Бесплатная электронная книга

УЧУСЬ

Embarcadero Delphi

Free unaffiliated eBook created from
Stack Overflow contributors.

#delphi

Около

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

глава 1: Начало работы с Embarcadero Delphi

замечания

Delphi - это язык общего назначения, основанный на диалекте Object Pascal, корнем которого является Borland Turbo Pascal. Он поставляется с собственной IDE, разработанной для поддержки быстрой разработки приложений (RAD).

Он позволяет создавать кросс-платформенную разработку (скомпилированную) приложений из одной базы кода. В настоящее время поддерживаются платформы Windows, OSX, iOS и Android.

Он поставляется с двумя визуальными структурами:

- VCL: Visual Component Library, специально разработанная для разработки Windows, обертывает собственные средства управления Windows и поддерживает создание пользовательских.
- FMX: межплатформенная платформа FireMonkey для всех поддерживаемых платформ

Версии

Версия	Цифровая версия	Наименование товара	Дата выхода
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 для .NET	2003-12-22
2005	9,0	Borland Delphi 2005	2004-10-12

Версия	Цифровая версия	Наименование товара	Дата выхода
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
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
X E8	22,0	Embarcadero RAD Studio XE8	2015-04-07
10 Сиэтл	23,0	Embarcadero RAD Studio 10 Сиэтл	2015-08-31
10.1 Берлин	24,0	Embarcadero RAD Studio 10.1 Берлин	2016-04-20
10.2 Токио	25,0	Embarcadero RAD Studio 10.2 Токио	2017-03-22

Examples

Привет, мир

Эта программа, сохраненная в файле *HelloWorld.dpr* , компилируется в консольное приложение, которое выводит «Hello World» на консоль:

```
program HelloWorld;

{$APPTYPE CONSOLE}

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

Покажите 'Hello World', используя VCL

Эта программа использует VCL, библиотеку компонентов интерфейса UI по умолчанию для Delphi, для печати «Hello World» в окне сообщения. VCL обертывает большинство используемых компонентов WinAPI. Таким образом, их можно использовать намного проще, например, без необходимости работать с Window Handles.

Чтобы включить зависимость (например, `Vcl.Dialogs` в этом случае), добавьте блок `uses` включая список единиц, разделенных запятыми, заканчивающийся точкой с запятой.

```
program HelloWindows;

uses
  Vcl.Dialogs;

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

Показать «Hello World» с помощью WinAPI MessageBox

Эта программа использует Windows API (WinAPI) для печати «Hello World» в окне сообщения.

Чтобы включить зависимость (например, `Windows` в этом случае), добавьте блок uses, включая список единиц, разделенных запятыми, заканчивающийся точкой с запятой.

```
program HelloWorld;

uses
  Windows;

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

Кросс-платформенный Hello World с использованием FireMonkey

XE2

```
program CrossPlatformHelloWorld;

uses
  FMX.Dialogs;

{$R *.res}

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

Большинство поддерживаемых Delphi платформ (Win32 / Win64 / OSX32 / Android32 / iOS32 / iOS64) также поддерживают консоль, поэтому пример `WriteLn` подходит для них хорошо.

Для платформ, для которых требуется графический интерфейс (любое устройство iOS и некоторые устройства Android), приведенный выше пример FireMonkey работает хорошо.

Прочитайте Начало работы с Embarcadero Delphi онлайн:
https://riptutorial.com/ru/delphi/topic/599/начало-работы-с-embarcadero-delphi

глава 2: Loops

Вступление

Язык Delphi обеспечивает 3 типа цикла

`for` - iterator для фиксированной последовательности над целым числом, строкой, массивом или перечислением

условие `repeat-until` quit проверяется после каждого хода, цикл выполняется минимум один раз tmeeven

`while do` do условие проверяется перед каждым поворотом, цикл никогда не может выполняться

Синтаксис

- для OrdinalVariable: = LowerOrdinalValue для UpperOrdinalValue действительно начать { loop-body} end;
- для OrdinalVariable: = UpperOrdinalValue downto LowerOrdinalValue действительно начать {loop-body} end;
- для EnumerableVariable в коллекции do begin {loop-body} end;
- repeat {loop-body} до {break-condition};
- в то время как {условие} действительно начинает {loop-body} конец;

Examples

Перерыв и продолжение в циклах

```
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.
```

Выход:

1
3
4
5
6
7
Конец.

Повторять-До

```
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.
```

Этот короткий пример печати на консоли `Type a words to echo. Enter an empty string to exit.` , дождитесь ввода типа пользователя, повторите его и снова ожидайте ввод в бесконечном цикле - пока пользователь не войдет в пустую строку.

Пока делаете

```
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.
```

В этом примере напечатайте консоль текстового содержимого файла `WhileEOF.dpr` используя `WhileEOF.dpr` «Не `WhileEOF.dpr` `While not(EOF)` . Если файл пуст, `ReadLn-WriteLn` не выполняется.

Прочитайте Loops онлайн: https://riptutorial.com/ru/delphi/topic/9931/loops

глава 3: Дженерики

Examples

Сортировка динамического массива через общий 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
));
```

Простое использование 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;
```

По убыванию от TList что делает его конкретным

```
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;
```

Сортировка TList

```
var List: TList<TDateTime>;

...

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

Прочитайте Дженерики онлайн: https://riptutorial.com/ru/delphi/topic/4054/дженерики

глава 4: Для циклов

Синтаксис

- для OrdinalVariable: = LowerOrdinalValue для UpperOrdinalValue действительно начать { loop-body} end;
- для OrdinalVariable: = UpperOrdinalValue downto LowerOrdinalValue действительно начать {loop-body} end;
- для EnumerableVariable в коллекции do begin {loop-body} end;

замечания

- Синтаксис Delphi `for` -loop не дает ничего, чтобы изменить количество шагов от `1` до любого другого значения.
- При циклировании с переменными порядковыми значениями, например локальными переменными типа `Integer` , верхнее и нижнее значения будут определяться только один раз. Изменения таких переменных не будут влиять на счетчик циклов циклов.

Examples

Простой цикл

Цикл `for` выполняет итерацию от начального значения до конечного значения включительно.

```
program SimpleForLoop;

{$APPTYPE CONSOLE}

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

Выход:

1
2
3
4
5
6
7

8
9
10

Зацикливание символов строки

2005

Следующие итерации над символами строки `s` . Он работает аналогично для циклизации элементов массива или набора, если тип переменной управления циклом (`c` , в этом примере) соответствует типу элемента повторяющегося значения.

```
program ForLoopOnString;

{$APPTYPE CONSOLE}

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

Выход:

E
Икс

м
п
L
е

Обратное направление для цикла

Цикл `for` выполняет итерацию от начального значения до конечного значения включительно, как пример «обратного отсчета».

```
program CountDown;

{$APPTYPE CONSOLE}

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

Выход:

10
9
8
7
6
5
4
3
2
1
0

Для цикла, использующего перечисление

Цикл `for` циклически перебирает элементы в перечислении

```
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.
```

Выход:

Воскресенье
понедельник
вторник
среда
Четверг
пятница
суббота

Для массива

Цикл `for` циклирует элементы в массиве

```
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.
```

Выход:

1,1
2,2
3,3

Прочитайте Для циклов онлайн: https://riptutorial.com/ru/delphi/topic/4643/для-циклов

глава 5: Запуск других программ

Examples

CreateProcess

Следующая функция инкапсулирует код для использования API-интерфейса `CreateProcess` для запуска других программ.

Он настраивается и может подождать, пока процесс вызова не завершится или не вернется немедленно.

Параметры:

- `FileName` - полный путь к исполняемому файлу
- `Params` - параметры командной строки или используйте пустую строку
- `Folder` - рабочая папка для вызываемой программы - если пустой путь будет извлечен из `FileName`
- `WaitUntilTerminated` - если истинная функция будет ждать завершения процесса
- `WaitUntilIdle` - если истинная функция вызовет функцию WaitForInputIdle и дождитесь, пока указанный процесс завершит обработку своего начального ввода и пока не будет ожидающих ввода пользователя
- `RunMinimized` - если истинный процесс будет запущен с минимальным
- `ErrorCode` - если функция не работает, это будет содержать обнаруженный код ошибки Windows

```
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;
```

Использование функции выше

```
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;
```

Документация CreateProcess

Прочитайте Запуск других программ онлайн: https://riptutorial.com/ru/delphi/topic/5180/запуск-других-программ

глава 6: Запуск потока при сохранении графического интерфейса пользователя

Examples

Отзывчивый графический интерфейс с использованием потоков для фоновой работы и PostMessage для отчета из потоков

Сохранение графического интерфейса при длительном процессе требует либо очень сложных «обратных вызовов», чтобы позволить графическому интерфейсу обрабатывать свою очередь сообщений, либо использовать потоки (фоновые) (рабочие).

Уклонение от любого количества потоков для выполнения некоторых работ обычно не является проблемой. Веселье начинается, когда вы хотите, чтобы графический интерфейс показывал промежуточные и конечные результаты или сообщал о прогрессе.

Отображение чего-либо в графическом интерфейсе требует взаимодействия с элементами управления и / или очереди сообщений / насоса. Это всегда должно быть сделано в контексте основного потока. Никогда в контексте какой-либо другой темы.

Есть много способов справиться с этим.

В этом примере показано, как вы можете это сделать, используя простые потоки, позволяя графическому интерфейсу получить доступ к экземпляру потока после его завершения, установив `FreeOnTerminate` в значение `false` и сообщив, когда поток «сделан» с помощью `PostMessage` .

Примечания к условиям гонки: ссылки на рабочие потоки хранятся в массиве в форме. Когда поток завершен, соответствующая ссылка в массиве становится нулевой.

Это потенциальный источник условий гонки. Как и использование «Running» boolean, чтобы было легче определить, есть ли еще какие-то потоки, которые нужно закончить.

Вам нужно будет решить, нужно ли вам защищать этот ресурс с помощью блокировок или нет.

В этом примере, в его нынешнем виде, нет необходимости. Они изменяются только в двух местах: в `StartThreads` метода и `HandleThreadResults` метода. Оба метода выполняются только в контексте основного потока. Пока вы держите его таким образом и не начинаете называть эти методы из контекста разных потоков, у них нет возможности для создания условий гонки.

Нить

```
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;
```

Конструктор просто устанавливает частные члены и устанавливает FreeOnTerminate в False. Это необходимо, так как это позволит основному потоку запрашивать экземпляр потока для его результата.

Метод execute выполняет свой расчет, а затем отправляет сообщение в дескриптор, который он получил в своем конструкторе, чтобы сказать, что он сделан:

```
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;
```

В этом примере важно использовать `PostMessage` . `PostMessage` «just» помещает сообщение в очередь сообщения сообщения основного потока и не ждет его обработки. Он асинхронный по своей природе. Если вы будете использовать `SendMessage` вы будете кодировать себя в рассол. `SendMessage` помещает сообщение в очередь и ждет, пока оно не будет обработано. Короче говоря, это синхронно.

Объявления для пользовательского сообщения UM_WORKERDONE объявляются как:

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

```
    unused: Integer;
    Result: LRESULT;
  end;
```

`UM_WORKERDONE` const использует `WM_APP` в качестве отправной точки для своего значения, чтобы гарантировать, что он не мешает никаким значениям, используемым Windows или Delphi VCL (как рекомендовано MicroSoft).

форма

Любая форма может использоваться для запуска потоков. Все, что вам нужно сделать, это добавить к нему следующих членов:

```
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;
```

О, и примерный код предполагает существование `Memo1: TMemo;` в объявлениях формы, которые он использует для «ведения журнала и отчетности».

`FRunning` можно использовать для предотвращения щелчка GUI во время работы. `FThreads` используется для хранения указателя экземпляра и дескриптора созданных потоков.

Процедура запуска потоков имеет довольно простую реализацию. Он начинается с проверки того, есть ли уже набор ожидающих потоков. Если это так, он просто выходит. Если нет, он устанавливает флаг в значение true и запускает потоки, предоставляющие каждому свой собственный дескриптор, чтобы они знали, куда отправлять сообщение «сделано».

```
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;
```

Рукоятка потока также помещается в массив, потому что это то, что мы получаем в сообщениях, сообщающих нам, что поток выполнен и наличие его вне экземпляра потока облегчает доступ. Наличие дескриптора, доступного вне экземпляра потока, также позволяет нам использовать `FreeOnTerminate` для `True` если нам не нужен экземпляр для получения его результатов (например, если они были сохранены в базе данных). В этом случае, конечно, нет необходимости хранить ссылку на экземпляр.

Веселье в реализации HandleThreadResult:

```
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;
```

Этот метод сначала ищет поток, используя дескриптор, полученный в сообщении. Если совпадение найдено, оно извлекает и сообщает результат потока с помощью экземпляра (`FreeOnTerminate` был `False` , помните?), А затем заканчивается: освобождение экземпляра и установка ссылки на экземпляр и дескриптор на нуль, указав, что этот поток не является более актуальным.

Наконец, он проверяет, работает ли какой-либо из потоков. Если ни один не найден, сообщается «все сделано», а флаг `FRunning` установлен на `False` чтобы можно было запустить новую партию работы.

Прочитайте Запуск потока при сохранении графического интерфейса пользователя онлайн: https://riptutorial.com/ru/delphi/topic/1796/запуск-потока-при-сохранении-графического-интерфейса-пользователя

глава 7: Измерение временных интервалов

Examples

Использование API Windows GetTickCount

Функция `GetTickCount` Windows API возвращает количество миллисекунд с момента запуска системы (компьютера). Простейший пример:

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

Обратите внимание, что `GetTickCount` возвращает 32-разрядный `DWORD` поэтому он обертывает каждые 49.7 дней. Чтобы избежать обертывания, вы можете использовать `GetTickCount64` (доступный с Windows Vista) или специальные процедуры для расчета разности `GetTickCount64` :

```
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;
```

В любом случае эти процедуры возвращают неверные результаты, если интервал двух последующих вызовов `GetTickCount` превышает границу 49,7 дня.

Чтобы преобразовать миллисекунды в несколько секунд:

```
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;
```

Использование записи TStopwatch

Последние версии Delphi поставляются с записью TStopwatch, которая предназначена для измерения временных интервалов. Пример использования:

```
uses
  System.Diagnostics;

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

Прочитайте Измерение временных интервалов онлайн:
https://riptutorial.com/ru/delphi/topic/2425/измерение-временных-интервалов

глава 8: Интерфейсы

замечания

Интерфейсы используются для описания необходимой информации и ожидаемого вывода методов и классов без предоставления информации о явной реализации.

Классы могут **реализовывать** интерфейсы, а интерфейсы могут **наследовать** друг от друга. Если класс **реализует** интерфейс, это означает, что все функции и процедуры, открытые интерфейсом, существуют в классе.

Особым аспектом интерфейсов в delphi является то, что экземпляры интерфейсов имеют управление жизненным циклом, основанное на подсчете ссылок. Время жизни экземпляров класса должно управляться вручную.

Учитывая все эти аспекты, интерфейсы могут использоваться для достижения разных целей:

- Предоставление нескольких различных реализаций для операций (например, сохранение в файле, базе данных или отправка в виде E-Mail, все как интерфейс « SaveData»)
- Сокращение зависимостей, улучшение развязки и, таким образом, улучшение кода обслуживания и проверки кода
- Работайте с экземплярами в нескольких единицах, не беспокоясь о пожизненном управлении (хотя даже здесь существуют ошибки, остерегайтесь!)

Examples

Определение и реализация интерфейса

Интерфейс объявляется как класс, но без модификаторов доступа (`public` , `private` , ...). Кроме того, никакие определения не допускаются, поэтому переменные и константы не могут использоваться.

Интерфейсы всегда должны иметь *уникальный идентификатор* , который можно сгенерировать, нажав `Ctrl` + `Shift` + `G`.

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

Чтобы реализовать интерфейс, имя интерфейса должно быть добавлено за базовым классом. Кроме того, класс должен быть потомком `TInterfacedObject` (это важно для

управления *жизненным* `TInterfacedObject`).

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

Когда класс реализует интерфейс, он должен включать все методы и функции, объявленные в интерфейсе, иначе он не будет компилироваться.

Следует отметить, что модификаторы доступа не имеют никакого влияния, если вызывающий абонент работает с интерфейсом. Например, все функции интерфейса могут быть реализованы как `strict private` члены, но все равно могут быть вызваны из другого класса, если используется экземпляр интерфейса.

Реализация нескольких интерфейсов

Классы могут реализовывать более одного интерфейса, в отличие от наследования более чем одного класса (*множественное наследование*), которое невозможно для классов Delphi. Чтобы достичь этого, имя всех интерфейсов должно быть добавлено запятыми позади базового класса.

Разумеется, класс реализации должен также определять функции, объявленные каждым из интерфейсов.

```
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;
```

Наследование для интерфейсов

Интерфейсы могут наследовать друг от друга, точно так же, как и классы. Таким образом, реализующий класс должен реализовывать функции интерфейса и всех базовых интерфейсов. Таким образом, однако, компилятор не знает, что класс имплантации также реализует базовый интерфейс, он знает только о интерфейсах, которые явно указаны. Вот почему использование `as ISuperInterface` в `TImplementer` не будет работать. Это также приводит к обычной практике, чтобы явно реализовать все базовые интерфейсы (в этом случае `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;
```

Свойства в интерфейсах

Поскольку объявление переменных в интерфейсах невозможно, «быстрый» способ определения свойств (`property Value: TObject read FValue write FValue;`) не может быть использован. Вместо этого, Getter и setter (каждый только в случае необходимости) также должны быть объявлены в интерфейсе.

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

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

Следует отметить, что класс реализации не должен объявлять свойство. Компилятор примет этот код:

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

Однако одно предостережение заключается в том, что таким образом свойство можно получить только через экземпляр интерфейса, а не через сам класс. Кроме того, добавление свойства в класс повышает читаемость.

Прочитайте Интерфейсы онлайн: https://riptutorial.com/ru/delphi/topic/4885/интерфейсы

глава 9: Использование RTTI в Delphi

Вступление

Delphi предоставила информацию о времени выполнения (RTTI) более десяти лет назад. Однако даже сегодня многие разработчики не в полной мере осознают свои риски и выгоды.

Короче говоря, Runtime Type Information - это информация о типе данных объекта, который устанавливается в память во время выполнения.

RTTI предоставляет возможность определить, является ли тип объекта определенным классом или его потомком.

замечания

RTTI IN DELPHI - ОБЪЯСНЕННЫЙ

Информация о времени выполнения в Delphi - может ли она что-нибудь для вас делать? статья Брайана Лонга дает большое представление о возможностях RTTI Delphi. Брайан объясняет, что поддержка RTTI в Delphi была добавлена прежде всего для того, чтобы среда разработки могла выполнять свою работу, но разработчики также могут воспользоваться ею для достижения определенных упрощений кода. В этой статье также представлен большой обзор классов RTTI, а также несколько примеров.

Примеры включают: чтение и запись произвольных свойств, общие свойства без общего предка, копирование свойств из одного компонента в другой и т. Д.

Examples

Основная информация о классе

В этом примере показано, как получить родословную компонента с использованием свойств `ClassType` и `ClassParent` . Он использует кнопку `Button1: TButton` и список `ListBox1: TListBox` на форме `TForm1` .

Когда пользователь нажимает кнопку, в поле списка добавляется имя класса кнопки и имена его родительских классов.

```
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;
```

Окно списка содержит следующие строки после нажатия пользователем кнопки:

- TButton
- TButtonControl
- TWinControl
- TControl
- TСотропепЬ
- TPersistent
- TObject

Прочитайте Использование RTTI в Delphi онлайн: https://riptutorial.com/ru/delphi/topic/9578/использование-rtti-в-delphi

глава 10: Использование try, кроме, и, наконец,

Синтаксис

1. Try-except: try [statements] кроме [[[на E: ExceptionType do statement]] [else statement] | [заявления];

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

Examples

Простой пример. Наконец, чтобы избежать утечек памяти

Используйте `try` - `finally` чтобы избежать утечки ресурсов (например, памяти) в случае возникновения исключения во время выполнения.

Следующая процедура сохраняет строку в файле и предотвращает утечку `TStringList` .

```
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;
```

Независимо от того, возникает ли исключение при сохранении файла, `SL` будет освобожден. Любое исключение отправляется вызывающему абоненту.

Исключительно безопасный возврат нового объекта

Когда функция *возвращает* объект (вместо того, чтобы *использовать* тот, который был передан вызывающим), будьте осторожны, исключение не вызывает утечку объекта.

```
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;
```

Наивные программисты могут попытаться поймать все типы исключений и вернуть `nil` от такой функции, но это всего лишь частный случай общей непривлекательной практики ловли всех типов исключений без их обработки.

Try-finally, вложенный внутри try-except

Блок `try finally` может быть вложен внутри блока `try except` .

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

Если в `UseResource` возникает `UseResource` , выполнение переходит к `ReleaseResource` . Если исключением является `EResourceUsageError` , то выполнение перейдет к обработчику исключений и вызовет `HandleResourceErrors` . Исключения любого другого типа пропустят обработчик исключений выше и перейдут к следующей `try` - `except` блокировки стека вызовов.

Исключения в `AcquireResource` или `ReleaseResource` заставят выполнение перейти к обработчику исключений, пропуская блок `finally` , либо потому, что соответствующий блок `try` еще не введен, либо потому, что блок `finally` *уже* введен.

Попробовать - кроме вложенного внутри try-finally

Блок `try except` может быть вложен в блок `try finally` .

```
AcquireResource;
try
```

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

Если в `EResourceUsageError` появляется `UseResource2` , то выполнение перейдет к обработчику исключений и вызовет `HandleResourceError` . Исключение будет считаться обработанным, поэтому выполнение продолжит `UseResource3` , а затем `ReleaseResource` .

Если в `UseResource2` возникает исключение любого другого типа, то обработчик исключений здесь не будет применяться, поэтому выполнение перепрыгнет через вызов `UseResource3` и перейдет непосредственно к блоку `finally` , где будет вызываться `ReleaseResource` . После этого выполнение перейдет к следующему применимому обработчику исключений, поскольку исключение пузырится вверх по стеке вызовов.

Если исключение возникает в любом другом вызове в приведенном выше примере, тогда `HandleResourceErrors` *не* будет вызываться. Это связано с тем, что ни один из других вызовов не возникает внутри блока `try` соответствующего этому обработчику исключений.

Попробуйте, наконец, с двумя или более объектами

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

Если вы не инициализируете объекты с помощью `nil` вне блока `try-finally` , если один из них не может быть создан, AV будет происходить в блоке finally, потому что объект не будет равен nil (поскольку он не был инициализирован) и приведет к исключению.

Метод `Free` проверяет, равен ли объект нулю, поэтому инициализация обоих объектов с помощью `nil` позволяет избежать ошибок при их освобождении, если они не были созданы.

Прочитайте Использование try, кроме, и, наконец, онлайн:
https://riptutorial.com/ru/delphi/topic/3055/использование-try--кроме--и--наконец-

глава 11: Использование анимаций в Firemonkey

Examples

Вращающийся треугольник

1. Создайте пустое приложение Multi-Device (Firemonkey).
2. Drop Прямоугольник на форме.
3. В окне Инспектор объектов (F11) найдите RotationAngle, нажмите кнопку выпадающего списка и выберите «Создать новую TFloatAnimation».
4. Окно инспектора объектов автоматически переключается на недавно добавленную TFloatAnimation, вы также можете просмотреть его в меню «Структура» (Shift + Alt
    - F11).
5. В инспекторе объектов TFloatAnimation заполняется длительность с любым числом (в секундах). В нашем случае давайте возьмем 1. Оставьте свойство StartValue как есть, а в типе StopValue - 360 (градусы, так что все идет кругом). Также включите опцию Loop (эта петля анимации, пока вы не остановите ее из кода).

Теперь у нас есть анимация. Осталось только включить его: сбросить две кнопки по форме, сначала вызвать «Старт», второй - «Остановить». в событии OnClick первой кнопки пишите:

```
FloatAnimation1.Start;
```

OnClick второго кода кнопки:

```
FloatAnimation1.Stop;
```

Если вы изменили имя своего TFloatAnimation - также измените его при вызове «Пуск» и «Стоп».

Теперь запустите свой проект, нажмите кнопку «Пуск» и наслаждайтесь.

Прочитайте Использование анимаций в Firemonkey онлайн:
https://riptutorial.com/ru/delphi/topic/5383/использование-анимаций-в-firemonkey

глава 12: Класс TStringList

Examples

Вступление

TStringList является потомком класса TStrings для VCL. TStringList может использоваться для хранения и управления списком строк. Хотя первоначально предназначенный для строк, любой тип объектов можно также манипулировать с помощью этого класса.

TStringList широко используется в VCL, когда целью является поддержание списка строк. TStringList поддерживает богатый набор методов, которые обеспечивают высокий уровень настройки и простоту манипулирования.

Следующий пример демонстрирует создание, добавление строк, сортировку, извлечение и освобождение объекта TStringList.

```
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 имеет множество пользовательских случаев, включая обработку строк, сортировку, индексацию, спаривание значений ключа и разделение разделителей между ними.

Сопоставление ключевых значений

Вы можете использовать TStringList для хранения пар ключ-значение. Это может быть полезно, если вы хотите сохранить настройки, например. Настройки состоят из клавиши (Идентификатор настройки) и значения. Каждая пара Key-Value хранится в одной строке StringList в формате Key = Value.

```
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;
```

В этом примере список Stringlist имеет следующее содержимое до его уничтожения:

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

Примечание по эффективности

Под капотом `TStringList` выполняет поиск по ключевым `TStringList` путем прямого цикла всех элементов, поиска разделителя внутри каждого элемента и сравнения части имени с данным ключом. Не нужно говорить, что это оказывает огромное влияние на производительность, поэтому этот механизм следует использовать только в некритических, редко повторяющихся местах. В тех случаях, когда важна

производительность, следует использовать `TDictionary<TKey,TValue>` из `System.Generics.Collections` который реализует поиск хеш-таблицы или сохраняет ключи в **отсортированном** `TStringList` со значениями, хранящимися как `Object` s, с использованием алгоритма поиска двоичных файлов.

Прочитайте Класс TStringList онлайн: https://riptutorial.com/ru/delphi/topic/6045/класс-tstringlist

глава 13: Получение обновленных данных TDataSet в фоновом потоке

замечания

Этот пример FireDAC и другие, которые я планирую отправить, будут избегать использования собственных вызовов для асинхронного открытия набора данных.

Examples

Пример FireDAC

В приведенном ниже примере кода показан один из способов получения записей с сервера MSSql в фоновом потоке с использованием FireDAC. Протестировано для Delphi 10 Сиэтл

Как написано:

- Поток извлекает данные с помощью собственного TFDConnection и TFDQuery и передает данные в FDQuery формы при вызове Sychronize ().
- Execute извлекает данные только один раз. Он может быть изменен для повторного запроса запроса в ответ на сообщение, отправленное из потока VCL.

Код:

```
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.
```

Прочитайте Получение обновленных данных TDataSet в фоновом потоке онлайн: https://riptutorial.com/ru/delphi/topic/4114/получение-обновленных-данных-tdataset-в-фоновом-потоке

глава 14: Создание легко удаляемых проверок времени выполнения

Вступление

Это показывает, как обычная процедура проверки выполнения во время вашего собственного создания может быть легко включена, чтобы она не генерировала накладные расходы кода при ее отключении.

Examples

Тривиальный пример

```
{$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;
```

Концепция в основном такова:

Определенный символ используется для включения кода. Он также останавливает код в явном виде, что означает, что проще поставить точку останова в процедуру проверки.

Однако настоящая красота этой конструкции заключается в том, что вы *больше не* хотите проверки. Комментируя ***$ DEFINE*** (поместите '//' перед ним), вы не только удалите контрольный код, но также *включите* ***встроенную*** подпрограмму и, таким образом, удалите все накладные расходы со всех мест, где вы вызывали рутина! Компилятор полностью удалит все следы вашего чека (при условии, что для самой встраивания установлено значение «Вкл.» Или «Авто», конечно).

Приведенный выше пример по существу похож на понятие «утверждения», и ваша первая строка может установить результат в TRUE или FALSE в зависимости от использования.

Но теперь вы также можете использовать этот способ построения для кода, который отслеживает трассировку, метрики, что угодно. Например:

```
   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.
```

Прочитайте Создание легко удаляемых проверок времени выполнения онлайн: https://riptutorial.com/ru/delphi/topic/10541/создание-легко-удаляемых-проверок-времени-выполнения

глава 15: Струны

Examples

Строковые типы

Delphi имеет следующие типы строк (в порядке их популярности):

Тип	Максимальная длина	Минимальный размер	Описание
`string`	2 Гб	16 байт	Управляемая строка. Псевдоним для `AnsiString` через Delphi 2007 и псевдоним для `UnicodeString` с Delphi 2009.
`UnicodeString`	2 Гб	16 байт	Управляемая строка в формате UTF-16.
`AnsiString`	2 Гб	16 байт	Управляемая строка в формате ANSI в формате Unicode. Начиная с Delphi 2009, он содержит явный индикатор кодовой страницы.
`UTF8String`	2 Гб	16 байт	Управляемая строка в формате UTF-8, реализованная как `AnsiString` с кодовой страницей UTF-8.
`ShortString`	255 символов	2 байта	Унаследованная, фиксированная длина, неуправляемая строка с очень небольшими накладными расходами
`WideString`	2 Гб	4 байта	Предназначен для COM-взаимодействия, управляемой строки в формате UTF-16. Эквивалентен типу Windows `BSTR` .

`UnicodeString` и `AnsiString` - подсчет ссылок и копирование на запись (COW).
`ShortString` и `WideString` не подсчитываются и не имеют семантики COW.

Струны

```
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
```

Символов

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
```

Предложение `uses` должно быть `System.Character` если версия XE2 или выше.

Верхний и нижний регистры

```
uses
  SysUtils;

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

присваивание

Присвоение строки различным типам строк и поведение среды выполнения в отношении них. Распределение памяти, подсчет ссылок, индексированный доступ к символам и ошибки компилятора, описанные кратко, где это применимо.

```
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 }
```

Подсчет ссылок

Подсчет ссылок на строки является потокобезопасным. Для защиты процесса используются блокировки и обработчики исключений. Рассмотрим следующий код с комментариями, указывающими, где компилятор вводит код во время компиляции для управления подсчетами ссылок:

```
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.
```

Как показано выше, введение временной локальной строки для сохранения модификаций параметра связано с теми же накладными расходами, что и внесение изменений непосредственно в этот параметр. Объявление строки `const` исключает подсчет ссылок, когда параметр строки действительно доступен только для чтения. Однако, чтобы избежать утечек детали реализации вне функции, рекомендуется всегда использовать один из `const` , `var` , или `out` на строковом параметре.

Кодировки

Строковые типы, такие как UnicodeString, AnsiString, WideString и UTF8String, хранятся в памяти с использованием их соответствующей кодировки (подробнее см. «Типы строк»). Присвоение одного типа строк в другой может привести к конверсии. Строка типа предназначена для независимого кодирования - вы никогда не должны использовать ее внутреннее представление.

Класс `Sysutils.TEncoding` предоставляет метод `GetBytes` для преобразования `string` в `TBytes` (массив байтов) и `GetString` для преобразования `TBytes` в `string` . Класс `Sysutils.TEncoding` также предоставляет множество предопределенных кодировок в качестве свойств класса.

Один из способов решения кодировок - использовать только тип `string` в вашем приложении и использовать `TEncoding` каждый раз, когда вам нужно использовать определенную кодировку - как правило, в операциях ввода-вывода, вызовах DLL и т. Д. ...

```
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;
```

Прочитайте Струны онлайн: https://riptutorial.com/ru/delphi/topic/3957/струны

кредиты

S. No	Главы	Contributors
1	Начало работы с Embarcadero Delphi	Charlie H, Community, Dalija Prasnikar, Florian Koch, Jeroen Wiert Pluimers, René Hoffmann, RepeatUntil, Rob Kennedy, Vadim Shakun, w5m, Y.N, Zam
2	Loops	Y.N
3	Дженерики	Rob Kennedy, Steffen Binas, Uli Gerhardt
4	Для циклов	Filipe Martins, Jeroen Wiert Pluimers, John Easley, René Hoffmann, Rob Kennedy, Siendor, Y.N
5	Запуск других программ	Dalija Prasnikar
6	Запуск потока при сохранении графического интерфейса пользователя	Fr0sT, Jerry Dodge, Johan, kami, LU RD, Marjan Venema
7	Измерение временных интервалов	Fr0sT, John Easley, kludg, Rob Kennedy, Victoria, Wolf
8	Интерфейсы	Florian Koch, Willo van der Merwe
9	Использование RTTI в Delphi	Petzy, René Hoffmann
10	Использование try, кроме, и, наконец,	EMBarbosa, Fabio Gomes, Johan, MrE, Nick Hodges, Rob Kennedy, Shadow
11	Использование анимаций в Firemonkey	Alexander Petrosyan
12	Класс TStringList	Charlie H, Fabricio Araujo, Fr0sT, KaiW
13	Получение обновленных	MartynA

	данных TDataSet в фоновом потоке	
14	Создание легко удаляемых проверок времени выполнения	Alex T
15	Струны	AlekXL, Dalija Prasnikar, EMBarbosa, Fabricio Araujo, Johan, Radek Hladík, René Hoffmann, RepeatUntil, Rob Kennedy, Rudy Velthuis