

DELPHI PROGRAMMING FOR BEGINNERS

BY YURIY KALMYKOV

embarcadero®
An Idera, Inc. Company



About the Author

Yuriy Kalmykov is a well-known expert in software development and author of many programming publications and textbooks, including "Teaching Delphi Programming in Schools". This book is a result of twenty-five years of instructing students as a member of the Informatics and Control Processes faculty at the National Research Nuclear University ME-PhI (Moscow Engineering Physics Institute) and teaching in top preparatory schools in Moscow.

Find additional free educational resources for teaching and learning to program at LearnDelphi.org

This book is licensed under a Creative Commons Attribution-ShareAlike 4.0 International license.

Full license <https://creativecommons.org/licenses/by-sa/4.0/>



Contents

MODULE 1. Introducing Delphi. Simple Components	3
MODULE 2. Handling a Button Press Event	16
MODULE 3. Variables and Types of Variables. Type Conversion	21
MODULE 4. Standard Math Functions	27
MODULE 5. Logical Expressions. Variables of Boolean Type. Logical Operations	30
MODULE 6. Conditional Execution in Program. IF...THEN...ELSE statement	34
MODULE 7. Nested If...Then...Else statement. Practicing Task Solving	39
MODULE 8. Procedures	42
MODULE 9. Functions	48
MODULE 10. Graphics	51
MODULE 11. Loops	54
MODULE 12. Strings	58
MODULE 13. Strings and Conversion To or From Numeric Types	63
MODULE 14. TMemo Control	67
MODULE 15. TMemo Control (Continued)	72
MODULE 16. Random Numbers, Constants, User Types and Arrays	76
MODULE 17. Single Dimensional Static Array	79
MODULE 18. Array Sorting and Selection Sort	84
MODULE 19. StringGrid Control	88
MODULE 20. StringGrid Practice	93
MODULE 21. Two-Dimensional Arrays	100
MODULE 22. Date and Time	107
MODULE 23. Timer	112
MODULE 24. Text Files	117
MODULE 25. Standard File Dialogs	132

Introducing Delphi. Simple Components

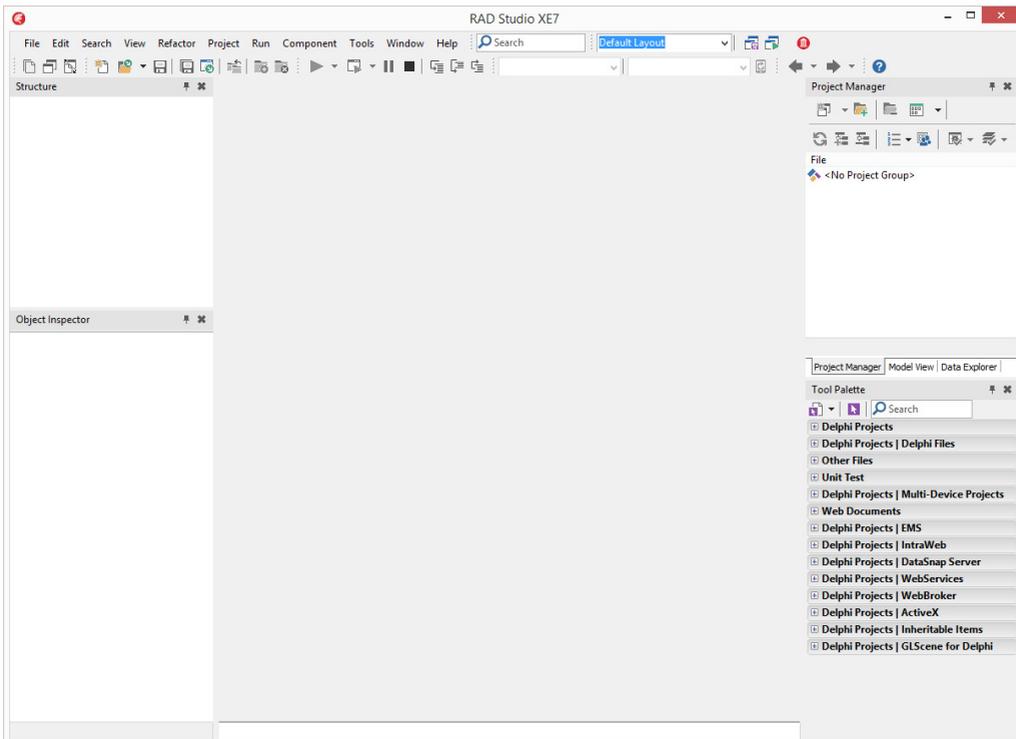
The development of personal desktop computers led to the introduction of multi-tasking, multi-user operating systems such as Microsoft Windows. However, as a result, the process of creating software has become much more complicated. Visual integrated development environments (IDE) and rapid application development (RAD) systems were created by market leaders to facilitate the interaction with operating systems, reduce coding time, and improve the quality of the code.

Visual Programming is a process of building software applications in which a user can simultaneously design, edit, debug and test the app using Visual IDE. In other words, Visual Programming is a combination of two interconnected processes: designing an application window visually; and writing the code.

Delphi, a powerful Pascal compiler with a number of significant improvements for creating Windows applications, was first released in 1996. It is a high-level software development system with an integrated toolset for developing complete applications; it is a tool for rapid application development. Visual design and event-based programming concepts are the pillars of Delphi ideology. Using those concepts significantly improves application design processes and greatly reduces development time.

Visual design allows users to lay out the application and see the results of the process prior to even starting the actual program. There are other benefits too: to assign a property value to a control or element of the app, it is not necessary to write multiple lines of code. All it takes is to change that value in a Properties window. This change will automatically generate or update the code.

Run Delphi



The integrated development environment Delphi is a multi-window system configured by setting elements of the user interface.

For any programmer's action in the window, Delphi automatically makes changes in the underlying program code.

The main Delphi window has MainMenu and ToolBar.

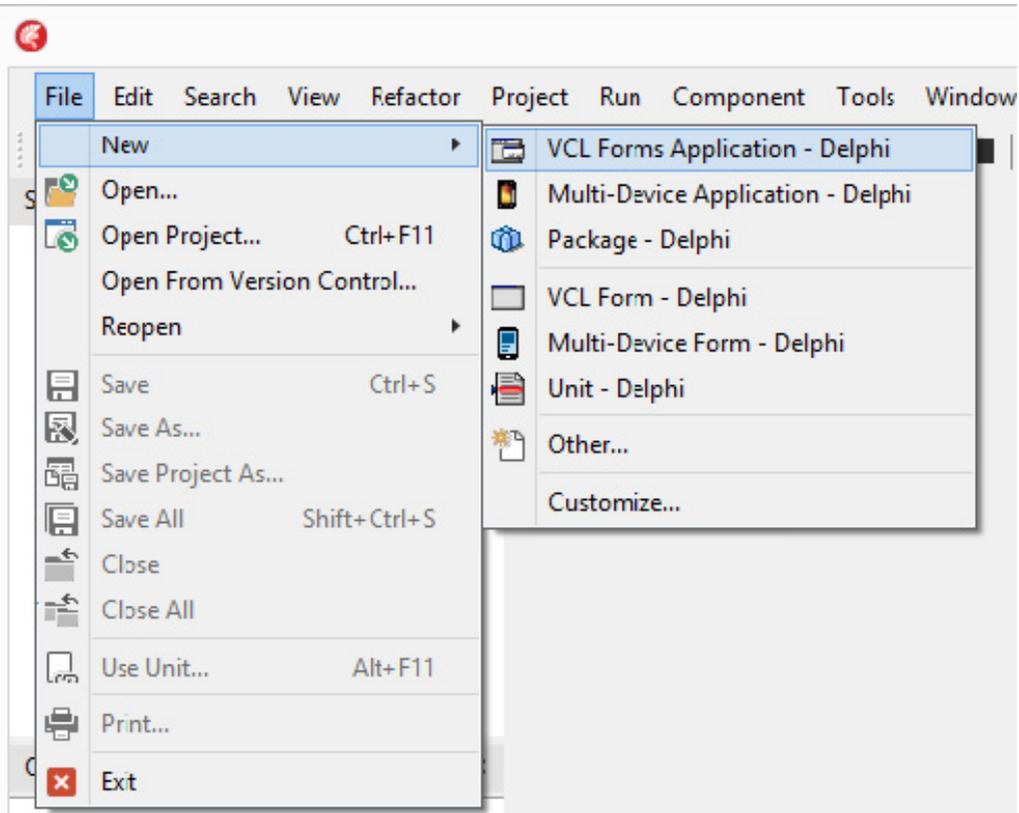


MainMenu contains all you need to manage the project.

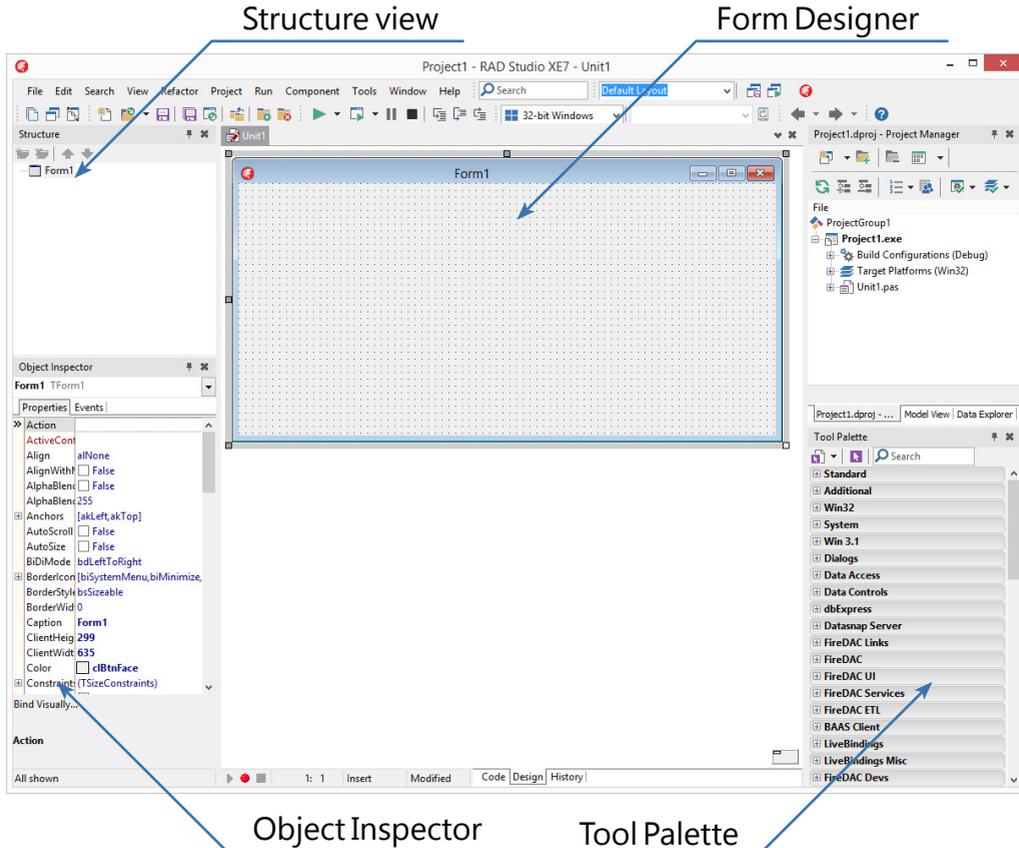
ToolBar contains buttons for fast access to the most popular options of the MainMenu.

For example, to run the program we can choose the **Run** option in the MainMenu **Run** dropdown, press the F9 key, or click a green triangle on the ToolBar.

To start writing a new program, it is necessary to select the **New** item in the **File** menu and then, in the opened list, to choose **VCL Forms Application — Delphi**.



The following Delphi windows layout will appear:



Let us parse out the windows.

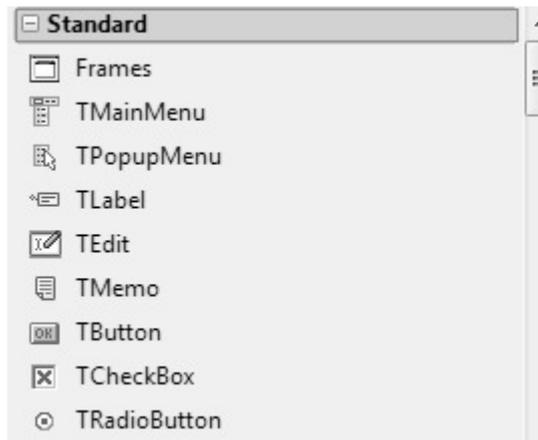
Delphi is an Object Oriented Programming language. An object is a self-contained entity having properties (characteristics or distinctive signs) and a set of actions or behaviors. The created object can be moved from one program to another. Delphi includes hundreds of ready to use objects (components), which are presented in the Tool Palette. These components are grouped into the tabs.



The main tabs of the Tool Palette are:

1. Standard
2. Additional
3. System
4. Data Access
5. Data Controls
6. Dialogs

For now, we will work with the Standard tab in the Tool Palette. It contains standard interface elements most programs can't do without.



Form window is a project for the future application. Visual programming allows you to create programs by manipulating components and placing them in a form.

When you first run Delphi, it automatically suggests a new project with the blank Form window named *Form1*. This form contains all basic window elements: the title *Form1*, minimize, close and maximize buttons, and a window system menu button. This is the main window of our program. We will add graphical items from Tool Palette here.

Text boxes, command buttons, and other Form controls are called components (Form components) in Delphi. In the program, the form and components are considered objects. So, the window with components' properties is called the **Object Inspector**.

The **Object Inspector** window has two tabs. On the first tab, you can see all available properties of the selected component. The left column has a list and the right column contains current default values.

The second tab, *Events*, has possible event handlers for the selected component. The left columns contains names, the right one has relevant properties or procedures.

Structure View displays all objects (components) placed on the form.

Stop running the application

- Click close button
- Menu **Run** — command *Program Reset* (**Ctrl+F2**)

Saving the Project

Any Delphi program comprises a large number of various files such as the project file, one or several units, etc. The project file is generated by the Delphi environment automatically, and it is not available for manual editing. That is why it has a unique extension (*.dpr), and it is not displayed in the Code Editor.

The project should be saved using the Save All command. Every project should be saved to a separate folder. The project folder and files should

be named appropriately. You should not accept the default names. File names may contain only Latin characters. Numerals can be used in file names beginning from the second character. Other symbols are invalid for file naming.

You have to assign unique names to the files **Project1.dpr** and **Unit1.pas** (they are different). Other files will be assigned default names.

When renaming the files **Project1.dpr** and **Unit1.pas**, follow the Delphi code standard, according to which all files must have an XXX prefix — for example: **prjMyTask1.dpr** and **untMyTask1.pas** (XXX are letters corresponding to the abbreviated name of whatever you are saving: **Project1 — prj** and **Unit1 — unt**). This rule also applies when renaming simple components.

This brings us to some of the simple components available in the Delphi environment.

These components are considered simple not because it is easier to use them compared to other components. They are so called because, during the creation of the graphical user interface, they are often used in quite complex combinations with other components. Let's have a look at the following simple components: form, label, text box, and button.

Form

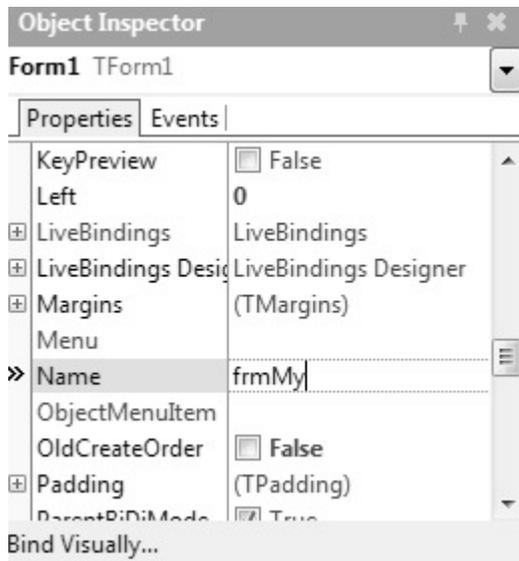
Of all these components, form is the only object that is not in the Tool Palette. A form is generated automatically when you create a new application. If you need to use several forms in an application, you can add more forms to your project. We will learn how to do that later. When working in Delphi, imagine that you are an artist and the form is your canvas. You "paint" the graphical user interface of the application by placing components on the form. Then, by changing the properties of components, you change their appearance.

After that, you will work on the component's interaction with the user. User interaction means that the form components react to user actions.

Properties of components are their characteristics. The name, caption, size or color of the component, and size or color of component's label are examples of properties.

Let's review some properties of the Form object. Many properties of different components are the same but the Form has some unique properties.

The first property we will examine is the **Name** property. All components must have this property, because the name is needed to call the component in the program. For the name, use a trigram prefix describing the component's type. For the Form, for example, we will use the prefix *frm* (shorter name of Form). If we want to call the Form *My*, we should assign the value *frmMy* to the Name property.



Let examine three simple components from the Standard tab of the Tool Palette.

TLabel

The component **TLabel** is used to output text that can be changed by user (of course, it can also be changed by a program). Let's see how to work with TLabels on a concrete example. Do the next steps:

1. Create a new project.
2. Put TLabel on the form. To do so, you need to double-click on the TLabel icon on the Tool Palette. Another way to place TLabel on the

form is to click once on the TLabel icon and then click on any place on the form. This is more convenient because the TLabel is placed where you want. To delete TLabel from the Form you need to select it (click on it with the mouse button — small black squares show that it is selected) and press the <Delete> key. To deselect TLabel, you need to click the mouse button somewhere out of TLabel. Experiment with placement and removal of TLabels.

3. Move the TLabel by dragging and dropping. To do so, move the pointer to the TLabel, and press and hold down the button on the mouse to “grab” the TLabel. “Drag” the TLabel to the desired location by moving the pointer to the new location. «Drop» the object by releasing the button.

4. Change the property Name of TLabel to `IblMyLabel` (by default it was called `Label1`). Click on the Name property in the Object Inspector and type ***IblMyLabel***. Make sure you change the TLabel property, not the Form property (it is a common mistake of beginners). For this purpose, TLabel has to be selected; the title of the list at the top of the Object Inspector will say ***Label1 TLabel*** (as soon as you change the name, it will change to ***IblMyLabel TLabel***). After typing the new name, save it by pressing the <***Enter***> key.

5. Check that the caption of TLabel has changed on `IblMyLabel`. It happens because by default the caption of the TLabel is its name. Change the caption. To do so, choose the Caption property in the Object Inspector, type the new caption ‘It’s my first label!’ and press <***Enter***> key. The new text will appear on a form.

6. Change the color of the TLabel background. Choose Color property, click on the arrow, choose yellow from the drop-down list, and click on it.

7. Change the font and text color of the TLabel. Choose the Font property and click on the three dots (ellipsis). In the Font window, change the font to Arial, the style to Bold Italic, and the size to 20. Choose red from the drop-down list and click on the OK button.

8. Add one more TLabel on the form. Try another way this time — click on the TLabel icon in the Tool Palette, move the pointer to any place on the form, and click again. A new TLabel should appear.

9. Change the Name property of the new TLabel to **lblAnother**, and the **Caption** property to 'Another label'.

10. Now select the form. You can do it in two ways: click on any place outside of the labels or choose **Form1** in the Structure View. If the form is visible, the first way is more convenient, of course, but if there are lots of forms in the project and the form you need is covered by others, then the second way is better.

11. Change the form properties: set the Name property value to **frm-LabelExample** and the Caption property value to 'Example of Label'.

12. So you have just created a simple application, which, to say the truth, does nothing yet. Run it. You can do so using one of three ways: click on the Run icon (the green triangle), choose the command Run in the menu Run, or press the **<F9>** key.

13. Click on the X button at the top right corner of the form to quit the application.

TEdit

The component TEdit stores text that can be put into the component both during the creation of the application and during the execution. The text seen in the text box comes from the property **Text**. Property **MaxLength** defines the maximum number of symbols in the text box. The **MaxLength** value 0 means there is no limit on number of characters. The text font is set using the **Font** property. If you set **ReadOnly property value to True**, the user will not be able to change the text of TEdit. The walkthrough below will help you get a better handle on TEdit functionality.

1. Create a new project.
2. Place **TEdit** on the form. Just as with **TLabel**, you can do it in two ways: by double-clicking TEdit icon in the Tool Palette or by clicking the TEdit icon and then clicking any place in the form.
3. Change the size of TEdit. Put the mouse pointer on one of the small black squares, press and hold down the left mouse button, and drag the small black square (and TEdit border with it) in the necessary direction. When you are at the necessary size, release the button. If there

are no small black square around the TEdit, then the TEdit is not selected. In this case, select the TEdit by clicking on it first.

4. Move TEdit to another place using drag and drop. Put the mouse pointer on TEdit, press and hold the left mouse button, and drag TEdit to the new location. When TEdit reaches the necessary location, release the mouse button.

5. Assign the value **edtMyText** to the property **Name**. Click the Name property in the Object Inspector and type **edtMyText**. As for TLabel, be sure you change the TEdit property, not the form. In the Structure View, you will see **Edit1: TEdit** (as soon as you change the name **edtMyText: TEdit** will appear).

6. Choose the **Text** property in Object Inspector and enter its new value: "This is a text edit control". Record the new name by pressing the <**Enter**> key. Note that as you enter the text in Object Inspector, the TEdit text on the form is changed too.

7. Change the text color of TEdit to blue. Click on the + sign near the Font property. A list of additional Font properties will appear. Select the Color property and click on the arrow: the list of available colors will appear. Find blue and click on it.

8. Select the form. You can do it in one of two ways: click any place on the form not covered by the text box, or choose a form name in the Structure View. Change the property Name to **frmEditBoxExample**, and the Caption property to 'Text Box Example'.

9. Press the <**F9**> key to run the new application. Experiment with the text box by typing any text into it.

10. To quit the application, click on the X button at the top right corner of the form.

11. Set the property **ReadOnly** to **True**.

12. Press the **F9** key to run the application again. Try to change the text: as you can see, now it is impossible. You may wonder why you would need a text box where you cannot input anything. However, later in the book, you will see that it is a quite useful tool because you can change a value of the ReadOnly property programmatically, allowing or not allowing the user to enter data.

13. Click on the X button to quit the application.

TButton

TButton is usually used to start the execution of some piece of code or the entire program. In other words, by clicking the TButton control, the user prompts the program to perform a certain action. At that time, the button changes to look as if it were pressed.

You can assign hot-key combinations to buttons. During execution, pressing those hot keys is equivalent to using the left mouse click on the button.

Perform a set of actions:

1. Create a new project.
2. In the Object Inspector, change the form property Name to **frmButtonExample** and **Caption** to 'Button Example'.
3. Place the button on the form. Double-click the TButton icon in the Tool Palette, or click on its icon and then click on any place of the form.

Change the TButton property to **btnMyButton**. Click on the **Name** property in the Object Inspector and type **btnMyButton**. Be sure you change the TButton property, not the form. In the Structure View, you will see **Button1: TButton** (as soon as you change the name, **btnMyButton: TButton** will appear).

4. Change the property Caption to **&Run**. Note that the letter following & is underlined (in this case it is R).
5. Change the size and location of the button.
6. Press **<F9>** to run the application.
7. Click on the button. The button appears pressed.
8. Activate the button by press the **<R>** key. As you can see, the button does not look pressed when you use the hot key to activate it. Meanwhile, since there is no program code connected to the button, we don't see any button reaction. Nevertheless, you can believe it is activated.

9. Click on the X button at the top right corner of the form to quit the application.

The *btnMyButton* caption appears as **Run**, not **Run**. Putting the symbol & before any character of the Caption property assigns a hot key to the button. In the button caption, the character following & is underlined. It tells users that a hot key is connected to the button. Users can activate the button from the keyboard by pressing Alt and the key with the underlined letter.

What do you do if a caption itself has to have &? After all, if you put it into caption, it underlines the next letter and it is not itself visible. To solve this problem, use the following rule: The symbol & is seen in the caption if the Caption property has two & together, i. e., &&. For example, to create the caption *This & That*, you write in **Caption** property *This && That*. Hot-key combinations are not assigned in this situation.

As you can see, in Delphi you can create a simple form (and application) in a minute. The most complicated graphical user interfaces can be created in the same way.

Now you are familiar with properties of the most popular Delphi components: forms, label, edit boxes and buttons. To improve on your new knowledge, experiment with these components. Try to change their other properties.

Handling a Button Press Event

Visual components are able to generate and handle several dozens kinds of events: mouse click, button press, keyboard press, windows opening, etc.

When user presses any mouse button, Windows sends a message “*This mouse button was clicked*” to the application. The program will respond to this event if a programmer programmed a reaction to it.

Response to an event is a result of a system event happening.

A program response to an event is programmed as a list of actions to be done; this list is entered in a **Code Editor** window. For a program to react to a mouse button click, it is necessary to write a program fragment called *event handler*.

Code Editor Window

This window has title Unit1 at the start.

You can open several source code files in the editor window. All opened files occupy their own tabs and their tab labels will display their source code file names. There will be three units if program has three windows. All three units will be displayed in the source Code Editor.

Simple events contain only events source, pointed to by the Sender parameter in the **Event Handler Procedure**. **Event Handler Procedures** for handling complex events need additional parameters such as the coordinates of the mouse pointer.

The selection of a control element produces an **OnClick** event, also called **Press Event**. Usually it originates from the mouse click on a component. The **OnClick** event is one of most frequently used events in the development of an application. For some components, the **OnClick** event

can originate from different actions upon the control element.

As an example, we will write the press event handler of the command button **btnMy**, placed on the form **frmMy**:

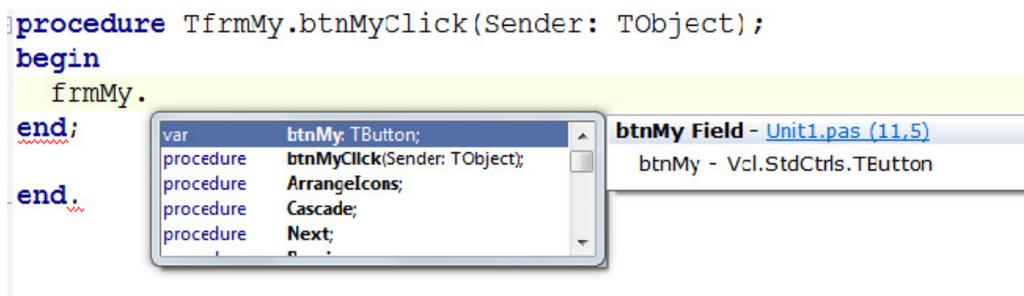
Double-click that button.

Delphi automatically generates code to minimize keyboard input, providing us with the complete mouse button press procedure body.

```
procedure TfrmMy.btnMyClick(Sender: TObject);  
begin  
end;
```

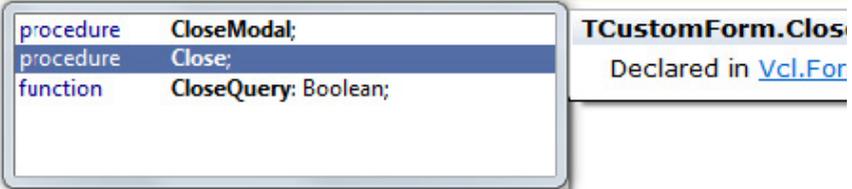
The cursor will be located between the begin and end keywords. Here we will write an instruction to programmatically close the form window — a complete analogue of the system button in the window title. As this window is main and the only one in application, the whole program will be terminated.

Type **frmMy** and add a dot ('.'). After a few moments Delphi displays the code completion options. Those options are methods and properties available to **frmMy** component.



We can find the appropriate property or method by navigating the list. As the list is relatively large, we can speed up the search by entering a few characters of the name and the number of elements in the list will decrease considerably.

```
procedure TfrmMy.btnMyClick(Sender: TObject);  
begin  
    frmMy.clo  
end;  
end.
```



We end code completion entry by pressing Enter.

The final code for the response to the **btnMy** command button press looks like the following:

```
procedure TfrmMy.btnMyClick(Sender: TObject);  
begin  
    frmMy.Close;  
end;
```

The program will exit if you press command button **btnMy** after the program has run. In this example, we used the method (procedure) **Close**. Calling a method shows what will be done with the component. You can see a method reference syntax in the example: the name of the component goes first, then there is a delimiter (dot) and finally the name of method specified after the delimiter.

```
<component_name>.<method>;
```

Every component has its own set of methods. Sometimes identical methods can be found in different components.

Let us demonstrate one more method, **Clear**. This method clears the content and can be applied, for example, to the component **Edit**.

```
Edit1.Clear
```

As we mentioned above, components have methods and properties.

Properties are attributes of components. Properties can be changed or specified as follows: first we reference the name of component, then we put the delimiter (dot), and after the delimiter, we specify the name of the property (Delphi allows us to choose one from code completion list). Then we put assignment operator and, finally, we specify the value of the property.

```
<component_name>.<property>:=<value>;
```

Most of the component properties correspond to ones displayed in the Object Inspector window. There are also some properties that are not reflected there; we will examine these later. For now, let us write a program in which pressing the command button will fill the form window with white background. In this program, we have to use a **Color** property.

```
procedure TfrmMy.btnMyClick(Sender: TObject);
begin
    frmMy.Color:=clWhite;
end;
```

Not only we can specify component properties, we can also read them. One use of this power is to assign a value of property of one component to some property of another component. The syntax looks like this:

```
<component1_name>.<property1>:=<component2_
name>.<property2>;
```

For example:

```
lblMy.Caption:=edtMy.Text;
```

This instructs the program to put the text from text box **edtMy** into the caption of label **lblMy**.

Multiple actions are separated by semicolon.

For example:

```
procedure TfrmMy.btnMyClick(Sender: TObject);
```

```
begin
    edit1.Clear;
    edit1.Color:=clBlue;
end;
```

Two actions will be performed after the button is pressed: text box will be cleared and its background will be filled with blue.

Exercises

Exercise 1.

Create a form that contains a label and two buttons. The first button enables the label, and the second button disables the label.

Exercise 2.

Create a form that contains a label, a text box, and a button. A button press moves the text from the text box to the label and clears the text-box entry.

Exercise 3.

“Traffic light”: Create a form that contains three labels and three buttons. Each button should enable an appropriate label and disables the other labels.

Exercise 4.

Create a form that contains a label and four groups of buttons; each group contains three buttons. The first group (three buttons) changes a background color of the label. The second group (three buttons) changes the label's font color. The third group (three buttons) changes the label's font size. And the final button group (three buttons) changes the name of font family of the label.

Variables and Types of Variables.

Type Conversion

To compute a value for a single expression with given inputs, and do it only once, it is not necessary to write a full-blown software program. However, if the process of calculations has to be done frequently and with different inputs every time, it makes sense to automate this process.

Assignment Operator

If you want to convert degrees (DEGR) to radians (RAD), the conversion formula will look like this in Delphi:

```
RAD:= (DEGR * 3.14) / 180;
```

The assignment operator is a very basic and important instruction in any programming language. It assigns a value of a computed expression on the right side of the assignment symbol:= to a variable on the left side of the assignment symbol. The assignment symbol consists of two separate symbols (: and =); however it is interpreted in Delphi as a single element.

The way it works internally is that the value of the expression on the right of the assignment operator is calculated first, and then that resulting value is assigned to the variable on the left of the assignment operator.

The format of assignment operator is as following:

```
<variable name>:= <expression>;
```

Now, let us talk about variables. A variable is a location in memory that has a name. The size of the memory allocated for a variable depends on

the value to be written into it. In theory, it could be possible to allocate only large memory “slots” for all variables. However, if that were the case, we would quickly run out of available memory.

In order to know exactly how much memory to allocate for a variable, we need to know its **data type**.

You can use variables in Delphi for variety of tasks and reasons. Variables need to be defined before they can be used. The **var** keyword is used to start a section of variable definitions, which is located after the procedure definition and before the keyword **begin**. The format for variable definition is:

```
Var  
    <variable name>: <data type>;
```

A variable name is a sequence of characters with the following syntax:

- Can contains letters, numbers (0–9), or underscore;
- Must begin with a letter.

The variable name can be of any length (the number of characters is not limited) and is not case sensitive.

When thinking about naming a variable, note that you should avoid one-character names except for temporary or loop counter variables.

Loop counter variables are the names I and J. Other instances of one-character variable names are S (string) or R (radius). One-character variable names should always be in uppercase, but it is always preferable to use more meaningful names. It is recommended that you not use the letter I as a variable name to avoid confusion with the number 1 (one).

Example:

```
procedure TfrmMy.btnMyClick(Sender: TObject);  
Var  
    k: Integer;  
begin  
end;
```

In this example, a variable **k** of type **Integer** is declared.

There are many data types in Delphi. Let's take a look at some of them.

Integers can be described with the following data types:

Data Type	Range	Representation
Integer	-2,147,483,648 to 2,147,483,647	32-bit, signed
Cardinal	0 to 4,294,967,295	32-bit, positive integers
ShortInt	-128 to 127	8-bit, signed
SmallInt	-32,768 to 32,767	16-bit, signed
LongInt	-2,147,483,648 to 2,147,483,647	32-bit, signed
Int64	-2^{63} to $2^{63}-1$	64-bit, signed
Byte	0 to 255	8-bit, unsigned
Word	0 to 65535	16-bit, unsigned
LongWord	0 to 4,294,967,295	32-bit, unsigned

We will mostly use **Integer** throughout this book.

Real numbers can also be represented by various data types. We will use data type **Real**.

Data Type	Range	Significant Digits (Precision)	Bytes
Real48	$\pm 2.9 \times 10^{-39}$ to $\pm 1.7 \times 10^{38}$	11-12	6
Real	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15-16	8
Single	$\pm 1.5 \times 10^{-45}$ to $\pm 3.4 \times 10^{38}$	7-8	4
Double	$\pm 5.0 \times 10^{-324}$ to $\pm 1.7 \times 10^{308}$	15-16	8
Extended	$\pm 3.6 \times 10^{-4951}$ to $\pm 1.1 \times 10^{4932}$	19-20	10
Comp	-2^{63} to $2^{63}-1$	19-20	8
Currency	-922337203685477.5808 to +922337203685477.5807	19-20	8

You know about normalized scientific notation (also known as exponential notation) for representation of real numbers. For example:

$$3,28 \cdot 10^{17} \quad 1,4 \cdot 10^{-9} \quad -5,101 \cdot 10^4$$

In Delphi, those numbers are recorded as:

3.28e+17	1.4e-09	-5.101e+4
328e15	0.14e-8	-5101e+1
0.328e+18	140e-11	-510100e-1

Arithmetic Expressions

Let's take a look at assignment operators one more time. The data type on the left should match the data type of the value of the arithmetic expression on the right.

An arithmetic expression is a legal mathematical expression built up using constants, variables, functions, arithmetic operators (such as +, *, —, / and exponent), and parentheses, (and).

Operators used in arithmetic expressions:

Operator	Name of the operator	Data Types of Operands	Result Data Type
+	Addition	At least one of Operands is Real	Real
		Integer	Integer
-	Subtraction	At least one of Operands is Real	Real
		Integer	Integer
*	Multiplication	At least one of Operands is Real	Real
		Integer	Integer
/	Division	Real, Integer	Real
Div	Integer Division	Integer	Integer
Mod	Remainder of Integer Division	Integer	Integer

Examples:

$$13 \text{ div } 4 = 3$$

$$13 \text{ mod } 4 = 1$$

$$-13 \text{ div } 4 = -3$$

$$-13 \text{ mod } 4 = -1$$

$$13 \text{ div } -4 = -3$$

$$13 \text{ mod } -4 = 1$$

$$-13 \text{ div } -4 = 3$$

$$-13 \text{ mod } -4 = -1$$

$$0 \text{ div } 2 = 0$$

$$1 \text{ mod } 2 = 1$$

$$2 \text{ div } 5 = 0$$

$$2 \text{ mod } 5 = 2$$

If the value of the expression is Integer, it can be assigned to a Real data type variable, but not vice versa.

Adding Two Numbers

Create form *frmCalc* with two TEdit textboxes *edtA* and *edtB*, a button *btnAdd* and label *lblResult*.

Set property **Caption** for the button, and leave properties **Text** for the textboxes blank. Double-click the button, the code window will appear and we can start coding a button click event handler.

First, we will declare the variables:

```
Var  
    a, b, c: Single;
```

We will get the a and b values from the textboxes *edtA* and *edtB* respectively. Note that we enter string values in the textboxes, so we have to convert them into numeric data type. We will use `StrToFloat` (string) function for that purpose.

```
Begin  
    a:=StrToFloat(edtA.Text); //enter the number A  
    b:=StrToFloat(edtB.Text); //enter the number B  
    c:=a+b; //Add numbers A and B  
    lblResult.Visible:=true; //Make label with Answer  
visible  
    lblResult.Caption:=FloatToStr(c); //Set the value  
of the answer to the label (converting it to string  
first)  
end;
```

The result of adding two variables is stored in a third variable (c), and then that result is converted back to string using `FloatToStr (number)` and displayed in the label `lblResult`. To make the code look even prettier, let's write it as follows:

```
lblAnswer.Caption:=edtA.Text+' '+edtB.  
Text+'='+FloatToStr(c);
```

When working with **StrToFloat** and **FloatToStr** in Delphi, it is important to watch for proper formatting of the decimal separator. In the program (Unit), only period (".") is allowed as a decimal separator. However, in the input and output windows on the form, the decimal separator depends on Windows regional settings. For example, in many European countries, comma is used to separate the integral and the fractional parts of a decimal numeral.

Functions **IntToStr(integer)** and **StrToInt(string)** are used to convert integers to strings and strings to integers respectively.

Exercises

Exercise 1.

Using two text boxes, one label, and four buttons, create a calculator for four basic arithmetic operations.

Exercise 2.

Write a program to convert Fahrenheit to Celsius and vice versa. ($T_f = 9/5 * T_c + 32$).

Exercise 3.

Write a program to convert speeds from kilometers per hour into meters per second and vice versa.

Standard Math Functions

In expressions, it is possible to use standard mathematical functions with standard operators and operations. You should pay attention to the argument and result types to get the correct results.

Standard functions in Delphi language

Function	Computation	Input parameters	Result type	Examples
ABS (X)	Returns the absolute value of a number.	X — REAL or INTEGER value	Same as parameter type	ABS (2.0) = 2.0000e+00;
SQR (X)	Returns the square of a number.	X — REAL or INTEGER value	Same as parameter type	SQR (3) = 9; SQR (-2.0) = 4.0000e+00;
SQRT (X)	Returns the square root of a number.	X — REAL or INTEGER value	REAL	SQRT (16) = 4.0000e+00; SQRT (25.0) = 5.0000e+00;
EXP (X)	Returns the exponentiation of a number.	X — REAL or INTEGER value	REAL	EXP (0) = 1.00000e+00; EXP (-1.0) = 3.67879e-01;
LN (X)	Returns the natural logarithm of a number.	X — REAL or INTEGER value	REAL	LN (1) = 0.00000e+00 LN (7.5) = 2.01490e+00
SIN (X)	Returns the sine of parameter.	X — REAL or INTEGER value, in radians	REAL	SIN (0) = 0.00000e+00; SIN (1.0) = 8.41471e-01
COS (X)	Returns the cosine of parameter.	X — REAL or INTEGER value, in radians	REAL	COS (0) = 1.00000e+00; COS (1.0) = 8.41471e-01
ARCTAN (X)	Returns the arctangent of parameter.	X — REAL or INTEGER value	REAL	ARCTAN (0) = 0.0000e+00 ARCTAN (-1.0) = -7.8539e-01

Function	Computation	Input parameters	Result type	Examples
ROUND (X)	Rounds a real value to the closest (by absolute value) integer value.	X — REAL	INTEGER	ROUND(3.1) = 3; ROUND(-3.1) = -3; ROUND(3.8) = 4; ROUND(-3.8) = -4; Note: numbers with fractional part equal to .5 are rounded to the nearest even number. ROUND(3.5) = 4; ROUND(2.5) = 2;
TRUNC (X)	Returns integer part of parameter by dropping its fractional part.	X — REAL	INTEGER	TRUNC(3.1) = 3; TRUNC(-3.1) = -3; TRUNC(3.8) = 3;
INT (X)	Returns integer part of parameter by dropping its fractional part.	X — REAL	REAL	INT(3.1) = 3.00000E+00 INT(-3.1) = -3.00000E+00 INT(3.8) = 3.00000E+00

Exercises

Exercise 1.

Given a real number, display its integral and fractional parts separately.

Exercise 2.

Assuming that Earth is an ideal sphere with the radius $R=6350\text{km}$, write a program to calculate the distance to the line of horizon from the point of given height from the Earth's surface.

Exercise 3.

Compute and display the sum and the product of three numbers entered using the keyboard.

Exercise 4.

The triangle is specified by the coordinates of its vertices. Compute and display the perimeter and area of the triangle.

Exercise 5.

Compute the height of the tree given the distance to the tree and its viewing angle. The result should be displayed as following:

Tree height equals to 2 m 87 cm

Logical Expressions. Variables of Boolean Type. Logical Operations

The right side of the assignment statement can be not only an arithmetic expression but an expression of another type — for example, a logical expression.

A logical (Boolean) expression is an expression that results in a value of either TRUE or FALSE. The name *Boolean* is chosen in honor of English mathematician George Boole who laid the foundation of mathematical logic. The terms *Boolean* and *logical* are usually used as synonyms.

The value of a logical expression can be assigned to a Boolean variable.

Example of logical variable:

```
Var
Exist: Boolean;
```

Logical expressions can include: arithmetic expressions, relational operators and logical operators.

Relational Operators

Relational operators are used to compare two values. The result of the comparison has either a **TRUE** or **FALSE** value.

=	—	equal to
<>	—	not equal to
<	—	less than
<=	—	less than or equal to
>	—	greater than
>=	—	greater than or equal to

Example:

```

Var
  X: Real;
  Exist, Ok: Boolean;
begin
  X:= 2.5;
  Ok:= X > 0;
  Exist:= X = 3-27;
end.

```

As a result of this program execution, the **Ok** variable stores the **TRUE** value and the **Exist** variable stores **FALSE**.

Logical Operators

Logical operators are used with logical values and they return a logical value too.

Let's examine the following logical operators:

NOT

AND

OR

Logical Operations and Values

X	Y	Not X	X And Y	X Or Y
False	False	True	False	False
False	True	True	False	True
True	False	False	False	True
True	True	False	True	True

The value of the expression is calculated in a certain order.

Table of Operator Precedence

Expression Type	Operator
Evaluation in parentheses	()
Function evaluation	functions
Unary Operators	not, unary '-'
Operators like multiplication	* / div mod and
Operators like addition	+ — or
Relational operators	= <> < > <= >=

Operators with the same precedence are evaluated from left to right according to their order in the expression.

As an example, let's examine the order of the operators and find the value of the next statement:

`(a*2>b) or not (c=7) and (d-1<=3), if a=2, b=4, c=6, d=4.`

`(2*2>4) or not (6=7) and (4-1<=3)`

`(4>4) or not (6=7) and (3<=3)`

`false or not false and true`

`false or true and true`

`false or true`

`True`

The mathematical statement $-4 < x \leq 18.3$ in Delphi will be written as:

`(x > -4) and (x<=18.3)`

Exercises

Exercise 1.

Create a form with a label and a button which turns the label on and off.

Exercise 2.

Traffic light: Create a form with three labels and three buttons. Each button turns on its own label (red, yellow, green) with its corresponding color and turn off the others.

Exercise 3.

Create a form with two text boxes and one button. When the button is pressed, the label True should appear if the number in the first text box is greater than the number in another, and the label False should appear otherwise.

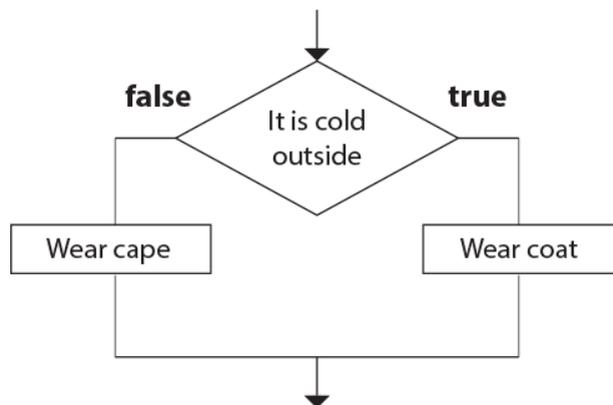
Exercise 4.

A three-digit integer is entered into the text box. Write a program that displays the label True if the sum of any of two digits is equal to the third digit or False otherwise.

Conditional Execution in Program. IF...THEN...ELSE statement

So far we have written programs with a sequential course of actions. In these programs, actions were performed as they were written, one after another. This program structure is called *linear program structure*.

In real life, we often face various choices. We put on coats if it is cold outside, otherwise we wear capes. This situation can be presented as the following flowchart:



Depending on the trueness of the statement "it is cold outside" in the diamond shaped box, we execute either "wear cape" or "wear coat" actions.

This structure is called a *branching structure*.

Choosing one from two or more alternatives is quite common the programming. In Delphi language, the selection is performed using an **If...Then...Else** statement.

Syntax:

```
If <logical expression> Then
    <statement 1>
Else
    <statement 2>;
```

The order of execution of **If...Then...Else**:

First, the *logical expression* is evaluated,

If the *logical expression* evaluated to **True**, then *statement 1* is executed,

Otherwise (*logical expression* evaluated to **False**), *statement 2* is executed.

Example 1.

Given two integer numbers, compute the maximum value of them.

Solution:

```
procedure TfrmMy.btnMaxClick(Sender: TObject);
var
    a, b, m: integer;
begin
    a:=StrToInt(edtA.Text);
    b:=StrToInt(edtB.Text);
    if a>b then
```

```
        m:=a
    else
        m:=b;
    lblMax.Caption:=IntToStr(m)
end;
```

It should be noted that the **If...Then...Else** statement does not need a semicolon before the **Else** keyword, because it is a single complex statement, and there cannot be **Else** without **If**.

You can use compound statements when Delphi language permits only a single action but you need to execute several actions. A compound statement consists of several statements enclosed in the operator brackets **begin...end**.

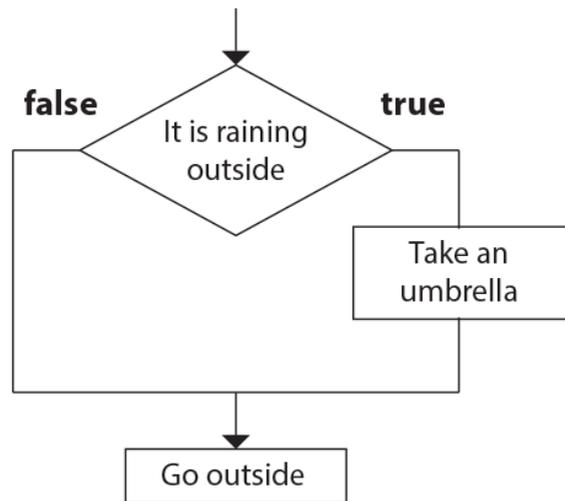
For example, if it is necessary to execute more than single statement in the **Then** or **Else** branches, then these statements are enclosed in the operator brackets **begin...end**:

```
...
If x<0 Then
begin
    y:=1;
    k:=k+1;
end
Else
    begin
        y:=2*x;
        k:=x;
    end;
end;
```

compound statement

compound statement

Sometimes choices are different:



This structure is represented by the shorter statement ***If...Then***.

Syntax:

```
If <logical expression> Then  
    <statement 1>;
```

The short version of the conditional statement gets executed just like the long version. If the *logical expression* evaluates to **True**, then *statement 1* gets executed. Otherwise no statements are executed in the course of execution of ***If...Then*** statement.

Exercises

Exercise 1.

Two integers m and n are entered by the user.

If n is divisible by m , then the result of the division of m by n should be displayed. Otherwise the program should display the message "n cannot be divided by m".

Exercise 2.

Given a three-digit integer, decide whether it is a **palindrome**, i.e., whether it reads the same left to right and right to left.

Exercise 3.

Find the maximum and minimum of three different real numbers.

Exercise 4.

A home contains n apartments numbered sequentially, starting with the number a . Figure out whether the sum of all apartment numbers is even or not.

Nested If...Then...Else statement. Practicing Task Solving

Statements that follow **Then** and **Else** can be conditional statements too; in this case, the If...Then...Else statement is called a nested statement.

For example:

```
If <logical expression 1> Then
  <statement 1>
Else
  If <logical expression 2> Then
    <statement 2>
  Else
    <statement 3>;
```

Each of the instructions (instruction 1, instruction 2, instruction 3) can be compound; i.e., in Then or Else it is possible to place more than one instruction, having bracketed them in a **begin... end** block.

For example:

```
If <logical expression 1> Then
  begin
    <statement 1>
    If <logical expression 2> Then
      begin
        <statement 2>
        <statement 3>
      end
    end
end
```

```
    Else
    <statement 4>;
END
Else
    If <logical expression 3> Then
    <statement 5>
    Else
    begin
    <statement 6>;
    <statement 7>;
    end;
```

Let's examine the next fragment of code:

```
If <logical expression 1> Then
    If <logical expression 2> Then
    <statement 1>
Else
    <statement 2>;
```

It can seem ambiguous, since it isn't clear to what **Then Else** belongs (to the first **Then** or to the second **Then**). There is a rule that **Else** belongs to the closest **If**.

To avoid confusion and possible mistakes, it is desirable to put the nested **If... Then... Else** inside the **begin... end** block.

Exercises

Exercise 1.

Find the square roots of the quadratic equation $Ax^2+Bx+C=0$ ($A \neq 0$). If the equation has no valid roots, display the corresponding message.

Exercise 2.

Find out if a given year is a leap year. A year is a leap year if it is divisible by 4, but among years that are also divisible by 100, leap years are only

those that are divisible by 400. For example, 1700, 1800 and 1900 are not leap years, but 2000 is a leap year.

Exercise 3.

Real numbers X, Y ($X \neq Y$) are given. Replace the smaller number with its arithmetic mean, and replace the greater number with its triple product.

Exercise 4.

Integer k ($1 \leq k \leq 180$) is given. Find what digit is in the k -position of 10111213 ... 9899 sequence of all two-digit numbers in order.

Procedures

It is not sufficient to know the operators of some programming language to write programs with ease, using as little time and introducing as few errors as possible. Rather, it is necessary to become proficient with structured or procedural programming skills and principles.

The practical application of those principles allow you to:

- easily write programs
- create understandable and readable programs
- find program errors faster, and
- more quickly change and improve programs.

One of the main principles of structured programming is called **top-down programming**. This principle states that the main task of any program has to be split into several subtasks. Then, every subtask should be split into even more simple subtasks and this process continued until you end up with a set of small, simple tasks that are easy to implement.

Separate subtasks are called **procedures** in Delphi parlance.

Let us solve the task of finding quadratic equation roots. The equation will be specified as three coefficients.

First we create a new form named **frmQuadEq** and in the **Caption** property we enter "Solution of quadratic equation".

We will create the text boxes **edtA**, **edtB** and **edtC** for values of the corresponding variables A, B and C. We accept the **Text** property of these boxes for faster entry of the coefficients.

A label **lblCoefs** is put above those text boxes; its **Caption** property should contain "Enter coefficients of quadratic equation".

We also need labels **lblX1** and **lblX2** to show the values of the roots of the equation. Another label **lblNo** contains the text “No real roots are found” in the **Caption** property. We set the **Visible** property of these labels to False so that they will not be shown at the program start.

We also need a button **btnFind** to run the solution process.

Let’s double-click and start writing code.

First we document our solution using comments. Comments in Delphi can be placed inside curly braces or, more conveniently, after double slashes //:

```
//Enter quadratic equation coefficients A, B and C
//Compute discriminant D
//IF D >= 0 then
    //Compute equation roots X1 and X2
    //display roots
//else
    //Notifying user that there is no solution
```

This overview of the solution gives us the number of needed variables, procedures, and functions. If a single line of a solution has to be expressed in several Delphi statements, then we will write a procedure; otherwise a single statement will be sufficient.

Given the above consideration, let’s rewrite the solution in Delphi:

```
procedure TfrmQuadEq.btnFindClick(Sender: TObject);
var
    A, B, C, D, X1, X2: real;
begin
    coefInput(A, B, C); //reading of coefficients A,
B, C
    D:=sqr(B)-4*A*C; //compute the discriminant D
    if D>=0 then //If discriminant >= 0, then
```

```
begin
    calc(A, B, D, X1, X2); //compute roots x1, x2
from A, B and D
    prn(X1, X2); //print roots x1, x2
end
else //else
    lblNo.Visible:=true; //Notifying user about ab-
sence of solution
end;
```

In the body of the procedure, we see the names of the actions that the program has to execute to solve the task. Those actions are called procedures; values inside parentheses are called procedure parameters.

When Delphi encounters the name of an action that is not a keyword or statement, it assumes that it is the procedure call. It then transfers the control to the procedure with the given name (in our case, names are `coeffInput`, `calc` and `prn`). After executing the procedure statements execution returns back into the calling routine, and the program executes the next statement after the procedure call.

Procedure call syntax:

```
<procedure name> (<call parameters list>);
```

Procedures should be defined in the **Implementation** section before referring to them in the call. You have to write the procedure before calling it.

Procedure definition syntax:

```
procedure <procedure name> (<formal parameters
list>);
    <declarations>
```

```
begin
    <statements>
end;
```

As we see, the procedure call uses the call parameters, and the procedure definition uses formal parameters. Formal parameters list specifies types for all parameters. There is one-to-one correspondence between call parameters and formal parameters.

The number, order, and types of call and formal parameters should be identical.

Let's define our procedures, beginning with the prn procedure.

In this procedure, the formal parameters X_f and X_s correspond to the call parameters X_1 and X_2 . Formal parameters X_f and X_s are value parameters; they are passed from caller to Prn and are not returned back to the caller.

```
Procedure Prn(Xf, Xs: real);
begin
    frmQuadEq.lblX1.Visible:=true;
    frmQuadEq.lblX1.Caption:='x1'+FloatToStr(xf);
    frmQuadEq.lblX2.Visible:=true;
    frmQuadEq.lblX2.Caption:='x2'+FloatToStr(xs);
end;
```

Here we make labels with the root values visible and put the answers there. We would get an error if we wrote **lblX1.Visible:=true**. In the user-defined procedures, we have to write the complete name of the component (specifying to which form it belongs because there can be several forms in the program, in general).

Let's write a procedure to input coefficients. We have to take coefficients from the text boxes and put them into corresponding variables. For the values of variables to be returned from procedure to main program, we have to use variable parameters. Variable parameters not only pass data to procedures, they also put data back into the variables of the main

program. To define parameters as variables, we have to use the keyword **var** before them. Call parameters should be variables for the procedure's variable parameters:

```
Procedure coefInput (var k1, k2, k3: real);
begin
    k1:=StrToFloat (frmQuadEq.edtA.Text);
    k2:=StrToFloat (frmQuadEq.edtB.Text);
    k3:=StrToFloat (frmQuadEq.edtC.Text);
end;
```

The formal variable parameters $k1$, $k2$ and $k3$ correspond to the call parameters a , b and c . Please note the use of `StrToFloat` function. We have to use it because the `Text` property of a text box is of type string and our variables are type `Real`. We have to convert types using this function, and the use of the function also allows code completion to properly recognize your intent.

Let's define the final procedure — how to compute the roots of equations. We have to return values of roots, so the formal parameters Xf and Xs will be variable parameters. We do not need to return first and second coefficients and discriminant values. They will be passed as value parameters.

```
procedure Calc(k1, k2, dis: real; var Xf, Xs:
Real);
begin
    Xf:=(-k2+Sqrt (dis)) / (2*k1);
    Xs:=(-k2-Sqrt (dis)) / (2*k1);
end;
```

After we run the program and enter the coefficients, we'll see one or another set of labels on the form. Some labels will remain visible if we enter another set of coefficients to produce another set of labels. To get improve the behavior of the program, we'll define another procedure that will put everything into a clean start state. This will be procedure without parameters that we'll call it ***Init***:

```
procedure Init;  
begin  
    frmQuadEq.lblX1.Visible:=false;  
    frmQuadEq.lblX2.Visible:=false;  
    frmQuadEq.lblNo.Visible:=false;  
end;
```

Exercises

Exercise 1.

Create a program that exchanges values between variables **A** and **B** and exchanges values of variables **C** and **D**. Define a procedure that exchanges the values of two variables.

Exercise 2.

Given the lengths of the sides of two triangles, find the sum of their perimeters and their squares. Define a procedure to compute the perimeter and area of a triangle if you know the lengths of its sides.

Exercise 3.

Find a square of a ring with inner radius R_1 and outer radius R_2 . Define and use this procedure to compute the area of a circle.

Exercise 4.

Find an area of convex quadrilateral specified by coordinates of vertices.

Functions

Now let's examine the functions. Functions are the same as procedures except that they return a value in addition to executing statements.

Delphi has a lot of built in functions but there are many situations when you need to write your own functions. For example, we need to solve a task using the tangent function $Tg(A)$ multiple times, but there is no such built in function. Below is a simple example:

There is a right triangle. The angle (in degrees) and the adjacent leg's length are given. Calculate another leg.

Double-click the button to get into code editor:

```
procedure TfrmCatheti.btnRunClick(Sender: TObject);
var
    a, b, alfa: real;
begin
    alfa:=StrToFloat(edtAlfa.Text); //Set angle alfa
    a:=StrToFloat(edtAlfa.Text); //Set adjacent leg a
    b:=a*tg(alfa); //Calculate another leg b
    lblB.Caption:=FloatToStr(b); //Output b
end;
```

We needed only to describe our function. Functions are described in the *Implementation* section. When you declare a function, you specify its name, the number and type of parameters it takes, and the type of its return value.

Function declaration format:

```
Function <name of function> (<list of parameters >):
< type of return value >;
    < local declarations >
begin
    <statements>
end;
```

Function has both value parameters and variable parameters. The parameters are governed by the same rules that apply to the procedure's parameters.

Function should have at least one assignment operator, which assigns a value to the function name.

```
Function tg(x: real): real;
var
    y: real;
Begin
    y:=x/180*pi;
    tg:=sin(y)/cos(y);
End;
```

Here we used one additional variable called a local variable; it is declared inside of the function.

Exercises

Exercise 1.

Calculate the value of the expression: $x = \frac{\sqrt{6}+6}{2} + \frac{\sqrt{13}+13}{2} + \frac{\sqrt{21}+21}{2}$,
using function $y = \sqrt{x} + x$

Exercise 2.

Calculate the value of the expression $x = \frac{15 + \sqrt{8}}{8 + \sqrt{15}} + \frac{6 + \sqrt{12}}{12 + \sqrt{6}} + \frac{7 + \sqrt{21}}{21 + \sqrt{7}}$,
using function $y = a + \sqrt{b}$

Exercise 3.

Given coordinates of three vertices of a triangle, calculate its perimeter.

Exercise 4.

The user will enter variables a, b, c. Calculate the value of the expression

$t = \frac{\max(a, b, c) - \min(a, b, c)}{2 + \max(a, b, c) \cdot \min(a, b, c)}$, where functions min(a, b, c) and max(a, b, c) return, respectively, the minimum and maximum of three numbers.

Exercise 5.

Calculate the value of the expression, where

Exercise 6.

Coordinates of points A, B, and C, lying on the plane, are entered from the keyboard. Find whether it is possible to construct a nondegenerate triangle on these points. If it is possible, calculate its side lengths, heights, area, perimeter and angles in degrees. Output the results on the screen, having specified which angles, sides, heights were calculated. For example, side AB=..., angle BCA=... and so on. When writing the program, define functions to calculate the length of sides, sizes of angle, and heights.

If the triangle can't be constructed, output the corresponding message.

Graphics

Delphi has vast graphics capabilities. We will discuss some of them in this chapter. Drawings will be placed into **TPaintBox** object.

Choose the *Additional* tab on the component palette. In the icon menu for this tab, choose **TPaintBox**. As a painter stretches canvas over a frame, we stretch our component over the form.

Our painting media is the **Canvas** property of component. The origin is in the upper left corner of the object, the X axis points horizontally to the right, and the Y axis points vertically to the bottom. Image dimensions can be viewed in the balloon next to the mouse cursor when it is over the object or they can be queried programmatically through the **Width** and **Height** properties of the object. We can change the name of object to **pbxEx** and compute the center of the **TPaintBox** component:

```
x0:= pbxEx.Width div 2;  
y0:= pbxEx.Height div 2;
```

The **Canvas** image gets constructed using various drawing tools. We can draw using a pen, fill drawings with brushes, and display text using fonts.

To change the color and width of the lines of any figure, we use the **Pen** object:

```
pbxEx.Canvas.Pen.Color:=clRed;  
pbxEx.Canvas.Pen.Width:=3;
```

Figure lines will be drawn using red and slightly wider than the default line width.

The **Brush** object is used to fill the inner space of closed figures.

```
pbxEx.Canvas.Brush.Color:=clGreen;
```

The color of the brush will be green from now on.

A simple example of how to set a brush color to white and fill the whole **Canvas** with white color:

```
pbxEx.Canvas.Brush.Color:= clwhite;  
pbxEx.Canvas.FillRect(ClientRect;
```

The fill and line styles can be set using the **Style** property. We will leave it to the reader to find particulars of working with style on his or her own.

Let's examine standard graphic primitives that can be put into **Canvas**.

Point

To set a point (X, Y) to a particular color, we use the **Pixels[X, Y]** property:

```
pbxEx.Canvas.Pixels[x, y]:=clRed;
```

In other words, only one pixel at cords X and Y is set to a specific color — red here.

Line

Lines can be drawn with the **LineTo** method:

```
pbxEx.Canvas.LineTo(X, Y)
```

A line will be drawn from the current pen position to the specific coordinates. The cursor will be moved to the coordinates specified.

A pen can be moved to point (X, Y) without any drawing using the **MoveTo** method:

```
pbxEx.Canvas.MoveTo(X, Y)
```

All other figures do not move the pen.

Rectangle

To draw a rectangle, we can use the **Rectangle** method:

```
pbxEx.Canvas.Rectangle(X1, Y1, X2, Y2
```

This call will draw a rectangle with axis-aligned sides and a diagonal from (X1, Y1) to (X2, Y2).

The current fill brush and line styles will be used. By default, the line color is black and the fill color is white.

Ellipse

An ellipse is specified by the enclosing rectangle. The ellipse's axes are parallel to X and Y axes. As with a rectangle, ellipse gets drawn filled.

```
pbxEx.Canvas.Ellipse(X1, Y1, X2, Y2)
```

Exercises

Exercise 1.

Draw a snowman.

Exercise 2.

Draw a house.

Exercise 3.

Think out about your own picture and draw it on **Canvas**. Use different primitives, lines, line widths, and fill colors.

Loops

Loop is a repeatedly executed sequence of statements. There are two loop types, loop with a precondition (**While...do...** loop) and post-condition loop (**Repeat...Until...** loop)

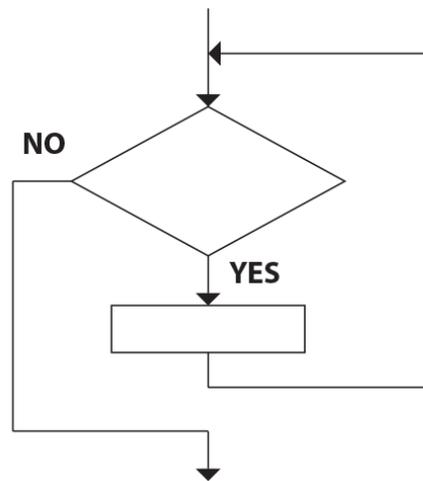
Precondition loop **While...do...** in Delphi has the following format.

```
While <logical expression> do
    <statement>;
```

Loop **While...do** repeatedly executes the same statement if the conditional expression is True. A conditional expression usually changes inside the loop and is calculated before the statement execution. So if the expression value is *False* from the very beginning, the statement won't be executed at all.

If you need to execute multiple statements, you need to use **begin... end** after **do**:

```
While <logical expression> do
begin
    <statement 1>;
    <statement 2>;
    <statement 3>;
end;
```



While...do flowchart

The post-condition loop **Repeat...Until. ...** has the format:

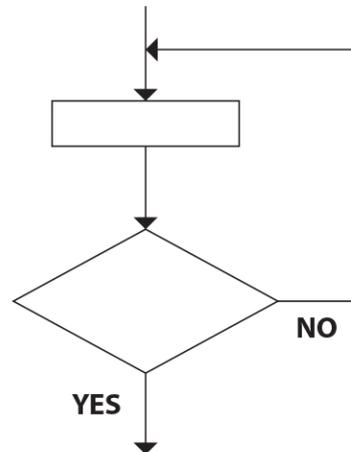
```
Repeat
    <statement 1>;
    <statement 2>;
    ...
    <statement n>;
Until <logical expression>
```

In this kind of loop, the statements from the loop body (the part of the program between **Repeat** and **Until**) are executed first. Then the logical expression is evaluated: if **False**, then the statements execution is repeated, else (if the logical expression is True) the loop ends. So in **Repeat...Until** the loop body is always executed at least once and the logical expression is the condition of loop completion.

If the number of body loop executions is known, you can use the loop with a counter **For**.

Format:

```
For <loop counter>:= <initial value> To <final value>
do
    <statement>;
or
For <loop counter>:= <initial value> DownTo <final
value> Do
    <statement>;
```



Repeat... Until flowchart

Where:

<loop counter> — variable of Integer type, accepting a value with initial to final with a step +1 in case of **To**, and -1 in case of **DownTo**. The value of the loop counter changes automatically

<initial value> — any Integer expression; the initial value of loop counter

<final value > — any Integer expression; the final value of loop counter

<statement> — the statement is executed till the loop counter is not greater (in case of **To**) or less (in case of **DownTo**) than final value. If you need to execute multiple statements, you need to use a **begin...end** block after **do**.

Initial and final values are calculated only once on the loop entry.

Loop **FOR** is also a pre-condition loop.

Exercises

Exercise 1.

Draw N horizontal lines at the same distance from each other. Number N is set through the text box.

Exercise 2.

Draw N squares with side S located horizontally at the same distance from each other. Numbers N and S are set using text boxes.

Exercise 3.

Draw the checkered field consisting of N lines and the M columns. Numbers N and M are set using text boxes.

Exercise 4.

Draw the chessboard consisting of N lines and the M columns. Numbers N and M are set using text boxes.

Exercise 5.

Draw N concentric circles. The radius of the smallest circle is r and the

radius of the largest circle is R. Numbers N, r and R are set using text boxes.

Exercise 6.

Draw the circle consisting of N points of R radius with the set coordinates of the center X_c, Y_c . All parameters are entered into text boxes. Coordinates of a point on the circle are defined as follows:

$$X = X_c + R \cos(\alpha)$$

$$Y = Y_c - R \sin \alpha$$

Angle α changes from 0 till 2π with step $2\pi/n$.

Exercise 7.

Given three numbers representing the lengths of the segments, find out if it is possible to build a triangle with those segments. If the answer is yes, construct the triangle; if not, provide an appropriate error message.

Exercise 8.

Draw N nested squares, having defined a procedure for drawing squares from lines.

Exercise 9.

Draw a simplified analog clock. The time is entered into a text box. When a button is pressed, the hands show the time.

Strings

We have already had some experience with strings in Delphi. For example, properties like **Caption** or **Text** can only have string values. What are strings exactly, and how can you work with them?

A string is a sequence of symbols inside single quotes. To declare a string variable, you have to use type **String**:

```
Var  
    s: String;
```

This definition means that the program can use a string variable with unlimited length.

Strings can be concatenated. String concatenation is denoted by the plus sign. For example:

```
Var  
    s, st: String;  
Begin  
    s:='We learn'; //We put "We learn" into first  
variable  
    st:=' Delphi; //We put " Delphi" into second  
variable  
    s:=s+st; //Concatenate these two strings.  
End;
```

When the program executes, we will get the string "We learn Delphi" in the S variable.

Strings can be compared.

Comparison is performed symbol by symbol using the comparison relation of symbols (comparing their internal representation). Latin characters have alphabetical ordering and digits have a natural ordering where $0 < 1 < \dots < 9$.

Example:

'AB' > 'AA'

'A' < 'AB'

'DC' > 'ABCDE'

'ABCE' > 'ABCD'

Several standard functions and procedures work with strings. The following table summarizes them.

Standard String Functions and Procedures

String Procedures		
Name and Parameters	Parameter Types	Semantics
Delete(St, Pos, N)	St: string; Pos, N: integer;	Delete N symbols from string St, starting from position Pos.
Insert(St1, St2, Pos)	St1, St2: string; Pos: integer;	Insert symbols of string St1 into string St2 starting from position Pos.
String functions		
Name and Parameters	Parameter Types	Semantics
Copy(St, Pos, N)	Result type: string; St: string; Pos, N: integer;	Result is the copy of N symbols from string St starting from position Pos.
Length(St)	Result type: integer; St: string;	Computes the length (symbol count) of string St.
Pos(St1, St2)	Result type: integer; St1, St2: string;	Searches for the first occurrence of substring St1 in string St. The result is the position of the first substring symbol. Returns 0 if the substring was not found.

Examples:

Delete procedure:

Value of St	Statement	Value of St after Statement Execution
'abcdef'	Delete(St,4,2)	'abcf'
'Turbo-Pascal'	Delete(St,1,6)	'Pascal'

Insert procedure:

Value of St1	Value of St2	Statement	Value of St2 after Statement Execution
'Turbo'	'-Pascal'	Insert(St1, St2,1)	'Turbo-Pascal'
'-Pascal'	'Turbo'	Insert(St1, St2,6)	'Turbo-Pascal'

Copy function:

Value of St	Statement	Value of Str after Statement Execution
'abcdefg'	Str:=Copy(St,2,3);	'bcd'
'abcdefg'	Str:=Copy(St,4,4);	'defg'

Length function:

Value of St	Statement	Value of N after Statement Execution
'abcdefg'	N:=Length(St);	7
'Turbo-Pascal'	N:=Length(St);	12

Pos function:

Value of St2	Statement	Value of N after Statement Execution
'abcdef'	N:=Pos('de', St2);	4
'abcdef'	N:=Pos('r', St2);	0

Extended example:

The input of the procedure is a string that contains two words delimited by single space symbol.

This procedure should change the order of the words.

```
Procedure Change(var s: String);
var
    s1: string;
begin
    s1:= copy(s,1, pos(' ', s)-1); // copy first word
in s (word is everything before the space)
    delete(s,1, pos(' ', s)); //delete first word and
space(s will contain second word)
    s:=s+' '+s1; //add space and first word to the end
end;
```

Note. The symbol ' ' means space; it is used for readability.

Exercises

Exercise 1.

Write a program that, after a press of a button, exchanges the second and third words of the text in the text box. The text-box content should contain exactly three words, separated by a space.

Exercise 2.

When the button is pressed, change all spaces in a text box to exclamation marks.

Exercise 3.

Count the number of periods in the text box string.

Exercise 4.

Count the number of occurrences of the string "abc" in the text box content.

Exercise 5.

A string with one open and one close parenthesis is entered into a text box. Output a string between these parentheses in a different text box.

Exercise 6.

Count the number of words separated by a space in a string entered into a text box.

Exercise 7.

Replace all occurrences of word "dog" with the word "cat" in the text box.

Exercise 8.

Reverse a string in a text box.

Exercise 9.

Count the number of space-separated words in text box. Switch the second and next-to-last words.

Strings and Conversion To or From Numeric Types

Procedure **Val(st, X, code)** transforms the string **st** into an integer or real value that is stored in variable **X**. The resulting type depends on the type of **X**. The variable parameter **Code** indicates success or error.

After computation, if the transformation was successful, the **Code** variable contains zero and the result is placed into **X**. If not, it contains the position index of an erroneous symbol in the **st** string and the value of **X** remains unchanged.

The **st** string can contain leading or trailing spaces. For real values, the integer and fractional parts are delimited by a period.

Example:

To check the correctness of a keyboard input of numeric data and their transformation into variables **X** and **Y**, we will use the procedure **Val**.

Variables **cx** and **cy** will be set to zero, if strings **sx** and **sy** contain valid representations of numbers. Otherwise, they will have non-zero values.

```
Val (sx, x, cx);
Val (sy, y, cy);
If (cx=0) and (cy=0) Then
    begin
        <statements>
    end
Else
    begin
        <statement>
```

end

Procedure **Str**(*X* [*:* **Width** [*:* **Decimal**]], *st*) transforms the integer or real number *X* into the string *st*. Parameters **Width** and **Decimal** control the transformation.

Parameter **Width** defines the complete length of the resulting string representation of *X*. Parameter **Decimal** defines the number of symbols for the fractional part of the number and is only applicable for real values.

Example:

```
var
  x: integer;
  y: real;
  s: string;
begin
  y:=3.5;
  Str(y, s);
  ... .;
  ... .
end;
```

Variable *s* will contain **3.500000000000000E+0000**.

However, if we apply transformation like this **str(y:6:2, s)**, the result will be `␣3.50`. The missing (zero) leading symbols of the integer part are replaced with spaces (denoted by `␣` sign).

Another example is:

```
Str(12345.6:5:2, s) .
```

In this case, the integer part is completely output because the constraint cannot be satisfied.

The result is **12345.6**.

A further example is:

```
Str(12.3456:5:2, s) .
```

In this case, the fractional part will be rounded to the specified number of decimal places.

The result is **12.35**.

Exercises

Exercise 1.

A text box contains a string. Delete all of the digits from the string and display the result in a separate label.

Exercise 2.

A text box contains a string. Compute the sum of all the digits in the string.

Exercise 3.

A text box contains a string with words and numbers, which are separated by a single space. Compute the sum of all the numbers.

Exercise 4.

A text box contains a string in the form of **number1+number2=**. Put the sum of **number1** and **number2** after the equal string.

Example:

```
Input string: 12.35+7.123=
```

```
Resulting string: 12.35+7.123=19.473
```

Exercise 5.

A text box contains a string with one open and one close parenthesis. There are several digits between the parentheses. Compute the average of the digits. Replace the digits inside the parentheses with the computed result.

Exercise 6.

Two text boxes contain strings in the form of **Name space Number**, where **Number** is the height of a man in centimeters. Output the name of the tallest man. If the heights are equal, output both names.

Exercise 7.

A text box contains a number. Output this number in binary.

Exercise 8.

A text box contains a number. Output this number in hexadecimal.

TMemo Control

TMemo is an edit box that handles multiple lines of text.

You can dynamically create **TMemo** content using its properties and methods.

Property **Font** and **ReadOnly** of **TMemo** are the same as the properties of **TEdit**. The text property has all of the **TMemo** text but this property is only available when the application is running.

Lines Property

The text of the **TMemo** control is stored in **Lines** property and represents the numbered set of lines (numbering starts from 0) — the first line has an index of 0 and the second has an index of 1, etc.

Example:

```
Var
    s: string;
Begin
    s:= memEx.Lines[2]; //the value of the 2nd line
is assigned to s variable
    memEx.Lines[3]:='Hi'; //string constant 'Hi'
//is assigned to the 4rd line
... .
end;
```

A program can only access the lines of **TMemo** control in this way if the lines have been created via the Object Inspector, entered from the keyboard during program execution or created by the corresponding

method. An attempt to access a non-existing line will cause a run time error.

You can fill **TMemo** using **Lines** property. To do this, press the button with an ellipsis on **Lines** property in the Object Inspector. The **Lines** editor window will open. You can input the lines in the window, pressing **<Enter>** at the end of each line.

When you have finished, press the **<OK>** button.

Count Property

Count property returns the total number of lines.

```
k:=memEx.Lines.Count;
```

Count is **ReadOnly**, i.e., you cannot change/edit it.

WordWrap Property

If **WordWrap** is true, the edit control wraps the text at the right margin so that it fits in the client area.

MaxLength Property

MaxLength defines the maximum number of characters that can be entered into the edit control. If the **MaxLength** is 0, the **TMemo** does not limit the length of the text entered.

ScrollBars Property

ScrollBars property determines whether the **TMemo** control has scroll bars. **TMemo** ignores the **MaxLength** property if it is vertical or if both of the scrollbars are turned on.

Alignment Property

Alignment property changes the way that the text is formatted in the text edit control.

Methods

Let us examine some methods.

Delete Method

This method deletes the line with the specified index. Other lines will automatically be moved and renumbered.

```
memEx.Lines.Delete(0); //deletes the first line,  
i.e. line with index 0
```

Example:

Only leave the odd lines.

```
k:=memEx.Lines.Count;  
For i:=k-1 Downto 1 Do  
  if (i mod 2) <>0 then  
    memEx.Lines.Delete(i);
```

Exchange Method

Exchange swaps the positions of two lines with the specified indexes.

```
memEx.Lines.Exchange(0,1); // swaps 0 and 1 lines
```

Move Method

This moves a line to a new position.

Example:

```
memEx.Lines.Move(1, 5);
```

During this method execution, the line with index 1 (i.e., the second line, as numbering starts at 0) will be removed from the text. All of the lines following it will be moved up one position and the fifth line will replace the line.

Exercises

Exercise 1.

Fill a **TMemo**. Write a program that, by clicking on the button, only leaves the following in the **TMemo**:

- The first four lines, if there are more than four lines.
- The first line, if there are less than four lines.

Exercise 2.

Fill a **TMemo** and text edit.

Write a program that, by clicking on the button, outputs to the label:

- a) The index of the line in **TMemo**, which is the same as the line in the text edit, or message that there is no such line.
- b) The indexes of the lines that contain a combination of symbols that have been entered into the text edit or message that there are no such lines.

Exercise 3.

Fill a **TMemo** with text of a few lines. One line should have 'cat' and another line should have 'dog'. Write a program to swap them.

Exercise 4.

Fill a **TMemo**.

Write a program to find the longest line and put it to the top.

Exercise 5.

Fill a **TMemo** with numbers, one number per line.

Write a program that will double all of the even numbers and move them to the second **TMemo** control. Divide all of the odd numbers by two and move them to the third **TMemo**.

TMemo Control (Continued)

Let us examine a few more useful properties and methods.

Clear

Completely clears **TMemo** content.

```
memEx.Lines.Clear
```

Methods Append and Add

If the **TMemo** is empty or **Clear** was called, then you can fill a **TMemo** by calling the **Append** method. The **Append** method adds a new line to the end of a **TMemo**.

Example:

Fill a **TMemo** with numbers from 1 to 10, one number per line.

```
...
memEx.Lines.Clear;
For i:=1 to 10 do
begin
    memEx.Lines.Append (IntToStr(i));
end;
```

Method **Add** works the same as **Append**.

```
memEx.Lines.Add ('This example uses A string
List.');
```

As you can see, both of the methods use string as a parameter.

Add returns the index of the new string.

```
b:=memEx1.Lines.Add(edtEx1.Text);
lblEx1.Caption:=IntToStr(b);
```

In this example, the text from the text box edtEx1 was added to the **TMemo**, called memEx1. The number of the new line was output to the label lblEx1.

You can remove, save and recover text fragments from **TMemo** using **Clipboard**.

Insert Method

This inserts a line with a specified index. Other lines will automatically move.

```
memEx.Lines.Insert(2,''); // adds empty line
instead of the line with index 2, which is moved, not
deleted
```

Lines Sorting in TMemo

The sorting of lines can be carried out using the following algorithm.

Find the index of a line with the minimal value (strings are compared character by character A<B<C ..<Z and, etc.)

The first line and the minimum line are swapped.

Repeat these statements (N-1) times, where N — number of lines in the **TMemo**.

Let us write this program.

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);
begin
  //For every i-line of TMemo
```

```
    //find index of minimal line starting from i
    //swap it with current.
end.
```

We describe the needed variables, operators and the function of finding the minimal line.

As a result, we will get the following program:

```
function PlMin(numStr: integer): integer;
var
    k, m: integer;
begin
    m:=NumStr;//Let line on the num_str-position is
minimum
    for k:=m+1 to frmEx1.memEx1.Lines.Count do//For
all //lines starting from num_str position
        if frmEx1.memEx1.Lines[k]< frmEx1.memEx1.
Lines[m] then //If current line less than minimum
then
            m:=k;//change position of minimal
            PlMin:=m;//assign the minimal line position to
function
end;
procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
    i, j: integer;
begin
    for i:=0 to memEx1.Lines.Count do //For every
line of TMemo
        begin
            j:=plmin(i); //find position of minimal line
starting from i
```

```
    memEx1.Lines.Exchange(i, j); //swap them  
end;  
end;  
end.
```

Exercises

Exercise 1.

Fill a **TMemo** with any numbers, one number per line.

Write a program to put all of the positive numbers into the second **TMemo** and all of the negative numbers into the third **TMemo**.

Exercise 2.

Fill a **TMemo** with a class name list. Write a program to sort the list.

Exercise 3.

Fill two **TMemos**.

Write a program that, by clicking on the button, outputs the identical lines from the first and second **TMemos** to the third **TMemo**, otherwise outputs the message that there are no identical lines.

Exercise 4.

Fill a **TMemo** with numbers, one number in each line.

Write a program that sorts the numbers.

Random Numbers, Constants, User Types and Arrays

RANDOM Function

We can use **Random** function to obtain evenly distributed random numbers. This function can be called in two ways:

- Calling **Random** without any parameters will return a random **real** number in the range of 0 (inclusive) to 1 (non-inclusive);
- Calling **Random(N)** with an integer parameter **N** will return an integer random number in the range of 0 to **N-1**.

If you need to obtain a random number in the range of a to b (both inclusive), you have to use the following formula:

$$\text{Random}(b-a+1) + a.$$

If you need a random real number in the range of a (inclusive) to b (non-inclusive), you can use the following formula:

$$(b-a) * \text{Random} + a.$$

Every run of the program will produce the same sequence of random numbers.

You have to call Randomize procedure to obtain different sets of random numbers in your program. To assure that you do not accidentally restart a random number generator from (possibly) the same point, organize your program in such a way that Randomize is only called before any calls to **Random**.

Named Constants

Constants are entities that cannot change their values during a program execution. Just like variables, constants have types. They can be anonymous and named.

Examples of anonymous constants:

100	—	Integer
2.7	—	Real
7.12457e-03	—	Real
TRUE	—	Boolean
14.0	—	Real
-37	—	Integer
'asderc'	—	String

It is very useful to have named constants. Constants are declared in the declaration part of a program, along with its value.

Named constant definition syntax:

```
Const
    <constant name> = <value>;
```

Rules governing constant names are the same as they are for variable names. The type of named constant is automatically derived from its value.

Named constants are usually used to enhance readability and program maintenance, as well as to denote critical parameters.

User-defined Types

Delphi allows us to define our own types.

An example of user type definition and its use is:

```
type
    kg = integer;
```

```
    cm = integer;  
var  
    Weight: kg;  
    Size: cm;
```

User types improve program readability. Program maintenance also gets better — we can always change the definition of a type (for example, from ***Integer*** to ***Real***) if, during program development, we find an inadequacy in the current type's definition. The change only needs to happen in one place, not in all of the definitions of all the variables of an inadequate type.

Exercises

Exercise 1.

Let us play a game — “Guess the Number”.

Let the computer choose a random number from the range, specified by two text boxes. The third text box will be used to enter our guesses.

Write a program that shows the result of a guess after pressing a button. The result can either be an acknowledgement of the correct guess or a comparison between the computer-chosen number and our guess (larger or smaller).

Add a button that allows users to peep the computer chosen number.

Single Dimensional Static Array

An **Array** is a collection of elements (values or variables of the same type), which are each identified by an array index or key with a general name. For example, a class list. In this array, the name of every element is 'Student' and every student's name has a position in the list (index). In this case, the name is the value of the element.

Array format:

```
<array name>: Array [<initial index>..  
final index>] Of <element type>;
```

An array format includes:

Array name.

Keyword — **Array**.

The initial and final value of index — these set the index range, e.g., 1..40, —2..2, 0..10. The lower limit is the minimal possible value of index and the higher limit shows the greatest value. The lower limit cannot be greater than the higher limit. They are separated by two dots. The index's type must be an enumeration, i.e., a type with predefined previous and next elements. Now we know the following enumeration types: all of the integer types and the Boolean type. Only constants can be used as an index, i.e., the size of an array is determined during program writing and cannot be changed during the run time.

Element's type.

An array example is:

```
A: array[1..10]of real;
```

Array A consists of 10 elements and every element is a real number.

You can access an array element by an index. An index can be an expression of a proper type.

For example:

```
X:=5;  
A[x+2]:=4.5
```

Here, the value of 4.5 is assigned to the array element with an index of 7 (5+2).

Access via an index allows tasks to be solved more efficiently. For example, to assign 0 to all of the array elements, it is enough to write the loop:

```
For i:=1 To 10 do  
A[i]:=0;
```

To use an array as a parameter of a procedure, its own type should describe the array, for example:

```
Type  
MyArray = array[1..30] of integer;  
.....  
.....  
Var  
A: MyArray;
```

Let us examine the task of filling an N-element array. The values of array elements should be in the specified range. Output the array values to a **TMemo** and find the maximum element.

Write the algorithm in comments.

```
Begin  
//Set range of array values
```

```
//Fill array with numbers from the specified range
//Output array into TMemo
//Find maximal element Amax
//Output Amax
End.
```

Let us write a statement to every phrase:

```
Begin
    SetArrayRange(rMin, rMax); //Set range of array
values
    FillArray(a, rMin, rMax); //Fill array with
numbers from the specified range
    OutputArray(a, n); //Output array into TMemo
    Amax:=max(a); //Find maximal element Amax
    lblAmax.Caption:=IntToStr(Amax); //Output Amax
End.
```

Before we write these procedures and functions, please note that an array should be set as a parameter of the procedure, described by its own type.

```
Const
    n=20;
Type
    array_n_elements=array[1..n] of integer;
procedure SetArrayRange(var ch1, ch2: integer);
begin
    ch1:=StrToInt(frmArr.edtCh1.Text);
    ch2:=StrToInt(frmArr.edtCh2.Text);
end;
procedure FillArray(var a: array_n_elements; rMin,
rMax: integer);
```

```
var
    i: integer;
begin
    randomize;
    For i:=1 to n do
        a[i]:=random(rMax-rMin)+rMin;
    end;
    procedure OutputArray(a: array_n_elements; n:
integer);
    var
        i: integer;
    begin
        frmArr.mem_ish.Lines.Clear;
        for i:=0 to n-1 do
            frmArr.memIsh.Lines.Append(IntToStr(a[i+1]));
        end;
    function max(a: array_n_elements): integer;
    var
        i, m: integer;
    begin
        m:=a[1];
        for i:=1 to n do
            if a[i]>m then
                m:=a[i];
                max:=m;
            end;
        end;
    procedure TfrmArr.btnClick(Sender: TObject);
    var
        rMin, rMax: integer;
        Amax: integer;
        A: array_n_elements;
```

```
begin
    SetArrayRange(rMin, rMax); //Set range of array
values
    FillArray(a, rMin, rMax); //Fill array with
numbers from the specified range
    OutputArray(a, n); //Output array
    Amax:=max(a); //Find maximal element Amax
    lblAmax.Caption:=IntToStr(Amax); //Output Amax
end;
end.
```

Exercises

Exercise 1.

Fill an array with random numbers from the range. Input the range limits via text boxes.

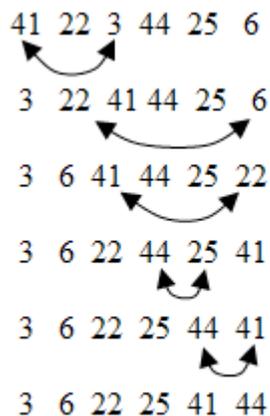
- Output the array to a **TMemo**.
 - To labels, count and output the:
 - Sum of the elements.
 - Average of the elements.
 - Quantity of the positive and negative elements.
- Maximal and minimal elements.
- Split the array into two arrays — one with positive elements and the other with negative elements. Output the arrays to two new **TMemos**.

Array Sorting and Selection Sort

The process of sorting can be defined as a process of ordering a set of objects by comparing their attributes. Array sorting is a well-known area in computer science and there are many algorithms that are used to sort arrays. We will examine one of the simplest and most understandable array sorting algorithms — the selection sort.

An algorithm finds the smallest element in an N-element array and exchanges it with the first element in the array. Then, the algorithm finds the smallest element in the remaining (N-1) elements and exchanges it with the second element in the array. The same operation is performed with N-2 elements, N-3 elements and so on, until there is only one element left. This element is the largest.

An example of algorithm execution is shown below. The arcs under the text show the exchanges of elements:



Let us examine the sorting of an array in ascending order. First, we write the algorithm in comments.

```

//Input of array value range, chB and chE.
//Filling array A with numbers in the range from
chB to chE (inclusively).
//Output array A to first memo field..
//Sort the array A in ascending order.
//Output array A to the second memo field.

```

Next, we write the algorithm as a sequence of procedure calls.

```

procedure TfrmArr.btnGoClick(Sender: TObject);
var
    chB, chE: integer;
    A: array_n_elements;
begin
    Input(chB, chE); //Input array value range,
//from chB to chE (inclusively).
    FillArr(a, chB, chE); //Fill the array with
random numbers
//in the range from chB to chE
    OutputArray1(a, n); //Output array A into TMemor
1.
    SortArr(a); //Sort the array A in ascending
order.
    OutputArray2(a, n); //Output array A, into TMemor
2.
end;

```

The first three procedures were discussed in the previous module. Here, we will only write the procedure to sort the array. It is almost the same as the procedure to sort a ***TMemo***.

```

procedure SortArr(var a: array_n_elements);
var

```

```
    i: integer;
begin
    for i:=1 to n-1 do //For each element in array
        change(a[i], a[NumMin(a, i)]); //exchange current
element and the least element in the range from
current to the end of array.
    end;
```

Before this procedure, we will write a procedure to exchange the two elements, given their indices and function, in order to find the smallest element, starting from the specified position:

```
function NumMin(a: array_n_elements; start:
integer): integer;
var
    i, m: integer;
begin
    m:=start;
    for i:=m+1 to n do
        if a[i]<a[m] then
            m:=i;
        NumMin:=m;
    end;
procedure change(var one, two: integer);
var
    temp: integer;
begin
    temp:=one;
    one:=two;
    two:=temp;
end;
```

Exercises

Exercise 1.

Enter a value range from the keyboard, fill the array with the integer values from the range entered.

Output the array into a **TMemo**.

Sort the array in descending order and output it into a second **TMemo**.

Exercise 2.

Enter a value range from the keyboard and fill the array with integer values from the range entered.

Output the array into a **TMemo**.

Put all of the positive elements in the beginning of the array and all of the zero and negative elements to the end of array (**do not use sorting**). Output the array into a second **TMemo**.

Exercise 3.

Enter a value range from the keyboard, fill the array with real values from the range entered.

Output the array into a **TMemo**.

Exchange the first half of the array with the second half. Output the array into a second **TMemo**.

StringGrid Control

StringGrid displays and collects string data in a grid format.

StringGrid is on the **Additional** tab. We use the prefix **sgd** for the names of this component. For our examples, we will use the name, **sgdMy**.

A grid can have fixed cells. The fixed cells need to display headers of columns and rows and for manually changing their sizes.

Usually, the left column and top row are fixed. However, using **FixedCols** and **FixedRows** properties, you can set other numbers of fixed columns and rows. If these properties are set to 0, the grid has no fixed part.

Generally:

```
sgdMy.FixedCols:=1; //Number of fixed columns = 1
sgdMy.FixedRows:=1; //Number of fixed rows = 1
```

Unfixed cells of a grid can contain any number of columns and rows and the number can be changed programmatically. If a grid is larger than the component window, the corresponding scrollbars will be automatically added. When the grid is scrolled, fixed rows and columns do not scroll out of sight but their content changes (the headers of rows and columns).

Cells are the main property of **StringGrid**. **Cells** property is the collection of cells and every cell can have text. Cells have two coordinates — the column coordinate of the cell and the row coordinate of the cell. The first row is row zero and the first column is column zero.

Cells have type **String**.

Cells can be manually filled during run time. To allow this, you should press + on the Object Inspector to open property **Options** and set **True** to property **goEditing**.

The cell value can be set programmatically, using an assignment operator. Use the indexes to access the cell. Remember that **the first index is the column number and the second index is the row number**.

Here is an example:

```
sgdMy.Cells [1,1]:= 'The left top unfixed cell';
sgdMy.Cells [0,0]:= 'Numbers: ';
```

String numbers will be set to the first cell of the grid, i.e., to the first cell of the fixed part.

The **ColCount** and **RowCount** property values define the size of the grid.

ColCount and **RowCount** can be changed during program creating and the run time. However, their values should be greater by at least one more than the corresponding values of **FixedCols** and **FixedRows**.

Let us set the columns' count to three and the rows' count to five in sgdMy.

```
sgdMy.ColCount:=3;
sgdMy.RowCount:=5;
```

The property **FixedColor** sets the color of the fixed cells and **Color** property sets the color of the other cells.

Let us write a program as an example. By clicking on the first button, the **StringGrid** is created. The number of columns, rows, fixed columns and fixed rows are entered into four text boxes. By clicking on the second button, the fixed cells are painted in a green color. By clicking on the third button, the other cells are painted in a red color.

```
Procedure GetGridParam(var n1, n2, n3, n4:
integer);
Begin
    n1:=StrToInt (frmGrid.edtLine.Text);
    n2:=StrToInt (frmGrid.edtStolb.Text);
```

```
    n3:=StrToInt (frmGrid.edtFline.Text);
    n4:=StrToInt (frmGrid.edtFstolb.Text);
End;
Procedure CreateGrid(n1, n2, n3, n4: integer);
Begin
    frmGrid.sgdMy.RowCount:=n2;
    frmGrid.sgdMy.ColCount:=n1;
    frmGrid.sgdMy.FixedCols:=n4;
    frmGrid.sgdMy.FixedRows:=n3;
end;
procedure TfrmGrid.btnTablClick(Sender: TObject);
var
    nl, ns, nfl, nfs: integer;
begin
    GetGridParam(nl, ns, nfl, nfs); //Gets grid
parameters
    CreateGrid(nl, ns, nfl, nfs); //Creates grid the
defined parameters
end;
procedure TfrmGrid.btnCelRedClick(Sender: TObject);
begin
    frmGrid.sgdMy.Color:=clRed;
end;
procedure TfrmGrid.btnFCGreenClick(Sender:
TObject);
begin
    frmGrid.sgdMy.FixedColor:=clGreen;
end;
```

Some Useful StringGrid Properties

Property	Description
BorderStyle: TBorderStyle;	Specifies the appearance of the grid border: bsNone — no border; bsSingle — 1 pixel border.
ColCount: LongInt;	Contains the number of grid columns.
DefaultColWidth: Integer;	Contains the default column width.
RowCount: LongInt;	Contains the number of grid rows.
DefaultRowHeight: Integer;	Contains the default row height.
Color: TColor;	Controls the color of the cells.
FixedCols: Integer;	Controls the number of fixed columns.
FixedRows: Integer;	Controls the number of fixed columns.
FixedColor: TColor;	Controls the color of the fixed cells.
GridHeight: Integer;	Contains the grid height.
GridWidth: Integer;	Contains the grid width.
GridLineWidth:	Specifies the width of the grid lines.

Exercises

Exercise 1.

Create a **StringGrid**.

In text boxes, set the:

- Number of rows.
- Number of columns.
- Number of fixed rows.
- Number of fixed columns.

Create three buttons for changing the color of the fixed cells (one button for one color).

Create three buttons for changing the color of the unfixed cells (one button for one color).

Create a text box and two buttons.

After clicking on the first button, the **TMemo** displays the column with an index, which is specified in the text box. After clicking on the second button, the **TMemo** displays the row with an index, which is specified in the text box.

Exercise 2.

Create a **StringGrid**.

Students are conducting experiments in a Physics laboratory. The following data are given: the number of experiments, scale interval and values, which are measured in the scale intervals.

Input the number of experiments (5–10) and the scale intervals into text boxes.

The values measured are added into the table from the keyboard. For all of the experiments, add to the table values and calculate the minimum and maximum values.

For example, the number of experiments = 5 and the scale intervals = 10. Then, the table will look like:

Number of Experiments	Scale Intervals	Measured Values	Result	Max/Min
1	10	83	830	Max
2	10	51	510	
3	10	67	670	
4	10	49	490	Min
5	10	75	750	

StringGrid Practice

Finding the numbers of elements with a specific attribute.

Problems of finding the number of specific grid elements are solved almost the same as for a **TMemo** or single-dimensional array. You only need to add the second loop for the second index.

Exercise 1.

Find the number of negative elements in every row of the integer matrix.

Solution

Fill a **StringGrid** using the random numbers generator. Remember that numbering starts at 0. The result will be output to the **TMemo**.

Let us write a **Delphi** program.

We will put two text boxes on the form to set the number of rows and columns, **StringGrid** and **TMemo**, as well as two buttons:

1 — Creates and fills **StringGrid**.

2 — Calculates the number of negative elements in every row.

Let us write two procedures to handle the buttons' click event.

```
procedure TForm1.Button1Click(Sender: TObject);
var
    i, j, n, m: integer;
begin
    n:=strtoint(edit1.Text);
    m:=strtoint(edit2.Text);
```

```
    stringgrid1.RowCount:=n;
    stringgrid1.ColCount:=m;
    for i:=0 to n-1 do
    for j:=0 to m-1 do
    stringgrid1.Cells[j,
i]:=inttostr(random(100)-50);
    end;
procedure TForm1.Button2Click(Sender: TObject);
var
    i, j, n, m, k: integer;
begin
    n:=stringgrid1.RowCount;
    m:=stringgrid1.ColCount;
    for i:=0 to n-1 do
begin
    k:=0;
    for j:=0 to m-1 do
    if strtoint(stringgrid1.Cells[j, i])<0 then
    k:=k+1;
    memol.Lines.Append(inttostr(k));
end
end;
```

Exercise 2.

Replace all of the negative elements of a matrix with the opposite ones.

Possible solutions:

- Multiply the negative element by (-1).
- Find the absolute values of the negative elements.
- Put a minus sign before the negative element.

Program 1

```
procedure TForm1.Button2Click(Sender: TObject);
var
    i, j, n, m: integer;
begin
    n:=stringgrid1.RowCount;
    m:=stringgrid1.ColCount;
    for i:=0 to n-1 do
begin
    for j:=0 to m-1 do
        if strtoint(stringgrid1.Cells[j, i])<0 then
            stringgrid1.Cells[j, i]:=inttostr((-
1)*strtoint(stringgrid1.Cells[j, i]));
        end
    end;
end;
```

Program 2

```
procedure TForm1.Button2Click(Sender: TObject);
var
    i, j, n, m: integer;
begin
    n:=stringgrid1.RowCount;
    m:=stringgrid1.ColCount;
    for i:=0 to n-1 do
begin
    for j:=0 to m-1 do
        stringgrid1.Cells[j, i]:=inttostr(abs(strtoint(s
tringgrid1.Cells[j, i])));
    end
end;
```

Program 3

```
procedure TForm1.Button2Click(Sender: TObject);
var
    i, j, n, m: integer;
begin
    n:=stringgrid1.RowCount;
    m:=stringgrid1.ColCount;
    for i:=0 to n-1 do
    begin
        for j:=0 to m-1 do
            if strtoint(stringgrid1.Cells[j, i])<0 then
                stringgrid1.Cells[j, i]:=inttostr(-
strtoint(stringgrid1.Cells[j, i]));
        end
    end;
end;
```

Exercise 3.

Find whether or not a square matrix is symmetric, with respect to the main diagonal.

Solution

Use the rule: the matrix is symmetric, if for every $i = 1, 2, \dots, n$ and $j = 1, 2, \dots, n$, where $i > j$, ***StringGrid1.Cells[i, j] = StringGrid1.Cells [j, i]***. So, we can write the program:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    i, j, n: integer;
begin
    n:=strtoint(edit1.Text);
    stringgrid1.RowCount:=n;
    stringgrid1.ColCount:=n;
```

```

    for i:=0 to n-1 do
    for j:=0 to n-1 do
    stringgrid1.Cells[j, i]:=inttostr(random(2));
    label2.Caption:='';
end;
procedure TForm1.Button2Click(Sender: TObject);
var
    i, j, n, m: integer;
    simm: boolean;
begin
    n:=stringgrid1.RowCount;
    simm:= True;
    {Assume that the matrix is symmetric}
    i:=2;
    While simm and (i < n) Do
    Begin
        j:=1;
        While (j < i) and (stringgrid1.Cells[j, i]
= stringgrid1.Cells[i, j]) Do
            j:=j+1;
            simm:=(j=i);
            i:=i+1
    End;
    if simm then
        label2.Caption:='Matrix is symmetric'
    else
        label2.Caption:='Matrix is not symmetric'
    end;
end;

```

Exercise 4.

Fill $n \times m$ matrix in a special way:

1 2 3 4 5 ... n
... n+2 n+1

Solution

To fill the matrix in this way, we need to define the rule of filling. In this case, the rule will be:

If the row is odd, then ***stringgrid1.Cells[j, i] := inttostr(i*m + j+1)***, or else, (if the row is even) then ***stringgrid1.Cells[j, i] := inttostr((i+1)*m-j)***.

We will write a program according to this rule:

```
procedure TForm1.Button1Click(Sender: TObject);
var
    i, j, n, m: integer;
begin
    n:=strtoint(edit1.Text);
    m:=strtoint(edit2.Text);
    stringgrid1.RowCount:=n;
    stringgrid1.ColCount:=m;
    for i:=0 to n-1 do
        for j:=0 to m-1 do
            if i mod 2 =0 then
                stringgrid1.Cells[j, i]:= inttostr(i*m + j+1)
            else
                stringgrid1.Cells[j, i]:= inttostr((i+1)*m-j)
        end;
    end;
```

Exercises

Exercise 1.

Fill $n \times n$ matrix with 0 and 1, according to the following scheme (for example, for a 5×5 matrix):

```
1 1 1 1 1
0 1 1 1 0
0 0 1 0 0
0 0 0 0 0
0 0 0 0 0
```

Exercise 2.

Fill $n \times n$ matrix, according to the following scheme (for example, for a 5×5 matrix):

```
0 0 0 0 0
0 0 0 0 0
0 0 1 0 0
0 1 1 1 0
1 1 1 1 1
```

Exercise 3.

Fill $m \times n$ matrix with random values from the range. The highest and lowest values of the range will be entered from the keyboard.

Exercise 4.

Fill $m \times n$ matrix with random values from the range. The highest and lowest values of the range will be entered from the keyboard.

Find the smallest element in every row and swap it with the element from the main diagonal.

Exercise 5.

Fill $m \times n$ matrix with random values from the range. The highest and lowest values of the range will be entered from the keyboard.

For every column, find and output the product of elements from the odd rows.

Exercise 6.

Fill $m \times n$ matrix with random values from the range. The highest and lowest values of the range will be entered from the keyboard.

Find and output the product of the positive elements from the odd columns.

Two-Dimensional Arrays

As we already know, an array is an ordered collection of elements of the same type, with one name and different indexes. It is true for any array that its elements lay next to each other (in memory).

First Element	Second Element	Third Element	...	Nth Element
---------------	----------------	---------------	-----	-------------

We use a single-dimensional array when we work with simple sequences of data, be it numbers, strings or other types.

It should be noted that all arrays can be seen, in essence, as one-dimensional. For convenience, two- and multi-dimensional arrays are needed in data representation and manipulation.

Two-dimensional arrays have two indexes and N-dimensional arrays have N indexes.

It is convenient to use a two-dimensional array in the process of solving problems that involve tabular data.

For example, input data should be represented and manipulated as a table with three columns and five rows.

One Column					Two Column					Three Column				
el 1	el 2	el 3	el 4	el 5	el 6	el 7	el 8	el 9	el 10	el 11	el 12	el 13	el 14	el 15

Elements of the array

Here, we have an array that consists of three columns and each of these consists of five rows. We can say that we have an array of arrays.

This kind of array will be put into a ***StringGrid***, as follows:

	Column One	Column Two	Column Three
Row One	1 element	6 element	11 element
Row Two	2 element	7 element	12 element
Row Three	3 element	8 element	13 element
Row Four	4 element	9 element	14 element
Row Five	5 element	10 element	15 element

Let us declare this array.

First, we will declare the column array.

```
Const
  m=3;
Type
  col_array=array[1..m] of row_array;
  Add the declaration of the row array.
Const
  m=3;
  n=5;
Type
  row_array=array[1..n] of integer;
  col_array=array[1..m] of row_array;
```

Let us examine another example, where the input data should be represented as an array with three rows, each with five columns.

Row One					Row Two					Row Three				
el 1	el 2	el 3	el 4	el 5	el 6	el 7	el 8	el 9	el 10	el 11	el 12	el 13	el 14	el 15

Elements of the array.

As we can see, we have an array that consists of three rows, each with

five columns. Again, we have an array of arrays.

In this case, the array will be put into ***StringGrid*** as follows:

	Column One	Column Two	Column Three	Column Four	Column Five
Row One	el 1	el 2	el 3	el 4	el 5
Row Two	el 6	el 7	el 8	el 9	el 10
Row Three	el 11	el 12	el 13	el 14	el 15

Let us define the array type.

First, let us declare the array of rows.

```
Const
    n=3;
```

```
Type
```

```
    row_array=array[1..n] of col_array;
```

Then, we add the declaration of array of columns:

```
Const
```

```
    n=3;
    m=5;
```

```
Type
```

```
    col_array =array[1..m] of integer;
    row_array =array[1..n] of col_array;
```

As you can see from these examples, two-dimensional (and N-dimensional) arrays can be represented as array of arrays. Two-dimensional arrays can be an array of rows or an array of columns.

We can address the elements of array A as follows:

A[1][4] or A[3][5] or A[I][j], where I is an index of a two-dimensional array and j is an index of a one-dimensional array (an element of a two-dimensional outer array).

Let us work out an example problem.

We have to fill a two-dimensional array on M columns of N integer elements with random numbers in the range of -10 to 10 , inclusively. We have to output this array into a **StringGrid** control element. The number of rows and columns of **StringGrid**'s fixed cells should be 0.

First, we declare the array type.

```
Const
    n=3;
    m=6;
Type
    array_1=array[1..n] of integer;
    array_2=array[1..m] of array_1;
```

Now, we can write the solution:

```
Var
    C: array_2;
Begin
    FillArray(c);
    //Fill array C
    TableOutput(c); //Output array C into text table
end;
```

Now, we can write the procedures out. We start with the procedure to fill an array of columns.

```
procedure FillArray(var arr: array_2);
    var i: integer;
begin
    Randomize;
    for i:=1 to m do //for i in range from 1 to M
        FillColumn(arr[i]); //fill i-th array column
    end;
```

Now, we need a procedure to fill out the column with random num-

bers in the range of -10 to 10.

```
procedure FillColumn(var col: array_1);
var
    j: integer;
begin
    for j:=1 to n do //For j in range from 1 to n
        ma_1[j]:=Random(21)-10; //jth element of array
        //gets assigned with random value in range
        //from -10 to +10 inclusively
    end;
```

The procedure to output the resulting array to the **StringGrid** is defined below.

```
procedure TableOutput(arr: array_2);
var
    i, j: integer;
begin
    frm2arr.sgdMy.FixedCols:=0; //Number of fixed
columns
    frm2arr.sgdMy.FixedRows:=0; //Number of fixed
rows
    frm2arr.sgdMy.ColCount:=m; //Columns
    frm2arr.sgdMy.RowCount:=n; //Rows
    for i:=1 to m do
        for j:=1 to n do
            frm2arr.sgdMy.Cells[i-1, j-1]:=IntToStr(arr[i][j]);
        end;
```

Let us see what would change if we output the array line-by-line, as if we transposed it. In this case, we would have three rows, each with six columns. We have to remember that the cells of the **StringGrid** are first indexed by a column and then, by a row.

The procedure to output an array to the **StringGrid** will be similar to the following:

```

procedure TableOutput(arr: array_2);
var
  i, j: integer;
begin
  frm2arr.SgdMy.ColCount:=n; //n - number of array
rows
  frm2arr.SgdMy.RowCount:=m; //m - number of array
columns
  for i:=1 to n do
    for j:=1 to m do
      frm2arr.SgdMy.Cells[i-1, j-1]:=IntToStr(arr[j][i]);
    end;
end;

```

Exercises

Exercise 1.

Fill with random numbers a two-dimensional array, consisting of M columns with N rows each.

The range should be entered in separate text boxes.

Output the array into a **StringGrid** table. The number of fixed cells' rows and columns should be set to 0.

The following values should be written to the **Memo** component:

Sum of all of the array elements.

Maximal and minimal elements amongst the elements of the array.

Sum of the elements of each column.

Everything should be written with a commentary.

For example:

Sum of elements = 234.

Maximal element = 68.

Minimal element = 5.

Sum of elements of one column = 94.

Sum of elements of two columns = 43.

Exercise 2.

Fill a two-dimensional array of real numbers with random values. The array should have N rows and M columns.

The range of random values should be input from separate **Edit** components.

Create a **StringGrid** component. Set the number of columns and rows in the fixed cells.

Output the array into the **StringGrid** component.

Compute the sum of elements of each array row and output into a separate cell in each row of the **StringGrid** component.

Date and Time

To work with a date and time value, there is a special data type ***TDateTime*** in ***Delphi***.

The variables are declared in the following way:

```
Var  
    a, b: TDateTime;
```

TDateTime is a double that holds the date-time value. The integral part of a ***TDateTime*** value is the number of days that have passed since December 30, 1899.

The fractional part of a ***TDateTime*** value is the time of day.

To find the fractional number of days between two dates, simply subtract the two values — unless one of the ***TDateTime*** values is negative. Similarly, to increment a date and time value by a certain fractional number of days if the ***TDateTime*** value is positive, add the fractional number to the date and time value.

When working with negative ***TDateTime*** values, computations must handle the time portion separately. The fractional part reflects the fraction of a 24-hour day, without considering the sign of the ***TDateTime*** value. For example, 6:00 am on 12.29.1899 is -1.25 , not $-1 + 0.25$, which would be -0.75 . There are no ***TDateTime*** values between -1 and 0 .

Procedures to Work with Dates

Let us consider the following procedures:

DecodeDate(Date: TDateTime; var Year, Month, Day: Word);
extracts year, month and day values from a given ***Date TDateTime*** type value.

For example, if variable A has the date March 1, 2008, then the **DecodeDate** (A, y, m, d) will return y=2008, M=3, d=1.

DecodeTime(Time: TDateTime; var Hour, Min, Sec, MSec: Word); extracts the hour, minute, second and millisecond values from a given **Time TDateTime** type value.

Functions to Work with Dates

Let us assume that we have three variables of a **TDateTime** type:

```
Var
    a, b, c: TDateTime;
Begin
    a:=Date; //Returns the current date
    b:=Time; //Returns the current time
    c:=Now; //Returns the current date and time
End;
```

These functions take the date and time from the computer settings. So, if the computer settings are incorrect, these functions return the wrong date and time.

There are functions that convert the date and time to string.

TimeToStr(t: TDateTime): String; converts a **TDateTime** value **Time** into a formatted time string. The time is formatted using the **LongTimeFormat** value, which, in turn, uses the **TimeSeparator** value.

DateToStr(t: TDateTime): String; converts a **TDateTime** value **Date** into a formatted date string.

The date is formatted using the **ShortDateFormat** value, which, in turn, uses the **DateSeparator** value, for example **'03:04:2015'**.

DateTimeToStr(t: TDateTime): String; — converts a **TDateTime** value **DateTime** into a formatted date and time string. The string comprises:

Date in **ShortDateFormat**

One blank

Time in ***LongTimeFormat***

Also, there are functions that convert a string to date, time or date and time.

```
StrToTime(S: String): TDateTime;  
StrToDate(S: String): TDateTime;  
StrToDateTime(S: String): TDateTime;
```

Here are some features of the string format:

- For time, you can skip seconds. For example '9:10' will be converted as '9:10:00' by default.
- Data should be correct.
- If the year is only specified by two digits, it will be converted as a year from 1930–2030 range.
- If the year is not specified, it will be converted as the current year.
- Day and month must be specified.

We will consider some more functions for work with dates and times.

DayOfWeek(Date: TDateTime): Word returns an index number for the day of the week: 1 = Sunday, 2 = Monday, etc.

DecodeDateFully(Date: TDateTime; var Year, Month, Day, DOW: Word): Boolean — extracts the Year, Month, Day and DOW (day of the week). Returns True for a leap year;

EncodeDate(var Year, Month, Day: Word): TDateTime generates a TDateTime return value from the passed Year, Month and Day values;

EncodeTime(Hour, Min, Sec, Msec: Word): TDateTime generates a TDateTime return value from the passed Hour, Min, Sec and MSec (millisecond) values.

Operations with Dates

- Two dates can be subtracted and thus, the whole part of a difference is the number of days between the dates.
- There is no sense in adding up the two dates but we can add or subtract the integer. In this case, we get a new date, which differs in the corresponding number of days -backwards or forwards.

Time can be summed up in the following way:

For example, there are two ways to know what the time will be in 1,5 hours:

```
Time+1,5/24
```

or

```
Time+StrToTime('1:30')
```

Exercises

Exercise 1.

A **StringGrid** is filled with the last names, dates of birth and dates of the Nobel Prize awarding. Determine the last name of the youngest laureate.

Exercise 2.

Let us assume that a school has lessons of equal duration. Breaks in this school also have equal duration, which are different from the lessons' duration. Create a school schedule using: the start time of the first lesson, lessons' duration, breaks' duration and the number of lessons. All of the quantities are entered into text boxes. The schedule should be presented in the following manner:

	Start Time	End Time
Lesson 1	9:00	9:40
Break	9:40	9:50
Lesson 2	9:50	10:30
Break	10:30	10:40
...		

Exercise 3.

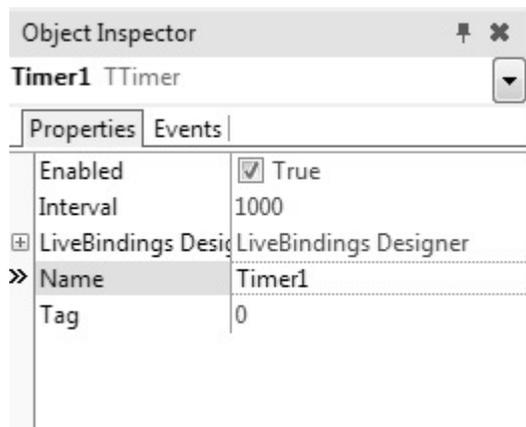
A date is entered into a text box. Compute how many whole years, months and days separate this date from the present day. Output the result into different labels.

Timer

Sometimes, it is necessary that some statements in a program execute in a certain time period. We will consider the following task: paint the form in a red color in five seconds after the program running.

To solve this task, we will use the component **Timer** from the **System** tab.

Put the **Timer** on the form at any place. It will not be seen during the run time. Let us learn these component properties in the Object Inspector.



In addition to the **Name**, we are interested in the **Enabled** and **Interval** properties. If **Enabled** is **True**, then the timer is running, otherwise, the timer is switched off. The **Interval** property specifies the timer reaction time in milliseconds. By default, the **Interval** is 1,000 or one second.

We will set the **Interval** value at 5,000 (five seconds).

We will click on the timer on the form and a timer event handler will

be created. There, we will add a statement to change the form color. Here you are:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    form1.Color:=clRed;
end;
```

After the run of the program, we will see that, after a while, the form will be painted in a red color.

Let us correct the program a bit. We will put the button on the form. We will add the following statement into the button click event handler:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    timer1.Enabled:=true;
end;
```

We will set the **Enabled** property value to **False**. Now, after the program run, nothing will happen. However, after clicking the button, the form will be painted to a red color in five seconds. So, after the button is clicked, the **Enabled** property will be set to true, the timer will be turned on and, in five seconds, the timer handler will be executed.

Let us change the timer handler in the following way:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    form1.Color:=256*256*random(256)+256*random(256)
+random(256);
end;
```

Here, we have assigned a random color.

We will change the button click event handler too:

```
procedure TForm1.Button1Click(Sender: TObject);
```

```
begin
    timer1.Enabled:=true;
    timer1.Interval:=1500;
end;
```

Here, we have set the timer interval to 1,5 seconds.

Let us run the program and click on the button. What will we see? Every 1,5 seconds, the color of the form changes randomly. So, the running timer calls the timer event handler constantly in the set interval. To stop the timer, the **Enabled** property should be set to **False**.

We will add more buttons to the form and write into its click event handler:

```
procedure TForm1.Button2Click(Sender: TObject);
begin
    timer1.Enabled:=false;
end;
```

Now, by clicking the first button, we will start the color changing. We will stop it by clicking on the second button.

If we want the timer to only be fired once, we need to set **False** to the **Enabled** property in the timer event handler. So, if we want this to happen after the 1,5 second color changes once, the timer event handler should be changed in the following way:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    form1.Color:=256*256*random(256)+256*random(256)
+random(256); timer1.Enabled:=false;
end;
```

To switch off a timer in a certain interval, you should add another timer. For example, to stop the color changing in 20 seconds, we will add the second timer (do not forget to set the **Enabled** property to **False**). We will write in the second timer event handler:

```
procedure TForm1.Timer2Timer(Sender: TObject);
begin
    timer1.Enabled:=false;
    timer2.Enabled:=false;
end;
```

We will change the first button click event handler and the first timer event handler in the following way:

```
procedure TForm1.Button1Click(Sender: TObject);
begin
    timer1.Enabled:=true;
    timer1.Interval:=1500;
    timer2.Enabled:=true;
    timer2.Interval:=20000;
end;
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    form1.Color:=256*256*random(256)+256*random(256)
+random(256);
end;
```

We will run the program. After the first button is clicked, the form color will start to change, with an interval of 1,5 seconds. The color changing will stop in 20 seconds.

Using times, it is possible to create moving objects. Let us consider one more example.

Let us put any component on the left part of the form, e.g., a text box. We will correct the first timer event handler in the following way:

```
procedure TForm1.Timer1Timer(Sender: TObject);
begin
    edit1.Left:=edit1.Left+15;
end;
```

Run the program and click on the first button. We will see that the text box will move from left to right.

So, timers allow us to measure time intervals during run time, dynamically change parameters and create moving objects.

Exercises

Exercise 1.

Using a timer, write a program to 'paint' starry sky. The stars are lit and, after a while, die away.

Exercise 2.

Write a program that creates a penalty timekeeper.

The time interval is set to ***TEdit***. After clicking the button, the timer starts to count down in seconds. As soon as the time is 0, the color of the ***TEdit*** background becomes red and the timer starts to count up the penalty time.

Exercise 3.

Write a program to create a digital clock, which shows the time, date and day of the week.

Exercise 4.

Write a program that imitates a clock with hands. The second hand has to move every second, minute — every minute, hour — and every hour (it is possible to make the hour hand move smoothly, in proportion to the past part of the hour).

Text Files

Writing Data to Text File

Any contemporary program has to perform some work on files.

A **File** is a named part of a disk or any other information medium that has some information written into it.

These files can have special purposes, like files from standard libraries, or they can be data files for or from a program. All files have their formats. The specifics of the file format depend on the data encoding in the file.

One of the most used file formats is the **text file** format.

A **text file** contains one or more lines of arbitrary length and each line can contain arbitrary characters. Each **text file** line ends with a special end-of-line character and the file ends with an end-of-file character. You can edit a **text file** with the **Notepad** editor or any other text file editor.

A **text file** can be seen as an aggregate of combined data, which have one common name. **Text file** names usually have the extension, **.txt** but this extension does not guarantee that the file is a **text file**. Also, **text file** names can have different extensions from **.txt**. Here are examples of standard extensions for **text files: .ini, .log, .inf, .dat, .bat**. We will use the **txt** extension.

Text files are sequential access files and can only be accessed in sequential order.

In sequential access files, data records are placed in the order that they were written into the file, sequentially one after another. To find a specific data record, the program has to sequentially scan the file from the beginning to the data needed. Consequently, the time needed to access the data record linearly depends on its position in the file.

Such files should be used if a program processes (almost) all of the data contained in it and the content of the file seldom changes. Thus, the main drawback of these files is that it is hard to update existing data, change data records to new ones and insert new records.

Let us see how to work with text files in *Delphi*.

A program cannot access a disk directly. It should use variables to hold data in RAM. Thus, we need a variable to hold a reference to a file. Such a variable is called a **file variable** and can be defined as follows:

```
Var  
    f: TextFile;
```

Before using a file, the program has to associate the **file variable** and the file to be used, whether it exists or needs to be created. To do this, the program has to use the **AssignFile** procedure, which has two parameters: **file variable** and string with a local or complete path to the file.

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);  
var  
    f: TextFile;  
begin  
    AssignFile(f, 'my.txt');  
end;
```

In this example, we associated the **file variable f** and the file named **my.txt**, which is (or will be) located in the current directory (directory from which the program has run). The file name, given as a second parameter, is a local file name. We should use the full path of the file if it is not located in the current directory.

Attention: associating the **file variable** with the file name does not check whether the file exists or not. It does not open an existing file, nor does it create a new one.

A **text file** can be either read or written. It is impossible to simultaneously read from and write to a **text file**.

To perform read or write operations on the file, we should open it. We can open a file to be read or to be written.

First, we look into the process of writing the file.

To open a file for writing we have to use the operator, **Rewrite(f)**, where **f** is a **file variable**.

This operation creates an empty file with the name associated with the **AssignFile**. The content of the file will be erased if such a file already exists.

Now we can write information into the file, using the operators, **write** and **writeln**. They work like their equivalents to print on console. The only difference is that we have to specify the **file variable** at the beginning of the arguments, indicating that output goes to the file and not to the console.

Below is an example procedure that writes various information into a **text file "example.txt"**.

```
Procedure ZapFile;
var
  f: TextFile;
  x, y: integer;
  Ok: boolean;
begin
  AssignFile(f, 'example.txt');
  Rewrite(f);
  x:=10;
  y:=7;
  Writeln(f, x); //Writes 10 to first line of file,
//cursor goes to second line.
  Write(f, x+2); //Writes 12 (10+2) to second
line, cursor remains at the end of second line
  Write(f, 'Hello'); //At the end of second line
(at cursor position) will be written a word "Hello",
```

```
cursor remains at the end of second line
    Write(f, x, y); //Writes 10 then 7 without space
at cursor position, cursor remains at the end of
second line
    Writeln(f, x, y); //Writes 10 then 7 without
space at cursor position, cursor moves to third line
    Ok:=5>7;
    Writeln (f, Ok); //Writes value False (value of
variable OK) to the third line, cursor move to fourth
line
    Writeln(f, x, ' ', y); //Writes 10, then two
spaces, then 7 into fourth line, cursor goes to fifth
line
    Writeln(f, 'x=', x); //Writes "x=10" to fifth line
of file.
    CloseFile(f);
end;
```

You can see that we closed the file with the **CloseFile(f)** statement. The **CloseFile** procedure **should be called after you have finished working with the file**. It closes the file and finishes writing. Without that call, some information can be lost.

As we said above, the call to **Rewrite** a procedure always creates a new file. However, there are cases when it is necessary to add information to an existing file. In such cases, the file can be opened with the **Append** procedure. This procedure does not create a new file but opens an existing one and puts the cursor at the end of the file. Consequently, all of the **write** and **writeln** statements have the information from the cursor position on them.

You can only use **Append(f)** on an existing file. Otherwise, the run time system will display an error and the program will terminate. After finishing work with a file that was opened with **Append(f)**, we have to close it with a **CloseFile(f)** call.

The file argument in procedures should always be declared as a **variable argument**, using the keyword, **var**.

Below is an example of a program that creates two files, "test1.txt" and "test2.txt", and writes several lines with zeroes to them. The first file should be three lines with five zeroes and the second file should contain four lines with three zeroes. The zeroes are separated by two spaces.

```
procedure ZapZero(var fl: TextFile; n, m: integer);
var
    i, j: integer;
begin
    Rewrite(fl);
    for i:=1 to n do
    begin
        for j:=1 to m do
            write(fl,0:3);
            writeln(fl)
        end;
    CloseFile(fl)
    end;
procedure TForm1.Button1Click(Sender: TObject);
var
    f, g: TextFile;
begin
    AssignFile(f, 'test1.txt');
    AssignFile(g, 'test2.txt');
    ZapZero(f,3,5);
    ZapZero(g,4,3);
end;
```

Reading Data from a Text File

Let us look into the process of reading data from a file. We open the file for reading with the call to **Reset(f)**.

Attention: the file that is about to be opened for reading should exist on a disk. Otherwise, a run time error will display an error message and terminate the program execution.

If the file is successfully opened, then the cursor is set to the beginning of the first line. We can read that line by calling the procedure, **ReadLn(f, s)**.

After the **ReadLn** statement is complete, the **s** variable will have the content of the first line and the cursor will be set at the beginning of the next line. If we repeat the statement, then the variable will be assigned to the content of the second line, the cursor will be set to the beginning of the third line and so on. After we have finished working with the file, we have to close it, using the **CloseFile(f)** procedure call, just like when writing files.

Below is an example of a procedure that will write the first line of a file **'my.txt'** into **memEx1 Memo** component.

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
    f: TextFile;
    s: string;
begin
    AssignFile(f, 'my.txt'); //Associates file name
and file variable
    Reset(f); //Opens file for reading
    ReadLn(f, s); //Read first line of file
    memEx1.Lines.Append(s); //Puts that line into
Memo
    CloseFile(f); //Closes file
end;
```

This procedure will always only put the first line from the file. We can use the loops to read several lines from the file. The example below will read and put five lines into the **TMemo** from the file '**my.txt**'.

```

procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
    f: TextFile;
    s: string;
    i: integer;
begin
    AssignFile(f, 'my.txt'); //Associates file name
and File variable
    Reset(f); //Opens file for reading
    For i:=1 to 5 do //Iterates 5 times
    begin
        ReadLn(f, s); //Reads a line from file
        memEx1.Lines.Append(s); //Puts line into
Memo
    end;
    CloseFile(f); //Close file
end;

```

As we have already noted, **text files** are sequential access files, i.e., it is not possible to access a specific line without reading the preceding lines. For example, we need to read four lines if we want to access the fifth line of the file. We can use the **ReadLn(f)** statement without specifying any variables.

```

procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
    f: TextFile;
    s: string;
    i: integer;
begin

```

```
    AssignFile(f,'my.txt'); //Associates file and file
variable
    Reset(f); //Opens file for reading
    For i:=1 to 4 do //iterates 4 times:
        ReadLn(f); // skips the line
        ReadLn(f, s); //Reads fifth line from file.
        memEx1.Lines.Append(s); //Puts line into
Memo component
    CloseFile(f); //Closes file
end;
```

In the example above, **ReadLn(f)** just moves the cursor to the beginning of the next line, without reading any data.

The number of lines in a file is not always known a priori. So, is it possible to read all of the lines from a file without knowing their count? We can use the **EOF(f)** Boolean function (**EOF** stands for **End Of File**). This function returns **True** if the cursor is at the end of the file, otherwise it is **False**. Here is an example of a procedure that reads all of the lines of a **text file** and puts them into a **Memo** component:

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
    f: TextFile;
    s: string;
begin
    AssignFile(f,'my.txt'); //Associates file name
and File variable
    Reset(f); //Opens file for reading
    While not EOF(f) do //Iterates while to end-of-
file
    begin
        ReadLn(f, s); //Reads the line from file
        memEx1.Lines.Append(s); //Puts line into TMemo
```

```
end;  
CloseFile(f); //Closes file  
end;
```

Please note that, in all of our examples, we used a **ReadLn** statement, not **Read**. The last example will not finish its work if we change **ReadLn** to **Read**. This is because **Read** does not move the cursor beyond the end of the current line and will read empty lines when the cursor is at the end of the line. This will result in a never-ending loop, which will read an infinite amount of empty lines and put them into **Memo**. This is because the cursor will never get to the end of the file. Thus, if we read lines from the file, we need to use a **ReadLn** statement.

It is also possible to know whether the cursor points to the end of the line. We can use the **EOLN(f)** Boolean function for this. This function returns **True** when the cursor is at the end of the line and returns **False** if it is not.

Now we know how to work with a **text file** when we read data from a file line by line. We can also read real or integer values from a file, which are separated by one or several spaces. In this case, we can read variables directly. Let us look at an example of how to do this.

Let us assume that we have a **text file** that contains numbers, which are separated by spaces (any non-zero number of spaces). An example is given in the figure on the right.

Right after opening the file, let us use **Read(f, x)** statement, where **f** is **a file variable** and **x** is an integer variable. After executing this statement, the **x** variable will be set to 10 and the cursor will be positioned at the delimiter (space) right after 0. After executing **Read(f, x)** for a second time, the cursor will skip all of the delimiters (spaces). Then, value 7 will be read into the **x** variable and the cursor will stop at the delimiter (space) right after 7. Executing **Read(f, x)** for a third time will skip all of the delimiters (spaces) before number 25. The number 25 will be read into the **x** variable and the cursor will stop after 25. Generally speaking, the **Read(f, x)** for an integer variable **x** statement skips all of the delimiters before the number, reads the number into the variable and puts the cursor to the non-numeric symbol after the number.

Here is an example:

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
    f: TextFile;
    s: string;
    x, i, k: integer;
begin
    AssignFile(f, 'my.txt'); //Associates file name
and File variable
    Reset(f); //Opens file for reading
    k:=0;
    For i:=1 to 5 do //Executes five times:
        begin
            Read(f, x); // Reads another number and
            k:=k+x; // Adds it to sum
        end;
    CloseFile(f); //Closes file
end;
```

After executing this procedure, the ***k*** variable will contain **57** — the sum of the first five numbers **10, 7, 25, 14** and **1**. Please note that ***Read(f, x)*** read the numeric data into an integer variable, skipping all of the delimiter symbols. The delimiters for ***Read*** are the spaces and end-of-line symbols.

In the example above, what will happen if we change the ***Read(f, x)*** statement into a ***ReadLn(f, x)*** statement? The main difference between ***Read*** and ***ReadLn*** is that ***ReadLn*** will skip all of the data on the line until the cursor reaches the next line.

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
    f: TextFile;
    s: string;
```

```
x, i, k: integer;
begin
  AssignFile(f,'my.txt'); //Associates file name
and File variable
  Reset(f); //Opens file for reading
  k:=0;
  For i:=1 to 5 do //executes five times:
    begin
      ReadLn(f, x); // Reads another number and
      k:=k+x; // Adds it to sum
    end;
  CloseFile(f); //Closes file
end;
```

After executing the procedure above, the **k** variable will contain the number, **86** — the sum of the numbers in the beginning of the five lines — **10, 14, 24, 27 u 11**.

A problem that is often encountered is to read all of the numbers from the line of the file. Furthermore, the amount of numbers is not known in advance, which can be problematic. It seems logical to use the **EOLN(f)** function to check the end-of-line situation and mostly repeat one of the above examples. In this case, the procedure will look like the following:

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
  f: TextFile;
  s: string;
  x, k: integer;
begin
  AssignFile(f,'my.txt'); //Associates file name
and File variable
  Reset(f); //Opens file for reading
  k:=0;
```

```
While not EOLN(f) do //Loops while not end-of-  
line  
begin  
    Read(f, x); // Reads another number  
    k:=k+x; // Adds it to sum  
end;  
CloseFile(f); //Closes file  
end;
```

After the execution of this example, the **k** variable should contain the number **42** — a sum of the numbers from the first line **10, 7, 25**. However, this will not be the case. The answer will vary as to whether or not the line that is read contains spaces after the last number. If there are no spaces, the answer will be expected to be one.

Let us look at what would be happening if there were spaces after the number **25** in our example file. In this case, after reading **25**, the cursor will be put at the space right after the digit five. The call to **EOLN(f)** will return **False** because the space is not an end-of-line symbol. The loop will execute once more and **Read(f, x)** will skip all of the delimiters, including the end-of-line symbol, until the next number is read. As a result, the **k** variable will contain a sum of numbers from several lines. The number of lines and the amount of numbers that are read depend on the number of consecutive lines that contain spaces before the end-of-line.

To avoid this problem, it is necessary to use the function, **SeekEOLN(f)**, instead of **EOLN(f)**, **when reading numbers**. This function first skips all of the spaces and, only then, does it check for the end-of-line symbol. In the example above, if we change the call to **EOLN(f)** to **SeekEOLN(f)**, our example will work correctly, regardless of the number of spaces before the end-of-line symbol.

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);  
var  
    f: TextFile;  
    s: string;  
    x, k: integer;
```

```
begin
    AssignFile(f,'my.txt'); //Associates file name
and File variable
    Reset(f); //Opens file for reading
    k:=0;
    While not SeekEOLN(f) do //Loops while not end-
of-line
        begin
            Read(f, x); // Read next number
            k:=k+x; // Adds it to sum
        end;
    CloseFile(f); //Closes file
end;
```

Delphi also has the function, **SeekEOF(f)**, which works just like **SeekEOLN(f)** but skips the spaces and end-of-line symbols. You should use it when reading numbers from a file, using **Read(f, x)** statement.

A procedure to compute the sum of all of the numbers in a file can look like this:

```
procedure TfrmEx1.btnEx1Click(Sender: TObject);
var
    f: TextFile;
    s: string;
    x, k: integer;
begin
    AssignFile(f,'my.txt'); //Associates file name
and file variable
    Reset(f); //Opens file for reading
    k:=0;
    While not SeekEOF(f) do //loops until not end-
of-file
```

```
begin
    Read(f, x); // Reads next number
    k:=k+x; // Adds it to sum
end;
CloseFile(f); //Closes file
end;
```

Use the following rule to guide yourself when you need your program to read data from a **text file**: if you read data as **lines**, use **EOLN(f)** and **EOF(f)** functions and if you read data as **numbers**, use **SeekEOLN(f)** and **SeekEOF(f)** functions.

When a program attempts to read a number from non-numeric data (**EOF** or just text), the run time system will display an error message and terminate the program execution. **Abort** will also be executed when a program attempts to read a real number into an integer variable or number format in a file that does not conform to **Delphi's** format (a comma separator for the fractional part, instead of a period).

Exercises

Exercise 1.

Write a program that finds the shortest line in a file and writes its content into a **Label** component. If there are several equally short lines, the program should write the last line found. The name of the file should be read from the text box.

To test the program behavior, use a **text file** that has been created using an external text editor (**Notepad** being one).

Exercise 2.

Write a program that reads lines from a file and outputs the palindrome lines to **TMemo**. The file name should be entered into the text box (a dialog box can be used).

To test the program behavior, use a **text file** that has been created using an external text editor (**Notepad** being one).

Exercise 3.

A **text file** contains several lines. Each line contains several integer numbers, which are separated by one or more spaces. Write a program that will read the file and write to another file the lines that only have an even sum of numbers. The file names are entered into text boxes (you can use dialog boxes).

To test the program behavior, use a **text file** that has been created using an external text editor (**Notepad** being one)

Exercise 4.

A **text file** contains information about children and schools that have taken part in the Olympiad. The first line contains the number of children records. Each of the subsequent lines has the following format:

```
<surname> <initials> <school number>
```

where <surname> is a string containing no more than 20 symbols, <initials> is a string containing four symbols (character, period, character, period), <school number> is one or two digit numbers. <surname> and <initials> and <initials> and <school number> are separated by a single space. An example of an input string is below:

```
Smith J.C. 57
```

Write a program that will output to a label the school number with the smallest number of participants. The school numbers should be listed and if there are several schools with the same smallest number of participants, they should be separated by a comma. The number of lines is greater than 1,000.

Standard File Dialogs

In the previous module, we reviewed the typical and standard way to work with files. However, **Delphi** has an option to work with files with **File Dialog** windows, i.e., select files for reading or writing through the standard **File Open** and **File Save Dialog** windows.

Components of the standard file dialog windows are located on the **Dialogs** tab in the palette. When placed on the form, these components are not visible during the program execution.

We will use the **OpenDialog** component (open file icon) to read the information from the file (a file must exist) and **SaveDialog** (disk icon) to write to the existing file or a new file. The name of the new file will be entered from the keyboard.

The components have the property **FileName** and the Boolean method **Execute**.

The following is an algorithm of working with file dialog windows:

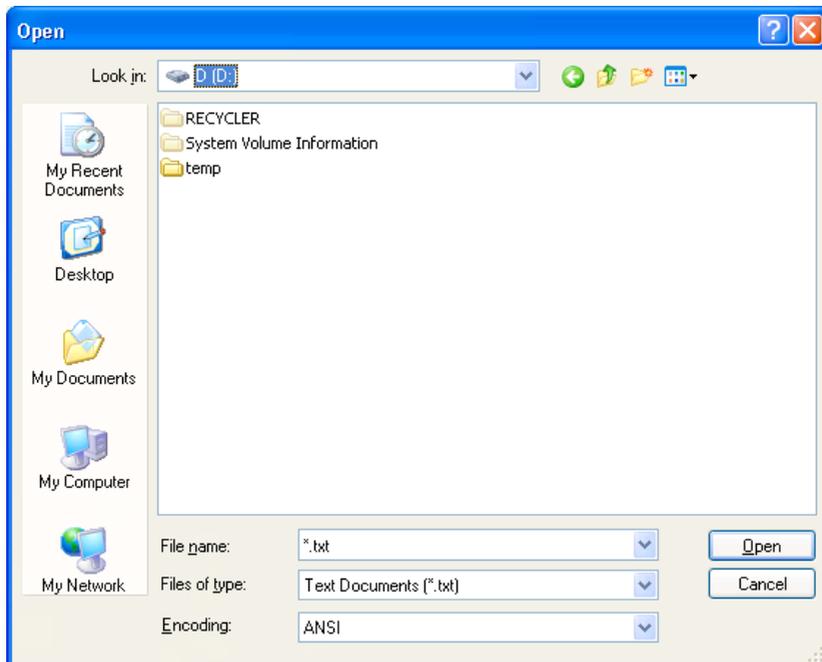
```
Var
    Okf1, Okf2: Boolean;
    StF1, StF2: string;
Begin
    //File for reading selected?
    //If the File for reading selected then
    Begin
        //Find File Name from Dialog
        //Assign File variable to the File
        //Open the File for Read
        <process File information: read, process, etc>
```

```
//Close the File
End;
//File for writing selected?
//If the File for writing selected then
Begin
//Find File name from Dialog
//Assign File variable to the File
//Open the File for Write
<write information to File >
//Close the File
End;
end;
```

The first line of the program will be:

```
Okf1:= OpenDialogEx1.Execute;
```

When we run the program, the following standard file **OpenDialog** will be displayed. We will select the file we need in this dialog window.



Pressing the **Open** button will open the file for reading. The execute method will return **True** and the name of the file will be assigned to the **FileName** property of the **OpenDialogEx1** component.

Let us continue the program:

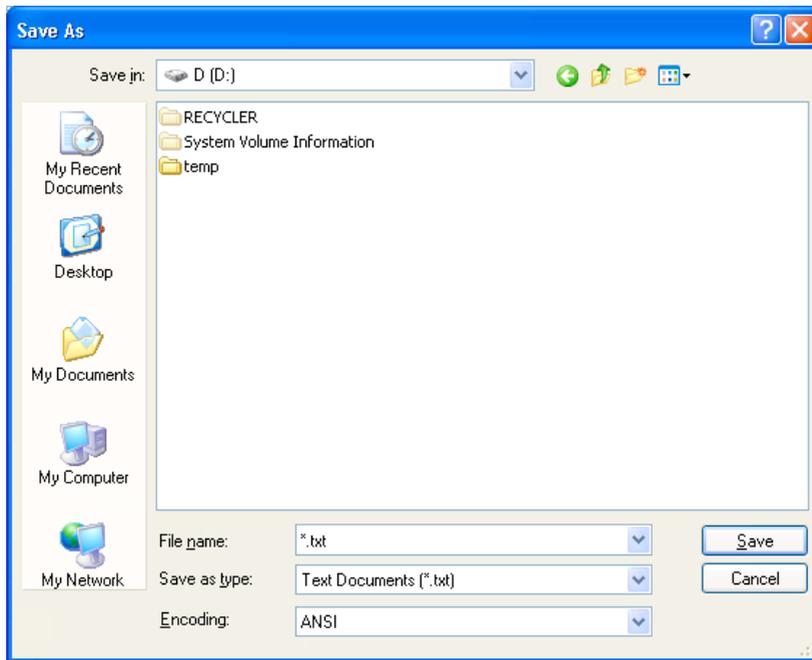
```
If Okf1 then
//if the File for Read is selected then
Begin
Stf1:= OpenDialogEx2.FileName;
//Grab the file name from Dialog and assign to Stf1
variable
AssignFile(f, Stf1); //Assign File Variable to the
File
Reset(f); //Open File for Read
.....; //process the data in the File
Closefile(f); // Close the File
End;
```

In this example, after the file is opened, we can replace the "....." with operators that are necessary for processing the data in order to solve the problems, read information from the file, process the data, etc. We have already tried this in previous modules.

To write the information into the file:

```
Okf2:=SaveDialogEx2.Execute;
//Is File selected for Write?
```

When this command is executed, the standard **File Save Dialog** is displayed. There, we can select an existing file or type a new file name in the "File Name" textbox. The name of the selected file (and the path to it) will be assigned to **SaveDialogEx1.FileName** property.



The operation will be cancelled if you press the **Cancel** button.

The code for recording information into the file will look like this:

```
Okf2:=SaveDialogEx2.Execute;
//Is File selected for Write?
If Okf2 then
//If File for Write is selected, then
Begin
Stf2:= SaveDialogEx2.FileName;
//Select File Name from Dialog
AssignFile(g, Stf2);
//Assign File Variable to the File
Rewrite(g); //Open File for Write
..... .; //write data to the file, process data
Closefile(g); //Close the File
End;
End;
```

By now you should know how to work with files in **Dialog** mode.

Exercises

Exercise 1.

Create a **text file** in **Notepad**. Using **Standard File Dialogs**, write a program that will copy all of the strings of even length from that file to File2, and all of the strings of odd lengths to File 3.

Exercise 2.

Create a **text file** in **Notepad** so that each line of the file contains integers that are separated from each other by one or more spaces. Using **Standard File Dialogs**, write a program that will calculate the sum of all numbers and record that sum in a new file.

Exercise 3.

Create a **text file** in **Notepad** so that each line of the file contains random integers and random words that are separated from each other by one or more spaces. Using **Standard File Dialogs**, write a program that will calculate the sum of all numbers and record that sum in a new file.

