

Practical Book

# INFORMATION TECHNOLOGY

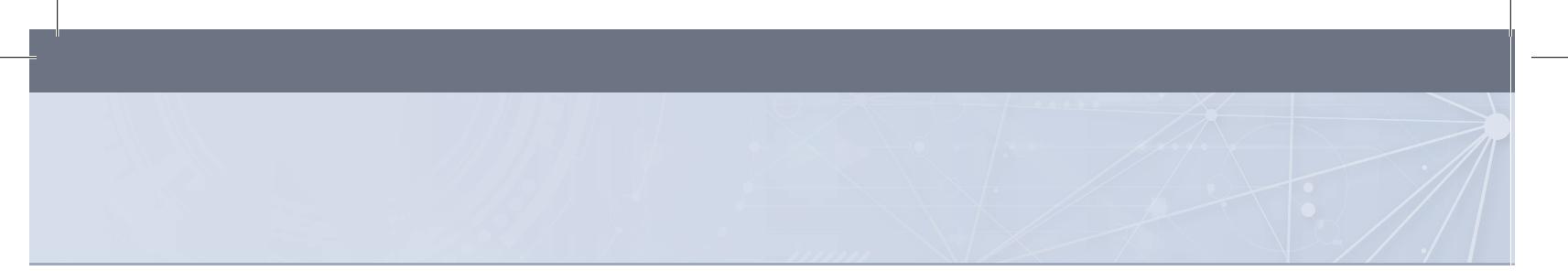
IT



basic education

Department:  
Basic Education  
REPUBLIC OF SOUTH AFRICA





**MTN South Africa, through MTN SA Foundation, is a proud supporter of the CAT and IT digital books.**

As an organisation rooted in technology, we believe in providing a new bold digital world to communities we operate in. This unique digital book provides the fundamental knowledge necessary for a sound grounding from which to make practical use of the complete and indispensable application-oriented information regarding Computer Applications Technology (CAT) and Information Technology (IT). It is a foundational reference for today's secondary school learners and teachers alike - as well as for the next generation of CAT and IT students.

**Information Technology Practical Book Grade 11**

**ISBN 978-1-928388-53-1**

First published in 2019 © 2019. Copyright in the text remains with the contributors.

**Quality Assurance team for Information Technology**

Allison Philander, Carina Labuscagne, David Peens, Denise van Wyk, Edward Gentle, Jugdeshchand Sewnanen, Julian Carstens, Magdalena Brits, Shamiel Dramat, Shani Nunkumar and Zainab Karriem

**Restrictions**

You may not make copies of this book in part or in full – in printed or electronic or audio or video form – for a profit seeking purpose.

**Rights of other copyright holders**

All reasonable efforts have been made to ensure that materials included are not already copyrighted to other entities, or in a small number of cases, to seek permission from and acknowledge copyright holders. In some cases, this may not have been possible. The publishers welcome the opportunity to redress this with any unacknowledged copyright holders.

# Contents

<b>Term 1</b>	
<b>Chapter 1</b>	Grade 10 Revision and Mathematical Functions
Introduction	1
<b>Unit 1.1</b>	Errors, debugging and mathematical methods
<b>Unit 1.2</b>	Mathematical methods
Consolidation activity	13
<b>Chapter 2</b>	Nested loops
Introduction	19
<b>Unit 2.1</b>	Nested Loops
<b>Unit 2.2</b>	Using nested loops
<b>Unit 2.3</b>	Creating shapes using nested loops
Consolidation activity	37
<b>Term 2</b>	
<b>Chapter 3</b>	Arrays
Introduction	41
<b>Unit 3.1</b>	Arrays
<b>Unit 3.2</b>	Searching and sorting arrays
<b>Unit 3.3</b>	Parallel arrays
Consolidation activity	67
<b>Chapter 4</b>	String and date manipulation
Introduction	69
<b>Unit 4.1</b>	Built-in string methods
<b>Unit 4.2</b>	Delimited strings
<b>Unit 4.3</b>	Built-in Date-Time methods
Consolidation activity	97
<b>Term 3</b>	
<b>Chapter 5</b>	Text files
Introduction	101
<b>Unit 5.1</b>	Introduction to text files
<b>Unit 5.2</b>	Reading from a text file
<b>Unit 5.3</b>	Writing to a text file
<b>Unit 5.4</b>	Creating reports
Consolidation activity	124
<b>Chapter 6</b>	User-defined methods
Introduction	129
<b>Unit 6.1</b>	Introduction to user-defined methods
<b>Unit 6.2</b>	Procedures
<b>Unit 6.3</b>	Functions
<b>Unit 6.4</b>	Basic input validation techniques
Consolidation activity	152
<b>Term 4</b>	
<b>Chapter 7</b>	User interfaces
Introduction	155
<b>Unit 7.1</b>	Multi-form user interfaces
<b>Unit 7.2</b>	Dynamic Instantiation of objects
<b>Chapter 8</b>	Databases
Introduction	167
<b>Unit 8.1</b>	Creating a database
<b>Unit 8.2</b>	Connecting to a database
<b>Unit 8.3</b>	Reading data from a database
<b>Unit 8.4</b>	Writing data to a database
<b>Unit 8.5</b>	Manipulating data
Consolidation activity	198
	<b>Annexure A – Grade 10 Revision</b> 201
	<b>Annexure B – Naming convention of components</b> 238
	<b>Annexure C – Programming and visible characters</b> 240
	<b>Glossary</b> 243
	<b>QR Code list</b> 245

# Dear Learner

Welcome to the *IT Practical Grade 11* textbook, and welcome to programming.

If this is your first time learning how to program, don't worry. This textbook has been designed to teach anyone – regardless of experience – how to program. If you follow along with all the examples then you will be an experienced programmer who has written more than 50 programs by the end of this book.

Programming and programming languages, much like real languages, can only be learned through practice. You cannot sit at home and learn to speak French from a textbook. In the same way, you cannot read this book and hope to be a programmer at the end of it. Instead, you will need to write every bit of code and create every program shown in this book. Even if all you do is follow the steps of the examples on your own computer, you will learn how to write code. Once you have mastered the code, you will be able to comfortably use it in your own programs.

For you to master programming, try to work through as many of the programs given to you. Each program has been designed to both teach you new concepts and reinforce existing concepts. The book will start by teaching you how to create simple programs. However, by the end of the book you will be creating useful programs and fun games to play.

Programming is not only about knowing and using the programming language. There are also important theoretical concepts that you will need to understand, and planning and problem-solving tools that you will need to master. The best-coded program in the world will not be useful if it solves the wrong problem. This book has therefore been divided into the following chapters:

- Chapter 1: Errors, debugging and mathematical methods
- Chapter 2: Nested loops
- Chapter 3: Arrays
- Chapter 4: String and date manipulation
- Chapter 5: Text files
- Chapter 6: User-defined methods
- Chapter 7: User interfaces
- Chapter 8: Databases

Before getting started with algorithms, watch the video in the QR code.



To give you the most opportunities to learn, this book will give three types of programming activities:

### Examples

Examples will guide you through the creation of a program from start to finish. All you need to do with examples is to follow the step-by-step guidance provided to you.

#### Example 2.1 Create the shape below using the special character '\*'\*

We know that there are six rows and each row has 6 '\*'s. The code to draw this shape:

```
*****  
*****  
*****  
*****  
*****  
*****  
  
Line 1:  for i := 1 to 6 do  
Line 2:    begin  
Line 3:      sLine := '';  
Line 4:      for j := 1 to 6 do  
Line 5:          sLine := sLine +'*';  
Line 6:      memDisplay.Lines.Add(sLine);  
Line 7:    end;
```

### Guided activities

Guided activities have a program that you need to create on your own. Your teacher will provide you with the solution. These solutions should be used as an opportunity to compare your program, and to see where you may have made errors or left something out.

#### Guided activity 2.1

You need to develop a multiplication table for a primary school learner as shown below:

$$\begin{array}{ll} 1 \times 1 = 1 & 2 \times 1 = 2 \\ 1 \times 2 = 2 & 2 \times 2 = 4 \\ 1 \times 3 = 3 & 2 \times 3 = 6 \end{array}$$

You are required to create the 1 times and 2 times multiplication table. In the 1 times multiplication table, you only need to find the product of 1 multiplied by a multiplier from 1 to 3. This is also true for the 2 times multiplication table.

Let's create an algorithm and flowchart for the problem.

#### ALGORITHM

```
for I = 1 to 2  
begin  
  for J = 1 to 3  
  begin
```

#### FLOWCHART



### Activities

Activities are programs that your teacher can give to you as classroom activities or homework. With these programs, you will only be assessed on how well your program works, so use your creativity to come up with a solution!



#### Activity 1.1

- 1.7.1 What is the difference between syntax, runtime and logic errors?
- 1.7.2 Give two examples of each of the following errors:
  - a. Syntax errors
  - b. Runtime errors
  - c. Logic errors

### 'Take note' and 'Did you know' boxes

The boxes provide extra, interesting content that might caution you to 'take note' of something important; or give you additional information. Note that the content in the 'Did you know' boxes will not be part of your exams.



#### Take note

If you cannot see all the lines that you have generated in the Memo component, then you need to set the ScrollBars property to ssVertical.



#### Did you know

In general, it is better to prevent a user from making a mistake than to inform the user that they have made a mistake afterwards.



### New words

**obfuscated** – the deliberate act of creating source or machine code that is difficult for humans to understand

### New words

These are difficult words that you may not have encountered before. A brief explanation for these words are given.



### WHAT'S AN ALGORITHM?



<https://www.youtube.com/watch?v=6hf0vs8pY1k>

### QR Codes, Videos and Screen captures

These will link you to online content. When you are in the eBook, you can easily access the links.

### Consolidation activities

This is a revision activity based on what you have covered in the chapter. Take time to answer the questions on your own. Your teacher may also use these to assess your performance during class.

#### CONSOLIDATION ACTIVITY

#### Chapter 1: Grade 10 Revision and Mathematical Functions

#### QUESTION 1 GENERAL PROGRAMMING SKILLS

##### INSTRUCTIONS

Open the incomplete program in the Question1\_1 folder found in the Question 1 folder.

Add your name and surname as a comment in the first line of the Question1\_1\_u.pas file.

Compile and execute the program. The program currently has no functionality.

# ERRORS, DEBUGGING AND MATHEMATICAL METHODS

## CHAPTER UNITS



Unit 1.1: Errors, debugging and validation

Unit 1.2: Mathematical methods



## Learning outcomes

At the end of this chapter you should be able to:

- consolidate your knowledge of the work done in Grade 10
- use the different mathematical methods: Random(), RandomRange(), Round(), Trunc(), Frac(), Ceil(), Floor(), Sqr(), Sqrt(), Inc(), Dec(), Pi and Power()
- use the mathematical methods to solve programs.

## INTRODUCTION

This chapter serves to consolidate all the knowledge you acquired in your Grade 10 studies (also see Annexure A for more Grade 10 revision). It also focuses on new content that includes work on mathematical methods. You will use these mathematical methods to solve problems during this year.

## 1.1 Errors, debugging and validation



**WHAT'S AN ALGORITHM?**



<https://www.youtube.com/watch?v=6hf0vs8pY1k>

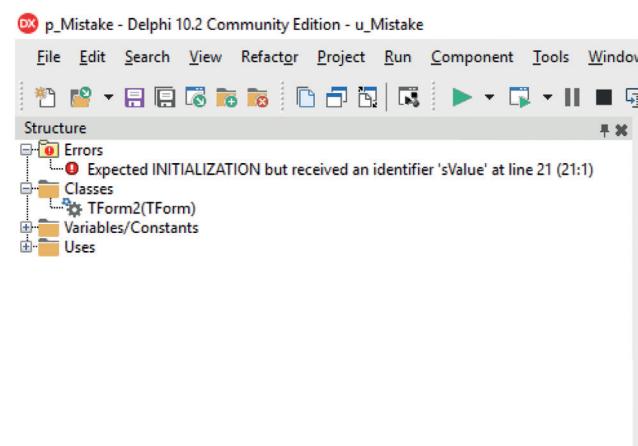
Programming errors can generally be grouped into three categories: syntax errors, runtime errors and logic errors.

Syntax refers to the rules of the programming language. For example, in Delphi, this includes the rules that all statements have to end with a semicolon and that the colon-equals operator (:=) is used to assign values. Syntax errors occur when you break the rules of the programming language.

Common syntax errors include:

- Leaving out a semicolon at the end of a statement.
- Adding a semicolon at the end of a line which is not the end of a statement (for example, in an IF-THEN-ELSE statement).
- Leaving out the command “var” when declaring variables.
- Assigning a variable using the equals sign.
- Making a typing mistake in the name of a variable.
- Not surrounding strings with the single quotation marks.
- Supplying variables to a function in the incorrect order or supplying an incorrect number of variables to a function.

With Delphi, RAD Studio will provide you with information on the error in the Structure panel at the top left Code screen, as well as in the compiler when you try to run the program. Double clicking on the mistake in the Structure panel will jump to the line with the problem.



**Figure 1.1:** The Structure panel will inform you of most syntax mistakes

Runtime errors occur when you ask your program to do a task that is either impossible or is impossible under certain circumstances. In most instances, runtime errors will cause your program to crash. Runtime errors can occur in almost any program, but some examples include:

- Doing mathematical calculations on strings.
- Performing illegal mathematical operations (such as dividing numbers by 0)
- Reading and using values from empty textboxes.
- Reading a list value that has not been created.
- Activating a ListBox's OnClick event when nothing is selected.
- Combining strings and numbers without changing their data types.

One way to solve a runtime error is to **step through** your program. You can also search for the error code online to see if it helps you to identify the cause of the error.

A screenshot of a Google search results page. The search query is "delphi i/o error 102". The results show a link to a Stack Overflow question titled "delphi - Delphi7, I/O error 102, When Assigning a file - Stack ...". Below the link, there are three more links: "How do I solve I/O error 103 in delphi" (22 May 2017), "delphi - Why is my code causing a I/O 104 error?" (22 May 2017), and "file - Delphi I/O error 103" (17 Oct 2016). The page indicates about 1,680,000 results found in 0,47 seconds.

**Figure 1.2:** Error 102 is caused by a file assignment problem

The final, and most difficult errors to solve are logic errors. These errors occur when there is a logical error or design problem in your program. While logic errors can cause applications to crash or give error messages, the program can also work without issue but give incorrect answers.

One way to identify a logic error is to use RAD Studio to track the value of the variable with the incorrect result. By seeing how the value changes with each step of the application, you can usually identify the point at which the mistake occurs. You will learn more about this technique in a later chapter.

## VALIDATING DATA

Data validation is a technique used by programmers to check (or **validate**) the information that users enter before processing it. This allows programmers to prevent common errors from occurring by making sure that the information entered is correct before it is used. The goal of input validation is to prevent users from accidentally or purposefully entering incorrect data into your program.

If your program automatically generates the data it will use, you can test the data before using it. You can also improve the algorithm generating the data to ensure that only the correct types of data are generated for your program.

In contrast, when a user is asked to supply data for your program, many unpredictable things can happen. The user may have:

- misunderstood what data is expected
- clicked the next button without entering data
- entered the correct data in an incorrect format
- entered the correct type of data, but an incorrect value.



### New words

**step through** – to step through means that you are working through a program line by line  
**validate** – to try and lessen the number of errors during the process of data input in programming

This incorrect data can cause your application to crash, or even worse, provide incorrect output.



### Take note

']' refers to a null/empty string.



### Did you know

In general, it is better to prevent a user from making a mistake than to inform the user that they have made a mistake afterwards.

## DIFFERENT TYPES OF INPUT VALIDATION

You can use different types of input validation in your program, including:

- **Required input validation** prevents the processing until certain required inputs are given. When you must read a value and perform a calculation from an *Edit* component, you may want to test whether the *Edit* component has a value before proceeding with calculations.

```
...
if edtAmount.text = '' then
    ShowMessage('Enter a value')
Else
    rAmount := StrToFloat(edtAmount.text);
...
```

- **Type validation** ensures that the data entered is the correct data type. In your calculator application, you could prevent the user from entering any values that are not numbers. Alternatively, you could inform the user that an invalid input was entered if he or she tries to do a calculation with letters and request them to enter the correct data.
- **Length validation** ensures that the data entered is the correct length.

### Note

The Length function determines the length of a string.

```
...
sPassword := edtPassword.text;
if length (sPassword) >= 8 then
    ...
Else
    rAmount := StrToFloat(edtAmount.text);
...
```

- Range validation is used to ensure that number or date falls within a specific range. For example, in a form asking a user's age, you might use range validation to ensure the user's age is between 0 and 120.

```
iAge := StrToInt(edtAge.text);
if (iAge > 0) AND (iAge < 120) then
    ...
Else
    ShowMessage('Enter an age in the range 1 to 119');
...
```

- Pattern matching validation ensures that the data entered matches a specific pattern. For example, all email addresses would match the pattern that they contain a bunch of letters or numbers, followed by an "@" sign, followed by more letters or numbers, followed by one or two groups of a full stop with letters.

## IMPLEMENTATION OF INPUT VALIDATION

Input validation can be implemented in several different ways:

- One way to implement input validation is to inform the user of the problem before they try to process the data. This could be in the form of an error message or a disabled button with an error message.
- A second way of implementing input validation is to check the data before it is processed. With this implementation, you build certain checks or conditional statements into your program to ensure that you do not process incorrect data. When these statements identify incorrect data, you send a message informing the user of the problem.

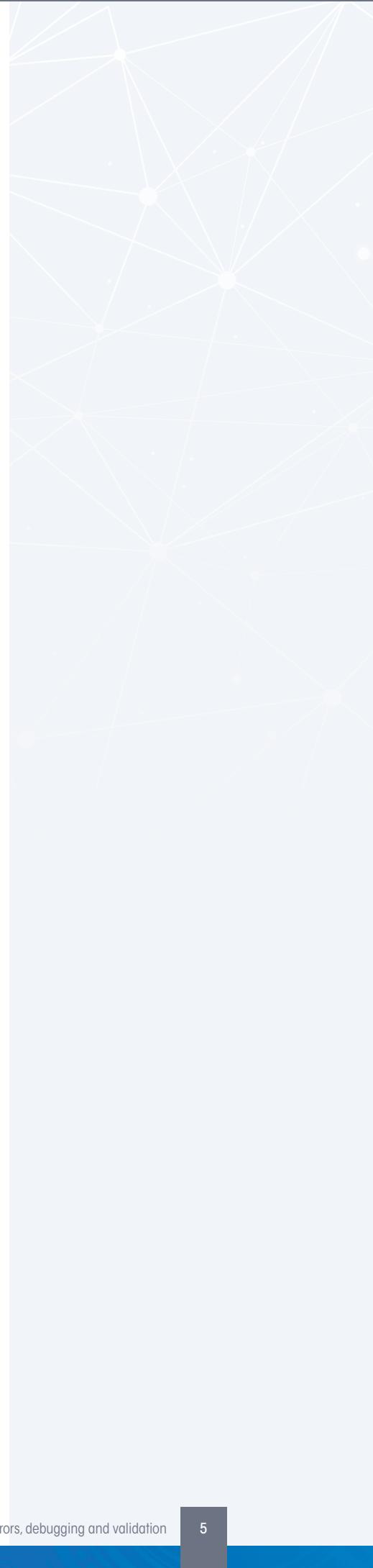


### Activity 1.1

**1.7.1** What is the difference between syntax, runtime and logic errors?

**1.7.2** Give two examples of each of the following errors:

- a. Syntax errors
- b. Runtime errors
- c. Logic errors



## 1.2 Mathematical methods

### METHODS

A method is a segment of pre-written programming code that performs a specific task. Methods are written by the developers of Delphi. There are two types of methods: functions and procedures. You have already worked with the data conversion functions `IntToStr`, `FloatToStr`, `StringToFloat` and `StringToInt` as well as formatting functions `FloatToStrF` and `Format`. In this chapter, we will discuss mathematical functions `ROUND`, `TRUNC`, `FRAC`, `CEIL`, `FLOOR`, `SQR`, `SQRT` and `PI` and the procedures `INC` and `DEC`.

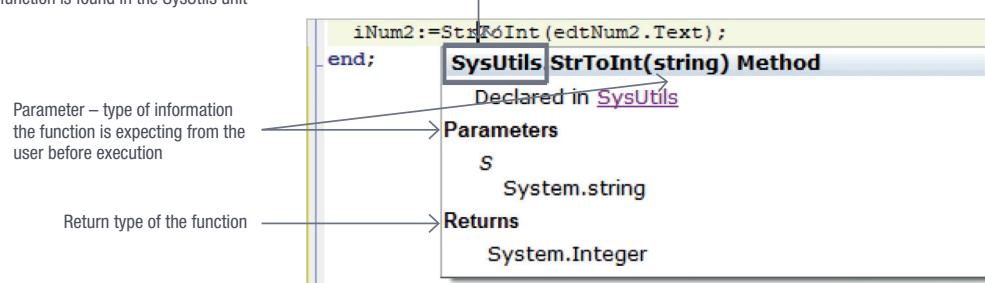
Methods are provided in Units for use by programmers. For example, the Math Unit hosts a collection of mathematical functions. The names of the Units that the programmers are most likely to use are automatically included in the `Uses` clause of the Unit of the Form.

#### **uses**

```
Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
Dialogs, StdCtrls, ComCtrls;
```

You can determine from which Unit a function or procedure is, by hovering your mouse over the function or procedure name. A tooltip appears displaying the information about the function or procedure. If you hover on the `StrToInt` function the tooltip below will appear:

`StrToInt` function is found in the `SysUtils` unit



When a programmer uses a function or procedure and the Unit name of the function or procedure does NOT appear in the `Uses` clause, then a compiler error will occur. The programmer needs to add the name of the Unit to the `Uses` clause. For example, if you are using the functions from the Math Unit then you have to add the Math Unit to the `Uses` clause:

```
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics,
  Controls, Forms, Dialogs, StdCtrls, ComCtrls, ExtCtrls, Math;
```

If you do not know the Unit name of the function or procedure, then do the following:

- Type the name of the function or procedure in the Delphi editor.
- Click on the name of the function or procedure you type in the bullet above.
- Press `<Ctrl> + <F1>` simultaneously.
- Delphi Help will provide you with the information of the function or procedure.

Functions and procedures can be called from any part of a program. When the functions or procedures are called, control from the program is transferred to the function or procedure. The programming code for the function or procedure is then executed. Once the programming code for the function or procedure terminates, control is transferred back to the main program.

## IMPORTANT INFORMATION ABOUT FUNCTIONS

Delphi prewritten functions can be called from any part of a program. A function is always called within another statement and returns a single value. For example:

The StrToInt function in the statement below is called in an assignment statement and returns an integer value.

```
iNum := StrToInt(edtValue.Text);
```

## PREDEFINED MATHEMATICAL FUNCTIONS

### RANDOM() FUNCTION

You learnt about the **Random** function in Grade 10. The RANDOM function generates a random number from 0 to less than 1.

```
rNumber := Random()
```

If you want to generate a number from a to b, then use the formula:

```
iNumber := Random(b - a + 1) + a
```

Example: To generate a random number in the range 10 to 99 the code is:

```
iNumber := Random(90) + 10
```

### RANDOMRANGE() FUNCTION

**Syntax:** RandomRange(Num1,Num2)

The **RandomRange** function generates a random integer number from Num1 to one less than Num2.

Example:

```
iNum := RandomRange(1,7);
```

This statement will generate a random integer number from 1 to 6.

### ROUND() FUNCTION

The **Round** function rounds a real number to an integer value.

The 'Banker's Rounding' method is used for rounding numbers. The real number X is rounded to the nearest whole number. If X is exactly halfway between two whole numbers, the result is always the even number of the two whole numbers.

For example:

```
rX := 14.5;  
iAns := Round(rX);
```

Steps to complete the rounding:

- Step 1: Establish between which two whole numbers rX lies. Example 14.5 lies between 14 and 15
- Step 2: Establish whether rX lies exactly halfway between the two whole numbers. In this case 14.5 lies exactly halfway between 14 and 15
- Step 3:
  - If rX lies exactly halfway, then round to the nearest even whole number. In this case 14.5 will be rounded down to 14
  - If rX does not lie exactly halfway between the two whole numbers, then round normally according to mathematical rules to the nearest whole number



## New words

**Random** – to generate a random number from 0 to less than 1

**RandomRange** – to generate a random integer number from Num1 to one less than Num2

**Round** – to round a real number to an integer value

**Trunc** – to remove or chop off the decimal part of the real number. It returns an integer after the truncation

**Frac** – to return the decimal part of a real number

**Ceil** – to round a real number up to the highest integer value

Other Examples:

STATEMENT	iANS
iAns := Round(12.4)	12
iAns := Round(12.5)	12 (rounds down to even)
iAns := Round(12.6)	13
iAns := Round(13.4)	13
iAns := Round(13.5)	14 (rounds up to even)
iAns := Round(13.6)	14



## Activity 1.8

**1.8.1** Study the Delphi statement below and then answer the questions that follow:

`redDisplay.Lines.Add(IntToStr(Round(rNum));`

- List the function(s) used in the statement.
- Indicate to which Unit the function(s) belong(s).

**1.8.2** Determine the value of iAns in each of the following statements:

- `iAns := Round (8.5);`
- `iAns := Round (20.4);`
- `iAns := Round (20.7);`
- `iAns := Round(21.5);`

## TRUNC() FUNCTION

The **Trunc** function truncates (removes or ‘chops off’) the decimal part of a real number. It returns an integer after the truncation.

Examples:

STATEMENT	iANS
<code>iAns := Trunc(12.4)</code>	12
<code>iAns := Trunc(12.5)</code>	12
<code>iAns := Trunc(12.8)</code>	12

## FRAC() FUNCTION

The **Frac** function returns the decimal part of a real number. It returns a real number.

Examples:

STATEMENT	rANS
<code>rAns := Frac(12.4)</code>	0.4
<code>rAns := Frac(12.5)</code>	0.5
<code>rAns := Frac(12.8)</code>	0.8

## CEIL() FUNCTION

The **Ceil** function rounds a real number up to the highest integer value. The Ceil Function belongs to the Math Unit and you must add the Math Unit to the Uses clause before using it, otherwise you will get a compiler error.

Examples:

STATEMENT	iANS
iAns := Ceil(12.4)	13
iAns := Ceil(18.5)	19
iAns := Ceil(-12.8)	-12 (remember that -12 is greater than -13)

### FLOOR() FUNCTION

The **Floor** function rounds a real number down to the lowest integer value. The Floor Function belongs to the Math Unit.

Examples:

STATEMENT	iANS
iAns := Floor(12.4)	12
iAns := Floor(18.5)	18
iAns := Floor(-12.8)	-13 (remember that -13 is less than -12)

### SQR() FUNCTION

The **Sqr** function returns the square of an integer or real number. The return value is the same type as the number being squared.

Examples:

STATEMENT	iANS	rANS
iAns := Sqr(12)	144	
rAns := Sqr(5.5)		30.25

### SQRT() FUNCTION

The **Sqrt** function returns the square root of a number. The result type is always real. Remember that the square root of a negative number is undefined.

Examples:

STATEMENT	rANS
rAns := Sqrt(144)	12
rAns := Sqrt(31.36)	5.6

### PI

**PI** is a predefined constant that returns a real number giving a useful approximation of the value of Pi.

Example:

STATEMENT	OUTPUT
redDisplay.Lines.Add(FloatToStr(Pi));	3.14159265358979



### New words

**Floor** – to round a real number down to the lowest integer value

**SQRT** – to return the square root of a number

**Pi** – is a predefined constant that returns a real number giving a useful approximation of the value Pi



## New words

**POWER** – to raise a base to a power and returns a real answer

**INC** – to increment the ordinal type variable passed to it

## POWER FUNCTION

Syntax: Power(base,power)

The **POWER** function raises a base to a power and returns a real answer. Both the *base* and *power* are real numbers.

Example:

STATEMENT	OUTPUT
redDisplay.Lines.Add(FloatToStr(Power(12.5,2)))	156.25
redDisplay.Lines.Add(FloatToStr(Power(3,4)))	81

## PREDEFINED PROCEDURES

A procedure is also a pre-written subroutine designed to perform a specific purpose. Unlike a function, a procedure can return no result, one result or more than one result. A call to a procedure is a standalone statement whilst a call to a function is always within another statement. Just like a function, a procedure is only executed when it is called. We are going to focus on two procedures: INC and DEC.

### INC() PROCEDURE

The **INC** procedure increments the ordinal type variable passed to it. In this chapter we will work with ordinal type integers and characters.

The default is to increment the ordinal variable by 1 unit:

Inc(iAns);

This statement is equivalent to iAns := iAns + 1;

However, you can increment an ordinal type variable by an integer set by yourself:

Inc(iAns,5);

This statement is equivalent to iAns := iAns + 5;

Example of code:

```
...
Line 1:    iNum := 6;
Line 2:    Inc(iNum);
Line 3:    Inc(iNum,5);
...
```

Explanation of code:

- In line 1, the value 6 is assigned to iNum
- In line 2, the value stored in iNum is increased by 1 – iNum holds the value 7.
- In line 3, the value stored in iNum is increased by 5 – iNum hold the value 12

Example of Code:

```
...
Line 1:    cLetter1 := 'A';
Line 2:    cLetter2 := 'T';
Line 3:    Inc(cLetter1);
Line 4:    Inc(cLetter2,2);
Line 5:    redDisplay.Lines.Add(cLetter1 + ' ' + cLetter2);
...
```

Explanation of code:

- In line 1, the value 'A' is assigned to cLetter1
- In line 2, the value 'T' is assigned to cLetter2
- In line 3, the value of cLetter1 is incremented by 1. The ordinal value of A is incremented by 1. You will learn more about ordinal values and ASCII values later on in this book. cLetter1 holds the value 'B'
- In line 4, the value of cLetter2 is incremented by 2. Again, it's the ordinal value of 'T' that is incremented by 2. cLetter2 holds the value 'V'
- In line 5, the values B V will be displayed

### DEC() PROCEDURE

The **DEC** procedures decrements an ordinal type variable. The default is to decrement the ordinal variable by 1 unit, but you can supply an integer to decrement by a different amount. Example of code using the Dec procedure:

When the code segment above is traced:

```
Begin
    ...
Line 1:    iNum1 := 7;
Line 2:    iNum2 := -5;
Line 3:    cLetter1 := 'B';
Line 4:    cLetter2 := 'T';
Line 5:    Dec(iNum1);
Line 6:    Dec(iNum2,3);
Line 7:    Dec(cLetter1);
Line 8:    Dec(cLetter2,2);
Line 9:    redisplay.Lines.Add(IntToStr(iNum1));
Line 10:   redisplay.Lines.Add(IntToStr(iNum2));
Line 11:   redisplay.Lines.Add(cLetter1);
Line 12:   redisplay.Lines.Add(cLetter2);
    ...
End;
```

LINE NUMBERS	INUM1	INUM2	CLETTER1	CLETTER2	OUTPUT
1	7				
2		-5			
3			B		
4				T	
5	6				
6		-8			
7			A		
8				R	
9					6
10					-8
11					A
12					R
Stop					



### New words

**DEC** – to decrement an ordinal type variable



### Take note

If you cannot see all the lines that you have generated in the Memo component, then you need to set the ScrollBars property to ssVertical.



### Activity 1.9

- 1.9.1** Open the **SquareCubeRoot\_p** project from the 01 – Square Cube and Square Root folder and create an OnClick event for the [Calculate] button to do the following:

- Generate a random number in the range 10 to 20 (inclusive).
- Display the random number in the EditBox.
- For each number from 1 to the random generated number, determine the square, cube and square root of the number. Tabulate the output. Display the number, its square, cube and square root with appropriate messages. Numeric values must be formatted to two decimal places.
- Save and run the project.

Number	Square	Cube	Square Root
1	1.00	1.00	
2	4.00	8.00	1.41
3	9.00	27.00	1.73
4	16.00	64.00	2.00
5	25.00	125.00	2.24

- 1.9.2** Open the **CircleAreaCircum\_p** project from the 01 – Circle Area and Circumference folder and create an OnClick event for the Calculate button to read the radius of a circle and calculate and display the radius, area and circumference of the circle. The formulae to calculate:

- Area :  $\pi r^2$
- Circumference :  $2\pi r$

Save and run your program.

Radius	5.40
Area	91.61
Circumference	33.93

- 1.9.3** Open the **LearnerAlphabet\_p** file from the 01 – Learner Alphabet folder. A letter of the alphabet is allocated in the following manner to each learner:

Learner 1: Z      Learner 2: X  
Learner 3: V      Learner 4: T      ....

Do the following:

- Create an OnCreate event to initialise variables for learner number and the letter of the alphabet that will be assigned to the first learner. Display the details for the first learner.
- Create an OnClick event for the Generate button to generate and display the next line of the sequence.

Learner: 1	Z
Learner: 2	X
Learner: 3	V
Learner: 4	T
Learner: 5	R
Learner: 6	P
Learner: 7	N
Learner: 8	I

**QUESTION 1 GENERAL PROGRAMMING SKILLS****INSTRUCTIONS**

Open the incomplete program in the Question1\_1 folder found in the Question 1 folder inside the 01 – Question 1 folder.

Add your name and surname as a comment in the first line of the Question1\_1\_u.pas file.

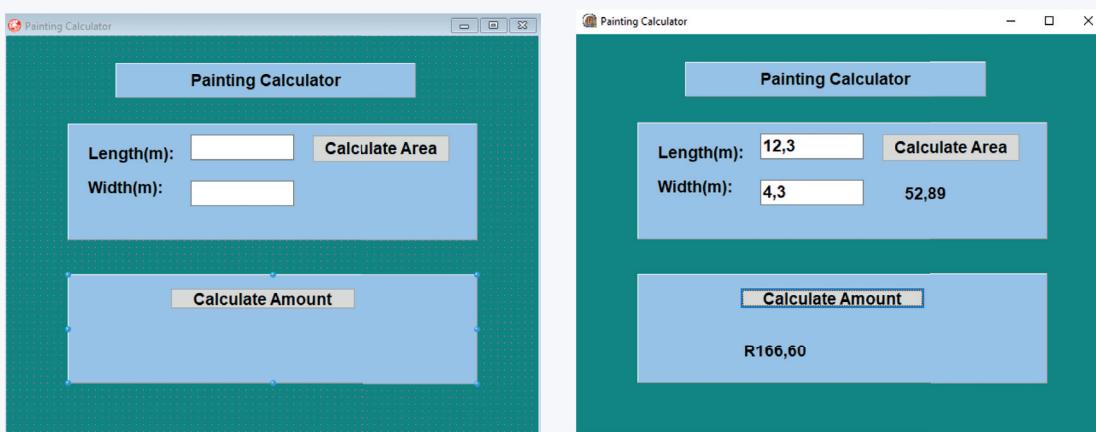
Compile and execute the program. The program currently has no functionality.

**SCENARIO**

Congratulations! You have been selected to take part in your schools Cricket tour in New Zealand in the upcoming December holidays. Unfortunately, there are some costs involved and your parents have stated that you will have to come up with some of the money yourself. Fortunately, you are an IT student and are able to use the skills you have learnt in IT to raise some funds.

You have decided to create two applications that will help you to keep track of the money you will make.

- 1.1** To earn some money, you have decided to do some painting for your neighbours and family. You charge a rate per m<sup>2</sup>. Open the Delphi project **Question1\_1\_p.dpr**, the form contains the following GUI:



- 1.1.1** In the *btnCalcArea* button's OnClick event.

- a. Declare the necessary variables for your calculations.
  - Length
  - Width
- b. Get the Length and Width as input from the two edits and assign them to your variables.
- c. Calculate the Area of the wall and display the output in *lblArea*.

- 1.1.2** In the *btnCalcAmount* button's OnClick event.

- a. Calculate the amount of money earned. Your fees are R3.15 per m<sup>2</sup> area.
- b. Display the output in *lblAmount* as Currency. Above is a screenshot of output.

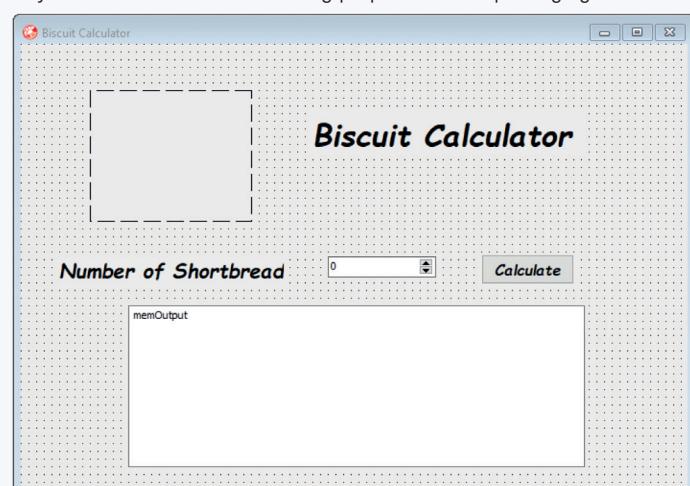
**CONSOLIDATION ACTIVITY****Chapter 1: Errors, debugging and mathematical methods *continued***

- 1.2** Open the incomplete program in the Question1\_2 folder inside the 01 – Question 1 folder.

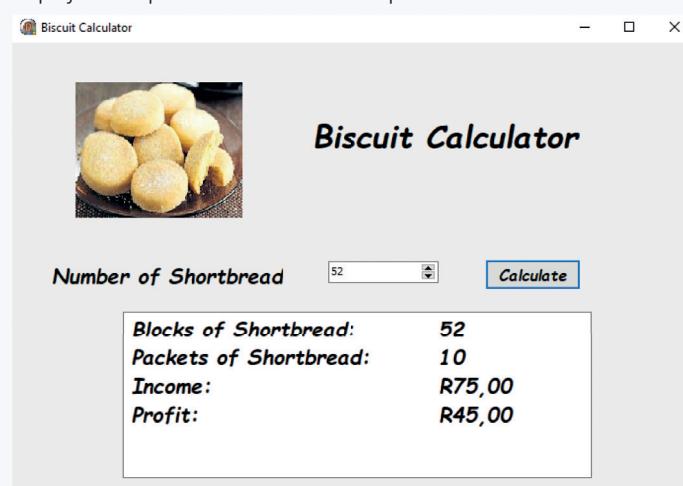
Open the Delphi project **Question 1\_2\_p.dpr** which contains the following GUI.

During the week, you have decided to make and sell square Shortbread biscuits at school. It costs you R30 to make a batch of biscuits that gives approximately 50 blocks of shortbread. You sell the biscuits in packets of 5 blocks for R7.50 per packet.

Any shortbread that breaks during preparation and packaging are not sold.

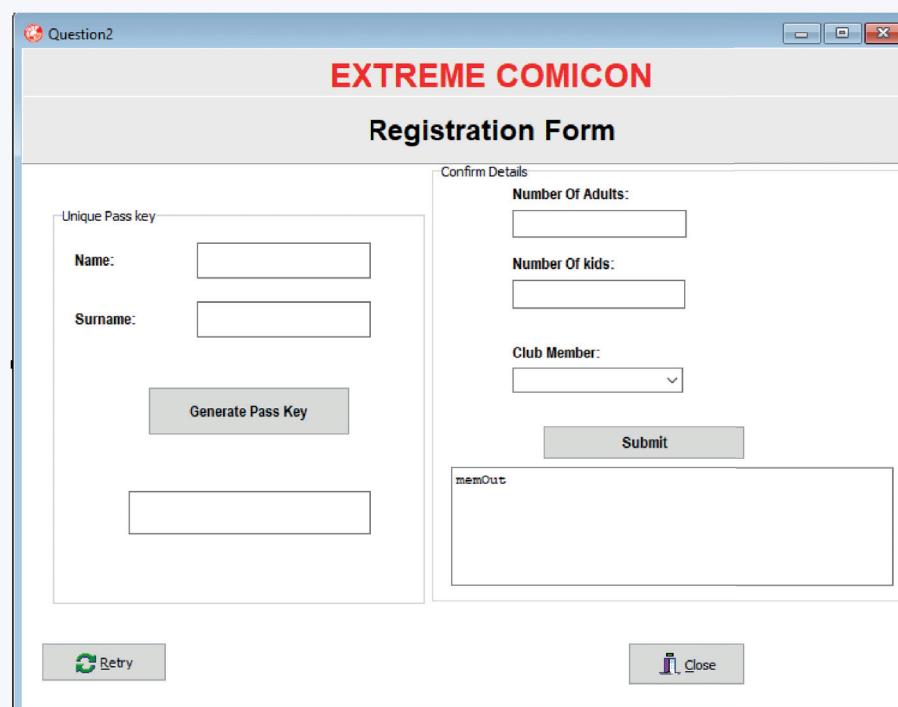


- 1.2.1** Insert the picture *Shortbread.jpg* into the image component and resize it to fit.
- 1.2.2** *spnShort* values needs to be between 40 and 55.
- 1.2.3** Get the number of usable blocks of shortbread from the spinner and calculate the number of packets that can be made.
- 1.2.4** Calculate the income and the profit which is made from selling that packets of shortbread.
- 1.2.5** Clear the *memOutput*.
- 1.2.6** Display the output as follows in *memOutput*. Use the test data to check it.



**QUESTION 2**

In the 01 – Question2 folder, open the project file **Question2\_p.dpr**. Figure 2-1 shows the GUI design.

**Figure 2-1**

Due to the growing number of Comicon enthusiasts, each year the number of participants increases making waiting to just get in considerably time-consuming. Extreme Comicon wants you to create a program that will make access to the Comicon by using a pass key and details hassle free. Add code to accomplish this task.

**2.1 Button [Generate Pass Key]**

Add code in the button to generate a unique pass key for members to be able to gain access:

- Two letters of the name starting at the second letter.
- The first four letters of surname
- The letters should all be uppercase
- Any four random digits
- Display the reference number in the TEdit edtPassKey

Example of Input/Output:

A screenshot of a Windows application window titled "Unique Pass key". It has "Name:" and "Surname:" fields with "Ishmail" and "Thyssen" respectively. Below is a "Generate Pass Key" button. At the bottom is a large TEdit control containing the generated pass key "SHTHYS443430".

## CONSOLIDATION ACTIVITY

### Chapter 1: Errors, debugging and mathematical methods *continued*

#### 2.2 Button [Submit]

Add code for the button to receive process and provide feedback on the various prices that has to be paid.

##### 2.2.1 Retrieve and save input from the input fields accordingly.

2.2.2 Adults pay R40 and kids pay R20 (these totals will never change), calculate the total amount that needs to be paid. Also take into consideration that members get 10% discount and non-members pay full price.

2.2.3 Provide code to display the details if the format is as follows:

Confirm Details

Number Of Adults:

Number Of kids:

Club Member:

**Submit**

memOut

Registration Details: <tab><tab> <current date>  
The adult ticket price is: <R> <adultprice> <2decimal places>  
The kids ticket price is: <R> <kidspice> <2decimal places>  
The total ticket price is: <R> <totalprice> <2decimal places>

#### Example of Input/Output

Confirm Details

Number Of Adults:

Number Of kids:

Club Member:

**Submit**

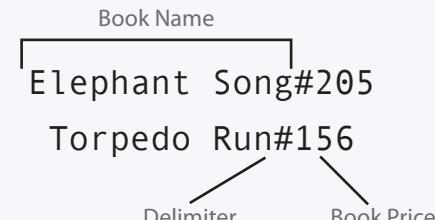
Registration Details: 2018/09/26  
The Adult ticket price is:R72.00  
The Kids ticket price is:R36.00  
The Total ticket price is:R108.00

## QUESTION 3

#### 3.1 Open the Question 3\_1 folder found in the Question 3 folder.

Open the file **Delimiting\_p.dpr**

In Grade 11 your syllabus includes the use of text files for storing data long-term. A common way of storing data of multiple categories is by saving strings in the text file that are divided by what we call delimiters. These strings would look something like the following:



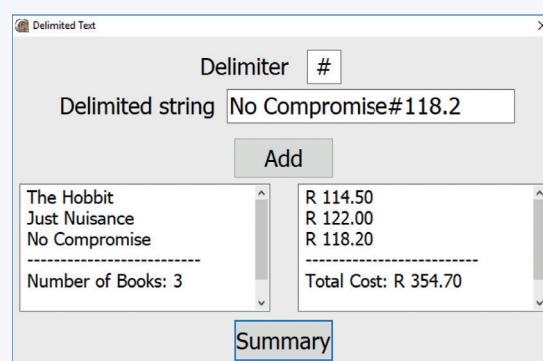
As a programmer you always save the name of the book first, then a delimiter character of your choice and then the price of the book.

NOTE: The delimiter can be any character, but it should be a character that is highly unlikely to ever be part of any of the strings involved so that the position of the delimiter character can tell us where to split the strings and then be removed.

**CONSOLIDATION ACTIVITY****Chapter 1: Errors, debugging and mathematical methods *continued***

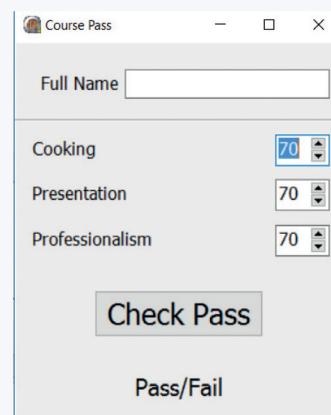
- 3.1.1** By using the delimiter character given in `edtDelimiter`, the string in `edtDelimitedString`, and your knowledge of string manipulation code the following in the `OnClick` event of the `btnAdd` button:
- Split the delimited string into its separate substrings.
- Output the name of the book in `memBook` and the price of the book into `memPrice` with the correct currency formatting.
- 3.1.2** Add code to the `OnClick` event of `btnSummary` that will:
- Show the total number of books in `memBook` and the total price of all the books in `memPrice` using the correct currency formatting.

Example:



- 3.2** Open the Question 3\_2 folder found in the 01 – Question 3 folder.

Open the project file **CoursePass\_p.dpr**



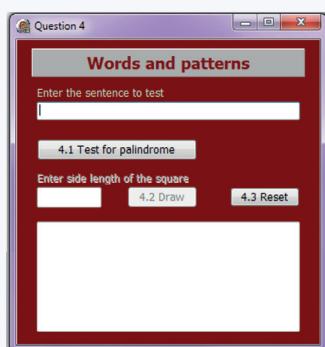
*Cooking for Business* is a company that offers short cooking courses for people who would like to start working in the food industry. This application is going to be used by participants to check if they have passed the course.

- 3.2.1** Use the `OnExit` event of `edtName` to validate the name that the user types in by using a loop to check if each character is either an uppercase letter, a lowercase letter, a space or a hyphen (-). If a character does not satisfy these criteria, then a relevant message should show, and focus should go back to `edtName`.
- 3.2.2** There are two pass criteria, namely:
- At least 80 for cooking, more than 60 for presentation
  - At least 80 for professionalism, above 50 for cooking and presentation
- Complete the IF-statement under the `OnClick` event of `btnCheckPass` to show whether the participants passed or failed.

**QUESTION 4 PROBLEM SOLVING**

Open the Question 4 folder.

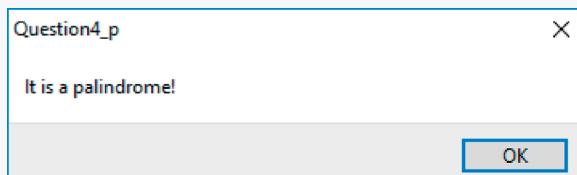
Load the project file **Question4\_p.dpr**



- 4.1** A palindrome is a word or phrase that is written exactly the same from front to back and back to front, for example:

- noon
- civic
- radar
- madam
- able was I ere I saw elba

Your task is to complete the [4.1 Test for palindrome] button code. When the button is pressed, the text in the text field must be tested. If it is a palindrome, a popup window must appear with a suitable message, for example:



If it is not a palindrome, it should also show a popup window with a suitable message. Capital letters and lowercase letters can be ignored. Once a palindrome is found, the [4.2 Draw] button and side length text field `edtSide` must become useable to the user.

- 4.2** Write the code for the [4.2 Draw] button. When the button is pressed, the value must be taken from the text field. This value must be the side length of a square filled with an X O X O X O pattern. Use the memo block for the output. Each odd row must start with an X and every even row must start with an O. Place spaces between the characters e.g.

A square with a side length of 5 will look like this

X O X O X  
O X O X O  
X O X O X  
O X O X O  
X O X O X

- 4.3** Write code for the reset button to clear the EditBoxes and MemoBox and to disable the [4.2 Draw] button and the side length EditBox.

Ensure that your name and surname has been entered as a comment in the first line of the program file.

Save your program

# NESTED LOOPS

## CHAPTER UNITS

- Unit 2.1 Nested loops
- Unit 2.2 Using nested loops
- Unit 2.3 Creating shapes using nested loops



## Learning outcomes



At the end of this chapter you should be able to

- describe the concept of a nested loop
- use nested loops in algorithms
- create simple nested loops in applications
- draw shapes using special symbols
- use trace tables to debug a nested loop.

## INTRODUCTION

In Annexure A we consolidated most work done in Grade 10. Looping formed an integral part of the Grade 10 curriculum. We focused on the three loops:

- FOR-loop
- WHILE-loop
- REPEAT-loop

In this chapter we will focus on nested loops. A nested loop is one loop within another loop. You will learn how to apply nested loops to develop algorithms, flowcharts and programs to solve programming problems. You will also use trace tables to test the correctness of algorithms, flowcharts and programs.

## 2.1 Nested Loops

A nested loop is a loop within a loop, that is, one loop that is placed inside the body of another loop. We refer to the first loop as the **outer** loop and the second loop as the **inner** loop. We say that the second loop (inner loop) is *nested* within the first loop (outer loop).

### STRUCTURE OF A NESTED LOOP

A nested loop is structured as follows:

```
Start of the first loop
begin
    ...
    Start of second the loop
    begin
        ...
        end;
    ...
    end;
```

**Note:**

- the second loop (inner loop) is nested within the first loop (outer loop)
- the inner loop and outer loop can be represented by a FOR-loop, WHILE-loop or REPEAT-loop
- the outer loop and inner loop can both be the same loop type, that is, either two FOR-loops or two WHILE-loops or two REPEAT-loops.

Look at the following examples:

#### Example Nested FOR-loop

```
for loopVariable1 := initialValue to/downto endValue do
begin
    statement(s);
    ...
    for loopVariable2 := initialValue to/downto endValue do
begin
    statement(s);
    ...
end;           //inner loop
...
end;           //outer loop
```

### Example Nested WHILE-loop

```
while condition1 do
begin
    statement(s);
    ...
    while condition2 do
    begin
        ...
        statement(s);
        ...
    end; //inner loop
    ...
end; // outer loop
```



### New words

**outer loop** – the outer part of a nested loop

**inner loop** – the inner part of a nested loop

### Example Nested REPEAT-loop

```
Repeat
    statement(s);
    ...
    Repeat
        statement(s);
        ...
        until condition2; //inner loop
    ...
until condition1; //outer loop
```

- The outer loop and inner loop can be a combination of the different loop types. Below is an example of a nested loop that is a combination of two different loops:

### Example NESTED loop

```
for loopVariable1 := initialValue to/downto endValue do
begin
    statement(s);
    ...
    while condition1 do
    begin
        statement(s);
        ...
    end; //inner loop
    ...
end; //outer loop
```



### Guided activity 2.1

You need to develop a multiplication table for a primary school learner as shown below:

$$1 \times 1 = 1 \quad 2 \times 1 = 2$$

$$1 \times 2 = 2 \quad 2 \times 2 = 4$$

$$1 \times 3 = 3 \quad 2 \times 3 = 6$$

You are required to create the 1 times and 2 times multiplication table. In the 1 times multiplication table, you only need to find the product of 1 multiplied by a multiplier from 1 to 3. This is also true for the 2 times multiplication table.

Let's create an algorithm and flowchart for the problem.

ALGORITHM	FLOWCHART
<pre>for I = 1 to 2 begin     for J = 1 to 3     begin         Answer = I * J         Display I, '*', J, '=', Answer     end end</pre> <p>The I-loop is the outer loop. The J-loop is the inner loop. The I-loop will run twice. The J-loop will run three times. The statements within the J-loop will be executed six times.</p>	<pre>graph TD     Start([Start]) --&gt; I_I1[I ← 1]     I_I1 --&gt; CondI{I ≤= 2}     CondI -- True --&gt; J_J1[J ← 1]     J_J1 --&gt; CondJ{J ≤= 3}     CondJ -- True --&gt; Process[Answer ← I * J]     Process --&gt; Display[/Display I, *, J, =, Answer/]     Display --&gt; J_Jplus1[J ← J + 1]     J_Jplus1 --&gt; CondJ     CondJ -- False --&gt; I_Iplus1[I ← I + 1]     I_Iplus1 --&gt; CondI     CondI -- False --&gt; Stop([Stop])</pre> <p>The flowchart illustrates the execution of nested loops. It starts at 'Start' and initializes <math>I \leftarrow 1</math> (1). It then checks if <math>I \leq 2</math> (2). If true, it initializes <math>J \leftarrow 1</math> (3) and checks if <math>J \leq 3</math> (4). If true, it performs the calculation <math>Answer \leftarrow I * J</math> (5), displays the result (6), increments <math>J \leftarrow J + 1</math> (7), and loops back to check <math>J \leq 3</math>. If <math>J \leq 3</math> is false, it increments <math>I \leftarrow I + 1</math> (8) and loops back to check <math>I \leq 2</math>. Once <math>I \leq 2</math> is false, the process stops at 'Stop'.</p>

The nested loop executes in the following manner:

- When the outer loop is executed for the first time, it triggers the execution of the inner loop. Control is transferred from the outer loop to the inner loop. The inner loop executes from its initial value to its end value and control is transferred back to the outer loop.
- For each change of value of the outer loop value, the inner loop is triggered. This process continues until the outer loop reaches its end value.
- Therefore in the example above, the inner loop will be triggered twice and for each trigger, the inner loop is executed three times.



### Guided activity 2.1 *continued*

The trace table for the flowchart:

BOX NO	I	J	ANSWER	I <= 2	J <= 3	OUTPUT
1	1					
2				T		
3		1				
4					T	
5			1			
6						1 * 1 = 1
7		2				
4					T	
5			2			
6						1 * 2 = 2
7		3				
4					T	
5			3			
6						1 * 3 = 3
7		4				
4					F	
8	2					
2				T		
3		1				
4					T	
5			2			
6						2 * 1 = 2
7		2				
4					T	
5			4			
6						2 * 2 = 4
7		3				
4					T	
5			6			
6						2 * 3 = 6
7		4				
4					F	
8	3					
2				F		
Stop						



## Guided activity 2.1

*continued*

**Note:**

- Boxes 5 and 6 are executed six times

The equivalent Delphi code for algorithm and flowchart:

```
var i, j, iAnswer: Integer;
begin
  memDisplay.Lines.Clear;
  for I := 1 to 2 do // outer loop
  begin
    for J := 1 to 3 do //inner loop
    begin
      iAnswer := I * J;
      memDisplay.Lines.Add(IntToStr(I) + ' * ' + IntToStr(J) + ' = ' + IntToStr(iAnswer));
    end;
    memDisplay.Lines.Add(' ');
  end;
```

Alternatively, the equivalent code for the flowchart can be written using nested WHILE-loops. Remember that you can use a FOR-loop or a WHILE-loop for a counter driven loop.

```
...
memDisplay.Lines.Clear;
I := 1;
while I <= 2 do
begin
  J := 1;
  while J <= 3 do
  begin
    iAnswer := I * J;
    memDisplay.Lines.Add(IntToStr(I) + ' * ' + IntToStr(J) + ' = ' + IntToStr(iAnswer));
    J := J + 1;
  end;
  memDisplay.Lines.Add(#13);
  I := I + 1;
end;
```



## 2.2 Using nested loops



### EXPLAINING BINARY NUMBERS



<https://www.youtube.com/watch?v=LpuPe81bc2w>  
<https://qrs.ly/6dab1zo>

### CONVERTING A BINARY NUMBER TO A DECIMAL NUMBER

In Grade 10 you also learned how to convert binary numbers into decimal numbers. Let's revise this work.

#### EXAMPLE TO CONVERT A BINARY NUMBER TO A DECIMAL NUMBER

To convert the binary number  $1101_2$  to a decimal number, you write the binary number in expanded notation and simplify:

$$\begin{aligned}1101_2 &= 1 * 2^3 + 1 * 2^2 + 0 * 2^1 + 1 * 2^0 \\&= 1 * 8 + 1 * 4 + 0 * 2 + 1 * 1 \\&= 8 + 4 + 0 + 1 \\&= 13\end{aligned}$$

#### Note:

- Any number raised to the power zero ( $N^0$ ) is equal to 1.
- You *must* follow the BODMAS rules.



#### Guided activity 2.2

Convert a binary number into a decimal number

Algorithm to convert a binary number into a decimal number:

```

Line 1      Read BinNum
Line 2      Index = length(BinNum)
Line 3      DecNum = 0
Line 4      for I = 1 to length (BinNum)
Line 5      begin
Line 6          Digit = Convert to integer BinNum[i]
Line 7          Index = Index-1
Line 8          if I = length(BinNum)
Line 9              DecNum = DecNum + 1 * Digit
Line 10         else
Line 11             begin
Line 12                 Prod = 1
Line 13                 for J = 1 to Index
Line 14                     Prod = Prod*2
Line 15                 DecNum = DecNum + Digit*Prod
Line 16             end;
Line 17             Display DecNum

```



## Guided activity 2.2

Convert a binary number into a decimal number *continued*

### Note:

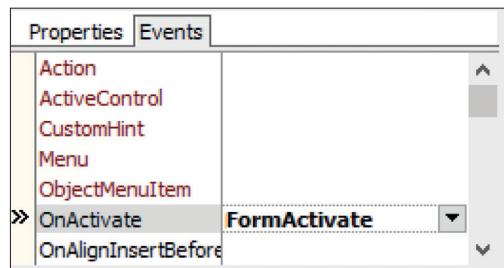
- Line 1: Read the binary number as a string *BinNum*.
- Line 2: Determine and store the length of the binary string in *Index*. The value of *Index* will be used to calculate  $2^{\text{Index}}$  for each binary digit.
- Line 3: Set the initial value of the decimal number *DecNum* to zero.
- Line 4: Set the outer *I*-loop to run from 1 to the length of the string.
- Line 5: The outer loop begins.
- Line 6: Extract the character from loop position *I* in the binary string and convert it to integer and store the value in *Digit*.
- Line 7: Each digit *Digit* will be multiplied by  $2^{\text{Index}}$ . We need to determine the value of the *index* using the statement *Index = Index - 1*. The initial value *Index* is set to the length of the binary string in Line 2. In this line the value of *Index* is decremented by 1. Suppose  $1101_2$  was read, then for initial value of *Index* for  $2^{\text{Index}}$  for the first digit is set as follows:
  - *Index* set to 4 in line 2
  - *Index* set to 3 in this line.
- Line 8: Determine whether the loop value for *I* equals the length of the binary string *BinNum*. If this is true, this means that the last character of the string *BinNum* has been reached.
- Line 9: If the condition in Line 8 is true, then calculate *DecNum = DecNum + 1 \* Digit*. Remember that the last digit is multiplied by  $2^0$ .  $2^0 = 1$
- Line 10: Else the condition in Line 8 is false
- Line 11: Else part begins
- Line 12: We are going to calculate  $2^{\text{Index}}$  using repeated multiplication. Therefore *Prod = 1*
- Line 13: Set the inner *J*-loop to run from 1 to the value of *Index*
- Line 14: The statement *Prod = Prod \* 2* is executed *Index* number of times
- Line 15: Set the value of *DecNum*: *DecNum = DecNum + Digit \* Prod*
- Line 16: Check whether the last value of the *I*-loop has been reached. If the last value of *I* has not been reached, the loop continues to execute
- Line 17: Display the decimal value *DecNum*

## ONACTIVATE EVENT

In Grade 10, you worked with the *OnClick* and *OnCreate* events. Another event we can use is called the *OnActivate* event. We use the *OnActivate* to perform special processing when the form receives focus. A form becomes active when focus is transferred to it.

To create an *OnActivate* event:

- click on the form component
- click on the *Event* tab in the Object inspector
- open the event handler framework for an event – *double click in the grey cell next to the name of the event*.



- An *OnActivate* event handler is created. Type the code to be executed within the BEGIN ... END of the event handler:

```
[begin code]
procedure TForm1.FormActivate(Sender: TObject);
begin
  edtNumber.SetFocus;
end;
[end code]
```



### Activity 2.2

Refer to the algorithm to convert a binary number into decimal number above .

- 2.2.1** Draw a trace table for the algorithm using an input value  $101_2$ .
- 2.2.2** Open the **BinaryToDecimal\_p** project in the 02 – Binary to Decimal Folder.
  - Create an *OnActivate* event on the form to set the cursor to focus on the **EditBox**.
  - Create an *OnClick* event for the [Convert To Decimal] button to convert a binary number to a decimal number. Use the algorithm to convert a binary number to a decimal number.
  - Run and execute the program.
  - Save the program.

## CONVERTING A DECIMAL NUMBER TO A BINARY NUMBER

In Grade 10, you learnt to convert a decimal number to binary number using the method shown below. For example: Convert 10 to binary.

NUMBER	REMAINDER
2	10
2	5
2	2
2	1
0	



$$10 = 1010_2$$

In the next guided activity, we will create an algorithm to convert a decimal number to a binary number. Work through the algorithm and make sure you understand each step.



### Guided activity 2.3

Algorithm to convert a decimal number to a binary number

```
Read number
BinNum = ''
Repeat
  Remainder = integer (remainder of number/2)
  BinNum = string value(remainder) + BinNum
  Number = integer (division of Number/2)
Until number = 0
Display BinNum
```



### Activity 2.3

**2.3.1** Open **DecimalToBinary\_p** project in the 02 – Decimal to Binary Folder to convert the first N integer numbers starting from 1 into binary. If N is 12, then numbers 1 to 12 will be converted to binary. Do the following:

- Create an OnActivate event for the form to set the focus on the EditBox.
- Read in the value of N from the EditBox.
- Write code for the [Convert To Binary] button to convert the N integer numbers into binary.
- Display the decimal number and its binary equivalent number in columns as shown below:

The screenshot shows a Windows application titled "Convert Decimal to Binary". It has a simple interface with a label "Enter Decimal Number" and an edit box containing "10". Below these is a button labeled "Convert To Binary". To the right of the input area is a scrollable memo box displaying a table of decimal and binary equivalents from 1 to 10. The table is as follows:

Decimal	Binary
1	1
2	10
3	11
4	100
5	101
6	110
7	111
8	1000
9	1001
10	1010

- Change the property of the MemoBox so that you can scroll through the display if all the data cannot be seen.

**2.3.2** The hexadecimal number system has 16 digits: 0–9, A–F.

You will remember from Grade 10 that the Hexadecimal numbers A–F represent the decimal numbers 10–15 respectively. Open the **ConvertToHex\_p** project in 02 – Hexidecimal to Decimal Vice Versa folder and do the following:

- Create an OnClick event for the [Convert to Dec] button to read in a hexadecimal number and convert the number to decimal. Display the Hexadecimal number and the Decimal number

The screenshot shows a Windows application titled "Convert Hex to Decimal & Vice Versa". It features two main sections. The top section is for hex-to-dec conversion, with a label "Enter Hexadecimal Number" and an edit box containing "13", and a button labeled "Convert to Dec". The bottom section is for dec-to-hex conversion, with a label "Enter Decimal Number" and an edit box, and a button labeled "Convert To Hex". To the right of these sections is a scrollable memo box showing a conversion table between hex and decimal values. The table includes rows for F, 15, 13, and 19.

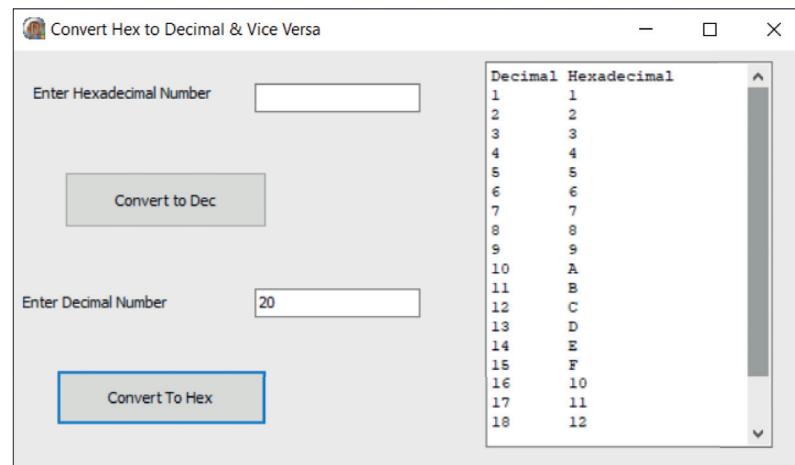
Hexadecimal Number	Decimal Number
F	15
13	19



### Activity 2.3

*continued*

- Create an OnClick event for the [Convert to Hex] button to read in a decimal number. Convert all decimal numbers from 1 to the number into their hexadecimal equivalent number. Display the decimal number and their corresponding hexadecimal number in columns.



**Note:** Scroll down to view 19–20.

## CREATING CIPHERS

Ciphers are used to send encrypted messages. These ciphers would use algorithms to hide the messages so that only people who knew the algorithm could unlock the messages and understand their meaning. In this section you will learn about two different types of ciphers. These are:

- Number cipher
- Caesar cipher

### NUMBER CIPHER

In a number cipher each character in a message is converted to an encrypted number in the following manner:

- All the characters that are going to be used to create messages are identified and a character list is compiled. Example:  
ABCDEF<sup>GHIJKLMNOPQRSTUVWXYZ</sup> 0123456789.,!?:-+="( )[]

**Note:** a space character appears after the character 'Z' in the character list.

- A message is created using the character list and all letter characters are converted to uppercase.
- Each character in the character list has a numerical position in the list:  
ABCDEF<sup>GHIJKLMNOPQRSTUVWXYZ</sup> 0123456789.,!?:-+="( )[]  
A is in position 1  
B is in position 2  
C is in position 3  
...

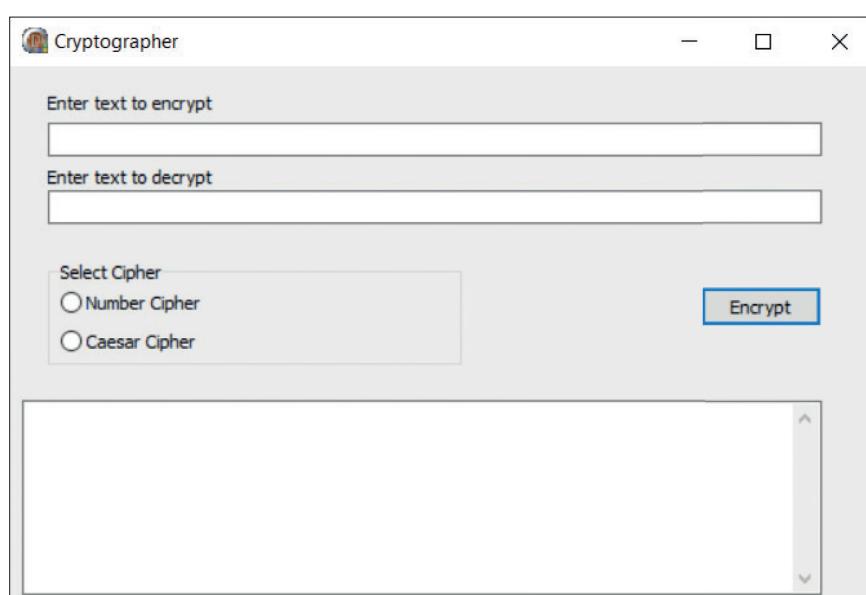
- The numerical position of each character in the message in the character list is found. Example: the numerical position of each character in the message Stop the war is:  
19 20 15 16 27 20 8 5 27 23 1 18
- The numerical positions of the characters is decreased by a fixed number. For example, let's decrease the numerical position of the characters by 5. The encrypted message is then:  
14 15 10 11 22 15 3 0 22 18 -4 13  
The encrypted characters are separated by space.



#### Guided activity 2.4

#### The Cryptographer

Open the **Cryptographer\_p** project in the 02 – Cryptographer folder and create an OnClick event for the [Encrypt] Button to do the following:



- Set the characters of the character list in the string variable *sCharacterList*. Set the initial value of the encrypted string *sOutput* to null.

```
sOutput := '';
sCharacterList := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ
0123456789.,!?:-+="()[]';
```

Read the message that you want encrypted from the EditBox and covert all letter characters to uppercase:

```
sText :=UpperCase(edtText.Text);
```

- Check if the [Number Cipher radio] button is selected. If the [Number Cipher] radio button is selected, then:
  - Extract each character from the message and check for its numerical position in the character list *sCharacterList*.
  - Store the numerical position in *iNumber*
  - Add the numerical position and a space to variable *sOutput*



## New words

**Caesar cipher** – a substitution cipher on which each letter in plaintext is ‘shifted’ a certain number of places down the alphabet

**encrypted message** – to encode information to prevent anyone other than its intended recipient from viewing it



## Guided activity 2.4

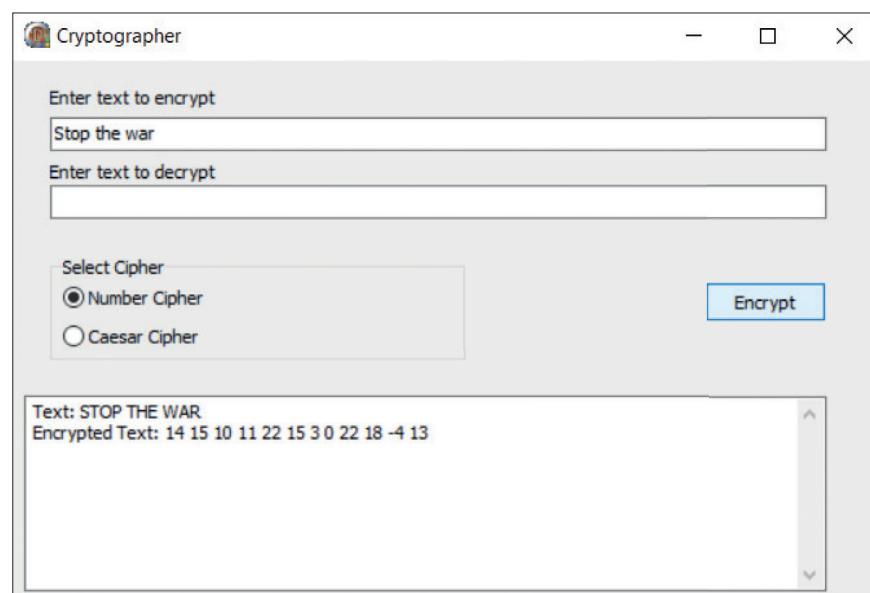
### The Cryptographer *continued*

```
if rgpCipher.ItemIndex = 0 then
begin
  for i := 1 to Length(sText) do
  begin
    sChar := sText[i];
    for j := 1 to Length(sCharacterList) do
      if sChar = sCharacterList[j] then
        iNumber := j;
    iNumber:= iNumber-5;
    sOutput := sOutput + IntToStr(iNumber) + ' ';
  end;
end;
```

- Display the message string sText and the encrypted message sOutput

```
memDisplay.Lines.Add('Message: ' + sText);
memDisplay.Lines.Add('Encrypted Message: ' + sOutput);
```

- Save and run the project.



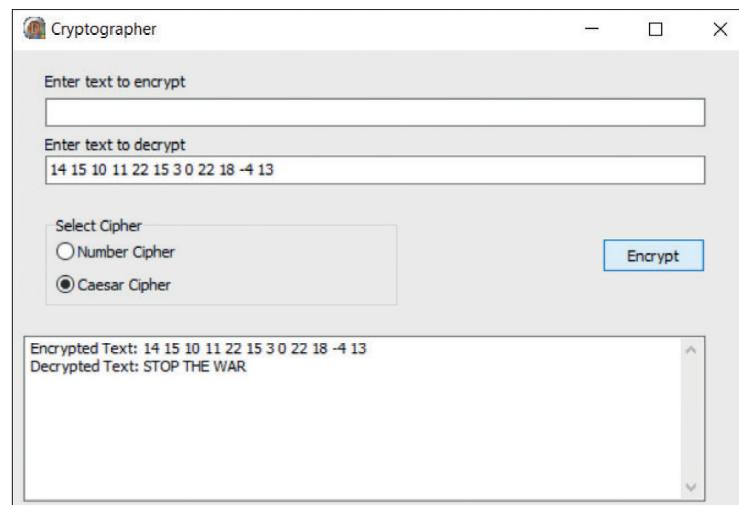
### CAESAR CIPHER

In the Number cipher you converted all the characters in a message to numbers, and then decreased the numbers by a fixed value to obtain an encrypted message. In a **Caesar cipher**, you take the **encrypted message**, and add the fixed value that was subtracted from it and convert the numbers back to corresponding characters from the character set.



## Activity 2.4

- 2.4.1** Open the **Cryptographer\_p** project in the 02 – Cryptographer folder and add code to the OnClick event for the [Encrypt] Button to do the following:



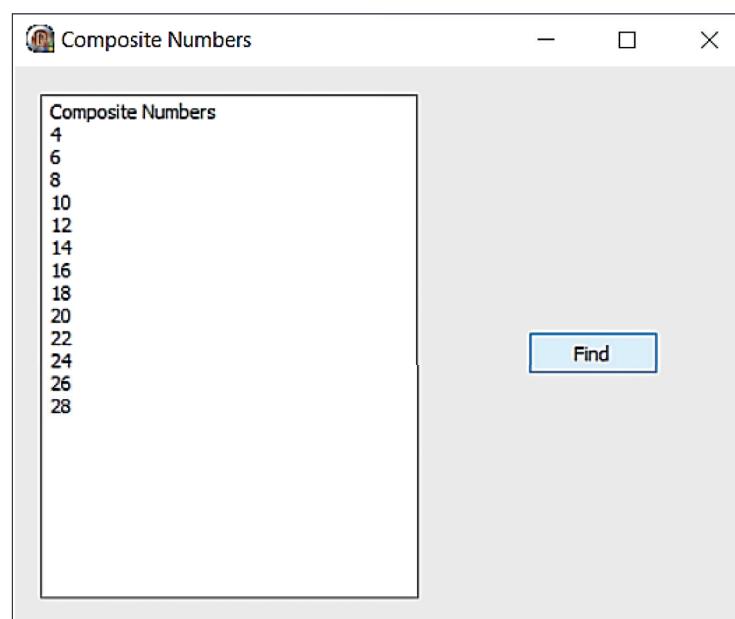
- If the radio button Caesar Cipher is selected then:
  - Set a new string variable to null.
  - Read the message that you want to decrypt in the appropriate EditBox.
  - Isolate the first number in the encrypted message and convert the number to an integer and add 5 to the number.
  - Find the character that is stored in the character list in the integer position calculate in bullet 2. Add the found character to the string created in bullet.
  - Continue with bullets 2 and 3 until the last number in the encrypted message is decrypted to the correct character.
- Save and run your project.

- 2.4.2** A composite number is a number that has 3 or more factors.

For example: 4 is a composite number because it has three factors: 1 2 4

Open the **CompositeNumber\_p** project in the 02 – Composite Number Folder.

Write code for the OnClick event for the [Find] button to display all the composite numbers that are less than 30.



## 2.3 Creating shapes using nested loops

You can create simple geometrical shapes using special characters such as '\*' or digits.

### Example 2.1

Create the shape below using the special character '\*'.

We know that there are six rows and each row has 6 '\*'s. The code to draw this shape:

```
*****  
*****  
*****  
*****  
*****  
*****
```

```
Line 1:   for i := 1 to 6 do  
Line 2:     begin  
Line 3:       sLine := '';  
Line 4:         for j := 1 to 6 do  
Line 5:           sLine := sLine +'*';  
Line 6:           memDisplay.Lines.Add(sLine);  
Line 7:     end;
```

#### Note:

- Line 1: The outer *i*-loop will run six times because there are six rows.
- Line 3: String *sLine* is set to null for each row.
- Line 4-5: In each row, 6 '\*'s must be joined together.
- Line 6: Display the string *sLine* before moving to the next row.

### Example 2.2

Creating a shape

We know that there are six rows and each row has 6 '\*'s. The code to draw this shape:

```
*
```

```
**
```

```
***
```

```
****
```

```
*****
```

The code to create this shape:

```
Line 1:   for i := 1 to 5 do  
Line 2:     begin  
Line 3:       sLine := '';  
Line 4:         for j := 1 to i do  
Line 5:           sLine := sLine +'*';  
Line 6:           memDisplay.Lines.Add(sLine);  
Line 7:     end;
```

### Example 2.2

### Creating a shape *continued*

#### Note:

- Line 1: The outer *i*-loop runs from 1 to 5 because we have five rows.
- Line 3: String sLine for each row is set to null.
- Line 4 & 5: Each row does not display the same number of \*s. Therefore the inner *j*-loop cannot run to a fixed constant value. We see from the table below that there is a relationship between the row number and the number of \*s that will be displayed. Hence the inner loop runs from 1 to *i*.

ROW NUMBER	NUMBER OF *S
1	1
2	2
3	3
4	4
5	5

- Line 6: Display the string sLine before moving to the next row.
- If we want to display the same shape formatted as follows:

```
*  
**  
***  
****  
*****
```

- You would need to replace the statement:

```
memDisplay.Lines.Add(sLine);
```

with

```
memDisplay.Lines.Add(Format('%10s',[sLine]));
```

### Example 2.3

### Creating a shape

Create the shape below:

```
*  
**  
***  
****  
*****  
*****  
****  
***  
**  
*
```

### Example 2.3 Creating a shape *continued*

The code to create the shape:

```
for i := 1 to 6 do
begin
    sLine := '';
    for j := 1 to i do
        sLine := sLine +'*';
    memDisplay.Lines.Add(sLine);
end;

for i := 5 downto 1 do
begin
    sLine := '';
    for j := 1 to i do
        sLine := sLine +'*';
    memDisplay.Lines.Add(sLine);
end;
```



### Activity 2.5

**2.5.1** Study the code segment below and determine the output:

```
for i := 1 to 5 do
begin
    sLine := '';
    for j := 1 to i do
        sLine := sline + IntToStr(i) ;
    memDisplay.Lines.Add(sLine);
end;
```

**2.5.2** Open the **Shapes\_p** project in the 02 – Shapes Folder and do the following:

- Set the font property of the MemoBox to Courier New.
- Create an OnClick event for the [Pattern 1] button to create and display the pattern alongside. The stars are separated by spaces. The memo component must be cleared before displaying the pattern.
- Create an OnClick event for the [Pattern 2] button to create and display the pattern below.

The memo component must be cleared before displaying the pattern.

```
*  
* *  
* * *  
* * * *  
* * * * *
```

```
*  
* * *  
* * * *  
* * * * *  
* * * * * *
```

- Create an OnClick event for the [Pattern 3] button to create and display the pattern below.  
The memo component must be cleared before displaying the pattern.

Options
1
2 3
4 5 6
7 8 9 10
11 12 13 14 15

**QUESTION 1**

- 1.1** Assume that the value for variable iSize is 5. Determine the output that will be produced by the following code segment:

```
for i := 1 to iSize do
begin
    sLine := '';
    for j := iSize downto i do
        sLine := sLine +'*';
    memDisplay.Lines.Add(sLine);
end;
```

- 1.2** Study the code below:

```
for i := 1 to 4 do
begin
    for j := 1 to 4 do
begin
    sLine := sLine + j + ' ';
    memOutput.Add(sLine);
end;
end;
```

The code was written to produce the output below:

1 2 3 4  
2 3 4  
3 4  
4

The program has errors. Correct the program so that it produces the correct output.

**QUESTION 2**

- 2.1** Open the **ConPatterns\_p** project in the 02 – ConShapes Folder and do the following:

- 2.1.1** Code the OnClick event for the [Pattern 1] button to test your code for question 1.2 above.  
**2.1.2** Code the OnClick event for the [Pattern 2] button to display the stars as shown below. The stars are separated by spaces.

```
* * * * *  
* * * * *  
* * * * *  
* * * * *  
* * * * *
```

## CONSOLIDATION ACTIVITY

### Chapter 2: Nested loops *continued*

- 2.1.3 Code the OnClick event for the [Pattern 3] button to display the stars as shown below. The stars are separated by spaces.

```
*  
* *  
* * *  
* * * *  
* * * * *  
* * * * *  
* * * *  
* * *  
* *  
*
```

### QUESTION 3

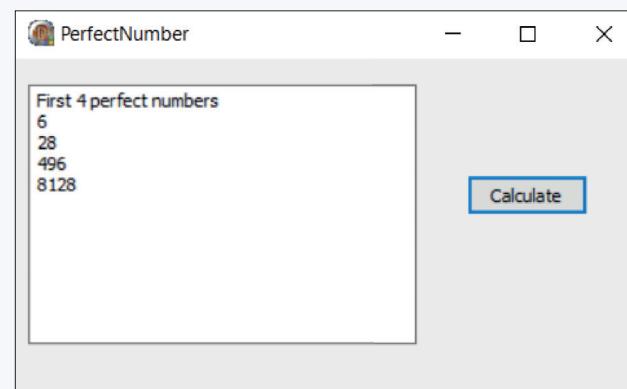
- 3.1 A number is a perfect number if its factors (excluding the number itself) sums up to the number.

Example: The factors of 6 are: 1 2 3 6.

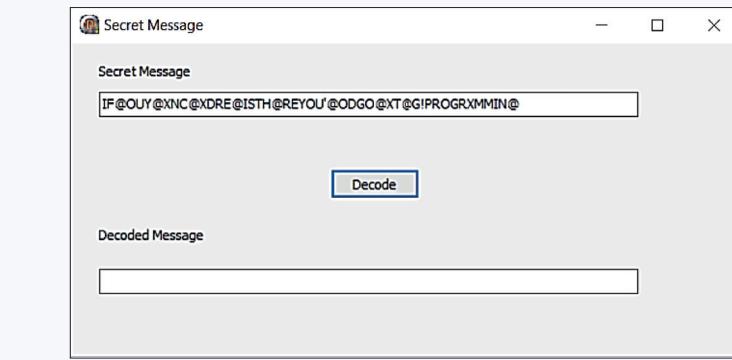
$1 + 2 + 3 = 6$ . Therefore 6 is a perfect number.

Open the **PerfectNumber\_p** project in the 02 – Perfect Numbers Folder. Create an OnClick event for the [Calculate] button to calculate the first four perfect numbers.

Sample Run:



- 3.2 Open the **SecretMessage\_p** project in the 02 – Secret message folder. The following interface will display when the program is run:



## CONSOLIDATION ACTIVITY

### Chapter 2: Nested loops *continued*

The following rules were applied to encode the secret message:

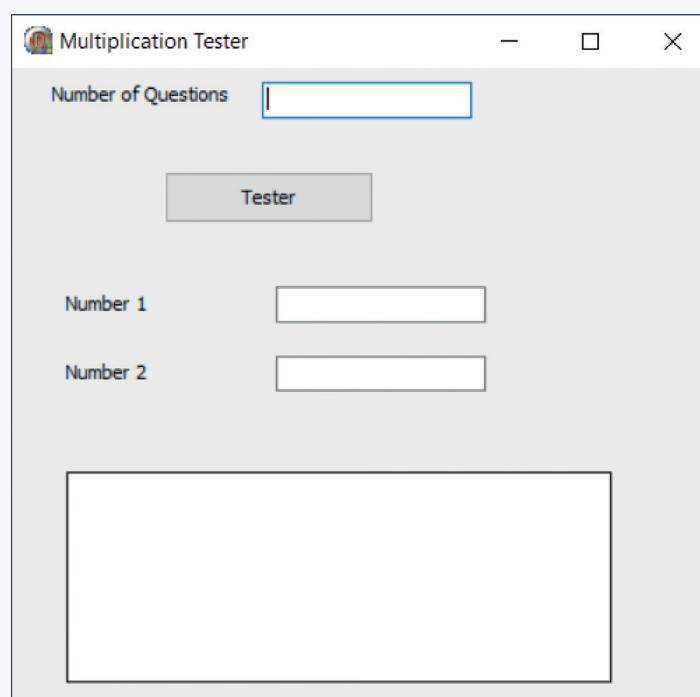
- All the characters were converted to uppercase.
- The characters in a word were rearranged only if the word length was three or more characters. In the rearrangement, the last two characters of the word were moved to the beginning of the word. Example the word 'you' became 'ouy'.
- A space character was replaced by the '@' character.
- The 'A' character was replaced by the 'X' character.

Write code for the OnClick event for the [Decode] button, using the rules above to decode the secret message.

#### QUESTION 4

A multiplication tester program is designed to test the product of two numbers. The user is prompted for the number of questions the user would like to work through. For each question, two random numbers in the range 1 to 10 (inclusive) are generated and the user is asked to supply the product. If correct, four points are awarded.

If incorrect, the user is given another chance; if correct this time, only two points are awarded. If the user fails on the second chance, the program provides the answer and no points are awarded.



Open the **MultiplicationTester\_p** project in the 02 – Multiplication Tester Folder and create an OnClick event for the [Tester] button to do the following:

- Read the number of questions from the EditBox edtNumQuestions.
- Generate two numbers iNum1 and iNum2 and display the numbers in the edtNum1 and edtNum2 EditBoxes respectively.
- Prompt the user for the correct answers using an Input box.
- Display whether the answer is correct or incorrect using a ShowMessage box.
- Display the Points earned in the memDisplay box.

**CONSOLIDATION ACTIVITY****Chapter 2: Nested loops *continued***

Example output for Question 4

Enter answer: Question 2 X

2\*4  
9

OK Cancel

Multiplicationtester\_p X

Incorrect

OK

Enter answer: Question 3 X

6\*2  
8

OK Cancel

Multiplicationtester\_p X

Incorrect

OK

Enter answer: Question 3 X

6\*2  
12

OK Cancel

Multiplicationtester\_p X

Correct

OK

**Total Points: 6**

CHAPTER  
3

# ARRAYS

## CHAPTER UNITS

- Unit 3.1 Arrays
- Unit 3.2 Searching and sorting arrays
- Unit 3.3 Parallel arrays



## Learning outcomes



At the end of this chapter you should be able to:

- describe the concept of an array
- define the structure and syntax of an array
- use different input sources to add data to arrays
- perform calculations using arrays
- format and display the output of an array
- describe the concept and use of parallel arrays
- use parallel arrays in calculations
- search for data in single and parallel arrays
- sort single and parallel arrays.

## INTRODUCTION

In many applications, large amounts of data need to be stored and accessed randomly. While it might be possible to create a variable for each data item that needs to be stored (such as your website's usernames and passwords), this could cause a number of issues. These include:

1. It may require you to create thousands or even millions of variables.
2. You would need to update the code of your application and manually create new variables every time the data changes (or someone new registers on your website).
3. Since each of these variables are independent of the others and have unique names, there is no easy way to loop through all of them.

So, instead of creating separate variables for each item, this chapter will teach you how to create an array that can store multiple values in the same structure.



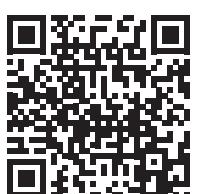
## New words

- homogenous** – elements of the same type
- index** – the position of the element in an array
- array** – is a data structure that stores a set of values (elements) of the same type linked to a single variable name

## 3.1 Arrays



### DELPHI ARRAYS



<https://www.youtube.com/watch?v=a7V8P4zE0ss>

TRO -SHORT LINK



### Take note

"Range check error" will appear if you try an access an element outside the index range.

- The elements of an integer array declared non-locally (globally) will by default be assigned 0.
- The elements of an integer array declared locally will by default be assigned random values.
- The elements of a string array will by default be assigned null values. However, it is good practice to initialise the values of an array.



### Remember!

An integer variable declared non-locally (globally) will by default be assigned 0.

### INTRODUCTION TO ONE-DIMENSIONAL ARRAYS

An array is a data structure that stores a set of values (elements) of the **same type** (**homogeneous**) linked to a single variable name. During this year, we will be working with one-dimensional (1D) arrays.

A one-dimensional array contains only one row for storing data. The elements in an array are ordered. The ordering property implies that the first, second, ..., last element can be identified or referenced.

Here is an example of a one-dimensional array, arrNames

Index	1	2	3	4	5
arrNames	Arhaan	Joyce	Peter	Andile	Zinhle

#### Take note of the following:

The name of the array is arrNames. We add the prefix **arr** to the name of the array. A meaningful name, which is dependent on the contents it holds, should be given to the array. This array contains elements of data type string. You can also use arrays of data type integer, real or character.

**Index** refers to the position of the element in an array. An element in the array is referenced using the name of the array and its index position within square brackets. For example, arrNames[2] refers to the second element 'Joyce' in the array; arrNames[4] refers to the fourth element 'Andile' in the array.

### DECLARATION OF AN ARRAY

Just like a variable, an array must be declared before it is used. Here is an example of the declaration of an array:

```
arrName: array[StartIndex..LastIndex] of Type;
```

- The reserved word **array** indicates that it is an array data structure.
- [StartIndex..LastIndex] indicates the size of the array. *StartIndex* indicates the starting position of the index and *LastIndex* indicates the last position of the index. For example, [1..10] OR [21..30] OR ['A'..'J'] OR ['a'..'j'] indicates that the array will hold 10 elements of data type.
- *Type* refers to the data type of the elements in the array. Remember that all elements in an array must have the same data type. This can be strings, integers, reals or even components (like labels or images).
- Each element is referenced by the name of the array followed by its index position. For example,

*arrName[3]*

*arrName[i]* where *i* is in the range 1 to *lastIndex*

- Delphi has a feature that allows you to check whether you are within a range of the array by adding the **{\$R+}** in the implementation section of the program:
  - Implementation
  - {\$R \*.dfm}
  - {\$R+}

This is an instruction to the compiler to do range checking.

## DECLARING AND POPULATING AN ARRAY SIMULTANEOUSLY

You can declare and populate an array at the same time. Like variable declarations, an array can be a constant array or an array where elements can vary.

### DECLARING AND POPULATING A CONSTANT ARRAY

You can declare a constant array, which means that the elements in this array cannot be modified. This constant declaration can be declared locally or non-locally.

```
const
  arrDays:array[1..7]of string = ('Sun','Mon','Tue','Wed','Thur','Fri','Sat');
  arrMonths:array[1..12] of string = ('Jan','Feb','Mar','Apr','May','Jun','July',
  'Aug','Sept','Oct','Nov','Dec');
```

### DECLARING AND POPULATING A NON-CONSTANT ARRAY

The elements in this array can be modified and this array can be declared locally or non-locally.

```
Var
  arrNames:array[1..6] of String = ('Tom','Jerry','Mickey', 'Mouse',
  'Daisy','Donald');
  arrPoints:array[1..10] of integer = (56,45,78,36,45,62,25,78,96,25);
```

## POPULATING AN ARRAY

### ASSIGNING VALUES TO AN ARRAY IN A SPECIFIC POSITION

The statements below demonstrate how elements in an array are assigned values:

```
arrNames[1] := 'Arhaan';
arrValue[7] := Sqr(5) ;
arrNumPeople[iCount] := 6;
arrMaxRainfall[i] := arrMaxRainFall[i + 1];
arrMark[4] := StrToInt(edtMark.text);
```

### ASSIGNING FIXED VALUES TO AN ARRAY

Array *arrSchools* is declared non-locally (globally). This method of assigning fixed values are used when the values are relatively fixed and the likelihood of it changing are slim. The code below is implemented on a *formActivate* event.

```
procedure TForm1.FormActivate(Sender: TObject);
begin
  arrSchools[1] := 'New Forest';
  arrSchools[2] := 'Stanger High';
  arrSchools[3] := 'Mountview High';
  arrSchools[4] := 'Fairmont High';
  arrSchools[5] := 'Mandini Academy';
end;
```

## READING DATA INTO THE ELEMENTS OF AN ARRAY FROM THE USER

To add values to an array, an *InputBox* is commonly used with a FOR-loop:

```
for i := 1 to 5 do  
    arrScores[i]:=StrToInt(InputBox('Scores','Enter score',''));
```

By using the *InputBox* with an array and a FOR-loop, you can ask the user to enter a value for each element of the array.

## DISPLAYING ELEMENTS FROM AN ARRAY

### DISPLAYING AN ELEMENT FROM AN ARRAY

You can display individual elements of an array. Here is the code you can use to display individual elements:

```
ShowMessage(arrName[1]); // display the element from index position 1  
memDisplay.Lines.Add('The 7th element is: '+ IntToStr(arrValue[7]));
```

### DISPLAYING ALL THE ELEMENTS IN AN ARRAY

Suppose you have five elements in *arrScore* and you want to display these elements in the MemoBox *memDisplay*. You can do this as follows:

#### Method 1: Displaying elements using an output statement for each element

```
memDisplay(IntToStr(arrScore[1]));  
memDisplay(IntToStr(arrScore[2]));  
memDisplay(IntToStr(arrScore[3]));  
memDisplay(IntToStr(arrScore[4]));  
memDisplay(IntToStr(arrScore[5]));
```

Method 1 is fine as long as you have only a few elements in the array. However, what happens when you have 100 elements. Using this program will become cumbersome because you will have 100 output statements. Instead, you can use a loop to display a large amount of elements.

#### Method 2: Using a loop to display elements

```
For i := 1 to 5 do  
    memDisplay.Lines.Add(IntToStr(arrScores[i]));
```



#### Take note

We access all five elements in the array using a loop. There are five elements in the array and we run the loop from 1 to 5. The loop counter is linked to the index position of the elements in an array. Each time the loop counter is incremented, it points to the next element in the array.



### Activity 3.1

**3.1.1** Pen and paper activity. Write down code that will do the following:

- Declare an array called *arrName* that can store three string values.
- Store the names of a friend, a family member and a pet in the array (in that order).
- Use the *ShowMessage* method to display the following names:
  - a. Pet's name
  - b. Family member's name

**3.1.2** Pen and paper activity. Write down code that will do the following:

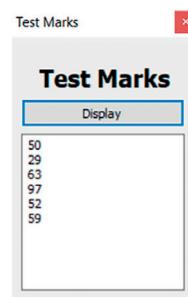
Declare the following four array variables.

NAME	INDEX	TYPE
arrAlphabet	1 to 26	Char
arrProvinces	1 to 9	String
arrAges	1 to 120	Integer
arrCardInDeck	1 to 52	Boolean

Assign the name of South Africa's nine provinces into an array *arrProvinces*.

Assign the value FALSE to every second value of array *arrCardInDeck*.

Create a FOR-loop that assigns random numbers between 1 and 20 (inclusive) to array *arrNumbers*. Array *arrNumbers* is type integer containing 50 elements.



**3.1.3** Open the **TestMarks\_p** project in the 03 – Test Marks Folder. The project is supposed to display the marks of the learners which are stored in the array. However, there are errors in the program. Correct the errors in the program so that it produces the correct output as shown on the right hand side.

**3.1.4** Open the **FamilyTree\_p** project in the 03 – Family Tree Folder and do the following:



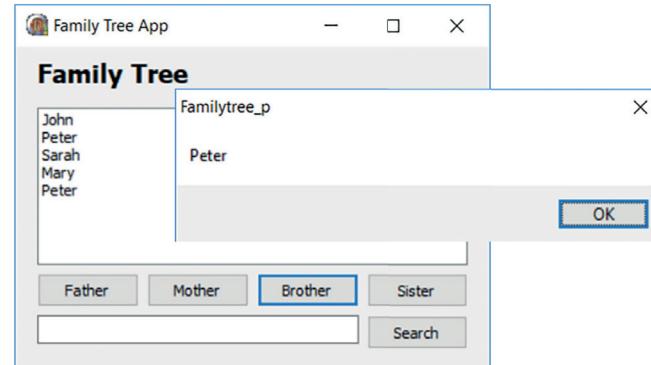
- a. Declare a non-local (global) array called *arrNames* since the values in this array will be accessed by each of the buttons.
- b. Create an *OnCreate* event for the form to assign the name of a family member to each of the elements in the array:

Father: John  
Mother: Mary  
Brother: Peter  
Sister: Sarah
- c. Create an *OnClick* event for [Father] button to display the name of the father from the array in a ShowMessage box and also added to the ListBox *lstFamily*.
- d. Create *OnClick* events for the [Mother], [Brother] and [Sister] buttons to display the relevant names from the array in a ShowMessage dialogue box and in the ListBox *lstFamily*.



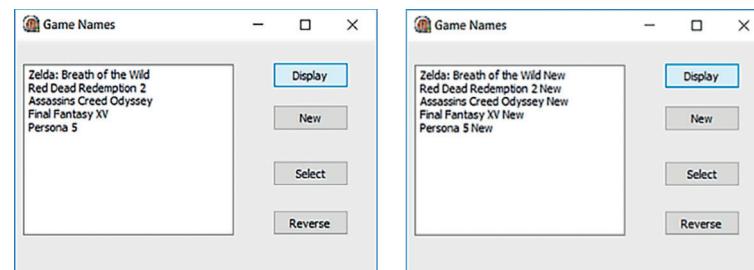
### Activity 3.1 *continued*

- e. Save and test your application. You should now be able to display the names of your different family members by clicking on their buttons.



- 3.1.5 The names of games are stored in an array. Open the **GameNames\_p** project in the 03 – Game Names Folder and do the following:

- Write code for the [Display] button to display the names of the games.
- Write code for the [New] button to add the text 'New' to every second element of the array starting from the first element.
- Write code for the [Select] button to randomly select a game and display the name of the game using a DialogBox.



- Write code for the [Reverse] button to reverse the characters of each element in the array. Display the array.



## BASIC MATHEMATICAL OPERATIONS

We can manipulate an array to perform mathematical calculations.

Let's find the maximum value and the position of the maximum value in an array.

```
...
{1} iMax := arrMarks[1];
{2} iPosition := 1;
{3} for i := 2 to 5 do
{4}   begin
{5}     if arrMarks[i] > iMax then
{6}       begin
{7}         iMax := arrMarks[i];
{8}         iPosition := i;
{9}       end;
{10}    end;
{11}    memDisplay.Lines.Add('The highest mark: '+ IntToStr(iMax) +' at
position '+ IntToStr(iPosition));
...
...
```

Assume that *arrMarks* contains the following values: 45, 90, 12, 40 and 72. Let's trace through the code segment above.

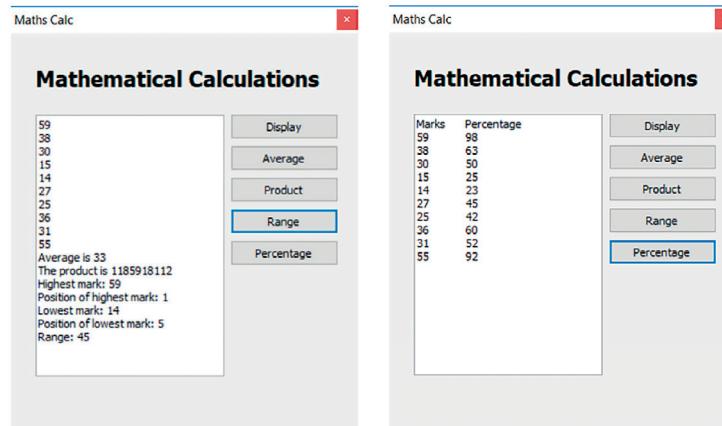
#	i	iMAX	iPOSITION	arrMARKS[i]	i <= 5	arrMARKS[i] > iMAX	OUTPUT
				arrMarks[1]:=45			
				arrMarks[2]:=90			
				arrMarks[3]:=12			
				arrMarks[4]:=40			
				arrMarks[5]:=72			
1		45					
2			1				
3	2				TRUE		
5						TRUE	
7		90					
8			2				
3	3				TRUE		
5						FALSE	
3	4				TRUE		
5						FALSE	
3	5				TRUE		
5						FALSE	
3	6				FALSE		
11							The highest mark is 90 at position 2



### Activity 3.2

The marks are stored in array *arrMarks*. Open the **MathsCalc\_p** project in the 03 – Math Calc Folder and write code for the following buttons:

- [Display] button to display the elements of the array.
- [Average] button to find and display the sum and average of the marks.
- [Product] button to find and display the product of the elements.
- [Range] button to find and display the highest and lowest mark and the position of the highest and lowest mark in the array and the range of the marks. The range is the difference between the highest and lowest mark.
- [Percentage] button to convert each mark to a percentage. The marks are out of 60 marks. Display the original mark and the rounded percentage calculated to zero decimal places.



## COUNTING ELEMENTS BASED ON A CRITERIA

When we need to determine *how many* elements there are in an array that meet a *specified criterion*, you can use a counting algorithm. Counting uses a linear approach where you loop through the array and test each item based on the criteria specified. Should the element meet the criteria, you increment the counter variable by one. Look at the following algorithm that shows how this works in practise:

### Algorithm

```
Count ← 0
For loop i ← 1 to length of array
    If array[i] matches criteria
        Count ← Count + 1
End FOR-loop
```



### Activity 3.3

**3.3.1** An array called *arrNumbers* has been declared. Open the **Counters\_p** project in the 03 – Counters Folder and write code for the following:

- [Even/Odd] button: Determine and display the number of even and odd numbers.
- [Negative/Positive] button: Determine and display the number of negative and positive numbers. *Remember that zero is neither negative nor positive.*



### Activity 3.3 *continued*

- c. [Composite] button: Determine how many numbers are composite. A *composite number* is a number that has more than two factors.
- All outputs must be accompanied by suitable messages.
- 3.3.2 An array called *arrNames* has been declared. Open the **ClassList\_p** project in the 03 – Class List Folder and write code for the [Find by letter] button to prompt the user to input a letter of the alphabet (use an *InputBox*). Count how many names in the given array *arrNames* begin with the letter provided by the user. Display the count value using a dialogue box.

## USING AN ARRAY AS A COUNTER

You can also use an array as a counter.

### Example 3.1

Your school plans a market day to raise funds. The schools will have 10 stalls selling different categories of items. They need to keep track of the popularity of each stall for future planning.

Since we are counting the popularity of 10 stalls, we would need to use 10 counter variables. This will become too cumbersome. A better solution would be to use an array as a counter where each stall is numbered from 1 to 10 and associated with each **index** position of an element in an array.

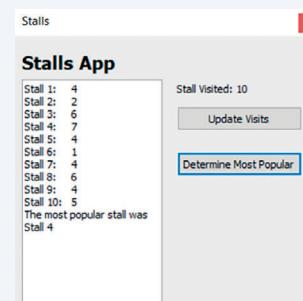
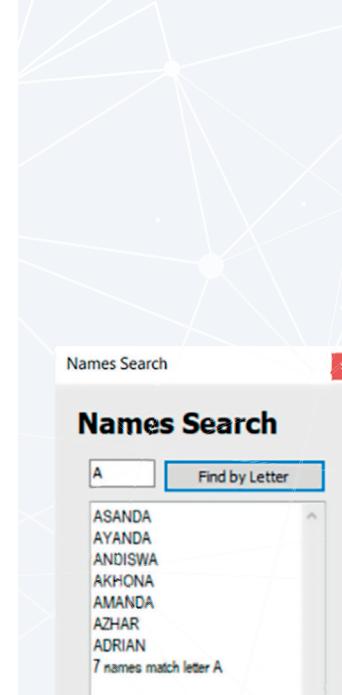
You need to set each element in the counter array to zero as shown below:

Index	1	2	3	4	5	6	7	8	9	10
arrPopCount	0	0	0	0	0	0	0	0	0	0

As customers visit a stall, the corresponding element matching the stall number is updated. If a customer visited stall number 3, the element at index position 3 is incremented by one.

Index	1	2	3	4	5	6	7	8	9	10
arrPopCount	0	0	1	0	0	0	0	0	0	0

To determine the most popular stall, you need to find the stall with the highest number of customers.



### Activity 3.4

- 3.4.1 Using the school market day example, open the **PopularStall\_p** project in the 03 – Popular Stall Folder and write code for the following:
- Declare an array *arrPopCount*.
  - [Update Visits] button: Every time this button is clicked, a random number in the range 1 to 10 is generated. This number represents the stall number that a customer has visited. The randomly generated number will be displayed on a label *lblStallNumber*. Keep count of the stalls visited using the array *arrPopCount*. Display the stalls visited as each stall is visited.
  - [Determine Most Popular] button: Determine and display the most popular stall.

## INSERTING AND DELETING ELEMENTS IN AN ARRAY

### INSERTING AN ELEMENT IN AN ARRAY

You can insert an element into a specific position in an array.

#### Example 3.2

Let's insert a value, 56, at the 4<sup>th</sup> position in the array below:

Index	1	2	3	4	5	6	7	8	9	10
arrHighScores	45	25	18	62	71	32	28	17	59	85



Remember, the number of elements in the array increases by one so the upper bound of the array must be increased by one.

All the elements from the 4<sup>th</sup> position must move to the right starting with moving the last element first:

```
arrHighScores[11] := arrHighScores[10];
arrHighScores[10] := arrHighScores[9];
...
arrHighScores[5] := arrHighScores[4];
The code for the movement of the elements:
for i := 10 downto 4 do
    arrHighScores[i + 1] := arrHighScores[i];
Alternatively:
for i := 11 downto 5 do
    arrHighScores[i] := arrHighScores[i-1];
```

You **cannot** start the movement of elements from the 4<sup>th</sup> element first, because it will overwrite the values on the right-hand side.

Now insert the value 56 at the 4<sup>th</sup> position: arrHighScores[4] := 56;

Index	1	2	3	4	5	6	7	8	9	10	11
arrHighScores	45	25	18	56	62	71	32	28	17	59	85

### DELETING AN ELEMENT FROM AN ARRAY

You can delete an element from a specific position in an array.

#### Example 3.3

Let's delete the value 18 from the 3<sup>rd</sup> position in the array below:

Index	1	2	3	4	5	6	7	8	9	10
arrHighScores	45	25	18	62	71	32	28	17	59	85

### Example 3.3 *continued*

All the elements from the 4<sup>th</sup> position must move to the left starting with the movement of the element at the 4<sup>th</sup> position first:

```
arrHighScores[3] := arrHighScores[4];
arrHighScores[4] := arrHighScores[5];
...
arrHighScores[9] := arrHighScores[10];
The code for the movement of the elements:
for i := 3 to 9 do
    arrHighScores[i] := arrHighScores[i + 1];
Alternatively:
for i := 4 to 10 do
    arrHighScores[i-1] := arrHighScores[i];
```

Although we moved the elements to the left, the last element still exists.

Index	1	2	3	4	5	6	7	8	9	10
arrHighScores	45	25	56	62	71	32	28	17	59	59

When we display, we must only display the first nine elements.

### REMOVING DUPLICATE ELEMENTS IN AN ARRAY

When we want to remove duplicate elements in an array, we use two arrays.

### Example 3.4

We want to remove duplicate names from an array *arrNames*. The array represented below is split in two lines due to space constraints.

Index	1	2	3	4	5	6	7	8	9	10
arrNames	Sanele	Sanele	Simon	Helen	Michele	Ulrich	Ulrich	Ulrich	Roland	Gerry

Index	11	12	13	14	15	16	17	18	19
arrNames	Henry	Ulrich	Sifiso	Roland	Gerry	Sue	Frankie	Andrew	Gerry

Follow the steps below:

- Use a second array *arrTemp* to hold the non-duplicate values.
- Declare array *arrTemp* to hold the same number of values as array *arrNames* to cater for the case where *arrNames* may not contain any duplicate values and hence all the values from *arrNames* will be copied to *arrTemp*.
- Loop through each element in *arrNames* and check whether it appears in *arrTemp*.
- If it does not appear in *arrTemp*, then the element is added to *arrTemp*.

Here is the code to remove duplicate values:

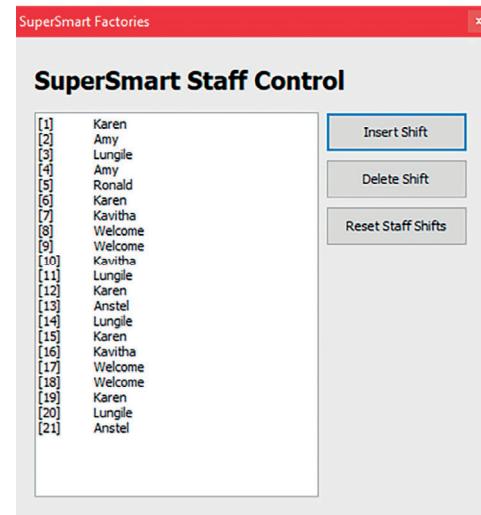
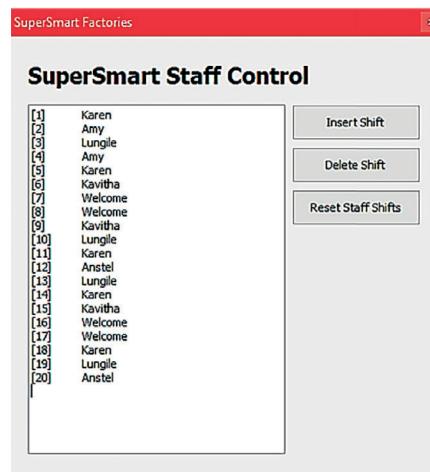
```
//The first element in arrTemp is set to first element in arrNames  
arrTemp[1] := arrNames[1];  
  
// iCounter will keep count of the number of elements in arrTemp  
//Since the first element is assigned to arrTemp, iCounter set to 1  
iCounter := 1;  
  
//Loop through each element in arrNames. Loop starts from 2 because the first  
//element is catered for  
for i := 2 to 19 do  
begin  
  
    //Boolean variable bFlag to check whether a duplicate match is found  
    //in arrTemp.  
    bFlag := true;  
  
    //Initialise counter value for inner loop to loop through the elements  
    //of arrTemp.  
    j := 1;  
  
    //The loop terminates either when a match is found or when the last  
    //element in arrTemp is reached  
    while (j <= iCounter) AND (bFlag = True) do  
begin  
  
    //Check whether the element in arrNames has a match in arrTemp  
    if arrNames[i] = arrTemp[j] then  
begin  
        // bFlag set to false as soon as a match is found in arrTemp  
        bFlag := false;  
end;  
    // Increment inner loop value  
    inc(j);  
end; // end inner loop  
  
    //If no match is found then increment the counter for arrTemp and  
    //store the value of the element in arrNames in the new counter  
    //position in arrTemp  
    if bFlag then  
begin  
    Inc(iCounter);  
    arrTemp[iCounter] := arrNames[i];  
end;  
end;  
  
// Copy the values from arrTemp into arrNames  
for i := 1 to iCounter do  
arrNames[i] := arrTemp[i];
```



### Activity 3.5

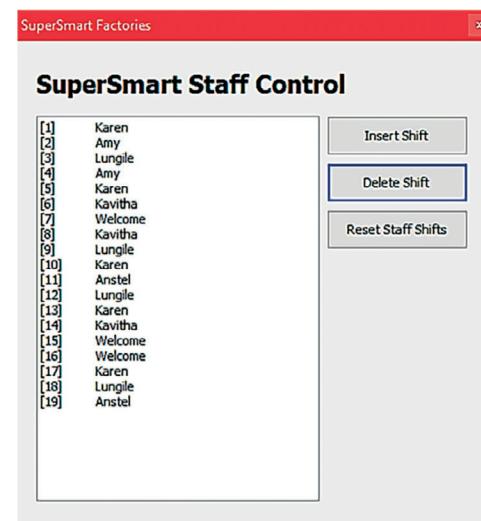
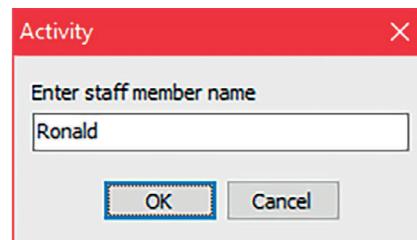
SuperSmart Factories requires a program to assist them with processing the placement of weekly shifts for staff members. Every time a staff member clocks-in, their name is added to an array named *arrStaff*. The maximum number of shifts is 50, however, the exact number of names present in the array is stored in the global variable *iCount*.

- 3.5.1** Open project **StaffShifts\_p** in the 03 – Staff Shifts Folder. The following interface is provided:



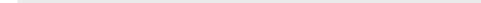
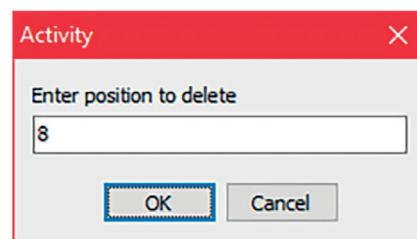
- 3.5.2** Write code for the [Insert Shift] button that will:

- Prompt the user to input the name of the staff member using an Input Box.
- Insert the new shift at position 5 in the array.
- Increment the value of *iCount* by 1.



- 3.5.3** Write code for the [Delete Shift] button that will:

- Prompt the user to input the position they wish to delete using an Input Box.
- Remove the specified value from the array.
- Decrease the value *iCount* by 1.





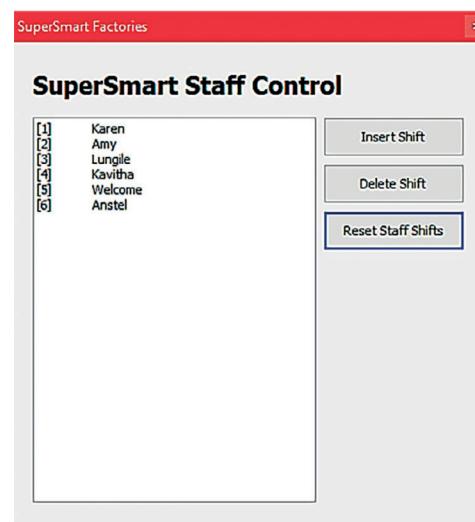
### Activity 3.5

*continued*

- 3.5.5** At the end of every week, all staff shifts are reset to a value of 1. This means that each staff member's name should appear in the array only once.

Write code for the [Reset Staff Shifts] button that will:

- Iterate through the array *arrStaff* and remove all duplicates.
- Modify the value of *iCount* to reflect the number of items remaining in the array after the duplicates have been removed.



## 3.2 Searching and sorting arrays

### SEARCHING ARRAYS

When working with large amounts of data, it is impossible to know the exact location of each element. So, if you want to search and locate specific elements in an array, you could use two methods:

- **linear** search
- **binary** search.

#### LINEAR SEARCH

When using a linear search, you can loop through the elements of the array to find the value that you are looking for. The loop will terminate when the:

- value is found
- last index position is reached and the value is not found.

To search for the first occurrence of a value in an array, you can use the following code:

```
i := 1;
iPos := 0;
bFlag := false;
sSearch := Input('Name', 'Enter name', '');
while (i <= 10) and (bFlag = false) do
begin
    inc(i);
    if arrNames[i] = sSearch then
begin
    bFlag := true;
    iPos := i;
end;
end;
if bFlag = true then
    memDisplay.Lines.Add('Found: ' + arrNames[iPos])
else
    memDisplay.Lines.Add(sSearch + ' not found');
```



#### New words

**linear search** – is a process that checks every element in the list sequentially until the desired element is found

**binary search** – is an algorithm used in computer science to locate a specified value (key) within an array

To search for all occurrences of a value in an array, you can use the following code:

```
...
Const
    arrNames:array[1..10] of string = ('Jack','Sanele','Alonso',
'Arhaan','Zinhle','Brian','Paul','Sarah','Akira','Zainab');
...
sSearch := Input('Name', 'Enter name', '');
bFlag := false;
for i := 1 to length(arrNames) do
begin
    if sSearch = arrNames[i] then
begin
    bFlag := true;
    memDisplay.Lines.Add(arrNames[i]);
end;
end;
if bFlag = false then
    memDisplay.Lines.Add('Name not found');
```



#### Take note

The **length** method of array returns the length of the array. The length method is used when the array is pre-populated.

## New words

**bubble sort** – to compare adjacent elements

**selection sort** – to select the element that should go in each array position either in ascending or descending order sequence

## SORTING ARRAYS

Arrays can be sorted in ascending or descending order. This year you will learn about two ways in which arrays can be sorted:

- **bubble** sort
- **selection** sort.

### BUBBLE SORT

The bubble sort compares adjacent elements. To sort elements in descending order, compare two adjacent elements. If the first element is less than the second element then swap the elements.

#### Pass 1 :

1	2	3	4	5	
arr	6	10	3	2	9

$\text{arr}[1] < \text{arr}[2]$  → true; swap elements

1	2	3	4	5	
arr	10	6	3	2	9

$\text{arr}[2] < \text{arr}[3]$  → false

$\text{arr}[3] < \text{arr}[4]$  → false

$\text{arr}[4] < \text{arr}[5]$  → true; swap elements

1	2	3	4	5	
arr	10	6	3	9	2

The smallest element “bubbles” to the last position

Because this is a descending sort, the smallest element “bubbles” to the last position in the array.

#### Pass 2 :

1	2	3	4	5	
arr	10	6	3	9	2

$\text{arr}[1] < \text{arr}[2]$  → false

$\text{arr}[2] < \text{arr}[3]$  → false

$\text{arr}[3] < \text{arr}[4]$  → true; swap elements

1	2	3	4	5	
arr	10	6	9	3	2

The second smallest element is stored in the second last position

### Pass 3 :

1	2	3	4	5	
arr	10	6	9	3	2

$\text{arr}[1] < \text{arr}[2]$  → false  
 $\text{arr}[2] < \text{arr}[3]$  → true; swap elements

1	2	3	4	5	
arr	10	9	6	3	2
↓					The third smallest element is stored in the third last position

### Pass 4 :

1	2	3	4	5	
arr	10	9	6	3	2

$\text{arr}[1] < \text{arr}[2]$  → false

The list is fully sorted in descending order.

The checks take place as follows:

PASS 1		PASS 2		PASS 3		PASS 4	
First Element Index	Second Element Index						
i	$i + 1$						
1	2	1	2	1	2	1	2
2	3	2	3	2	3		
3	4	3	4				
4	5						

You can write the bubble sort program using a nested FOR-loop. The outer loop is determined by the number of passes to be made. The inner loop always starts from index position 1 and ends with each pass at 5, 4, 3 and 2 consecutively.

Here is the code for a bubble sort using two nested FOR-loops:

```
....  
int [] arr = {5,9,2,1,8};  
int temp;  
for i:=arr.length-1 downto 1 do ←  
    for j:=0 to i do ↑  
        if (arr[j]<arr[j+1]) ←  
        Begin ←  
            temp=arr[j]; ←  
            arr[j]=arr[j+1]; ←  
            arr[j+1]=temp; ←  
        End;  
....
```

Outer loop

- Loop executed four times
- Loop starts at four and moves down to 1
- Downto loop because it will influence the inner loop's end value

Inner loop

- End value received from outer loop
- First time it will receive value 4
- Second time it will receive value 3
- Third time it will receive value 2
- Fourth time it will receive value 1

Adjacent elements compared

### BUBBLE SORT USING A FLAG

Although the bubble sort is a more efficient sort than the selection sort, the bubble sort program segment above is a fixed pass sort, that is, it will do all the passes even if the array is already sorted. The bubble sort with a flag is an improvement of the bubble sort. It is not a fixed pass sort. The order of the elements in the array influences the number of times the nested loop will be executed.

The bubble sort with a flag works as follows:

- the outer FOR-loop becomes a WHILE-loop
- a Boolean variable *bFlag* is set to true to **assume** that the array is already sorted. If any swapping takes place in the inner loop, then the array is still not sorted and Boolean variable *bFlag* is set to FALSE.

Here is the code for a bubble sort with a flag:

```
....  
repeat  
    bFlag := true; ←  
  
    for j := 1 to 9 do  
    begin  
        if arrNumbers[j] > arrNumbers[j + 1] then  
        begin  
            iTemp := arrNumbers[j];  
            arrNumbers[j] := arrNumbers[j + 1];  
            arrNumbers[j + 1] := iTemp;  
            bFlag := false;  
        end;  
    end;  
until bFlag = true;  
....
```

Boolean variable *bFlag*:

- Set to FALSE – indicates array not sorted
- Set to TRUE – indicates array is sorted

New words

**assume** – supposed to be the case, without proof

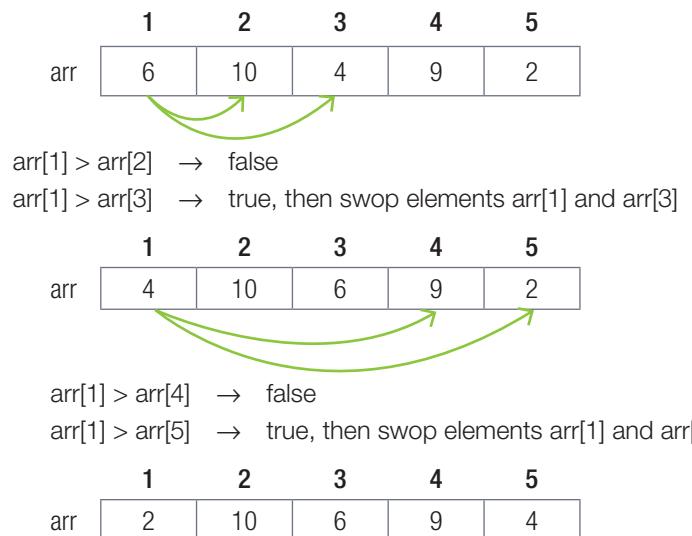
### SELECTION SORT

The selection sort is one of the simplest sorting algorithms. Although it is an easy sort to program, it is one of the least efficient. The algorithm offers no way to end the sort early, even if it begins with an already sorted list. It works by selecting the element that should go in each array position either in ascending or descending order sequence.

### Steps to sort an array in ascending order:

- **Step 1:** Compare the element at index position1 with each element in the array. If the element at index position1 is greater than the element it is being compared to, then the elements must be swapped.

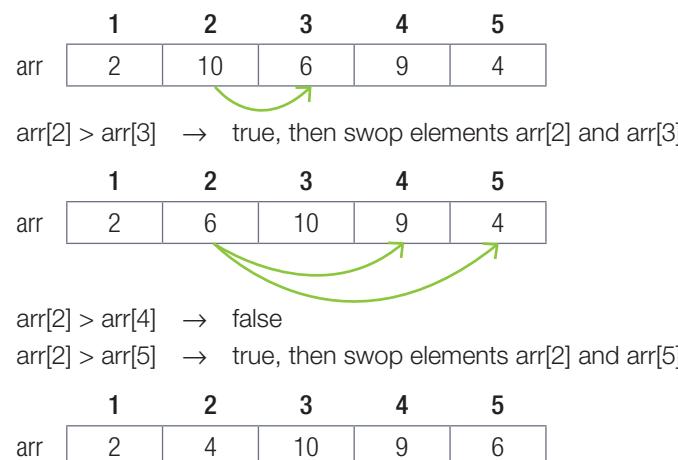
Example:



At the end of the comparisons for index position1, the smallest element is in index position1

- **Step 2:** Compare the element at index position2 with each element in the array after index position2. If the element at index2 is greater than the element it is being compared to, then the elements must be swapped.

Example:



At the end of the comparisons for index position2, the second smallest element is in index position2.

- **Step 3:** Continue in this way for all the locations of the array. In the last step only one comparison takes place. The second last element arr[4] will be compared to the last element arr[5].
- The selection sort will be implemented using nested loops.
- An element has to be kept constant until it is compared to every element to its right. The index position of the outer loop (variable i) is used to keep an element constant.

- The inner loop (variable  $j$ ) caters for the indices of the elements to the right of the constant element
- The comparisons take place as follows based on the index positions:

STEP 1		STEP 2		STEP 3		STEP 4	
Outer loop -i	Inner loop -j						
1	2	2	3	3	4	4	5
			4		5		
			5				

Here is the code for a selection sort:

```

...
// declare and initialize array elements
int [] arr = {5,9,2,8,1};
int temp;

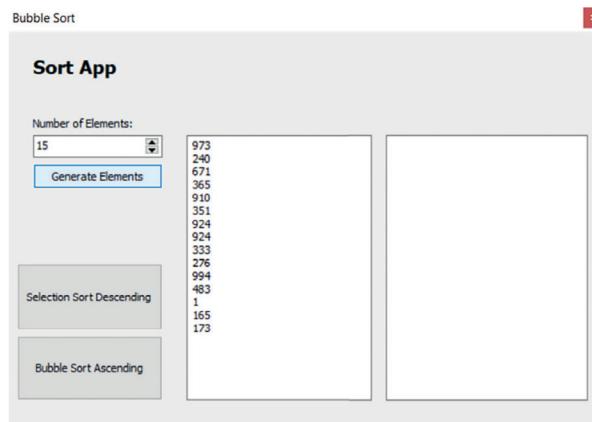
// outer loop holds index of element being compared
for i := 1 to arr.length-1 do
    // inner loop hold index of elements being compared to
    for j := i+1 to arr.length do
        // checks if the element is > the element being compared to
        if (arr[i]>arr[j])
            Begin
                // if true, swap elements
                temp=arr[i];
                arr[i]=arr[j];
                arr[j]=temp;
            End;

```



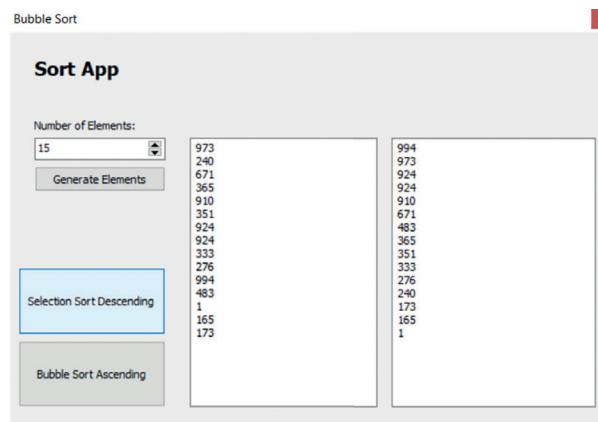
### Activity 3.6

- 3.6.1** Open **SortApp\_p** in the 03 – Sort Application folder. The following interface has been provided:

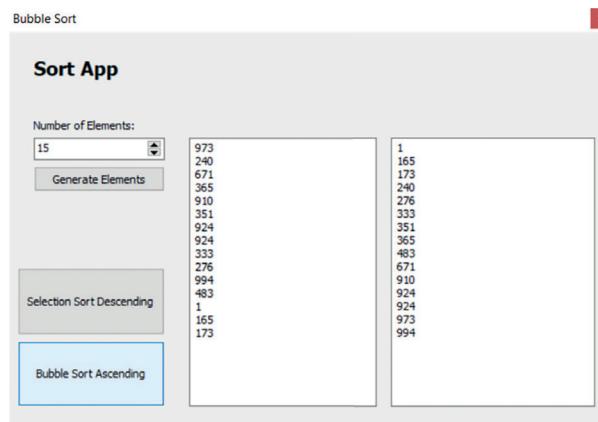


The code for the [Generate Elements] button has been provided that populates and displays the array with the number of elements specified in the *SpinEdit*. Note that the number of items in the array is stored in the global variable *items*.

- 3.6.2** Write code for the [Selection Sort Descending] button that will sort the array in Descending Order using the Selection Sort algorithm. Display the array in the *Memo Box memSorted*.



- 3.6.3** Write code for the [Bubble Sort Ascending] button that will sort the array in Ascending Order using the Bubble Sort algorithm. Display the array in the *Memo Box memSorted*.



## BINARY SEARCH

Whilst the linear search is effective with smaller arrays, larger arrays would result in very poor efficiency in terms of the number of tests that need to be made to search for a specific value. The search would start at the beginning of the array and terminate only once the specific value is found. So, if an array has 10 000 elements and the specific value is found in the second last position, the linear search will run up to 9 999 tests for the specific value. In the worst case scenario, all elements will be tested if the specific value is not found in the array. This means that if the array has 10 000 elements, then 10 000 tests will be made.

The binary search is a much more efficient method of locating a specific value in larger arrays. It searches for a specific value in a **sorted** array. The array can be sorted in ascending or descending order.

Let's assume that we want to search for the value 222 in an array. The binary search works as follows:

- **Step 1:** Ensure that the array is sorted.

Index	1	2	3	4	5	6	7	8	9	10
arrNumbers	85	99	158	159	180	199	201	222	248	301

- **Step 2:** We need to search within a range in the array. The range is given by a lower bound (start index position) and an upper bound (end index position). We need to find the mid-position of this **search range**.

- To find the mid-position of the search range:

mid-position ← (lowerBound + upperBound) div 2

- In the initial search range (index positions 1 to 10) the mid-position is 5. The element at position 5 is 180.



### New words

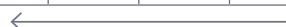
**sorted** – to sort an element in numerical order

- **Step 3: Test 1:** We test whether the element (180) at the mid-position is equal to the search value (222). If the test is TRUE, then the value is found and the search ends. In this case the test is FALSE.

- *Test 2:* If *test1* is FALSE, then we test whether the search value is less than the mid-position element. If the *test2* is TRUE, then it means:
    - the search value is on the left of the mid-position element and that the search range will now focus on these elements.
    - all elements from the mid-position element to the last element in the search range can be eliminated from the search.

- The upperBound is set to mid-position-1. The lowerBound is still 1:

Index	1	2	3	4	5	6	7	8	9	10
arrNumbers	85	99	158	159	180	199	201	222	248	301



- *Test 3:* If *test2* is FALSE, then we test whether the search value is greater than the mid-position element. If the *test3* is TRUE, then it means:
    - the search value is on the right of the mid-position element and that the search range will now focus on these elements.
    - all elements from the mid-position element to the *first element in the search range* can be eliminated from the search.

- The lowerBound is set to midposition +1

Index	1	2	3	4	5	6	7	8	9	10
arrNumbers	85	99	158	159	180	199	201	222	248	301

- Step 4:** Go back to step 2 until the search value is found **or** the start index value is greater than end index value.
- Look at the pseudocode below for this search:

#### PSEUDOCODE

```
// The array is sorted in ascending order
searchValue ← User Input
lowerBound ← 1
upperBound ← Length(array)
Found ← FALSE
While (Found = false) and (lowerBound <= upperBound)
begin
    Midpoint ← (Start + End) div 2
    If searchValue = array[Midpoint]
        Found ← TRUE
    Else
        If searchValue > array[Midpoint]
            lowerBound ← Midpoint + 1
        Else
            upperBound ← Midpoint - 1
end while
```

Search value is found

- The search value lies to the left of the midpoint when the array is sorted in ascending order
- The search value lies to the left of the midpoint when the array is sorted in ascending order

- The search value lies to the right of the midpoint when the array is sorted in ascending order
- The upper bound of the search range is set to one less than the midpoint



#### Activity 3.7

##### 3.7.1 Open the **BinarySearch\_p** project and write code for the following:

- [Generate] button: to generate 20 values in the range 10 to 99 and store the values in an array.
- [Sort] button: Sort the array in descending order.
- [Display] button: Display the elements in the array.
- [Search] button: Prompt the user to enter a number in the range 10 to 99. Determine whether the number is found in the array using the binary search. If the number is found, then display the number and its position in the array. If the number is not found in the array, an appropriate message must be displayed.

### 3.3 Parallel arrays

Remember that an array can only store elements of the same data type. However, if you want to store **related** information of different data types, you need different arrays. For example, if you want to store the names and marks of learners, then you need to store the names in one array and the marks in another array. But, the names and marks are related. Each element in one array is linked to an element in the second array by its index position. For example, in the arrays below, Peter's name is stored in *arrNames[3]* and his mark is stored in *arrMarks[3]*.

These linked arrays are called parallel arrays.

	1	2	3	4	5
arrNames	John	Mary	Peter	Sarah	Sanele
arrMarks	56	78	91	65	81

↓      ↓      ↓      ↓      ↓      ↓

#### SORTING PARALLEL ARRAYS

When you sort an array, elements are swapped. When you use parallel arrays and you want to sort, for example, the *arrMarks* array in descending order to create a merit list, the linked elements in the parallel array *arrNames* must be swapped simultaneously.

Look at the code below that shows how elements in parallel arrays can be swapped:

```

...
for i := 1 to 4 do
begin
    for j := i + 1 to 5 do
    begin
        if arrMarks[i] < arrMarks[j] then
        begin
            iTemp := arrMarks[i];
            arrMarks[i] := arrMarks[j];
            arrMarks[j] := iTemp;
            sTemp := arrNames[i];
            arrNames[i] := arrNames[j];
            arrNames[j] := sTemp;
        end;
    end;
end;
...

```



#### New words

**related information** –  
information belonging in  
the same group

#### SEARCHING PARALLEL ARRAYS

To search for a specific value in one array and access the corresponding element in the linked array, you can search for the specific value in the one array and if a match is found, use the index position of the match to access the element from the linked array.

Look at the code below that shows how elements in parallel arrays can be searched:

```
i := 1;
iPos := 0;
bFlag := false;
sSearch := Input('Name', 'Enter name', '');
while (i <= 10) and (bFlag = false) do
begin
    if arrNames[i] = sSearch then
begin
    bFlag := true;
    iPos := i;
end;
inc(i);
end;
if bFlag = true then // or if iPos > 0 then
    memDisplay.Lines.Add('Found: ' + arrMarks[iPos])
else
    memDisplay.Lines.Add(sSearch + ' not found');
```

Index position of element = sSearch



### Activity 3.8

Open project **SearchingSorting\_p** that provides the following user interface:

Names:	Assignment Marks:	Exam Marks:
ANGELA	64	97
DEEPIKA	97	90
MIKASHA	79	90
MASEEHA	94	99
SIAN	92	70
SHALEENA	92	62
LUNGELO	94	77
ADHIKAR	88	69
KHULASANDE	63	90
NDUMISO	61	98
YOSHODA	65	84
ERICA	60	86
PHILILE	92	72
NOXOLO	66	84
PEARL	64	75
SINENHLANHLA	74	72
ASHVEER	97	82
VINISHA	100	82
NIKHIL	77	76
PRASHANTHI	61	86



### Take note

The exam and assignment arrays are randomly generated and therefore will not match the screenshot above.

The [Load Data] button given, has been coded to populate four parallel arrays:

- *arrNames*: An array of type String that holds 20 learners' names.
- *arrAssignment*: A parallel array of type Integer that holds 20 learners' assignment marks.
- *arrExam*: A parallel array of type Integer that holds 20 learners' exam marks.
- *arrAverages*: A parallel array of type Real that will be populated during the activity.

Write code for each of the buttons (1–9) to perform the following tasks:

- [1] button: Display the Assignment Mark and Exam Mark for 'Shaleena' (Element 6).
- [2] button: Calculate and display the average Assignment Mark.
- [3] button: Determine and display the name of the learner with the lowest Exam Mark.



### Activity 3.8

*continued*

- [4] button: Calculate the average of each learner's score [(Exam Mark + Assignment Mark)/2] and store the calculated value in its corresponding position in *arrAverages*. Display a confirmation message that the averages were calculated successfully.
- [5] button: Display a list of learner names who have scored above the average exam mark. Display a heading displaying the average exam mark followed by the list of names.
- [6] button: Determine and display the learner with the highest exam mark.
- [7] button: Prompt the user to input a Name to search for using an Input Box. Use the Linear Search to identify the name. If the name is found, display the learner's Assignment, Exam and Average marks. If the name is not found, display a suitable message to the user.
- [8] button: Use either the Bubble Sort or Selection Sort to sort the arrays based on the learners' names (*arrNames*). The names should be sorted alphabetically and the display should be updated in the three MemoBoxes.
- [9] button: Prompt the user to input a Name to search for using an Input Box. Use the Binary Search to identify the name. If the name is found, display the learner's Assignment, Exam and Average marks. If the name is not found, display a suitable message to the user.

**QUESTION 1:** Paper-based activity

You have three parallel arrays containing data, as shown below.

There are 400 seats in total in Parliament. The number of seats each party gets is determined by their ratio of votes.

arrCandidates	arrVotes	arrSeats	
Digital Party	11750		
Data Party	98115		
USB Party	11391		
Android Party	49415		

- 1.1 Determine the total votes cast by calculating the sum of the values in *arrVotes*. Store the sum in a global variable called *iTotal*.
- 1.2 Use *iTotal* (from 1.1) to determine the number of seats each party will get. The number of seats depends on the ratio of votes that a particular party received. Store the number of seats for each party in *arrSeats*. (Round your answer to an Integer value).
- 1.3 Write code to determine which party received the highest number of votes. Display the Party name (from *arrCandidates*) and the number of votes (from *arrVotes*).
- 1.4 Create a sorting algorithm and flowchart that can be used to sort *arrVotes* in descending order.
- 1.5 Create code that can be used to sort all the arrays based on the number of seats in Parliament; arranged in ascending order.

**QUESTION 2**

For this application, open the project saved in the 03 – Question 2 – Password Strength Folder. Once done, save the project in the same folder. For this question, an array called *arrPasswords* has been declared and populated in the [Generate 20 Random Passwords] button.

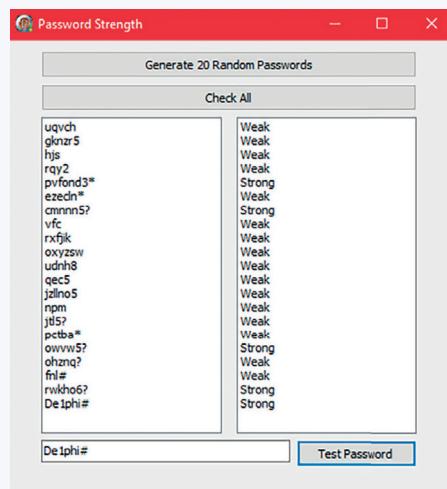
- 2.1 Write code for the [Test Password] button that reads the password from the EditBox *edtPassword* and determines whether the password is strong or weak. A strong password needs to meet all the following criteria:
  - The length of the password is a minimum of 6 characters.
  - The password contains at least one digit.
  - The password contains at least one letter.
  - The password contains at least one of the following special characters: \*, #, ?, \$.

## CONSOLIDATION ACTIVITY

### Chapter 3: Arrays *continued*

Otherwise, the password is considered to be weak.

- 2.2 Add the password, as well as the strength of the password, to the ListBoxes as shown below.



- 2.3 When the [Check All] button is pressed, use a FOR-loop to check the strength of all the passwords in the array.  
2.4 Record the strength of all passwords in a parallel array named *arrStrength*.  
2.5 Display all the passwords, as well as the strength of the passwords, as shown above.

## QUESTION 3

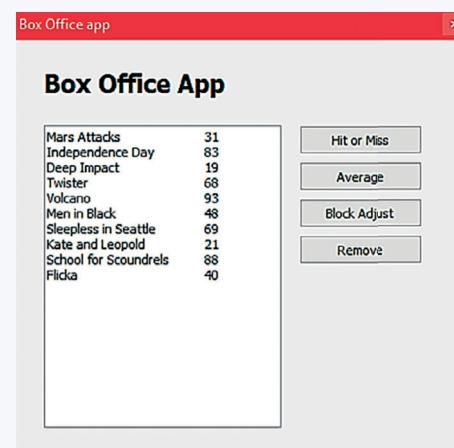
Open project **BoxOffice\_p** in the 03 – Question 3 – Box Office Folder that provides the following interface:

Two parallel arrays have been declared and initialised:

- *arrMovies* – an array of type String containing the names of movies.
- *arrTickets* – a parallel array of type Integer containing the number of tickets sold for each movie in *arrMovies*.

The variable *items* tracks the number of items in the Array.

- 3.1 When the [Hit or Miss] button is pressed: Determine and display which movie has sold the greatest number of tickets. At the same time, determine and display which movie has sold the least number of tickets.  
3.2 When the [Average] button is pressed: Determine and display the average ticket sales across all the movies being shown.  
3.3 A businessman has sponsored 100 tickets per movie to a local school. Write code for the [Block Adjust] button, which will increase the ticket sales of all movies by 100.  
3.4 Movies have a limited run on circuit. This means that eventually, it is removed from cinema listings and is released to home video or streaming services. Write code for the [Remove] button which will prompt the user to input the name of a movie. Locate the movie in the movies array and remove its entries from both arrays.



# STRING AND DATE MANIPULATION

CHAPTER  
**4**

## CHAPTER UNITS



- Unit 4.1 Built-in string methods
- Unit 4.2 Delimited strings
- Unit 4.3 Built-in Date-Time methods



## Learning outcomes

At the end of this chapter you should be able to

- describe the following concepts: method-call, method-overload and method signature
- use the following built-in string methods: Pos, Copy, Insert and Delete
- read delimited strings
- extract data from delimited strings
- store the data from delimited strings in an array
- use Date methods.

## INTRODUCTION

In Grade 10, you learned how to manipulate strings by using FOR-loops to access each character in the string. By doing this, you were able to:

- find a character in the string
- replace a character in a string
- delete a character from a string
- insert a character into a string.

In this chapter, you will learn about built-in string methods that allow you to do the same thing without having to use a FOR-loop. You will also learn how to use these methods to manipulate individual words and phrases from strings. We will also focus on the use of Date methods.

## 4.1 Built-in string methods



### Take note

If you hover your cursor over a method in Delphi, it will display the unit the method belongs to.



### New words

**concatenates** – to joins strings together into one result string

**method overloading** – to have more than one method with the same name

**method signature** – is the number of arguments and their data type

In Chapter 1, you worked with built-in mathematical methods. These built-in methods are made accessible to programmers through modules that we call units. In this section, you will learn more about string methods that allow you to manipulate and work with strings. In Grade 10 you used the following methods IntToStr(), StrToInt(), StrToFloat(), FloatToStr, Input Box() and Length().

### LENGTH() FUNCTION

In Grade 10, you learnt that the length functions returns the number of characters in a string. For example:

```
iCount := length('It's hot today') // the value of
// iCount will be
// 14
```

### CONCAT() FUNCTION

The CONCAT-function **concatenates** (joins) strings (String1, String2 ...) together into one result string. It is equivalent to the + operator. Thus far you have been using the + operator to join strings, for example, sSentence:=sSentence+sWord

Syntax: **CONCAT(*sString1, sString2, ...*)**

- *sString1*: refers to the first string to be joined.
- *sString2*: refers to the second string to be joined.
- ...: indicates more strings can be joined.

Here are some examples:

```
sMsg := 'it''s a good day';
sMsg := sMsg +' for fishing'; //note the space at the
beginning of the string ' for fishing'
OR
sMsg := Concat(sMsg,' for fishing');
sCities := Concat('Paris','London','New York'); //will
join all the strings without spaces //between the strings
```

### COPY() FUNCTION

The COPY-function returns a copy of a certain number of characters from a string. The result is a string that is part of the original string, which we refer to as a substring.

Syntax 1: **COPY(*sString, iStart, iNumCharacters*)**

- *sString*: the string you are copying from.
- *iStart*: the position to start copying from.
- *iNumCharacters*: the number of characters to copy.

A substring (part of a string) will be copied from string *sString*, starting at position *iStart*, *iNumCharacters* number of characters will be returned.

**Example:**

```
sResult := Copy('PAM is cool',5,2) - will result in sResult := 'is'
```

Syntax 2: `Copy(sString, iStart)`

- *sString*: the string you are copying from.
- *iStart*: the position to start copying from to the end of the string.

**Example:**

```
sResult := (copy('PAM is cool',8) - will result in sResult := 'cool'
```

## METHOD-OVERLOADING AND METHOD-SIGNATURE

We can see from the discussion above that we can call the COPY-function using different arguments:

- The first COPY-function's method signature is its argument (sString, iStart, iNumCharacters).
- The second COPY-function's method signature is the arguments (sString, iStart).

We can have more than one method with the same name. This is called **method-overloading**.

Delphi differentiates between the two COPY-functions using a **method signature**. The method signatures refers to the number of arguments and their data types. Look at Table 4.1 below for some examples.

**Table 4.1: Examples of method signatures**

EXAMPLES	OUTCOME
<pre>sWord := 'Hamburger'; sAbrev := Copy(sWord, 1, 1);</pre>	<code>sAbrev := 'H'</code>
<pre>sWord := 'Hamburger'; sAbrev := Copy(sWord, 1, 3);</pre>	<code>sAbrev := 'Ham'</code>
<pre>sWord := 'Hamburger'; sAbrev := Copy(sWord, 20, 1);</pre>	<code>sAbrev := ''</code> '' denotes empty string
<pre>var   sName : String;   cInitial : char;    sName := 'James';   cInitial := Copy(sName, 1, 1);   //copy one character</pre>	Type mismatch error. The result of the Copy function is a string and cannot be stored in a char variable  <b>Note:</b> To store a single character using a copy function we use: <code>cInitial := Copy(sName, 1, 1)[1] ;</code> then <code>cInitial := 'J'</code>
<pre>sName := ' Nelson Mandela' iLen := length(sName) sLast3 := copy(sName, iLen - 2)</pre>	<code>sLast3 := 'ela'</code>



## Activity 4.1

**4.1.1** Study the declaration below and determine the output in the table provided:

```
Var sName, sSurname, sID : string;
    cLetter : char;
Begin
    sName := 'Nathaniel';
    sSurname := 'Khune'
    sID := '9607225212087'
End;
```

	STATEMENT	OUTPUT OF lblOutput.Caption
1	lblOutput.Caption := Copy(sName,5,1)	
2	lblOutput.Caption := Copy(sName,1,0)	
3	lblOutput.Caption := Copy(sName,20,2)	
4	lblOutput.Caption := Copy(sName,6,4)	
5	lblOutput.Caption := Copy(sName,1,1) + ' ' + Copy(sSurname,1,5)	
6	lblOutput.Caption := Copy(sSurname,1,Length(sSurname))	
7	lblOutput.Caption := 'Day: ' + Copy(sID,5,2)	
8	lblOutput.Caption := 'Year: 19' + Copy(sID,1,2)	
9	lblOutput.Caption := IntToStr(length(Copy(sName,3,5)));	
10	lblOutput.Caption := IntToStr(length(Copy(sName,1,8)));	

## POS() FUNCTION

The **Pos** function returns the start position of one string within another string as an integer. For example, the start position of ‘mark’ in ‘fish market’ is 6.

Syntax: **Pos(sSubstring, sString);**

- **sSubstring:** string to be found.
- **sString:** string in which to look for sSubstring.

### Note:

- sSubstring can be a character or a string.
  - Pos looks for an exact string and is case sensitive.
  - Pos returns the start position of the substring sSubstring in the string sString.
- Examples:
- iPos := Pos('at','Leave at 10') will result in iPos := 7.
  - iPos := Pos('v','Leave at 10') will result in iPos := 4.
- If there are many occurrences of the substring sSubstring in the string sString, then the start position of the first occurrence of the substring is returned.



### New words

**Pos** – to return to the start position of one string within another string as an integer

### Example:

```
iPos := Pos('a','Leave at 10') will result in iPos := 3
```

- The start position of the first character of the substring `sSubstring` is returned.
- If the substring `sSubstring` is not found within `sString` then it returns a value 0

**Example:**

`iPos := Pos('@', 'Leave at 10') will result in iPos := 0;`

**Table 4.2: Other examples**

STATEMENTS	VALUE OF iPLACE
<pre>var s: String;     iPlace : Integer; ... s := 'DELPHI PROGRAMMING'; iPlace := Pos('HI', s);</pre>	5
<pre>sWord := 'School'; iPlace := Pos('o',sWord);</pre>	4
<pre>sWord := 'School'; iPlace := Pos('oo',sWord);</pre>	4
<pre>sWord := 'School'; iPlace := Pos('s',sWord);</pre>	0 - because Pos is case sensitive



### Activity 4.2

**4.2.1** Study the declaration below and determine the value of `iAns`. Complete the table below.

```
Var sName, sSub: string;
    cLetter : char;
Begin
    sName := 'Information Technology';
    sSub := 'Tech';
    cLetter := 'I';
End;
```

NO	STATEMENT	iANS
a.	<code>iAns := Pos('n', sName);</code>	
b.	<code>iAns := Pos(cLetter, sName);</code>	
c.	<code>iAns := Pos(sSub, sName);</code>	
d.	<code>iAns := Pos('man', sName);</code>	
e.	<code>iAns := Pos(sName, sSub);</code>	

**4.2.2** Study the Delphi code segment below and answer the questions that follow:

```
Var sFullNames, sName, sSurname : string;  
    iPos : Integer;  
.....  
    sFullNames := edtFullNames.Text;  
    iPos := Pos(' ', sFullNames) //Position of space  
.....
```

Assume that the string 'Ariana Grande' is read for the *sFullNames* variable using the *edtFullNames* EditBox:

- a. Write a Delphi statement that will extract the first name from *sFullNames* and store the name in *sName*. This code should work for any full name read.
- b. Write a Delphi statement that will extract the surname from *sFullNames* and store the surname in *sSurname*. This code should work for any full name read.

## SETLENGTH() PROCEDURE

The **SETLENGTH** procedure changes the size of a string.

Syntax: `setLength(sString, iNewLength);`

- **sString:** the string whose size will change.
- **iNewLength:** the new size of *sString*.



### New words

**SETLENGTH** – to change the size of a string

When changing the size of a string *sString*, the new size *iNewLength* may be smaller, the same size or larger than the existing string length. In all cases, the size is in characters, and a new string is created regardless.

#### Note:

- If the new size of the string is smaller than the original size of the string, the string gets truncated, that means that if the string contains 15 characters and the new size is 10, then the last 5 characters are 'chopped' off.

#### Example:

```
sString := 'Change is the only constant';  
setLength(sString, 13);
```

After the statements above are executed, the value of *sString* will be:

'Change is the'

- If the new size of the string is larger than the original length of the string, then space for extra characters are added but these extra characters are **not initialised**. This can create odd effects.

#### Example:

```
sString := 'Plenty';  
setLength(sString, 15)  
memDisplay.Lines.Add(sString);  
sStr := 'Plenty';  
setLength(sStr, 10);  
memDisplay.Lines.Add(sStr);
```

After the statements above are executed, the results returned could be:

PlentyFormat  
Plenty

We find odd effects in the output. This is because the additional characters are not initialised.



### Activity 4.3

4.3.1 Given code:

```
sText := 'Creative people will benefit most from changes in technology';
setLength(sText, 15);
sMessage := 'Hacking is';
setLength(sMessage, 14);
```

What will be the values *sText* and *sMessage* after the statements are executed?

4.3.2 Given code:

```
Line 1: sText := 'abcdefg';
Line 2: setLength(sText, 15);
```

- Write code to initialise the extra places created in line 2 with a '-' character.
- Why is this initialisation of the extra space so important?

## INSERT() PROCEDURE

The **Insert** procedure inserts one string into another string. Unlike functions, the INSERT procedure changes the string, that is, after the procedure is called, the previous value of the string is permanently changed.

Syntax: **Insert(*sSubstring*, *sString*, *Position*)**

- sSubstring:** string to be inserted.
- sString:** string in which *sSubstring* will be inserted.
- Position:** integer position where *sSubstring* is to be inserted.



### New words

**Insert** – to insert one string into another string

#### Note:

- Any characters to the right of the insertion position will be moved to the right.
- The length of the string will increase by the number of characters inserted.

#### Example:

Study the Delphi code segment below.

```
...
sPhrase := 'IT is my favorite subject';
Insert('really ', sPhrase, 7);
Insert('really ', sPhrase, 6);
Insert('really ',' IT is my favorite subject', 6);
...
```

The outcome after each statement in the code segment is executed.

STATEMENTS	OUTCOME
sPhrase := 'IT is my favourite subject';	sPhrase := 'IT is my favourite subject';
Insert('really ', sPhrase, 7);	sPhrase := 'IT is really my favourite subject'
Insert('really ', sPhrase, 6);	sPhrase := 'IT is really my favourite subject'
Insert('really ', IT is my favourite subject', 6);	Gives an error. The second argument must be a variable so that a value can be assigned to it



### Activity 4.4

Study the code below and then complete the table.

```
Var sName, sSub: string;
    cLetter : char;
Begin
    sPhrase := 'Hard work pays off'; //18 characters
    sSub := 'Very';
    cLetter := 'f';
End;
```

	STATEMENT	OUTPUT
1	Insert(sSub,sPhrase,1); lblOutput.Caption := sPhrase;	
2	Insert(sSub + ' ',sPhrase,1); lblOutput.Caption := sPhrase;	
3	Insert(cLetter,sPhrase,Length(sPhrase)); lblOutput.Caption := sPhrase;	
4	Insert('thorough ',sPhrase,Pos(' ', sPhrase) + 1); lblOutput.Caption := sPhrase;	
5	Insert('Only ',sPhrase,Pos(',', sPhrase) + 1); lblOutput.Caption := sPhrase;	

## DELETE() PROCEDURE

The **Delete** procedure deletes a number of characters *iNumOfCharacters* from a string *sString* starting from a start position *iStartPosition*.

Syntax: Delete(**sString**, **iStartPosition**, **iNumofCharacters**)

- **sString:** String from which characters will be deleted.
- **iStartPosition:** Integer position of first character to be deleted.
- **iNumofCharacters:** Number of characters to be deleted.



### New words

**Delete** – to delete a number of characters from a string starting from a start position

**Note:**

- Any characters to the right of the deleted characters will be moved to the left.
- The length of the string will decrease by the number of characters deleted.

**Example:**

```

...
sPhrase := 'IT is not my most favourite subject'
Delete(sPhrase, 7, 4);
Delete(sPhrase, 10, 5);
Delete(sPhrase, 1, 100);
Delete('I love IT', 1, 1);
...

```

The outcome after the Delete statements are executed one after the other:

STATEMENT	OUTCOME
Delete(sPhrase, 7, 4);	sPhrase := 'IT is my most favourite subject'
Delete(sPhrase, 10, 5);	sPhrase := 'IT is my favourite subject'
Delete(sPhrase, 1, 100);	sPhrase := " " //empty string
Delete('I love IT', 1, 1);	Gives an error. First argument must be a string variable to be assigned a value, not a constant string

**Activity 4.5**

Given: sValue:= 'Johannesburg'

Write down independent Delphi statements to do the following:

- 4.5.1 Determine the length of sValue.
- 4.5.2 Remove the string 'hannes' from sValue.
- 4.5.3 Find the position of 'ann' in sValue.
- 4.5.4 Return characters 3 to 5 from sValue.
- 4.5.5 Insert the string 'hannes' after the letters 'Jo' in sValue.

**Take note**

If the method has a return value, store this value in a variable.

**Guided activity 4.1****String functions**

Open the **StringProblems\_p** project in the 04 – String Problems Folder. You should see the following user interface when you run the project.



### Guided activity 4.1

#### String functions *continued*

Create OnClick events for buttons: **Question 1**, **Question 2**, **Question 3**, **Question 4** and **Question 5** as follows:

- **Question 1 button**

Determines whether the input text contains a comma or not. Displays an appropriate message in the MemoBox indicating whether the comma was found or not. If the comma was found, display the position of the comma in the input string.

```
sInput := edtInput.Text;
iPos := Pos(',', sInput);
if iPos > 0 then
    memOutput.Text := 'Comma found at position ' + IntToStr(iPos)
else
    memOutput.Text := 'Comma not found';
```

**Notes:**

- The POS function is used to determine the position of the comma in the input string.
- If the value returned by the POS function in *iPos* is equal to 0, then a comma is not found in the input string; otherwise the comma is found in position *iPos*.

- **Question 2 button**

Deletes any characters in the input text up to (and including) the first space. If there is no space in the text, no characters are deleted. Display the changed input text in the MemoBox.

```
sInput := edtInput.Text;
iPos := Pos(' ', sInput);
Delete(sInput, 1, iPos);
memOutput.Text := sInput;
```

**Notes:**

- Pos function is used to determine the position of the first space.
- All characters can then be deleted from the start of the text to the position of the space.
- If no space is found, Pos will return a value of 0 and no characters will be deleted.

- **Question 3 button**

Move the first three characters of the input text to the end of the input text. Display the changed input text in the MemoBox.

```
sInput := edtInput.Text;
sCharacters := Copy(sInput, 1, 3);
Delete(sInput, 1, 3);
sInput := sInput + sCharacters;
memOutput.Text := sInput;
```

**Notes:**

- The first three characters are copied to a new string *sCharacters* before being deleted from the input string *sInput*.
- After the first three characters are deleted from *sInput*, they can be added to the end of *sInput*.
- *sInput* is displayed in the memo component.

- **Question 4 button**

Deletes all copies of the lowercase character 's' from the input text. Display the changed input string in the Memo box.



## Guided activity 4.1

### String functions *continued*

```
sInput := edtInput.Text;
iPos := Pos('s', sInput);
while iPos <> 0 do
begin
  Delete(sInput, iPos, 1);
  iPos := Pos('s', sInput);
end;
memOutput.Text := sInput;
```

#### Notes:

- The position of the first 's' is determined using the POS function and stored in *iPos*.
- While *iPos* <> 0, delete the 's' from *sInput* and find the next 's' in *sInput*. Continue this process while *iPos* <> 0.
- By using a WHILE-DO-loop, you can continue searching through your input text *sInput* and deleting all 's' characters until no 's' characters are found.

#### • Question 5 button

Inserts a space after every character of the text. Display the changed input string in the MemoBox.

```
sInput := edtInput.Text;
i := 1;
while i <= Length(sInput) do
begin
  i := i + 1;
  Insert(' ', sInput, i);
  i := i + 1;
end;
memOutput.Text := sInput;
```

#### Notes:

- You can use a WHILE-loop with the Insert command to add a space after each character.
- With this code, you need to make sure that your loop does not add a space after the space it previously added. This will cause your program to go into an infinite loop where each iteration of the loop simply adds another space after your previously added spaces.
- Therefore, in the code above, the value of 'I' is incremented twice. The first time it increases it selects the next character, while the second increase moves 'I' past the newly added space.
- Save and run your project.



## Activity 4.6

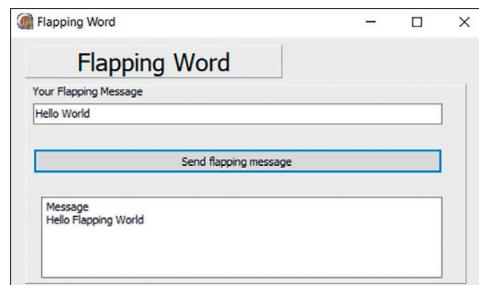
- 4.6.1** To make messages more interesting, the word 'flapping' must be added between the first two words of a message. Open the **FlappingWord\_p** project in 04 – Flapping Word Folder and do the following:





## Activity 4.6 *continued*

- Create an OnClick event for the [Send Flapping Message] button to read a message from the EditBox
- Add the word ‘flapping’ in between the first two words of a message.
- Display the message in the MemoBox.
- Save and run the project.



**4.6.2** Coming up with interesting, new compliments in a relationship can be challenging, especially after you have been in a relationship for a long time. To solve this problem, you have decided to create the Love Letters application that will randomly generate new compliments whenever you click on a button. Open the **LoveLetters\_p** project in the 04 – Love Letters Folder.



When you click on the [Generate compliment] button, the application should do the following:

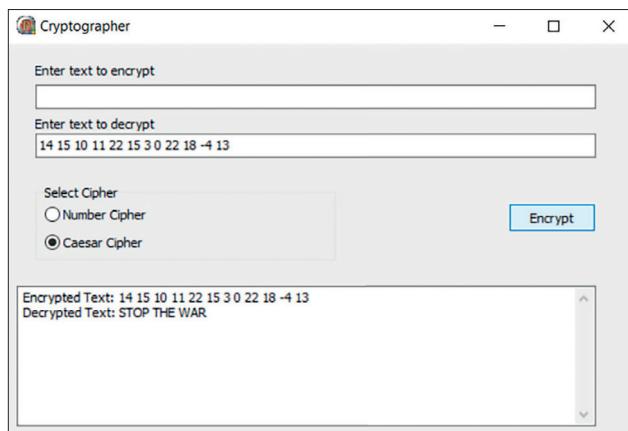
- Store several greetings, pet names and descriptions in three arrays. These arrays have already been created for you.
- Create a random compliment in the following format: <GREETING>, <PETNAME> you look <DESCRIPTION> today! <GREETING>, <PETNAME> and <DESCRIPTION> are randomly selected from their respective arrays.
- Using the string functions, replace the <GREETING>, <PETNAME> and <DESCRIPTION> keywords in your compliment with the randomly selected phrases from the arrays.  
Display your automatically generated compliment.  
The message below shows an example of a simple compliment that could be used in your application.  
<GREETING> <PETNAME>, you look <DESCRIPTION> today!
- By replacing the keywords, you might end up with one of the following compliments:
  - Hi Love, you look stunning today!
  - Hey Gorgeous, you look amazing today!





## Activity 4.6 *continued*

- 4.6.3** Open the **Cryptographer\_p** project in the 04 – Cryptographer Folder and add code to the OnClick event for the [Encrypt] button to do the following:



An encrypted message was created in the following manner:

- a character's logical numerical position was obtained from the character list below:  
`sCharacterList := 'ABCDEFGHIJKLMNOPQRSTUVWXYZ 0123456789.,!?:-+=()"[]';`  
Example: The logical numerical position of 'A' is 1, 'B' is 2 and so on.
- The character's numerical position (obtained in bullet 1) was incremented by 5.
- If the radio button Caesar Cipher is selected then decrypt and display the message.
- Save and run your project.

## CHARACTERS AND ASCII CODES

In Grade 10 you learned about ASCII codes. Remember that a character is any letter, number, space, punctuation mark or symbol. Each character is represented by an ASCII code. ASCII code stands for American Standard Code for Information Interchange. ASCII coding is a standard way to represent characters using numeric codes. See the characters and their corresponding ASCII code on the ASCII table in Annexure A.

Here are some examples:

- The ASCII code for 'A' is 65
- The ASCII code for 'a' is 97
- The ASCII code for '?' is 63

### ORD() FUNCTION

The **ORD** function returns the ordinal value (ASCII) of a character.

Syntax: **Ord(cChar)**

- **cChar:** represents a character.

**Examples:**

```
iNum := ord('B');           // will return the ordinal value 66
iNum := ord('z');           // will return the ordinal value 122
```



### New words

**ORD** – to return the ordinal value of a character

**CHR** – to return the corresponding character of an ASCII code

**VAL** – to convert a string to a numeric value

### CHR() FUNCTION

The **CHR** function returns the corresponding character of an ASCII code.

Syntax: **Chr(iNum)**

- **iNum:** represents an ordinal number of a character.

**Examples:**

```
cChar := chr(66);           // will return the character 'A'
cChar := chr(122);           // will return the character 'z'
```

### VAL() PROCEDURE

The **VAL** procedure converts a string to a numeric value.

Syntax: **Val(sString, Result, iCode);**

- **sString:** represents the string that must be converted.
- **Result:** represents the value that must be returned and can be either an integer or real number.
- **iCode:** Stores an integer value based on the success of the code conversion. If the conversion is successful, *iCode* is 0. If the conversion is unsuccessful, *iCode* stores the position where an error first occurred in the conversion.

**Examples:**

```
Val('4450', result, iCode);
```

The string '4450' will be converted to 4450 and stored in *result*. *iCode* will hold the value 0 because the conversion was successful.

```
Val('44A0', result, icode);
```

The string '44A0', will not convert successfully to an integer value because of the non-numeric character 'A'. *Result* store the value 44 and *iCode* will store the value 3.

## STR() PROCEDURE

The **STR** procedure converts an integer or real number into a string, with optional basic formatting.

Syntax: **Str(*Number, sString*)**;

- **Number:** represents an integer or real number to be converted.
- **sString:** holds the converted *Number* as a string.

**Examples:**

```
iNum := 123;  
rNum := 345.24;  
Str(iNum, sConvert1);  
ShowMessage(sConvert1);  
Str(rNum, sConvert2);  
ShowMessage(sConvert2);
```

When the code above is executed then the following is displayed:

123

3.45240000000000E+0002 ← Real numbers are displayed in floating-point format.



### New words

**STR** – to convert an integer or real number into a string, with optional basic formatting

**Upcase** – to convert a single letter character to uppercase

**UpperCase** – to converts lowercase characters in a string to uppercase

**LowerCase** – to converts uppercase characters in a string to lowercase

## UPCASE() FUNCTION

The **Upcase** function converts a single letter ('a'..'z') character to uppercase. If the letter is already in uppercase, it is left unchanged.

Syntax: **Upcase(*cLetter*)**;

- **cLetter:** represents a small letter in the range 'a' to 'z'.

**Examples:**

```
cChar := 'z';  
cLetter := Upcase('t')           // cLetter will store the letter 'T'  
cCh := Upcase(cChar);          // cCh will store the letter 'Z'  
cLetter2 := UpCase('S');        // cLetter2 will store letter 'S'
```

## UPPERCASE() FUNCTION

The **UpperCase** function converts lowercase characters in a string to uppercase.

Syntax: **Uppercase(*sString*)**

- **sString:** represents the string that must be converted to uppercase.

**Example:**

```
sMessage := uppercase('Horse');      //sMessage will store 'HORSE'
```

## LOWERCASE() FUNCTION

The **LowerCase** function converts uppercase characters in a string to lowercase.

Syntax: **Lowercase(*sString*)**

- **sString:** represents the string that must be converted to lowercase.

**Example:**

```
sMessage := lowercase('Save The World');    //sMessage will store  
                                         // 'save the world'
```

## COMPARETEXT FUNCTION

The **CompareText** Function compares two strings, *sString1* and *sString2* for equality, ignoring case. It is not case sensitive. It returns an integer value.

Syntax: CompareText(*sString1*, *sString2*)

### Note:

- The function returns 0 if the strings are equal.
- If *sString1* is greater than *sString2*, the function returns an integer greater than 0.
- If *sString1* is less than *sString2*, the function returns an integer less than 0.

### Example 1:

```
iNum := CompareText('Heidi','heidi')
```

*iNum* will store the value 0 because when the case is ignored, the strings are equal.

### Example 2:

```
sName1 := 'John';
sName2 := 'Janet';
iNum := CompareText(sName1, sName2);
```

*iNum* will store a value greater than 0 because *sName1* is greater than *sName2*.

### Example 3:

```
sName1 := 'John';
sName2 := 'Janet';
iNum := CompareText(sName2, sName1);
```

*iNum* will store a value less than 0 because *sName2* is less than *sName1*.



### Activity 4.7

**4.7.1** Write Delphi statements to display the ordinal value (ASCII) of the following characters.

- 'J'
- 'G'
- '9'

**4.7.2** Read a *sName* string from ComboBox *cmbNames* and convert it to uppercase.

**4.7.3** Study the code segment below and then answer the questions that follow:

```
IF (compareText(sString1,sString2) = 0) THEN
    memDisplay.Lines.Add('Found')
ELSE
    IF (compareText(sString1,sString2) > 0) THEN
        memDisplay.Lines.Add('Descending')
    ELSE
        memDisplay.Lines.Add('ascending');
```

What will be the output if the following strings are read for *sString1* and *sString2* respectively?

- 'happy' and 'Unhappy'
- 'School' and 'school'
- 'Plain' and 'Plane'



### New words

**CompareText** – to compare two strings for equality, ignoring case



## Activity 4.7 *continued*

**4.7.4** Open the project **PasswordEncrypter\_U** in the 04 – Passwoerd Encrypter Folder and write code for the [Encrypt] button.

- Change the letters of the alphabet from the password to the next alphabetical letter, for example if the letter is 'A', the letter must become 'B'. If the letter is 'B', the letter must become 'C', and so on. If the letter is 'Z', the letter must become 'A'.
- Display the encrypted password in the password EditBox.
- Sample output:



## 4.2 Delimited strings

In everyday life, specific characters in written text are used to show where one piece of data ends and the next piece of data begins. In books:

- spaces indicate where one word ends, and another word starts.
- full stops indicate where one sentence ends, and another sentence starts.
- line breaks indicate where one paragraph ends, and another paragraph starts.

These characters are called **delimiters** since they show the start and end (or limits) of individual pieces of data.

In programming, the most common delimiter is the comma symbol (,), which delimits the data stored in comma-separated value (CSV) files. In CSV files, each line of text represents one row of data, while each comma indicates where a column of data starts and ends. An example of data stored in a CSV file is shown in the images below.

```
name,surname,phone
Wilma,Hughes,0821112223
Bongi,Mohoje,0732345678
Rudy,Smith,0845656565
Tokozile,Shabangu,0820000000
Gugile,Dlamini,0731234567
```

	A	B	C
1	name	surname	phone
2	Wilma	Hughes	0821112223
3	Bongi	Mohoje	0732345678
4	Rudy	Smith	0845656565
5	Tokozile	Shabangu	0820000000
6	Gugile	Dlamini	0731234567

**Figure 4.1:** Comma-separated values

The first image shows the CSV file opened in a text editor, while the second image shows the same CSV file opened in a spreadsheet application. The spreadsheet application uses the row and column delimiters to split the data into rows and columns, where each point of data (such as a phone number) can be accessed on its own.

By using the different string functions, it is possible to take a delimited string and interact with the data points inside the string. To see how this is done, work through the following examples and guided activities.



### Guided activity 4.2

#### Delimited string splitter (algorithm)

Data is stored in the following format: 'August#37#18'. The information in the string is delimited by the # symbol. So 'August#37#18' contains three pieces of information: month, temperature and rainfall. To split the pieces of information:

- Extract the string from the EditBox *edtEnter* and stored in a variable called *sLine*.

```
sLine := edtEnter.text;
```

- Identify the position of the first delimiter.

```
iPos := Pos('#',sLine);
```

- Copy the substring starting from position 1 to *iPos* -1.

```
Month := copy(sLine,1,iPos-1);
```

- Delete the substring *sMonth* from *sLine*.

```
Delete(sLine,1,iPos);
```

#### New words

**delimiters** – to show the start and ends of individual pieces of data



### Guided activity 4.2

#### Delimited string splitter (algorithm) *continued*

- Identify the position of the second delimiter.

```
iPos := Pos('#', sLine);
```

- Copy the substring starting from position 1 to iPos -1.

```
iTemp := StrToInt(copy(sLine, 1, iPos-1));
```

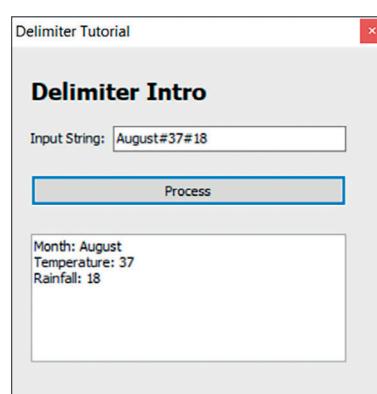
- Delete the extracted substring from sLine

```
Delete(sLine, 1, iPos);
```

- The information that now remains in *sLine* is the rainfall data:

```
iRainfall := StrToInt(sLine);
```

Open the **DelimiterTut\_p** project in the 04 – Delimiter Strings Folder and complete the coding for the [Process] button to extract and split the string from the EditBox as shown below.



In the GUI example above, we knew that the string had exactly two delimiters. What happens when a string does not have a fixed number of delimiters? For example: Different sentences have different number of words which are delimited by the spaces.

The code to obtain each individual word in a sentence is as follows:

```
sSentence := InputBox('Sentence', 'Enter String', 'The  
quick brown fox jumps over the lazy dog');

iPos := pos(' ', sSentence);

while iPos <> 0 do ←
begin
    sWord := Copy(sSentence, 1, iPos - 1);
    redOut.Lines.Add(sWord);
    Delete(sSentence, 1, iPos);

    iPos := Pos(' ', sSentence);

    if iPos = 0 then
    begin
        redOut.Lines.Add(sSentence);
    end;
end;
```

Loops until no delimiters (space) found in *sSentence*

Looks for the next delimiter in *sSentence*.  
If no delimiter is found, the remaining string is displayed.



### Guided activity 4.2

#### Delimited string splitter (algorithm) *continued*

The program above can be adjusted to store the words of the sentence in an array instead of displaying them.

The adjusted code is:

```
...
Var
    arrWords : Array[1..20] of String;
...
sSentence := InputBox('Sentence', 'Enter String', 'The
quick brown fox jumps over the lazy dog');

iCount := 0; <-- {iCount keeps count of the index position of the array and is initialised to 0.}

iPos := pos(' ', sSentence);
while iPos >> 0 do
begin
    sWord := Copy(sSentence, 1, iPos - 1);
    inc(iCount);
    arrWords[iCount] := sWord;
    Delete(sSentence, 1, iPos);

    iPos := Pos(' ', sSentence); <-- {The extracted word (sWord) is stored in the array at position iCount.}

    if iPos = 0 then
begin
    inc(iCount);
    arrWords[iCount] := sWord;
end;

end;
```

{iCount keeps count of the index position of the array and is initialised to 0.}

{The extracted word (sWord) is stored in the array at position iCount.}

{Looks for the next delimiter in sSentence. If no delimiter is found, the remaining string in the next counter position in the array}



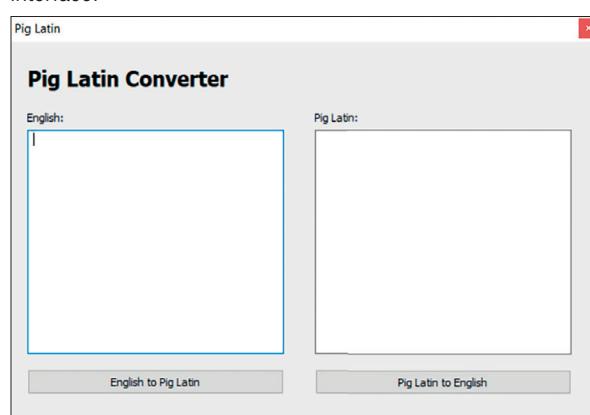
### Activity 4.8

#### English to Pig Latin

Pig Latin is a language game that children play in order to hide what they are saying from people not familiar with Pig Latin. To convert a word to Pig Latin, you place the first character of the word at the end of the word and add the letters "ay". This means that the word 'Delphi' would become 'Elphiday', and the phrase 'Hello world' would become 'Ellohay orldway'. For this project, you need to create a program that can convert from English to Pig Latin, and from Pig Latin to English.

To convert text from English to Pig Latin (and back), you will need to start by identifying all the spaces in the text.

- Open the project saved in the **Pig Latin\_p** project in the 04 – Pig Latin Folder. You should see the following user interface.

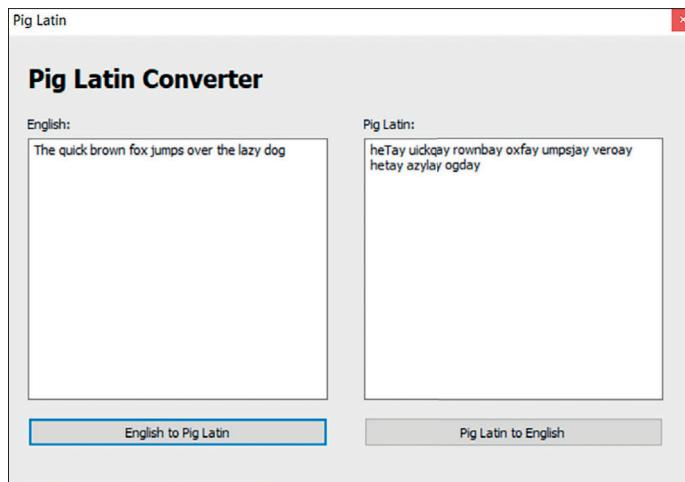




### Activity 4.8

#### English to Pig Latin *continued*

- Write code for the [English to Pig Latin] button to extract data from MemoBox *memEnglish* and then convert the input text into Pig Latin. The converted text should be displayed in the Pig Latin MemoBox *memPL*.
- Write code for the [Pig Latin to English] button that will extract data from MemoBox *memPL* and then convert the extracted text to English from Pig Latin. The converted text should be displayed in the MemoBox *memEnglish*.



## 4.3 Built-in Date-Time methods

The Delphi programming language includes methods and components which are specifically designed to manipulate date and time information.

### THE TDATETIME DATA TYPE

The data type *TDateTime* is capable of storing date and time data in a single variable.

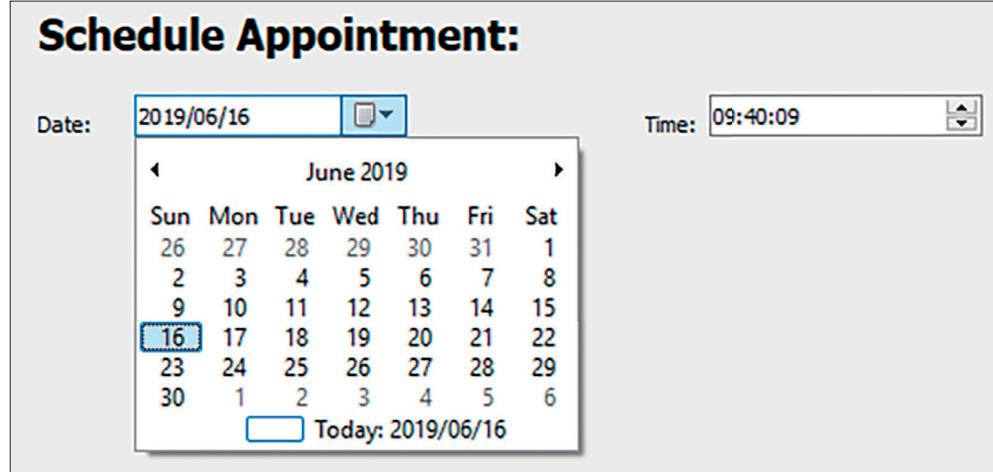
```
var  
  dtToday : TDateTime;
```

The Prefix *dt* will be used for a variable of *TDateTime*

A variable of type *TDateTime* can receive its value through initialisation, from a *DateTimePicker* component or by assigning it to the system date and time.

### THE DATETIMEPICKER COMPONENT

Delphi includes a *DateTimePicker* component that enables the user to select a date from a calendar-style interface.



The component's *Kind* property determines whether it will be used to select a Date or Time.

### DATE FUNCTION

The **Date** function returns the current date in the local time zone. Because the return value is a *TDateTime* type, the **time** component is set to zero (start of day).

Syntax function date: **TDateTime**;

**Example:**

```
Var dtToday:TDateTime;  
...  
  dtToday := Date;
```

## TIME FUNCTION

The **Time** function returns the current time in the local time zone. Because the return value is *TDateTime* type, the **date** component is set to 30/12/1899.

Syntax: function Time: **TDateTime**

Example:

```
Var dtToday : TDateTime;  
...  
    dtToday := time;
```

## NOW FUNCTION

The **Now** function returns both the system date and time to a variable of type *TDateTime*.

```
var  
    dtToday : TDateTime;  
  
begin  
    dtToday := Now;           //Gets current system date and  
                           //time  
end;
```

## ASSIGNING VALUES TO THE TDATETIME VARIABLE

From a **DateTimePicker** component:

```
dtToday := dtpRegistered.Date;
```

By Assignment:

```
dtToday := StrToDate('2019/06/22');
```

From the System Date:

```
dtToday := Date;
```

## CONVERTING DATA IN A TDATETIME VARIABLE INTO A STRING AND VICE VERSA

Data in a *TDateTime* variable will need to be converted to string for the following reasons:

- Most components (such as *ShowMessage* dialogs, *RichEditBoxes* and *Labels*) used commonly for output, receive and display their output as a String.
- If the Date/Time data requires processing using String Handling methods (such as **Copy** or **Pos**), it is necessary to convert the *TDateTime* data into a String and vice versa.

The table below lists the most commonly used conversion functions between the two data types.

**Table 4.3:** The most commonly used conversion functions

FUNCTION	WHAT IT DOES	EXAMPLE
DateToStr	Converts date information stored in a variable of type <code>TDateTime</code> into a variable of type <code>String</code> .	<pre>dtToday := Date; sToday := DateToStr(dtToday); ShowMessage(sToday); //Will output current date in the default //system date format</pre>
StrToDate	Converts date information stored in a variable of type <code>String</code> into a variable of type <code>TDateTime</code> .	<pre>sToday := '2019/06/14'; dtToday := StrToDate(dtToday); //The date formatting must match the system //formatting.</pre>
TimeToStr	Converts time information stored in a variable of type <code>TDateTime</code> into a variable of type <code>String</code> .	<pre>dtToday := Time; sToday := TimeToStr(dtToday); ShowMessage(sToday); //Will output current time in the default //system time format.</pre>
StrToTime	Converts time information stored in a variable of type <code>String</code> into a variable of type <code>TDateTime</code> .	<pre>sTime := '11:15:50'; dtTime := StrToTime(sTime); //The time formatting must match the system //formatting.</pre>
DateTimeToStr	Converts <code>TDateTime</code> values into a formatted date and time string	<pre>dtDate := StrToDate('18/07/2018 11:15');  ShowMessage('Date &amp; Time: ' + DateTimeToStr(dtDate));  //Will display 18/07/2018 11:15</pre>

## FORMATDATETIME FUNCTION

The `FormatDateTime` function converts data of type `TDateTime` into data of type `String`. The returned String can be formatted to return the Date/Time data in a user-defined format.

DATE FORMATTING	TIME FORMATTING
yy	2 digit year
yyyy	4 digit year
mm	2 digit month
mmm	Short month name (example Jan)
mmmm	Long month name (example January)
dd	2 digit day
dd	Short day name (example Sun)
ddd	Long day name (example Sunday)

**Example:**

```
Var  
    dtToday : TDateTime;  
    sToday : String;  
...  
    dtToday := Now;  
    sToday := FormatDateTime('ddd dd mmmm yyyy @ hh:nn:ss', dtToday);  
    showMessage(sToday);
```

Assuming the user runs the program on 16 June 2020 at 10:16:43, the output value will display as:

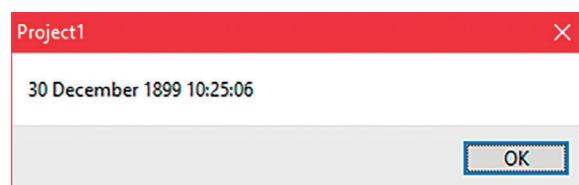
Tuesday 16 June 2020 @ 10:16:43

**Notes:**

- The TDateTime data type is capable of storing date and time information in a single variable.
- If we don't provide a value for the Date, the variable will revert to the earliest system date.
- If we don't provide a value for the time, the variable will revert to midnight.

**Example 1:**

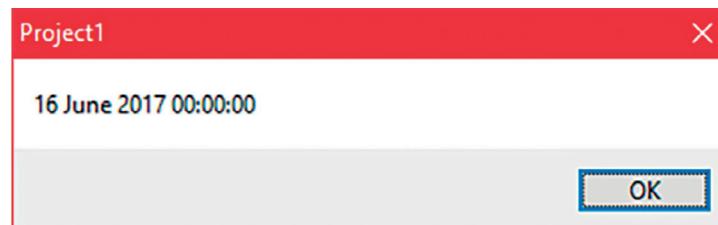
```
Var  
    dtToday : TDateTime;  
    sToday : String;  
...  
    dtToday := Time;  
    sToday := FormatDateTime('dd mmmm yyyy hh:nn:ss',dtToday);  
    showMessage(sToday);  
...
```



In this case, since the variable *dtToday* was assigned only to Time, the date value reverted to the earliest system date (30 December 1899). The default date depends on the hardware the program is running on.

**Example 2:**

```
Var  
    dtToday : TDateTime;  
    sToday : String;  
...  
    dtToday := StrToDate('16/06/2017');  
    sToday := FormatDateTime('dd mmmm yyyy hh:nn:ss',dtToday);  
    showMessage(sToday);  
...
```



In this case, since the variable *dtToday* was assigned only to a specific date, the time value reverted to the earliest system time which is midnight.

It is important to remember that a *TDateTime* variable has 2 parts (Date and Time) and both parts need to be initialised / set before processing starts.

## ASSIGNING THE DATE AND TIME IN A SINGLE STATEMENT

The following assignment statements will set both the date and time values in the *TDateTime* variable *dtToday*.

```
System Date and Time:  
dtToday := Now;
```

OR

```
dtToday := Date + Time;
```

From DateTimePicker components:

```
dtToday := dtpRegDate.Date + dtpRegTime.Time;
```

Using Specific Values:

```
dtToday := StrToDate('2019/06/16') + StrToTime('10:46:00');
```

## ISLEAPYEAR FUNCTION

The **IsLeapYear** function returns true if a given calendar value is a leap year. Year can have a value 0...9999.

Syntax: Function *isLeapYear*(const year: word):Boolean;

**Example:**

```
...  
  
iYear := StrToInt(InputBox('Year','Enter a Year',''));  
if isLeapYear(iYear) then  
    ShowMessage(IntToStr(iYear) +' is a leap year')  
else  
    ShowMessage(IntToStr(iYear)) +' is not a leap year');
```

## THE TIMER COMPONENT

The **Timer** is an invisible component that executes code written in its *OnTimer* event at fixed intervals.



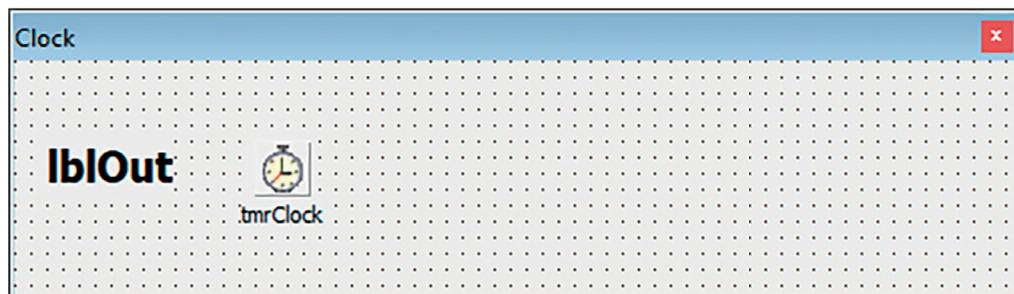
The timer can be switched on and off by toggling its *Enabled* property.

If the Timer is enabled, the code in the *OnTimer* event will execute automatically at intervals determined by the *Interval* property. This property is specified in milliseconds and defaults to 1000 milliseconds or 1 execution per second.



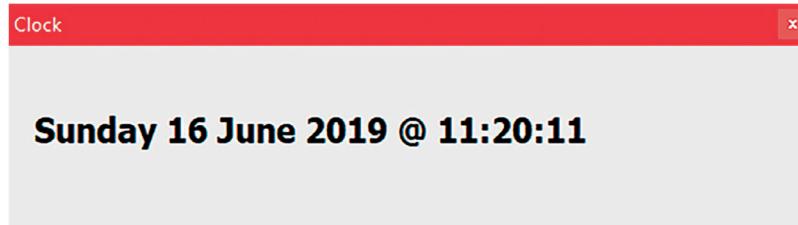
### Guided activity 4.3

1. Open project **Clock\_p** in the 04 – Clock Folder. The following interface is provided:



2. Create an *OnTimer* event for the *tmrClock* component.
3. Declare two variables: *dtToday* of type *TDateTime* and *sToday* of type *String*.
4. Extract the system date and time and assign it to *dtToday*.
5. Call the *FormatDateTime* function, assigning it to *sToday*. Display the Long Day name, the date, the long month name, a four-digit year. Follow this with the "@" symbol and include the time with hours, minutes and seconds.
6. Display *sToday* in the label *lblOut*.
7. Save and run the project.

```
Var
    dtToday : TDateTime;
    sToday : String;
...
dtToday := Now;
sToday := FormatDateTime('dddd dd mmmm yyyy @ hh:nn:ss', dtToday);
lblOut.Caption := sToday;
...
```





## Activity 4.9

- 4.9.1** State the value of *sOutput* in each of the following:

*dtToday* is a variable of type *TDateTime* and *sOutput* is a variable of type String.

- a. *dtToday* := Date;  
*sOutput* := TimeToStr(*dtToday*);
- b. *dtToday* := StrToDate('2003/03/13') + StrToTime('15:00:00');  
*sOutput* := FormatDateTime('dd mmm yy', *dtToday*);

- 4.9.2** The following data structures have been provided:

```
Var
  dtToday : TDateTime;
  sToday : String;
  arrDetails : Array[1..7] of String;
...
  dtToday := Now;
  sToday := FormatDateTime('dddd dd mmmm yyyy hh:nn:ss', dtToday);
```

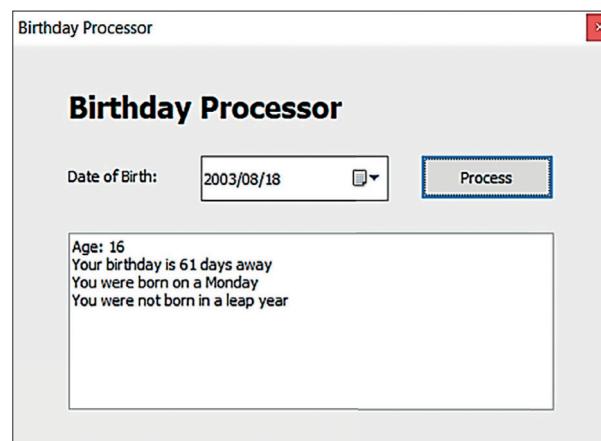
Write code to separate the delimited data in *sToday* and store each part in Array *arrDetails*.



## Activity 4.10

Open **BirthdayProcessor\_p** project in the 04 – Birthday Processor Folder. The following interface is provided:

- Declare two variables of type *TDateTime*: One to store the current date and the other to store the user's Date of Birth.
- Extract date of birth from the DateTime picker.
- Get the system date.
- Use the extracted data to determine and display:
  - The user's age
  - The number of days to the user's birthday. If the user's birthday has already passed for the current year, then display a suitable message.
  - The day (example: Sunday) when the user was born.
  - Whether the person was born on a leap year or not.



**QUESTION 1**

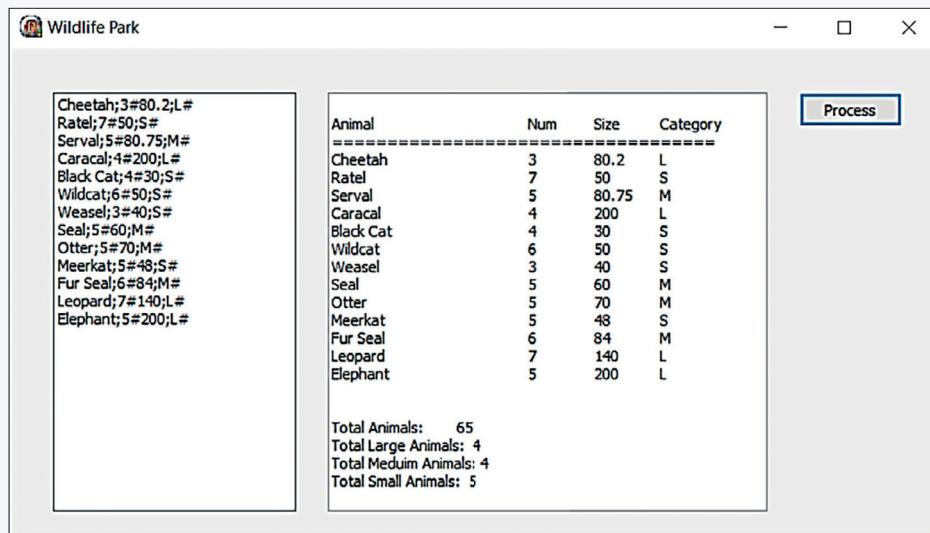
Open the **Wildlife\_p** project in the 04 – Wildlife Folder. The ListBox *lstEnclosures* stores information about the details of each enclosure in the Wildlife Park in the following format: <Type of animal>;<Number of animals currently in the enclosure>#<Size of the enclosure in square metres>;<Category of animals based on size>#

Example of the data for the **first five** enclosures in the ListBox *lstEnclosures*:

- Cheetah;3#80.2;L#
- Ratel;7#50;S#
- Serval;5#80.75;M#
- Caracal;4#200;L#
- Black-footed Cat;4#30;S#

**1.1** Write code for the [Process] button to do the following:

- a. Read each line of text from the ListBox *lstEnclosures* and display the text in the format shown in the sample run below.
- b. Display the total number of animals found at the Wildlife Park.
- c. Determine and display the number of animals in each category 'L', 'M' and 'S'.



**QUESTION 2**

- 2.1** Open the project **PhoneNumbers\_p** project in the Phone Numbers Folder.

The given program generates an array of phone numbers for you to work with. The array called *arrPhoneNos* holds 20 strings. Data in the array will look like this:

```
arrPhoneNos [1] := '086 New Hill';
arrPhoneNos [2] := '086 Dial Bar';
arrPhoneNos [3] := '086 Bay View';
arrPhoneNos [4] := '086 Kya Sand';
arrPhoneNos [5] := '086 SowetoN';
arrPhoneNos [6] := '086 Casa Sol';
arrPhoneNos [7] := '086 The Havn';
arrPhoneNos [8] := '086 Get Food';
arrPhoneNos [9] := '086 Thai Plc';
arrPhoneNos [10] := '086 Cleaner';
arrPhoneNos [11] := '086 Casa Rok';
arrPhoneNos [12] := '086 Rix Taxi';
arrPhoneNos [13] := '086 Air Time';
arrPhoneNos [14] := '086 Dial Bed';
arrPhoneNos [15] := '086 Dial Car';
arrPhoneNos [16] := '086 Dial Hlp';
arrPhoneNos [17] := '086 Kya Rosa';
arrPhoneNos [18] := '086 Bay Sand';
arrPhoneNos [19] := '086 Cater 4 U';
arrPhoneNos [20] := '086 1to1 Air';
```

- a. Write code for the [Convert] button to convert all the alphanumeric characters in the *arrPhoneNos* array into normal telephone numbers. Replace the alphabetic characters (upper case and lower case) in the telephone numbers with the corresponding numbers given below:

A, B, C	2
D, E, F	3
G, H, I	4
J, K, L	5
M, N, O	6
P, Q, R, S	7
T, U, V	8
W, X, Y, Z	9

The numeric values in the telephone numbers remain as they are.

**Note:** The resulting numerical phone number must be formatted as follows: 3 digits, space, 3 digits, space, 4 digits (for example 086 345 6546)

Store the normal telephone numbers in a new array *arrNewPhoneNos*.

- b. Write code for the [Display] button to display the original alphanumeric number and the new numerical version.

Example of output:

Original Number	Converted Number
086NewHill	086 639 4455
086DialBar	086 342 5227
086BayView	086 229 8439
086KyaSand	086 592 7263
086SowetoN	086 769 3866
086CasaSol	086 227 2765
086TheHavn	086 843 4286
086GetFood	086 438 3663
086HaiPlc	086 842 4752
086Cleaner	086 253 2637
086CasaRok	086 227 2765
086RixTaxi	086 749 8294
086AirTime	086 247 8463
086DialBed	086 342 5233
086DialCar	086 342 5227
086DialHlp	086 342 5457
086KyaRosa	086 592 7672
086BaySand	086 229 7263
086Cater4U	086 228 3748
0861to1Air	086 186 1247

- c. Write code for the [Duplicate] button: Use the numerical phone numbers to check that there are no duplicates in the array. If duplicates are found, the program must display the duplicate numbers. If no duplicates are found, a suitable message must be displayed. At the end of the list there must be a summary stating how many duplicates were found (if any).

Example of the output:

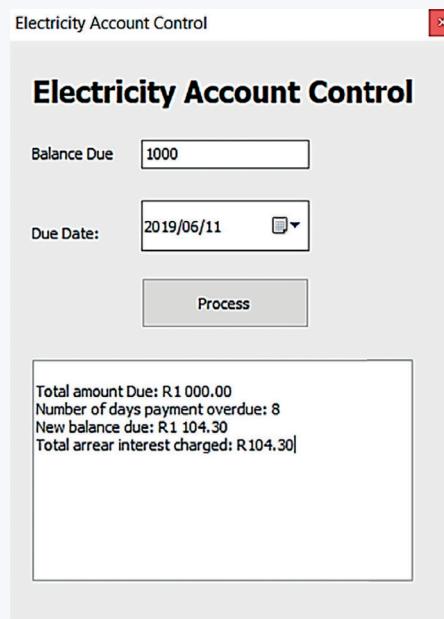
Duplicates  
086 342 5227  
086 227 2765  
The number of duplicates : 2

**QUESTION 3**

- 3.1** The town electricity account control department requires help in calculating arrears interest on outstanding payments. Arrears interest is calculated daily and added to the outstanding amount owing. In terms of their policy, arrears interest is calculated per day as follows:

DAYS	ARREARS INTEREST % PER DAY
1–7	1%
8–14	3%
>14	5%

Open the **ElectricityAcc\_p** project in the 04 – Electricity Account Folder and write code for the [Process] button to do the following:



- Read the balance due amount from the EditBox.
- Select the date from the date picker component of the due date by when the balance due was payable.
- Calculate the following:
  - number of days payment is overdue as of the current date,
  - total amount owing after arrears interest is added and
  - total arrear interest charged.
- Display the original balance due, the number of days that payment is overdue, the new balance due, the total arrear interest charged.

Save and run the project.

# TEXT FILES

## CHAPTER OVERVIEW



- Unit 5.1 Introduction to text files
- Unit 5.2 Reading from a text file
- Unit 5.3 Writing to a text file
- Unit 5.4 Creating reports



## Learning outcomes

At the end of this chapter you should be able to:

- create text files
- use text file functions and procedures
- read and display data from a text file and write data to a text file
- use string manipulation functions to read delimited text files
- catch and handle errors
- create reports using data from a text file.

## INTRODUCTION

You learnt that a variable stores information temporarily in your computer's memory. When working with large volumes of data, it can be time-consuming to enter the data from the keyboard each time the program is executed. We can use information stored in text files as input.

Text files are stored on a storage medium. Text files provide a simple, convenient and permanent way of storing textual data and are commonly used for importing and exporting data to and from programs. A text file does not have any formatting or structure and therefore you can transfer them between programs. You can store information permanently in a text file and read and manipulate this information using a Delphi program. By the end of this chapter, you should be able to use the different text file functions and procedures to open a text file, read data from a text file and write data to a text file.

## 5.1 Introduction to text files



Take note

Text files have a **.txt** extension

### WHAT IS A TEXT FILE?

A text file contains text with no formatting, that is, no formatting features such as bold, underline, tables, styles and so on. Since the text file has no formatting, it can be used by different programs.

### CREATING A TEXT FILE

You can create a text file by using any one of the methods below:

- A text editor like NotePad:  
Open the NotePad program, type the text and save the file. The file will have a .txt extension.
- The Delphi Code Editor:  
In Delphi:
  - Click on *File, New, Other, Other file, Text File, OK*. Select \*.txt as the file extension.
  - Type the text and save the file.
- Writing Delphi Code.

#### Note:

- The text file must be saved in the same folder as the project file.



#### Activity 5.1

- 5.1.1** Create a text file called **IndustrialRevolution.txt** using NotePad. The text file must contain the following text:

```
IndustrialRevolution.txt - Notepad
File Edit Format View Help
Fourth Industrial Revolution
Everyone is talking about the fourth industrial revolution
The fourth industrial revolution will change how we produce, how we consume, how we communicate and
even how we live
Get ready for change
```

- 5.1.2** Create a text file called **StudyTips.txt** using the Delphi code editor. The text file must contain the following text:

```
StudyTips.txt
- Study Tips
- Try not to let stress get to you!
- Remember: you got this
- Don't leave it to the last minute
- Organise your study space
- Use flowcharts and diagrams
7 Practice on old exam papers
```

### DISPLAYING THE CONTENTS OF A TEXT FILE

You can display the contents of a text file in a MemoBox component or a RichEdit component.

#### Examples:

- memDisplay.Lines.LoadFromFile('Cities.txt');
- redDisplay.Lines.LoadFromFile('SeasonVisits.txt');

File found on storage medium

### Note:

If the text file is not saved in the same folder as the project file, then you need to indicate the path for the text file.

Example:

```
memDisplay.Lines.LoadFromFile('c:\DelphiProjects\Cities.txt');
```

## SAVING THE CONTENTS OF A MEMOBOX OR RICHEDIT COMPONENT AS A TEXT FILE

The contents of a MemoBox or RichEdit component can be saved as a text file.

### Examples:

- memDisplay.Lines.SaveToFile('NewCities.txt');
  - redDisplay.Lines.SaveToFile('Tours.txt');
- Name of file to store the contents of the MemoBox or RichEdit component



### Activity 5.2

Open the **TextLoadSave\_p** project from the 05 – Text Load Save Folder and write code to do the following:

- 5.2.1 [Load Text File] button: Display the contents of the text file **Sports.txt** using the MemoBox component. Make the MemoBox component scrollable vertically.
- 5.2.2 [Create Text File] button: Save the contents of the RichEdit component to a file **Fruits.txt** and display a message 'Saved to file'.

## PHYSICAL FILE VERSUS LOGICAL FILE NAME

The **physical file** name refers to the external file name found on a storage device and contains the actual data.

The file name that you will use in your Delphi program must be declared as you would any other variable in your program:

```
Var  
  <fileVariableName>: Textfile;
```

The *fileVariableName* refers to the **logical file** name that you use in your Delphi program and **not** the physical file name on your storage device. The logical file name is a variable (in RAM) that points to the physical file on your storage medium.

## LAYOUT OF DATA IN A TEXT FILE

Each line of a text file has an **end of line <eoln>** marker. The <eoln> marker is added to the end of a line when the [Enter] button is pressed. An **end of file <eof>** marker is added to the end of a file when the file is saved.



### New words

**physical file** – to name an external file name found on a storage device and contains the actual data

**logical file** – is a variable (in RAM) that points to the physical file on your storage medium

**end of line <eoln>** – to indicate the end of the line when the [Enter] button is pressed

**end of file <eof>** – to indicate the end of a file when the file is saved

The <eoln> and <eof> markers refer to Boolean functions eof() and eoln() and return a value of either true or false. Example:

```
Fourth Industrial Revolution <EOLN>
Everyone is talking about the fourth industrial revolution <EOLN>
It will change how we produce, how we consume, how we communicate and even
how we live <EOLN>
Get ready for change <EOLN>
<EOF>
```

## TEXT FILE PROCEDURES

PROCEDURE	DESCRIPTION	SYNTAX
AssignFile	<p>Creates a link between the logical file name and the physical file found on a storage medium.</p> <p>This procedure does not check whether the file exists or not. You need to use programming code to check whether a physical file exists or not.</p>	<pre>AssignFile(tName, 'file.txt');</pre> <p>Links the logical file name tName to the physical file 'File.txt'</p>
Rewrite	<p>Create a new file with write only access(writing) and sets the file pointer at the beginning of the file.</p> <p>If the file already exists, the contents of the file is lost.</p> <p>A buffer is created in RAM when this procedure is executed. The buffer is opened for writing and the file pointer is set to the beginning of the buffer area in RAM.</p>	<pre>Rewrite(tName);</pre>
Reset	<p>Opens an existing file for read only access and sets the file pointer at the beginning of the file so that text can be read from the beginning of the text file.</p> <p>If the file does not exist, the program will crash. Programming code must be used to determine whether a file exists or not.</p>	<pre>Reset(tName);</pre>
Append	<p>Opens the file for writing and sets the file pointer to the end of the file so that text can be written to the end of the text file</p> <p>Note that the file that you want to append to must exist for you to make use of the Append procedure.</p> <p>It only allows you to add data to the end of the file.</p>	<pre>Append(tName);</pre>
Writeln	<p>Writes a line of text at the current position of the file pointer and adds an end-of-line marker after the written text. The file pointer moves to the next line ready to write a new line.</p>	<pre>Writeln(tName, sOutput);</pre> <p>Writes the contents of variable sOutput to the current position of the file pointer and positions the file pointer to the next line.</p>

PROCEDURE	DESCRIPTION	SYNTAX
Write	Write does not place an end-of-line marker after writing text to a file, so when you make use of the write procedure all the text that you write to the file will be written on one line.	<code>Write(tName, sOutput);</code>
Readln	Reads the selected line of text from the current file pointer position in the file and moves the file pointer to the next line.	<code>Readln(tName, sInput);</code> Reads the contents of a line at the current position of the file pointer and stores the contents in variable sInput
Read	Reads all the characters up to the end-of-line marker, or until Eof(FileVariable) becomes true. The Read procedure does not read the end-of-line marker with the characters.	<code>Read(tName, sInput);</code>
CloseFile	Closes the link between the text file's logical name and the physical file name. You cannot read or write to the text file once it is closed.	<code>CloseFile(tName);</code>

## TEXT FILE FUNCTIONS

FUNCTION	DESCRIPTION	SYNTAX
Eof	Returns a Boolean value that indicates whether you have reached the end of the file or not. A return value of true indicates that the end of the file has been reached.	<code>bEndOfFile := Eof(tName);</code>
Eoln	Returns a Boolean value that indicates whether you have reached the end of a line or not. A return value of true indicates that the end of line has been reached.	<code>bEndOfLine := eoln(tName);</code>
FileExists	Returns a Boolean value that indicates whether a file with the given path exists or not.	<code>bFileExists := FileExists('file.txt');</code>



### Activity 5.3

Complete the following activity using pen and paper.

- 5.3.1 Explain what the purpose of the AssignFile procedure is.
- 5.3.2 Explain the difference between the Rewrite, Reset and Append procedures.
- 5.3.3 What type of value is returned by the Eof function?
- 5.3.4 Explain the purpose of the CloseFile procedure.

## 5.2 Reading from a text file

### STEPS TO FOLLOW WHEN READING FROM A TEXT FILE

- **Step 1:** Declare the text file variable name (logical file name) that you will use to refer to the file in Delphi. This is not the name of the physical file on your storage device, but instead is the name of a variable that points to the file on the storage medium.

Example:

```
Var tLearners: TextFile;
```

- **Step 2:** Link (or assign) a physical file to this text file variable. By doing this, your computer knows which file on your storage medium to access when you read or write information.

Example:

```
AssignFile(tLearners,'Learners.txt');
```

- **Step 3:** Indicate how you would like to use the file. The three options available are Rewrite, Reset and Append.

Examples:

```
Reset(tLearners);
OR Rewrite(tLearners);
OR Append(tLearners);
```

- **Step 4:** Read the contents of the line at the position of the file pointer and store the contents in the variable *sLine*. Once the line has been read, the file pointer points to the next line.

Example:

```
Readln(tLearners,sLine);
```

- **Step 5:** Close the file. Whenever you are done with your file, you need to make sure that you close the link between the text file variable in Delphi and the physical file, allowing other applications to read this file.

Example:

```
CloseFile(tLearners);
```

### FILEEXISTS FUNCTION

Your program can crash if you try to read from or write to a file that does not exist. You can use the **FileExists** function to determine whether a file exists or not.

Syntax: Function FileExists(FileName:string):Boolean;

#### Note:

- The **FileExists** function returns the value TRUE if the physical file name *FileName* exists
- The program searches for the file name in the current directory



#### New words

**FileExists** – to determine whether a file exists or not

### Example 1:

```
if FileExists('Learners.txt') then
    ShowMessage('File is found on storage medium')
Else
    ShowMessage('File not found on storage medium');
```

### Example 2:

```
bFileFound := FileExists('Learners.txt');
if bFileFound then
    ShowMessage('File is found on storage medium')
Else
    ShowMessage('File not found on storage medium');
```

## EXCEPTION HANDLING

Another way to prevent a program from crashing when a file does not exist is to use **Exception Handling** in Delphi.

An **exception** is generally an error condition or event that interrupts the flow of your program. In order to prevent an interruption in the flow of the program, we use the **try...except** statement:

```
try
    ...
except
    ...
end;
```

#### Note:

- The **Try** command tells Delphi to try and run code where errors may occur. If no errors occur, the code runs like it normally would. The exception block (**except**) is ignored, and control is passed to the statement following the **end** keyword
- However, if an error occurs in the Try section, Delphi ‘catches’ the error and the **except** code will take over.

The **try...except** statement can be used to check whether a file exists or not:

```
Try
    Reset(tLearners);
Except
    ShowMessage('File does not exist');
    Exit;
End;
```

## READING ALL LINES FROM A TEXT FILE

The lines in a Text File can only be accessed sequentially. This means that the values can only be read in the order in which they were written. To read all the data from a text file, the *Readln* and *Eof* methods can be combined with a WHILE-DO-loop. The WHILE-DO-loop continues to read lines from the text file, one line at a time, and adds them to the *ListBox*, until the end-of-file is reached.



### New words

**Exception Handling** – a way to prevent a program from crashing when a file does not exist

**exception** – is generally an error condition or event that interrupts the flow of your program

### While loop to read all the data from the Text File

```
var  
  tName: TextFile; ← Declare a file variable (tName)  
  sLine: string; ← Declare a String variable (sLine) which will store a line of  
begin  
  AssignFile(tName,'filename.txt'); ← Links the text file variable tName and the physical file  
  if not FileExists('filename.txt') then  
    begin  
      ShowMessage('The file does not exist');  
      Exit; ← Exits the current procedure  
    end  
  else  
    begin  
      Reset(tName); ← Opens an existing file for reading and sets the file pointer  
      to the beginning of the file  
      Do this in the Try... Except OR Test if the file exists.  
      while not Eof(tName) do ← Loop until the EOF marker has been reached.  
        begin  
          ReadLn(tName,sLine); ← Reads a line of text from the text file tName into a String  
          variable (sLine)  
          lstLines.Items.Add(sLine); ← Add the line of text sLine to the ListBox lstLines  
        end; //While  
      CloseFile(tName); ← Closes the link between the text file variable and the  
      physical file on storage  
    end; //event handler
```

#### Note:

- To read all the text from a text file, a line at a time, you need to loop through the text file reading a line at a time and adding the line of text to a ListBox until the EOF marker is reached.
- If you need to read a specific line of code from a text file, then you also have to loop through the text file until the specific line is found or until the EOF marker is reached.

Remember that a text file is sequential in nature and you have to read data in the order in which it was written to the text file. You cannot jump directly to the line you want to read.

## READING THE FIRST LINE OF TEXT AND STORING IT IN A VARIABLE

### Reading the first line

```
AssignFile(tName, 'filename.txt');  
if not FileExists('filename.txt') then  
  begin  
    showMessage('The file does not exist');  
    Exit;  
  end;  
else  
  begin  
    Reset(tName);  
    ReadLn(tName, sOutput);  
    CloseFile(tName);  
  End;
```

## READING THE FIFTH LINE OF TEXT AND STORING IT IN A VARIABLE

### Reading 5<sup>th</sup> line

```
AssignFile(tName, 'filename.txt');
if not FileExists('filename.txt') then
begin
  ShowMessage('File does not exist');
  Exit;
end;
Reset(tName);
for i := 1 to 5 do
begin
  ReadLn(tName, sOutput);
End;
CloseFile(tName);
ShowMessage(sOutput);
```

## RICHEDIT COMPONENT

The *RichEdit* component works in the same basic way as a *Memo* component but offers additional formatting options. We will use the *RichEdit* component to tabulate output. To do this, you will need to change the following properties of your *RichEdit* component.

PROPERTY	DESCRIPTION	POSSIBLE VALUES
Font.Color	Sets the color of the font.	clBlack clWhite clBlue clRed clYellow
Paragraph.Numbering	Sets the type of bullet points to add to each line.	nsBullet nsNone
Paragraph.Alignment	Sets the alignment of your paragraphs to left, right or center.	taLeftJustify taRightJustify taCenter
Paragraph.TabCount	Sets the number of tab stops that you want to use with your TRichEdit component.	Any positive integer such as 1, 2 or 3
Paragraph.Tab[Index]	Index indicates which tab stop you are working with. It sets the end of the tab stop as a number of pixels from the left of the TRichEdit component.  Take note: the index starts at 0.	Any positive integer such as 100, 200 or 250

### Example 5.1

Suppose we want to display information in the following format:

ClientNo	Full Name	Client Type
AS67	Julian Bernard	Premium
AK99	Audrey Gilliam	Basic
AU95	Catherine Browning	Professional
AR18	Evan Guzman	Professional
AO90	Rhonda Lowery	Premium
AF84	Griffith Hodge	Prestige

```
...
// sets the number of tab stops to 2
redDisplay.Paragraph.TabCount := 2;
// the first tab stop at index position 0 is set to 100
redDisplay.Paragraph.Tab[0] := 100;
// the first tab stop at index position 1 is set to 200
redDisplay.Paragraph.Tab[1] := 200; // the second tab stop
```

```
Move a tab position
    ↓
redDisplay.Lines.Add('ClientNo' + #9 + 'Full Name' + #9 + 'Client Type');
redDisplay.Lines.Add('=====');
for i := 1 to 6 do
begin
  sClientNo := edtClientNo.text;
  sFullName := edtFullName;
  sType := edtType;
  redDisplay.Lines.Add(sClientNo + #9 + sFullName + #9 + sType);
end;
...
...
```



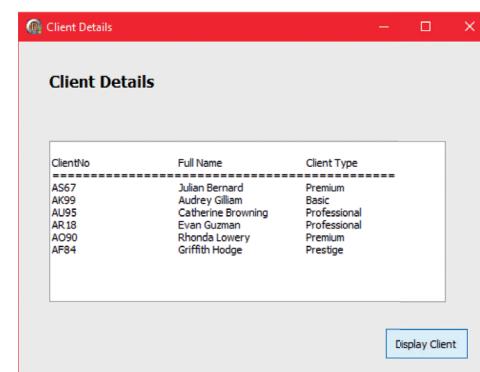
### Activity 5.4

**5.4.1** The data for each client is stored on a separate line as follows in a text file **Clients.txt**:

```
<Client No>
<Name and Surname>
<Client Type>
<Client No>
<Name and Surname>
<Client Type>

...
```

- Open the **Clients\_p** project from the 05 – Clients Folder and create an OnClick event for the [Display Client] button to display the Client No, Full Name of the client and the Client Type from the text file as shown.
- Save and run the program.





## Activity 5.4 *continued*

- 5.4.2** The data for each client is stored as follows in a text file **ClientsDelimited.txt** in the 05 – Clients Delimited Folder:

```
<Client No>,<Name and Surname>,<Client Type>
```

Open the **ClientsDelimited\_p** project from the 05 – Clients Delimited Folder and create an *OnClick* event for the [Display Client] button to:

- Display the Client No, Full Name of the client and the Client Type from the text file. Ensure that you can scroll through the information in the richEditBox.
- Display the total number of Clients and the total number of clients that are of client type 'Prestige'
- Save and run the program.

- 5.4.3** The details of users at an Internet Café is stored in a text file **LoginDetails.txt** in the following format:

```
<name>,<age>,<email address>,<password>
```

Client Details		
ClientNo	Full Name	Client Type
AS67	Julian Bernard	Premium
AC69	Audrey Gillam	Basic
AL85	Catherine Downing	Professional
AR18	Evan Guzman	Professional
AO90	Rhonda Lowery	Premium
AF84	Griffith Hodge	Prestige
AH23	Stewart Evans	Premium
AM69	Nell Conner	Prestige
AQ79	Kane Osborne	Basic
AP75	Alfred Maxwell	Professional
AQ99	Forrest Meja	Professional
AC85	Russell Hancock	Prestige
AV22	Nichole Kidd	Basic
AF74	Eleanor Farrell	Basic
AF52	Kellie Hooper	Professional
AD39	Sheila Hayes	Basic
AL93	Josie Moreno	Basic
AQ13	Briar Armstrong	Prestige
AZ57	Fatima Washington	Premium
Total number of clients: 19		
Total number of Prestige Clients: 4		

**Hint:** You can view a list of valid emails and passwords by opening the **LoginDetails.txt** file in Notepad.

Open the **VerifyLogin\_p** project from the 05 – Verify Login Folder.

Create an *OnClick* event for the [Login] button to do the following:

- Read the email address and password from the EditBoxes.
- Compare the input email and password combination against the email and password combinations stored in the text file called **LoginDetails.txt**.
- Show a message, using an output DialogBox, indicating whether a correct or incorrect combination of email and password was entered.
- Save and run the project.

The figure consists of two side-by-side screenshots of a Windows application window titled "Login". Both screenshots show two text input fields: "Email" containing "vakarianknight@outlook.com" and "Password" containing "@valanche". Below the fields is a blue "Login" button. In the bottom-left screenshot, a red dialog box titled "Verifylogin\_p" displays the message "You are successfully logged in" with an "OK" button. In the bottom-right screenshot, a red dialog box displays the message "Email and password do not match" with an "OK" button.

**Figure 5.1:** Sample Data where credentials are input correctly

**Figure 5.2:** Sample Data where credentials are input incorrectly

## 5.3 Writing to a text file

Now that you know how to create a file and assign it to a text file variable, you are ready to start adding data to it. You already know that a text file can be created using a Text editor (Notepad or Word processing program) or the Delphi Code Editor. We are now going to learn how to create a text file using Delphi programming code.

You need to consider the format in which text will be written to the text file. The format used must facilitate easy reading and interpretation at a later stage.

### CREATING AND ADDING DATA TO A NEW FILE

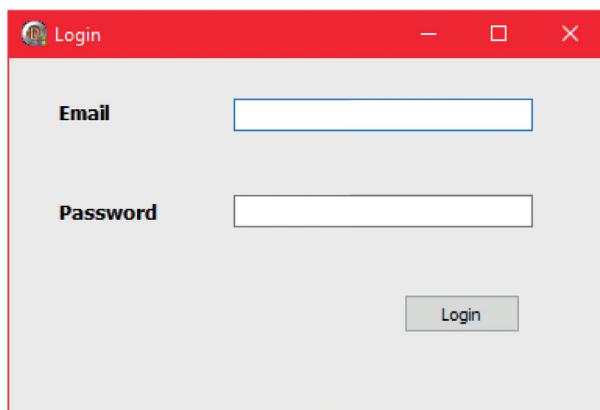
You need to create a file before writing information to the file and determine the format that will be used to write information to the text file.



#### Guided activity 5.1

- 5.1.1 Open the **StoringLogin\_p.dproj** project from the 05 – Storing Login Folder.
- 5.1.2 The OnClick event for the [Login] button demonstrates how information is written to the text file. The user enters an email address and password in the EditBoxes. The email address and password will be written to the new text file Login.txt in the following format:

```
<email address>,<password>
```



```
procedure TForm1.btnLogInClick(Sender: TObject);
Var
  tFileName: TextFile; ←
  sUserName, sPassword, sLine: String;
begin
  ↓
  AssignFile(tFileName, 'Login.txt');
  Rewrite(tFileName)←
  sUserName := edtUserName.Text;
  sPassword := edtPassword.Text;
  sLine := sUserName + ',' + sPassword;
  Writeln(tFileName, sLine);←
  closeFile(tFileName);←
  showMessage('File has been created');
end;
```

Declare a text file variable

Links the text file variable with the new external file.

Physically creates the text file on your storage medium. If the file already exists, the contents of the file are lost.

Writes a line to the text file at the current file pointer position

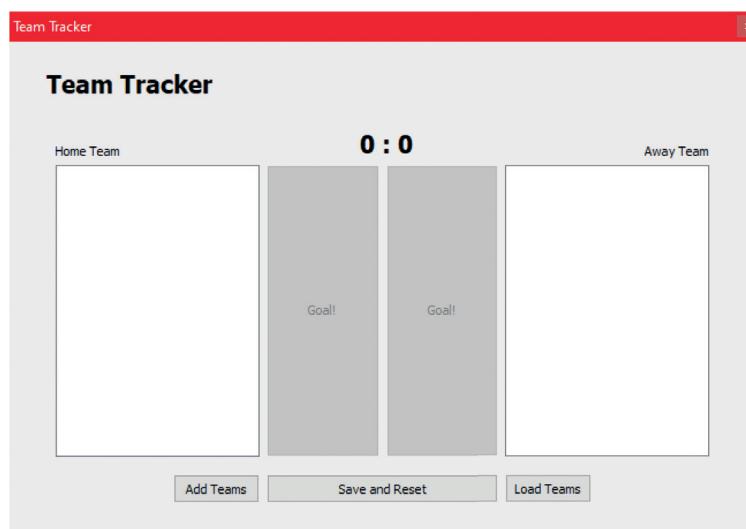
Closes the text file

## ADDING DATA TO AN EXISTING FILE



### Guided activity 5.2

Open the **TeamTracker\_p** project from the 05 – Team Tracker Folder. The following interface will display when the program is run.



- The aim of this application is to track the goals scored during a match between two teams.
- The user starts using the application by clicking the [Load Teams] button. This loads a list of participating teams into the ListBoxes *lstHome* and *lstAway* from a text file.
- The two teams currently playing are selected from the ListBoxes *lstHome* and *lstAway*.
- Every time a team scores a goal, a [Goal!] button for the away or home team is clicked depending on who scored the goal. If the Home team scores a goal, *btnHome* button is clicked and their score is increased by 1. If the Away team scores a goal, *btnAway* button is clicked and their score is increased by 1.
- The scores are actively updated in the *lblScore* label.
- After the match is complete, the [Save and Reset] button is clicked which adds the results of the match to a text file named **results.txt** and then resets the application by deselecting the teams in the two ListBoxes and setting both teams' scores back to 0.
- The [Add Teams] button allows the user to add a new team to the roster of participating teams. The team is added to both ListBoxes as well as to the text file containing the list of teams.

Write code to do the following:

- Declare three global variables as shown below:

#### TeamTracker global variables

```
tTeam : TextFile;  
iHomeScore, iAwayScore : Integer
```

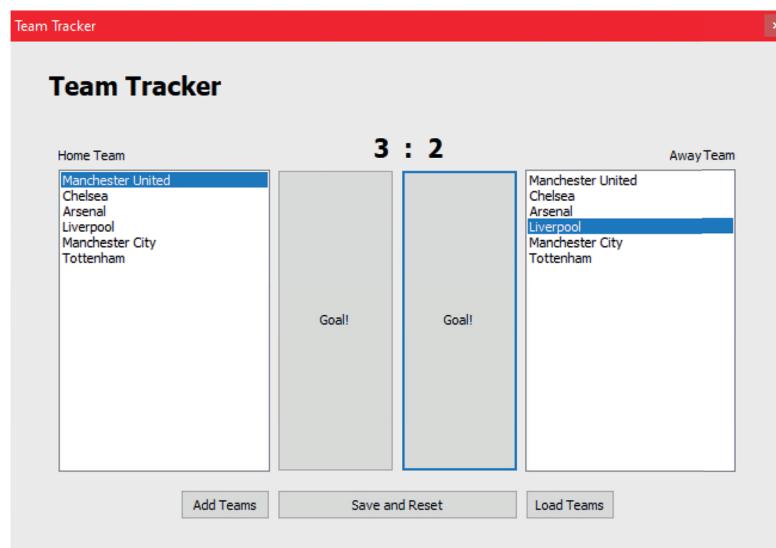
- In the OnClick event of the [Load Teams] button do the following:
  - Declare a local variable called *sTempTeam*.
  - Assign the *tTeam* variable to a file called **teams.txt** using the *AssignFile* procedure.
  - Test if the text file **teams.txt** exists and open(*reset*) the file for reading. If the file does not exist output a suitable message and terminate the program.
  - Use a WHILE-DO-loop with the *Readln* method, to read each line of text from the text file and store the result in *sTempTeam*.
  - As you read a line from the text file, add *sTempTeam* to the Home and Away ListBoxes (*lstHome*, *lstAway*).
  - Enable the *btnHome* and *btnAway* buttons.



### Guided activity 5.2

*continued*

- In the `onClick` event for the `btnHome` button and `btnAway` buttons increment the value of `iHomeScore` and `iAwayScore` respectively.
  - Update the `lblScore` label to show the new score (using the `iHomeScore` and `iAwayScore` variables).
- Save and test your application.
  - Select a team from `lstHome` ListBox
  - Select a team from `lstAway` ListBox.
  - Click on `btnHome` and `btnAway` to increment the scores of the participating teams. You can decide how many times to click the respective buttons, which will set the closing score of the match. In the screenshot below, the user clicked the `btnHome` button three times and the `btnAway` button two times.



## APPENDING DATA

You can add data to an existing file. Data is normally added to the end of the file. You need to position the file pointer to the end of the text file before writing to the file. The file pointer can be moved in one of two ways:

- Looping through the contents of the file until the EOF marker is reached
- Using the `append` procedure. The `append` procedure opens an existing file for writing, sets the file pointer to the end of the file and allows you to add data to the file.



### New words

**append** – to open an existing file for writing, set the file pointer to the end of the file and allows you to add data to the file

### Example 5.2 Team Tracker adding teams to an existing file

The application will now enable users to add their own team names to the existing 'teams.txt' file. Open your **TeamTracker\_p** project in the 05 – Team Tracker Folder and do the following:

- In the `OnClick` event of the [Add Teams] button write code to do the following:
  - Create a local variable called `sTeamName`.
  - Use an input box to ask the user to enter a team name and store the value in the `sTeamName` variable.
  - Add the value of `sTeamName` to both the ListBoxes.
  - Open the `tTeams` file using the `append` procedure.
  - Write the value of `sTeamName` to the `tTeams` file using the `WriteLn` procedure.
  - Close the `tTeam` file.

### Example 5.2

### Team Tracker adding teams to an existing file *continued*

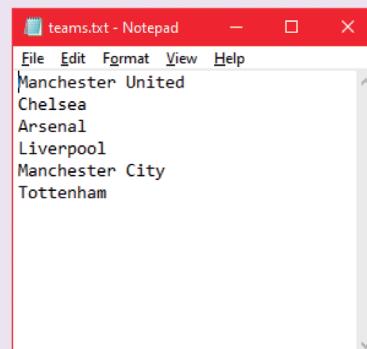
The code for saving to an existing file should now look as follows:

#### Writing to file

```
procedure TfrmTeamTracker.btnAddClick(Sender: TObject);
var
  sTeamName : String;
begin
  sTeamName := InputBox('New team','Enter the name of your team:','');
  lstHome.Items.Add(sTeamName);
  lstAway.Items.Add(sTeamName);
  Append(tTeams);← Opens an existing file for writing and allows you to add data to the end of the file
  WriteLn(tTeams, sTeamName);
  CloseFile(tTeams);
```

This code will add every team name entered by the user to both the ListBoxes as well as to the **teams.txt** file.

- Save and test your application. Once you have added a few teams, open the text file on your storage medium to verify that the teams have been added to the file.



### Example 5.3

### Team tracker appending results

To see how you can save the data from your Team Tracker application to file, work through the following example

- Open the existing **TeamTracker\_p** project.
- Declare the following global variables:

#### Team tracker global variables

```
tTeamTracker: TextFile;
```

- In the OnClick event of the [Save and Reset] button write code to do the following:
  - Assign the *tTeamTracker* variable to a file called **results.txt** using the *AssignFile* method.
  - Test if the file **results.txt** exists. If the file **does not** exist, a new file should be created.

#### If FileExists condition

```
If not FileExists('results.txt') then
begin
  Rewrite(tTeamTracker);
  WriteLn(tTeamTracker, 'home team,away team,home score,away score');
end
```

### Example 5.3 Team tracker appending results *continued*

- if the text file **exists** do the following:
  - Declare two local variables called *sHomeName* and *sAwayName*.
  - Set the value of *sHomeName* to equal the selected item in the *lstHome* ListBox.
  - Set the value of *sAwayName* to equal the selected item in the *lstAway* ListBox.
  - Use the *Append* procedure to open the text file. This will ensure that any new lines of text you write to the file are added to the end of the file, rather than overwriting the existing file.
  - Use the *WriteLn* procedure to add the name of the home team, name of the away team, home score and away score, which are all stored in their respective variables (separated by commas), to the text file in the following format:

```
<home team name>,<away team name>,<home score>,<away score>
```

- Close the file using the *CloseFile* procedure.

The code for saving to the file should now look as follows:

#### Writing to file

```
else
begin
    sHomeName := lstHome.Items[lstHome.ItemIndex];
    sAwayName := lstAway.Items[lstAway.ItemIndex];
    Append(tTeamTracker);
    WriteLn(tTeamTracker, sHomeName + ',' +
        sAwayName + ',' + IntToStr(iHomeScore) + ',' +
        IntToStr(iAwayScore));
    CloseFile(tTeamTracker);
end;
```

- The next step is to **reset** your application back to the starting point, so that a new score can be added to the text file. To do this:
  - Set the values of the *iHomeScore* and *iAwayScore* variables to 0 and clear the *lblScore* label's caption.
  - Add the following two lines of code to *btnSaveAndReset* button's event.

#### Deselect ListBoxes

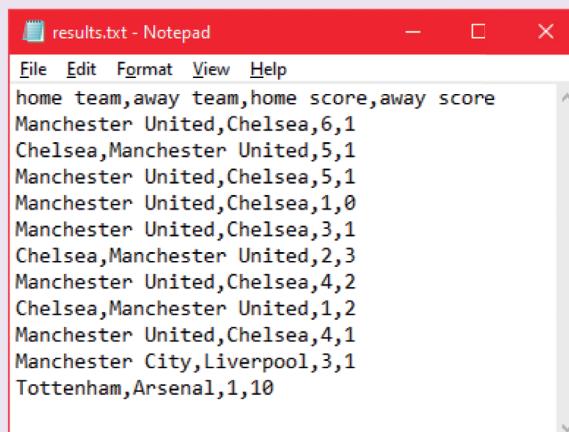
```
lstHome.ItemIndex := -1;
lstAway.ItemIndex := -1;
```

### Example 5.3

#### Team tracker appending results *continued*

By changing the index of the selected item to -1, you are telling the ListBox to deselect all items. As a result, each time you press the [Save and reset] button, you will save the data to your Text File before resetting the application to its starting position.

- Save and test your application. Using the application, add scores for a few matches to your Text file.



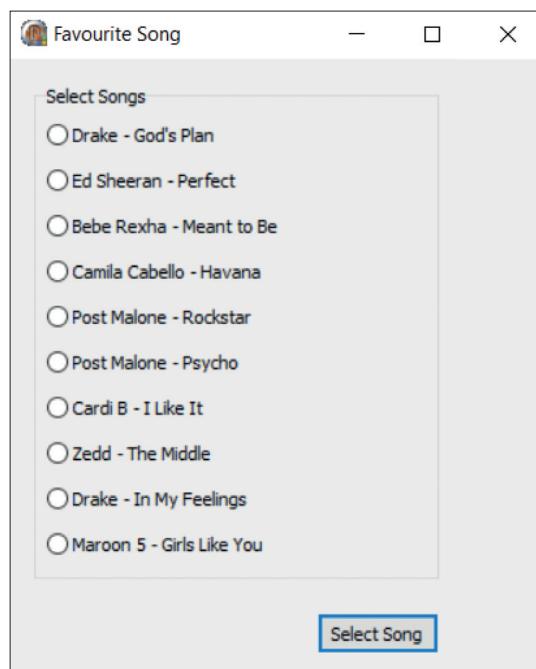
The Text file will be saved in the same folder as the executable file for your application.



### Activity 5.5

- 5.5.1** Open the **FavouriteSong\_p** project from the 05 – Favourite Song Folder. Create an *OnClick* event for the [Select Song] button to write code to do the following:

- Select your favourite song name from the Radio Group and add the name of the song to the end of text file **FavouriteSong.txt**.
- Your application should ensure that:
  - The user has made a selection from the Radio Group.
  - The text file exists before attempting to write to it.
- Save and run your program.



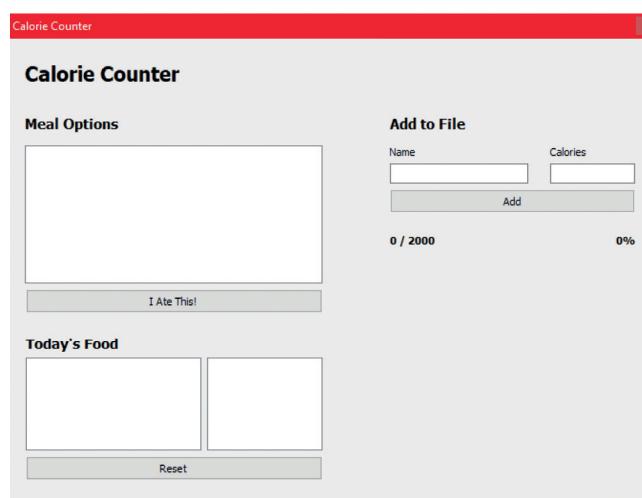


## Activity 5.5 *continued*

**5.5.2** One of the best ways for people to lose weight is to count the calories they eat. If they eat fewer calories than they burn in a day, they should lose weight. You have decided to help people lose weight by creating a calorie counting application. The calorie counter will consist of four sections:

- A section to allow users to add the names of foods, as well as the calories they contain, to an existing text file.
- A section that reads the text file and displays the meal options to the user. The user can then select a meal and inform the program that they ate the meal.
- A section that displays all the meals eaten in a day, as well as the calories consumed per meal.
- A graph that shows how the food eaten per day compares to the daily limit of 2000 calories.

Open the project **CalorieCounter\_p** from the 05 – Calorie Counter Folder. The following interface will display when the program is run.



- A list of available foods and their calories are stored in a text file **Food.txt** in the project folder.
- The text file contains delimited data in the following format:

<Food Name>, <Calories>

The comma (,) acts as a delimiter, separating the food name from the calories.

The first three lines from the text file:

BigMac Burger,600  
Medium Pizza,1800  
Green Salad,50

- To display the graph, a *TImage* component will be used and 6 jpeg images are provided in the project folder. These images contain graph values in steps of 20s.

IMAGE FILE NAME	RANGE OF CALORIE CONSUMPTION (%)
0.jpg	0..19
20.jpg	20..39
40.jpg	40..59
60.jpg	60..79
80.jpg	80..99
100.jpg	> 99



## Activity 5.5 *continued*

- d. In addition, some data structures have been declared for you. These data structures and their purposes are listed below:

DATA STRUCTURE	PURPOSE
arrFood	A global array of type String which can hold a maximum of 50 items. This array will hold the names of the various food items extracted from the text file.
arrCalories	A global array of type Integer which can hold a maximum of 50 items. This parallel array will hold the calories of the various food items in arrFood.
iMax	A global Integer variable which will be used to track the number of elements in arrFood and arrCalories.
iTotalCalories	A global Integer variable which tracks the total calories a user has consumed.

- e. Write code to add the functionality described for each of the events listed below.

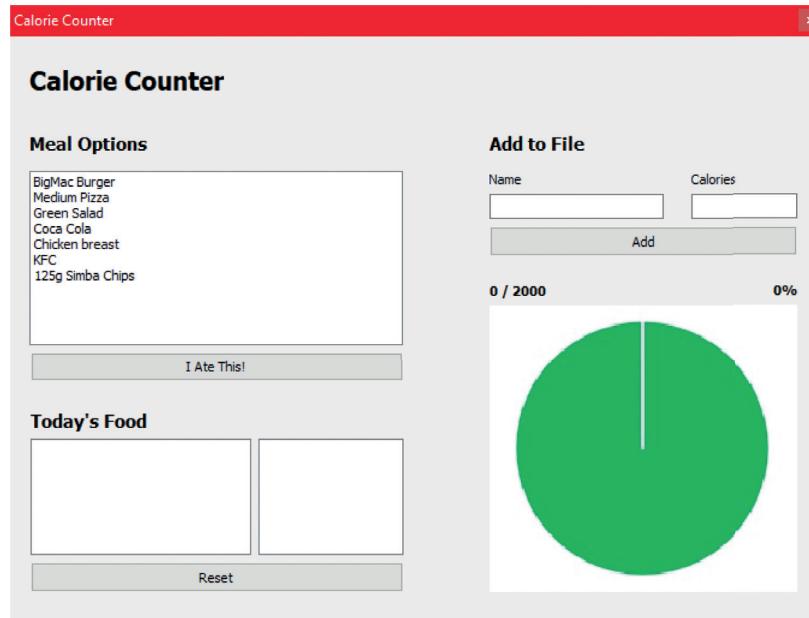
### Form Show

- Connect to the text file Food.txt, opening the text file for reading.
- Loop through the file, reading each line and processing it, extracting the food item's name and its calorie value.
- Add the food item name to the *lstOptions* ListBox.
- Increment the *iMax* variable and use it to add the food item name to the *arrFood* array and its corresponding calorie value to the *arrCalories* array.
- Load image file 0.jpg into the *imgGraph* image component.

- To load an image into an Image component, we use the following code:

```
imgGraph.Picture.LoadFromFile('0.jpg');
```

- Save and test your application.





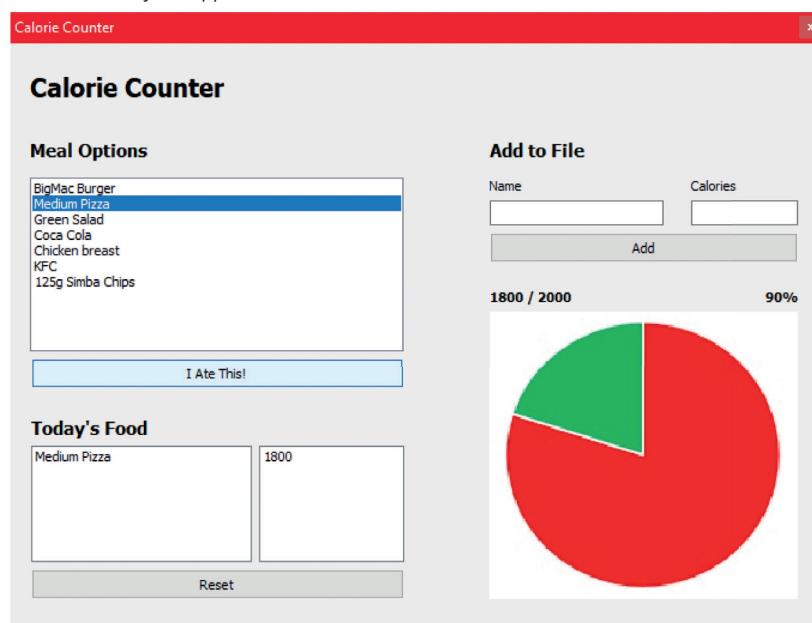
## Activity 5.5 *continued*

### [I Ate This!] button

- Extract the Item Index of the food item selected from the *IstOptions* ListBox.
- Use the Item Index and the *arrFood* array and *arrCalories* array to add the food item and its corresponding calorie value to ListBoxes *IstFoodNames* and *IstCalories* respectively.
- Increment the variable *iTotalCalories* by the calories consumed (the calories should be extracted from the *arrCalories* array).
- Calculate the percentage of calorie consumption using the following equation:

$$\text{Percentage(%)} = (\text{iTotalCalories} / 2000) * 100$$

- Based on the range of the percentage (%), load the relevant graph image. Refer to the graph image filenames and ranges in the table, in the introduction to this activity.
- Update the Calorie Counter in the *lblCalories* label and the Percentage Counter in the *lblPercentage* label.
- Save and test your application.



### [Reset] button

- Clear all items in the *IstFoodNames* and *IstCalories* ListBoxes.
- Assign *iTotalCalories* to 0.
- Load image file '0.jpg' into the *imgGraph* image component.
- Display '0 / 2000' in the *lblCalories* label and '0%' in the *lblPercentage* label.
- Save and test your application.

### [Add] button

- Extract data from the *edtFoodName* and *edtCalories* EditBoxes.
- Add the extracted data to the text file Food.txt using the Food Name and the Calories with the comma separating delimiter.

<Food Name>, <Calories>

- Add the Food Name to the *IstOptions* ListBox.
- Increment the value of *iMax* and add the Food Name to the *arrFood* array and the calories to the *arrCalories* array.
- Save and test your application.

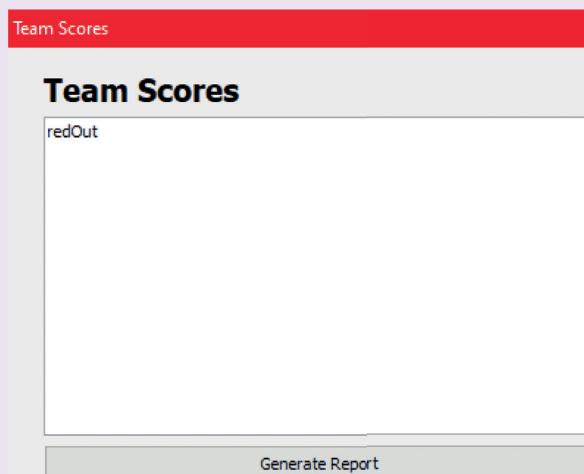
## 5.4 Creating reports

You can use the data stored in text files to produce meaningful reports. Example: Users vote for their favourite song which is then stored in a text file. From the data in the text file, you can determine the song of the year or the top three songs, how many people voted for each song and so on.

### Example 5.4 Team Scores report

For this project, you will read the data you created in your Team Tracker project. To do this, open the **TeamScores\_p** project from the 05 – Team Scores Folder.

The following user interface is provided:



- The folder contains a text file **results.txt** with delimited data formatted as follows:

```
<home team>,<away team>,<home score>,<away score>
```

The comma acts as a delimiter and there are three delimiters in each line.

For example:

Manchester United,Chelsea,6,1

- Manchester United is the home team
- Chelsea is the away team
- 6 was the home team's score
- 1 was the away team's score

- Create an OnClick event for the [Generate Report] button and write code to do the following:
  - Assign a Text File variable *tScores* to the physical file '**results.txt**'.
  - Create a conditional statement that checks if '**results.txt**' does not exist.
  - If the file does not exist, display an error message and close the application.
  - If the file exists: Open the *tScores* file for reading, by using the *Reset* procedure
  - Use a while-loop to read each line from the text file and split the line to display the home team's name, the scores, and the away team's name, separated by tab spaces in the RichEdit Box *redOut*. You will have to set Paragraph tab spaces accordingly.

#### Example 5.4 Team Scores report *continued*

- Save and run your application. You should now be able to read the results from your **results.txt** file and display them in your application.

HOME	SCORE	AWAY
Manchester United	6-1	Chelsea
Chelsea	5-1	Manchester United
Manchester United	5-1	Chelsea
Manchester United	1-0	Chelsea
Manchester United	3-1	Chelsea
Chelsea	2-3	Manchester United
Manchester United	4-2	Chelsea
Chelsea	1-2	Manchester United
Manchester United	4-1	Chelsea
Manchester City	3-1	Liverpool
Tottenham	1-10	Arsenal

**Generate Report**

Congratulations, you have just used all the techniques you learned in Grade 11 to create this report.

While the application may not look like much at this time, the techniques used to read the data and create this application form the basis of most applications working with data, from data analysis tools to music players and games.



#### Activity 5.6

- 5.6.1** Open the **ShopSales\_p** project from the 05 – Report Shop Sales Folder.

The data files provided in a text file named **Sales.txt** which contains delimited text in the following format:

**<PRODUCT>#<SALES>**

The # symbol acts as a delimiter, separating the product name from its sales.

**For example:**

**Bottled Water#32**

- The product name is: Bottled Water
- The sales for Bottled Water is: 32

In the *OnClick* event of [Load Data] button write code to do the following:

- Test whether file **Sales.txt** exists or not.
- If **Sales.txt** exists:
  - Assign and open the file for reading.
  - Loop through the text file, reading each line and processing the delimited text.
  - Display the product name followed by a tab space and then the product sales.
  - Calculate and display the total number of products sold.
  - Calculate and display the average number of products sold.

Product	Sales
Bottled Water	72
Fizzy Drink	45
Small Chips	88
Pie	53
Energy Drink	93
Chewing Gum	27
Fruit Juice	21
Burger	82
Hot Dog	30
Vitamin Juice	72

Total Sales: 583  
Average Sales: 58.3



## Activity 5.6 *continued*

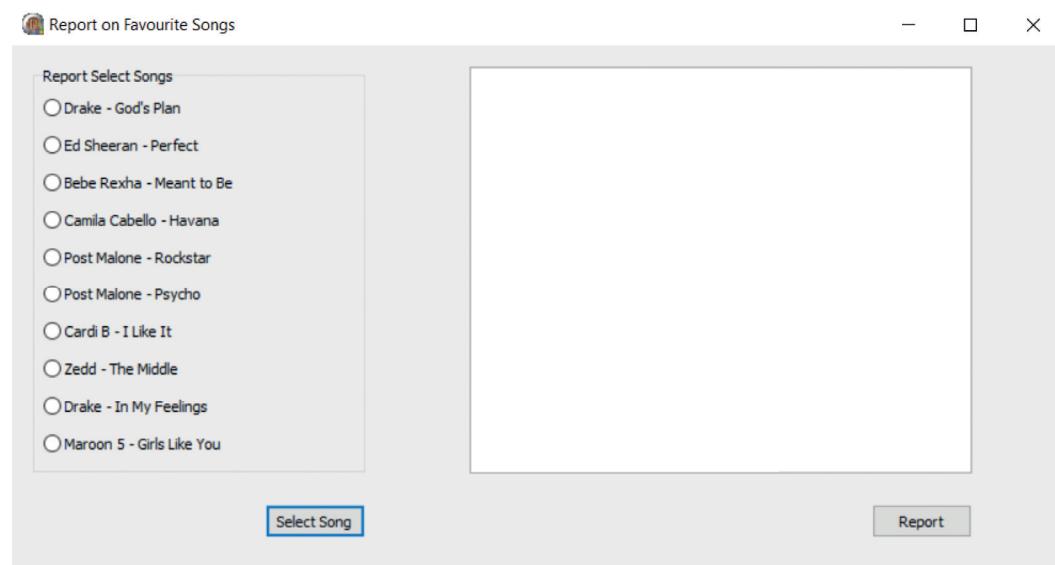
**5.6.2** Open the **ReportFavouriteSong\_p** project from the 05 – Report on Favourite Songs Folder:

The given code for the [Select Song] button allows users to vote for their favourite song.

There are 10 possible songs. Every time a user votes for a song, the song's name is added as a new line to the text file **FavouriteSong.txt**.

Create an *OnClick* event for the [Report] button which will:

- Count how many times each song name appears in the text file **FavouriteSong.txt**.
- Create a report that shows the 10 song names and the number of times each song was voted for.
- Save and run the program.



**QUESTION 1**

Open the project **GamingWebsite\_p** from the 05 – Gaming Websites Folder.

An advertising agency is tracking the popularity of various gaming websites.

Data on the websites has been captured in a text file named **sites.txt** with delimited text formatted as follows:

```
<SiteName>#<AverageRating>#<TotalRating>#<NumberOfRatings>
```

The hash(#) symbol acts as a delimiter.

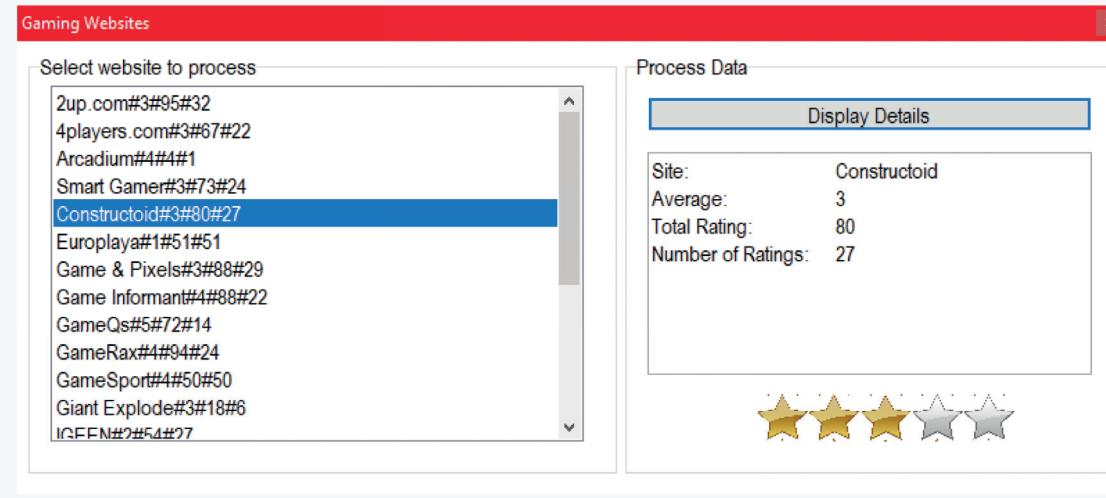
```
Constructoid#3#80#27
```

**1.1** In the form's *OnShow* event write code to:

- Test if the file **sites.txt** exists. If it exists, connect to, and open the file for reading.
- Loop through the text file, reading each line and adding it to the *lstSites* ListBox.  
In the *OnClick* event of the [Display] Button write code to:
  - Extract the selected item from the *lstSites* ListBox.
  - Process the extracted data, separating the delimited data.
  - Display the Site Name, Average, Total and Number of Ratings in separate lines in the *redOut* RichEdit component.

Six image files are provided in the project folder holding images representing ratings ranging from 0 stars to 5 stars. For example, the file name for the image with 5 stars is **5.jpg**.

Use the Average Rating to generate a file name by combining the rating with '.jpg'. Use this filename to load the appropriate image to the *imgRating* Image Box.



**QUESTION 2**

Open project **WebsiteUsers\_p** from the 05 – Website Users Folder.

Details on registered users for a popular website are stored in a text file **Users.txt**.

Storing data into this text file without any security would put the system at risk.

As a result, a user's name is **obfuscated** (hidden) by inserting dummy data between the actual data.

The actual data is found at the even positions of the provided text. For example: FTBaYIHoKnU represents the name Talon

F	T	B	a	Y	I	H	o	K	n	U
1	2	3	4	5	6	7	8	9	10	11

First 3 lines from the Text File:

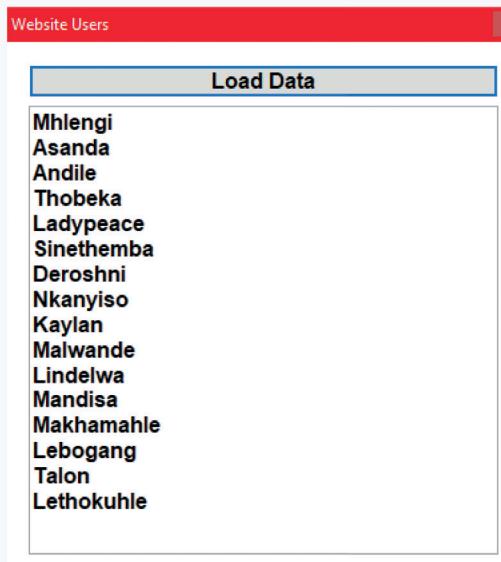
```
rMMhUl0eCnDgKiB
SAWsIaNnCdHaI
VABnOdKiF1VeK
```

**New words**

**obfuscated** – the deliberate act of creating source or machine code that is difficult for humans to understand

In the OnClick event of the [Load Data] button write code to:

- Check if the **Users.txt** text file exists. If the file does not exist, display an error message and terminate the program.
- Connect to the file and open it for reading.
- Loop through the text file, extracting a single line from the file each time the loop runs.
- Process the extracted line, copying only the characters at even positions in the extracted line.
- Display the processed line (decrypted) in the *redOut* RichEdit Box.

**QUESTION 3**

Open project **StaffLogin\_p** from the 05 – Staff Login Folder.

A business requires a login system for employees when they interact with a work computer. Details on users have been captured in a text file named **staff.txt**. The text file contains delimited data formatted as follows:

```
<NAME>#<ID NUMBER>#<PASSWORD>
```

## CONSOLIDATION ACTIVITY

### Chapter 5: Text files *continued*

For example:

Mike#9908045421087#ke

- Mike is the staff member's name
- 9908045421087 is the staff member's ID Number
- ke is the staff member's password

For convenience, all passwords have been set to the last two letters of the staff member's name.

**Note:** all staff members were born before the year 2000.

The following data structures have been declared globally:

DATA STRUCTURE	TYPE	PURPOSE
arrName	Array [1..50] of String	Stores names of employees
arrPassword	Array [1..50] of String	Parallel array to <i>arrName</i> , storing employee passwords
arrGender	Array [1..50] of Char	Parallel array to <i>arrName</i> and <i>arrPassword</i> : stores gender of employees – M for Male and F for Female
arrAge	Array[1..50] of Integer	Parallel array to <i>arrName</i> , <i>arrPassword</i> and <i>arrGender</i> . Stores age of employees
iMax	Integer	Stores the number elements in the arrays
CURRENTYEAR	Integer constant	Stores the current year – This value should be changed in its declaration from 2019 to the current year, if necessary

#### Explanation of the SA ID Number:

Example: 9908045421087

- 99 is the year of Birth
- 08 is the month of Birth
- 04 is the day of Birth
- Digits 7–10 indicate whether the ID belongs to a Male or Female. If the number  $\geq 5000$ , the ID belongs to a male. If the number  $< 5000$ , the ID belongs to a female.

In this case, the gender code is 5421 meaning that this ID belongs to a male.

#### 3.1 In the Form's *onShow* event write code to:

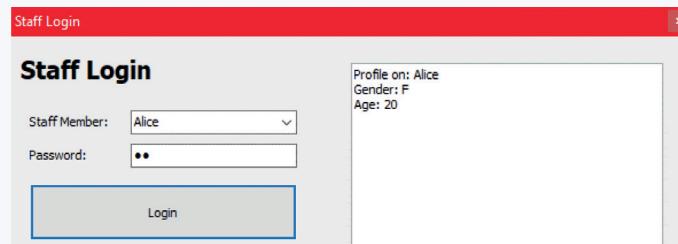
- Connect to and open text file **staff.txt** for reading.
- Loop through the file, reading each line.
- Process each line, separating the name, ID number and Password.
- Increment the value of the *iMax* variable.
- Add the user's name to the *arrName* array.
- Add the user's password to the *arrPassword* array.
- From the user's ID Number:
  - Determine their Gender and store either 'M' or 'F' in the relevant index position of the *arrGender* array.
  - Calculate the user's age by determining the difference between the year value from their ID Number and the constant CURRENTYEAR. Store the user's age in the relevant index position of the *arrAge* array.
- Add the user's name to the *cmbLogin* ComboBox.
- After the loop terminates, close the text file.

3.2 In the *OnClick* event of the [Login] button write code to:

- Extract the Item Index from the *cmbLogin* ComboBox to determine the index position (in *arrName*) of the user attempting to login.
- Compare the password extracted from the *edtPassword* EditBox with a password from the *arrPassword* array (the Index position extracted from *cmbLogin* can be used to determine which element from *arrPassword* is tested). If the input password matches the corresponding password in *arrPassword*, display the user's details in the *redOut* RichEdit box (see the screenshot below for the format).
- If the passwords do not match, display an appropriate message.
- Furthermore, management requires that a log be kept of everyone who logs in.

When a user successfully logs in:

- Check whether text file **Log.txt** exists. If it does not exist, create a new file using the Rewrite procedure. If it exists, open it for writing using the Append procedure.
- Add the user's name (who logged in successfully) as a new line in the **Log.txt** file.
- Close the file.

3.3 In the *OnClick* event of the [Register] button write code to:

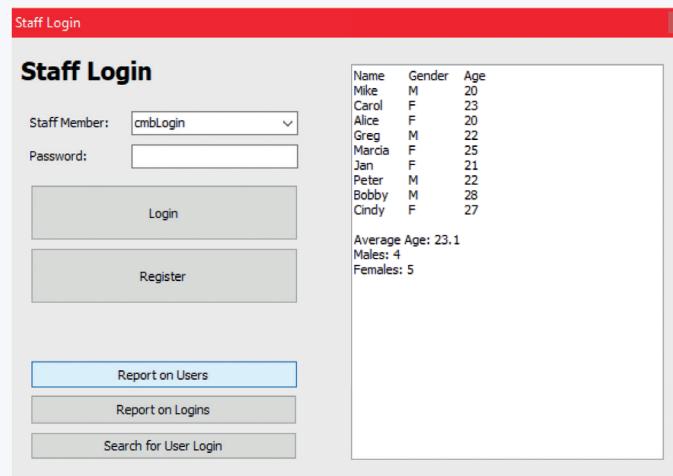
- Use InputBox dialogs to prompt the user to enter their name, ID Number and Password.
- Add the input data to the text file **staff.txt**, formatted with the '#' delimiter separating the data in the following format:

<NAME>#<ID NUMBER>#<PASSWORD>

**Note:** Due to the design of the program, you'll have to terminate the application and run it again for the new name to appear in the ComboBox.

3.4 In the *OnClick* event of the [Report on Users] button write code to:

- Display the contents of the arrays *arrNames*, *arrGender* and *arrAge* in neat columns, with suitable headings.
- Calculate and display:
  - The average age of all staff members.
  - The number of male staff members and the number of female staff members.



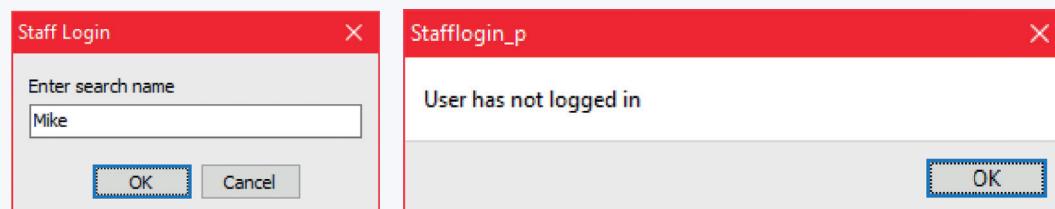
- 3.5 In the *OnClick* event of the [Report on Logins] button write code to:  
Open file **log.txt** text file for reading. Loop through the file, displaying all logins.  
Count and display the total number of logins.

```
Marcia
Peter
Greg
Bobby
Bobby
Alice

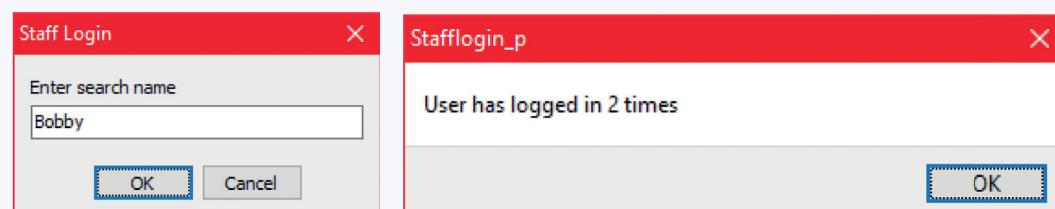
6 logins in total
```

- 3.6 In the *OnClick* event of the [Search for User Login] button write code to:
- Prompt the user to input the name of a staff member using a DialogBox.
  - Loop through the **log.txt** text file to count the number of times the staff member has logged in.
  - If the staff member has never logged in, display a DialogBox with the message 'User has not logged in'.
  - If the staff member has logged in, display the number of times their name appears in the log file using a DialogBox.

#### Sample Data 1



#### Sample Data 2



# USER-DEFINED METHODS

## CHAPTER UNITS

- Unit 6.1 User-defined methods
- Unit 6.2 Procedures
- Unit 6.3 Functions
- Unit 6.4 Basic input validation techniques



## Learning outcomes



At the end of this chapter you should be able to:

- define and describe user-defined methods
- give the structure of a function and a procedure
- differentiate between a function and a procedure
- use functions and procedures to solve problems
- explain the relationship between actual parameters and formal parameters
- explain how value parameters work
- perform basic input validation using code.

## INTRODUCTION

You learned that a method is a subprogram (small piece of code) written to perform a specific task. The string and mathematical methods that you learned about in previous chapters are built-in (or pre-defined) methods found in Delphi.

In this chapter, you are going to focus on *user-defined methods*. *User-defined* methods are written by the user to perform a specific task. Just like built-in methods, *user-defined* methods can be defined and used anywhere in a program.

## 6.1 Introduction to user-defined methods

Delphi has a large number of built-in methods. In the previous chapters you worked with mathematical and String methods such as *Random*, *Copy* and *Inc*. You also used some methods of components such as *Clear* and *SetFocus*. You also learnt about data conversion methods such as *IntToStr* and *StrToFloat*. By calling one of these methods, you can use a single line of code to complete a task that would normally take you multiple lines to do without the method.

Methods make a program:

- Modular – a program is broken into simpler subtasks and these subtasks are kept as separate modules. The use of modular programming structure enhances the accuracy and clarity of a program.
- More readable – It divides a program into smaller and more understandable tasks.
- Easier to debug.
- Easier to update.
- Less repetitive – it can shorten code, that is, if a method is called several times, it saves repetition of code.
- Simpler to understand.
- More efficient.

There are two types of methods:

- procedures
- functions.

The differences between a function and a procedure are listed in the table below.

**Table 6.1: Differences between a function and a procedure**

FUNCTION	PROCEDURE
<p>A call to a function is always within another statement.</p> <p><b>Examples:</b></p> <pre>iNum := StrToInt(edtInput.text); rSquare := sqr(rVal));</pre> <ul style="list-style-type: none"> <li>• where <i>StrToInt</i> &amp; <i>Sqr</i> are function methods</li> </ul>	<p>A call to a procedure is a stand-alone statement.</p> <p><b>Examples:</b></p> <pre>memDisplay.Clear; edtAmount.SetFocus;</pre> <ul style="list-style-type: none"> <li>• <i>Clear</i> and <i>SetFocus</i> are procedure methods of the MemoBox and EditBox respectively.</li> </ul>
<p>A function always returns a single value.</p> <p><b>Example:</b></p> <pre>memoDisplay.lines. Add(FloatToStr(sqrt(25))); rX := Random();</pre> <ul style="list-style-type: none"> <li>• The function <i>sqr</i> calculates the square root of 25 and returns the value of the calculation.</li> <li>• The <i>FloatToStr</i> function converts this value into a string.</li> <li>• The return value from the <i>FloatToStr</i> function conversion is displayed in a MemoBox</li> <li>• A function starts with the keyword, Function.</li> <li>• A function name is assigned a data type.</li> </ul>	<p>A procedure does not return a value through its name.</p> <p><b>Example:</b></p> <pre>ShowMessage('Delphi is fun'); AssignFile(tFile, 'Scores.txt');</pre> <ul style="list-style-type: none"> <li>• The <i>ShowMessage</i> procedure displays text but does not return a value</li> <li>• The <i>AssignFile</i> procedures assigns the logical file <i>tFile</i> to the physical file <b>Scores.txt</b> that is stored on a storage medium.</li> <li>• A procedure starts with the keyword, Procedure.</li> <li>• A procedure name is not assigned a data type.</li> </ul>

## USER DEFINED PROCEDURES AND FUNCTIONS

In addition to using built-in methods, programmers can write their own methods.

**User-defined** methods work in much the same basic way as built-in methods except that the user creates the procedures and functions. However, unlike built-in methods, *user-defined* methods:

- need to be created from scratch by the programmer.
- requires more effort from the programmer to create a method.
- also allows the programmer to create a method that does exactly what he or she wants it to do.

Once the method has been created, it can be used as often as it is needed in an application by simply calling the method's name.



### Activity 6.1

**6.1.1** What is a *user-defined* method?

**6.1.2** What is the difference between a function and a procedure?



### Did you know

All events (such as a button's *OnClick* event) are procedures. To verify this, you can create an *OnClick* event for any button and look at the code. Before the event's name, you will see the keyword, **Procedure**.



### New words

**user-defined** – is methods written by programmers themselves

## 6.2 Procedures

### PROCEDURES WITH NO PARAMETERS

You have already used procedures with no parameters. Examples of procedures with no parameters include the following:

- Randomize
- edtNumber.setFocus
- memDisplay.Clear

*setFocus* is a procedure with no parameters of the EditBox component and *Clear* is a procedure with no parameters of the MemoBox component.

#### Example 6.1    memDisplay.Clear;



- The word *procedure* indicates that the *Clear* method is a procedure.
- The semi-colon(;) immediately after *Clear* indicates that the procedure does not have parameters.

### DECLARING AND DEFINING A PROCEDURE WITH NO PARAMETERS

The procedure is declared as a method of the form as follows:

```
private
    procedure NameOfProcedure;
```

The procedure is defined in the body of the program as follows:

```
procedure <className>.<NameOfProcedure>;
var .... //variables with local scope
begin
    // body of the procedure
end;
```

#### Example 6.2

Below is code for the **ThreeNumbers\_p** project that contains the following:

- A procedure called **SumOfNumbers** that determines and displays the sum of three numbers
- A procedure called **Highest** that determines and display the highest of three numbers
- A procedure called **Line** that displays a line made up of the '=' symbol

### Example 6.2 *continued*

- [Determine] button: that reads three numbers and calls procedures **SumOfNumbers**, **Line** and **Highest**

```
unit ThreeNumbers_u;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, StdCtrls, ComCtrls;
type
  TfrmThreeNumbers = class(TForm)
    redResults: TRichEdit;
    btnDetermine: TButton;
    edtNum1: TEdit;
    edtNum2: TEdit;
    edtNum3: TEdit;
    lblNum1: TLabel;
    lblNum2: TLabel;
    lblNum3: TLabel;
    procedure btnDetermineClick(Sender: TObject);

private
  { Private declarations }
  procedure SumofNumbers; } ← The declaration of the procedures
  procedure Highest; } ← The procedures are in the Private section because
  procedure Line;   they will only be assessed within this unit.

public
  { Public declarations }
end;
var
  frmThreeNumbers: TfrmThreeNumbers;
  iNum1, iNum2, iNum3: Integer; ← iNum1, iNum2 and iNum3 are declared non-locally (globally)
                                because three different procedures need to access the values.
  sDisplay: string;
implementation
{$R *.dfm}

procedure TfrmThreeNumbers.btnDetermineClick(Sender: TObject);
begin
  iNum1 := StrToInt(edtNum1.Text);
  iNum2 := StrToInt(edtNum2.Text);
  iNum3 := StrToInt(edtNum3.Text);
  SumOfNumbers;
  Line;
  Highest;
  Line;
end;
```

Procedure calls  
A procedure is called (activated) by its name. Control is transferred from the calling statement to the procedure. Once the procedure execution is complete, control is transferred to the next statement after calling the statement.

### Example 6.2 *continued*

```
procedure TfrmThreeNumbers.Highest; ← Name of the Procedure
var iLarge:Integer; ↑ Name of the Class appears before the procedure name. This indicates
begin that the procedure belongs to the class (form TfrmThreeNumbers).
    if (iNum1 > iNum2) and (iNum1 > iNum3) then
        iLarge := iNum1
    else
        if iNum2 > iNum3 then
            iLarge := iNum2
        else
            iLarge := iNum3;
    sDisplay := 'Highest Number: '+ IntToStr(iLarge);
end;

procedure TfrmThreeNumbers.SumofNumbers;
var iSum:Integer;
begin
    iSum := iNum1 + iNum2 + iNum3;
    sDisplay := 'Sum: '+ IntToStr(iSum);
end;
procedure TfrmThreeNumbers.Line;
begin
    redResults.Lines.Add(sDisplay);
    redResults.Lines.Add('=====');
end;

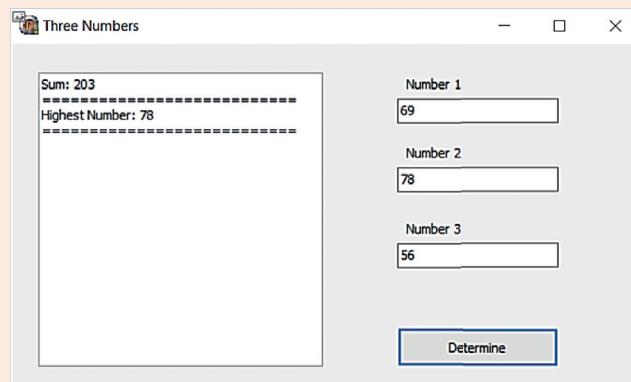
end.
```

User defined procedure *Highest*

User defined procedure *SumOfNumbers*

User defined procedure *Line*

When the program is executed, it will display the following:



#### Notes:

- In the example, because *iNum1*, *iNum2* and *iNum3* are declared non-locally (globally), their values can be changed from any procedure in the program.
- The procedure *Line* is called twice in the procedure **Determine**.
- You can type the procedure or alternatively let Delphi quickly create a procedure framework for you.

## Example 6.2 *continued*

- Once you declare a procedure, you can easily create a framework for the procedure in the following manner:
  - Place your cursor anywhere in the name of the procedure in the procedure declaration statement.

For example:

Private

Procedure Smallest;

← Click anywhere in the name of the procedure

- Press <Shift> + <Ctrl> + <c> simultaneously. The following framework will appear:

```
procedure TfrmThreeNumbers.Smallest;
begin
end;
```

- If you don't use the class name before the procedure name, you cannot access the components of the class as you would normally do.

```
procedure TfrmThreeNumbers.SumofNumbers;
var iSum:Integer;    ↑
begin
  iSum := iNum1 + iNum2 + iNum3;
  TfrmThreeNumbers.redResults.Lines.Add('Sum: ' + IntToStr(iSum));
end;
```

↑ If the form name is removed from the procedure definition, then the form name has to be added before all components in the body of the procedure



## Activity 6.2

- 6.2.1** Indicate whether the statement below is true or false:

'Non-local (global) variables values can be changed in a procedure.'

- 6.2.2** Explain and correct the following error:

"undeclared identifier:SumOfNumbers" error when the *SumOfNumbers* procedure is called. The *SumOfNumbers* procedure has been correctly defined.

```
procedure TfrmThreeNumbers.btnCloseClick(Sender: TObject);
begin
  iNum1:=StrToInt(edtNum1.Text);
  iNum2:=StrToInt(edtNum2.Text);
  iNum3:=StrToInt(edtNum3.Text);
  SumOfNumbers;
  Highest;
end;
```

- 6.2.3** Open the **ThreeNumbers\_p** project in the 06 – Three Numbers Folder and do the following:

- Add a procedure **Smallest** to determine and display the smallest of the three numbers.
- Add a procedure **SwapColour** to swap the form's current colour from either black to white or white to black, that is, if the form's colour is black, it must change to white and vice versa.
- Create an *onCreate* event to set the forms colour to black.
- Swap the colour of the form using the procedure **SwapColour** as follows:
  - when the [Determine] button is clicked
  - just before the **Determine** procedure is exited.



## Activity 6.2

*continued*

**6.2.4** Open the **Shapes\_p** project in the 06 – Shapes Folder and do the following:

- a. Write a procedure **Square** that will create and display a square shape as follows:

**Square**

```
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

- b. Write a procedure **Triangle** that will create and display a triangle as follows:

**Triangle**

```
*
```

```
**
```

```
***
```

```
***
```

```
****
```

```
****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

- c. Write code for the *OnClick* event for the [Create] button to display a square, triangle and a square as shown below:

```
Square  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```

**Triangle**

```
*
```

```
**
```

```
***
```

```
***
```

```
****
```

```
****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

```
*****
```

**Square**

```
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****  
*****
```



## Activity 6.2 *continued*

**6.2.5** Open the **UniqueNumbers\_p** project in the 06 – Unique Numbers Folder and do the following:

- a. Write a procedure **Generate** to randomly generate 10 unique values in the range 10 to 99 and store the values in *arrNumbers*.
- b. Write a procedure **Display** to display the elements of *arrNumbers* horizontally.
- c. Write a procedure **Sort** to sort the elements of *arrNumbers* in descending order.
- d. Write code for the *OnClick* event for the [Determine] button to:
  - generate the unique numbers
  - display the original array with an appropriate message
  - sort the elements in the array
  - display the sorted array with an appropriate message.

## PROCEDURES WITH PARAMETERS

In the previous section you created procedures that could be called without any additional information to complete a specific task. Compare these procedures with a procedure like the **ShowMessage** procedure. When calling **ShowMessage**, you use the following syntax:

### ShowMessage syntax

```
ShowMessage(sOutput);
```

Without providing a value to **ShowMessage**, the procedure cannot display your message.

## DECLARING AND DEFINING A PROCEDURE WITH PARAMETERS

To add parameters to a procedure, you can use the following syntax:

### Procedure with parameters

```
procedure ClassName.ProcedureName(parameterName1 : type1; parameterName2 : type2, ...);  
var  
    var1 : Type;  
begin  
  
    Statement1;  
    Statement2;  
    ...  
    Statement1000;  
end;
```

Formal parameter list  
Note: Semi-colons are used to separate data types

### Notes:

- The procedure with parameter/s is declared as a method of the form as follows:

```
private  
procedure NameOfProcedure(List of formal parameters);  
Example:  
Private  
Procedure ThreeNumbers(iNo1,iNo2,iNo3:integer);
```

- The variable(s) declared next to the *procedureName* in the procedure definition is/are known as **formal parameter**(s). It is declared in the same way as you would declare a variable.
- Formal parameter(s) has local scope.



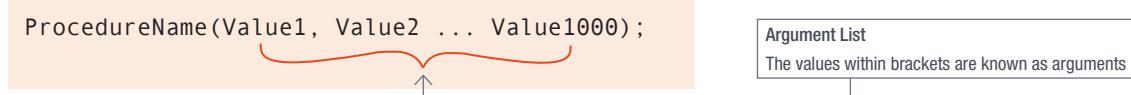
### New words

**formal parameter** – to declare variable(s) next to the procedure name

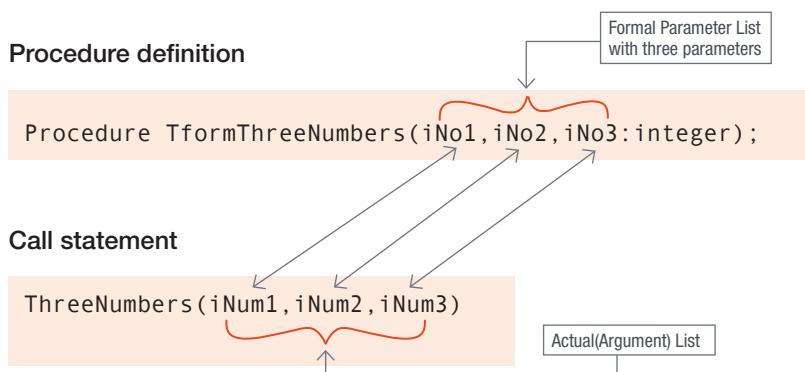
- There are two types of formal parameters:
  - Variable parameters:** Variable parameters are preceded by the keyword VAR. You will learn about variable parameters during your tertiary studies in programming.
  - Value parameters:** This will be discussed later.

## CALLING A PROCEDURE WITH PARAMETERS

To call a procedure with parameters, you indicate the name of the procedure followed by a list of values. For example:



### Relationship between arguments and parameters



#### Notes:

- The values you pass to the procedure must be the same number, order, and of the same type as the parameters (method signature) defined in the formal parameter list:
  - If there are three formal parameters in the formal parameter list, then the call statement must supply three arguments.
  - The data type of each argument value must match the data type of its corresponding formal parameter.
  - The order of the arguments must be in the same order as in the formal parameter list.
- The value of the first argument is assigned to the first formal parameter, the second argument value is assigned to the second parameter and so on.
- Value parameters:** When a procedure is called, memory locations are created for each of the formal parameters and the values of the arguments are assigned to the corresponding formal parameters. Changes made to a value parameter will not affect its corresponding argument. When the procedure is exited, the memory locations of the formal parameters 'die' away.
- Since the formal parameters are value parameters, the arguments in the call statement can be:
  - variables
  - constants
  - expressions.



#### New words

**value parameter** – when a procedure is called, memory locations are created for each of the formal parameters and the values of the arguments are assigned to the corresponding formal parameters. Changes made to a value parameter will not affect its corresponding argument. When the procedure is exited, the memory locations of the formal parameters 'die' away

- Here is an example of call statements:

```

Var x,y,z:integer;
      sLine:string;
      bFlag:boolean;
      ...
Sort(x,y,z);           // arguments are variables
Sort(5,6,3);           // arguments are constants
Sort(x + 1,y-2,z-y);  // arguments are expressions
Sort(x,6,z-y);         // arguments are a combination of a variable, a
                        constant and an expression
Check(true,'Green',6); // arguments are constants of type Boolean,
                        string and integer

```

## METHOD SIGNATURE

Each method (procedure or function) has a **method signature**. The name of the method and its formal parameter list is referred to as the method signature of a method. You can get more than one method with the same name. This is called **method overloading**. In method overloading a procedure is differentiated from another procedure by its method signature.

### Example of a procedure with a parameter

```

procedure TFrmGreetings.SayHello(sWhat:string)
begin
  ShowMessage('Hello ' + sWhat) ;
end;

```



### New words

**method overloading** – to have more than one method with the same name  
**method signature** – to name a method and its formal parameters list

### Note:

- The name of the form is *TfrmGreeting*
- The name of the procedure is **SayHello**
- The procedure has one formal parameter *sWhat* of type String
- The procedure displays the word ‘Hello’ followed by the string *sWhat* that it received. For example, if it received ‘Tarzan’ then it would display ‘Hello Tarzan’.



### Guided activity 6.1

A company needs to provide a quotation to lay grass and to fence a rectangular property. Use the AreaPerimeter\_p project from the 06 – Area Perimeter folder. Do the following:

- Create an *OnFormCreate* event to initialise a variable *rTotalCost* to 0.
- Write a procedure **CalculateCost** that will receive unit cost, the unit of measurement and a message. The message will either indicate ‘Cost to lay Grass’ or ‘Cost to Fence’. The amount payable is calculated by multiplying the unit cost by the unit of measurement. Display the message and the amount payable.
- [Read] button: Prompts the user to input the length and breadth of a property.
- [Lay Grass Cost] button: The area of the rectangular property is calculated and the user is prompted for the unit cost of laying grass per m<sup>2</sup>. The procedure **CalculateCost** is called with the arguments unit cost per square metre, the area and the message ‘Cost to lay Grass’.
- [Fence Property Cost] button: The perimeter is calculated and the user is prompted for the unit cost per metre for the fencing. The procedure **CalculateCost** is called with the arguments unit price per metre, the perimeter and the message ‘Cost to fence’.



### Guided activity 6.1

*continued*

- [Total cost] button: Displays the total cost of laying grass and of fencing the property.

```
unit AreaPerimeter_U;
interface
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls, ComCtrls, ExtCtrls;
type
  TfrmAreaPerimeter = class(TForm)
    redDisplay: TRichEdit;
    btnRead: TButton;
    btnCostGrass: TButton;
    btnCostFence: TButton;
    btnTotalCost: TButton;
    procedure btnReadClick(Sender: TObject);
    procedure btnCostGrassClick(Sender: TObject);
    procedure btnCostFenceClick(Sender: TObject);
    procedure btnTotalCostClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
  private
    { Private declarations }
    procedure CalculateCost(rCost, rMeasurement: real; sMessage: string); Procedure CalculateCost declaration.
  public
    { Public declarations }
  end;

var
  frmAreaPerimeter: TfrmAreaPerimeter; Variables declared non-locally (globally) because they will be used in different procedures.
  rLength, rBreadth, rTotalCost : real; } } ←
implementation
{$R *.dfm} Reads and displays the length and breadth.

procedure TfrmAreaPerimeter.btnReadClick(Sender: TObject);
begin
  rLength := StrToFloat(inputbox('','Enter length of a property',''));
  rBreadth := StrToFloat(inputbox('','Enter breadth of a property',''));
  redDisplay.Lines.Add('Length: ' + #9 + #9 + FloatToStrF(rLength,ffFixed,10,2));
  redDisplay.Lines.Add('Breadth: ' + #9 + #9 + FloatToStrF(rBreadth,ffFixed,10,2));
end;

procedure TfrmAreaPerimeter.btnCostGrassClick(Sender: TObject);
var rArea,rCost:Real;
begin
  rArea := rLength * rBreadth;
  rCost := StrToFloat(inputbox('','Enter cost per square metre of grass',''));
  redDisplay.Lines.Add('Area: ' + #9 + #9 + FloatToStrF(rArea,ffFixed,10,2));
  redDisplay.Lines.Add('Unit price of Grass: ' + #9 + FloatToStrF(rCost,
  ffCurrency,10,2));
  CalculateCost(rCost,rArea,'Cost to lay Grass'); ← Calculates and displays the area, prompts user for cost per square metre and displays this value.
end; } ← Call procedure CalculateCost and sends the values: cost per square metre, area and 'Cost to lay grass'.
```



## Guided activity 6.1 *continued*

```
procedure TfrmAreaPerimeter.btnCostFenceClick(Sender: TObject);
var rPeri,rCost:Real;
begin
    rPeri := 2 * (rLength + rBreadth);
    redDisplay.Lines.Add('Perimeter: '+ #9 + FloatToStrF(rPeri,ffFixed,10,2));
    rCost := strtofloat(inputbox('', 'Enter cost per meter of fencing', ''));
    redDisplay.Lines.Add('Unit Price of Fence:' + #9 + FloatToStrF(rCost,
    ffCurrency,10,2));
    CalculateCost(rCost,rPeri,'Cost to fence'); ← Call procedure CalculateCost and
                                                sends the values: cost per metre,
                                                perimeter and 'Cost to fence'.
end;

procedure TfrmAreaPerimeter.btnTotalCostClick(Sender: TObject);
begin
    redDisplay.Lines.Add('Total Cost: ' + #9 + FloatToStrF(rTotalCost,
    ffCurrency,10,2));
end;

procedure TfrmAreaPerimeter.CalculateCost(rCost, rMeasurement:
real;sMessage:string);
var rAmountPayable: real;
begin
    rAmountPayable := rCost * rMeasurement;
    redDisplay.Lines.Add(sMessage + #9 + FloatToStrF(rAmountPayable, ffCurrency,8,2));
    rTotalCost := rTotalCost + rAmountPayable; ← rAmountPayable is added
                                                to the total cost rTotalCost.
end;

procedure TfrmAreaPerimeter.FormCreate(Sender: TObject);
begin
    rTotalCost := 0; ← The total cost rTotalCost set to 0 when the form is created.
end;

end.
```

Calculates and displays the perimeter, prompts user for cost per metre and displays this value.

Call procedure CalculateCost and sends the values: cost per metre, perimeter and 'Cost to fence'.

Procedure CalculateCost is defined with three formal parameters: *rCost*, *rMeasurement* and *sMessage*. Two parameters are type real and one parameter type string.

The amount payable *rAmountPayable* is calculated. The message received and the amount payable is displayed.

*rAmountPayable* is added to the total cost *rTotalCost*.

The total cost *rTotalCost* set to 0 when the form is created.



## Activity 6.3

- 6.3.1** Answer the following questions in your own words.
- Identify one built-in procedure with parameters and one built-in function without parameters.
  - Identify three built-in procedures without parameters and three built-in functions with parameters.
  - Explain what modular programming is and list three advantages of modular programming.
- 6.3.2** Open the **AreaPerimeter\_p** project in the 06 – Area Perimeter Folder and write code to do the following:
- Add a procedure **Discount** to give a discount of 10% on the total cost if the total cost exceeds R5000. Display the discount and final amount payable.
  - Call procedure **Discount** so that it also displays the discount and final cost payable.

Example of sample runs:

Length:	8.00
Breadth:	3.00
Area:	24.00
Unit price of Grass:	R70.00
Cost to lay Grass:	R2 240.00
Perimeter:	22.00
Unit Price of Fence:	R120.00
Cost to fence:	R2 880.00
Total Cost:	R5 120.00
Discount:	R512.00
Final Amount:	R4 608.00

- 6.3.3** Study the following procedures and identify the mistakes made in each procedure.

### Procedure 1

```
SetFormHeight(iHeight : Integer);
begin
    frmMain.Height := iHeight;
end;
```

### Procedure 2

```
procedure FormSize(iHeight, iWidth, sText);
begin
    frmMain.Height := iHeight;
    frmMain.Width := iWidth;
    frmMain.Color := clWhite;
    lblResult.Caption := sText;
end;
```

### Procedure 3

```
procedure ShowValue(iValue, iNumberOfTimes : Integer, sDescription : String);
var
    i, iNumberOfTimes : Integer;
begin
    for i := 1 to iNumberOfTimes do
        ShowMessage(sDescription + ':' + IntToStr(iValue));
end;
```

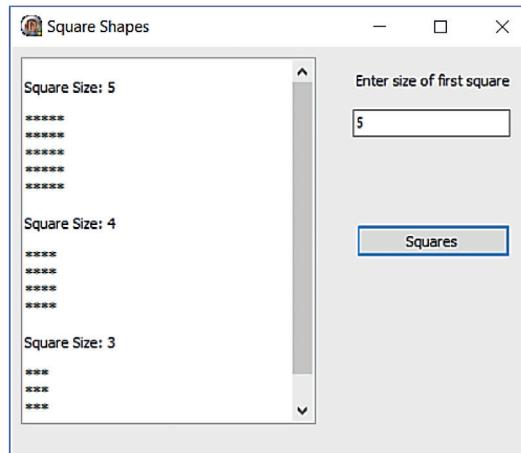


## Activity 6.4

**6.4.1** Open **SquareShapes\_p** project in the 06 – Square Shapes Folder and do the following:

- Write a procedure **Square** that will receive an integer value *iSize* and will draw a square of size *iSize*. Example if *iSize* is 10, then the following square will be drawn:  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*
- Create an *OnClick* event for the [Squares] button that will read the size of the square from the EditBox and store the value in variable *iSquareSize*. Display three squares with each square's size one smaller than the size of the previous square displayed. The size of the first square will be size read from the EditBox.

**Note:** Call procedure **Square** to display the squares.



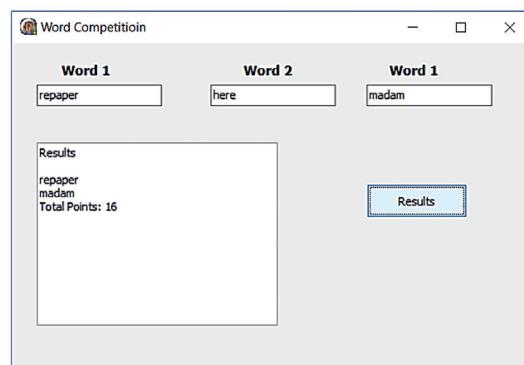
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*  
\*\*\*\*\*

**6.4.2** A word competition is being held at your school. Learners are required to come up with three words that are palindromes. A palindrome is a word that is spelt the same forwards and backwards.

Learners are awarded 1 point per letter of each palindromic word plus an additional 2 points. A prize is awarded if the learner scores 20 points and above.

Open the project **WordCompetition\_p** from the 06 – Word Competition Folder and do the following:

- Write a procedure **Palindrome** that will accept a word and determine whether the word is a palindrome or not. Display the word if it is a palindrome.
- Write a procedure **AwardPoints** that will accept a palindromic word and determine the points awarded to the word. The points are added to a variable *iTotalPoints*.
- Write code for the *OnClick* event for the [Results] button to do the following:
  - Read three words from the EditBoxes and store the words in *arrWords*.
  - Call procedure **Palindrome** to determine whether each word is a palindrome or not.
  - Call procedure **AwardPoints** when calculating the points.
  - Display the number of palindrome words.
  - Display the total points earned.
  - Display whether a learner is awarded a prize.



## 6.3 Functions

Functions work in the same way as procedures – they also perform a specific task and can be called from anywhere in your program.

### Syntax of a function without parameters

```
Function FunctionName: ReturnType;
var
  var1 : Type;
begin
  Statement1;
  Statement2;
  ...
Result := Value of ReturnType; ←
end;
```

A function starts with the keyword **FUNCTION**.

Function name is assigned a data type.

A function always returns a value of the data type in the function header. The value is assigned to a reserved word *Result* OR can be assigned to the function name.

### Syntax of a function with parameters

```
function FunctionName(parameter1 : Type; parameter2 : Type) : ReturnType;
var
  var1 : Type;
begin
  Statement1;
  Statement2;
  ...
Result := Value of ReturnType;
end;
```

### Calling a function

Since a function always returns a value, a call to a function is always within another statement such as:

- an assignment statement
- a selection statement
- an output statement.



### Guided activity 6.2

#### CalculatePower

Write a function **CalculatePower** that accepts two integer numbers *iBase* and *iExponent* and returns the results of *iBase* raised to the power *iExponent*.

#### CalculatePower function

```
function Tnameofform.CalculatePower(iNumber, iExponent : Integer) : Integer;
var
  i, iOutput : Integer;
begin
  iOutput := 1;
  for i := 1 to iExponent do
    iOutput := iOutput * iNumber;
  Result := iOutput
end;
```



## Guided activity 6.2

### CalculatePower continued

Write a function **CalculateFactorial** that returns the factorial of a number. The number is received as a parameter. The factorial (indicated by the ! symbol) of a number N is determined as follows:

$N! = 1 * 2 * 3, \dots, *N$

#### Factorial function

```
function TnameOfForm.CalculateFactorial(iNumber : Integer) : Integer;
var
  i, iOutput : Integer;
begin
  iOutput := 1;
  for i := 1 to iNumber do
    iOutput := iOutput * i;
  Result := iOutput;
end;
```



## Activity 6.5

**6.5.1** Open the **Vowels\_p** project in the 06 – Vowels Folder and do the following:

- Write a function **CountVowels** that receives a word and returns the number of vowels in the word.
- Write a function **RemoveVowels** that receives a word and returns the word with the vowels removed.
- Write code for the [Process] Button. Each element of the array *arrWords* contains a word. For each element, display the word, the number of vowels in the word and the word with the vowels removed. The display must be in tabular form. Call function **CountVowels** and **RemoveVowels**.

Save and run your program.

Word	#Vowels	Vowels Removed
Suddenly	2	Sddnly
Tomorrow	3	Tmrww
Change	2	Chng
Essentially	4	ssntllly
Deadlines	4	Ddlns
Support	2	Spprt
Necessary	3	Ncssry
Nightmare	3	Nghtmr
Stories	3	Strs
Finally	2	Fnlly

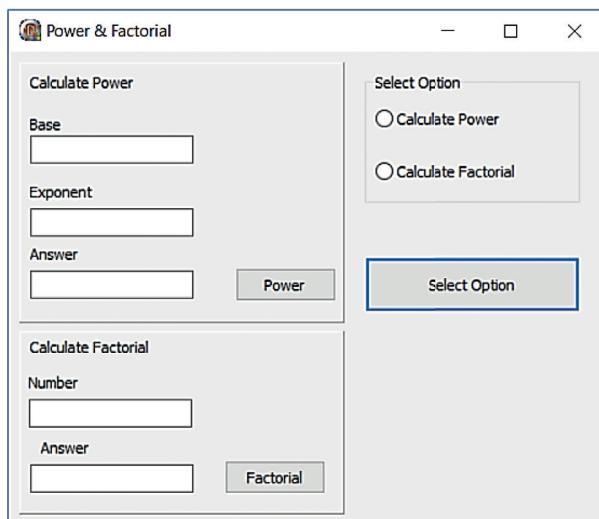
**Process**



## Activity 6.5

continued

- 6.5.2 Open the **PowerFactorial\_p** project in the 06 – Power Factorial Folder and write code to do the following:



a. Write a function **CalculatePower** that accepts two integer numbers *iBase* and *iExponent* and returns the results of *iBase* raised to the power *iExponent*.

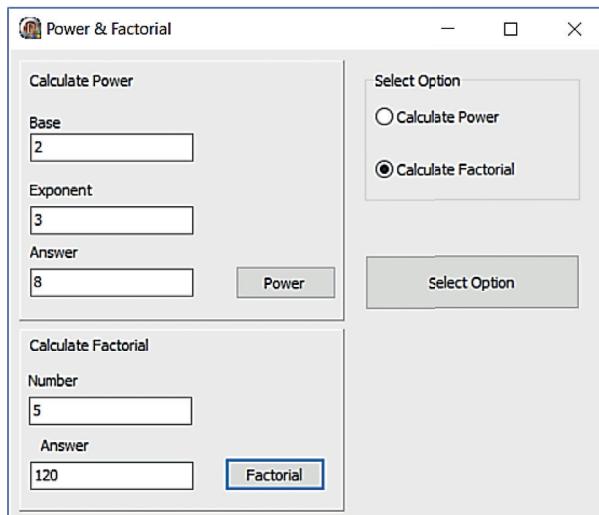
b. Write a function **CalculateFactorial** that returns the factorial of a number:

The number is received as a parameter. The factorial (indicated by ! symbol) of a number *N* is determined as follows:

$$N! = 1 * 2 * 3, \dots, * N$$

- c. [SelectOption] button: Select an option, **Calculate Power** or **Calculate Factorial**. If no option has been selected, then display a message 'No option selected'. If the Calculate Power option is selected then enable the panel to calculate power. If the **Calculate Factorial** option is selected, then enable the panel to calculate a factorial.
- d. [Power] button: Reads the data: the base and exponent from the EditBoxes, calculates and displays the results in the *answer* EditBox. Call the function **CalculatePower** to perform the power calculation.
- e. [Factorial] button: Reads a number from the EditBox, calculates the factorial and displays the factorial in the EditBox. Call the function **CalculateFactorial** to calculate the factorial of a number.

Sample Output after both the [Calculate Power] and [Calculate Factorial] buttons are selected:





## Activity 6.5 *continued*

- 6.5.3** Open the **ImprovedWordCompetition\_p** project in the 06 – Improve Word Competition Folder and write code as follows:
- Write a function **CheckPalindrome** that will accept a word as a parameter and return TRUE if the word is a palindrome; otherwise it will return FALSE.
  - Write a function **AwardPoints** that will accept a word as a parameter and calculates the points a user is awarded. The points are calculated by adding the number of characters in the word plus an additional two points.
  - Write code for the *OnClick* event for the [Results] button and do the following:
    - Read the three words from the edit component and store the words in an array *arrWords*.
    - Call the method **CheckPalindrome** for each word to test whether it is a palindrome or not. If a word is a palindrome, then call the **AwardPoints** to calculate the points awarded for a word.
    - Display the words that are palindromes.
    - Display how many of the words are palindromes.
    - Display the total points *iTotalPoints* obtained by a competitor.

## FOR ENRICHMENT ONLY

### METHODS WITH ARRAY AS PARAMETER

You can pass an array to a method. To use an array as a parameter in a method, you need to declare a new data type as shown below:

```
...
uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs, StdCtrls;

type
  arrGuesses=array[1..9] of Integer; <-- New data type arrGuesses declared above the class definition
  TfrmMake15 = class(TForm)
  ...
  
```

The method is then declared:

```
private
  function isUnique(arrVal:arrGuesses):boolean;
  
```

The method is then defined:

```
function TfrmMake15.isUnique(arrVal: arrGuesses): boolean;
var bFlag:Boolean;
  i,j: Integer;
begin
  bFlag := True;
  for i := 1 to 8 do
    for j := I + 1 to 9 do
      if arrVal[I] = arrVal[J] then
        begin
          bFlag := False;
        end;
  result := bFlag;
end;
```

## 6.4 Basic input validation techniques

For any data input, a validation check is used to ensure that the data received is actually what is required by the application. Validation checks are a safeguard against incompatible values that could cause interruptions in the flow of the program or cause the program or operating system to crash. Validation techniques do not determine whether the data input is accurate. Different validation techniques are used to validate data input.

### DATA TYPE CHECK

You can test whether the data entered is of the required data type using three options. These are data type check for:

- an Integer
- a Real number
- a String with letters and space.

#### DATA TYPE CHECK FOR AN INTEGER

```
function TValidation.isValidInteger(sValue, sFieldName: String): boolean;
Var
  bFlag: boolean;
  iNum: integer;
begin
  bFlag := False;
  try<
    begin
      iNum := StrToInt(sValue);
      bFlag := true;
    end;
  except<
    ShowMessage('Invalid Number for ' + sFieldName + 'Error');
    bFlag := False;
  end;
  result := bFlag;
end;
```

**Exception Handling used.**

- Try Section  
The string is converted to an integer and *bFlag* set to TRUE indicating the conversion was fine.
- Except Section  
This is activated when the conversion from string to integer did not take place correctly. *bFlag* is set to FALSE.

#### DATA TYPE CHECK FOR A REAL NUMBER

```
function TValidation.isReal(sValue, sFieldName: String): boolean;
Var
  bFlag: boolean;
  rNum: real;
begin
  bFlag := False;
  try
    begin
      rNum := StrToFloat(sValue);
      bFlag := true;
    end;
  except
    ShowMessage('Invalid Number for ' + sFieldName + 'Error');
    bFlag := False;
  end;
  result := bFlag;
end;
```

## DATA TYPE CHECK FOR A STRING WITH LETTERS AND SPACE

```
function TValidation.isValidStringAZ(sValue: String): boolean;
Var
  bFlag: boolean;
  i, iLength: Integer;
begin
  if Length(sValue) <> 0 then
  begin
    i := 1;
    bFlag := true;
    while (i <= Length(sValue)) AND (bFlag = true) do
    begin
      if not(upcase(sValue[i]) in ['A' .. 'Z', ' ']) then
      begin
        bFlag := False;
      end;
      inc(i);
    end;
  end
  else
  begin
    bFlag := False;
  end;
  result := bFlag;
end;
```

## RANGE CHECK

Range validations are used to ensure that data input matches the expected range limitations imposed by the application. For example, in the case of validating a cell phone number, you need to check whether the number contains only digits and 10 characters in length.

### CHECKS WHETHER A CELLPHONE NUMBER IS VALID OR INVALID

```
function TValidation.IsValidCellNo(sCellNum: string): boolean;
var
  c: Integer;
  bFlag: boolean;
begin
  bFlag := true;
  c := 1;
  sCellNum := StringReplace(sCellNum, ' ', ' ', [rfReplaceAll]);
  if (Length(sCellNum) = 10) AND (sCellNum[1] = '0') then
  begin
    while (c <= 10) AND (bFlag = true) do
    begin
      if not(sCellNum[c] IN ['0' .. '9']) then
      begin
        bFlag := False;
        MessageDlg('Must contain digits only', mtError, [mbOK], 0);
      end;
      inc(c);
    end;
  end
  else
  begin // length not equal to 10
    bFlag := False;
    MessageDlg('Invalid Phone Number! Phone Number must be 10 digits long',
               mtError, [mbOK], 0);
  end;
  result := bFlag;
end;
```

↓  
Uses the *StringReplace* function to remove spaces from the cell phone number

### CHECKS WHETHER THE DATE ENTERED IS A VALID DATE

```
function TValidation.IsValidDate(sDay: String; iMonth: Integer;
    sYear: String): boolean;
var
    bFlag: Boolean;
    mm, yy: Integer;
begin
    mm := iMonth;
    bFlag := true;
    if (mm = 4) OR (mm = 6) OR (mm = 11) then
        if (StrToInt(sDay) > 30) then
            begin
                ShowMessage('Only 30 days in month ' + IntToStr(mm) + ' Invalid Date');
                bFlag := False;
            end;
    if (mm = 2) then
        begin
            yy := StrToInt(sYear);
            if (yy mod 4 = 0) then
                begin
                    if (StrToInt(sDay) > 29) then
                        ShowMessage('Only 30 days in Feb ' + IntToStr(mm) + ' Invalid Date');
                        bFlag := False;
                end
            else
                begin
                    if not(StrToInt(sDay) = 28) then
                        begin
                            ShowMessage('Only 28 days in Feb ' + IntToStr(mm));
                            bFlag := False;
                        end;
                end;
        end;
    result := bFlag;
end;
```

### PRESENCE CHECK

When a presence check is set up on a field then that field cannot be left blank, some data must be entered into it. The presence check does not check that the data is correct, only that some data is present. For example:

```
function TValidation.CheckEmpty(sValue: string): Boolean;
var
    bFlag: Boolean;
begin
    bFlag := true;
    if (sValue = '') then
        begin
            bflag := false;
            showmessage('Please enter a value');
        end;
end;
```



## Activity 6.6

**6.6.1** A South African ID number is a 13-digit number that is defined by the following format: YYMMDDSSSSCAZ

- The first 6 digits (YYMMDD) are based on your date of birth. For example, 13 March 1998 is displayed as 980313.
- The next 4 digits (SSSS) are used to define your gender. Females are assigned numbers in the range 0000–4999 and males from 5000–9999.
- The next digit (C) shows if you're a SA citizen status with 0 denoting that you were born a SA citizen and 1 denoting that you're a permanent resident.
- The last digit (Z) is a checksum digit – used to check that the number sequence is accurate using a set formula called the [Luhn algorithm](#). The checksum digit (Z) is calculated as follows:

A = The sum of the odd-positioned digits (positions 1, 3, 5, 7, 9, 11 and excluding the 13th digit)

B = The concatenation of the even-positioned digits (positions 2, 4, 6, 8, 10, 12)

C = The sum of the digits of the result B x 2

D = A + C

Z =  $10 - (D \bmod 10)$

Example: Calculate the checksum digit for the ID Number: 8801235111088

$$A: 8 + 0 + 2 + 5 + 1 + 0 = 16$$

$$B: 813118$$

$$C: 813118 \times 2 = 1626236$$

Sum of the digits in 1626236

$$= 1 + 6 + 2 + 6 + 2 + 3 + 6$$

$$= 26$$

$$D: 42$$

$$Z: 10 - (42 \bmod 10)$$

$$= 10 - 2$$

$$= 8$$

Open **ValidateInfo\_p** project in the 06 – Validate Information Folder and write code to do the following:

- Create a function **isNotEmpty** to receive an ID Number and return a value false if the ID number is empty; otherwise return true.
- Create a function **CalculateCheckSum** to receive an ID number and calculate and return the checksum digit as a string.
- Create a function **isValidId** to receive an ID number and return true if the ID number is valid; otherwise return false. An ID number is valid if:
  - It is 13 characters in length.
  - All the characters are digits.
  - The calculated checksum digit is equal to the last digit in the ID number. Call function **CalculateCheckSum** to return the calculation for the checksum digit.
- [Validate and Extract] button: Read an ID number from the EditBox *edtIdNo*.
  - Validate that data has been entered for the ID number in the *edtIdNo* EditBox by calling the **isNotEmpty** function.
  - Call function **IsValidId** to determine whether the ID number entered is valid.
  - Extract the date of birth from the ID number and display the date of birth in the format yyyy/mm/dd.
  - Display the gender of a person as either 'Male' or 'Female' using the gender digits in the ID number.
  - Display the citizenship status of a person as either 'South African' or 'Permanent Citizen' using the citizenship status digit in the ID number.

The screenshot shows a window titled "Validation Information". It contains four input fields: "Enter Id Number" with the value "8801235111088", "Date of Birth (yyyy/mm/dd)" with the value "1988/01/23", "Gender" with the value "Male", and "Citizenship" with the value "South African". At the bottom is a blue "Validate and Extract" button.

### New words

**Luhn algorithm** – is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers, IMEI numbers, and Canadian Social Insurance Numbers

**QUESTION 1**

Open the **LearnerYearMark\_p** project from the 06 – Learner Year Mark folder.

- 1.1** Create a function **CalculateYearMark** to calculate a learner's year mark. The function accepts the following parameters:

- Test 1
- Test 2
- Assignment
- Exam

The average of the two tests form a quarter of the learner's year mark, the assignment another quarter of the year mark and the exam forms half of the learner's year mark. The final year mark must be rounded up to the nearest integer.

Do the following:

- a. Write code for the function **CalculateYearMark**.
- b. Create an OnClick event for the [Calculate Learner Year Mark] button to read the marks for Test1, Test2, Assignment and Exam. Call the function **CalculateYearMark** to calculate the year mark.
- c. Display the year mark in the *edtYearMark* EditBox.

Open the **ConvertMeasure\_p** project from the 06 – Convert Measurement folder.

- 1.2** Create a function **Convert** that receives number of bytes and the unit of measurement (KB, MB, GB or TB) to convert the bytes to. The declaration of the function **Convert**:

Function **Convert**(NoBytes:integer;UnitOfMeasure:String):String;

Use the table below to convert to the required unit of measurement.

UNIT OF MEASUREMENT	NUMBER OF BYTES
Kilobyte (KB)	1 000 bytes
Megabyte (MB)	1 000 000 bytes
Gigabyte (GB)	1 000 000 000 bytes
Terabyte (TB)	1 000 000 000 000 bytes

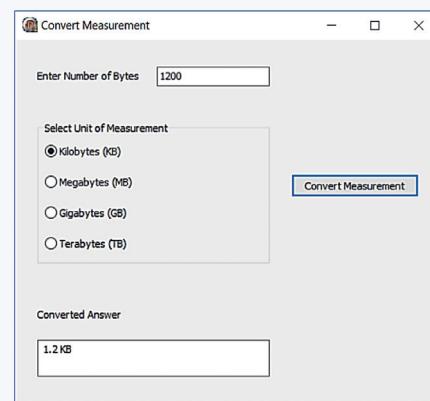
The function must return a string containing the answer of the conversion and the unit of measurement.

**Example:** if the function was called as follows:

```
sAnswer := Convert(1200, 'KB');
//then the function will return 1.200 KB.
```

Do the following:

- a. Write code for function **Convert**.
- b. Create an OnClick event for the [Convert Measurement] button to:
  - i. Read the number of bytes from the EditBox *edtBytes*.
  - ii. Select the unit of measurement to convert bytes to kilobytes.
  - iii. Call function **Convert** to do the conversion and display the conversion answer.



**QUESTION 2**

- 2.1 Replace delimiters of a string with new delimiters.

**Example:**

Replace the space delimiters in a string with the delimiter '\*'.

The string:

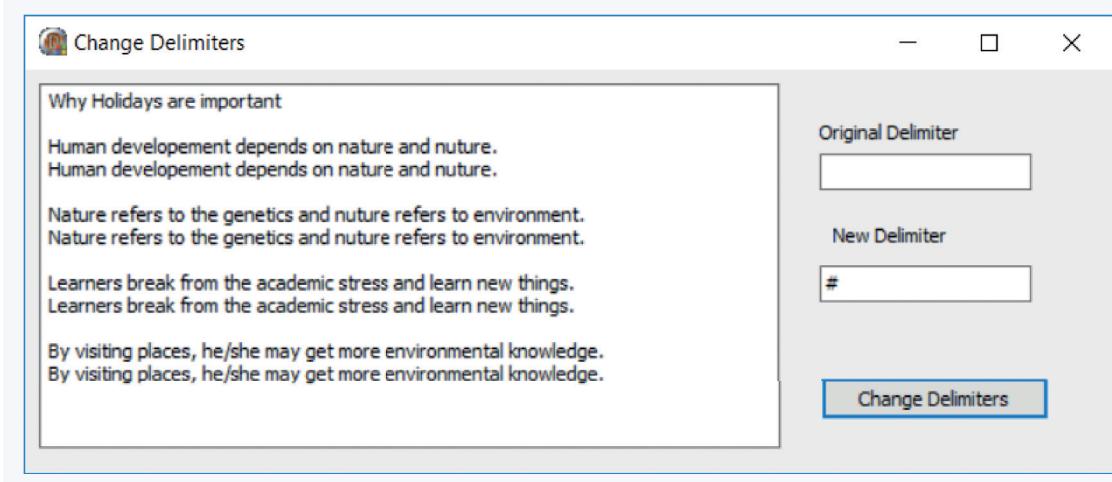
The fat cat jumps high.

**Becomes:**

The\*fat\*cat\*jumps\*high.

Open the **ChangeDelimiter\_p** project in the 06 – Change Delimiter Folders and do the following:

- a. Write a procedure **ChangeDelimiters** to accept a sentence, the current delimiter of the string and the new delimiter of the sentence. Replace the current delimiter with the new delimiter. Display the original sentence and the new sentence one below the other.
- b. Write a procedure **ReadFileData** to read the data stored in the text file **Sentences.txt** and store the values in an array **arrSentences**. You may assume that the text file may not contain more than 10 entries.
- c. Create an OnClick event for the [ChangeDelimiters] button to:
  - i. Read the original delimiter and the new delimiter from the EditBoxes.
  - ii. Call procedure **ReadFileData**
  - iii. Call procedure **ChangeDelimiters** to change the original delimiter of the strings stored in **arrSentences** with the new delimiter.



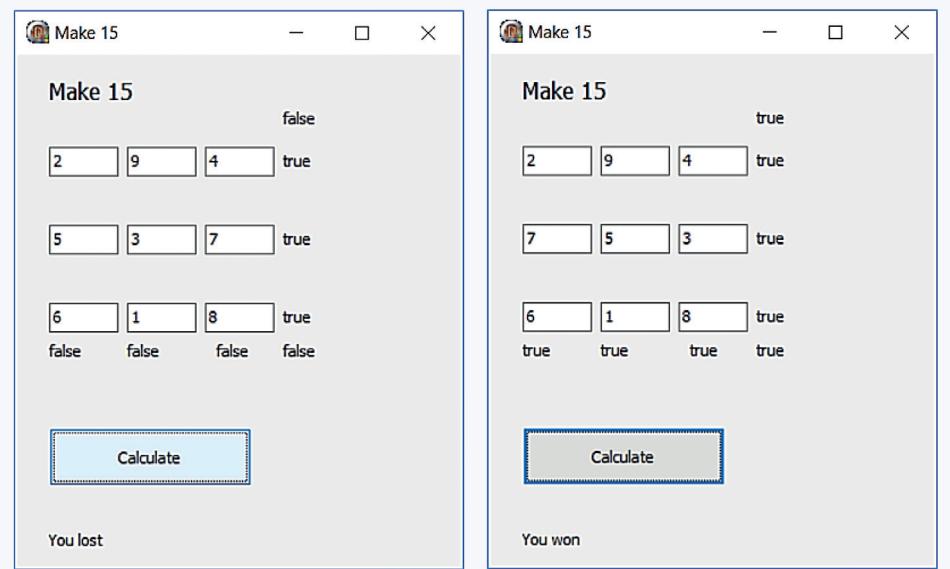
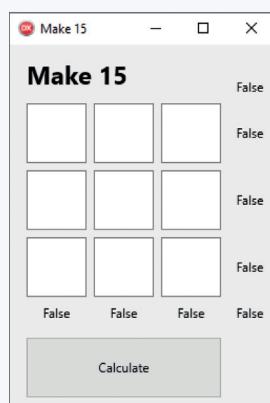
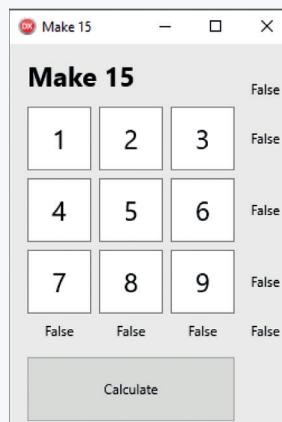
**2.2** The Make 15 Puzzle is a game. In this game, you have nine boxes, arranged in a three-by-three square. The goal of Make 15 is to enter values into these boxes so that all the rows, columns and diagonals add up to 15. The trick is that you are not allowed to use any number in the range 1 to 9, more than once.

Open the **Make15\_p** project in the 06 – Make15 Folder. You should see the following user interface for your puzzle.

**Note:** the EditBoxes are named as shown in the image alongside. EditBox *Edt1* is therefore the EditBox in the top-left corner, while EditBox *edt9* is the EditBox in the bottom-right corner.

- Write a function **IsUnique** that checks whether the 9 values entered by the user are unique.
- Write a function **IsFifteen** that accepts three integer parameters and returns a String ‘True’ if the total of the three numbers is 15; otherwise it returns the value ‘False’.
- Create an OnClick event for the [Calculate] button to:
  - Store the values from entered in the EditBoxes *edt1* to *edt9* in an array *arrValues*. The value of EditBox *edt1* is stored in *arrvalues[1]*, the value of EditBox *edt2* is stored in *arrValues[2]* and so on.
  - Call the **isUnique** function to check whether the values entered are unique values. If the values are not unique, display an appropriate message. If the values are unique, then call the **isFifteen** function and pass three values of the first row that is the first three elements of the array *arrValues*. Assign the return value of function **isFifteen** to the label next to the first row. Call the **isFifteen** function for the other rows, columns and diagonals. Determine **whether** a user won or not and display the message in the message label.

Save and test your application.



# USER INTERFACES

## CHAPTER UNITS



Unit 7.1 Multi-form user interface

Unit 7.2 Dynamic instantiation of objects



## Learning outcomes

At the end of this chapter you should be able to:

- describe the basics of human-computer interaction
- use the Hide and Show methods with components
- dynamically instantiate a passive component
- dynamically instantiate an active component
- add an event to a dynamically instantiated component
- use more than one form
- pass data between forms.

## INTRODUCTION

In Grade 10, you learned how to analyse user interfaces. A big part of this exercise was recording all the screens the application uses and understanding how to navigate between these screens.

Up to now all your applications have used a single screen. While this has worked thus far, there are serious drawbacks to creating single-screen applications:

- It limits the number of components you can use in your program.
- It limits the overall scope and size of your program.
- User interfaces can quickly become confusing.

In this chapter, you will learn how to add additional screens (or forms) to your application. This will allow you to create simpler, easier to use user interfaces that group similar functions on the same screen. You will also learn how to dynamically add elements to your screens. This means that components like labels, images and textboxes can be created using code. By the end of the chapter you should be comfortable using both techniques, as both these techniques are critical in creating professional applications.

## 7.1 Multi-form user interfaces

Multi-form user interfaces will only be tested in your practical assessment task (PAT) and not in your exams. To create a multi-form (or multi-screen) user interface, you need to follow three steps. These are:

- creating a second form
- moving between the forms
- passing data between the forms.

### ADDING AND ACCESSING A NEW FORM IN AN EXISTING PROJECT

You can add another form to your current project. Each form is an **independent** application that can run on its own.



#### Guided activity 7.1

Follow the steps below to add a form to an existing project.

**Step 1:** Open the existing project **FormSwapper\_p** project in the 07 – Form Swapper Folder

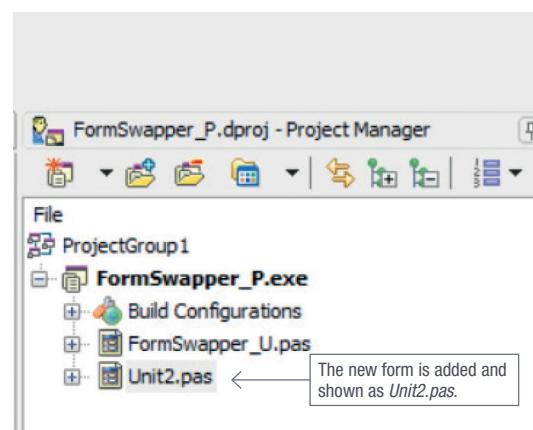
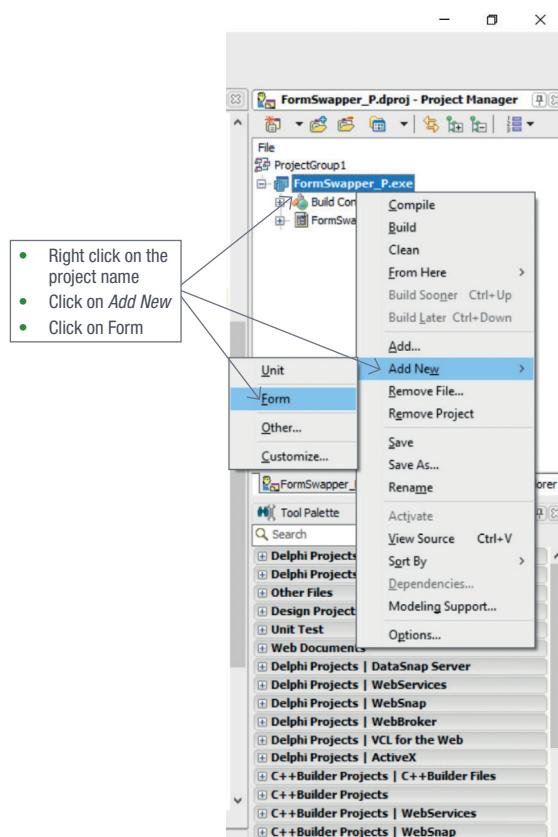
**Step 2:** To add an additional form to the current project, go to the *Project Manager* panel. Remember that the project manager is found above the *Tool Palette*.

**Step 3:** The new form will now be added to the project and will be shown in the project manager. The default name of the new form is *Unit2.pas*.



#### New words

**independent** – to run on its own

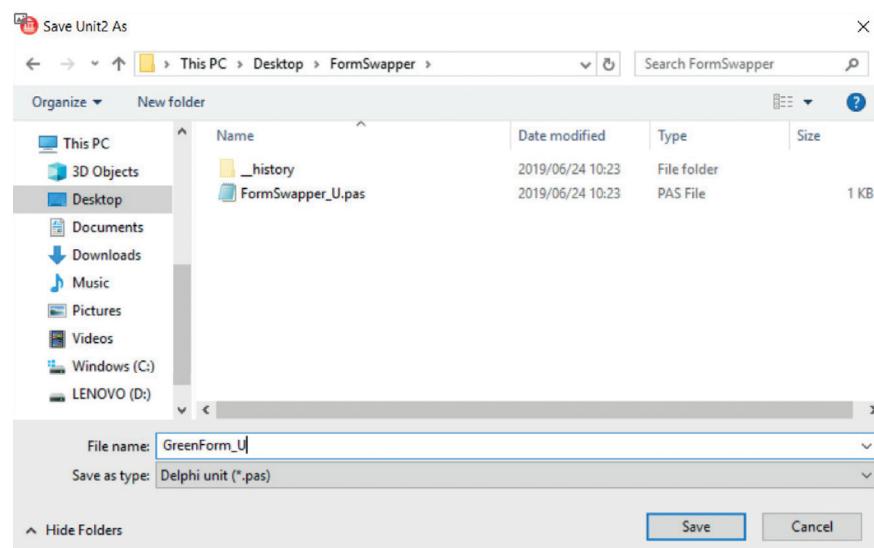




## Guided activity 7.1 *continued*

**Step 4:** The new form must now be saved. To save the new form:

- Click on *File* and select *Save*.
- The Save File DialogBox appears.
- Navigate to the project folder 07 – Form Swapper and open the project folder. Type the name of the new form **GreenForm\_u**.
- Click *Save*.
- The new form is now part of the project.



**Step 5:** You can now open any one of the forms using the *Project Manager* panel. Click on the form that you need to open. The opened form displays in its own tab in the form section of the Delphi interface.

- Each open form is shown in its own tab.
- Two forms are open, *FormSwapper\_U* and *GreenForm\_U*.
- Currently the *FormSwapper\_U* is selected and displayed.
- To switch between the forms, click on the form's tab.

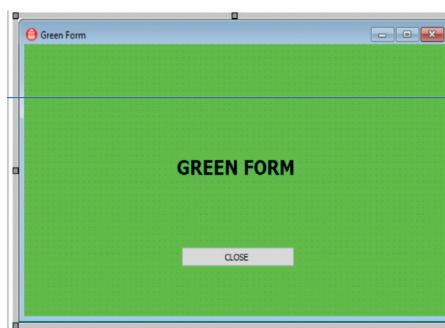


### Guided activity 7.1

*continued*

**Step 6:** Make the following changes to the *GreenForm\_u* form:

- Change the form caption to *Green Form*.
- Change the *Color* property to *cLime*.
- Add a label *lblMessage* with the caption *GREEN FORM*.
- Add a button *btnClose*.



**Step 7:** Moving between the forms:

- Select the *FormSwapper\_u* tab.
- In order to access the *GreenForm\_u* from the *FormSwapper\_u* form, you need to add the name of the green form in the uses section of the *FormSwapper\_u* form as follows:
  - Create an OnClick event for the [Move to Green Form] button and add the following code:

#### USES

```
Windows, Messages, SysUtils, Variants, Classes, Graphics,  
Controls, Forms, Dialogs, StdCtrls, GreenForm_U;  
procedure TfrmYellow.btnNewScreenClick(Sender: TObject);  
begin  
    frmGreen.Show;  
    frmYellow.Hide;  
end;
```

**Step 8:** Adding code to the *GreenForm\_u* form:

- Create an OnClick event for the [Close] button and add the code:

```
procedure TfrmGreen.btnCloseClick(Sender: TObject);  
begin  
    frmGreen.Close; // OR self.close;  
end;
```

#### Note:

The CLOSE method of a form closes any specified form. You can therefore close any form that is opened. The form closes not the application.

The statement:

```
Application.Terminate;
```

terminates the entire program.

The statement:

```
Self.close;
```

closes the current form.

**Step 9:** Save and run the project. The *FormSwapper\_u* form is set as the main form and will therefore display first when the program is executed. We will look at changing the main form to another form later on.

## SHARING DATA BETWEEN FORMS

Data can be shared between two forms in different ways:

- By including the name of the new unit in the *USES* section of the main form, you can access all variables added to the *PUBLIC* section of the new unit.
- By including the name of the unit in the *USES* section of the main form, you can access the properties of its components through the form object (Example : `frmSignUp.lblName.Text`).
- By sharing data between forms using text files.
- By sharing data between forms using a database.

In most situations, you will use the first or second option to share data between forms.



### Activity 7.1

A text file **Users.txt** stores the information of all users of FriendBook in the following format:

name , age , email , password

Here is an example of data in the text file:

Vijay,100,vakarianknight@outlook.com,Alien  
Arhaan,14,arhaan@gmail.com,Tookols  
Rayan,29,rayan@gmail.com,Secret  
Rachael,45,rachael@yahoo.com,45#Pc

**7.1.1** Open the **FriendBook\_p** project in the 07 – FriendBook Folder and do the following:

- Add a new form *FriendBookLogin\_u* to the **FriendBook\_p** project.
- Add components to the *FriendBookLogin\_u* form as shown below:



### Did you know

When using two forms, you **cannot** add both forms to each other's *USES* section. This creates a **circular dependency** that will cause the application to crash. Instead, you have to choose a parent form (typically your first form) and place the child form's name in its *USES* section.



### New words

**circular dependency** –  
to cause an application to crash



## Activity 7.1

*continued*

**7.1.2** The FriendBook logo is found in the project folder.

- Write an *OnClick* event for the [Login] button to read the email and password from the EditBoxes and look for a match of the email address and password in the **Users.txt** text file. If a match is found then display a message 'Match found – you are logged in'. If no match is found then display the message 'No match found'.
- Create an *OnClick* event for the label that contains the text 'sign up' to display the Sign in screen.

**7.1.3** The *FriendBookSignUp\_u* form is used to sign up friends for FriendBook.

Do the following:

The screenshot shows a Windows-style application window titled "SIGN UP". Inside the window, there is a large blue circular logo with the words "friend book" written in a white, lowercase, sans-serif font. Below the logo are five text input fields arranged vertically. Each field has a label to its left: "Name", "Age", "Email", "Password", and "Confirm Password". After the "Confirm Password" field is a "SIGN UP" button. At the bottom of the window, there is a link that says "Click here to return to the Login Form".

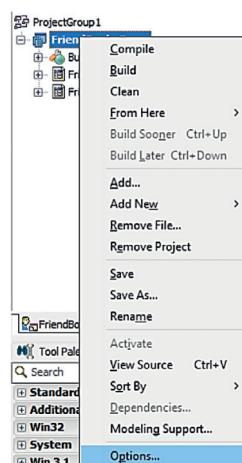
- Create an *OnClick* event for the [Sign Up] button to read the information for the name, age, email, password and confirm password from the EditBoxes. Check if the password and confirm password data match. If the password and confirm password matches, write the name, age, email and password to the **User.txt** text file in the required format. If the password and confirm password data does not match, then display an appropriate message.
- Create an *OnClick* event for the label that contains the text 'Click here to return to the Login Form' that will display the Login screen.

## SETTING THE MAIN FORM

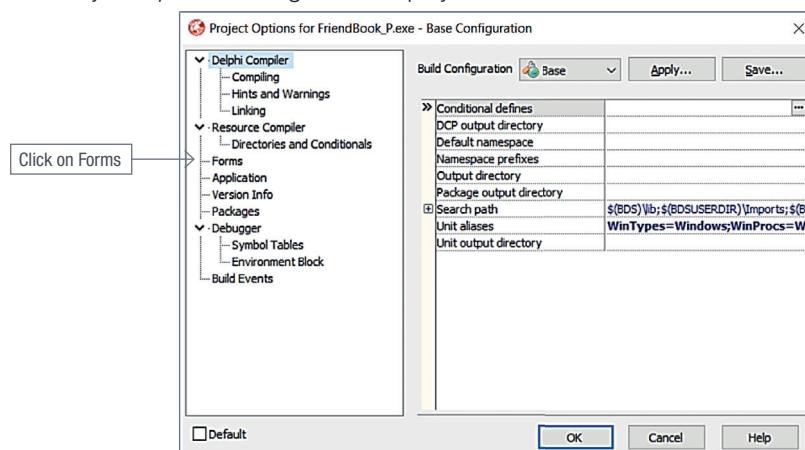
When you create a project, the initial form created is the Main form. This means that every time you run the program this form will execute first. When you are working with multiple forms, you may want to set the main form to another form.

To do this:

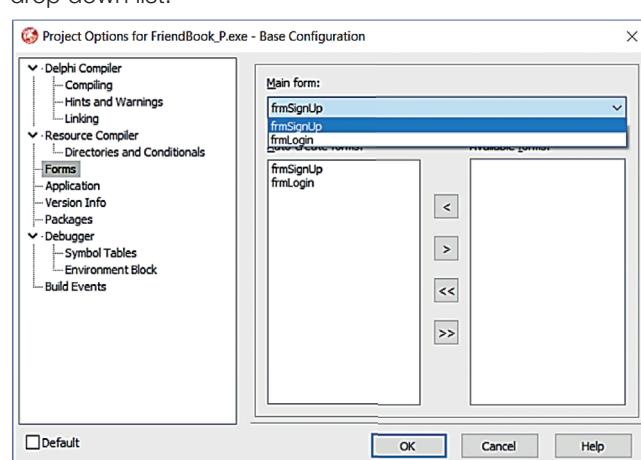
1. In the *Project Manager* panel, right-click on the name of the project and select *Options* from the drop down list.



2. The *Project Options DialogBox* is displayed.



3. Select the form that you want to set as the main form from the main form drop down list.



## Activity 7.2

- 7.2.1 Open the **Change MainForm\_p** project in the 07 – Change Main Form Folder.  
Change the main form to *frmLogin*.

- 7.2.2 Save and run the project.

## 7.2 Dynamic instantiation of objects

Up until now you have created a user interface and then coded using the user interface. In many situations the user interface will not always remain the same, such as in the case of a game. In a game the user interface changes as the game progresses. The programmer may create characters, obstacles and enemies. When a component or object is created during run-time we refer to this as **dynamic instantiation** of objects.



### New words

#### **dynamic instantiation** –

when a component or object is created during run-time

### VISUAL COMPONENTS

In Delphi, all components can be created using the *Create* method of their class. To dynamically create a label component, we need to declare the name of the label as a variable:

```
...
var
  frmMain: TfrmMain;
  lblMessage: TLabel; ← TLabel refers to the name of the class of a label
implementation
  ...
```

↑  
Name of the label variable

#### Instantiation of the label component object

The label is created by calling the *Create* method of the *TLabel* class.

```
lblMessage := TLabel.Create(Self); OR lblMessage := TLabel.Create(frmMain);
```

↑  
The label is a child of the form  
*frmMain*.

↑  
*Self* refers to the *frmMain* form.  
It is also parent.

↑  
The label belongs to the *frmMain* form. Form  
*frmMain* is referred to as a parent of the label.

#### Set the properties for the label

The TWO required properties are the *Parent* and *Caption* property.

```
lblMessage.Parent := Self; OR lblMessage.Parent := frmMain;
lblMessage.Caption := 'Hello, World!';
```

You can set other properties of label as is required using programming code such as:

```
lblMessage.Parent := Self;
lblMessage.Text := 'Hello, World!';
lblMessage.Left := 100;
lblMessage.Top := 100;
lblMessage.Width := 100;
lblMessage.Height := 20;
```

You can create any component in a similar manner as you did for the label component. In order to assign values to the component properties, you need to know the names of the most important properties.

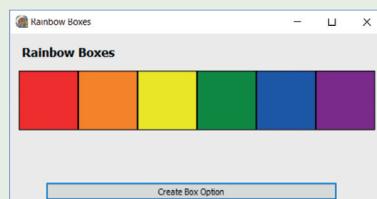
The table below lists these properties.

PROPERTY	COMPONENTS	PARAMETERS	DESCRIPTION
Parent	All	Form	All components must be assigned to a parent form. This is done by assigning the value 'Self' to the <i>Parent</i> property.
Left	All	Integer	The number of pixels the component is moved from the left of the form.
Top	All	Integer	The number of pixels the component is moved from the top of the form.
Width	All	Integer	The width of the component in pixels.
Height	All	Integer	The height of the component in pixels.
Text	Edit	String	The text shown by text boxes.
Caption	Button, Label	String	The text shown by buttons and labels.
Bitmap. CreateFromFile	Image	String	A string of the file path of the image that will be displayed.
Items.Add	ListBox, Memo	String	The string that will be shown on one row of a ListBox or memo.

To see how this is done, work through the following example.

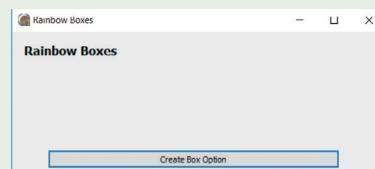
### Example 7.1 Rainbow boxes

For this project, you will create an application that creates rainbow coloured boxes every time you click on the [Create Box Option] button:



To do this:

1. Create a new project **RainbowBoxes\_p** project and save it in a folder named 07 – Rainbow Boxes.
2. Create the following user interface.



3. Add the unit *ExtCtrls* to the USES section of your program. This will allow you to dynamically create the *TShape* component.
4. Declare the following two global variables.

```
i : Integer = 0;  
shpDynamicBox: TShape;
```

The integer variable *i* is declared and initialised at the same time. This type of declaration can only be done with global variables. The variable *i* is assigned the value 0.

```
i: integer = 0;
```

The variable *shpDynamicBox*, is a *TShape* variable. The *TShape* variable was chosen since it has a *Brush.Color* property which accepts a colour variable. You will use this property to change the colour of the boxes.

### Example 7.1 Rainbow boxes *continued*

5. Create an *OnClick* event for the [Create a box] button and do the following:

Instantiate the shape object *shpDynamicBox*.

Set the properties of the newly created *shpDynamicBox* component as follows:

PROPERTY	VALUE
Parent	Self
Height & Width	80
Top	50
Left	$12 + i * 80$

While most of this code sets static properties for the rectangle, the *Left* property increases by 80 with each new box. The calculation for the *Left* property ensures that each time the value of variable *I* increases, the next box will be placed 80 pixels further to the right. The different boxes will be placed next to each other rather than on top of each other.

6. Create the following CASE statement to set the colour of the box based on the value of variable *I*.

#### Colour elements

```
Case i of
  0 : shpDynamicBox.Brush.Color := clRed;
  1 : shpDynamicBox.Brush.Color := $000080FF;
  2 : shpDynamicBox.Brush.Color := clYellow;
  3 : shpDynamicBox.Brush.Color := clGreen;
  4 : shpDynamicBox.Brush.Color := clBlue;
  5 : shpDynamicBox.Brush.Color := clPurple;
  6 : shpDynamicBox.Brush.Color := clFuchsia;
end;
```

7. Set the Name property of the box equal to the text *shpBox* followed by the value of variable *I*. This name could be used to change a box's properties at a later time.

Example:

```
shpDynamicBox.Name := 'shpBox' + IntToStr(I);
```

8. Finally increase the value of variable *I* by 1.

9. Save and run your project.

#### Code for the OnClick CreateABox procedure

```
procedure TfrmRainbowBoxes.btnAddClick(Sender: TObject);
begin
  shpDynamicBox := TShape.Create(Self);
  shpDynamicBox.Parent := Self;
  shpDynamicBox.Height := 80;
  shpDynamicBox.Width := 80;
  shpDynamicBox.Top := 50;
  shpDynamicBox.Left := 12 + I * 80;
  Case i of
    0 : shpDynamicBox.Brush.Color := clRed;
    1 : shpDynamicBox.Brush.Color := $000080FF;
    2 : shpDynamicBox.Brush.Color := clYellow;
    3 : shpDynamicBox.Brush.Color := clGreen;
    4 : shpDynamicBox.Brush.Color := clBlue;
    5 : shpDynamicBox.Brush.Color := clPurple;
    6 : shpDynamicBox.Brush.Color := clFuchsia;
  end;
  shpDynamicBox.Name := 'shpBox' + IntToStr(I);
  I := I + 1;
end;
```

## INTERACTIVE COMPONENTS

Creating an interactive component with an event (such as an *OnClick*, *OnChange* or *OnTimer* event), can be done in seven steps:

- **Step 1:** Declare the component variable.
- **Step 2:** Create the component and assign it to the variable.
- **Step 3:** Set the component properties.
- **Step 4:** Declare the name of a custom procedure in the Public section of your form.
- **Step 5:** Add the (*Sender: TObject*) parameter to the procedure's name.
- **Step 6:** Create the custom procedure for your event in the code.
- **Step 7:** Assign the name of the procedure to the appropriate event property.

### CREATING A BUTTON WITH AN ONCLICK EVENT

The first three steps of this procedure are identical to those of visual components:

- **Step 1:** Declare the component variable.
- **Step 2:** Create the component and assign it to the variable.
- **Step 3:** Set the component properties.

In the next four steps you will create a button interactive component with an *OnClick* event:

- **Step 4:** Declare the name of the custom procedure that you want to link your button or timer to in the *Public* section.

```
type
  TfrmMain = class (TForm)
  private
    { Private declarations }
  public
    procedure ButtonClick (Sender: TObject);
    { Public declarations }
  end;
```

Custom procedure *ButtonClick* has been declared.  
The procedure becomes a method of the form object.

*TObject* parameter called *Sender*. This parameter contains the object (button in this case) that called the procedure.

- **Step 5:** When you have multiple buttons activating the same procedure, the *Sender* parameter allows your form to understand which button was clicked to activate the procedure.
- **Step 6:** The second last step is to create your custom procedure (technically, your form method). This can be done in the same way as all the custom procedures you created in Chapter 6.

The code below shows an example of the **ButtonClick** form procedure.

```
Dynamic OnClick method
procedure TfrmMain.ButtonClick(Sender: TObject);
begin
  ShowMessage('Hello, Button!');
end;
```

This procedure should also include the *Sender* parameter.

- **Step 7:** Finally, the method name (excluding the *TfrmMain* part) can be assigned to the appropriate event property where the button is created. The method above can be assigned to the dynamic button as follows:

```
btnDynamic := TButton.Create(Self);
btnDynamic.Parent := Self;
btnDynamic.Text := 'Click me!';
btnDynamic.Left := 10;
btnDynamic.Top := 10;
btnDynamic.OnClick := ButtonClick;
```

Event name *OnClick*.

ButtonClick method assigned to button's *btnDynamic* *OnClick* property.

## CREATING TIMERS WITH AN ONTIMER EVENT

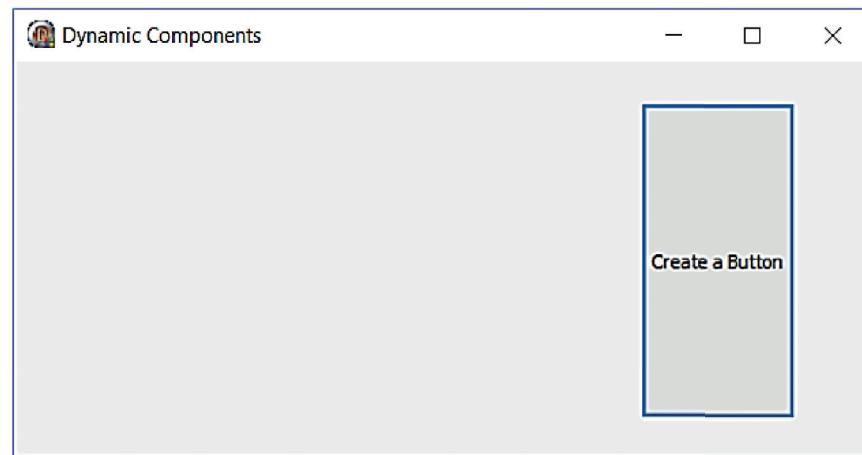
Timers are created in roughly the same way as a button, with only two differences:

- the timer must be enabled.
- an appropriate interval must be set.

The form method must be assigned to the OnTimer event.

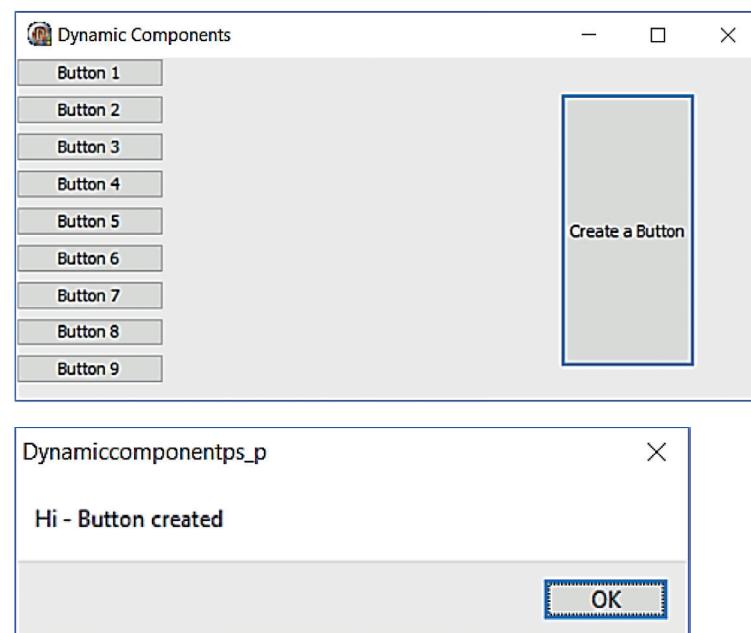
### Activity 7.3 Dynamic components

Open the **DynamicComponent\_p** project in the 07 – Dynamic Components Folder. You should see the following user interface.



**7.3.1** Create an OnClick event for the [Create a Button] button to do the following:

- Each time the [Create a Button] button is clicked, it creates a new button and displays a message as shown below:



- Save and run the project.

# DATABASES

## CHAPTER UNITS



- Unit 8.1 Creating a database
- Unit 8.2 Connecting to a database
- Unit 8.3 Reading data from a database
- Unit 8.4 Writing data to a database
- Unit 8.5 Manipulating data



## Learning outcomes

At the end of this chapter you should be able to

- create a simple database using Microsoft Access
- create a connection to a database using Delphi
- use a data module to connect to a database
- display the data from a database in Delphi
- access fields and records within a database
- select appropriate records using Delphi components or code
- modify values in a database using Delphi code
- manipulate database records using code
- use algorithms when working with databases
- filter a database using code with criteria.

## INTRODUCTION

Databases play a critical role in permanently storing data in applications. Whether it is a music library in a media player, the graphics and dialogue for a game, or a user's settings in an application, databases provide a structured, logical method to store data. In this chapter, you will learn how databases can be used in Delphi applications. The information stored in the database will be accessed and manipulated via programming code. The focus will be on connecting to the database, obtaining the data and using the data in your application.

## 8.1 Creating a database

In Chapter 7 of the Grade 11 IT Theory textbook, you learned how to create a database. The example below will very briefly take you through the process of creating the simple, single-table database that will be used in the **FriendBook** application you created in Activity 7.1 on page 159.

### FRIENDBOOK DATABASE STRUCTURE

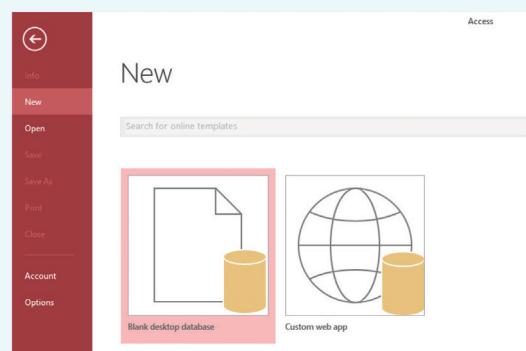
The structure below stores information about your friends in a single-table database with the following fields:

FIELD NAME	DATA TYPE	PURPOSE
Surname	ShortText	Stores user surname
FirstNames	ShortText	Stores user's first name(s)
DateOfBirth	Date/Time	Stores user's Date of Birth
Class	ShortText	Stores user's Grade/Division
ProfileViews	Integer	Stores number of times a user's profile has been viewed
Bio	ShortText	Stores a short description of the user

#### Example 8.1 Creating a database for FriendBook

To create an Access database:

1. Open the Start Menu and type 'Access'.
2. Click on the *Access* application to open it.



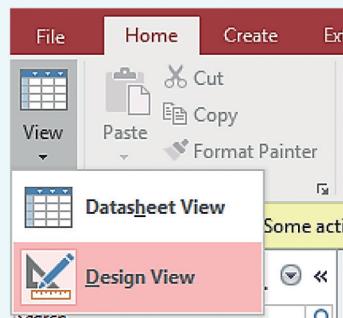
3. Double click on *Blank desktop database* option in the main window.
4. Click on the *Open* icon next to the *File Name* text box.
5. In the *New Database* window that opens, select an appropriate folder to save your database and click *OK*.
6. Select *Microsoft Access Database (2002-2003)* in the *Save as type* drop down list.
7. Enter the name *FriendbookDB.mdb* in the *File name* text box at the bottom of the *New Database* window.
8. Click *OK* to close the window, then click on the *[Create]* button to create the database.
9. Find the *Table1* table in the *All Access Objects* panel on the left side of Access.



### Example 8.1

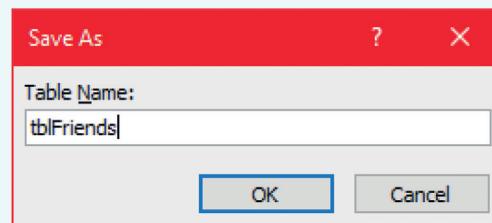
### Creating a database for FriendBook *continued*

10. Click on the Home tab in the ribbon. Click on View and choose *Design View*.



You will be prompted to name and save your table.

11. Enter the name *tblFriends* for the table and press Enter.



*Table1* will be renamed to *tblFriends*.

The table opens in *Design View*.

Indicates the Primary Key

For programming purposes: we will use naming conventions of variables, excluding the data type prefix.

Data Type of the field. To access other data types, click on the drop down list.

Description of the type of data the field holds. The description makes it possible for any programmer to make use of the database as it explains what the fields represent.

Field Properties

General | Lookup

Field Size	Long Integer
New Values	Increment
Format	
Caption	
Indexed	Yes (No Duplicates)
Text Align	General

The data type determines the kind of values that users can store in the field. Press F1 for help on data types.

Access will add a field named *ID* to your table automatically. *ID* is of type AutoNumber. This data type can be used to automatically generate a unique primary key for each record in numerical order starting from 1. Since this is our first database, we will use the AutoNumber as our primary key, however, in future databases, we will create a primary key manually.

#### Note:

- When generating Autonumbers, the computer always remembers the last Autonumber generated and adds 1 to it. For instance, if there are 15 records in a table and you delete the 15th record, when you add a new record, the Autonumber generated will be 16.



#### Did you know

Remember to save your work regularly. The easiest way to save a database in Access is to press the CTRL + S hotkey.

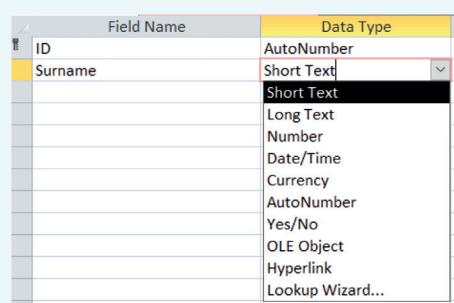
### Example 8.1 Creating a database for FriendBook *continued*

#### Adding Fields to a Table

Once you have created your table, you can create the fields that will be used in your application and later on, data will be added to these fields.

To do this:

1. Type the field name *Surname* as your first field.
2. Select the Short Text option as the data type.



3. In the Field Properties options, set a Field Size for Surname. It defaults to 255, but since surnames would not take up that many characters, it is in good design principles to set a smaller size. In the example below the size is set to 40.

General	
Field Size	40
Format	
Input Mask	
Caption	
Default Value	
Validation Rule	
Validation Text	
Required	No
Allow Zero Length	Yes
Indexed	No
Unicode Compression	Yes
IME Mode	No Control
IME Sentence Mode	None
Text Align	General

4. Add a second field of type Short Text called *FirstNames*. Set the field size, as necessary.
5. Add the remaining fields for *tblFriends*, choosing the appropriate data types and setting the Field Sizes, if necessary.
6. When all the fields have been added, Save changes to the database.

tblFriends	
Field Name	Data Type
ID	AutoNumber
Surname	Short Text
FirstNames	Short Text
DateOfBirth	Date/Time
Class	Short Text
ProfileViews	Number
Bio	Short Text

### Example 8.1 Creating a database for FriendBook *continued*

#### Adding Records to the table

1. Click on *Home, View, DataSheet View*. This will take you to DataSheet View.

2. In Datasheet View, enter the details for at least five friends:

ID	Surname	FirstNames	DataOfBirth	Class	ProfileView	Bio	ColourR	ColourG	ColourB
1	Mangenya	Vincent	2004/06/12	12A	5	I'm a popular friend	255	0	0
*(New)									

- a. The *ID* field will be automatically generated.
- b. Fill in the friend's surname and first names.
- c. Select the *DateOfBirth* using the *DateTimePicker* component.
- d. Type in the friend's Grade and Division in the field *Class*.
- e. For *ProfileViews*, initialise the field to 0.
- f. In the field *Bio*, type a short description about the friend.

3. Add records until you have at least five records in your table, then save the database.

4. With the database table and a few records created, you are ready to start using your database in an application. Save changes to your database.



#### Activity 8.1

Create Microsoft Access databases and add at least five records to the tables in the database.

Appropriate field names and data types should be used.

**8.1.1** A database called **ContactsDB** which stores information about your friends in *tblInfo*. Fields to store the following data should be created:

- a. Name
- b. Surname
- c. Age
- d. Phone number
- e. E-mail address
- f. Last date spoken

**8.1.2** A database called **ShopsDB** which stores information about a variety of different online or physical shops that you use in *tblShops*. Fields to store the following data should be created:

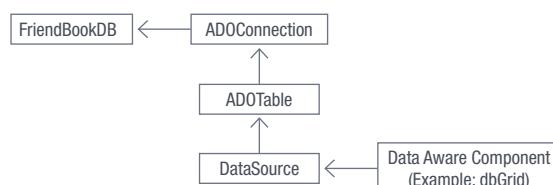
- a. Name
- b. Physical address
- c. Web address
- d. Type (e.g. electronics, clothes, fast food, groceries)
- e. Number of times you have used them
- f. Last time you used them

## 8.2 Connecting to a database

The first step to using a database in your application is to create a connection between your application and the database. To do this, you will use the following invisible components:

- **TADOConnection:** As the name suggests, the TADOConnection component is used to create a connection to an external database.
- **TADOTable:** The TADOTable component uses the database connection to connect to a specific table inside your database.
- **TDataSource:** The TDataSource component creates a connection between your database table(s) and the Delphi components. A DataSource can contain data from an entire table, a part of a table or data from several tables combined.

In our FriendBook example, we would diagrammatically represent the connection as shown below:



**Figure 8.1:** The connection between your application and the database

The first two components can be found from the *dbGo* list in RAD Studio's *Tool Palette*, while the *TDataSource* component can be found in the *Data Access* list of the *Tool Palette*.



**Figure 8.2:** The dbGo list from the Tool Palette

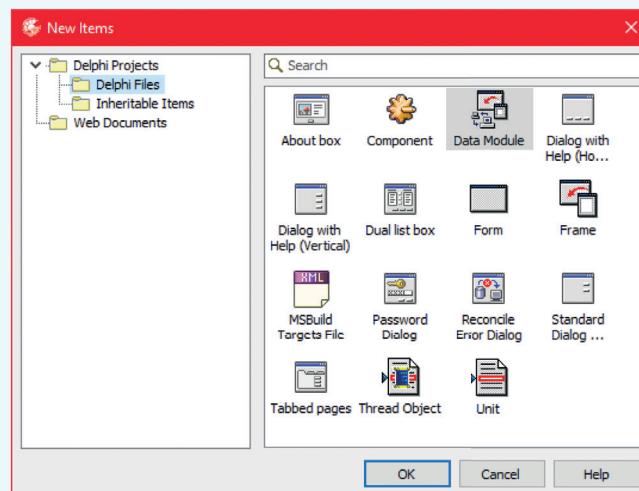
While these components can be added directly to your form, this links them to a specific form, making it harder to use the database in a multi-form application. Use of multiple forms is necessary for your PAT, so it is better to put our database connection components into a data module that we can share across forms. The database connection components can be added to a new type of unit called a **data module**. The example below shows how a data module can be created and used in a project.

## ADDING A DATA MODULE

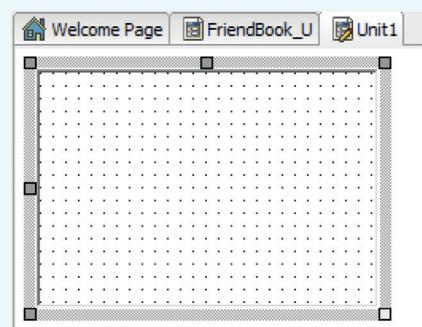
### Example 8.2 Adding a data module to your project

Open the **FriendBook\_p** project from the 08 – FriendBook\_1 Folder.

1. In the *Project Manager* panel in the top right corner of the screen, right click on the **FriendBook\_p.exe** project.
2. Select the *Add New* option and click on *Other*. A *New Items* window should now open.



3. Inside the *New Items* window, select the *Data Module* option and click *OK*. This will add a new data module to your project.



4. Save the data module in your project folder with the name **conFriendBook**.
5. Rename the data module form to **dbmFB**.
6. Select your main form and open the *Code* screen.
7. Add *conFriendBook* to the **Uses** section of your main form's code. Save and close your project.

```
unit FriendBook_u

interface

uses
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls,
  Forms, Dialogs, ComCtrls, StdCtrls, Keyboard, ExtCtrls, Grids,
  DBGrids, conFriendBook;
```

By adding the *conFriendBook* data module to a form, you gain access to any database connections created inside the data module. By separating the database connections from any form, you can import the database into each form through the **Uses** section.

## ADDING DATABASE CONNECTION COMPONENTS

Once you have created the data module, there are two methods of connecting to a database that we will discuss:

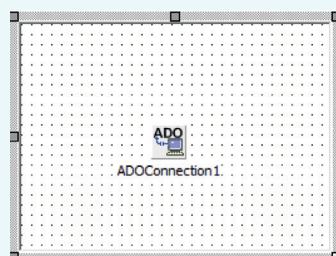
- Method 1: Connection via Connection Wizard (Object Inspector)
- Method 2: Connection via Dynamic Connection (Code)

### METHOD 1: CONNECTION USING THE OBJECT INSPECTOR

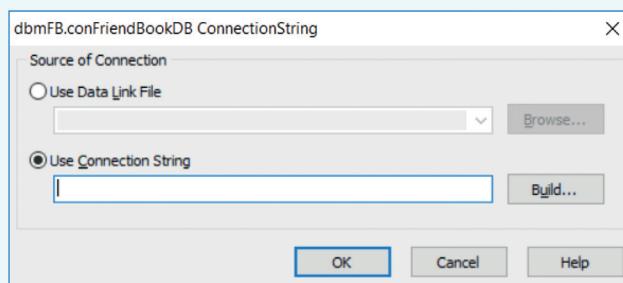
#### Example 8.3 Adding the database connection components

Open your **FriendBook\_p** project from the 08 – FriendBook\_1 Folder and open the **conFriendBook** data module.

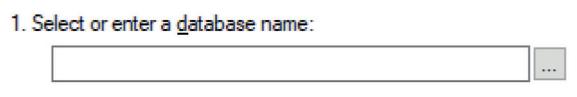
1. In the *Tools Palette*, open the *dbGo* list and drag a *TADOConnection* to the data module.



2. Rename the component to **conFriendBookDB**.
3. Change the connection's *LoginPrompt* property to False.
4. Double click on the **conFriendBookDB** component in your data module. This will open the *ConnectionString* window.



5. Click on the [Build] button.
6. In the *Data Link Properties* window, select the *Microsoft Jet 4.0 OLE DB Provider* option and click *Next*.
7. In the *Connection* tab, click on the three dots button next to the *Database name* textbox.

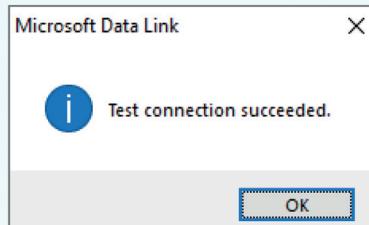


8. Using the *Select Access Database* screen, browse to your **FriendBookDB** database.
9. Select the **FriendBookDB.mdb** database and click *Open*.
10. In the *Connection* tab of the *Data Link Properties* window, click on the [Test Connection] button. You should receive a message stating that the test connection succeeded.



#### Did you know

If the database is in the same folder as your Delphi project, you can simply enter the database name and file extension in the text box. This will allow the application to work on any computer, even if the project is moved to a different folder.



### Example 8.3

### Adding the database connection components *continued*

11. Click *OK*, then click *OK* again in the *ConnectionString* window.

12. Change the *Connected* property of *conFriendBookDB* to *True*.

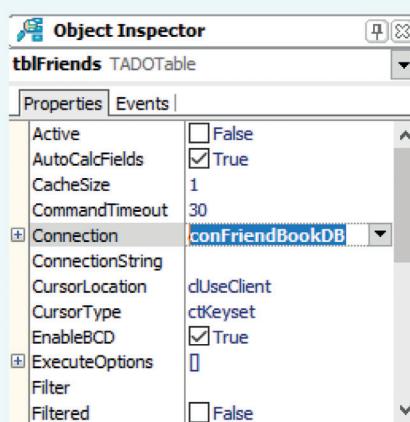
Congratulations! You have just connected your database to your Delphi project.

#### Connecting to a specific table

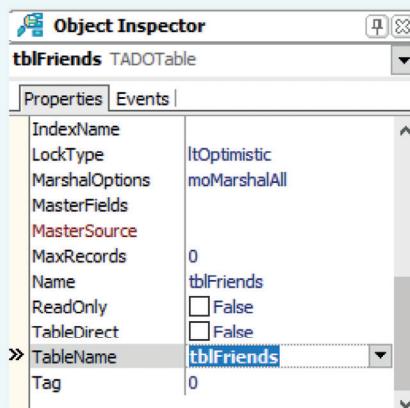
The next step is to use this connection to connect to a specific table and to make this table available in Delphi.

To do this:

1. From the *dbGo* list in the *Tool Palette*, drag a *TADOTable* component to your data module.
2. Rename the table component to *tblFriends*. Note that this is the same name we gave to our table when creating it in Microsoft Access.
3. Click on the dropdown list next to the *Connection* property of your *tblFriends* component and select *conFriendBookDB*.



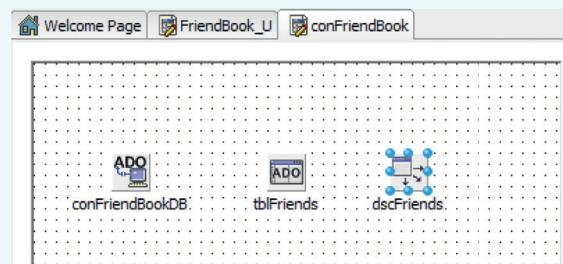
4. Change the value of the *TableName* property to the name of the table you are connecting to (*tblFriends*). If your connection was made correctly, you will be able to select the table from a dropdown list in the *Object Inspector*.



5. Change the table's *Active* property to *True*.
6. From the *Data Access* list in the *Tool Palette*, drag a *TDataSource* component to your data module.

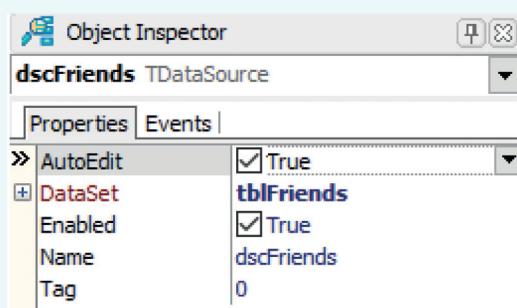
7. Rename the Data Source to *dscFriends*.

You should now have the following three components on your data module:



### Example 8.3 Adding the database connection components *continued*

8. Select your *dscFriends* component and set the value of its *DataSet* property to *tblFriends*.



CONNECTING TO AN ACCESS DATABASE



<https://www.youtube.com/watch?v=k8pGVnNHfE>

9. Save and close your application.

You now have a connection between your Delphi project and the *tblFriends* table in your Microsoft Access database.

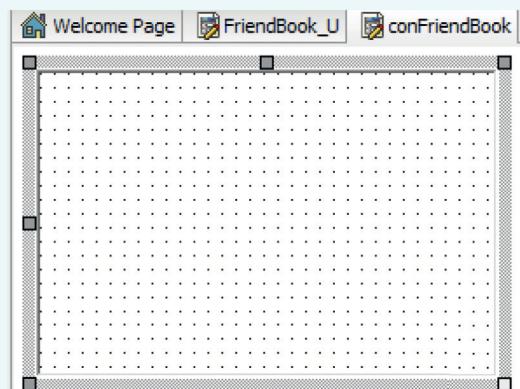
### METHOD 2: CONNECTION VIA CODE CONSTRUCT

#### Example 8.4 Connecting an Access database and your Delphi application using code construct

This method establishes a connection between the Access database and your Delphi application using code construct. Since we are writing code to point the *ADOConnection* to the external database, we must first ensure that our database resides in the same folder as our project.

Move your Access Database *FriendBookDB.mdb* to the 08 – FriendBook\_2 Folder.

1. Open the **FriendBook\_p** project from the 08 – FriendBook\_2 Folder. This project has an empty *DataModule* already created and saved.
2. Double click on **conFriendBook.pas** in the *Project Manager* to open the *DataModule*.



3. Since we are connecting the database using code, we do not need to add any components to the *DataModule* from the Palette.

#### Example 8.4

Connecting an Access database and your Delphi application using code construct *continued*

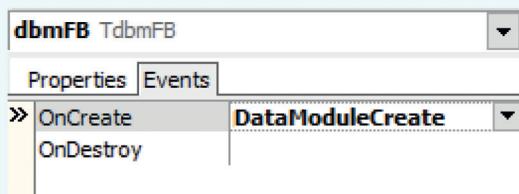
4. Switch to **Code** view.
5. Under **USES**, add: 'ADODB' and 'DB'.

```
uses
  SysUtils, Classes, ADODB, DB;
```

6. Declare the ADOConnection, ADOTable and DataSource under **Public**.

```
type
  TconFriendBookDB = class (TDataModule)
  private
    { Private declarations }
  public
    { Public declarations }
    conFriendBookDB: TADOConnection;
    dscFriends: TDataSource;
  end;
```

7. Switch to Design view and create an *OnCreate* event for the DataModule.



8. In the **OnCreate** method, write the following code:

```
conFriendBookDB := TADOConnection.Create(dbmFB);
tblFriends := TADOTable.Create(dbmFB);
dscFriends := TDataSource.Create(dbmFB);

conFriendBookDB.Close;
conFriendBookDB.ConnectionString := 'Provider=Microsoft.Jet.
OLEDB.4.0;Data Source =
+ ExtractFilePath(ParamStr(0)) + 'FriendBookDB.mdb' + ';
Persist Security Info = False';
conFriendBookDB.LoginPrompt := False;
conFriendBookDB.Open;

tblFriends.Connection := conFriendBookDB;
tblFriends.TableName := 'tblFriends';

dscFriends.DataSet := tblFriends;
tblFriends.Open;
```

These lines instantiate the Database component objects.

Responsible for setting up the ADOConnection and pointing it to the Access database. ExtractFilePath locates the database file within the project folder.

Sets up connection to specific table in the Access database via the ADOTable.

Connects the DataSource to the ADOTable to make it accessible to Data-aware components in the GUI.

Opens the table, making it accessible to the application's components

9. Save and close your application.

You now have a dynamic connection between your Delphi project and the Microsoft Access database.

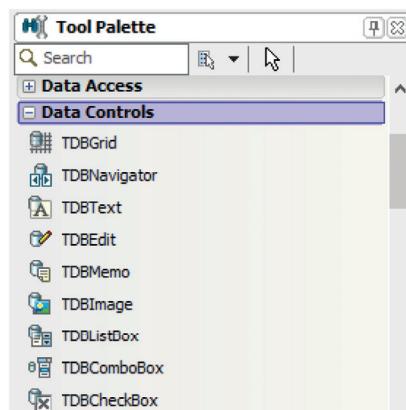
## 8.3 Reading data from a database

Once a database connection has been made, you can start using and displaying the data in your application.

### DISPLAYING DATA ON A DBGRID

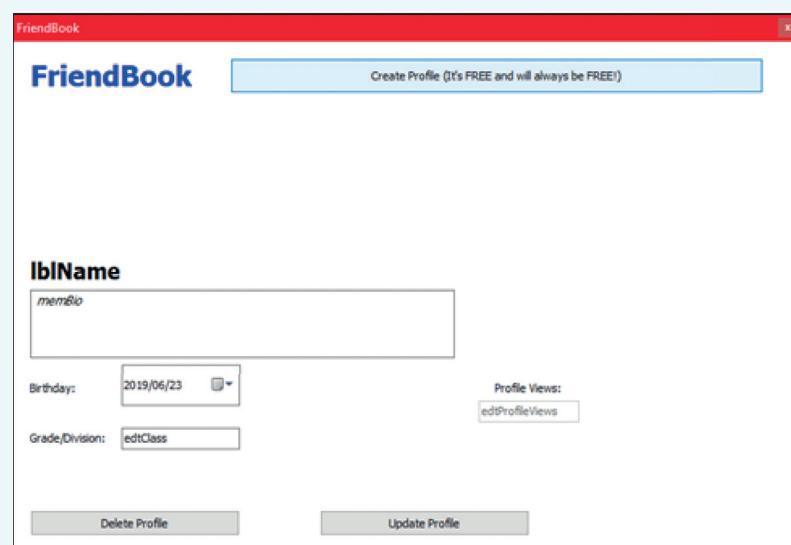
The *DBGrid* is a data-aware component because it is used to read data directly from a *DataSource*.

We will now use a *DBGrid* to display the contents of a database table in our Delphi application. The *TDBGrid* component can be added from the *Data Controls* list in the *Tool Palette*.



#### Example 8.5 Creating the user interface

Open the **FriendBook\_p** project in the 08 – FriendBook\_3 Folder. The basic user-interface has already been built. It is, however, missing an important component which you will have to add. To do this, the following basic user interface has been provided:



1. In the *Tool Palette*, find the *Data Controls* list and drag a *TDBGrid* to your form.

### Example 8.5 Creating the user interface *continued*

2. Change the name property to *dbgFriends* and resize it as shown in the image below.

The screenshot shows the FriendBook application window. At the top, there's a red header bar with the title 'FriendBook' and a button labeled 'Create Profile (It's FREE and will always be FREE!)'. Below the header is a large, empty DBGrid. Underneath the DBGrid, there's a section titled 'lblName' containing a memo field ('memBio'). To the right of the memo field are two buttons: 'Birthday:' with a date picker set to '2019/06/23' and 'Profile Views:' with a button labeled 'editProfileViews'. Below these are two more buttons: 'Grade/Division:' with a button labeled 'editClass' and 'Delete Profile' and 'Update Profile' at the bottom.

3. To display the data from your database, you need to connect *dbgFriends* to your datasource. To do this:
  - a. In the *Design* screen, create an *OnShow* event for the form.
  - b. In the *OnShow* event, write the following line to connect the *DBGrid* to *DataSource* in the *DataModule*.

```
dbgFriends.DataSource := dbmFB.dscFriends;
```

4. Save and test your application. You should now be able to see the records from your database on the *FriendBook* form.

The screenshot shows the FriendBook application window after saving and testing. The DBGrid now displays a list of records with columns for 'ID', 'Surname', 'FirstNames', and 'DataOfBirth'. The data in the grid is as follows:

ID	Surname	FirstNames	DataOfBirth
4	Raman	Kamalin	2003/03/01
5	Nyawo	Nothondo	2004/12/24
1	Mangenyela	Vincent	2004/06/12
2	Hooseen	Zubair	2003/02/10
3	Sithole	Penny	2004/05/01

The rest of the interface remains the same, with the 'lblName' section, profile edit fields, and buttons at the bottom.

The only problem remaining with the user interface of the *FriendBook* form is that a few of the columns are too wide. To solve this problem:

5. Navigate to the *OnShow* event for the form.

### Example 8.5 Creating the user interface *continued*

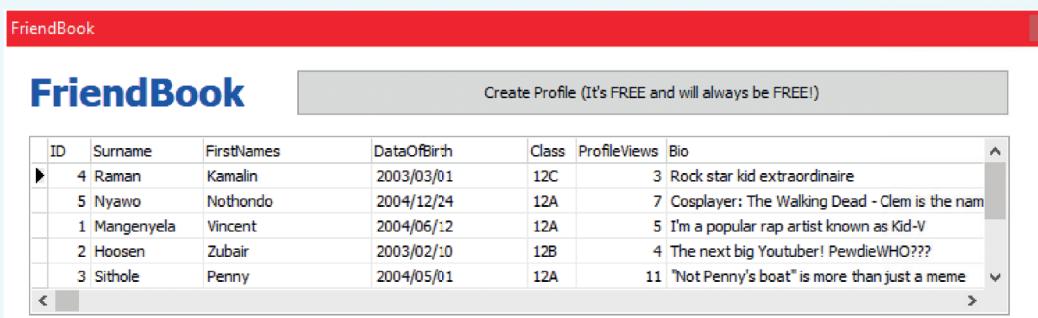
6. Inside the *OnShow* event, add the following lines of code.

#### Column width

```
dbgFriends.Columns[0].Width := 30;  
dbgFriends.Columns[1].Width := 80;  
dbgFriends.Columns[2].Width := 120;
```

This code changes the width of the first 3 columns of *dbgFriends*.

7. Using similar code, set the width of the remaining columns to appropriate values.
8. Once done, save and test your application.



Congratulations, you have just linked a database to a grid in your application! By using this grid, you can not only view the data from your database, but you can also change the data by clicking on any values and entering a new value.



### Activity 8.2

Both questions in this activity are based on the SoftDrink Application. This main form contains a DBGrid named *dbgSoftDrink*. An *OnShow* event is already created, but no code has been written for it. A DataModule named *conSoftDrink* has been created and saved to the project folder. The DataModule's object name is *dbmSoftDrink*. An *OnCreate* event is created for the data module.

The folder also contains a database named **SoftDrinkDB.mdb** that contains a single table named *tblSoftDrink* that has been populated with several records.

#### 8.1.1 Open project **SoftDrinkApp\_p** from the 08 – SoftDrink\_1 Folder.

Open the DataModule *conSoftDrink*. Your aim is to create a connection between **SoftDrinkDB.mdb** (The Access database) and the Delphi project using the Connection Wizard (method 1).

- Add the following components, renaming them as shown in the table:

COMPONENT	NAME
ADOConnection	conSoftDrinkDB
ADOTable	tblSoftDrink
DataSource	dscSoftDrink

- Complete the necessary steps in using the Object Inspector to connect the data components to the external database.
- Link the *DBGrid* in the GUI to the DataSource, to make the table visible to the user.
- Save and test your application.



## Activity 8.2 *continued*

### 8.1.2 Open project **SoftDrinkApp\_p** in the 08 – SoftDrink\_2 Folder.

- Open the DataModule *conSoftDrink*.

Your aim is to create a connection between **SoftDrinkDB.mdb** (The Access database) and the Delphi project using code construct (method 2).

- Add **ADODB** and **DB** to the USES section of the DataModule.

Declare the following components under Public in the DataModule:

```
public
  conSoftDrinkDB : TADOConnection;
  tblSoftDrink : TADOTable;
  dscSoftDrink : TDataSource;
```

- Write the necessary code in the *OnCreate* event to instantiate and connect the data components to the external database.
- Write the necessary code in the *OnShow* event in the main form to connect the *DBGrid* to the DataSource.
- Save and test your application.

## 8.4 Writing data to a database

In this unit, you will learn about two different ways to write data to a database.

Methods used to create a new record in a Database table:

METHOD	DESCRIPTION	EXAMPLE
Append	Creates an empty record at the end of a database table and selects it, allowing you to add values to the record.	<code>tblFriends.Append;</code>
Insert	Creates an empty record at the current position in a database table and selects it, allowing you to add values to the record.	<code>tblFriends.Insert;</code>
Post	Saves all the changes you have made to records. Without running this command, all changes will be discarded when the application is closed.	<code>tblFriends.Post;</code>

The **append** method adds an empty row to the end of your database table. The **insert** method adds an empty row at the current position of your database table. This means that if the pointer is at position 3, the new record will be inserted at position 3. The existing records from 3 onwards will be moved down accordingly: for example, record 3 will become record 4.

Once this new row has been created (using either append or insert), you can then set the values in this row, much like how you would change the values of a variable. For example, to set the values of the *Surname* and *FirstNames* fields in your *tblFriends* table, you can use the following syntax.

### Adding values to database fields

```
tblFriends.Append;
tblFriends['Surname'] := 'Goolam';
tblFriends['FirstNames'] := 'Noorjahan';
tblFriends.Post;
```

Looking at the syntax, after the append method, you use the table name followed by the name of the field (as a string in square brackets). You then use the assignment statement (:=) to set the value of the field. Once you have added values for all the fields, you use the **post command** to permanently save the values to the database table.

To see how this is used in practice, work through the following example.

#### Example 8.6 Adding records to a table

To add a record to a table:

1. Open the **FriendBook\_p** project in the 08 – FriendBook\_3 Folder
2. Create an *OnClick* event for the [Create Profile] button.



### New words

**insert** – to add an empty row at the current position of your database table

**append** – to add an empty row to the end of your database table

**post command** – to permanently save the values to the database table

### Example 8.6 Adding records to a table *continued*

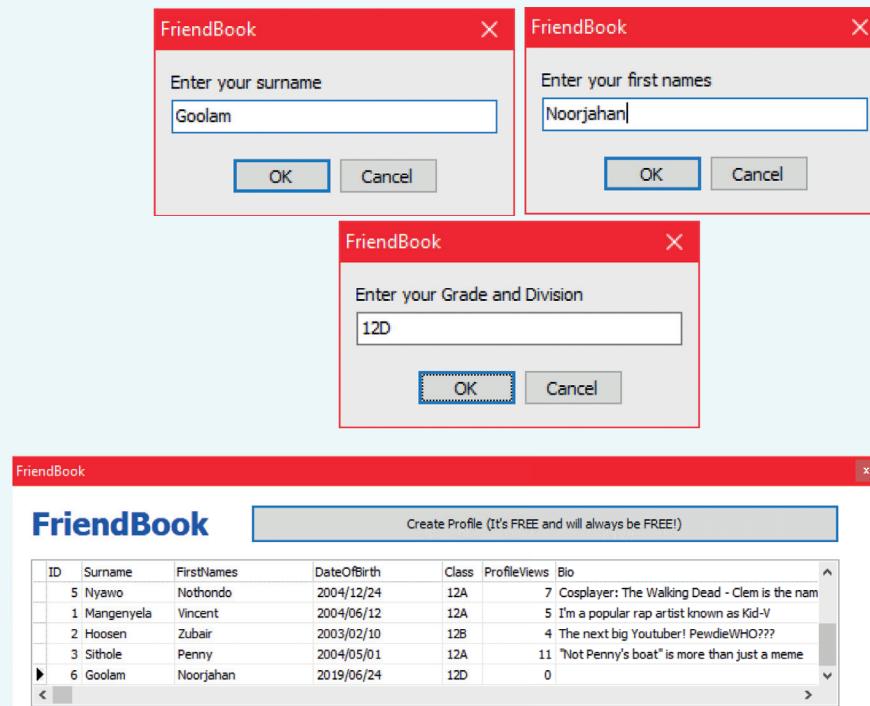
3. Use InputBox dialogs to get input to allow a new user to sign up for FriendBook.

```
procedure TfrmAddCards.btnAddCardClick(Sender: TObject);
begin
  dbmFB.tblFriends.Append;                                Adds a blank row at the end of the table

  dbmFB.tblFriends['Surname'] := InputBox('FriendBook',
    'Enter your surname', '');                            Values set from user input
  dbmFB.tblFriends['FirstNames'] := InputBox('FriendBook',
    'Enter your first names', '');                       Fields initialised with default values
  dbmFB.tblFriends['Class'] := InputBox('FriendBook',
    'Enter your Grade/Division', '');                    Fields initialised with default values
  dbmFB.tblFriends['ProfileViews'] := 0;
  dbmFB.tblFriends['DateOfBirth'] := Date;               Fields initialised with default values
  dbmFB.tblFriends['Bio'] := '';

  dbmFB.tblFriends.Post;                                 Writes new record to the table in the database on the storage medium

  showMessage('Congratulations! You''re registered.
  Login to complete your profile.');
end;
```



#### Note:

- Since *tblFriends* is an object on the data module, you need to first refer to *dbmFB* whenever you want to access the table.
- Some fields were set to Default values (such as *DateOfBirth*) and will be updated when we learn how to display and edit data.

In order to maintain Data Integrity, we normally validate input data before writing it to the database. This code does not include validation; however, it is important to note that professional applications protect data integrity by using validation techniques.

## THE WITH KEYWORD

Since our data components reside within the DataModule (*dbmFB*), we have to reference it whenever we interact with our database table. In larger applications, referencing the DataModule repetitively can become cumbersome.

Delphi includes a structure called **WITH** that enables us to reference a component or variable once and then avoid having to repeat this reference within the **WITH** block of code.

### Example 8.7 Display the user's Surname, First Names and Class using DialogBoxes

#### METHOD 1: Using the DataModule reference in every line:

```
showMessage(dbmFB.tblFriends['Surname']);  
showMessage(dbmFB.tblFriends['FirstNames']);  
showMessage(dbmFB.tblFriends['Class']);
```

#### METHOD 2: Using the WITH command

```
WITH dbmFB DO  
Begin  
    showMessage(tblFriends['Surname']);  
    showMessage(tblFriends['FirstNames']);  
    showMessage(tblFriends['Class']);  
End;
```

Using the **WITH** command is not compulsory but can make larger blocks of code more readable. Whether you use it or not depends on your programming style.

## DISPLAYING AND UPDATING DATA

We can extract data from a specific field of a selected record and display this data in a component that makes reading and editing easier for the user.



### Guided activity 8.1

- 8.1.1 Click on the DBGrid *dbgFriends*.
- 8.1.2 Create a *OnCellClick* event.
- 8.1.3 Write the following code in the event:

```
lblName.Caption := dbmFB.tblFriends['FirstNames'] + ' ' + dbmFB.  
tblFriends['Surname'];  
memBio.Text := dbmFB.tblFriends['Bio'];  
dtpBirthday.Date := dbmFB.tblFriends['DateOfBirth'];  
edtClass.Text := dbmFB.tblFriends['Class'];  
edtProfileViews.Text := dbmFB.tblFriends['ProfileViews'];
```

This code extracts data from the database table and displays it in standard Input/Output components such as EditBoxes.



## Guided activity 8.1 continued

- 8.1.4** When the user clicks on a profile in the *DBGrid*, the application will transfer the selected user's details to the individual components below.

The screenshot shows the FriendBook application interface. At the top, there is a DBGrid displaying a list of users with columns: ID, Surname, FirstNames, DateOfBirth, Class, ProfileViews, and Bio. The row for user ID 1, Vincent Mangenyela, is highlighted. Below the DBGrid, a detailed profile view for Vincent Mangenyela is shown. The profile includes his first name (Vincent), bio ('I'm a popular rap artist known as Kid-V'), birthday (2004/06/12), grade/division (12A), and profile views (5). At the bottom of the profile view, there are 'Delete Profile' and 'Update Profile' buttons.

### Editing an existing record

Now that we have the data available in standard input components such as *EditBoxes*, we can read changes the user may effect into these components and then update the captured data.

- 8.1.5** Create an *OnClick* event for the button [Update Profile].

- 8.1.6** First we must set our **ADOTable** into **Edit mode**.

```
dbmFB.tblFriends.Edit;
```

- 8.1.7** Next, we transfer data from our Input components into the table.

```
dbmFB.tblFriends['Bio'] := memBio.Text;
dbmFB.tblFriends['DateOfBirth'] := dtpBirthday.Date;
dbmFB.tblFriends['Class'] := edtClass.Text;
```

This code transfers data from the Input Components to the Database table

- 8.1.8** Finally, we write the changes to the database.

```
dbmFB.tblFriends.Post;
showMessage('Profile updated');
```

### Noorjahan Goolam

The screenshot shows the FriendBook application interface. A profile view for Noorjahan Goolam is displayed, showing her bio ('Future civil engineer. Building awesomeness!'), birthday (2004/04/29), grade/division (12D), and profile views (0). At the bottom, there are 'Delete Profile' and 'Update Profile' buttons.

The screenshot shows the Friendbook\_p application window with a single message: 'Profile updated'. There is an 'OK' button at the bottom.

The screenshot shows the FriendBook application interface with the DBGrid updated. The row for user ID 6, Noorjahan Goolam, now has a profile view count of 0, indicating the update was successful.

## MODIFYING FIELDS AUTOMATICALLY

It is sometimes necessary to update a field's value automatically. In our example, the field *ProfileViews* counts the number of times a user's profile has been viewed. Therefore it should automatically increase by 1 every time a user's profile is clicked on.

### Example 8.8 Modifying fields automatically

1. Navigate to the *dbgFriendsCellClick* event.
2. Each time a profile is clicked, we will increment the record's *ProfileViews* value.
3. Add the following code to achieve this:

```
dbmFB.tblFriends.Edit;  
dbmFB.tblFriends['ProfileViews'] := dbmFB.tblFriends['ProfileViews'] + 1;  
dbmFB.tblFriends.Post;  
edtProfileViews.Text := dbmFB.tblFriends['ProfileViews'];
```

4. Now, whenever the user clicks on a record in the *DBGrid*, the *ProfileViews* value will increment automatically.

### Deleting a current record from a table

An important feature in a Database application is the ability to remove a record from a table.

While it is not compulsory, it is considered good design to ask the user to confirm their action before deleting data from a database table.

### For confirmation, we usually use a MessageDlg DialogBox.

1. Create an *OnClick* event for the [Delete Profile] button.
2. Add the following code to the event:

MessageDlg is a customisable DialogBox that allows us to specify its type (mt), the available buttons (mb) and a customised message to the user

```
if MessageDlg('Are you sure?', mtConfirmation, mbYesNo, 0) = mrYes then  
begin  
    dbmFB.tblFriends.Delete; ← Permanently deletes the selected record  
    showMessage('Record deleted');  
end  
else ← If the user clicks No (mrNo)  
begin  
    showMessage('Delete cancelled');  
end;
```

The user's response is gauged based on a response constant which begins with mr (Message Response). In this case, if the user clicks Yes, we will proceed with deleting the record.

### Note:

- The **MessageDlg** is a function which has four parameters:
  - The message, which is of type String
  - The message type, which begins with **mt**
  - The message buttons, which begins with **mb**
  - The last parameter is the Help Context, to which we usually send **0**.
- The function returns an Integer value depending on which button the user clicks. You can use the Message Response constants (beginning with **mr**) to check which response was clicked.
- When coding the MessageDlg, you can type the first two letters (**mt**, **mb**, **mr**) and press *Ctrl + Space* to see a list of available options for each parameter.

### Example 8.8

Modifying fields automatically *continued*

The screenshot shows two windows of the FriendBook application. The top window displays a list of profiles and a detailed view for 'Noorjahan Goolam'. The bottom window shows the updated list after changes were made.

**FriendBook Application (Top Window):**

- Profile View:** Shows a table with columns: ID, Surname, FirstNames, DateOfBirth, Class, ProfileViews, Bio.
- Noorjahan Goolam Profile:** Shows the bio: "Future civil engineer. Building awesomeness!" and fields for Birthday (2004/04/29) and Grade/Division (12D). A confirmation dialog box is overlaid, asking "Are you sure?" with Yes and No buttons.
- Buttons:** Delete Profile and Update Profile.

**FriendBook Application (Bottom Window):**

- Profile View:** Shows the same table as the top window, but with updated data for the row corresponding to 'Noorjahan Goolam'.

ID	Surname	FirstNames	DateOfBirth	Class	ProfileViews	Bio
4	Raman	Kamalin	2003/03/01 12:47:59	12C	9	Rock star kid extraordinaire
5	Nyawo	Nothondo	2004/12/24	12A	17	Cosplayer: The Walking Dead - Clem is the nam
7	Goolam	Noorjahan	2004/04/29	12D	0	Future civil engineer. Building awesomeness!
1	Mangenya	Vincent	2004/06/12	12A	1	I'm a popular rap artist known as Kid-V
2	Hoosen	Zubair	2003/02/10	12B	9	The next big Youtuber! PewdieWHO???

## 8.5 Manipulating data

Once we have established a connection to a database table, we can write code to perform a variety of processing tasks to turn the extracted data into useful information.

When we process data from a database, we follow a similar pattern to processing data from a text file.

- You start by selecting the first record (or line) of the database table.
- Next, you create a WHILE-Do-loop that runs until you reach the end of file
- Inside the loop, you read the fields you are looking for and store the data in variables.
- Using these values, you complete any processing needed by your application.
- You then move to the next record and repeat the process.

### BASIC STRUCTURE WHEN MANIPULATING DATA

#### PSEUDOCODE

```
Move one pointer to the first Record
Loop While NOT at end of table
Begin Loop
    Read data from table
    Process data from table
    Move to next record
End Loop
```

#### DELPHI CODE

```
tblDams.First;
while NOT tblDams.EOF do
begin
    ...
    ...
    ...
    ...
    ...
tblDams.Next;
end;
```

The table below shows the table methods that allow you to step through your database.

METHOD	DESCRIPTION	EXAMPLE
First	Navigates to the first record in the table.	tblDams.First
Next	Moves to the next record in the table.	tblDams.Next
Prior	Moves to the previous record in the table	tblDams.Prior
Last	Moves to the last record in the table	tblDams.Last
Eof	Returns TRUE if the pointer (iterator) is at the end of a database table. Otherwise returns FALSE.	tblDams.EOF
RecordCount	Function that returns the number of records in a Database table.	tblFriends.Post;

## Example 8.9 Dams Application

For this project, you will use the functions listed above to perform calculations and apply filters to a database.  
To do this:

1. Open the project **DamsApp\_p** in the 08 – DamsApplication Folder. You should see the following user interface.

The screenshot shows a Microsoft Access application window titled "Dams DB". It displays two tables side-by-side:

tblDams						
DamID	DamName	River	YearCompleted	DamLevel	Capacity	HeightOfWall
1	Albasini Dam	Levubu River	1952	20053	28200	100
2	Albert Falls Dam	Umgene River	1976	142339	288100	100
3	Allemanskraal Dam	Sand River	1960	128666	174500	100
4	Alphen Dam	Bonte River	1990	237	700	100
5	Armenia Dam	Leeu River	1954	7056	13000	100
6	Beervlei Dam	Groot River	1957	50460	85800	100

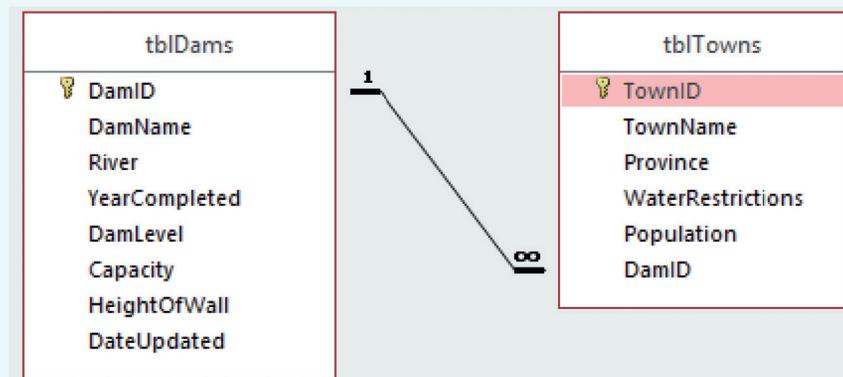
TownID	TownName	Province	WaterRestrictions	Population	DamID	HeightOfWall
1	Willowmore	Eastern Cape	False	7100	6	100
2	Queenstown	Eastern Cape	False	55800	14	100
3	East London	Eastern Cape	False	423500	18	100
4	Kareedouw	Eastern Cape	False	3400	23	100
5	Tsomo	Eastern Cape	False	10600	107	100

Below the tables are several buttons:

- redOut
- Navigate: First, Previous, Next, Last
- Average Capacity
- Search by Dam Name
- Calculate % Full
- List with Null Height
- List of Dams post 2000
- Count names by letter

In this application, the data module and all the database connections have already been made. Your only task will be to create the *OnClick* events for the different buttons and write code to make each button function.

The database **DamsDB** contains the following fields and relationship:



Take note of the primary keys and foreign key as these fields are necessary when accessing data across tables.

2. Create *OnClick* events for the [First], [Previous], [Next] and [Last] buttons.

### Navigating to the first record in the **tblDams**

In the *OnClick* event for **btnFirst**, add the following line of code:

```
dbmDamsDB.tblDams.First;
```

### Example 8.9 Dams Application *continued*

#### Navigating to the previous record in the *tblDams*

In the *OnClick* event for *btnPrevious*, add the following line of code:

```
dbmDamsDB.tblDams.Prior;
```

#### Navigating to the next record in the *tblDams*

In the *OnClick* event for *btnNext*, add the following line of code:

```
dbmDamsDB.tblDams.Next;
```

#### Navigating to the last record in the *tblDams*

In the *OnClick* event for *btnLast*, add the following line of code:

```
dbmDamsDB.tblDams.Last;
```

3. Save and test your application. The [First], [Previous], [Next] and [Last] buttons should now function, allowing you to navigate through *tblDams*.

## CALCULATING THE SUM AND AVERAGE WITH DATA FROM A DATABASE TABLE



### Guided activity 8.2

Calculating the Average Capacity of all dams (Project DamsApp\_p)

- 8.2.1 Create an *OnClick* event for the [Average Capacity] button.
- 8.2.2 The field *Capacity* in *tblDams* contains the maximum quantity of water that each dam can store. We will now write code to calculate the average capacity based on the data in the table. In order to calculate the average, we first need to determine the total capacity (sum).

We begin by declaring two variables: *rSum* and *rAve*. Since *rSum* will be used to determine the sum by adding each *Capacity* value to it, we need to initialise it to 0.

```
rSum := 0;
```

- 8.2.3 Our next step is to move the iterator (pointer) to the first record in the table. This is necessary because (by user action or processing), the iterator could be at any point in the table. We need to make sure that processing will start at the first record.

```
dbmDamsDB.tblDams.First;
```

- 8.2.4 Next, we need to use a WHILE-Do-loop to iterate through every record. This is necessary as we have to process every record, extracting its capacity value and adding it to *rSum*.

```
while NOT dbmDamsDB.tblDams.EOF do ←  
begin  
...  
end;
```

The loop repeats as long as the iterator (pointer)  
is NOT pointing to the End of File (table)

To add each dam's capacity value to *rSum*, we include the following line inside the WHILE-Do-loop:

```
rSum := rSum + dbmDamsDB.tblDams['Capacity'];
```



### Guided activity 8.2

#### Calculating the Average Capacity of all dams *continued*

- 8.2.5 Next, within the WHILE-Do-loop, we instruct the iterator to move to the next record in order to continue the process of calculating the total capacity.

```
dbmDamsDB.tblDams.Next;
```

- 8.2.6 After the WHILE-Do-loop terminates, we will now have the total capacity stored in rSum. We can now use this total to calculate the Average.

```
rAve := rSum / dbmDamsDB.tblDams.RecordCount;
```

- 8.2.7 Finally, we will output the average to the Output Area (*redOut*).

```
redOut.Lines.Add('Average Capacity: ' + FloatToStr(rAve));
```

- 8.2.8 Save and run your application. Clicking the [Average Capacity] button should now calculate and display the average capacity of all dams.

## CALCULATIONS BASED ON A SPECIFIC RECORD

In the previous button, we wrote code that processed the entire table, by reading and processing each value consecutively.

Depending on the scenario, it may be necessary to process a single record only.



### Guided activity 8.3

#### Determining % of water currently in a specific dam (Project DamsApp\_p)

- 8.3.1 The field *Capacity* indicates the maximum amount of water that a dam can store.

- 8.3.2 The field *DamLevel* indicates the current amount of water in the dam.

- 8.3.3 To express how full the dam is as a percentage, we can use the following formula:

```
(DamLevel / Capacity) * 100
```

- 8.3.4 When the user clicks on a dam in the **tblDams DBGrid** and then clicks on the [Calculate % Full] button, we will extract the relevant values from the table, calculate the percentage and display the percentage to the Display Area (*redOut*).

#### Writing the code:

1. Create an OnClick event for the [Calculate % Full] button.
2. Declare a variable of type String for the selected dam's name and three variables of type Real to store the *DamLevel*, *Capacity* and calculated percentage.

```
var  
  sName : String;  
  rLevel, rCapacity, rPerc : Real;
```



### Guided activity 8.3

Determining % of water currently in a specific dam *continued*

3. Next, write the code below, which will extract, process and output the necessary data.

```
sName := dbmDamsDB.tblDams['DamName'];
rLevel := dbmDamsDB.tblDams['DamLevel'];
rCapacity := dbmDamsDB.tblDams['Capacity']; } ← Calculates the % using the provided formula.

rPerc := (rLevel / rCapacity) * 100; } ← Outputs calculated data

redOut.Clear;
redOut.Lines.Add('Dam Name: ' + sName);
redOut.Lines.Add('Level: ' + FloatToStr(rLevel));
redOut.Lines.Add('Capacity: ' + FloatToStr(rCapacity));
redOut.Lines.Add('Percentage Full: ' +
FloatToStrF(rPerc, ffFixed, 3, 2)); } ← Outputs calculated data
```

4. Save and run your application. Clicking on a Dam in *tblDams* **dbGrid** and then clicking the [Calculate % Full] button should now calculate and display the percentage level of the selected dam.

The screenshot shows a Windows application interface. At the top is a title bar with the text 'DamsApp\_p'. Below it is a toolbar with icons for New, Open, Save, Print, and Exit. The main area contains a **dbGrid** control named 'tblDams' displaying a list of dams. The grid has columns: DamID, DamName, River, YearCompleted, DamLevel, Capacity, and a small icon column. The fourth row, representing 'Alphen Dam', is selected and highlighted with a blue border. Below the grid is a **Details** window titled 'Alphen Dam'. It displays the following information:  
Dam Name: Alphen Dam  
Level: 237  
Capacity: 700  
Percentage Full: 33.90

## DISPLAYING DATA BASED ON A SPECIFIC CRITERION

A very common process in programming is searching through a dataset for records that meet certain criteria and then using these values in further processing; or simply displaying them.



### Guided activity 8.4

Display all Dams completed after the year 2000 (Project DamsApp\_p)

The field **YearCompleted** indicates the year when each dam's construction was concluded.

We will now loop through *tblDams* and test each dam's **YearCompleted** value based on our criteria. If the value meets our criteria, we will display the dam's name.

- 8.4.1 Create an OnClick event for the [List of Dams post 2000] button.



### Guided activity 8.4

Display all Dams completed after the year 2000 *continued*

- 8.4.2 The code and explanations are provided below:

```
redOut.Clear;
redOut.Lines.Add('List of Dams completed after 2000'); } ← Clears Display Area and displays a Heading

dbmDamsDB.tblDams.First; ← Sets pointer to first record

while not dbmDamsDB.tblDams.EOF do ← Loops through entire Dams table
begin

    if dbmDamsDB.tblDams['YearCompleted'] > 2000 then ← Tests for criterion
    begin
        redOut.Lines.Add(dbmDamsDB.tblDams[ 'DamName' ]); ← If condition is met, Dam's name is displayed
    end;

    dbmDamsDB.tblDams.Next; ← Moves pointer to next record for testing

end;
```

- 8.4.3 Save and run your application. Clicking on the [List of Dams post 2000] button should now display a list of dam names for dams completed after the year 2000.



## SEARCHING FOR RECORDS BASED ON USER INPUT

Searching based on user input uses an algorithm very similar to the one used in the previous Guided activity. The only difference is instead of using a constant condition, the condition will compare the field being tested against the user's input.



### Guided activity 8.5

Search for a Dam Name specified by the user (Project DamsApp\_p)

- 8.5.1 Create an *OnClick* event for the [Search by Dam Name] button.
- 8.5.2 Our aim is to prompt the user to input the name of a Dam. We then loop through *tb/Dams* and identify all dams that match the input name. The dam's name, level and capacity will be displayed if it is found. If no results are found, an appropriate message is displayed instead.



### Guided activity 8.5

#### Search for a Dam Name specified by the user *continued*

The code and explanations are provided below:

```
sTarget := InputBox('Dams DB', 'Enter Dam Name', ''); ← Gets user's input – value being searched for.  
  
bFound := FALSE; ← Flag variable to determine if search was successful or not.  
  
dbmDamsDB.tblDams.First; ← Moves pointer to first record; the search will begin at the first record.  
  
while not dbmDamsDB.tblDams.EOF do ← Loops through entire table. NOTE: if searching for a single value, the flag can be included in the While condition to terminate the loop as soon as the target is found  
begin  
  
    if dbmDamsDB.tblDams['DamName'] = sTarget then ← Tests each Dam Name against target  
    begin  
        redout.Lines.Add('Dam Name: ' + dbmDamsDB.tblDams['DamName']);  
        redOut.Lines.Add('Capacity: ' +  
                        FloatToStr(dbmDamsDB.tblDams['Capacity']));  
        redOut.Lines.Add('Dam Level: ' +  
                        FloatToStr(dbmDamsDB.tblDams['DamLevel']));  
  
        bFound := TRUE; ← If the Dam Name matches the target, the relevant fields are output and the Flag is changed to TRUE indicating that the search item was found  
    end;  
  
    dbmDamsDB.tblDams.Next; ← Moves the pointer to the next record for testing  
  
end;  
  
if NOT bFound then ← When the While loop terminates and the flag is still FALSE, it means that the search target was not found.  
begin  
    showMessage('Dam not found'); ← Displays an appropriate message  
end;
```

- 8.5.3** Save and run your application. Clicking on the [Search by Dam Name] button should now prompt the user to input the name of a dam. The application should then locate and display the target dam's details or display the 'Dam not found' DialogBox if the target was not located.

The screenshot shows the application's user interface. At the top is a dialog box titled "Dams DB" with a red header bar. It contains a label "Enter Dam Name" and a text input field containing "Woodstock Dam". Below the input field are two buttons: "OK" and "Cancel". In the main window, there is a left panel displaying dam details: "Dam Name: Woodstock Dam", "Capacity: 373300", and "Dam Level: 322366". To the right of this panel is a "Navigate" section with buttons for "First", "Previous", "Next", and "Last". Below the "Navigate" section are several other buttons: "Average Capacity", "Calculate % Full", "List of Dams post 2000", "Search by Dam Name" (which is highlighted in blue), "List with Null Height", and "Count names by letter".

## IDENTIFY FIELDS WITH NULL VALUES

When processing values from a database table, it is sometimes necessary to identify fields which contain a null (empty) value. Note that **null** should not be confused with **0** or an empty string (''). If a numeric field is assigned to 0 or a String field is assigned to an empty string, both are considered to hold values. Having a null value means that the field is empty and has not been initialised. Fields with null values can cause run-time errors when looping through a table or cause logical errors when processing calculations such as averages.



### New words

**null** – to represent an empty value



#### Guided activity 8.6

Identify Dams with NULL values in the HeightOfWall field (Project DamsApp\_p)

- 8.6.1** Create an OnClick event for the [List with Null Height] button.

Our aim to display the dam names for all dams which have a null value for the *HeightOfWall* field.

The code and explanations are provided below:

```
redOut.Clear;  
redOut.Lines.Add('List of Dams with empty HeightOfWall values');  
  
dbmDamsDB.tb1Dams.First; ← Moves pointer to first record  
  
while not dbmDamsDB.tb1Dams.EOF do ← Loops through entire table  
begin  
  
    if dbmDamsDB.tb1Dams['HeightOfWall'] = null then ← Tests each HeightOfWall  
        begin                                         against the constant null  
            dbmDamsDB.tb1Dams['DamName']; ← which is used to test for  
        end;                                            empty variables / fields.  
  
    redOut.Lines.Add(dbmDamsDB.tb1Dams['DamName']); ← Displays Dam Name if  
end;                                                 condition is met  
  
dbmDamsDB.tb1Dams.Next; ← Moves pointer to next record  
  
end;
```

- 8.6.2** Save and run your application. Clicking on the [List with Null Height] button should now loop through *tb1Dams* and identify the dams with null *HeightOfWall* values and display the corresponding dam names into the Output Area.

**List of Dams with empty HeightOfWall values**  
Beervlei Dam  
Bronkhorstspruit Dam  
Dap Naude Dam  
Driekloof Dam  
Driekoppies Dam  
Elandskloof Dam  
Floriskraal Dam  
Vondo Dam

## COUNTING FIELDS THAT MATCH SPECIFIC CRITERIA

Statistical calculations often require us to count the number of records which meet certain criteria. The process is very similar to the methods we've already explored.



### Guided activity 8.7

Count the number of dams with names beginning with a specified letter  
(Project DamsApp\_p)

In this activity, the user will input a letter of the alphabet. We will then loop through the *tblDams* table and count the number of dams that have a name beginning with that letter.

- 8.7.1** Create an *OnClick* event for the [Count names by Letter] button.

The code and explanations are provided below:

```
sTarget := InputBox('Dams DB', 'Enter Search Letter', '')[1]; ← Letter being searched for is input by the user

iCount := 0; ← Whenever we count, it is compulsory to initialise our Counter to 0

dbmDamsDB.tblDams.First; ← Moves pointer to first record

while not dbmDamsDB.tblDams.EOF do ← Loops through entire table
begin
    sDamName := dbmDamsDB.tblDams['DamName']; ← Extracts dam name from tblDams for processing

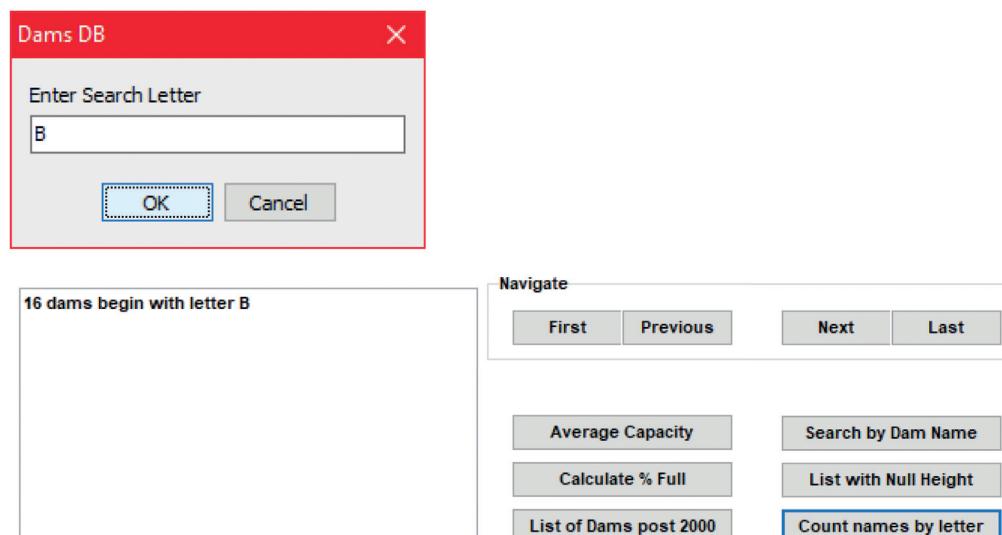
    if sDamName[1] = sTarget then ← Tests 1st character in Dam Name to letter provided by user. If they match, the Counter variable (iCount) is increased by 1.
    begin
        inc(iCount);
    end;

    dbmDamsDB.tblDams.Next; ← Moves pointer to next record

end; ← Output to user

redOut.Text := IntToStr(iCount) + ' dams begin with letter ' + sTarget;
```

- 8.7.2** Save and run your application. Clicking on the [Count names by Letter] button should now prompt the user to input a letter of the alphabet and then loop through *tblDams*. It will then identify dam names beginning with provided letter and increase the counter every time a dam is identified. Output will be displayed in the Rich Edit Box.



## READING DATA ACROSS RELATED TABLES

Most professional databases have multiple tables and it is often necessary to read and manipulate data across the tables. In Grade 11, we will be learning the basics of reading data across multiple tables. In Grade 12, this concept will be expanded upon, including the use of SQL statements to manipulate datasets.

### THE FOREIGN KEY

Whenever we work with multiple tables, it is important to study the **Entity Relationship Diagram** and determine the Primary keys and Foreign keys as these are necessary for us to successfully extract and manipulate the data.



#### New words

**Entity Relationship Diagram** – to show the relationships of entity sets stored in a database



#### Take note

You will learn more about Database design in Chapter 9 IT Theory.

In the case of **DamsDB**:

- *DamID* is Primary Key for *tblDams*.
- *TownID* is the Primary Key for *tblTowns*.
- *DamID* is the Foreign Key in *tblTowns* and links the two tables.



### Guided activity 8.8

#### Finding the Dam linked to a Town (Project DamsApp\_p)

When the user clicks on a Town in the **dbGrid** linked to *tblTowns*, we will identify the *DamID* (Foreign Key) connected to the town. We will then loop through *tblDams* and search for the *DamID* (Primary Key) in *tblDams*. Once we find it, we will display the Dam Name and its associated River from *tblDams*.

**8.8.1** Create a *OnCellClick* event for *dbGrid* *dbgTowns*.

**8.8.2** Our first step will be to extract the *DamID* from *tblTowns*. This will enable us to identify the foreign key we will be searching for.

```
sDamID := dbmDamsDB.tblTowns['DamID'];
```

**8.8.3** Our next step is to point the other table to the first record, so that we can loop through it and search for the foreign key (*sDamID*).

```
dbmDamsDB.tblDams.First;
```

**8.8.4** We can now loop through *tblDams*, searching for the *DamID* extracted from *tblTowns*.

```
while NOT dbmDamsDB.tblDams.EOF do
```

**8.8.5** Inside the WHILE-Do-loop, we compare each Primary Key (*DamID*) in *tblDams* with the extracted Foreign Key from *tblTowns* (*sDamID*). When we identify a match, we have found the corresponding record in *tblDams*. We can then output the Dam Name and River Name (or any other fields, if necessary).

```
if dbmDamsDB.tblDams['DamID'] = sDamID then
begin
    redOut.Lines.Add('Dam Name: ' + dbmDamsDB.tblDams['DamName']);
    redOut.Lines.Add('River: ' + dbmDamsDB.tblDams['River']);
end;
```

**8.8.6** Finally, within the WHILE-Do-loop, we move the iterator to the next record:

```
dbmDamsDB.tblDams.Next;
```



### Guided activity 8.8

Finding the Dam linked to a Town *continued*

#### Notes:

- A flag can be used to terminate the loop once a match is found. This will improve the efficiency of the solution.
- If referential integrity is enabled in the database relationship, there is no need to cater for a 'match not found' message as the foreign key is guaranteed to exist in the other table.

**8.8.7** Save and test the application. By clicking on a town in the **dbGrid** linked to *tblTowns*, the corresponding dam data should be displayed in the RichEditBox.

The screenshot shows a Delphi application window. At the top is a **dbGrid** control displaying a table with columns: TownID, TownName, Province, WaterRestrictions, Population, and DamID. The first row, which has a blue selection bar, contains the values: 1 Willowmore, Eastern Cape, False, 7100, 6. Below the dbGrid is a **RichEditBox** containing the text: "Dam Name: Beervlei Dam" and "River: Groot River". To the right of the RichEditBox is a **Panel** labeled "Navigate" with buttons for First, Previous, Next, and Last. Below the panel are several buttons: Average Capacity, Search by Dam Name, Calculate % Full, List with Null Height, List of Dams post 2000, and Count names by letter.

### CONSOLIDATION ACTIVITY

Chapter 8: Databases

#### QUESTION 1

Select the correct option for the following questions.

- 1.1** Which of the following components is not used to display a table from a database in Delphi?
  - a. TADOConnection
  - b. TDBGrid
  - c. TADOTable
  - d. TADODatasource
- 1.2** Which of the following options best describes the relationship between the database and Delphi database components.
  - a. Database → DataSource → Connection → Table → Grid
  - b. Database → Connection → DataSource → Table → Grid
  - c. DataSource → Database → Table → Connection → Grid
  - d. Database → Connection → Table → DataSource → Grid
- 1.3** Which line below can be used to change the value of the 'Age' field of the selected record to 5.
  - a. `tblChildren['Age'] := 5;`
  - b. `tblChildren.Age := 5;`
  - c. `tblChildren.Age.Set(5);`
  - d. None of the above
- 1.4** What is the first step you need to follow when stepping through data from a database in your application.
  - a. Select the first record of your table.
  - b. Read the fields that you would like to use in your application.
  - c. Create a WHILE-Do-loop that runs until you reach the end of the file.
  - d. Move to the next record of your table.

**QUESTION 2**

Open the project **ShopApp\_p** in the 08 – ShopApp Folder.

The following user interface is provided.

**Shop App**

tblProducts									
ProdID	Descrip	SuppPrice	Stock	OnSpecial	Markup	SellPrice	ExpDate	SuppID	
P0001	Jelly Babies		3	15	True	4	0	2018/10/26	\$1
P0002	Jelly Tots		5	52	False	2	0	2018/09/09	\$4
P0003	Skittles		3.5	18	False	4	0	2018/07/15	\$2
P0004	Smarties		5	55	True	3	0	2018/11/20	\$3
P0005	Chocnuts		8.5	22	False	2	0	2018/07/05	\$11
P0006	Whispers		9.5	16	False	3	0	2018/10/22	\$10
P0007	Sour Worms		3.5	34	True	5	0	2018/10/27	\$9

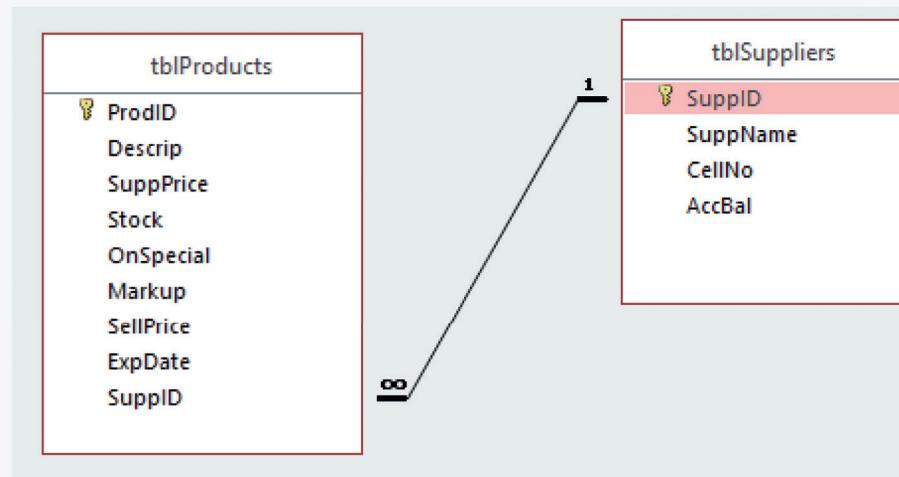
  

tblSuppliers			
SuppID	SuppName	CellNo	AccBal
S1	Khairah	0826220801	100
S2	Tarique	0815025939	0
S3	Emmanuel	0835930039	3030.53
S4	Nosipho	0825950393	50
S5	Yoline	0762939393	0

2.1 Specials   2.4 Total Product Value   2.7 Delete Product  
2.2 Count Specials   2.5 Decrease Stock   2.8 Search for Product  
2.3 Average Stock   2.6 Add Supplier   2.9 Calculate Selling Price

**Output Area:**  
redOut

The application is already connected to the Access database **ShopDB**, which uses has the following tables and relationship:



The DataModule provided is named **dbmShopDB** and contains two ADOTable components named *tblProducts* and *tblSuppliers*.

Write code for each event (2.1 to 2.11) to perform the tasks described below:

- 2.1** [2.1 Specials] Display a list of product names which are on special.

Products are on special when the *OnSpecial* field is set to TRUE.

The product names (*Descrip*) should be displayed to the Output Area.

- 2.2** [2.2 Count Specials] Count and display the number of products that are on special.

- 2.3** [2.3 Average Stock] Calculate and display the average stock available for all products.

- 2.4** [2.4 Total Product Value] Determine and display the Total Product Value (in Rand) for the selected product. The total product value is calculated by multiplying a product's Supplier Price (*SuppPrice*) by the number in stock (*Stock*).

**CONSOLIDATION ACTIVITY****Chapter 8: Databases *continued***

- 2.5** [2.5 Decrease Stock] When products are sold, their stock needs to be decreased accordingly.  
Write code to decrease the stock value of the selected product by 1.  
Display a confirmation message.
- 2.6** [2.6 Add Supplier] Write code to add a new Supplier to the Supplier table *tblSuppliers*.
- Generate a *SupplID* by combining the letter 'S' with the next available number in the table.  
(Hint: Use the RecordCount function + 1)
  - For the fields, *SuppName* and *CellNo*, use the InputBox function to get values from the user.
  - Assign the *AccBal* (Account Balance) field to 0.
  - Write the record to the database and display a confirmation message.
- 2.7** [2.7 Delete Product] Write code to Delete the selected product record after displaying a Confirmation Message.
- 2.8** [2.8 Search for Product] Write code to prompt the user to input a Product Name. Loop through *tblProducts* and if the product is found, display the product's stock. If the product is not found, display an appropriate message.
- 2.9** [2.9 Calculate Selling Price] The selling price for all products has been initialised to 0. The actual selling price is calculated using the following formula:

```
Selling Price ← Supplier Price * (1 + (Markup / 100))
```

Write code to loop through *tblProducts* and calculate and permanently store the Selling Price (*SellPrice*) for all products.

- 2.10** Write code to display the Supplier's Details of a particular product. When the user clicks on a Product in *dbgProducts*, extract the *SupplID* (foreign key) associated with that product. Loop through *tblSuppliers* and find the corresponding Supplier ID and display the Supplier's name and mobile number.

**Note:** each product has only 1 supplier.

- 2.11** Write code to display all products supplied by a particular supplier. When the user clicks on a Supplier in *dbgSuppliers*, extract the *SupplID* (Primary Key) associated with the supplier. Loop through *tblProducts* and identify all products that are associated with that supplier. Display a list of the products names.

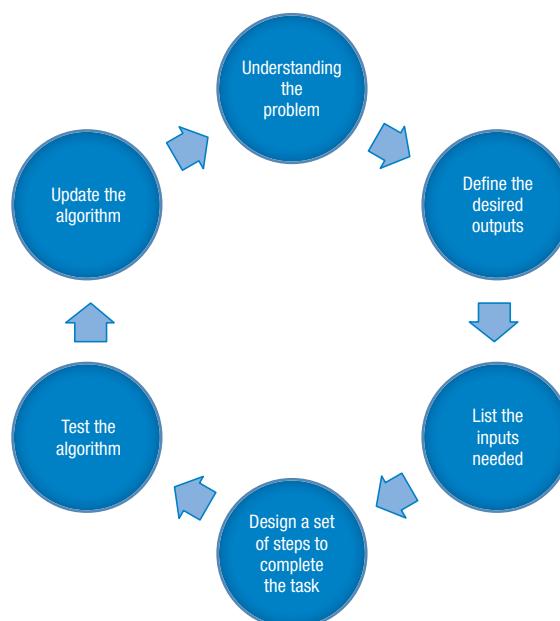
**Note:** a supplier may supply more than one item.

## ALGORITHMS

In Grade 10 you learned that an algorithm is a plan drawn up to solve a particular problem in a finite number of steps. An algorithm can be represented using:

- a flowchart: A flowchart is a tool that can be used to visually show how an algorithm works.
- pseudocode: Is written using the same logic and structures as a programming language, but it does not have to follow any programming language's syntax or rules.
- a programming language: An algorithm is translated into programming code for execution.

Remember that you must follow the problem solving steps to create high quality algorithms. These are shown in the Figure A.1 below.



**Figure A.1:** Problem solving steps to create high quality algorithms

While there are no specific rules about how to write an algorithm, once it is complete it should meet the following criteria:

- There may be zero input or many inputs.
- There must be a limited number of steps.
- The steps must be:
  - **unambiguous**.
  - Each step should:
    - consist of a single task.
    - be at the most basic level that cannot be broken into simpler tasks.
- All repetitions must have clear ending conditions.
- There must be at least one result (or output).



### WHAT'S AN ALGORITHM?



<https://www.youtube.com/watch?v=6hf0vs8pY1k>



### New words

**unambiguous** – not open to more than one interpretation

## FLOWCHART

In Grade 10 you also learned that a flowchart is a tool that is used to visually show how an algorithm works.

The following symbols are used to create a flowchart:

**Table A.1:** Symbols used in a flowchart

ELEMENT	FUNCTION	SHAPE
Terminal	Indicates the start and end of an algorithm.	Begin / End
Input / Output	Used to input data (reading data) or output data (displaying data).	Input / Output
Processing	Gives an instruction that the algorithm must execute.	Instruction
Decision	Shows a decision (or condition) which affects the algorithm's behaviour.	Decision
Connector	Connects one element of the algorithm to the next element. Show the direction in which you move from one element to the next.	→

Work through the following Guided activity to remind yourself about how to represent an algorithm.



### Guided activity A.1

#### Isolating the digits in a number

Read in an integer number, then determine and display each digit of the number on a new line.

##### Algorithm

Line 1: Read Number

Line 2: Digits = ''

Line 2: Repeat

Line 3: Remainder = Integer remainder of (Number /10)

Line 4: Number = Number /10 (the integer answer ignoring the decimal value)

Line 5: Digits = String(Remainder) + Digits

Line 6: Until Number = 0

Line 7: Display Digits



#### Take note

Display each digit on a new line in the order that they appear in the number.

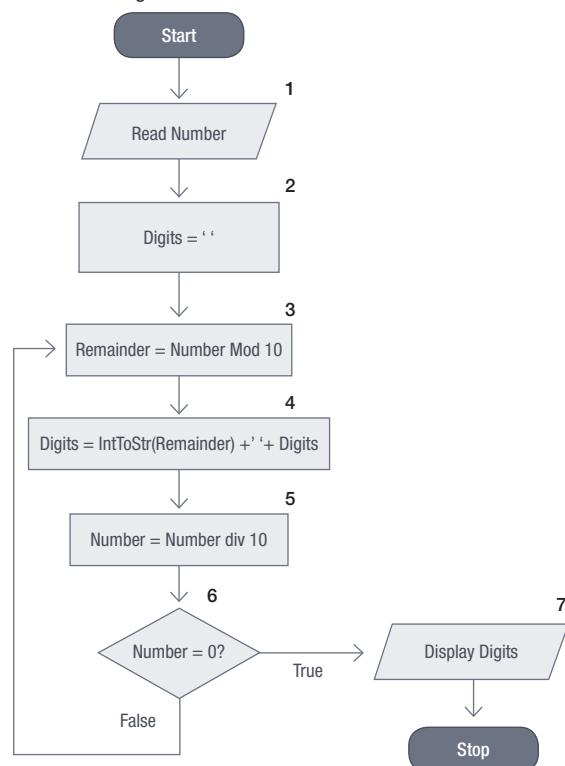


### Guided activity A.1

### Isolating the digits in a number *continued*

#### Flowchart

Here is an example of a flowchart for this algorithm:



#### Trace table

You can also draw the trace table for the flowchart.

STEP#	NUMBER	DIGITS	REMAINDER	NUMBER = 0?	OUTPUT
1	958				
2					
3			8		
4	95				
5		8			
6				F	
3			5		
4	9				
5		5 8			
6				F	
3			9		
4	0				
5		9 5 8			
6				T	
7					9 5 8



### Guided activity A.1

Isolating the digits in a number *continued*

#### IPO Chart

You can also use an Input, Processing and Output (IPO) chart to show exactly what input, processing and output is needed in your application. The IPO chart below is for the Isolate Digit algorithm:

INPUT	PROCESSING	OUTPUT
Number	Repeat Remainder $\leftarrow$ Number mod 10 Digit $\leftarrow$ Remainder + ‘ ‘ + Digits //Add a space and digits to Remainder and assign digits Number $\leftarrow$ Number div 10 Until Number = 0	Digits



### Activity A.1

**A.1.1** A number is a prime number if it has only 2 factors, the number 1 and the number itself.

Do the following:

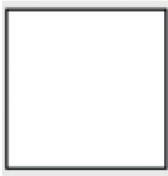
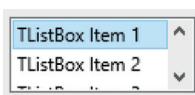
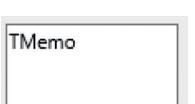
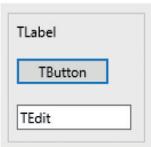
- Create an algorithm to determine whether a number is prime or not.
- Draw the corresponding flowchart for your algorithm.
- Draw an IPO chart for the problem.
- Trace through your flowchart using a value of 6.

In Grade 10 you also learned how to create a Graphical User Interface (GUI) by adding components.

Remember that components are added in the Delphi Design screen. Here are the components that you should know by now:

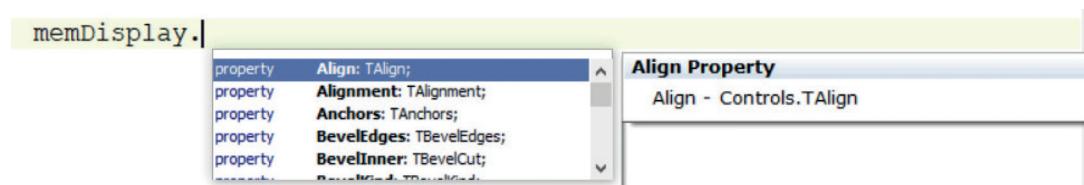
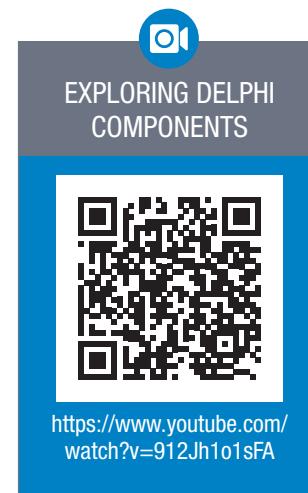
**Table A.2:** Components used when creating a GUI

NAME	DESCRIPTION	COMPONENT/IMAGE
TForm	The basis of the user interface in Delphi. All UI elements are added to the form.	
TPageControl	TPageControl is a container component that is used to create a multiple page dialog or tabbed notebook. TPageControl displays multiple overlapping pages that are TTabSheet objects. The user selects a page by clicking the page's tab that appears at the top of the control.	
TLabel	A simple label that can be used to display text in the UI.	
TButton	A button that can be pressed by users to trigger an event.	
TBitButton	TBitBtn has all the features of the TButton and adds the ability to set a glyph.	
TEdit	A text box that can be used to obtain text input from users.	
TCheckBox	A checkbox that users can tick or untick.	
TGroupBox and TRadioButton	A group of checkboxes (called radio buttons) where only one item can be selected at a time.	
TComboBox	A ComboBox combines an EditBox with a scrollable list.	
TSpinEdit	A TSpinEdit component is a TEdit with a TSpinButton attached. The Value of the EditBox is numerical. By pressing the up and down buttons of TSpinButton, the Value is increased or decreased.	
TImage	A frame that displays an image.	

NAME	DESCRIPTION	COMPONENT/IMAGE
TShape	Add a TShape object to a form to draw a simple geometric shape on the form.	
TListBox	A box showing a list of items that can be selected.	
TMemo	A box that works as a large TEdit component, allowing users to enter multiple lines of text.	
TRichEdit	A box that is similar to a TEdit and TMemo but offers more formatting properties.	
TPanel	A frame that can be used to group different elements together visually.	
TTimer	An invisible component that triggers an OnTimer event at intervals.	

#### Notes:

- When you create (or **instantiate**) a component, you are creating an “**instance**” of the component’s class. An instance is also referred to as an object.
- Remember the **naming convention** for components, for example, the prefix for a button component is btn.
- Each component has its own **properties** which can be changed on the Properties Tab in the Object Inspector or through programming code.  
Example: btnSelect.Caption := ‘Choose’. The Caption property of the button is changed to ‘Choose’.
- Components also have **methods**. Methods are predefined instructions. Whenever a method is activated, it activates the lines of code attached to the method for execution.  
Example btnSort.Hide.
- When you type a component’s name followed by a dot(.), all the component’s properties and methods (function and procedures) come up in a scrollable list for selection.  
Example:



### Example of IPO Chart with components for the Isolate Digit Algorithm

INPUT	PROCESSING	OUTPUT
Number	Repeat <i>Remainder</i> $\leftarrow$ Number mod 10 Number $\leftarrow$ Number div 10 Add <i>Remainder</i> to <i>Digits</i> + a space Until Number = 0	Digits
Input components		Output components
edtNumber		memDisplay

## EVENTS

Delphi is an event-driven programming language. An event is linked to some code which responds to some action. This action may be created by user interaction, the system or code logic. An event handler manages the execution of an event. An example is the *OnClick* event for a button.

In Grade 10 you learnt about the following event handlers:

- **OnClick:** Almost every visible component in Delphi can have an *OnClick* event that activates when the user clicks on the component. It is most often used with buttons but can also easily be used with labels and images.
- **OnCreate:** An event specific to the form, it activates when the form is created the first time. As a result, it can be used to add code that will run every time the application is opened.
- **OnTimer:** An event specific to the timer, it activates every few milliseconds, based on the value of the interval property of the timer.

New words

**instantiate** – represent as or by an instant

**instance** – an example or single occurrence of something

**naming convention** – to name things (generally agreed scheme)

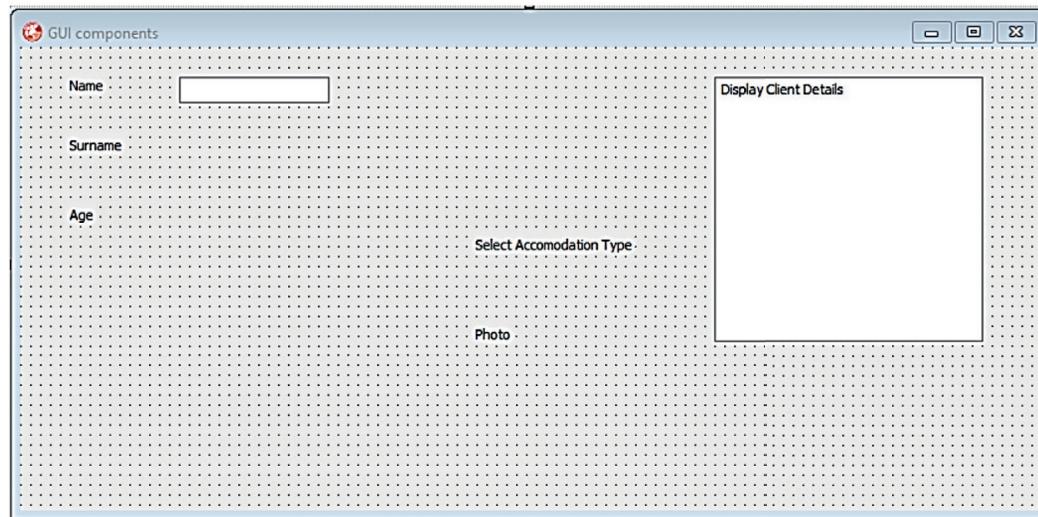
**properties** – the components or building blocks

**methods** – predefined instructions



### Activity A.2

- A.2.1 Open the **GUIComponents\_p** project located in the Annex – GUI Components folder.





## Activity A.2

*continued*

Modify the form so that it displays as shown below:

The screenshot shows a Windows application window titled "GUI components". Inside, there are several user interface elements:

- Name:** A text input field.
- Surname:** A text input field.
- Age:** A numeric spin box with a value of 0.
- Gender:** Radio button group with options "Male" and "Female".
- Places Visited:** A checkbox group with options "Paris", "London", and "Rome".
- Select Accommodation Type:** A dropdown combobox labeled "cmbType".
- Photo:** A placeholder area for a photo, indicated by a dashed rectangle.
- Display Client Details:** A large empty rectangular area.
- Display:** A button at the bottom right.

Note the following:

- Add all the missing components using appropriate captions.
- Use the component naming convention to name the components appropriately and meaningfully based on their purpose.
- The values for the combobox:
  - 2 Star
  - 3 Star
  - 4 Star
  - 5 Star
- Save and run the project.

**A.3****Variables**

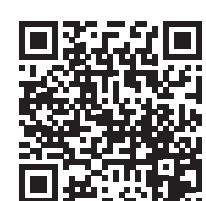
In Grade 10 you learnt that variables are used to store information for later retrieval. You can assign values to variables or read the value stored in a variable. Study the table below to remind yourself about the variable types you learnt about.

**Table A.3: Variable types**

VARIABLE TYPE	DESCRIPTION
String	Strings are made up of a sequence of numbers, letters and symbols. A string variable can contain anything from a single character to many characters. For example: <code>sName := 'Stefan'</code>
Char	Chars, or characters, are made up of a single number, letter or symbol. As with strings, chars need to be surrounded by single quotation marks. For example: <code>cGender := 'm';</code>
Integer	An integer can contain any positive or negative whole number (i.e. a number without a decimal). These numbers can be used in calculations. For example: <code>iMaximum := 5 + 4;</code>
Real/Double	A real/double can contain any positive or negative numbers (including decimal numbers). As with integers, these numbers can be used in calculations. For example: <code>rTotal := 1.2;</code>
Boolean	Booleans can contain only one of two values: TRUE or FALSE. This is often used in conditional statements, where the program completes a specific task if a condition is TRUE. For example: <code>bFactor := True;</code>

All variable and component names must be unique. When naming variables and components, remember to adhere to the following naming conventions:

- It must start with a letter, followed by letters or digits or the underscore(\_) character. No other special characters are allowed.
- Names should describe the data they will contain.
- Names should use camel case, which means the first word or letter is in lowercase, and each word afterwards starts with an uppercase letter.  
Example: `countEvenNumbers`.
- Component names should start with three letters describing the component.  
Example: `btnChangeText` for a button or `lblHelloWorld` for a label.
- Variable names should start with a single letter describing the type of variable:
  - i Integer
  - r Real/Double
  - s String
  - c Char
  - b Boolean
- For example: `sName` for a string or `iMaximum` for an integer.


**LEARNING ABOUT  
VARIABLES**


<https://www.youtube.com/watch?v=vKmLQcuz5ds>



## CONVERTING REAL NUMBERS TO STRINGS



<https://www.youtube.com/watch?v=7YqvMLJb64>

## NUMBERS

Numbers are represented by:

- Integers (numbers without a decimal point)
- Reals (numbers with a decimal point)

The operations that can be performed on numbers are listed in Table A.4 below:

**Table A.4: Operations**

MATHEMATICAL OPERATOR	MATHEMATICAL OPERATION
+	Addition
-	Subtraction
*	Multiplication
/	Real division
DIV	Integer division
MOD	Modulus (Remainder after integer division)

If you add, subtract, multiply, divide (DIV) or modulus (MOD) two integer numbers, then the answer will be an integer number.

EXPRESSION	ANSWER
$37 + 5$	42
$37 - 5$	32
$37 * 5$	185
$37 \text{ DIV } 5$	7
$37 \text{ MOD } 5$	2

If you add, subtract, multiply or divide (/) two real numbers, then the answer will be a real number.

EXPRESSION	ANSWER
$37.83 + 5.2$	43.03
$37.83 - 5.2$	32.63
$37.83 * 5.2$	196.716
$37.83 / 5.2$	7.275
$37.83 / 5$	7.566

In the last row of the table above: 5 is included in the real arithmetic calculation  $37.93 / 5$  because:

- Integers are a subset of real numbers. Every integer can be represented as a Real
- 5 can be written as 5.0.

## ORDER OF PRECEDENCE

Basic mathematical operators in mathematical expressions are evaluated using the BODMAS-rule just like you would do when working in Mathematics. This is known as the order of precedence.

**Table A.5:** The order of precedence

OPERATOR	PRECEDENCE
Brackets ()	Highest level
* / DIV MOD	Second level – from left to right – whichever one comes first
+ -	Third level – from left to right – whichever one comes first

### Example A.1

### Using the order of precedence in calculations

$$\begin{aligned} & 2 + 3 * 26 / (16 - 3) - 5 \\ & = 2 + 3 * 26 / 13 - 5 \quad (\text{Level 1 – Brackets}) \\ & = 2 + 72 / 13 - 5 \quad (\text{Level 2 – multiplication}) \\ & = 2 + 6 - 5 \quad (\text{Level 2 – division}) \\ & = 8 - 5 \quad (\text{Level 3 – addition}) \\ & = 3 \quad (\text{Level 3 – subtraction}) \end{aligned}$$

## CONVERSION OF DATA TYPES AND FORMATTING

Often, we need to convert between data types. The functions used to convert data types is shown in Table A.6 below.

**Table A.6:** Function used to convert data types

FUNCTION	DESCRIPTION	EXAMPLE
StrToInt(sInput)	Converts the sInput value to an integer.	StrToInt('500');
StrToFloat(sInput)	Converts the sInput value to a real.	StrToInt('500.1');
IntToStr(iInput)	Converts the iInput value to a string.	IntToStr(500);
FloatToStr(rInput)	Converts the rInput value to a string.	FloatToStr(500.1);

The FloatToStrF function also converts a float number into a string format:

- Syntax: FloatToStrF(Value, Format, total number of digits, number of decimal digits)
- Format can be any format from the table below:

FORMATS ALLOWED	DESCRIPTION
ffCurrency	Formats the value with the currency included
ffExponent	Formats the value in scientific notation
ffFixed	Formats the value with the number of decimal as specified
ffGeneral	General number format. Includes decimals only when required
ffNumber	Corresponds to ffFixed, but uses the thousands separator



## MATH OPERATIONS



[https://www.youtube.com/watch?v=s\\_PNKJ2x1x4](https://www.youtube.com/watch?v=s_PNKJ2x1x4)

## MATHEMATICAL METHODS

You learnt about the following mathematical methods in Grade 10:

METHOD	PURPOSE
Random()	<p>Generates a random number from 0 to less than 1.</p> $rNumber := Random()$ <p>If you want to generate a number from <math>a</math> to <math>b</math>, then use the formula:</p> $iNumber := Random(b - a + 1) + a$ <p>Example: To generate a random number in the range 10 to 99 the code is:</p> $iNumber := Random(90) + 10$
Round()	<p>The Round function rounds a real number to an integer value.</p> <p>The real number <math>X</math> is rounded to the nearest whole number. If <math>X</math> is exactly halfway between two whole numbers, the result is always the even number of the two whole numbers.</p> <p>Examples: <math>iAns := Round(12.4)</math> is 12  <math>iAns := Round(12.5)</math> is 12 (rounds down to even)  <math>iAns := Round(12.6)</math> is 13  <math>iAns := Round(13.5)</math> is 14 (rounds up to even)</p>
Trunc()	<p>The Trunc function truncates (removes or 'chops off') the decimal part of a real number. It returns an integer after the truncation.</p> <p>Examples: <math>iAns := Trunc(12.4)</math> is 12  <math>iAns := Trunc(12.8)</math> is 12</p>
Sqr()	<p>The SQR function returns the square of an integer or real number.</p> <p>The return value is the same type as the number being squared.</p> <p>Examples: <math>iSqrAns := Sqr(5)</math> is 25  <math>rSqrAns := Sqr(6.2)</math> is 38.44</p>
Sqrt()	<p>The SQRT function returns the square root of a number. The result type is always real. Remember that the square root of a negative number is undefined.</p> <p>Examples: <math>rAns := Sqrt(31.36)</math> is 5.6  <math>rAns := Sqrt(144)</math> is 12.0</p>

### Note:

- To prevent Delphi from generating the same set of random numbers, you need to place the randomize command at the start of your application:  
Example:

```
begin
  ...
  Randomize
  iNum := Random(20) + 1;
  rValue := Random();
  ...
end;
```

- This will ensure that each time your application runs, a new set of random numbers will be generated

## FORMATTING NUMBERS IN A MEMOBOX

You can format data in a MemoBox using the Format function.

```
Format('%8s%10.2f',['Average',53.861])
```

### Note:

- The format function has two arguments
- The **first argument** '%8s%10.2f' is a string that holds instructions for formatting. In the control code used:
  - % symbol indicates that the text which follows is formatting instructions and not normal text.
  - There are two formatting instructions in '%8s%10.2f'
- First formatting instruction %8s: 8 indicates that the string to be displayed must contain 8 characters.
- Second formatting instruction %10.2f: 10 indicates that the string to be displayed must contain 10 characters. 2 indicates that the 10 characters must have 2 decimal points.
- The s and f indicate the type of data that will be formatted. Data types are indicated as follows:

SYMBOL	DATA TYPE
s	String
f	Floating point number
d	Integer
m	Monetary value (currency symbol will be shown – what symbol shows depends upon the Windows regional settings)

- The **second argument** ['Average',53.861] holds the value/s that needs to be converted into a formatted string.
  - These value/s must appear within square brackets.
  - The data types in the formatting instructions must match the data types of the values in the second argument.
  - The values can be variables or constants.
- If the display in the MemoBox has to be aligned correctly, you need to change the font property of the MemoBox to a fixed font (where the shape of the character does not influence the output) such as Courier New or Lucida Console.

Here are some examples:

```
memDisplay.Lines.Add(Format('%10s%10.2f', ['value', 45.345]));
memDisplay.Lines.Add(Format('%10s%10s%10s', ['Number', 'Square', 'Square Root']));
memDisplay.Lines.Add(Format('%10d%10d%10.2f', [10, Sqr(10), Sqrt(10)]));
```

## SCOPE OF A VARIABLE

Variables can be declared as follows:

- At the beginning of the unit; or
- They can be declared in an **event**, **procedure** or **function**.

Variables declared in an event handler, procedure or function are only created in the computer's memory at the start of the event, procedure or function execution and only exist as long as the event, procedure or function is being executed. These variables cease to exist once the event, procedure or function has terminated. We call these variables **local variables**, because they have a local scope, that is, they cannot be used in another event, procedure or function.



### New words

- first argument** – is a string that holds instructions for formatting
- second argument** – holds the values that needs to be converted into a formatted string
- event** – an occurrence of something
- procedure** – an official way of doing something
- function** – the operation of something in a particular way
- local variable** – variables that have a local scope



### Take note

We know that a variable must first be declared before it is used in a program. However, where it is declared in a program, determines its scope.

We call variables declared at the beginning of a program **global** or **non-local** variables. These variables have class scope because they can be used in any event, method or procedure. You will learn about this in later chapters.



### Take note

We will only declare global/non-local variables in the Var and implementation section of the Unit.

## COMMENT STATEMENTS

Comments are used by programmers to make a program more easier to read and understand its purpose.

Comments are used to explain:

- what a section of programming code does.
- how a certain piece of code works (the logic used).

Any text preceded by two forward slashes (//) is considered a comment and is ignored by Delphi. You can also block comment code a group of statements by using curly bracket{} so that the code within the {} is ignored during execution.



### New words

**global** – is a programming language construct, a variable that is declared outside and function and is accessible to all the functions throughout the program

**non-local** – is a variable that is not defined within the local scope



### Activity A.3

- A.3.1** Open **DistanceConversion\_p** project located in the Annex – Distance Conversion folder. Distance in certain countries are measured in inches, feet, yards and miles as follows:

$$12 \text{ inches} = 1 \text{ foot}$$

$$3 \text{ feet} = 1 \text{ yard}$$

$$1760 \text{ yards} = 1 \text{ mile}$$

Write code for the [Convert] button that will read in the distance in inches and convert the distance to foot (feet), yards and miles.

The screenshot shows a Windows application window titled "Distance Conversion". In the top-left corner, there is a small icon of a globe. On the right side of the title bar are standard window control buttons for minimizing, maximizing, and closing. Below the title bar, there is a text input field labeled "Distance in Inches" containing the value "126720". To the right of this input field is a "Convert" button. At the bottom of the application window, there is a table showing the conversion results:

_inches	126720
Feet	10560
Yards	3520
Miles	2

- A.3.2** Open **TileCost\_p** project from the Annex – Tile Cost folder. Andile wants to tile his room floor. He must calculate the square metre ( $m^2$ ) that he requires for his room. In addition, he needs to add 10% additional  $m^2$  to his requirements to cater for breakages. He likes a tile that costs R150 per  $m^2$ . The tiles are sold in boxes. Each box holds tiles with a measurement of 2.3  $m^2$ . Write code for the [Tile Calculator] button to read in the length and breadth of Andile's room, determine and print the following:

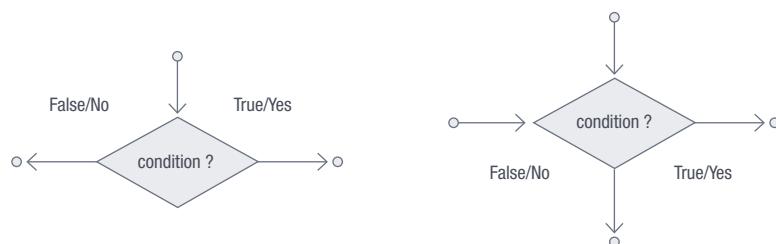
- The total  $m^2$  tiles required inclusive of breakages.
- The number of boxes of tiles that must be bought.
- The total amount that will be paid for the boxes of tiles bought. Format the answer to currency.
- Add comments to your code.
- Save and execute your program.

The screenshot shows a Windows application window titled "Calculating Tile Cost". In the top-left corner, there is a small icon of a globe. On the right side of the title bar are standard window control buttons for minimizing, maximizing, and closing. Below the title bar, there is a section titled "Enter Dimensions of the Room". It contains two text input fields: "Length" with the value "4.5" and "Breadth" with the value "3". To the right of these fields is a "Tile Calculator" button. At the bottom of the application window, there is a table showing the calculated results:

Tile Square metres	14.85
Boxes Required	7
Cost of Tiles	R2 415.00

In every day life you often make decisions to solve a problem. The same is true for when you write or develop a program – you have to use the decision-making process regularly. In the decision-making process, conditions (criteria) are tested. A condition evaluates one of the two Boolean values: TRUE or FALSE. The outcome of the condition determines which one of the two paths (the YES/TRUE path or the NO/FALSE path) will be followed. We say that decision making causes branching to occur in the normal sequential program flow.

In a flowchart, a decision is represented by a diamond symbol. Here are some examples:



#### Take note

Branching along TRUE or FALSE paths.

## BOOLEAN EXPRESSIONS

A Boolean expression is an expression that evaluates to TRUE or FALSE. There are three categories of Boolean expressions:

- Boolean variable
- Simple Boolean expressions
- Compound (complex) Boolean expressions.

## BOOLEAN VARIABLES

You can assign a Boolean value to a Boolean variable, for example:

- bFound := True;
- bValid := False;

## SIMPLE BOOLEAN EXPRESSIONS

Comparison operators are used to create conditions:

COMPARISON OPERATOR	DESCRIPTION	EXAMPLE OF CONDITIONS
>	Greater than	6 > 7
>=	Greater than or equal to	iNumber >= 50
=	Equal to	sStr[3] = sStr[8]
<	Less than	rNum < 56.3
<=	Less than or equal to	5 <= 8
<>	Not equal to	item <> 9



## GLOBAL AND LOCAL VARIABLES



[https://www.youtube.com/watch?v=LPRLU1\\_dGJE](https://www.youtube.com/watch?v=LPRLU1_dGJE)

## COMPOUND BOOLEAN EXPRESSIONS

We use logical operators to create compound Boolean expressions. The logical operators used are:

- AND logical operator
- OR logical operator
- NOT logical operator

In this section we will look at each logical operator in a bit more detail.

### AND LOGICAL OPERATOR

The AND operator is used to test two conditions in the format:

(condition1) AND (condition2)

Both conditions must evaluate to true for the result to be true. If either condition1 or condition2 is false, then the result will be false, for example: if A = 5 and B = 7 then:

BOOLEAN EXPRESSION	CONDITION1	CONDITION2	RESULT
(A > 2) AND (B <= 7)	True	True	True
(A > 2) AND (B > 7)	True	False	False
(A = 3) AND (B > 5)	False	True	False
(A = 3) AND (B > 7)	False	False	False

### OR LOGICAL OPERATOR

The OR operator is used to test two conditions in the format:

(condition1) OR (condition2)

Both conditions must evaluate to false for the result to be false. If either condition1 or condition2 is true, then the result will be true, for example: if A = 5 and B = 7 then:

BOOLEAN EXPRESSION	CONDITION1	CONDITION2	RESULT
(A > 2) OR (B <= 7)	True	True	True
(A > 2) OR (B > 7)	True	False	True
(A = 3) OR (B > 5)	False	True	True
(A = 3) OR (B > 7)	False	False	False

### NOT LOGICAL OPERATOR

The NOT operator negates the result of the condition in the format:

NOT(condition)

If the condition evaluates to true, then NOT(condition) evaluates to false. If the condition evaluates to false, then NOT(condition) evaluates to true, for example: if A = 5 and B = 7 then:

BOOLEAN EXPRESSION	CONDITION	RESULT
NOT(A > 2)	True	False
NOT(B > 7)	False	True

## THE ORDER OF PRECEDENCE

You learnt about the order of precedence of mathematical operators. Similarly, Boolean operators also have order of precedence. The order of precedence from highest to lowest precedence is as follows:

OPERATOR/S	LEVEL
NOT	1 (highest level of precedence)
* , /, DIV, MOD, AND	2
+ , - , OR	3
=, <>, <, >, <=, >=	4

### Example A.2

#### The order of precedence

Assume the value of the variables is as follow: Gender = 'F', Age = 15 and Sport is 'Netball'.

Evaluate the following Boolean expressions:

(Gender = 'F') OR (Age > 10) AND (Sport = 'Cricket')

= TRUE OR TRUE AND FALSE

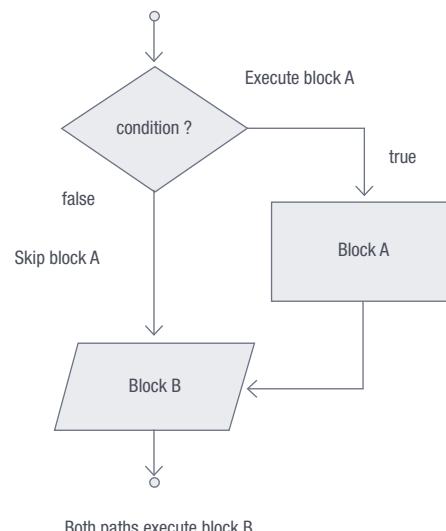
= TRUE OR FALSE

= TRUE

## IF-THEN STATEMENT

In Delphi, you can use the IF-THEN statement to make decisions. IF and THEN are keywords in Delphi. The IF-THEN statement executes the statement/s following the THEN keyword if the condition is true and skips the execution of these statement/s when the condition is false.

In the flow chart below, block A represents statement/s following the THEN keyword. Note how block A is skipped if the condition is FALSE.



#### Remember!

You cannot use IF and THEN as variable names. If you do, then you will get an error 'Declaration expected but IF found'.

## SYNTAX OF IF-THEN STATEMENT

Delphi syntax of an IF-THEN statement, if one statement follows the THEN keyword:

```
If <condition> then  
    <statement1>;
```

The THEN-keyword is not followed by a semicolon as it is not the end of the statement. The <statement1> is followed by a semicolon and ends the IF-THEN statement.

Sometimes, if a condition is true, we need to execute more than one statement.

The syntax of an IF-THEN statement if more than one statement follows the THEN keyword. Multiple statements are grouped together within the keywords Begin and End as shown below. It is seen as one group of statements to be executed in the THEN part.

```
If <condition> then  
begin  
    <statement1>;  
    <statement2>;  
    ...  
end;
```

### Example A.3

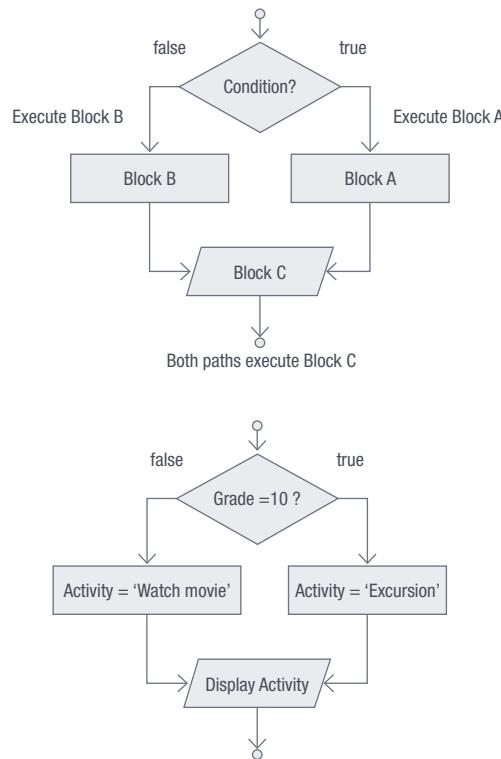
```
If Gender = 'F' then  
begin  
    iNumGirls := iNumGirls + 1;  
    lblNumGirls.caption := IntToStr(iNumGirls);  
end;
```

## IF-THEN-ELSE STATEMENT

In many situations you may want something to happen if a condition is met, and something else to happen if the condition is not met. For example, all Grade 10 learners will go for an excursion, all other learners will watch a movie. A decision needs to be made on the grade of a learner to determine which activity they will participate in. In the condition if the grade is 10 then the learner will go on an excursion, else the learner will watch a movie.

The IF-THEN-ELSE statement executes statement/s following the THEN keyword if the condition is true and executes statement/s following the ELSE keyword when the condition is false.

In the flow chart below, block A represents statement/s following the THEN keyword. Block B represents statement/s following the ELSE keyword. Both paths execute Block C.



### SYNTAX OF IF-THEN-ELSE STATEMENT

Delphi syntax of an IF-THEN-ELSE statement if one statement follows the THEN and ELSE keywords:

```
If <condition> THEN
  <statement1>
ELSE
  <statement2>;
```

- The THEN and ELSE keywords are not followed by a semicolon
- The statement before the ELSE statement does not have a semi-colon
- If more than one statement appears in the THEN or ELSE part, then the statements must appear in a BEGIN...END block
- A single statement does not appear in a BEGIN...END block

```
If <condition> THEN
begin
  <statement1>;
  ...
  <statement4>
End
ELSE
begin
  <statement5>;
  <statement6>;
  ...
end;
```

## NESTED IF-THEN STATEMENTS

A nested IF-THEN statement occurs when one conditional statement is placed inside another conditional statement. By doing this your program first checks if the outer condition has been met before looking at the inner conditional statement.

### SYNTAX OF THE NESTED IF STATEMENT

```
IF <condition1> THEN
    IF <condition2> THEN
        <statement1>
    ELSE
        begin
            <statement5>;
            <statement6>;
            ...
        end;
```

#### Example A.4

The code snippet below shows an example of a nested-if statement.

```
if iValue > 0 then
begin
    if iValue < 100 then
        ShowMessage('Number is between 0 and 100')
    else
        ShowMessage('Number is 100 or above');
end;
```

#### Note:

- The outer conditional statement (*iValue > 0*) is tested first. If this condition is true, the inner conditional statement (*iValue < 100*) is tested next.
- If the conditional statement (*iValue < 100*) is true, then the *ShowMessage('Number is between 0 and 100')* statement is executed else the *ShowMessage('Number is 100 or above')*, is executed.

### Example A.5

A company is handing out bags using the following criteria:

- If a person is a male and drinks Coke then he qualifies for a bag; otherwise no bag
- If a person is female and drinks Fanta then she qualifies for a bag; otherwise no bag

```
if Gender = 'Male' then
begin
  if Drink = 'Coke' then
    ShowMessage('You get a bag')
  else
    ShowMessage('No bag');
end;
if Gender = 'Female' then
begin
  if Drink = 'Fanta' then
    ShowMessage('You get a bag')
  else
    ShowMessage('No bag');
end;
```

## CASE-STATEMENTS

Another type of decision making structure in Delphi is the CASE-statement. Instead of using a sequence of cascading IF-THEN-ELSE-IF for decision making, the CASE-statement provides a tidy way of dealing with decision making. The cascading IF-THEN-ELSE-IF allows you to execute a block code among many alternatives. If you are checking on a value of an ordinal type variable using an IF-THEN-ELSE-IF, it is better to use the CASE-statement.

The CASE-statement uses the following syntax:

#### Syntax of CASE-statement

```
CASE <variable> OF
  value1 : statement1;
  value2 : statement2;
  value3 : statement3;
  ELSE
    statement4;
end;
```

#### Notes:

- Start with the keyword CASE followed by a <variable> followed by the keyword OF. There is no semicolon after the OF keyword.
- The CASE-statement does not have a begin but has an end.
- <variable> represents a variable name. The data type of the variable can be integer or character.
- Different cases:
  - Value1, Value2 and Value3 refer to the cases against which <variable> will be compared and must be of the same data type as <variable>.
  - Each case is followed by a colon.
  - The statements following the colon indicates what code must be executed for that case.

Example for case Value2, Statement2 must be executed.

- Again, if more than one statement need to be executed in a case, it must be in a BEGIN... END block
- A case can be represented by:
  - An integer : 5
  - A character : 'A'
  - Range : 3..5
- If the same action is required for more than one case then cases can be grouped as follows:

#### Case cLetter Of

```
'a','e','i','o','u': ShowMessage ('Vowel'); //grouped cases are
//separated by comma

Else
    ShowMessage ('Not a vowel')
End;
```

- When a match is found during the comparison, control of the program passes to that case and the code of that case is executed and the CASE-statement is exited.
- The ELSE statement is optional. It is used to provide a default if none of the cases match the <variable>.

## IN OPERATOR

You can also test if an element is included in a set of values using the IN operator. The test returns true if the element is found in the set of values; otherwise the test returns false.

#### IN operator Syntax

```
Element IN [set of Values]
```

#### Notes:

- Element is the variable that is being tested against the s
- Element variable/value can only be an ordinal data type (integer, char and Boolean). The values in the [set of values] must match the data type of element
- The IN operator checks whether the Element is found in the set [set of Values]. The set of values appear within square brackets []

#### Example A.6

#### Case iGrade of

```
10: ShowMessage ('Go to movies')
11: ShowMessage ('uShaka Marine')
12: ShowMessage ('Drakensberg Excursion')
End;
```

### Example A.7

```
Var cLetter:char;  
....  
bFound := cLetter IN ['a','b','c'];  
if bFound then  
    ShowMessage('Letter Valid')  
else  
    ShowMessage('Letter Invalid');  
....
```

#### Notes

- If cLetter has the value 'a' then bFound will be true and 'Letter Valid' will display. Remember that 'a' is not the same as 'A'
- If cLetter has any other value than 'a', 'b' or 'c' then bFound will be false and 'Letter Invalid' will display.

As with CASE-statements, when using the IN operator the set of values can be:

- a range of values (with the minimum and maximum values separated by two full stops).
- individual values (with the values separated by commas).
- a combination of a range of values and individual values.

**Table A.7: Examples of statements with IN operator**

If iMonth in [1..5] then ...	Checks whether iMonth is one of the first 5 months. The set of number can also be written as a range [1..5] because the numbers are inclusive
If iNum in [1..5, 8..53] ...	Checks whether iNum is in the set 1 to 5, 8 and 50 to 53
If cLetter IN ['A'..'Z', 'a'..'z'] then ...	'A'..'Z' and 'a'..'z' indicate a range of uppercase and lowercase letters

When creating a conditional statement, the IN-operator is used in place of the equals operator. This is because you are using the conditional statement to see if your value can be found inside the set.

The screenshot shows a Windows application window titled "Client Data". On the left, there are input fields for "Name" (text box), "Surname" (text box), "Age" (spin box with value 0), "Gender" (radio buttons for "Male" and "Female"), and a "Places Visited" section with checkboxes for "Paris", "London", and "Rome". Below these is a dropdown menu for "Select Accommodation Type". On the right, there is a large empty rectangular area labeled "Display Client Details". At the bottom right of the form is a "Display" button.

- A.4.1** Open the **ClientData\_p** project in the Annex – Client Data folder. The form is designed to capture the details of a travel client.

Do the following:

- The image component imgPicture must display the *Destinations* image found in your *Client Data* folder. Set the *stretch* and *proportional* properties to true.
- Create an OnClick event for the [Display] button to:
  - Read the name, surname, age, gender, places travelled and Accommodation type for each client.
  - A client is classified according to the number of places visited as follows:

NUMBER OF PLACES VISITED	CATEGORY
1 Place	Occasional
2 Places	Frequent
3 Places	Avid
No Place	Starter

- Display the name, surname, age, gender, places travelled, Category and Accommodation type for each client.

#### Example of sample output

The screenshot shows the same "Client Data" application window. The input fields now contain "Name: Jenny", "Surname: Peters", "Age: 26", and "Gender: Female". In the "Places Visited" section, only the checkbox for "London" is checked. The "Select Accommodation Type" dropdown is set to "3 Star". On the right, the "Display Client Details" panel contains the following information:
 

Name:	Jenny
Surname:	Peters
Age:	26
Gender:	Female
Places visited	London
Traveller Type	Occasional
Accommodation Type	3 Star

 Below this panel is a small image of a bridge at night over water.

## A.5 Looping

Loops repeat certain lines of code until a specific condition is met. In most programming languages, including Delphi, there are three types of loop constructs:

- FOR-loop
- REPEAT-loop
- WHILE-loop

These three loops generally do the same thing: They repeat a number of instructions until some condition is met. However, they differ in how they decide when to start and stop the loops.

### FOR-LOOP

The FOR-loop is known as an iteration loop since it runs for a specific number of iterations. The syntax for the FOR-loop is:

```
FOR iCount ← Minimum to Maximum do
  BEGIN // body of loop
    // 1 or more instruction(s)
  END // NEXT iCount called
```

The FOR-loop needs a counter, which will be called *iCount* in the next example.

- The loop starts with the reserved/keyword word: FOR and the same line ends with a DO.
- After the loop definition line, we have a start-end-block of code, that will be executed several times.
- Counting from a minimum value being 1 (in the example) to a maximum being 10 (as displayed). For each iteration the loop increases the counter iCount with one (1). If one more than the maximum value (11) is reached by the counter, the iteration comes to an end and the next statement following the loop is executed.

All loops work on the I-T-C (initialise-Test-Change) principle:

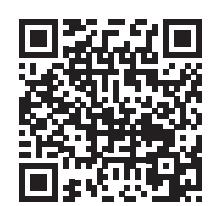
I for Initialise: The variable/s that appear in the loop condition must be initialised. These variables are called the loop control variable/s.

T for Test: The loop control variable/s in the condition of the loop is tested.

C for Change: Inside the body of the loop the loop condition variable/s should be changed to ensure that the loop terminated at some point.



#### FOR LOOPING



[https://www.youtube.com/watch?v=kYgXRi\\_m0Ak](https://www.youtube.com/watch?v=kYgXRi_m0Ak)

### Example A.8

#### The ITC principle for the For-loop below

```
...
iSum := 0;
For i := 2 to 5 do
Begin
    iSum := iSum + i;
    memDisplay.Lines.Add(IntToStr(i));
End; //for loop
memDisplay.Lines.Add(IntToStr(i));
...
```

The loop control variable is i.

I: Initial value of i is 2

T: Test is  $i \leq 5$

C: The value of i is incremented by 1 when the end of the loop is reached



#### New words

**incremental** – relating to or denoting an increase or addition

**decremental** – the act or process of decreasing or becoming gradually less

#### IN DELPHI WE DO GET TWO TYPES OF FOR-LOOPS:

The **incremental** FOR-loop can be identified by the command **TO**:

For  $iCount := 1$  to 10 do ... . It typically starts with a small value, which is increased by one each time to reach the maximum or the end-value.

Delphi source code for an incremental for-loop:

```
Var
  iCount : Integer;
begin
  for iCount := 1 to 10 do
  begin
    // execute instruction(s)
  end;           // increment by 1 (always 1)
end;
```

The **decremental** FOR-loop can be identified by the command **downto**: For  $iCount := 10$  downto 1 do ... .

It starts with a large (maximum) value to be decreased by one each time to reach the minimum or the end-value.



#### Did you know

When used in a for-loop, most people simply call the loop-counter variable *i* or *j*.

Delphi source code for a decremental for-loop:

```
Var
  iCount : Integer;
begin
  for iCount := 10 downto 1 do
  begin
    // execute instruction(s)
  end;           // decrement by 1 (always -1)
end;
```

Since the integer loop control variable is either incremented or decremented with each iteration, these loop control variables can be used within the body of the loop if you need an increasing or decreasing integer. For example: using the loop control variable to access each character in a string.

## REPEAT LOOP

The REPEAT...UNTIL loop has got its control structure at the end of the loop-block. That means one iteration is always executed before the condition is tested! The REPEAT...UNTIL loop is called a post-conditional loop because it checks whether it should continue running at the end of each loop.

### Syntax of REPEAT...UNTIL loop

```
Repeat  
    // one or more instruction(s)  
Until (condition = true);
```

#### Note:

- The REPEAT...UNTIL loop structure does not need a BEGIN and END line.
- The REPEAT marks the begin of the loop and UNTIL marks the end of the loop.
- The statements in the loop body are executed repeatedly until the condition (a Boolean expression) evaluates to true, that is, the loop body executes when the condition is false and terminates when the condition becomes true.
- The condition is tested only after the loop body has been executed.
- The loop body is executed at least once.
- We say that this is an ICT loop. The test control variable is initialised before the loop. The change takes place within the loop and the test takes place at the end of the loop.

## WHILE-LOOP

The WHILE-loop does not necessarily run a specific number of times. Instead, the while-loop is a **conditional** (like the REPEAT) but puts its condition first before executing the looping block. Only if the condition is satisfied, the loop body will execute, that is, the loop body executes while the condition is true and exits the loop when the condition is false.



### New words

**conditional** – to put its condition first before executing the looping back

The WHILE-loop is called a pre-conditional loop since the condition that determines whether it should run is found at the start of the loop. If the condition is true, the loop activates and continues to repeat until the condition is no longer true. If the condition is not met initially, the entire loop is skipped.

### Syntax of WHILE..DO loop

```
WHILE (condition = True) do  
BEGIN  
    // 1 or more instruction  
END // Test Condition again
```

#### Note:

- The loop starts with the keyword WHILE followed by the condition and then the keyword DO
- DO is not followed by a semi-colon
- The body of the loop appears within a BEGIN ... END block
- The loop executes while the condition is true and exits the loop once the condition evaluates to false
- The loop condition is tested at the beginning of the loop. If the loop condition evaluates to false upon entering the loop, then the loop is not executed at all.

Differences between a WHILE-loop and a REPEAT-loop:

WHILE-LOOP	REPEAT LOOP
• Pre-Test loop. The condition is tested at the beginning of the loop	• Post-Test loop. The condition is tested at the end of the loop
• Since the condition is tested at the beginning of the loop, the loop may not be executed at all if the condition is false	• Since the condition is tested at the end of the loop, the loop is executed at least once
• The loop executes while the condition is true	• The loop executes while the condition is false
• The loop terminates when the condition becomes false	• The loop terminates when the condition becomes true
• Works on the ICT principle	• Works on the ICT principle

 FOR AND WHILE LOOPS



[https://www.youtube.com/watch?v=FM1IemZvP\\_Y](https://www.youtube.com/watch?v=FM1IemZvP_Y)

## USING LOOPS WITH COMPONENTS

Loops are often used with components like ListBoxes to show the results from the loop. This is because the ListBox's Items.Add method allows you to create a list of items that can then be shown in the ListBox. ListBoxes are therefore an easy way to both store and display the results of a loop. This also works with components like the TComboBox and the TRadioGroup.

With the use of a simple loop, you can quickly add new items to these components, as shown by the code snippet below.

### Adding items to components

```
for i := 1 to 10 do
begin
  sValue := IntToStr(i);
  TComboBox.Items.Add(sValue);
  TRadioGroup.Items.Add(sValue);
  TListBox.Items.Add(sValue);
end;
```

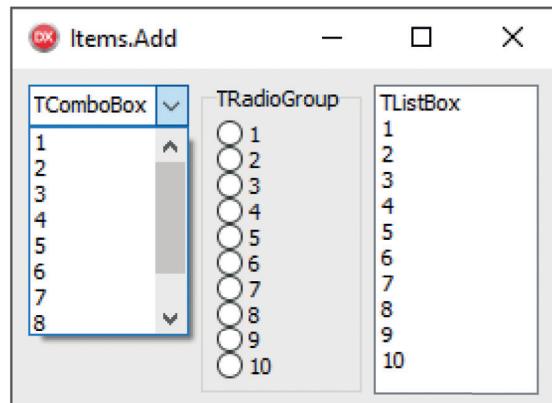


Figure A.2: Different components with the `Items.Add` method

Once an item has been added to one of these components, you can read the value using the following syntax.

#### Reading an item from a particular position in a ListBox

```
sValue := TListBox.Items[0]; // Reads the first value.  
sValue := TListBox.Items[1]; // Reads the second value.  
...  
sValue := TListBox.Items[i - 1]; // Reads the i-th value.
```

Examples of assigning values to a particular position in a ListBox:

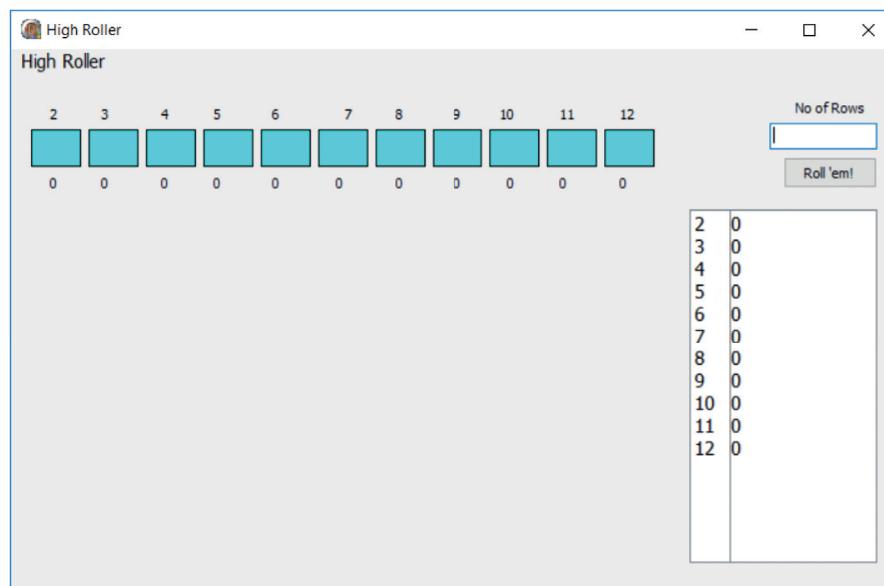
```
lstDisplay.Items[0] := 'Name';  
lstRolls.Items[iTotal-2] := IntToStr(iCurrent);
```



#### Guided activity A.1 High roller

For the high roller application, you will create a bar chart that shows which numbers occur most frequently when you roll two dice. To create this application:

- A.1.1 Create a new folder named High Roller.
- A.1.2 Create a new **HighRoller\_p** project and save it in the folder Annex – High Roller.
- A.1.3 Create the following user interface:



- Use labels to display the possible outcome when the two dices are thrown.
- Set the Shape property of TShape shapes to stRectangle and change their Color property to a colour of your choice.
- Use labels below the TShapes to display how many times a total was generated.
- Place two *ListBoxes* next to each other. The first *ListBox* displays the possible totals when the two dices are rolled. The second *ListBox* displays how many times a particular total was achieved
- Add an *EditBox* to read the number of times the dices must be thrown

- A.1.4 Create an *OnActivate* event to:

Clear the *ListBox* *lstRolls*. The *ListBox* *lstRolls* will keep count of the number of times each total was generated when the two dices were thrown. Initially it sets all counters to zero. Clears the *ListBox*. Set the initial counter value of each total to zero (0). Eleven such counters are set, that is, counters for totals from 2 to 12 are only possible. The two dices cannot roll a total of 0 or 1.



### Guided activity A.1

High roller *continued*

- It also sets the totals (2 to 12) that can be generated in the ListBox lstNum.

```
lstRolls.Items.Clear;
for i := 2 to 12 do
begin
  lstRolls.Items.Add('0');
  lstnum.Items.Add(IntToStr(i));
end;
```

- A.1.5** Create an *OnClick* event for the [Roll'em] button.

- Declare the local variables shown below:

```
Var i, iDie1, iDie2, iTotal, iCurrent, iMaxHeight, iMaxCount:
  Integer;
```

- Since you will be using random numbers for the dice and want Delphi to generate different random numbers each time, you need to use the *Randomize* command
- Read the number of times you want to roll both dices from the EditBox edtNum.
- For 1 to the value read from the edtNum EditBox:  
Randomly generate two numbers in the range 1 to 6.
- Calculate the total for both dices and store it in variable iTotal.
- For the calculate total, read its counter from the ListBox lstRolls and increment by 1 and store the value in iCurrent:

```
iCurrent := StrToInt(lstRolls.Items[iTotal-2]) + 1;
```

- Remember that the *ListBox* displays the first item in position 0. So if iTotal is 4, then the counter counting the number of times 4 occurs will be displayed in position 2.
- Set the value of lstRolls[iTotal-2] to the iCurrent value

```
lstRolls.Items[iTotal-2] := IntToStr(iCurrent);
```

- The code is:

#### Dice roll for-loop

```
for i := 1 to StrToInt(edtNum.Text) do
begin
  iDie1 := Random(6) + 1;
  iDie2 := Random(6) + 1;
  iTotal := iDie1 + iDie2;
  iCurrent := StrToInt(lstRolls.Items[iTotal-2]) + 1;
  lstRolls.Items[iTotal-2] := IntToStr(iCurrent);
end;
```



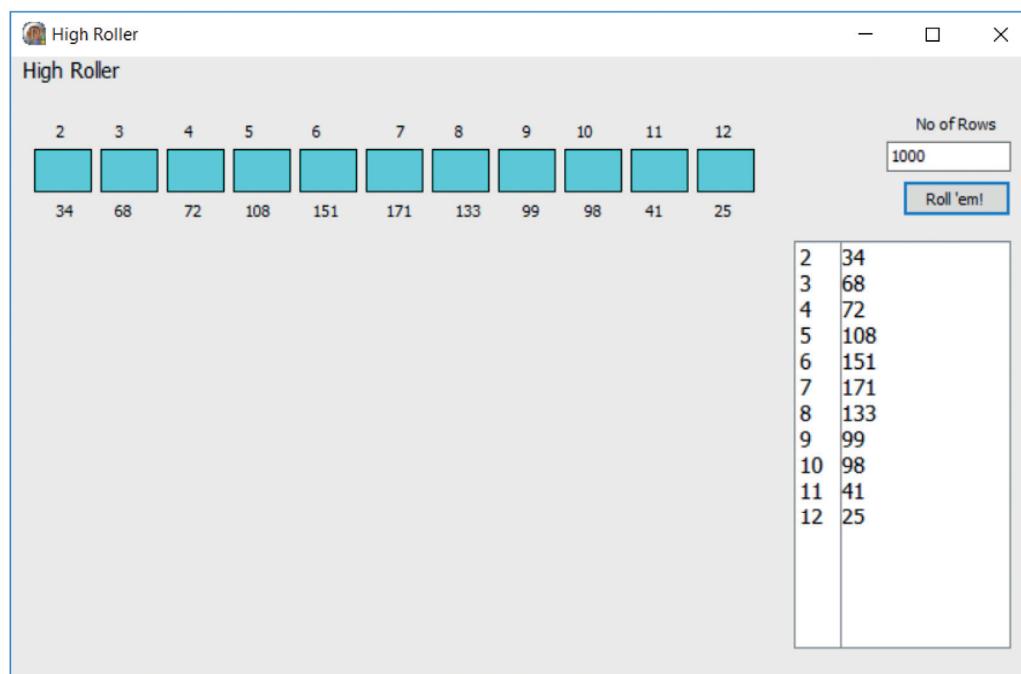
### Guided activity A.1 High roller *continued*

- Set the caption of the 11 labels below the rectangle shapes to equal to the corresponding counter value from the *lstRolls* ListBox.

#### Update labels text

```
lblCount2.caption := lstRolls.Items[0];
lblCount3.caption := lstRolls.Items[1];
lblCount4.caption := lstRolls.Items[2];
lblCount5.caption := lstRolls.Items[3];
lblCount6.caption := lstRolls.Items[4];
lblCount7.caption := lstRolls.Items[5];
lblCount8.caption := lstRolls.Items[6];
lblCount9.caption := lstRolls.Items[7];
lblCount10.caption := lstRolls.Items[8];
lblCount11.caption := lstRolls.Items[9];
lblCount12.caption := lstRolls.Items[10];
```

- A.1.6** Save and test your application. Here is an example of sample output:



- A.1.7** Use a FOR-loop with a value from 2 to 12 to identify which item in the *ListBox* has the largest value and store this value in the *iMaxCount* variable.

#### Finding largest value

```
iMaxCount := 0;
for i := 2 to 12 do //OR for i := 0 to 10 do
begin
if StrToInt(lstRolls.Items[i-2]) >= iMaxCount then
  iMaxCount := StrToInt(lstRolls.Items[i-2]);
end;
lstRolls.Items.Add('Max: '+ IntToStr(iMaxCount));
```

- A.1.8** Set the value of variable *iMaxHeight* to equal to the Height property of the form minus 150.

```
iMaxHeight := frmHighRoller.Height-150;
```



### Guided activity A.1

#### High roller *continued*

- A.1.9 To calculate the height of the first box, use the following code.

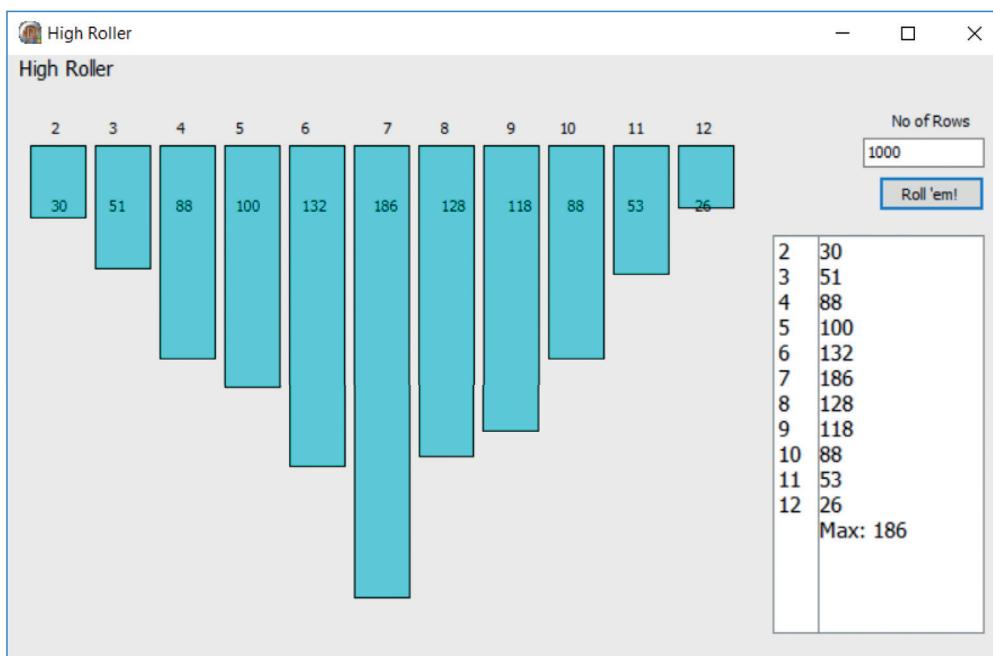
##### Shape height

```
Shp1.Height := Round((StrToInt(lstRolls.Items[0]) / iMaxCount) *  
                      iMaxHeight);
```

- The counter for the first total 2 is stored in position 0 in the lstRolls ListBox. This value is retrieved and converted to integer.
- Calculate the rounded percentage of the value retrieved in bullet one divided by the maximum variable iMaxCount . The percentage is rounded since the height property can only accept an integer value.
- The percentage is then multiplied with the maximum allowed height of the shapes. Each bar's height will therefore be equal to a percentage of the maximum possible height.

- A.1.10 Add the same formula to all the other shapes, making sure to update the index of lstRolls.Items for each shape.

- A.1.11 Save and test your application.



### Activity A.5

- A.5.1 1. Open the **IsolateDigits\_p** project from the Annex – Isolate Digits folder and do the following:

- Create an OnClick event for the [Isolate Digits – Method 1] button to isolate the digits of a number. Use the method to isolate digits as shown in the Guided activity in Unit 1.1 of this chapter. Display the isolated digits in the order in which they appear in the number on separate lines.
- Create an OnClick event for the [Isolate Digits – Method 2] button to isolate the digits of a number using an alternate method to the one in bullet 1. Display the isolated digits in the order in which they appear in the number on separate lines.

- A.5.2 You can generate any Fibonacci sequence given the first and second term. In a Fibonacci sequence, the third term is generated by summing the previous two term:

2 5 7 12 19 ...

Open **FibonacciSequence\_p** project from the Annex – Fibonacci Sequence folder and do the following:

- Read in the first two terms from the EditBoxes
- Create an OnClick event for the [Calculate] button to determine how many terms of the sequence must be added together to give a sum just greater than 100.



### Activity A.5 *continued*

- Display the terms that are used in the summation, the number of terms used and the sum of the terms.
- Save and run your project.

**Fibonacci Sequence**

Enter value for Term1

Enter value for Term2

Fibonacci Sequence  
2 3 5 8 13 21 34 55  
Number of terms: 8  
Sum=141

**Calculate**

## A.6 Strings

Just like there are functions that are useful to perform on numbers, there are also functions that are specifically useful to perform on strings. This section will look at how strings can be combined, compared and manipulated.

### COMBINING STRINGS

The syntax to combine strings is straightforward, you simply add all the strings together:

```
sNew := String1 + String2 + ... + String1000
```

This technique can be used to combine any number of strings. However, doing so without considering the spaces of the combined strings can often result in one long word without any spaces.

### FORMATTING CHARACTERS

One way to ensure that your spacing is correct when combining strings is to make use of the tab and newline characters. These symbols are described in the table below.

**Table A.8:** The Tab and Newline characters

NAME	CODE	DESCRIPTION	EXAMPLE
Tab	#9	Adds a flexible amount of space so that the text following the tab is aligned to a specific position in the component.	'M1: ' + #9 + '71%'; 'Average: ' + #9 + '85%;' // M1: 71% // Average: 85%
Newline	#13	Adds a line break to the string so that the text following the newline character appears on a new line.	'M1: 71%' + #13 + 'M2: 73%'; // M1: 71% // M2: 73%

To add the two formatting characters to a string, you use the plus operator to combine their character values (including the hash symbol) to an existing string. Occasionally, it will be necessary to use more than one tab character to align texts of different lengths. The only way to know if you have added the correct number of tab characters is to test the application and see the result. It should immediately be visible if you have added the incorrect number of tab characters.

Incorrect	Correct
Mark 1: 80%	Mark 1: 80%
Mark 2: 86%	Mark 2: 86%
Average: 83%	Average: 83%

**Figure A.3:** Incorrect and correct number of tab character

## COMPARING STRINGS

Conditional statements can be used to compare two or more strings. For example:

```
if sValue1 = sValue2 then  
    Statement1;
```

**Note:**

- The comparison is not affected by length.
- The comparison is carried out on a letter by letter basis.
- The comparison is case sensitive.

In some applications you need to compare the length of two strings, rather than the values of the strings. This is especially useful when doing string validation. To find the length of a string, you can use the *Length* function, which returns an integer value containing the number of characters:

```
sPassword := 'SuperSecretCode1';  
iLength := Length(sPassword); // 16
```

This integer can then be used in a conditional statement:

```
if iLength >= 8 then  
    Statement1;
```

## SCROLLING THROUGH A STRING

With square bracket syntax, you can read or write a specific character by placing the number of the character in square brackets after the name of the string. The following code snippet shows how this can be done.

### Accessing characters in a string

```
sValue := 'Hello, World!';  
cFirst := sValue[1]; // H  
cSecond := sValue[2]; // e  
sValue[13] := '?'; // sValue = 'Hello, World?'
```



### Did you know

When a value is set for a character in a string, the existing character is replaced by the new character.

Being able to access each of individual characters in a string allows you to manipulate the string in a number of different ways:

- You can create conditional statements based on specific characters.
- You can copy certain characters.
- You can compare specific characters in different strings.
- You can assign new values to individual characters.
- You can delete individual characters.

**MANIPULATING  
STRINGS IN DELPHI**

[https://www.youtube.com/  
watch?v=t9oszMtEdQs](https://www.youtube.com/watch?v=t9oszMtEdQs)

In order to do most of these tasks, you need to combine the square bracket notation with a FOR-loop that allows you to iterate through the string. To scroll through each character in a string, your FOR-loop should run from the first character in the string to the last character (given by the length of the string).

#### Iterating through a string

```
sPhrase := 'I love programming!';
iLength := Length(sPhrase);
for i := 1 to iLength do
  ShowMessage(sPhrase[i]);
```

## MANIPULATING STRINGS

Once you know how to scroll through a string, you can use this technique to manipulate the string. The table below briefly describes different ways in which strings can be manipulated.

**Table A.9:** Ways in which strings can be manipulated

GOAL	ALGORITHM DESCRIPTION
Finding a character	Use a for-loop to scroll through a string sString. Compare each character in the string to the search character. Once the search character is found, return the value of the for-loop's counter as the position of the search character.
Replacing a character	Use the finding a character algorithm to find a specific character. Once the character is found, use the for-loop's counter to set a new value for this character.
Deleting a character at a specific position	Use a for-loop to scroll through a string. Add all the characters before the delete position to sStart string, and all the characters after the delete position to sEnd string. Overwrite the input string by combining sStart and sEnd strings.
Inserting a character	Use a for-loop to scroll through a string sString. Add all the characters before the insert position to sStart string, and all the characters after the insert position (including the insert position) to sEnd string. Overwrite the input string sString by combining the sStart string, the new character, and the sEnd string.

You will learn how to replace each of these algorithms with Delphi functions later this year.



## Activity A.6

**A.6.1** Create algorithms for:

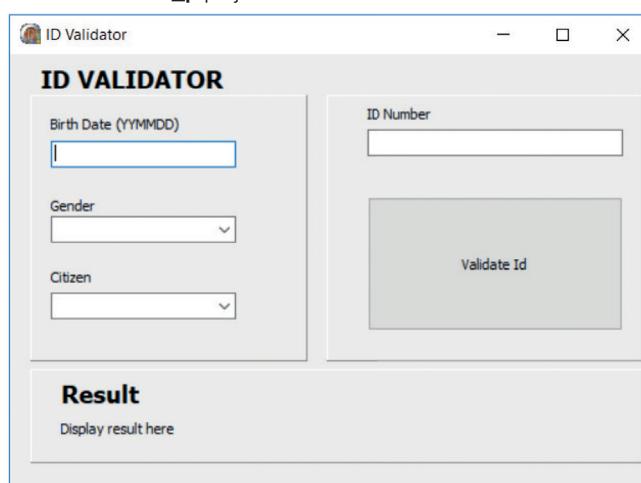
- Finding a character sChar in sLine string.
- Replacing a character in sLine string with a character sReplace.

**A.6.2** Your South African ID number is a unique 13-digit number given to all South African citizens. This number is used as your unique identifier in all government databases and appears on your government identity document, passport and driver's license.

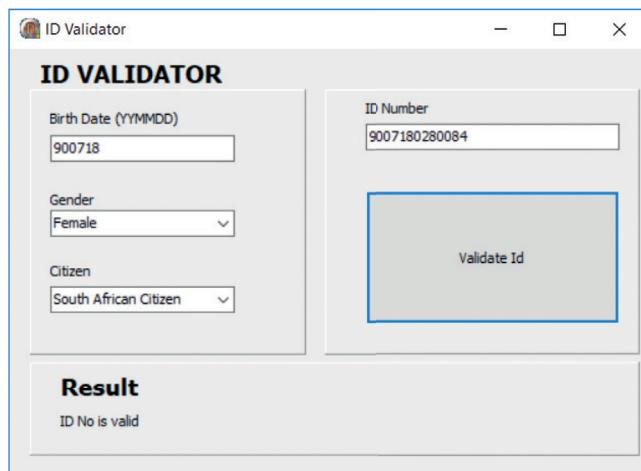
Each ID number uses the following format: YYMMDDSSSSCAZ.

- The first six digits (YYMMDD) are based on your date of birth. For example, if you were born on 23 December 2002, your first six digits would be 021223.
- The next four digits (SSSS) make a unique number between 0000 and 9999 used to distinguish between people born on the same date. Numbers under 5 000 are given to women, while numbers equal to or above 5 000 are given to men.
- The next digit (C) is used to indicate if you are a South African citizen (0) or a permanent resident (1).
- The second last digit (A) was used to indicate a person's race during apartheid, but today is always an 8.
- The final digit (Z) uses the Luhn algorithm to determine if all the previous numbers were entered correctly. We will not look at the Luhn algorithm in this case study.

Open the **IDValidator\_p** project from the Annex – ID Validator folder and do the following:



- Read the Birth date and ID Number for yourself
- Create an OnClick event for the Validate ID button to verify whether your ID number correctly shows your birth date, gender and citizenship. Display an appropriate message.



## B Naming convention of components

COMPONENT NAME	PREFIX	ICON	BRIEF DESCRIPTION
<b>Standard Group</b>			
Button	btn		Most used to activate an action.
Label	lbl		Commonly used to display information.
Edit	edt		Used for single line input, but also displays information.
Memo	mem		* Multiple line display organized in lines.
Panel	pnl		A container hosting other components.
ListBox	lst		Multiple line display. Able to display left aligned columns.
Radio Button	rad		Toggles selection.
Radio Group	rgp		Grouped Radio buttons – only one selectable.
ComboBox	cmb		Multiple line capturing. Selection of item through drop-down.
Check Box	chk		Toggles selection.
Main Menu	mnu		Main menu with submenus. Activates actions.
<b>Additional Group</b>			
BitButton	btt		Button with icon - used to activate actions.
String Grid	sgd		Two dimensional grid with cells to capture text.
Image	img		Component to host pictures (bitmaps, jpgs).
Shape	shp		* Basic shape like circle, rectangle or ellipse.
<b>Win32</b>			
RichEdit	red		Memo with RTF capabilities.
Page Control	pgc		Special page component hosting tab sheets.
Progress Bar	prb		* Rectangle capable indicating progress via growing colour bar.
Status Bar	stb		* A sub dividable bar at the bottom of form indicating status.
<b>System Group</b>			
Timer	tmr		* Count down timer – initializing action as count-down reaches 0.
<b>Samples Group</b>			
Spin Edit	sed		Integer input component with pre-set range to select from.
Calendar	cld		* Calendar with Month layout for selecting days.

COMPONENT NAME	PREFIX	ICON	BRIEF DESCRIPTION
<b>Data Access Group</b>			
DataSource	dsr		Component to connect data-aware component with dataset.
<b>dbGo Group</b>			
ADOConnection	con		Component to connect with the database (Access).
ADOTable	tbl		Dataset component to reflect the contents of a single table.
ADOQuery	qry		Dataset component to reflect a result.
<b>Data Controls Group</b>			
DBGrid	dbg		Data-aware component reflecting the contents of a dataset.
DBNavigator	dbn		* Data-aware component interacting with a dataset.
DBText	dt		* Data-aware EditBox reflecting a field value from a record.
Form	frm		The initial Form – not categorised under any group.

## PROGRAMMING CHARACTERS

DECIMAL NUMBER	CHARACTER	NAME
0	NUL	Null
1	SOH	Start of Heading
2	STX	Start of Text
3	ETX	End of Text
4	EOT	End of Transmission
5	ENQ	Enquiry
6	ACK	Acknowledgement
7	BEL	Bell
8	BS	Backspace
9	HT	Horizontal Tab
10	LF	Line Feed
11	VT	Vertical Tab
12	FF	Form Feed
13	CR	Carriage Return
14	SO	Shift Out
15	SI	Shift In
16	DLE	Data Link Escape
17	DC1	Device Control 1
18	DC2	Device Control 2
19	DC3	Device Control 3
20	DC4	Device Control 4
21	NAK	Negative Acknowledgement
22	SYN	Synchronous Idle
23	ETB	End of Transmission Block
24	CAN	Cancel
25	EM	End of Medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File Separator
29	GS	Group Separator
30	RS	Record Separator
31	US	Unit Separator

The next 95 characters are all visible characters that you can see on the screen.

## VISIBLE CHARACTERS

Decimal	Character	Decimal	Character	Decimal	Character
32	SPACE	38	&	45	-
33	!	39	'	46	.
34	"	40	(	47	/
35	#	41	)	48	0
36	\$	42	*	49	1
37	%	43	+	50	2
51	3	74	J	97	a
52	4	75	K	98	b
53	5	76	L	99	c
54	6	77	M	100	d
55	7	78	N	101	e
56	8	79	O	102	f
57	9	80	P	103	g
58	:	81	Q	104	h
59	;	82	R	105	i
60	<	83	S	106	j
61	=	84	T	107	k
62	>	85	U	108	l
63	?	86	V	109	m
64	@	87	W	110	n
65	A	88	X	111	o
66	B	89	Y	112	p
67	C	90	Z	113	q
68	D	91	[	114	r
69	E	92	\	115	s
70	F	93	]	116	t
71	G	94	^	117	u
72	H	95	_	118	v
73	I	96	@	119	w
120	x				
121	y				

Decimal	Character	Decimal	Character	Decimal	Character
122	z				
123	{				
124					
125					
126	~				

The 127th character is the *DELETE* character, which is used when something needs to be removed or deleted.

To write the word ‘English’, a computer would thus receive the following values:

E	n	g	l	i	s	h
69	110	103	108	105	115	104

However, the computer would receive these characters as bytes. Can you calculate the 8-digit binary numbers for each of these characters?

# Glossary

## A

**append** to open an existing file for writing, set the file pointer to the end of the file and allows you to add data to the file

**append** to add an empty row to the end of your database table

**array** is a data structure that store a set values (elements) of the same type linked to a single variable name

**assume** supposed to be the case, without proof

## B

**binary search** is an algorithm used in computer science to locate a specified value (key) within an array

**bubble sort** to compare adjacent elements

## C

**Caesar cipher** a substitution cipher on which each letter in plaintext is ‘shifted’ a certain number of places down the alphabet

**Ceil** to round a real number up to the highest integer value

**CHR** to return the corresponding character of an ASCII code

**circular dependency** to cause an application to crash

**CompareText** to compare two strings for equality, ignoring case

**concatenates** to joins strings together into one result string

**conditional** to put its condition first before executing the looping back

## D

**data module** a sealed, removable storage module containing magnetic disks and their associated access arms and read/write heads

**DEC** to decrement an ordinal type variable

**decremental** the act or process of decreasing or becoming gradually less

**Delete** to delete a number of characters from a string starting from a start position

**delimiters** to show the start and ends of individual pieces of data

**dynamic instantiation** when a component or object is created during run-time

## E

**encrypted message** to encode information to prevent anyone other than its intended recipient from viewing it

**end of file <eof>** to indicate the end of a file when the file is saved

**end of line <eoln>** to indicate the end of the line when the [Enter] button is pressed

**Entity Relationship Diagram** to show the relationships of entity sets stored in a database

**event** an occurrence of something

**exception** is generally an error condition or event that interrupts the flow of your program

**Exception Handling** a way to prevent a program from crashing when a file does not exist

## F

**FileExists** to determine whether a file exists or not

**first argument** is a string that holds instructions for formatting

**Floor** to round a real number down to the lowest integer value

**formal parameter** to declare variable(s) next to the procedure name

**Frac** to return the decimal part of a real number

**function** the operation of something in a particular way

## G

**global** is a programming language construct, a variable that is declared outside and function and is accessible to all the functions throughout the program

## H

**homogenous** elements of the same type

## I

**INC** to increment the ordinal type variable passed to it

**incremental** relating to or denoting an increase or addition

**independent** to run on its own

**index** the position of the element in an array

**inner loop** the inner part of a nested loop

**Insert** to insert one string into another string

**insert** to add an empty row at the current position of your database table

**instance** an example or single occurrence of something

**instantiate** represent as or by an instant  
**linear search** is a process that checks every element in the list sequentially until the desired element is found

## L

**local variable** variables that have a local scope  
**logical file** is a variable (in RAM) that points to the physical file on your storage medium  
**LowerCase** to converts uppercase characters in a string to lowercase  
**Luhn algorithm** is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers, IMEI numbers, and Canadian Social Insurance Numbers

## M

**method overloading** to have more than one method with the same name  
**method overloading** to have more than one method with the same name  
**method signature** is the number of arguments and their data type  
**method signature** to name a method and its formal parameters list  
**methods** predefined instructions

## N

**naming convention** to name things (generally agreed scheme)  
**non-local** is a variable that is not defined within the local scope  
**null** to represent an empty value

## O

**ORD** to return the ordinal value of a character  
**outer loop** the outer part of a nested loop

## P

**physical file** to name an external file name found on a storage device and contains the actual data  
**Pi** is a predefined constant that returns a real number giving a useful approximation of the value Pi  
**Pos** to return to the start position of one string within another string as an integer  
**post command** to permanently save the values to the database table  
**POWER** to raise a base to a power and returns a real answer

**procedure** an official way of doing something  
**properties** the components or building blocks

## R

**Random** to generate a random number from 0 to less than 1  
**RandomRange** to generate a random integer number from Num1 to one less than Num2  
**related information** information belonging in the same group  
**Round** to round a real number to an integer value

## S

**second argument** holds the values that needs to be converted into a formatted string  
**selection sort** to select the element that should go in each array position either in ascending or descending order sequence  
**SETLENGTH** to change the size of a string  
**sorted** to sort an element in numerical order  
**SQRT** to return the square root of a number  
**step through** to step through means that you are working through a program line by line  
**STR** to convert an integer or real number into a string, with optional basic formatting

## T

**Trunc** to remove or chop off the decimal part of the real number. It returns an integer after the truncation

## U

**unambiguous** not open to more than one interpretation  
**Upcase** to convert a single letter character to uppercase  
**UpperCase** to converts lowercase characters in a string to uppercase  
**user-defined** is methods written by programmers themselves  
**VAL** to convert a string to a numeric value  
**validate** to try and lessen the number of errors during the process of data input in programming  
**value parameter** when a procedure is called, memory locations are created for each of the formal parameters and the values of the arguments are assigned to the corresponding formal parameters. Changes made to a value parameter will not affect its corresponding argument. When the procedure is exited, the memory locations of the formal parameters 'die' away

# QR Code list

You can use the QR codes on these pages to link to online content for further information on these topics.

## Dear Learner

WHAT MOST SCHOOLS DON'T TEACH.....	iv
------------------------------------	----

## Chapter 1

WHAT'S AN ALGORITHM?.....	2
---------------------------	---

## Chapter 2

EXPLAINING BINARY NUMBERS.....	26
CONVERT DECIMAL TO HEXADECIMAL.....	30

## Chapter 3

DELPHI ARRAYS.....	42
--------------------	----

## Chapter 8

CONNECTING TO AN ACCESS DATABASE .....	176
WHAT'S AN ALGORITHM?.....	201
EXPLORING DELPHI COMPONENTS.....	206
LEARNING ABOUT VARIABLES .....	209
CONVERTING REAL NUMBERS TO STRINGS.....	210
MATH OPERATIONS.....	212
GLOBAL AND LOCAL VARIABLES .....	216
FOR LOOPING .....	225
FOR AND WHILE LOOPS.....	228
MANIPULATING STRINGS IN DELPHI.....	236

## ANNEXURES

### Notes

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---

---