Start programming

using

Object Pascal

Written by: Motaz Abdel Azeem
Edited by: Pat Anderson, Jason Hackney

code.sd

18.June.2011

Introduction

This book is written for programmers who want to learn the Object Pascal Language. It is also suitable as a first programming book for new students and non-programmers. It illustrates programming techniques in general in addition to the Object Pascal Language.

The Object Pascal Language

The first appearance of the Pascal Language supporting Object Oriented programming was in 1983 by Apple computer company. After that Borland supported Object Oriented programming for their famous Turbo Pascal line.

Object Pascal is a general purpose hybrid (structured and object oriented programming) language. It can be used for a vast range of applications, like learning, game development, business applications, Internet applications, communication applications, tools development, and OS kernels.

Delphi

After the success of Turbo Pascal, Borland decided to port it to Windows and introduced component driven technology to it. Soon Delphi became the best RAD (Rapid Application Development) tool at that time.

The first version of Delphi was released in 1995 with a rich set of components and packages that supported Windows and Database applications development.

Free Pascal

After Borland dropped support for the Turbo Pascal line, the Free Pascal team started an open source project to write a compatible compiler for Turbo Pascal from scratch, and then make it compatible with Delphi. This time the Free Pascal compiler was targeting additional platforms and operating systems like Windows, Linux, Mac, ARM, and WinCE.

Version 1.0 of the Free Pascal compiler was released in July 2000.

Lazarus

Free Pascal is a compiler, and it lacks an Integrated Development Environment (IDE) similar to the

Delphi IDE for Windows. The Lazarus project was started to provide an IDE for Free Pascal. It provides a source code editor, debugger, and contains a lot of frameworks, packages, and component libraries similar to the Delphi IDE.

Version 1.0 of Lazarus has not been released yet, but there are a lot of applications developed with beta versions of Lazarus. A lot of volunteers write packages and components for Lazarus, and the community is growing.

Object Pascal features

Object Pascal is a very easy and readable language for beginners, its compilers are very fast, and the applications it produces are reliable, fast and can be compared with C, and C++. You can write robust and large applications with its IDEs (Lazarus and Delphi) without complexity.

Author: Motaz Abdel Azeem

I am graduated from Sudan University of Science and Technology in 1999. and I started learning Pascal as a second language after BASIC. Since then, I've been using it continuously, and I found it a very easy and powerful tool, specially after I studied C, and C++. Then I moved to Delphi. Since then I have been using Delphi and Lazarus for all my applications.

I live in Khartoum. My current job is Software Developer.

First Editor

Pat Anderson graduated from Western Washington State College in 1968 and Rutgers Law School in 1975. He works as the City Attorney of Snoqualmie, Washington. Pat began programming on a Radio Shack TRS-80 Model III in 1982 with the built-in BASIC interpreter, but soon discovered Turbo Pascal. He has owned all versions of Turbo Pascal from 4.0 to 7.0, and every version of Delphi from 1.0 to 4.0. Pat took a hiatus from programming from 1998 until 2009, when he came upon Free Pascal / Lazarus, which reignited his passion for programming.

Second Editor

Jason Hackney is a graduate of Western Michigan University's College of Aviation. He works full-time as a professional pilot for a power company based in southeast Michigan. Jason has been a casual programmer since his first exposure to the Commodore 64 around 1984. Briefly introduced to Turbo Pascal in 1990, he recently rekindled latent programming interest after discovering Linux, Lazarus, and Free Pascal.

License:

The License for this book is **Creative Commons**.

Environment for book examples

We will use Lazarus and Free Pascal for all the examples in this book. You can get the Lazarus IDE, including the Free Pascal compiler, from this site: http://lazarus.freepascal.org.

If you are using Linux, then you can get Lazarus from the software repository. In Ubuntu you can use the command:

```
sudo apt-get install lazarus
```

In Fedora you can use the command:

```
yum install lazarus
```

Lazarus is a free and open source application. And it is available on many platforms. Applications written in Lazarus can be re-compiled on another platform to produce executables for that platform. For example if you write an application using Lazarus in Windows, and you want to produce a Linux executable for that application, you only need to copy your source code to Lazarus under Linux, then compile it.

Lazarus produces applications that are native to each operating system, and it does not require any additional libraries or virtual machines. For that reason, it is easy to deploy and fast in execution.

Using Text mode

All examples in the first chapters of this book will be console applications (text mode applications/ command line applications), because they are easy to understand and standard. Graphical user interface applications will be introduced in later chapters.

Contents

Chapter One

Language Basics

Chapter Two

Structured Programming

Chapter Three

The Graphical User Interface

Chapter Four

Object Oriented

Programming

Chapter One

Language Basics

Our First Application

After installing and running Lazarus, we can start a new program from the main menu:

Project/New Project/Program

We will get this code in the Source Editor window:

```
program Project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

{$IFDEF WINDOWS}{$R project1.rc}{$ENDIF}

begin
end.
```

We can save this program by clicking File/Save from the main menu, and then we can name it, for example, first.lpi

Then we can write these lines between the *begin* and *end* statements:

```
  Writeln('This is Free Pascal and Lazarus');
  Writeln('Press enter key to close');
  Readln;
```

The complete source code will be:

```
program first;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
```

```
{$IFDEF WINDOWS}{$R first.rc}{$ENDIF}

begin
  Writeln('This is Free Pascal and Lazarus');
  Writeln('Press enter key to close');
  Readln;
end.
```

The *Writeln* statement displays the text on the screen (Console window). *Readln* halts execution to let the user read the displayed text until he/she presses enter to close the application and return to the Lazarus IDE.

Then press *F9* to run the application or click the button:

After running the first program, we will get this output text:

```
This is Free Pascal and Lazarus
Press enter key to close
```

If we are using Linux, we will find a new file in a program directory called (first), and in Windows we will get a file named first.exe. Both files can be executed directly by double clicking with the mouse. The executable file can be copied to other computers to run without the need of the Lazarus IDE.

Notes

If the console application window does not appear, we can disable the debugger from the Lazarus menu:

Environment/Options/Debugger
In Debugger type and path select (None)

Other examples

In the previous program change this line:

```
  Writeln('This is Free Pascal and Lazarus');
```

to this one:

```
  Writeln('This is a number: ', 15);
```

Then press *F9* to run the application.

You will get this result:

```
This is a number: 15
```

Change the previous line as shown below, and run the application each time:

Code:

```
  Writeln('This is a number: ', 3 + 2);
```

Output:

```
This is a number: 5
```

Code:

```
  Writeln('5 * 2 = ', 5 * 2);
```

Output:

```
5 * 2 = 10
```

Code:

```
  Writeln('This is real number: ', 7.2);
```

Output:

```
This is real number: 7.2000000000000E+0000
```

Code:

```
  Writeln('One, Two, Three : ', 1, 2, 3);
```

Output:

```
One, Two, Three : 123
```

Code:

```
  Writeln(10, ' * ',  3 , ' = ', 10 * 3);
```

Output:

```
10 * 3 = 30
```

We can write different values in the *Writeln* statement each time and see the result. This will help us understand it clearly.

Variables

Variables are data containers. For example, when we say that X = 5, that means X is a variable, and it contains the value 5.

Object Pascal is a strongly typed language, which means we should declare a variable's type before putting values into it. If we declare X as an integer, that means we should put only integer numbers into X during its life time in the application.

Examples of declaring and using variables:

```
program FirstVar;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x: Integer;
begin
  x:= 5;
  Writeln(x * 2);
  Writeln('Press enter key to close');
  Readln;
end.
```

We will get 10 in the application's output.

Note that we used the reserved word *Var*, which means the following lines will be variable declarations:

```
  x: Integer;
```

This means two things:

1. The variable's name is X; and
2. the type of this variable is Integer, which can hold only integer numbers without a fraction. It could also hold negative values as well as positive values.

And the statement:

```
  x:= 5;
```

means put the value 5 in the variable X.

In the next example we have added the variable Y:

```
var
  x, y: Integer;
begin
  x:= 5;
  y:= 10;
  Writeln(x * y);
  Writeln('Press enter key to close');
  Readln;
end.
```

The output of the previous application is:

```
50
Press enter key to close
```

50 is the result of the formula (x * y).

In the next example we introduce a new data type called ***character***:

```
var
  c: Char;
begin
  c:= 'M';
  Writeln('My first letter is: ', c);
  Writeln('Press enter key to close');
  Readln;
end.
```

This type can hold only one letter, or a number as an alphanumeric character, not as value.

In the next example we introduce the *real* number type, which can have a fractional part:

```
var
  x: Single;
begin
  x:= 1.8;
  Writeln('My Car engine capacity is ', x, ' liters');
  Writeln('Press enter key to close');
```

```
  Readln;
end.
```

To write more interactive and flexible applications, we need to accept input from the user. For example, we could ask the user to enter a number, and then get this number from the user input using the ***Readln*** statement / procedure:

```
var
  x: Integer;
begin
  Write('Please input any number:');
  Readln(x);
  Writeln('You have entered: ', x);
  Writeln('Press enter key to close');
  Readln;
end.
```

In this example, assigning a value to *X* is done through the keyboard instead of assigning it a constant value in the application.

In the example below we show a multiplication table for a number entered by the user:

```
program MultTable;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x: Integer;
begin
  Write('Please input any number:');
  Readln(x);
  Writeln(x, ' * 1 = ', x * 1);
  Writeln(x, ' * 2 = ', x * 2);
  Writeln(x, ' * 3 = ', x * 3);
  Writeln(x, ' * 4 = ', x * 4);
  Writeln(x, ' * 5 = ', x * 5);
  Writeln(x, ' * 6 = ', x * 6);
  Writeln(x, ' * 7 = ', x * 7);
  Writeln(x, ' * 8 = ', x * 8);
```

```
  Writeln(x, ' * 9 = ', x * 9);
  Writeln(x, ' * 10 = ', x * 10);
  Writeln(x, ' * 11 = ', x * 11);
  Writeln(x, ' * 12 = ', x * 12);
  Writeln('Press enter key to close');
  Readln;
end.
```

Note that in the previous example all the text between single quotation marks (') is displayed in the console window as is, for example:

```
  ' * 1 = '
```

Variables and expressions that are written without single quotation marks are evaluated and written as values.

See the difference between the two statements below:

```
  Writeln('5 * 3');
  Writeln(5 * 3);
```

The result of first statement is:

```
5 * 3
```

Result of the second statement is evaluated then displayed:

```
15
```

In the next example, we will do mathematical operations on two numbers (x, y), and we will put the result in a third variable (Res):

```
var
  x, y: Integer;
  Res: Single;
begin
  Write('Input a number: ');
  Readln(x);
  Write('Input another number: ');
  Readln(y);
  Res:= x / y;
  Writeln(x, ' / ', y, ' = ', Res);
  Writeln('Press enter key to close');
  Readln;
end.
```

Since the operation is division, it might result in a number with a fraction, so for that reason we have

declared the Result variable (Res) as a real number (*Single*). ***Single*** means a real number with single precision floating point.

Sub types

There are many sub types for variables, for example, Integer number subtypes differ in the range and the number of required bytes to store values in memory.

The table below contains integer types, value ranges, and required bytes in memory:

Type	Min value	Max Value	Size in Bytes
Byte	0	255	1
ShortInt	-128	127	1
SmallInt	-32768	32767	2
Word	0	65535	2
Integer	-2147483648	2147483647	4
LongInt	-2147483648	2147483647	4
Cardinal	0	4294967295	4
Int64	-9223372036854780000	9223372036854775807	8

We can get the minimum and maximum values and bytes sizes for each type by using the Low, High, and SizeOf functions respectively, as in the example below:

```
program Types;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes;

begin
  Writeln('Byte: Size = ', SizeOf(Byte),
   ', Minimum value = ', Low(Byte), ', Maximum value = ',
   High(Byte));

  Writeln('Integer: Size = ', SizeOf(Integer),
    ', Minimum value = ', Low(Integer), ', Maximum value = ',
    High(Integer));

  Write('Press enter key to close');
  Readln;
end.
```

Conditional Branching

One of the most important features of intelligent devices (like computers, programmable devices) is that they can take actions in different conditions. This can be done by using conditional branching. For example, some cars lock the door when the speed reaches or exceeds 40 K/h. The condition in this case will be:

If speed is >= 40 and doors are unlocked, then lock door.

Cars, washing machines, and many other gadgets contains programmable circuits like micro controllers, or small processors like ARM. Such circuits can be programmed using assembly, C, or Free Pascal according to their architecture.

The If condition

The If condition statement in the Pascal language is very easy and clear. In the example below, we want to decide whether to turn on the air-conditioner or turn it off, according to the entered room temperature:

Air-Conditioner program:

```
var
  Temp: Single;
begin
  Write('Please enter Temperature of this room :');
  Readln(Temp);
  if Temp > 22 then
    Writeln('Please turn on air-condition')
  else
    Writeln('Please turn off air-condition');

  Write('Press enter key to close');
  Readln;
end.
```

We have introduced the *if then else* statement, and in this example: if the temperature is greater than 22, then display the first sentence::

```
Please turn on air-conditioner
```

else, if the condition is not met (less than or equal to 22) , then display this line:

```
Please turn off air-conditioner
```

We can write multiple conditions like:

```
var
  Temp: Single;
begin
  Write('Please enter Temperature of this room :');
  Readln(Temp);
  if Temp > 22 then
    Writeln('Please turn on air-conditioner')
  else
  if Temp < 18 then
    Writeln('Please turn off air-conditioner')
  else
    Writeln('Do nothing');
```

You can test the above example with different temperature values to see the results.

We can make conditions more complex to be more useful:

```
var
  Temp: Single;
  ACIsOn: Byte;
begin
  Write('Please enter Temperature of this room : ');
  Readln(Temp);
  Write('Is air conditioner on? if it is (On) write 1,',
   ' if it is (Off) write 0 : ');
  Readln(ACIsOn);

  if (ACIsOn = 1) and (Temp > 22) then
    Writeln('Do nothing, we still need cooling')
  else
  if (ACIsOn = 1) and (Temp < 18) then
    Writeln('Please turn off air-conditioner')
  else
  if (ACIsOn = 0) and (Temp < 18) then
    Writeln('Do nothing, it is still cold')
  else
  if (ACIsOn = 0) and (Temp > 22) then
    Writeln('Please turn on air-conditioner')
  else
    Writeln('Please enter a valid values');
```

```
  Write('Press enter key to close');
  Readln;
end.
```

In the above example, we have used the new keyword (***and***) which means if the first condition returns True (ACIsOn = 1), and the second condition returns True (Temp > 22), then execute the *Writeln* statement. If one condition or both of them return False, then it will go to the else part.

If the air-conditioner is connected to a computer via the serial port for example, then we can turn it on/off from that application, using serial port procedures/components. In this case we need to add extra parameters for the if condition, like for how long the air-conditioner was operating. If it exceeds the allowable time (for example, 1 hour) then it should be turned off regardless of room temperature. Also we can consider the rate of losing coolness, if it is very slow (at night), then we could turn if off for longer time.

Weight program

In this example, we ask the user to enter his/her height in meters, and weight in Kilos. Then the program will calculate the suitable weight for that person according to the entered data, and then it will tell him/her the results:

```
program Weight;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  Height: Double;
  Weight: Double;
  IdealWeight: Double;
begin
  Write('What is your height in meters (e.g. 1.8 meter) : ');
  Readln(Height);
  Write('What is your weight in kilos : ');
  Readln(Weight);
  if Height >= 1.4 then
```

```
    IdealWeight:= (Height - 1) * 100
  else
    IdealWeight:= Height * 20;

  if (Height < 0.4) or (Height > 2.5) or (Weight < 3) or
  (Weight > 200) then
  begin
    Writeln('Invalid values');
    Writeln('Please enter proper values');
  end
  else
  if IdealWeight = Weight then
    Writeln('Your weight is suitable')
  else
  if IdealWeight > Weight then
    Writeln('You are under weight, you need more ',
     Format('%.2f', [IdealWeight - Weight]), ' Kilos')
  else
    Writeln('You are over weight, you need to lose ',
     Format('%.2f', [Weight - IdealWeight]), ' Kilos');

  Write('Press enter key to close');
  Readln;
end.
```

In this example, we have used new keywords:

1. ***Double:*** which is similar to *Single*. Both of them are real numbers, but Double has a double precision floating point, and it requires 8 bytes in memory, while single requires only 4 bytes.

2. The second new thing is the keyword (***Or***) and we have used it to check if one of the conditions is met or not. If one of condition is met, then it will execute the statement. For example: if the first condition returns *True* (Height < 0.4), then the *Writeln* statement will be called: *Writeln('Invalid values')*; . If the first condition returns *False*, then it will check the second one, etc. If all conditions returns *False*, it will go to the *else* part.

3. We have used the keywords ***begin end*** with the if statement, because *if statement* should execute one statement. *Begin end* converts multiple statements to be considered as one block (statement), then multiple statements could be executed by *if condition.* Look for these two statemetns:

```
    Writeln('Invalid values');
    Writeln('Please enter proper values');
```

   It has been converted to one statement using *begin end*:

```
  if (Height < 0.4) or (Height > 2.5) or (Weight < 3) or
```

```
    (Weight > 200) then
    begin
      Writeln('Invalid values');
      Writeln('Please enter proper values');
    end
```

4. We have used the procedure *Format*, which displays values in a specific format. In this case we need to display only 2 digits after the decimal point. We need to add the *SysUtils* unit to the *Uses* clause in order to use this function.

```
What is your height in meters (e.g. 1.8 meter) : 1.8
What is your weight in kilos : 60.2
You are under weight, you need more 19.80 Kilos
```

Note:

This example may be not 100% accurate. You can search the web for weight calculation in detail. We meant only to explain how the programmer could solve such problems and do good analysis of the subject to produce reliable applications.

Case .. of statement

There is another method for conditional branching, which is the *Case .. Of* statement. It branches execution according to the *case* ordinal value. The Restaurant program will illustrate the use of the *case of* statement:

Restaurant program

```
var
  Meal: Byte;
begin

  Writeln('Welcome to Pascal Restaurant. Please select your order');
  Writeln('1 - Chicken      (10$)');
  Writeln('2 - Fish         (7$)');
  Writeln('3 - Meat         (8$)');
  Writeln('4 – Salad        (2$)');
  Writeln('5 - Orange Juice (1$)');
  Writeln('6 - Milk         (1$)');
  Writeln;
  Write('Please enter your selection: ');
  Readln(Meal);

  case Meal of
    1: Writeln('You have ordered Chicken,',
        ' this will take 15 minutes');
    2: Writeln('You have ordered Fish, this will take 12 minutes');
    3: Writeln('You have ordered meat, this will take 18 minutes');
    4: Writeln('You have ordered Salad, this will take 5 minutes');
    5: Writeln('You have ordered Orange juice,',
         ' this will take 2 minutes');
    6: Writeln('You have ordered Milk, this will take 1 minute');
  else
    Writeln('Wrong entry');
  end;
  Write('Press enter key to close');
  Readln;
end.
```

If we write the same application using the if condition, it will become more complicated, and will contain duplications:

Restaurant program using If condition

```
var
  Meal: Byte;
begin
  Writeln('Welcome to Pascal restaurant, please select your meal');
  Writeln('1 - Chicken      (10$)');
  Writeln('2 - Fish         (7$)');
  Writeln('3 - Meat         (8$)');
  Writeln('4 - Salad        (2$)');
  Writeln('5 - Orange Juice (1$)');
  Writeln('6 - Milk         (1$)');
  Writeln;
  Write('Please enter your selection: ');
  Readln(Meal);

  if Meal = 1 then
    Writeln('You have ordered Chicken, this will take 15 minutes')
  else
  if Meal = 2 then
    Writeln('You have ordered Fish, this will take 12 minutes')
  else
  if Meal = 3 then
     Writeln('You have ordered meat, this will take 18 minutes')
  else
  if Meal = 4 then
    Writeln('You have ordered Salad, this will take 5 minutes')
  else
  if Meal = 5 then
    Writeln('You have ordered Orange juice,' ,
    ' this will take 2 minutes')
  else
  if Meal = 6 then
    Writeln('You have ordered Milk, this will take 1 minute')
  else
    Writeln('Wrong entry');

  Write('Press enter key to close');
  Readln;
end.
```

In the next example, the application evaluates students' marks and converts them to grades: A, B, C, D, E, and F:

Students' Grades program

```
var
  Mark: Integer;
begin
  Write('Please enter student mark: ');
  Readln(Mark);
  Writeln;

  case Mark of
    0 .. 39 : Writeln('Student grade is: F');
    40 .. 49: Writeln('Student grade is: E');
    50 .. 59: Writeln('Student grade is: D');
    60 .. 69: Writeln('Student grade is: C');
    70 .. 84: Writeln('Student grade is: B');
    85 .. 100: Writeln('Student grade is: A');
  else
    Writeln('Wrong mark');
  end;

  Write('Press enter key to close');
  Readln;
end.
```

In the previous sample we have used a range, like (0 .. 39), which means the condition will return *True* if the *Mark* value exists in this range.

Note:
The Case statement works only with ordinal types like *Integers*, *char*, but it doesn't work with other types like *strings*, and *real* numbers.

Keyboard program

In this example, we will get a character from keyboard and the application will tell us the row number of the entered key on the keyboard:

```
var
  Key: Char;
begin
  Write('Please enter any English letter: ');
  Readln(Key);
  Writeln;
```

```
  case Key of
    'q', 'w', 'e', 'r', 't', 'y', 'u', 'i', 'o', 'p':
      Writeln('This is in the second row in keyboard');

    'a', 's', 'd', 'f', 'g', 'h', 'j', 'k', 'l':
      Writeln('This is in the third row in keyboard');

    'z', 'x', 'c', 'v', 'b', 'n', 'm':
      Writeln('This is in the fourth row in keyboard');
  else
    Writeln('Unknown letter');
  end;


  Write('Press enter key to close');
  Readln;
end.
```

Note that we have used a new technique in the case condition, which is a set of values:

```
    'z', 'x', 'c', 'v', 'b', 'n', 'm':
```

which means execute case branch statement if the Key was one of these values set:
z, x, c, v, b, n or m

We can also mix ranges with values set like this:

```
    'a' .. 'd', 'x', 'y', 'z':
```

which means execute the statement if the value falls between a and d, or equal to x, y or z.

Loops

Loops are used to execute certain parts of code (statements) for a specific number of times, or until a condition is satisfied.

For loop

You can execute for statements for a specific number of cycles using a counter like this example:

```
var
  i: Integer;
  Count: Integer;
begin
  Write('How many times? ');
  Readln(Count);
  for i:= 1 to Count do
    Writeln('Hello there');

  Write('Press enter key to close');
  Readln;
end.
```

We should use ordinal types like *Integer*, *Byte*, and *Char* in *for* loop variables. We call this variable a *loop variable* or *loop counter*. The value of *loop counter* can be initialized with any number, and we can also determine the last value of *loop counter*. For example, if we need to count from 5 to 10, then we can do this:

```
  for i:= 5 to 10 do
```

We can display *loop counter* in every cycle of the loop, as in the modified example below:

```
var
  i: Integer;
  Count: Integer;
begin
  Write('How many times? ');
  Readln(Count);
  for i:= 1 to Count do
  begin
    Writeln('Cycle number: ', i);
    Writeln('Hello there');
  end;

  Write('Press enter key to close');
  Readln;
end.
```

Note that this time we need to repeat two statements, and for that reason we have used the *begin . . end* keywords to make them one statement.

Multiplication Table using *for* loop

The for loop version of the Multiplication Table program is easier and more concise:

```
program MultTableWithForLoop;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x, i: Integer;
begin
  Write('Please input any number: ');
  Readln(x);
  for i:= 1 to 12 do
    Writeln(x, ' * ', i, ' = ', x * i);

  Writeln('Press enter key to close');
  Readln;
end.
```

Instead of writing the *Writeln* statement 12 times, we write it once inside a loop which is executed 12 times.

We can make the *for* loop statement iterate in a backward direction using ***downto*** keyword instead of ***to*** keyword using this syntax:

```
  for i:= 12 downto 1 do
```

Factorial program

Factorial in mathematics is the multiplication of a number by each one of its predecessors down to the number 1. For example, 3! = 3 * 2 * 1 = 6.

```
var
  Fac, Num, i: Integer;
begin
  Write('Please input any number: ');
  Readln(Num);
  Fac:= 1;
  for i:= Num downto 1 do
    Fac:= Fac * i;
  Writeln('Factorial of ', Num ,' is ', Fac);

  Writeln('Press enter key to close');
  Readln;
end.
```

Repeat Until loop

Unlike the *for* loop which repeats for a specific number of cycles, the *Repeat* loop has no counter. It loops until a certain condition occurs (Returns True), then it will go to the next statement after the loop.

Example:

```
var
  Num : Integer;
begin
  repeat
    Write('Please input a number: ');
    Readln(Num);
  until Num <= 0;
  Writeln('Finished, please press enter key to close');
  Readln;
end.
```

In the previous example, the program enters the loop, then it asks the user to enter a number. If the number is less than or equal to zero, it will exit the loop. If the entered number is greater than zero, the loop will continue.

Restaurant program using Repeat loop

```
var
  Selection: Char;
  Price: Integer;
  Total: Integer;
begin
  Total:= 0;
  repeat
    Writeln('Welcome to Pascal Restaurant. Please select your order');
    Writeln('1 - Chicken      (10 Geneh)');
    Writeln('2 - Fish         (7 Geneh)');
    Writeln('3 - Meat         (8 Geneh)');
    Writeln('4 – Salad        (2 Geneh)');
    Writeln('5 - Orange Juice (1 Geneh)');
    Writeln('6 - Milk         (1 Geneh)');
    Writeln('X - nothing');
    Writeln;
    Write('Please enter your selection: ');
    Readln(Selection);

    case Selection of
      '1': begin
             Writeln('You have ordered Chicken, this will take 15 minutes');
             Price:= 10;
           end;
      '2': begin
             Writeln('You have ordered Fish, this will take 12 minutes');
             Price:= 7;
```

```
          end;
      '3': begin
             Writeln('You have ordered meat, this will take 18 minutes');
             Price:= 8;
           end;
      '4': begin
             Writeln('You have ordered Salad, this will take 5 minutes');
             Price:= 2;
           end;
      '5': begin
             Writeln('You have ordered Orange juice, this will take 2 minutes');
             Price:= 1;
           end;
      '6': begin
             Writeln('You have ordered Milk, this will take 1 minute');
             Price:= 1;
           end;
      else
      begin
        Writeln('Wrong entry');
        Price:= 0;
      end;
    end;

    Total:= Total + Price;

  until (Selection = 'x') or (Selection = 'X');
  Writeln('Total price           = ', Total);
  Write('Press enter key to close');
  Readln;
end.
```

In the previous example, we used these techniques:

1. Adding *begin end* in case branches to convert multiple statements to one statement
2. We initialized (put a starting value in a variable) the variable *Total* to zero, to accumulate the total price of orders. Then we added the selected price in every loop to the *Total* variable:

```
Total:= Total + Price;
```

4. We put two options to finish the orders, either capital X, or small x. Both characters are different in representation (storage) in computer memory.

Note:
We could replace this line:

```
  until (Selection = 'x') or (Selection = 'X');
```

By this abbreviated code:

```
  until UpCase(Selection) = 'X';
```

This will change the *Selection* variable to upper case if it is a lowercase letter, and the condition will return *True* in both cases (x or X)

While loop

The *while* loop is similar to the *repeat* loop, but it differs from it in these aspects:

1. In the *while* , the condition is checked first before entering the loop, but *repeat* enters the loop first then it checks the condition. That means *repeat* always executes its statement(s) once at least, but *while* loop could prevent entering the first cycle if the condition returns *False* from the beginning.

2. *while l*oop needs *begin end* if there are multiple statements that need to be executed in a loop, but *repeat* does not need *begin end*, its block (repeated statements) starts from the *repeat* keyword to the *until* keyword.

Example:

```
var
  Num: Integer;
begin
  Write('Input a number: ');
  Readln(Num);
  while Num > 0 do
  begin
    Write('From inside loop: Input a number : ');
    Readln(Num);
  end;
  Write('Press enter key to close');
  Readln;
end.
```

Factorial program using while loop

```
var
  Fac, Num, i: Integer;
begin
  Write('Please input any number: ');
  Readln(Num);
  Fac:= 1;
  i:= Num;
  while i > 1 do
  begin
    Fac:= Fac * i;
    i:= i - 1;
  end;
  Writeln('Factorial of ', Num ,' is ', Fac);

  Writeln('Press enter key to close');
  Readln;
```

```
end.
```

The *while* loop has no loop counter, and for that reason we have used the variable *i* to work as a loop counter. The loop counter value is initialized by the number for which we need to get its factorial, then we decrease it manually in each cycle of the loop. When *i* reaches 1, the loop will break.

Strings

The *String* type is used to declare variables that can hold a chain of characters. It can be used to store text, a name, or a combination of characters and digits like a car license plate number.

In this example we will see how we can use the *string* variable to accept a user name:

```
var
  Name: string;
begin
  Write('Please enter your name : ');
  Readln(Name);
  Writeln('Hello ', Name);

  Writeln('Press enter key to close');
  Readln;
end.
```

In the next example, we will use strings to store information about a person:

```
var
  Name: string;
  Address: string;
  ID: string;
  DOB: string;
begin
  Write('Please enter your name : ');
  Readln(Name);
  Write('Please enter your address : ');
  Readln(Address);
  Write('Please enter your ID number : ');
  Readln(ID);
  Write('Please enter your date of birth : ');
  Readln(DOB);
  Writeln;
  Writeln('Card:');
  Writeln('-----------------------------------------');
  Writeln('| Name    : ', Name);
  Writeln('| Address : ', Address);
  Writeln('| ID      : ', ID);
  Writeln('| DOB     : ', DOB);
  Writeln('-----------------------------------------');

  Writeln('Press enter key to close');
  Readln;
end.
```

Strings can be concatenated to produce larger strings. For example we can concatenate *FirstName*, *SecondName*, and *FamilyName* into another string called *FullName*, similar to the next example:

```
var
  YourName: string;
  Father: string;
  GrandFather: string;
  FullName: string;
begin
  Write('Please enter your first name : ');
  Readln(YourName);
  Write('Please enter your father name : ');
  Readln(Father);
  Write('Please enter your grand father name : ');
  Readln(GrandFather);
  FullName:= YourName + ' ' + Father + ' ' + GrandFather;
  Writeln('Your full name is: ', FullName);

  Writeln('Press enter key to close');
  Readln;
end.
```

Note that in this example we have added a space (' ') between names (like *YourName* + ' ' + *Father*) to make a separation between names. The space is a character too.

We can do many operations on strings, like searching for subtext in a string, copying one string to another string variable, or converting text characters to capital or lowercase like the examples below:

This line converts the letters in the *FullName* string value to capital letters:

```
  FullName:= UpperCase(FullName);
```

And this converts it to lowercase letters:

```
  FullName:= LowerCase(FullName);
```

In the next example, we need to search for the letter *a* in a user name using the *Pos* function. The *Pos* function returns the first occurrence (Index) of that character in a string, or returns *zero* if that letter or substring does not exist in the searched text:

```
var
  YourName: string;
begin
  Write('Please enter your name : ');
  Readln(YourName);
  If Pos('a', YourName) > 0 then
    Writeln('Your name contains a')
  else
    Writeln('Your name does not contain a letter');

  Writeln('Press enter key to close');
  Readln;
end.
```

If the name contains a capital *A*, then *Pos* function will not point to it, because *A* is different from *a* as we mentioned earlier.

To solve this problem we can convert all user name letters to lowercase case, and then we can do the search:

```
  If Pos('a', LowerCase(YourName)) > 0 the
```

In the next modification of the code, we will display the position of the letter *a* in a user name:

```
var
  YourName: string;
begin
  Write('Please enter your name : ');
  Readln(YourName);
  If Pos('a', LowerCase(YourName)) > 0 then
  begin
    Writeln('Your name contains a');
    Writeln('a position in your name is: ',
      Pos('a', LowerCase(YourName)));
  end
  else
    Writeln('Your name does not contain a letter');

  Write('Press enter key to close');
  Readln;
end.
```

You may notice that if the name contains more than one letter *a* , the function *Pos* will return the index of the first occurrence of the letter *a* in the user name.

You can get the count of characters in a string using the function *Length*:

```
  Writeln('Your name length is ', Length(YourName), ' letters');
```

And you can get the first letter/character of a string using its index number:

```
  Writeln('Your first letter is ', YourName[1]);
```

And the second character:

```
  Writeln('Your second letter is ', YourName[2]);
```

The last character:

```
  Writeln('Your last letter is ', YourName[Length(YourName)]);
```

You can also display a string variable character by character using a *for* loop:

```
  for i:= 1 to Length(YourName) do
    Writeln(YourName[i]);
```

Copy function

We can copy part of a string using the function *copy*. For example, if we need to extract the word '*world*' from the string '*hello world*', we can do it if we know the position of that part, as we have done in the example below:

```
var
  Line: string;
  Part: string;
begin
  Line:= 'Hello world';

  Part:= Copy(Line, 7, 5);

  Writeln(Part);

  Writeln('Press enter key to close');
  Readln;
end.
```

Note that we have used the *Copy* function using this syntax:

```
  Part:= Copy(Line, 7, 5);
```

Here is an explanation of above statement:

- Part:= This is the string variable in which we will put the function result (sub string, '*world*').
- Line This is the source string which contains the '*Hello world*' sentence.
- 7 This is the starting point or index of sub string that we need to extract, in this case it is the character *w*.
- 5 This is the length of the extracted part. In this case it represents the length of the word '*world*'.

In the next example, we will ask the user to enter a month name, like *February*, then the program will type the short version of it, like *Feb*:

```
var
  Month: string;
  ShortName: string;
begin
  Write('Please input full month name e.g. January : ');
  Readln(Month);
```

```
  ShortName:= Copy(Month, 1, 3);

  Writeln(Month, ' is abbreviated as : ', ShortName);

  Writeln('Press enter key to close');
  Readln;
end.
```

Insert procedure

The *Insert* procedure inserts a substring into a string. Unlike the string concatenation operator (+) which links two substrings together, *Insert* adds the substring in the middle of another string.

For example, we can insert the word *Pascal* into the string '*Hello world*', resulting in '*Hello Pascal World*' as in the following example:

```
var
  Line: string;
begin
  Line:= 'Hello world';

  Insert('Pascal ', Line, 7);

  Writeln(Line);

  Writeln('Press enter key to close');
  Readln;
end.
```

Parameters of the Insert procedure are:

- '**Pascal**' This is the substring that we need to insert inside the destination string.
- **Line** This is the destination string that will contain the result of the operation.
- **7** This is the position to start the insertion in the destination string. In this case it will be after the seventh character, which is the first space of '*Hello world*'.

Delete procedure

This procedure is used to delete a character or substring from a string. We need to know the start position and the length of substring that should be deleted.

For example, if we need to delete the letters *ll* from the string '*Hello World*' to make '*Heo World*', we can do it like this:

```
var
  Line: string;
begin
  Line:= 'Hello world';
  Delete(Line, 3, 2);
  Writeln(Line);
  Writeln('Press enter key to close');
  Readln;
end.
```

Trim function

This function is used to remove spaces from the start and the end of strings. If we have a string that contains the text ' *Hello* ' it will be '*Hello*' after using this function. We can not display spaces in a terminal window unless we put characters between them. Look at the example below:

```
program TrimStr;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  Line: string;
begin
  Line:= ' Hello ';

  Writeln('<', Line, '>');

  Line:= Trim(Line);

  Writeln('<', Line, '>');

  Writeln('Press enter key to close');
  Readln;
end.
```

In the foregoing example, we have used the unit *SysUtils,* which contains the *Trim* function.

There are another two functions that remove spaces from only one side of a string, before/after. These functions are: *TrimRight*, *TrimLeft*.

StringReplace function

The *StringReplace* function replaces characters or substrings with other characters or substrings in the desired string.

```
program StrReplace;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  Line: string;
  Line2: string;
begin
  Line:= 'This is a test for string replacement';
  Line2:= StringReplace(Line, ' ', '-', [rfReplaceAll]);
  Writeln(Line);
  Writeln(Line2);
  Write('Press enter key to close');
  Readln;
end.
```

The parameters of the *StringReplace* function are:

1. Line: This is the original string that we need to modify.
2. ' ': This is the substring that we need to replace. It is *space* in this example.
3. '-': This is the alternative substring that we want to replace the previous one in original string.
4. [rfReplaceAll]: This is the replacement type. In this case we need to replace all occurrence of the *space* substring.

We can use only one string variable and discard the variable *Line2* as modified example shows, but we will lose the original text value.

```
var
  Line: string;
begin
  Line:= 'This is a test for string replacement';
  Writeln(Line);
  Line:= StringReplace(Line, ' ', '-', [rfReplaceAll]);
  Writeln(Line);
  Write('Press enter key to close');
  Readln;
end.
```

Arrays

An Array is a chain of variables of the same type. If we need to declare an array of 10 Integer variables, we can do it like this:

```
  Numbers: array [1 .. 10] of Integer;
```

We can access single variables in the array using its index. For example, to put a value in the first variable in the array we can write it as:

```
  Numbers[1]:= 30;
```

To put a value in the second variable, use the index 2:

```
  Numbers[2]:= 315;
```

In the next example, we will ask the user to enter 10 student marks and put them in an array. Eventually we will go through them to extract pass/fail results:

```
var
  Marks: array [1 .. 10] of Integer;
  i: Integer;
begin
  for i:= 1 to 10 do
  begin
    Write('Input student number ', i, ' mark: ');
    Readln(Marks[i]);
  end;

  for i:= 1 to 10 do
  begin
    Write('Student number ', i, ' mark is : ', Marks[i]);
    if Marks[i] >= 40 then
      Writeln(' Pass')
    else
      Writeln(' Fail');
  end;

  Writeln('Press enter key to close');
  Readln;
end.
```

We can modify the previous code to get the highest and lowest student marks:

```
var
  Marks: array [1 .. 10] of Integer;
  i: Integer;
  Max, Min: Integer;
begin
```

```
  for i:= 1 to 10 do
  begin
    Write('Input student number ', i, ' mark: ');
    Readln(Marks[i]);
  end;

  Max:= Marks[1];
  Min:= Marks[1];

  for i:= 1 to 10 do
  begin
    // Check if current Mark is maximum mark or not
    if Marks[i] > Max then
      Max:= Marks[i];

    // Check if current value is minimum mark or not
    if Marks[i] < Min then
      Min:= Marks[i];

    Write('Student number ', i, ' mark is : ', Marks[i]);
    if Marks[i] >= 40 then
      Writeln(' Pass')
    else
      Writeln(' Fail');
  end;

  Writeln('Max mark is: ', Max);
  Writeln('Min mark is: ', Min);
  Writeln('Press enter key to close');
  Readln;
end.
```

Note that we consider the first mark (*Marks[1]*) as the maximum and minimum mark, and then we will compare it with the rest of the marks.

```
  Max:= Marks[1];
  Min:= Marks[1];
```

Inside the loop we compare *Max*, *Min* with each mark. If we find a number that is larger than Max, we will replace *Max* value with it. If we find a number that is less than *Min*, then we will replace *Min* with it.

In the previous example we have introduced comments:

```
    // Check if current Mark is maximum mark or not
```

We have started the line with the characters //, which means that this line is a comment and will not affect the code and will not be compiled. This technique is used to describe parts of code to other programmers or to the programmer himself/herself.

// is used for short comments. If we need to write multi-line comments we can surround them by *{}* or *(**)*

Example:

```
  for i:= 1 to 10 do
  begin
    { Check if current Mark is maximum mark or not
      check if Mark is greater than Max then put
      it in Max }
    if Marks[i] > Max then
      Max:= Marks[i];

    (* Check if current value is minimum mark or not
       if Min is less than Mark then put Mark value in Min
    *)
    if Marks[i] < Min then
      Min:= Marks[i];

    Write('Student number ', i, ' mark is : ', Marks[i]);
    if Marks[i] >= 40 then
      Writeln(' Pass')
    else
      Writeln(' Fail');
  end;
```

We can also disable part of our code temporarily by commenting it:

```
  Writeln('Max mark is: ', Max);
// Writeln('Min mark is: ', Min);
  Writeln('Press enter key to close');
  Readln;
```

In the above code, we have disabled the procedure for writing the minimum student mark.

Note:

We can declare an array as having a *zero* based index the same as in the C language:

```
  Marks: array [0 .. 9] of Integer;
```

This can hold 10 elements too, but the first item can be accessed using the index 0:

```
  Numbers[0]:= 30;
```

Second item:

```
  Numbers[1]:= 315;
```

Last item:

```
  Numbers[9]:= 10;
```

or

```
  Numbers[High(Numbers)]:= 10;
```

Records

While *arrays* can hold many variables of the same type, *records* can hold variables of different types, and these variables are called '*Fields*'.

This group of variables/fields can be treated as a single unit or variable. We can use records to store information that belong to the same object, for example, car information:

1. *Car type*: string variable
2. *Engine size*: real number
3. *Production year*: integer value

We can collect these different types in one *record* which represents a Car as in the following example:

```
program Cars;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TCar = record
    ModelName: string;
    Engine: Single;
    ModelYear: Integer;
  end;


var
  Car: TCar;
begin
  Write('Input car Model Name: ');
  Readln(Car.ModelName);
  Write('Input car Engine size: ');
  Readln(Car.Engine);
  Write('Input car Model year: ');
  Readln(Car.ModelYear);

  Writeln;
  Writeln('Car information: ');
  Writeln('Model Name  : ', Car.ModelName);
  Writeln('Engine size : ', Car.Engine);
  Writeln('Model Year  : ', Car.ModelYear);

  Write('Press enter key to close..');
  Readln;
end.
```

In this example ,we have defined a new type (*Record*) using the '*type'* keyword:

```
type
  TCar = record
    ModelName: string;
    Engine: Single;
    ModelYear: Integer;
  end;
```

We have added the letter (*T*) to *Car* to indicate that this is a type and not a variable. Variable names could be like: *Car, Hour, UserName*, but type names should be like: *TCar, THouse, and TUserName*. This is a standard in the Pascal language.

When we need to use this new type, then we should declare a variable of that type, for example:

```
var
  Car: TCar;
```

If we need to store a value in one of its variables/fields, we should access it like this:

```
  Car.ModelName
```

Records will be used in the *Random access files* section of this book.

Files

Files are important elements of operating systems and applications. Operating system components are represented as files, and information and data are represented in files too, like photos, books, applications, and simple text files.

Operating systems control files management like: reading, writing, editing and deleting files.

Files are divided into many types according to many perspectives. We can group files into two types: *executable* files, and *data* files. For example compiled binary Lazarus applications are executable files, while Pascal source code (.pas) are data files. Also PDF books, JPEG pictures, are data files.

We can divide data files into two types according to the representation of their contents:

1. *Text files*: which are simple text files that can be written, or read using any simple tool including operating system command lines like *cat, vi* commands in Linux and *type, copy con* commands in Windows.

2. *Binary data files*: These are more complex and need special applications to open them. For example, picture files can not be opened using simple command line tools, instead they should be opened using applications like *GIMP*, *Kolour Paint*, *MS Paint,* etc. If we open picture, or voice files, we will get unrecognizable characters which mean nothing to the user. Examples of binary data files are database files, which should be opened using the proper application.

There is another way to categorize files according to access type:

1. *Sequential access files*: An example of a sequential file is a text file, which has no fixed size record. Each line has its own length, so we can't know the position (in characters) for the start of the third line for example. For that reason, we could open the file for read only, or writing only, and we can also append text only to the end of file. If we need to insert text in the middle of a file, then we should read file contents into memory, do the modifications, erase the entire file from disk, then overwrite it the file with the modified text.

2. *Random access files*: This type of file has a fixed size record. A record is the smallest unit that we can read and write at one time, and it could be *Byte*, *Integer*, *string*, or a user defined record. We can read and write at the same time, for example, we could read the record number 3 and copy it to the record number 10. In this case, we can know exactly the position of each record in the file. Modifying records is simple. In random access files, we can replace/overwrite any record without affecting the rest of the file.

Text files

Text files are the simplest files, but we should write to them in only one direction (only forward). We can not go back while writing to a text file. We should also define the operation mode before opening the file: *Read*, *Write* or *Append* (writing at the end).

In this example, we will display the contents of a text file selected by a user. For example, a user may have a text file name like *c:\test\first.pas* in Windows or */home/user/first.pas* in Linux:

Reading text file program

```
program ReadFile;

{$mode objfpc}{$H+}
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, sysUtils
  { you can add units after this };

var
  FileName: string;
  F: TextFile;
  Line: string;
begin
  Write('Input a text file name: ');
  Readln(FileName);
  if FileExists(FileName) then
  begin
    // Link file variable (F) with physical file (FileName)
    AssignFile(F, FileName);

    Reset(F); // Put file mode for read, file should exist

    // while file has more lines that does not read yet do the loop
    while not Eof(F) do
    begin
      Readln(F, Line); // Read a line from text file
      Writeln(Line);   // Display this line in user screen
    end;
    CloseFile(F); // Release F and FileName connection
  end
  else // else if FileExists..
    Writeln('File does not exist');
  Write('Press enter key to close..');
  Readln;
end.
```

In Linux we can enter these file names:

```
/etc/resolv.conf
```
or
```
/proc/meminfo
/proc/cpuinfo
```

We have used new types, functions and procedures to manipulate text files which are:

1.
```
  F: TextFile;
```

TextFile is a type that is used to declare a text file variable. This variable can be linked to a text file for later usage.

2.
```
  if FileExists(FileName) then
```

FileExists is a function contained in the *SysUtils* unit. It checks for a file's existence. It returns *True* if the file exists in the storage media.

3.
```
  AssignFile(F, FileName);
```

After we have checked for the file's existence, we can use the *AssignFile* procedure to link the file variable (*F*) with the physical file. Any further usage of *F* variable will represent the physical file.

4.
```
  Reset(F); // Put file mode for read, file should exist
```

The *Reset* procedure opens text files for reading only, and puts the reading pointer at the first character of the file.

If there is no read permission for that file for the current user, an error message will appear, like *access denied*.

5.
```
  Readln(F, Line); // Read a line from text file
```

The procedure *Readln* is used to read a complete line from the file and put it in the variable *Line*. Before reading a line from text file, we should make sure that the file has not reached the end. We can do that with the next function, *Eof*.

6.
```
  while not Eof(F) do
```

The *Eof* function returns *True* if the file has reached its end. It is used to indicate that the Read operation couldn't be used anymore and we have read all of the file's contents.

7.

```
   CloseFile(F); // Release F and FileName connection
```

After finishing reading from or writing to a file, we should close it to release the file, because the *Reset* procedure reserves the file in the operating system and prevents writing and deletion by another application while the file is open.

We should use *CloseFile* only when the *Reset* procedure succeeds in opening the file. If *Reset* fails (for example, if the file does not exist, or is being used by another application) in that case we should not close the file.

In the next example we will create a new text file and put some text in it:

Creating and writing into text file

```
var
  FileName: string;
  F: TextFile;
  Line: string;
  ReadyToCreate: Boolean;
  Ans: Char;
  i: Integer;
begin
  Write('Input a new file name: ');
  Readln(FileName);

  // Check if file exists, warn user if it is already exist
  if FileExists(FileName) then
  begin

    Write('File already exist, did you want to overwrite it? (y/n)');
    Readln(Ans);
    if upcase(Ans) = 'Y' then
      ReadyToCreate:= True
    else
      ReadyToCreate:= False;
  end
  else // File does not exist
    ReadyToCreate:= True;

  if ReadyToCreate then
  begin
    // Link file variable (F) with physical file (FileName)
    AssignFile(F, FileName);

    Rewrite(F); // Create new file for writing

    Writeln('Please input file contents line by line, '
     , 'when you finish write % then press enter');
```

```
    i:= 1;
    repeat
      Write('Line # ', i, ':');
      Inc(i);
      Readln(Line);
      if Line <> '%' then
        Writeln(F, Line); // Write line into text file
    until Line = '%';

    CloseFile(F); // Release F and FileName connection, flush buffer
  end
  else // file already exist and user does not want to overwrite it
    Writeln('Doing nothing');
  Write('Press enter key to close..');
  Readln;
end.
```

In this example, we have used many important things:

1. *Boolean* type:

```
  ReadyToCreate: Boolean;
```

This type can hold only one of two values: either *True* or *False*. These values can be used directly in *if condition*, *while* loop or *repeat* loop.

In the previous example, we have used the if condition like this:

```
  if Marks[i] > Max then
```

Which eventually turns to *True* or *False*.

2. *UpCase* function:

```
  if upcase(Ans) = 'Y' then
```

This statement is executed when the file exists. The program will warn the user about overwriting an existing file. If he/she wants to continue, then he/she should enter a lowercase *y* or capital *Y*. The *UpCase* function will convert the character into a capital letter if it is lowercase.

3. *Rewrite* procedure:

```
  Rewrite(F); // Create new file for writing
```

The Rewrite procedure is used to create a new empty file. If the file already exists, it will be erased and overwritten. It also opens the file for writing only in case of text files.

4. *Writeln(F, ..)* procedure:

```
Writeln(F, Line); // Write line into text file
```

This procedure is used to write string or variables in text file, and appends them with end of line characters, which are a carriage return/line feed combination (CR/LF), represented as the characters for the numbers 13 and 10 respectively.

These characters can not be displayed in a console window, but it will move the screen display cursor to a new line.

5. *Inc* procedure:

```
Inc(i);
```

This procedure increases an integer variable by one. It is equivalent to the statement:

```
i:= i + 1;
```

6. *CloseFile* procedure:

```
CloseFile(F); // Release F and FileName connection, flush buffer
```

As we mentioned earlier, the *CloseFile* procedure releases a file in the operating system. In addition, it has an additional job when writing to a text file, which is flushing the writing buffer.

Buffering of text files is a feature that makes dealing with text files faster. Instead of writing a single line or character directly to disk or any other storage media (*which is very slow compared with writing into memory*), the application will write these entries into a memory buffer. When the buffer reaches its full size, it will be flushed (forced to be written) into permanent storage media like a hard disk. This operation makes writing faster, but it will add the risk of losing some data (in the buffer) if the power is suddenly lost. To minimize data loss, we should close the file immediately after finishing writing to it, or calling the *Flush* procedure to flush the buffer explicitly.

Appending to a text file

In this example, we want to open an existing text file for writing at the end of it, without removing its original contents using the procedure *Append*.

Add to text file program

```
var
  FileName: string;
  F: TextFile;
  Line: string;
  i: Integer;
begin
  Write('Input an existed file name: ');
  Readln(FileName);
  if FileExists(FileName) then
  begin
    // Link file variable (F) with physical file (FileName)
    AssignFile(F, FileName);

    Append(F); // Open file for appending

    Writeln('Please input file contents line by line',
      'when you finish write % then press enter');
    i:= 1;
    repeat
      Write('Line # ', i, ' append :');
      Inc(i);
      Readln(Line);
      if Line <> '%' then
        Writeln(F, Line); // Write line into text file
    until Line = '%';
    CloseFile(F); // Release F and FileName connection, flush buffer
  end
  else
    Writeln('File does not exist');
  Write('Press enter key to close..');
  Readln;
end.
```

After we run this application and enter an existing text file, we can view it with the *cat / type* commands, or by double clicking on it in its directory to see the appended data.

Random access files

As we mentioned earlier, the second type of file according to an access type perspective is *Random access* , or *direct access*. This type of file has a fixed size record, so that we can jump to any record for reading or writing at any time.

There are two types of random access files: *typed* files and *untyped* files.

Typed files

Typed files are used for files that contain the same type of data that has the same size of records, for example, if a file contains records of *Byte* type, that means the record size = 1 byte. If the file contains real numbers (*Single*), that means all record sizes are 4 bytes, etc.

In the next example, we will show how to use file of *Byte*:

Marks program

```
var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  Rewrite(F); // Create file
  Writeln('Please input students marks, write 0 to exit');

  repeat
    Write('Input a mark: ');
    Readln(Mark);
    if Mark <> 0 then  // Don't write 0 value
      Write(F, Mark);
  until Mark = 0;
  CloseFile(F);

  Write('Press enter key to close..');
  Readln;
end.
```

In this example, we have used this syntax to define a typed file:

```
    F: file of Byte;
```

Which means: the file contains records of Byte data type, which can hold values from 0 to 255.

And we have created the file and opened it for writing using the *Rewrite* procedure:

```
  Rewrite(F); // Create file
```

Also we have used the function *Write* instead of *Writeln* to write records in the typed file:

```
      Write(F, Mark);
```

Writeln is not suitable for typed files, because it appends CR/LF characters on each line of written text, but *Write* stores the record as it is without any additions. In this case, if we have entered 10 records (of Byte), file size will be 10 bytes on disk.

In the next example, we will show how to display the previous file's contents:

Reading student marks

```
program ReadMarks;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    Reset(F); // Open file
    while not Eof(F) do
    begin
      Read(F, Mark);
      Writeln('Mark: ', Mark);
    end;
    CloseFile(F);
  end
  else
    Writeln('File (marks.dat) not found');

  Write('Press enter key to close..');
  Readln;
end.
```

In the next example, we will show how to append new records without deleting the existing data:

Appending student marks program

```
program AppendMarks;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    FileMode:= 2; // Open file for read/write
    Reset(F); // open file
    Seek(F, FileSize(F)); // Go to beyond last record
    Writeln('Please input students marks, write 0 to exit');

    repeat
      Write('Input a mark: ');
      Readln(Mark);
      if Mark <> 0 then  // Don't write 0 value in disk
        Write(F, Mark);
    until Mark = 0;
    CloseFile(F);
  end
  else
    Writeln('File marks.dat not found');

  Write('Press enter key to close..');
  Readln;
end.
```

After running this program and entering new records, we can run it again to see the appended data.

Note that we have used *Reset* for opening the file for writing instead of the *Rewrite* procedure. *Rewrite* erases all data for existing files, and creates an empty file if the file does not exist, while *Reset* can only open an existing file without erasing its contents.

Also we have assigned 2 to the *FileMode* variable to indicate that we need to open the file for read/write access mode. 0 in *FileMode* means read only, 1 means write only, 2 (*Default*) means read/write.

```
    FileMode:= 2; // Open file for read/write
    Reset(F); // open file
```

Reset puts the read/write pointer to the first record, and for that reason if we start to write records immediately, we will overwrite the old records, so we have used the *Seek* procedure to move the read/write pointer to the end of file. *Seek* can be used only with random access files.

If we try to access a record position that does not exist with the *Seek* procedure (for example, record number 100, while we have only 50 records), we will get an error.

We have also used the *FileSize* function, which returns the current record count in the file. It is used in combination with the *Seek* procedure to jump to the end of the file:

```
        Seek(F, FileSize(F)); // Go to after last record
```

Note that this example can be used if the Student marks file already exists; if not, then we should run the first program (Storing Student Marks) because it uses *Rewrite* which can create new files.

We can combine both methods (*Reset* and *Rewrite*) according to whether the file exists, as we have done in the next example:

Create and append student marks program

```
program ReadWriteMarks;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  F: file of Byte;
  Mark: Byte;
begin
  AssignFile(F, 'marks.dat');
  if FileExists('marks.dat') then
  begin
    FileMode:= 2; // Open file for read/write
    Reset(F); // open file
    Writeln('File already exist, opened for append');
    // Display file records
    while not Eof(F) do
    begin
      Read(F, Mark);
      Writeln('Mark: ', Mark);
    end
  end
```

```
  else // File not found, create it
  begin
    Rewrite(F);
    Writeln('File does not exist,, not it is created');
  end;

  Writeln('Please input students marks, write 0 to exit');
  Writeln('File pointer position at record # ', FilePos(f));
  repeat
    Write('Input a mark: ');
    Readln(Mark);
    if Mark <> 0 then  // Don't write 0 value
      Write(F, Mark);
  until Mark = 0;
  CloseFile(F);

  Write('Press enter key to close..');
  Readln;
end.
```

Note that in this example, we didn't use the *Seek* procedure, instead we read all file contents first. This operation (Read all file) moves the pointer to the end of file.

In the next example, we will use a file of records to store Cars information.

Cars database program

```
program CarRecords;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

type
  TCar = record
    ModelName: string[20];
    Engine: Single;
    ModelYear: Integer;
  end;

var
  F: file of TCar;
  Car: TCar;
begin
  AssignFile(F, 'cars.dat');
```

```
  if FileExists('cars.dat') then
  begin
    FileMode:= 2; // Open file for read/write
    Reset(F); // open file
    Writeln('File already exist, opened for append');
    // Display file records
    while not Eof(F) do
    begin
      Read(F, Car);
      Writeln;
      Writeln('Car # ', FilePos(F), ' --------------------------');
      Writeln('Model : ', Car.ModelName);
      Writeln('Year  : ', Car.ModelYear);
      Writeln('Engine: ', Car.Engine);
    end
  end
  else // File not found, create it
  begin
    Rewrite(F);
    Writeln('File does not exist, created');
  end;

  Writeln('Please input car informaion, ',
        'write x in model name to exit');
  Writeln('File pointer position at record # ', FilePos(f));

  repeat
    Writeln('--------------------------');
    Write('Input car Model Name : ');
    Readln(car.ModelName);
    if Car.ModelName <> 'x' then
    begin
      Write('Input car Model Year : ');
      Readln(car.ModelYear);
      Write('Input car Engine size: ');
      Readln(car.Engine);
      Write(F, Car);
    end;
  until Car.ModelName = 'x';
  CloseFile(F);

  Write('Press enter key to close..');
  Readln;
end.
```

In the previous example, we declared *TCar* type to define car information. The first field (*ModelName*) is a string variable, but we limited its maximum length [20]:

```
  ModelName: string[20];
```

We should use this declaration for string variables before we use them in files, because the default *ANSI String* variables has a different and unlimited storage type in memory, but for typed files, every data type width should be defined.

File copying

All file types, like text files, binary files, are based on byte units, which are the smallest representation of data in computer memory and on disk. Every file should contain one byte, two bytes, etc, or no bytes at all. Every byte could hold an integer or a character code from 0 to 255. We can open all file types using the *File of Byte* or *File of Char* declaration.

We can copy any file to another one using *File of Byte* files, and the result will be a new file that is identical to the source file's contents.

Copy files using file of byte

```
program FilesCopy;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };


var
  SourceName, DestName: string;
  SourceF, DestF: file of Byte;
  Block: Byte;
begin
  Writeln('Files copy');
  Write('Input source file name: ');
  Readln(SourceName);

  Write('Input destination file name: ');
  Readln(DestName);
  if FileExists(SourceName) then
  begin
    AssignFile(SourceF, SourceName);
    AssignFile(DestF, DestName);

    FileMode:= 0;   // open for read only
    Reset(SourceF); // open source file
    Rewrite(DestF); // Create destination file

    // Start copy
    Writeln('Copying..');
    while not Eof(SourceF) do
    begin
      Read(SourceF, Block); // Read Byte from source file
      Write(DestF, Block);  // Write this byte into new
```

```
                                                // destination file
    end;
    CloseFile(SourceF);
    CloseFile(DestF);

  end
  else // Source File not found
    Writeln('Source File does not exist');

  Write('Copy file is finished, press enter key to close..');
  Readln;
end.
```

After running the previous example, we should enter an existing source file and a new destination file. In Linux we could enter file names like this:

```
Input source file name: /home/motaz/quran/mishari/32.mp3
Input destination file name: /home/motaz/Alsajda.mp3
```

In Windows we could enter something like this:

```
Input source file name: c:\photos\mypphoto.jpg
Input destination file name: c:\temp\copy.jpg
```

If the source file exists in the same directory as the *FileCopy* program, we could enter only the file name like this:

```
Input source file name: test.pas
Input destination file name: testcopy.pas
```

If we use this method to copy large files, it will take a very long time compared with operating system copy procedures. That means the operating system uses a different technique to copy files. If we want to copy a 1 megabyte file, that means the *while* loop will repeat about 1 million times, that means a million read and a million write operations. If we replace the *file of Byte* declaration with *file of Word*, that means it will take about 500,000 cycles for read and write, but this will work only for the files whose size is even not odd. It will succeed if a file contains 1,420 bytes, but it will fail with a file of 1,423 bytes.

To copy a file of any kind using a faster method, we should use *untyped* files.

Untyped files

Untyped files are random access files,that have a fixed record length, but are not linked to any data type. Instead, they treat data (*records*) as an array of bytes or characters.

Copy files using untyped files program

```
program FilesCopy2;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  SourceName, DestName: string;
  SourceF, DestF: file;
  Block: array [0 .. 1023] of Byte;
  NumRead: Integer;
begin
  Writeln('Files copy');
  Write('Input source file name: ');
  Readln(SourceName);

  Write('Input destination file name: ');
  Readln(DestName);

  if FileExists(SourceName) then
  begin
    AssignFile(SourceF, SourceName);
    AssignFile(DestF, DestName);

    FileMode:= 0;   // open for read only
    Reset(SourceF, 1); // open source file
    Rewrite(DestF, 1); // Create destination file

    // Start copy
    Writeln('Copying..');
    while not Eof(SourceF) do
    begin
      // Read Byte from source file
      BlockRead(SourceF, Block, SizeOf(Block), NumRead);
      // Write this byte into new destination file
      BlockWrite(DestF, Block, NumRead);
    end;
    CloseFile(SourceF);
    CloseFile(DestF);
```

```
  end
  else // Source File not found
    Writeln('Source File does not exist');

  Write('Copy file is finished, press enter key to close..');
  Readln;
end.
```

New things in the previous example are:

1. Declaration type of files:

```
  SourceF, DestF: file;
```

2. Read/write variable (Buffer)

```
  Block: array [0 .. 1023] of Byte;
```

We have used an array of bytes (*1 Kilo bytes, and it can be modified*) to read and copy files blocks.

3. Opening an untyped file requires an additional parameter:

```
  Reset(SourceF, 1); // open source file
  Rewrite(DestF, 1); // Create destination file
```

The additional parameter is the record size, which is the minimum element that can be read/written at a time. Since we need to copy any file type, that means it should be one byte, because it can be used with any file size.

4. Read procedure:

```
  BlockRead(SourceF, Block, SizeOf(Block), NumRead);
```

The *BlockRead* procedure is used with untyped files. It reads a bunch of data at a time.

The parameters of the *BlockRead* procedure are:

- SourceF: This is the source file variable that we need to copy.
- Block : This is the variable or array that will store the current data being read and written.
- SizeOf(Block): This is the desired number of records that we need to read at one time. For example ,if we enter 100 that means we need to read 100 records (bytes on this case). If we use the *SizeOf* function, that means we need to read records typical to the number of the container (Array of Bytes).
- NumRead: We have told the *BlockRead* function to read a specific number of records (1024), and some times it succeeds in reading all of that amount, and sometimes it gets only part of it. For example, suppose that we need to read a file that contains only 100 bytes, that means

BlockRead could read only 100 bytes. Reading fewer records than desired happens also at the last block of the file. If the file contains 1034 bytes for example, that means in the first loop, we will get 1024, but in next loop we will get only 10 bytes left, and *Eof* function will return True. We need the *NumRead* value to use with the *BlockWrite* procedure.

5. Writing to untyped files:

```
  BlockWrite(DestF, Block, NumRead);
```

This is the write procedure, and it is similar to the *BlockRead* procedure, but it has some differences: 1. Instead of using the *SizeOf* procedure, we have used *NumRead*, because *NumRead* contains the actual read Block size. 2. The fourth parameter *NumWritten* (which is not used in this example) is not important, because we always get records written as we desire, unless the disk is full.

After we run this application, notice the speed of copying large files. If the file contains 1 Megabytes, that means we need only about one thousand reading/writing cycles to copy the entire file.

In the next example, we will use untyped files to read any file and display it as it is stored in memory or on disk. As we know, files are list of bytes.

Display file contents program

```
program ReadContents;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

var
  FileName: string;
  F: file;
  Block: array [0 .. 1023] of Byte;
  i, NumRead: Integer;
begin
  Write('Input source file name: ');
  Readln(FileName);

  if FileExists(FileName) then
  begin
    AssignFile(F, FileName);

    FileMode:= 0;   // open for read only
```

```
    Reset(F, 1);

    while not Eof(F) do
    begin
      BlockRead(F, Block, SizeOf(Block), NumRead);
      // display contents in screen
      for i:= 0 to NumRead - 1 do
        Writeln(Block[i], ':', Chr(Block[i]));
    end;
    CloseFile(F);

  end
  else // File does not exist
    Writeln('Source File does not exist');

  Write('press enter key to close..');
  Readln;
end.
```

After running this example and entering a text file name for example, we will see for the first time CR/LF values (13/10), because we display each character code (ASCII). In Linux we will find only line feed (LF) which has a value of 10 in decimal. This is a line delimiter in text files.

We can display other types of files, like pictures, executables, to see how they look from the inside.

We have used the *Chr* function in this example to get the numerical value of characters, for example the letter *a* is stored in memory in a byte as 97, and capital *A* is stored as 65.

Date and Time

Date and time are two of the most important issues in programming. It is important for the applications that store transaction or any operation information, like purchasing, bills payment, etc, they need to store the date and time of that transaction. Later they could determine the transactions and operations that happened during the last month or current month, for example.

An applications log is one operation that relies on date/time recording. We need to know when some application starts, stops, generates an error, or crashes.

TDateTime is the type in Object Pascal that we can use to store date/time information. It is a double precision floating point number that occupies 8 bytes in memory. The fractional part of this type represents time, and the integral part represents the number of days that have passed since 30/Dec/1899.

In the next example, we will show how to display the current date/time value using the *Now* function.

```
program DateTime;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes , SysUtils
  { you can add units after this };

begin
  Writeln('Current date and time: ', DateTimeToStr(Now));
  Write('Press enter key to close');
  Readln;
end.
```

We have used the *DateTimeToStr* function contained in the *SysUtils* unit to convert a *TDateTime* type returned by the *Now* function to a readable date/time string representation.

If we do not use this conversion, we will get an encoded date/time displayed as a real number:

```
  Writeln('Current date and time: ', Now);
```

There are also two date/time conversion functions that display the date part only and the time part only :

```
  Writeln('Current date is ', DateToStr(Now));
  Writeln('Current time is ', TimeToStr(Now));
```

The *Date* function returns only the date part of today, and the *Time* function returns the current time only:

```
  Writeln('Current date is ', DateToStr(Date));
  Writeln('Current time is ', TimeToStr(Time));
```

These two functions put zeroes in the other part, for example, the *Date* function returns the current day and puts zero in the time part (Zero in time means 12:00 am). The *Time* function returns the current system time, and puts zero in the date part (Zero in date means 30/12/1899).

We can check this by using *DateTimeToStr* function:

```
  Writeln('Current date is ', DateTimeToStr(Date));
  Writeln('Current time is ', DateTimeToStr(Time));
```

The *DateTimeToStr* displays date/time according to the computer's date/time configuration. Its result may vary between two computer systems, but the *FormatDateTime* function will display date and time in the format written by the programmer regardless of computer configuration:

```
  Writeln('Current date is ',
     FormatDateTime('yyyy-mm-dd hh:nn:ss', Date));
  Writeln('Current time is ',
    FormatDateTime('yyyy-mm-dd hh:nn:ss', Time));
```

In the next example, we will treat date/time as a real number, and we will add and subtract values from it:

```
begin
  Writeln('Current date and time is ',
    FormatDateTime('yyyy-mm-dd hh:nn:ss', Now));
  Writeln('Yesterday time is ',
    FormatDateTime('yyyy-mm-dd hh:nn:ss', Now - 1));
  Writeln('Tomorrow time is ',
    FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1));
  Writeln('Today + 12 hours is ',
    FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1/2));
  Writeln('Today + 6 hours is ',
    FormatDateTime('yyyy-mm-dd hh:nn:ss', Now + 1/4));
  Write('Press enter key to close');
  Readln;
end.
```

If we add one to or subtract one from a date, it will add/subtract a complete day. If we add, for example, ½ or 0.5, this will add half a day (12 hours).

In the next example, we will generate a date value using a specific year, month and day:

```
var
  ADate: TDateTime;
begin
  ADate:= EncodeDate(1975, 11, 16);
  Writeln('My date of birth is: ', FormatDateTime('yyyy-mm-dd', ADate));
  Write('Press enter key to close');
  Readln;
end.
```

The *EncodeDate* function accepts *year*, *month* and *days* as inputs, and returns year/month/day values in one variable of type *TDateTime*.

The *EncodeTime* function accepts *hour*, *minute*, *second* and *Milli second* and returns time values as one *TDateTime* value:

```
var
  ATime: TDateTime;
begin
  ATime:= EncodeTime(19, 22, 50, 0);
  Writeln('Almughrib prayer time is: ', FormatDateTime('hh:nn:ss', ATime));
  Write('Press enter key to close');
  Readln;
end.
```

Date/time comparison

You can compare two date/time variables the same as comparing real numbers. For example in real numbers: 9.3 is greater than 5.1, the same happens for *TDateTime* values. Now + 1, which represents tomorrow, is greater than today (Now), and Now + 1/24 which means an hour after now is greater than Now – 2/24 which means two hours before now.

In the next example, we will put the date 1/Jan/2012 in a variable and compare it with current date and check if this date is passed or not yet.

```
var
  Year2012: TDateTime;
begin
  Year2012:= EncodeDate(2012, 1, 1);

  if Now < Year2012 then
    Writeln('Year 2012 is not coming yet')
  else
    Writeln('Year 2012 is already passed');
  Write('Press enter key to close');
```

```
  Readln;
end.
```

We can add new functions to this example, which displays the remaining days or passed days to/from that date:

```
var
  Year2012: TDateTime;
  Diff: Double;
begin
  Year2012:= EncodeDate(2012, 1, 1);
  Diff:= Abs(Now - Year2012);

  if Now < Year2012 then
    Writeln('Year 2012 is not coming yet, there are ',
      Format('%0.2f', [Diff]), ' days Remaining ')
  else
    Writeln('First day of January 2012 is passed by ',
      Format('%0.2f', [Diff]), ' Days');
  Write('Press enter key to close');
  Readln;
end.
```

Diff is a real number variable that will hold the difference between current date and the 2012 date. We also used the *Abs* function, which returns the absolute value of a number (the number without the negative sign).

News recorder program

In this example, we will use text files to store News titles, and in addition, we will store date and time too.

After closing and opening the application again, it will display the previously entered news titles with their date/time:

```
program news;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
```

```
  { you can add units after this };

var
  Title: string;
  F: TextFile;
begin
  AssignFile(F, 'news.txt');
  if FileExists('news.txt') then
  begin       // Display old news
    Reset(F);
    while not Eof(F) do
    begin
      Readln(F, Title);
      Writeln(Title);
    end;
    CloseFile(F); // reading is finished from old news
    Append(F);   // open file again for appending
  end
  else
    Rewrite(F);

  Write('Input current hour news title: ');
  Readln(Title);
  Writeln(F, DateTimeToStr(Now), ', ', Title);
  CloseFile(F);

  Write('Press enter to close');
  Readln;
end.
```

Constants

Constants are similar to variables. They have names and can hold values, but differ from variables in modifying that value. Constant values can not be modified while running the application, and their values should be determined before compiling the application.

We have used constants in a different way before, without naming them, just as an Integer value or string as in this example:

```
  Writeln(5);
  Writeln('Hello');
```

The value 5 will never change after running the application. 'Hello' is also a string constant.

We can define constants using the *Const* keyword after the application's uses clause as in the following example:

Fuel Consumption program

```
program FuelConsumption;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };


const GallonPrice = 6.5;

var
  Payment: Integer;
  Consumption: Integer;
  Kilos: Single;
begin
  Write('How much did you pay for your car''s fuel: ');
  Readln(Payment);
  Write('What is the consumption of your car? (Kilos per Gallon): ');
  Readln(Consumption);

  Kilos:= (Payment / GallonPrice) * Consumption;

  Writeln('This fuel will keep your car running for : ',
    Format('%0.1f', [Kilos]), ' Kilometers');
  Write('Press enter');
  Readln;
end.
```

In the previous example, we will calculate the kilometers that the car could run with its current fuel according to these facts:

1. Car fuel consumption: we have used the *Consumption* variable to store kilometers per gallon for the current car
2. Fuel: we have used the *Payment* variable to store how much money we paid for the current fuel
3. Gallon Price: we have used the *GallonPrice* constant to store the price for a gallon of fuel for the current country. This value shouldn't be entered by the user; instead, it should be defined by the programmer.

Constants are recommended when using the same value many times in the application. If we need to change this value, we can do it once at the header of code.

Ordinal types

Ordinal types are integer values that use literal indications instead of numbers. For example, if we need to define language variables (Arabic/English/French) we could use the value 1 for Arabic, 2 for English, and 3 for French. Other programmers wouldn't know the values for 1, 2 and 3 unless they find comments with these values. It will be more readable if we do it with ordinal types as in the below example:

```
program OrdinalTypes;

{$mode objfpc}{$H+}


uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TLanguageType = (ltArabic, ltEnglish);

var
  Lang: TLanguageType;
  AName: string;
  Selection: Byte;
begin
  Write('Please select Language: 1 (Arabic), 2 (English)');
  Readln(Selection);

  if Selection = 1 then
    Lang:= ltArabic
  else
  if selection = 2 then
    Lang:= ltEnglish
  else
    Writeln('Wrong entry');

  if Lang = ltArabic then
    Write('ماهو اسمك: ')
  else
    if Lang = ltEnglish then
      Write('What is your name: ');

  Readln(AName);

  if Lang = ltArabic then
  begin
    Writeln('مرحباً بك ', AName);
    Write('الرجاء الضغط على مفتاح إدخال لإغلاق البرنامج');
  end
  else
  if Lang = ltEnglish then
```

```
  begin
    Writeln('Hello ', AName);
    Write('Please press enter key to close');
  end;
  Readln;

end.
```

Integer, *character* and *boolean* types are ordinal types, while real numbers and strings are not.

Sets

Set types can hold multiple properties or characteristics in one variable. Sets are used only with ordinal values.

For example, if we need to define the operating system's support for applications, we can do it as in the following:

1. Define ordinal type that represents operating systems: *TApplicationEnv*:

```
  TApplicationEnv = (aeLinux, aeMac, aeWindows);
```

2. Define the application as set of *TApplicationEnv*:
   for example:

```
  FireFox: set of TApplicationEnv;
```

3. Put operating system values in the application set variable:

```
  FireFox:= [aeLinux, aeWindows];
```

```
program Sets;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

type
  TApplicationEnv = (aeLinux, aeMac, aeWindows);

var
  FireFox: set of TApplicationEnv;
  SuperTux: set of TApplicationEnv;
  Delphi: set of TApplicationEnv;
  Lazarus: set of TApplicationEnv;
begin
  FireFox:= [aeLinux, aeWindows];
  SuperTux:= [aeLinux];
  Delphi:= [aeWindows];
  Lazarus:= [aeLinux, aeMac, aeWindows];

  if aeLinux in Lazarus then
    Writeln('There is a version for Lazarus under Linux')
  else
    Writeln('There is no version of Lazarus under linux');

  if aeLinux in SuperTux then
    Writeln('There is a version for SuperTux under Linux')
```

```
  else
    Writeln('There is no version of SuperTux under linux');

  if aeMac in SuperTux then
    Writeln('There is a version for SuperTux under Mac')
  else
    Writeln('There is no version of SuperTux under Mac');

  Readln;
end.
```

Also we can use set syntax for other ordinal types, like Integers:

```
  if Month in [1, 3, 5, 7, 8, 10, 12] then
    Writeln('This month contains 31 days');
```

Or character:

```
if Char in ['a', 'A'] then
    Writeln('This letter is A');
```

Exception handling

There are two types of errors: compilation errors like using a variable without defining it in the Var section, or writing statements with incorrect syntax. These types of errors prevent the application from compiling, and the compiler displays a proper message and points to the line containing the error.

The second type is the runtime error. This type of error occurs while the application is running, for example, division by zero, In a line like this:

```
x:= y / z;
```

This is valid syntax, but at run time a user could enter 0 in the *Z* variable, and then the application will crash and display a *division by zero* error message.

Trying to open a nonexistent file will generate a runtime error also (*File not found*), or trying to create a file in a read only directory.

The compiler cannot catch such errors, which only occur after running the application.

To build a reliable application that will not crash from runtime errors, we should use exception handling.

There are different methods of exception handling in Object Pascal:

Try except statement

Syntax:

```
try
  // Start of protected code
  CallProc1;
  CallProc2;
  // End of protected code

except
on e: exception do // Exception handling
begin
  Writeln('Error: ' + e.message);
end;
end;
```

Example for division:

```
program ExceptionHandling;

{$mode objfpc}{$H+}
```

```
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, sysutils
  { you can add units after this };


var
  x, y: Integer;
  Res: Double;
begin
  try
    Write('Input x: ');
    Readln(x);
    Write('Input y: ');
    Readln(y);
    Res:= x / y;
    Writeln('x / y = ', Res);

  except
  on e: exception do
  begin
    Writeln('An error occurred: ', e.message);
  end;
  end;
  Write('Press enter key to close');
  Readln;
end.
```

There are two sections of the try statement. The first one is the block that we need to protect, which resides between *try* … *except*.The other part exists between *except* .. *end*.

If something goes wrong in the first section (*try except*), the application will go to the *except* section (*except.. end*) and it will not crash, but rather it will display the proper error message and continue execution.

This is the line that could raise the exception if the value of *y* is zero:

```
    Res:= x / y;
```

If there is no exception, the *except* .. *end* part will not be executed.

Try finally

Syntax:

```
try
  // Start of protected code
  CallProc1;
  CallProc2;
```

```
  // End of protected code

finally
  Writeln('This line will be printed in screen for sure');
end;
```

Division program using *try finally* method:

```
program ExceptionHandling;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };

var
  x, y: Integer;
  Res: Double;
begin
  try
    Write('Input x: ');
    Readln(x);
    Write('Input y: ');
    Readln(y);
    Res:= x / y;
    Writeln('x / y = ', Res);

  finally
    Write('Press enter key to close');
    Readln;
  end;
end.
```

This time the *finally* .. *end* part will be executed in all cases, regardless of whether there is an error.

Raise an exception

Sometimes we need to generate/raise an exception to prevent some logical error. For example if the user enters the value 13 for Month variable, we could raise an exception telling him/her that he/she has violated the range of months.

Example:

```
program RaiseExcept;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };


var
  x: Integer;
begin
  Write('Please input a number from 1 to 10: ');
  Readln(X);
  try

    if (x < 1) or (x > 10) then  // rais exception
      raise exception.Create('X is out of range');
    Writeln(' X * 10 = ', x * 10);

  except
  on e: exception do  // Catch my exception
  begin
    Writeln('Error: ' + e.Message);
  end;
  end;
  Write('Press enter to close');
  Readln;

end.
```

If the user enters a value outside of 1 to 10, an exception will be generated (*X is out of range*), and if there is no exception handling for this part, the application will crash. Because we have written *try except* around the code that contains the *raise* keyword, the application will not crash, but instead it will display the error message.

Chapter Two

Structured Programming

Introduction

Structured programming appeared after the expansion of programs. Large applications become unreadable and unmaintainable when they use unstructured code in one file.

In structured programming, we can split an application's source code into smaller pieces, called *procedures* and *functions*, and a we can also combine the procedures and functions that relate to one subject in a separate code file called a *unit*.

Structured programming benefits:

1. **Partitioning** an application's code to readable modules and procedures.

2. Code **re-usability**: procedures and functions can be called from any part of the code many times without the need to re-write and duplicate code.

3. Programmers could **share** and **participate** in one project at the same time. Each programmer could write their own procedures and functions in a separate unit, then they can integrate these units in the project.

4. Application **maintenance** and **enhancement** becomes easy: we can find bugs easily in procedures, also it is easy to improve procedures and functions, write new ones, add new units, etc.

5. Introducing **modules** and **layers**: we can divide an application into different logical layers and modules: for example we can write a unit that contains procedures that read/write data from/into files, and another unit that represent rules and validation layer, and a third one as a user interface layer.

Procedures

We have already used some *procedures* in the previous chapter, like *Writeln*, *Readln*, *Reset*, etc, but this time we need to write our own *procedures* that can be used by our applications.

In the next example we have written two procedures: *SayHello* and *SayGoodbye*:

```
program Structured;

{$mode objfpc}{$H+}
```

```
uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };


procedure SayHello;
begin
  Writeln('Hello there');
end;

procedure SayGoodbye;
begin
  Writeln('Good bye');
end;

begin  // Here main application starts
  Writeln('This is the main application started');
  SayHello;
  Writeln('This is structured application');
  SayGoodbye;
  Write('Press enter key to close');
  Readln;
end.
```

We see that the procedure looks like a small program, with its own *begin..end*, and it can be called from the main application's code.

Parameters

In the next example, we introduce *parameters*, which are variables passed to a procedure when calling it:

```
procedure WriteSumm(x, y: Integer);
begin
  Writeln('The summation of ', x, ' + ', y, ' = ', x + y)
end;

begin
  WriteSumm(2, 7);
  Write('Press enter key to close');
  Readln;
end.
```

In the main application, we have called the *WriteSumm* procedure and passed the values *2*, *7* to it, and the procedure will receive them in *x*, *y* integer variables to write the summation result of them.

In the next example, we have rewritten the restaurant application using procedures:

Restaurant program using procedures

```
procedure Menu;
begin
  Writeln('Welcome to Pascal Restaurant. Please select your order');
  Writeln('1 - Chicken      (10$)');
  Writeln('2 - Fish         (7$)');
  Writeln('3 - Meat         (8$)');
  Writeln('4 – Salad        (2$)');
  Writeln('5 - Orange Juice (1$)');
  Writeln('6 - Milk         (1$)');
  Writeln;
end;

procedure GetOrder(AName: string; Minutes: Integer);
begin
  Writeln('You have ordered : ', AName, ', this will take ',
    Minutes, ' minutes');
end;


// Main application

var
  Meal: Byte;
begin

  Menu;
  Write('Please enter your selection: ');
  Readln(Meal);

  case Meal of
    1: GetOrder('Chicken', 15);
    2: GetOrder('Fish', 12);
    3: GetOrder('Meat', 18);
    4: GetOrder('Salad', 5);
    5: GetOrder('Orange juice', 2);
    6: GetOrder('Milk', 1);
  else
    Writeln('Wrong entry');
  end;
  Write('Press enter key to close');
  Readln;
end.
```

Now the main application becomes smaller and more readable. The details of the other parts are separated in procedures, like the *get order* and *display menu* procedures.

Functions

Functions are similar to *procedures*, but have an additional feature, which is returning a value. We have used functions before, like *UpperCase,* which converts and returns text to upper case, and *Abs,* which returns the absolute value of a number.

In the next example, we have written the *GetSumm* function, which receives two integer values and returns their summation:

```
function GetSumm(x, y: Integer): Integer;
begin
  Result:= x + y;
end;

var
  Sum: Integer;
begin
  Sum:= GetSumm(2, 7);
  Writeln('Summation of 2 + 7 = ', Sum);
  Write('Press enter key to close');
  Readln;
end.
```

Notice that we have declared the function as Integer, and we have used the *Result* keyword to represent the function's return value.

In the main application, we have used the variable *Sum*, in which we receive the function result, but we could eliminate this intermediate variable and call this function inside the *Writeln* procedure. This is one difference between functions and procedures. We can call functions as input parameters in other procedures or functions, but we can not call procedures as parameters of other functions and procedures:

```
function GetSumm(x, y: Integer): Integer;
begin
  Result:= x + y;
end;

begin
  Writeln('Summation of 2 + 7 = ', GetSumm(2, 7));
  Write('Press enter key to close');
  Readln;
end.
```

In the next example, we have rewritten the Restaurant program using functions:

Restaurant program using functions

```
procedure Menu;
begin
  Writeln('Welcome to Pascal Restaurant. Please select your order');
  Writeln('1 - Chicken      (10$)');
  Writeln('2 - Fish         (7$)');
  Writeln('3 - Meat         (8$)');
  Writeln('4 – Salad        (2$)');
  Writeln('5 - Orange Juice (1$)');
  Writeln('6 - Milk         (1$)');
  Writeln('X - nothing');
  Writeln;
end;

function GetOrder(AName: string; Minutes, Price: Integer): Integer;
begin
  Writeln('You have ordered: ', AName, ' this will take ',
    Minutes, ' minutes');
  Result:= Price;
end;


var
  Selection: Char;
  Price: Integer;
  Total: Integer;
begin
  Total:= 0;
  repeat
    Menu;
    Write('Please enter your selection: ');
    Readln(Selection);

    case Selection of
      '1': Price:= GetOrder('Checken', 15, 10);
      '2': Price:= GetOrder('Fish', 12, 7);
      '3': Price:= GetOrder('Meat', 18, 8);
      '4': Price:= GetOrder('Salad', 5, 2);
      '5': Price:= GetOrder('Orange juice', 2, 1);
      '6': Price:= GetOrder('Milk', 1, 1);
      'x', 'X': Writeln('Thanks');
      else
      begin
        Writeln('Wrong entry');
        Price:= 0;
      end;
    end;

    Total:= Total + Price;

  until (Selection = 'x') or (Selection = 'X');
  Writeln('Total price           = ', Total);
  Write('Press enter key to close');
```

```
  Readln;
end.
```

Local Variables

We can define variables locally inside a procedure or function to be used only inside its code. These variables can not be accessed from the main application's code or from other procedures and functions.

Example:

```
procedure Loop(Counter: Integer);
var
  i: Integer;
  Sum: Integer;
begin
  Sum:= 0;
  for i:= 1 to Counter do
    Sum:= Sum + i;
  Writeln('Summation of ', Counter, ' numbers is: ', Sum);
end;

begin   // Main program section
  Loop;
  Write('Press enter key to close');
  Readln;
end.
```

In the procedure Loop, there are two local variables *Sum* and *I*. Local variables are stored in Stack memory, which is a part of memory that allocates variables temporarily until the procedure's execution is finished. That means it will be unaccessible and can be overwritten when program execution reaches this line of code:

```
  Write('Press enter key to close');
```

Global variables can be accessed from the main program and other procedures and functions. They can hold values until the application is closed, but this can break the structure of the program and make it hard to trace errors, because any procedure can change global variable values, which may result in unknown values and misbehavior when we forget to initialize them.

Defining local variables guarantees their privacy, which helps the procedures and functions to be ported or called from anywhere without worry about global variables values.

News database application

In this example we have three procedures and one function: *Add News title*, *display all News*, *searching*, and *displaying menu* to let the user select the function that he/she want to execute:

```
program news;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this };

type
  TNews = record
    ATime: TDateTime;
    Title: string[100];
  end;

procedure AddTitle;
var
  F: file of TNews;
  News: TNews;
begin
  AssignFile(F, 'news.dat');
  Write('Input current news title: ');
  Readln(News.Title);
  News.ATime:= Now;
  if FileExists('news.dat') then
  begin
    FileMode:= 2; // Read/Write
    Reset(F);
    Seek(F, System.FileSize(F)); // Go to last record to append
  end
  else
    Rewrite(F);
  Write(F, News);
  CloseFile(F);
end;

procedure ReadAllNews;
var
  F: file of TNews;
  News: TNews;
begin
  AssignFile(F, 'news.dat');
  if FileExists('news.dat') then
  begin
    Reset(F);
    while not Eof(F) do
    begin
      Read(F, News);
      Writeln('------------------------------');
```

```
      Writeln('Title: ', News.Title);
      Writeln('Time : ', DateTimeToStr(News.ATime));
    end;
    CloseFile(F);
  end
  else
    Writeln('Empty database');
end;


procedure Search;
var
  F: file of TNews;
  News: TNews;
  Keyword: string;
  Found: Boolean;
begin
  AssignFile(F, 'news.dat');
  Write('Input keyword to search for: ');
  Readln(Keyword);
  Found:= False;
  if FileExists('news.dat') then
  begin
    Reset(F);
    while not Eof(F) do
    begin
      Read(F, News);
      if Pos(LowerCase(Keyword), LowerCase(News.Title)) > 0 then
      begin
        Found:= True;
        Writeln('------------------------------');
        Writeln('Title: ', News.Title);
        Writeln('Time : ', DateTimeToStr(News.ATime));
      end;
    end;
    CloseFile(F);
    if not Found then
      Writeln(Keyword, ' Not found');
  end
  else
    Writeln('Empty database');
end;

function Menu: char;
begin
  Writeln;
  Writeln('..........News database.........');
  Writeln('1. Add news title');
  Writeln('2. Display all news');
  Writeln('3. Search');
  Writeln('x. Exit');
  Write('Please input a selection : ');
  Readln(Result);
end;
```

```
// Main application

var
  Selection: Char;

begin

  repeat
    Selection:= Menu;
    case Selection of
      '1': AddTitle;
      '2': ReadAllNews;
      '3': Search;
    end;
  until LowerCase(Selection) = 'x';
end.
```

This application is easy to read and has compact and clear code in the main section. We can also add new features to any procedure without affecting or modifying other parts.

Functions as input parameters

As we said before, we can call the function as a procedure/function input parameter, because we can treat it as a value.

See the example below:

```
function DoubleNumber(x: Integer): Integer;
begin
  Result:= x * 2;
end;

// Main

begin
  Writeln('The double of 5 is : ', DoubleNumber(5));
  Readln;
end.
```

Note that we have called the *DoubleNumber* function from inside the *Writeln* procedure.

In the next modification of the example, we will use an intermediate variable to store the function's result, and then use it as input to the *Writeln* procedure:

```
function DoubleNumber(x: Integer): Integer;
begin
  Result:= x * 2;
end;

// Main

var
  MyNum: Integer;
begin
  MyNum:= DoubleNumber(5);
  Writeln('The double of 5 is : ', MyNum);
  Readln;
end.
```

We can also call functions within if conditions and loop conditions:

```
function DoubleNumber(x: Integer): Integer;
begin
  Result:= x * 2;
end;

// Main

begin
  if DoubleNumber(5) > 10 then
    Writeln('This number is larger than 10')
```

```
  else
    Writeln('This number is equal or less than 10);
  Readln;
end.
```

Procedure and function output parameters

In the previous usage of functions, we found that we can return only one value, which is the result of the function, but how can we return more than one value in functions or procedures?

Let us do this experiment:

```
procedure DoSomething(x: Integer);
begin
  x:= x * 2;
  Writeln('From inside procedure: x = ', x);
end;

//  main

var
  MyNumber: Integer;
begin
  MyNumber:= 5;

  DoSomething(MyNumber);
  Writeln('From main program, MyNumber = ', MyNumber);
  Writeln('Press enter key to close');
  Readln;
end.
```

In this example, the *doSomething* procedure receives *x* as an Integer value, then it multiplies it by two, and then finally it displays it.

In the main part of the program, we declared the variable *MyNumber* as an Integer, put the number *5* in it, and then passed it as a parameter to the *DoSomething* procedure. In this case, the *MyNumber* value (5) will be copied into the *x* variable.

After calling the function, *X* 'svalue will be 10, but when we display *MyNumber* after procedure calling we will find that it still holds the value *5*. That means *MyNumber* and *X* have two different locations in memory, which is normal.

This type of parameter passing is called calling by value, which does not affect the original parameter *MyNumber.* We also could use constants to pass such values, e.g:

```
DoSomething(5);
```

Calling by reference

If we add the *var* keyword to the declaration of *DoSomething's x* parameter, things will be different now:

```
procedure DoSomething(var x: Integer);
begin
  x:= x * 2;
  Writeln('From inside procedure: x = ', x);
end;

//  main

var
  MyNumber: Integer;
begin
  MyNumber:= 5;

  DoSomething(MyNumber);
  Writeln('From main program, MyNumber = ', MyNumber);
  Writeln('Press enter key to close');
  Readln;

end.
```

This time *MyNumber's* value will be changed according to *x*, which means they are sharing the same memory location.

We should pass a variable (not a constant) to the procedure this time, using the same type: if the parameter is declared as Byte, then *MyNumber* should be declared as Byte; if it is Integer, then it should be Integer.

The next example will generate an error when calling *DoSomething,* which requires a variable for its parameter:

```
  DoSomething(5);
```

In *calling by value*, the previous code could be used, because it cares only about having a value passed as its parameter, and 5 is a value, but in *calling by reference* the program cares about having a variable passed as its parameter and then acts on its value.

In the next example, we will pass two variables, and then the procedure will swap their values:

```
procedure SwapNumbers(var x, y: Integer);
var
  Temp: Integer;
begin
  Temp:= x;
  x:= y;
  y:= Temp;
```

```
end;

//  main

var
  A, B: Integer;
begin

  Write('Please input value for A: ');
  Readln(A);

  Write('Please input value for B: ');
  Readln(B);

  SwapNumbers(A, B);
  Writeln('A = ', A, ', and B = ', B);
  Writeln('Press enter key to close');
  Readln;
end.
```

Units

A *Unit* in Pascal is a library that contains procedures, functions, constants, and user defined types that can be used in many applications.

Units are used to achieve these goals:

1. Accumulate procedures and functions that are frequently used in applications in external units. This achieves code re-usability in software development.

2. Combine procedures and functions that are used to perform certain tasks in one entity. Instead of populating the main application's source code with unrelated procedures, it is better to divide the application into logical modules using units.

To create a new unit, go to File/New Unit in the Lazarus menu, and Lazarus creates this template:

```
unit Unit1;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

implementation

end.
```

After creating a new unit, we should save it using a specific name, like *Test*. It will be saved in a file named *Test.pas*, but the unit name will remain *Test* in the application.

Then we can start writing procedures, functions, and other re-usable code:

```
unit Test;

{$mode objfpc}{$H+}

interface


uses
  Classes, SysUtils;

const
  GallonPrice = 6.5;

  function GetKilometers(Payment, Consumption: Integer): Single;
```

```
implementation


function GetKilometers(Payment, Consumption: Integer): Single;
begin
  Result:= (Payment / GallonPrice) * Consumption;
end;

end.
```

We have written the *GallonPrice* constant and the *GetKilometers* function to be called from any program.

Also we have put the function's header in the *Interface* part of the unit to make it accessible from outside the unit. Applications can access only the *Interface* part of units.

To use this unit, we have created a new program in the same directory as the unit (Test.pas), and then we have added this unit in *uses* clause:

```
program PetrolConsumption;

{$mode objfpc}{$H+}


uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils
  { you can add units after this }, Test;


var
  Payment: Integer;
  Consumption: Integer;
  Kilos: Single;
begin
  Write('How much did you pay for your car''s petrol: ');
  Readln(Payment);
  Write('What is the consumption of your car (Kilos per one Gallon) ');
  Readln(Consumption);

  Kilos:= GetKilometers(Payment , Consumption);

  Writeln('This petrol will keep your car running for: ',
    Format('%0.1f', [Kilos]), ' Kilometers');
  Write('Press enter');
  Readln;
end.
```

If we need to go to the *GetKilometers* function's code, we can press the *Ctrl* key then click its name with the mouse to display the *GetKilometers* function's source code. Lazarus/Delphi will immediately open the *Test* unit and display that function.

We can also open the *Test* unit in the editor by moving the cursor to the unit's name (*Test*), and then press *Alt + Enter*.

We can access units from programs in these conditions:

1. The Unit file exists in the same directory, as the application, as we have done in the previous example.
2. Adding the unit to the project, by opening the unit in Lazarus then clicking Project/Add Editor File to project.
3. Adding the unit's path in Project/Compiler Options/Compiler Options/Paths/Other Unit Files.

Units in Lazarus and Free Pascal

Units represent one of the most important building blocks in Lazarus and Free Pascal. We find that most procedures, functions, types, classes and components exist in *units*.

Free Pascal has a very rich collection of units that provide functionality used in all different types of applications. For that reason it is fast and easy to develop applications using Lazarus and Free Pascal.

Examples of Free Pascal/ Lazarus units that contain general procedures and functions: *SysUtils*, and *Classes*.

Units written by the programmer

Programmers can use Lazarus units and they can write their own units. *Units* that are written by programmers represent their special needs for their applications. For example, if a developer is writing software for a car garage, then he/she could make a *unit* that contains procedures for adding a new car, searching for a car using plate id, etc.

Putting procedures and functions in *units* makes the application more readable and easier to be developed by more than one developer, because each one could concentrate on one module (*unit*) or more, then they could test their *units* functionality independently (*Unit testing*), then eventually they could integrate these *units* together in one project.

Hejri Calendar

The *Hejri* calender is based on Moon months, and was created by Muslims. We need to create a unit that helps us to convert the *Gregorian* calendar to the Moon (*Hejri*) Calender based on these facts:

1. The first day of *Hejri* 16 July 622 in the *Gregorian* calender.
2. The *Hejri* year contains 354.367056 days.
3. The *Hejri* month contains 29.530588 days.

The *Hejri* unit can be used to get current the moon phase.

This is the code of the *Hejri* unit:

```
{
*******************************************************************************

  HejriUtils:   Hejri Calnder converter, for FreePascal and Delphi
  Author:       Motaz Abdel Azeem
  email:        motaz@code.sd
  Home page:    http://motaz.freevar.com/
  License:      LGPL
  Created on:   26.Sept.2009
  Last modifie: 26.Sept.2009

*******************************************************************************
}

unit HejriUtils;

{$IFDEF FPC}
{$mode objfpc}{$H+}
{$ENDIF}

interface

uses
  Classes, SysUtils;

const
  HejriMonthsEn: array [1 .. 12] of string = ('Moharram', 'Safar', 'Rabie Awal',
    'Rabie Thani', 'Jumada Awal', 'Jumada Thani', 'Rajab', 'Shaban', 'Ramadan',
    'Shawal', 'Thi-Alqaida', 'Thi-Elhajah');

  HejriMonthsAr: array [1 .. 12] of string = ('محرم' , 'صفر' ,'ربيع أول',
    'ربيع ثاني' ,'جمادى الأول' ,'جمادى الآخر' ,'رجب' ,'شعبان' ,'رمضان',
    'شوال' ,'ذي القعدة' ,'ذي الحجة');

  HejriYearDays = 354.367056;
  HejriMonthDays = 29.530588;

  procedure DateToHejri(ADateTime: TDateTime; var Year, Month, Day: Word);
  function HejriToDate(Year, Month, Day: Word): TDateTime;

  procedure HejriDifference(Year1, Month1, Day1, Year2, Month2, Day2: Word;
```

```
    var YearD, MonthD, DayD: Word);


implementation

var
  HejriStart : TDateTime;

procedure DateToHejri(ADateTime: TDateTime; var Year, Month, Day: Word);
var
  HejriY: Double;
  Days: Double;
  HejriMonth: Double;
  RDay: Double;
begin
  HejriY:= ((Trunc(ADateTime) - HejriStart - 1) / HejriYearDays);
  Days:= Frac(HejriY);
  Year:= Trunc(HejriY) + 1;
  HejriMonth:= ((Days * HejriYearDays) / HejriMonthDays);
  Month:= Trunc(HejriMonth) + 1;
  RDay:= (Frac(HejriMonth) * HejriMonthDays) + 1;
  Day:= Trunc(RDay);
end;

function HejriToDate(Year, Month, Day: Word): TDateTime;
begin
  Result:= (Year - 1) * HejriYearDays + (HejriStart - 0) +
    (Month - 1) * HejriMonthDays + Day + 1;
end;

procedure HejriDifference(Year1, Month1, Day1, Year2, Month2, Day2: Word; var
  YearD, MonthD, DayD: Word);
var
  Days1: Double;
  Days2: Double;
  DDays: Double;
  RYear, RMonth: Double;
begin
  Days1:= Year1 * HejriYearDays + (Month1 - 1) * HejriMonthDays + Day1 - 1;
  Days2:= Year2 * HejriYearDays + (Month2 - 1) * HejriMonthDays + Day2 - 1;
  DDays:= Abs(Days2 - Days1);

  RYear:= DDays / HejriYearDays;
  RMonth:= (Frac(RYear) * HejriYearDays) / HejriMonthDays;
  DayD:= Round(Frac(RMonth) * HejriMonthDays);

  YearD:= Trunc(RYear);
  MonthD:= Trunc(RMonth);

end;

initialization

  HejriStart:= EncodeDate(622, 7, 16);

end.
```

The *HejriUtils* Unit contains these procedures and functions:

1. *DateToHejri*: This procedure is used to convert Gregorian date to *Hejri* date, example of using it:

```
program Project1;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes  , HejriUtils, SysUtils
  { you can add units after this };

var
  Year, Month, Day: Word;
begin
  DateToHejri(Now, Year, Month, Day);
  Writeln('Today in Hejri: ', Day, '-', HejriMonthsEn[Month],
    '-', Year);
  Readln;
end.
```

2. *HejriToDate*: This function is used to convert the *Hejri* date to a Gregorian *TDateTime* value.
3. *HejriDifference*: This procedure is used to calculate the difference in *years*, *days*, and *months* between two *Hejri* dates.

Procedure and function Overloading

Overloading mean we can write two or more procedures/functions with the same name but different parameters. Different parameters means different parameter types or a different number of parameters.

For example, we may need to write two versions of the Sum function, where the first one accepts and returns integer numbers, and the second one accepts and returns real numbers:

```
program sum;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };


function Sum(x, y: Integer): Integer; overload;
begin
  Result:= x + y;
end;

function Sum(x, y: Double): Double; overload;
begin
  Result:= x + y;
end;

var
  j, i: Integer;
  h, g: Double;
begin
  j:= 15;
  i:= 20;
  Writeln(J, ' + ', I, ' = ', Sum(j, i));

  h:= 2.5;
  g:= 7.12;
  Writeln(H, ' + ', G, ' = ', Sum(h, g));

  Write('Press enter key to close');
  Readln;

end.
```

Notice that we have used the reserved word *overload,* which means: this function is overloaded.

Default value parameters

We can put default values for procedures and functions in parameters. In this case we can ignore sending these parameters, and the default values will be used.

Look at the next example:

```
program defaultparam;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };


function SumAll(a, b: Integer; c: Integer = 0;
  d: Integer = 0): Integer;
begin
  result:= a + b + c + d;
end;

begin
  Writeln(SumAll(1, 2));
  Writeln(SumAll(1, 2, 3));
  Writeln(SumAll(1, 2, 3, 4));
  Write('Press enter key to close');
  Readln;
end.
```

In this example, we have called the *SumAll* function three times, the first time using two values, the second time using three values, and the third time using four values. *A* and *B* are mandatory parameters and should be sent in all cases because they have no default values.

We can ignore sending default parameter values from left to right, for example, if we want to ignore sending a value for *c*, then we should not send a value for *d*.

Default parameters should start from right to left. We can not declare a default parameter and declare another one that has no default value next to it, like the following example:

```
function ErrorParameters(a: Integer; b: Integer = 10; c: Integer; x: string);
```

We should put the most important parameters on the left, and unimportant ones (that can be ignored) on the right side of the function/procedure's header.

Sorting

Data sorting is part of the data structure topic, and we introduce it here because it needs procedures and functions to implement it.

Sorting is always used with arrays and lists. Suppose that we have an array of integers, and we need to sort it in ascending or in descending order. We can do this using different methods:

Bubble sort algorithm

This algorithm is one of the simplest sorting algorithms. It compares the first item of array with second one, and in case of ascending order, it will swap the first one with second if the first is greater than the second. After that it compares the second item with third one until it reaches the end of the array. Then it repeats this operation until no swap operation happens, which means the array is already sorted.

Suppose that we have 6 items in an array containing these values:

```
7
10
2
5
6
3
```

We will find that these values need more than one cycle to be sorted.

The first number will be compared with the second one, and a swap operation will occur if the first one is greater than the second, and then the second will be compared with third, until the fifth item is compared with the sixth one.

After the first cycle, the array will be like this:

```
7
2
5
6
3
10
```

After the second cycle:

```
2
5
6
```

```
3
7
10
```

Third:

```
2
5
3
6
7
10
```

Fourth:

```
2
3
5
6
7
10
```

We will get a sorted array after the fourth cycle, because in the fourth cycle no swap operation happens. That means only three cycles are required with that data, but the fourth one is used to check for the occurrence of a sort.

Notice that the small values float like bubbles over the larger values.

Next is a *bubble sort* program:

```
program BubbleSortProj;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes;

procedure BubbleSort(var X: array of Integer);
var
  Temp: Integer;
  i: Integer;
  Changed: Boolean;
begin
  repeat  // Outer loop
    Changed:= False;
    for i:= 0 to High(X) - 1 do // Inner loop
      if X[i] > X[i + 1] then
```

```
      begin
        Temp:= X[i];
        X[i]:= X[i + 1];
        X[i + 1]:= Temp;
        Changed:= True;
      end;
  until not Changed;
end;

var
  Numbers: array [0 .. 9] of Integer;
  i: Integer;
begin
  Writeln('Please input 10 random numbers');
  for i:= 0 to High(Numbers) do
  begin
    Write('#', i + 1, ': ');
    Readln(Numbers[i]);
  end;

  BubbleSort(Numbers);
  Writeln;
  Writeln('Numbers after sort: ');

  for i:= 0 to High(Numbers) do
  begin
    Writeln(Numbers[i]);
  end;
  Write('Press enter key to close');
  Readln;
end.
```

We can modify this sort to a *descending* sort by changing the *greater than (>)* operator to *less than (<)*, in this line:

```
   if X[i] < X[i + 1] then
```

In the next example, we will sort students' grades from larger to smaller marks (d*escending order*):

Sorting students' marks

```
program smSort;   // Students' marks sort

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };
```

```
type
  TStudent = record
    StudentName: string;
    Mark: Byte;
  end;


procedure BubbleSort(var X: array of TStudent);
var
  Temp: TStudent;
  i: Integer;
  Changed: Boolean;
begin
  repeat
    Changed:= False;
    for i:= 0 to High(X) - 1 do
      if X[i].Mark < X[i + 1].Mark then
      begin
        Temp:= X[i];
        X[i]:= X[i + 1];
        X[i + 1]:= Temp;
        Changed:= True;
      end;
  until not Changed;
end;

var
  Students: array [0 .. 9] of TStudent;
  i: Integer;

begin
  Writeln('Please input 10 student names and marks');
  for i:= 0 to High(Students) do
  begin
    Write('Student #', i + 1, ' name : ');
    Readln(Students[i].StudentName);

    Write('Student #', i + 1, ' mark : ');
    Readln(Students[i].Mark);
    Writeln;
  end;

  BubbleSort(Students);
  Writeln;
  Writeln('Marks after Bubble sort: ');
  Writeln('-----------------------');

  for i:= 0 to High(Students) do
  begin
    Writeln('# ', i + 1, '   ', Students[i].StudentName,
      ' with mark (', Students[i].Mark, ')');
  end;
  Write('Press enter key to close');
  Readln;
end.
```

The *Bubble sort* algorithm is very simple, and programmers could memorize it. It is also suitable only

for a small amount of data, or data that is nearly sorted. In the case of a large amount of unsorted data, it could take a long time, so that it is better to use another algorithm.

Selection Sort algorithm

This type of sorting is similar to the *bubble sort*, but it is faster with a large amount of data. It contains two loops, the outer loop and the inner loop. In the inner loop, the cycles decrease in every outer loop cycle until it reaches only two cycles.

```
program SelectionSort;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes
  { you can add units after this };


procedure SelectSort(var X: array of Integer);
var
  i: Integer;
  j: Integer;
  SmallPos: Integer;
  Smallest: Integer;
begin
  for i:= 0 to High(X) -1 do // Outer loop
  begin
    SmallPos:= i;
    Smallest:= X[SmallPos];
    for j:= i + 1 to High(X) do // Inner loop
      if X[j] < Smallest then
      begin
        SmallPos:= j;
        Smallest:= X[SmallPos];
      end;
      X[SmallPos]:= X[i];
      X[i]:= Smallest;
  end;
end;



// Main application

var
  Numbers: array [0 .. 9] of Integer;
  i: Integer;

begin
  Writeln('Please input 10 random numbers');
  for i:= 0 to High(Numbers) do
```

```
  begin
    Write('#', i + 1, ': ');
    Readln(Numbers[i]);
  end;

  SelectSort(Numbers);
  Writeln;
  Writeln('Numbers after Selection sort: ');

  for i:= 0 to High(Numbers) do
  begin
    Writeln(Numbers[i]);
  end;
  Write('Press enter key to close');
  Readln;
end.
```

In spite of that, it is faster than a *bubble sort*, but a *bubble sort* becomes faster (fewer outer cycles) if the data is sorted or semi-sorted, while a *selection sort* always has the same number of cycles regardless of the data's initial order.

Shell sort algorithm

This is a very fast sort when there is a large amount of data, and its behavior is similar to a *bubble sort* if the data is semi-sorted, but it is more complicated than the two previous sort algorithms.

Its name comes from its inventor, Donald Shell.

```
program ShellSort;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes;


procedure ShellS(var X: array of Integer);
var
  Done: Boolean;
  Jump, j, i: Integer;
  Temp: Integer;
begin
  Jump:= High(X);
  while (Jump > 0) do // Outer loop
  begin
    Jump:= Jump div 2;
```

```
    repeat          // Intermediate loop
      Done:= True;
      for j:= 0 to High(X) - Jump do // Inner loop
      begin
        i:= j + Jump;
        if X[j] > X[i] then // Swap
        begin
          Temp:= X[i];
          X[i]:= X[j];
          X[j]:= Temp;
          Done:= False;
        end;

      end; // end of inner loop
    until Done; // end of intermediate loop
  end; // end of outer loop
end;

var
  Numbers: array [0 .. 9] of Integer;
  i: Integer;

begin
  Writeln('Please input 10 random numbers');
  for i:= 0 to High(Numbers) do
  begin
    Write('#', i + 1, ': ');
    Readln(Numbers[i]);
  end;

  ShellS(Numbers);
  Writeln;
  Writeln('Numbers after Shell sort: ');

  for i:= 0 to High(Numbers) do
  begin
    Writeln(Numbers[i]);
  end;
  Write('Press enter key to close');
  Readln;
end.
```

String sorting

We can sort strings like names, address, etc., using the same comparison operators (> and <) that we used to sort integers.

Comparison occurs from the left characters to the right, for example the letter *A* is smaller than *B*, and the name '*Ahmed*' is smaller than '*Badr*'. If two strings have the same first letters, then comparison will go to the next character. Example '*Ali*' and '*Ahmed*', this time *h* is smaller than *l*.

Sorting students name program

```
program smSort;   // Students mark sort by name

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes;

type
  TStudent = record
    StudentName: string;
    Mark: Byte;
  end;


procedure BubbleSort(var X: array of TStudent);
var
  Temp: TStudent;
  i: Integer;
  Changed: Boolean;
begin
  repeat
    Changed:= False;
    for i:= 0 to High(X) - 1 do
      if X[i].StudentName > X[i + 1].StudentName then
      begin
        Temp:= X[i];
        X[i]:= X[i + 1];
        X[i + 1]:= Temp;
        Changed:= True;
      end;
  until not Changed;
end;

var
  Students: array [0 .. 9] of TStudent;
  i: Integer;
```

```
begin
  Writeln('Please input 10 student names and marks');
  for i:= 0 to High(Students) do
  begin
    Write('Student #', i + 1, ' name : ');
    Readln(Students[i].StudentName);

    Write('Student #', i + 1, ' mark : ');
    Readln(Students[i].Mark);
    Writeln;
  end;

  BubbleSort(Students);
  Writeln;
  Writeln('Marks after Bubble sort: ');
  Writeln('----------------------');

  for i:= 0 to High(Students) do
  begin
    Writeln('# ', i + 1, '  ', Students[i].StudentName,
     ' with mark (', Students[i].Mark, ')');
  end;
  Write('Press enterkey to close');
  Readln;
end.
```

Sort algorithms comparison

In this example, we will compare the three sorting algorithms that we have mentioned in this chapter with a large array of Integer. We will measure the time used by each algorithm:

```
program SortComparison;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Classes, SysUtils;

//  Bubble sort

procedure BubbleSort(var X: array of Integer);
var
  Temp: Integer;
  i: Integer;
  Changed: Boolean;
begin
  repeat
    Changed:= False;
```

```
    for i:= 0 to High(X) - 1 do
      if X[i] > X[i + 1] then
      begin
        Temp:= X[i];
        X[i]:= X[i + 1];
        X[i + 1]:= Temp;
        Changed:= True;
      end;
  until not Changed;
end;


// Selection Sort

procedure SelectionSort(var X: array of Integer);
var
  i: Integer;
  j: Integer;
  SmallPos: Integer;
  Smallest: Integer;
begin
  for i:= 0 to High(X) -1 do // Outer loop
  begin
    SmallPos:= i;
    Smallest:= X[SmallPos];
    for j:= i + 1 to High(X) do // Inner loop
      if X[j] < Smallest then
      begin
        SmallPos:= j;
        Smallest:= X[SmallPos];
      end;
      X[SmallPos]:= X[i];
      X[i]:= Smallest;
  end;
end;

// Shell Sort

procedure ShellSort(var X: array of Integer);
var
  Done: Boolean;
  Jump, j, i: Integer;
  Temp: Integer;
begin
  Jump:= High(X);
  while (Jump > 0) do // Outer loop
  begin
    Jump:= Jump div 2;
    repeat          // Intermediate loop
      Done:= True;
      for j:= 0 to High(X) - Jump do // Inner loop
      begin
        i:= j + Jump;
        if X[j] > X[i] then // Swap
        begin
          Temp:= X[i];
          X[i]:= X[j];
          X[j]:= Temp;
```

```
          Done:= False;
        end;

      end; // inner loop
    until Done; // innermediate loop end
  end; // outer loop end
end;

// Randomize Data

procedure RandomizeData(var X: array of Integer);
var
  i: Integer;
begin
  Randomize;
  for i:= 0 to High(X) do
    X[i]:= Random(10000000);
end;

var
  Numbers: array [0 .. 59999] of Integer;
  i: Integer;
  StartTime: TDateTime;

begin

  Writeln('----------------- Full random data');
  RandomizeData(Numbers);
  StartTime:= Now;
  Writeln('Sorting.. Bubble');
  BubbleSort(Numbers);
  Writeln('Bubble sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
  Writeln;

  RandomizeData(Numbers);
  Writeln('Sorting.. Selection');
  StartTime:= Now;
  SelectionSort(Numbers);
  Writeln('Selection sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
  Writeln;

  RandomizeData(Numbers);
  Writeln('Sorting.. Shell');
  StartTime:= Now;
  ShellSort(Numbers);
  Writeln('Shell sort tooks (mm:ss): ',
   FormatDateTime('nn:ss', Now - StartTime));

  Writeln;
  Writeln('----------------- Nearly sorted data');
  Numbers[0]:= Random(10000);
  Numbers[High(Numbers)]:= Random(10000);
  StartTime:= Now;
  Writeln('Sorting.. Bubble');
  BubbleSort(Numbers);
  Writeln('Bubble sort tooks (mm:ss): ',
```

```
    FormatDateTime('nn:ss', Now - StartTime));
  Writeln;

  Numbers[0]:= Random(10000);
  Numbers[High(Numbers)]:= Random(10000);
  Writeln('Sorting.. Selection');
  StartTime:= Now;
  SelectionSort(Numbers);
  Writeln('Selection sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));
  Writeln;

  Numbers[0]:= Random(10000);
  Numbers[High(Numbers)]:= Random(10000);
  Writeln('Sorting.. Shell');
  StartTime:= Now;
  ShellSort(Numbers);
  Writeln('Shell sort tooks (mm:ss): ',
    FormatDateTime('nn:ss', Now - StartTime));

  Write('Press enterkey to close');
  Readln;
end.
```

Chapter Three

The Graphical User Interface

Introduction

The Graphical User Interface (GUI) is a new alternative to the console interface. It contains *forms*, *buttons*, *message boxes*, *menus*, *check boxes* and graphical components. It is easier for users to use GUI applications than console applications.

The GUI is used for business applications, operating system applications, games, and development tools like Lazarus, and many other applications.

Our First GUI application

To create a new GUI application, click on the Lazarus menu:

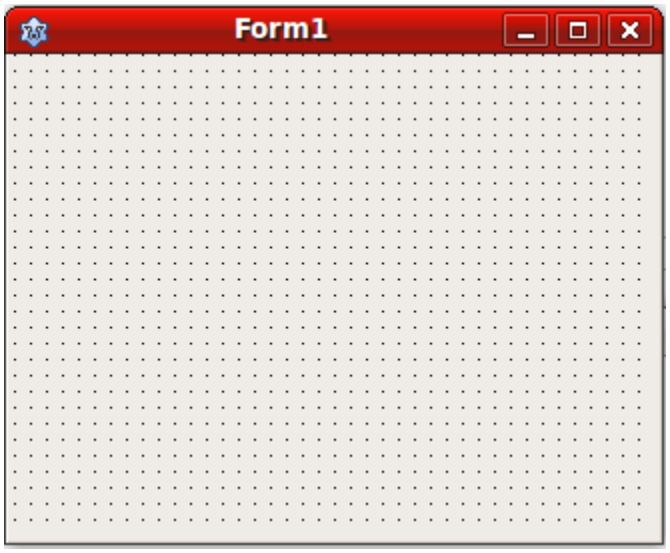
Project/New Project/Application

Then save the application by clicking on:

File/Save All

We can then create a new folder, such as *firstgui,* in which to save our project files. Then we should save the main unit, for example, *main.pas*, and finally we should choose a project name, such as *firstgui.lpi.*

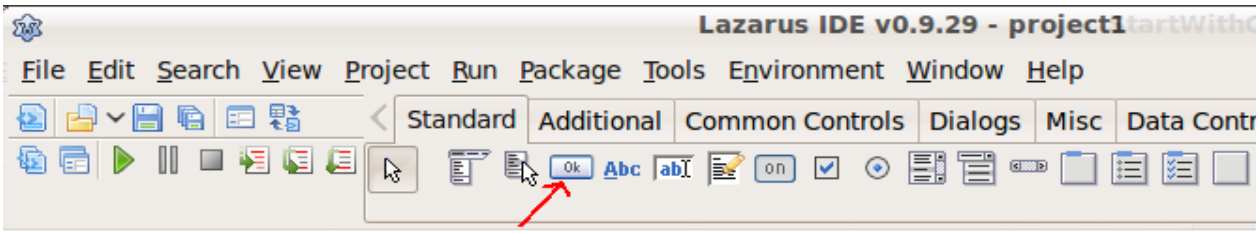
In the main unit, we can press F12 to view the form associated with the main unit, which will look like this:



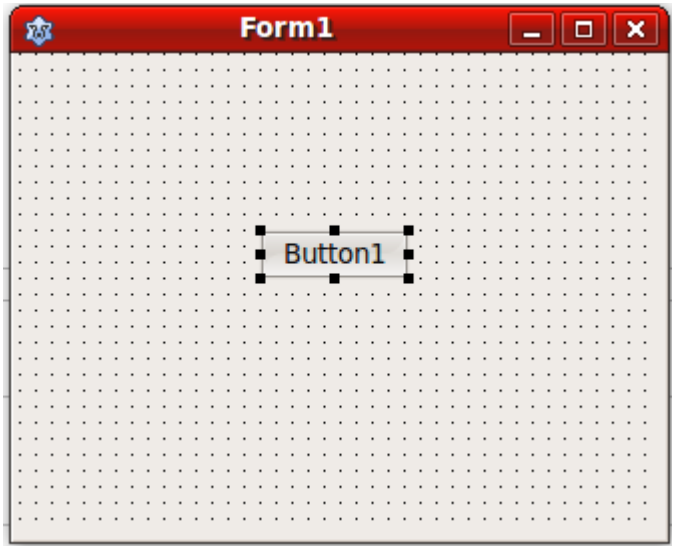
If we run the application, we will get this window:

Then we can close the application to return to the designer.

After that, we drop a *button* from standard component page on the form:

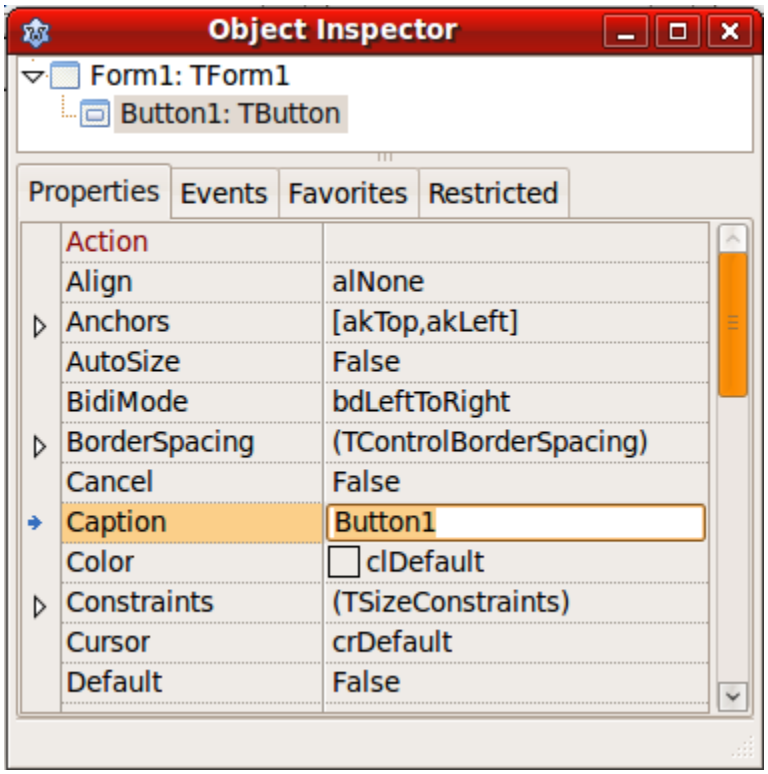


Then put the button at any place on the form like this picture:



We can display the button's properties (like caption name, width, etc) in the *Object Inspector* window. If it does not appear in Lazarus's windows, then we can display it by clicking *Window/Object Inspector*

from the main menu or by pressing *F11*. Then we will get this window:



We should make sure that we have selected the *button* not the *form* before we do any modification to its properties. Notice the Inspector window has tabs, the first being *Properties* and the second being *Events*. Each tab displays its own page.

Then we can change the text that is displayed in the button by changing its *Caption* property to *Click*.

After that, select the *Events* tab of the *Object Inspector* to display the *Events* page and double click the *OnClick* event, which will create this code template in the main unit:

```
procedure TForm1.Button1Click(Sender: TObject);
begin

end;
```

Then write this line in the *OnClick* event code template that Lazarus created for us:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Hello world, this is my first GUI application');
end;
```

By doing this, when we run the application, it will display a dialog window containing the desired message when we click the button with our mouse.

In Linux, we will find *firstgui* as an executable file in the same project directory, and *firstgui.exe* if we are using Windows. These are the executable files that we can copy to another computer and can run without the need of the Lazarus tool.

In the previous example, there are many important issues:

1. Main application file: which is stored in the *firstgui.lpi* file in the example. We can show it by clicking Project/Source. We will get this source code:

```
program firstgui;

{$mode objfpc}{$H+}

uses
  {$IFDEF UNIX}{$IFDEF UseCThreads}
  cthreads,
  {$ENDIF}{$ENDIF}
  Interfaces, // this includes the LCL widgetset
  Forms
  { you can add units after this }, main;

begin
  Application.Initialize;
  Application.CreateForm(TForm1, Form1);
  Application.Run;
end.
```

We may need to modify this code in certain cases for special purposes, but in most cases, we can leave it as it is, and let Lazarus manage it automatically.

2. Main unit: It is the unit that contains the definition of the form that appears automatically when we run the application.

   Below is the main unit code after we have completed the code for the *OnClick* event:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics, Dialogs,
  StdCtrls;

type

  { TForm1 }

  TForm1 = class(TForm)
```

```
    Button1: TButton;
    procedure Button1Click(Sender: TObject);
  private
    { private declarations }
  public
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{ TForm1 }

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Hello world, this is my first GUI application');
end;

initialization
  {$I main.lrs}

end.
```

In the main unit header, we find the *TForm1* declaration, which is similar to the *record* type, but is a *Class*. We will talk about classes in the next chapter: Object Oriented Programming. *Button1* is declared inside the *TForm1* class.

We can show this unit source code by pressing *Ctrl-F12* then selecting *main* unit.

3. Object Inspector/Properties: In this page of the Object Inspector, we can view and modify any component's properties, like a Button's caption or location, a form's color, or a labels font, etc. These components are similar to records, and their properties are similar to a record's fields.

4. Object Inspector/Events: This page of the Object Inspector contains a component's events that we can receive, like *On Click* event in a button, *Key Press* of an edit box, *double click* in a label, etc. When we click on any event of a component, Lazarus creates a new procedure template for us to write the code that will be called when this event occurs. We can write Pascal code for this event like the example below for a button's *OnClick* event:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage('Hello world, this is my first GUI application');
end;
```

This procedure will be called when the user clicks that button. This procedure is called an event handler.

Second GUI application

In this example, we will let the user enter his/her name in an edit box, and then when he/she clicks on a button, a greeting message will be displayed in a label.

To write this application, do the following:

- Create a new application and save it as *inputform*. Save the main unit as *main.pas*, then drop these components on the form from the standard component tab:

2 Labels
Edit box
Button

Then modify some properties of the above components as shown:

Form1:
Name: fmMain
Caption: Input form application

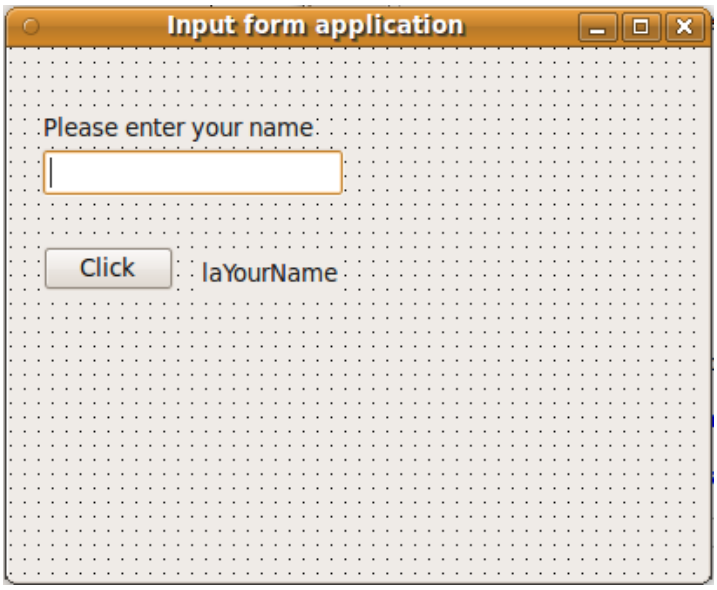
Label1:
Caption: Please enter your name

Label2:
Name: laYourName

Edit1:
Name: edName
Text:

Button1:
Caption: Click

Put the components on the form to look like the picture below:



- Put this code in the *OnClick* event handler:

```
procedure TfmMain.Button1Click(Sender: TObject);
begin
  laYourName.Caption:= 'Hello ' + edName.Text;
end;
```

Then we can run the application and write our name in the edit box, and then click the button.

In the previous example, we used the field *Text* in the edit box *edName* to read what the user types. This is the graphical alternative to the *Readln* procedures used in console applications to receive user input.

We also used the *Caption* property of the label *laYourName* to display a message. This is one of the alternative methods to *Writeln* in GUI applications.

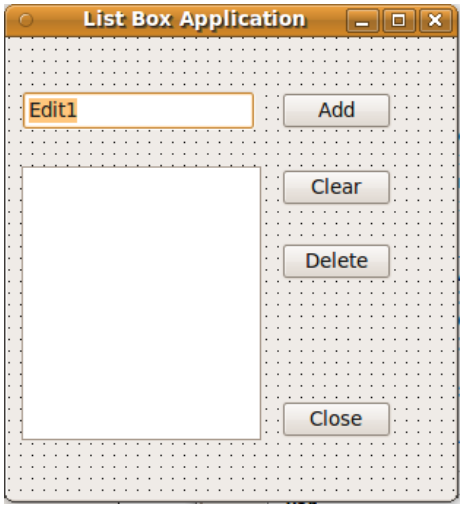
ListBox application

In the next example, we need to add text in a list box, delete it, and clear the list. To do this application follow these steps:

- Create a new application and put 4 buttons, an Edit box, and a List Box (*TListBox*) on its main form
- Change the name of buttons to these names:

```
btAdd, btClear, btDelete, btClose
```

- Change the captions of the buttons according to their names as shown in this figure:



- Write these event handlers for the buttons' *OnClick* events:

```
procedure TForm1.btAddClick(Sender: TObject);
begin
  ListBox1.Items.Add(Edit1.Text);
end;

procedure TForm1.btClearClick(Sender: TObject);
begin
  ListBox1.Clear;
end;

procedure TForm1.btDeleteClick(Sender: TObject);
var
  Index: Integer;
begin
  Index:= ListBox1.ItemIndex;
  if Index <> -1 then
    ListBox1.Items.Delete(Index);
end;

procedure TForm1.btCloseClick(Sender: TObject);
begin
```

```
  Close;
end;
```

Clicking the Add button will insert the text of the edit box into the list. The Delete button will delete current selected list item. The Clear button will clear the entire list. Finally, the Close button will close the application.

Text Editor Application

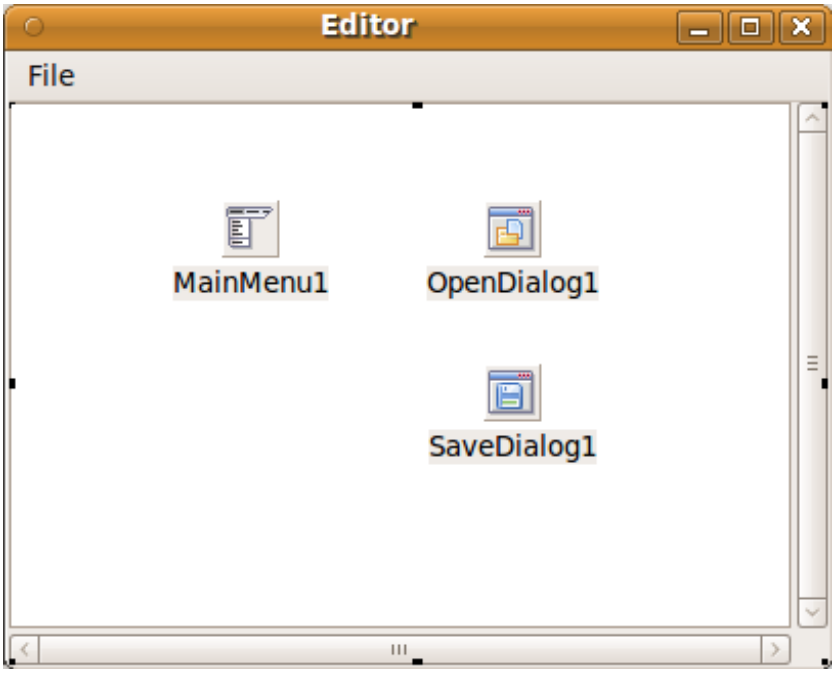
In this example, we will show how to create a simple text editor application.
Follow these steps:

- Create a new application and put these components on it's main form:

  - TMainMenu
  - TMemo: Change it's align property to *alClient* and ScrollBars to *ssBoth*
  - TOpenDialog and TSaveDialog from Dialogs page of components palette.

- Double click on the *MainMenu1* component and add *File* menu and a sub menu containing *Open File*, *Save File*, and *Close* menu items.

Our form will look like this:



- For the *Open File* item's *OnClick* event write this code:

```
if OpenDialog1.Execute then
  Memo1.Lines.LoadFromFile(OpenDialog1.FileName);
```

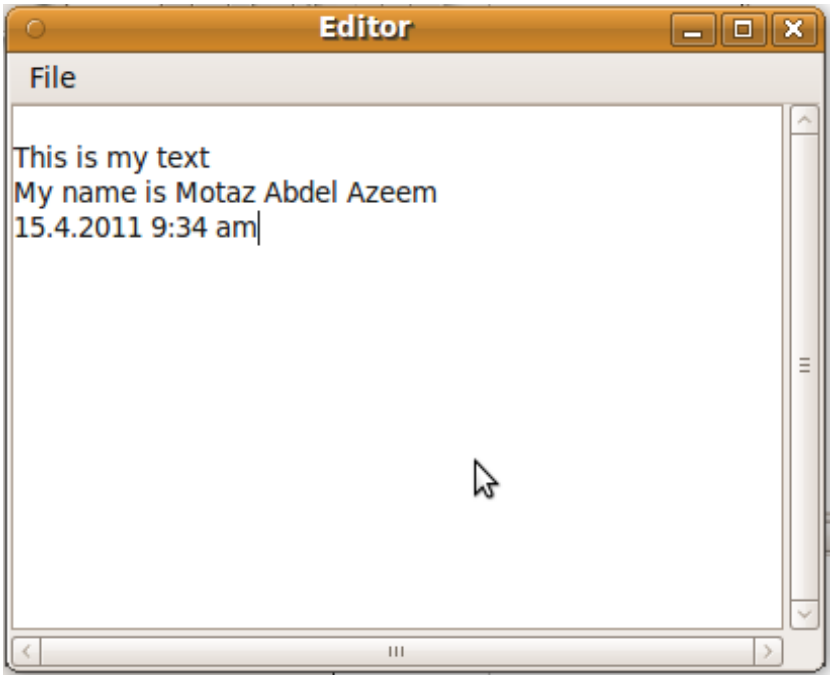
- For the *Save File* item write this code:

```
if SaveDialog1.Execute then
  Memo1.Lines.SaveToFile(SaveDialog1.FileName);
```

- For the *Close* item write:

```
Close;
```

After running this text editor application, we can write any text and then save it to disk. We also can open existing text files, like *.pas* files:



News Application

This time we need to write an application for storing news titles using these steps:

- Create a new application and name it *gnews*
- Add two buttons of type type *TButton*
- Add a text box (*TEdit*)
- Add memo (*TMemo*)
- Change the component's properties according to the following values:

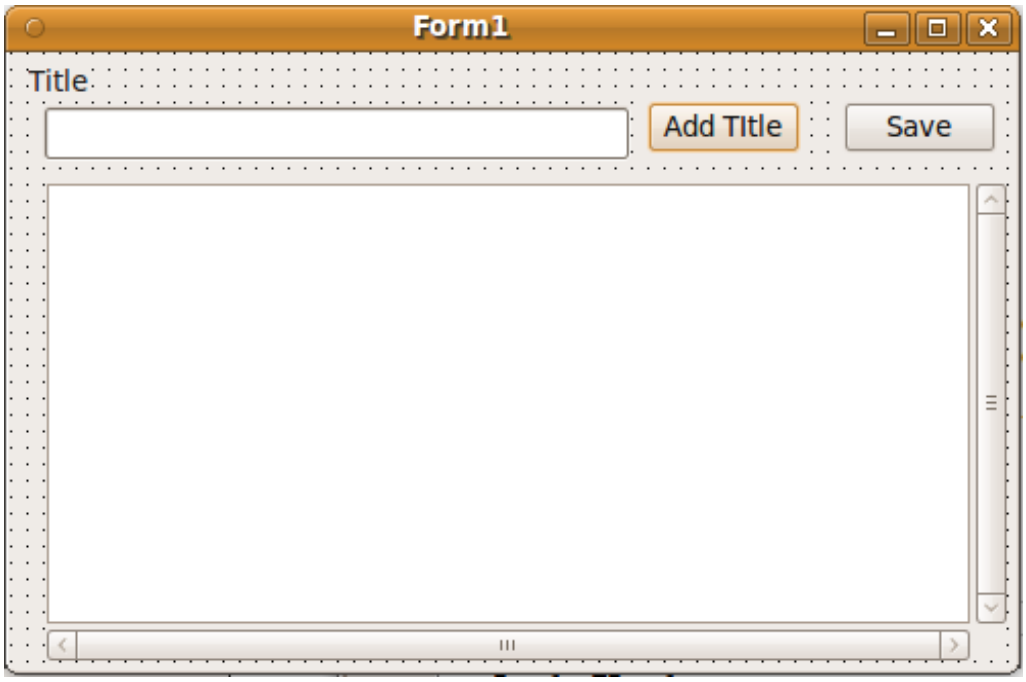
Button1
Caption: Add Title

Button2
Caption: Save
Anchors: Left=False, Right=True

Edit1:
Text=

Memo1
ScrollBars: ssBoth
ReadOnly: True
Anchors: Top=True, Left=Ture, Right=True, Bottom=Ture

Then we will get a form like this:



- For the *Add Title* button's *OnClick* event write this code:

```
Memo1.Lines.Insert(0,
  FormatDateTime('yyyy-mm-dd hh:nn', Now) + ': ' + Edit1.Text);
```

- For the *Save* button's *OnClick* event write:

```
Memo1.Lines.SaveToFile('news.txt');
```

- For the main form's *OnClose* event write this code to save entered news:

```
Memo1.Lines.SaveToFile('news.txt');
```

- For the main form's *OnCreate* event write the code that loads previously saved news titles if any exist:

```
if FileExists('news.txt') then
  Memo1.Lines.LoadFromFile('news.txt');
```

Application with a Second form

In the previous GUI applications, we have used only one form, but in real applications sometimes we need multiple forms.

To write a multiple formGUI application, do the following steps:

1. Create a new application and save it in a new folder called *secondform.*
2. Save the main unit as *main.pas*, and name the form component as *fmMain*. Save the project as *secondform.lpi.*
3. Add a new form by clicking *File / New Form*. Save this new unit as *second.pas* and name its form component *fmSecond*.
4. Add a label in the second form and write in its *Caption* property '**Second Form**'. Increase its font size from the label's *Font.Size* property.
5. Go back to the main form and put a button on it.
6. Add this line after main unit *implementation* section:

```
uses second;
```

7. For the *OnClick* event of the button, write this code:

```
fmSecond.Show;
```

Then run the application and click the button to display the second form.

Chapter Four

Object Oriented Programming

Introduction

In Object Oriented Programming, we describe the entities of an application as objects. For example, we can represent car information as an object that contains model name, model year, price, and this object has the ability to save this data in a file.

The object contains:

1. **Properties** which store status information., These values can be stored in variables.
2. Procedures and functions which are called **Methods**. These methods represents the actions that can be done in this object.
3. **Events**: these events could be received by the object, like the mouse moves over the object, a mouse click, etc
4. **Event handlers**: These are the procedures that will be executed if an event occurs.

Properties and methods that are related to each other can be represented as one object:

Object = Code + Data

An example of Object Oriented Programming is the GUI that we have used in the previous chapter. In that chapter we have used a lot of objects like *buttons*, *labels*, and *forms*. Each object contains properties like *Caption*, *Width*, and have methods like, *Show*, *Hide*, *Close*, etc. Also they have events like *OnClick*, *OnCreate*, *OnClose*, etc. The code that is written by the programmer to respond to specific events like the *OnClick* code represents event handlers.

First example: Date and Time

We have written an object that contains date and time with actions that work for date and time.
We have created a new unit which is called *DateTimeUnit*, and we have created a *class* in it called *TmyDateTime*.

Class is the type of an object. If we need to use that class, we should declare an instance of it. This instance is called *object*, the same as we have done with *Integer*, and *String* types. We declare instances of classes as variables (*I*, *J*, *Address*, etc) in order to use them.

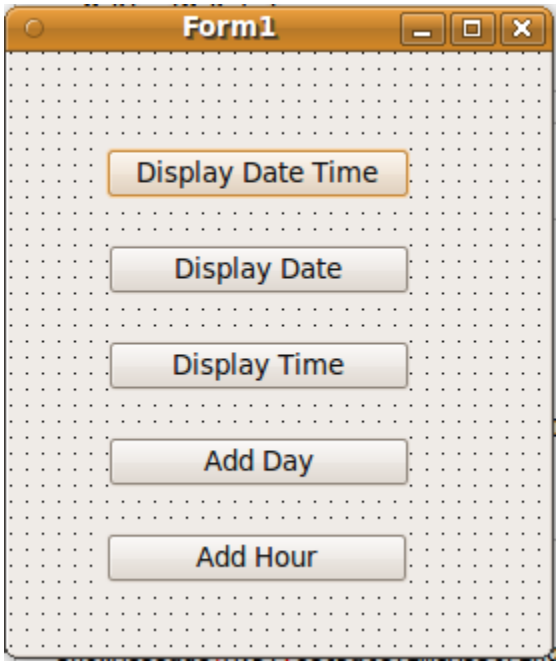
This is the code of unit:

```
unit DateTimeUnit;

{$mode objfpc}{$H+}

interface

uses
```

```
  Classes, SysUtils;

type
  { TMyDateTime }

  TMyDateTime = class
  private
    fDateTime: TDateTime;
  public
    function GetDateTime: TDateTime;
    procedure SetDateTime(ADateTime: TDateTime);
    procedure AddDays(Days: Integer);
    procedure AddHours(Hours: Single);
    function GetDateTimeAsString: string;
    function GetTimeAsString: string;
    function GetDateAsString: string;
    constructor Create(ADateTime: TDateTime);
    destructor Destroy; override;
  end;

implementation

{ TMyDateTime }

function TMyDateTime.GetDateTime: TDateTime;
begin
  Result:= fDateTime;
end;

procedure TMyDateTime.SetDateTime(ADateTime: TDateTime);
begin
  fDateTime:= ADateTime;
end;

procedure TMyDateTime.AddDays(Days: Integer);
begin
  fDateTime:= fDateTime + Days;
end;

procedure TMyDateTime.AddHours(Hours: Single);
begin
  fDateTime:= fDateTime + Hours / 24;
end;

function TMyDateTime.GetDateTimeAsString: string;
begin
  Result:= DateTimeToStr(fDateTime);
end;

function TMyDateTime.GetTimeAsString: string;
begin
  Result:= TimeToStr(fDateTime);
end;

function TMyDateTime.GetDateAsString: string;
begin
  Result:= DateToStr(fDateTime);
end;
```

```
constructor TMyDateTime.Create(ADateTime: TDateTime);
begin
  fDateTime:= ADateTime;
end;

destructor TMyDateTime.Destroy;
begin
  inherited Destroy;
end;

end.
```

We have put five buttons on the main form as shown:



We have written the following code in the *OnClick* event of each button:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics, Dialogs,
  StdCtrls, DateTimeUnit;

type

  { TForm1 }
```

```
  TForm1 = class(TForm)
    Button1: TButton;
    Button2: TButton;
    Button3: TButton;
    Button4: TButton;
    Button5: TButton;
    procedure Button1Click(Sender: TObject);
    procedure Button2Click(Sender: TObject);
    procedure Button3Click(Sender: TObject);
    procedure Button4Click(Sender: TObject);
    procedure Button5Click(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    MyDT: TMyDateTime;
    { public declarations }
  end;

var
  Form1: TForm1;

implementation


{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);
begin
  MyDT:= TMyDateTime.Create(Now);
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  ShowMessage(MyDT.GetDateTimeAsString);
end;

procedure TForm1.Button2Click(Sender: TObject);
begin
  ShowMessage(MyDT.GetDateAsString);
end;


procedure TForm1.Button3Click(Sender: TObject);
begin
  ShowMessage(MyDT.GetTimeAsString);
end;

procedure TForm1.Button4Click(Sender: TObject);
begin
  MyDT.AddHours(1);
end;

procedure TForm1.Button5Click(Sender: TObject);
begin
  MyDT.AddDays(1);
```

```
end;


initialization
  {$I main.lrs}

end.
```

In this example, notice these important issues:

In the *DateTimeUnit* unit:

1. Defining *TmyDateTime* as *class*, which is the keyword for defining new classes.
2. Introducing the *Constructor* method: *Create*. This is a special procedure that is used to create objects in memory and initialize them.
3. Introducing the *Destructor* method: *Destroy*. This is a special procedure that is called to dispose of object's memory after finishing using it.
4. There are two sections in that class: *private*: which contains properties and methods that can not be accessed from outside the class unit. The other section is *public,* which contains properties and methods that can be accessed from outside the unit. If a class does not contains a public section, that means it can not be used at all.

Main unit:

1. We have added *DateTimeUnit* in the *Uses* clause of the main form to access its class.
2. We have declared the *MyDT* object inside the unit:

```
  MyDT: TMyDateTime;
```

3. We have created the object and initialized it in the main Form's *OnCreate* event:

```
  MyDT:= TMyDateTime.Create(Now);
```

That is the method of creating objects in the Object Pascal language.

News application in Object Oriented Pascal

In this example, we want to rewrite the News application using Object Oriented methods. We also have to categorize news into separate files.

We have created a new GUI application and named it *oonews*.

The next example is a new unit that contains the *TNews* class that has news functionality:

```
unit news;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type
  TNewsRec = record
    ATime: TDateTime;
    Title: string[100];
  end;

  { TNews }

  TNews = class
    private
      F: file of TNewsRec;
      fFileName: string;
    public
      constructor Create(FileName: string);
      destructor Destroy; override;
      procedure Add(ATitle: string);
      procedure ReadAll(var NewsList: TStringList);
      function Find(Keyword: string;
        var ResultList: TStringList): Boolean;
  end;

implementation

{ TNews }

constructor TNews.Create(FileName: string);
begin
  fFileName:= FileName;
end;

destructor TNews.Destroy;
begin
  inherited Destroy;
end;

procedure TNews.Add(ATitle: string);
var
```

```
  Rec: TNewsRec;
begin
  AssignFile(F, fFileName);
  if FileExists(fFileName) then
  begin
    FileMode:= 2; // Read/write access
    Reset(F);
    Seek(F, FileSize(F));
  end

  else
    Rewrite(F);

  Rec.ATime:= Now;
  Rec.Title:= ATitle;
  Write(F, Rec);
  CloseFile(F);

end;

procedure TNews.ReadAll(var NewsList: TStringList);
var
  Rec: TNewsRec;
begin
  NewsList.Clear;
  AssignFile(F, fFileName);
  if FileExists(fFileName) then
  begin
    Reset(F);
    while not Eof(F) do
    begin
      Read(F, Rec);
      NewsList.Add(DateTimeToStr(Rec.ATime) + ' : ' + Rec.Title);
    end;
    CloseFile(F);
  end;

end;

function TNews.Find(Keyword: string; var ResultList: TStringList): Boolean;
var
  Rec: TNewsRec;
begin
  ResultList.Clear;
  Result:= False;
  AssignFile(F, fFileName);
  if FileExists(fFileName) then
  begin
    Reset(F);
    while not Eof(F) do
    begin
      Read(F, Rec);
      if Pos(LowerCase(Keyword), LowerCase(Rec.Title)) > 0 then
      begin
        ResultList.Add(DateTimeToStr(Rec.ATime) + ' : ' + Rec.Title);
        Result:= True;
      end;
    end;
```

```
    CloseFile(F);
  end;

end;

end.
```

In the main form we have added *Edit box*, *ComboBox*, three *buttons*, *Memo*, and two *labels* components as shown:

In the main unit, we have written this code:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,
    Dialogs, News, StdCtrls;

type
```

```
  { TForm1 }

  TForm1 = class(TForm)
    btAdd: TButton;
    btShowAll: TButton;
    btSearch: TButton;
    cbType: TComboBox;
    edTitle: TEdit;
    Label1: TLabel;
    Label2: TLabel;
    Memo1: TMemo;
    procedure btAddClick(Sender: TObject);
    procedure btSearchClick(Sender: TObject);
    procedure btShowAllClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    NewsObj: array of TNews;
    { public declarations }
  end;

var
  Form1: TForm1;

implementation

{ TForm1 }

procedure TForm1.FormCreate(Sender: TObject);
var
  i: Integer;
begin
  SetLength(NewsObj, cbType.Items.Count);
  for i:= 0 to High(NewsObj) do
    NewsObj[i]:= TNews.Create(cbType.Items[i] + '.news');

end;

procedure TForm1.btAddClick(Sender: TObject);
begin
  NewsObj[cbType.ItemIndex].Add(edTitle.Text);
end;

procedure TForm1.btSearchClick(Sender: TObject);
var
  SearchStr: string;
  ResultList: TStringList;
begin
  ResultList:= TStringList.Create;
  if InputQuery('Search News', 'Please input keyword', SearchStr) then
    if NewsObj[cbType.ItemIndex].Find(SearchStr, ResultList) then
    begin
      Memo1.Lines.Clear;
      Memo1.Lines.Add(cbType.Text + ' News');
      Memo1.Lines.Add('-------------------------------------------------');
      Memo1.Lines.Add(ResultList.Text);
    end
```

```
    else
      Memo1.Lines.Text:= SearchStr + ' not found in ' +
        cbType.Text + ' news';
  ResultList.Free;
end;

procedure TForm1.btShowAllClick(Sender: TObject);
var
  List: TStringList;
begin
  List:= TStringList.Create;
  NewsObj[cbType.ItemIndex].ReadAll(List);
  Memo1.Lines.Clear;
  Memo1.Lines.Add(cbType.Text + ' News');
  Memo1.Lines.Add('------------------------------------------------------------');
  Memo1.Lines.Add(List.Text);
  List.Free;
end;

procedure TForm1.FormClose(Sender: TObject; var CloseAction: TCloseAction);
var
  i: Integer;
begin
  for i:= 0 to High(NewsObj) do
    NewsObj[i].Free;

  NewsObj:= nil;

end;

initialization
  {$I main.lrs}

end.
```

In the previous example, notice these issues:

1. We have used a Dynamic array, which is an array that can be allocated, expanded, reduced and disposed at run time according to it's usage. We declare a dynamic array of News objects like this:

```
  NewsObj: array of TNews;
```

At run time and before using it, we should initialize it using the *SetLength* procedure:

```
  SetLength(NewsObj, 10);
```

Which means allocate 10 elements for that array. This is similar to the declaration of a normal array:

```
  NewsObj: array [0 .. 9] of TNews;
```

A normal array's size will remain constant during the time an application is running, but a dynamic array's size can be increased and decreased.

In this example, we have initialized the array according to categories that exist in the combo box:

```
SetLength(NewsObj, cbType.Items.Count);
```

If we add more categories in *ComboBox.Items*, the dynamic array's size will increase accordingly.

2. The *TNews* type is a *Class*, and we can not use it directly,.We must declare an object instance of it like *NewsObj*.
3. At the end of the application, we have released the objects, then released the dynamic array:

```
for i:= 0 to High(NewsObj) do
  NewsObj[i].Free;

NewsObj:= nil;
```

Queue Application

A queue is an example of one of type of data structure. It is used to insert and store elements sequentially and delete them in the order in which the elements were inserted. Its rule is called *First-in-first-out*.

In the next example, we have written a unit called *Queue*, which contains the *TQueue* class. The *TQueue* class can be used to store data like names, and get them sequentially. Getting data from a queue deletes that data. For example, if the queue contains 10 items, and we get and read 3 items, 7 items will be left in the queue.

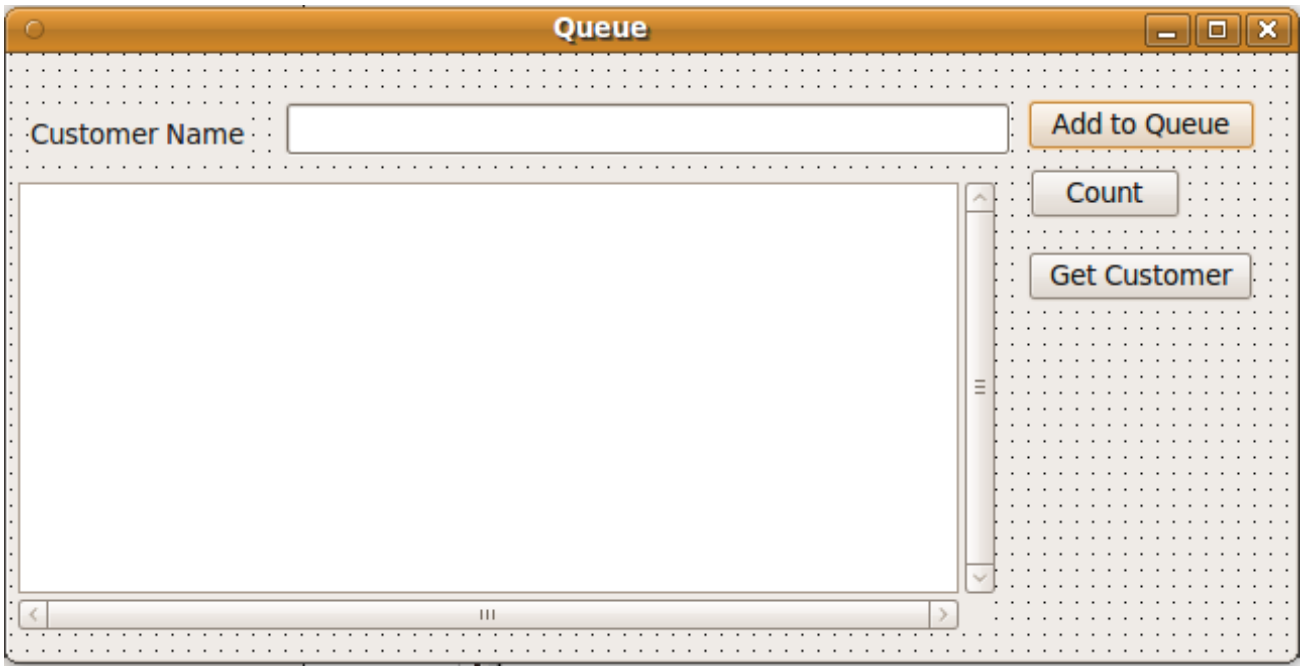
Queue unit:

```
unit queue;
// This unit contains TQueue class,
// which is suitable for any string queue

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils;

type

  { TQueue }

  TQueue = class
    private
      fArray: array of string;
      fTop: Integer;
    public
      constructor Create;
      destructor Destroy; override;
      function Put(AValue: string): Integer;
      function Get(var AValue: string): Boolean;
      function Count: Integer;
      function ReOrganize: Boolean;
  end;

implementation

{ TQueue }

constructor TQueue.create;
begin
  fTop:= 0;
end;

destructor TQueue.destroy;
begin
  SetLength(fArray, 0); // Erase queue array from memory
  inherited destroy;
```

```
end;


function TQueue.Put(AValue: string): Integer;
begin
  if fTop >= 100 then
    ReOrganize;

  SetLength(fArray, Length(fArray) + 1);
  fArray[High(fArray)]:= AValue;
  Result:= High(fArray) - fTop;
end;

function TQueue.Get(var AValue: string): Boolean;
begin
  AValue:= '';
  if fTop <= High(fArray) then
  begin
    AValue:= fArray[fTop];
    Inc(fTop);
    Result:= True;
  end
  else // empty
  begin
    Result:= False;

    // Erase array
    SetLength(fArray, 0);
    fTop:= 0;
  end;
end;

function TQueue.Count: Integer;
begin
  Result:= Length(fArray) - fTop;
end;

function TQueue.ReOrganize: Boolean;
var
  i: Integer;
  PCount: Integer;
begin
  if fTop > 0 then
  begin
    PCount:= Count;
    for i:= fTop to High(fArray) do
      fArray[i - fTop]:= fArray[i];

    // Truncate unused data
    setLength(fArray, PCount);
    fTop:= 0;
    Result:= True; // Re Organize is done

  end
  else
    Result:= False; // nothing done
  end;
end;
```

```
end.
```

The main form for the Queue application:



The code of the main unit:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,
 Dialogs, Queue, StdCtrls;

type

  { TfmMain }

  TfmMain = class(TForm)
    bbAdd: TButton;
    bbCount: TButton;
    bbGet: TButton;
    edCustomer: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    procedure bbAddClick(Sender: TObject);
    procedure bbCountClick(Sender: TObject);
    procedure bbGetClick(Sender: TObject);
```

```
    procedure FormClose(Sender: TObject; var CloseAction: TCloseAction);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    MyQueue: TQueue;
    { public declarations }
  end;

var
  fmMain: TfmMain;

implementation


{ TfmMain }

procedure TfmMain.FormCreate(Sender: TObject);
begin
  MyQueue:= TQueue.Create;
end;

procedure TfmMain.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
  MyQueue.Free;
end;

procedure TfmMain.bbCountClick(Sender: TObject);
begin
  Memo1.Lines.Add('Queue length is: ' + IntToStr(MyQueue.Count));
end;

procedure TfmMain.bbAddClick(Sender: TObject);
var
  APosition: Integer;
begin
  APosition:= MyQueue.Put(edCustomer.Text);
  Memo1.Lines.Add(edCustomer.Text + ' has been added as # ' +
    IntToStr(APosition + 1));
end;

procedure TfmMain.bbGetClick(Sender: TObject);
var
  ACustomer: string;
begin
  if MyQueue.Get(ACustomer) then
  begin
    Memo1.Lines.Add('Got: ' + ACustomer + ' from the queue');
  end
  else
    Memo1.Lines.Add('Queue is empty');
end;

initialization
  {$I main.lrs}

end.
```

In the *TQueue* class, we have used the *Put* method to insert a new item by expanding the dynamic array and then put the new item in the new last element of the array.

When calling the *Get* method to remove the item at the top of the queue, the *fTop* pointer will be moved to the next item in the queue.

Removing items from the top of a the dynamic array will result in moving the *fTop* pointer, but the items of dynamic array will remain in memory and will occupy space, because we can delete only from the bottom of the dynamic array, and for that reason if the number of deleted items reaches 100, the *ReOrganize* method will be called to shift the queue elements at the top of the dynamic array, then delete unused elements of the array.

This is the code that shifts queue elements to the top of the dynamic array:

```
    for i:= fTop to High(fArray) do
      fArray[i - fTop]:= fArray[i];
```

And this is the code for cutting the dynamic array from the bottom:

```
    // Truncate unused data
    setLength(fArray, PCount);
    fTop:= 0;
```

In this example, we find that object oriented programming introduces information hiding. Access to sensitive data will be denied, like access to variables. Instead we can use specific methods that will not result in odd behavior for the object.

Sensitive data and methods will be located in the private section of the class declaration:

```
    private
      fArray: array of string;
      fTop: Integer;
```

Programmers who use this class can not access these variables directly. If they were able to access those variables, then they could corrupt the queue by modifying *fTop* or fArray by accident. For example, suppose that *fTop* has been changed to 1000 while the queue contains only 10 elements, this will result in an access violation error at run-time.

As an alternative for using variables directly, we have implemented *Put* and *Get* methods to add and remove items from array safely. This method is like using gates to control the inside elements.
This feature of OOP is called *encapsulation*.

Object Oriented File

In the first chapter, we used different types of files, and we manipulated them using structured programming (procedures and functions). This time we will access files using a file object.

One of Object Oriented Pascal's file classes is *TFileStream*, which contains methods and properties to manipulate files.

Using this method makes things more standard and predictable for programmers. For example to open and initialize a file, we will use the *Create* constructor, as any other object in Pascal, but in the structured method of using files, initializing files is done by *AssignFile*, *Rewrite*, *Reset* and *Append* procedures.

Copy files using TFileStream

In this example, we will copy files using the *TFileStream* type.

To do this, create a new application and put these components on the main form:

TButton, TOpenDialog, TSaveDialog

For the *OnClick* event of the button, write this code:

```
procedure TfmMain.Button1Click(Sender: TObject);
var
  SourceF, DestF: TFileStream;
  Buf: array [0 .. 1023] of Byte;
  NumRead: Integer;
begin
  if OpenDialog1.Execute and SaveDialog1.Execute then
  begin
    SourceF:= TFileStream.Create(OpenDialog1.FileName, fmOpenRead);
    DestF:= TFileStream.Create(SaveDialog1.FileName, fmCreate);
    while SourceF.Position < SourceF.Size do
    begin
      NumRead:= SourceF.Read(Buf, SizeOf(Buf));
      DestF.Write(Buf, NumRead);
    end;
    SourceF.Free;
    DestF.Free;
    ShowMessage('Copy finished');
  end;
end;
```

This method is very similar to using untyped files, except that it uses an object oriented file.

We can also use a simpler method of copying files:

```
procedure TfmMain.Button1Click(Sender: TObject);
var
  SourceF, DestF: TFileStream;
begin
  if OpenDialog1.Execute and SaveDialog1.Execute then
  begin
    SourceF:= TFileStream.Create(OpenDialog1.FileName, fmOpenRead);
    DestF:= TFileStream.Create(SaveDialog1.FileName, fmCreate);
    DestF.CopyFrom(SourceF, SourceF.Size);
    SourceF.Free;
    DestF.Free;
    ShowMessage('Copy finished');
  end;
end;
```

The *CopyFrom* method copies the entire file contents to the destination file because we gave it the source file's size : SourceF.Size

Inheritance

Inheritance in Object Oriented Programming means creating a new class from an existing one and inheriting its methods and properties. After inheriting existing properties and methods, we can add new methods and properties to the new class.

As an example of inheritance, we want to create new a Integer queue class from our old string queue class. Instead of writing the Integer queue from scratch, we can inherit from string queue.

To inherit from string queue, add a new unit and put the string queue unit in its *uses* clause. Then declare the new integer queue as:

```
  TIntQueue = class(TQueue)
```

The new unit's name is *IntQueue*, and we have introduced two new methods: *PutInt*, and *GetInt*.

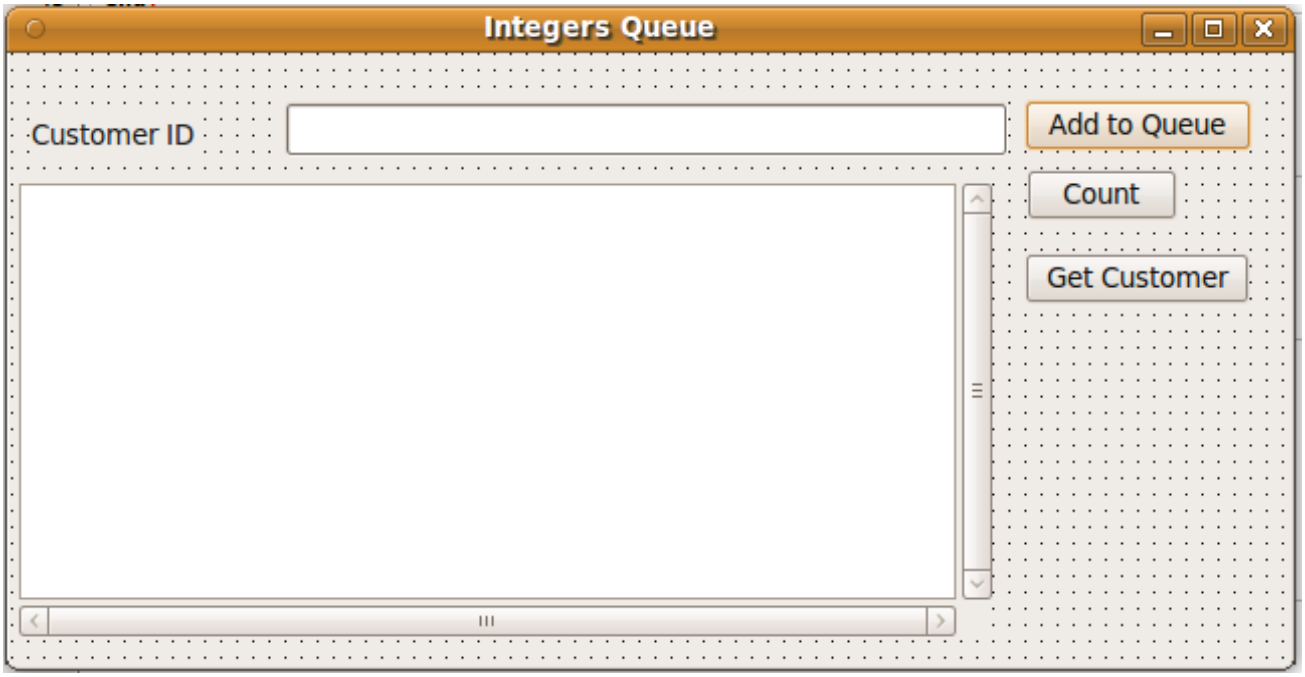
The complete code of the unit that contains the *TIntQueue* class is:

```
unit IntQueue;

// This unit contains TIntQueue class, which is inherits TQueue
// class and adds PutInt, GetInt methods to be used with
// Integer queue

{$mode objfpc}{$H+}

interface
```

```
uses
  Classes, SysUtils, Queue;

type

  { TIntQueue }

  TIntQueue = class(TQueue)

  public
    function PutInt(AValue: Integer): Integer;
    function GetInt(var AValue: Integer): Boolean;

  end;

implementation

{ TIntQueue }

function TIntQueue.PutInt(AValue: Integer): Integer;
begin
  Result:= Put(IntToStr(AValue));
end;

function TIntQueue.GetInt(var AValue: Integer): Boolean;
var
  StrValue: string;
begin
  Result:= Get(StrValue);
  if Result then
    AValue:= StrToInt(StrValue);

end;

end.
```

Note that we didn't write new *Create*, *Destroy* and *Count* methods, because they already exist in the parent class *TQueue*.

To use the new integer queue class, we have created a new application and added these components:



The unit's code is:

```
unit main;

{$mode objfpc}{$H+}

interface

uses
  Classes, SysUtils, FileUtil, LResources, Forms, Controls, Graphics,
    Dialogs, IntQueue, StdCtrls;

type

  { TfmMain }

  TfmMain = class(TForm)
    bbAdd: TButton;
    bbCount: TButton;
    bbGet: TButton;
    edCustomerID: TEdit;
    Label1: TLabel;
    Memo1: TMemo;
    procedure bbAddClick(Sender: TObject);
    procedure bbCountClick(Sender: TObject);
    procedure bbGetClick(Sender: TObject);
    procedure FormClose(Sender: TObject;
       var CloseAction: TCloseAction);
    procedure FormCreate(Sender: TObject);
  private
    { private declarations }
  public
    MyQueue: TIntQueue;
    { public declarations }
  end;
```

```
var
  fmMain: TfmMain;

implementation


{ TfmMain }

procedure TfmMain.FormCreate(Sender: TObject);
begin
  MyQueue:= TIntQueue.Create;
end;

procedure TfmMain.FormClose(Sender: TObject; var CloseAction: TCloseAction);
begin
  MyQueue.Free;
end;

procedure TfmMain.bbCountClick(Sender: TObject);
begin
  Memo1.Lines.Add('Queue length is: ' + IntToStr(MyQueue.Count));
end;

procedure TfmMain.bbAddClick(Sender: TObject);
var
  APosition: Integer;
begin
  APosition:= MyQueue.PutInt(StrToInt(edCustomerID.Text));
  Memo1.Lines.Add(edCustomerID.Text + ' has been added as # '
   + IntToStr(APosition + 1));
end;

procedure TfmMain.bbGetClick(Sender: TObject);
var
  ACustomerID: Integer;
begin
  if MyQueue.GetInt(ACustomerID) then
  begin
    Memo1.Lines.Add('Got: Customer ID : ' + IntToStr(ACustomerID) +
    ' from the queue');
  end
  else
    Memo1.Lines.Add('Queue is empty');
end;

initialization
  {$I main.lrs}

end.
```

Note that we have used the properties and methods of *TQueue* in addition to *TIntQueue* properties and methods.

In this case ,we call the original class *TQueue* the base class or *ancestor*, and we call the new class the *descendant*.

Instead of creating a new class, we could modify the string queue unit and add *IntPut* and *IntGet* to handle integers, but we have created the new class *TIntQueue* to illustrate inheritance. There is also another reason: suppose that we didn't have the original source code of *TQueue*, like having only the compiled unit file (*ppu* in Lazarus) or (*.dcu* in Delphi). In this case we couldn't see the source code and of course we couldn't modify it. Inheritance will be the only way to add more functionality to this queue.

The end

Code.sd

18.June.2011