EBook Gratis

APRENDIZAJE

Embarcadero Delphi

Free unaffiliated eBook created from
Stack Overflow contributors.

#delphi

Tabla de contenido

Acerca de

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

Capítulo 1: Comenzando con Embarcadero Delphi

Observaciones

Delphi es un lenguaje de propósito general basado en un dialecto Object Pascal con sus raíces que proviene de Borland Turbo Pascal. Viene con su propio IDE diseñado para soportar el rápido desarrollo de aplicaciones (RAD).

Permite el desarrollo de aplicaciones nativas (compiladas) multiplataforma desde una base de código única. Las plataformas actualmente soportadas son Windows, OSX, iOS y Android.

Viene con dos marcos visuales:

- VCL: Visual Component Library, diseñada específicamente para el desarrollo de Windows, incluye controles y soporte nativos de Windows para crear controles personalizados.
- FMX: Framework multiplataforma FireMonkey para todas las plataformas compatibles

Versiones

Versión	Version numerica	Nombre del producto	Fecha de lanzamiento
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 para .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

Versión	Version numerica	Nombre del producto	Fecha de lanzamiento
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 Tokio	25.0	Embarcadero RAD Studio 10.2 Tokyo	2017-03-22

Examples

Hola Mundo

Este programa, guardado en un archivo llamado *HelloWorld.dpr* , se compila en una aplicación de consola que imprime "Hello World" en la consola:

```
program HelloWorld;

{$APPTYPE CONSOLE}

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

Mostrar 'Hello World' usando el VCL

Este programa utiliza VCL, la biblioteca de componentes de interfaz de usuario predeterminada de Delphi, para imprimir "Hola mundo" en un cuadro de mensaje. La VCL envuelve la mayoría de

los componentes WinAPI utilizados comúnmente. De esta manera, se pueden usar mucho más fácilmente, por ejemplo, sin la necesidad de trabajar con los controladores de ventana.

Para incluir una dependencia (como `Vcl.Dialogs` en este caso), agregue el bloque de `uses` que incluye una lista de unidades separadas por comas que terminan con un punto y coma.

```
program HelloWindows;

uses
  Vcl.Dialogs;

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

Mostrar 'Hello World' usando WinAPI MessageBox

Este programa utiliza la API de Windows (WinAPI) para imprimir "Hello World" en un cuadro de mensaje.

Para incluir una dependencia (como `Windows` en este caso), agregue el bloque de usos que incluye una lista de unidades separadas por comas que terminan con un punto y coma.

```
program HelloWorld;

uses
  Windows;

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

Plataforma multiplata Hello World utilizando FireMonkey

XE2

```
program CrossPlatformHelloWorld;

uses
  FMX.Dialogs;

{$R *.res}

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

La mayoría de las plataformas compatibles con Delphi (Win32 / Win64 / OSX32 / Android32 / iOS32 / iOS64) también son compatibles con una consola, por lo que el ejemplo de `WriteLn` les queda bien.

Para las plataformas que requieren una GUI (cualquier dispositivo iOS y algunos dispositivos Android), el ejemplo anterior de FireMonkey funciona bien.

Lea Comenzando con Embarcadero Delphi en línea:
https://riptutorial.com/es/delphi/topic/599/comenzando-con-embarcadero-delphi

Capítulo 2: Bucles

Introducción

El lenguaje Delphi proporciona 3 tipos de bucle

`for` - iterator para secuencia fija sobre entero, cadena, matriz o enumeración

`repeat-until` : la condición para salir se verifique después de cada turno, el bucle se ejecuta al menos una vez

`while do` condición `while do` - do se verifica antes de cada turno, el bucle nunca podría ejecutarse

Sintaxis

- para OrdinalVariable: = LowerOrdinalValue to UpperOrdinalValue comience {loop-body} end;
- para OrdinalVariable: = UpperOrdinalValue down to LowerOrdinalValue do comienza {loop-body} end;
- para EnumerableVariable in Collection, comience {loop-body} end;
- repita {loop-body} hasta que {break-condition};
- mientras que {condición} comienza {bucle-cuerpo} termina;

Examples

Romper y continuar en bucles

```
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.
```

Salida:

1
3
4

5
6
7
Terminar.

Repetir hasta

```
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.
```

Este ejemplo corto se imprime en la consola `Type a words to echo. Enter an empty string to exit.` , espere el tipo de usuario, haga eco y vuelva a esperar la entrada en un bucle infinito, hasta que el usuario ingrese la cadena vacía.

Mientras hace

```
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.
```

Este ejemplo imprime para consolar el contenido de texto del archivo `WhileEOF.dpr` usando la condición `While not(EOF)` . Si el archivo está vacío, entonces no se ejecuta el ciclo `ReadLn-WriteLn` .

Lea Bucles en línea: https://riptutorial.com/es/delphi/topic/9931/bucles

Capítulo 3: Clase TStringList

Examples

Introducción

TStringList es un descendiente de la clase TStrings del VCL. TStringList se puede utilizar para almacenar y manipular la lista de cadenas. Aunque originalmente estaba destinado a cadenas, cualquier tipo de objetos también se pueden manipular usando esta clase.

TStringList se usa ampliamente en VCL cuando el propósito es mantener una lista de cadenas. TStringList admite un amplio conjunto de métodos que ofrecen un alto nivel de personalización y facilidad de manipulación.

El siguiente ejemplo muestra la creación, adición de cadenas, clasificación, recuperación y liberación de un objeto 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 tiene una variedad de casos de usuarios que incluyen manipulación de cadenas, clasificación, indexación, emparejamiento clave-valor y separación de delimitadores entre ellos.

Emparejamiento clave-valor

Puede usar una TStringList para almacenar pares clave-valor. Esto puede ser útil si desea almacenar configuraciones, por ejemplo. Una configuración consiste en una clave (el identificador

de la configuración) y el valor. Cada par clave-valor se almacena en una línea de StringList en el formato clave = valor.

```
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;
```

En este ejemplo, la lista de cadenas tiene el siguiente contenido antes de que se destruya:

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

Nota sobre el rendimiento

Bajo el capó, `TStringList` realiza una búsqueda de claves haciendo un bucle directo a través de todos los elementos, buscando un separador dentro de cada elemento y comparando la parte del nombre con la clave dada. No es necesario decir que tiene un gran impacto en el rendimiento, por lo que este mecanismo solo debe utilizarse en lugares no críticos, que rara vez se repiten. En los casos en los que el rendimiento es importante, se debe usar `TDictionary<TKey,TValue>` de `System.Generics.Collections` que implementa la búsqueda de tablas hash o para mantener las claves **ordenadas** en `TStringList` con los valores almacenados como `Object` , utilizando el algoritmo de búsqueda binaria.

Lea Clase TStringList en línea: https://riptutorial.com/es/delphi/topic/6045/clase-tstringlist

Capítulo 4: Creación de cheques de error de tiempo de ejecución fácilmente extraíbles

Introducción

Esto muestra cómo una rutina de verificación de errores de tiempo de ejecución de su propia creación se puede incorporar fácilmente para que no genere ninguna sobrecarga de código cuando se apaga.

Examples

Ejemplo trivial

```
{$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;
```

El concepto es básicamente este:

El símbolo definido se utiliza para activar el uso del código. También evita que el código esté explícitamente en línea, lo que significa que es más fácil poner un punto de interrupción en la rutina de verificación.

Sin embargo, la verdadera belleza de esta construcción es cuando ya *no* quieres el cheque. Comentando la ***$ DEFINE*** (poner '//' en frente de ella) que no sólo va a eliminar el código de verificación, sino que también se *encienda la* ***línea*** de la rutina y por lo tanto eliminar cualquier gasto general de todos los lugares donde se invoca ¡la rutina! El compilador eliminará todos los rastros de su cheque por completo (asumiendo que la alineación en sí misma está configurada en "On" o "Auto", por supuesto).

El ejemplo anterior es esencialmente similar al concepto de "aserciones", y su primera línea podría establecer el resultado en VERDADERO o FALSO según sea apropiado para el uso.

Pero ahora también es libre de utilizar esta forma de construcción para el código que realiza el registro de seguimiento, las métricas, lo que sea. Por ejemplo:

```
   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.
```

Lea Creación de cheques de error de tiempo de ejecución fácilmente extraíbles en línea: https://riptutorial.com/es/delphi/topic/10541/creacion-de-cheques-de-error-de-tiempo-de-ejecucion-facilmente-extraibles

Capítulo 5: Ejecutando otros programas

Examples

Proceso de creación

La siguiente función encapsula el código para usar la API de Windows `CreateProcess` para iniciar otros programas.

Es configurable y puede esperar hasta que el proceso de llamada finalice o regrese inmediatamente.

Parámetros:

- `FileName` - ruta completa al ejecutable
- `Params` - Parámetros de línea de comandos o utilizar cadena vacía
- `Folder` - carpeta de trabajo para el programa llamado - si la ruta vacía será extraída de `FileName`
- `WaitUntilTerminated` : si la función true esperará a que el proceso finalice la ejecución
- `WaitUntilIdle` : si la función true llamará a la función WaitForInputIdle y esperará hasta que el proceso especificado haya terminado de procesar su entrada inicial y hasta que no haya ninguna entrada de usuario pendiente
- `RunMinimized` - si el proceso verdadero se ejecuta minimizado
- `ErrorCode` - si la función falla este código de error de Windows contendrá encontrado

```
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;
```

Uso de la función anterior

```
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;
```

Documentación de CreateProcess

Lea Ejecutando otros programas en línea: https://riptutorial.com/es/delphi/topic/5180/ejecutando-otros-programas

Capítulo 6: Ejecutar un hilo manteniendo GUI sensible

Examples

Interfaz gráfica de usuario receptiva que usa hilos para trabajos en segundo plano y PostMessage para informar sobre los hilos

Mantener una interfaz gráfica de usuario receptiva mientras se ejecuta un proceso largo requiere o bien algunas "devoluciones de llamada" muy elaboradas para permitir que la interfaz gráfica de usuario procese su cola de mensajes, o el uso de subprocesos (de fondo) (trabajador).

Iniciar cualquier cantidad de subprocesos para hacer algún trabajo por lo general no es un problema. La diversión comienza cuando desea que la GUI muestre resultados intermedios y finales o informe sobre el progreso.

Mostrar cualquier cosa en la GUI requiere interactuar con los controles y / o la cola / bomba de mensajes. Eso siempre debe hacerse en el contexto del hilo principal. Nunca en el contexto de cualquier otro hilo.

Hay muchas maneras de manejar esto.

Este ejemplo muestra cómo puede hacerlo utilizando subprocesos simples, permitiendo que la GUI acceda a la instancia del subproceso después de que termine configurando `FreeOnTerminate` en `false` e informando cuando un subproceso se "hace" con `PostMessage` .

Notas sobre las condiciones de carrera: las referencias a los subprocesos de trabajo se mantienen en una matriz en el formulario. Cuando se termina un hilo, la referencia correspondiente en la matriz se anula.

Esta es una fuente potencial de condiciones de carrera. Como es el uso de un booleano "En ejecución" para que sea más fácil determinar si todavía hay algún hilo que necesita terminar.

Tendrá que decidir si necesita proteger estos recursos utilizando bloqueos o no.

En este ejemplo, tal como está, no hay necesidad. Solo se modifican en dos ubicaciones: el método `StartThreads` y el método `HandleThreadResults` . Ambos métodos solo se ejecutan en el contexto del hilo principal. Mientras lo mantengas así y no comiences a llamar estos métodos desde el contexto de diferentes hilos, no hay forma de que produzcan condiciones de carrera.

Hilo

```
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;
```

El constructor simplemente establece los miembros privados y establece FreeOnTerminate en False. Esto es esencial ya que permitirá al hilo principal consultar la instancia del hilo para ver su resultado.

El método de ejecución realiza su cálculo y luego publica un mensaje en el identificador que recibió en su constructor para decir que se realizó:

```
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;
```

El uso de `PostMessage` es esencial en este ejemplo. `PostMessage` "solo" pone un mensaje en la cola de la bomba de mensajes del hilo principal y no espera a que se maneje. Es de naturaleza asíncrona. Si fueras a usar `SendMessage` , estarías codificándote en un encurtido. `SendMessage` pone el mensaje en la cola y espera hasta que se haya procesado. En resumen, es síncrono.

Las declaraciones para el mensaje UM_WORKERDONE personalizado se declaran como:

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

La const. `UM_WORKERDONE` usa `WM_APP` como punto de partida para su valor para garantizar que no interfiera con ningún valor usado por Windows o Delphi VCL (según lo recomendado por MicroSoft).

Formar

Cualquier forma se puede utilizar para iniciar hilos. Todo lo que necesitas hacer es agregar los siguientes miembros:

```
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;
```

Ah, y el código de ejemplo supone la existencia de un `Memo1: TMemo;` en las declaraciones del formulario, que utiliza para "registro e informes".

Se puede utilizar `FRunning` para evitar que se `FRunning` clic en la GUI mientras se está realizando el trabajo. `FThreads` se utiliza para mantener el puntero de instancia y el identificador de los subprocesos creados.

El procedimiento para iniciar los hilos tiene una implementación bastante sencilla. Comienza con una verificación de si ya hay un conjunto de subprocesos en espera. Si es así, simplemente sale. Si no, establece el indicador en verdadero e inicia los subprocesos proporcionando a cada uno su propio identificador para que sepan dónde publicar su mensaje "terminado".

```
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;
```

El identificador del subproceso también se coloca en la matriz porque eso es lo que recibimos en los mensajes que nos dicen que se ha realizado un subproceso y tenerlo fuera de la instancia del subproceso hace que sea un poco más fácil acceder. Tener el identificador disponible fuera de la instancia del hilo también nos permite usar `FreeOnTerminate` en `True` si no necesitamos la instancia para obtener sus resultados (por ejemplo, si se han almacenado en una base de datos). En ese caso, por supuesto, no habría necesidad de mantener una referencia a la instancia.

La diversión está en la implementación de 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;
```

Este método primero busca el hilo usando el identificador recibido en el mensaje. Si se encontró una coincidencia, recupera e informa el resultado del hilo usando la instancia (`FreeOnTerminate` era `False` , ¿recuerdas?), Y luego finaliza: libera la instancia y configura la referencia de la instancia y el identificador a cero, lo que indica que este hilo no es ya relevante

Finalmente, comprueba si alguno de los subprocesos todavía se está ejecutando. Si no se encuentra ninguno, se informa "todo hecho" y el indicador de `FRunning` establece en `False` para que se pueda iniciar un nuevo lote de trabajo.

Lea Ejecutar un hilo manteniendo GUI sensible en línea:
https://riptutorial.com/es/delphi/topic/1796/ejecutar-un-hilo-manteniendo-gui-sensible

Capítulo 7: Genéricos

Examples

Ordenar una matriz dinámica a través de TArray genérico.

```
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
));
```

Uso simple de 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;
```

Descendiendo de TList haciéndolo específico

```
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;
```

Ordenar una lista

```
var List: TList<TDateTime>;

...

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

Lea Genéricos en línea: https://riptutorial.com/es/delphi/topic/4054/genericos

Capítulo 8: Instrumentos de cuerda

Examples

Tipos de cuerdas

Delphi tiene los siguientes tipos de cadena (en orden de popularidad):

Tipo	Longitud maxima	Talla minima	Descripción
`string`	2GB	16 bytes	Una cadena manejada. Un alias para `AnsiString` través de Delphi 2007 y un alias para `UnicodeString` partir de Delphi 2009.
`UnicodeString`	2GB	16 bytes	Una cadena gestionada en formato UTF-16.
`AnsiString`	2GB	16 bytes	Una cadena administrada en formato ANSI anterior a Unicode. A partir de Delphi 2009, lleva un indicador de página de código explícito.
`UTF8String`	2GB	16 bytes	Una cadena administrada en formato UTF-8, implementada como `AnsiString` con una página de códigos UTF-8.
`ShortString`	255 caracteres	2 bytes	Una cadena heredada, de longitud fija, no administrada con muy poca sobrecarga
`WideString`	2GB	4 bytes	Diseñado para interoperabilidad COM, una cadena administrada en formato UTF-16. Equivalente al tipo `BSTR` Windows.

`UnicodeString` y `AnsiString` son referencias contabilizadas y copia en escritura (COW). `ShortString` y `WideString` no se cuentan como referencia y no tienen semántica COW.

Instrumentos de cuerda

```
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
```

Los caracteres

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
```

La cláusula de `uses` debe ser `System.Character` si la versión es XE2 o superior.

SUPERIOR y minúscula

```
uses
  SysUtils;

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

Asignación

Asignación de cadenas a diferentes tipos de cadenas y cómo se comporta el entorno de ejecución con respecto a ellas. La asignación de memoria, el recuento de referencias, el acceso indexado a los caracteres y los errores del compilador se describen brevemente cuando corresponda.

```
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 }
```

Recuento de referencias

Contar referencias en cadenas es seguro para subprocesos. Se utilizan candados y controladores de excepciones para salvaguardar el proceso. Considere el siguiente código, con comentarios que indican dónde el compilador inserta el código en el momento de la compilación para administrar los recuentos de referencias:

```
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.
```

Como se muestra arriba, la introducción de una cadena local temporal para contener las modificaciones a un parámetro implica la misma sobrecarga que hacer modificaciones directamente a ese parámetro. La declaración de una cadena `const` solo evita el conteo de referencias cuando el parámetro de cadena es realmente de solo lectura. Sin embargo, para evitar filtrar detalles de implementación fuera de una función, es recomendable utilizar siempre uno de `const` , `var` o `out` en el parámetro de cadena.

Codificaciones

Los tipos de cadena como UnicodeString, AnsiString, WideString y UTF8String se almacenan en una memoria con su codificación respectiva (consulte Tipos de cadena para obtener más detalles). Asignar un tipo de cadena a otro puede resultar en una conversión. La cadena de tipo está diseñada para ser independiente de la codificación; nunca debe usar su representación interna.

La clase `Sysutils.TEncoding` proporciona el método `GetBytes` para convertir `string` a `TBytes` (matriz de bytes) y `GetString` para convertir `TBytes` a `string` . La clase `Sysutils.TEncoding` también proporciona muchas codificaciones predefinidas como propiedades de clase.

Una forma de lidiar con las codificaciones es usar solo el tipo de `string` en su aplicación y usar la `TEncoding` cada vez que necesite usar una codificación específica, generalmente en operaciones de E / S, llamadas a DLL, 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;
```

Lea Instrumentos de cuerda en línea: https://riptutorial.com/es/delphi/topic/3957/instrumentos-de-cuerda

Capítulo 9: Interfaces

Observaciones

Las interfaces se utilizan para describir la información necesaria y el resultado esperado de los métodos y clases, sin proporcionar información de la implementación explícita.

Las clases pueden **implementar** interfaces, y las interfaces pueden **heredarse** unas de otras. Si una clase está **implementando** una interfaz, esto significa que todas las funciones y procedimientos expuestos por la interfaz existen en la clase.

Un aspecto especial de las interfaces en delphi es que las instancias de las interfaces tienen una administración de por vida basada en el conteo de referencias. El tiempo de vida de las instancias de clase debe gestionarse manualmente.

Teniendo en cuenta todos estos aspectos, las interfaces se pueden utilizar para lograr diferentes objetivos:

- Proporcionar múltiples implementaciones diferentes para las operaciones (por ejemplo, guardar en un archivo, base de datos o enviar como correo electrónico, todo como interfaz "Guardar datos")
- Reduzca las dependencias, mejorando el desacoplamiento y haciendo así que el código sea más fácil de mantener y verificable
- Trabaje con instancias en varias unidades sin tener problemas con la administración de por vida (aunque incluso aquí existen trampas, ¡tenga cuidado!)

Examples

Definir e implementar una interfaz.

Una interfaz se declara como una clase, pero sin modificadores de acceso (`public` , `private` , ...). Además, no se permiten definiciones, por lo que no se pueden usar variables y constantes.

Las interfaces siempre deben tener un *identificador único* , que se puede generar presionando `Ctrl` + `Shift` + `G`.

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

Para implementar una interfaz, el nombre de la interfaz debe agregarse detrás de la clase base. Además, la clase debe ser descendiente de `TInterfacedObject` (esto es importante para la *administración de* por *vida*).

```
TDatabaseRepository = class(TInterfacedObject, IRepository)
```

```
    function  SaveKeyValuePair(aKey: Integer; aValue: string): Boolean;
end;
```

Cuando una clase implementa una interfaz, debe incluir todos los métodos y funciones declarados en la interfaz, de lo contrario no compilará.

Una cosa que vale la pena mencionar es que los modificadores de acceso no tienen ninguna influencia, si la persona que llama trabaja con la interfaz. Por ejemplo, todas las funciones de la interfaz se pueden implementar como miembros `strict private` , pero aún se pueden llamar desde otra clase si se utiliza una instancia de la interfaz.

Implementando multiples interfaces

Las clases pueden implementar más de una interfaz, en lugar de heredar de más de una clase (*herencia múltiple*) que no es posible para las clases de Delphi. Para lograr esto, el nombre de todas las interfaces se debe agregar separados por comas detrás de la clase base.

Por supuesto, la clase implementadora también debe definir las funciones declaradas por cada una de las 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;
```

Herencia para interfaces

Las interfaces pueden heredarse unas de otras, exactamente igual que las clases. Por lo tanto, una clase implementadora tiene que implementar funciones de la interfaz y todas las interfaces base. De esta manera, sin embargo, el compilador no sabe que la clase implementadora también implementa la interfaz base, solo conoce las interfaces que se enumeran explícitamente. Es por eso que el uso `as ISuperInterface` en `TImplementer` no funcionaría. Eso también resulta en la

práctica común, implementar explícitamente todas las interfaces base, también (en este caso `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;
```

Propiedades en interfaces

Dado que la declaración de variables en interfaces no es posible, no se puede utilizar la forma "rápida" de definir propiedades (`property Value: TObject read FValue write FValue;`). En su lugar, el Getter y el setter (cada uno solo si es necesario) también deben declararse en la interfaz.

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

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

Una cosa que vale la pena mencionar es que la clase implementadora no tiene que declarar la propiedad. El compilador aceptaría este código:

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

Una advertencia, sin embargo, es que de esta manera solo se puede acceder a la propiedad a través de una instancia de la interfaz, y no a través de la propia clase. Además, agregar la propiedad a la clase aumenta la legibilidad.

Lea Interfaces en línea: https://riptutorial.com/es/delphi/topic/4885/interfaces

Capítulo 10: Medición de intervalos de tiempo

Examples

Usando la API de Windows GetTickCount

La función `GetTickCount` API de Windows devuelve el número de milisegundos desde que se inició el sistema (computadora). El ejemplo más simple sigue:

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

Tenga en cuenta que `GetTickCount` devuelve `DWORD` 32 bits por lo que se ajusta cada 49.7 días. Para evitar el `GetTickCount64` , puede usar `GetTickCount64` (disponible desde Windows Vista) o rutinas especiales para calcular la diferencia de ticks:

```
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;
```

De todos modos, estas rutinas devolverán resultados incorrectos si el intervalo de dos llamadas subsiguientes de `GetTickCount` excede el límite de 49.7 días.

Para convertir milisegundos a segundos ejemplo:

```
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;
```

Usando el registro de TStopwatch

Las versiones recientes de Delphi se envían con el registro TStopwatch que es para la medición del intervalo de tiempo. Ejemplo de uso:

```
uses
  System.Diagnostics;

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

Lea Medición de intervalos de tiempo en línea:
https://riptutorial.com/es/delphi/topic/2425/medicion-de-intervalos-de-tiempo

Capítulo 11: Para loops

Sintaxis

- para OrdinalVariable: = LowerOrdinalValue to UpperOrdinalValue comience {loop-body} end;
- para OrdinalVariable: = UpperOrdinalValue down to LowerOrdinalValue do comienza {loop-body} end;
- para EnumerableVariable in Collection, comience {loop-body} end;

Observaciones

- La sintaxis de Delphi `for` bucle no proporciona nada para cambiar la cantidad del paso de `1` a cualquier otro valor.
- Cuando se realiza un bucle con valores ordinales variables, por ejemplo, variables locales de tipo `Integer` , los valores superiores e inferiores se determinarán solo una vez. Los cambios en dichas variables no tendrán ningún efecto en el recuento de iteraciones de los bucles.

Examples

Simple para bucle

Un bucle `for` repite desde el valor inicial hasta el valor final incluido.

```
program SimpleForLoop;

{$APPTYPE CONSOLE}

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

Salida:

1
2
3
4
5
6
7
8

9
10

Bucle sobre los caracteres de una cadena

2005

Lo siguiente itera sobre los caracteres de la cadena `s` . Funciona de manera similar para el bucle sobre los elementos de una matriz o un conjunto, siempre que el tipo de la variable de control de bucle (`c` , en este ejemplo) coincida con el tipo de elemento del valor que se está iterando.

```
program ForLoopOnString;

{$APPTYPE CONSOLE}

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

Salida:

mi
X
una
metro
pag
l
mi

Dirección inversa para bucle

Un bucle `for` repite desde el valor inicial hasta el valor final incluido, como un ejemplo de "cuenta atrás".

```
program CountDown;

{$APPTYPE CONSOLE}

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

Salida:

10
9
8
7
6
5
4
3
2
1
0

Para bucle utilizando una enumeración.

Un bucle `for` iterar a través de elementos en una enumeración

```
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.
```

Salida:

domingo
lunes
martes
miércoles
jueves
viernes
sábado

Para en la matriz

Un bucle `for` itera a través de elementos en una matriz

```
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.
```

Salida:

1,1
2,2
3,3

Lea Para loops en línea: https://riptutorial.com/es/delphi/topic/4643/para-loops

Capítulo 12: Recuperar datos actualizados de TDataSet en un hilo de fondo

Observaciones

Este ejemplo de FireDAC, y los otros que planeo enviar, evitarán el uso de llamadas nativas para abrir el conjunto de datos de forma asíncrona.

Examples

Ejemplo de FireDAC

El ejemplo de código a continuación muestra una forma de recuperar registros de un servidor MSSql en un subproceso en segundo plano utilizando FireDAC. Probado para Delphi 10 Seattle

Como esta escrito:

- El hilo recupera los datos utilizando su propia TFDConnection y TFDQuery y transfiere los datos a la FDQuery del formulario en una llamada a Sychronize ().
- El Ejecutar recupera los datos solo una vez. Podría modificarse para ejecutar la consulta repetidamente en respuesta a un mensaje publicado desde el hilo de VCL.

Código:

```
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.
```

Lea Recuperar datos actualizados de TDataSet en un hilo de fondo en línea: https://riptutorial.com/es/delphi/topic/4114/recuperar-datos-actualizados-de-tdataset-en-un-hilo-de-fondo

Capítulo 13: Usando animaciones en firemonkey

Examples

Trectangle giratorio

1. Crear en blanco la aplicación Multi-Device (Firemonkey).
2. Drop Rectangle on Form.
3. En la ventana del inspector de objetos (F11), encuentre el botón desplegable de RotationAngle y seleccione "Crear nuevo TFloatAnimation".
4. La ventana del inspector de objetos se cambia automáticamente a un nuevo TFloatAnimation, también puede verlo en el menú Estructura (Mayús + Alt).
    - F11).
5. En el inspector de objetos de TFloatAnimation, complete la duración con cualquier número (en segundos). En nuestro caso, tomemos 1. Deje la propiedad StartValue como está, y en el tipo StopValue - 360 (grados, por lo que todo gira). También permite activar la opción Loop (esta animación de bucles hasta que la detenga desde el código).

Ahora tenemos nuestra animación configurada. Todo lo que queda es encenderlo: soltar dos botones en el formulario, llamar primero "Inicio", segundo - "Detener". en el evento OnClick del primer botón escribe:

```
FloatAnimation1.Start;
```

OnClick del segundo código del botón:

```
FloatAnimation1.Stop;
```

Si cambió el nombre de su TFloatAnimation: cámbielo también cuando llame a Iniciar y Detener.

Ahora ejecute su proyecto, haga clic en el botón Inicio y disfrute.

Lea Usando animaciones en firemonkey en línea:
https://riptutorial.com/es/delphi/topic/5383/usando-animaciones-en-firemonkey

Capítulo 14: Usando RTTI en Delphi

Introducción

Delphi proporcionó información de tipo de tiempo de ejecución (RTTI) hace más de una década. Sin embargo, incluso hoy en día, muchos desarrolladores no son plenamente conscientes de sus riesgos y beneficios.

En resumen, la información de tipo de tiempo de ejecución es información sobre el tipo de datos de un objeto que se configura en la memoria en tiempo de ejecución.

RTTI proporciona una manera de determinar si el tipo de un objeto es el de una clase en particular o uno de sus descendientes.

Observaciones

RTTI EN DELPHI - EXPLICADO

La información de tipo de tiempo de ejecución en Delphi: ¿puede hacer algo por usted? El artículo de Brian Long proporciona una excelente introducción a las capacidades RTTI de Delphi. Brian explica que el soporte RTTI en Delphi se agregó primero y principalmente para permitir que el entorno de tiempo de diseño haga su trabajo, pero que los desarrolladores también pueden aprovecharlo para lograr ciertas simplificaciones de código. Este artículo también proporciona una gran visión general de las clases RTTI junto con algunos ejemplos.

Los ejemplos incluyen: leer y escribir propiedades arbitrarias, propiedades comunes sin ancestros comunes, copiar propiedades de un componente a otro, etc.

Examples

Información básica de la clase

Este ejemplo muestra cómo obtener la ascendencia de un componente utilizando las propiedades `ClassType` y `ClassParent` . Utiliza un botón `Button1: TButton` y un cuadro de lista `ListBox1: TListBox` en un formulario `TForm1` .

Cuando el usuario hace clic en el botón, el nombre de la clase del botón y los nombres de las clases principales se agregan al cuadro de lista.

```
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;
```

El cuadro de lista contiene las siguientes cadenas después de que el usuario haga clic en el botón:

- TButton
- Control de TButton
- TWinControl
- TControl
- TComponente
- TPersistente
- Objeto

Lea Usando RTTI en Delphi en línea: https://riptutorial.com/es/delphi/topic/9578/usando-rtti-en-delphi

Capítulo 15: Uso de try, except, y finalmente.

Sintaxis

1. Try-except: try [sentencias] excepto [[[en E: ExceptionType do statement]] [else statement] | [declaraciones] terminan;

   Trate-finalmente: intente [sentencias] finalmente [sentencias] final;

Examples

Simple intento ... finalmente ejemplo para evitar pérdidas de memoria

Utilice `try` - `finally` para evitar pérdidas de recursos (como la memoria) en caso de que ocurra una excepción durante la ejecución.

El siguiente procedimiento guarda una cadena en un archivo y evita que la `TStringList` fugas.

```
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;
```

Independientemente de si se produce una excepción al guardar el archivo, `SL` se liberará. Cualquier excepción irá a la persona que llama.

Devolución segura de excepciones de un nuevo objeto.

Cuando una función *devuelve* un objeto (en lugar de *usar* uno que ha pasado la persona que llama), tenga cuidado de que una excepción no provoque la fuga del objeto.

```
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;
```

Los programadores ingenuos pueden intentar capturar todos los tipos de excepción y devolver `nil` partir de esa función, pero eso es solo un caso especial de la práctica general desalentada de capturar todos los tipos de excepción sin manejarlos.

Try-finalmente anidado dentro de try-except

Un bloque `try` - `finally` puede estar anidado dentro de un `try` - `except` bloque.

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

Si ocurre una excepción dentro de `UseResource` , la ejecución saltará a `ReleaseResource` . Si la excepción es un `EResourceUsageError` , la ejecución saltará al controlador de excepciones y llamará a `HandleResourceErrors` . Las excepciones de cualquier otro tipo omitirán el controlador de excepciones de arriba y aumentarán hasta el siguiente `try` , `except` bloquee la pila de llamadas.

Las excepciones en `AcquireResource` o `ReleaseResource` harán que la ejecución vaya al controlador de excepciones, saltándose el bloque `finally` , ya sea porque el bloque `try` correspondiente aún no se ha ingresado o porque el bloque `finally` *ya se* ha ingresado.

Prueba-excepto anidado dentro de prueba-finalmente

Un bloque `try` - `except` que puede estar anidado dentro de un bloque `try` - `finally` .

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

```
finally
  ReleaseResource;
end;
```

Si se produce un `EResourceUsageError` en `UseResource2` , la ejecución saltará al controlador de excepciones y llamará a `HandleResourceError` . La excepción se considerará manejada, por lo que la ejecución continuará `UseResource3` y luego `ReleaseResource` .

Si ocurre una excepción de cualquier otro tipo en `UseResource2` , entonces el controlador de excepciones que se muestra aquí no se aplicará, por lo que la ejecución saltará sobre la llamada `UseResource3` e irá directamente al bloque `finally` , donde se `ReleaseResource` . Después de eso, la ejecución saltará al siguiente controlador de excepción aplicable, ya que la excepción aumenta la pila de llamadas.

Si se produce una excepción en cualquier otra llamada en el ejemplo anterior, *no* se llamará a `HandleResourceErrors` . Esto se debe a que ninguna de las otras llamadas se produce dentro del bloque `try` correspondiente a ese controlador de excepciones.

Probar-finalmente con 2 o más objetos.

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

Si no inicializa los objetos con `nil` fuera del bloque `try-finally` , si uno de ellos no se crea, se producirá un AV en el bloque finally, porque el objeto no será nulo (ya que no se inicializó) y causará una excepción

El método `Free` comprueba si el objeto es nulo, por lo que al inicializar ambos objetos con `nil` evitan errores al liberarlos si no se crearon.

Lea Uso de try, except, y finalmente. en línea: https://riptutorial.com/es/delphi/topic/3055/uso-de-try--except--y-finalmente-

Creditos

S. No	Capítulos	Contributors
1	Comenzando con Embarcadero Delphi	Charlie H, Community, Dalija Prasnikar, Florian Koch, Jeroen Wiert Pluimers, René Hoffmann, RepeatUntil, Rob Kennedy, Vadim Shakun, w5m, Y.N, Zam
2	Bucles	Y.N
3	Clase TStringList	Charlie H, Fabricio Araujo, Fr0sT, KaiW
4	Creación de cheques de error de tiempo de ejecución fácilmente extraíbles	Alex T
5	Ejecutando otros programas	Dalija Prasnikar
6	Ejecutar un hilo manteniendo GUI sensible	Fr0sT, Jerry Dodge, Johan, kami, LU RD, Marjan Venema
7	Genéricos	Rob Kennedy, Steffen Binas, Uli Gerhardt
8	Instrumentos de cuerda	AlekXL, Dalija Prasnikar, EMBarbosa, Fabricio Araujo, Johan, Radek Hladík, René Hoffmann, RepeatUntil, Rob Kennedy, Rudy Velthuis
9	Interfaces	Florian Koch, Willo van der Merwe
10	Medición de intervalos de tiempo	Fr0sT, John Easley, kludg, Rob Kennedy, Victoria, Wolf
11	Para loops	Filipe Martins, Jeroen Wiert Pluimers, John Easley, René Hoffmann, Rob Kennedy, Siendor, Y.N
12	Recuperar datos actualizados de TDataSet en un hilo de fondo	MartynA
13	Usando animaciones en firemonkey	Alexander Petrosyan
14	Usando RTTI en	Petzy, René Hoffmann

	Delphi	
15	Uso de try, except, y finalmente.	EMBarbosa, Fabio Gomes, Johan, MrE, Nick Hodges, Rob Kennedy, Shadow