eBook Gratuit

# APPRENEZ Embarcadero Delphi

eBook gratuit non affilié créé à partir des **contributeurs de Stack Overflow.**

#delphi

# Table des matières

# À propos

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

# Chapitre 1: Démarrer avec Embarcadero Delphi

## Remarques

Delphi est un langage général basé sur un dialecte Object Pascal dont les racines proviennent de Borland Turbo Pascal. Il est livré avec son propre IDE conçu pour prendre en charge le développement rapide d'applications (RAD).

Il permet le développement d'applications natives (compilées) entre plates-formes à partir d'une base de code unique. Les plateformes actuellement prises en charge sont Windows, OSX, iOS et Android.

Il est livré avec deux cadres visuels:

- VCL: Visual Component Library spécialement conçue pour le développement Windows intégrant les contrôles Windows natifs et la prise en charge de la création de contrôles personnalisés.
- FMX: framework multiplate-forme FireMonkey pour toutes les plates-formes prises en charge

## Versions

| Version | Version numérique | Nom du produit | Date de sortie |
|---|---|---|---|
| 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 pour .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 |

| Version | Version numérique | Nom du produit | Date de sortie |
|---|---|---|---|
| 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 |
| XE8 | 22.0 | Embarcadero RAD Studio XE8 | 2015-04-07 |
| 10 Seattle | 23.0 | Embarcadero RAD Studio 10 Seattle | 2015-08-31 |
| 10.1 Berlin | 24.0 | Embarcadero RAD Studio 10.1 Berlin | 2016-04-20 |
| 10,2 Tokyo | 25,0 | Embarcadero RAD Studio 10.2 Tokyo | 2017-03-22 |

# Examples

## Bonjour le monde

Ce programme, enregistré dans un fichier nommé *HelloWorld.dpr* , compile en une application console qui affiche "Hello World" sur la console:

```
program HelloWorld;

{$APPTYPE CONSOLE}

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

## Montrer 'Hello World' en utilisant la VCL

Ce programme utilise VCL, la bibliothèque de composants d'interface utilisateur par défaut de Delphi, pour imprimer "Hello World" dans une boîte de message. La VCL enveloppe la plupart des composants WinAPI couramment utilisés. De cette façon, ils peuvent être utilisés beaucoup plus facilement, par exemple sans avoir à travailler avec des poignées de fenêtre.

Pour inclure une dépendance (comme `Vcl.Dialogs` dans ce cas), ajoutez le bloc `uses` , y compris une liste d'unités séparées par des virgules et se terminant par un point-virgule.

```
program HelloWindows;

uses
  Vcl.Dialogs;

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

## Montrer 'Hello World' en utilisant WinAPI MessageBox

Ce programme utilise l'API Windows (WinAPI) pour imprimer "Hello World" dans une boîte de message.

Pour inclure une dépendance (comme `Windows` dans ce cas), ajoutez le bloc uses, y compris une liste d'unités séparées par des virgules se terminant par un point-virgule.

```
program HelloWorld;

uses
  Windows;

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

## Cross-platform Hello World en utilisant FireMonkey

XE2

```
program CrossPlatformHelloWorld;

uses
  FMX.Dialogs;

{$R *.res}

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

La plupart des plates-formes prises en charge par Delphi (Win32 / Win64 / OSX32 / Android32 / iOS32 / iOS64) prennent également en charge une console, de sorte que l'exemple `WriteLn` leur convient bien.

Pour les plates-formes nécessitant une interface graphique (tout appareil iOS et certains appareils Android), l'exemple de FireMonkey ci-dessus fonctionne bien.

Lire Démarrer avec Embarcadero Delphi en ligne:
https://riptutorial.com/fr/delphi/topic/599/demarrer-avec-embarcadero-delphi

# Chapitre 2: Boucles

## Introduction

Le langage Delphi fournit 3 types de boucle

`for` - itérateur pour séquence fixe sur entier, chaîne, tableau ou énumération

`repeat-until` qu `repeat-until` - condition de sortie vérifie après chaque tour, boucle en cours d'exécution au moins une fois

`while do` condition `while do` - do est vérifiée avant chaque tour, la boucle ne peut jamais être exécutée

## Syntaxe

- pour OrdinalVariable: = LowerOrdinalValue to UpperOrdinalValue ne commence {loop-body} end;
- pour OrdinalVariable: = UpperOrdinalValue downOrdinalValue down commence {loop-body} end;
- pour EnumerableVariable dans Collection, commencez {loop-body} end;
- répétez {loop-body} jusqu'à {break-condition};
- while {condition} commence {loop-body} end;

## Examples

### Pause et continuer dans les boucles

```
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.
```

**Sortie:**

1
3

4
5
6
7
Terminer.

## Répète jusqu'à

```
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.
```

Cet exemple court d'impression sur la console `Type a words to echo. Enter an empty string to exit.` , attendez le type d'utilisateur, renvoyez-le en écho et attendez à nouveau l'entrée en boucle infinie - jusqu'à ce que l'utilisateur entre la chaîne vide.

## Tout faire

```
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.
```

Cet exemple imprime pour `WhileEOF.dpr` contenu `WhileEOF.dpr` fichier `WhileEOF.dpr` en utilisant la condition `While not(EOF)` . Si le fichier est vide, la boucle `ReadLn-WriteLn` n'est pas exécutée.

Lire Boucles en ligne: https://riptutorial.com/fr/delphi/topic/9931/boucles

# Chapitre 3: Classe TStringList

## Examples

### introduction

TStringList est un descendant de la classe TStrings de la VCL. TStringList peut être utilisé pour stocker et manipuler la liste des chaînes. Bien qu'initialement prévu pour les chaînes, tout type d'objet peut également être manipulé à l'aide de cette classe.

TStringList est largement utilisé dans VCL lorsque l'objectif est de maintenir une liste de chaînes. TStringList prend en charge un ensemble complet de méthodes offrant un haut niveau de personnalisation et une facilité de manipulation.

L'exemple suivant illustre la création, l'ajout de chaînes, le tri, la récupération et la libération d'un objet 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 possède une variété de cas d'utilisateurs, y compris la manipulation de chaînes, le tri, l'indexation, l'appariement des valeurs-clés et la séparation des délimiteurs.

### Appariement valeur-clé

Vous pouvez utiliser une TStringList pour stocker des paires clé-valeur. Cela peut être utile si vous souhaitez stocker des paramètres, par exemple. Un paramètre se compose d'une clé (l'identifiant

du paramètre) et de la valeur. Chaque paire valeur-clé est stockée sur une ligne du format StringList dans le format 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;
```

Dans cet exemple, la liste de chaînes contient le contenu suivant avant sa destruction:

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

### Note sur la performance

Sous le capot, `TStringList` effectue une recherche de clé en effectuant une boucle droite sur tous les éléments, en recherchant un séparateur dans chaque élément et en comparant la partie du nom à la clé donnée. Nul besoin de dire que cela a un impact énorme sur les performances, ce mécanisme ne doit donc être utilisé que dans des endroits peu critiques et rarement répétés. Dans les cas où les performances sont importantes, il faut utiliser `TDictionary<TKey,TValue>` de `System.Generics.Collections` qui implémente la recherche de table de hachage ou conserver les clés dans `TStringList` **trié** avec des valeurs stockées comme `Object` -s utilisant un algorithme de recherche binaire.

Lire Classe TStringList en ligne: https://riptutorial.com/fr/delphi/topic/6045/classe-tstringlist

# Chapitre 4: Cordes

## Examples

### Types de chaînes

Delphi a les types de chaînes suivants (par ordre de popularité):

| Type | Longueur maximale | Taille minimum | La description |
|---|---|---|---|
| `string` | 2 Go | 16 octets | Une chaîne gérée. Un alias pour `AnsiString` via Delphi 2007 et un alias pour `UnicodeString` partir de Delphi 2009. |
| `UnicodeString` | 2 Go | 16 octets | Une chaîne gérée au format UTF-16. |
| `AnsiString` | 2 Go | 16 octets | Une chaîne gérée au format ANSI pré-Unicode. À partir de Delphi 2009, il comporte un indicateur de page de code explicite. |
| `UTF8String` | 2 Go | 16 octets | Une chaîne gérée au format UTF-8, implémentée en tant que `AnsiString` avec une page de codes UTF-8. |
| `ShortString` | 255 caractères | 2 octets | Une chaîne héritée, de longueur fixe, non gérée avec très peu de charge |
| `WideString` | 2 Go | 4 octets | Destiné à l'interopérabilité COM, une chaîne gérée au format UTF-16. Equivalent au type Windows `BSTR` . |

`UnicodeString` et `AnsiString` sont des références comptées et des copies sur écriture (COW). `ShortString` et `WideString` ne sont pas comptés par référence et n'ont pas de sémantique COW.

### Cordes

```
uses
  System.Character;

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

## Chars

2009

```
uses
  Character;

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

La clause `uses` doit être `System.Character` si la version est XE2 ou supérieure.

## Majuscule et minuscule

```
uses
  SysUtils;

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

## Affectation

Assigner une chaîne à différents types de chaînes et comment l'environnement d'exécution se comporte à leur égard. Allocation de mémoire, comptage de références, accès indexé aux caractères et erreurs de compilation décrites brièvement, le cas échéant.

```
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 }
```

## Comptage de référence

Le comptage des références sur les chaînes est sécurisé pour les threads. Les verrous et les gestionnaires d'exceptions sont utilisés pour protéger le processus. Considérez le code suivant, avec des commentaires indiquant où le compilateur insère du code au moment de la compilation pour gérer les comptes de référence:

```
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.
```

Comme indiqué ci-dessus, l'introduction d'une chaîne locale temporaire pour contenir les modifications apportées à un paramètre implique la même surcharge que pour apporter des modifications directement à ce paramètre. Déclarer une chaîne `const` n'évite que le comptage de références lorsque le paramètre de chaîne est réellement en lecture seule. Cependant, pour éviter toute fuite de détails d'implémentation en dehors d'une fonction, il est conseillé de toujours utiliser l'un des paramètres `const` , `var` ou `out` on string.

## Encodages

Les types de chaînes comme UnicodeString, AnsiString, WideString et UTF8String sont stockés dans une mémoire à l'aide de leur codage respectif (voir Types de chaînes pour plus de détails). L'affectation d'un type de chaîne à une autre peut entraîner une conversion. Type string est conçu pour encoder indépendamment - vous ne devriez jamais utiliser sa représentation interne.

La classe `Sysutils.TEncoding` fournit la méthode `GetBytes` pour convertir la `string` en `TBytes` (tableau d'octets) et `GetString` pour convertir `TBytes` en `string` . La classe `Sysutils.TEncoding` fournit également de nombreux encodages prédéfinis en tant que propriétés de classe.

Une façon de traiter les encodages consiste à utiliser uniquement `string` type de `string` dans votre application et à utiliser `TEncoding` chaque fois que vous devez utiliser un codage spécifique - généralement dans les opérations d'E / S, les appels 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;
```

Lire Cordes en ligne: https://riptutorial.com/fr/delphi/topic/3957/cordes

# Chapitre 5: Création de vérifications d'erreur d'exécution facilement amovibles

## Introduction

Cela montre comment une routine de vérification des erreurs d'exécution de votre propre fabrication peut être facilement incorporée afin de ne pas générer de surcharge de code lorsqu'elle est désactivée.

## Examples

### Exemple 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;
```

Le concept est essentiellement celui-ci:

Le symbole défini est utilisé pour activer l'utilisation du code. Cela empêche également le code d'être explicitement intégré, ce qui signifie qu'il est plus facile de placer un point d'arrêt dans la routine de vérification.

Cependant, la vraie beauté de cette construction est que vous *ne* voulez plus le chèque. En commentant le ***$ define*** (mettre « // » en face de celui - ci) , vous non seulement supprimer le code de vérification, mais vous pourrez aussi *passer sur* la ***ligne*** pour la routine et de supprimer ainsi les frais généraux de tous les endroits où vous avez appelé la routine! Le compilateur supprimera entièrement toutes les traces de votre vérification (en supposant que l'inclinaison proprement dite soit bien sûr définie sur "On" ou "Auto").

L'exemple ci-dessus est essentiellement similaire au concept des "assertions", et votre première ligne pourrait définir le résultat sur TRUE ou FALSE en fonction de l'utilisation.

Mais vous êtes maintenant également libre d'utiliser cette méthode de construction pour le code qui fait de la trace, de la métrique, etc. Par exemple:

```
   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.
```

Lire Création de vérifications d'erreur d'exécution facilement amovibles en ligne: https://riptutorial.com/fr/delphi/topic/10541/creation-de-verifications-d-erreur-d-execution-facilement-amovibles

# Chapitre 6: Exécuter d'autres programmes

## Examples

### CreateProcess

La fonction suivante encapsule le code d'utilisation de l'API Windows `CreateProcess` pour lancer d'autres programmes.

Il est configurable et peut attendre que le processus d'appel se termine ou qu'il revienne immédiatement.

Paramètres:

- `FileName` - chemin d'accès complet à l'exécutable
- `Params` - paramètres de ligne de commande ou utiliser une chaîne vide
- `Folder` - dossier de travail pour le programme appelé - si le chemin vide sera extrait de `FileName`
- `WaitUntilTerminated` - si la fonction true attendra que le processus termine son exécution
- `WaitUntilIdle` - si la fonction true appelle la fonction WaitForInputIdle et attend que le processus spécifié ait fini de traiter son entrée initiale et jusqu'à ce qu'il n'y ait plus de saisie utilisateur en attente
- `RunMinimized` - si le vrai processus sera exécuté minimisé
- `ErrorCode` - si la fonction échoue, cela contiendra le code d'erreur Windows rencontré

```
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;
```

### Utilisation de la fonction ci-dessus

```
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;
```

Documentation CreateProcess

Lire Exécuter d'autres programmes en ligne: https://riptutorial.com/fr/delphi/topic/5180/executer-d-autres-programmes

# Chapitre 7: Exécuter un thread tout en gardant l'interface graphique réactive

## Examples

### Interface graphique réactive utilisant des threads pour le travail en arrière-plan et PostMessage pour générer des rapports à partir des threads

Garder une interface graphique réactive lors de l'exécution d'un processus long nécessite soit des "callbacks" très élaborés pour permettre à l'interface graphique de traiter sa file d'attente de messages, soit l'utilisation de threads (en arrière-plan) (worker).

Lancer n'importe quel nombre de threads pour faire un travail n'est généralement pas un problème. Le plaisir commence lorsque vous souhaitez que l'interface graphique affiche les résultats intermédiaires et finaux ou que vous décriviez les progrès.

L'affichage de tout élément dans l'interface graphique nécessite une interaction avec les contrôles et / ou la file d'attente / pompe de messages. Cela devrait toujours être fait dans le contexte du thread principal. Jamais dans le contexte d'un autre fil.

Il y a plusieurs façons de gérer cela.

Cet exemple montre comment vous pouvez le faire en utilisant des threads simples, permettant à l'interface graphique d'accéder à l'instance de thread une fois celle-ci terminée en définissant `FreeOnTerminate` sur `false` et en signalant qu'un thread est "terminé" en utilisant `PostMessage` .

Remarques sur les conditions de course: Les références aux threads de travail sont conservées dans un tableau du formulaire. Lorsqu'un thread est terminé, la référence correspondante dans le tableau devient nil-ed.

C'est une source potentielle de conditions de course. Tout comme l'utilisation d'un booléen "Running" pour déterminer plus facilement s'il reste des threads à terminer.

Vous devrez décider si vous devez protéger ces ressources à l'aide de verrous ou non.

Dans cet exemple, tel qu'il est, il n'y a pas besoin. Ils ne sont modifiés qu'à deux emplacements: la méthode `StartThreads` et la méthode `HandleThreadResults` . Les deux méthodes ne fonctionnent que dans le contexte du thread principal. Tant que vous continuez ainsi et que vous ne commencez pas à appeler ces méthodes dans le contexte de threads différents, il leur est impossible de créer des conditions de course.

## Fil

```
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;
```

Le constructeur définit simplement les membres privés et définit FreeOnTerminate sur False. Ceci est essentiel car cela permettra au thread principal d'interroger l'instance du thread pour connaître son résultat.

La méthode execute effectue son calcul, puis publie un message dans le descripteur reçu dans son constructeur pour indiquer que c'est fait:

```
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;
```

L'utilisation de `PostMessage` est essentielle dans cet exemple. `PostMessage` "just" place un message dans la file d'attente de la pompe de messages du thread principal et n'attend pas qu'il soit traité. C'est de nature asynchrone. Si vous deviez utiliser `SendMessage` vous vous `SendMessage` dans un cornichon. `SendMessage` place le message dans la file d'attente et attend qu'il soit traité. En bref, c'est synchrone.

Les déclarations pour le message personnalisé UM_WORKERDONE sont déclarées comme suit:

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

Le const `UM_WORKERDONE` utilise `WM_APP` comme point de départ pour sa valeur afin de s'assurer qu'il n'interfère pas avec les valeurs utilisées par Windows ou la VCL Delphi (comme recommandé par

MicroSoft).

## Forme

Toute forme peut être utilisée pour démarrer des threads. Il vous suffit d'y ajouter les membres suivants:

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

Oh, et l'exemple de code suppose l'existence d'un `Memo1: TMemo;` dans les déclarations du formulaire, qu'il utilise pour "journalisation et reporting".

Le `FRunning` peut être utilisé pour empêcher que l'interface graphique ne commence à être cliquée pendant le travail. `FThreads` est utilisé pour contenir le pointeur d'instance et le handle des threads créés.

La procédure pour démarrer les threads a une implémentation assez simple. Il commence par vérifier si un ensemble de threads est déjà en attente. Si c'est le cas, il ne fait que sortir. Si ce n'est pas le cas, il attribue la valeur true à l'indicateur et lance les threads en fournissant à chacun son propre handle afin qu'il sache où publier son message "done".

```
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;
```

Le handle du thread est également placé dans le tableau car c'est ce que nous recevons dans les messages qui nous indiquent qu'un thread est terminé et qu'il est plus facile à accéder en dehors de l'instance du thread. Avoir le handle disponible en dehors de l'instance du thread nous permet également d'utiliser `FreeOnTerminate` défini sur `True` si nous n'avions pas besoin de l'instance pour obtenir ses résultats (par exemple s'ils avaient été stockés dans une base de données). Dans ce cas, il ne serait bien sûr pas nécessaire de garder une référence à l'instance.

Le plaisir réside dans l'implémentation 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;
```

Cette méthode recherche d'abord le thread en utilisant le handle reçu dans le message. Si une correspondance a été trouvée, elle récupère et rapporte le résultat du thread en utilisant l'instance ( `FreeOnTerminate` était `False` , souvenez-vous?), Puis se termine: libérer l'instance et définir à la fois la référence d'instance et le handle sur nil, indiquant que ce thread n'est pas plus pertinent.

Enfin, il vérifie si l'un des threads est toujours en cours d'exécution. Si aucun n'est trouvé, "all done" est signalé et l'indicateur `FRunning` défini sur `False` afin qu'un nouveau lot de travail puisse être démarré.

Lire Exécuter un thread tout en gardant l'interface graphique réactive en ligne: https://riptutorial.com/fr/delphi/topic/1796/executer-un-thread-tout-en-gardant-l-interface-graphique-reactive

# Chapitre 8: Génériques

## Examples

### Trier un tableau dynamique via TArray.Sort générique

```
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
));
```

### Utilisation 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;
```

### Descendant de TList le rendant spécifique

```
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;
```

## Trier un TList

```
var List: TList<TDateTime>;

...

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

Lire Génériques en ligne: https://riptutorial.com/fr/delphi/topic/4054/generiques

# Chapitre 9: Interfaces

## Remarques

Les interfaces sont utilisées pour décrire les informations nécessaires et la sortie attendue des méthodes et des classes, sans fournir d'informations sur l'implémentation explicite.

Les classes peuvent **implémenter des** interfaces et les interfaces peuvent **hériter les** unes des autres. Si une classe **implémente** une interface, cela signifie que toutes les fonctions et procédures exposées par l'interface existent dans la classe.

Un aspect particulier des interfaces dans delphi est que les instances d'interfaces ont une gestion de la durée de vie basée sur le comptage des références. La durée de vie des instances de classe doit être gérée manuellement.

Compte tenu de tous ces aspects, des interfaces peuvent être utilisées pour atteindre différents objectifs:

- Fournir plusieurs implémentations différentes pour les opérations (par exemple enregistrer dans un fichier, une base de données ou envoyer un courrier électronique, le tout en tant qu'interface "SaveData")
- Réduire les dépendances, améliorer le découplage et ainsi rendre le code plus facile à maintenir et à tester
- Travaillez avec des instances dans plusieurs unités sans être gêné par la gestion de la durée de vie (même si des pièges existent, faites attention!)

## Examples

### Définition et implémentation d'une interface

Une interface est déclarée comme une classe, mais sans modificateurs d'accès ( `public` , `private` , ...). De plus, aucune définition n'est autorisée, donc les variables et les constantes ne peuvent pas être utilisées.

Les interfaces doivent toujours avoir un *identifiant unique* , qui peut être généré en appuyant sur `Ctrl` + `Maj` + `G`.

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

Pour implémenter une interface, le nom de l'interface doit être ajouté derrière la classe de base. En outre, la classe doit être un descendant de `TInterfacedObject` (cela est important pour la *gestion* de la *durée de vie* ).

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

Lorsqu'une classe implémente une interface, elle doit inclure toutes les méthodes et fonctions déclarées dans l'interface, sinon elle ne sera pas compilée.

Une chose à noter est que les modificateurs d'accès n'ont aucune influence si l'appelant travaille avec l'interface. Par exemple, toutes les fonctions de l'interface peuvent être implémentées en tant que membres `strict private` , mais peuvent toujours être appelées à partir d'une autre classe si une instance de l'interface est utilisée.

## Implémentation de plusieurs interfaces

Les classes peuvent implémenter plusieurs interfaces, par opposition à l'héritage de plusieurs classes ( *Multiple Inheritance* ), ce qui n'est pas possible pour les classes Delphi. Pour ce faire, le nom de toutes les interfaces doit être ajouté à la classe de base.

Bien entendu, la classe d'implémentation doit également définir les fonctions déclarées par chacune des 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;
```

## Héritage des interfaces

Les interfaces peuvent hériter les unes des autres, exactement comme les classes aussi. Une classe d'implémentation doit donc implémenter des fonctions de l'interface et de toutes les interfaces de base. De cette façon, cependant, le compilateur ne sait pas que la classe implicite implémente également l'interface de base, elle ne connaît que les interfaces explicitement

répertoriées. C'est pourquoi l'utilisation `as ISuperInterface` sur `TImplementer` ne fonctionnerait pas. Cela se traduit également par la pratique courante d'implémenter explicitement toutes les interfaces de base (dans ce cas, `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;
```

## Propriétés dans les interfaces

La déclaration de variables dans les interfaces n'étant pas possible, la manière "rapide" de définir les `property Value: TObject read FValue write FValue;` ( `property Value: TObject read FValue write FValue;` ) ne peut pas être utilisée. Au lieu de cela, le Getter et le setter (chacun seulement si nécessaire) doivent également être déclarés dans l'interface.

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

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

Une chose à noter est que la classe d'implémentation n'a pas à déclarer la propriété. Le compilateur accepterait ce code:

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

Une mise en garde, cependant, est que de cette manière, la propriété ne peut être accédée que via une instance de l'interface, non pas à travers la classe elle-même. De plus, l'ajout de la propriété à la classe augmente la lisibilité.

Lire Interfaces en ligne: https://riptutorial.com/fr/delphi/topic/4885/interfaces

# Chapitre 10: Mesure des intervalles de temps

## Examples

### Utilisation de l'API Windows GetTickCount

La fonction `GetTickCount` API Windows renvoie le nombre de millisecondes depuis le démarrage du système (ordinateur). L'exemple le plus simple est le suivant:

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

Notez que `GetTickCount` renvoie 32 bits `DWORD` afin qu'il encapsule tous les 49,7 jours. Pour éviter l'encapsulation, vous pouvez utiliser `GetTickCount64` (disponible depuis Windows Vista) ou des routines spéciales pour calculer la différence de `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;
```

Quoi qu'il en soit, ces routines `GetTickCount` des résultats incorrects si l'intervalle de deux appels suivants de `GetTickCount` dépasse la limite de 49,7 jours.

Pour convertir des millisecondes en secondes exemple:

```
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;
```

### Utilisation de l'enregistrement TStopwatch

Les versions récentes de Delphi sont livrées avec l'enregistrement TStopwatch qui sert à mesurer l'intervalle de temps. Exemple d'utilisation:

```
uses
  System.Diagnostics;

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

Lire Mesure des intervalles de temps en ligne: https://riptutorial.com/fr/delphi/topic/2425/mesure-des-intervalles-de-temps

# Chapitre 11: Pour les boucles

## Syntaxe

- pour OrdinalVariable: = LowerOrdinalValue to UpperOrdinalValue ne commence {loop-body} end;
- pour OrdinalVariable: = UpperOrdinalValue downOrdinalValue down commence {loop-body} end;
- pour EnumerableVariable dans Collection, commencez {loop-body} end;

## Remarques

- La syntaxe de Delphi `for` -loop ne fournit rien pour modifier le montant de l'étape de `1` à toute autre valeur.
- Lors de la mise en boucle avec des valeurs ordinales variables, par exemple des variables locales de type `Integer` , les valeurs supérieure et inférieure seront déterminées une seule fois. Les modifications apportées à ces variables n'auront aucun effet sur le nombre d'itérations des boucles.

## Examples

### Simple pour la boucle

Une boucle `for` itère de la valeur de départ à la valeur finale incluse.

```
program SimpleForLoop;

{$APPTYPE CONSOLE}

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

**Sortie:**

1
2
3
4
5
6
7
8

9
dix

## En boucle sur les caractères d'une chaîne

2005

Ce qui suit itère sur les caractères de la chaîne `s` . Il fonctionne de la même façon pour la lecture en boucle des éléments d'un tableau ou d'un ensemble, à condition que le type de la variable de contrôle de boucle ( `c` , dans cet exemple) corresponde au type d'élément de la valeur itérée.

```
program ForLoopOnString;

{$APPTYPE CONSOLE}

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

**Sortie:**

E
X
une
m
p
l
e

## Sens inverse pour la boucle

Une boucle `for` une itération de la valeur initiale à la valeur finale incluse, comme exemple de décompte.

```
program CountDown;

{$APPTYPE CONSOLE}

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

**Sortie:**

dix
9
8
7
6
5
4
3
2
1
0

## Pour une boucle utilisant une énumération

A `for` itérer en boucle à travers les articles dans une énumération

```
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.
```

**Sortie:**

dimanche
Lundi
Mardi
Mercredi
Jeudi
Vendredi
samedi

## Pour en tableau

Une boucle `for` itérer les éléments d'un tableau

```
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.
```

**Sortie:**

1,1
2,2
3,3

Lire Pour les boucles en ligne: https://riptutorial.com/fr/delphi/topic/4643/pour-les-boucles

# Chapitre 12: Récupération des données TDataSet mises à jour dans un thread d'arrière-plan

## Remarques

Cet exemple FireDAC, ainsi que les autres que je compte soumettre, éviteront l'utilisation d'appels natifs pour ouvrir de manière asynchrone le jeu de données.

## Examples

### Exemple FireDAC

L'exemple de code ci-dessous montre un moyen d'extraire des enregistrements d'un serveur MSSql dans un thread d'arrière-plan à l'aide de FireDAC. Testé pour Delphi 10 Seattle

Comme écrit:

- Le thread récupère les données en utilisant ses propres TFDConnection et TFDQuery et transfère les données dans le FDQuery du formulaire dans un appel à Sychronize ().
- L'exécution récupère les données une seule fois. Il pourrait être modifié pour exécuter la requête à plusieurs reprises en réponse à un message publié à partir du thread VCL.

Code:

```
type
  TForm1 = class;

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

TForm1 = class(TForm)
  FDConnection1: TFDConnection;
  FDQuery1: TFDQuery;
  DataSource1: TDataSource;
```

```
  DBGrid1: TDBGrid;
  DBNavigator1: TDBNavigator;
  Button1: TButton;
  procedure FormDestroy(Sender: TObject);
  procedure Button1Click(Sender: TObject);
  procedure FormCreate(Sender: TObject);
private
public
  QueryThread : TFDQueryThread;
end;

var
Form1: TForm1;

implementation

{$R *.dfm}

{ TFDQueryThread }

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

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

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

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

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

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

```
  QueryThread.Free;
end;

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

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

end.
```

Lire Récupération des données TDataSet mises à jour dans un thread d'arrière-plan en ligne: https://riptutorial.com/fr/delphi/topic/4114/recuperation-des-donnees-tdataset-mises-a-jour-dans-un-thread-d-arriere-plan

# Chapitre 13: Utilisation d'animations dans Firemonkey

## Examples

### TRectangle rotatif

1. Créer une application Multi-Device (Firemonkey) vierge.
2. Déposez le rectangle sur la forme.
3. Dans la fenêtre de l'inspecteur d'objets (F11), sélectionnez RotationAngle, cliquez sur le bouton déroulant et sélectionnez "Créer une nouvelle TFloatAnimation".
4. La fenêtre de l'inspecteur d'objets est automatiquement remplacée par une nouvelle TFloatAnimation ajoutée, vous pouvez également l'afficher dans le menu Structure (Maj + Alt
    - F11).
5. Dans l'inspecteur d'objets de TFloatAnimation, remplissez la durée avec n'importe quel nombre (en secondes). Dans notre cas, prenons 1. Laissez la propriété StartValue en l'état, et dans le type StopValue - 360 (Degrés, donc tout tourne). Permet également d'activer l'option Boucle (l'animation en boucle jusqu'à ce que vous l'arrêtiez à partir du code).

Maintenant, nous avons notre animation mise en place. Il ne reste plus qu'à l'activer: Déposez deux boutons sur le formulaire, appelez le premier "Start", le second - "Stop". dans l'événement OnClick du premier bouton écrire:

```
FloatAnimation1.Start;
```

OnClick du deuxième code du bouton:

```
FloatAnimation1.Stop;
```

Si vous avez changé le nom de votre TFloatAnimation - Modifiez-le également lorsque vous appelez Start and Stop.

Maintenant, lancez votre projet, cliquez sur le bouton Démarrer et profitez-en.

Lire Utilisation d'animations dans Firemonkey en ligne:
https://riptutorial.com/fr/delphi/topic/5383/utilisation-d-animations-dans-firemonkey

# Chapitre 14: Utilisation de try, sauf et enfin

## Syntaxe

1. Try-except: try [instructions] sauf [[[sur E: ExceptionType]]] [else instruction] | [déclarations] se terminent;

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

## Examples

### Simple try..finally exemple pour éviter les fuites de mémoire

Utilisez `try` - `finally` pour éviter les fuites de ressources (telles que la mémoire) en cas d'exception lors de l'exécution.

La procédure ci-dessous enregistre une chaîne dans un fichier et empêche la `TStringList` de `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;
```

Qu'une exception se produise lors de la sauvegarde du fichier, `SL` sera libéré. Toute exception ira à l'appelant.

### Retour d'exception-sécurité d'un nouvel objet

Lorsqu'une fonction *retourne* un objet (par opposition à l' *utilisation d'* un objet transmis par l'appelant), faites attention qu'une exception ne provoque pas de fuite de l'objet.

```
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;
```

Les programmeurs naïfs peuvent tenter d'attraper tous les types d'exception et renvoyer `nil` partir d'une telle fonction, mais ce n'est là qu'un cas particulier de la pratique générale découragée d'attraper tous les types d'exception sans les manipuler.

## Essayez-enfin imbriqué dans try-except

Un bloc `try` - `finally` peut être imbriqué dans un bloc `try` - `except` .

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

Si une exception se produit dans `UseResource` , l'exécution passera à `ReleaseResource` . Si l'exception est une `EResourceUsageError` , l'exécution passe alors au gestionnaire d'exceptions et appelle `HandleResourceErrors` . Les exceptions de tout autre type ignoreront le gestionnaire d'exceptions ci-dessus et remonteront jusqu'au prochain `try` - `except` bloquer la pile d'appels.

Les exceptions à `AcquireResource` ou `ReleaseResource` entraînera l' exécution d'aller au gestionnaire d'exception, sauter le `finally` bloc, soit parce que le correspondant `try` bloc n'a pas encore été entré ou parce que le `finally` bloc a *déjà* été saisi.

## Essayez, sauf imbriqué dans try-finally

Un bloc `try` - `except` peut être imbriqué dans un bloc `try` - `finally` .

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

```
  UseResource3;
finally
  ReleaseResource;
end;
```

Si une `EResourceUsageError` se produit dans `UseResource2` , l'exécution passe alors au gestionnaire d'exceptions et appelle `HandleResourceError` . L'exception sera considérée comme gérée, donc l'exécution continuera à `UseResource3` , puis `ReleaseResource` .

Si une exception d'un autre type se produit dans `UseResource2` , alors le gestionnaire d'exception `UseResource2` ici ne s'appliquera pas, l'exécution passera par-dessus l'appel `UseResource3` et ira directement au bloc `finally` , où `ReleaseResource` sera appelé. Après cela, l'exécution passera au gestionnaire d'exceptions suivant lors de la création de l'exception.

Si une exception se produit dans un autre appel dans l'exemple ci-dessus, `HandleResourceErrors` *ne* sera *pas* appelé. En effet, aucun des autres appels ne se produit dans le bloc `try` correspondant à ce gestionnaire d'exceptions.

### Essayez-enfin avec 2 objets ou plus

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

Si vous n'initialisez pas les objets avec `nil` dehors du bloc `try-finally` , si l'un d'eux ne peut pas être créé, un AV se produira sur le bloc finally, car l'objet ne sera pas nul (car il n'a pas été initialisé) et provoquera une exception.

La méthode `Free` vérifie si l'objet est nul, l'initialisation des deux objets avec `nil` évite les erreurs lors de leur libération s'ils n'ont pas été créés.

Lire Utilisation de try, sauf et enfin en ligne: https://riptutorial.com/fr/delphi/topic/3055/utilisation-de-try--sauf-et-enfin

# Chapitre 15: Utiliser RTTI dans Delphi

## Introduction

Delphi a fourni des informations de type d'exécution (RTTI) il y a plus de dix ans. Pourtant, même aujourd'hui, de nombreux développeurs ne sont pas pleinement conscients de ses risques et de ses avantages.

En bref, les informations sur le type d'exécution sont des informations sur le type de données d'un objet qui est défini en mémoire au moment de l'exécution.

RTTI permet de déterminer si le type d'un objet est celui d'une classe particulière ou de l'un de ses descendants.

## Remarques

RTTI IN DELPHI - EXPLIQUÉ

L' information sur le type d'exécution dans Delphi - Peut-il faire quelque chose pour vous? L'article de Brian Long constitue une excellente introduction aux fonctionnalités RTTI de Delphi. Brian explique que la prise en charge de RTTI dans Delphi a été ajoutée avant tout pour permettre à l'environnement de conception de faire son travail, mais que les développeurs peuvent également en tirer parti pour obtenir certaines simplifications de code. Cet article fournit également un excellent aperçu des classes RTTI avec quelques exemples.

Exemples: lecture et écriture de propriétés arbitraires, propriétés communes sans ancêtre commun, copie des propriétés d'un composant à un autre, etc.

## Examples

### Informations de base sur la classe

Cet exemple montre comment obtenir l'ascendance d'un composant à l'aide des propriétés `ClassType` et `ClassParent` . Il utilise un bouton `Button1: TButton` et une zone de liste `ListBox1: TListBox` sur un formulaire `TForm1` .

Lorsque l'utilisateur clique sur le bouton, le nom de la classe du bouton et les noms de ses classes parentes sont ajoutés à la liste.

```
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;
```

La zone de liste contient les chaînes suivantes après que l'utilisateur clique sur le bouton:

- TButton
- TButtonControl
- TWinControl
- TControl
- Composant
- TPersistent
- TObject

Lire Utiliser RTTI dans Delphi en ligne: https://riptutorial.com/fr/delphi/topic/9578/utiliser-rtti-dans-delphi

# Crédits

| S. No | Chapitres | Contributeurs |
|---|---|---|
| 1 | Démarrer avec Embarcadero Delphi | Charlie H, Community, Dalija Prasnikar, Florian Koch, Jeroen Wiert Pluimers, René Hoffmann, RepeatUntil, Rob Kennedy, Vadim Shakun, w5m, Y.N, Zam |
| 2 | Boucles | Y.N |
| 3 | Classe TStringList | Charlie H, Fabricio Araujo, Fr0sT, KaiW |
| 4 | Cordes | AlekXL, Dalija Prasnikar, EMBarbosa, Fabricio Araujo, Johan, Radek Hladík, René Hoffmann, RepeatUntil, Rob Kennedy, Rudy Velthuis |
| 5 | Création de vérifications d'erreur d'exécution facilement amovibles | Alex T |
| 6 | Exécuter d'autres programmes | Dalija Prasnikar |
| 7 | Exécuter un thread tout en gardant l'interface graphique réactive | Fr0sT, Jerry Dodge, Johan, kami, LU RD, Marjan Venema |
| 8 | Génériques | Rob Kennedy, Steffen Binas, Uli Gerhardt |
| 9 | Interfaces | Florian Koch, Willo van der Merwe |
| 10 | Mesure des intervalles de temps | Fr0sT, John Easley, kludg, Rob Kennedy, Victoria, Wolf |
| 11 | Pour les boucles | Filipe Martins, Jeroen Wiert Pluimers, John Easley, René Hoffmann, Rob Kennedy, Siendor, Y.N |
| 12 | Récupération des données TDataSet mises à jour dans un thread d'arrière-plan | MartynA |
| 13 | Utilisation d'animations dans | Alexander Petrosyan |

| | | |
|---|---|---|
| | Firemonkey | |
| 14 | Utilisation de try, sauf et enfin | EMBarbosa, Fabio Gomes, Johan, MrE, Nick Hodges, Rob Kennedy, Shadow |
| 15 | Utiliser RTTI dans Delphi | Petzy, René Hoffmann |