FREE eBook

LEARNING

Embarcadero Delphi

Free unaffiliated eBook created from
Stack Overflow contributors.

#delphi

Table of Contents

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: embarcadero-delphi

It is an unofficial and free Embarcadero Delphi ebook created for educational purposes. All the content is extracted from Stack Overflow Documentation, which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Embarcadero Delphi.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Embarcadero Delphi

Remarks

Delphi is a general-purpose language based on an Object Pascal dialect with its roots coming from Borland Turbo Pascal. It comes with its own IDE designed to support rapid application development (RAD).

It allows cross-platform native (compiled) application development from a single code base. Currently supported platforms are Windows, OSX, iOS and Android.

It comes with two visual frameworks:

- VCL: Visual Component Library specifically designed for Windows development wrapping Windows native controls and support for creating custom ones.
- FMX: FireMonkey cross-platform framework for all supported platforms

Versions

Version	Numeric version	Product name	Release date
1	1.0	Borland Delphi	1995-02-14
2	2.0	Borland Delphi 2	1996-02-10
3	3.0	Borland Delphi 3	1997-08-05
4	4.0	Borland Delphi 4	1998-07-17
5	5.0	Borland Delphi 5	1999-08-10
6	6.0	Borland Delphi 6	2001-05-21
7	7.0	Borland Delphi 7	2002-08-09
8	8.0	Borland Delphi 8 for .NET	2003-12-22
2005	9.0	Borland Delphi 2005	2004-10-12
2006	10.0	Borland Delphi 2006	2005-11-23
2007	11.0	CodeGear Delphi 2007	2007-03-16
2009	12.0	CodeGear Delphi 2009	2008-08-25

Version	Numeric version	Product name	Release date
2010	14.0	Embarcadero RAD Studio 2010	2009-08-15
XE	15.0	Embarcadero RAD Studio XE	2010-08-30
XE2	16.0	Embarcadero RAD Studio XE2	2011-09-02
XE3	17.0	Embarcadero RAD Studio XE3	2012-09-03
XE4	18.0	Embarcadero RAD Studio XE4	2013-04-22
XE5	19.0	Embarcadero RAD Studio XE5	2013-09-11
XE6	20.0	Embarcadero RAD Studio XE6	2014-04-15
XE7	21.0	Embarcadero RAD Studio XE7	2014-09-02
XE8	22.0	Embarcadero RAD Studio XE8	2015-04-07
10 Seattle	23.0	Embarcadero RAD Studio 10 Seattle	2015-08-31
10.1 Berlin	24.0	Embarcadero RAD Studio 10.1 Berlin	2016-04-20
10.2 Tokyo	25.0	Embarcadero RAD Studio 10.2 Tokyo	2017-03-22

Examples

Hello World

This program, saved to a file named *HelloWorld.dpr*, compiles to a console application that prints "Hello World" to the console:

```
program HelloWorld;

{$APPTYPE CONSOLE}

begin
  WriteLn('Hello World');
end.
```

Show 'Hello World' using the VCL

This progam uses VCL, the default UI components library of Delphi, to print "Hello World" into a message box. The VCL wrapps most of the commonly used WinAPI components. This way, they can be used much easier, e.g. without the need to work with Window Handles.

To include a dependency (like `Vcl.Dialogs` in this case), add the `uses` block including a comma-separated list of units ending with an semicolon.

```
program HelloWindows;

uses
  Vcl.Dialogs;

begin
  ShowMessage('Hello Windows');
end.
```

Show 'Hello World' Using WinAPI MessageBox

This program uses the Windows API (WinAPI) to print "Hello World" into a message box.

To include a dependency (like `Windows` in this case), add the uses block including a comma-separated list of units ending with an semicolon.

```
program HelloWorld;

uses
  Windows;

begin
  MessageBox(0, 'Hello World!', 'Hello World!', 0);
end.
```

Cross-platform Hello World using FireMonkey

XE2

```
program CrossPlatformHelloWorld;

uses
  FMX.Dialogs;

{$R *.res}

begin
  ShowMessage('Hello world!');
end.
```

Most of the Delphi supported platforms (Win32/Win64/OSX32/Android32/iOS32/iOS64) also support a console so the `WriteLn` example fits them well.

For the platforms that require a GUI (any iOS device and some Android devices), the above FireMonkey example works well.

Read Getting started with Embarcadero Delphi online:
https://riptutorial.com/delphi/topic/599/getting-started-with-embarcadero-delphi

Chapter 2: Creating easily removable runtime error checks

Introduction

This shows how a runtime error check routine of your own making can be easily incorporated so that it doesn't generate any code overhead when it is turned off.

Examples

Trivial example

```
{$DEFINE MyRuntimeCheck} // Comment out this directive when the check is no-longer required!
                         // You can also put MyRuntimeCheck in the project defines instead.

   function MyRuntimeCheck: Boolean;  {$IFNDEF MyRuntimeCheck} inline;  {$ENDIF}
   begin
      result := TRUE;
      {$IFDEF MyRuntimeCheck}
        // .. the code for your check goes here
      {$ENDIF}
   end;
```

The concept is basically this:

The defined symbol is used to turn on the use of the code. It also stops the code being explicitly in-lined, which means it is easier to put a breakpoint into the check routine.

However, the real beauty of this construction is when you *don't* want the check anymore. By commenting out the ***$DEFINE*** (put '//' in-front of it) you will not only remove the check code, but you will also *switch on* the ***inline*** for the routine and thus remove any overheads from all the places where you invoked the routine! The compiler will remove all traces of your check entirely (assuming that inlining itself is set to "On" or "Auto", of course).

The example above is essentially similar to the concept of "assertions", and your first line could set the result to TRUE or FALSE as appropriate to the usage.

But you are now also free to use this manner of construction for code that does trace-logging, metrics, whatever. For example:

```
   procedure MyTrace(const what: string);  {$IFNDEF MyTrace} inline;  {$ENDIF}
   begin
      {$IFDEF MyTrace}
        // .. the code for your trace-logging goes here
      {$ENDIF}
   end;
...
MyTrace('I was here');   // This code overhead will vanish if 'MyTrace' is not defined.
```

```
MyTrace( SomeString );   // So will this.
```

Read Creating easily removable runtime error checks online:
https://riptutorial.com/delphi/topic/10541/creating-easily-removable-runtime-error-checks

Chapter 3: For Loops

Syntax

- for OrdinalVariable := LowerOrdinalValue to UpperOrdinalValue do begin {loop-body} end;
- for OrdinalVariable := UpperOrdinalValue downto LowerOrdinalValue do begin {loop-body} end;
- for EnumerableVariable in Collection do begin {loop-body} end;

Remarks

- Delphi's `for`-loop syntax does not provide anything to change step amount from `1` to any other value.
- When looping with variable ordinal values, e.g. local variables of type `Integer`, the upper and lower values will be determined only once. Changes to such variables will have no effect on the loops iteration count.

Examples

Simple for loop

A `for` loop iterates from the starting value to the ending value inclusive.

```
program SimpleForLoop;

{$APPTYPE CONSOLE}

var
  i : Integer;
begin
  for i := 1 to 10 do
    WriteLn(i);
end.
```

Output:

1
2
3
4
5
6
7
8
9
10

Looping over characters of a string

2005

The following iterates over the characters of the string `s`. It works similarly for looping over the elements of an array or a set, so long as the type of the loop-control variable (`c`, in this example) matches the element type of the value being iterated.

```
program ForLoopOnString;

{$APPTYPE CONSOLE}

var
  s : string;
  c : Char;
begin
  s := 'Example';
  for c in s do
    WriteLn(c);
end.
```

Output:

E
x
a
m
p
l
e

Reverse-direction for loop

A `for` loop iterates from the starting value down to the ending value inclusive, as a "count-down" example.

```
program CountDown;

{$APPTYPE CONSOLE}

var
  i : Integer;
begin
  for i := 10 downto 0 do
    WriteLn(i);
end.
```

Output:

10
9
8

7
6
5
4
3
2
1
0

For loop using an enumeration

A `for` loop iterate through items in an enumeration

```
program EnumLoop;

uses
  TypInfo;

type
  TWeekdays = (Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday);

var
  wd : TWeekdays;
begin

  for wd in TWeekdays do
    WriteLn(GetEnumName(TypeInfo(TWeekdays), Ord(wd)));

end.
```

Output:

Sunday
Monday
Tuesday
Wednesday
Thursday
Friday
Saturday

For in array

A `for` loop iterate through items in an array

```
program ArrayLoop;
{$APPTYPE CONSOLE}
const a : array[1..3] of real = ( 1.1, 2.2, 3.3 );
var f : real;
begin
  for f in a do
    WriteLn( f );
end.
```

Output:

1,1
2,2
3,3

Read For Loops online: https://riptutorial.com/delphi/topic/4643/for-loops

Chapter 4: Generics

Examples

Sort a dynamic array via generic TArray.Sort

```
uses
  System.Generics.Collections, { TArray }
  System.Generics.Defaults; { TComparer<T> }

var StringArray: TArray<string>; { Also works with "array of string" }

...

{ Sorts the array case insensitive }
TArray.Sort<string>(StringArray, TComparer<string>.Construct(
  function (const A, B: string): Integer
  begin
    Result := string.CompareText(A, B);
  end
));
```

Simple usage of TList

```
var List: TList<Integer>;

...

List := TList<Integer>.Create; { Create List }
try
  List.Add(100); { Add Items }
  List.Add(200);

  WriteLn(List[1]); { 200 }
finally
  List.Free;
end;
```

Descending from TList making it specific

```
type
  TIntegerList = class(TList<Integer>)
  public
    function Sum: Integer;
  end;

...

function TIntegerList.Sum: Integer;
var
  Item: Integer;
begin
  Result := 0;
```

```
  for Item in Self do
     Result := Result + Item;
end;
```

Sort a TList

```
var List: TList<TDateTime>;

...

List.Sort(
  TComparer<TDateTime>.Construct(
    function(const A, B: TDateTime): Integer
    begin
      Result := CompareDateTime(A, B);
    end
  )
);
```

Read Generics online: https://riptutorial.com/delphi/topic/4054/generics

Chapter 5: Interfaces

Remarks

Interfaces are used to describe the needed information and the expected output of methods and classes, without providing information of the explicit implementation.

Classes can **implement** interfaces, and interfaces can **inherit** from each other. If a class is **implementing** an interface, this means all functions and procedures exposed by the interface exist in the class.

A special aspect of interfaces in delphi is that instances of interfaces have a lifetime management based on reference counting. The lifetime of class instances has to be managed manually.

Considering all these aspects, interfaces can be used to achieve different goals:

- Provide multiple different implementations for operations (e.g. saving in a file, database or sending as E-Mail, all as Interface "SaveData")
- Reduce dependencies, improving the decoupling and thus making the code better maintainable and testable
- Work with instances in multiple units without getting troubled by lifetime management (though even here pitfalls exist, beware!)

Examples

Defining and implementing an interface

An interface is declared like a class, but without access modifiers (`public`, `private`, ...). Also, no definitions are allowed, so variables and constants can't be used.

Interfaces should always have an *Unique Identifier*, which can be generated by pressing `Ctrl` + `Shift` + `G`.

```
IRepository = interface
    ['{AFCFCE96-2EC2-4AE4-8E23-D4C4FF6BBD01}']
    function  SaveKeyValuePair(aKey: Integer; aValue: string): Boolean;
end;
```

To implement an interface, the name of the interface must be added behind the base class. Also, the class should be a descendant of `TInterfacedObject` (this is important for the *lifetime management*).

```
TDatabaseRepository = class(TInterfacedObject, IRepository)
    function  SaveKeyValuePair(aKey: Integer; aValue: string): Boolean;
end;
```

When a class implements an interface, it must include all methods and functions declared in the

interface, else it won't compile.

One thing worth noting is that access modifiers don't have any influence, if the caller works with the interface. For example all functions of the interface can be implemented as `strict private` members, but can still be called from another class if an instance of the interface is used.

Implementing multiple interfaces

Classes can implement more than one interface, as opposed to inheriting from more than one class (*Multiple Inheritance*) which isn't possible for Delphi classes. To achieve this, the name of all interfaces must be added comma-separated behind the base class.

Of course, the implementing class must also define the functions declared by each of the interfaces.

```
IInterface1 = interface
    ['{A2437023-7606-4551-8D5A-1709212254AF}']
    procedure Method1();
    function Method2(): Boolean;
end;

IInterface2 = interface
    ['{6C47FF48-3943-4B53-8D5D-537F4A0DEC0D}']
    procedure SetValue(const aValue: TObject);
    function  GetValue(): TObject;

    property Value: TObject read GetValue write SetValue;
end;

TImplementer = class(TInterfacedObject, IInterface1, IInterface2)
    // IInterface1
    procedure Method1();
    function Method2(): Boolean;

    // IInterface2
    procedure SetValue(const aValue: TObject);
    function  GetValue(): TObject

    property Value: TObject read GetValue write SetValue;
end;
```

Inheritance for interfaces

Interfaces can inherit from each other, exactly like classes do, too. An implementing class thus has to implement functions of the interface and all base interfaces. This way, however, the compiler doesn't know that the implenting class also implements the base interface, it only knows of the interfaces that are explicitly listed. That's why using `as ISuperInterface` on `TImplementer` wouldn't work. That also results in the common practice, to explicitly implement all base interfaces, too (in this case `TImplementer = class(TInterfacedObject, IDescendantInterface, ISuperInterface)`).

```
ISuperInterface = interface
    ['{A2437023-7606-4551-8D5A-1709212254AF}']
    procedure Method1();
```

```
    function Method2(): Boolean;
end;

IDescendantInterface = interface(ISuperInterface)
    ['{6C47FF48-3943-4B53-8D5D-537F4A0DEC0D}']
    procedure SetValue(const aValue: TObject);
    function  GetValue(): TObject;

    property Value: TObject read GetValue write SetValue;
end;

TImplementer = class(TInterfacedObject, IDescendantInterface)
    // ISuperInterface
    procedure Method1();
    function Method2(): Boolean;

    // IDescendantInterface
    procedure SetValue(const aValue: TObject);
    function  GetValue(): TObject

    property Value: TObject read GetValue write SetValue;
end;
```

Properties in interfaces

Since the declaration of variables in interfaces isn't possible, the "fast" way of defining properites (`property Value: TObject read FValue write FValue;`) can't be used. Instead, the Getter and setter (each only if needed) have to be declared in the interface, too.

```
IInterface = interface(IInterface)
    ['{6C47FF48-3943-4B53-8D5D-537F4A0DEC0D}']
    procedure SetValue(const aValue: TObject);
    function  GetValue(): TObject;

    property Value: TObject read GetValue write SetValue;
end;
```

One thing worth noting is that the implementing class doesn't have to declare the property. The compiler would accept this code:

```
TImplementer = class(TInterfacedObject, IInterface)
    procedure SetValue(const aValue: TObject);
    function  GetValue(): TObject
end;
```

One caveat, however, is that this way the property can only be accessed through an instance of the interface, noth through the class itself. Also, adding the property to the class increases the readability.

Read Interfaces online: https://riptutorial.com/delphi/topic/4885/interfaces

Chapter 6: Loops

Introduction

Delphi language provide 3 types of loop

`for` - iterator for fixed sequence over integer, string, array or enumeration

`repeat-until` - quit condition is checking after each turn, loop executing at minimum once tmeeven

`while do` - do condition is checking before each turn, loop could be never executed

Syntax

- for OrdinalVariable := LowerOrdinalValue to UpperOrdinalValue do begin {loop-body} end;
- for OrdinalVariable := UpperOrdinalValue downto LowerOrdinalValue do begin {loop-body} end;
- for EnumerableVariable in Collection do begin {loop-body} end;
- repeat {loop-body} until {break-condition};
- while {condition} do begin {loop-body} end;

Examples

Break and Continue in Loops

```
program ForLoopWithContinueAndBreaks;

{$APPTYPE CONSOLE}

var
  var i : integer;
begin
  for i := 1 to 10 do
    begin
      if i = 2 then continue; (* Skip this turn *)
      if i = 8 then break;    (* Break the loop *)
      WriteLn( i );
    end;
  WriteLn('Finish.');
end.
```

Output:

1
3
4
5
6

7
Finish.

Repeat-Until

```
program repeat_test;

{$APPTYPE CONSOLE}

var s : string;
begin
  WriteLn( 'Type a words to echo. Enter an empty string to exit.' );
  repeat
    ReadLn( s );
    WriteLn( s );
  until s = '';
end.
```

This short example print on console `Type a words to echo. Enter an empty string to exit.`, wait for user type, echo it and waiting input again in infinite loop - until user entering the empty string.

While do

```
program WhileEOF;
{$APPTYPE CONSOLE}
uses SysUtils;

const cFileName = 'WhileEOF.dpr';
var F : TextFile;
s : string;
begin
  if FileExists( cFileName )
    then
      begin
        AssignFile( F, cFileName );
        Reset( F );

        while not Eof(F) do
          begin
            ReadLn(F, s);
            WriteLn(s);
          end;

        CloseFile( F );
      end
    else
      WriteLn( 'File ' + cFileName +  ' not found!' );
end.
```

This example print to console the text content of `WhileEOF.dpr` file using `While not(EOF)` condition. If file is empty then `ReadLn-WriteLn` loop is not executed.

Read Loops online: https://riptutorial.com/delphi/topic/9931/loops

Chapter 7: Retrieving updated TDataSet data in a background thread

Remarks

This FireDAC example, and the others I'm planning to submit, will avoid the use of native calls to asynchronously open the dataset.

Examples

FireDAC example

The code sample below shows one way to retrieve records from an MSSql Server in a background thread using FireDAC. Tested for Delphi 10 Seattle

As written:

- The thread retrieves data using its own TFDConnection and TFDQuery and transfers the data to the form's FDQuery in a call to Sychronize().
- The Execute retrieves the data only once. It could be altered to run the query repeatedly in response to a message posted from the VCL thread.

Code:

```
type
  TForm1 = class;

TFDQueryThread = class(TThread)
private
  FConnection: TFDConnection;
  FQuery: TFDQuery;
  FForm: TForm1;
published
  constructor Create(AForm : TForm1);
  destructor Destroy; override;
  procedure Execute; override;
  procedure TransferData;
  property Query : TFDQuery read FQuery;
  property Connection : TFDConnection read FConnection;
  property Form : TForm1 read FForm;
end;

TForm1 = class(TForm)
  FDConnection1: TFDConnection;
  FDQuery1: TFDQuery;
  DataSource1: TDataSource;
  DBGrid1: TDBGrid;
  DBNavigator1: TDBNavigator;
  Button1: TButton;
```

```
  procedure FormDestroy(Sender: TObject);
  procedure Button1Click(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
public
  QueryThread : TFDQueryThread;
end;

var
Form1: TForm1;

implementation

{$R *.dfm}

{ TFDQueryThread }

constructor TFDQueryThread.Create(AForm : TForm1);
begin
  inherited Create(True);
  FreeOnTerminate := False;
  FForm := AForm;
  FConnection := TFDConnection.Create(Nil);
  FConnection.Params.Assign(Form.FDConnection1.Params);
  FConnection.LoginPrompt := False;

  FQuery := TFDQuery.Create(Nil);
  FQuery.Connection := Connection;
  FQuery.SQL.Text := Form.FDQuery1.SQL.Text;
end;

destructor TFDQueryThread.Destroy;
begin
  FQuery.Free;
  FConnection.Free;
  inherited;
end;

procedure TFDQueryThread.Execute;
begin
  Query.Open;
  Synchronize(TransferData);
end;

procedure TFDQueryThread.TransferData;
begin
  Form.FDQuery1.DisableControls;
  try
    if Form.FDQuery1.Active then
      Form.FDQuery1.Close;
    Form.FDQuery1.Data := Query.Data;
  finally
    Form.FDQuery1.EnableControls;
  end;
end;

procedure TForm1.FormDestroy(Sender: TObject);
begin
  QueryThread.Free;
end;
```

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  if not QueryThread.Finished then
    QueryThread.Start
  else
    ShowMessage('Thread already executed!');
end;

procedure TForm1.FormCreate(Sender: TObject);
begin
  FDQuery1.Open;
  QueryThread := TFDQueryThread.Create(Self);
end;

end.
```

Read Retrieving updated TDataSet data in a background thread online:
https://riptutorial.com/delphi/topic/4114/retrieving-updated-tdataset-data-in-a-background-thread

Chapter 8: Running a thread while keeping GUI responsive

Examples

Responsive GUI using threads for background work and PostMessage to report back from the threads

Keeping a GUI responsive while running a lengthy process requires either some very elaborate "callbacks" to allow the GUI to process its message queue, or the use of (background) (worker) threads.

Kicking off any number of threads to do some work usually isn't a problem. The fun starts when you want to make the GUI show intermediate and final results or report on the progress.

Showing anything in the GUI requires interacting with controls and/or the message queue/pump. That should always be done in the context of the main thread. Never in the context of any other thread.

There are many ways to handle this.

This example shows how you can do it using simple threads, allowing the GUI to access the thread instance after it is finished by setting `FreeOnTerminate` to `false`, and reporting when a thread is "done" using `PostMessage`.

Notes on race conditions: References to the worker threads are kept in an array in the form. When a thread is finished, the corresponding reference in the array gets nil-ed.

This is a potential source of race conditions. As is the use of a "Running" boolean to make it easier to determine whether there are still any threads that need to finish.

You will need to decide whether you need to protect theses resource using locks or not.

In this example, as it stands, there is no need. They are only modified in two locations: the `StartThreads` method and the `HandleThreadResults` method. Both methods only ever run in the context of the main thread. As long as you keep it that way and don't start calling these methods from the context of different threads, there is no way for them to produce race conditions.

Thread

```
type
  TWorker = class(TThread)
  private
    FFactor: Double;
    FResult: Double;
    FReportTo: THandle;
```

```
  protected
    procedure Execute; override;
  public
    constructor Create(const aFactor: Double; const aReportTo: THandle);

    property Factor: Double read FFactor;
    property Result: Double read FResult;
  end;
```

The constructor just sets the private members and sets FreeOnTerminate to False. This is essential as it will allow the main thread to query the thread instance for its result.

The execute method does its calculation and then posts a message to the handle it received in its constructor to say its done:

```
procedure TWorker.Execute;
const
  Max = 100000000;var
  i : Integer;
begin
  inherited;

  FResult := FFactor;
  for i := 1 to Max do
    FResult := Sqrt(FResult);

  PostMessage(FReportTo, UM_WORKERDONE, Self.Handle, 0);
end;
```

The use of `PostMessage` is essential in this example. `PostMessage` "just" puts a message on the queue of the main thread's message pump and doesn't wait for it to be handled. It is asynchronous in nature. If you were to use `SendMessage` you'd be coding yourself into a pickle. `SendMessage` puts the message on the queue and waits until it has been processed. In short, it is synchronous.

The declarations for the custom UM_WORKERDONE message are declared as:

```
const
  UM_WORKERDONE = WM_APP + 1;
type
  TUMWorkerDone = packed record
    Msg: Cardinal;
    ThreadHandle: Integer;
    unused: Integer;
    Result: LRESULT;
  end;
```

The `UM_WORKERDONE` const uses `WM_APP` as a starting point for its value to ensure that it doesn't interfere with any values used by Windows or the Delphi VCL (as recommended by MicroSoft).

Form

Any form can be used to start threads. All you need to do is add the following members to it:

```
private
  FRunning: Boolean;
  FThreads: array of record
    Instance: TThread;
    Handle: THandle;
  end;
  procedure StartThreads(const aNumber: Integer);
  procedure HandleThreadResult(var Message: TUMWorkerDone); message UM_WORKERDONE;
```

Oh, and the example code assumes the existence of a `Memo1: TMemo;` in the form's declarations, which it uses for "logging and reporting".

The `FRunning` can be used to prevent the GUI from starting being clicked while the work is going on. `FThreads` is used to hold the instance pointer and the handle of the created threads.

The procedure to start the threads has a pretty straightforward implementation. It starts with a check whether there already is a set of threads being waited on. If so, it just exits. If not, it sets the flag to true and starts the threads providing each with its own handle so they know where to post their "done" message.

```
procedure TForm1.StartThreads(const aNumber: Integer);
var
  i: Integer;
begin
  if FRunning then
    Exit;

  FRunning := True;

  Memo1.Lines.Add(Format('Starting %d worker threads', [aNumber]));
  SetLength(FThreads, aNumber);
  for i := 0 to aNumber - 1 do
  begin
    FThreads[i].Instance := TWorker.Create(pi * (i+1), Self.Handle);
    FThreads[i].Handle := FThreads[i].Instance.Handle;
  end;
end;
```

The thread's handle is also put in the array because that is what we receive in the messages that tell us a thread is done and having it outside the thread's instance makes it slightly easier to access. Having the handle available outside the thread's instance also allows us to use `FreeOnTerminate` set to `True` if we didn't need the instance to get its results (for example if they had been stored in a database). In that case there would of course be no need to keep a reference to the instance.

The fun is in the HandleThreadResult implementation:

```
procedure TForm1.HandleThreadResult(var Message: TUMWorkerDone);
var
  i: Integer;
  ThreadIdx: Integer;
  Thread: TWorker;
  Done: Boolean;
begin
```

```
  // Find thread in array
  ThreadIdx := -1;
  for i := Low(FThreads) to High(FThreads) do
    if FThreads[i].Handle = Cardinal(Message.ThreadHandle) then
    begin
      ThreadIdx := i;
      Break;
    end;

  // Report results and free the thread, nilling its pointer and handle
  // so we can detect when all threads are done.
  if ThreadIdx > -1 then
  begin
    Thread := TWorker(FThreads[i].Instance);
    Memo1.Lines.Add(Format('Thread %d returned %f', [ThreadIdx, Thread.Result]));
    FreeAndNil(FThreads[i].Instance);
    FThreads[i].Handle := nil;
  end;

  // See whether all threads have finished.
  Done := True;
  for i := Low(FThreads) to High(FThreads) do
    if Assigned(FThreads[i].Instance) then
    begin
      Done := False;
      Break;
    end;
  if Done then
  begin
    Memo1.Lines.Add('Work done');
    FRunning := False;
  end;
end;
```

This method first looks up the thread using the handle received in the message. If a match was found, it retrieves and reports the thread's result using the instance (`FreeOnTerminate` was `False`, remember?), and then finishes up: freeing the instance and setting both the instance reference and the handle to nil, indicating this thread is no longer relevant.

Finally it checks to see if any of the threads is still running. If none is found, "all done" is reported and the `FRunning` flag set to `False` so a new batch of work can be started.

Read Running a thread while keeping GUI responsive online:
https://riptutorial.com/delphi/topic/1796/running-a-thread-while-keeping-gui-responsive

Chapter 9: Running other programs

Examples

CreateProcess

Following function encapsulates code for using `CreateProcess` Windows API for launching other programs.

It is configurable and can wait until calling process finishes or return immediately.

Parameters:

- `FileName` - full path to executable
- `Params` - command line parameters or use empty string
- `Folder` - working folder for called program - if empty path will be extracted from `FileName`
- `WaitUntilTerminated` - if true function will wait for process to finish execution
- `WaitUntilIdle` - if true function will call WaitForInputIdle function and wait until the specified process has finished processing its initial input and until there is no user input pending
- `RunMinimized` - if true process will be run minimized
- `ErrorCode` - if function fails this will contain encountered Windows Error Code

```
function ExecuteProcess(const FileName, Params: string; Folder: string; WaitUntilTerminated,
WaitUntilIdle, RunMinimized: boolean;
  var ErrorCode: integer): boolean;
var
  CmdLine: string;
  WorkingDirP: PChar;
  StartupInfo: TStartupInfo;
  ProcessInfo: TProcessInformation;
begin
  Result := true;
  CmdLine := '"' + FileName + '" ' + Params;
  if Folder = '' then Folder := ExcludeTrailingPathDelimiter(ExtractFilePath(FileName));
  ZeroMemory(@StartupInfo, SizeOf(StartupInfo));
  StartupInfo.cb := SizeOf(StartupInfo);
  if RunMinimized then
    begin
      StartupInfo.dwFlags := STARTF_USESHOWWINDOW;
      StartupInfo.wShowWindow := SW_SHOWMINIMIZED;
    end;
  if Folder <> '' then WorkingDirP := PChar(Folder)
  else WorkingDirP := nil;
  if not CreateProcess(nil, PChar(CmdLine), nil, nil, false, 0, nil, WorkingDirP, StartupInfo,
ProcessInfo) then
    begin
      Result := false;
      ErrorCode := GetLastError;
      exit;
    end;
  with ProcessInfo do
    begin
      CloseHandle(hThread);
```

```
      if WaitUntilIdle then WaitForInputIdle(hProcess, INFINITE);
      if WaitUntilTerminated then
        repeat
          Application.ProcessMessages;
        until MsgWaitForMultipleObjects(1, hProcess, false, INFINITE, QS_ALLINPUT) <>
WAIT_OBJECT_0 + 1;
      CloseHandle(hProcess);
    end;
end;
```

Usage of above function

```
var
  FileName, Parameters, WorkingFolder: string;
  Error: integer;
  OK: boolean;
begin
  FileName := 'C:\FullPath\myapp.exe';
  WorkingFolder := ''; // if empty function will extract path from FileName
  Parameters := '-p'; // can be empty
  OK := ExecuteProcess(FileName, Parameters, WorkingFolder, false, false, false, Error);
  if not OK then ShowMessage('Error: ' + IntToStr(Error));
end;
```

CreateProcess documentation

Read Running other programs online: https://riptutorial.com/delphi/topic/5180/running-other-programs

Chapter 10: Strings

Examples

String types

Delphi has the following string types (in order of popularity):

Type	Maximum length	Minimum size	Description
`string`	2GB	16 bytes	A managed string. An alias for `AnsiString` through Delphi 2007, and an alias for `UnicodeString` as of Delphi 2009.
`UnicodeString`	2GB	16 bytes	A managed string in UTF-16 format.
`AnsiString`	2GB	16 bytes	A managed string in pre-Unicode ANSI format. As of Delphi 2009, it carries an explicit code-page indicator.
`UTF8String`	2GB	16 bytes	A managed string in UTF-8 format, implemented as an `AnsiString` with a UTF-8 code page.
`ShortString`	255 chars	2 bytes	A legacy, fixed-length, unmanaged string with very little overhead
`WideString`	2GB	4 bytes	Intended for COM interop, a managed string in UTF-16 format. Equivalent to the Windows `BSTR` type.

`UnicodeString` and `AnsiString` are reference counted and copy-on-write (COW).
`ShortString` and `WideString` are not reference counted and do not have COW semantics.

Strings

```
uses
  System.Character;

var
  S1, S2: string;
begin
  S1 := 'Foo';
  S2 := ToLower(S1); // Convert the string to lower-case
  S1 := ToUpper(S2); // Convert the string to upper-case
```

Chars

2009

```
uses
  Character;

var
  C1, C2: Char;
begin
  C1 := 'F';
  C2 := ToLower(C1); // Convert the char to lower-case
  C1 := ToUpper(C2); // Convert the char to upper-case
```

The `uses` clause should be `System.Character` if version is XE2 or above.

UPPER and lower case

```
uses
  SysUtils;

var
  S1, S2: string;
begin
  S1 := 'Foo';
  S2 := LowerCase(S1); // S2 := 'foo';
  S1 := UpperCase(S2); // S1 := 'FOO';
```

Assignment

Assigning string to different string types and how the runtime environment behaves regarding them. Memory allocation, reference counting, indexed access to chars and compiler errors described briefly where applicable.

```
var
  SS5: string[5]; {a shortstring of 5 chars + 1 length byte, no trailing `0`}
  WS: Widestring; {managed pointer, with a bit of compiler support}
  AS: ansistring; {ansistring with the default codepage of the system}
  US: unicodestring; {default string type}
  U8: UTF8string;//same as AnsiString(65001)
  A1251: ansistring(1251); {ansistring with codepage 1251: Cryllic set}
  RB: RawbyteString; {ansistring with codepage 0: no conversion set}
begin
  SS5:= 'test'; {S[0] = Length(SS254) = 4, S[1] = 't'...S[5] = undefined}
  SS5:= 'test1'; {S[0] = 5, S[5] = '1', S[6] is out of bounds}
  SS5:= 'test12'; {compile time error}
  WS:= 'test'; {WS now points to a constant unicodestring hard compiled into the data segment}
  US:= 'test'+IntToStr(1); {New unicode string is created with reference count = 1}
  WS:= US; {SysAllocateStr with datacopied to dest, US refcount = 1 !}
  AS:= US; {the UTF16 in US is converted to "extended" ascii taking into account the codepage
in AS possibly losing data in the process}
  U8:= US; {safe copy of US to U8, all data is converted from UTF16 into UTF8}
  RB:= US; {RB = 'test1'#0 i.e. conversion into RawByteString uses system default codepage}
  A1251:= RB; {no conversion takes place, only reference copied. Ref count incremented }
```

Reference counting

Counting references on strings is thread-safe. Locks and exception handlers are used to safeguard the process. Consider the following code, with comments indicating where the compiler inserts code at compile time to manage reference counts:

```
procedure PassWithNoModifier(S: string);
// prologue: Increase reference count of S (if non-negative),
//           and enter a try-finally block
begin
  // Create a new string to hold the contents of S and 'X'. Assign the new string to S,
  // thereby reducing the reference count of the string S originally pointed to and
  // brining the reference count of the new string to 1.
  // The string that S originally referred to is not modified.
  S := S + 'X';
end;
// epilogue: Enter the `finally` section and decrease the reference count of S, which is
//           now the new string. That count will be zero, so the new string will be freed.

procedure PassWithConst(const S: string);
var
  TempStr: string;
// prologue: Clear TempStr and enter a try-finally block. No modification of the reference
//           count of string referred to by S.
begin
  // Compile-time error: S is const.
  S := S + 'X';
  // Create a new string to hold the contents of S and 'X'. TempStr gets a reference count
  // of 1, and reference count of S remains unchanged.
  TempStr := S + 'X';
end;
// epilogue: Enter the `finally` section and decrease the reference count of TempStr,
//           freeing TempStr because its reference count will be zero.
```

As shown above, introducing temporary local string to hold the modifications to a parameter involves the same overhead as making modifications directly to that parameter. Declaring a string `const` only avoids reference counting when the string parameter is truly read-only. However, to avoid leaking implementation details outside a function, it is advisable to always use one of `const`, `var`, or `out` on string parameter.

Encodings

String types like UnicodeString, AnsiString, WideString and UTF8String are stored in a memory using their respective encoding (see String Types for more details). Assigning one type of string into another may result in a conversion. Type string is designed to be encoding independent - you should never use its internal representation.

The class `Sysutils.TEncoding` provides method `GetBytes` for converting `string` to `TBytes` (array of bytes) and `GetString` for converting `TBytes` to `string`. The class `Sysutils.TEncoding` also provides many predefined encodings as class properties.

One way how to deal with encodings is to use only `string` type in your application and use `TEncoding` every time you need to use specific encoding - typically in I/O operations, DLL calls, etc...

```
procedure EncodingExample;
var hello,response:string;
    dataout,datain:TBytes;
    expectedLength:integer;
    stringStream:TStringStream;
    stringList:TStringList;

begin
  hello := 'Hello World!Привет мир!';
  dataout := SysUtils.TEncoding.UTF8.GetBytes(hello); //Conversion to UTF8
  datain := SomeIOFunction(dataout); //This function expects input as TBytes in UTF8 and
returns output as UTF8 encoded TBytes.
  response := SysUtils.TEncoding.UTF8.GetString(datain); //Convertsion from UTF8

  //In case you need to send text via pointer and length using specific encoding (used mostly
for DLL calls)
  dataout := SysUtils.TEncoding.GetEncoding('ISO-8859-2').GetBytes(hello); //Conversion to ISO
8859-2
  DLLCall(addr(dataout[0]),length(dataout));
  //The same is for cases when you get text via pointer and length
  expectedLength := DLLCallToGetDataLength();
  setLength(datain,expectedLength);
  DLLCall(addr(datain[0]),length(datain));
  response := Sysutils.TEncoding.GetEncoding(1250).getString(datain);

   //TStringStream and TStringList can use encoding for I/O operations
   stringList:TStringList.create;
   stringList.text := hello;
   stringList.saveToFile('file.txt',SysUtils.TEncoding.Unicode);
   stringList.destroy;
   stringStream := TStringStream(hello,SysUtils.TEncoding.Unicode);
   stringStream.saveToFile('file2.txt');
   stringStream.Destroy;
end;
```

Read Strings online: https://riptutorial.com/delphi/topic/3957/strings

Chapter 11: Time intervals measurement

Examples

Using Windows API GetTickCount

The Windows API `GetTickCount` function returns the number of milliseconds since the system (computer) was started. The simplest example follows:

```
var
  Start, Stop, ElapsedMilliseconds: cardinal;
begin
  Start := GetTickCount;
  // do something that requires measurement
  Stop := GetTickCount;
  ElapsedMillseconds := Stop - Start;
end;
```

Note that `GetTickCount` returns 32-bit `DWORD` so it wraps every 49.7 days. To avoid wrapping, you may either use `GetTickCount64` (available since Windows Vista) or special routines to calculate tick difference:

```
function TickDiff(StartTick, EndTick: DWORD): DWORD;
begin
  if EndTick >= StartTick
    then Result := EndTick - StartTick
    else Result := High(NativeUInt) - StartTick + EndTick;
end;

function TicksSince(Tick: DWORD): DWORD;
begin
  Result := TickDiff(Tick, GetTickCount);
end;
```

Anyway these routines will return incorrect results if the interval of two subsequent calls of `GetTickCount` exceeds the 49.7 day boundary.

To convert milliseconds to seconds example:

```
var
  Start, Stop, ElapsedMilliseconds: cardinal;
begin
  Start := GetTickCount;
  sleep(4000); // sleep for 4 seconds
  Stop := GetTickCount;
  ElapsedMillseconds := Stop - Start;
  ShowMessage('Total Seconds: '
      +IntToStr(round(ElapsedMilliseconds/SysUtils.MSecsPerSec))); // 4 seconds
end;
```

Using TStopwatch record

Recent versions of Delphi ships with the TStopwatch record which is for time interval measurement. Example usage:

```
uses
  System.Diagnostics;

var
  StopWatch: TStopwatch;
  ElapsedMillseconds: Int64;
begin
  StopWatch := TStopwatch.StartNew;
  // do something that requires measurement
  ElapsedMillseconds := StopWatch.ElapsedMilliseconds;
end;
```

Read Time intervals measurement online: https://riptutorial.com/delphi/topic/2425/time-intervals-measurement

Chapter 12: TStringList class

Examples

Introduction

TStringList is a descendant of the TStrings class of the VCL. TStringList can be used for storing and manipulating of list of Strings. Although originally intended for Strings, any type of objects can also be manipulated using this class.

TStringList is widely used in VCL when the the purpose is there for maintaining a list of Strings. TStringList supports a rich set of methods which offer high level of customization and ease of manipulation.

The following example demonstrates the creation, adding of strings, sorting, retrieving and freeing of a TStringList object.

```
procedure StringListDemo;
var
   MyStringList: TStringList;
   i: Integer;

Begin

   //Create the object
   MyStringList := TStringList.Create();
   try
      //Add items
      MyStringList.Add('Zebra');
      MyStringList.Add('Elephant');
      MyStringList.Add('Tiger');

      //Sort in the ascending order
      MyStringList.Sort;

      //Output
      for i:=0 to MyStringList.Count - 1 do
        WriteLn(MyStringList[i]);
   finally
      //Destroy the object
      MyStringList.Free;
   end;
end;
```

TStringList has a variety of user cases including string manipulation, sorting, indexing, key-value pairing and delimiter separation among them.

Key-Value Pairing

You can use a TStringList to store Key-Value pairs. This can be useful if you want to store settings, for example. A settings consists of a Key (The Identifier of the setting) and the value.

Each Key-Value pair is stored in one line of the StringList in Key=Value format.

```
procedure Demo(const FileName: string = '');
var
   SL: TStringList;
   i: Integer;
begin
     SL:= TStringList.Create;
     try
        //Adding a Key-Value pair can be done this way
        SL.Values['FirstName']:= 'John';   //Key is 'FirstName', Value is 'John'
        SL.Values['LastName']:= 'Doe';   //Key is 'LastName', Value is 'Doe'

        //or this way
        SL.Add('City=Berlin');  //Key ist 'City', Value is 'Berlin'

        //you can get the key of a given Index
        IF SL.Names[0] = 'FirstName' THEN
         begin
              //and change the key at an index
              SL.Names[0]:= '1stName';  //Key is now "1stName", Value remains "John"
         end;

        //you can get the value of a key
        s:= SL.Values['City']; //s now is set to 'Berlin'

        //and overwrite a value
        SL.Values['City']:= 'New York';

        //if desired, it can be saved to an file
        IF (FileName <> '') THEN
         begin
              SL.SaveToFile(FileName);
         end;
     finally
        SL.Free;
     end;
end;
```

In this example, the Stringlist has the following content before it is destroyed:

```
1stName=John
LastName=Doe
City=New York
```

Note on performance

Under the hood `TStringList` performs key search by straight looping through all items, searching for separator inside every item and comparing the name part against the given key. No need to say it does huge impact on performance so this mechanism should only be used in non-critical, rarely repeated places. In cases where performance matters, one should use `TDictionary<TKey,TValue>` from `System.Generics.Collections` that implements hash table search or to keep keys in **sorted** `TStringList` with values stored as `Object`-s thus utilizing binary find algorithm.

Read TStringList class online: https://riptutorial.com/delphi/topic/6045/tstringlist-class

Chapter 13: Use of try, except, and finally

Syntax

1. Try-except: try [statements] except [[[on E:ExceptionType do statement]] [else statement] | [statements] end;

   Try-finally: try [statements] finally [statements] end;

Examples

Simple try..finally example to avoid memory leaks

Use `try-finally` to avoid leaking resources (such as memory) in case an exception occurs during execution.

The procedure below saves a string in a file and prevents the `TStringList` from leaking.

```
procedure SaveStringToFile(const aFilename: TFilename; const aString: string);
var
  SL: TStringList;
begin
  SL := TStringList.Create; // call outside the try
  try
    SL.Text := aString;
    SL.SaveToFile(aFilename);
  finally
    SL.Free // will be called no matter what happens above
  end;
end;
```

Regardless of whether an exception occurs while saving the file, `SL` will be freed. Any exception will go to the caller.

Exception-safe return of a new object

When a function *returns* an object (as opposed to *using* one that's passed in by the caller), be careful an exception doesn't cause the object to leak.

```
function MakeStrings: TStrings;
begin
  // Create a new object before entering the try-block.
  Result := TStringList.Create;
  try
    // Execute code that uses the new object and prepares it for the caller.
    Result.Add('One');
    MightThrow;
  except
    // If execution reaches this point, then an exception has occurred. We cannot
    // know how to handle all possible exceptions, so we merely clean up the resources
```

```
    // allocated by this function and then re-raise the exception so the caller can
    // choose what to do with it.
    Result.Free;
    raise;
  end;
  // If execution reaches this point, then no exception has occurred, so the
  // function will return Result normally.
end;
```

Naive programmers might attempt to catch all exception types and return `nil` from such a function, but that's just a special case of the general discouraged practice of catching all exception types without handling them.

Try-finally nested inside try-except

A `try`-`finally` block may be nested inside a `try`-`except` block.

```
try
  AcquireResources;
  try
    UseResource;
  finally
    ReleaseResource;
  end;
except
  on E: EResourceUsageError do begin
    HandleResourceErrors;
  end;
end;
```

If an exception occurs inside `UseResource`, then execution will jump to `ReleaseResource`. If the exception is an `EResourceUsageError`, then execution will jump to the exception handler and call `HandleResourceErrors`. Exceptions of any other type will skip the exception handler above and bubble up to the next `try`-`except` block up the call stack.

Exceptions in `AcquireResource` or `ReleaseResource` will cause execution to go to the exception handler, skipping the `finally` block, either because the corresponding `try` block has not been entered yet or because the `finally` block has *already* been entered.

Try-except nested inside try-finally

A `try`-`except` block may be nested inside a `try`-`finally` block.

```
AcquireResource;
try
  UseResource1;
  try
    UseResource2;
  except
    on E: EResourceUsageError do begin
      HandleResourceErrors;
    end;
  end;
  UseResource3;
```

```
finally
  ReleaseResource;
end;
```

If an `EResourceUsageError` occurs in `UseResource2`, then execution will jump to the exception handler and call `HandleResourceError`. The exception will be considered handled, so execution will continue to `UseResource3`, and then `ReleaseResource`.

If an exception of any other type occurs in `UseResource2`, then the exception handler show here will not apply, so execution will jump over the `UseResource3` call and go directly to the `finally` block, where `ReleaseResource` will be called. After that, execution will jump to the next applicable exception handler as the exception bubbles up the call stack.

If an exception occurs in any other call in the above example, then `HandleResourceErrors` will *not* be called. This is because none of the other calls occur inside the `try` block corresponding to that exception handler.

Try-finally with 2 or more objects

```
Object1 := nil;
Object2 := nil;
try
  Object1 := TMyObject.Create;
  Object2 := TMyObject.Create;
finally
  Object1.Free;
  Object2.Free;
end;
```

If you do not initialize the objects with `nil` outside the `try-finally` block, if one of them fails to be created an AV will occur on the finally block, because the object won't be nil (as it wasn't initialized) and will cause an exception.

The `Free` method checks if the object is nil, so initializing both objects with `nil` avoids errors when freeing them if they weren't created.

Read Use of try, except, and finally online: https://riptutorial.com/delphi/topic/3055/use-of-try--except--and-finally

Chapter 14: Using Animations in Firemonkey

Examples

Rotating TRectangle

1. Create blank Multi-Device (Firemonkey) application.
2. Drop Rectangle on Form.
3. In Object inspector window (F11) find RotationAngle click on drop down button, and select "Create New TFloatAnimation".
4. Object inspector window is automatically switched to a newly added TFloatAnimation, you can also view it in Structure menu (Shift + Alt
    - F11).
5. In Object inspector of TFloatAnimation fill duration with any number (in seconds). In our case lets take 1. Leave StartValue property as it is, and in StopValue type - 360 (Degrees, so it all goes round). Also lets turn Loop option on (this loops animation until you stop it from code).

Now we have our animation set up. All is left is to turn it on: Drop two buttons on form, call first one "Start", second one - "Stop". in OnClick event of first button write:

```
FloatAnimation1.Start;
```

OnClick of second button code:

```
FloatAnimation1.Stop;
```

If you changed name of your TFloatAnimation - Also change it when calling Start and Stop.

Now run your project, click Start button and enjoy.

Read Using Animations in Firemonkey online: https://riptutorial.com/delphi/topic/5383/using-animations-in-firemonkey

Chapter 15: Using RTTI in Delphi

Introduction

Delphi provided Runtime Type Information (RTTI) more than a decade ago. Yet even today many developers aren't fully aware of its risks and benefits.

In short, Runtime Type Information is information about an object's data type that is set into memory at run-time.

RTTI provides a way to determine if an object's type is that of a particular class or one of its descendants.

Remarks

RTTI IN DELPHI - EXPLAINED

The Run-Time Type Information In Delphi - Can It Do Anything For You? article by Brian Long provides a great introduction to the RTTI capabilities of Delphi. Brian explains that the RTTI support in Delphi has been added first and foremost to allow the design-time environment to do its job, but that developers can also take advantage of it to achieve certain code simplifications. This article also provides a great overview of the RTTI classes along with a few examples.

Examples include: Reading and writing arbitrary properties, common properties with no common ancestor, copying properties from one component to another, etc.

Examples

Basic Class Information

This example shows how to obtain the ancestry of a component using the `ClassType` and `ClassParent` properties. It uses a button `Button1: TButton` and a list box `ListBox1: TListBox` on a form `TForm1`.

When the user clicks the button, the name of the button's class and the names of its parent classes are added to the list box.

```
procedure TForm1.Button1Click(Sender: TObject) ;
var
  ClassRef: TClass;
begin
   ListBox1.Clear;
   ClassRef := Sender.ClassType;
   while ClassRef <> nil do
   begin
     ListBox1.Items.Add(ClassRef.ClassName) ;
     ClassRef := ClassRef.ClassParent;
```

```
    end;
end;
```

The list box contains the following strings after the user clicks the button:

- TButton
- TButtonControl
- TWinControl
- TControl
- TComponent
- TPersistent
- TObject

Read Using RTTI in Delphi online: https://riptutorial.com/delphi/topic/9578/using-rtti-in-delphi

Credits

S. No	Chapters	Contributors
1	Getting started with Embarcadero Delphi	Charlie H, Community, Dalija Prasnikar, Florian Koch, Jeroen Wiert Pluimers, René Hoffmann, RepeatUntil, Rob Kennedy, Vadim Shakun, w5m, Y.N, Zam
2	Creating easily removable runtime error checks	Alex T
3	For Loops	Filipe Martins, Jeroen Wiert Pluimers, John Easley, René Hoffmann, Rob Kennedy, Siendor, Y.N
4	Generics	Rob Kennedy, Steffen Binas, Uli Gerhardt
5	Interfaces	Florian Koch, Willo van der Merwe
6	Loops	Y.N
7	Retrieving updated TDataSet data in a background thread	MartynA
8	Running a thread while keeping GUI responsive	Fr0sT, Jerry Dodge, Johan, kami, LU RD, Marjan Venema
9	Running other programs	Dalija Prasnikar
10	Strings	AlekXL, Dalija Prasnikar, EMBarbosa, Fabricio Araujo, Johan, Radek Hladík, René Hoffmann, RepeatUntil, Rob Kennedy, Rudy Velthuis
11	Time intervals measurement	Fr0sT, John Easley, kludg, Rob Kennedy, Victoria, Wolf
12	TStringList class	Charlie H, Fabricio Araujo, Fr0sT, KaiW
13	Use of try, except, and finally	EMBarbosa, Fabio Gomes, Johan, MrE, Nick Hodges, Rob Kennedy, Shadow
14	Using Animations in Firemonkey	Alexander Petrosyan

15	Using RTTI in Delphi	Petzy, René Hoffmann