Practical Book

INFORMATION TECHNOLOGY

12

Contents

Term 1

Term 2

Term 3

Dear Learner

Welcome to the *IT Practical Grade 12* 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.

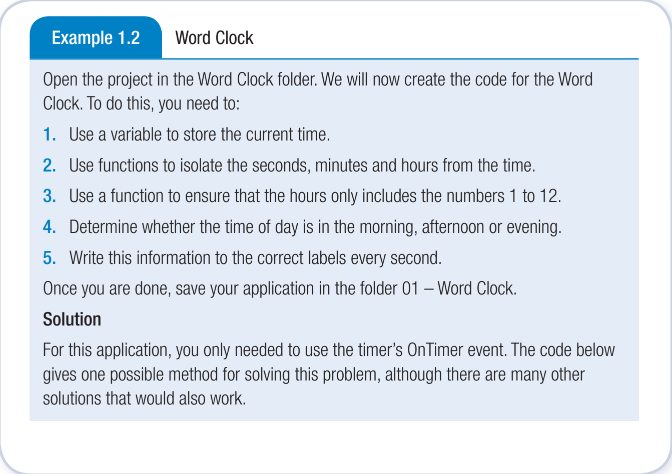
Programming is not only about knowing and using the programming language. There are also important theoretical concepts that you will need to understand, and planning and problem-solving tools that you will need to master. The best-coded program in the world will not be useful if it solves the wrong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

- Chapter 1: Programming fundamentals
- Chapter 2: Object-oriented programming
- Chapter 3: Two-dimensional arrays
- Chapter 4: Databases and SQL

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 1.2 Word Clock

Open the project in the Word Clock folder. We will now create the code for the Word Clock. To do this, you need to:

1. Use a variable to store the current time.
2. Use functions to isolate the seconds, minutes and hours from the time.
3. Use a function to ensure that the hours only includes the numbers 1 to 12.
4. Determine whether the time of day is in the morning, afternoon or evening.
5. Write this information to the correct labels every second.

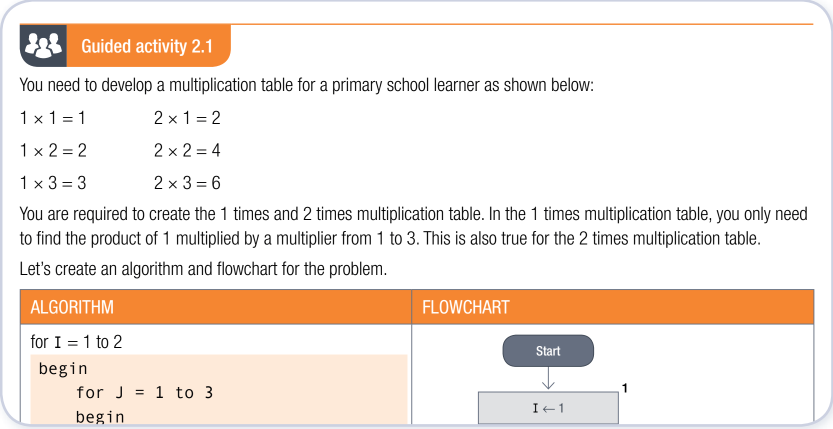
Once you are done, save your application in the folder 01 – Word Clock.

Solution

For this application, you only needed to use the timer's OnTimer event. The code below gives one possible method for solving this problem, although there are many other solutions that would also work.

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:

1 × 1 = 1	2 × 1 = 2
1 × 2 = 2	2 × 2 = 4
1 × 3 = 3	2 × 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.

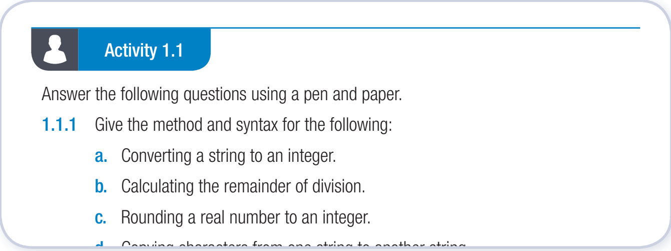
Let's create an algorithm and flowchart for the problem.

ALGORITHM	FLOWCHART

for I = 1 to 2
begin
for J = 1 to 3
begin
	Start
	I ← 1
	1

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

Answer the following questions using a pen and paper.

1.1.1 Give the method and syntax for the following:

a. Converting a string to an integer.
b. Calculating the remainder of division.
c. Rounding a real number to an integer.
d. Copying characters from one string to another string

‘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

When creating mock data for a database, it is useful to use a site such as Mockaroo Random data generator. https://mockaroo.com/

Did you know

Programming is a way of thinking – not a rote skill. Learning about ‘For’ loops is not learning to program, any more than learning about pencils is learning how to draw.

New words

Double – The Double data type can store 15 - 16 digits in the fraction part of a decimal number and thus support greater precision.

Debugging – The process of detecting and removing existing and potential errors (also called ‘bugs’) in software code that can cause it to behave unexpectedly or crash.

New words

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

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.

4 TIPS TO SOLVE PROGRAMMING PROBLEMS

https://www.youtube.com/watch?v=IsLeirHcoJQ

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. You teacher may also use these to assess your performance during class.

CONSOLIDATION ACTIVITY Chapter 1: Programming fundamentals

QUESTION 1

Komani Game Reserve in the Eastern Cape offers accommodation and the chance to see three of the ‘Big Five’ animals. Do the following:

- Open the incomplete program in the 01 – Question1 folder.
- Compile and execute the program. The program has no functionality currently.
- Follow the instructions below to complete the code for QUESTION 1.1 to QUESTION 1.5.

1.1 A picture file called elephant.png has been included in the root folder of the **Question1_p** Delphi project.

TERM 1

PROGRAMMING FUNDAMENTALS

CHAPTER 1

CHAPTER UNITS

Unit 1.1	Problem solving
Unit 1.2	Procedures and functions in the Delphi run–time libraries
Unit 1.3	Procedures and functions
Unit 1.4	User interface design
Unit 1.5	Databases

Learning outcomes

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

- use algorithms, flowcharts and pseudocode to plan applications
- debug applications using a variety of different techniques
- create applications using the Delphi IDE
- list and describe the most commonly used properties
- create events for Delphi applications
- use number variables and functions in applications
- use string variables and functions in applications
- use arrays and files in applications
- use different looping structures in applications
- use conditional structures in applications
- create user-defined methods for applications
- describe and implement the principles of user interface design
- dynamically create Delphi components in applications
- create and read text files in Delphi applications
- create a connection to a database using Delphi components
- use data from a database in applications.

INTRODUCTION

In this chapter, you will briefly look at all the concepts learned in Grade 10 and Grade 11. To ensure this information is fresh in your mind, this chapter contains five new programs for you to create. These include a statistical simulator, a fake virus application, and a word clock. Work carefully to complete these programs and the units in this chapter.

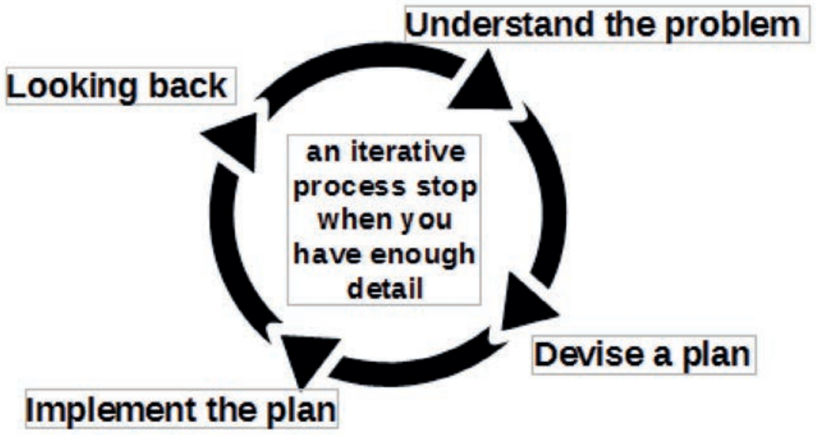
The remaining chapters of this book will rely on the fact that you are competent and confident with your knowledge about the concepts that we revise in Chapter 1.

UNIT 1.1 Problem solving

SOFTWARE ENGINEERING: A GUIDE FOR YOUR PAT

Problem solving refers to the process through which a solution is found to a complex problem. Since programmers spend most of their time solving problems, they need to have a method to systematically tackle new problems.

The problem-solving method consists of four steps. These are illustrated in the diagram below and explained briefly:



Some basic strategies for getting to grips with a problem:
[Source: Adapted from How to solve it, G. Polya, 1973, Princeton U.P.]

FIRST	UNDERSTAND THE PROBLEM
You have to understand the problem.	• Do research to understand the details of the problem. • Do you understand the scope of the problem? • What are the input data? • What tasks must be performed? • What is the required output?
SECOND	DEVISING A PLAN
Find the connection between the data and the unknown. You may need to consider alternate problems if an immediate connection cannot be found. You should eventually come up with a plan for the solution.	• Have you written a related or similar program before? • Have you seen the same problem in a slightly different form? • Could you restate the problem? • Could you introduce some alternate method? • Focus on IPO – it will help you isolate the tasks that must be done. • Could you solve a part of the problem? • Could you derive something useful from the data? • Could you change the data so that the required output and the data are closer to each other? • Did you use all the data? • Create detailed notes of your progress.
THIRD	IMPLEMENT THE PLAN
Carry out your plan.	• When carrying out your plan of the solution, check each step. • Can you show that each line of code is correct? • Test each task as you complete it.

Did you know

Programming is a way of thinking – not a rote skill. Learning about 'For' loops is not learning to program, any more than learning about pencils is learning how to draw.

4 TIPS TO SOLVE PROGRAMMING PROBLEMS

https://www.youtube.com/watch?v=IsLeirHcoJQ

FOURTH	LOOKING BACK
Examine the solution obtained.	• Can you check the result? • Can you derive the result differently? • Can you use the result, or the method, for some other program? • Solve the problem.

THE PROBLEM-SOLVING METHOD

One of the best problem-solving strategies is: do something.

If, at any time, you get stuck on trying to figure out a solution to a problem, brainstorm your solution using a pencil and paper before trying to write the code for the program. Remember that the most important part of planning is creating a detailed implementation plan that describes how you will solve the problem. To create this plan, you need to make a list (or checklist) of every task that needs to be completed.

Take note

First solve the problem, then write the code.
– John Johnson

In Grade 10 we looked at the user story. This tool is well-suited to help you as you create your detailed implementation plan.

Example 1.1 Creating a detailed implementation plan

TEMPLATE: USER STORIES

Key	
aUser	Specific type of user
member	Member of the fitness club
admin	The System Administrator

User Stories			Confirmations	
As a...	I want to be able to... (What)	So that I can.. (Why)	Success...	Failure...
member	Use the Login form Login to the system with user-name and password	Signup or view progress	Members navigator screen appears	No such login
member	see a list of training programs	view data about training program	Training program information is displayed on program screen upon selection of program type	Incorrect selection, database not connected
member	select a training program	Enter/select information on signup screen	Only available programs active for selection. Those with trainer and not yet full. Popup message to confirm that signup data is saved	Program full. Already running. Program not available

Once we have used our user story to create an implementation plan to solve our problem, we have a good idea about which tasks are needed to solve our problem.

Next we can start looking at screen layout for the tasks – this is driven by the user stories.

Example 1.1 Creating a detailed implementation plan *continued*

Here is an activity diagram, which provides an excellent way of capturing the broad picture.

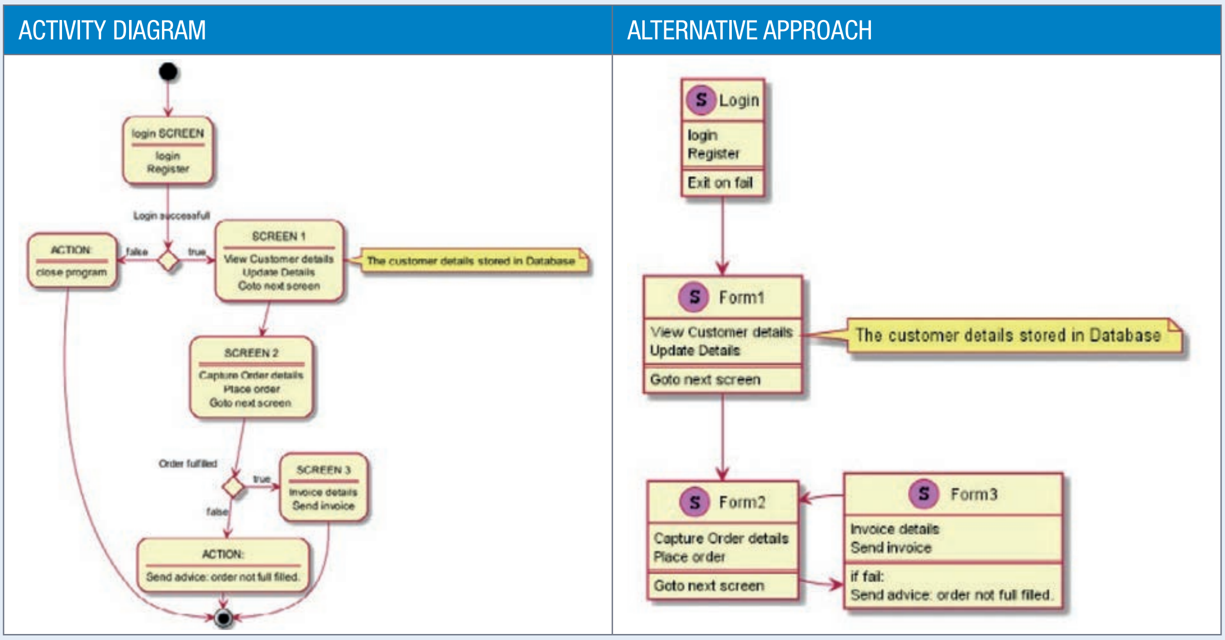


Figure 1.1: *Activity diagram – Overview of system (UML)*

Once we have completed our activity diagram, we can now consider building a prototype in Delphi, building screens and linking them. However, there will be no functionality yet.

We call this evolutionary prototyping and it demonstrates the logical program flow and navigation between screens. This helps you ensure that the layouts are functional and satisfy the usability requirements.

Take note

In Unit 1.4 of this chapter, we will discuss useful design tips.

Also see Shneiderman's Eight Golden Rules of Interface Design in Annexure A.

WHAT'S UML AND WHY DO YOU NEED IT?

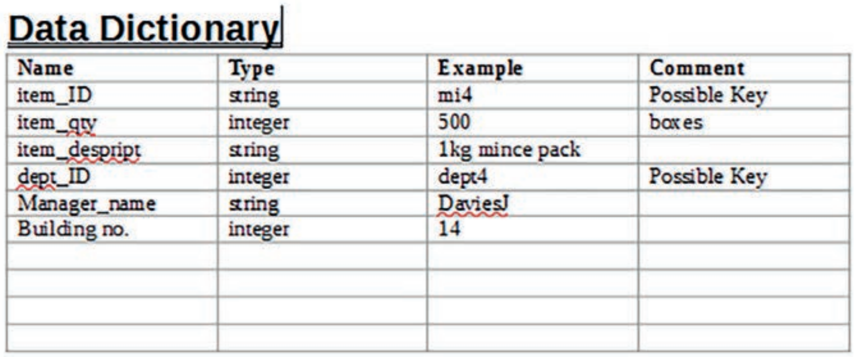
https://www.youtube.com/watch?v=8CBnAmYnwk0

WHAT ABOUT THE DATA?

Because we have an implementation plan as well as a basic Delphi prototype to help us solve our problem in place, we can begin looking at the data we need to complete the tasks we have identified. By using the 'user stories' you can get a good idea of what data you will need.

In Grade 10 you also looked at a tool called the 'noun-verb analysis'. This tool is useful for extracting this information from the user stories. Look at the the second column of Figure 1.1 above. Work through the following steps:

- Work through the 'user stories' template and collect all of the nouns (YELLOW). In this way you can determine what objects or entities you need to manipulate. The verbs (GREEN) will give you more details of how this should be done.
- List all the details of the entities. You can unpack them using the information you researched earlier.
- Create a 'data dictionary' in which you can add the name of the data fields you have identified and their types. An example is shown below:



Data Dictionary

Name	Type	Example	Comment
item_ID	string	mi4	Possible Key
item_qty	integer	500	boxes
item_despript	string	1kg mince pack	
dept_ID	integer	dept4	Possible Key
Manager_name	string	DaviesJ	
Building no.	integer	14	

Now you are ready to set up a database if your problem requires it.

- Create the UML diagram to illustrate your database and classes. An example is shown below:

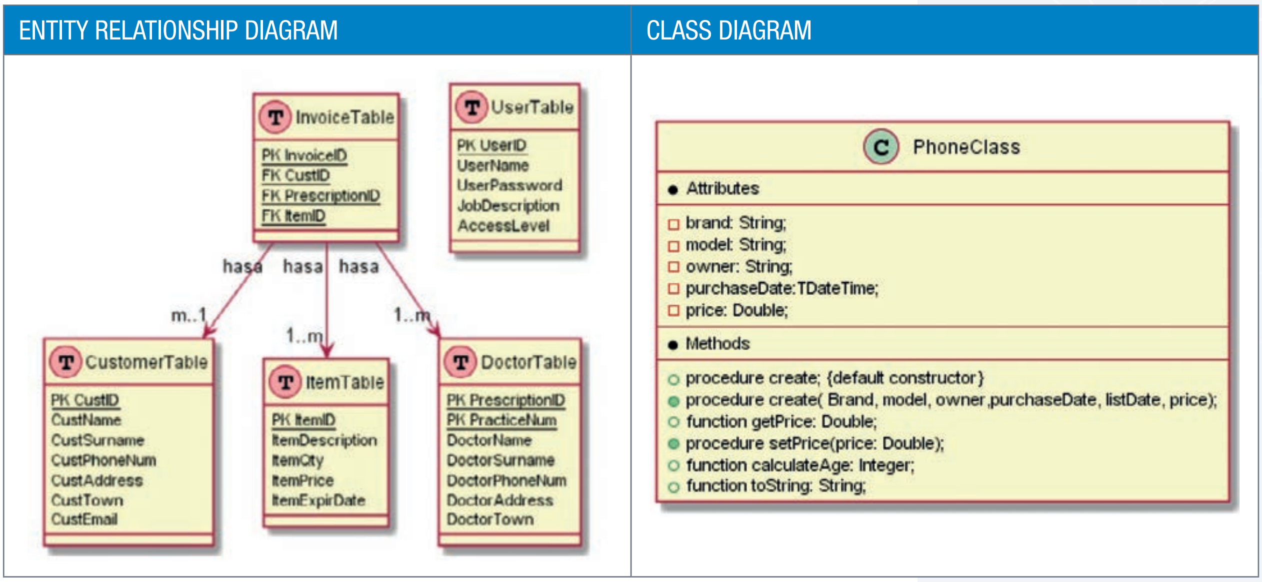


Figure 1.2: *UML diagram to illustrate your database and classes*

- Think about how you will complete each of the tasks identified in the user stories. Start by ordering the user stories by importance: ‘essential’, ‘important’ and ‘nice to have’. Consider how things will interact with each other.
- Test each small sub-program as you go. To do this, you can use the **debugging** techniques such as trace-tables, variable watches, error catching and trace printing to find errors.
- Make good use of flowcharts or Pseudocode to plan the implementation of algorithms you require. Remember the principles of IPO as you continue with the designs of the algorithms. This will ensure that you end up with a robust solution.
- When you get stuck with the implementation of a particular sub-task, create a small, standalone program to sort out the problem, and then move it back into the main program. Because of the detailed implementation plan, you can systematically complete all the small, individual tasks listed on your plan.
- Check to see if the plan solved the problem. To do this, test that each sub-task meets its goal.

Did you know

Nicholas Wirth used this title on his famous Pascal book: **Algorithms + Data Structure = Program**

New words

Double – The Double data type can store 15 - 16 digits in the fraction part of a decimal number and thus support greater precision.

Debugging – The process of detecting and removing existing and potential errors (also called ‘bugs’) in software code that can cause it to behave unexpectedly or crash.

Take note

Your program might:

- fail to solve the problem
- only partially solve the problem
- create new problems.

If your evaluation reveals any remaining problems, you can repeat the problem-solving process, focusing on these new problems.

- Test the program for any calculation errors or logical problems. To do this, you can use techniques like trace tables, variable watches, error catching. Make use of these debugging tools and/or trace printing to find errors.
- Output samples to ensure the program works as expected.
- Look back to your initial problem statement and the ‘user stories’ and compare your output to the requirements. This final check ensures that you have not only met the requirements listed on your plan, but also the original requirements from the user stories. Remember to include extreme values, for example boundary values, minimum values, and so on, in your test data to ensure the program works for any value.

UNIT

1.2 Procedures and functions in the Delphi runtime libraries

In this unit you will learn about some of the procedures and functions in the Delphi run–time libraries. These will help you as you create different programs to solve problems that you may need to solve as a programmer.

DATE AND TIME

The DateTime variable is a container for a variable of type double. When you look at the variable it shows a date and time (for example, 1 January 2019 06:00:00), but the actual value stored in the variable is a number with a decimal (for example, 43418.25). When working with this value, the whole number represents the full days since 30 December 1899, while the decimal value represents a fraction of a day (that is, the time).

To create a DateTime variable, you can use one of the techniques shown in the table below:

FUNCTION	DESCRIPTION	EXAMPLE
`tBirthDate := dNumber;`	If you know the exact number of days since 30 December 1899, you can assign it directly to the DateTime variable.	`tBirthDate := 43646.261;`
`tDate := StrToDateTime('dd/mm/yyyy hh:mm:ss');`	The StrToDateTime function can convert a date string into a DateTime. In the date string, the values are given in the following order: day/month/year hour:minute:second	`tDate := StrToDateTime('30/06/2019 06:15:27');`
`tDate := EncodeDateTime(yyyy, mo, d, h, mi, s);`	The EncodeDateTime function accepts integer values for the year, month, day, hour, minute and seconds of the DateTime.	`tDate := EncodeDateTime(2019, 06, 30, 06, 15, 27);`

There are also four named functions you can use to access specific times. They are:

- Yesterday: Returns the DateTime of 00:00:00, the previous day.
- Date: Returns the DateTime of 00:00:00, today.
- Tomorrow: Returns the DateTime of 00:00:00, tomorrow.
- Now: Returns the current date and time.

To do calculations with a DateTime variable, you add (or subtract) the number or fractions of days to the variable. If you want to set the DateTime's value equal to now, seven days ago, you can subtract seven days from the current DateTime as follows:

Seven days ago

```
tLastWeek := Now – 7;
```

Did you know

It is usually easier to remember the calculation used to obtain the decimal unit than the exact decimal unit.

To add or subtract times, you need to add or subtract the appropriate decimal value. The table below shows the decimal value per unit for hours, minutes and seconds.

UNIT OF MEASUREMENT	UNITS PER DAY	DECIMAL PER UNIT
Hour	24 hours per day	1 / 24 = 0.041667
Minute	24 * 60 = 1 440 minutes per day	1 / 24 / 60 = 0.00069
Second	24 * 60 * 60 = 86 400 second per day.	1 / 24 / 60 / 60 = 0.0000116

FUNCTIONS 1 - DELPHI

https://www.youtube.com/watch?v=biFMeKdzMjE

One of the biggest advantages of the DateTime variable is the functions related to it. The table below shows some of the most useful DateTime functions:

FUNCTION	DESCRIPTION
`iDay := DayOfTheMonth(tDate);`	Returns an integer value between 1 and 31 for the day of the month.
`iDay := DayOfTheWeek(tDate);`	Returns an integer value between 1 and 7 for the day of the week.
`iYear := YearOf(tDate)`	Returns an integer value containing the year.
`iMonth := MonthOf(tDate)`	Returns an integer value between 1 and 12 containing the month.
`iDays := DaysBetween(tDate1, tDate2);`	Returns an integer value of the number of days between two dates.
`iDays := DaysInAMonth(tDate);`	Returns the number of days in a given date's month (between 28 and 31).
`iDays := DaysInAYear(tDate);`	Returns the days in a given date's year (either 365 or 366).
`bLeapYear := IsLeapYear(iYear);`	Returns a True value if the specified year is a leap year, and a False value otherwise.
`FormatSettings. LongMonthNames[index]`	A built-in array containing the month names (i.e. LongMonthNames[1] = 'January').

PROCEDURES - DELPHI

https://www.youtube.com/watch?v=9Myahw6q68I

FUNCTION	DESCRIPTION

`FormatSettings.`
`LongDayNames[index]`
	A built-in array containing the day names (i.e. LongDayNames[1] = 'Monday').

sOutput := FormatDateTime(sFormat, tDate);	Returns a formatted string of the DateTime, based on the "sFormat" input. You can create a formatting string by combining the following letters that represent units of time: • Y = year • M = month • D = day • H = hour • N = minute • S = second • Z = millisecond For example, the format string 'DD-MM-YYYY' will return a value like '25-12-2030'.

To use these methods, you need to add the DateUtils library to your project. This is done by including the text **DATEUTILS** in the **USES** section of your application. Once added, you can use any of the functions inside the DateUtils library.

Example 1.2 Word Clock

Open the project in the Word Clock folder. We will now create the code for the Word Clock. To do this, you need to:

1. Use a variable to store the current time.
2. Use functions to isolate the seconds, minutes and hours from the time.
3. Use a function to ensure that the hours only includes the numbers 1 to 12.
4. Determine whether the time of day is in the morning, afternoon or evening.
5. Write this information to the correct labels every second.

Once you are done, save your application in the folder 01 – Word Clock.

Solution

For this application, you only needed to use the timer's OnTimer event. The code below gives one possible method for solving this problem, although there are many other solutions that would also work.

Example 1.2 Word Clock *continued*

OnTimer event

```
procedure TfrmWordClock.tmrTimerTimer(Sender: TObject);
var
  tTime : TDateTime;
  sTimeOfDay : String;
  iHours : Integer;
begin
  lblSeconds.Caption := FormatDateTime('s', Now) + ' seconds and';
  lblMinutes.Caption := FormatDateTime('n', Now) + ' minutes past';
  tTime := Now - Date;

  if tTime < 0.5 then
    sTimeOfDay := ' in the morning';
  if (tTime >= 0.5) and (tTime < 0.75) then
    sTimeOfDay := ' in the afternoon';
  if tTime >= 0.75 then
    sTimeOfDay := ' in the evening';

  iHours := StrToInt(FormatDateTime('h', tTime)) mod 12;
  if iHours = 0 then
    iHours := 12;

  lblHours.Caption := IntToStr(iHours) + sTimeOfDay;
end;
```

Once you are done, run and save your application.

In this code, the first two lines make use of the FormatDateTime function to display the seconds or minutes of the current time (**NOW**). Since the timer updates every second, the time calculated as **NOW** will increase every second, resulting in the labels showing the newest time every second.

The tricky part of this code is to update the third line, which shows whether the hours are in the morning or evening, as well as showing the hours between 1 and 12 (and not 0 to 24). To do this:

- Start by subtracting the current date (date function) from the current date and time (now function).
- This will return only the time which will be a fraction between 0 and 1 (with 0 representing 00:00 AM, 0.5 representing 12:00 PM and 1 representing 00:00 AM the next day).
- With this decimal value, you can use IF-THEN statements to determine whether it is currently the morning, afternoon or evening.
- Make sure the hours are always between 1 and 12. To do this, you start by isolating the hours from the time. While this can be done in a number of ways, the code above uses the FormatDateTime function to display a string that only contains the hours. This string is then converted into an integer.

- By using the mod 12 mathematical operator, you calculate the remainder of the hours divided by 12. This will always return the correct hour value under 12. However, when it is 12 o'clock in the morning or night, the remainder will be 0, which is incorrect. To fix this, an IF-THEN statement is used to change the value from 0 to 12.
- The hours are converted back to a string and assigned, together with the time of the day, to the label's caption.

ARRAYS AND FILES

ARRAYS

Arrays are variables that can be used to store multiple related variables of the same type (called elements). As long as there is space left in the array, you can continue writing new information to it. Each array has an array index, which is an integer that indicates the number of the element you want to access in the array. An array can be created using the following syntax:

```
aName : Array[FirstIndex..LastIndex] of Type;
```

FOR-loops are often used to access each of the items in your array. For example, if the value of each banking transaction is stored in an array variable, you can find the total of all transactions as follows:

For-loops and arrays

```
for i := 0 to Length(aTransaction) do
begin
  dTotal := rTotal + aTransaction[i];
end;
```

CREATING AN ARRAY

https://www.youtube.com/watch?v=kW7s90JvZLM

FILES

To store data used or created by your program permanently, you need to store data in a file on your computer. When working with files in an application, you need to follow five steps. These are:

- declare the TextFile variable
- assign a text file name to the variable
- indicate how the file will be used (rewrite, reset or append)
- use the file in the application
- close the file

The code snippet below, which writes the phrase “Hello, World!” to a file, shows each of these steps.

File syntax

```
var
  fOutputFile : TextFile;   // Declare the variable

begin
  AssignFile(fOutputFile, 'output.txt');   // Assign the variable
  Rewrite(fOutputFile);   // Indicate how the file will be used
  WriteLn(fOutputFile, 'Hello, World!');   // Use the file
  CloseFile(fOutputFile);   // Close the file
end;
```

The following functions can be used to write to or read from a TextFile.

FUNCTION	DESCRIPTION
ReadLn(fName, sOutput);	Returns the first line of your text file as a string saved in sOutput variable, before moving the cursor to the next line.
bDone := Eof(fName);	Returns a Boolean value that indicates whether the end of file (EOF) has been reached.
WriteLn(fName, sInput);	Writes a string to the TextFile before moving to the next line.
Write(fName, sInput);	Writes a string to a TextFile without moving to the next line.
Eof(fName)	Returns a Boolean value indicating whether the end of file has been reached.

To read all the data from a TextFile, the two functions from the table are usually combined with a WHILE-DO-loop, as shown in the code snippet below.

While loop to read files

```
while not Eof(fName) do
  ReadLn(fName, sOutput);
```

Activity 1.1

Answer the following questions using a pen and paper.

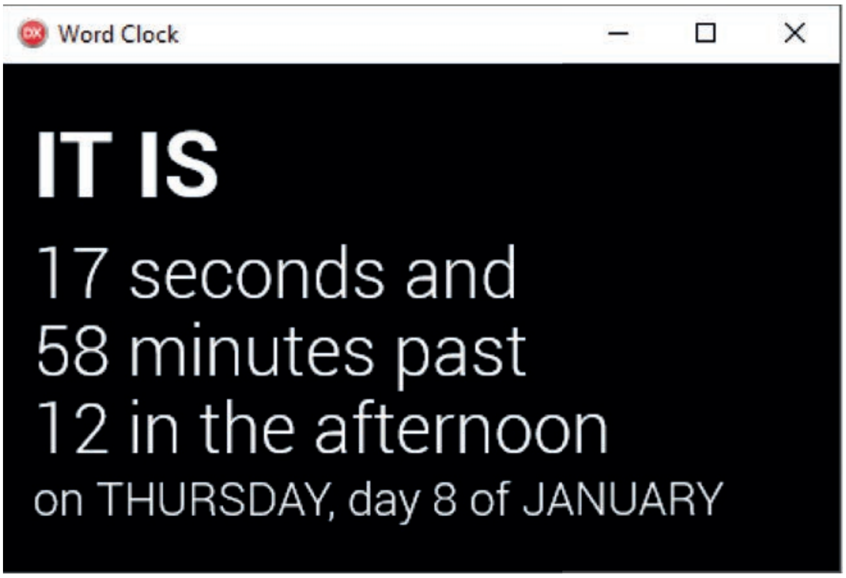
1.1.1 Give the method and syntax for the following:

- **a.** Converting a string to an integer.
- **b.** Calculating the remainder of division.
- **c.** Rounding a real number to an integer.
- **d.** Copying characters from one string to another string.
- **e.** Finding the year of a date and time.
- **f.** Determining if a year is a leap year.
- **g.** Declaring an array with 5 integer elements.
- **h.** Opening a file to add text to it.

1.1.2 What is the difference between the APPEND, REWRITE and RESET procedures?

Activity 1.2

Update your Word Clock activity so that it looks as follows. The date at the bottom of the clock should update automatically and be shown in a single label.



IT IS

17 seconds and
58 minutes past
12 in the afternoon

on THURSDAY, day 8 of JANUARY

Figure 1.3: *The World Clock*

CODE YOUR OWN

Procedures and functions are groups of statements that are created that complete a specific task. These procedures and functions can be created for a number of different reasons. They:

- break the code into smaller, more manageable pieces.
- make the code easier to understand.
- make it easier to find problems.
- make it easier to add new functions or update the code.
- allow you to re-use certain algorithms from different events.

For these reasons, professional programmers try to make use of procedures and functions whenever their code becomes too long or difficult to manage. A general guideline is that you should try to not have the same code repeated, keep procedures (including events) to less than 15 lines of code, and they should almost never be more than 30 lines of code. Just as importantly, each procedure should do only one thing and it should do this thing well.

Both procedures and functions can accept variable inputs (called parameters), but only functions can return a value. The following syntax is used to create a procedure:

Procedure with value parameters

```
procedure ProcedureName(parameterName1 : type1;
parameterName2 : type2);
var
  var1 : Type;
begin
  Statement1;
  Statement2;
  ...
  Statement1000;
end;
```

Once a parameter value is sent to a procedure, it works exactly the same as any other variable. The only difference between parameters and variables is that the parameters are defined next to the procedure name and that the parameters start with a value. To call a procedure, the following syntax is used.

Calling a procedure

```
ProcedureName(Value1, Value2 ... Value1000);
```

Where the values must be the same number, in the same order and of the same type as the parameters in the declaration the procedure.

There are only three differences between the function syntax and the procedure syntax:

- All functions start with the key word **function**.
- After the parameters are listed, functions must specify the variable type of the return value.

- Inside the function, a value must be assigned to the variable *Result*. This value will be returned once the last line of the function has been executed.

Function syntax

```
function FunctionName(parameter1 : Type; parameter2 : Type): ReturnType;
var
  var1 : Type;
begin
  Statement1;
  Statement2;
  ...
  Result := Value of ReturnType;
end;
```

A function can be called in the same way as a procedure except that its return value must be assigned to a variable. To call a function:

Calling a function

```
ReceivedValue := FunctionName(Value1, Value2 ... Value1000);
```

Custom functions and procedures must be declared after the **implementation** section of a project but before the first automatically created procedure. This allows all the automatic procedures following the custom functions to make use of the custom functions.

Custom functions and procedures also have access to the form's global variables and components. However, in order to access the components, the procedure and functions must first use the form's name, followed by a full stop, followed by the component's name. This is shown in the code below.

Accessing a component from user defined methods

```
sValue1 := frmName.lblData.Caption;
sValue2 := frmName.edtData.Text;
```

Example 1.4 Use reading from and writing to a file to copy the contents

Open the project in the folder 01 – Copy File and insert the following code:

Declare the procedure *copyMyFile* in the private section:

```
  procedure copyMyFile(fromFile : String; toFile :String);
```

Press <Control + shift + C> to create the stub for the procedure.

Example 1.4 Use reading from and writing to a file to copy the contents *continued*

Add the following code:

```
procedure TForm1.copyMyFile(fromFile, toFile: String);
var
  infile : TextFile;
  outFile: TextFile;
  line : String;
begin
  if Not FileExists(fromFile) then
   showMessage(fromfile + ‘ NOT FOUND’)
  else
  begin
   AssignFile(inFile, fromFile);
   Reset(infile);
   AssignFile(outFile, toFile);
   Rewrite(outFile);
   while not eof(inFile) do
   begin
     readln(infile, line);
     writeln(outFile, line);
   end;
   closeFile(infile);
   closeFile(outFile);
   showMessage(‘File copied’);
  end;
```

Add the code below to the [Copy File] button Onclick event:

```
procedure TForm1.btnCopyFileClick(Sender: TObject);
var
  fromFile: String;
  toFile: String;
begin
   fromFile := InputBox('File Copy', 'Copy from – filename:','');
   toFile := InputBox('File Copy', 'Copy to – filename:','');
   copyMyFile(fromFile, toFile);
End;
```

Run the program and enter the filename **Acronyms.txt** . This file will be copied to a file with the name of your choice, that is, the name you enter in the second input box.

Activity 1.3

Using pen and paper, create the following user-defined functions.

1.3.1 A custom function that accepts a number and an exponent and calculates the solution.

1.3.2 A function that returns only the largest value from an array.

1.3.3 A function that returns the n^th^ largest value from an array, where n is provided by the user.

UNIT

1.4 User interface design

The difference between an application that is used by millions of people and an application that is left untouched is often the user interface. This is because applications with easy, fun, beautiful, simple and intuitive interfaces are enjoyable to use, while poorly designed user interfaces can be incredibly confusing and frustrating to use.

This unit looks at the principles of user interface design, how a multi-form user interface can be created and how dynamic user interfaces can be created.

WHAT IS USER INTERFACE DESIGN?

https://www.youtube.com/watch?v=WtoK7BzalsA

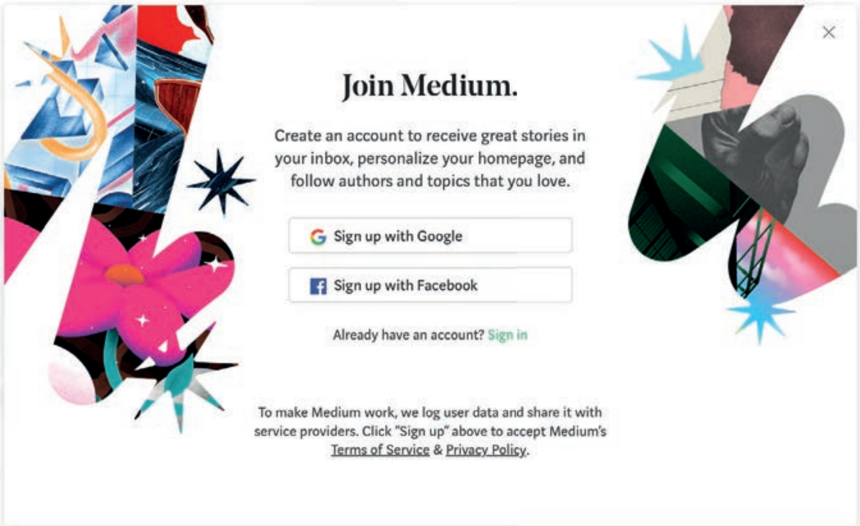
PRINCIPLES OF USER INTERFACE DESIGN

The following principles should be used when creating user interfaces:

- Put users in control
- Minimise the effort
- Eliminate useless items
- Give visual cues
- Give feedback
- Be consistent
- **Accommodate all users.** The first step to building a good user interface is to give users control. This means that users should be able use the application and complete tasks in ways that feel intuitive to them.
- The next core concept is to **minimise the effort it takes to use the program**. This means you should not ask users to enter the same information more than once, not use unfamiliar jargon or strange terms, make the transition to the next step obvious, help users to provide the correct information, and protect users work so that they never accidentally lose work.

Take note

Also see Shneiderman's Eight Golden Rules of Interface Design in Annexure A.



Join Medium.

Create an account to receive great stories in your inbox, personalize your homepage, and follow authors and topics that you love.

Already have an account? Sign in

To make Medium work, we log user data and share it with service providers. Click "Sign up" above to accept Medium's Terms of Service & Privacy Policy.

Figure 1.4: *Medium (https://medium.com/) minimises user effort allowing users sign-in with a Google or Facebook account*

- Next, **visual elements that are not helping users should be eliminated or hidden until needed.** In this way, users are not overwhelmed by all the options and it is clear to users what they are expected to do. Similarly, you should also eliminate steps or tasks that are irrelevant to the solution of the problem.
- One way in which applications can be made simpler is by **using visual cues.** These cues help users to understand where they are in your application, what they are doing, how the application works and what you expect them to do next. Popular visual cues include colours, photos, arrows, icons, animations or a single larger component.

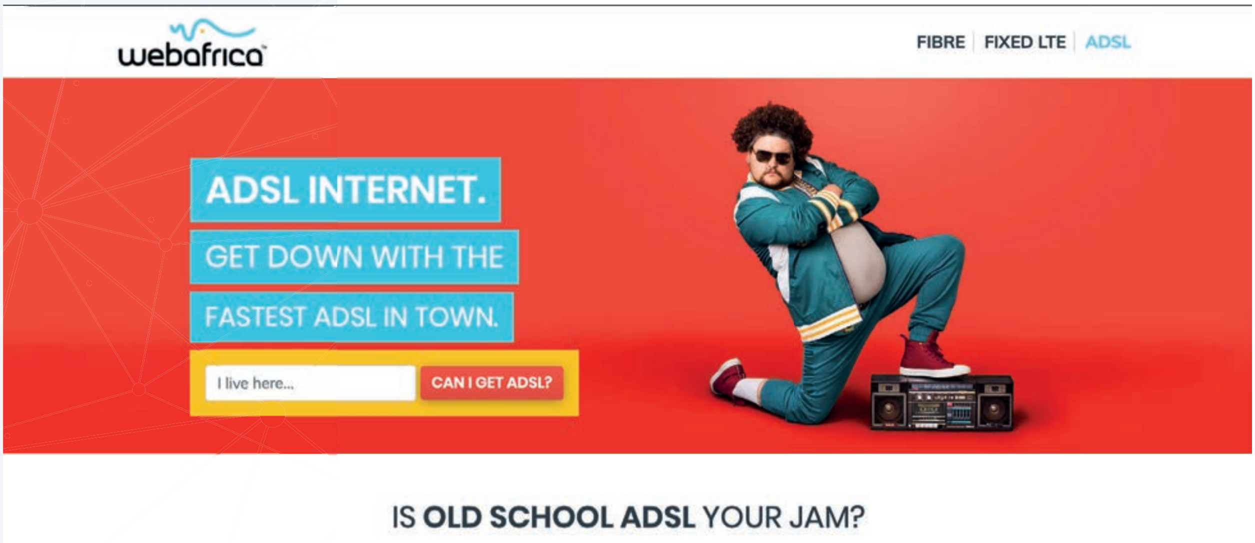


Figure 1.5: *WebAfrica uses bright colours to draw the attention to interactive elements*

- The fifth key concept is to **give users feedback.** This means that users should receive an acknowledgement when they interact with your application. This can include a progress bar, a message, a loading animation, a button changing colours or sound. By providing user with immediate feedback, they never have to worry that their action was not recorded.
- **Consistency in user interface design** is another key concept. Users who have used some parts of your application should be able to predict how other parts of your application will work. Similarly, items should generally remain in the same place, be the same size and have the same colour across your screens.

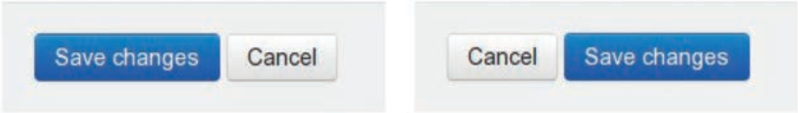


Figure 1.6: *Buttons with inconsistent positions can create frustrated users*

- The final concept is to **accommodate different users**. Users can differ according to:
  - What they want from your application.
  - Their skills and experience.
  - Their ability to use your application.
  - The tools they are most comfortable with.
  - Their expectations.

If you want all these users to enjoy using your application, you need to carefully consider each element of your application.

MULTI-FORM USER INTERFACES

To create a multi-form (or multi-screen) user interface, you need to follow three steps:

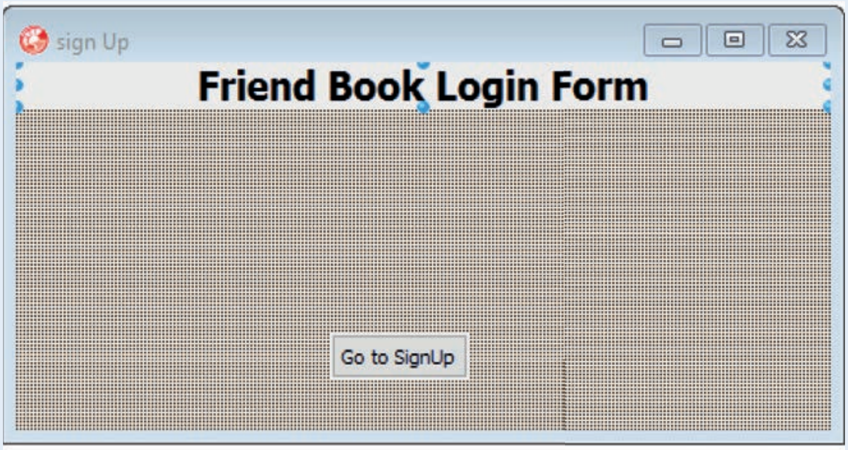
- Create the second form
- Open the second form
- Pass data between the forms

The example below shows how to create a second form.

Example 1.5 Creating a form

Load the project FriendBookLogin

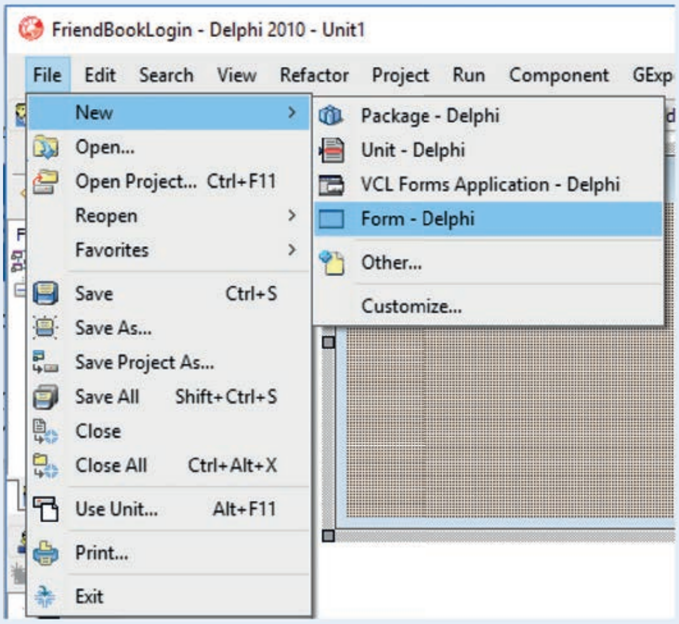
You will see this form has been created for you.



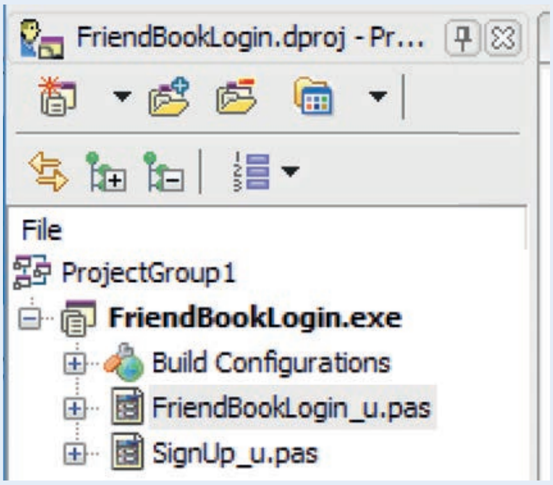
To a create a new form:

1. Open the *File* menu in RAD Studio.
2. Select the *New* option, the click on the *Form – Delphi* option.

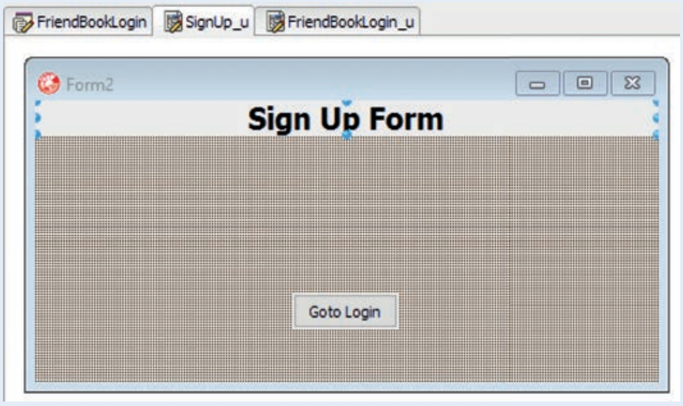
Example 1.5 Creating a form *continued*



3. You should see a new form open in your project, and **Unit2.pas** appear in the *Project Manager* panel.
4. Save this unit in the same folder as your project and give the name **SignUp-u**.



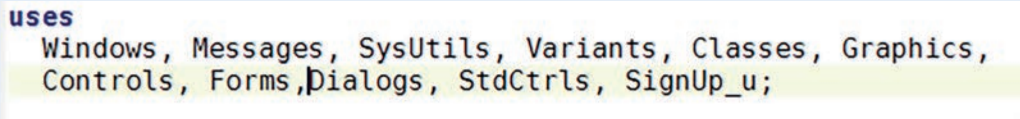
5. You can now edit the second form by clicking on its *Design* tab.
6. Copy the design as illustrated:



Example 1.6 Linking forms

To link forms in a Delphi application:

1. Find the USES statement at the top of your first form's *FriendBookLogin_u* code .
2. Add the name of the unit **SignUp_u** to the uses section. This allows your first form to access your second form.



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

3. In the OnClick event of the [Go To SignUp] button add the second form's Show method *frmSignUp.Show.*
4. To close the second form, either click on the [Close] button in the top-right corner or run the form's Close method frmSignUp.Close.
5. Experiment by changing the frmSignUp.Show to frmSignUp.Showmodal. Explain in your own words how these two instructions differ.

Once you have created your second form, you need to link the two forms so that you can swap between them.

The final step is to pass information between the forms. There are two different ways to do this:

- By including a unit in the USES section, you gain access to its **visible** variables through the form e.g. frmSignUp.iNumber. You also gain access to the properties of its components directly e.g. frmSignUp.lblName.Text.
- By saving the data in an external file (such as a textfile), any unit can read the data from the file.

In most situations, you will use the first option to share data between forms. However, saving the data to a file can be useful if the data needs to be stored permanently.

By using these techniques to create new forms, link the forms and share data between the forms, you can add any number of forms to an application. This helps you to create a more organised application where each form serves a specific purpose.

DYNAMIC USER INTERFACES

Dynamic user interfaces are interfaces in which some or all of the user interface components are created using code or while the application is running. By creating user interface components using code, you can dynamically update your user interface based on user decisions, amount of data available or on certain conditions being met.

The following section will look at the different ways to create visual and interactive components.

VISUAL COMPONENTS

To create a non-interactive component using code, you need to follow three steps:

- Declare an empty variable of the component's class.
- Create the component and assign it to the empty variable.
- Use the component variable to assign values to the required properties.

BUILDING EFFECTIVE USER INTERFACES

https://www.youtube.com/watch?v=7LWiM7OJick

For example, when creating a TLabel component, you should start by declaring a global label variable.

Declaring global components

```
var
  frmMain: TfrmMain;
  lblDynamicLabel : TLabel;
```

Next, you should use the TLabel class's Create method to create and assign a label to this variable.

Creating the component

```
procedure TfrmDynamicItems.btnCreateLabelClick(Sender:
TObject);
begin
  lblDynamicLabel := TLabel.Create(Self);
end;
```

Finally, you need to set the properties of the label to ensure it is visible to the user.

Standard label properties

```
lblDynamicLabel.Parent := Self;
lblDynamicLabel.Text := 'Hello, World!';
lblDynamicLabel.Top := 100;
lblDynamicLabel.Left := 100;
lblDynamicLabel.Width := 100;
lblDynamicLabel.Height := 20;
```

The table below lists a few of the most important properties.

PROPERTY	COMPONENTS	PARAMETERS	DESCRIPTION
Parent	All	Form	All components must be assigned to a parent form. The component will be closed once the parent is closed.
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.
Caption	Label, Button	String	The text shown by labels and buttons.
Text	Edit	String	The text shown by text boxes.
Bitmap.CreateFromFile	Image	String	A string of the file path of the image that will be displayed.
Items.Add	ListBox, ComboBox, RadioGroup	String	The string that will be shown on one row of a list box, combo box or radio group.
Lines.Add	RichEdit, Memo	String	The string that will be shown on one row of a rich edit of memo

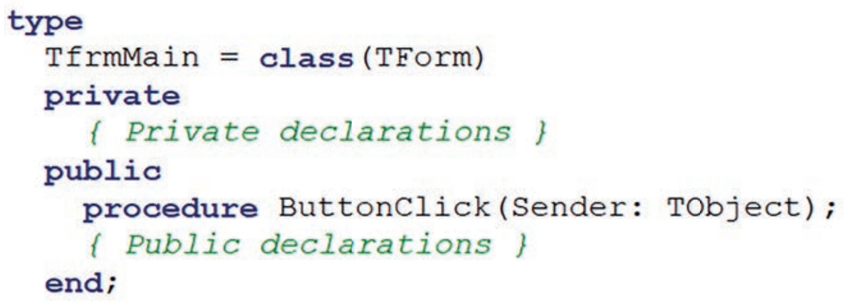
INTERACTIVE COMPONENTS

Creating an interactive component during run–time with an event (such as an OnClick, OnChange or OnTimer event), can be done in seven steps:

- Declare the component variable.
- Create the component and assign it to the variable.
- Set the component properties.
- Declare the name of a custom procedure in the *public* section of your form.
- Add the (Sender: TObject) parameter to the procedure's name to identify the type of component.
- Create the custom procedure for your event in the code.
- Assign the name of the procedure to the appropriate event property (for example, the OnClick property).

The first three steps of this procedure are identical to those of visual components.

For the fourth step, you need to add the name of the custom event handler, that will be linked to your interactive component's event, to the *public* section of your code. The *public* section can be found inside the *type* section, and above the global variables and *implementation* sections. The image below shows a custom procedure called *ButtonClick* added to the public section. .



```
type
  TfrmMain = class(TForm)
  private
    { Private declarations }
  public
    procedure ButtonClick(Sender: TObject);
    { Public declarations }
  end;
```

Figure 1.7: *A custom procedure in the public section*

As shown in the image above, all custom event handlers must have TObject parameter called *Sender*.

The next step is to create the custom event handler, which is technically a method linked to your form. This can be done in the same way as the custom procedures created in the previous unit. However, since this procedure is a form method that forms part the form, it needs to include the full name of the form before the procedure name. The code below shows an example of the *ButtonClick* form procedure.

Dynamic OnClick method

```
procedure TfrmMain.ButtonClick(Sender: TObject);
begin
  ShowMessage('Hello, Button!');
end;
```

Finally, the method name (excluding the “TfrmMain” part) can be assigned to the appropriate event property when the button is created.

```
btnDynamic := TButton.Create(Self);
btnDynamic.Parent := Self;
btnDynamic.Text := 'Click me!';
btnDynamic.Left := 10;
btnDynamic.Top := 10;
btnDynamic.OnClick := ButtonClick;
```

Example 1.7 Photo Bomb

At the start of the 21st century when most computers were still using Windows XP, the most common type of virus was a virus that caused infected computers to automatically open hundreds of new windows containing advertisements. These windows would flood the user’s screen and open more quickly than they could be closed. Since most of these advertisements were for adult-only websites, these viruses were especially embarrassing for their victims!

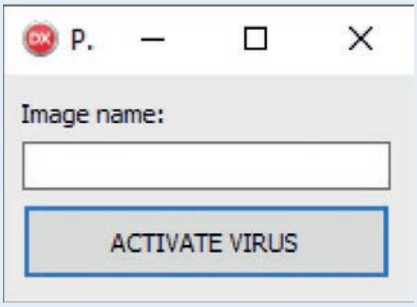
For this application, you will simulate these old viruses. To start the virus, the user will enter the name of an image and click on a button with the text [Activate Virus]. This button will increase the size of the form and create a new copy of the image every few milliseconds. The position of each image should be randomly selected inside the form.

You can use your own image or the teacher will provide the **stop.bmp** image to you. Take note, this image must be placed in your application’s win32\debug folder.

The project should be saved in the folder called 01 – Photo Bomb.

Solution

For this application, you can create a very simple user interface. Even though it is not visible, this interface includes a disabled timer.



Once the [*Activate Virus*] button is clicked, the following OnClick event is run.

Activate Virus OnClick event

```
procedure TfrmPhotoBomb.btnActivateVirusClick(Sender: TObject);
begin
  frmPhotoBomb.Height := 640;
  frmPhotoBomb.Width := 800;
  tmrTimer.Enabled := True;
end;
```

This event simply resizes the form and enables the timer. Once the timer has been enabled, the following OnTimer event activates every few milliseconds (based on the timer’s interval).

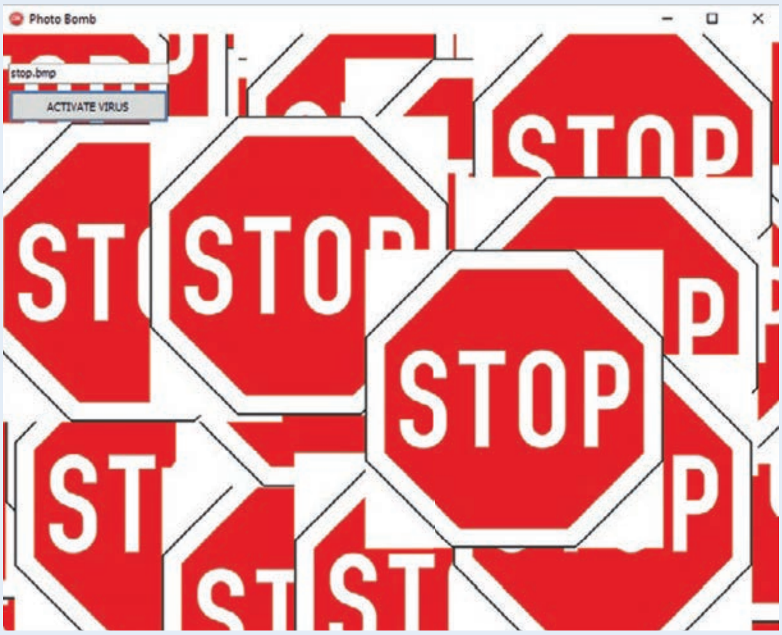
Example 1.7 Photo Bomb *continued*

OnTimer event

```
procedure TfrmPhotoBomb.tmrTimerTimer(Sender: TObject);
var
  iRandomTop, iRandomLeft : Integer;
  imgPhoto : TImage;
begin
  randomize;

  iRandomTop := Random(frmPhotoBomb.Height) – 150;
  iRandomLeft := Random(frmPhotoBomb.Width) – 150;

  imgPhoto := TImage.Create(Self);
  imgPhoto.Parent := Self;
  imgPhoto.Top := iRandomTop;
  imgPhoto.Left := iRandomLeft;
  imgPhoto.Width := 300;
  imgPhoto.Height := 300;
  imgPhoto.Picture.LoadFromFile(edtImageName.Text);
end;
```

Looking at this code, you start by randomly selecting a coordinate for the top and left properties of the image. These positions are based on the height and width of the form minus half the height and width of the image. This will ensure that at least half the height and width of the image is always visible, regardless of the randomly selected position.

The next step is to dynamically create the image. This includes setting all the standard properties for the image, including the image's parent, top, left, width, height properties, and using the Picture.LoadFromFile( ) method.



stop.bmp

Activity 1.4

Answer the following questions in your own words.

1.4.1 List and describe four principles of user interface design.

1.4.2 What are the three steps needed to create a multi-form application?

1.4.3 List five common properties that should be set when dynamically creating a label.

1.4.4 What is the syntax for an OnClick event declaration used with a dynamically created button?

Activity 1.5

Create the following multi-form application:

1.5.1 The project has two forms.

1.5.2 Each form contains a textbox with a default value and a button.

1.5.3 When the button on the first form is pressed:

- **a.** swap the values between the two textboxes.
- **b.** hide the active first form and display the inactive second form.

UNIT

1.5 Databases

When programmers want to save data, their first choice is to use a database. This section will look at five parts of using a database with your application:

- Creating a database
- Connecting a Delphi project to a database
- Reading data from a database
- Writing data to a database
- Using the data from a database
- Modifying/manipulating the data from a database

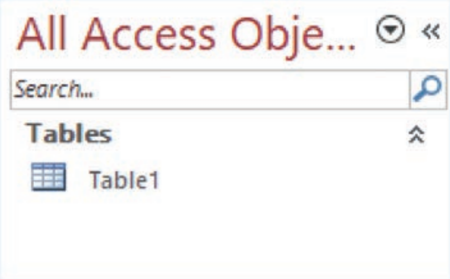
CREATING A DATABASE

The example below shows you how to create a simple, single-table database.

Example 1.8 Creating a database

To create an Access database:

1. Open Microsoft Access from the *Start Menu*.
2. Double click on the *Blank database* option in the main window.
3. In the window that opens, click on the *Open* icon 📂.
4. Select an appropriate folder to save your database and click *OK*.
5. Select *Microsoft Access Database (2002-2003)* as the file type.
6. Enter the name of your database in the *File name* text box.
7. Click OK to close the window, then click on the *Create* button to create the database.
8. Find the *Table1* table in the *All Access Objects* panel on the left side of Access.



9. Right click on this table and select the *Rename* option.
10. Enter the name of your table and press Enter.
11. Inside the table, click on the *Click to Add* option at the top of the table.
12. Select the field type then enter a name for the field.
13. Continue adding fields to your table until all fields have been added.
14. Enter values into these fields to create new records.

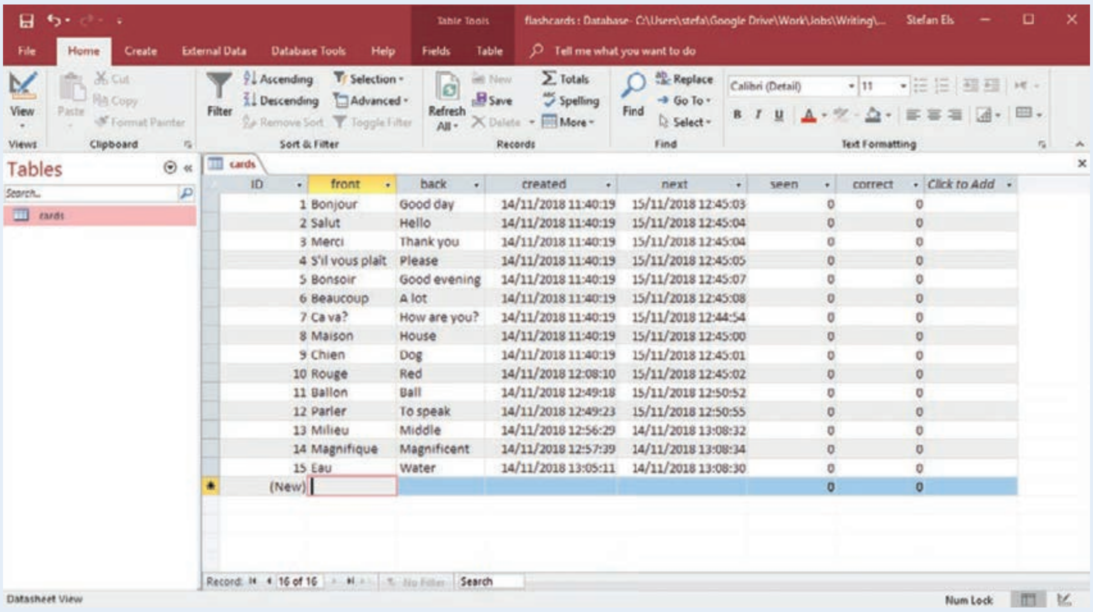
Take note

When creating mock data for a database, it is useful to use a site such as Mockaroo Random data generator. https://mockaroo.com/

DATABASES IN DELPHI

https://www.youtube.com/watch?v=dwb0wv6lJqA

Example 1.8 Creating a database *continued*



cards

ID	front	back	created	next	seen	correct	Click to Add
1	Bonjour	Good day	14/11/2018 11:40:19	15/11/2018 12:45:03	0	0	
2	Salut	Hello	14/11/2018 11:40:19	15/11/2018 12:45:04	0	0	
3	Merci	Thank you	14/11/2018 11:40:19	15/11/2018 12:45:04	0	0	
4	S'il vous plait	Please	14/11/2018 11:40:19	15/11/2018 12:45:05	0	0	
5	Bonsoir	Good evening	14/11/2018 11:40:19	15/11/2018 12:45:07	0	0	
6	Beaucoup	A lot	14/11/2018 11:40:19	15/11/2018 12:45:08	0	0	
7	Ca va?	How are you?	14/11/2018 11:40:19	15/11/2018 12:44:54	0	0	
8	Maison	House	14/11/2018 11:40:19	15/11/2018 12:45:00	0	0	
9	Chien	Dog	14/11/2018 11:40:19	15/11/2018 12:45:01	0	0	
10	Rouge	Red	14/11/2018 12:08:10	15/11/2018 12:45:02	0	0	
11	Ballon	Ball	14/11/2018 12:49:18	15/11/2018 12:50:52	0	0	
12	Parler	To speak	14/11/2018 12:49:23	15/11/2018 12:50:55	0	0	
13	Milieu	Middle	14/11/2018 12:56:29	14/11/2018 13:08:32	0	0	
14	Magnifique	Magnificent	14/11/2018 12:57:39	14/11/2018 13:08:34	0	0	
15	Eau	Water	14/11/2018 13:05:11	14/11/2018 13:08:30	0	0	
(New)					0	0	

Record: 16 of 16

15. Save the database in the same folder as your application and close Microsoft Access.

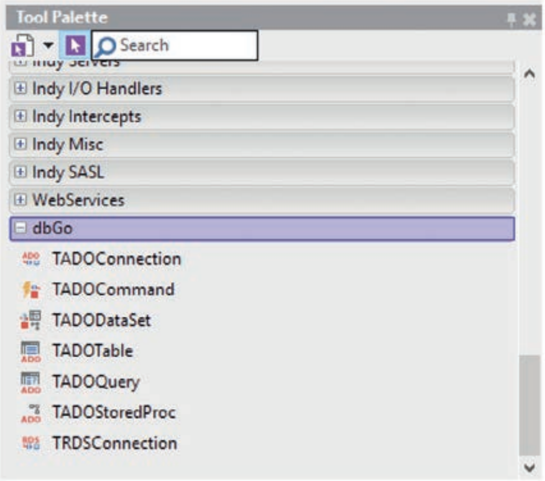
CONNECTING TO A DATABASE

To connect to a database in Delphi, you will need to use three invisible components:

- **TADOConnection:** Creates a connection to an external database.
- **TADOTable:** Uses the database connection to connect to a specific table inside your database.
- **TDataSource:** Create a connection between your TADOTable and Delphi visual components.

The first two of the components can be found from the *dbGo* list in RAD Studio's *Tool Palette*, while the *DataSource* component can be found from the *Data Access* list.

These components should not be added directly to a unit, but rather to a data module. A data module is a special form or container to keep all the database components organised and together. By separating the database connections from any form, you can import the database into each form through the "uses" section.



Tool Palette

Search

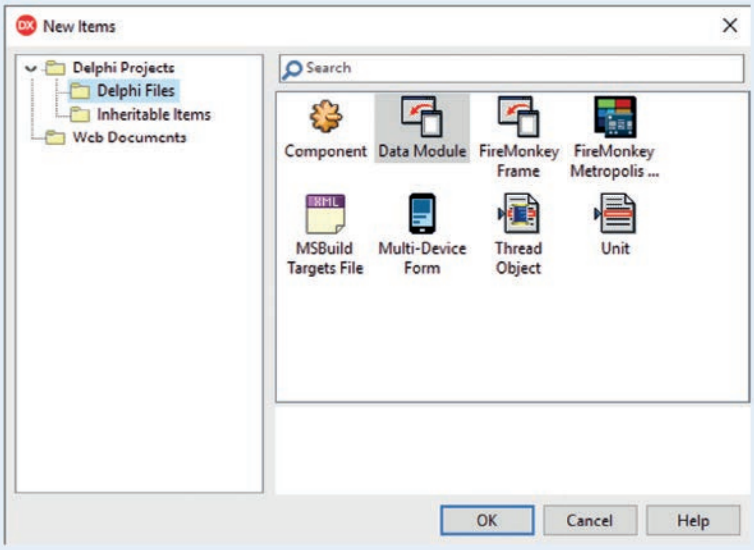
- Indy I/O Handlers
- Indy Intercepts
- Indy Misc
- Indy SASL
- WebServices
- dbGo
  - TADOConnection
  - TADOCommand
  - TADODataSet
  - TADOTable
  - TADOQuery
  - TADOStoredProc
  - TRDSConnection

Figure 1.8: *The dbGo list from the Tool Palette*

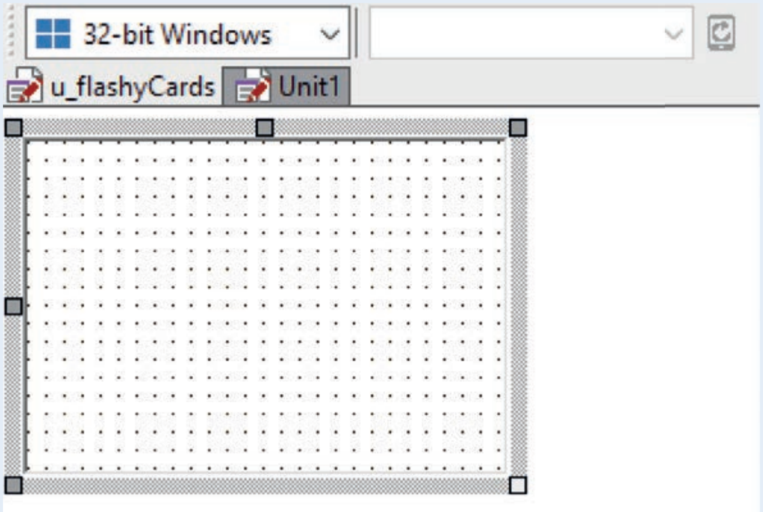
Example 1.9 Adding a data module

To add a data module to your project:

1. In the *Project Manager* panel in the top right corner of the screen, right click on the .exe project file.
2. Select the *Add New* option and click on *Other*.
3. Inside the *New Items* window, select the *Data Module* option and click *OK*.



4. This will add a new data module to your project.



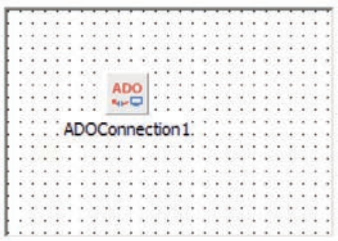
5. Save the data module in your project folder and rename the data module form (for example, *dbmName*).
6. Select your main form and open the *Code* screen.
7. Add the data module's name (e.g. *dbmName*) to the USES section of your main form's code.

Once you have created the data module, you can add the database connection components to it.

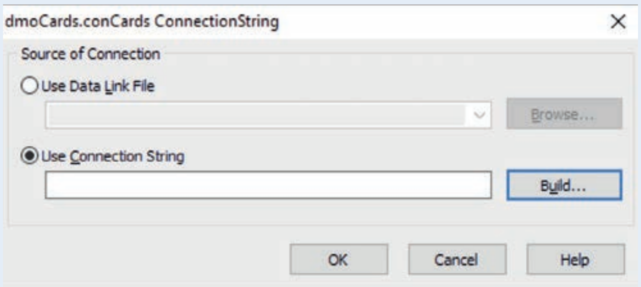
Example 1.10 Adding database connection components

To add the database connection components:

1. Add a *TADOConnection* component to the data module and change the name, for example, conCards.



2. Change the connection's *LoginPrompt* property to False to avoid receiving a login prompt every time you run the application.
3. Double click on the connection component in your data module. This will open the *Connection String* window.



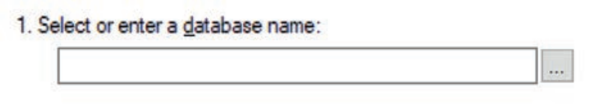
dmoCards.conCards ConnectionString

Source of Connection

○ Use Data Link File

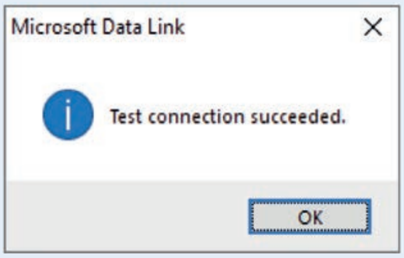
● Use Connection String

4. Click on the [Build] button.
5. In the *Data Link Properties* window, select the *Microsoft Jet 4.0 OLE DB Provider* (.mdb database) or Microsoft Database Engine (.accdb database) option and click *Next*.
6. In the *Connection* tab, enter your database name and extension in the textbox.



1. Select or enter a database name:

7. Select your database and click *Open for the .mdb database option*.
8. In the *Connection* tab click on the *Test Connection* button. You should receive a message stating that the test connection succeeded.



Microsoft Data Link

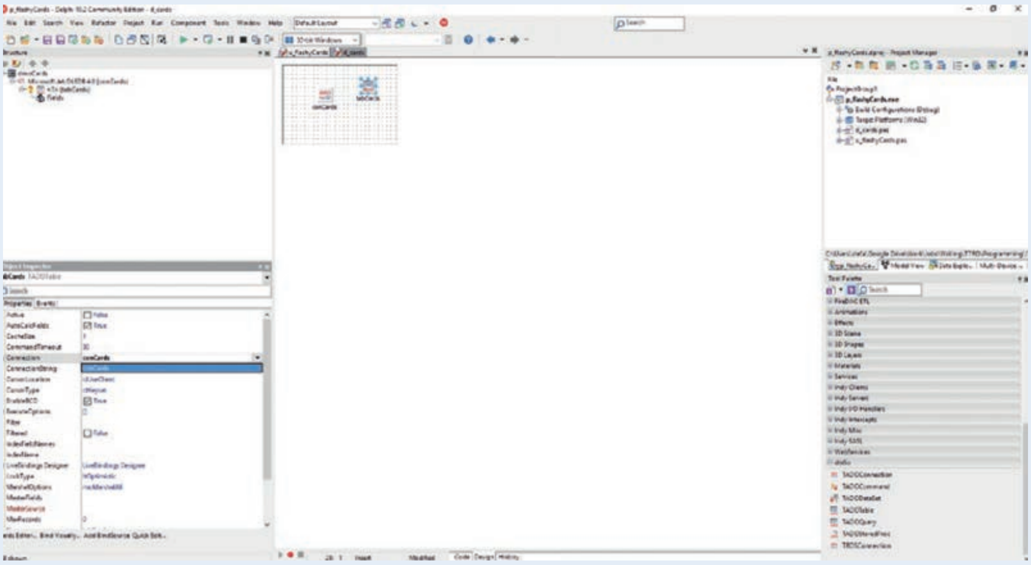
Test connection succeeded.

9. Click *OK*, then click *OK* again.

Example 1.10 Adding database connection components *continued*

Now that the database has been connected to your Delphi project, you can add a connection to a specific table. To do this:

1. Add a *TADOTable* component to your data module.
2. Click on the dropdown list next to the *Connection* property of your table and select the name of your database connection component.



3. Change the value of the *TableName* property to the name of the table you are connecting to.
4. Change the table's *Active* property to True.
5. Add a *TDataSource* component to your application.
6. Change its *DataSet* property to the name of your *TADOTable* component.
7. Save your application.

You now have a connection directly to a table on your database.

READING DATA FROM A DATABASE

Once a database connection has been made, you can start using and displaying the data in your application. The easiest way to do this is to display the database table as a table (or TDBGrid component) in Delphi. The *TDBGrid* component can be added from the *Data Controls* list in the *Tool Palette*.

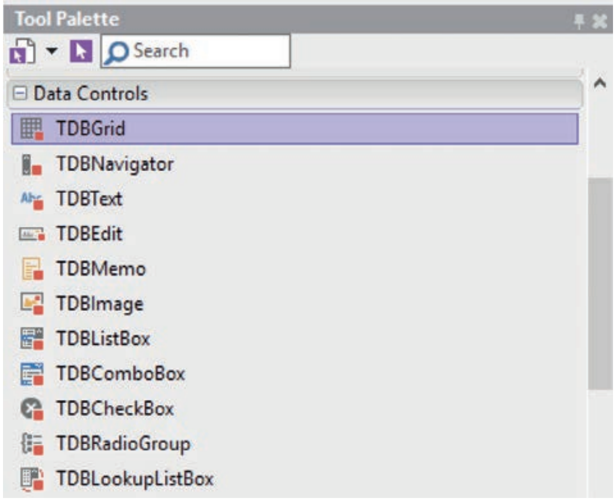


Figure 1.9: *The TDBGrid component in the Grids list*

Example 1.11 Using the grid component

To link a database table to a grid component:

1. Add a *TDBGrid* to your form.
2. Select the Grid component and change its *DataSource* property to the name of your *TDataSource* component.
3. Save and test our application. If all the connections were made correctly, you should now see the data from your database table appear on your grid component.

WRITING DATA TO A DATABASE

To add a record to a database, you will use two of the *TADOTable* component's methods.

Method	Description	Example
Append	Creates an empty record and selects it, allowing you to add values to the record.	tblCards.Append;
Insert	Inserts an empty record after the currently selected record and selects the new, empty record.	tblCards.Insert;
Post	Saves all the changes you have made to records. Without running this command, all changes will be discarded.	tblCards.Post;

By using the append or insert method, an empty row is added to the end of your database. You can then set the values in this row, much like you would change the values of a variable.

Adding values to database fields

```
tblCards.Append;
tblCards[‘fieldName1’] := 'Hello';
tblCards[‘fieldName2’] := 'World';
tblCards.Post;
```

Take note, the field names are placed inside single-quotation marks, inside square brackets, after the table name. This syntax will always be used to access specific fields in a table.

USING THE DATA FROM A DATABASE

To use the data from a database (without first showing it in a grid), you use an algorithm that is similar to the algorithms used to read text files:

- Selecting the first record of the database.
- Create a while loop that runs until you reach the end of the file.
- Inside the loop, read the relevant fields for your application.
- Using these values, manipulate the data as needed by your application.
- Finally, move to the next record and repeat the process.

The table below shows the table methods that allow you to iterate through the data.

Method	Description	Example
First;	Selects the first record in your table, allowing you to read the value of its fields.	tblName.First;
Next;	Selects the next record in your table. This must be included in your while loop.	tblName.Next;
Eof;	Returns the value True if there are no records left in your table.	tblName.Eof;

Once you have selected the correct record, you can use the tableName[‘fieldName’] syntax to read the fields’ data. The code below shows an example of these methods being used.

Using the data methods

```
dbmData.tblName.First;
while not dbmData.tblName.Eof do
begin
  sValue := dbmData.tblName['fieldName'];
  dbmData.tblName.Next;
end;
```

MODIFYING THE DATA FROM A DATABASE

- To modify data saved in a database:
- Select the record you want to modify.
- Set the record to edit mode (using the Edit method).
- Set the new values for the record.
- Post the new values to the database.

The following code shows how this can be done, using the Locate method to select the correct record.

Modifying data

```
dbmData.tblName.First;
dbmData.tblName.Locate('fieldName', searchValue, []);
dbmData.tblName.Edit;
dbmData.tblName['fieldName'] := newValue;
dbmData.tblName.Post;
```

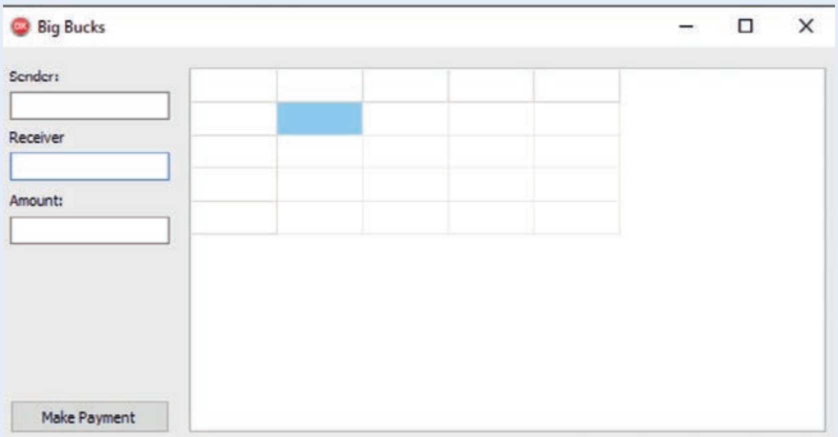
To see how a database can be used in practice, work through the following example.

Example 1.12 Big Bucks setup

You and a group of friends have decided to create Big Bucks, an application to track debts within your circle of friends. The application will record every payment between friends and automatically determine if a payment is a new loan or a repayment.

To create Big Bucks:

1. Open a new application and save it in the folder 01 – Big Bucks.
2. Create the following user interface.

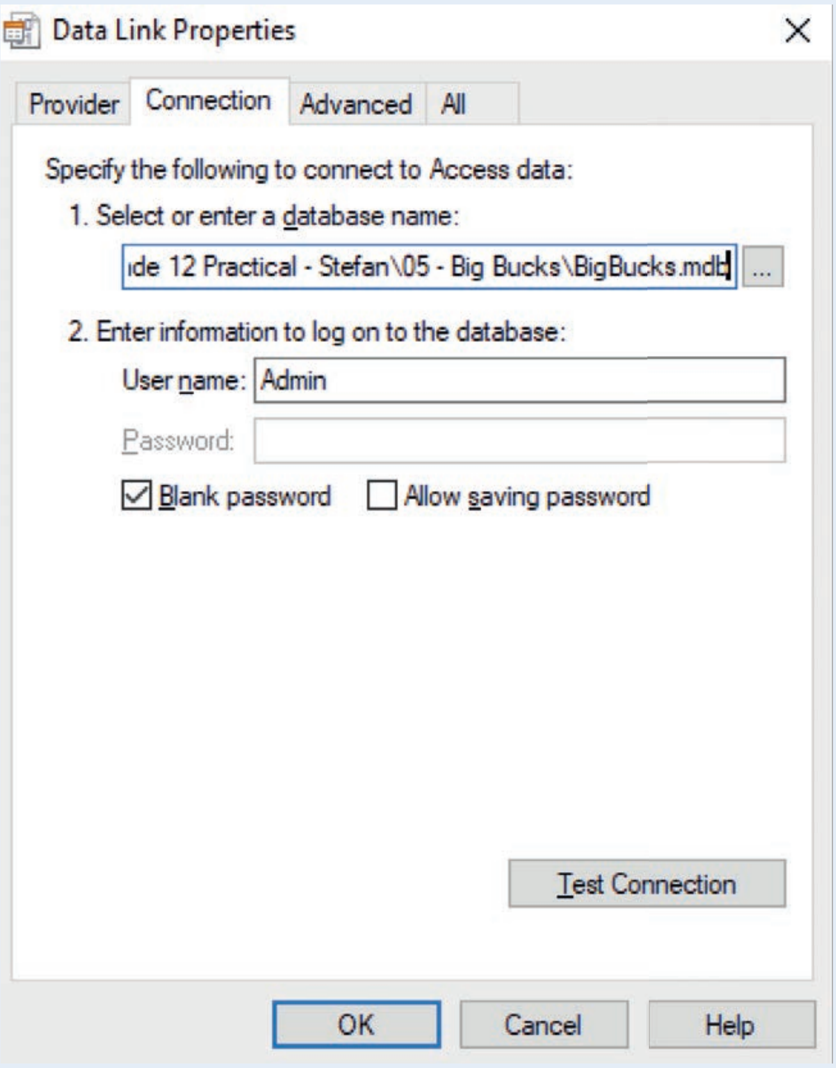


The box on the right of the table is a TStringGrid component called *grdBigBucks*.

3. Create a new data module called *bigBucks_d.pas*.
4. Change the name property of the data module to *dbmBigBucks*.
5. Add a TADOConnection component to the data module (called *conBigBucks*) and a *TADOTable* component to the data module (called *tblBigBucks*).
6. Add a *TDataSource* component to the data module called *sorBigBucks*.
7. Copy the **BigBucks.mdb** database from your teacher to your project's folder.
8. Select the *conBigBucks* connection and click on the [Edit ConnectionString] button at the bottom left of the *Object Inspector*.
9. Click on the Build button.
10. Select the *Microsoft Jet 4.0 OLE DB Provider* option and click *Next*.

Example 1.12 Big Bucks setup *continued*

11. Use the … button to select the "BigBucks.mdb" database then click on the [Test Connection] button.



Data Link Properties

Specify the following to connect to Access data:

1. Select or enter a database name: ıde 12 Practical - Stefan\05 - Big Bucks\BigBucks.mdb

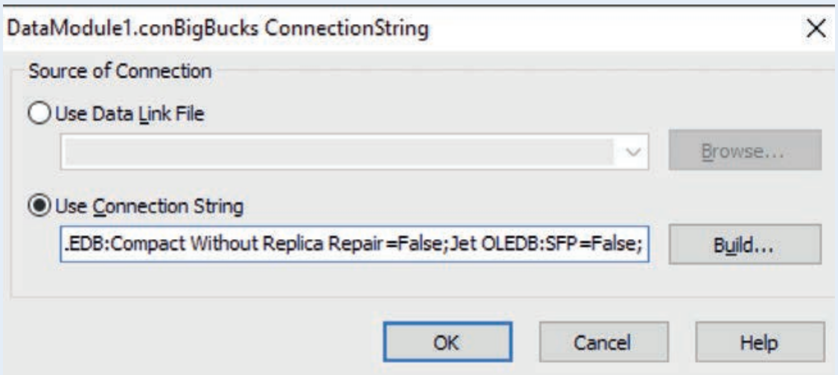
2. Enter information to log on to the database:

User name: Admin

Password:

☑ Blank password ☐ Allow saving password

12. If the test is successful, click on the *OK* button, then click on the *OK* button in the *Connection String* window.



DataModule1.conBigBucks ConnectionString

Source of Connection

○ Use Data Link File

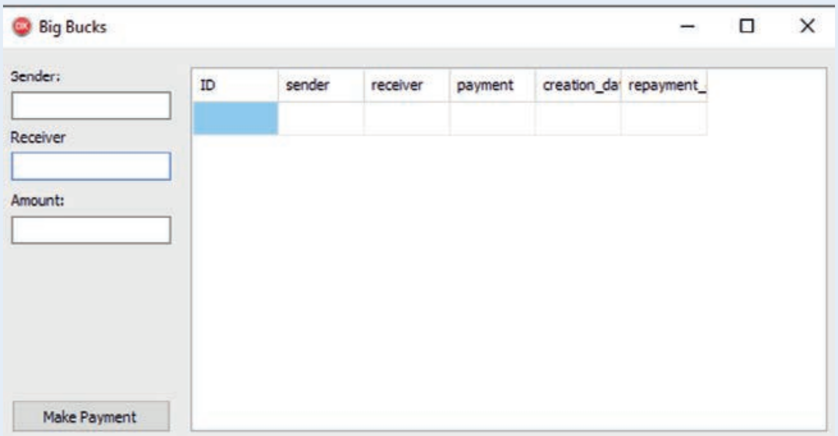
● Use Connection String: .EDB:Compact Without Replica Repair=False;Jet OLEDB:SFP=False;

Example 1.12 Big Bucks setup *continued*

13. Change the connection's *LoginPrompt* property to False and the *Connected* property to True.
14. Select the *tblBigBucks* table component.
15. Change the *Connection* property to *conBigBucks*, the *TableName* property to *Payments* and the *Active* property to True.
16. Select the *sorBigBucks* component and set the DataSet property to *tblBigBucks*.
17. Open the code editor of the **bigBucks_u** unit.
18. Add *bigBucks_d* to the **USES** section of the code. This imports the data module into your unit.

```
uses
    Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
    System.Classes, Vcl.Controls, Vcl.Forms, Vcl.Dialogs, Vcl.StdCtrls,
    Vcl.Grids, bigBucks_d;
```

19. Select the *grdBigBucks* component and set the *DataSource* property to *sorBigBucks*.
20. Save and run your application. You should see the headings from your *Payments* table in your string grid.



ID	sender	receiver	payment	creation_da	repayment_

Congratulations, you have just created the database connection for your application. In the next activity, you will create the code that uses this database.

Example 1.13 Big Bucks code

In the previous example, you set up the database connection for your Big Bucks application. In this example, you will allow users to add a payment of money between two friends. This payment will have one of three impacts:

- If no current debt exists between the two friends, the payment will record a new debt between the friends.
- If receiver already owes the sender money, an additional debt will be created.
- If the sender owes the receiver money, the payment will be used to first pay-off this debt (and any other debts that are found). If money is left over, a new debt will be created between the sender and receiver.

Example 1.13 Big Bucks code *continued*

To create this application:

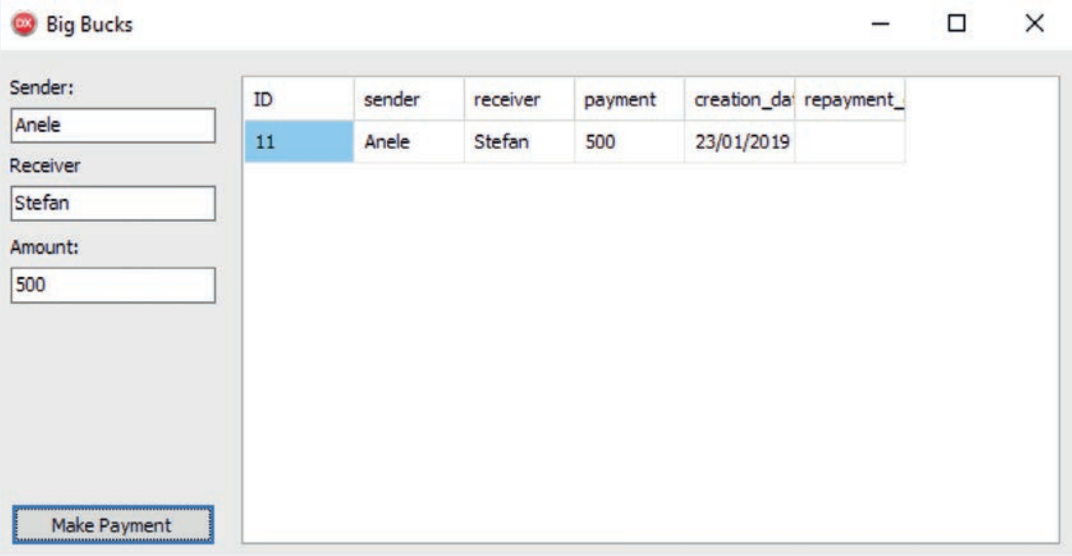
1. Open the project saved in your 01 – Big Bucks folder.
2. Save the values entered into the three text boxes into variables called *sSender*, *sReceiver* and *rRemainder*.
3. Add the following conditional statement to your code.

Appending condition

```
if rAmount > 0 then
begin
  dbmBigBucks.tblBigBucks.Append;
  dbmBigBucks.tblBigBucks['sender'] := sSender;
  dbmBigBucks.tblBigBucks['receiver'] := sReceiver;
  dbmBigBucks.tblBigBucks['payment'] := rAmount;
  dbmBigBucks.tblBigBucks['creation_date'] := Now;
  dbmBigBucks.tblBigBucks.Post;
end;
```

This conditional statement checks if the amount in *rAmount* is positive before adding the data to the database. While this condition may not be relevant right now, it will be used later to check if there is any money left after a repayment was made. It is important to remember that, when you add a record to a database, start with the Append (or Insert) function and end with the Post method. The lines in between these two statements specify the values that need to be added to the different database fields.

4. Save and test your application. You should now be able to add debts to the database.



Sender: Anele

Receiver Stefan

Amount: 500

ID	sender	receiver	payment	creation_da	repayment_
11	Anele	Stefan	500	23/01/2019	

The next step is to check if a payment can be made against an existing debt before any new debts are made. To do this:

5. Before you append the data, read the first record of the database. This can be done using the *dbmBigBucks.tblBigBucks.First* method.
6. Create a WHILE-DO loop that repeats while the database is not at the end of the file and while *rAmount* is larger than 0.
7. At the end of the WHILE-DO loop, use the table's *Next* method to move to the next record in the database. Without this line, the WHILE-DO loop may continue running forever.

Example 1.13 Big Bucks code *continued*

WHILE-DO-loop

```
dbmBigBucks.tblBigBucks.First;
while (not dbmBigBucks.tblBigBucks.Eof) and (rAmount > 0) do
begin
  // Check if an existing debt exists between the sender and receiver
  // Check if the existing debt is equal to the current payment
  // Check if the existing debt is larger than the current payment
  // Check if the existing debt is smaller than the current payment
  dbmBigBucks.tblBigBucks.Next;
  end;
```

8. Create a conditional statement inside the WHILE-DO loop to check if the payment is a repayment of debt.

   You know that the payment is a repayment if the database contains a record where the current sender was the receiver, while the current receiver was the sender. This means you need to read each record inside the while statement and see if this condition is met.

IF-THEN statement

```
if (sSender = dbmBigBucks.tblBigBucks['receiver']) and (sReceiver =
dbmBigBucks.tblBigBucks['sender']) then
begin
  // Store value of existing debt
  // Check if the existing debt is equal to the current payment
  // Check if the existing debt is larger than the current payment
  // Check if the existing debt is smaller than the current payment
end;
```

9. When a record is found where this condition is met, store the value of the *payment* field in a variable called *rExisting*.
10. Create a condition to check if *rAmount* is greater than or equal to *rExisting*.
11. Inside the condition, run *dbmBigBucks.tblBigBucks.Delete* method to delete the selected record.
12. Set the value of *rAmount* to equal *rAmount* minus *rExisting*.

Cancelling a debt

```
if rAmount >= rExisting then
begin
  dbmBigBucks.tblBigBucks.Delete;
  rAmount := rAmount – rExisting;
end;
```

Taking a look at this code, the condition checks if the amount paid is enough to fully repay the debt. If it is, the existing debt record is deleted and *rAmount* is adjusted to reflect the debt that has been repaid. If there is any money left in *rAmount*, the WHILE-DO loop will continue looking through the database to see if there are additional debts to repay. If there are no debts to repay, the WHILE-DO loop will exit. Since *rAmount* is still larger than 0, a new debt will be added to the database.

13. Save and test your application. You should now be able to remove debts by making a payment in the opposite direction.
14. Create a condition to check if *rExisting* is smaller than *rAmount*. This will mean that the debt is not fully repaid and needs to be adjusted based on the amount repaid.
15. Inside the conditional statement, run the command *dbmBigBucks.tblBigBucks.Edit* to edit the table.
16. Set the *payment* field of the table equal to *rExisting* minus *rAmount*.

Example 1.13 Big Bucks code *continued*

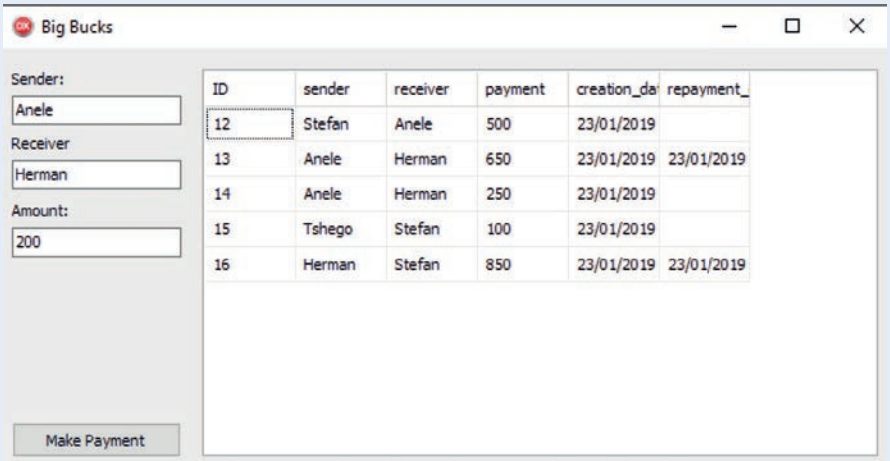
17. Set the *repayment_date* field equal to the date returned by the *Now* function.
18. Use the *Post* command to make these changes to the database.
19. Set *rAmount* equal to 0, since all the money has been used to repay the debt.

Partially repaying a debt

```
if rAmount < rExisting then
begin
  dbmBigBucks.tblBigBucks.Edit;
  dbmBigBucks.tblBigBucks['payment'] := rExisting – rAmount;
  dbmBigBucks.tblBigBucks['repayment_date'] := Now;
  dbmBigBucks.tblBigBucks.Post;
  rAmount := 0;
end;
```

Since *rAmount* will always be 0 for a partial repayment, the WHILE-DO loop will end once this payment is made and no additional changes will be made to the database.

20. Save and test your application. You should now be able to add debts, fully repay debts and partially repay debts.



Sender: Anele

Receiver: Herman

Amount: 200

ID	sender	receiver	payment	creation_da	repayment_
12	Stefan	Anele	500	23/01/2019	
13	Anele	Herman	650	23/01/2019	23/01/2019
14	Anele	Herman	250	23/01/2019	
15	Tshego	Stefan	100	23/01/2019	
16	Herman	Stefan	850	23/01/2019	23/01/2019

Congratulations, you just created an application that can easily store hundreds (or even thousands) of money transfers between people! To do this, you needed to create a database connection, add information to the database, read information from the database, and modify existing data on the database.

Databases will be covered in more detail in Chapter 4.

Activity 1.6

Answer the following questions using pen and paper.

1.6.1 Which two Delphi components are needed to connect to a database and database table?

1.6.2 Which Delphi component can be used to display the data from a database?

1.6.3 Write down Delphi commands to do the following:

a. Select the next record in a table.

b. Set the value of the database field *Name* to Kagiso.

c. Check if the database field *Number* has a value of 7.

d. Save changes made to the database.

e. Read each record in a table.

QUESTION 1

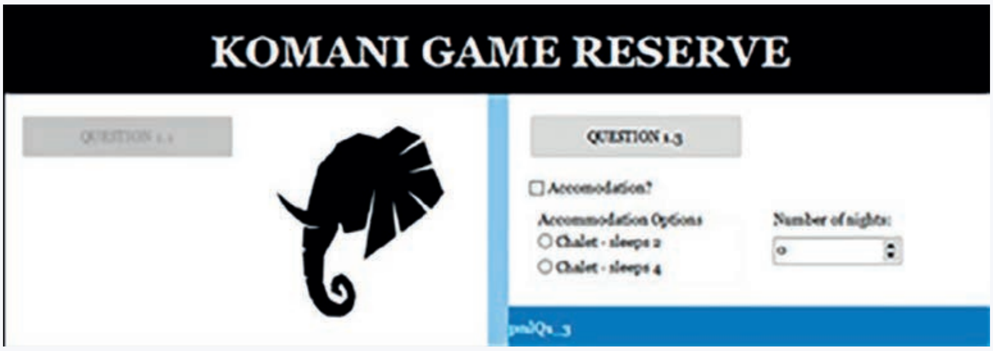
Komani Game Reserve in the Eastern Cape offers accommodation and the chance to see three of the 'Big Five' animals. Do the following:

- Open the incomplete program in the 01 – Question1 folder.
- Compile and execute the program. The program has no functionality currently.
- Follow the instructions below to complete the code for QUESTION 1.1 to QUESTION 1.5.

1.1 A picture file called elephant.png has been included in the root folder of the **Question1_p** Delphi project. Write code for the following:

- Display the text "KOMANI GAME RESERVE" on *pnlHeader*.
- Change the colour of *pnlHeader* to black.
- Change the font colour of *pnlHeader* to white.
- Display the **elephant.png** picture on the *imgQ1_1* component.
- Disable *btnQ1_1*.

Example output:


KOMANI GAME RESERVE

1.2 The game reserve is situated on a rectangular piece of land and the dimensions of the reserve are as follows:

- length: 40.4 km
- width: 27.8 km

The values for the length and width of the game reserve are stored in two global constants called *dReserveLength* and *dReserveWidth*, respectively.

Calculate the perimeter of the game reserve and display the result on *lblQ1_2A* in the following format:

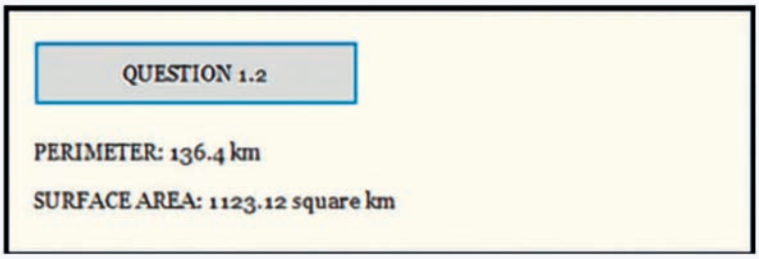
```
PERIMETER: <calculated perimeter> km
```

The surface area of a rectangle is calculated with the following formula: A = W × L

Calculate the surface area of the game reserve and display the result on *lblQ1_2B* in the following format:

```
SURFACE AREA: <calculated surface area> square km
```

Example output:



```
PERIMETER: 136.4 km
SURFACE AREA: 1123.12 square km
```

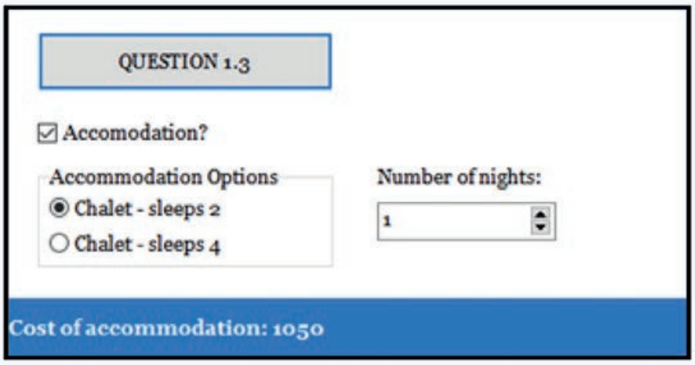
1.3 Visitors who want to spend more than 1 day in the reserve have a choice between the following two accommodation options:

OPTION	RATE
Chalet – sleeps two	R1 050.00
Chalet – sleeps four	R1 850.00

Write code for the following:

- Check whether the user has indicated that accommodation is required in the *cbxAccommodation* component.
- In case the user has indicated that accommodation is required, check which accommodation option was chosen in *rgpChaletOption* and extract the number of nights he/she needs accommodation for from sedNumNights.
- Calculate and display the cost of accommodation on *pnlQ1_3*. If the user did NOT indicate that accommodation is required, display the text "NO ACCOMMODATION" on *pnlQ1_3*.

Example output:



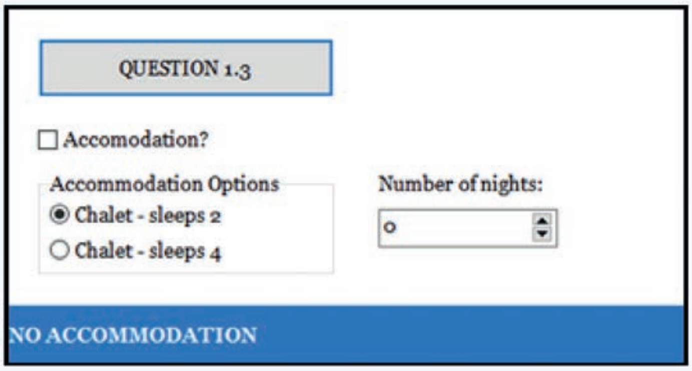
QUESTION 1.3

☑ Accomodation?

Accommodation Options
● Chalet - sleeps 2
○ Chalet - sleeps 4

Number of nights: 1

Cost of accommodation: 1050



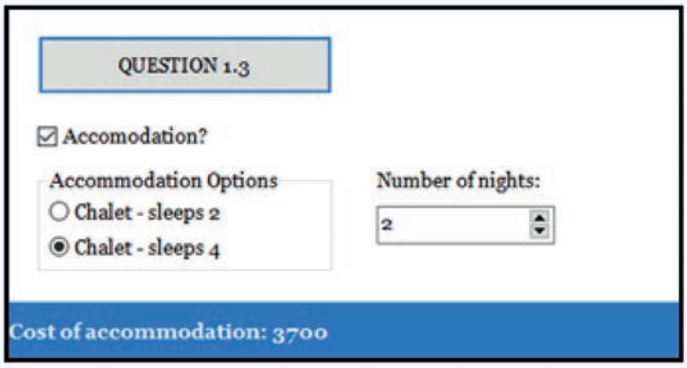
QUESTION 1.3

□ Accomodation?

Accommodation Options
● Chalet - sleeps 2
○ Chalet - sleeps 4

Number of nights: 0

NO ACCOMMODATION



QUESTION 1.3

☑ Accomodation?

Accommodation Options
○ Chalet - sleeps 2
● Chalet - sleeps 4

Number of nights: 2

Cost of accommodation: 3700

1.4 The game reserve hosts three of the Big Five. These are elephants, buffalo and lions. Visitors are able to report sightings, which give other visitors an idea of which animals there are to see on a given day. Sightings are compiled into a string of keys (single characters) that indicate which animals have been sighted.

Each key in the string represents the following animals:

KEY	ANIMAL
B	Buffalo
E	Elephant
L	Lion

Buffalo and elephant sightings are common, but lion sightings are rare.

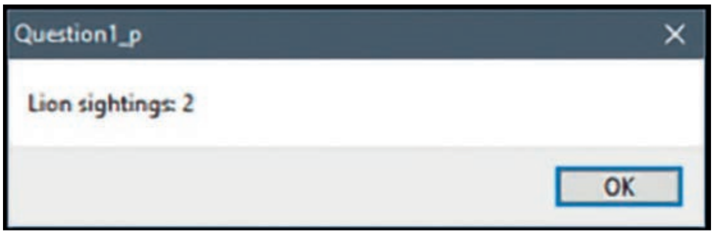
Write code for the following:

- Extract the sightings from *edtSightings*.
- Use the string of keys to create a list of animals sighted and display this list on *redQ1_4*.
- Count the number of lion sightings and display the result on a message box in the following format:

```
Lion sightings: <number of lion sightings>
```

Example of output if EEBEBEEBLEELB was entered as the sightings string:

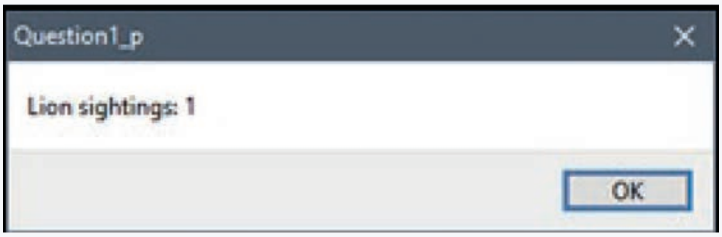
```
Elephant
Elephant
Buffalo
Elephant
Buffalo
Elephant
Elephant
Buffalo
Lion
Elephant
Elephant
Lion
Buffalo
```



Lion sightings: 2

Example of output if EBBEBL was entered as the sightings string:

```
Elephant
Buffalo
Buffalo
Elephant
Buffalo
Lion
```



Lion sightings: 1

1.5 Potential visitors need to be able to make a reservation. A reference code needs to be generated for each reservation.

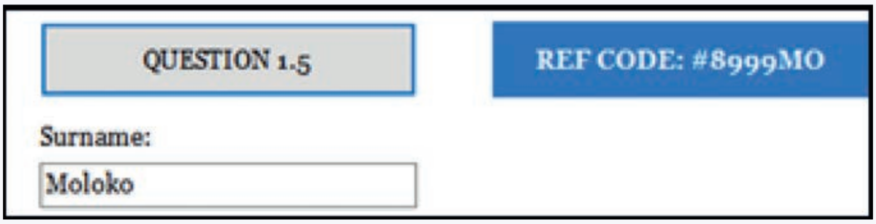
Reference codes are compiled as follows:

- The first part of the reference code is a hash-symbol (#).
- The second part of the reference code is a random number ranging from 1000 to 9999 (both inclusive).
- The third part of the reference code consists of the first two letters of the visitor's surname. Both letters are in UPPER CASE.

Write code for the following:

- Test if the visitor's surname is entered in the *edtSurname* component. If *edtSurname* is empty, display an appropriate error message and set the focus to *edtSurname*.
- If the visitor's surname is entered, extract it from *edtSurname* and generate a reference code.
- Display the reference code on *pnlQ1_5*.

Example output:



REF CODE: #8999MO

Surname: Moloko

QUESTION 2 Database manipulation

This section consists of two questions. The following important notes are applicable to both questions:

- You are NOT allowed to modify or add to the supplied data in any way.
- Good programming techniques must be applied when coding your solutions.
- NO marks will be assigned for hardcoding. Use control structures and variables where necessary.
- NO FILTERS MAY BE USED.

SCENARIO:

The HealthActive gym is currently running a healthy living program where a member's health status is captured and checked.

The **Health.mdb** database contains one table called *Members*.

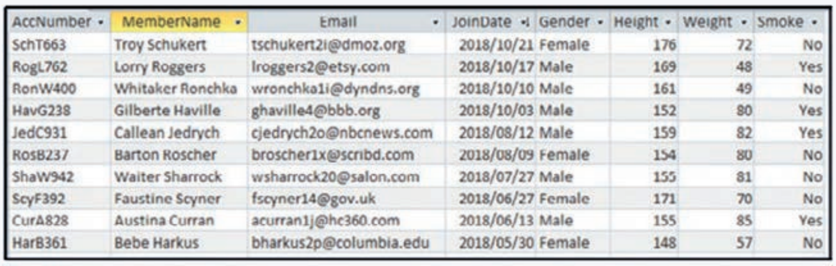
The *Members* table is structured with the following fields:

FIELD	DATA TYPE	DESCRIPTION
AccNumber	Text	A unique account number that consists of the first 3 letters of the member's surname, followed by the member's initial and a random 3-digit number.
MemberName	Text	Contains a member's name and surname.
Email	Text	Contains a member's email address
JoinDate	Date	Contains the date (YYYY/MM/DD) on which the member has joined the health program.
Gender	Text	Describes a member's gender as Male or Female.
Height	Number	Contains a member's height in centimeters.
Weight	Number	Contains a member's weight in kilograms.
Smoke	Boolean	Describes whether the member is a smoker(Yes) or non-smoker (No).

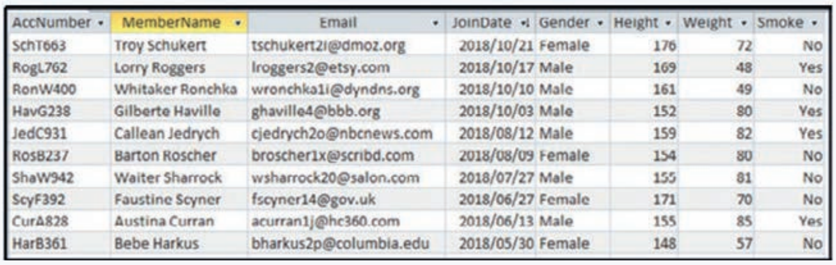
Take note

- Connection code has been provided.
- When the btnDBRestore button is clicked, the data in the database will be restored to the original data.
- The name of the table to be used in your code must be tblMember, which is the TADOTable object connected to the database.

Example data from the *Members* table:



AccNumber	MemberName	Email	JoinDate	Gender	Height	Weight	Smoke
SchT663	Troy Schukert	tschukert2i@dmoz.org	2018/10/21	Female	176	72	No
RogL762	Lorry Roggers	lroggers2@etsy.com	2018/10/17	Male	169	48	Yes
RonW400	Whitaker Ronchka	wronchka1i@dyndns.org	2018/10/10	Male	161	49	No
HavG238	Gilberte Haville	ghaville4@bbb.org	2018/10/03	Male	152	80	Yes
JedC931	Callean Jedrych	cjedrych2o@nbcnews.com	2018/08/12	Male	159	82	Yes
RosB237	Barton Roscher	broscher1x@scribd.com	2018/08/09	Female	154	80	No
ShaW942	Waiter Sharrock	wsharrock20@salon.com	2018/07/27	Male	155	81	No
ScyF392	Faustine Scyner	fscyner14@gov.uk	2018/06/27	Female	171	70	No
CurA828	Austina Curran	acurran1j@hc360.com	2018/06/13	Male	155	85	Yes
HarB361	Bebe Harkus	bharkus2p@columbia.edu	2018/05/30	Female	148	57	No



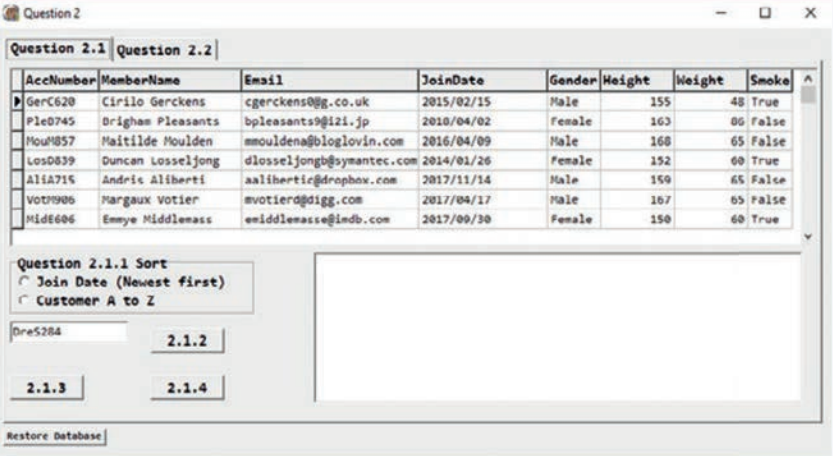
AccNumber	MemberName	Email	JoinDate	Gender	Height	Weight	Smoke
SchT663	Troy Schukert	tschukert2i@dmoz.org	2018/10/21	Female	176	72	No
RogL762	Lorry Roggers	lroggers2@etsy.com	2018/10/17	Male	169	48	Yes
RonW400	Whitaker Ronchka	wronchka1i@dyndns.org	2018/10/10	Male	161	49	No
HavG238	Gilberte Haville	ghaville4@bbb.org	2018/10/03	Male	152	80	Yes
JedC931	Callean Jedrych	cjedrych2o@nbcnews.com	2018/08/12	Male	159	82	Yes
RosB237	Barton Roscher	broscher1x@scribd.com	2018/08/09	Female	154	80	No
ShaW942	Waiter Sharrock	wsharrock20@salon.com	2018/07/27	Male	155	81	No
ScyF392	Faustine Scyner	fscyner14@gov.uk	2018/06/27	Female	171	70	No
CurA828	Austina Curran	acurran1j@hc360.com	2018/06/13	Male	155	85	Yes
HarB361	Bebe Harkus	bharkus2p@columbia.edu	2018/05/30	Female	148	57	No

Do the following:

- Compile and execute the program in the 01 – Question2 folder. The program currently has limited functionality
- Complete the code for each question as described in [Question 2.1] and [Question 2.2].
- The program contains a graphical user interface with two-tab sheets labelled [*Question 2.1*] and [*Question 2.2*].

2.1 Data processing

Select tab sheet [*Question 2.1*], which displays the following user interface when the program is run:



AccNumber	MemberName	Email	JoinDate	Gender	Height	Weight	Smoke
GerC620	Cirilo Gerckens	cgerckens0@g.co.uk	2015/02/15	Male	155	48	True
PleD745	Brigham Pleasants	bpleasants9@121.jp	2010/04/02	Female	163	86	False
MouM857	Maitilde Moulden	mmouldena@bloglovin.com	2016/04/09	Male	168	65	False
LosD839	Duncan Losseljong	dlosseljongb@symantec.com	2014/01/26	Female	152	60	True
AliA715	Andris Aliberti	aalibertic@dropbox.com	2017/11/14	Male	159	65	False
VotM906	Margaux Votier	mvotierd@digg.com	2017/04/17	Male	167	65	False
MidE606	Emmye Middlemass	emiddlemasse@imdb.com	2017/09/30	Female	150	60	True

Question 2.1.1 Sort
○ Join Date (Newest first)
○ Customer A to Z

DreS284

Complete the code to meet the requirements specified in QUESTION 2.1.1 to QUESTION 2.1.3

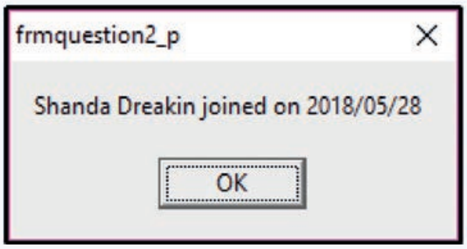
2.1.1 Radiogroup [2.1.1]

Write code for the OnClick event of the radiogroup that will sort the contents of the *Members* table and display it on the database grid called *dbgTable* as follows:

- Radiobutton 1: From newest to oldest according to the join date.
- Radiobutton 2: Ascending order of customer names and surnames.

2.1.2 The user will be asked to enter an account number. Write code to search the database table to check if the member's account number appears in the table. If it is found, the date on which the member joined the program must be displayed in the following format:

Example of output if Account number: *DreS284* is typed into *edtSearch*.



Shanda Dreakin joined on 2018/05/28

If the record is not found, a suitable message must be displayed.

2.1.3 A person's body mass index (BMI) is a measure for indicating nutritional status in adults. It is defined as a person's weight in kilograms divided by the square of the person's height in metres (kg/m^2). For example, an adult who weighs 70 kg and whose height is 1.75 m will have a BMI of 22.9.

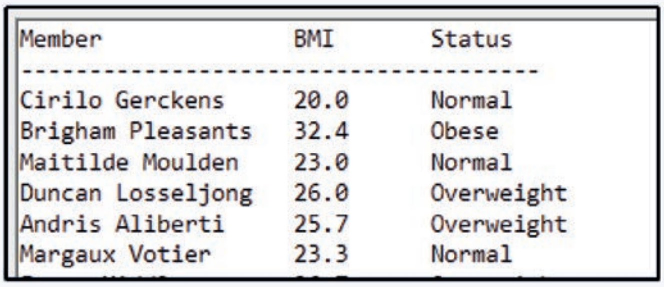
70 (kg)/1.752 (m^2) = 22.9 BMI

For adults over 20 years old, BMI falls into one of the following categories:

BMI	NUTRITIONAL STATUS
Below 18.5	Underweight
18.5 – 24.9	Normal
25.0 – 29.9	Overweight
30.0 or higher	Obese

Write code to display a list of all members their BMI rounded to 1 decimal and their respective nutritional health status, neatly in columns on the *redOutput*:

Example output:



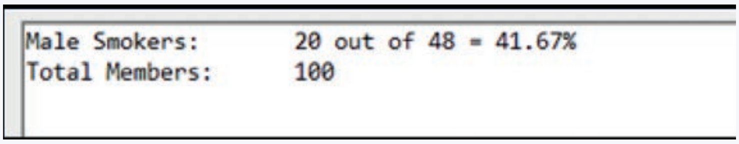
```
Member              BMI       Status
----------------------------------------
Cirilo Gerckens     20.0      Normal
Brigham Pleasants   32.4      Obese
Maitilde Moulden    23.0      Normal
Duncan Losseljong   26.0      Overweight
Andris Aliberti     25.7      Overweight
Margaux Votier      23.3      Normal
```

2.1.4 Write code to calculate and display on the *redOutput* which percentage of male members are smokers.

Display:

- the number of male members who are smokers
- the percentage of the male members who are smokers rounded to two decimals places
- the total number of members in the table.

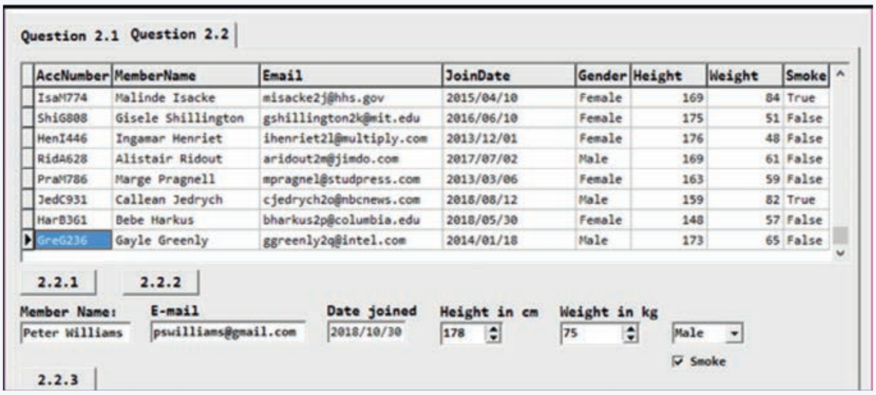
Example of output:



```
Male Smokers:       20 out of 48 = 41.67%
Total Members:      100
```

2.2 Data maintenance

Select tab sheet [*Question 2.2*], which displays the following user



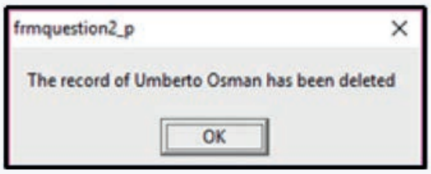
AccNumber	MemberName	Email	JoinDate	Gender	Height	Weight	Smoke
IsaM774	Malinde Isacke	misacke2j@hhs.gov	2015/04/10	Female	169	84	True
ShiG808	Gisele Shillington	gshillington2k@mit.edu	2016/06/10	Female	175	51	False
HenI446	Ingamar Henriet	ihenriet2l@multiply.com	2013/12/01	Female	176	48	False
RidA628	Alistair Ridout	aridout2m@jimdo.com	2017/07/02	Male	169	61	False
PraM786	Marge Pragnell	mpragnel@studpress.com	2013/03/06	Female	163	59	False
JedC931	Callean Jedrych	cjedrych2o@nbcnews.com	2018/08/12	Male	159	82	True
HarB361	Bebe Harkus	bharkus2p@columbia.edu	2018/05/30	Female	148	57	False
GreG236	Gayle Greenly	ggreenly2q@intel.com	2014/01/18	Male	173	65	False

Member Name: Peter Williams
E-mail: pswilliams@gmail.com
Date joined: 2018/10/30
Height in cm: 178
Weight in kg: 75
Male
☑ Smoke

Complete the code to meet the requirements specified in QUESTION 2.2.1 to QUESTION 2.2.3.

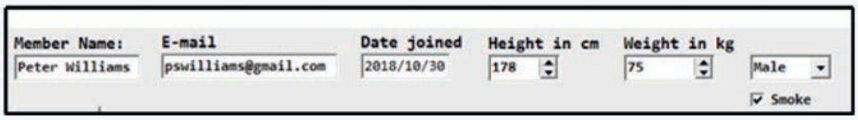
2.2.1 The email address for the member with member number PraM786 has been incorrectly captured as mpragnell2n@studiopress.c. Add code to correct the email address to mpragnel@studpress.com

2.2.2 You want the user to be able to delete the record that he/she selects. Write the code to delete the current record from the table. Keep track of the name of the member that will be deleted. Display a message once the record has been deleted, as follows (if the third record in the table was selected):



The record of Umberto Osman has been deleted

2.2.3 The details of a new member have been supplied in the appropriate components. Complete the code to insert this member into a new record in the table. The account number must be constructed using the first 3 letters of the surname, the member's initial and any random 3 digit number (between 100 and 999).



Member Name: Peter Williams
E-mail: pswilliams@gmail.com
Date joined: 2018/10/30
Height in cm: 178
Weight in kg: 75
Male
☑ Smoke

QUESTION 3

SCENARIO

You have been asked by the organisers of a Gymnastics competition in Strand to complete a program to manipulate the numbers of learners from 10 schools in the vicinity who are taking part.

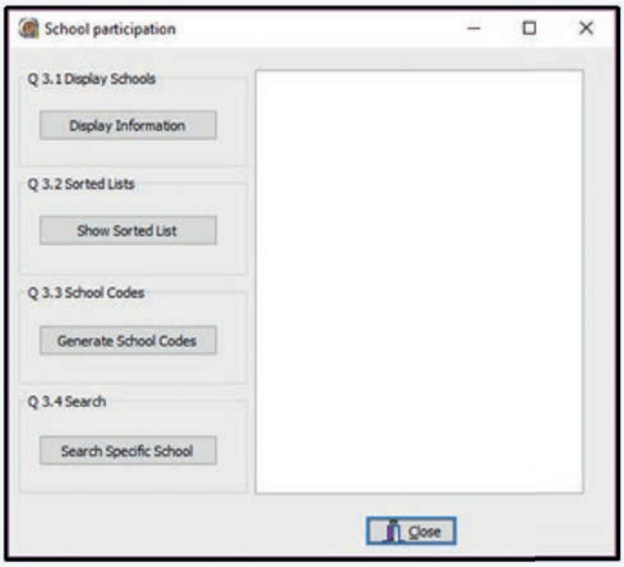
Open the project in the folder 01 – Question 3 folder.

- The incomplete main form unit called **Question3_u.pas**
- Currently the program has no functionality.

An incomplete form class *Question3_u* is provided with the following graphical user interface:

CONSOLIDATION ACTIVITY Chapter 1: Programming fundamentals *continued*

Complete the code for each section of QUESTION 3 as specified in QUESTION 3.1 to QUESTION 3.4 below.



In the given program the following two global parallel arrays have been created:

- a String array called *arrSchools* with the names of the ten schools that have registered to take part.
- an Integer array called *arrNumGymnasts* with the number of the learners from each school who will be competing.

3.1 When this button is clicked a call to the **DisplayArrays** procedure is made.

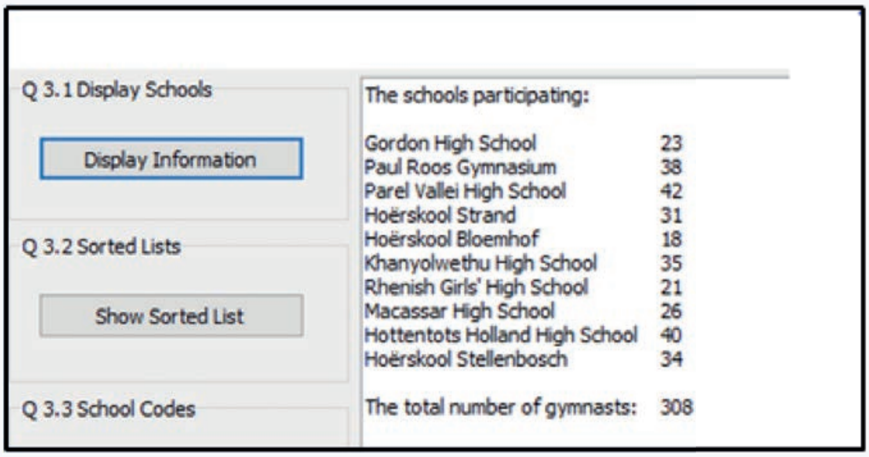
Do the following:

- Create a procedure called **DisplayArrays** that will receive the heading as a parameter.

In the implementation of the procedure:

- display the heading as indicated in the parameter
- loop through the two given arrays and display the information in two columns in the rich edit *redOutput*, as indicated in the *btnDisplayClick* event handler
- Call the procedure and use the heading "The schools participating" in the parameter to display the information.
- Add code to determine and display the total number of gymnasts.

Example output:

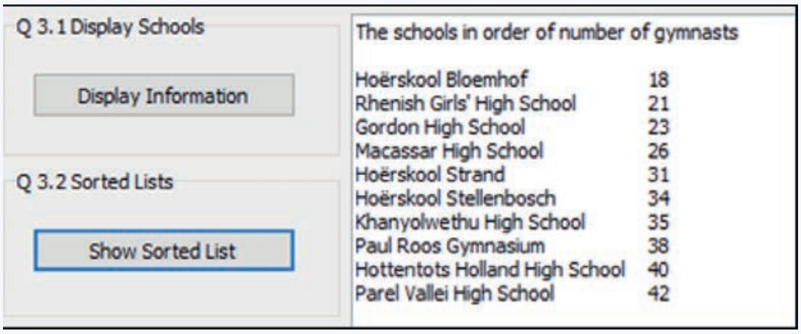


```
The schools participating:

Gordon High School             23
Paul Roos Gymnasium            38
Parel Vallei High School       42
Hoërskool Strand               31
Hoërskool Bloemhof             18
Khanyolwethu High School       35
Rhenish Girls' High School     21
Macassar High School           26
Hottentots Holland High School 40
Hoërskool Stellenbosch         34

The total number of gymnasts:  308
```

3.2 The organisers want a list of the schools with the number of gymnasts sorted from lowest to highest as indicated in the screen shot below. Write the code to sort the arrays to achieve this.

Use the **DisplayArrays** procedure with the appropriate heading as a parameter to display the sorted list.

Example output:



```
The schools in order of number of gymnasts

Hoërskool Bloemhof            18
Rhenish Girls' High School    21
Gordon High School            23
Macassar High School          26
Hoërskool Strand              31
Hoërskool Stellenbosch        34
Khanyolwethu High School      35
Paul Roos Gymnasium           38
Hottentots Holland High School 40
Parel Vallei High School      42
```

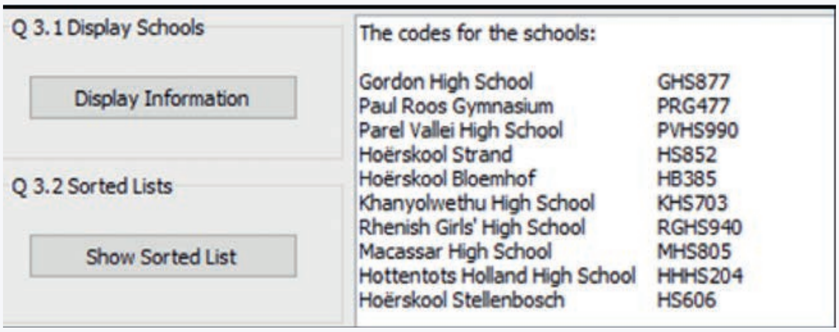
3.3 The organisers want a code for each of the schools. The code consists of the first letter of each of the words of the school's name and a random number in the range 100 to 999 (both included), for example, RGHS367 for Rhenish Girls' High School.

Do the following:

- Create a function called **GenerateCode** that will receive the **name of the school** as a parameter and return the generated code.

In the implementation section of the function extract the first letter from each word in the name of the school and then add a randomly generated number between 100 and 999 (both included).

In the *btnCodeClick* event handler loop through the array called *arrSchools*, call the **GenerateCode** function for each of the schools in the list and display the generated codes as indicated in the screenshot below:



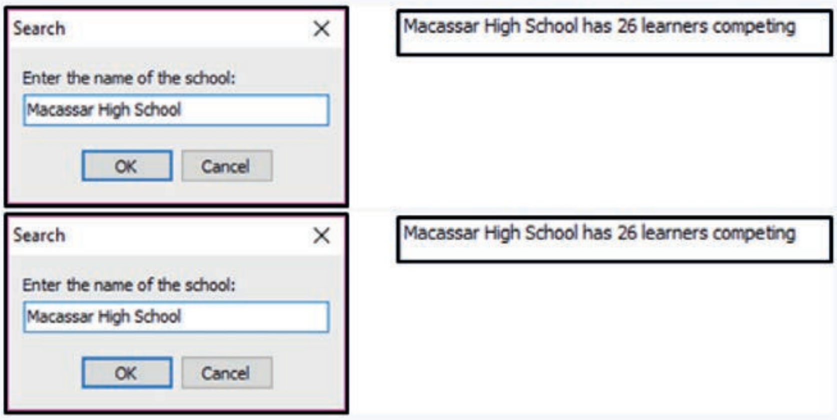
```
The codes for the schools:

Gordon High School             GHS877
Paul Roos Gymnasium            PRG477
Parel Vallei High School       PVHS990
Hoërskool Strand               HS852
Hoërskool Bloemhof             HB385
Khanyolwethu High School       KHS703
Rhenish Girls' High School     RGHS940
Macassar High School           MHS805
Hottentots Holland High School HHHS204
Hoërskool Stellenbosch         HS606
```

Take note

- Your code should provide for the fact that each user might type the name of the school in a different case (lower and/or upper case)
- If the name of the school was not found, an appropriate message should be displayed in the rich edit.

3.4 You want the user to type in the name of a school and then determine and display how many learners from that school are competing. Use an *Inputbox* to get the name of the school to search for, and then display the number of learners competing in *redOuput*, as indicated in the screenshot below.

Example output:



Search

Enter the name of the school: Macassar High School

```
Macassar High School has 26 learners competing
```

Search

Enter the name of the school: Macassar High School

```
Macassar High School has 26 learners competing
```

QUESTION 4

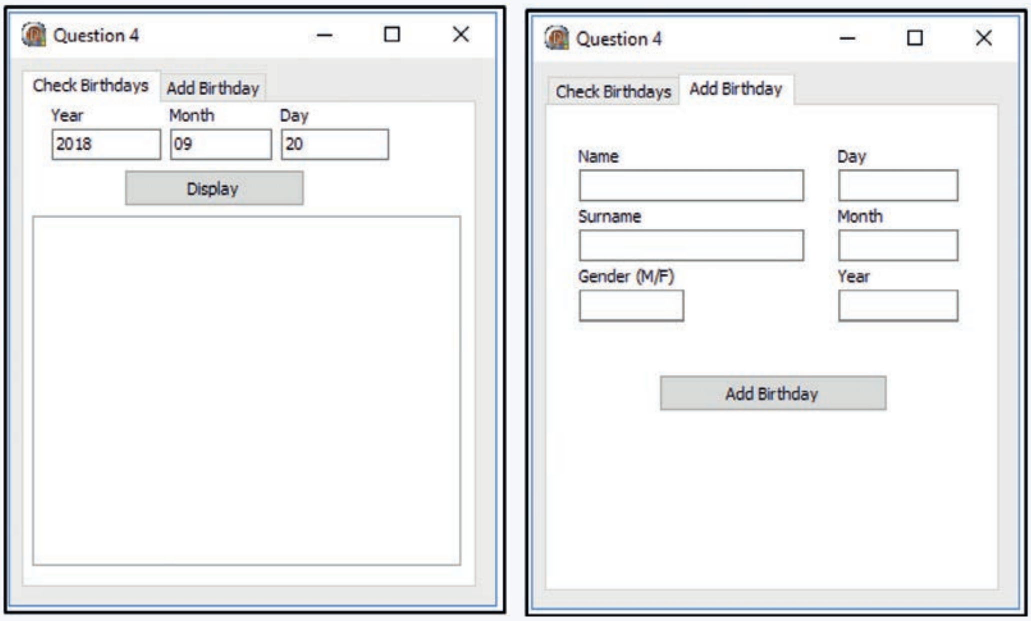
SCENARIO:

Your principal asked you as an IT student to help write a program which will determine whether anyone in your school has a birthday on any given day. He wishes to use the program every morning before school to check whose birthdays he has to announce today.

Do the following:

- Compile and execute the program in the 01 – Question 4 folder.
- Complete the code for each question, as described in QUESTION 4.1 to QUESTION 4.3.

Supplied GUI:



Year: 2018 Month: 09 Day: 20

4.1 Add code to the On Create event handler of the form to do the following:

4.1.1 Display the current (today's) date in the edit boxes provided

4.1.2 Check if the file, **Birthdays.txt** exists. Display a suitable message if the file does not exist.

The text file contains data in the following format:

```
Name,Surname,Gender,Year-Month-Day
```

Example of the first 5 lines:

```
Noluvo,Mdlungwana,F,2001-07-12
Laureka,Wallace,F,2000-03-01
Michelle,van Heerden,F,2001-05-23
Maria,Kok,F,2000-09-19
Brewster,Attew,M,2001-12-09
```

4.2 Work on the [Check Birthdays] tab sheet. Add code to the event handler of *btnDisplay*.

4.2.1 Create suitable variables for each component of a line from the text file

4.2.2 Open the **Birthdays.txt** file for reading. Assign this file to the given global text file variable, *tfBirthdays*.

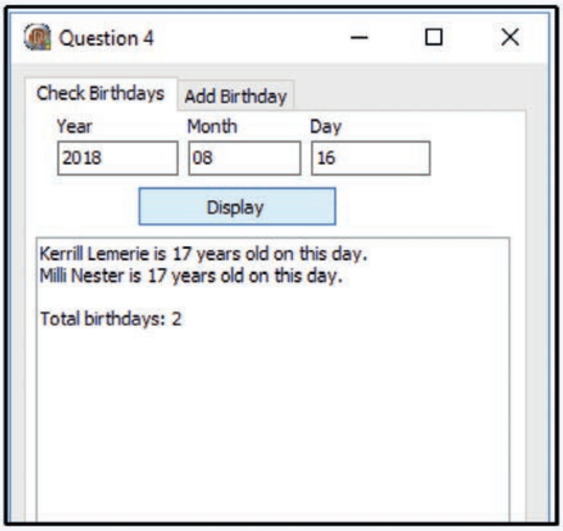
4.2.3 Ensure that all the data is removed from *redBirthdays* when the button is clicked.

4.2.4 Make use of variables to store the current year, month and day. Ensure that the program will still work if any other day of the year is entered.

4.2.5 Loop through the text file and determine if any of the entries are on the date entered in the edit boxes. Display the name, surname and age of the records that match.

4.2.6 Determine the total number of birthdays for this day and display it with the output.

Example output: (for the date 2018/08/16)

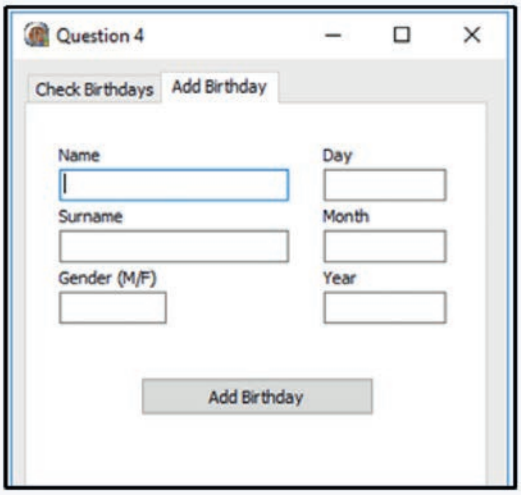


Year: 2018
Month: 08
Day: 16

```
Kerrill Lemerie is 17 years old on this day.
Milli Nester is 17 years old on this day.

Total birthdays: 2
```

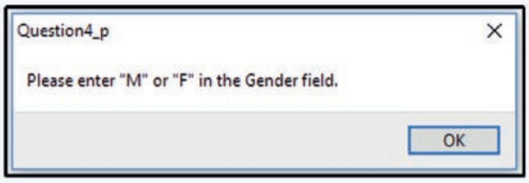
CONSOLIDATION ACTIVITY Chapter 1: Programming fundamentals *continued*

4.3 Study the interface for the [AddBirthday] tab sheet



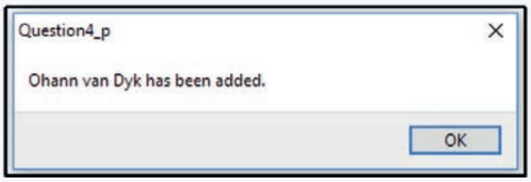
4.3.1 Write code for the [Add Birthday] button to extract all the information from the components.

Add validation to check that only ‘M’ or ‘F’ was added to *edtGender*. Display a message if anything else was added.



Please enter "M" or "F" in the Gender field.

4.3.2 Once the button is clicked, the student’s information has to be added to the text file, and a suitable message has to be displayed.



Ohann van Dyk has been added.

QUESTION 5

5.1 Which type of loop can be used to read each entry in an array?

a. CASE

b. FOR-loop

c. REPEAT-loop

d. WHILE-DO loop

5.2 Determine which of the following are objects, properties or events.

EXAMPLE	TYPE
TButton	
Top	
Caption	
OnClick	
TADOQuery	

5.3 Give an example of a binary variable.

5.4 Name ONE way of preventing programming errors.

5.5 A user is required to input a FOUR-character security code which contains ONE alphabetical character and THREE digits. The first character must be alphabetical, for example "D845".

Write an algorithm, using pseudocode, to validate the code for the correct format once the code has been entered. **NOTE:** Use at least ONE loop must be part of your solution.

OBJECT-ORIENTED PROGRAMMING

CHAPTER 2

CHAPTER UNITS	
Unit 2.1	Defining a user-defined class
Unit 2.2	Using the class

Learning outcomes

At the end of this chapter you should be able to

- describe the class as a data type
- discuss the different access specifiers
- describe attributes and methods as part of a class
- define a class:
  - add attributes to a class
  - declare and implement methods in a class
- instantiate objects of the class
- use the object in your application.

INTRODUCTION

In this chapter we will look at **Object-Oriented Programming (OOP)** in more detail. To understand OOP and user-defined classes, think about your national ID card. Each South African citizen is issued with an ID smart card using an ID blueprint.

REPUBLIC OF SOUTH AFRICA
NATIONAL IDENTITY CARD

Surname:
ABCDEFGHIJKLMNOPQRSTUVXZY
Names:
ABCDEFGHIJKLMNOPQRSTUVXZY
Sex:
M
Nationality:
RSA
Identity Number:
0123456789012
Date of Birth:
16 JUL 1969
Country of Birth:
RSA
Status:
CITIZEN
Signature

Conditions:
This card has been issued by the Department of Home Affairs in terms of the Identification Act, Act 68 of 1997 (Amended)

Date of Issue: 2013-00-30

RSA

For verification or enquiry purposes contact (012) xxx xxxx or (012) xxx xxxx
If found please return to the Department of Home Affairs

123456789

Figure 2.1: *An ID card is like a custom class*

Example of some of the details captured for your ID:

- photo
- name
- surname
- identity number
- nationality
- date of birth

We refer to these data items as fields.

OOP IN DELPHI

https://www.youtube.com/watch?v=gRUvQgIZ5jl

Watch out!

We use analogies to describe the concept of an object; don't get too attached to these analogies. They only serve to illustrate a point. Instead, concentrate on drawing out the abstract nature of an object as we apply an everyday term to a programming technique.

New words

Object-Oriented Programming (OOP) – refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure

New words

attributes – the data fields of the class

behaviour – the code that provides the interaction with the attributes

instances of the class – a data type that describes the attributes and behaviour of the object to be model electronically

encapsulation – the grouping of attributes and behaviour in one entity

The ID smart card can be:

- scanned to transfer the data on it
- presented to verify the person's identity.

We say the fields used to gather data are **attributes** of the card and what the card can be used for is the **behaviour** of the card.

The attributes and behaviour form a blueprint for creating an ID smart card. We refer to the blueprint as a **class**. The **instance** of a class is referred to as an **object**.

From the one blueprint each South African citzen can get an Identity Smart Card i.e many **Instances of the class** or card objects can be created. Like an array a class can have many data items, however whereas an array can only have data Items of the same type whereas a class can have data items of different data types. As a record in a database has data items of different types, so does the data items of a class. The grouping of data together with the functions to interact with the data is called **encapsulation**.

CLASS	A data type that describes the attributes and behaviour of the object to be model electronically
OBJECT	An instance of the class
ATTRIBUTES	The data fields of the class
BEHAVIOUR	The code that provide the interaction with the attributes
ENCAPSULATION	The grouping of attributes and behaviour in one entity

OOP-techniques enable programmers to:

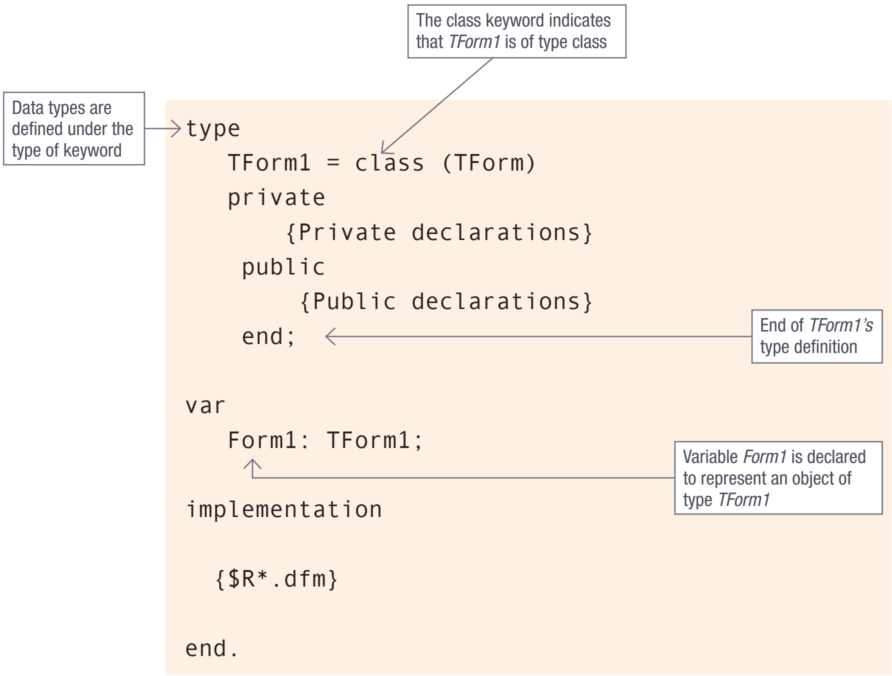
- create independent modules that are not influenced by other parts of the program
- create reusable code
- improve the integrity of a program and its data.

In this chapter you will learn how to declare classes, implement its methods, instantiate an object of the class and use the object's methods in an application.

UNIT

2.1 Defining a user-defined class

After the introduction, you might think that classes are a brand-new concept that you will have to learn from scratch. Fortunately this is not the case. You have been using multiple classes in every single program you have created, like the TForm class. If you create a new Delphi project and open the code, you will see the following lines near the top of your code.



```
type
   TForm1 = class (TForm)
   private
       {Private declarations}
    public
        {Public declarations}
    end;

var
   Form1: TForm1;

implementation

  {$R*.dfm}

end.
```

The class keyword indicates that *TForm1* is of type class
Data types are defined under the type of keyword
End of *TForm1's* type definition
Variable *Form1* is declared to represent an object of type *TForm1*

Figure 2.2: *TForm1 class*

By looking at the declaration of the *TForm1* class above, you can see the *basic structure* of a class definition, using the key words: *Type*, *class*, *private*, *public* and *end*. These are the essential keywords for declaring a class and *implementation* to provide it with behaviour.

- *TForm1* has *TForm* as its base class.
- The keyword, *private*, marks the section where the attributes and methods are declared, that should not be accessible from outside the class.
- The keyword, *public*, marks the section where we declare methods that serves as the interface to the internal features of the class. These methods are accessible from outside the class.
- The *implementation* section is where all the events, procedures and functions are placed. Any active code that you want to execute should be placed in this section.

USER DEFINED OBJECTS

https://www.youtube.com/watch?v=0uMu4dgJZ8M

Take note

Notice how the *Form1* variable is declared to be of type *TForm1*. An instance of class *TForm1*, also called an object, can be assigned to *Form1*.

DECLARING A CLASS

The code block below shows the basic structure of a class declaration.

Class declaration syntax

```
type
ClassName = class(optional BaseClass)
 Private
//declare attributes and private methods here
 Public
//declare public methods here
 End;
```

In order to understand whether to declare a public or a private method, you need to understand how **access specifiers** work. As their name suggests, access specifiers tell your class which methods should be visible (or accessible) to units outside the class, and which methods should only be visible inside the class.

New words

Access specifiers – is a defining code element that can be determine which elements of a program are allowed to access a specific variable or other piece of data

There are four types of access specifiers:

ACCESS SPECIFIER	DESCRIPTION
Private	Private attributes and methods can only be accessed from inside this class.
Protected	Protected attributes and methods can only be accessed by this class or any classes based on this class.
Public	Public methods can be accessed in any program where this class is used.
Published	Published attributes and methods are very similar to public attributes and methods, but may be changed from the RAD Studio's Object Inspector.

Class attributes or fields are only declared in the private section of the class. This data hiding ensures that access is only given to class members and protects the integrity of the data.

Once the attributes and methods have been declared, you need to write code to implement the methods. This is done in the *implementation* section. Once the methods are implemented your class definition is complete and ready to be tested.

Activity 2.1

Answer the following questions in your own words:

2.1.1 What is a class?

2.1.2 What is an object?

2.1.3 Give an example of two classes that you have used in your programs.

2.1.4 Write down the syntax for declaring a class.

2.1.5 What is the difference between the public and private keywords?

Activity 2.1 *continued*

2.1.6 Why should class attributes be private and not public?

2.1.7 Find the errors in the following basic class structure:

```
implementation
 className : class(optional BaseClass);
 Private
Attribute1: String;
Method1;
 Public
Attribute2: Integer;
Method2;
```

To define a class in Delphi, you need to take the following steps:

- create a new unit file that will only contain the code for the class
- in the TYPE section name the class
- add attributes and methods in the PRIVATE section
- add methods in the PUBLIC section
  - constructor method
  - accessor (or getters) methods
  - mutator (or setters) methods
  - other auxiliary/helper methods
  - *ToString* method for quick access to the state of an object.

Read the following case study, which will be used throughout this unit to look at each of the steps above.

Case Study Second-hand phones

Imagine you are building a web application that facilitates buying and selling second-hand phones. Once the site is up and running, you expect there to be hundreds of users selling and buying phones at any time. You will recognise a phone as an entity similar to the ID-card. Each phone has a number of attributes you need to record, including the phone's brand, the phone's model number, the seller's price and the phone's date of purchase. How would you store this information?

The best way to store this information is to encapsulate it in a custom class named *TPhone*.

Your *TPhone* class might have the following essential properties:

- Brand: String
- Model: String
- Owner: String
- PurchaseDate: TDateTime
- Price: Double

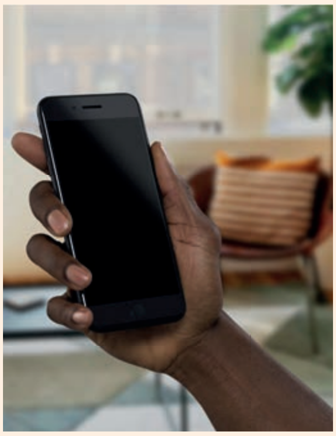


Figure 2.3: *How would you store the data needed to sell phones?*

We could also consider a method to work out the age of the phone. So the buyer can get a quick feel about the value for money of the offer.

An object of the type *TPhone* can store the brand, model, owner, *purchaseDate* and price, together with methods to access and manipulate it, such as a method that is able to return the age of the phone.

Once you have defined the *TPhone* class, you can create one or more *TPhone* objects in your application (or even a list or array of *TPhone* objects). Each of these objects will contain a unique set of information and will represent a phone.

CLASS UNIT FILE

The first step in creating your new application that facilitates buying and selling second hand phones (see Case study above) is to create a new unit file for the new class. Just as a data module separates your data from any specific form, the unit file separates the class from a form. This allows you to use the class in multiple forms in a multi-form application and even in other applications. By separating the class from the form, you keep the code clean, making it easier to keep track of your code and reduce the repetition of code.

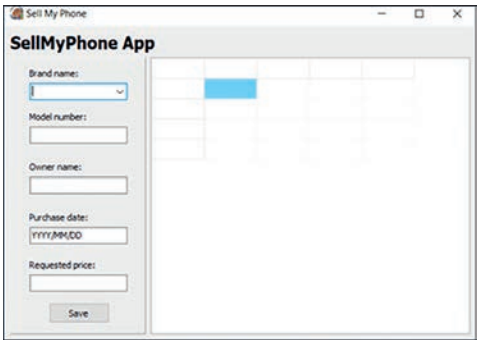
In order to create an object to be used in your application, a design specification for your model needs to be written. This is then captured in a class diagram. The class diagram lists the attributes and behaviour for all objects of the class.

CREATING A CLASS UNIT FILE

Open the project saved in the sellMyPhone folder.

When you open the project, you should see the following user interface:

The white box on the right is a *TStringGrid* component from the Grids list. You are now ready to create the class unit file for your project.

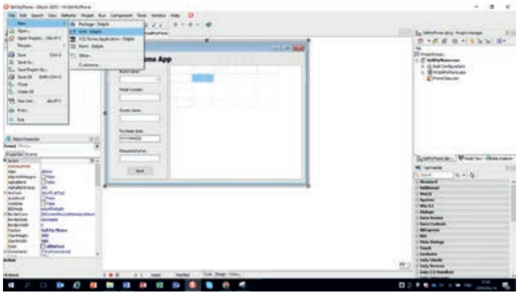


1. In RAD Studio, open the *File* Menu and select the *New* option.

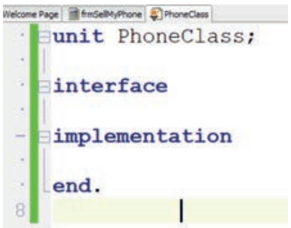
Take note

Your image might differ slightly, depending on your version of RAD Studio.

2. Select the *Unit* option. You will see a new unit appear in the Project Manager panel in the top right corner.

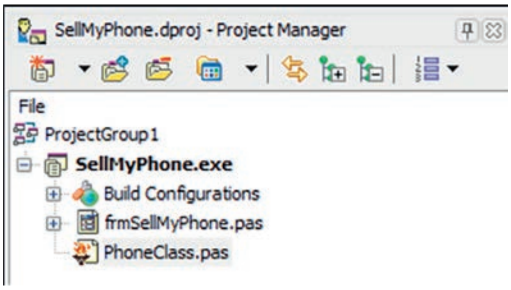


3. Now press <ctrl+shift+s> (save all), when prompted name your Unit “PhoneClass” and save the unit in your project folder named “SellMyPhone”.



```
unit PhoneClass;

interface

implementation

end.
```

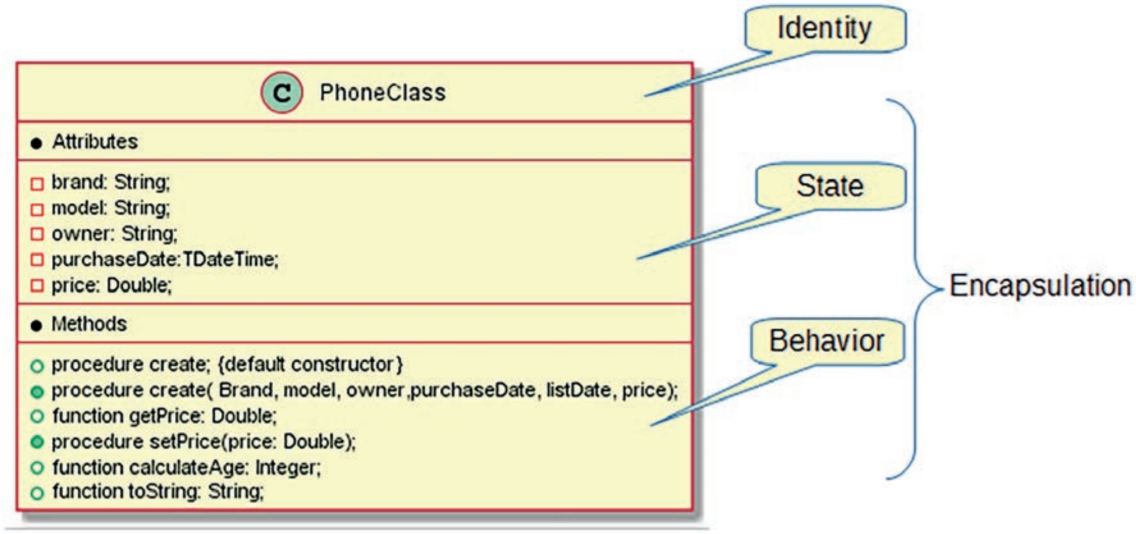
The Pascal source file, PhoneClass.pas, for the newly created unit will appear in the project manager, under the form class.



SellMyPhone.dproj - Project Manager

File
ProjectGroup1
SellMyPhone.exe
Build Configurations
frmSellMyPhone.pas
PhoneClass.pas

In the case of our *PhoneClass*, the formal class diagram looks like this:


PhoneClass
Attributes
brand: String;
model: String;
owner: String;
purchaseDate: TDateTime;
price: Double;
Methods
procedure create; {default constructor}
procedure create(Brand, model, owner,purchaseDate, listDate, price);
function getPrice: Double;
procedure setPrice(price: Double);
function calculateAge: Integer;
function toString: String;
Identity
State
Behavior
Encapsulation

As we develop the app, you will see how this directs the process.

A class definition is a coded version of the class diagram with all methods implemented. Once the class has been defined, it can be used in an application like any other data type.

ADDING AN EMPTY CLASS DECLARATION

4. Between the *Interface* and *implementation* lines of your code, add the following statement:

   This imports the core utilities of Delphi into your class, allowing you to use them.

```
unit PhoneClass
interface
uses SysUtils;
implementation
```

5. Underneath the *uses* line, you need to set your class name in the *Type* section of the class. To do this, add the following code type:

```
    TPhone = class
        private
        public
    end
```

   This tells your program that you are defining a new class of type “TPhone”.

```
unit PhoneClass;
interface
uses SysUtils
type
    TPhone = class
        private
        public
end;
Implementation
End.
```

This is the last step needed to create your class unit and an empty class declaration. Save and run your application.

DECLARING THE ATTRIBUTES

6. Attributes are variables that hold the state of the object, the class models, and are declared like any other variable – except they are put in the private section of the class declaration.

   By placing the attributes in the private section, you ensure that they cannot be accessed by any units outside of the class unit. The data is hidden and its integrity is protected. In the next section, you will see how values can be added to these variables.

SYNTAX

```
unit name;
interface
uses
    Imported Units
type
ClassName = class
 Private
   attribute1: datatype1;
   attribute2: datatype2;
    ...
   methods
  end;
 Implementation
 End.
```

IMPLEMENTATION

Phone

```
unit PhoneClass;
interface
uses SysUtils;
type
  TPhone = class
    Private
      Brand: String;
      Model: String;
      Owner: String;
      PurchaseDate: TDateTime;
      Price: Double;

 end;
Implementation
End.
```

Save and run your application, make sure there are no syntax errors in the code that was added.

DECLARING THE METHODS

Methods are the active code that bring behaviour to an object. Methods declared in the public section serves as an interface to the object. Those declared in the private section, only assist internally with the implementation of the behaviour.

7. A phone object needs to be created (constructed) for the user to interact with. A special method only used once and started with the Delphi keyword **constructor** is used to instantiate the phone object. This method is usually called *create* to give a clear sense of its purpose.

8. Now we declare the methods that will act upon the attributes.

   Some of the data is fixed and cannot be altered. We consider the following use cases:

   - Users need to get the Price in order to view it before and when making changes.
   - Users need to set the Price in order to lower the price if the phone is not selling.
   - Users need to find out the age of the phone. (Why don't we store a value for the age?)
   - Programmers may need to print out the state of an object. For this we use a function named *toString* that return a string of all the values stored in the attributes.

<table>
<tr><th>SYNTAX</th><th>IMPLEMENTATION</th></tr>
<tr><td><pre>
unit Unit;
interface
uses Other units that the program
     will depend upon
type
ClassName = class
       (OptionalBaseClass)
       (Default is TObject)
 Private
 { Now declare attributes …}
   attribute1: datatype1;
   attribute2: datatype2;
    ...
  {private methods here}
Public
{Now declare public methods here}
constructor name(Optional
parameters: datatypes);
procedure name(Optional parameters:
datatypes);

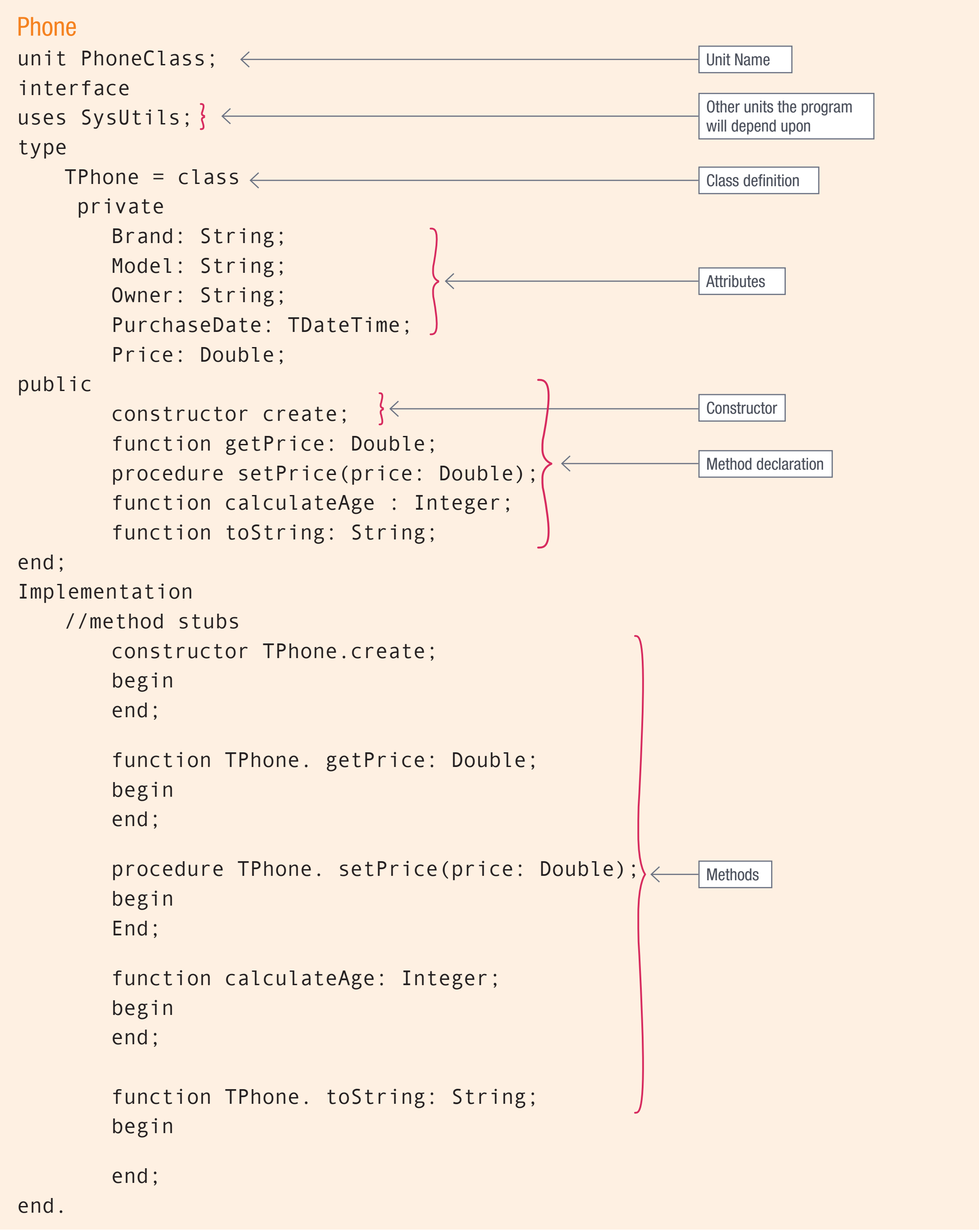
function name(optional parameters:
datatypes): datatype;
End;
Implementation
End.
</pre></td><td>Phone<pre>
unit PhoneClass;
interface
uses SysUtils;
type
  TPhone = class
    Private
      Brand: String;
      Model: String;
      Owner: String;
      PurchaseDate: TDateTime;
      Price: Double;
    Public
    Constructor create;
    function getPrice: Double;
    procedure setPrice(price: Double);
    function calculateAge: Integer;
    function toString: String;
  end;
implementation
end.
</pre></td></tr>
<tr><td colspan="2">Save and run your application.</td></tr>
</table>

CREATING THE METHOD STUBS IN THE IMPLEMENTATION SECTION

Now that the basic class structure has been defined, we need to complete the definition by implementing the methods listed. These methods will be coded in the *implementation* section, that is, underneath the *implementation* keyword.

9. Place the cursor on the line *constructor create*; and press ctrl + shift+ C and watch how RAD studio generates a method stub (an empty method) for all the declared methods. Placing the cursor on any one of the other method declarations will also work.
10. If you make a change to the declaration of a method you must ensure that you make the same correction to the implementation.

IMPLEMENTATION



Phone

```
unit PhoneClass;
interface
uses SysUtils;
type
    TPhone = class
     private
        Brand: String;
        Model: String;
        Owner: String;
        PurchaseDate: TDateTime;
        Price: Double;
public
        constructor create;
        function getPrice: Double;
        procedure setPrice(price: Double);
        function calculateAge : Integer;
        function toString: String;
end;
Implementation
    //method stubs
        constructor TPhone.create;
        begin
        end;

        function TPhone. getPrice: Double;
        begin
        end;

        procedure TPhone. setPrice(price: Double);
        begin
        End;

        function calculateAge: Integer;
        begin
        end;

        function TPhone. toString: String;
        begin

        end;
end.
```

Save and run your application, make sure there are no syntax errors in the code that was added.

ADDING CODE TO THE METHOD STUBS

The final step in defining the class is to provide it with behaviour, that is, the code to be executed when calling the methods of the class.

11. We begin with the constructors. There are two versions: Default constructor and default values for each attribute according to its data type, assigned by Delphi.
    - Where we have data, the constructor receives them as parameters and assigns them to the attributes. The constructor creates an instance of a class. This is called instantiation – it generates an object and initialises the attributes of the object.

SYNTAX

```
constructor name(
    Parameters: datatype); overload;
```

Because we will now have two definitions for the constructor, we need to tell the compiler about this, so we add the *overload* keyword to both constructor declarations.

IMPLEMENTATION

```
constructor create(
    brand, model, owner: String;
    purchaseDate: TDateTime;
    price: Double); overload;
```

IMPLEMENTATION

In the class definition underneath the default constructor insert the new constructor declaration and press *ctrl+shift+C* to generate the stub for the Parameterised constructor.

It is helpful to use the full names of the attributes in the constructor's definition. This makes it easier to identify the attributes in the next step.

Now complete the code for the constructor as indicated in the right hand column.

Note: the **Self** keyword refers to the attribute of the phone.

Add the assignments as shown:

```
Public
constructor create; overload;
constructor create(
    brand, model, owner: String;
    purchaseDate: TDateTime;
    price: Double); overload;

implementation
constructor TPhone.create(
    brand, model, owner: String;
    purchaseDate: TDateTime;
    price: Double);
begin
  Self.brand  := brand;
  Self.model  := model;
  Self.owner  := owner;
  Self.purchaseDate := purchaseDate;
  Self.price  := price;
end;
```

12. Now we add code to the body of the **other** methods to introduce behaviour into the object as follows:

getPrice returns the value of price which is stored as a *Double*.

```
function  TPhone.getPrice: Double;
  begin
     result := price;
  end;
```

SetPrice uses the parameter for price to set the value of the phone object's attribute.

```
procedure TPhone.setPrice(price:
Double);
     begin
      Self.price := price;
    end;
```

CalculateAge returns an integer.

TimeDate: the current date is stored as the number of days since 30 December 1899. We can access it through the system variable, now. Its decimal part represents the time.

Subtracting the *purchaseDate* from *now*, and rounding the result gives the number of days between them. This is then divided by 365 (div, that is, integer division) for the number of years.

```
function TPhone.calculateAge: Integer;
var
     numDays: Integer;
 begin
 numDays :=   round(now -  purchaseDate);
 result := numDays DIV 365;
end;
```

ToString returns a string representation of the current state of the object, that is, the values of its attributes.

We build the string using a simple format.

Use *formatDateTime* ('dddddd', date) to get the date into a string format dd mmm yyyy, for example, 05 June 2020.

Format the price as money.

Finally, we return it as a result.

```
function TPhone.toString: String;
var output: String;
begin
  Output :=
  'Brand: ' + brand + sLineBreak +
  'Model: ' + model + sLineBreak +
  'Owner: ' + owner + sLineBreak +
  'purchaseDate: ' +  formatdatetime('ddd
  ddd', purchaseDate) + sLineBreak +
   'Price: ' + Format('%m',[price]);
   Result := output;
end;
```

Save and run your application.

This completes the class definition, that is, class and attributes declared; all methods implemented.

The next step is to test the class. Once completed, it can then be used in any application that needs this model of a cell phone.

Note: The design of this *TPhone* class was to solve a specific problem. **Reusability** is an important OOP principle. You should always try to design a class with reuse in mind, so that your class can be used or be extended for other scenarios. To best achieve reusability, you must only model the essential features that can be applied to many scenarios.

New words

reusability – is an important OOP principle.

Activity 2.2 Defining classes: Pen and paper

In each case, clearly indicate where in the class unit file the code to declare the class, the attributes and methods should be placed.

Do not implement the methods.

2.2.1 A *TCar* class containing the attributes *Model*, *Brand*, *Year*, *RetailPrice* and the following methods:

a. A default and a parameterised constructor.

b. A *getDetailedModel* function.

c. A *setRetailPrice* procedure and *getRetailPrice* function.

d. A *toString* function.

e. A *getVATPrice* function.

Name the unit *CarClass*.

2.2.2 A *TSong* class containing the attributes *Artist*, *Song*, *Album*, *TrackNumber* and *Duration* (in seconds) and the following methods:

a. A Default and a parameterised constructor.

b. Getters and setters for all attributes.

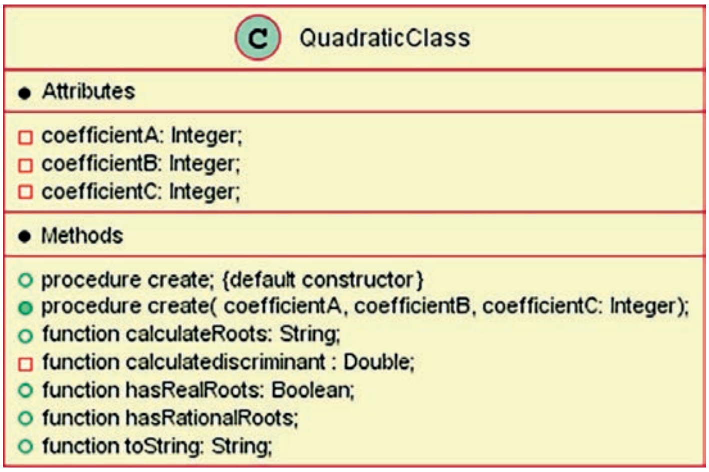
c. A *toString* method.

d. A *getQuickReference* method that returns a string.

e. A *getMinuteDuration* method that returns a string.

Name the unit *SongClass*.

2.2.3 A *TQuadratic* class for the class diagram given below:



C QuadraticClass

● Attributes

□ coefficientA: Integer;
□ coefficientB: Integer;
□ coefficientC: Integer;

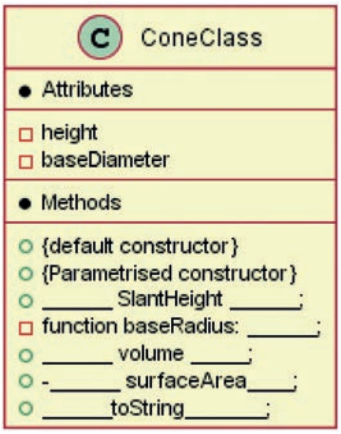
● Methods

○ procedure create; {default constructor}
● procedure create(coefficientA, coefficientB, coefficientC: Integer);
○ function calculateRoots: String;
□ function calculatediscriminant : Double;
○ function hasRealRoots: Boolean;
○ function hasRationalRoots;
○ function toString: String;

Name the unit *QuadraticClass*.

2.2.4 A *TCone* class according to the partial Cone class diagram given below. Complete the declaration of the class, adding getters and setters that you may need.

Name the unit *ConeClass*.



C ConeClass

● Attributes

□ height
□ baseDiameter

● Methods

○ {default constructor}
○ {Parametrised constructor}
○ ______ SlantHeight ______;
□ function baseRadius: ______;
○ ______ volume _____;
○ -______ surfaceArea____;
○ ______toString________;

Activity 2.3 Defining classes: Delphi projects

2.3.1 Open the project saved in the carPriceList folder. Add a unit *CarClass* and provide code to implement the *TCar* class you wrote in Activity 2.2.1.

a. Complete the constructor, getters and setters you have generated.

b. The **toString** method returns a string that displays the lines *attribute label: attribute value* below each other.

c. The **getDetailedModel** function returns a string in the format Year Brand Model.

d. The **getVATPrice** function returns the car's retail price multiplied by 1.15.

2.3.2 Open the project saved in the *myPlaylist* folder. Add a unit *SongClass* and implement the *TSong* class you wrote in Activity 2.2.2. Complete the class definition as follows:

a. Code the constructor, getters and setters you have generated.

b. The **toString** method returns a string that displays the lines *attribute label: attribute value* below each other.

c. The **getQuickReference** function returns a string in the format 'Song,~Album~Artist'.

d. The **getMinuteDuration** function converts the song duration from seconds to minutes and seconds and returns a string in the format mm:ss.

2.3.3 Open the project saved in the *quadraticEquations* folder. Add the unit *QuadraticClass* and code the class definition you wrote in Activity 2.2.3. Implement:

a. The constructors.

b. **CalculateDiscriminant**.

$$discriminant = b^2 - 4ac$$

c. **calculateRoots** to return the roots in format, 'x1 = 0.00 : x2 = 0.00'.
Use the formula:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

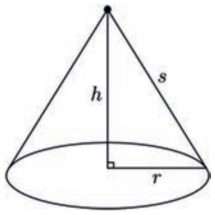
d. **HasRealRoots** should return true if the discriminant is >= 0.

e. **HasRationalRoots** should return true if the discriminant is a perfect square.

f. A toString method to return the quadratic in the form: $ax^2 + bx + c = 0$

Hint: use chr (178) to display *superscript 2*.

2.3.4 Open the project saved in the *coneCalculations* folder. Add the unit *ConeClass* and code the class definition you wrote in Activity 2.2.4. Implement the methods listed in the class diagram and the getters and setters you added.

	*s* = slant height *r* = radius of base *h* = height	Slant height: Volume: Surface area:	$s = \sqrt{r^2 + h^2}$ $V = \pi r^2 \frac{h}{3}$ $A = \pi r\,(r + s)$
There are two possible implementations of cone calculations: • Trigonometry, as exemplified by: *http://mathcentral.uregina.ca/QQ/database/QQ.09.07/s/marija1.html* • Pythagoras, as exemplified by: *https://mathbitsnotebook.com/Geometry/3DShapes/3DCones.html*			

2.3.5 Update your *TPhone* class by adding getters for *Brand*, *Model* and *Owner*.

LET'S REVISE

TERM AND DESCRIPTION	CODE EXAMPLES

CLASS:
● data structure designed to model the state and behaviour of a class of Object
● when declared under the *type* keyword it becomes a datatype
● variables declared of this datatype can store objects or instances of the class
	`type`
	`  TPhone = class`
	`    private`
	`    public`
	`end;`

ATTRIBUTES:
● set of variables
● declared under the private keyword
● variables can have different datatypes
● refers to the state of the object
	`type`
	`  TPhone = class`
	`    Private`
	`      Brand : String;`
	`      Model : String;`
	`      Owner : String;`
	`      PurchaseDate : TDateTime;`
	`      Price : Double/Real;`
	`    public`
	`end;`

METHODS:
● it is used to allow an object to perform actions
● will provide results to applications that are normally declared with *public* access so that they can be 'called' by application programs (e.g. main Form)
● serves as the interface to an object within an application
● is how the object receives data and gives information
● there are five basic types of methods:
○ constructor methods
○ accessor (or getters) methods
○ mutator (or setters) methods
○ auxiliary (or helper) methods
○ ToString method
	`type`
	`  TPhone = class`
	`    private`
	`    public`
	`      constructor create;`
	`      function getPrice : Double;`
	`      procedure setPrice (price : Double);`
	`      function calculateAge : Integer;`
	`      function toString : String;`
	`end;`

a. Constructor:
● a method that is called to create the object, and instance of the class
● through its parameters the object is assigned a unique initial state
● calling a constructor without parameters creates an object with default values
● must be called on the class name – not the object name – as the object does not exist prior to this call
● the object created when the constructor is called should be assigned to a declared object variable, otherwise is will be dangling in memory and cannot be used.
	`type`
	`  TPhone = class`
	`    private`
	`    public`
	`      constructor create (`
	`        brand, model, owner : String;`
	`        purchaseDate : TDateTime;`
	`        price : Double/Real);`
	`      ….`
	`end;`
	`Implementation`
	`   constructor TPhone.create (brand, model,`
	`   owner : String; purchaseDate : TDateTime;`
	`   price : Double/Real);`
	`   begin`
	`      Self.brand := brand;`
	`      Self.model := model;`
	`      Self.owner := owner;`
	`      Self.purchaseDate := purchase - Date;`
	`      Self.price := price;`
	`   end;`

TERM AND DESCRIPTION	CODE EXAMPLES

b. Accessor (or getters):
- functions that return the state of the attributes
- can transform outputs before displaying them
- it allows you to store data in one way, but present it to the application in a different, more useful way
- make debugging easier, as you can include breakpoints inside the accessors (getters) functions, allowing you to detect mistakes that occur at this point in the program
- getPrice returns the value of price which is stored as a *Double* or *Real*
	`type`
	`  TPhone = class`
	`    private`
	`    public`
	`       ……;`
	`       function getPrice : Double;`
	`       ……… .`
	`end;`
	`Implementation`
	`Function TPhone.getPrice : Double;`
	`    begin`
	`       Result := price;`
	`    end;`

c. Mutator (or setters):
- procedures that update the state of the object
- can include input validation, automatically rejecting any incorrect values
- can transform inputs before storing them
- allows you to build rules or conditions into the storage of variables
- make debugging easier, as you can include breakpoints inside the mutators (setters) functions, allowing you to detect mistakes that occur at this point in the program
- *setPrice* uses the parameter for price to set the value of the phone object's attribute
	`type`
	`  TPhone = class`
	`    private`
	`    public`
	`       ……`
	`       procedure setPrice (price : Double/Real);`
	`       ….`
	`end;`
	`Implementation`
	`procedure TPhone.setPrice (price : Double/Real);`
	`    begin`
	`       Self.price := price;`
	`    end;`

d. Auxiliary (or helper):
- Provide additional procedures or functions that assist the accessors (getters) and mutators (setters) in their work
- Enable you to reduce the complexity within a class by providing an abstraction of a complex algorithm within a method
- Some generate data that is used by other methods, often derived from the essential data encapsulated in the class
- Use the data sent to them by a calling method and returning the generated data
- Do not have the responsibility of changing the state e.g. *calculateAge* in the *TPhone* class used *PurchaseDate,* an attribute to return age
	`type`
	`  TPhone = class`
	`    private`
	`    public`
	`       …`
	`       function calculateAge : Integer;`
	`       …`
	`end;`
	`Implementation`
	`function TPhone.calculateAge : Integer;`
	`  Var`
	`      numDays : Integer;`
	`    begin`
	`       numDays := round(now - purchaseDate);`
	`       result := numDays DIV 365;`
	`    end;`

TERM AND DESCRIPTION

e. ToString:

- is used to display the state of the class, that is, the values of its attributes
- when displaying more than the state of the class, e.g. calculated results, then you should develop a function that returns a string version (preferably) of the information you want to display.

CODE EXAMPLES

```
type
  TPhone = class
    private
      Brand : String;
      Model : String;
      Owner : String;
      PurchaseDate : TDateTime;
      Price : Double/Real;
    public
       …
       function toString : String;
end;
Implementation
function TPhone.toString : String;
  Var
      output : String;
    begin
       Output := 'Brand: ' + brand +
       sLineBreak + ‘Model: ' + model +
       sLineBreak +
       'Owner: ' + owner + sLineBreak +
       'purchaseDate: ' +
       formatDateTime('dddddd', puchaseDate)
       + SLineBreak +
       'Price: ' + Format('%m', [price]);
       Result := output;
    end;
```

Did you know

- Generally, we let the method calling *calculateAge deal with the display of the age.*
- Using many short methods reduces complexity.
- Each method should only have one task.

Did you know

A method name followed by its parameters is called the **method signature**.

Overloading is declaring more than one method with the same name. Overloading allows us to have multiple methods that share the same name, but with a different number of parameters and types.

Example:

```
   Public
       Constructor create; overload;
       Constructor create (brand, model, owner : String;
purchaseDate : TDateTime; price : Double/Real); overload;
```

UNIT

2.2 Using the class

Before other units in your application can use your class, they need to import the class unit. Having access to the custom class allows the application to create objects of the imported class and call its public methods.

IMPORTING THE TPHONE CLASS

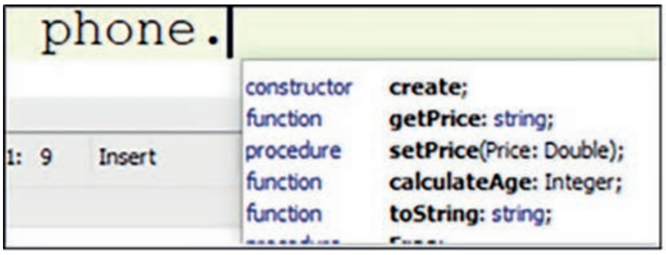
1. Open the main form of the application and add the unit name (PhoneClass) to the **USES** section at the top of the code. This imports the TPhone class to your main form.
2. In your main form, declare a variable called *phone* of type, *TPhone*, that is, *phone: TPhone*.
3. Save and run your application.

   If everything is correct, your application should open without any errors. This means that you have successfully created a custom class and added it to your application.

   To check if the *PhoneClass* interface is visible, double click on the [Save] button and type the word 'phone' in the event handler.

CLASS METHODS IN DELPHI

https://www.youtube.com/watch?v=TDTak5nsVD4

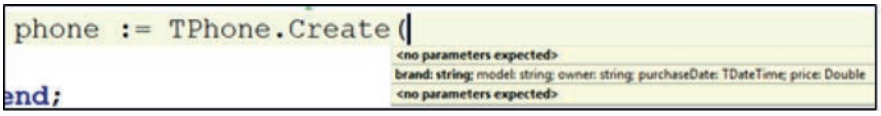


```
phone.
1:  9     Insert
constructor   create;
function      getPrice: string;
procedure     setPrice(Price: Double);
function      calculateAge: Integer;
function      toString: string;
```

See how the interface we defined for our class pops up. Everything defined in the public section of the class definition is visible. You will notice that we cannot see the attributes in the private section.

Now we can use the constructors to instantiate objects of type TPhone.

Make sure you use Delphi's popup tip to pass the parameters through in the correct order. For example:



```
phone := TPhone.Create(
                      <no parameters expected>
                      brand: string; model: string; owner: string; purchaseDate: TDateTime; price: Double
                      <no parameters expected>
end;
```

SYNTAX	IMPLEMENTATION

ObjectName = Classname.
name(optional parameters:
datatypes);
	phone1 := TPhone.create;
	phone2 := TPhone.create('Huawei',
	'P10', 'John Smith',
	StrtoDateTime('2017/7/21'),
	750.00);

TESTING THE TPHONE CLASS

A quick test to see if the all the methods are implemented correctly and if the correct data is stored in the attributes can be done as follows:

EXPLANATION	IMPLEMENTATION

4. Add the code in right hand column to the *onclick* event of button [Save] in the form class.
The first line creates a *TPhone* object.
The second line displays its state.
	phone := TPhone.create;
	showmessage(phone.toString);

5. Now instantiate the phone object with the following attributes:

   Brand: Huawei

   Money: P10

   Owner: John Smith

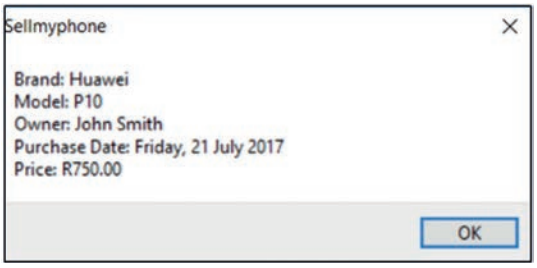
   Purchase Date: 2017/7/21

   Price: R750.00

   Replace code in the [Save] button with the code in the right hand column.

   Save and run the application.

```
phone := TPhone.create('Huawei',
'P10', 'John Smith',
StrtoDateTime('2017/7/21'),
750.00);
showmessage(phone.tostring);
```



Sellmyphone

Brand: Huawei
Model: P10
Owner: John Smith
Purchase Date: Friday, 21 July 2017
Price: R750.00

In a similar way you can test all of the methods:

Add the code on the right hand column to the [Save] button to test the *setPrice*, *getPrice* and *calculateAge* methods.

%m uses the same system values as *FormatCurr*.

%d converts an integer to a string.

For more quick formatting with *ShowMessagefmt('%d, f, s, m',[args])*.

```
phone := TPhone.create('Huawei',
'P10', 'John Smith',
StrtoDateTime('2017/7/21'),
750.00);
showmessage(phone.tostring);
phone.setPrice(650.00);
ShowMessagefmt('%m' ,[phone.
getPrice]);
ShowMessagefmt('%d', [phone.
calculateAge];
```

Save and run your application.

COMPLETING THE SELLMYPHONE APP

The *SellMyPhone* App. captures the input from the components, then instantiates a phone object, and finally displays the brand, model, age and price as a record in the string grid. These inputs could come from various sources like a text file, parallel arrays or a database.

Before we use the class add getters for the attribute *Brand* and *Model*. Replace the test code in the [Save] button with the code below:

EXPLANATION	IMPLEMENTATION

Declare the phone globally under the **implementation** keyword.
Declare a global variable row to manage the insertion point for the *StringGrid*.
	`Implementation`
	`var`
	` phone: TPhone;`
	` Row: Integer = 1;`

Declare local variable for inputs from the various components.	`procedure TfrmTPhone.btnSaveClick(Sender: TObject);`
	`var`
	`  Brand, Model, Owner : String;`
	`  PurchaseDate : tDateTime;`
	`  Price : Double;`

Read and store the inputs.	`begin`
	`  Brand := cbxBrand.Items[cbxBrand.ItemIndex];`
	`  Model := edtModel.Text;`
	`  Owner := edtOwner.Text;`
	`  PurchaseDate := StrToDate(edtDate.Text);`
	`  Price := StrToFloat(edtPrice.Text);`

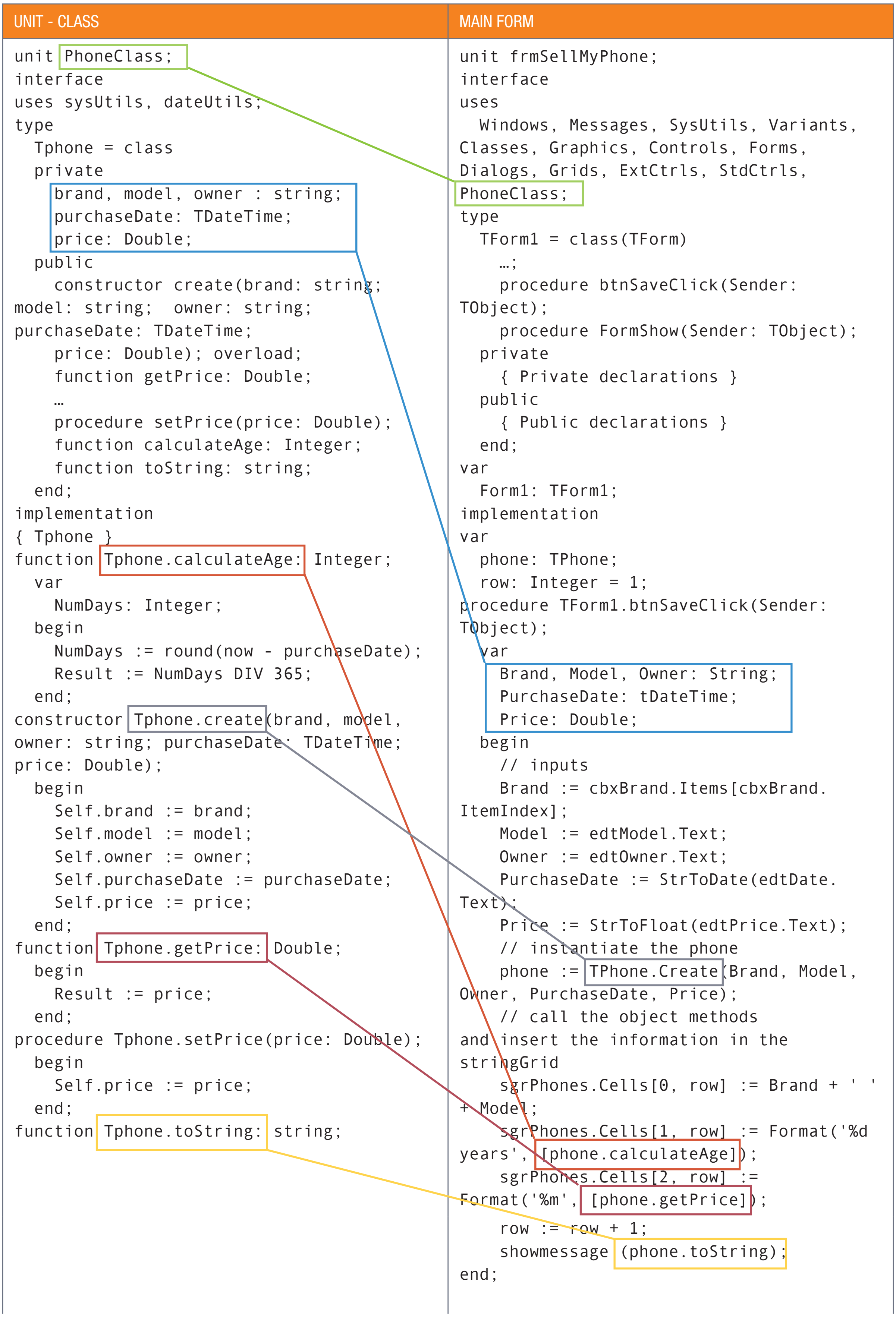
Instantiate the phone object calling the parameterised constructor and the input values provide by the user.	`  phone := TPhone.Create(Brand, Model, Owner, PurchaseDate, Price);`

Call various *TPhone* class methods to provide information about the phone object.	`  sgrPhones.Cells[0,row] := phone.getBrand + ' ' + phone.getModel;`
	`  sgrPhones.Cells[1,row] := Format('%d years', [phone.calculateAge]);`
	`  sgrPhones.Cells[2,row] := Format('%m', ,[phone.getPrice]);`
	`  row := row + 1.`
	`end;`

Save and run your application.

You could use a *TList* to store a list of phone objects. This will ensure that all the phone objects will be available while the program is running. If you want a more permanent copy of the list then send the data to a file or store it in a database.

PROGRAM INTERACTION WITH THE CLASS



UNIT - CLASS

```
unit PhoneClass;
interface
uses sysUtils, dateUtils;
type
  Tphone = class
  private
    brand, model, owner : string;
    purchaseDate: TDateTime;
    price: Double;
  public
    constructor create(brand: string;
model: string;  owner: string;
purchaseDate: TDateTime;
    price: Double); overload;
    function getPrice: Double;
    …
    procedure setPrice(price: Double);
    function calculateAge: Integer;
    function toString: string;
  end;
implementation
{ Tphone }
function Tphone.calculateAge: Integer;
  var
    NumDays: Integer;
  begin
    NumDays := round(now - purchaseDate);
    Result := NumDays DIV 365;
  end;
constructor Tphone.create(brand, model,
owner: string; purchaseDate: TDateTime;
price: Double);
  begin
    Self.brand := brand;
    Self.model := model;
    Self.owner := owner;
    Self.purchaseDate := purchaseDate;
    Self.price := price;
  end;
function Tphone.getPrice: Double;
  begin
    Result := price;
  end;
procedure Tphone.setPrice(price: Double);
  begin
    Self.price := price;
  end;
function Tphone.toString: string;
```

MAIN FORM

```
unit frmSellMyPhone;
interface
uses
  Windows, Messages, SysUtils, Variants,
Classes, Graphics, Controls, Forms,
Dialogs, Grids, ExtCtrls, StdCtrls,
PhoneClass;
type
  TForm1 = class(TForm)
    …;
    procedure btnSaveClick(Sender:
TObject);
    procedure FormShow(Sender: TObject);
  private
    { Private declarations }
  public
    { Public declarations }
  end;
var
  Form1: TForm1;
implementation
var
  phone: TPhone;
  row: Integer = 1;
procedure TForm1.btnSaveClick(Sender:
TObject);
  var
    Brand, Model, Owner: String;
    PurchaseDate: tDateTime;
    Price: Double;
  begin
    // inputs
    Brand := cbxBrand.Items[cbxBrand.
ItemIndex];
    Model := edtModel.Text;
    Owner := edtOwner.Text;
    PurchaseDate := StrToDate(edtDate.
Text);
    Price := StrToFloat(edtPrice.Text);
    // instantiate the phone
    phone := TPhone.Create(Brand, Model,
Owner, PurchaseDate, Price);
    // call the object methods
and insert the information in the
stringGrid
    sgrPhones.Cells[0, row] := Brand + ' '
+ Model;
    sgrPhones.Cells[1, row] := Format('%d
years', [phone.calculateAge]);
    sgrPhones.Cells[2, row] :=
Format('%m', [phone.getPrice]);
    row := row + 1;
    showmessage (phone.toString);
end;
```

PROGRAM INTERACTION WITH THE CLASS

UNIT - CLASS

```
  begin
    Result := 'Brand: ' + brand +
    sLineBreak + 'Model: ' + model +
    sLineBreak + 'Owner: ' + owner +
    sLineBreak + 'Purchase Date: ' +
    formatdatetime ('dddddd', purchaseDate)
    + sLineBreak + 'Price: ' + Format
    ('R%f', [price]);
  end;
end.
```

MAIN FORM

```
// given code
procedure TForm1.FormShow(Sender:
TObject);
  begin
    sgrPhones.Cells[0, 0] := 'Cell Phone';
    sgrPhones.Cells[1, 0] := 'Age';
    sgrPhones.Cells[2, 0] := 'Price';
  end;
end.
```

Activity 2.4 Pen and Paper

2.4.1 Write down the syntax for instantiating a class.

2.4.2 Given is the constructor declaration for a *TNGO* class:

Constructor create(Name: string; Funds : Double: Donations : Integer; Code: String); For (a) to (g) below, state whether an accessible NGO object will be created. If not, then state why.

a. NGO.create('Helping Hand', 23500.00, 50, 'HH007');

b. NGO := TNGO.create('Helping Hand', 'HH007', 23500.00, 50);

c. NGO := TNGO.create('Helping Hand', 23500.00, 50, 'HH007');

d. TNGO := NGO.create('Helping Hand', 23500.00, 50, 'HH007');

e. NGO := TNGO.create('Helping Hand', 23500.00, 50);

f. TNGO.create('Helping Hand', 23500.00, 50, 'HH007');

g. NGO := TNGO.create('Helping Hand', 23500.00, 50.00, 'HH007');

2.4.3 Give a Delphi statement to import the class, NGOClass into the form unit.

2.4.4 How will you ensure that the value of an attribute cannot change after the object is created?

2.4.5 What is the function of the toString-method?

2.4.6 What criteria will you use to decide if a method belong to the class or not?

2.4.7 Which one of the following is the correct syntax for the signature(header) of a *get-method*? Also state why the others are incorrect.

a. Procedure methodName(parameter: datatype);

b. Function methodName(parameter: datatype): dataType;

c. Procedure methodName: dataType;

d. Function methodName: dataType;

Did you know

Non-governmental organisations, or NGO were first called such in Article 71 in the Charter of newly formed United Nations in 1945. While NGOs have no fixed or formal definition, they are generally defined as non-profit entities independent of governmental influence

Activity 2.5

2.5.1 Open the project saved in the 02 – carPriceList folder. Provide code to:

a. Import the *CarClass* unit in the *frmCar* unit.

b. Declare the global variables Car of type *TCar* and an Integer variable Row initialised to 1, in the implementation section. Remember the *var* keyword.

c. In the [Save] button's onclick event handler, provide code to:

i. Store the inputs brand, model, year and retail price.

ii. Create the Car object

iii. Call the methods **getDetailedModel** and **getVATPrice** to display the model details and the retail price in the string grid.

2.5.2 Open the project saved in the 02 – myPlaylist folder. Provide code to:

a. Import the *SongClass* unit in the *frmSong* unit.

b. Declare a global variable *mySong* of type *TSong* and an Integer variable *Row* initialised to 1, in implementation section. Remember the *var* keyword.

c. In the [AddtoPlaylist] button's onclick event handler, provide code to:

i. Store the inputs Artist, Album, Song, TrackNumber and duration.

ii. Create the *mySong* object.

iii. Call the methods *getQuickReference* and *getMinuteDuration* to display the quick reference and the minute duration in the string grid.

Did you know

When using a class, always check for for potential helper methods. Any statement or function that will be used more than once and is only related to the class data should be moved to a private helper method inside the class.

Activity 2.5 *continued*

CHALLENGE

2.5.3 The mathematics teacher at school Overcrowded High is requesting your help. The teacher wants an application that will generate quadratic equations in the following categories:

- those that have rational roots
- those that have irrational roots.

He also wants only those equations for which coefficients *a*, *b* and *c* are non-zero.

The equations and their roots must be sent to a file. The teacher wants to print the file and give each of the 60 learners a unique set of three problems to solve.

Open the project *HelpMathTeacher* in the folder *quadraticEquations*. The code for the buttons [Save] and [Shuffle List] has been provided.

You are required to import the *QuadraticClass* and declare a variable of type *TQuadratic*.

In the Make list event-handler, the code to generate permutations of the coefficients *a*, *b* and *c* is given as three nested loops. Insert code in the nested loop to do the following:

a. Test for non-zero coefficients.

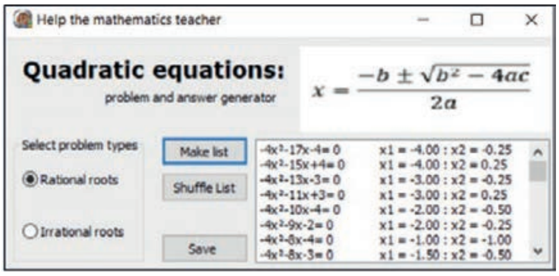
b. Create a Quadratic object.

c. Check which problem type is selected.

d. Test if the current object meets the selected problem type.

- Build a line with the quadratic equation and its roots.
- Add this line to the list box.

In b to d above, use the methods of the Quadratic class you imported.



```
-4x²-17x-4= 0        x1 = -4.00 : x2 = -0.25
-4x²-15x+4= 0        x1 = -4.00 : x2 = 0.25
-4x²-13x-3= 0        x1 = -3.00 : x2 = -0.25
-4x²-11x+3= 0        x1 = -3.00 : x2 = 0.25
-4x²-10x-4= 0        x1 = -2.00 : x2 = -0.50
-4x²-9x-2= 0         x1 = -2.00 : x2 = -0.25
-4x²-8x-4= 0         x1 = -1.00 : x2 = -1.00
-4x²-8x-3= 0         x1 = -1.50 : x2 = -0.50
```

2.5.4 The Maths Literature teacher has asked you to develop an application to help the learners check their homework based on cones. The App. is required to calculate three values for a cone:

- slant height, surface area and volume given the height and diameter of the base.

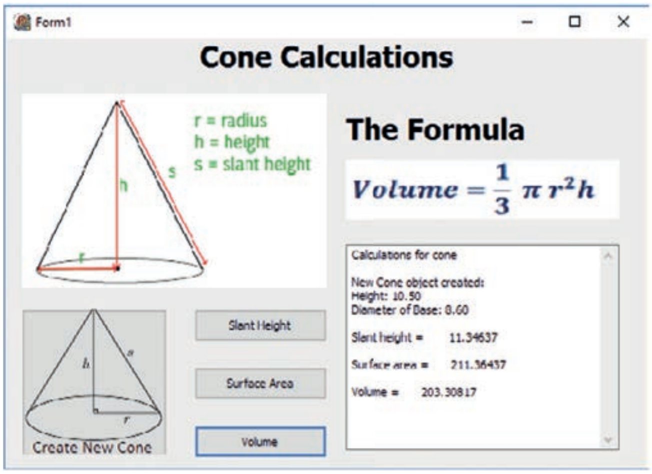
The following buttons have been provided in the Project file inside the 02 – Cone Calculation:

a. In the form class add code to import the class.

b. Button [Create New Cone]: demonstrates the parameterised constructor. Use two (2) *input boxes* to get the parameters required to instantiate the object, create the object and use the *toString* method to display the state of the Cone object in the memo. Code to enable the other buttons is given.

c. Button [Slant Height]: using the cone created in b, display the slant height in the memo.

d. Button [Volume]: using the cone created in b, display the volume in the memo.



```
Calculations for cone

New Cone object created:
Height: 10.50
Diameter of Base: 8.60

Slant height =     11.34637

Surface area =     211.36437

Volume =     203.30817
```

e. Button [Surface Area]: using the cone created in b, display the surface area in the memo.

One of the reasons custom classes are so powerful is that they allow you to keep lists of objects, with all the data related to each object stored in the object itself. This is a lot more reliable than creating multiple parallel lists, as inconsistencies can easily appear in parallel lists (for example, if an item is deleted from one list but not from the others).

Our *SellMyPhone* App. only uses one object at a time and reuses the object variable for each instatiation. Once a second phone is created you can't go back to the first object to change the price of the phone. You will have to create another phone object for the same cell phone and new price. This shortcoming motivates the use of a list to store objects so that we can use them in a meaningful way.

Did you know

You will not be examined on your ability to create arrays (or lists) of objects. However, when creating your own applications, custom objects will often be used with arrays and lists.

CONSOLIDATION ACTIVITY Chapter 2: Object-oriented programming

QUESTION 1

1.1 Explain the concept of encapsulation in object-orientated programming.

1.2 What is the purpose of a constructor in object-orientated programming?

1.3 Explain why you would use a getter without setters.

1.4 The following class diagram has been suggested as part of the transport program to manage their drivers:

DRIVER	
ATTRIBUTES	METHODS
- driverID : String - surname : String - firstName : String - cellNumber : String - fullTime : Boolean	+ getDriverID() : String + getSurname() : String + getFullTime() : Boolean + setFirstName(firstName : String) + toString() : String + calculateAge() : Integer

f. The minus sign (-) shows that the declaration is private while the plus sign (+) shows that the declaration is public. Explain the difference between a public declaration of attributes and a private declaration of attributes and methods.

g. Write down an example of an auxiliary method from the Driver class.

1.5 Define object-orientated programming.

QUESTION 2

For this application, open the project saved in the 02 – Question 2. Once done, save your project in the same folder.

Tourists visiting South Africa often travel to different areas of the country. Normally, guests to a bed and breakfast (B&B) have to pay the bill for all the extra items they ordered during their stay when they check out. To provide a service that makes them more appealing than other B&Bs, the Petersen Group has decided to let their guests transfer their accumulated extra costs between the guesthouses in each town. Guests will have to pay the bill for these items when they check out at the last guesthouse on their journey. They decided that the best way to manage this is to e-mail a text file indicating the extra costs of the guests to the next guesthouse.

You have been asked to write the program to handle the extra costs of the guests. The data is stored in a text file named "Extras.txt" in the following format: GuestNo#GuestName#ExtraType#CostPerItem
An example of some of the data in the text file:

```
1#Mr G Ferreira#Phone#7.05
2#Mrs L Honeywell#Drinks#71.95
3#Ms I Mendes#Kitchen#39.95
1#Mr G Ferreira#Kitchen#23.95
1#Mr G Ferreira#Drinks#7.15
4#Mr B Khoza#Taxi#127.25
```

2.1 Define a class named *TExtraItem*. Create appropriately named and typed private fields to hold the following data (suggested field names are given in brackets):

- guest number (guestNum)
- item type description (itemType)
- cost per item (cost)

2.2 Write a constructor method that accepts the guest number, the item description and the cost per item as parameters. All the fields must be initialised in the constructor.

2.3 Write an appropriately named *get* method (accessor method) to return the guest number.

2.4 The company uses a 25% markup on cost per item to determine profit. Write a method named *calculateProfit* that calculates and returns the profit (that is, cost*25/100).

2.5 Write a method named *calculatePrice* that calculates the final price of the item (that is, cost + the calculated profit).

2.6 Write a method named *toString* that builds and returns a string with information on the item formatted as follows:

Item type<tab>Cost<tab>Profit<tab>Final Price

Any numbers must be formatted to two decimal places.

2.7 Create an array named *arrItems* that holds *TExtraItem* objects. Write code in the *OnActivate* event handler of the form to read information from the text file *Extras.txt* according to the following steps:

a. Test if the text file exists. Display a suitable message if the file does not exist and terminate the program.

b. Use a loop to:

- read a line of text from the text file.
- separate the text into the guest number, item type and cost.
- use this information to create a new *TExtraItem* object and place the object in the array named *arrItmes*.

c. Use a counter field to keep track of how many items there are in the array.

2.8 When the user clicks the [List Items] button, the program must do the following:

a. Allow the user to enter a guest number.

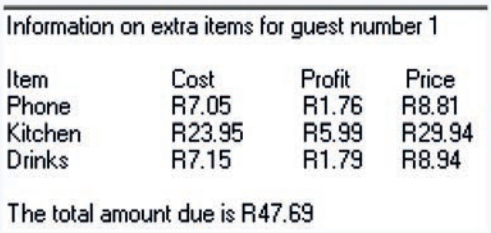
b. Search through the array. Each time an item for the guest is found:

- calculate the profit using the percentage mark-up and calculate the final price.
- display the information using the *toString* method.
- add the final price for each item to get a grand total.

c. When the search is complete the program must:

- display the total amount due for the guest.
- display an appropriate message to say that there are no extra charges for this guest, if no items have been found.

An example of the final output is shown below:



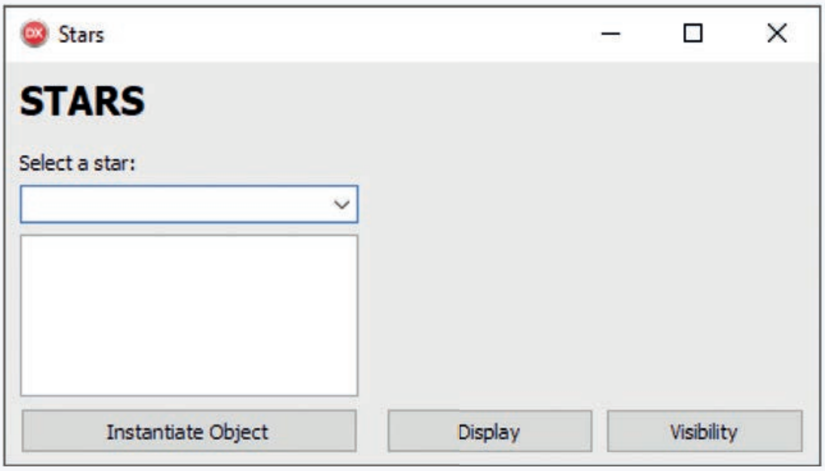
```
Information on extra items for guest number 1

Item          Cost     Profit   Price
Phone         R7.05    R1.76    R8.81
Kitchen       R23.95   R5.99    R29.94
Drinks        R7.15    R1.79    R8.94

The total amount due is R47.69
```

QUESTION 3

For this application, open the project saved in the 02 – Question 3. Once done, save your project in the same folder.

A constellation is a group of related stars that covers the night sky. Some stars are considered to be navigational, while others are passive. A navigational star is used to assist with calculating direction and movement.

The application you will be using has the following user interface.



3.1 Write code for a constructor method that will receive the name of the star, its magnitude, its distance from the Earth and the constellation it belongs to as parameters. Set the FOUR respective attributes to the received parameter values and initialise the *fNavigationalStatus* attribute to FALSE.

3.2 Write code to create an accessor method for the constellation attribute *fConstellation*.

3.3 Write code for a mutator method called **setNavigationalStatus**, which will receive a Boolean value as a parameter and set the navigational status attribute to the received value.

3.4 Write code for a method called **determineVisibility** that will determine and return a description of the visibility of the star. The visibility of a star depends on its distance from Earth in light years and its magnitude.

Use the following criteria to determine the description of visibility that applies to a star:

DISTANCE	MAGNITUDE	DESCRIPTION OF VISIBILITY
Fewer than 80 light years	Any value	Clearly visible
Between 80 and 900 light years (inclusive)	Up to 2	Hardly visible to the naked eye
	Larger than 2	Visible by means of standard optical aid
More than 900 light years	Any value	Only visible by means of specialised optical aid

3.5 Write code to create a *toString* method which returns a string formatted as follows:

- <name of star> belongs to the <constellation> constellation.
- The star has a magnitude of <magnitude> and is <distance from Earth> light years away from Earth.

If the star is a navigational star, add the following line:

- <name of star> is a navigational star.

Otherwise, add the line:

- <name of star> is a passive star.

3.6 For the [Instantiate Object] button, the user is required to select the name of a star in the combo box. Once done, write code to do the following:

a. Extract the name of the selected star from the combo box.

b. Use a conditional loop and search in the text file for the name of the selected star. The loop must stop when the name of the star has been found in the file.

c. If the name of the star has been found, do the following:
   - Instantiate a *TStar* object using the *objStarX* object variable that has been declared globally as part of the given code.
   - Test whether the star is a navigational star using the *aNavigationStars* array and set the value for the navigational status attribute accordingly.

d. If the name of the star has NOT been found in the text file, display a message to indicate that the star was not found.

3.7 When the [Display] button is clicked:

a. Display the details of the star in the rich edit component *redDescription* using the *toString* method.

b. Load the image of the constellation that the star belongs to into the *imgStar* component. The file name of the image to be displayed is the name of the constellation the star belongs to. All image files have the extension ".jpg".

3.8 The brightness and visibility of a star is dependent on the magnitude and the distance of the star from Earth. When the [Visibility] button is clicked, call the relevant methods to display the name and visibility of the star as a Message Box.

For example, if Mimosa is selected and the [Visibility] button is clicked, you should see the following message.

- Star: Mimosa
  Visibility: Hardly visible to the naked eye

TERM 2

CHAPTER 3

TWO-DIMENSIONAL ARRAYS

CHAPTER UNITS	
Unit 3.1	2D arrays
Unit 3.2	2D arrays with data
Unit 3.3	Application for 2D arrays

Learning outcomes

At the end of this chapter you should be able to

- describe the concept of 2D arrays
- define the structure of 2D arrays
- input data to 2D arrays using different sources
- use the data from 2D arrays
- output the data from 2D arrays using column and row headings.

INTRODUCTION

Imagine that you are writing a chess application. One of the main tasks of your application would be to record the position of the pieces after every move. To do this, your application would need to analyse each square of the chessboard after a move and record whether it contains a piece or not. How would you do this?

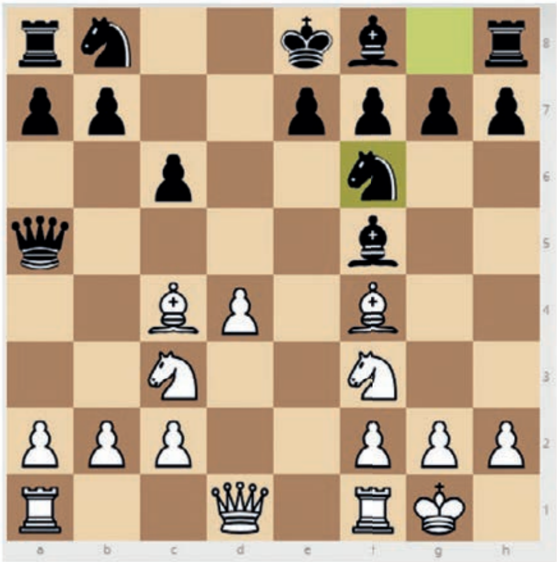


Figure 3.1: *A chessboard contains eight rows and eight columns*

2D ARRAYS

https://www.youtube.com/watch?v=ICepY3luREc

You could create 64 individual variables with each variable storing the name of the piece on it. However, with 64 variables, you would need to access each variable individually since you cannot use a loop to iterate through them. Furthermore, the computer would not understand the relationship between these squares (that is, that one square is above another one), making it very difficult to use in the rules of your game.

A better solution would be to create eight array variables, with each array containing the squares from one row. This would have the benefit of your program understanding how all the squares in the rows are related. However, you would still need to individually access the eight arrays, and your computer will still not understand the relationship between the different arrays.

In fact, the best way to handle this data is to create a single two-dimensional (or 2D) array. A 2D array uses two indices: one for the row and one for the column. This has a number of very significant advantages:

- You can store the data (elements) using a single variable.
- You can scroll through the elements using a nested FOR-loop.
- The relationship between the elements (that is, which squares are next to which) is captured in the indices, making it possible to use these relationships in your code.

In this chapter, you will learn more about 2D arrays and how they can be used in programming.

3.1 2D arrays

Since Grade 11, all your examples of arrays have been of one-dimensional arrays. A one-dimensional array stores an i-number of elements, of a specific type. These elements can be accessed using an index (usually "i"). For example, if you created an array to store the first five powers of three, the values could be represented using a single row with five elements.

aNumbers[i]	3	9	27	81	243

The third element in this array can be accessed using the array name and the index of the element in square brackets.

Accessing an element in a 1D array

```
iValue := aNumbers[3];
```

In contrast, 2D arrays store two-dimensional data which is represented using a grid with rows and columns. Each element in the 2D array has two indices (usually `I` and J), with the first index giving the row number and the second index giving the column number.

If you look at the 2D grid below, you will see that it is made up of five rows and three columns. The value of the first index indicates the number of the row. This means that *aNumbers[1, J]* (where value of the first index is 1) refers to the elements in the first row, while *aNumbers[2, J]* refers to the second row of elements. In contrast, the numbers in the second index indicates the column number. For example, *aNumbers[I, 3]* refers to the elements in the third column of the array.

The table below shows a visual representation of a 2D array.

	aNumbers[I,1]	aNumbers[i,2]	aNumbers[i,3]
aNumbers[1,j]	1	1	1
aNumbers[2,j]	2	4	8
aNumbers[3,j]	3	9	27
aNumbers[4,j]	4	16	64
aNumbers[5,j]	5	25	125

The numbers in the array were created by raising the index `I` (the row number) to the power of J (the column number).

The table effectively shows how 15 integer values can be represented uniquely using the same identifier, *aNumbers*, and the two indices surrounded by square brackets to distinguish between them. It shows 15 unique integer **variables** that can be accessed or be assigned new values. Variables must be declared before they can be used. How is the 2D array declared?

DECLARING A 2D ARRAY

To declare a 2D array, you can use the following syntax:

SYNTAX	EXAMPLE

2D array syntax
`var`
`  aName: Array[1..i, 1..j]`
`         of Type;`
	`var`
	`  aNumbers: Array[1..5,1..3]`
	`            of Integer;`

Where `I` gives the number of rows in the array and J gives the number of columns. As can be seen from this declaration, 2D arrays have the same general limitations as 1D arrays, namely:

- all the elements of the array must have the same type
- the array has a fixed size.

Activity 3.1 Declaring 2D arrays

Using a pen and paper, declare the following arrays.

3.1.1 10 rows and 10 columns of integers.

3.1.2 5 rows and 4 columns of strings.

3.1.3 500 rows and 3 columns of reals.

3.1.4 250 rows and 400 columns of strings.

3.1.5 an array to model weeks and days in the month of May

Have you noticed that the 2D array models displays things in a similar manner to a spreadsheet using only one data type? That data type can be any of the built-in datatypes as well as those you declare.

ACCESSING ELEMENTS IN A 2D ARRAY

To access a specific element in a 2D array, you need to give both the row and column number.

Example 3.1 Accessing and using an element in a 2D array

Use the *aNumber* array above to reference the numbers given in the examples below.

EXAMPLE	CODE
Assign 9 to *iValue.*	`iValue := aNumbers[3,2];`
Display the number 27.	`showMessage(IntToStr(aNumbers[3, 3])`
Determine the square root of 64.	`Sqrt(aNumber[4,3])`
Increase the cost by 5%.	`cost := cost + (aNumber[5,1] * 0.05)`

When working with 2D arrays, it is important to remember that the first index indicates the row number while the second index indicates the column number.

ASSIGNING VALUES TO 2D ARRAYS

Once declared, you can assign values to, and read values from the array by providing the column and row index.

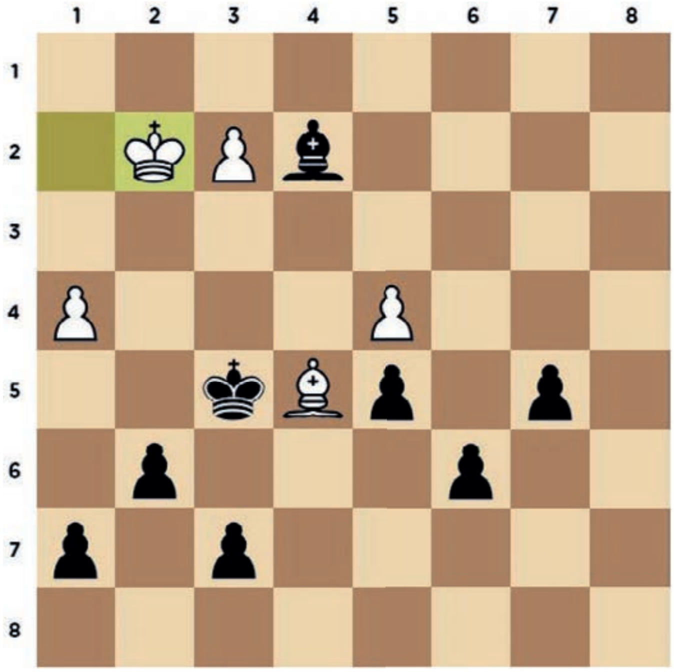
Example 3.2 Assigning values to and array element

EXAMPLE	CODE
Assign the value 92 to row 3 column 3 of *aNumbers*.	`aNumbers[3,3] := 92;`
Given 1 <= a <= 5. Assign the user input from *edtNumber* to Row a and column 1.	`aNumbers[a,1] := StrToInt(edtNumber.Text);`
Given 1 <= b <= 3. Assign the value of the spin edit to row 2 and column b.	`aNumbers[2,b] := sedNumber.Value;`
Given 1 <= a <= 5 and 1 <= b <= 3 Assign the value of 10 DIV 3 to row a column b.	`aNumbers[a,b] := 10 DIV 3;`

The activity below shows how this can be used in a real-life example.

Activity 3.2 Chess and 2D arrays

Look at the chessboard below. As you learned in the introduction of this chapter, the position of pieces on a chessboard can be represented using a 2D array with 8 rows and 8 columns.



Activity 3.2 Chess and 2D arrays *continued*

Based on this position, complete the following tasks.

3.2.1 Declare an array of strings called *aBoard*, to represent the 8 × 8 squares on the chess board.

3.2.2 Assign the position of all the pieces to the correct element in array *aBoard*. Make sure to use different strings for white and black pieces or access elements with the same pieces.

3.2.3 At the top right of the board, there are two yellow squares. These squares show that the white king moved from the first column to the second column in the previous move. How would you code this move using your array?

The names of the pieces are given in the table below:

NAME	WHITE PIECE	NAME	BLACK PIECE
w_King	♔	b_King	♚
w_Bishop	♗	b_Bishop	♝
w_Pawn	♙	b_Pawn	♟

Since the data is saved in an array, you can use the information as part of your program. For example, you can easily count which colour has the most pieces by stepping through your array and counting the first letters stored for each square ("w" for white or "b" for black).

USING LOOPS TO STEP THROUGH AN ARRAY

While assigning values one-by-one can be useful (as in the chess example), it is more common to use FOR-DO loops to assign values to an array. The nested FOR-DO loop is the perfect control structure to scroll through a 2D array. The indices must stay within the declared boundaries or an out of bound error will occur.

Use the length function to ensure that the indices remain within the boundaries.

```
var
    aNumbers : Array [1..7,1..5,1..8] of Integer;
begin
    showMessage(IntToStr(Length(aNumbers)));           //output 7
    showMessage(IntToStr(Length(aNumbers[1])));        //output 5
    showMessage(IntToStr(Length(aNumbers[1,1])));      //output 8
end;
```

Note how without any indices Length returns the first upper boundary when the lower boundary is 1. By fixing the first index, any number from 1 to 7 will work, giving the upper boundary of the second index, and so on. If a fourth index [,1..12] was given, how will the length expression change to return its upper boundary, 12?

Example 3.3 Nested-FOR-DO loop

1. Use a nested FOR-DO loop to assign the sum of the row and column numbers to each element of array *aNumbers*.

```
var
  aNumbers : Array[1..3, 1..2] of Integer;
  i, j: Integer;
begin
1:  for i := 1 to length(aNumbers) do
2:    for j := 1 to length(aNumbers[1]) do
3:      aNumbers[i, j] := i + j;
end;
```

The code will produce the following 2D array:

2	3
3	4
4	5

Visual representation of the array

To see how values are assigned to the array in this FOR-DO loop, you can use a trace table. By stepping through the code bit-by-bit, it becomes easier to see how each element of the array is assigned a value.

For the code snippet above, add more rows to the trace table below and complete tracing through the nested FOR-loop. Track the values of the following variables:

1. i
2. j
3. Name of array element (including indices)
4. Value of the array element.

Line number	i	j	aNumbers name	aNumbers[i,j] value
2		1		
3			aNumbers[1,1]	2
...				

2. Use the same nested loop pattern to extract/access data from the 2D array.

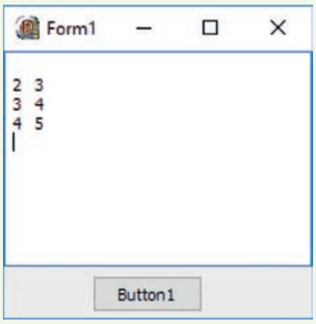
In this example we have two options:

- build a line of output and then when complete add it to the Memo. Note the line is set to null after adding to the Memo.
- build a single line which include line breaks then add.

Example 3.3 Nested-FOR-DO loop *continued*

```
for i := 1 to length(aNumbers) do
begin
 for j := 1 to length(aNumbers[1]) do
 line :=  line + format('%d   ', [aNumbers[i, j]]) ;
 Memo1.lines.add(line);
 line := '';
end;
```

```
//alternative
for i := 1 to length(aNumbers) do
begin
 for j := 1 to length(aNumbers[1]) do
 line :=  line + format('%d   ',  [aNumbers[i, j]]) ;
 line := line + sLineBreak;
end;
Memo1.lines.add(line);
```



```
2 3
3 4
4 5
```

Activity 3.3 Nested-FOR-DO loop

Now that you have seen how 2D arrays can be manipulated with nested FOR-DO loops, create the following 2D arrays using pen and paper (include the variable declarations):

3.3.1 An array containing the multiplication table for the first six numbers.

3.3.2 An array containing the powers table for the first ten numbers and three exponents. You can use the Power function to do this.

3.3.3 A 10 by 10 2D array of numbers
- Fill the array with random numbers in the range 100 to 999.
- Extract the numbers from the 2D array and in a memo component display as a 10 by 10 table.
- Insert an empty line and display the leading diagonal on the next line in the memo.

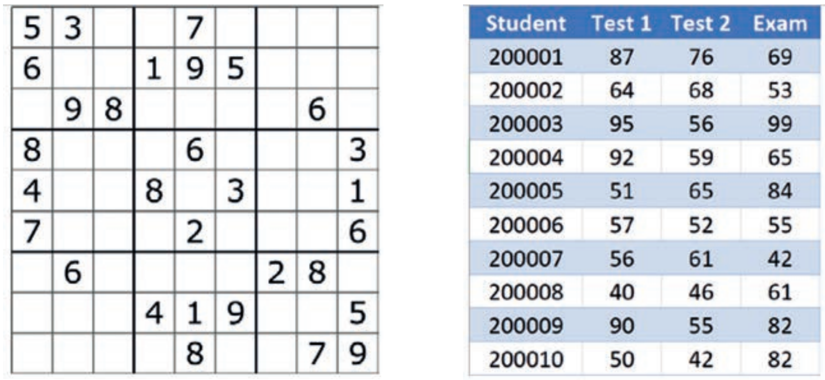
USES OF 2D ARRAYS

Up to now, you have looked at very specific uses of 2D arrays, but these arrays are incredibly useful in a number of different situations. Two of the most common general uses are when you need to store a few pieces of information about every item in a list, and when you need to store a grid of information. Here are some examples:

- An application's usernames and passwords can be read using a 2D array.
- The results of two tests and an exam for 500 students can be stored in a 500 by 3 array.
- A list of companies' incomes, expenses and balances can be store in a 2D array.

- The multiplication table is a grid that can be stored in a 2D array.
- The pixels of an image can be represented as a grid using a 2D array.
- The numbers on a sudoku puzzle can be stored in a 9 by 9 array.
- The 4 suits of cards containing 13 cards each can be stored in a 13 by 4 array.
- Various board games where the board has a grid like structure.

From the examples you will see that a 2D array is suitable for modelling, or storing, most data in a table form. A 2D array can also be used to store the data of a CSV file – as long as all of the elements have a common type.



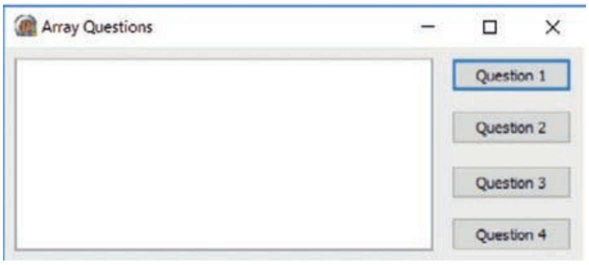
5	3			7				
6			1	9	5			
	9	8					6	
8				6				3
4			8		3			1
7				2				6
	6					2	8	
			4	1	9			5
				8			7	9

Student	Test 1	Test 2	Exam
200001	87	76	69
200002	64	68	53
200003	95	56	99
200004	92	59	65
200005	51	65	84
200006	57	52	55
200007	56	61	42
200008	40	46	61
200009	90	55	82
200010	50	42	82

Figure 3.2: *Tables and grids of information can be represented with 2D arrays*

Activity 3.4

Open the application saved in 03 – Array Questions. You should see the following user interface.



Create the following 2D arrays and display their values in the *lstAnswer* component. Make use of the tab space character (#09) to display the values from the columns separately.

3.4.1 The 6 by 6 multiplication table.

3.4.2 A 9 by 9 table containing a random value between 1 and 9 for each element.

3.4.3 A 10 by 5 table containing the sum of the indices.

3.4.4 A 7 by 5 table of randomly generated lowercase characters.

Did you know

The listbox component has a TabWidth property which determines how much space is used by the tab space character. This property has a default value of 0 which means tab space characters are not shown, unless you change the property.

For the rest of this chapter, you will be creating applications that work with 2D arrays.

3.2 2D arrays with data

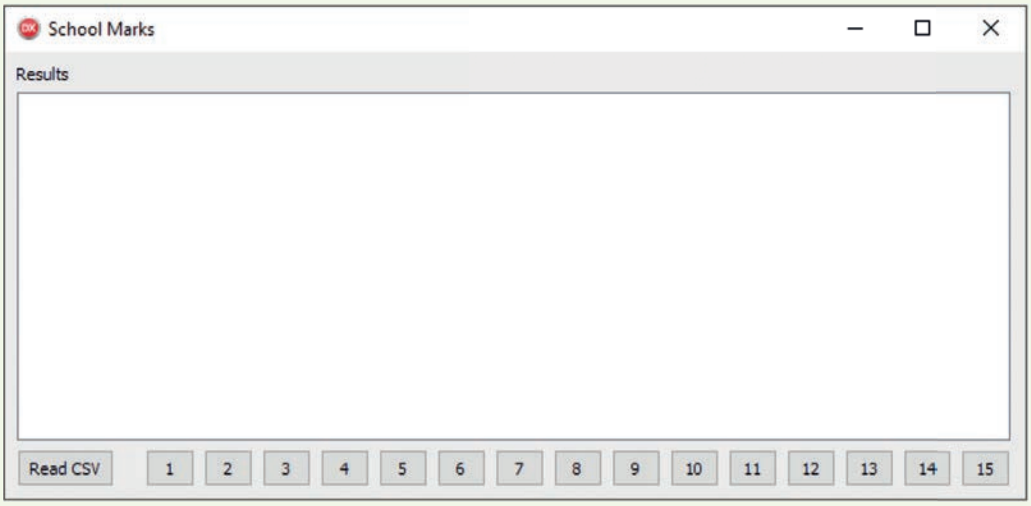
In the previous unit, the data for the arrays were either entered manually (as in the chess example) or generated automatically using numbers. An additional method can be used to add data to arrays which is to read the data from a text file or database. To see how this can be done, you will use the following dataset (saved in the **marks.csv** text file), which contains information about five students at your school.

STUDENT NUMBER	AGE	FEES OUTSTANDING	TEST 1	TEST 2	ASSIGNMENT	EXAM
200001	16	1500	75	72	97	81
200002	17	200	88	94	81	92
200003	17	350	85	53	76	84
200004	19	2300	64	59	63	52
200005	16	4000	62	74	82	71

However, before you can start working with the data you first need to add it to a 2D array.

Example 3.4 Adding data to an array

Open the project saved in your 03 – School Marks folder. You should see the following user interface.



In the following example testing for the existence of the file left out to keep the code simple.

The code for the *CSVIntoArray* procedure is given below:

Procedure code

```
procedure CSVIntoArray;
var
  sData : String;
  iComma, i, j : Integer;
  fCSV : TextFile;
  // aData : Array[1..5, 1..7] of Integer; (already declared globally)
begin
  AssignFile(fCSV, 'marks.csv');
  Reset(fCSV);
  ReadLn(fCSV, sData);
  for i := 1 to 5 do
```

Example 3.4 Adding data to an array *continued*

```
  begin
    ReadLn(fCSV, sData);
    for j := 1 to 7 do
    begin
      iComma := Pos(',', sData);
      if iComma := > 0 then
      begin
       aData[i,j] := StrToInt(Copy(sData,1,iComma-1));
       Delete(sData,1,iComma);
     end
     else
      aData[i,j] := StrToInt(Copy(sData,1,Length(sData)));
    end;
   end
end;
```

Activity 3.5

Based on the code in the example 3.4 above, answer the following questions:

3.5.1 What is the purpose of this procedure?

3.5.2 Describe the purpose of the following lines:

a. `for i := 1 to Length(aData) do`

b. `ReadLn(fCSV, sData);`

c. `for j := 1 to Length(aData[1]) do`

d. `iComma := AnsiPos(‘,’, sData);`

e. `aData[i, j] := StrToInt(Copy(sData, 1, iComma - 1));`

f. `Delete(sData, 1, iComma);`

3.5.3 When will the value of *iComma* be equal to 0?

3.5.4 Why is there a ReadLn function before the loop starts?

3.5.5 In the TStringlist version, describe the purpose of the following lines:

a. `inList := TStringList.Create;`

b. `inList.DelimitedText := sData;`

c. `aData[i,j] := strtoint(inList.Strings[j-1]);`

3.5.6 Why does *inList.Strings[j-1]* have j-1 as the index?

3.5.7 Create an event for the [Read CSV] button that displays the data from the array in the list box. Make sure to run the **CSVIntoArray** procedure at the start of this event.

WORKING WITH ROWS OF DATA

In your 2D array above, each row contains information about a single entity (in this case, a student). This is typical when you read data from a CSV file or a database into an array. In contrast, each column contains information about a specific field. For example, the first column contains all the student numbers, while the second column contains the students' ages.

When you want to work with the data of a row, you can keep the row number (or first index) constant while using a FOR-DO loop to update the values of the second index.

Example 3.5 Find the largest mark in the first row

Stepping through a row

```
iLargest := 0;
for j := 4 to Length(aData) do
begin
  if aData[1, j] > iLargest then
    iLargest := aData[1, j];
end;
```

In this snippet, a FOR-DO loop was created that repeats once for each column of the array. The FOR-DO loop's counter `I` is then used to iterate through each element of the first row of *aData* in order to find the largest value.

To see how this technique can be applied to your school's dataset, complete the following activity.

Activity 3.6 Stepping through a row

Using the 2D array created for *School Marks*, create events for the [1] and [2] buttons to calculate and display the following:

3.6.1 The 3rd student's average mark for his tests, assignment and exam.

3.6.2 The 2nd student's highest mark.

WORKING WITH COLUMNS OF DATA

As the previous case study shows, stepping through the values of a single row is useful when you want to make comparisons or do calculations for a specific item in your array. However, when you want to do calculations or make comparisons using all the values of a single field, you need to step through a column.

To step through a column, you keep the second index constant while using a FOR-DO loop to update the values of the first index.

Example 3.6 Determine the age of the oldest student

Stepping through the first column

```
iAge := 0;
for i := 1 to length(aData) do
begin
  if aData[i, 2] > iAge then
    iAge := aData[i, 2];
end;
```

Take note, the FOR-DO loop counter was changed to `I`. This is because the letter `I` is used for the first index of a 2D array. Inside the FOR-DO loop, the second index (column number) of the array is now held constant while the first index (row number) changes as the loop repeats.

To use this in an application, complete the following activity.

Activity 3.7 Stepping through a column

Using the 2D array created for *School Marks*, create events for the [3] and [4] buttons to calculate and display the following:

3.7.1 The total fees outstanding.

3.7.2 The average age of students.

Once complete, save the project in the 03 – School Marks folder.

STEPPING THROUGH ALL THE DATA

The final way to step through the data is to use a nested FOR-DO loop statement. With a nested FOR-DO loop statement, the outside loop can be used to systematically select each of the rows or columns. The inside FOR-DO loop can be used to step through the values of the selected columns or rows. By combining two FOR-DO loops, you can therefore step through all the data in a table.

Example 3.7 Determines the number of A's scored by the class

Stepping through all the data

```
iCount := 0;
for i := 1 to Length(aData) do
  for j := 4 to length(aData[1]) do
    if aData[i, j] > 80 then
      Inc(icount);
lstResults.Items.Add(IntToStr(iCount));
```

With this information in mind, complete the following activity.

Activity 3.8 Stepping through all the data

Using the 2D array created for *School Marks*, create events for the [5] and [6] buttons to calculate and display the following:

3.8.1 The highest single mark between the tests, assignment and exams.

3.8.2 The average of all the marks.

Once complete, save the project in the *School Marks* folder.

Activity 3.9

Using the 2D array created for *School Marks*, create events for the [7] to [15] buttons to calculate and display the following:

3.9.1 The 5th student's lowest mark.

3.9.2 If the 4th student obtained their personal highest mark for the assignment.

3.9.3 If the 2nd student's average is above 90.

3.9.4 The age of the oldest student.

3.9.5 The student number of the student who obtained the highest mark for the exam.

3.9.6 The student number with the highest outstanding fees.

3.9.7 The student with the highest average mark.

3.9.8 The lowest overall mark between all tests, exams and assignments.

3.9.9 The lowest mark of the student with the student number "200004".

Once complete, save the project in the *School Marks* folder.

UNIT

3.3 Applications for two-dimensional arrays

Now that you are more comfortable working with a two-dimensional array, you can use that knowledge to create the game Sudoku. In Sudoku, players start with a 9 × 9 grid with a few numbers inside each 3 × 3 square in the grid. The goal of the game is to fill each empty space in the grid with a number between 1 and 9, in such a way that there are no duplicate numbers within any:

- Row
- Column
- Diagonal
- 3 × 3 square

The image below shows an example of an incomplete and completed Sudoku game.

ARRAYS IN DELPHI - PART 1

https://www.youtube.com/watch?v=0MuOM3cT30M

			9	5				3
	5				3			
7	6			4			8	
1	2					9	4	5
		8		7		3		
6	3	4					1	8
	8			3			9	2
			6				3	
3				2	8			

8	4	1	9	5	7	2	6	3
9	5	2	8	6	3	4	7	1
7	6	3	2	4	1	5	8	9
1	2	7	3	8	6	9	4	5
5	9	8	1	7	4	3	2	6
6	3	4	5	9	2	7	1	8
4	8	6	7	3	5	1	9	2
2	7	5	6	1	9	8	3	4
3	1	9	4	2	8	6	5	7

Figure 3.3: *A starting and completed game of Sudoku*

To create a game of Sudoku in Delphi, you will need to do the following:

- Create a 9 × 9 array to store the user's values and a grid to display the values.
- Create a Duplicate Checker algorithm which checks if there are any duplicates in a list of numbers.
- Create a nested-loop that sends each of the nine rows to the Duplicate Checker.
- Create a nested-loop that sends each of the nine columns to the Duplicate Checker.
- Create a nested-loop that sends each of the nine 3 × 3 squares to the Duplicate Checker.
- Create a function to save or load starting positions.

ARRAYS IN DELPHI - PART 2

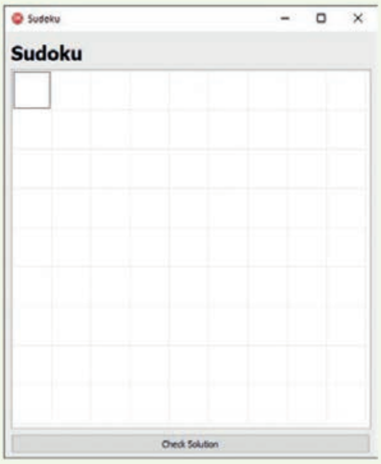
https://www.youtube.com/watch?v=-KKRZqlm09U

This chapter will complete the first four of these tasks. However, if you would like, you can complete the last two tasks on your own.

Example 3.8 Setting up Sudoku

To create the Sudoku game, you need to start by creating a grid to display the results and an array to store the results. You also need to build the link between the grid and the array, allowing values entered into the grid to be stored in the array. To do this:

1. Open the project saved in the 03 – Sudoku folder. You should see the following user interface.

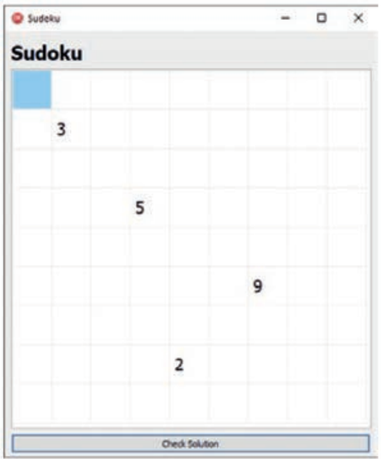


TASK/ALGORITHM

In the OnClick event of button [Check Solution]

Transfer digits from the grid to the 2D array

- Declare 9 × 9 array called *aNumbers.*
- Declare loop variables j and k and a 1D array of integers called *aToCheck*.
- Provide code to transfer the values in the grade to array *aNumbers*. Insert 0 if the cell is empty.
- Test with the values below:



IMPLEMENTATION

```
var
 aNumbers:  Array[0..8, 0..8] of Integer;
 j: integer;
 k: integer;
 aToCheck : Array[0..8] of Integer;
Begin
{transfer from grid to array}
for j := 0 to 8 do
  for k := 0 to 8 do
if grdSudoku.Cells[k, j] <> '' then
      aNumbers[j, k] :=
StrToInt(grdSudoku.Cells[k, j])
    else
      aNumbers[j, k] := 0;
end;
```

Save and test your application. To make sure the values from the string grid are being converted, add numbers or letters to your grid and click on the [Check Solution] button. The application should give an error for the letters but not the numbers.

Checking for duplicates

- Declare the function *hasDuplicates* to receive an array of integers in the private section.
- Set functions return type to Boolean.
- Press *Ctrl +Shift+C* to generate the function stub in the implementation section.
- Declare local variables called *bHasDuplicates*, “i” and “k”.
- *bHasDuplicates* to FALSE.
- Create a for-loop for “j” from 1 to 7
- Test if number is 0, skip else enter inner loop.
- Create a nested FOR-loop for “k” from “j” + 1 to 8.
- If number [j] = number [k] then set *bHasDuplicates* to TRUE.
- Return result.

```
private
function hasDuplicates(aCheckMe: Array of
integer): boolean;


implementation
function TfrmSudoku.hasDuplicates(aCheckMe:
Array of integer): boolean;
var
 j: integer;
 k: integer;
 bHasDuplicates : Boolean;
begin
bHasDuplicates := false;
for j := 0 to 7 do
 if aCheckMe[j] > 0 then
 for k := j + 1 to 8 do
   if aCheckMe[j] = aCheckMe[k] then
       bHasDuplicates := True;
result := bHasDuplicates;
end;
```

Save and test your application. Since you are not passing any values to this function, you cannot test the *HasDuplicate* function yet. However, by running the application you can make sure there are no syntax errors.

Well done! You now have a working duplicate checker.

In the OnClick event of button [Check Solution]

Checking the rows for duplicates

Declare boolean variable *bHasDuplicates*.

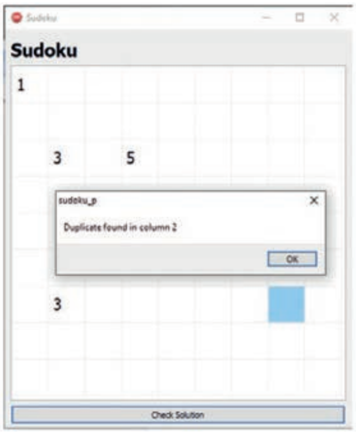
```
for j := 0 to 8 do
begin
  for k := 0 to 8 do
    aToCheck[k] := aNumbers[j, k];
  bHasDuplicates := HasDuplicates(aToCheck);
  if bHasDuplicates then
  ShowMessageFmt('Duplicate found in row %d',[j
  + 1]);
end;
```

Checking the columns for duplicates

Repeat the code above and replace the highlighted line with the ones in the right hand side.

Note how the inner loop is now referring to the column and not to the row.

```
 aToCheck[k] := aNumbers[k, j];
 if bHasDuplicates then
 ShowMessageFmt('Duplicate found in column %d',[j
 + 1]);
```



Duplicate found in column 2

Challenge

- Create a nested-loop that sends each of the nine 3 × 3 squares to the Duplicate Checker.
- Create a function to save or load starting positions.

Activity 3.10

Update your Sudoku game to inform the player when he or she wins the game. To win the game, the following conditions need to be met:

3.10.1 There should be no 0 values in the array.

3.10.2 There should be no duplicates in any rows or columns.

Once done, save your application in the *Sudoku* folder.

CONSOLIDATION ACTIVITY Chapter 3: Two-dimensional arrays

QUESTION 1

1.1 Which of the following lines can be used to access an element from a 2D array?

a. aNumbers[4;3]

b. aNumbers[4][3]

c. aNumbers[4,3]

d. aNumbers[4,3,7]

1.2 Write the code you would use to declare a 2D Boolean array with 6 rows and 15 columns called *aWinLose*.

1.3 A 2D array called *aRainfall* has been declared to contain the average rainfall per month for five towns.

a. What structure must be used to access the monthly rainfall for all towns.

b. Write the code used to show the monthly rainfall figures in table form.

1.4 A two-dimensional array called *aStock* has been used to record the quantities of the four items for the four departments, as shown in the table below.

	AFRIKAANS	HISTORY	TOURISM	DESIGN	TOTAL
Desktop	4	0	12	0	
Laptop	0	2	1	5	
Printer	2	1	1	0	
Scanner	0	1	0	1	

Write pseudocode to calculate the total stock per item and store these values in the array.

1.5 Three parallel arrays called *aPassengers*, *aStations* and *aMonths* have been declared to contain the number of passengers that pass through a train station each month.

a. Why it is NOT possible to use a 2D array instead of three parallel arrays to store this data as given?

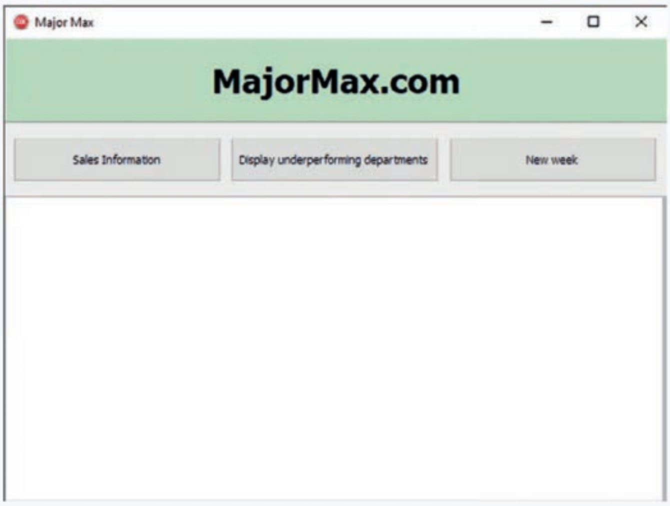
b. What changes would you make to store the data into a 2D array?

QUESTION 2

For this application, open the project saved in the 03 – Question 2 folder. Once done, save your project in the same folder.

The online-shopping website of MajorMax allows customers to buy items online from various departments at their store. The manager of the company must analyse their weekly sales figures.

The GUI below shows the interface of the program used by MajorMax to keep track of their weekly sales figures.

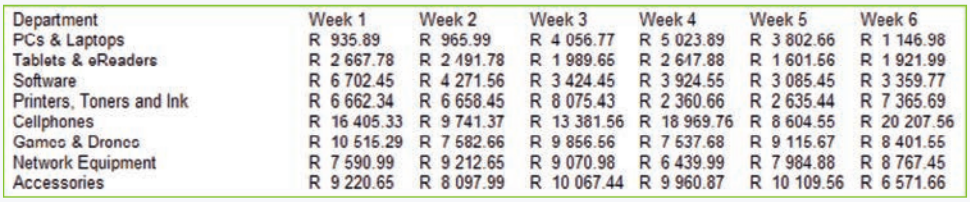


CONSOLIDATION ACTIVITY Chapter 3: Two-dimensional arrays *continued*

The program contains code that declares two arrays, *aDepartments* and *aSales*.

- The *aDepartments* array contains the names of the various departments that sell products online
- The *aSales* array is a two-dimensional array that contains the sales figures for the first six weeks of the year for each department. The rows in the array represent the various departments and the columns represent various weeks.

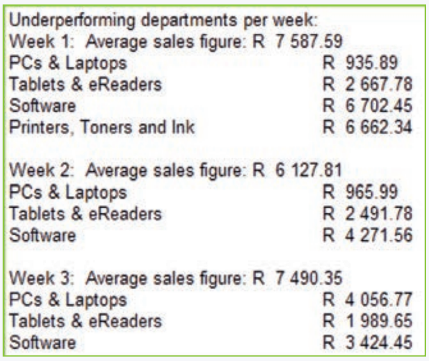
2.1 When the [Sales Information] button is clicked, display the content of the *aSales* array with suitable headings in the *redOutput* component provided. All monetary values must be displayed in currency format with TWO decimal places, as shown below.



```
Department               Week 1       Week 2      Week 3       Week 4       Week 5       Week 6
PCs & Laptops            R  935.89    R  965.99   R 4 056.77   R 5 023.89   R 3 802.66   R 1 146.98
Tablets & eReaders       R 2 667.78   R 2 491.78  R 1 989.65   R 2 647.88   R 1 601.56   R 1 921.99
Software                 R 6 702.45   R 4 271.56  R 3 424.45   R 3 924.55   R 3 085.45   R 3 359.77
Printers, Toners and Ink R 6 662.34   R 6 658.45  R 8 075.43   R 2 360.66   R 2 635.44   R 7 365.69
Cellphones               R 16 405.33  R 9 741.37  R 13 381.56  R 18 969.76  R 8 604.55   R 20 207.56
Games & Drones           R 10 515.29  R 7 582.66  R 9 856.56   R 7 537.68   R 9 115.67   R 8 401.55
Network Equipment        R 7 590.99   R 9 212.65  R 9 070.98   R 6 439.99   R 7 984.88   R 8 767.45
Accessories              R 9 220.65   R 8 097.99  R 10 067.44  R 9 960.87   R 10 109.56  R 6 571.66
```

2.2 A report of all underperforming departments per week is required. A department is underperforming when their sales figure is lower than the average sales for all the departments for that week. When the [Display underperforming departments] button is pressed, display a report with the sales figures of all underperforming departments. All monetary values must be displayed in currency format with TWO decimal places.

The output below shows an example of an underperforming report for the first three weeks:

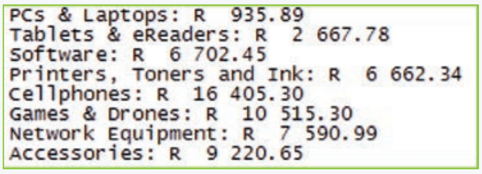


```
Underperforming departments per week:
Week 1:  Average sales figure: R  7 587.59
PCs & Laptops                       R  935.89
Tablets & eReaders                  R  2 667.78
Software                            R  6 702.45
Printers, Toners and Ink            R  6 662.34

Week 2:  Average sales figure: R  6 127.81
PCs & Laptops                       R  965.99
Tablets & eReaders                  R  2 491.78
Software                            R  4 271.56

Week 3:  Average sales figure: R  7 490.35
PCs & Laptops                       R  4 056.77
Tablets & eReaders                  R  1 989.65
Software                            R  3 424.45
```

2.3 Currently the data in the *aSales* array represents the sales figures for the first six weeks of the year. Before the sales figures for the next week (Week 7) can be added to the array and analysed, the current data for Week 1 must be backed up to a text file. To do this:

a. Use Delphi code to create a new text file. The name of the text file is the number of the week of the sales figures that are archived. For example, if the sales figures for Week 1 are archived in the file, then the name of the text file will be *Week 1.txt*.

b. Save the data (with headings) to the new text file, a shown below.

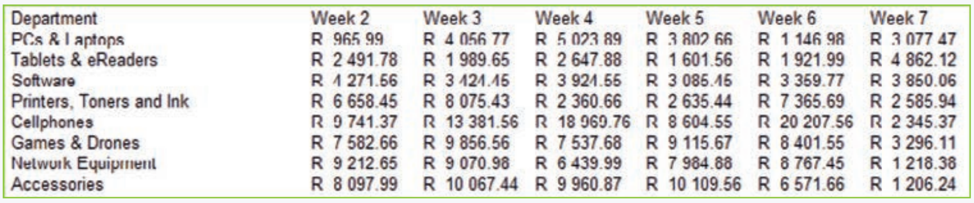


```
PCs & Laptops: R  935.89
Tablets & eReaders: R  2 667.78
Software: R  6 702.45
Printers, Toners and Ink: R  6 662.34
Cellphones: R  16 405.30
Games & Drones: R  10 515.30
Network Equipment: R  7 590.99
Accessories: R  9 220.65
```

c. When the data for Week 1 has been archived, the data for Week 2 in the arrSales array must be moved to the position of Week 1 in the array; the data for Week 3 must be moved to Week 2, and so on.

d. For test purposes, the sales data for the new week must be randomly generated values between R500 and R5 000.

e. Use code to update the labels used to display the number of the week. The new report should look as follows.



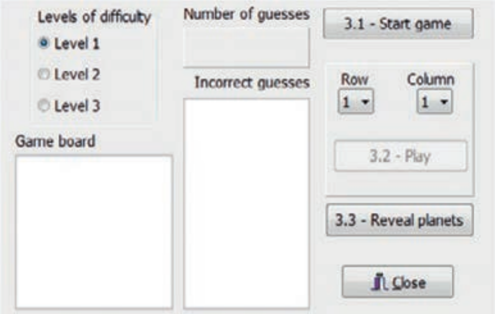
Department	Week 2	Week 3	Week 4	Week 5	Week 6	Week 7
PCs & Laptops	R 965.99	R 4 056.77	R 5 023.89	R 3 802.66	R 1 146.98	R 3 077.47
Tablets & eReaders	R 2 491.78	R 1 989.65	R 2 647.88	R 1 601.56	R 1 921.99	R 4 862.12
Software	R 1 271.56	R 3 121.16	R 3 921.55	R 3 085.16	R 3 359.77	R 3 850.06
Printers, Toners and Ink	R 6 658.45	R 8 075.43	R 2 360.66	R 2 635.44	R 7 365.69	R 2 585.94
Cellphones	R 9 741.37	R 13 381.56	R 18 969.76	R 8 604.55	R 20 207.56	R 2 345.37
Games & Drones	R 7 582.66	R 9 856.56	R 7 537.68	R 9 115.67	R 8 401.55	R 3 296.11
Network Equipment	R 9 212.65	R 9 070.98	R 6 439.99	R 7 984.88	R 8 767.45	R 1 218.38
Accessories	R 8 097.99	R 10 067.44	R 9 960.87	R 10 109.56	R 6 571.66	R 1 206.24

QUESTION 3

For this application, open the project saved in the 03 – Question 3 folder. Once done, save your project in the same folder.

A new game called Galaxy Explore is planned and needs to be developed. The purpose of the game is to prepare a grid with a number of randomly placed planets that are not visible to the player. The player must then guess the position of the planets on the grid. The grid will be referred to as the *Game board*.

The GUI below shows an early version of the user interface for the program.



The program must do the following:

- Populate the 2D array when the game starts.
- Allow the user to guess the positions of invisible planets placed randomly on the game board.
- Determine whether the player has won or lost and terminate the game. The player wins when he/she identifies two planets on the game board within five guesses.

3.1 When the [Start game] button is pressed:

a. The [Play] button must be enabled and the rich edit components must be cleared.

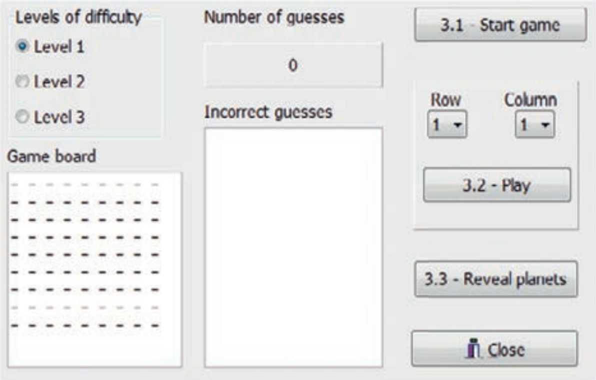
b. A dash character (-) represents an open space in the *aGame* and a hash character (#) represents a planet. The 2D array must first be populated with open-space characters.

c. The content of the *aGame* array must be displayed in the game board area.

d. The *aGame* array must be updated to include the appropriate number of planets. The level of difficulty selected from the radio group *rgbQ2* determines the number of planets. The following rules apply:

- Difficulty level 1: 50 positions in the array must be replaced by planets (#).
- Difficulty level 2: 40 positions in the array must be replaced by planets (#).
- Difficulty level 3: 30 positions in the array must be replaced by planets (#).

e. The value '0' must be displayed on the panel *pnlQ2NumberOfGuesses*, which indicates the total number of guesses. The image below shows the user interface after the [Start game] button is pressed.



0

```
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
```

3.2 When the [Play] button is clicked:

a. The player guesses the position of a planet by selecting a row number and a column number from the combo boxes provided.

b. If the position of a planet is guessed correctly:

- Update the display to show the position of the planet (#) that was guessed correctly.
- Update the number of correct guesses on the panel *pnlQ2NumberOfGuesses*.

c. In the *aGame* array, replace the planet character (#) that was guessed correctly with the “Y” character to show which planets were identified during the game play session. This information is required for the [Reveal Planets] button.

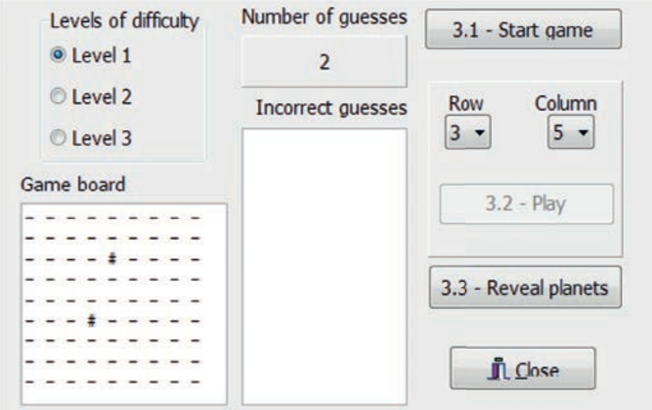
d. If the position guessed is NOT the position of a planet, display the row and column values of the incorrect guess in the *redQ2Incorrect* output area.

e. Allow the player to guess the positions of planets repeatedly until he or she wins or loses the game.

- A game is won as soon as the positions of two planets are correctly guessed.
- A game is lost if fewer than two planets are guessed within the allowed five guesses.

f. Use a message box to display a suitable message based on the outcome of the game, either ‘Game won’ or ‘Game lost’.

g. Disable the [Play] button when the game is over. The image below shows the output if the positions of two planets were guessed correctly within two guesses:



Levels of difficulty

● Level 1

○ Level 2

○ Level 3

Number of guesses: 2

Incorrect guesses

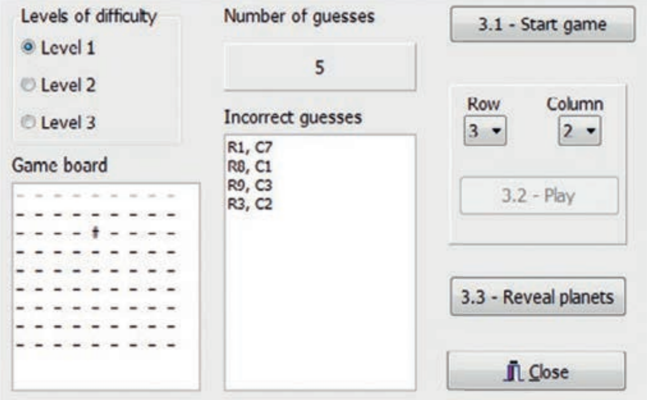
Row: 3

Column: 5

Game board

```
- - - - - - - - -
- - - - - - - - -
- - - - # - - - -
- - - - - - - - -
- - - - - - - - -
- - - # - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
```

The image below shows the output if the player lost the game. The position of only one planet was guessed correctly within five guesses:



Levels of difficulty

● Level 1

○ Level 2

○ Level 3

Number of guesses: 5

Incorrect guesses

```
R1, C7
R8, C1
R9, C3
R3, C2
```

Row: 3

Column: 2

Game board

```
- - - - - - - - -
- - - - - - - - -
- - - - # - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
- - - - - - - - -
```

3.3 When the [Reveal planets] button is clicked, write code to display the game board with all the randomly placed planets revealed.

TERM 3

CHAPTER 4

DATABASES AND SQL

CHAPTER UNITS	
Unit 4.1	Select and sort columns
Unit 4.2	Select columns and rows
Unit 4.3	Calculated columns
Unit 4.4	Aggregate functions
Unit 4.5	Data maintenance
Unit 4.6	Querying two tables
Unit 4.7	Database applications

WHAT IS SQL

https://www.youtube.com/watch?v=hWU2pvj_xlc

Learning outcomes

At the end of this chapter you should be able to

- select columns and calculated fields
- select columns and rows using conditions
- use functions and aggregate functions when selecting data
- update, add and delete data using SQL statements
- query one or two tables using SQL statements
- use the data from an SQL statement in a Delphi application.

INTRODUCTION

SQL stands for Structured Query Language, which is a standardised language that can be used to work with almost all **relational databases**. This includes database management software (DBMS) like Microsoft SQL Server, Oracle, SQLite, MySQL and Microsoft Access.

New words

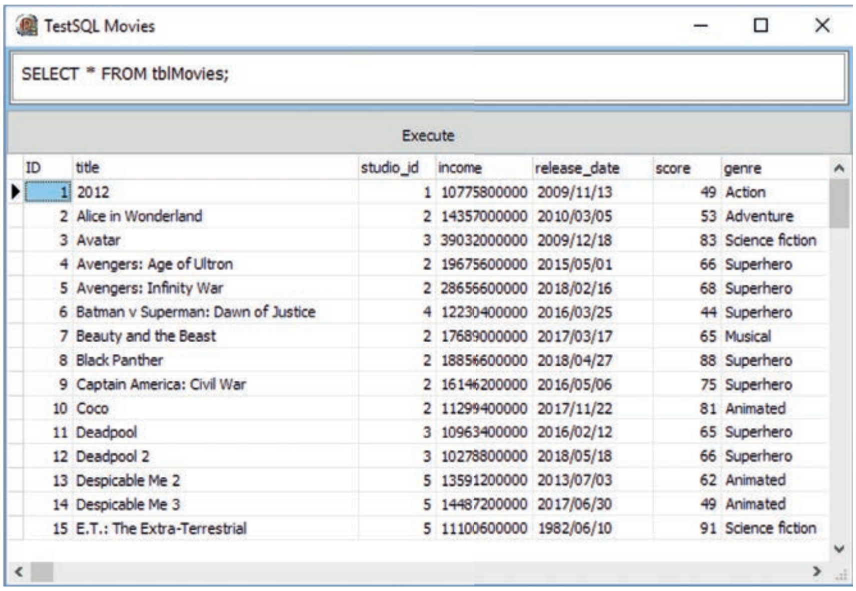
relational database – a database structured to recognise relations between stored items of information

SQL can also be used inside most programming languages, including Delphi, to communicate with databases. This allows you to complete almost any task on a database, including:

- adding, changing and deleting records in a table
- selecting data/records from tables
- displaying sorted records selected from a Database
- performing calculations on fields and records in tables
- grouping data/records from tables
- joining tables.

In this chapter, you will learn how to use SQL to do these tasks, both inside and outside of Delphi. For each unit inside this chapter, you will look at what the different SQL commands do, what their syntax is, and how they can be used in Delphi. To do this, your teacher will provide you with a database containing information on the 100 most successful movies in history, together with an executable file (**MoviesSQL.exe**). You will also be given a second database containing healthcare data on carnivores, together with the executable file (**CarnivoresSQL.exe**).

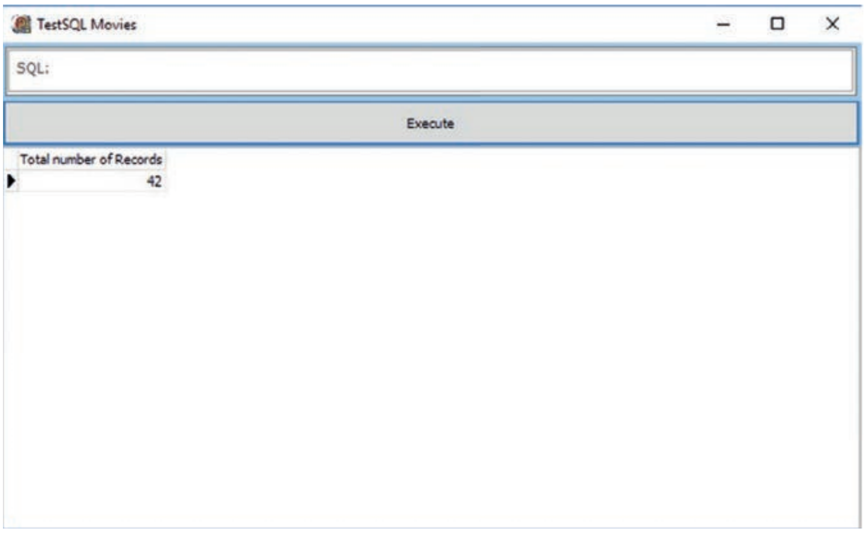
Illustrated below are some screenshots of the databases that you will use to test your SQL statements.



```
SELECT * FROM tblMovies;
```

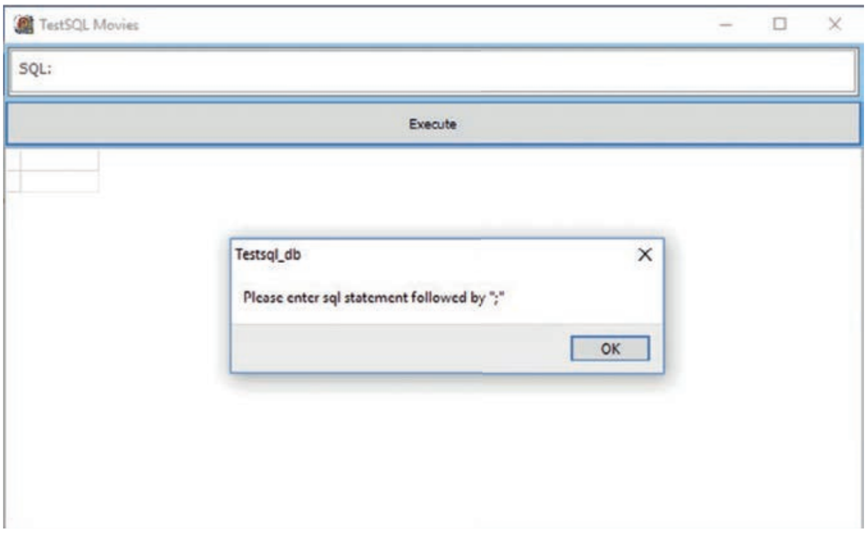
ID	title	studio_id	income	release_date	score	genre
1	2012	1	10775800000	2009/11/13	49	Action
2	Alice in Wonderland	2	14357000000	2010/03/05	53	Adventure
3	Avatar	3	39032000000	2009/12/18	83	Science fiction
4	Avengers: Age of Ultron	2	19675600000	2015/05/01	66	Superhero
5	Avengers: Infinity War	2	28656600000	2018/02/16	68	Superhero
6	Batman v Superman: Dawn of Justice	4	12230400000	2016/03/25	44	Superhero
7	Beauty and the Beast	2	17689000000	2017/03/17	65	Musical
8	Black Panther	2	18856600000	2018/04/27	88	Superhero
9	Captain America: Civil War	2	16146200000	2016/05/06	75	Superhero
10	Coco	2	11299400000	2017/11/22	81	Animated
11	Deadpool	3	10963400000	2016/02/12	65	Superhero
12	Deadpool 2	3	10278800000	2018/05/18	66	Superhero
13	Despicable Me 2	5	13591200000	2013/07/03	62	Animated
14	Despicable Me 3	5	14487200000	2017/06/30	49	Animated
15	E.T.: The Extra-Terrestrial	5	11100600000	1982/06/10	91	Science fiction

Figure 4.1: *A snippet of the database you will be using (created using* ***MoviesSQL.exe****)*



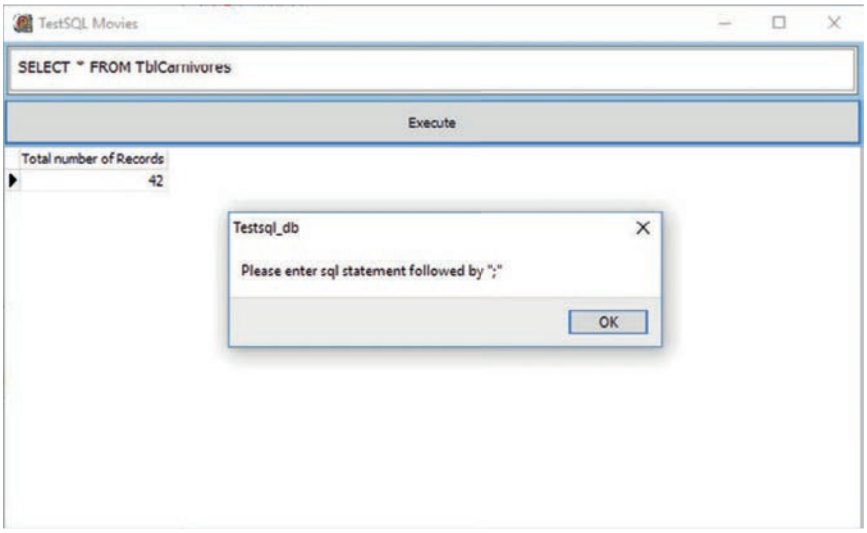
Total number of Records
42

Figure 4.2: *Opening screen of the database you will be using (created using* ***MoviesSQL.exe****)*



Please enter sql statement followed by ";"

Figure 4.3: *No SQL statement entered for the database you will be using (created using MoviesSQL.exe)*

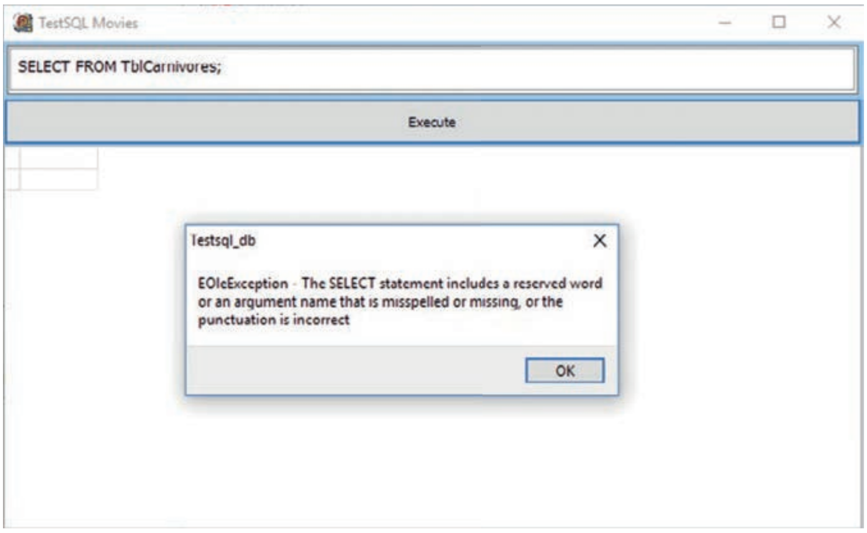


```
SELECT * FROM TblCarnivores
```

Total number of Records
42

Please enter sql statement followed by ";"

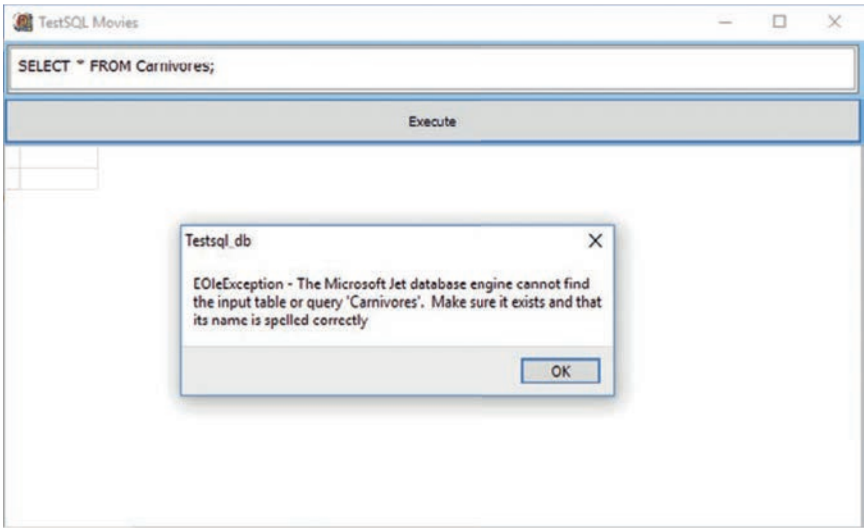
Figure 4.4: *Missing semicolon in the database you will be using (created using CarnivoresSQL.exe)*



```
SELECT FROM TblCarnivores;
```

EOleException - The SELECT statement includes a reserved word or an argument name that is misspelled or missing, or the punctuation is incorrect

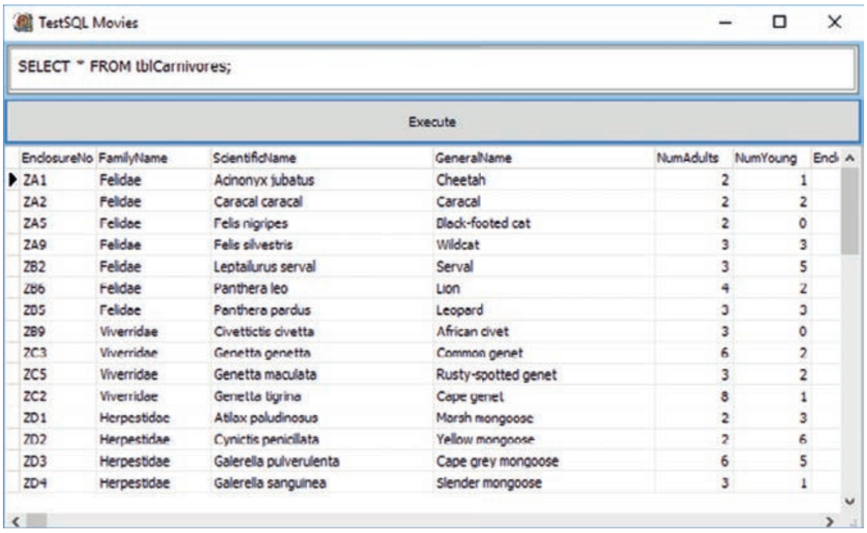
Figure 4.5: *Syntax Error in the database you will be using (created using CarnivoresSQL.exe)*



SELECT * FROM Carnivores;

EOleException - The Microsoft Jet database engine cannot find the input table or query 'Carnivores'. Make sure it exists and that its name is spelled correctly

Figure 4.6: *Missing or misspelled table in the database you will be using (created using CarnivoresSQL.exe)*



SELECT * FROM tblCarnivores;

EnclosureNo	FamilyName	ScientificName	GeneralName	NumAdults	NumYoung	Encl
ZA1	Felidae	Acinonyx jubatus	Cheetah	2	1	
ZA2	Felidae	Caracal caracal	Caracal	2	2	
ZA5	Felidae	Felis nigripes	Black-footed cat	2	0	
ZA9	Felidae	Felis silvestris	Wildcat	3	3	
ZB2	Felidae	Leptailurus serval	Serval	3	5	
ZB6	Felidae	Panthera leo	Lion	4	2	
ZB5	Felidae	Panthera pardus	Leopard	3	3	
ZB9	Viverridae	Civettictis civetta	African civet	3	0	
ZC3	Viverridae	Genetta genetta	Common genet	6	2	
ZC5	Viverridae	Genetta maculata	Rusty-spotted genet	3	2	
ZC2	Viverridae	Genetta tigrina	Cape genet	8	1	
ZD1	Herpestidae	Atilax paludinosus	Marsh mongoose	2	3	
ZD2	Herpestidae	Cynictis penicillata	Yellow mongoose	2	6	
ZD3	Herpestidae	Galerella pulverulenta	Cape grey mongoose	6	5	
ZD4	Herpestidae	Galerella sanguinea	Slender mongoose	3	1	

Figure 4.7: *Successly loading data in the database you will be using (created using CarnivoresSQL.exe)*

UNIT

4.1 Select and sort columns

The single most important SQL command you will learn is the SELECT command. This command selects the data from a database, allowing you to either use it in an application or show it to the user. For example, your school might use a database containing all the learners' information and marks. When they create your report card, they use the SELECT command to select specific information (certain fields) about you (such as your name, surname and marks) and place that in a report.

Other examples of SELECT being used include:

- social networks selecting messages and status updates relevant to you
- games selecting the correct enemies and graphics to show
- music applications selecting the correct song from the database.

In this unit, you will learn how to:

- select different fields
- select distinct values
- order selected data

SORTING DATABASES IN DELPHI

https://www.youtube.com/watch?v=kV2g0yKeB7s

Did you know

It is convention, but not required, to write SQL commands like SELECT and INSERT in uppercase letters. This makes it easier to read SQL statements. The basic structure of a SQL query is *always* the same with the *same order* in keywords even if some of the clauses are omitted.

SELECT FIELDS

The SELECT statement can consist of different clauses depending on the information you want to display. A SELECT query doesn't change the values in the database, it only displays what is in the database.

SELECT syntax

```
SELECT field_name1, field_name2, …, field_name100 FROM table_name;
```

The fields are separated with a comma (,) and only the listed fields will be displayed.

Looking at the *tblMovies* table from your example database, you could select and show the title and income of the tblMovies table. To do this, you would use the following query:

Selecting movies and income

```
SELECT title, income FROM tblMovies;
```

Title and income are field names and tblMovies is the name of the table.

Did you know

The asterisk (*) symbol can be used as a wildcard in place of field names to select *all* the fields.

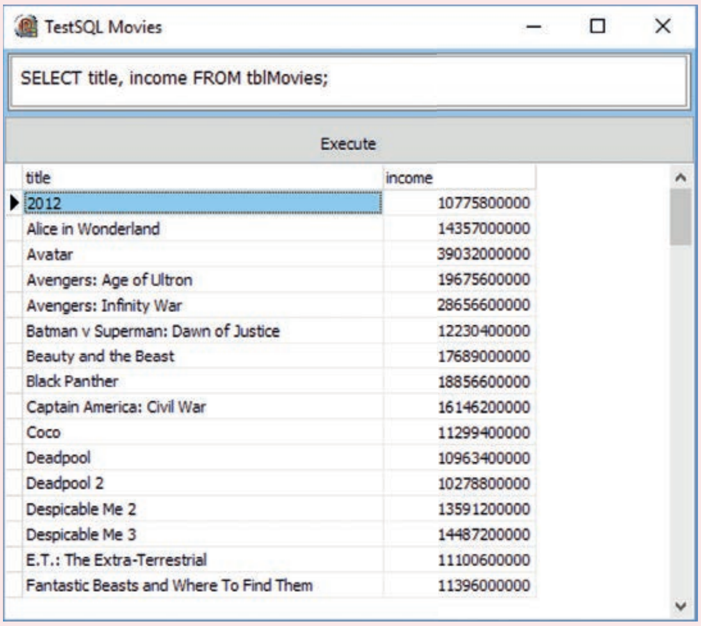
Example 4.1 Running SQL Queries in the testSQL app

1. Open the folder 04 – Movies; make sure the database **Movies.mdb** is present; and run the MoviesSQL.exe file by double-clicking it.
2. Enter the following SQL query in the SQL edit box at the top of the interface.

SELECT query

```
SELECT title, income FROM tblMovies;
```

3. Click on the [Execute] button under the SQL edit box. You should now see the data from your query displayed in the Query window below.



```
SELECT title, income FROM tblMovies;
```

title	income
2012	10775800000
Alice in Wonderland	14357000000
Avatar	39032000000
Avengers: Age of Ultron	19675600000
Avengers: Infinity War	28656600000
Batman v Superman: Dawn of Justice	12230400000
Beauty and the Beast	17689000000
Black Panther	18856600000
Captain America: Civil War	16146200000
Coco	11299400000
Deadpool	10963400000
Deadpool 2	10278800000
Despicable Me 2	13591200000
Despicable Me 3	14487200000
E.T.: The Extra-Terrestrial	11100600000
Fantastic Beasts and Where To Find Them	11396000000

Figure 4.8: *The Query1 table only shows the "title" and "income" fields*

Congratulations, you have just created your first SQL query!

Example 4.2 More examples to try

Use the procedure above to run these SQL statements.

Note that once the program is running, you may change the SQL and execute the new statement without restarting the program. Use the Snipping tool to record your results.

SHOW ALL:	SQL
1. The titles and release dates of the movies in the tblMovies table.	`SELECT title, release_date FROM tblMovies;`
2. The names and cities in the tblStudios table.	`SELECT name, city FROM tblStudios;`
3. The fields in the tblMovies table.	`SELECT * FROM tblMovies;`

Activity 4.1

4.1.1 For each of the example queries, how many records were selected?

4.1.2 How do the answers compare to the total number of records in the original tables?

4.1.3 In your own words describe what the "SELECT fields FROM table" statement do?

4.1.4 What will be the difference in output if number 1 in the table above is changed to SELECT release_date, title FROM *tblMovies*.

4.1.5 A group of learners were asked to display the title, genre and income of all the movies in the database, comment on the answers given below. If you think it's wrong then state why?

a. SELECT FROM *tblMovies* title, genre, income;

b. SELECT movies, genre, income FROM *tblMovies*;

c. SELECT title, genre, income, FROM *tblMovies*;

d. SELECT genre, income, title FROM *tblMovies*;

4.1.6 Using a pen and paper, write SQL queries to select the following data.

a. The *generalName* and *numadults* from the *tblCarnivores*

b. The *scientificName* and the *generalName* from the *tblCarnivores*

c. The *visitDate* and *reasonForVisit* from the *tblVetVists*

d. The *generalName*, *numadults*, *numYoung* and *enclosureSize* from the *tblCarnivores*

e. The *reasonForVisit* and *followUp* from the *tblVetVists*

Once you have written down the queries, test the queries by opening the folder testSQL_app; make sure the database **Carnivores.mdb** is present; and run the **CarnivoresSQL.exe** file as you did in the previous examples.

DISTINCT

There are many situations where you may want to only select the unique (or distinct) values in a field, that is, no duplicates in the same column (field). Placing the DISTINCT keyword directly after the SELECT keyword informs the DBMS that you want to select all distinct values from a specific column. The syntax for DISTINCT is shown below:

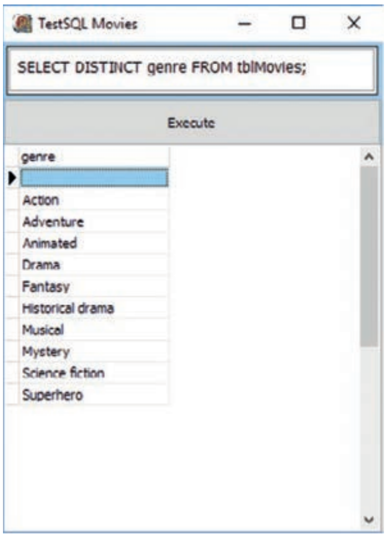
DISTINCT syntax

```
SELECT DISTINCT field_name FROM table_name;
```

Example 4.3

If you would like to know what the possible genres of movies in the database is, you only want to see the distinct genres.

```
SELECT DISTINCT genre FROM tblMovies;
```



SELECT DISTINCT genre FROM tblMovies;

genre
Action
Adventure
Animated
Drama
Fantasy
Historical drama
Musical
Mystery
Science fiction
Superhero

Figure 4.9: *A list of distinct genres from the database*

In this image, the first row is empty. This is because there are empty (or null) values in the genre column. Note no duplicate values appear in the column.

Example 4.4 More examples to try

SHOW:	SQL
1. The unique cities from the *tblStudios* table.	`SELECT DISTINCT city FROM tblStudios;`
2. The unique provinces from the *tblStudios* table.	`SELECT DISTINCT province FROM tblStudios;`
3. The unique release_date of movies from the *tblMovies* table.	`SELECT * FROM tblMovies`

Activity 4.2

4.2.1 Study table *tblCarnivores* and table *tblVetVisits* in the *Carnivores* database; correct the mistakes in the following SQL statements.

a. SELECT DISTINCT ReasonForVisit FROM Carnivores;

b. SELECT FROM tblVetVisits DISTINCT EnclosureNo;

Using a pen and paper, create queries to select the following data.

4.2.2 The unique *FamilyNames* for the animals in the *tblCarnivores*.

4.2.3 A list of the different reasons for the vet's visits.

4.2.4 A list of distinct enclosures.

Run **CarnivoresSQL.exe** to test your queries.

4.2.5 Based on the query results, how many unique animal family names are there?

Did you know

This is very useful if you need the values to fill a combo box with distinct values for user selection.

Queries like these are often used to create a list of unique items that meet some criteria. For example, you might create a query to identify movie studios that makes low-quality movies or to identify cities closely linked to the film industry. In Unit 4.2, you will learn how to add criteria to your selection.

ORDER BY

In the previous sections, you used the SELECT statement to select data. Now, you will learn how to sort the data that has been selected using the ORDER BY clause. To do this, you use the following syntax:

ORDER BY syntax

```
SELECT field_name1, field_name2,... FROM table_name
ORDER BY field_name1 order_type;
```

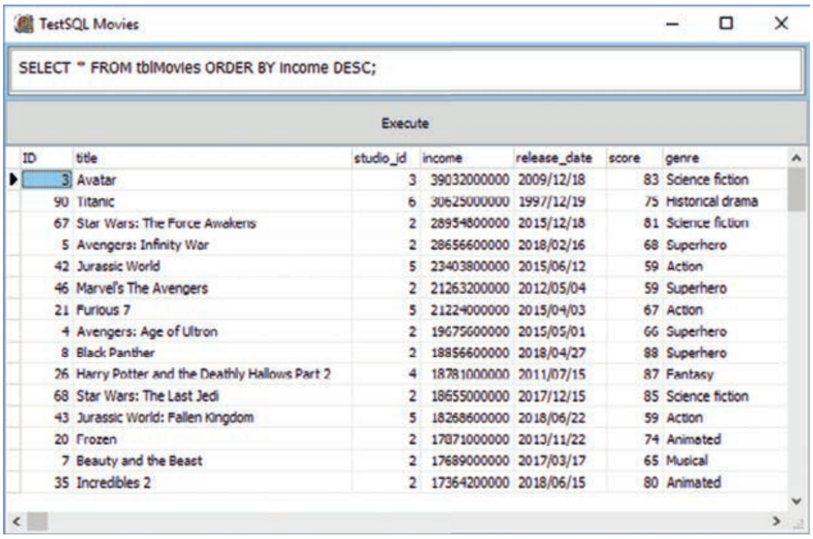
As you can see from the syntax, the ORDER BY clause introduces a new instruction to your SQL query. The instruction tells the DBMS to organise the results of the query based on the values of a specific field. The specific field must be one or more of the fields in the select-clause. If more, then the fields must be separated by commas. Each field must have its own order type indicated e.g. ORDER BY field1 ASC, field2 DESC;

Example 4.5

If you want to organise your list of movies from the highest income to the lowest income, you could use the following query.

Order from the Highest to the lowest income

```
SELECT * FROM tblMovies
ORDER BY income DESC;
```



```
SELECT * FROM tblMovies ORDER BY income DESC;
```

ID	title	studio_id	income	release_date	score	genre
3	Avatar	3	39032000000	2009/12/18	83	Science fiction
90	Titanic	6	30625000000	1997/12/19	75	Historical drama
67	Star Wars: The Force Awakens	2	28954800000	2015/12/18	81	Science fiction
5	Avengers: Infinity War	2	28656600000	2018/02/16	68	Superhero
42	Jurassic World	5	23403800000	2015/06/12	59	Action
46	Marvel's The Avengers	2	21263200000	2012/05/04	59	Superhero
21	Furious 7	5	21224000000	2015/04/03	67	Action
4	Avengers: Age of Ultron	2	19675600000	2015/05/01	66	Superhero
8	Black Panther	2	18856600000	2018/04/27	88	Superhero
26	Harry Potter and the Deathly Hallows Part 2	4	18781000000	2011/07/15	87	Fantasy
68	Star Wars: The Last Jedi	2	18655000000	2017/12/15	85	Science fiction
43	Jurassic World: Fallen Kingdom	5	18268600000	2018/06/22	59	Action
20	Frozen	2	17871000000	2013/11/22	74	Animated
7	Beauty and the Beast	2	17689000000	2017/03/17	65	Musical
35	Incredibles 2	2	17364200000	2018/06/15	80	Animated

Figure 4.10: *Movies from highest to lowest income*

As the image shows, at the end of 2018, Avatar was the highest grossing film, earning more than R39 billion.

Did you notice the word “DESC” at the end of the query? This keyword tells your database to show the results in *descending* order (from largest to smallest).

There are two order types you can use:

- **ASC**: Short for *ascending*, this organises your data from the smallest value to the largest value. ASC can be omitted and the data will automatically be order in *ascending* order.
- **DESC**: Short for *descending*, this organises your data from the largest value to the smallest value.

Example 4.6 More examples to try

SHOW:	SQL
1. All movies organised from the lowest income to the highest income.	`SELECT * FROM tblMovies ORDER BY income ASC;`
2. The scores of all the *tblMovies* ordered from the lowest to the highest score.	`SELECT scores FROM tblMovies ORDER BY scores;`
3. All movies organised from the newest movie to the oldest movie.	`SELECT * FROM tblMovies ORDER BY release_date DESC;`

Activity 4.3 Queries using ORDER BY

Using the database **Carnivores.mdb** create and test the following ordered queries using the *testSQL_app* for Carnivores:

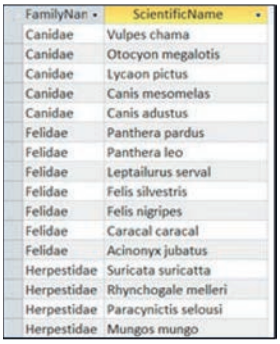
4.3.1 All of the general names of carnivores arranged alphabetically.

4.3.2 All of the vet's visits listed by date starting with the last visit.

4.3.3 All of the *scientificNames* of the Carnivores listed alphabetically.

4.3.4 The screenshot shows the fields *FamilyNames* and *ScientificName* selected from table *tblCarnivores* and arranged in a certain order.

Provide the SQL statement that could have produced this output.



FamilyNan	ScientificName
Canidae	Vulpes chama
Canidae	Otocyon megalotis
Canidae	Lycaon pictus
Canidae	Canis mesomelas
Canidae	Canis adustus
Felidae	Panthera pardus
Felidae	Panthera leo
Felidae	Leptailurus serval
Felidae	Felis silvestris
Felidae	Felis nigripes
Felidae	Caracal caracal
Felidae	Acinonyx jubatus
Herpestidae	Suricata suricatta
Herpestidae	Rhynchogale melleri
Herpestidae	Paracynictis selousi
Herpestidae	Mungos mungo

Activity 4.4 Select and sort columns

Create the following ordered queries and test in **MoviesSQL.exe**:

4.4.1 Select the income, score and title from the *tblMovies* table.

4.4.2 Select the *name*, *city* and *provinc*e fields from the *tblStudios* table.

4.4.3 Select the *title* and *score* fields and order the table based on the score (from highest to lowest) from the *tblMovies* table.

4.4.4 Select the *title*, *income* and *release_date* fields and order the table based on the release date (from oldest to newest) from the *tblMovies* table.

4.4.5 Select all cities from the *tblStudios* table. Every city must appear only once.

4.4.6 Select all movies organised by *genre* (ascending) and *title* (ascending).

4.4.7 Select all movies organised according to *score* (highest to lowest) and release *date* (oldest to the newest).

4.4.8 Select all movies organised by *genre* (ascending) and *income* (highest to lowest).

4.4.9 Select all movies organised by *studio_id* (from lowest to highest) and *score* (from highest to the lowest).

UNIT

4.2 Select columns and rows

In Unit 4.1 we selected columns and displayed all the data available in the selected columns. Now imagine your bank details occupy one row in a database table and your consultant only needs your record. Why fetch the whole table? There must be a way to select just the row that the consultant requires. This is exactly what the function of the WHERE clause is: to select the rows according to some specific conditions. In the case of your bank account, the consultant can select just the row with your details.

The WHERE syntax is given below:

WHERE syntax

```
SELECT field_names
FROM table_name
WHERE condition1 <Logic Operator > condition 2;
```

Take note

Every record that meets the conditions in the WHERE clause will be selected.

You use the same syntax as the normal SELECT syntax, but simply add the WHERE clause at the end of it (before the semicolon).

SQL does not register the line breaks in an SQL statement. As such, SELECT, FROM and WHERE can be placed on separate lines to make the statement easier to read. This is especially important with complex queries.

RELATIONAL OPERATORS

The Boolean expressions used with the WHERE command, are similar to those used in Delphi. The following table lists the relational operators that can be used in SQL and shows how it is applied in the WHERE clause.

Table 4.1: *Relational operators that can be used in SQL*

	MEANING	EXAMPLE
=	Equal	`WHERE genre = 'action';`
<>	Not equal to	`WHERE studio_id <> 4;`
>	Larger than	`WHERE income > 10000000000;`
>=	Larger than or equal to	`WHERE income >= 21224000000;`
<	Smaller than	`WHERE id < 10;`
<=	Smaller than or equal to	`WHERE studio_id <= 3;`

Note, in SQL, strings must be surrounded by single quotation marks normally used in Delphi.

To see how the operators are used, work through the following examples.

Example 4.7 Creating a query with WHERE and one CONDITION

To create a query with WHERE:

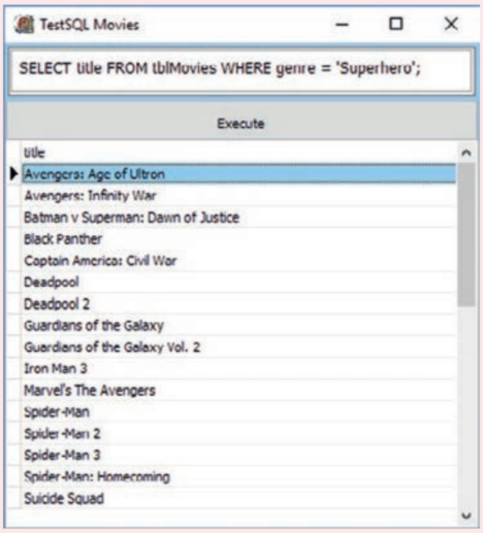
1. Using the database **Movies.mdb** create and test the following queries with the testSQL_app MoviesSQL.exe:

   WHERE query

   ```
   SELECT title FROM tblMovies WHERE genre = 'Superhero';
   ```

   This code will show all the movies in the database where the *genre* is 'Superhero'. Remember, in SQL, strings must be surrounded by quotation marks.

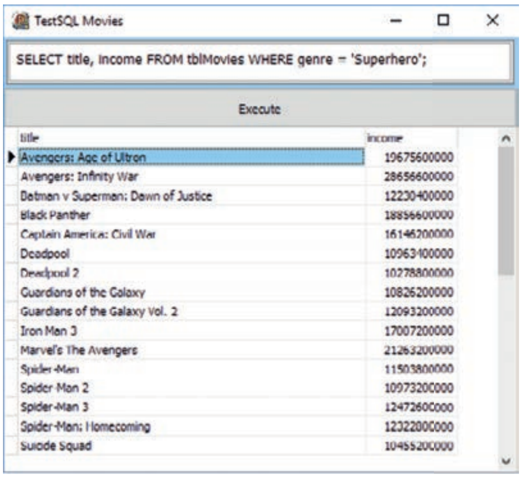
   By clicking on the [Execute] button, you should see the following result:



```
SELECT title FROM tblMovies WHERE genre = 'Superhero';
```

```
title
Avengers: Age of Ultron
Avengers: Infinity War
Batman v Superman: Dawn of Justice
Black Panther
Captain America: Civil War
Deadpool
Deadpool 2
Guardians of the Galaxy
Guardians of the Galaxy Vol. 2
Iron Man 3
Marvel's The Avengers
Spider-Man
Spider-Man 2
Spider-Man 3
Spider-Man: Homecoming
Suicide Squad
```

Example 4.8 More examples to try

SHOW:	SQL
As the results show, a lot of the most successful movies are superhero movies! But how much money are the studios earning on these movies? To answer this question, we need to look at the income of each movie in our list:	
1. WHERE query with income field added 	`SELECT title, income FROM tblMovies WHERE genre = 'Superhero';`

```
SELECT title, income FROM tblMovies WHERE genre = 'Superhero';
```

```
title                                   income
Avengers: Age of Ultron            19675600000
Avengers: Infinity War             28656600000
Batman v Superman: Dawn of Justice 12230400000
Black Panther                      18856600000
Captain America: Civil War         16146200000
Deadpool                           10963400000
Deadpool 2                         10278800000
Guardians of the Galaxy            10826200000
Guardians of the Galaxy Vol. 2     12093200000
Iron Man 3                         17007200000
Marvel's The Avengers              21263200000
Spider-Man                         11503800000
Spider Man 2                       10973200000
Spider-Man 3                       12472600000
Spider-Man: Homecoming             12322000000
Suicide Squad                      10455200000
```

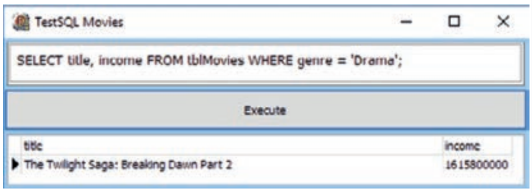
Example 4.8 More examples to try *continued*

SHOW:	SQL

Looking at the results, you will notice that the movie with the lowest income on the list earned more than R10 billion, while the movie with the highest income (Avengers: Infinity War) earned almost R29 billion.

Perhaps it is normal for successful movies to earn this much money. To see if this is true, you can compare superhero movies with drama movies:

2. Change the query to select movies where the genre is “drama”.



```
TestSQL Movies
SELECT title, income FROM tblMovies WHERE genre = 'Drama';
```

```
  title                                      income
▶ The Twilight Saga: Breaking Dawn Part 2    1615800000
```

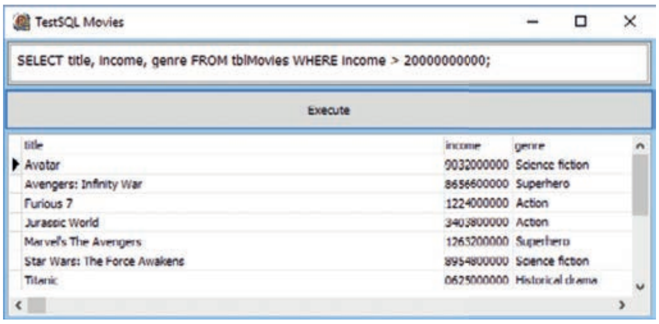
```
SELECT title, income FROM tblMovies
WHERE genre = 'Drama';
```

There appears to be only a single drama movie on the entire list: Twilight Breaking Dawn Part 2. So, even though drama movies are very popular, they do not attract as large an audience as superhero movies. This may explain why the studios are releasing so many superhero movies every year.

This begs the question: What other movie genres attract large audiences and have an income more than say R20 000 000 000, that is, twenty billion Rand?

3. To answer this we select movies of all genres where the income is greater than R20 000 000 000. Show the movies’ title, income and genre.

```
SELECT title, income, genre
FROM tblMovies WHERE income >
20000000000;
```



```
TestSQL Movies
SELECT title, income, genre FROM tblMovies WHERE income > 2000000000;
```

```
  title                            income      genre
▶ Avotar                           9032000000  Science fiction
  Avengers: Infinity War           8656600000  Superhero
  Furious 7                        1224000000  Action
  Jurassic World                   3403800000  Action
  Marvel's The Avengers            1263200000  Superhero
  Star Wars: The Force Awakens     8954800000  Science fiction
  Titanic                          0625000000  Historical drama
```

The list now only shows movies with an income greater than R20 billion. There are seven movies in total, with two science fiction movies, two action movies and two superhero movies and one historical drama.

Activity 4.5 Select and sort columns

Using the App. in the folder 04 – Movies with the database **Movies.mdb**; create and test queries that will select the following data:

4.5.1 All fields of fantasy movies.

4.5.2 The *title* and *date* of all movies with an income lower than R12 billion.

4.5.3 The *genre* of all movies with a score higher than or equal to 80.

4.5.4 The *name*, the *city* and *province* of all studios in the United States.

4.5.5 All movies where the *studio_id* is 2 organised by genre (ascending) and income (descending).

Using the App in the folder 04 – Carnivores with the database **Carnivores.mdb**; create and test queries that will select the following data:

4.5.6 All species of Carnivores that have an *endangered* rating of “VU”.

4.5.7 All animals kept in enclosure ZC2.

4.5.8 All animals where the number of adults is greater than 5.

WILDCARDS AND THE LIKE OPERATOR

Wildcards are special characters that can be used in SQL to represent any character or number of characters (letters or numbers) in a string. These wildcards are used in a WHERE clause in combination with the LIKE operator to find values that match a specific pattern in a field.

LIKE syntax

```
SELECT field_names FROM table_name
WHERE field_name LIKE 'wildcard_pattern';
```

Did you know

Visit https://www.w3schools.com/sql/sql_wildcards.asp for a list of wildcards.

There are two wildcard characters that can be used in wildcard patterns:

NAME	DESCRIPTION	DELPHI SYMBOL	ACCESS SYMBOL
Only one	This wildcard character is used in place of a single character. For example, the wildcard pattern 'c_p' will match the words 'cap', 'cup' and 'cop'.	_	?
Zero to many	This wildcard character is used in place of any number of characters, from zero to thousands. For example, the wildcard pattern 'c%p' will match words like 'cap' and 'cup', but also 'crop', 'clasp' and 'championship'.	%	*

By using these characters to query your database, you can search for movies starting with the word 'The' by placing a percentage symbol (in Delphi) or an asterisk symbol (in Access) after it.

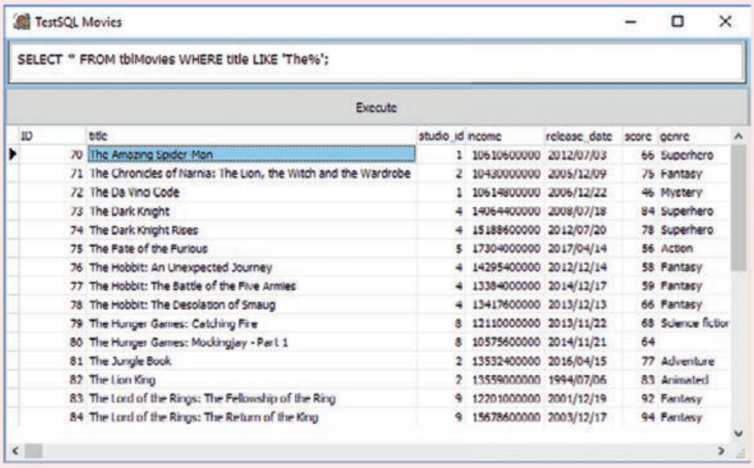
Example 4.9 Creating a query with WHERE and a CONDITION using LIKE with wildcards

Run the MoviesSQL app, enter and test the following queries:

Movies starting with "The"

```
SELECT * FROM tblMovies
WHERE title LIKE 'The%';
```

The results can be seen in the figure below.



```
SELECT * FROM tblMovies WHERE title LIKE 'The%';
```

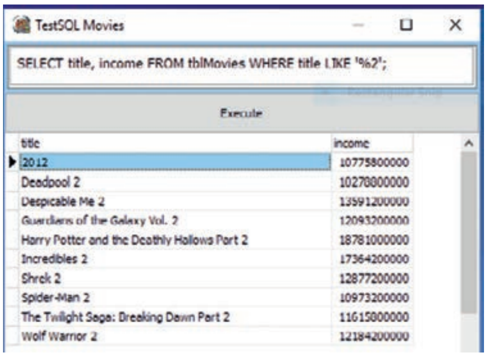
ID	title	studio_id	income	release_date	score	genre
70	The Amazing Spider-Man	1	10610600000	2012/07/03	66	Superhero
71	The Chronicles of Narnia: The Lion, the Witch and the Wardrobe	2	10430000000	2005/12/09	75	Fantasy
72	The Da Vinci Code	1	10614800000	2006/12/22	46	Mystery
73	The Dark Knight	4	14064400000	2008/07/18	84	Superhero
74	The Dark Knight Rises	4	15188600000	2012/07/20	78	Superhero
75	The Fate of the Furious	5	17304000000	2017/04/14	56	Action
76	The Hobbit: An Unexpected Journey	4	14295400000	2012/12/14	58	Fantasy
77	The Hobbit: The Battle of the Five Armies	4	13384000000	2014/12/17	59	Fantasy
78	The Hobbit: The Desolation of Smaug	4	13417600000	2013/12/13	66	Fantasy
79	The Hunger Games: Catching Fire	8	12110000000	2013/11/22	68	Science fictior
80	The Hunger Games: Mockingjay - Part 1	8	10575600000	2014/11/21	64	
81	The Jungle Book	2	13532400000	2016/04/15	77	Adventure
82	The Lion King	2	13559000000	1994/07/06	83	Animated
83	The Lord of the Rings: The Fellowship of the Ring	9	12201000000	2001/12/19	92	Fantasy
84	The Lord of the Rings: The Return of the King	9	15678600000	2003/12/17	94	Fantasy

Figure 4.11: *All movies whose names start with "The"*

Example 4.10 More examples to try

<table>
<tr><th>SHOW:</th><th>SQL</th></tr>
<tr><td>

1. The *title* and *income* of all movie titles ending with 2.



SELECT title, income FROM tblMovies WHERE title LIKE '%2';

<table>
<tr><th>title</th><th>income</th></tr>
<tr><td>2012</td><td>10775800000</td></tr>
<tr><td>Deadpool 2</td><td>10270800000</td></tr>
<tr><td>Despicable Me 2</td><td>13591200000</td></tr>
<tr><td>Guardians of the Galaxy Vol. 2</td><td>12093200000</td></tr>
<tr><td>Harry Potter and the Deathly Hallows Part 2</td><td>18781000000</td></tr>
<tr><td>Incredibles 2</td><td>17364200000</td></tr>
<tr><td>Shrek 2</td><td>12877200000</td></tr>
<tr><td>Spider-Man 2</td><td>10973200000</td></tr>
<tr><td>The Twilight Saga: Breaking Dawn Part 2</td><td>11615800000</td></tr>
<tr><td>Wolf Warrior 2</td><td>12184200000</td></tr>
</table>

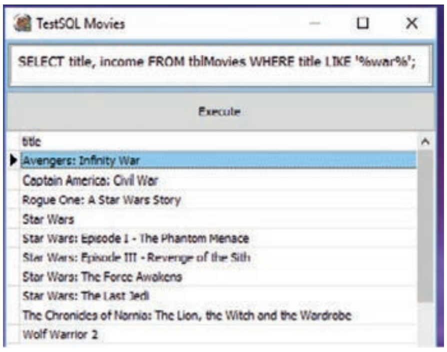
</td><td>

Movies ending with 2

```
SELECT title, income FROM tblMovies
WHERE title LIKE '%2';
```

</td></tr>
<tr><td>

2. The *title* and *income* of all movie titles containing the word ‘war’.



SELECT title, income FROM tblMovies WHERE title LIKE '%war%';

<table>
<tr><th>title</th></tr>
<tr><td>Avengers: Infinity War</td></tr>
<tr><td>Captain America: Civil War</td></tr>
<tr><td>Rogue One: A Star Wars Story</td></tr>
<tr><td>Star Wars</td></tr>
<tr><td>Star Wars: Episode I - The Phantom Menace</td></tr>
<tr><td>Star Wars: Episode III - Revenge of the Sith</td></tr>
<tr><td>Star Wars: The Force Awakens</td></tr>
<tr><td>Star Wars: The Last Jedi</td></tr>
<tr><td>The Chronicles of Narnia: The Lion, the Witch and the Wardrobe</td></tr>
<tr><td>Wolf Warrior 2</td></tr>
</table>

Movies containing the word war

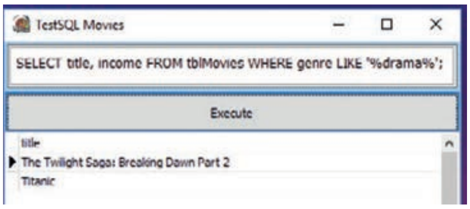
</td><td>

Movies containing the word war

```
SELECT title, income FROM tblMovies
WHERE title LIKE '%war%';
```

</td></tr>
<tr><td>

3. The *title* and *income* of all movie genres containing the word ‘drama’



SELECT title, income FROM tblMovies WHERE genre LIKE '%drama%';

<table>
<tr><th>title</th></tr>
<tr><td>The Twilight Saga: Breaking Dawn Part 2</td></tr>
<tr><td>Titanic</td></tr>
</table>

Note this time we found 2 Movies instead of the one found previously, where the condition was genre = ’drama’.

</td><td>

Movies having ‘drama’ in the genre

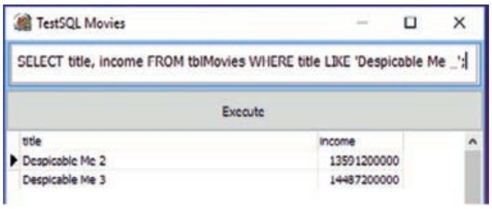
```
SELECT title, income FROM tblMovies
WHERE genre LIKE '%drama%';
```

</td></tr>
</table>

Example 4.10 More examples to try *continued*

<table>
<tr><th>SHOW:</th><th>SQL</th></tr>
<tr><td>

4. Say you don’t remember all of the sequel numbers of the ‘Despicable me’ movies, but you remember that they all have a space and a single digit at the end of the name. In this case you can make use of ‘ _ ’



SELECT title, income FROM tblMovies WHERE title LIKE 'Despicable Me _';

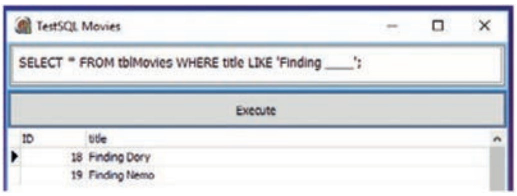
title	income
Despicable Me 2	13591200000
Despicable Me 3	14487200000

</td><td>

```
SELECT title, income FROM tblMovies
WHERE title LIKE 'Despicable Me _';
```

</td></tr>
<tr><td>

5. Display records where the *title* of the movies start with the word Finding and followed by a 4-letter word



SELECT * FROM tblMovies WHERE title LIKE 'Finding ____';

ID	title
18	Finding Dory
19	Finding Nemo

</td><td>

```
SELECT * FROM tblMovies
WHERE title LIKE 'Finding ____';
```

</td></tr>
<tr><td colspan="2">

A database used in business often has millions of records and the exact detail of what needs to be searched for is not always known. Using LIKE and wildcards we can reduce the dataset to a manageable size. For example, one only needs to know a part of a movie genre, or an address, to find what you are looking for.

</td></tr>
</table>

Activity 4.6

Using the App. in the folder 04 – Movies with the database **Movies.mdb**; create and test queries that will select the following data:

4.6.1 All movies ending with the word ‘out’.

4.6.2 All movies where the score starts with an 8 (using an underscore).

4.6.3 Spider-Man 2 and Spider-Man 3 (but no other Spider-Man movies).

4.6.4 All movies where the name starts with ‘Harry’ and ends with an ‘e’.

4.6.5 All movies ending with the word ‘man’.

Using the App. in the folder 04 – Carnivores with the database **Carnivores.mdb**; create and test queries that will select the following data:

4.6.6 All Carnivores where the name ends in ‘mongoose’.

4.6.7 All vet visits where the animal had an injury or were injured.

4.6.8 All Vet visits where the vet treated ears.

In Delphi **the ‘_’character** is useful if you want to select data with a **specific length**, but not specific characters. For example, if you wanted to select mobile phone numbers starting with 072, you could use the wildcard string ‘072_______’ (i.e. 7 underscore characters). This would find all 10-digit telephone numbers starting with 072.

BOOLEAN OPERATORS

As with the Delphi conditional statements, you can use Boolean operators to combine conditions in the WHERE clause. The table below shows three of the most common Boolean operators that can be used in SQL.

Table 4.2: *Most common Boolean operators that can be used in SQL*

OPERATOR	DESCRIPTION	EXAMPLE
AND	(condition1) AND (condition2)	`WHERE (genre = 'Superhero') AND (score> 80);`
OR	(condition1) OR (condition2)	`WHERE (genre = 'Drama') OR (genre = 'Historical drama');`
NOT	NOT (condition)	`WHERE NOT (country = 'United States');`

OPERATOR PRECEDENCE

The Boolean operators are executed in the following order:

Parenthesis
Multiplication, division
Subtraction, addition
NOT
AND
OR

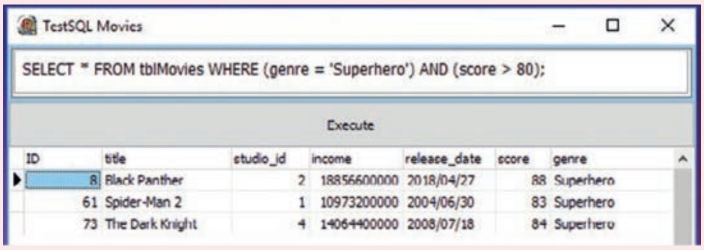
To change the order of operations, you will need to use of brackets – much the same way as you do in Mathematics.

Example 4.11 Creating a query with WHERE and compound CONDITIONS

Using the apps in the folder with the databases 04 – Movies **Movies.mdb** and 04 – Carnivores folder with the database **Carnivores.mdb**; create and test queries that will select the following data.

```
SELECT * FROM tblMovies
WHERE (genre = 'Superhero') AND (score > 80);
```

The query selects all fields of superhero movies where the score is larger than or equal to 80. This will show the following results.



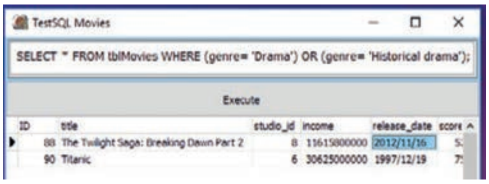
```
SELECT * FROM tblMovies WHERE (genre = 'Superhero') AND (score > 80);
```

ID	title	studio_id	income	release_date	score	genre
8	Black Panther	2	18856600000	2018/04/27	88	Superhero
61	Spider-Man 2	1	10973200000	2004/06/30	83	Superhero
73	The Dark Knight	4	14064400000	2008/07/18	84	Superhero

Figure 4.12: *All superhero movies with a score above 80*

Example 4.12 More examples to try

SHOW:	SQL

1. Select all the drama and historical drama movies

 	
SELECT * FROM tblMovies WHERE (genre= 'Drama') OR (genre= 'Historical drama');
ID | title | studio_id | income | release_date | score
88 | The Twilight Saga: Breaking Dawn Part 2 | 8 | 11615800000 | 2012/11/16 | 5
90 | Titanic | 6 | 30625000000 | 1997/12/19 | 7
	`SELECT * FROM tblMovies`
	`WHERE (genre = 'Drama') OR (genre =`
	`'Historical drama');`

2. Select all the fields of all movies with a score between 50 and 90. 50 and 90 included.	`SELECT * FROM tblMovies`
	`WHERE (score >= 50) AND (score <=`
	`90);`

3. All movies that are not superhero and not fantasy or animation movies.	`SELECT * FROM tblMovies`
	`WHERE NOT genre = 'superhero' AND`
	`NOT (genre = 'fantasy' or genre =`
	`'animated');`

Activity 4.7

4.7.1 The clause WHERE genre LIKE '%drama%' and the clause WHERE (genre = 'Drama') OR (genre = 'Historical drama') selected the same records.
Will this always be the case? Motivate your answer.

4.7.2 Which one of the following WHERE clauses will select action and superhero movies with and income less than R20 000 000 000:

- **a.** WHERE genre = 'action' OR genre = 'superhero' AND income < 20000000000;
- **b.** WHERE genre = 'action' AND genre = 'superhero' AND income <= 20000000000;
- **c.** WHERE genre = 'action' OR genre = 'superhero' AND income < 20000000000;
- **d.** WHERE (genre = 'action' OR genre = 'superhero') AND income < 20000000000;

4.7.3 Complete the following SQL statement:

SELECT * ______ tblMovies _______ genre = 'Animated' _______ score > 85;

Using the apps in the folder with the databases 04 – Movies **Movies.mdb** and 04 – Carnivores folder with the database **Carnivores.mdb**; create and test queries that will select the following data.

4.7.4 All movies with a *genre* of fantasy or a *studio_id* equal to 4.

4.7.5 All movies that are not superhero or action movies.

4.7.6 All action movies with a *score* less than 50 and those greater than 60.

4.7.7 All Carnivores where the adults are over 5 years old and they do not belong to the Viverridae family.

SPECIAL OPERATORS

SQL also includes a few special operators that help you to simplify complex conditions. These operators, their functions and their syntax are shown in the table below.

Table 4.3: *Special SQL operators*

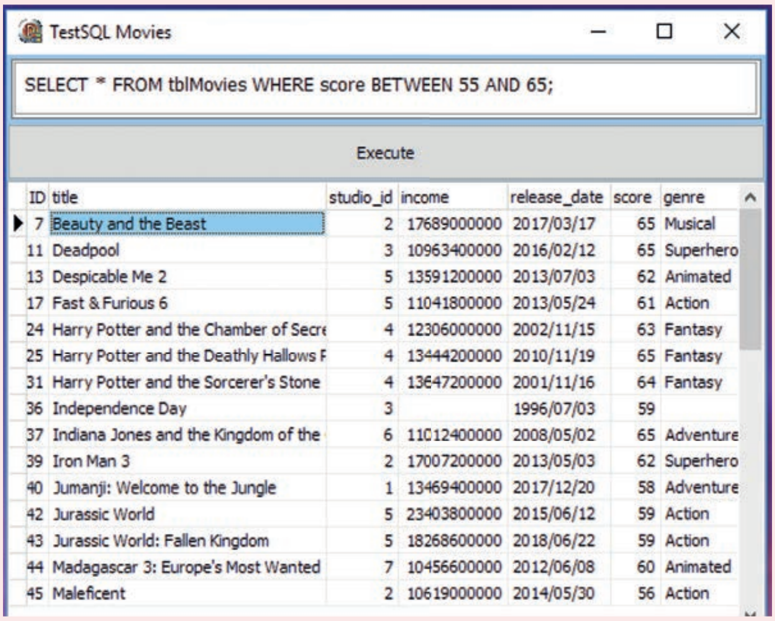
OPERATOR	FUNCTION	SYNTAX
BETWEEN	Selects all values found between the upper and lower values (including the upper and lower values)	WHERE field_name BETWEEN min_value AND max_value;
IN	Selects all records that have one of the values in the provided list of values.	WHERE field_name IN (value1, … , value50);
IS NULL	Selects all records that have no value for a specific field.	WHERE field_name IS NULL (NULL = field empty)
IS NOT NULL	Selects all records that have value for a specific field.	WHERE field_name IS NOT NULL;

Example 4.13

BETWEEN example

```
SELECT * FROM tblMovies
WHERE score BETWEEN 55 AND 65;
```

This query will result in the following list of movies.



```
SELECT * FROM tblMovies WHERE score BETWEEN 55 AND 65;
```

ID	title	studio_id	income	release_date	score	genre
7	Beauty and the Beast	2	17689000000	2017/03/17	65	Musical
11	Deadpool	3	10963400000	2016/02/12	65	Superhero
13	Despicable Me 2	5	13591200000	2013/07/03	62	Animated
17	Fast & Furious 6	5	11041800000	2013/05/24	61	Action
24	Harry Potter and the Chamber of Secre	4	12306000000	2002/11/15	63	Fantasy
25	Harry Potter and the Deathly Hallows F	4	13444200000	2010/11/19	65	Fantasy
31	Harry Potter and the Sorcerer's Stone	4	13647200000	2001/11/16	64	Fantasy
36	Independence Day	3		1996/07/03	59	
37	Indiana Jones and the Kingdom of the	6	11012400000	2008/05/02	65	Adventure
39	Iron Man 3	2	17007200000	2013/05/03	62	Superhero
40	Jumanji: Welcome to the Jungle	1	13469400000	2017/12/20	58	Adventure
42	Jurassic World	5	23403800000	2015/06/12	59	Action
43	Jurassic World: Fallen Kingdom	5	18268600000	2018/06/22	59	Action
44	Madagascar 3: Europe's Most Wanted	7	10456600000	2012/06/08	60	Animated
45	Maleficent	2	10619000000	2014/05/30	56	Action

Figure 4.13: *Movies with a score between 55 and 65*

Take note that each of these movies have a score between 55 and 65, and a few movies (like one of the Harry Potter movies) has a score of exactly 65.

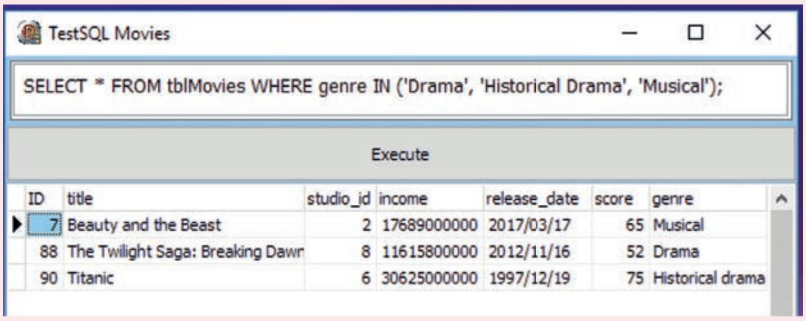
The next example shows how the IN command works.

Example 4.14

IN example

```
SELECT * FROM tblMovies
WHERE genre IN ('Drama', 'Historical Drama', 'Musical');
```

This query will create a list of movies where the genre is either “Drama”, “Historical Drama” or “Musical”.



SELECT * FROM tblMovies WHERE genre IN ('Drama', 'Historical Drama', 'Musical');

ID	title	studio_id	income	release_date	score	genre
7	Beauty and the Beast	2	17689000000	2017/03/17	65	Musical
88	The Twilight Saga: Breaking Dawn	8	11615800000	2012/11/16	52	Drama
90	Titanic	6	30625000000	1997/12/19	75	Historical drama

Figure 4.14: *All drama, historical drama and musicals*

While the same results could be achieved by writing a very long SQL query containing OR operators, the IN operator is a lot easier to write and read.

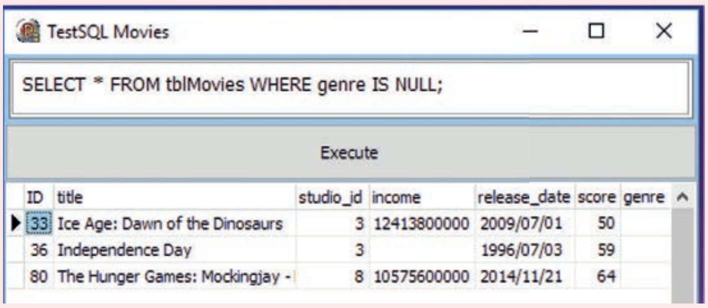
The next example shows how the IS NULL command works.

Example 4.15

IS NULL operator

```
SELECT * FROM tblMovies
WHERE genre IS NULL;
```

This final example query will select all records that do not have a value for the genre field. This can be especially useful if there are important differences between data with values and without values or when you are trying to find and fix any gaps in your data.



SELECT * FROM tblMovies WHERE genre IS NULL;

ID	title	studio_id	income	release_date	score	genre
33	Ice Age: Dawn of the Dinosaurs	3	12413800000	2009/07/01	50	
36	Independence Day	3		1996/07/03	59	
80	The Hunger Games: Mockingjay -	8	10575600000	2014/11/21	64	

Figure 4.15: *Movies without a genre*

Activity 4.8

Using the apps in the folder with the databases 04 – Movies **Movies.mdb** and 04 – Carnivores folder with the database **Carnivores.mdb**; create and test queries that will select the following data.

Use special operators in each.

4.8.1 *Title* and *studio_id* of all movies with a *studio_id* of 1, 4, 5 or 9.

4.8.2 *Title* and *genre* of all movies with an *income* between R12 billion and R18 billion.

4.8.3 *Title*, *date* and *score* of all movies without a release date.

4.8.4 *Title* and *genre* of all movies that are not action, adventure, fantasy or superhero.

4.8.5 *Title* of all superhero movies with a *score* between 60 and 80.

4.8.6 The *generalName* and *EnclosureSize* of all enclosures greater than 30 m^2 and less than 40 m^2

Up to now, dates from the SQL database have only been selected, without being manipulated or used in any conditions. This is because, dates, like strings, have their own rules and functions which need to be used.

DATES IN CONDITIONS

To use a date in a condition, you need to surround the date with the hash (#) symbol. The date does not have to follow a specific format since SQL does a good job of interpreting different dates. This means that, if you want to enter the date 5 January 2018, you could use a number of different formats, including:

- #5 January 2018#
- #5 Jan 18#
- #2018/01/05#

The one date format you should not use is the South African standard format of day/month/year (that is, #05/01/2018#) since SQL may interpret this as month/day/year, giving you incorrect results. Instead, it is safer to use the international date format which goes from the largest unit of time to the smallest (that is, year/month/day).

DATE FUNCTIONS

The following four date functions can also be used when working with dates in your database.

Table 4.4: *Date functions*

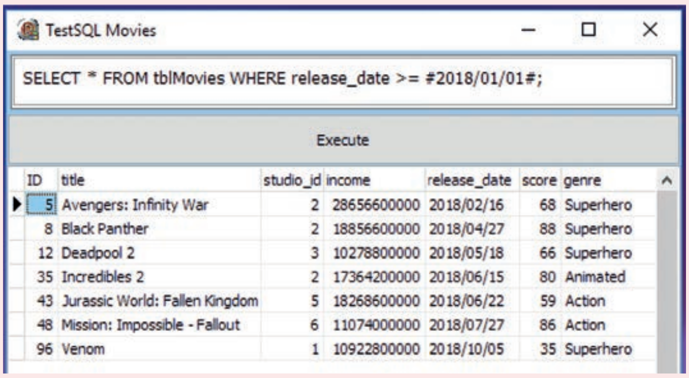
FUNCTION	DESCRIPTION	SYNTAX
YEAR	Returns an integer containing the year of the selected date.	YEAR(date)
MONTH	Returns an integer (between 1 and 12) containing the month of the selected date.	MONTH(date)
DAY	Returns an integer (between 1 and 31) containing the day of the selected date.	DAY(date)
DATE	Returns today's date as a date variable.	DATE( )

Example 4.16

1. Which movies from your database were released on or after 1 January 2018?

Date condition

```
SELECT * FROM tblMovies
WHERE release_date >= #2018/01/01#;
```



```
SELECT * FROM tblMovies WHERE release_date >= #2018/01/01#;
```

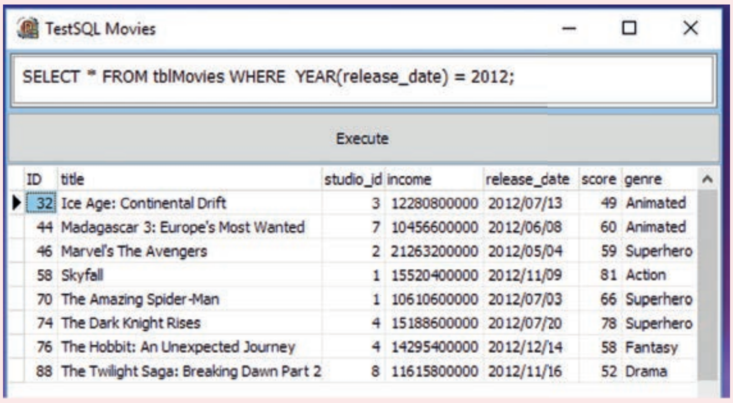
ID	title	studio_id	income	release_date	score	genre
5	Avengers: Infinity War	2	28656600000	2018/02/16	68	Superhero
8	Black Panther	2	18856600000	2018/04/27	88	Superhero
12	Deadpool 2	3	10278800000	2018/05/18	66	Superhero
35	Incredibles 2	2	17364200000	2018/06/15	80	Animated
43	Jurassic World: Fallen Kingdom	5	18268600000	2018/06/22	59	Action
48	Mission: Impossible - Fallout	6	11074000000	2018/07/27	86	Action
96	Venom	1	10922800000	2018/10/05	35	Superhero

Figure 4.16: *Movies released since the start of 2018*

2. Which movies from the database were released in 2012?

condition using a Date function

```
SELECT * FROM tblMovies WHERE YEAR(release_date) = 2012;
```



```
SELECT * FROM tblMovies WHERE  YEAR(release_date) = 2012;
```

ID	title	studio_id	income	release_date	score	genre
32	Ice Age: Continental Drift	3	12280800000	2012/07/13	49	Animated
44	Madagascar 3: Europe's Most Wanted	7	10456600000	2012/06/08	60	Animated
46	Marvel's The Avengers	2	21263200000	2012/05/04	59	Superhero
58	Skyfall	1	15520400000	2012/11/09	81	Action
70	The Amazing Spider-Man	1	10610600000	2012/07/03	66	Superhero
74	The Dark Knight Rises	4	15188600000	2012/07/20	78	Superhero
76	The Hobbit: An Unexpected Journey	4	14295400000	2012/12/14	58	Fantasy
88	The Twilight Saga: Breaking Dawn Part 2	8	11615800000	2012/11/16	52	Drama

When using dates in conditional statements, all the Boolean operators, including AND, OR and BETWEEN, can be used.

Activity 4.9

Using the apps in the folder with the databases 04 – Movies **Movies.mdb**; create and test queries that will select the following data.

4.9.1 Show all of the movies released before the start 1994.

4.9.2 Show all of the movies released after 10 June 2016.

4.9.3 Show all of the movies released between the start of 2000 and the end of 2005.

4.9.4 Show all of the movies released on the first day of the month.

4.9.5 Show all of the movies released in the last three months of the year.

4.9.6 Show all of the movies released in 2009.

Activity 4.10 Select columns and rows

Write down the following queries using pen and paper. Make sure that all calculated fields have field names.

4.10.1 Select the *title* and *score* of all fantasy movies.

4.10.2 Select all movies made by the second studio.

4.10.3 Select all movies with an *income* smaller than R13 billion and a *score* below 50.

4.10.4 Select all movies with a *score* above 90 that are not animated.

4.10.5 Select all, superhero, fantasy and science fiction movies using the IN operator.

4.10.6 Select all movies that do not have a *release date*.

4.10.7 Select all movies that contain the word 'me'.

4.10.8 Select all movies released before 1999.

UNIT 4.3 Calculated columns

CALCULATED FIELDS

https://www.youtube.com/watch?v=MAPYm9I9WQc

When working with datasets, you may wish to perform some calculations on a field before returning the selected data. For example, your Movies database is showing the income of the movies in Rands, even though most of the income was earned in dollars. In situations outside of South Africa, you may wish to first convert the income to dollars before displaying the results. To do this, you need to make use of a **calculated field**.

A calculated field is displayed in the same was as any other field on your table. The only difference between a normal field and a calculated field is that, as the name suggests, the calculated field is calculated each time you run your query. This means that the data from the calculated field is never stored on the database itself, but instead is recalculated each time.

New words

calculated field – the field that is calculated each time you run your query

To create a calculated field, you enter the calculation into your SQL query as a selected field.

Calculated field syntax

```
SELECT field_names, calculation
FROM table_name;
```

The problem with these queries is that the calculated fields do not have a heading. This makes it difficult to interpret the results. To assign a name to the heading, you need to use the **AS** clause, as shown in the syntax below.

Calculated field syntax with AS

```
SELECT field_names, calculation AS calculated_field_name
FROM table_name;
```

Calculated field names cannot be used in the other clauses such as ORDER BY.

SQL calculations use the same mathematical operators (and order of operations – BODMAS) as calculations inside Delphi.

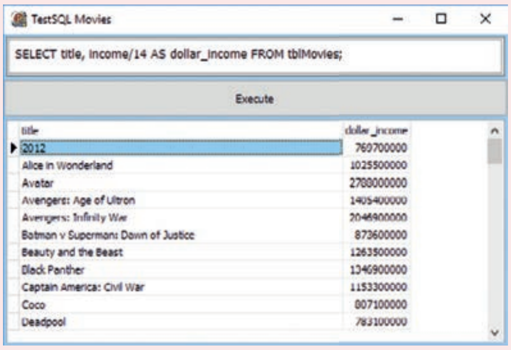
Example 4.17

In the query below, the income is divided by 14 to convert the Rands into Dollars. This is therefore a calculated field. Using the **AS** command, the calculated field is given the name “dollar_income”.

Calculated field example

```
SELECT title, income/14 AS dollar_income
FROM tblMovies;
```

Looking at the image alongside, you will see that there is a new calculated field called *dollar_income* which shows the original income of the movies divided by 14.



```
SELECT title, income/14 AS dollar_income FROM tblMovies;
```

title	dollar_income
2012	760700000
Alice in Wonderland	1025500000
Avatar	2780000000
Avengers: Age of Ultron	1405400000
Avengers: Infinity War	2046900000
Batman v Superman: Dawn of Justice	873600000
Beauty and the Beast	1263500000
Black Panther	1346900000
Captain America: Civil War	1153300000
Coco	807100000
Deadpool	783100000

Figure 4.17: *Selection with a calculated field*

Activity 4.11 Select columns and rows

Using the apps in the folder with the databases 04 – Movies **Movies.mdb** and 04 – Carnivores folder with the database **Carnivores.mdb**; create and test queries that will select the following data.

4.11.1 All movies showing the movie *title* and a calculated field called *lower_score* that subtracts 10 from the score.

4.11.2 All movies showing the movie *title* and a calculated field called *income_in_billions* that divides the income by 1 000 000 000.

4.11.3 All carnivores listing the general name and a calculated field *Total_Number_Animals* where the family name is 'Canidae'.

FUNCTIONS IN CALCULATIONS

Previously we used the DATE functions as part of the conditions to select records. Functions can also be used in calculated fields as part of the calculation or to format the result of a calculation.

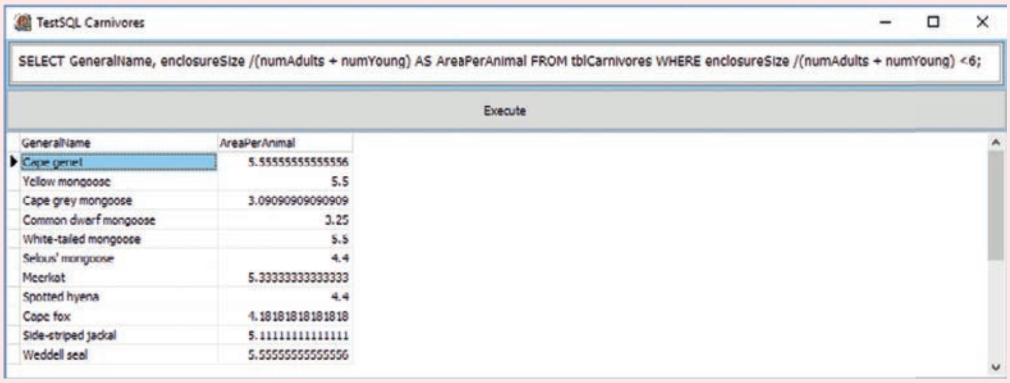
NUMBER FUNCTIONS

As you start using calculated fields more often, you will notice that these fields are often not formatted correctly.

Example 4.18

All carnivores listing the general name and a calculated field *AreaPerAnimal* where *area_per_animal* is less than 6 m^2 per animal to determine overcrowded enclosures. The calculated field name cannot be used in the WHERE clause, only the actual calculation.

```
SELECT GeneralName, enclosureSize /(numAdults + numYoung)AS AreaPerAnimal
FROM tblCarnivores
WHERE enclosureSize /(numAdults + numYoung) <6;
```



```
SELECT GeneralName, enclosureSize /(numAdults + numYoung) AS AreaPerAnimal FROM tblCarnivores WHERE enclosureSize /(numAdults + numYoung) <6;
```

GeneralName	AreaPerAnimal
Cape genet	5.55555555555556
Yellow mongoose	5.5
Cape grey mongoose	3.09090909090909
Common dwarf mongoose	3.25
White-tailed mongoose	5.5
Selous' mongoose	4.4
Meerkat	5.33333333333333
Spotted hyena	4.4
Cape fox	4.18181818181818
Side-striped jackal	5.11111111111111
Weddell seal	5.55555555555556

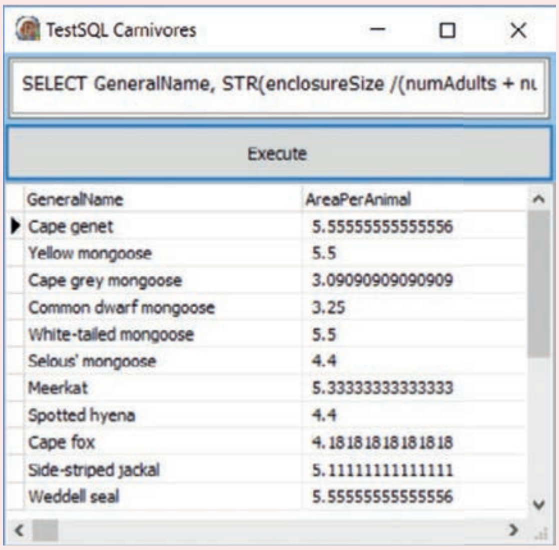
To fix this, you can use the four functions shown in the table below as part of your SQL commands:

FUNCTION	DESCRIPTION	SYNTAX
INT	Returns a whole number, discarding any decimal value.	INT(field)
ROUND	Rounds a number to the indicated number of decimals.	ROUND(field, decimals)
STR	Returns a DateTime or number as a string.	STR(field)
FORMAT	Returns a DateTime or number as a string in a specific format.	FORMAT(field, formatstring)

Example 4.18 *continued*

Replace the SELECT clause above with the ones below.

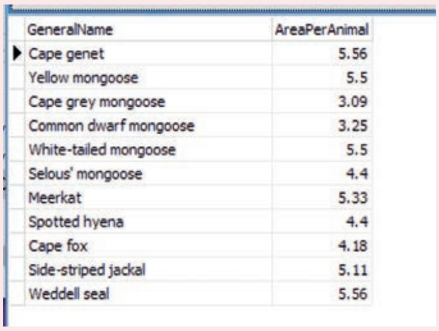
```
SELECT GeneralName, STR(enclosureSize /(numAdults + numYoung)) AS
AreaPerAnimal FROM tblCarnivores;
```



SELECT GeneralName, STR(enclosureSize /(numAdults + nu

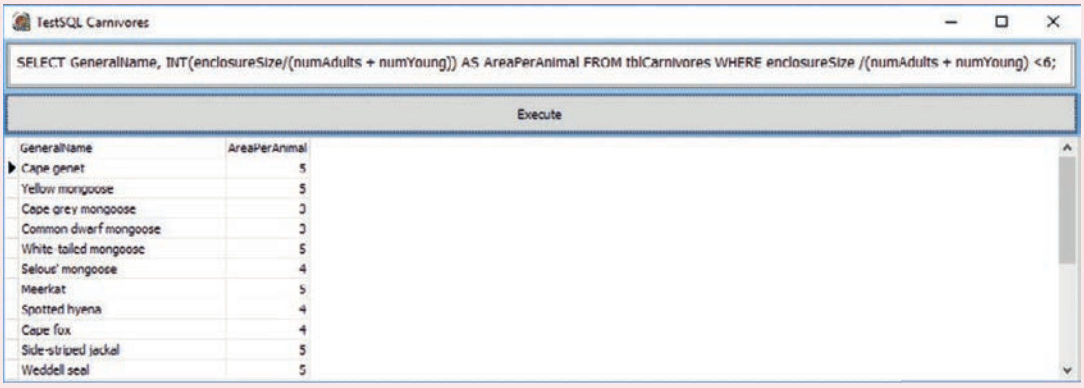
GeneralName	AreaPerAnimal
Cape genet	5.55555555555556
Yellow mongoose	5.5
Cape grey mongoose	3.09090909090909
Common dwarf mongoose	3.25
White-tailed mongoose	5.5
Selous' mongoose	4.4
Meerkat	5.33333333333333
Spotted hyena	4.4
Cape fox	4.18181818181818
Side-striped jackal	5.11111111111111
Weddell seal	5.55555555555556

```
SELECT GeneralName, ROUND(enclosureSize /(numAdults + numYoung),2) AS
AreaPerAnimal FROM tblCarnivores;
```



GeneralName	AreaPerAnimal
Cape genet	5.56
Yellow mongoose	5.5
Cape grey mongoose	3.09
Common dwarf mongoose	3.25
White-tailed mongoose	5.5
Selous' mongoose	4.4
Meerkat	5.33
Spotted hyena	4.4
Cape fox	4.18
Side-striped jackal	5.11
Weddell seal	5.56

```
SELECT GeneralName, INT(enclosureSize /(numAdults + numYoung)) AS
AreaPerAnimal FROM tblCarnivores;
```



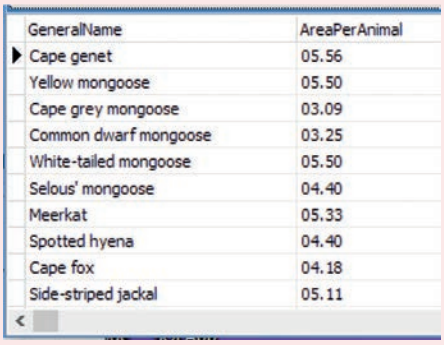
SELECT GeneralName, INT(enclosureSize/(numAdults + numYoung)) AS AreaPerAnimal FROM tblCarnivores WHERE enclosureSize /(numAdults + numYoung) <6;

GeneralName	AreaPerAnimal
Cape genet	5
Yellow mongoose	5
Cape grey mongoose	3
Common dwarf mongoose	3
White-tailed mongoose	5
Selous' mongoose	4
Meerkat	5
Spotted hyena	4
Cape fox	4
Side-striped jackal	5
Weddell seal	5

Example 4.18 *continued*

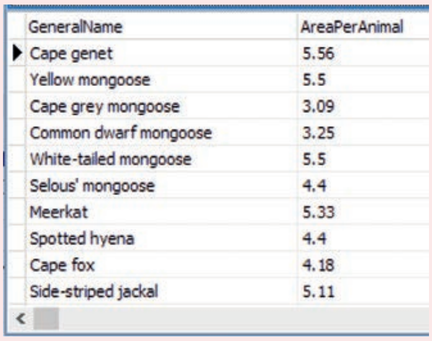
```
SELECT GeneralName, FORMAT(enclosureSize /(numAdults + numYoung),'00.00')
AS AreaPerAnimal FROM tblCarnivores;
```

‘00.00’



GeneralName	AreaPerAnimal
Cape genet	05.56
Yellow mongoose	05.50
Cape grey mongoose	03.09
Common dwarf mongoose	03.25
White-tailed mongoose	05.50
Selous' mongoose	04.40
Meerkat	05.33
Spotted hyena	04.40
Cape fox	04.18
Side-striped jackal	05.11

Output when ‘00.00’ is replaced with ‘##.##’



GeneralName	AreaPerAnimal
Cape genet	5.56
Yellow mongoose	5.5
Cape grey mongoose	3.09
Common dwarf mongoose	3.25
White-tailed mongoose	5.5
Selous' mongoose	4.4
Meerkat	5.33
Spotted hyena	4.4
Cape fox	4.18
Side-striped jackal	5.11

Take note that the format string from the FORMAT function takes similar string values to Delphi’s. A few FORMAT examples with their outputs are shown below:

- FORMAT(release_date, ‘dd mmmm yyyy’) — 15 October 2018
- FORMAT(release_date, ‘dd-mm-yy’) — 15-10-18
- FORMAT(release_date, ‘dd mmm yyyy hh:nn:ss’) — 15 Oct 2018 12:15:52
- FORMAT(income, ‘0.00’) — 15130000000.00
- FORMAT(income, ‘0.##’) — 15130000000
- FORMAT(income, “Currency”) — R15,130,000,000.00
- FORMAT(NumYoung/(NumYoung + NumAdults), ‘0%’) — 84%

Activity 4.12

4.12.1 How does the use of INT change the format of the output *AreaPerAnimal* compared to the original output?

4.12.2 Explain the difference in output when ‘00.00’ and ‘##.##’ was used to display *AreaPerAnimal*.

4.12.3 What will the *AreaPerAnimal* output look like if the format string is ‘#.00’?

Create the following queries with calculated fields using a pen and paper, making sure to give each calculated field a meaningful name.

4.12.4 Show all movies’ titles and incomes (converted to an integer).

4.12.5 Show all movies’ titles and incomes (rounded to 1 decimal point).

4.12.6 Show the movies’ titles and scores (divided by 100 and formatted as a percentage).

4.12.7 Show only the release dates of movies (formatted as 15/12/2018).

4.12.8 Show the movies’ titles, *income* (formatted as a currency) and *score* (divided by 100 and formatted as a percentage).

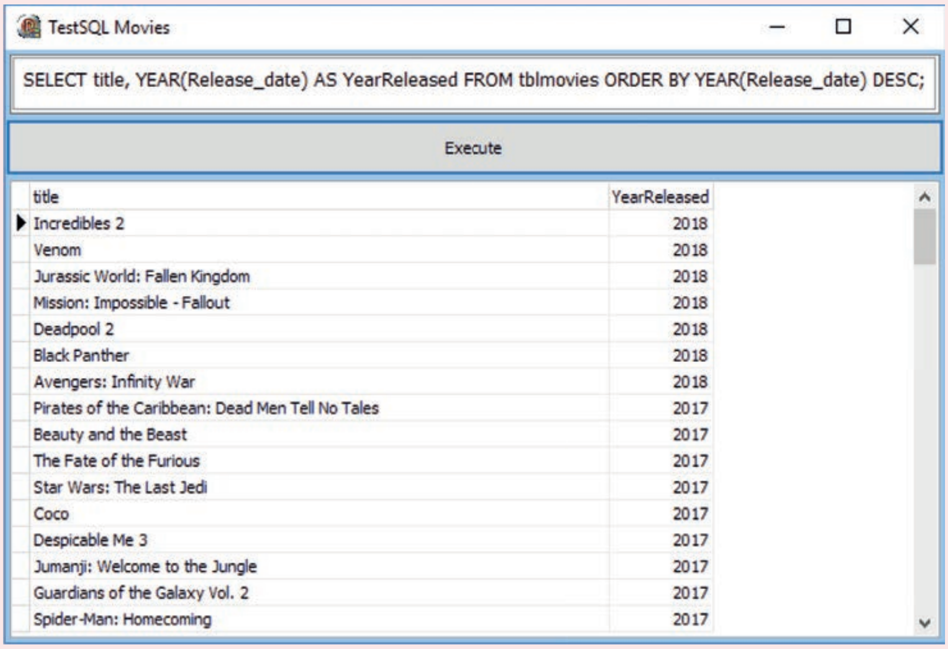
DATE FUNCTIONS

When creating calculated columns, the names of functions (such as YEAR, MONTH, and DAY) cannot be used as field names. See list of DATE functions in Unit 4.2 on page 127.

Example 4.19

Select all movie titles and the year of their release and display the latest movies first.

```
SELECT title, YEAR(Release_date) AS YearReleased FROM tblMovies ORDER BY
YEAR(Release_date) DESC;
```



```
SELECT title, YEAR(Release_date) AS YearReleased FROM tblmovies ORDER BY YEAR(Release_date) DESC;
```

title	YearReleased
Incredibles 2	2018
Venom	2018
Jurassic World: Fallen Kingdom	2018
Mission: Impossible - Fallout	2018
Deadpool 2	2018
Black Panther	2018
Avengers: Infinity War	2018
Pirates of the Caribbean: Dead Men Tell No Tales	2017
Beauty and the Beast	2017
The Fate of the Furious	2017
Star Wars: The Last Jedi	2017
Coco	2017
Despicable Me 3	2017
Jumanji: Welcome to the Jungle	2017
Guardians of the Galaxy Vol. 2	2017
Spider-Man: Homecoming	2017

Activity 4.13

Use MoviesSQL app with the databases “Movies.mdb”; create and test queries that will select the following data.

4.13.1 Show all movie titles, release_dates and the year of their release.

4.13.2 Show all the movie titles released on the first day of the month.

4.13.3 Show all the movie titles released in the last three months of the year.

4.13.4 Show all the movie titles released in 2009.

4.13.5 Show all the movie titles release dates and their age (in days).

4.13.6 Show all the movie titles release dates and their age (in months)

STRING FUNCTIONS

Just like Delphi, SQL allows you to manipulate strings in different ways. The table below shows four different string functions you can use in your SQL queries.

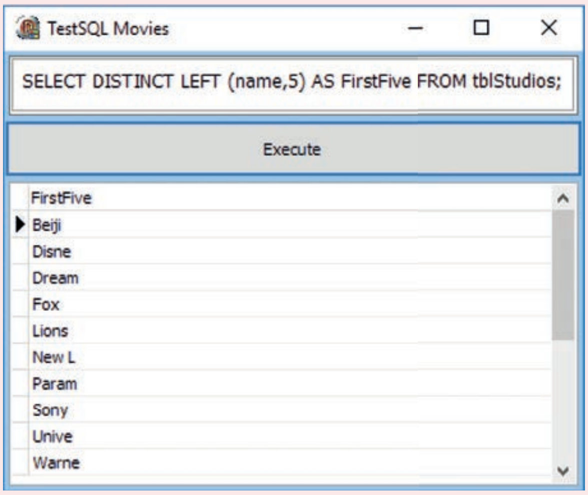
Table 4.5: *Four different string functions used in a SQL query*

FUNCTION	DESCRIPTION	SYNTAX
LEN	Returns the number of characters in a string.	LEN(string)
LEFT	Returns the indicated number of characters from the start of the string.	LEFT(string, num_Characters)
RIGHT	Returns the indicated number of characters from the end of the string.	RIGHT(string, num_Characters)
MID	Returns the indicated number of characters from a specified point in the string.	MID(string, first_character, characters)

Example 4.20

Show the first five characters of all studios' names.

```
SELECT DISTINCT LEFT (name, 5) AS FirstFive FROM tblStudios;
```



```
SELECT DISTINCT LEFT (name,5) AS FirstFive FROM tblStudios;
```

```
FirstFive
Beiji
Disne
Dream
Fox
Lions
New L
Param
Sony
Unive
Warne
```

Activity 4.14

Create the following queries using string functions. Make sure to give each calculated field a relevant name.

4.14.1 Show the second, third and fourth character of all studios' cities.

4.14.2 Show all movies' titles, as well as the length of the titles.

4.14.3 Show the first and last letter of all the studios' names.

4.14.4 Show the first 2 letters of the enclosure name from the table carnivores as the Enclosure types.

COMBINING STRINGS

The final type of calculated field is created by combining two strings. To do this, you simply add the one string to the second string using the plus (+) operator. This can be combined with functions such as STR and FORMAT to combine numbers or dates with strings.

Example 4.21

Appending the '%' symbol to the score.

Integer score to percentage

```
SELECT title, STR(score) + '%' AS score_percentage FROM tblMovies;
```

Activity 4.15

Create the following queries using *tblMovies* from the **Movies.mdb** database and the *tblVetVisits* from the **Carnivores.mdb** database with a combined string using pen and paper, making sure to give each calculated field a relevant name.

4.15.1 Show all movie titles' and incomes, where the *income* is converted to billions (i.e. divide by 1 000 000 000) and the string 'bn' is added to the end of it.

4.15.2 Show the movie studios' names and locations (as 'City, Province, Country').

4.15.3 Show all animals where the *followUp* field has the value false. List the animal name, reason for visit and the *Animal_ID*, and the string, 'No follow up required'.

Always remember to consider spaces when combining strings. In the third question, a space is added after each comma.

Activity 4.16 Calculated fields

Write down the following queries using *tblMovies* from the **Movies.mdb** database and **tblVetVisits** from the **Carnivores.mdb** database using pen and paper. Make sure that all calculated fields have appropriate field names.

4.16.1 Show all the movie titles' release dates and their age (in years).

4.16.2 Show month and day of vet's visits.

4.16.3 Select all movies released on the 12th day of the month.

4.16.4 Select the movie titles and the *income* divided by 1.15 to remove VAT. Use an appropriate heading and format.

4.16.5 Select all titles and show the year, month and day of release separately from the *tblMovies* table.

4.16.6 Display the first two and last two letters of each movie's *title*.

UNIT 4.4 Aggregate functions

Up to now, all the calculated fields have manipulated the values of an individual record. This is useful if you want to work with the data from records but is not very useful if you want to work with the overall data from the dataset. When you want to answer question about the overall dataset, you need to use the aggregate SQL functions.

In this section, you will learn about five aggregate functions.

AGGREGATE FUNCTION

https://www.youtube.com/watch?v=0s1w50wLC1w

Table 4.6: *Five aggregate functions*

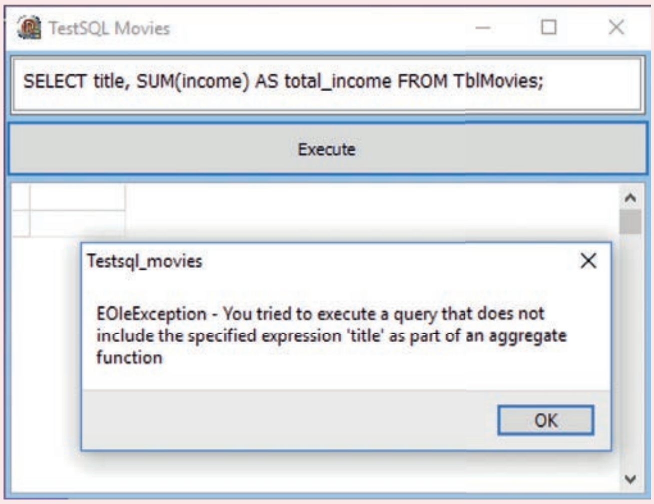
FUNCTION	DESCRIPTION	SYNTAX
SUM	Calculates the sum of all values in a field.	SUM(field_name)
AVG	Calculates the average value for a field.	AVG(field_name)
MIN	Returns the minimum value in a field.	MIN(field_name)
MAX	Returns the maximum value in a field.	MAX(field_name)
COUNT	Counts the number of records that have values in a field.	COUNT(field_name)

Each of these functions will complete a calculation on your dataset and return a single value as the result. Since these functions return a single value (rather than a value for each record), they cannot be used in a query with other fields (such as *title* or *genre*).

Example 4.22

```
SELECT title, SUM(income)
AS total_income
FROM tblMovies;
```

Doing this will result in an error.



SELECT title, SUM(income) AS total_income FROM TblMovies;

Testsql_movies

EOleException - You tried to execute a query that does not include the specified expression 'title' as part of an aggregate function

Figure 4.18: *Aggregate function with a normal field*

Here is the syntax for the aggregate function:

syntax for aggregate functions

```
SELECT Aggregate_function(field_name) AS calculated_name
FROM table_name WHERE condition;
```

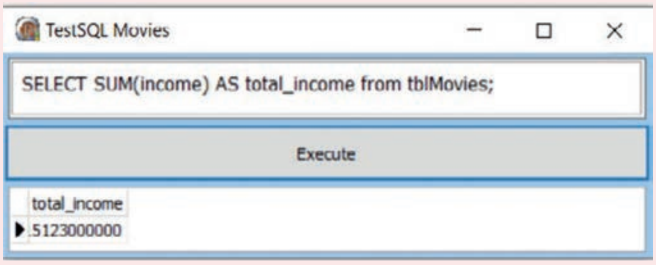
There are hundreds of different uses for the aggregate function. It allows you to answer questions about your entire dataset. Let's look at some examples:

Example 4.23

To find the sum of the income from all movies, you could do the following query:

SUM

```
SELECT SUM(income) AS total_income FROM tblMovies;
```



```
SELECT SUM(income) AS total_income from tblMovies;
```

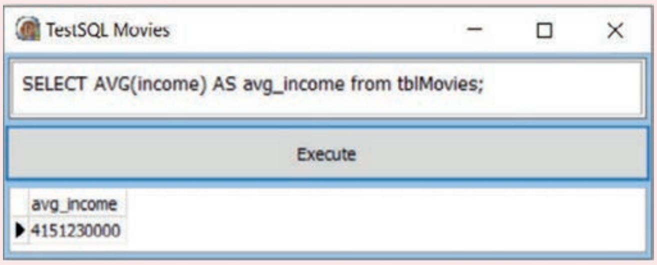
```
total_income
5123000000
```

Example 4.24

To find the average of the income from all movies, you could write the following query:

AVG

```
SELECT AVG(income) AS avg_income FROM tblMovies;
```



```
SELECT AVG(income) AS avg_income from tblMovies;
```

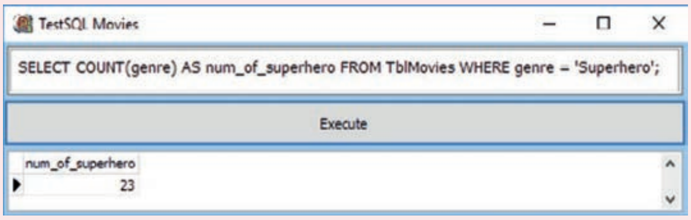
```
avg_income
4151230000
```

Example 4.25

To find the lowest score given to an action movie, you could use the following query.

MAX or MIN example

```
SELECT MIN(score) as lowest_score
FROM tblMovies
WHERE genre = 'Action';
```



```
SELECT COUNT(genre) AS num_of_superhero FROM TblMovies WHERE genre = 'Superhero';
```

```
num_of_superhero
            23
```

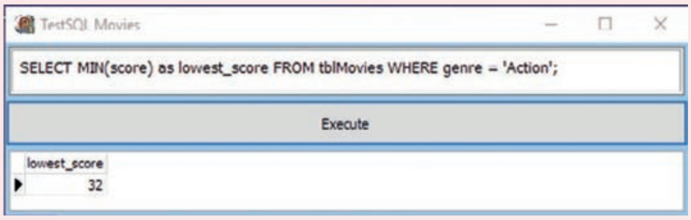
Figure 4.19: *Finding the lowest score given to an action movie*

Example 4.26

To count the number of superhero movies, you could write the following query:

COUNT with WHERE

```
SELECT COUNT(genre) AS num_of_superhero
FROM tblMovies
WHERE genre = 'Superhero';
```



```
SELECT MIN(score) as lowest_score FROM tblMovies WHERE genre = 'Action';
```

```
lowest_score
         32
```

Figure 4.20: *Number of superhero movies*

Activity 4.17

Using a pen and paper, write down SQL queries to answer the following questions.

4.17.1 How many movies have a score below 50?

4.17.2 What is the maximum income earned by any movie?

4.17.3 What is the minimum score earned by an animated movie?

4.17.4 What is the total income earned by all movies containing the words ‘Harry Potter’ in their title?

4.17.5 What was the average score received by all movies from *studio_id2*?

Once complete, use these queries in the *MoviesSQL* App.

GROUP BY

The problem with aggregate functions is that they only return a single value. If you want to compare the scores obtained by different genres of movies, you would need to write a separate query or create a separate calculated field for each genre. This is not very practical, especially not with large databases. To fix this problem, you can use the GROUP BY clause.

As the name suggests, the GROUP BY clause groups your records according to their value in a specific field. Once this has been done, the aggregate function is applied to each of these groups. The specific field must also be included in the SELECT clause to serve as a label for the groups. The GROUP BY clause comes at the end of the statement after the FROM and WHERE clauses.

The GROUP BY clause uses the following syntax:

GROUP BY syntax

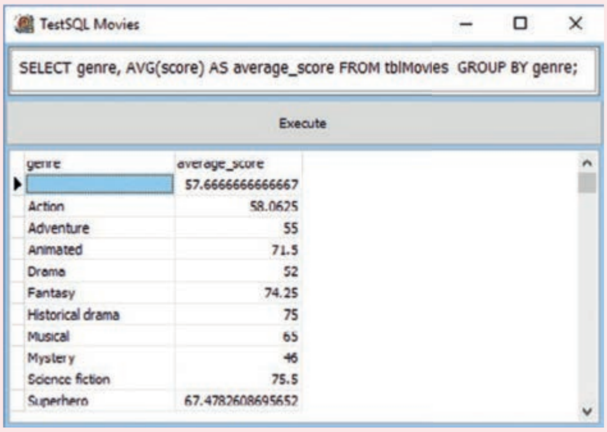
```
SELECT group_field_name, aggregate_function (value_field_name)
AS result_field_name
FROM table_name
GROUP BY group_field_name;
```

Example 4.27

Using the GROUP BY clause you could group all of the movies in your database according to genre and then calculate the average score for each genre. Using this syntax, you can view the average score of the different genres of movies in your database with the following query.

GROUP BY example

```
SELECT genre, AVG(score) AS average_score
FROM tblMovies
GROUP BY genre;
```



```
SELECT genre, AVG(score) AS average_score FROM tblMovies  GROUP BY genre;
```

genre	average_score
	57.6666666666667
Action	58.0625
Adventure	55
Animated	71.5
Drama	52
Fantasy	74.25
Historical drama	75
Musical	65
Mystery	46
Science fiction	75.5
Superhero	67.4782608695652

Figure 4.21: *Average score of different genres of movies*

Activity 4.18

Create queries from *tblCarnivores* that show the following information.

4.18.1 The number of movies on the list, per *genre*.

4.18.2 The total *income* earned, per studio.

4.18.3 The release date of the most recently released movie by each studio.

4.18.4 The average score of movies earning more than R14 000 000 000, per genre.

4.18.5 The number of movies earning more than R14 000 000 000, per studio.

HAVING

The HAVING clause allows you to add a condition to the grouped results, so that only groups meeting the condition will be shown. This is done by placing the HAVING clause after the GROUP BY clause, as shown in the syntax below.

HAVING syntax

```
SELECT group_field_name, aggregate_function(value_field_name)
AS result_field_name
FROM table_name
WHERE condition
GROUP BY group_field_name
HAVING group_condition;
```

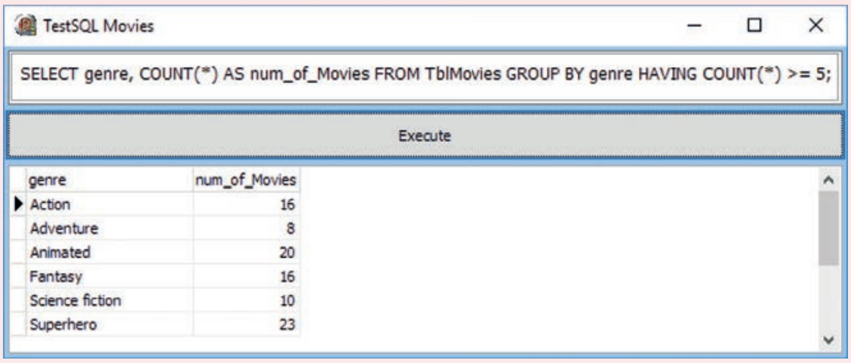
Using this syntax, the group condition will check if an aggregate function returns a result that is larger than, smaller than, or equal to a specific value.

Example 4.28

Create a query that shows number of movies per genre where the genre has five or more movies.

HAVING

```
SELECT genre, COUNT(*) AS num_of_Movies
FROM tblMovies
GROUP BY genre
HAVING COUNT(*) >= 5;
```



```
SELECT genre, COUNT(*) AS num_of_Movies FROM TblMovies GROUP BY genre HAVING COUNT(*) >= 5;
```

genre	num_of_Movies
Action	16
Adventure	8
Animated	20
Fantasy	16
Science fiction	10
Superhero	23

Figure 4.22: *Genres with more than 5 movies*

The calculated field name cannot be used in the HAVING clause, only the actual calculation.

Activity 4.19

Using the HAVING clause, create queries that show the following information.

4.19.1 The number of movies per *studio_id*, only showing studios with 10 or more movies.

4.19.2 The total *income* per studio where the total income earned is greater than R150 000 000 000.

4.19.3 The average score per *genre* where there are at least 5 movies in the *genre*.

Take note

In 4.19.3, the 'group' condition (HAVING clause) uses a different aggregate function to the aggregate function in the calculated field.

Activity 4.20 Aggregate functions

Write down the following queries using pen and paper.

- Make sure that all calculated fields have field names.
- Do not show any empty records.
- Use appropriate formatting.

4.20.1 Find the earliest release date for any movie.

4.20.2 Find the average *score* earned by all movies, rounded to 2 decimals.

4.20.3 Find the highest *income* earned by an animated movie.

4.20.4 Find the number of movies with an *income* larger than R15 billion.

4.20.5 Find the average *income* of movies with a *score* above 90.

4.20.6 Find the average *income* of movies based on their *genre*.

4.20.7 Find the maximum *score* of movies based on their studio.

4.20.8 Find the maximum *score* of movies based on their studio. Do not show empty studio_id's and display the studio names instead of the studio id number.

4.20.9 Find the minimum *score* of movies based on their release year (a calculated column).

4.20.10 Show the total *income* of all genres that have earned more than R100 billion.

4.20.11 Show the average *score* of all studios, rounded to 2 decimals, that have a *score* above 70.

UNIT

4.5 Data maintenance

In the previous units, you learned how to use the SELECT statement to select and display data from your database. In this unit, you will learn how you can make changes to a database by using the INSERT INTO, UPDATE and DELETE statements. These statements change the data values in the database.

HOW TO CONNECT AND INTERACT WITH A DATABASE

https://www.youtube.com/watch?v=dwb0wv6lJqA

INSERT INTO

To add additional records to a database, you use the INSERT INTO statement. The syntax for this statement is shown in the code snippet below:

INSERT INTO

```
INSERT INTO table_name (field_names)
VALUES (new_values);
```

When using the INSERT INTO statement, the different field names are separated using commas. Similarly, the values added to those fields must also be separated with commas and must be in the same order (and data type) as the fields listed in the previous line/listed field names. The field names in the first bracket can be omitted if values for ALL the fields are inserted. The values must then be in the same order as the field names in the database table.

Example 4.29

Create the following query to add the movie “Aquaman” to the tblMovies table:

INSERT INTO

```
INSERT INTO tblMovies (id, title, studio_id, income,
release_date, score, genre, studio_ID)
VALUES (101, "Aquaman", 4, 14353000000, #2018/12/21#,
55, “Superhero”, );
```

When inserting data into a table, pay careful attention to the following:

- The field names are spelled correctly.
- All field names are separated by a comma.
- The values are added in the same order and data type, as the field names.
- A value is added to the primary key field.
- The value added to the primary key field is unique.
- All string values are surrounded by double quotation marks.
- All date values are surrounded by the hash (#) symbol.

Activity 4.21

Using a pen and paper, write down the queries needed to add the following data to the correct tables.

4.21.1 *tblMovies* table:

ID	TITLE	STUDIO_ID	INCOME	RELEASE_DATE	SCORE	GENRE
102	Bohemian Rhapsody	3	10846000000	2 November 2018	49	Musical
103	Fantastic Beasts: The Crimes of Grindelwald	4	9069000000	16 November 2018	52	Fantasy
104	Ant-Man and the Wasp	2	8718000000		70	Superhero

4.21.2 *tblStudios* table

STUDIO_ID	NAME	CITY	PROVINCE	COUNTRY
11	Polybona Films	Beijing	Hebei	China

Rather than creating three INSERT INTO statements to add movies to your database, you can use a single query where the new records are separated by commas, as shown in the syntax below.

INSERT INTO syntax (multiple records)

```
INSERT INTO table_name (field_names)
VALUES (new_values1),
(new_values2),
…,
(new_values3);
```

As with a single record, you need to make sure that the values of the three records are aligned with the field names. If the record does not have a value for a specific field, a comma should still be added but the space before the next comma can be empty.

Example 4.30

The code below inserts the three movies Bohemian Rhapsody, Fantastic Beast and Ant-Man and the Wasp.

```
INSERT INTO tblMovies (ID, title, studio_id, income, release_date, score,
genre)
VALUES
(102, "Bohemian Rhapsody", 3, 10846000000, #2018/11/2#, 49, "Musical"),
(103, "Fantastic Beasts: The Crimes of Grindelwald", 4, 9069000000,
#2018/11/16#, 52, "Fantasy"),
(104, "Ant-Man and the Wasp", 2, 8718000000,, 70, "Superhero");
```

Take note

The *release_date* of Ant-Man and the Wasp is not known and therefore left blank between the commas.

UPDATE

The UPDATE statement allows you to make permanent changes to a record's values. This is useful to fix mistakes, update information or add missing information to existing records. However, if used incorrectly, it also has the ability to replace the data from your entire database with garbage, so it is important to be very careful when using the UPDATE statement.

The UPDATE statement has the following syntax:

UPDATE syntax

```
UPDATE table_name
SET field_name1 = Newvalue, field_name2 = Newvalue, ...
WHERE condition;
```

As with most SQL queries, you start by selecting the table you will be using. In the second line, you select the fields and set the new values. When setting the values, make sure that the values are in the correct format and follow the field's rules (such as adding a unique value for the primary key). Finally, in the last line, you use the WHERE statement to select the fields whose values you would like to change.

Example 4.31

In the *tblMovies* table, the movie 'Independence Day' does not have an income or genre. To add values to these fields, you can use the following query:

UPDATE example

```
UPDATE tblMovies
SET income = 11444000000, genre = 'Science Fiction'
WHERE title = 'Independence Day';
```

The WHERE clause is incredibly important when updating values, since the changes made by the UPDATE statement will be applied to each record meeting the WHERE clause conditions. If you leave out the WHERE clause, you will update each record in your table and have no way of recovering the old data.

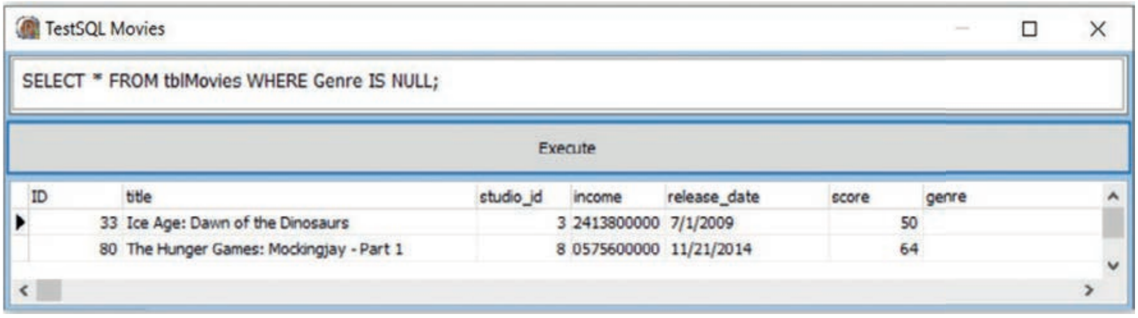
In order to ensure you do not update the incorrect records, follow these guidelines when creating a WHERE clause:

- When updating a single record, you can use the primary key to select the correct record.
- Before making changes to the data, first create a separate SELECT query to ensure that you are only selecting the appropriate records before using UPDATE statement.

With this information in mind, complete the following activity.

Activity 4.22

Take a look at the image below.



```
SELECT * FROM tblMovies WHERE Genre IS NULL;
```

ID	title	studio_id	income	release_date	score	genre
33	Ice Age: Dawn of the Dinosaurs	3	2413800000	7/1/2009	50	
80	The Hunger Games: Mockingjay - Part 1	8	0575600000	11/21/2014	64	

4.22.1 Using a pen and paper, write down the queries needed to add the missing information to the films.

MOVIE NAME	GENRE
Ice Age	Animals
Hunger Games	Action

Take note

The Lord of the Rings: The Two Towers was released in 2002 on 18 December.

4.22.2 Write down an SQL query that you use to increase the *income* of all films before the year 2000 by 10%.

DELETE FROM

The final SQL statement you will learn about is DELETE FROM, which deletes all records meeting the specified condition. The syntax for DELETE FROM is given in the code snippet below.

DELETE syntax

```
DELETE FROM table_name
WHERE condition;
```

As with the UPDATE statement, it is incredibly important that only the records you want to delete meet the WHERE condition, since any other records meeting this condition will also be deleted.

Example 4.32

To delete the 100th record from the tblMovies table, you could use the following query.

DELETE examples

```
DELETE FROM tblMovies
WHERE id = 100;
```

Activity 4.23

Using pen and paper, write down queries that will delete the following records:

4.23.1 The movie released on 5 May 2017.

4.23.2 The movie called 'Despicable Me 2'

4.23.3 All superhero movies.

4.23.4 All movies with 'Star Wars' in the title.

4.23.5 All movies released before the 5th day of the month.

4.23.6 All movies with a *score* below 60 and an *income* below 14 000 000 000.

Take note

First create a backup for the original disk before you write the update and delete SQLs

Activity 4.24 Data maintenance

Write down the following queries using pen and paper. Make sure that all calculated fields have field names.

4.24.1 Add the following South African film studio to the *tblStudios* table.

STUDIO_ID	NAME	FULL_NAME	CITY	PROVINCE	COUNTRY
12	CTFS	Cape Town Film Studios	Cape Town	Western Cape	South Africa

4.24.2 Add the following films to the *tblMovies* table using a single query.

ID	TITLE	STUDIO_ID	INCOME	RELEASE_DATE	SCORE	GENRE
105	Ready Player One	3		20 July 2018	64	Science Fiction
106	The Meg	3	7423000000	10 August 2018		
107	Mamma Mia! Here We Go Again	5	5526000000		60	Musical

4.24.3 Update the *score* of the movie with the ID 106 to 46.

4.24.4 Change the *genre* of all science fiction movies to 'Science-Fiction'.

4.24.5 Delete all movies released before the year 2000 from the *tblMovies* table.

4.24.6 Delete all studios not based in the United States from the *tblStudios* table.

UNIT

4.6 Querying two tables

In a relational database, there are many situations where you may want to select data from more than one table at the same time. To do this, you will use a special condition in the WHERE clause to link the two tables.

To create a link between two tables, you use the following syntax.

Join syntax

```
SELECT table1.field_name, table2.field_name
FROM table1, table2
WHERE table1.foreign_key = table2.primary_key;
```

As the syntax shows:

- The first line selects the field names and the tables from which they must be taken.
- The second line lists both the table names in the FROM clause, separating the table names with a comma.
- In the WHERE clause, you set the foreign key of one table equal to the primary key of the other table. This condition creates the link between the tables. If this condition is omitted in the WHERE clause your result set will display a lot of duplicate values.

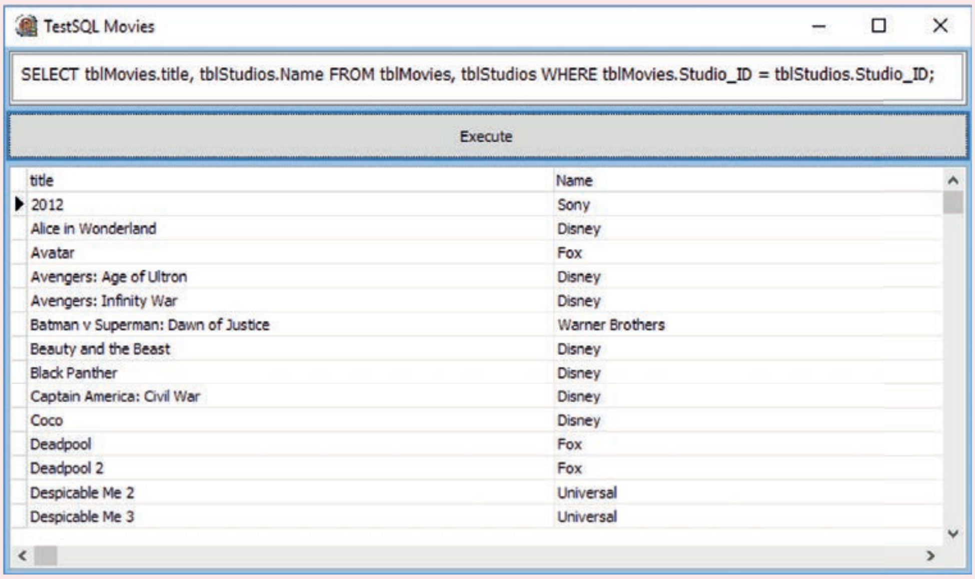
Example 4.33

The query below will show the name of the film as well as the name of the studio that produced it.

Join example

```
SELECT tblMovies.title, tblStudios.name
FROM tblMovies, tblStudios
WHERE tblMovies.studio_id = tblStudios.studio_id;
```

Running this query will return the following results:



```
SELECT tblMovies.title, tblStudios.Name FROM tblMovies, tblStudios WHERE tblMovies.Studio_ID = tblStudios.Studio_ID;
```

title	Name
2012	Sony
Alice in Wonderland	Disney
Avatar	Fox
Avengers: Age of Ultron	Disney
Avengers: Infinity War	Disney
Batman v Superman: Dawn of Justice	Warner Brothers
Beauty and the Beast	Disney
Black Panther	Disney
Captain America: Civil War	Disney
Coco	Disney
Deadpool	Fox
Deadpool 2	Fox
Despicable Me 2	Universal
Despicable Me 3	Universal

Figure 4.23: *Movie titles and studio names*

Example 4.33 *continued*

To filter this selection, you need to add a second condition to the WHERE command using the Boolean operator AND. This is shown in the syntax below.

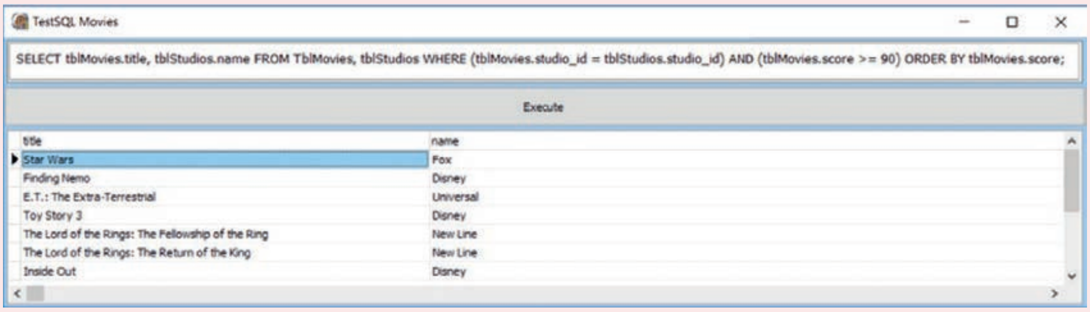
Join syntax with a condition

```
SELECT table1.field_name, table2.field_name
FROM table1, table2
WHERE table1.foreign_key = table2.primary_key AND condition;
```

If you would like to show the movie title and studio names of all movies with a score greater than or equal to 90, you could create the following SQL query.

Join with the condition example

```
SELECT tblMovies.title, tblStudios.name
FROM tblMovies, tblStudios
WHERE (tblMovies.studio_id = tblStudios.studio_id)
AND (tblMovies.score >= 90)
ORDER BY tblMovies.score;
```



```
SELECT tblMovies.title, tblStudios.name FROM TblMovies, tblStudios WHERE (tblMovies.studio_id = tblStudios.studio_id) AND (tblMovies.score >= 90) ORDER BY tblMovies.score;
```

title	name
Star Wars	Fox
Finding Nemo	Disney
E.T.: The Extra-Terrestrial	Universal
Toy Story 3	Disney
The Lord of the Rings: The Fellowship of the Ring	New Line
The Lord of the Rings: The Return of the King	New Line
Inside Out	Disney

Figure 4.24: *Filtered list of movie titles and studio names*

The previous SQL-statement got too long and can be shortened by introducing aliases for the table names. We have used aliases for calculated fields using the AS keyword, similarly we can use the AS keyword to set single-letter as an alias for a table name. Why? Count how many times tblMovies was used in the previous SQL statement and you will see how much shorter the statement will be if tblMovies gets replaced with the letter M. See the code below.

Note: once a table alias is used it must be applied everywhere in the statement.

using table aliases

```
SELECT M.title, S.name
FROM tblMovies AS M, tblStudios AS S
WHERE (M.studio_id = S.studio_id) AND (M.score >= 90)
ORDER BY M.score;
```

Activity 4.25

By joining the *tblMovies* and *tblStudios* table, create queries that show the following information.

4.25.1 The *title* and *date* from the *tblMovies* table and the *name* from the *tblStudios* table.

4.25.2 The *title*, *score* and *income* from the *tblMovies* table and the *city* and *province* from the *tblStudios* table.

4.25.3 The *title*, *score* and *income* from the *tblMovies* table and the *name* from the *tblStudios* table where the *score* is below 40.

4.25.4 The *title* from the *tblMovies* table and the *name* from the *tblStudios* table where the *city* is Los Angeles.

4.25.5 The *title*, *score* and *income* from the *tblMovies* table and the *name* from the *tblStudios* table where the *income* is above 15000000000.

Activity 4.26 Querying two tables

Write down the following queries and run them using the **MoviesSQL App.** Make sure that all calculated fields have field names.

4.26.1 The *title*, *score* and movie studio *name* for all movies.

4.26.2 The *title*, release year and studio *city* for all movies.

4.26.3 All fields from the *tblMovies* and *tblStudios* tables.

4.26.4 All fields from movies with a *score* above 90 not made in Hollywood.

4.26.5 All superhero movies made by Disney.

4.26.6 The average *score* of all movies, grouped by studio name.

4.26.7 Run the **CitiesInSA_App.exe**, write down queries that will answer the following questions, as well as the answers to these questions.

a. What is the largest city in Mpumalanga?

b. In which province is the city Allemansvlei?

c. What is the longitude and latitude of Kwazulu-Natal's capital city?

d. In which province is the city Nkwali and what is the province's population?

e. What are the capital cities and largest cities of all provinces with a population greater than 6 million people?

UNIT

4.7 Database applications

When creating an application that uses SQL, you should not expect your users to be able to write SQL queries. Instead, you should create applications that automatically builds the SQL queries based on the input from your application's components.

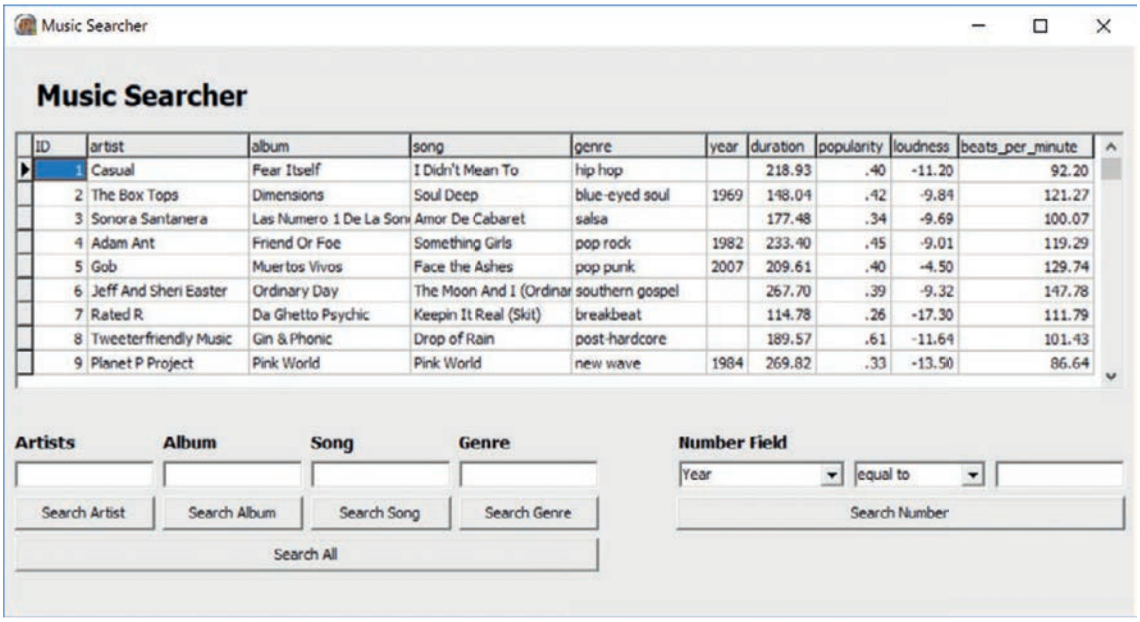
In this unit, you will create an application that builds these queries automatically using data entered in components.

THE MUSIC SEARCHER APP

For this application, the user interface, database and database connection has already been created. Your task is to build the SQL queries. Open the project in the Music searcher folder to enter the example code and complete Activity 4.27.

Activity 4.27

The user interface is shown below. As the image shows, the user interface contains a grid with a lot of information about different songs from a "music" table. At the bottom of the user interface you can perform different searches.



ID	artist	album	song	genre	year	duration	popularity	loudness	beats_per_minute
1	Casual	Fear Itself	I Didn't Mean To	hip hop		218.93	.40	-11.20	92.20
2	The Box Tops	Dimensions	Soul Deep	blue-eyed soul	1969	148.04	.42	-9.84	121.27
3	Sonora Santanera	Las Numero 1 De La Son	Amor De Cabaret	salsa		177.48	.34	-9.69	100.07
4	Adam Ant	Friend Or Foe	Something Girls	pop rock	1982	233.40	.45	-9.01	119.29
5	Gob	Muertos Vivos	Face the Ashes	pop punk	2007	209.61	.40	-4.50	129.74
6	Jeff And Sheri Easter	Ordinary Day	The Moon And I (Ordinar	southern gospel		267.70	.39	-9.32	147.78
7	Rated R	Da Ghetto Psychic	Keepin It Real (Skit)	breakbeat		114.78	.26	-17.30	111.79
8	Tweeterfriendly Music	Gin & Phonic	Drop of Rain	post-hardcore		189.57	.61	-11.64	101.43
9	Planet P Project	Pink World	Pink World	new wave	1984	269.82	.33	-13.50	86.64

4.27.1 The user must enter an artist name in the artist edit box. When the button [Search Artist] is clicked, the SQL query should return all records for the artist entered.

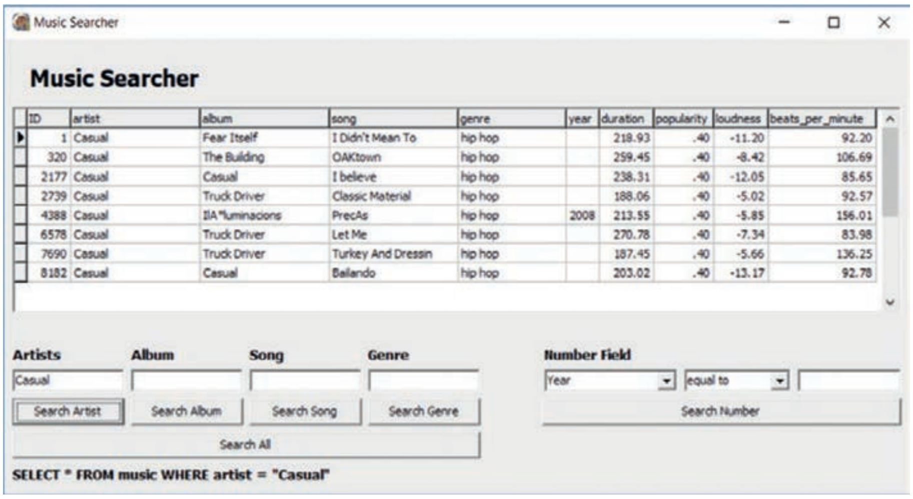
SQL query: a string in Delphi code

```
'SELECT * FROM music WHERE artist = "' + sArtist + '"';
```

NOTE: The double quotation marks around sArtist. This ensures that the value entered is inserted in the SQL statement as a string so that the test artist = "some value" won't give a data type mismatch error. Double quotation marks are not required if the field in the condition is not a String. For date tests the '#' symbols instead of the double quotes.

Activity 4.27 *continued*

If the user enters ‘Casual’ and click Search Artist output will be:



Music Searcher

ID	artist	album	song	genre	year	duration	popularity	loudness	beats_per_minute
1	Casual	Fear Itself	I Didn't Mean To	hip hop		218.93	.40	-11.20	92.20
320	Casual	The Building	OAKtown	hip hop		259.45	.40	-8.42	106.69
2177	Casual	Casual	I believe	hip hop		238.31	.40	-12.05	85.65
2739	Casual	Truck Driver	Classic Material	hip hop		188.06	.40	-5.02	92.57
4388	Casual	IIA "luminacions	PrecAs	hip hop	2008	213.55	.40	-5.85	156.01
6578	Casual	Truck Driver	Let Me	hip hop		270.78	.40	-7.34	83.98
7690	Casual	Truck Driver	Turkey And Dressin	hip hop		187.45	.40	-5.66	136.25
8182	Casual	Casual	Bailando	hip hop		203.02	.40	-13.17	92.78

Artists: Casual
Album:
Song:
Genre:
Number Field: Year; equal to

SELECT * FROM music WHERE artist = "Casual"

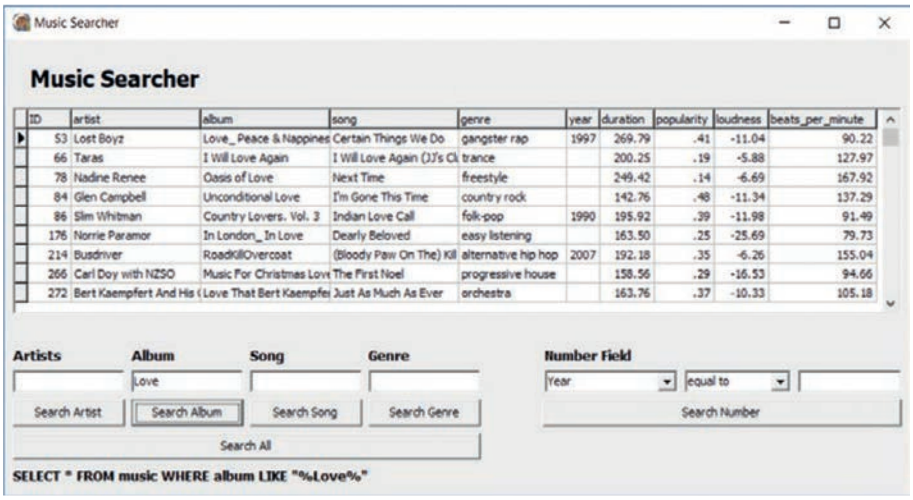
Search Artist event

```
var
  sSqlQuery : String;
  sArtist: String;
Begin
  sArtist := edtArtist.Text;
  sSqlQuery := 'SELECT * FROM music WHERE artist = "' + sArtist + '"';
  lblSqlQuery.Caption := sSqlQuery;
  dmoMusic.qryMusic.SQL.Text := sSqlQuery;
  dmoMusic.qryMusic.active := true;
end;
```

4.27.2 When the button [Search Album] is clicked a SQL query is needed to search for the any album containing the word entered. Let sAlbum be the variable to hold the user input. Remember the LIKE operator and wildcard ‘%’ must be used.

SQL query: a string in Delphi code

```
‘SELECT * FROM music WHERE album LIKE "%' + sAlbum + '%"';
```



Music Searcher

ID	artist	album	song	genre	year	duration	popularity	loudness	beats_per_minute
53	Lost Boyz	Love_ Peace & Nappines	Certain Things We Do	gangster rap	1997	269.79	.41	-11.04	90.22
66	Taras	I Will Love Again	I Will Love Again (JJ's Cl	trance		200.25	.19	-5.88	127.97
78	Nadine Renee	Oasis of Love	Next Time	freestyle		249.42	.14	-6.69	167.92
84	Glen Campbell	Unconditional Love	I'm Gone This Time	country rock		142.76	.48	-11.34	137.29
86	Slim Whitman	Country Lovers. Vol. 3	Indian Love Call	folk-pop	1990	195.92	.39	-11.98	91.49
176	Norrie Paramor	In London_ In Love	Dearly Beloved	easy listening		163.50	.25	-25.69	79.73
214	Busdriver	RoadKillOvercoat	(Bloody Paw On The) Kil	alternative hip hop	2007	192.18	.35	-6.26	155.04
266	Carl Doy with NZSO	Music For Christmas Love	The First Noel	progressive house		158.56	.29	-16.53	94.66
272	Bert Kaempfert And His (	Love That Bert Kaempfe	Just As Much As Ever	orchestra		163.76	.37	-10.33	105.18

Artists:
Album: Love
Song:
Genre:
Number Field: Year; equal to

SELECT * FROM music WHERE album LIKE "%Love%"

Activity 4.27 *continued*

Search Album event

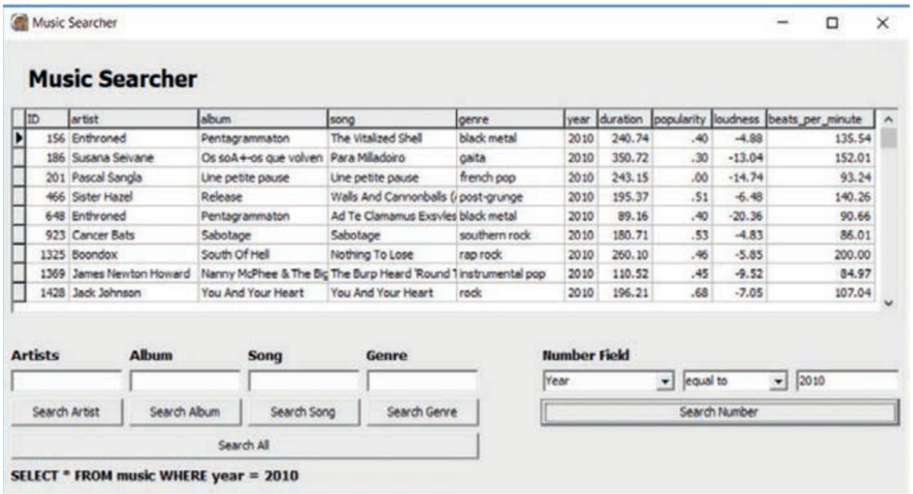
```
var
  sSqlQuery : String;
  sAlbum : String;
begin
  sAlbum := edtAlbum.Text;
  sSqlQuery := 'SELECT * FROM music WHERE album LIKE "%' + sAlbum +
  '%"';
  lblSqlQuery.Caption := sSqlQuery;
  dmoMusic.qryMusic.SQL.Text := sSqlQuery;
  dmoMusic.qryMusic.Active := True;
end;
```

4.27.3 When the button [Search Number] is clicked a SQL query must be built that includes the field selected, the relation operator selected and the value entered in the edit. Let *sField*, *sSymbol* and *sValue* hold the data entered by the user then the SQL statement will be:

SQL query: a string in Delphi code

```
'SELECT * FROM music WHERE '+ sField + sSymbol + sValue;
```

If the user is searching for songs released in the year 2010 the output will be:



ID	artist	album	song	genre	year	duration	popularity	loudness	beats_per_minute
156	Enthroned	Pentagrammaton	The Vitalized Shell	black metal	2010	240.74	.40	-4.88	135.54
186	Susana Seivane	Os soA+-os que volven	Para Miladoiro	gaita	2010	350.72	.30	-13.04	152.01
201	Pascal Sangla	Une petite pause	Une petite pause	french pop	2010	243.15	.00	-14.74	93.24
466	Sister Hazel	Release	Walls And Cannonballs (	post-grunge	2010	195.37	.51	-6.48	140.26
648	Enthroned	Pentagrammaton	Ad Te Clamamus Exsvles	black metal	2010	89.16	.40	-20.36	90.66
923	Cancer Bats	Sabotage	Sabotage	southern rock	2010	180.71	.53	-4.83	86.01
1325	Boondox	South Of Hell	Nothing To Lose	rap rock	2010	260.10	.46	-5.85	200.00
1369	James Newton Howard	Nanny McPhee & The Big	The Burp Heard 'Round T	instrumental pop	2010	110.52	.45	-9.52	84.97
1428	Jack Johnson	You And Your Heart	You And Your Heart	rock	2010	196.21	.68	-7.05	107.04

Number Field: Year, equal to, 2010

SELECT * FROM music WHERE year = 2010

Activity 4.27 *continued*

Search Number event

```
var
  sSqlQuery : String;
  sField, sSymbol, sValue: String;
Begin
  case cbxField.ItemIndex of
   0: sField := 'year';
   1: sField := 'duration';
   2: sField := 'popularity';
   3: sField := 'loudness';
   4: sField := 'beats_per_minute';
  end;
  case cbxSymbol.ItemIndex of
   0: sSymbol := ' = ';
   1: sSymbol := ' > ';
   2: sSymbol := ' < ';
  end;
  sValue := edtNumber.Text;
  sSqlQuery := 'SELECT * FROM music WHERE '+ sField + sSymbol + sValue;
  lblSqlQuery.Caption := sSqlQuery;
  dmoMusic.qryMusic.SQL.Text := sSqlQuery;
  DmoMusic.qryMusic.active := true;
end;
```

Did you know

The wildcard characters in Delphi are the percentage symbol (%) for zero to many characters, and the underscore symbol (_) for a single character.

Activity 4.28

4.28.1 Provide code for the [Search Song] and [Search Genre] buttons to complete the **Music Searcher App.** Use wild cards in both so that any song or any genre with the word entered can be found.

4.28.2 Provide code for the [Search all] button as follows:

- Create a query that always searches for all fields (using the AND operator).
- Place each of the values from the text boxes into the search query.
- If a textbox does not contain a value, replace this value with a wildcard character that will match all records.

Congratulations, you just created an application that allows users to search through thousands of songs with the click of a button!

Activity 4.29 Using data from components

Open the application saved in your Cities of South Africa folder. Create the following events for this application:

4.29.1 An event to insert new cities to your *cities* table.

4.29.2 An event to update the population of your *provinces* table.

4.29.3 An event to delete all cities meeting specific criteria from your *cities* table.

4.29.4 Using this application, complete the following tasks:

a. Add any city of your choice to the database.

b. Update the population of Gauteng to 14 717 000 (the 2018 estimate).

c. Delete the city with the *ID* of 6800.

CONSOLIDATION ACTIVITY Chapter 4: Databases and SQL

QUESTION 1

1.1 The following database called **students** shows the details of a number of students attending a university, as well as the students' average for their course's subjects. Use this table to answer the questions that follow.

STUDENT_NUMBER	NAME	SURNAME	COURSE	AVERAGE
8621	Anton	Potgieter	Electrical Engineering	62
8893	Kimberly	Burton	Information Technology	74
9672	Ayanda	Bisepe	Psychology	57
7896	Erika	van Vuuren	Electrical Engineering	82
2033	David	Mtsweni	Chemical Engineering	74
8877	Martin	Governor	Veterinary Sciences	92
6696	Blessing	Sisulu	Mathematics	42

Write a SQL query to:

a. View all the information, sorted from the lowest student number to the highest student number.

b. Show a list of courses without duplicates.

c. Show the *name*, *surname* and *average* of all the students that have the character 'a' in either their *name* or *surname*.

1.2 The following database called **Appliances** shows the products sold at an appliance and electronics store. It shows the price for these products in Rands and how many units of the product are stored in the store's warehouse.

PRODUCT	PRICE	TOTAL_UNITS
Kettle	95.95	116
DVD Player	159.00	563
Television	4999.00	250
Fridge	4850.50	189
Washing machine	2655.99	374
Dish washer	2910.00	116
Sound system	1999.90	123

Write a SQL query to:

a. view all the products that have a price above R2000?.

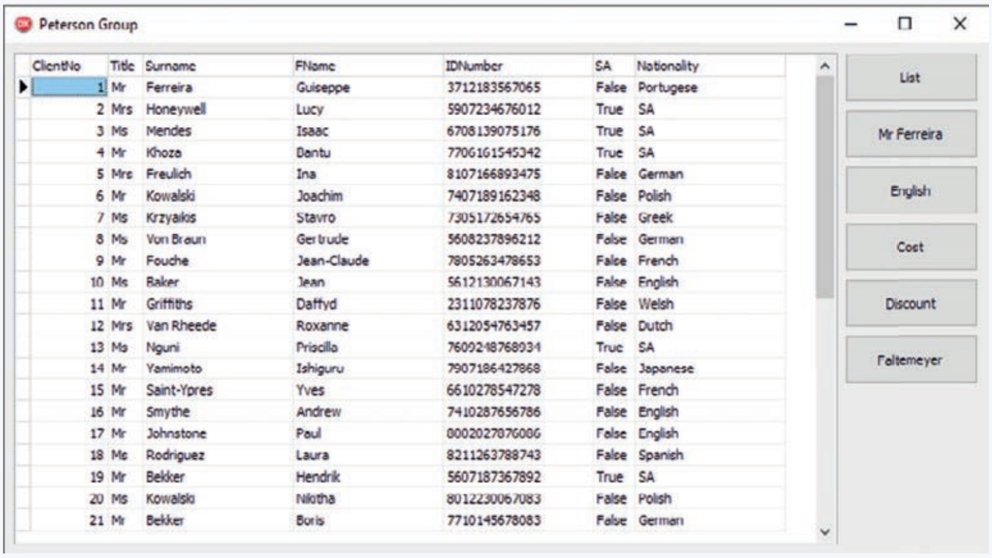
b. show the total value of all the stock.

QUESTION 2

For this application, open the project saved in the 04 – Question 2. Save your project.

The Petersen Group CC wants to use software to assist their staff in answering queries from management. In the development of the software, a database called **BandB.mdb** has been created. The program is incomplete. Your task will be to complete the program that will be used to answer queries from management.

2.1 Open the data module and ensure that all the database components have been connected correctly. You should see the following user interface.



ClientNo	Title	Surname	FName	IDNumber	SA	Nationality
1	Mr	Ferreira	Guiseppe	3712183567065	False	Portugese
2	Mrs	Honeywell	Lucy	5907234676012	True	SA
3	Ms	Mendes	Isaac	6708139075176	True	SA
4	Mr	Khoza	Bantu	7706161545342	True	SA
5	Mrs	Freulich	Ina	8107166893475	False	German
6	Mr	Kowalski	Joachim	7407189162348	False	Polish
7	Ms	Krzyakis	Stavro	7305172654765	False	Greek
8	Ms	Von Braun	Gertrude	5608237896212	False	German
9	Mr	Fouche	Jean-Claude	7805263478653	False	French
10	Ms	Baker	Jean	5612130067143	False	English
11	Mr	Griffiths	Daffyd	2311078237876	False	Welsh
12	Mrs	Van Rheede	Roxanne	6312054763457	False	Dutch
13	Ms	Nguni	Priscilla	7609248768934	True	SA
14	Mr	Yamimoto	Ishiguru	7907186427868	False	Japanese
15	Mr	Saint-Ypres	Yves	6610278547278	False	French
16	Mr	Smythe	Andrew	7410287656786	False	English
17	Mr	Johnstone	Paul	0002027876086	False	English
18	Ms	Rodriguez	Laura	8211263788743	False	Spanish
19	Mr	Bekker	Hendrik	5607187367892	True	SA
20	Ms	Kowalski	Nikitha	8012230067083	False	Polish
21	Mr	Bekker	Boris	7710145678083	False	German

Complete the SQL statements for each button as indicated by questions 2.2 to 2.7 that follow:

2.2 Complete the code in the [List] button by formulating an SQL statement to display all the fields from *tblClients* table sorted by *Surname* and then by *FName* (first name).

2.3 Complete the code in the [Mr Ferreira] button by creating a query that will calculate the total amount owed by Mr Ferreira (*ClientNo* field equal to 1). This is a calculated answer. The name of the calculated field must be *Total Due* and the result must be formatted to display the amount with TWO decimal places.

2.4 All the bookings for the English football fans have been cancelled. Complete the code for the [English] button by creating a query to delete all English clients (*Nationality* field equal to English).

2.5 The group uses a 25% markup when calculating the selling price. Using the *tblOrders* table, complete the code for the [Cost] button by creating a query that will list the *Date*, *Category*, *SellingPrice* and *Cost* (selling price minus the 25% mark-up) for each item ordered by Guiseppe Ferreira (*ClientNo* field equal to 1). Cost is a calculated field and must be named *Cost*.

Use the following formula to calculate the cost: Cost = Selling price × 0.75

2.6 The Petersen Group has decided that they want to support all tourists by giving them a R5 discount on the selling price of every item they have ordered if the item's selling price is R30 or more. Complete the code for the [Discount] button by writing a query that will reduce the selling price of the relevant items by R5 in the *tblOrders* table.

2.7 Complete the code for the [Faltemeyer] button by writing a query that will add the following client data to the *tblClients* table:
Mr Harald Faltemeyer, ID 7407185683074, Swedish
NB: *IDNumber* field only accepts strings, while the *SA* field only accepts boolean values.

QUESTION 3

The city of Coruscant has called for the development of an application to keep track of drivers that violate the city's speed limits.

The **TrafficViolations.mdb** database contains two tables called *tblOwners* and *tblViolations*.

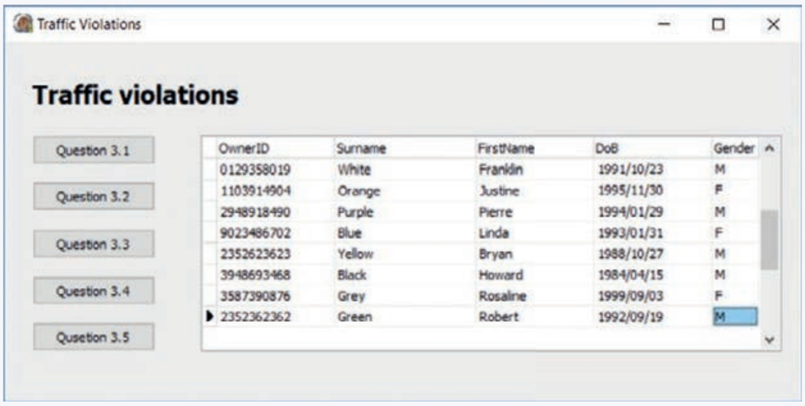
The *tblOwners* table has the following fields:

FIELD	DATA TYPE	DESCRIPTION
OwnerID (PK)	Text	An identification code unique to each owner.
Surname	Text	Contains an owner's surname.
FirstName	Text	Contains an owner's first name.
DoB	Text	Contains an owner's date of birth 'YYYY/MM/DD'.
Gender	Text	Describes an owner's gender as male (M) or female (F).

The *tblViolations* table is structured with the following fields:

FIELD	DATA TYPE	DESCRIPTION
ViolationID (PK)	Auto number	Unique ID for a speeding violation.
Location	Text	Location where a speeding violation was captured.
SpeedArea	Integer	The official speed limit for an area.
SpeedCaptured	Integer	The speed captured for a speeding violation.
ViolationDate	Text	Indicates the date of the speeding violation.
ViolationTime	Text	Indicates the time of the speeding violation.
OwnerID (FK)	Text	Identifies the owner of the speeding violation.

The *OwnerID* field has been used to link the two tables. The dates and times captured in the database are stored as strings.



OwnerID	Surname	FirstName	DoB	Gender
0129358019	White	Franklin	1991/10/23	M
1103914904	Orange	Justine	1995/11/30	F
2948918490	Purple	Pierre	1994/01/29	M
9023486702	Blue	Linda	1993/01/31	F
2352623623	Yellow	Bryan	1988/10/27	M
3948693468	Black	Howard	1984/04/15	M
3587390876	Grey	Rosaline	1999/09/03	F
2352362362	Green	Robert	1992/09/19	M

Complete SQL the code to answer QUESTION 3.1. to QUESTION 3.5.

3.1 All fields must be displayed for all records in the *tblOwners* table, sorted from Z to A on surname.

3.2 Concatenate the following in a field called *Owner* for all records in the *tblOwners* table that were born in or after the year 1993 (also display the date of birth):

- *Surname* followed by a comma and a space.
- The first letter of the owner name followed by a dot (.).

3.3 Using the *Violations* table, display the average speed recorded for each location where speeding violations were captured in a new field called *AverageSpeed*. The averages need to be rounded off to ONE decimal place.

3.4 Use both tables to display the *surname* of each owner and the number of traffic violations belonging to each owner in a field called *ViolationCount*.

3.5 Two records in the *tblOwners* table have been captured by error and need to be removed from the database. Remove the records identified by the keys (*OwnerID*) 2987593875 and 9867398476 from the *tblOwners* table.

ANNEXURE A

Shneiderman's 'Eight Golden Rules of Interface Design'

These rules were obtained from the text Designing the User Interface by Ben Shneiderman.
Shneiderman proposed this collection of principles that are derived heuristically from experience and applicable in most interactive systems after being properly refined, extended, and interpreted.
To improve the usability of an application it is important to have a well designed interface.
Shneiderman's 'Eight Golden Rules of Interface Design' are a guide to good interaction design.

1. STRIVE FOR CONSISTENCY.

Consistent sequences of actions should be required in similar situations; identical terminology should be used in prompts, menus, and help screens; and consistent commands should be employed throughout.

2. ENABLE FREQUENT USERS TO USE SHORTCUTS.

As the frequency of use increases, so do the user's desires to reduce the number of interactions and to increase the pace of interaction. Abbreviations, function keys, hidden commands, and macro facilities are very helpful to an expert user.

3. OFFER INFORMATIVE FEEDBACK.

For every operator action, there should be some system feedback. For frequent and minor actions, the response can be modest, while for infrequent and major actions, the response should be more substantial.

4. DESIGN DIALOG TO YIELD CLOSURE.

Sequences of actions should be organized into groups with a beginning, middle, and end. The informative feedback at the completion of a group of actions gives the operators the satisfaction of accomplishment, a sense of relief, the signal to drop contingency plans and options from their minds, and an indication that the way is clear to prepare for the next group of actions.

5. OFFER SIMPLE ERROR HANDLING.

As much as possible, design the system so the user cannot make a serious error. If an error is made, the system should be able to detect the error and offer simple, comprehensible mechanisms for handling the error.

6. PERMIT EASY REVERSAL OF ACTIONS.

This feature relieves anxiety, since the user knows that errors can be undone; it thus encourages exploration of unfamiliar options. The units of reversibility may be a single action, a data entry, or a complete group of actions.

7. SUPPORT INTERNAL LOCUS OF CONTROL.

Experienced operators strongly desire the sense that they are in charge of the system and that the system responds to their actions. Design the system to make users the initiators of actions rather than the responders.

8. REDUCE SHORTTERM MEMORY LOAD.

The limitation of human information processing in shortterm memory requires that displays be kept simple, multiple page displays be consolidated, windowmotion frequency be reduced, and sufficient training time be allotted for codes, mnemonics, and sequences of actions.
[Source: http://www.cs.utexas.edu/users/almstrum/cs370/elvisino/rules.html, accessed 13 June 2019]

ANNEXURE B

10 usability heuristics for user interface design

By Jakob Nielsen on January 1, 1995

Summary: Jakob Nielsen's 10 general principles for interaction design. They are called 'heuristics' because they are broad rules of thumb and not specific usability uidelines.

VISIBILITY OF SYSTEM STATUS

The system should always keep users informed about what is going on, through appropriate feedback within reasonable time.

MATCH BETWEEN SYSTEM AND THE REAL WORLD

The system should speak the users' language, with words, phrases and concepts familiar to the user, rather than system-oriented terms. Follow real-world conventions, making information appear in a natural and logical order.

USER CONTROL AND FREEDOM

Users often choose system functions by mistake and will need a clearly marked 'emergency exit' to leave the unwanted state without having to go through an extended dialogue. Support undo and redo.

CONSISTENCY AND STANDARDS

Users should not have to wonder whether different words, situations, or actions mean the same thing. Follow platform conventions.

ERROR PREVENTION

Even better than good error messages is a careful design which prevents a problem from occurring in the first place. Either eliminate error-prone conditions or check for them and present users with a confirmation option before they commit to the action.

(Read full article on preventing user errors.)

RECOGNITION RATHER THAN RECALL

Minimize the user's memory load by making objects, actions, and options visible. The user should not have to remember information from one part of the dialogue to another. Instructions for use of the system should be visible or easily retrievable whenever appropriate. (Read full article on recognition vs. recall in UX.)

FLEXIBILITY AND EFFICIENCY OF USE

Accelerators — unseen by the novice user — may often speed up the interaction for the expert user such that the system can cater to both inexperienced and experienced users. Allow users to tailor frequent actions.

AESTHETIC AND MINIMALIST DESIGN

Dialogues should not contain information which is irrelevant or rarely needed. Every extra unit of information in a dialogue competes with the relevant units of information and diminishes their relative visibility.

HELP USERS RECOGNIZE, DIAGNOSE, AND RECOVER FROM ERRORS

Error messages should be expressed in plain language (no codes), precisely indicate the problem, and constructively suggest a solution.

HELP AND DOCUMENTATION

Even though it is better if the system can be used without documentation, it may be necessary to provide help and documentation. Any such information should be easy to search, focused on the user's task, list concrete steps to be carried out, and not be too large.

I originally developed the heuristics for heuristic evaluation in collaboration with Rolf Molich in 1990 [Molich and Nielsen 1990; Nielsen and Molich 1990]. I since refined the heuristics based on a factor analysis of 249 usability problems [Nielsen 1994a] to derive a set of heuristics with maximum explanatory power, resulting in this revised set of heuristics [Nielsen 1994b].

REFERENCES

Molich, R., and Nielsen, J. (1990). Improving a human-computer dialogue, Communications of the ACM 33, 3 (March), 338-348.

Nielsen, J., and Molich, R. (1990). Heuristic evaluation of user interfaces, Proc. ACM CHI'90 Conf. (Seattle, WA, 1-5 April), 249-256.

Nielsen, J. (1994a). Enhancing the explanatory power of usability heuristics. Proc. ACM CHI'94 Conf. (Boston, MA, April 24-28), 152-158.

Nielsen, J. (1994b). Heuristic evaluation. In Nielsen, J., and Mack, R.L. (Eds.), Usability Inspection Methods, John Wiley Sons, New York, NY.

Note: Many people ask if they can use these heuristics in their own work. Yes, but please credit Jakob Nielsen and provide the address for this page [nngroup.com/articles/ten-usability-heuristics] or cite the paper above [Nielsen 1994a]. If you want to print copies of this page or reproduce the content online, however, please see the Copyright link below for details.

Did you know

For more information on interface design, you can visit these websites:

Bruce 'Tog' Tognazzini's list of basic principles for interface design (https://asktog.com/atc/principles-of-interaction-design/). The list is slightly too long for heuristic evaluation but serves as a useful checklist.

Examples of the 10 heuristics in Web applications (http://designingwebinterfaces.com/6-tips-for-a-great-flex-ux-part-5).

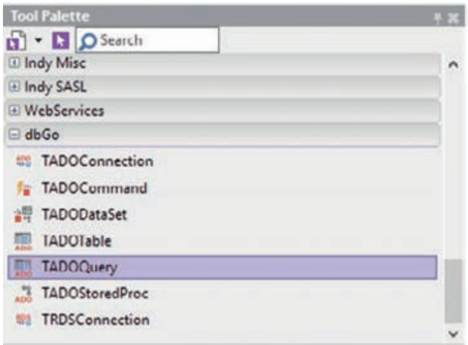
The 10 usability heuristics applied to everyday life (just for fun) (https://www.zenhaiku.com/archives/usability_applied_to_life.html).

Full set of 2,397 usability guidelines (across multiple reports) (https://www.nngroup.com/reports/).

Using SQL in Delphi

So far, you have only used SQL in Microsoft Access. In this Annexure, you will learn how to use SQL in Delphi. To do this, you will create a simple database application with a single textbox that you can use to enter SQL commands. Once an SQL command is entered and the *Filter* button is pressed, the grid will show the filtered database.

To create this application, you will need to use the *TADOQuery* component which can be found from the *dbGo* list in the *Tool Palette*. This component, together with the standard *TADOConnection* component will be added to a data module, from where they can be used in your application.



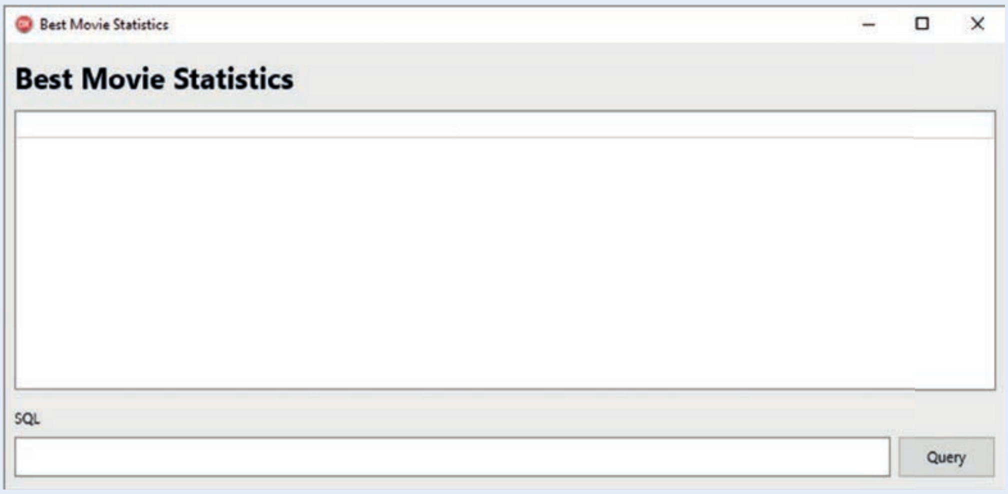
TADOQuery component in a dbGo list

Once the database connection has been set up and the *TADOQuery* component has been added to your data module, you can use the query component to create and SQL query. This is done by adding a string with the query to the *SQL.Text* property, as will be shown in the examples below.

Example C Creating the SQL connection in Delphi

To create the SQL connection in Delphi:

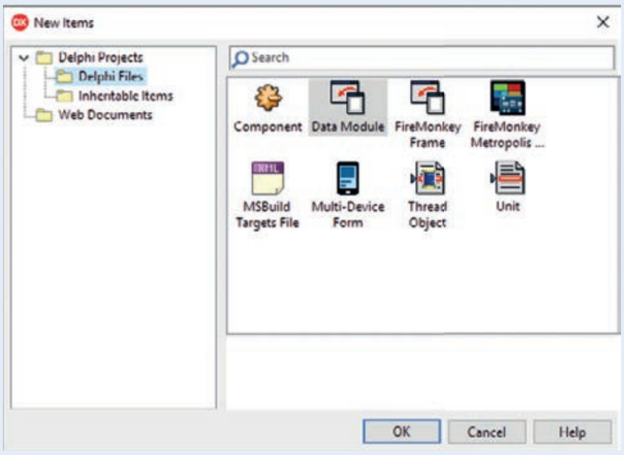
1. Create a new project and save it in the folder Best Movie Statistics.
2. Create the following user interface.



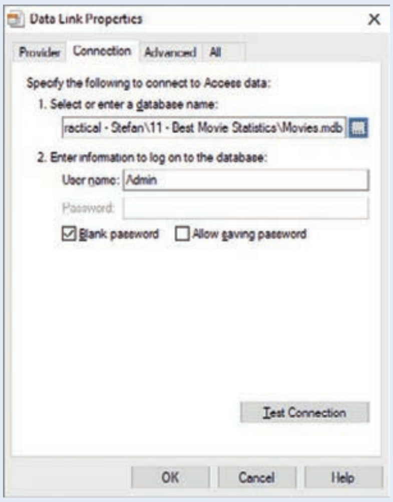
3. Right click on the executable project file in the *Project Manager*, select *Add New* and click on the *Other* option. This will open the *New Items* window.

Creating the SQL connection in Delphi *continued*

4. Select *Data Module* and click *OK*.



5. Save the data module as *movies_d* in the project folder.
6. Change the data module's name to *dmoMovies*.
7. Add a *TADOConnection* component, a *TADOQuery* component and a *TDataSource* component to the data module.
8. Rename the components to *conMovies*, *qryMovies* and *dsMovies*.
9. Select *conMovies* and click on the three dots button next to its *ConnectionString* property.
10. Click on the *Build* button.
11. Select the *Microsoft Jet 4.0 OLE DB Provider* and click *Next*.



Specify the following to connect to Access data:

1. Select or enter a database name:

ractical - Stefan\11 - Best Movie Statistics\Movies.mdb

2. Enter information to log on to the database:

User name: Admin

Password:

☑ Blank password ☐ Allow saving password

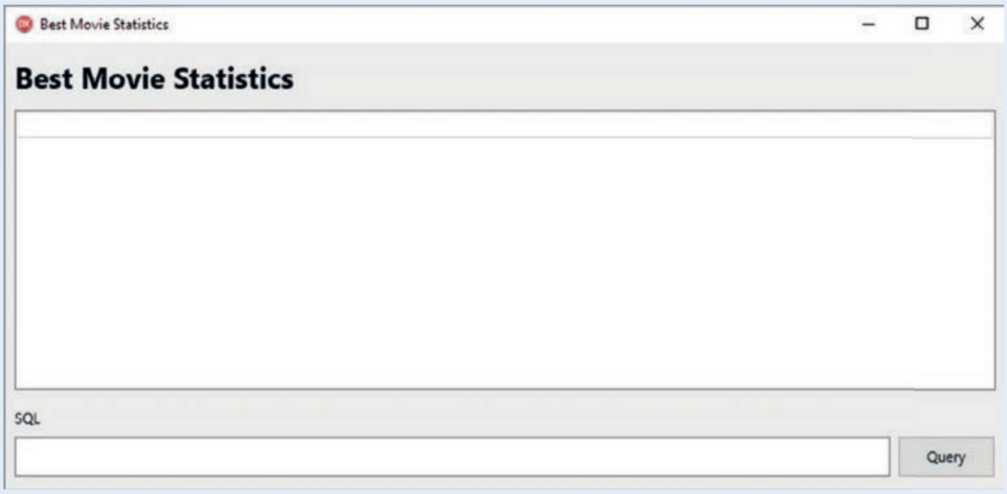
12. Click on the three dots button, select the **Movies.mdb** file and click Open.
13. Remove the 'path' from the connection string to be able to open the application on other computers
14. Test the connection, then click *OK*. Click *OK* again.
15. Change the *LoginPrompt* property of the database connection to False, and the *Connected* property to True.
16. Select *qryMovies* and set its *Connection* property to *conMovies*.
17. Select *dsMovies* and set its *DataSet* property to *qryMovies*. This connects the data source to your movies query.
18. Add *movies_d* to the *uses* list at the top of your main form.
19. Save and test your application. Your application should now open without a problem.

Congratulations on successfully creating the SQL query connection! As this example shows, creating the connection is very similar to using a *TADOTable* component to connect to a database. However, as the next example will show, the query component is more flexible as it allows you to enter different SQL queries.

Example C Using SQL in Delphi

To create the SQL connection in Delphi:

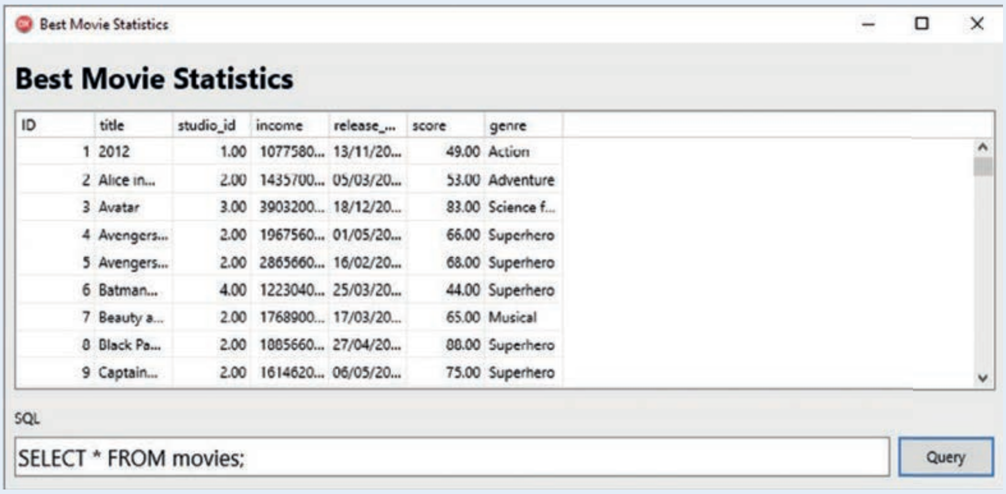
1. Create a new project and save it in the folder Best Movie Statistics.
2. Create the following user interface.



3. Right click on the executable project file in the *Project Manager*, select *Add New* and click on the *Other* option. This will open the *New Items* window.
4. In your event, set the *Active* property of your *qryMovies* component to True.
5. Save and test your application by entering a valid SQL query in the text box and the pressing the [Query] button. You should see that data appear in your grid component.

Did you know

Remember, whenever you access a component from a different form or data module, you always start by first accessing the form or data module.



ID	title	studio_id	income	release_...	score	genre
1	2012	1.00	1077580...	13/11/20...	49.00	Action
2	Alice in...	2.00	1435700...	05/03/20...	53.00	Adventure
3	Avatar	3.00	3903200...	18/12/20...	83.00	Science f...
4	Avengers...	2.00	1967560...	01/05/20...	66.00	Superhero
5	Avengers...	2.00	2865660...	16/02/20...	68.00	Superhero
6	Batman...	4.00	1223040...	25/03/20...	44.00	Superhero
7	Beauty a...	2.00	1768900...	17/03/20...	65.00	Musical
8	Black Pa...	2.00	1885660...	27/04/20...	88.00	Superhero
9	Captain...	2.00	1614620...	06/05/20...	75.00	Superhero

SQL

```
SELECT * FROM movies;
```

Well done, you now have a Delphi application that can create run SQL queries for you.

While this application is very straightforward, it shows how SQL queries can be run in Delphi. A later unit will show a more complex example of a Delphi application using SQL.

ANNEXURE
D

Component names and description

COMPONENT NAME	PREFIX	ICON	BRIEF DESCRIPTION
Standard Group			
Button	btn	OK	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.
List Box	lst		Multiple line display. Able to display left aligned columns.
Radio Button	rad	◉	Toggles selection.
Radio Group	rgp		Grouped Radio buttons – only one selectable.
Combo Box	cmb		Multiple line capturing. Selection of item through drop-down.
Check Box	chk	☒	Toggles selection.
Main Menu	mnu		Main menu with submenus. Activates actions.
Additional Group			
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.
Win32			
Rich Edit	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.
System Group			
Timer	tmr	⏱	* Count down timer – initializing action as count-down reaches 0.

COMPONENT NAME	PREFIX	ICON	BRIEF DESCRIPTION
Samples Group			
Spin Edit	`sed`		Integer input component with pre-set range to select from.
Calendar	`cld`	21	* Calendar with Month layout for selecting days.
Data Access Group			
DataSource	`dsr`		Component to connect data-aware component with dataset.
dbGo Group			
ADOConnection	`con`	ADO	Component to connect with the database (Access).
ADOTable	`tbl`	ADO	Dataset component to reflect the contents of a single table.
ADOQuery	`qry`	ADO	Dataset component to reflect a result.
Data Controls Group			
DBGrid	`dbg`		Data-aware component reflecting the contents of a dataset.
DBNavigator	`dbn`		* Data-aware component interacting with a dataset.
DBText	`dbt`	A	* Data-aware edit box reflecting a field value from a record.
Form	`frm`	▭	The initial Form – not categorised under any group.

ANNEXURE E

The ASCII table

In 1963, the American Standards Association published a table which linked 127 different letters and symbols to numbers. This table was called the **ASCII** table, which is short for the American Standard Code for Information Interchange.

With ASCII, the first 32 characters in the table are programming characters that cannot be shown on the screen. These include characters like a carriage return character (which shows where a new line should start) and a horizontal tab character which added some horizontal space. The full list of these 32 programming characters is given in the table below.

Table 14.1: *The 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.

Table 14.2: *Visible characters*

DECIMAL	CHARACTER
32	SPACE
33	!
34	“
35	#
36	$
37	%
38	&
39	‘
40	(
41	)
42	*
43	+
44	,
45	-
46	.
47	/
48	0
49	1
50	2
51	3
52	4
53	5
54	6
55	7
56	8
57	9
58	:
59	;
60	<
61	=
62	>
63	?
64	@
65	A
66	B
67	C
68	D
69	E
70	F
71	G
72	H
73	I
74	J
75	K
76	L
77	M
78	N
79	O
80	P
81	Q
82	R
83	S
84	T
85	U
86	V
87	W
88	X
89	Y
90	Z
91	[
92	\
93	]
94	^
95	_
96	@
97	a
98	b
99	c
100	d
101	e
102	f
103	g
104	h
105	i
106	j
107	k
108	l
109	m
110	n
111	o
112	p
113	q
114	r
115	s
116	t
117	u
118	v
119	w
120	x
121	y
122	z
123	{
124	|
125	|
126	~

The final 127th character is the DELETE character, which is used when something needs to be removed or deleted.

ANNEXURE F Enrichment

LISTS

Lists work in a similar way to arrays. They allow you to store a large number of elements that can be accessed using an index and which must all be of the same type. However, lists differ from arrays in three important ways:

- The size of a list is not fixed.
- Lists contain useful, built-in methods that allow you to manipulate the list and the data in the list.
- The index of the first list element is always 0.

Lists are created in three steps:

- Add the "Generics.Collections" to your application's "uses" section.
- Define the list in the variables section.
- Create the list in an event.

Once the "Generics.Collections" library has been added to your project, you can declare your variable in the variable section as follows.

```
var
  lName : TList<Type>;
```

Finally, the list needs to be created in an event. This is done with the following code:

```
lName := TList<Type>.Create;
```

While it takes a bit of effort to define the list, once it is done you can take advantage of the many advanced methods that are included in the TList object. These methods include:

FUNCTION	DESCRIPTION
Tlist.Add(Item);	Adds an element to the end of the list.
Tlist.Delete(Index);	Deletes the element at the index location.
TList.Clear;	Deletes all items from the list.
TList.Insert(Index, Item);	Inserts an item at the index location, increasing the index of all items following this location.
Tlist[Index] := Item;	Replaces the value of the element at the index location.
TList.Sort;	Sorts the items in the list in ascending order using an efficient QuickSort algorithm.
TList.IndexOf(Item);	Searches for the item entered and returns its index.

You have already learned about things such as Double, String, Array, and List. But what if you are dealing with a group of data types that form a single entity, that is, different but related data types that need to be grouped together to represent an entity – something similar to an ID card or an account. In this case it would be useful to consider the data as a special type, something that not only contains the data but also the functions that may be applied to it. We refer to this as **encapsulation**.

The *TList* type in Delphi is an example of this type of data. It contains all the essential features to hold items in a list as well as the functions to manage the list. We call the entity *TList* a class, and instances (objects) of this class will each have its own data plus the given functions to manage the data. (See the simplified Tlist API in Appendix A.)

In the code snippet below, constructors are used to create a list object and button object.

Constructor example

```
listNames := TList<String>.Create;
btnDynamic := TButton.Create(Self);
```

In the first line, the *Create* function is used to create a TList object with the type *String* and assign it to the TList object "listNames". In the second line, a TButton object is created and assigned to the object name "btnDynamic". In this example, the constructor function requires the "Self" parameter. We refer to this method of **abstracting** the essential common features of an entity to create a **data type**, as Object Oriented Programming (OOP). In this way, the data structure becomes a class for creating objects (entities) that include both data and functions (code). Programmers can use this to create instances of the entity much like using the *TList* object to create a number of lists with in your program.

In the next example we will demonstrate how useful it is to have a list of objects that can be filtered to get information.

New words

data type – a particular kind of data item, as defined by the value it can take, the programming language used, or the operations that can be performed on it.

abstracting – to create an object and assign it to an object name

Example F.1 Adding data to an array

Alternative: Using *TStringList* to extract data form a delimited string

You can use a *TStringList* to extract the parts between the delimiters from a delimited string. In this example we do this in the inner loop. You can read the data in a loop in exactly the same way as the example above. However sometimes when you have a small file and you may want to use the method below, it puts the complete file into a *TStringList* you can then use a second *TStringList* to remove the delimiters as illustrated.

In this example we do not need to specify the delimiter as it defaults to ‘,’.

```
Var
  sData: String;
  i, j: Integer;
  inputLines: TStringList;
  oneLine: TStringList;
  // aData : Array[1..5, 1..7] of Integer; (Global variable)
begin
  inputLines := TStringList.Create;
  oneLine := TStringList.Create;
  inputLines.LoadFromFile('marks.csv'); //put file into inputLines
  For i := 1 To 5 Do
  Begin
    oneLine.delimitedText := inputList[i]; // default delimiter is ','
    For j := 1 To 7 Do
      aData[i,j] := strtoint(oneLine[j-1]);//array 1-7 stringlist 0-6
  End;

Displaying the array data
CSVIntoArray;
for i := 1 to Length(aData) do
begin
  sOutput := '';
  for j := 1 to Length(aData[1]) do
    sOutput := sOutput + IntToStr(aData[i, j]) + #09;
  lbxResults.Items.Add(sOutput);
end;
```

This code loops through and creates rows of data separated by a tab space character. Once a single row has been saved in the *sOutput* variable, it is added to the listbox before a new row is started.

Take note, since you are looping through the rows of data, you need to place the row variable (“i”) in the outer-loop and the column variable (“j”) in the inner-loop.

Did you know

Many of the James Bond films, including Casino Royale, are set at the Monte Carlo Casino.

For this example, you need to build a Monte Carlo simulator that will analyse the probabilities in Blackjack. Specifically, you want to know how often someone will pick up one of the following combinations of cards:

- An ace and a jack (called Blackjack).
- Two cards with the same number.
- Two cards with a combined value of exactly 21.
- Two cards with a combined value larger than 16 and smaller than 21.
- Two cards with a combined value of exactly 16.
- Two cards with a combined value of under 16.

The value of cards is equal to the number on the card (so a two of hearts has a value of 2), except for the king, queen and jack which have a value of 10, and an ace which has a value of 11.

Example F.2 Monte Carlo simulator

A Monte Carlo simulation is a mathematical tool used to determine how likely certain outcomes are in uncertain situations. Rather than trying to use complex mathematics to calculate an exact probability, mathematicians create a computer simulation with all the relevant variables and then run hundreds of thousands of tests, recording the result of each test. After these tests have been run, the mathematicians can look at the results to see how often each outcome occurred.

The Monte Carlo simulation is named after the suburb Monte Carlo in the city of Monaco, which is famous for having one of the most prestigious casinos in the world. The name Monte Carlo was chosen because Monte Carlo simulations are great at analysing gambling probabilities and odds.

Figure 14.2: *The Monte Carlo Casino in Monaco (photo by Bohyunlee)*

In Blackjack, players with a hand value larger than 16 are likely to win, while players with a hand value less than 16 are likely to lose (unless both cards have the same value).

To create the Monte Carlo simulator, you will need to:

1. Create a user interface.
2. Create a virtual deck of cards.
3. Create a loop that will run the simulations a user-determined number of times.
4. Select two random cards from the deck (without being able to select the same card twice).
5. Calculate the combined value of the cards.
6. Record the category of result (blackjack, smaller than 16, and so forth).
7. Display the results.

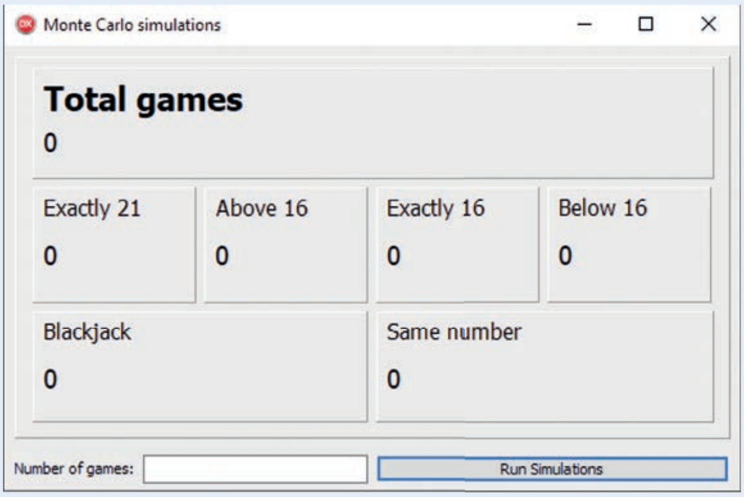
Example F.2 Monte Carlo simulator *continued*

Open the project in the F1 – Monte Carlo simulator folder and add code as described in the solution below:

Solution

To create this program, you needed to complete a number of tasks. For each of these tasks, there is more than one possible solution, so the code below will simply show one way to create the Monte Carlo simulator.

- The first step is to create the user interface.
- The user interface below simply shows the different criteria as well as the number of draws that matched these criteria.



- Create a deck of cards.
- In this solution, a list is used to create a deck of cards since the list functions could be useful to add or delete cards from the list. However, an array can also be used.

Creating the deck of cards

```
lCards := TList<Integer>.Create;
for i := 1 to 52 do
begin
  iCardValue := i mod 13;
  if iCardValue = 0 then
    iCardValue := 13;
  lCards.Add(iCardValue)
end;
```

- The FOR-loop runs 52 times, once for each card in the deck. By calculating the remainder of i mod 13, the loop ensures that all the values are between 1 to 13 (ace to king). These values are then added to the *iCards* list.

INTRODUCTION TO THE MONTE CARLO SIMULATION

https://www.youtube.com/watch?v=pDExObTMHTs

Example F.2 Monte Carlo simulator *continued*

- Create the loop that will run a specific number of times. This loop is shown below.

Game loop

```
iSimulations := StrToInt(edtNumberOfGames.Text);
for j := 1 to iSimulations do
begin
  // Select two random cards
  // Calculate the value of the two cards
  // Categorise the cards based on their values
end;
```

- Once you have your game loop, select two cards inside the loop. The code below shows how these cards are selected.

Selecting two cards

```
iRandom1 := Random(52);
iRandom2 := Random(52);
while iRandom1 = iRandom2 do
  iRandom2 := Random(52);

aSelectedCard[1] := lCards[iRandom1];
aSelectedCard[2] := lCards[iRandom2];
```

- The code starts by selecting two random numbers between 0 and 51. These numbers will be used to draw cards from the list of cards. Since the same card cannot be drawn twice, a WHILE-DO loop will repeat until *iRandom1* and *iRandom2* have different values. Once you have two unique integers, they are used to select two cards from the *iCards* list and assign it to the *aSelectedCard[1]* and *aSelectedCard[2]* array elements.
- The cards selected at this stage have a value of 1 (ace) to 13 (king). However, not all cards have the same value as their number. Specifically, cards with the number 11, 12 and 13 (jack, queen and king) have a value of 10, while cards with the number 1 (ace) have a value of 11.
- The next step is to store the values of the two selected cards.

Selecting two cards

```
iRandom1 := Random(52);
iRandom2 := Random(52);
while iRandom1 = iRandom2 do
  iRandom2 := Random(52);

aSelectedCard[1] := lCards[iRandom1];
aSelectedCard[2] := lCards[iRandom2];
```

- In this code, a FOR-loop is used to assign values to both the selected cards. Inside the FOR-loop, a case statement looks at the selected card's number and assigns an appropriate value to the *aCardValue[1]* and *aCardValue[2]* array elements. Once the values for the two cards has been assigned, they can be added together to obtain the total value.
- At this point, you know the value of both the cards as well as the specific cards that were selected. The last two steps is to assign the cards to a category before displaying the values for the categories in your UI.

Example F.2 Monte Carlo simulator *continued*

Categorising the cards

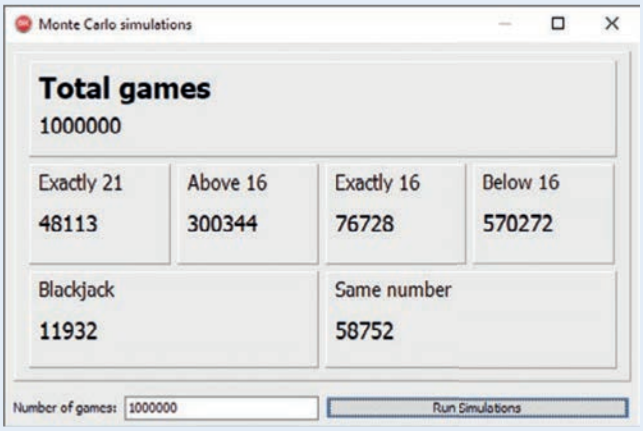
```
Case iTotalValue of
    1..15  : iBelow16 := iBelow16 + 1;
    16     : iExactly16 := iExactly16 + 1;
    17..20 : iAbove16 := iAbove16 + 1;
    21     : iExactly21 := iExactly21 + 1;
End;

// Checks if the cards have the same number
if aSelectedCard[1] = aSelectedCard[2] then
  iSameNumber := iSameNumber + 1;

// Checks if an ace and jack were selected
if ((aSelectedCard[1] = 1) and (aSelectedCard[2] = 11)) or
((aSelectedCard[2] = 1) and (aSelectedCard[1] = 11)) then
  iBlackJack := iBlackJack + 1;
```

- The CASE statement at the start of the code snippet looks at the total card value and increments the category variable based on this value. Once this is done, a separate IF-THEN statement is used to check if the two selected cards have the same number. Finally, an IF-THEN statement is used to check if one of the cards is an ace (1) and the other a jack (11) in order to increment the *iBlackJack* variable. This is the last step that occurs inside the loop.
- Once outside of the loop, all the values are converted to strings and written to the labels.

Writing the values

```
lblTotalGames.Caption := IntToStr(iSimulations);
lblExactly21.Caption := IntToStr(iExactly21);
lblAbove16.Caption := IntToStr(iAbove16);
lblExactly16.Caption := IntToStr(iExactly16);
lblBelow16.Caption := IntToStr(iBelow16);
lblBlackJack.Caption := IntToStr(iBlackJack);
lblSameNumber.Caption := IntToStr(iSameNumber);
```

- By running this application, you will see that your chances of picking up a BlackJack is roughly 1.2% (or 12 000 / 1 000 000)!



Total games
1000000

Exactly 21
48113

Above 16
300344

Exactly 16
76728

Below 16
570272

Blackjack
11932

Same number
58752

Number of games: 1000000

Glossary

A

abstracting to create an object and assign it to an object name

access specifiers is a defining code element that can be determine which elements of a program are allowed to access a specific variable or other piece of data

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 liked to a single variable name

assume supposed to be the case, without proof

attributes the data fields of the class

B

behaviour the code that provides the interaction with the attributes

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

calculated field the field that is calculated each time you run your query

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

data type a particular kind of data item, as defined by the value it can take, the programming language used, or the operations that can be performed on it.

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

encapsulation the grouping of attributes and behaviour in one entity

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

instances of the class a data type that describes the attributes and behaviour of the object to be model electronically

instantiate represent as or by an instant

L

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

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 t 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

Object-Oriented Programming (OOP) refers to a type of computer programming (software design) in which programmers define not only the data type of a data structure, but also the types of operations (functions) that can be applied to the data structure

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

relational database a database structured to recognise relations between stored items of information

reusability is an important OOP principle

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

V

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

Notes