embarcadero
Developer Camp

【T8】テクニカルセッション

「Delphi言語再入門」

株式会社シリアルゲームズ
取締役
細川　淳

embarcadero
Developer Camp

Classについて改めて

class

- Delphi を使う上で切っても切れない class 型
- class 型について、どの位の事をご存じですか？
- 良くは判らないけど、component もクラスだし、Form を作ったら強制的にクラスになっちゃうから、ただ何となく使っている？
- ここでは、そんな class 型について深く見ていくことにします
  - ただ、そもそも class とはなんぞや？という話はしません
  - もう少し深く見ていきます。

class 定義

- class 定義はヘルプより下記の様になっています

```
type
className = class [abstract | sealed] (ancestorClass)
   memberList
end;
```

- abstaract, sealed って？
- ancestorClass ？

class 属性

- abstract 属性
  - 抽象クラスを表す属性です
  - この属性を指定するとクラスは抽象クラスとなり生成が禁じられ……**ません**
    - 下位互換性を保つために抽象クラスの生成が許可されています

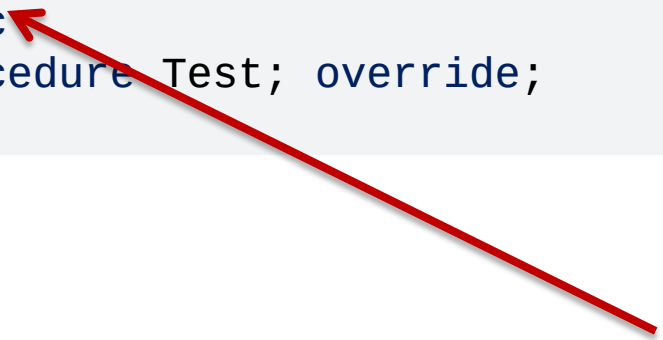
      →実質 abstract 属性は、あまり意味がありません。
    - しかし、abstract メソッドがある場合、警告が出ます
      - abstract メソッドについては後述

class 属性

- sealed 属性
  - sealed を付けて宣言すると、そのクラスを継承する事ができなくなります
    - Java の final と同じです。
      - Delphi にも final はあるのですが、メソッドに付けます。
    - ただし class helper は使えます
      - 拡張するだけなので当然ですね
      - class helper でメソッドを隠蔽可能です

class 属性 - sealed

```
type
  TTestClass = class sealed
  public
    procedure Test; virtual;
  end;

  TTestClass2 = class(TTestClass)
  public
    procedure Test; override;
  end;
```



ここで、下記のエラーが発生する
[DCC エラー] Unit1.pas(35): E2353 シールドクラス 'TTestClass' は拡張できません

class 属性 - sealed

```
type
  TTestClass = class sealed
  public
    procedure Test; virtual;
  end;

  TTestClass2 = class helper for TTestClass
    procedure Test;
  end;
```

class helper として拡張するのは OK

メソッドを隠蔽して元の Test を呼ばないようにできる
隠蔽した元の Test は

inherited Test;

として呼び出し可能

class 定義 - ancestorClass

- ancestorClass は継承元クラス
  - 何も指定しないと TObject を継承します
    - TObject もクラスとして定義されている(System.pas)ので TObject って一体なんだ！という話になりますが……
      - コンパイラマジックの一種と考えて良いと思います

- TObject
  - TObject は特別なクラスです。
  - 全てのクラス型の大元になります。
  - TObject のメソッドを使えないクラスはありません。
    - record 型もメソッドを持てるようになりましたが、class 型との最大の相違は TObject を先祖にもつか？ということです。
      つまり、**record 型は TObject を先祖にもちません**！

class とクラス参照型(メタクラス)

Delphi 言語では class も1つのオブジェクト(※)です。
例えば

```
Application.CreateForm(TForm1, Form1);
```

は、皆さん見たことがあると思います。

ここで、見て欲しいのが TForm1 と書いてあるところです。
class は型ですが、型そのものを引数として渡しています。
これは integer などでは不可能な事です。

このように class 型を参照する型を「クラス参照型」といいます(「メタクラス」ともいいます)

※より正確にいうと VMT へのポインタです

class とクラス参照型(メタクラス)

Application.CreateForm の宣言を見てみます

```
procedure TApplication.CreateForm(
  InstanceClass: TComponentClass;
  var Reference);

TComponentClass = class of TComponent;
```

こんな風になっています。

ここで、赤枠で囲った部分がクラス参照型の宣言です。

このようにクラス型そのものを渡す事によって、Factory を作ったり、多態性を確保する事が容易になります。

class とクラス参照型(メタクラス)

```
type
  TFigure = class(TObject)
  public
    procedure Draw; virtual; abstract;
  end;
  TFigureClass = class of TFigure;

  TTriangle = class(TFigure)
  public
    procedure Draw; override;
  end;

  TRect = class(TFigure)
  public
    procedure Draw; override;
  end;
```

クラス参照型を定義

三角形を描画する

四角形を描画する

```
  TFigureDrawer = class(TObject)
  private
    FFigure: TFigure;
  public
    constructor Create(const iFigureClass: TFigureClass);
    procedure Draw;
  end;

  constructor TFigureDrawer.Create(const iFigureClass: TFigureClass);
  begin
    FFigure = iFigureClass.Create;
  end;
```

渡されたクラスのインスタンスを作っている
描画される内容を TFigureDrawer は知らない

class クラスメソッド

class にはクラスメソッドといわれるものがあります。

具体的には

```
type
  TFoo = class(TObject)
  public
    class procedure Method1;
    class procedure Method2; static;
  end;
```

このように宣言の前に "class" をつけたメソッドです。

クラスメソッドはインスタンスを必要としません。

つまり

```
procedure Test;
begin
  TFoo.Method1;
  TFoo.Method2;
end;
```

のように呼び出すことができます。

さて、Method1(); と Method2(); の違いはなんでしょうか……？

classクラスメソッドと静的メソッド

普通に考えるとクラスメソッドはインスタンスが存在しないため、Self が存在しません。
それは、Self とは、インスタンスを表しているからです。
しかし！ここで、Delphi では Class もオブジェクトである、という事を思い出してください！
Method1; の中で Self を参照できてしまうのです！

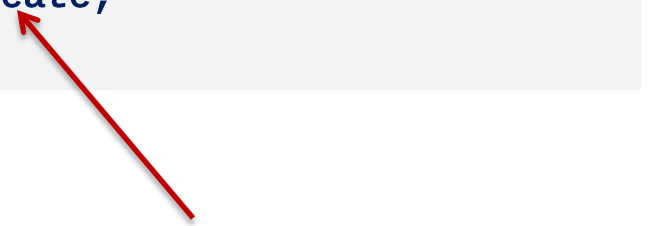
```
type
  TFoo = class(TObject)
  public
    class procedure Method1;
    class procedure Method2; static;
  end;


class procedure TFoo.Method1();
begin
  Self.Method2;
  Self.Create;
end;
```

クラスメソッド中の Self はクラスオブジェクトそのものを指します。
なので、このように Self をクラスのように使えるのです。

class クラスメソッドと静的メソッド

それに対して "static" 指定をされたクラスメソッドは、Self を持ちません。
このことから、static 指定されたメソッドを特に「静的メソッド」と呼びます。

```
type
  TFoo = class(TObject)
  public
    class procedure Method1;
    class procedure Method2; static;
  end;


class procedure TFoo.Method2();
begin
  Self.Create;
end;
```



ここで、下記のエラーが出ます

[DCC エラー] Unit1.pas(149): E2003 未定義の識別子 : 'Self'
[DCC エラー] Unit1.pas(149): E2076 このメソッドの呼び方はクラスメソッドの場合に限られます

class クラスメソッドと静的メソッド

```
type
  TFoo = class(TObject)
  public
    class procedure Method1;
    class procedure Method2; static;
  end;

procedure Test;
var
  Foo: TFoo;
begin
  Foo := TFoo.Create;
  TFoo.Method1();
  Foo.Method1();
  TFoo.Method2();
end;
```

ちなみに、バイナリレベルでも違いがでます。
static 指令がない Method1 の呼び出しでは、eax レジスタに Self を代入して呼び出しています。
しかし、Method2 ではシンプルに何もせずに呼び出しているだけです。
特にクラス参照型を欲しない場合は static 指令を付けた静的メソッドを使うとコードサイズが若干小さくなります。

```
Unit1.pas.58: TFoo.Method1;
0051A757 A16CA65100       mov eax,[$0051a66c]
0051A75C E853020000       call TFoo.Method1

Unit1.pas.59: Foo.Method1;
0051A761 8B45FC           mov eax,[ebp-$04]
0051A764 8B00             mov eax,[eax]
0051A766 E849020000       call TFoo.Method1

Unit1.pas.61: TFoo.Method2;
0051A76B E868020000       call TFoo.Method2
```

class メンバの可視性

- private, protected, public, published
  - これらの指定を、もちろん見たことがあると思います。
  - private, protected, public については、説明は不要かと思いますが、一応説明すると下表の様になります

可視性	自分自身	継承先クラス	同一ユニット
private	○		○
protected	○	○	○
public	○	○	○
strict private	○		
strict protected	○	○	

classメンバの可視性 - published

publishedとpublicは、公開範囲は同じように思えます
では、何が違うのでしょうか？

publishedはpublicと違いRTTIを生成します。
RTTIとは「実行時型情報」です。

※本来「型」とは、コンパイル時に必要なものです。
たとえば、Integer 型に Class のインスタンスを代入しようとすればコンパイルエラーになります。
しかし、Integer 型と Class のインスタンスは同じ4バイトです（32bitの時）。
CPU から見たときに違いはありません。
本来の「型」とは、プログラマを助けるために導入された概念とみなすことができます。

class メンバの可視性 - published

- published 指定されたメンバは
  - ソースコードレベルの可視性は同じ
  - RTTI が生成される
  - RTTI を介して外部のプログラムがメンバーを参照できる
    - オブジェクトインスペクタでプロパティを編集できるのは、このためです。
  - メソッドの overload はできない
    - もちろん public では overload できます。

class メンバの可視性 - published

public, published の違いについて誤解を恐れずにいうと

- public は、ソースコードレベルで公開
  - ソースコード中でメンバを参照できる
- published は、バイナリレベルで公開
  - コンパイル済みのバイナリに対して、参照できる

となります

class のバインディング

- RTTI が出てきたので、ここでバインディングについても見ていきます。

- バインディングとは

  – virtual
  – dynamic
  – override

  のことです。

class のバインディング

- virtual と dynamic
  - virtual と dynamic はソースコードレベルでは同じ動作をします。
  - どちらも、override 可能、abstract 指定可能です。
  - 違いをヘルプで引くと下記の様に書かれています。

バインディング	最適化	備考
virtual	実行速度を最適化	最も効率的な方法
dynamic	サイズを最適化	たまにしかオーバーライドされない時に使用する

  - 意味はわかるけど仕組みが判りませんね

class のバインディング

実際の所、実装上の違いは、どうなっているかというと

バインディング	意味
virtual	派生クラスでもメソッドテーブルが生成される
dynamic	基本クラスにのみメソッドテーブルが生成される

となります。

先ほどの表の「たまにしかオーバーライドされない時に使用する」というのは「派生先でメソッドテーブル」が生成されないため、コードサイズが小さくなる、という意味です。

class のバインディング - VMT

- VMT とは Virtual Method Table の事です。

メソッドの呼び出しは、Table への参照に置き換わります
概念的にはメソッドのポインタへの配列です。

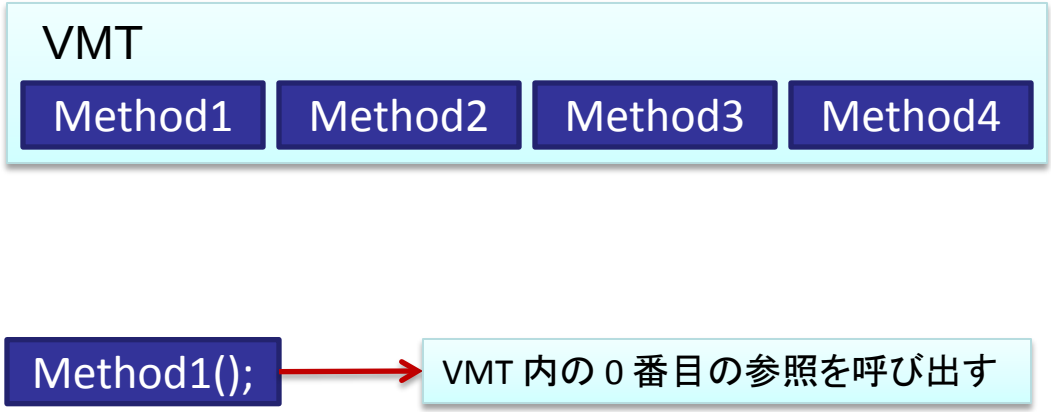
たとえば、ソースコードに

Method1();

という呼び出しコードがあった場合

VMT[0]();

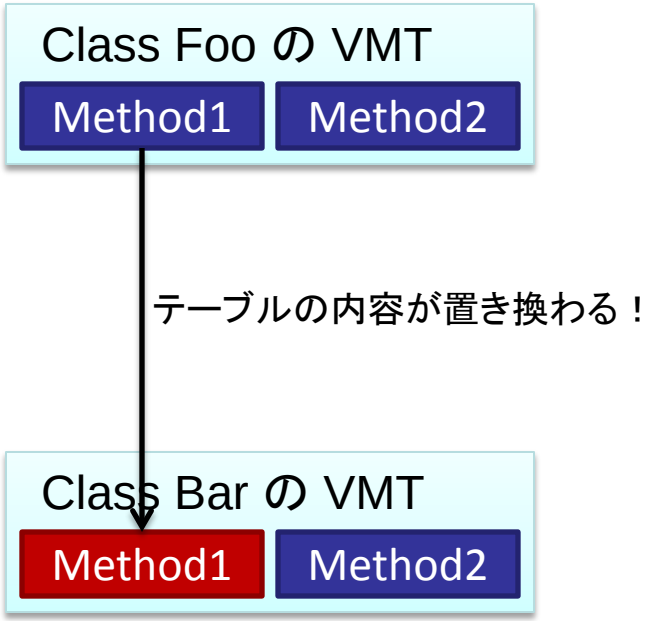
といったコードになる、ということです。



class のバインディング - VMT と override

- override

```
type
  TFoo = class(TObject)
  public
    procedure Method1(); virtual;
    procedure Method2(); virtual;
  end;

  TBar = class(TFoo)
  public
    procedure Method1(); override;
  end;
```

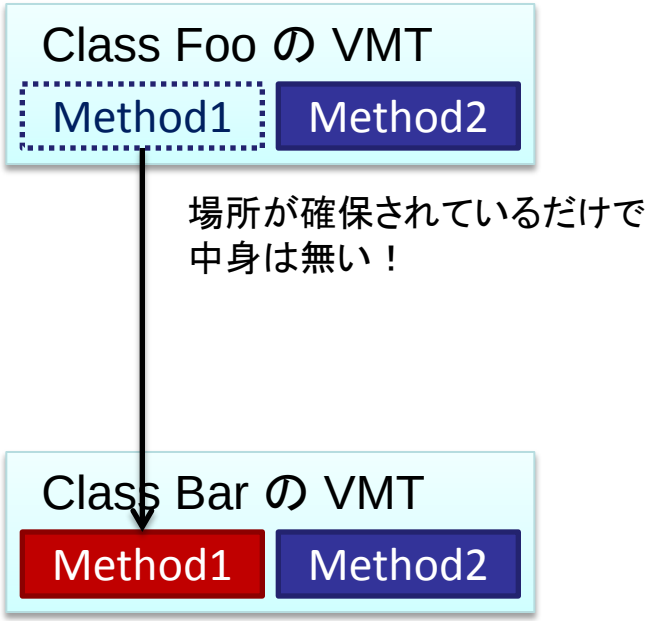



```
//擬似コードとして書くと
VMT[0] := @TFoo.Method1; // Method1 のアドレスが入った

// TBar で TFoo の Method1 を override すると
VMT[0] := @TBar.Method1; // TBar の Method1 アドレスに変わる

// 呼び出し側では下記のように呼び出しているだけなので、中身が入れ替われば呼び出し先も変わる！
VMT[0]();
```

class のバインディングと abstract

- abstract

```
type
  TFoo = class(TObject)
  public
    procedure Method1(); virtual; abstaract;
  end;

  TBar = class(TFoo)
  public
    procedure Method1(); override;
  end;
```




```
//擬似コードとして書くと
VMT[0] := nil; // アドレスは nil !

// TBar で TFoo の Method1 を override すると
VMT[0] := @TBar.Method1; // TBar の Method1 アドレスが入る
```

メソッドポインタ

今まで見てきた通り Class には2つの構成要素があります。

- Self　　インスタンスのアドレスを表す
- Method　　メソッドのアドレス

クラス参照型は、クラスオブジェクトのアドレスを渡しています。
アドレスは、32bit のマシンでは4バイトで表されます。

VMT に格納されているメソッドのアドレスも4バイトです。

Self が指すクラスのインスタンスも4バイトです。

メソッドポインタ

ここで、イベントについてちょっと見てます。

```
type
  TFoo = class(TObject)
  private
    FEvent: TNotifyEvent;
    procedure TestEvent(Sender: TObject);
  end;

procedure Test;
var
  Foo: TFoo;
begin
  Foo := TFoo.Create;
  Foo.FEvent := Foo.TestEvent;
end;
```

Foo と TestEvent で、それぞれ4バイト必要

Object	必要なバイト数
Self	4 byte
Method	4 byte
計	8 byte

Delphi の Pointer 型は4バイトしか格納できないけど……
どうやって、アドレスを代入しているのか？？

メソッドポインタ

Delphi では「メソッドポインタ」を使ってイベントを管理しています。

ヘルプを引用すると

> メソッドポインタは，特定のクラスインスタンスの特定のメソッドを指す特殊なポインタ型です。
>
> メソッドポインタはほかの手続き型と同じように動作しますが，手続き型と違って**クラスインスタンスへの隠されたポインタ**を保持しています。

つまり、メソッドポインタは、コードを書いていると気づきませんが、実は8バイトのポインタなのです。

メソッドポインタの宣言には下記の様に of object を指定します。

```
type
  TTestEvent = procedure(Sender: TObject; iFoo: TFoo) of object;

var
  Event: TTestEvent; // Event は8バイト
```

メソッドポインタ - TMethod

TMethod という面白い機構が存在します。
TMethod は 8 byte のレコードで、メソッドポインタを保持できます

```
type
  TFoo = class(TObject)
  private
    procedure Method;
  end;

procedure Test;
type
  TProc = procedure of object;
var
  Foo: TFoo;
  Proc: TProc;
  Method: TMethod;
begin
  Foo := TFoo.Create;
  Proc := Foo.Method;

  Method := TMethod(Proc);
  TProc(Method)();

  Foo.Free;
end;
```

メソッドポインタは TMethod へキャスト可能です。
そして、TMethod もメソッドポインタにキャストできます。

こんな風にも書けます

```
procedure Test;
type
  TProc = procedure of object;
var
  Foo: TFoo;
  Proc: TProc;
begin
  Foo := TFoo.Create;

  TMethod(Proc).Data:= Foo;
  TMethod(Proc).Code:= Foo.MethodAddress('Method')
  Proc;

  Foo.Free;
end;
```

メソッドポインタ - TMethod

TMethod は Code, Data という2つのメンバを持っています。
Code は、メソッドのアドレス
Data は、インスタンスのアドレス
です。
下記の様に Data の値を入れ替えると……？

```
type
  TProc = procedure of object;

  TFoo = class(TObject)
  private
    FMsg: String;
    procedure Method;
  end;

procedure TFoo.Method;
begin
  ShowMessage(FMsg);
end;
```

```
procedure Test;
var
  Foo: TFoo;
  Bar: TFoo;
  Proc: TProc;
  Method: TMethod;
begin
  Foo := TFoo.Create;
  Bar := TFoo.Create;

  Foo.FMsg := 'Foo'
  Bar.FMsg := 'Bar';

  Proc := Foo.Method;
  Method := TMethod(Proc);

  TProc(Method)();

  Method.data := Bar;
  TProc(Method)();
end;
```

embarcadero
Developer Camp

Q & A

embarcadero
Developer Camp

Classの演習

TMethodCaller を作る

いままで、通して来てクラスについての理解が深まったと思います。

ここで、少し演習をしてみます。

TMethodCaller の要件

- クラスとメソッド名を引数にもつ静的メソッド
  - メソッドの名前は仮に execute とした場合、下記の様に呼び出せる
    - TMethodCaller.Execute(TTestClass, 'Method');
- このメソッドを呼び出すとクラスを生成し、該当メソッドを呼び出す
  - つまり、上記の例は↓と同じ事をする
    - (TTestClass.Create).Method();
- なお、呼び出せるメソッドは引数を持たない手続きとします

TMethodCaller - RTTI

- RTTI を扱うクラスがあります。
- それを使うとメソッドの引数なども簡単に取得できます。
- それを使えば今回のように引数なしのメソッドではなくても、安全に呼び出すことができます。
  - 腕に自信がある方は、是非挑戦してみてください。

TMethodCaller - 宣言部

```
unit uMethodCaller;

interface

type
  TMethodCaller = class(TObject)
  private
    type TCalledProc = procedure of object;
  public
    class procedure Execute(
      const iClass: TClass;
      const iMethodName: String); static;
  end;
```

TMethodCaller - 実現部

```
implementation

class procedure TMethodCaller.Execute(
  const iClass: TClass;
  const iMethodName: String);
var
  Obj: TObject;
  Proc: TCalledProc;
begin
  Obj := iClass.Create;
  try
    TMethod(Proc).Data := Obj;
    TMethod(Proc).Code := Obj.MethodAddress(iMethodName);

    if (TMethod(Proc).Code <> nil) then
      Proc;
  finally
    Obj.Free;
  end;
end;

end.
```

TMethodCaller - 使い方

```
type
  TTestClass = class(TPersistent) // TObject から生成すると published のところで警告発生
  published
    procedure Test;
  end;

procedure TTestClass.Test;
begin
  ShowMessage('TEST');
end;

procedure TForm1.Button1Click(Sender: TObject);
begin
  TMethodCaller.Execute(TTestClass, 'Test');
end;
```