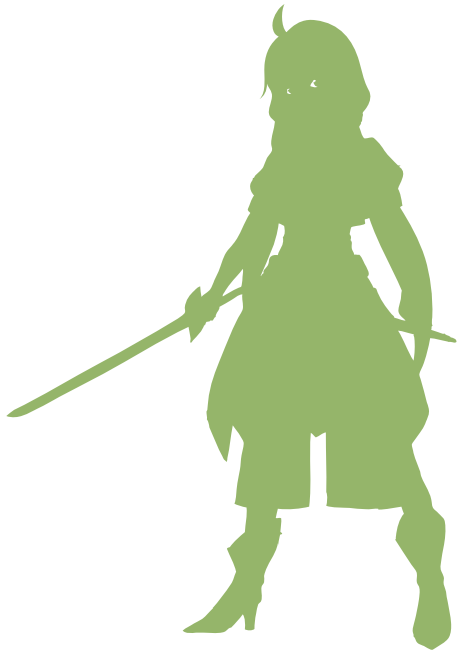
iOS、Android、Windowsライブラリ利用のためのインターフェイス活用術

第35回 エンバカデロ・デベロッパーキャンプ

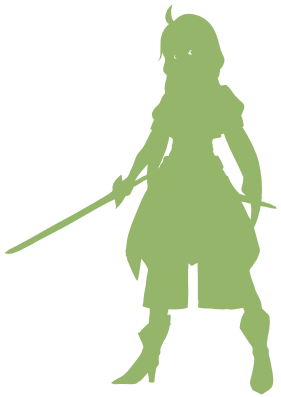
エンバカデロ・テクノロジーズ
セールスコンサルタント 毛利春幸



embarcadero®
DEVELOPER CAMP

はじめに

- 古くから存在するDelphiインターフェイスですが利用方法はどこまで理解できていますか？直接インスタンスを作成する事ができない基本的な利用方法から２つの基底クラス活用方法やオブジェクトの存続期間を管理におけるメリット、C++Builderでの、IInterface 利用方法とそのメリット、マルチデバイスでの活用方法の説明と、iOS/AndroidライブラリなどVCLからFiremonkeyまでのさまざまな利用方法をご説明いたします。



embarcadero DEVELOPER CAMP

インターフェイスってなんでしょう？

- 通常

var 変数A: クラス型

変数A := クラス型 .Create();

- インターフェイス

var 変数A: インターフェイス型

変数A := クラス型 .Create();

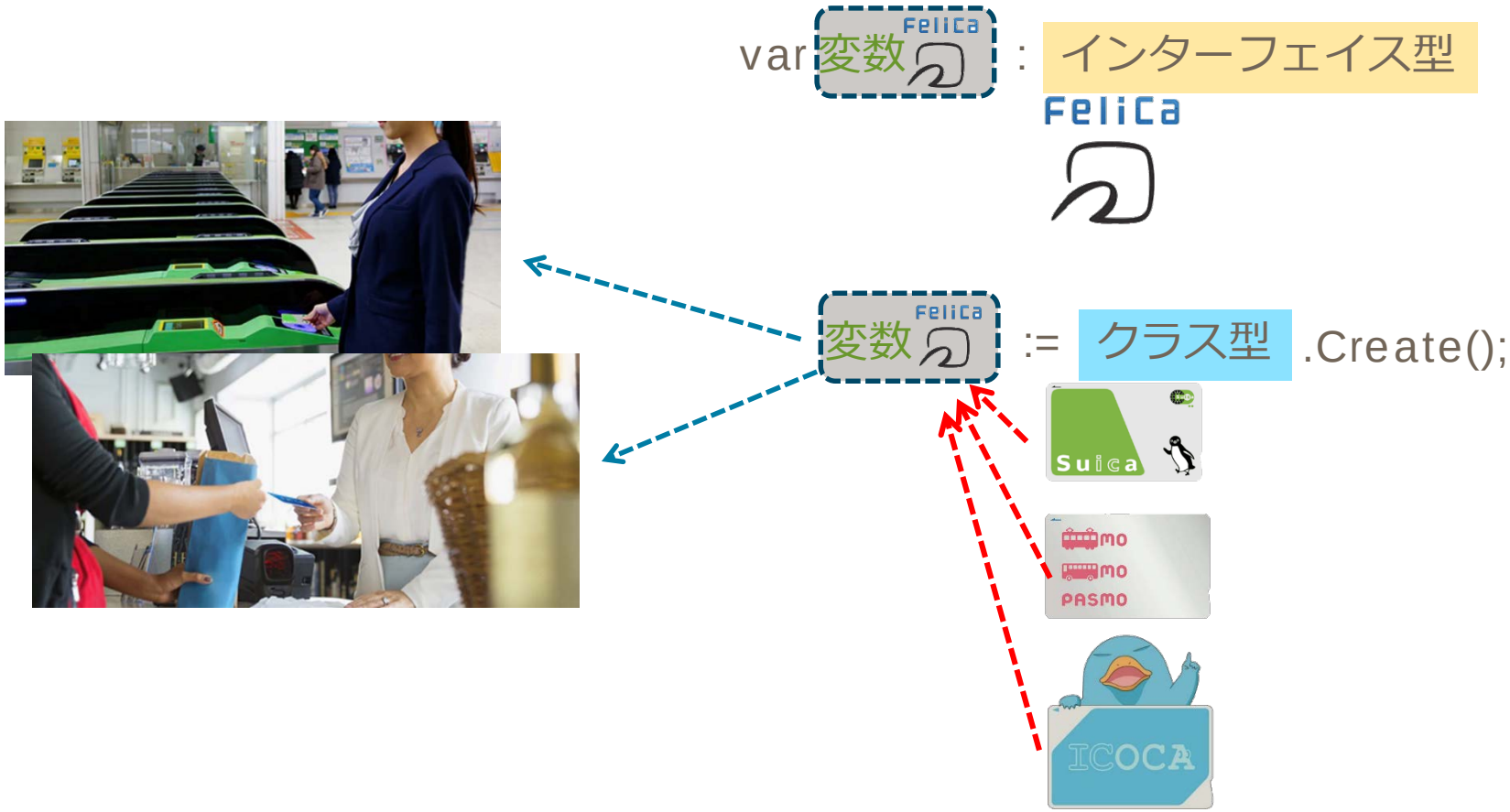
```
type
  IX = interface
    function to_string: String;
  end;
  TX1 = class(TInterfacedObject, IX)
    Fs: String;
    function to_string: String;
  end;

function TX1.to_string: String;
begin
  Result  := Fs;
end;

var
  ix1: IX;
begin
  try
    ix1 := TX1.Create;
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

つまりインターフェイスとは

- 共通関数を渡す為のインターフェイス



Delphi インターフェイスの特徴

- **インターフェイス**はクラスと同じように宣言しますが、**直接インスタンス化することはできません**

  - メソッドとプロパティのみ メンバーに指定
  - インターフェイスにはフィールドがないため、プロパティの read 指定子と write 指定子をメソッドにする必要がある
  - インターフェイスのメンバはすべて public となる
  - インターフェイスにはコンストラクタとデストラクタがない
  - メソッドは virtual、dynamic、abstract、override として宣言できない。インターフェイス自体はメソッドを実装していないため、これらの宣言に意味はない

Delphi インターフェイス用途

- インターフェイスはRTL内どのような箇所で使われているか
  - IHTTPRequestやIHTTPResponseなどのネットワークライブラリ
  - マルチデバイスのIObjectiveCや、IJava(IJavaClass)など
    - ネイティブAPIを利用する際利用
  - Open Tools API IOTAStringsなど
  - **WinRT**
    - IInspectable, IActivationFactory
  - その他いろいろ

```
var
  i:  IHTTPResponse;
  s:  String;
begin
  with TNetHTTPClient.Create(nil) do
  try
    i := Get('http://www.embarcadero.com’);
  finally
    DisposeOf;
  end;
  s := i.ContentAsString(TEncoding.UTF8);
end;
```

Delphiでシンプルなインターフェイスの作り方

- Delphi言語でインターフェイスを作ってみましょう
  - インスタンス内のテキストを取得する
    - 仮にITextとしました

```
var
  IText = interface
    function to_string: String;
  end;
```

C++従来のインターフェイス

- C++では従来から純粋仮想関数を利用したインターフェイスがあります

```
struct Itext
{
    virtual UnicodeString to_string() = 0;
    virtual ~IText(){};
};
```

←インターフェイス

```
struct TText: public Itext
{
    UnicodeString to_string(){return L"DeveloperCamp 35th";};
    ~TText(){};
};
```

この方法ではDelphiのインターフェイスが利用できません

C++BuilderでDelphiのインターフェイス

- __interfaceで作成したクラス継承元がIInterface

```
__interface IText : public IInterface
{
    virtual __fastcall UnicodeString to_string() = 0;
};
```

作成したインターフェイスを利用する

- 作成したITextを利用
  - ITextを継承したクラスを作成

```
  TText<T: class, constructor> = class(TInterfacedObject, IText)
    F_obj: T;
    function to_string: String;
    constructor Create;
    destructor Destroy; override;
  end;
var
  Text1: IText;
begin
  try
    Text1 := TText<TStringList>.Create;
    Writeln( Text1.to_string );
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

これだけでは便利かどうか解らない

C++Builder 作成したインターフェイスを利用する

- 作成したITextを利用
  - ITextを継承したクラスを作成

```
struct TText : TCppInterfacedObject<IText>
{
    __fastcall UnicodeString to_string(){return "C++Builder DeveloperCamp";};
    inline __fastcall~TText(void){};
};

int _tmain(int argc, _TCHAR* argv[])
{
    DelphiInterface<IText> text1 = new TText();
    std::wcout << text1->to_string().c_str() << std::endl;
    return 0;
}
```

インスタンスを作る時にDelphiInterfaceを宣言する

Delphi 共通のインターフェイスを利用

- 共通のインターフェイスを利用したクラスを作る事で
  変数や引数などが共通に利用可能になる

```
TTextReader = class(TInterfacedObject, IText)
  F_obj: TStringReader;
  function to_string: String;
  constructor Create;
  destructor Destroy; override;
end;
```

引数を共通にできるのでシンプル

```
func  := procedure (ltext: IText) begin
  Writeln(ltext.to_string);
end;
```

```
var
  Text1,
  Text2: IText;
begin
  try
    Text1 := TText<TStringList>.Create;
    Text2 := TTextReader.Create;
    Writeln( Text1.to_string );
    Writeln( Text2.to_string );
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

C++Builder 共通のインターフェイスを利用

- 共通のインターフェイスを利用したクラスを作る事で
  変数や引数などが共通に利用可能になる

```
struct TText: public TCppInterfacedObject<IText>
{
    UnicodeString __fastcall to_string(){return "TText"; };
};
struct TCppTextReader: public TCppInterfacedObject<IText>
{
    UnicodeString __fastcall to_string(){return "TTextReader"; };
};

int _tmain(int argc, _TCHAR* argv[])
{
    std::vector<_di_IText> v1;
    v1.push_back(new TText());
    v1.push_back(new TCppTextReader());

    for (std::vector<_di_IText>::iterator lv = v1.begin(); lv != v1.end(); lv++)
    {
        std::wcout << lv.operator *() ->to_string().c_str() << std::endl;
    }
    return 0;
}
```

vector<_di_IText>に
別型のインスタンスが入る

TAggregatedObject

- 参照回数カウントを共有できるクラスです
- C++だとTCppAggregatedObject<typename>

本日のセッションでは説明しませんが、詳しく知りたい人は下記URLへ
http://delphimaniacs.blogspot.jp/2013/12/delphi-interface.html

Delphi TProc, TFuncでの応用

- インターフェイスを使ったTProcやTFunc
  - インターフェイスを無名メソッドの代わりに利用

通常

```
var
  proc1: TProc;
begin
  proc1 := procedure begin end;
end.
```

```
var
  MyProc1:  IMyProc;//インターフェイス
  proc1:    TProc;
begin
  MyProc1 := TMyProc.Create;
  proc1 := TProc(MyProc1);
end.
```

```
TThread.CreateAnonymousThread(TProc(MyProc1)).Start;
```

C++Builder TProc, TFuncでの応用1

- C++BuilderのTProc, TFuncはTなのにインターフェイスです

変数を作る場合

```
DelphiInterface<TProc__1<TObject* > > proc1;
DelphiInterface<TProc__2<TObject*, TComponent*> > proc2;
```

こんな感じになります

この方法を使うと、lambdaを使わなくともDelphiのTProcにデータを渡せます

C++Builder TProc, TFuncでの応用2

- C++BuilderのTProcはインターフェイスなので、継承した型を作る

```
struct TMyProc: TCppInterfacedObject<TProc>
{
    void __fastcall Invoke(void)
    {
        for (int i = 0; i < 10; i++)
        {
            std::cout << i << std::endl;
        }
    };
};

typedef DelphiInterface<TProc> _di_TMyProc;

int _tmain(int argc, _TCHAR* argv[])
{
    _di_TMyProc myproc1 = new TMyProc();
    TThread::CreateAnonymousThread(myproc1)->Start();
    TThread::CreateAnonymousThread(_di_TMyProc(new TMyProc()))->Start();
    Sleep(10000);
    return 0;
}
```

インターフェイスでジェネリックス

- インターフェイスでもジェネリックスが利用可能です

```
ITextTest<T: class,constructor> = interface
  function to_string: String;
end;

TTextTest<T: class,constructor> = class(TInterfacedObject, ITextTest<T>)
  FObj: T;
  constructor Create;
  destructor Destroy; override;
  function to_string: String;
end;
```

```
var
  test1:  ITextTest<TStringList>;
  test2:  ITextTest<TTestBuilder>;
begin
  try
    test1 := TTextTest<TStringList>.Create;
    test2 := TTextTest<TTestBuilder>.Create;
    Writeln(test1.to_string);
    Writeln(test2.to_string);
  except
    on E: Exception do
      Writeln(E.ClassName, ': ', E.Message);
  end;
end.
```

型依存しないインターフェイスが作成でき共通化できます
これによりコーディングがシンプルになります

C++Builder テンプレート Delphi インターフェイス

- C++のテンプレートも利用可能です

```
struct TStringListX: public TStringList{};

template <typename T>__interface ITextTest: public System::Iinterface
{
    virtual UnicodeString __fastcall to_string() = 0;
};

template <typename T>struct TTextTest: public TCppInterfacedObject<ITextTest<T> >
{
    T* FObj;    __fastcall TTextTest()
    {
        FObj = new T();
    };
    UnicodeString __fastcall to_string()
    {
        return FObj->Text;
    }
};

int _tmain(int argc, _TCHAR* argv[])
{
    DelphiInterface<ITextTest<TStringList> >  text1 = new TTextTest<TStringList>();
    DelphiInterface<ITextTest<TStringListX> > text2 = new TTextTest<TStringListX>();
    return 0;
}
```

FireMonkeyでのインターフェイス

- プラットフォームサービス
  - 特定の実行時プラットフォームに実装されている
    実装されていないデバイスの場合もあります
    - IFMXPhoneDialerService,IFMXApplicationService,IFMXClipboardServiceなど

- デバイス事のライブラリ
  - iOS IObjectiveC派生 インターフェイス
  - Android IJava派生 インターフェイス

  - WinRT(VCL/FMX)
    - IInspectable, IActivationFactory派生 インターフェイス

プラットフォームサービス

- 50種類を超えるインターフェイス
  - FireMonkeyのコンポーネントでも利用しているので、すべて把握する必要は無いです
  - TPlatformServices(FMX.Platform.pas)内部にTDictionary<TGUID, IInterface>があり管理している

```
var
    ClipboardService: IFMXExtendedClipboardService;
begin
    if TPlatformServices.Current.SupportsPlatformService(IFMXExtendedClipboardService, ClipboardService) then
    begin
        // …
    end;
end;
```

プラットフォームサービス 例

- IFMXPhoneDialerServiceでスマホから電話をかける場合

```
procedure TForm1.Button1Click(Sender: TObject);
Var
  l_Phone: IFMXPhoneDialerService;
Begin
  if TPlatformServices.Current.SupportsPlatformService(IFMXPhoneDialerService, l_Phone) then
  begin
    l_Phone.Call('080xxxx9199’);
  end;
end;
```

iOSのライブラリを利用するNSMutableArray

- NSMutableArray = interface(NSArray) 配列を利用する
  - ネイティブのライブラリはインターフェイスです

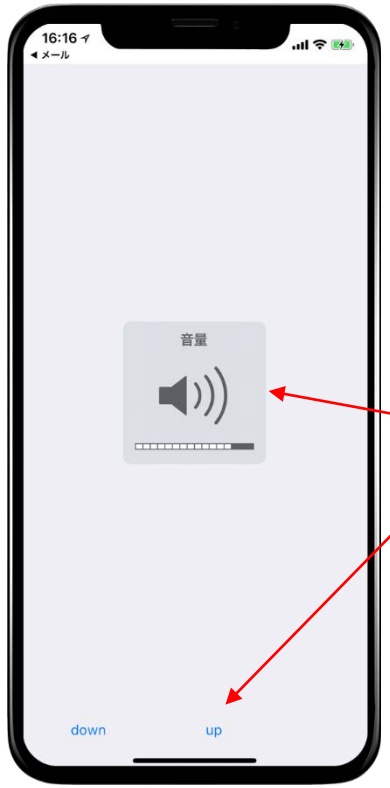
```
procedure TForm1.Button1Click(Sender: TObject);
var
  Array1: NSMutableArray;
  i:  Integer;
  s:  String;
begin
  Array1  := TNSMutableArray.Create;
  for i := 0 to 9 do
    Array1.addObject(StrToNSStr(Format('DeveloperCamp %d', [i])).init);

  for i := 0 to Array1.count-1 do
     s  := s + NSStrToStr( TNSString.Wrap( Array1.objectAtIndex(i)) ) + #10;

  Label1.Text := s;
end;
```

iOSのライブラリを利用する応用

- MPVolumeViewを利用しボリュームコントロールする



MPVolumeViewもインターフェイスが用意されています
(C++Builderの場合_di_MPVolumeView)

UP / Down ボタンで同期

iOSのライブラリを利用する応用

- MPVolumeViewを利用しボリュームコントロールする

MPVolumeViewもインターフェイスが用意されています
(C++Builderの場合_di_MPVolumeView)

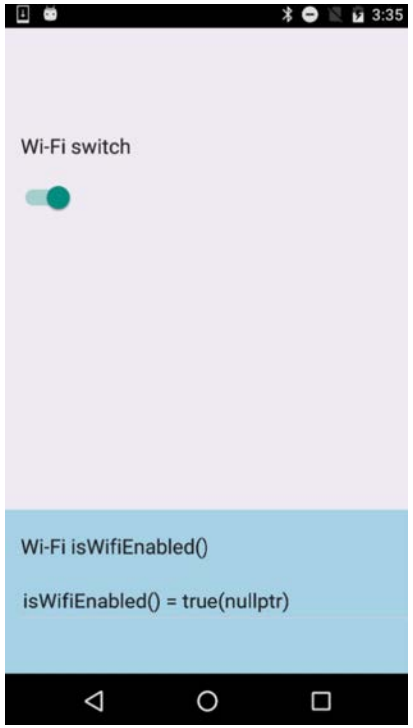
```
Fvv_  := TMPVolumeView.Create();
for i := 0 to Fvv_.subviews.count-1 do
begin
  if i = 0 then
  begin
    lSlider := TUISlider.Wrap(Fvv_.subviews.objectAtIndex(i));
    s := lSlider.value;
  end;
end;
```

Androidのライブラリを利用する

- org/json/JSONObject をDelphi / C++Builderで利用する
  - JJSONObject ← JObject ← IJavaInstance ← IJava
    上記のようにIJava継承のインターフェイスです
  - JStringもJObject継承です
    - JString とUnicodeString間の変換はStringToJString() JStringToString()が必要です

Androidのライブラリ応用

- android/net/wifi/WifiManager ライブラリを使ってWi-Fi制御
  - JWifiManager インターフェイス



WinRT

- **TWinRTImport**継承のFactory, Statics, Instance
  - WinRTの型はすべて上記3つの継承です
- インターフェイスはIInspectable継承
- 文字列はHSTRING(NativeUInt)
  - WindowsCreateString()でHSTRINGに変換

WinRT 呼び出す

- WinRT APIたくさんありますが、いくつかをご紹介
  - INetworkInformationStatics
    - ネットワーク情報
  - Json_IJsonObject
    - JSONオブジェクト
  - UserProfile_IGlobalizationPreferencesStatics
    - ユーザーズテータス
  - Power_IBatteryStatics
    - バッテリー状態
  - Http_IHttpClientFactory
    - http接続クライアント

インターフェイス活用術

- 共通インターフェイスを作る事で共通関数に渡せる
- マルチデバイスライブラリでインターフェイスが活躍する

THANKS!

www.embarcadero.com/jp

第35回 エンバカデロ・デベロッパーキャンプ