embarcadero
Developer Camp

【T5】Delphiテクニカルセッション

Delphi言語『再』入門
ビギナーからエキスパートまで！
意外と知らない言語機能や落とし穴

株式会社シリアルゲームズ
取締役
細川　淳

embarcadero
Developer Camp

Delphi ソースの構造

Delphi ソースの構造

- プロジェクトファイル（.dpr）
- ユニット(.pas)
- リソースファイル(.dfm など)

基本的にはこの3つのファイルでできています。
ここでは、プロジェクトファイルとユニットファイルを説明します。

Delphi プロジェクトファイル

- Project File にメインルーチンがあります。
  - 拡張子：dpr (Delphi PRoject)

```
program sample;

uses
  Windows, Forms,

{$R *.res}

begin
  Application.Initialize;
  Application.MainFormOnTaskbar := True;
  Application.CreateForm(TfrmMain, frmMain);
  Application.Run;
end.
```

Delphi プロジェクトファイル

- 予約語 program
  - program はプログラムの開始を宣言します。
  - 純粋な pascal には program (input, output) といった入出力宣言が必要でした。
    - Delphi では無視されます。

- begin - end.
  - program のブロックを示します。
  - begin で開始して
  - end. で終了です。
    - 「.」が重要です。
    - 「.」はソースの終端を表します。

Delphi プロジェクトファイル

- end. 以降には何を書いても無視されます。
  - Delphi XE 等では下記の警告が発生します。
    - [DCC 警告] *.dpr(40): W1011 'END' 以降へのテキストの記述。コンパイラはこれらを無視する

```
program sample;
(中略)
begin
(中略)
end.

この文章は無視されて、コンパイルは正常に終了します。
昔はここにプログラムの意図やメモを書いたりする場合もありました。
現在は、プログラムの最初にコメント文を入れることの方が主流です。
また、Delphi であれば、ToDo などのツールも使えます。
```

ユニットファイル

- ユニットファイルにはフォームなどの機能単位に分かれたプログラムを書きます。
  - 拡張子：pas
  - unit ファイルも最後は「end.」って終わります。

ユニットファイル

```
unit Unit2;

interface

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

type
  TForm2 = class(TForm)
  private
  public
  end;

var
  Form2: TForm2;

implementation

{$R *.dfm}

end.
```

ユニットファイルの構造

- interface
  - 宣言部です。
  - ここでクラスや関数、変数を宣言します。
  - ここで宣言したクラスや関数は、他のユニットから参照できます。
  - C/C++ で言うところのヘッダファイルのような物です。
  - Pascal は宣言が先にないとコンパイルできないため、次のようなコードの書き方があります。

ユニットファイルの構造

- 互いに参照し合うクラスは、このように自分は class であるとだけ、宣言することができます。
  - 似たような物に forward 宣言があります。
    - forward 宣言は関数について、先に宣言する物です。

```
type
  TContainer = class;

  TItem= class
  private
    Container : TContainer;
  end;

  TContainer = class
  private
    Item: TItem;
  end;
```

ユニットファイルの構造

- implementation
  - 実現部です。
  - ここで interface 部で宣言したクラスや関数を実装します
    - もちろん implementation 部でも関数やクラスを宣言できます。
    - implementation 部で宣言したクラスや関数は、このユニットの中でしか参照できません。

ユニットファイルの構造

- 初期化部と終了処理部
  - initialization
    - 初期化部
    - 初期化部はユニットがロードされたとき実行されます。
      - ロードには順序があります
  - finalization
    - 終了処理部
    - 終了処理部はユニットのロードの基本的には逆順序で実行されます。

ユニットファイルの構造

- 初期化部と終了処理部の例

```
unit exapmle;

interface

(中略)

implemenation

(中略)

initialization
  CoInitialize(nil);

finalization
 CoUninitialize;

end.
```

ユニットファイルの利用

- uses
  - uses を使ってユニットファイルを利用します。
  - uses で取り込んだユニットの interface 部で宣言したクラスや関数、変数を利用できます。
  - ここで注意したいのは循環参照です。

ユニットファイルの利用

- 循環参照の例
  - interface 部で互いに uses しあうと循環参照エラーが発生します。
    - implementation 部で uses しあってもエラーは起きません

```
unit exapmle1;

interface

uses
  example2;
```

```
unit exapmle2;

interface

uses
  example1;
```

ユニットファイルの利用

- 循環参照の解消
  - interface 部で循環参照してしまう時は
    - 型
    - 変数
  - などを互いに利用したいときがほとんどです。
  - その場合は、common.pas などのユニットを作り、このユニットを利用するようにします。

ユニットファイルの利用

```
unit common;

interface

type
  TTest = class
  end;
```

```
unit exapmle1;

interface

uses
  common;
```

```
unit exapmle2;

interface

uses
  common;
```

ユニットファイルの再帰的な構造

- ユニットファイルは－
  - 宣言部
    - 型宣言
    - 変数宣言
    - 関数の宣言
  - 実現部
- という構造をしています。
- これらは再帰的な構造としてみることもできます。

ユニットファイルの再帰的な構造

```
procedure Test;
type
  TSampleStr = AnsiString(2022);
var
  Str: TSampleStr;

  procedure InnerProc;
  var
    Str: String;
  begin
    (中略)
  end;

begin
 (中略)
end;
```

embarcadero
Developer Camp

Delphi 言語の基礎知識

予約語と指令

- 予約語
  - 予約語は for や if などコンテキストに関わらず単体で意味をなす単語です。

- 指令
  - 指令は、特定の予約語と一緒に使われるときに意味をなす単語です。
  - たとえば
    - property sample: String read FSample write SetSample;
  - property は予約語で、read write は指令です。
  - 指令は変数名などに使用できますが、混乱の元となるので指令と同じ名前をつけるのは避けた方が良いでしょう。

コンパイラ指令

- コンパイラ指令
  - コンパイラ指令は、コンパイラに対して特別な処理をさせるための指令です。
  - たとえば
    - {$R *.res}
  - は、コンパイル済みユニットファイル(.dcu) にリソースファイルを含める指示をしています。

呼び出し規約

- 呼び出し規約とは、関数を呼んだときに、スタックやレジスターをどのように使うかを指示します。
  - たとえば register 規約は、引数をレジスタに入れて渡すように指示します。

  - register (BCCでは __fastcall)（デフォルト）
  - pascal
  - cdecl
  - stdcall
  - safecall

- などがあります。

シンボル

- シンボルとは、単体で意味を持つ記号です。
  - 以下の記号がシンボルです。
    - #$&’()*+,-./:;<=>@[]^{}

- 特殊シンボルというものもあります。
  - これは、昔フルキーボードが一般的では無かった時代に、１つのシンボルを２つのシンボルで表すために使われました。

特殊シンボル	相当するシンボル
[	(.
]	.)
{	(*
}	*)

シンボル

- 特殊シンボル
  - コメントを意味する { } と (* *) の2つの記号は独立して使用できるので、以下のような記述が可能です。

```
(*
ここはコメントです。

{
   ここはコメントですが、(*   記号によって   {   なども意味をなしません。
}

ここもコメントです。
*)
```

型

- Delphi 言語は型に非常に厳しい言語です。
  - これは、Pascal から引き継いだ性格です。
  - ここでは、いくつかの型を紹介します。

クラス型

- class で宣言します。
- class は、変数、定数、型、関数といった「ある一連の作業を行う単位」を定義します。
  - オブジェクト指向の重要な機能です。
- class には可視性というものがあり、可視性によってアクセス制御を実現します。

クラス型

- private
  - 自身と同じユニットの中から参照できます。
  - C++ で言うところの friend のように扱うこともできます。
- strict private
  - これは純粋に自身からしか参照できません。
- protected
  - 自身と継承先のクラスから参照できます。
  - private と同じように同じユニットからも参照できます。
- strict protected
  - 自身と継承先でしか参照できません。

クラス型

- public
  - ソースレベルでこのクラスを参照できれば、誰でも参照できます。
- published
  - ここで宣言されたプロパティなどについては RTTI 情報が生成され、RTTI を参照できれば、誰でも参照できます。
  - Object Inspector でコンパイル済みのコンポーネントのプロパティをいじれるのは、この可視性によって RTTI が公開されているからです。

クラスリファレンス型

- クラスリファレンスはメタクラスとも言われます。
  - クラスを表す型です。

クラスリファレンス型

- たとえば、こんな風に使えます。

```
TTest = class
public
  procedure Show; virtual; abstract;
end;

TTestRef = class of TTest;

TTest1 = class(TTest)
public
  procedure Show; override;
end;

TTest2 = class(TTest)
public
  procedure Show; override;
end;
```

クラスリファレンス型

```
procedure TTest1.Show;
begin
  ShowMessage('Test1');
end;

procedure TTest2.Show;
begin
  ShowMessage('Test2');
end;

procedure ShowTest(iRef: TTestRef);
var
  Test: TTest;
begin
  Test := iRef.Create;
  Test.Show;
end;

procedure TForm2.FormCreate(Sender: TObject);
begin
  ShowTest(TTest1);
  ShowTest(TTest2);
end;
```

インターフェース型

- interface 型はメソッドとプロパティの定義のみを示し、実装は class 型に任せます。
- interface を利用すると、実装はどうあれ、interface で定義されたメソッドの存在が約束されます。

インターフェース型

```
type
  ITest = interface
    procedure Show;
  end;

  TTest1 = class(TInterfacedObject, ITest)
  public
    procedure Show;
  end;

  TTest2 = class(TInterfacedObject, ITest)
  public
    procedure Show;
  end;
```

インターフェース型

```
procedure TTest1.Show;
begin
  ShowMessage('Test1');
end;

procedure TTest2.Show;
begin
  ShowMessage('Test2');
end;

procedure TForm2.FormCreate(Sender: TObject);
var
  Test: TTest1;
  TestIntf: ITest;
begin
  Test := TTest1.Create;
  TestIntf := Test;
  TestIntf.Show;
end;
```

集合型

- 集合型は、集合を表す型です。
  - 列挙型を要素として集合型を定義します。
  - 集合はビットで値を表します。
  - 集合要素が8個しかない場合、8bit なので 1byte で表されます
    - そこで、8 個以内の要素しかない集合は Byte 型でキャストできます。
    - 同様に16個であれば、Word, 32個であれば DWord 型でキャストできます。

```
type
  TItem = (tiOne, tiTwo, tiThree);
  TItems = set of TItem;
begin
  Byte(TItem)
```

ポインタ型

- ポインタは、アドレスを表す型です。
  - ポインタでは、
    - 変数のアドレス
    - 関数のアドレス
    - メソッドのアドレス（イベント、クロージャ）
  - を表すことができます。
  - Delphi 言語では、C++ などと比べて関数やメソッドへのポインタを簡単に定義できます。

ポインタ型

```
unit Unit2;

interface

type
  PTest = ^TTest; // ポインタの定義では、このように TTest の定義が先になくてもOK
  TTest = packed record
    Name: String;
    Age: Integer;
    Data: packed array [0.. 9] of Byte;
  end;

  TForm2 = class(TForm)
    procedure FormCreate(Sender: TObject);
  private
    function ShowName(iTest: PTest): String;
  end;
```

ポインタ型

```
implementation

type
  TFunc = function(iTest: PTest): String of object;

procedure TForm2.FormCreate(Sender: TObject);
var
  Test: PTest;
  Func: TFunc;
begin
  Func := ShowName;

  New(Test);
  try
    Test^.Name := 'Asuka';
    Test^.Age := 14;

    Func(Test);
  finally
    Dispose(Test);
  end;
end;
```

```
function TForm2.ShowName(iTest: PTest): String;
begin
  ShowMessage(iTest^.Name);
end;

end.
```

embarcadero
Developer Camp

Delphi 言語の新機能

ジェネリクス

- ジェネリクスとは、型を柔軟に扱う機構です。
- 仮の型を指定して、使うときに型を指定します。

```
type
  TTest<T> = record
    Data: T;
  end;

procedure TForm2.FormCreate(Sender: TObject);
var
  StrTest: TTest<String>;
  IntTest: TTest<Integer>;
begin
  StrTest.Data := 'データ';
  IntTest.Data := 10;
end;
```

無名メソッド

- 無名メソッド(クロージャ)は最近の言語のトレンドです。
  - Script 言語でよく見る書き方を使用できます。
  - 無名メソッドはプロシージャや関数レベルで多態性を確保します。
  - また、特筆すべきは関数内関数と同じようにコンテキスト内の変数などにアクセスできます。

無名メソッド

```
type
  TProcRef = reference to procedure (Str: String);

procedure Sample1;
var
  Proc: TProcRef;
begin
  Proc := procedure (Str: String)
  begin
    ShowMessage(Str);
  end;

  Proc('Sample1');
end;
```

無名メソッド

```
procedure Sample2;
var
  Name: String;

  procedure Call(iProc:TProcRef);
  begin
    iProc('2');
  end;

begin
  Name := 'Sample';

  Call(procedure (Str: String)
    begin
      ShowMessage(Name + Str);
    end
  );
end;
```

class helper

- class helper はクラスの拡張を手助けします。
  - helper 対象のクラスを変更せずにクラスを拡張できます
  - .NET の 拡張メソッドや、Objective-C の category のような機能です
    - 別のユニットに class helper を定義しておいて、必要な時だけ、そのユニットを uses します。

class helper

```
type
  TObjectHelper = class helper for TObject
  private
    function GetInstance: TObject;
  public
    property Instance: TObject read GetInstance;
  end;

function TObjectHelper.GetInstance: TObject;
begin
  Result := Self;
end;
```

class helper

```
procedure TForm2.FormCreate(Sender: TObject);

  procedure Show(iSL: TStringList);
  begin
    ShowMessage(iSL.Text);
  end;

begin
  with TStringList.Create do
    try
      Add('Test');
      Show(Instance as TStringList);
    finally
      Free;
    end;
end;
```

演算子のオーバーロード

- Delphi 2007 から、演算子のオーバーロードが実装されています
  - C++ の演算子のオーバーロードとほとんど同じです。

```
type
  TTest = record
  private
    FValue: Integer;
  public
    class operator Add(a, b: TTest): Integer;
    constructor Create(iVal: Integer); reintroduce;
    property Value: Integer read FValue;
  end;
```

演算子のオーバーロード

```
constructor TTest.Create(iVal: Integer);
begin
  FValue := iVal;
end;

class operator TTest.Add(a, b: TTest): Integer;
begin
  Result := a.Value + b.Value;
end;

procedure TForm2.FormCreate(Sender: TObject);
var
  Test1: TTest;
  Test2: TTest;
begin
  Test1 := TTest.Create(1);
  Test1 := TTest.Create(2);

  ShowMessage(IntToStr(Test1 + Test2));
end;
```

関数のインライン化

- 関数をインライン化できるようになりました。
  - インライン化された関数は関数を呼び出すコードの代わりに、関数本体が展開されます。
  - ただし、これはコンパイラへの提案であり、必ずインライン化されるわけではありません。

```
procedure Test; inline;
begin
  ShowMessage('Test');
end;

procedure TForm2.FormCreate(Sender: TObject);
begin
  Test; // ShowMessage('Test')  が展開される？
end;
```

for in do

- for in do 文が追加されました。
  - for in do が使える代表的な型を示します。
    - 配列
    - 文字列
    - 集合
    - クラス
      - TList などが対応しています

for in do

- 集合型の例

```
procedure TForm2.FormCreate(Sender: TObject);
type
  TTest = (One, Two, Three);
  TTests = set of TTest;
const
  CTest: array [TTest] of String = ('One', 'Two', 'Three');
var
  Tests: TTests;
  Test: TTest;
begin
  Tests := [One, Three];

  for Test in Tests do
    ShowMessage(CTest[Test]);
end;
```

クラス型の拡張

- クラス型には以下の機能が追加されました
  - abstract class
  - sealed class
  - クラス定数
  - クラス変数
  - ネストクラス
  - final メソッド
  - sealed メソッド

その他

- DLL のインポートでは delayed 指令が実装されました。
  - procedure Test; external 'test.dll' name 'Test' delayed;

- Exit 関数に Result 値をわたせるようになりました。
  - Result := 10;
  - Exit;

  - Exit(10);

embarcadero
Developer Camp

付録

豆知識

- for 文の制御変数にはローカル変数しか使用できず、変更もできません。

```
procedure Test;
var
  i: Integer;

  procedure Add10(iInt: Integer);
  var
    i: Integer;
    Val: Integer;
  begin
    for i := 1 to 10 do
      Inc(Val, iInt);

    ShowMessage(IntToStr(Val);
  end;

begin
  for i := 1 to 10 do
    Add10(i);
end;
```

豆知識

- if then 文と if then else 文は違う文なので、else の前の式にはセミコロンを付けられない。

```
if (A = 1) then
  ShowMessage('1')
else
  ShowMessage('not 1');
```

- end が続くセミコロンは省略できる

```
function TForm2.ShowName(iTest: PTest): String;
begin
  ShowMessage(iTest^.Name)  // 省略できるが付いていた方が見やすい。
end;

end.
```

豆知識

- absolute 宣言
  - 変数のアドレスを指定することができます。
  - 例えば SysUtils.pas には以下のようなコードがあります

```
var
  FormatSettings: TFormatSettings absolute CurrencyString;
```

CurrencyString の宣言

```
var
  CurrencyString: string deprecated 'Use FormatSettings.CurrencyString';
  CurrencyFormat: Byte deprecated 'Use FormatSettings.CurrencyFormat';
    :
    :
```

TFormatSettings の宣言

```
type
  TFormatSettings = record
  public
    CurrencyString: string;
    CurrencyFormat: Byte;
      :
      :
```

embarcadero
Developer Camp

Q & A