Delphiならここまでできる！
iOS / Androidネイティブアプリ構築術

第33回 エンバカデロ・デベロッパーキャンプ
【T4】Delphi/C++テクニカルセッション

株式会社シリアルゲームズ
アプリケーション第３開発部
取締役
細川 淳

embarcadero
DEVELOPER CAMP

アジェンダ

- FireMonkey おさらい
  - 仕組み
  - 限界
- ネイティブコントロールとは
  - ControlType
- API について
- API をラップする

- FireMonkey のおさらい

FireMonkey のおさらい

- Delphi XE2 (2011)から搭載されました
  - 初期はバージョンアップのたびにドラスティックな変更が繰り返された…
    - ユニットの移動、名前の変更、機能の廃止・変更・追加・分割などなど…
    - その結果、企業ユーザーに敬遠された部分もある…
      - 継続的な開発が難しくなってしまう
    - 現在は、ある程度落ち着いてきたので…多分

- マルチプラットフォーム用フレームワーク
  - VCL は Windows 専用のフレームワーク
  - FMX は Windows / macOS / iOS / Android に対応

- Object Pascal で記述されている
  - VCL と同じく Object Pascal で記述されている
  - ただし、一部別言語で書かれているところがある

FireMonkey のおさらい - 仕組み

- FireMonkey の表層
  - 各 OS の描画方法
    - Windows　　GDI / GDI+ / DirectX など
    - macOS　　Quartz / OpenGL など
    - iOS　　OpenGL ES
    - Android　　OpenGL ES
  - 各 OS の見た目
    - Aero / modern UI / Aqua / フラットデザイン / LongShadow などなど
  - 当然ですが、**全く統一されていない！**

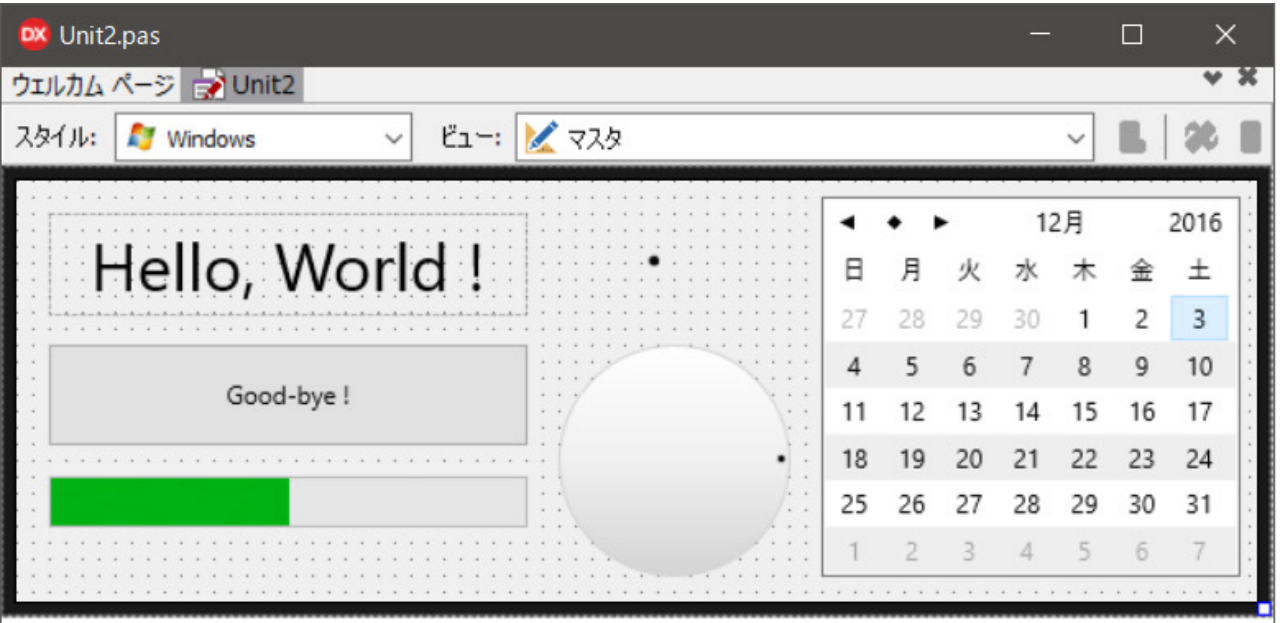
FireMonkey のおさらい - 仕組み

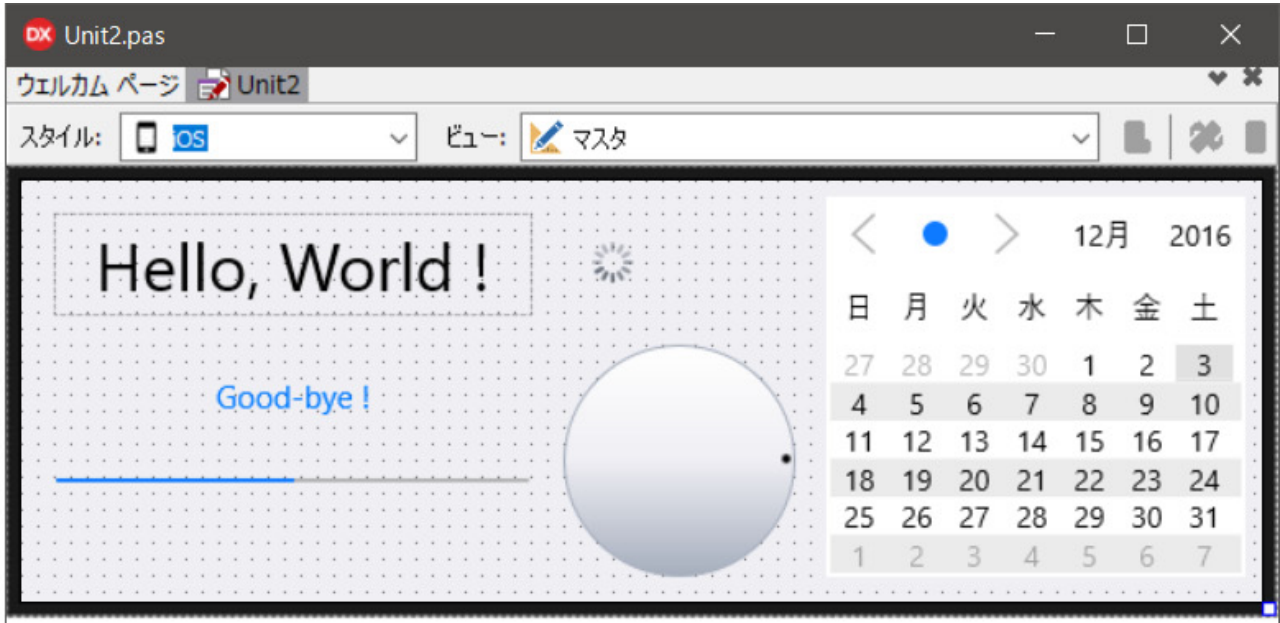
- 統一されていない！なら力ずくで統一させてやる！！

- 解決方法
  1. OS ではなく自分で描画する
  2. Style
  3. Align / Anchor
     - 今回の範囲から外れるので、この話は割愛します

FireMonkey おさらい - 仕組み

- 自分で描画する
  - 使う API 群は各 OS のネイティブを使う
    - Windows　　DirectX
    - macOS　　OpenGL
    - iOS　　OpenGL ES
    - Android　　OpenGL ES
  - 基本的にベクターグラフィクス
    - Bitmap を中に埋め込むこともできる
  - 自分で描画しているので何でもあり！
    - 全てのコントロールがコンテナになれる！
  - IDE は Windows で動いているにも関わらず他の見た目を表示できる！



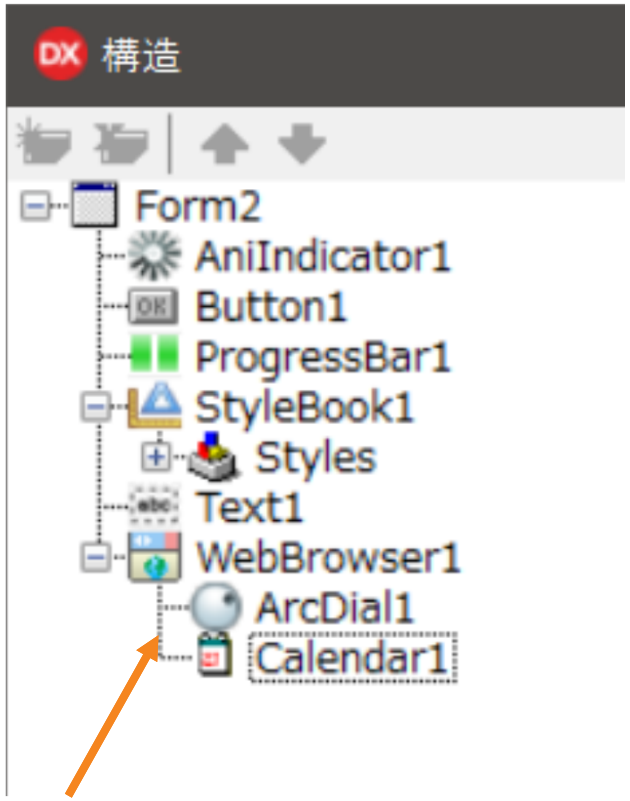


FireMonkey おさらい - 仕組み

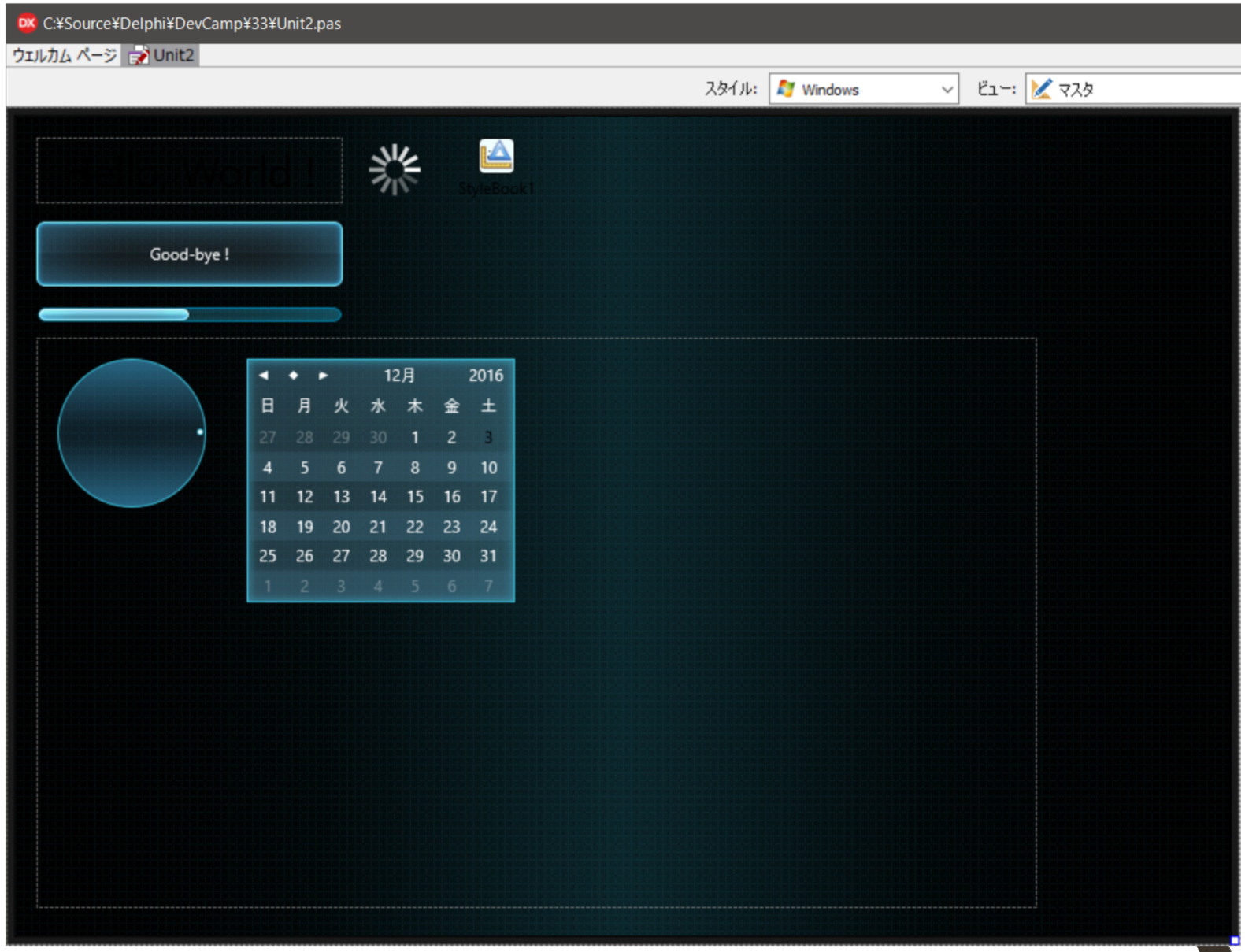
- Style
  - Style を置くともっと色々な見た目を実現できる
  - 各 OS の見た目にとらわれない自由な見た目の実現
    - 是非は置いておいて…
  - Style は自分でも作れる
    - 詳しくは前々回（31回）のデベロッパーキャンプの資料をどうぞ
      「VCL ユーザーのための FireMonkey 入門」

FireMonkey おさらい - 限界

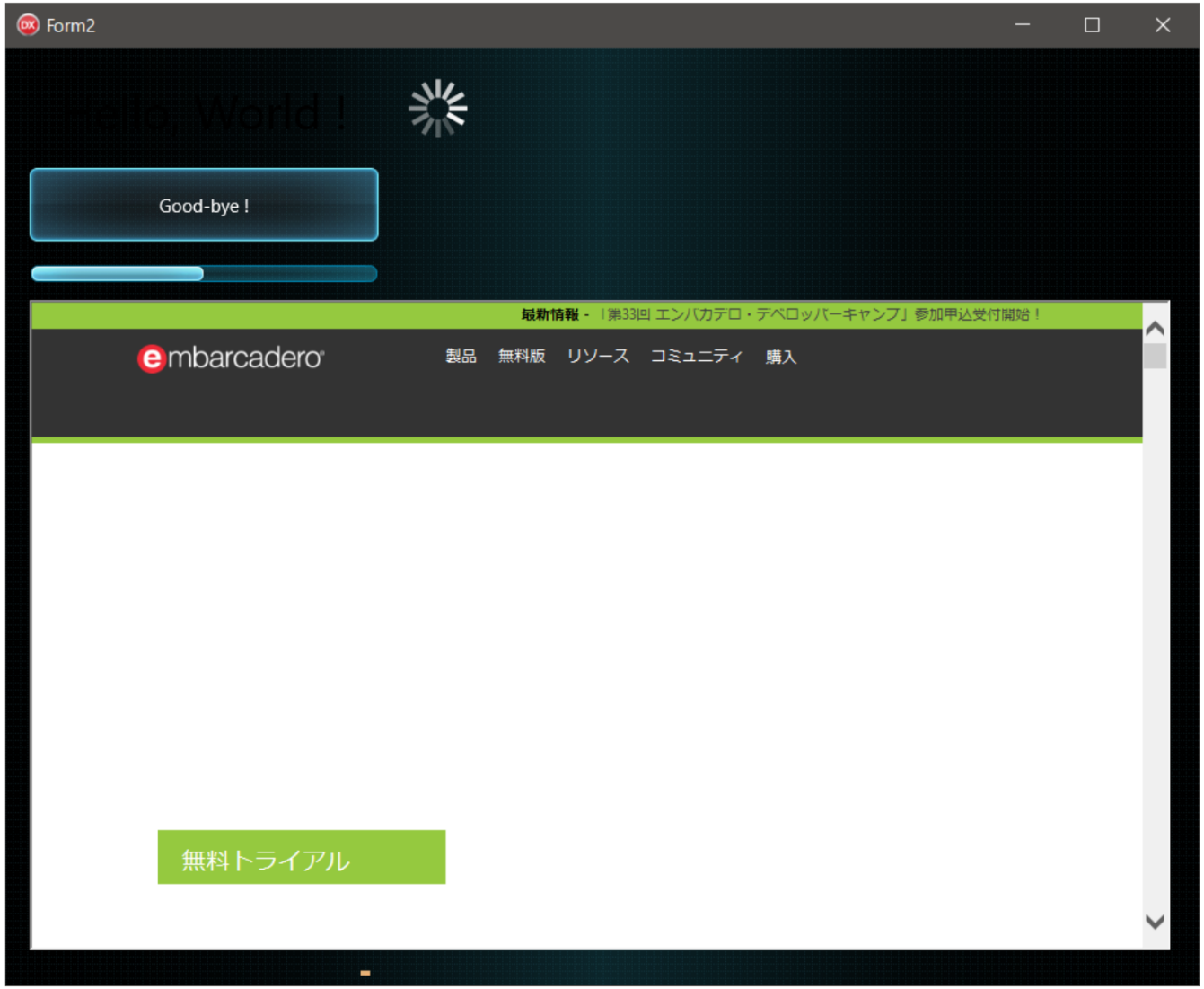
- 自分で描画することで各プラットフォームの差異を吸収できた！
- しかし、ここには恐ろしい罠が…！
- FireMonkey ワールドだけで事足りるのであれば問題はない
- しかし、少しでもネイティブコントロールが必要になったら…
- 最たる例が TWebBrowser コントロール



FireMonkey は前述のとおり全てのコントロールがコンテナになれるので、TWebBrowser の上に ArcDial, Calendar を載せてみた



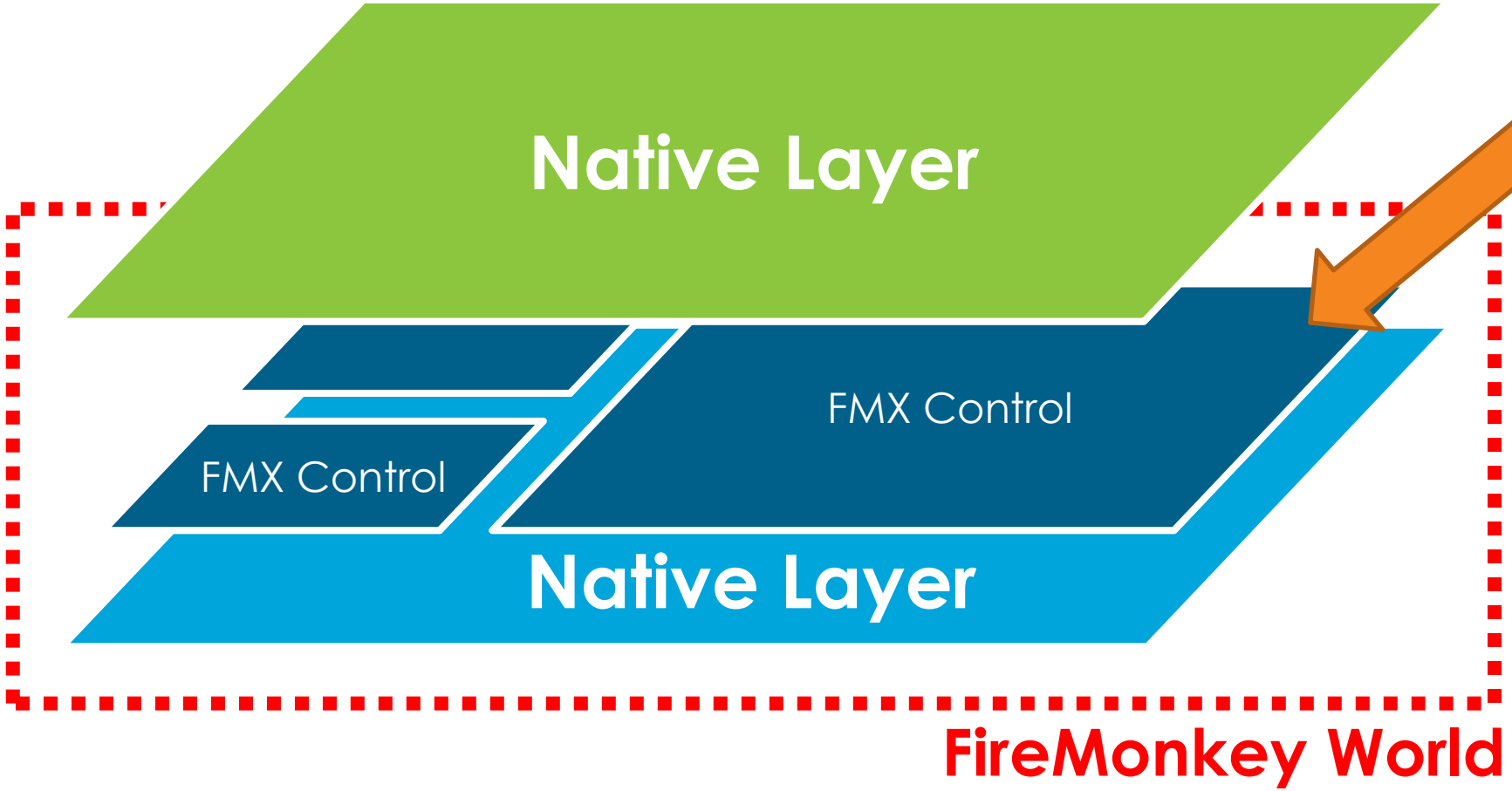
FireMonkey おさらい



いざ実行してみると、ArcDial, Calendar が表示されていない！

FireMonkey おさらい - 限界

FireMonkey の構造がこのようになっているため
Native Control より上に置けない！



各 OS に依存した構造上に
自分で描画している

FMX Control は「絵」！
OS から見た時に実体があ
るわけではない

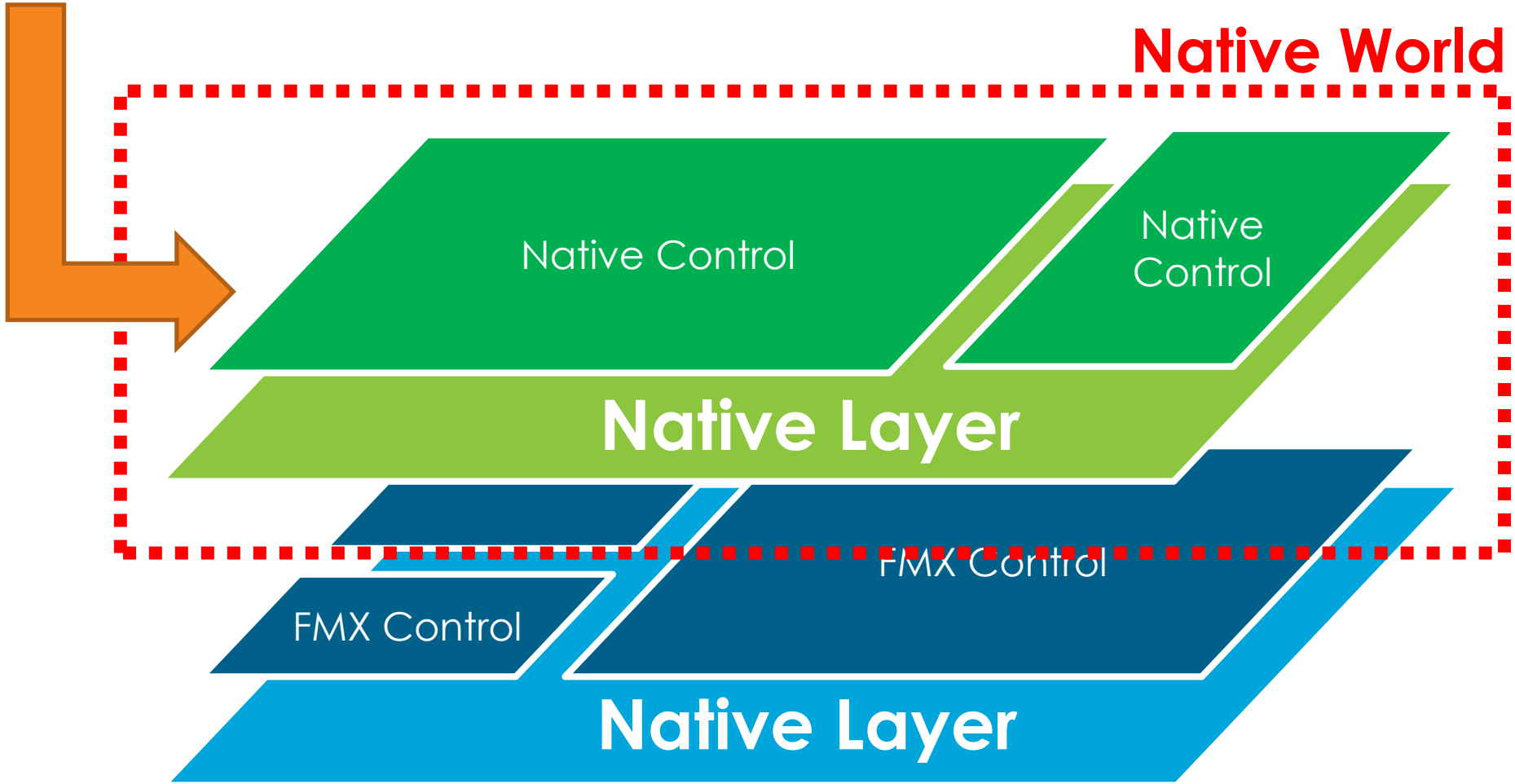
- Native Control とは？

NativeControl とは

- OS が用意しているコントロール群
  - Windows であれば EDIT や LISTBOX など
  - ≒ VCL がラッピングしているコントロール
- OS のルールの中で動くので当然 NativeControl の上に NativeControl を置ける
- OS が用意しているコントロールなので最新の機能が使える
  - たとえば、音声入力ができたり

NativeControl とは

当然ながら Native Control の上に Native Control を置ける



NativeControl とは

- FireMonkey は、OS が用意したコントロールを使わず、自分で描く道を選んだため、NativeControl は使えない…
- NativeControl が使えれば、色々と捗るのに…
- しかし…！！

NativeControl とは

- FireMonkey には Native Control を簡単に使える機構がある！
  - ただし、現在のところ iOS と Windows のみ

- ControlType プロパティ
  - Styled　　スタイルを使う（FireMonkey が描画する）
  - Platform　　プラットフォームに描画を任せる

- ControlType が使えるコントロール
  - Edit 系
  - iOS では TListView や TCalendar なども
  - 詳しくは DocWiki の次の項目を参照
    - FireMonkey のネイティブ iOS コントロール
    - FireMonkey のネイティブ Windows コントロール

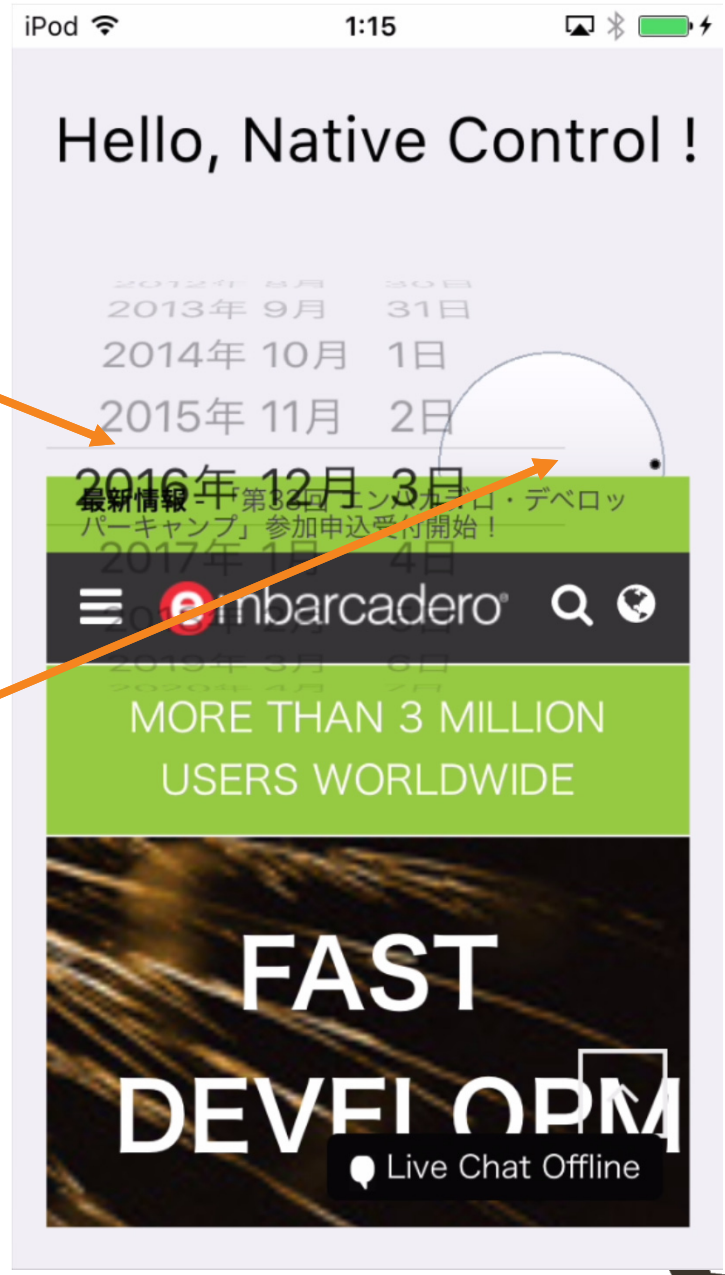
設計時

TCalendar は ControlType = Platform

設計時と実行時で見た目が違う！
TWebBrowser より上に表示されている！

TArcDial は Styled コントロールなので
下に表示されている

実行時



NativeControl とは

- FireMonkey の ControlType を使うと簡単に NativeControl が使える
- Android や macOS はサポートされていない…
- Android や macOS では、NativeControl は使えないの…？
- FireMonkey なら、きっとなんとかしてくれる…！

- API について

API について

- VCL では Win32 API をシームレスに呼べました
  例えば

```
begin
  SendMessage(Button1.Handle, WM_SETREDRAW, 0, 0); // 描画を抑制
end;
```

  のようにシームレスかつ簡単に Win32 API を呼び出せました

- FireMonkey でも！実は簡単に OS の API を呼び出せます！
  - Windows の時と同じように定義ファイルさえあればＯＫ
    - 他の開発環境のように開発メーカーからヘッダが提供されるまで待ったり、JNI ブリッジを書いたりしなくて良いのです
  - **NativeControl が使いたかったら自分で呼び出しちゃえば良い！**

API について

- FireMonkey で API を呼び出してみます
- 呼び出す API は Dialog です
- [Show Dialog] ボタンを押すと Dialog を表示します
- [Status] 部分にはダイアログ表示中「Showing」と表示します

API について - Windows

NativeLayer
II
Window

- Dialog の表示 - Windows 編
  - VCL と同じように Winapi.* ユニットを uses に追加します。
  - FMX.Platform.Win ユニットには Windows 用のクラス・関数群が用意されています。

```
implementation

{$R *.fmx}

uses
  FMX.Platform
  {$IFDEF MSWINDOWS}
  , FMX.Platform.Win
  , Winapi.Windows, Winapi.Messages
  {$ENDIF}
  ;
```

API について

- Dialog の表示 - Windows 編
  - MessageBox API を非常に簡単に呼び出せます

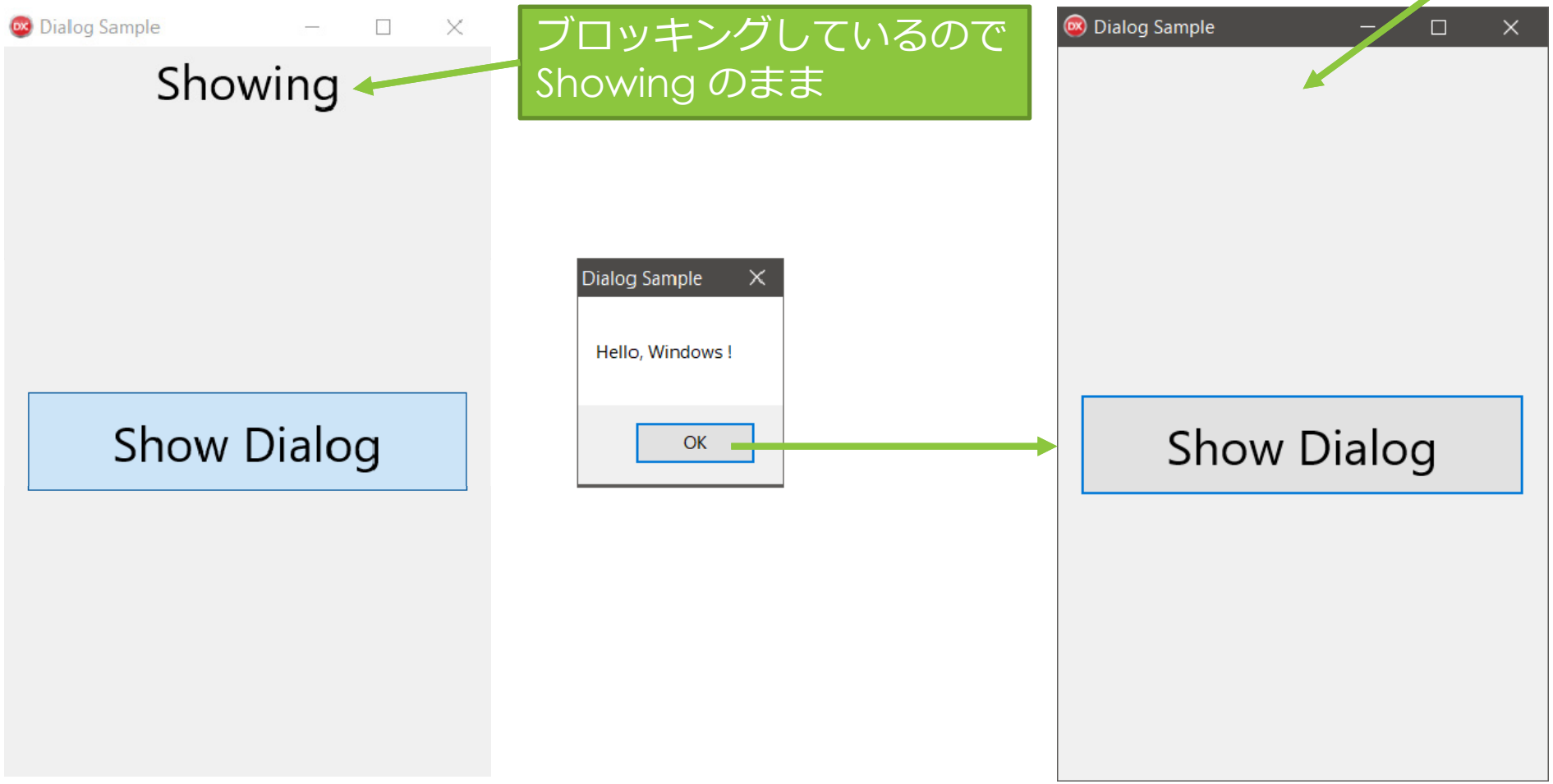
```
procedure TfrmSample.Button1Click(Sender: TObject);
begin
  Label1.Text := 'Showing';

  {$IFDEF MSWINDOWS}
  MessageBox(FormToHWND(Self), 'Hello, Windows !', 'Dialog Sample', MB_OK);
  Label1.Text := '';
  {$ENDIF}
end;
```

FMX.Platform.Win に定義されている関数
Form を渡すと HWND が返る

ブロッキングする関数なので
ダイアログを閉じないと
次の行は実行されない

API について

- Dialog の表示 - Windows 編



API について - Android

NativeLayer
II
View

- Dialog の表示 - Android 編
  - Java のクラスについて
    - Delphi では Java のクラスは全てインターフェースとして実装されています
    - ライフサイクルは参照カウント方式で管理されます
  - TJavaLocal について
    - Java のクラスはインターフェースなので、何らかの実装主体が必要です
    - それが TJavaLocal です

```
type
  TView = class(TJavaLocal, JView) // TJavaLocal を継承して Inteface を実体化します
  end;
```

API について

- Dialog の表示 - Android 編
  - Java のスレッドについて
    - Delphi で UI を触る時は Delphi の Main Thread で行います
    - 同様に Java の UI を触る時は Java の Main Thread (UI Thread) で行わないとなりません
    - これを上手く操るためのメソッドがあります

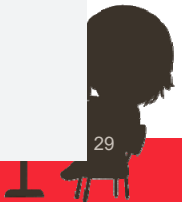
```
CallInUIthread(
  procedure
  begin
    // Java の UI を触れる
  end
);
```

```
TThread.Queue(
  TThread.Current,
  procedure
  begin
    // Delohi の UI を触れる
  end
);
```

これと同じ動作

```
mHandler.post(new Runnable() {
  @Override
  public void run()
  {
    // UI に対する動作
  }
});
```

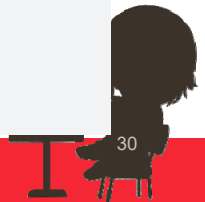
どちらのメソッドも次のタイミングまで
実行を遅らせる機能



API について

- Dialog の表示 - Android 編
  - Windows と同じように Androidapi.* ユニットを uses に追加します
    - これらのユニットが必要なクラスがあるので interface 部です
  - Winapi と違って Androidapi は細かく分かれているので必要なユニットを追加します

```
interface

uses
  System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.Graphics, FMX.Dialogs,
  FMX.Controls.Presentation, FMX.StdCtrls
  {$IFDEF ANDROID}
  , Androidapi.JNI.App
  , Androidapi.JNI.Widget
  , Androidapi.JNI.JavaTypes
  , Androidapi.JNI.GraphicsContentViewText
  , Androidapi.JNIBridge
  {$ENDIF}
  ;
```



API について

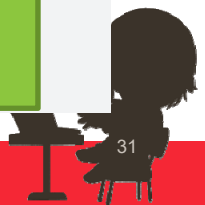
- Dialog の表示 - Android 編
  - Android のダイアログはブロッキングではないのでリスナが必要です。
  - 当然ながらリスナも Delphi で書けるのです！

```
type
  {$IFDEF ANDROID}
  TClickListener = class(TJavaLocal, JDialogInterface_OnClickListener)
  public
    procedure onClick(dialog: JDialogInterface; which: Integer); cdecl;
  end;
  {$ENDIF}
```

Java の呼び出しには JNI を使っているので
呼び出し規約は cdecl

android.content.DialogInterface.OnClickListener の定義

Delphi では Java のクラスは接頭辞として **J** がつきます。
またクラス内で定義されているクラスは
親クラス_クラス
のように定義されます。

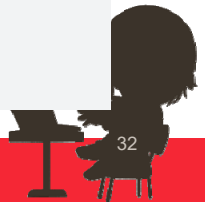


API について

- Dialog の表示 - Android 編
  - Java のクラスは Interface として実装されているので参照カウント方式で解放されます
  - 必要なクラスはクラスのメンバ変数に格納するなどして自動的に解放されないようにします

```
TfrmSample = class(TForm)
  Button1: TButton;
  Label1: TLabel;
  procedure Button1Click(Sender: TObject);
private
  {$IFDEF ANDROID}
  FClickListener: JDialogInterface_OnClickListener;
  FDialog: JAlertDialog;
  {$ENDIF}
end;
```

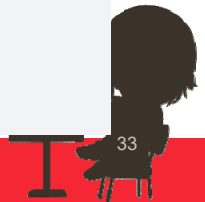
自動的に解放されないようにメンバ変数に格納する



API について

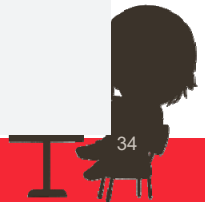
- Dialog の表示 - Android 編
  - クラス定義に必要ない部分は implementation の uses に追加します
  - Androidapi.Helpers と FMX.Helpers.Android には便利なヘルパ関数が定義されています

```
implementation

uses
  FMX.Platform
  {$IFDEF MSWINDOWS}
  , FMX.Platform.Win
  , Winapi.Windows, Winapi.Messages
  {$ENDIF}
  {$IFDEF ANDROID}
  , Androidapi.Helpers
  , FMX.Helpers.Android
  {$ENDIF}
  ;
```



API について

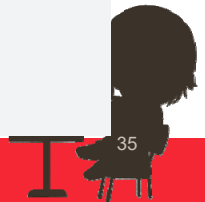
```
procedure TfrmSample.Button1Click(Sender: TObject);
begin
  Label1.Text := 'Showing';
  {$IFDEF ANDROID}
  CallInUIThread(
    procedure
    var
      Builder: JAlertDialog_Builder;
    begin
      FClickListener := TClickListener.Create;

      Builder := TJAlertDialog_Builder.JavaClass.init(TAndroidHelper.Context);
      Builder.setTitle(StrToJCharSequence('Dialog Sample'));
      Builder.setMessage(StrToJCharSequence('Hello, Android !'));
      Builder.setPositiveButton(StrToJCharSequence('OK'), FClickListener);

      FDialog := Builder.create;
      FDialog.show;
    end
  );
  {$ENDIF}
end;
```



API について

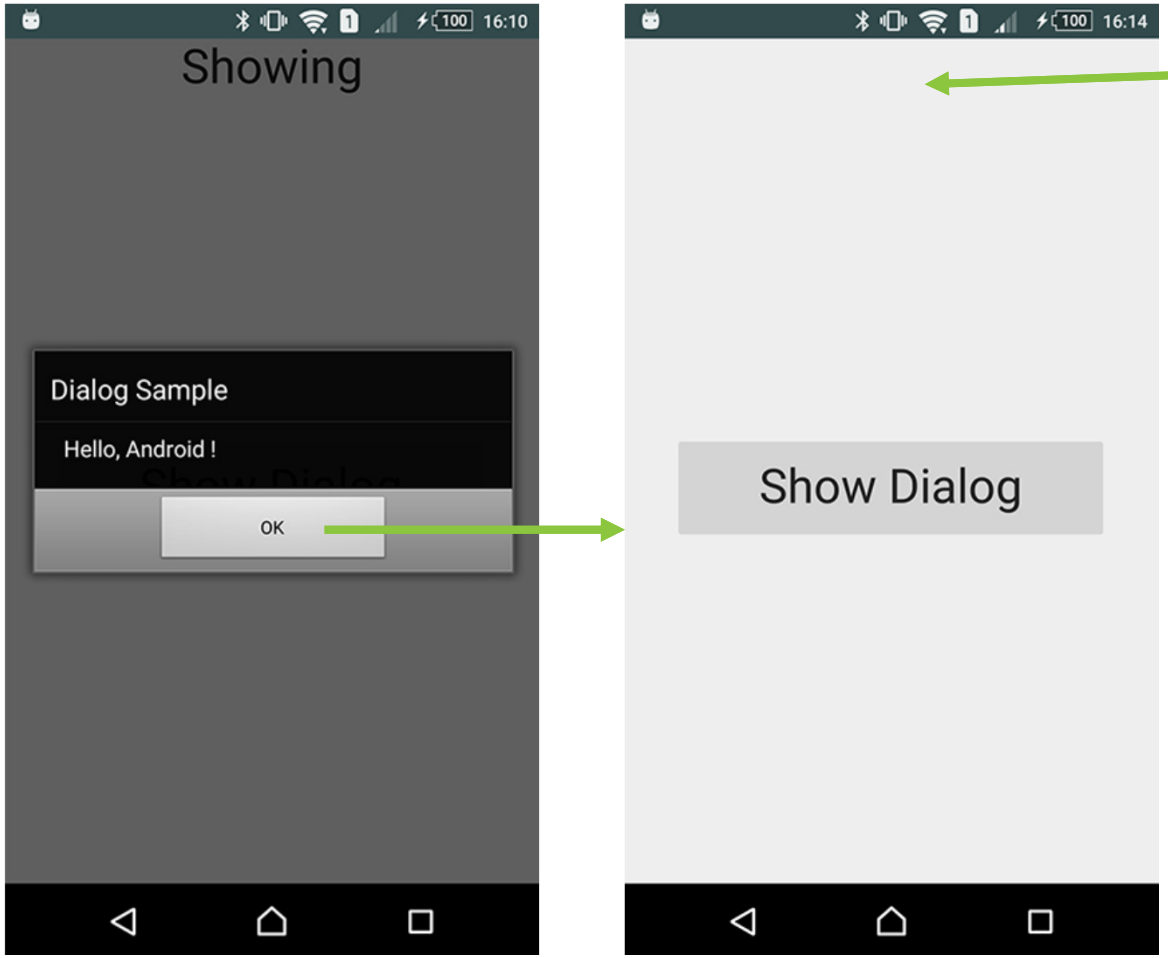
- Dialog の表示 - Android 編
  - ClickListener の実装です

```
{$IFDEF ANDROID}
{ TClickListener }

procedure TClickListener.onClick(dialog: JDialogInterface; which: Integer);
begin
  TThread.Queue(
    TThread.Current,
    procedure
    begin
      frmSample.Label1.Text := '';
    end
  );
end;
{$ENDIF}
```



API について

- Dialog の表示 - Android 編



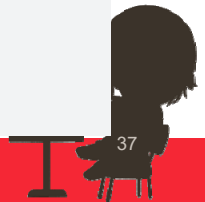
OnClickListener の onClick が呼ばれて
Label がクリアされている

API について

NativeLayer
II
View

- Dialog の表示 - iOS 編
  - iOSapi, Macapi を uses に追加します
    - macOS, iOS に共通した宣言は Macapi 名前空間に定義されます

```
interface

uses
  System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.Graphics, FMX.Dialogs,
  FMX.Controls.Presentation, FMX.StdCtrls
  {$IFDEF IOS}
  , Macapi.ObjectiveC
  , iOSapi.CocoaTypes
  , iOSapi.UIKit
  {$ENDIF}
  ;
```



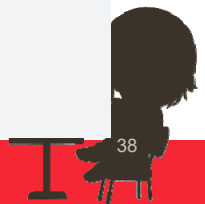
API について

- Dialog の表示 - iOS 編
  - Android と同様 iOS のクラスも Interface です
    - Android の TJavaLocal 同様に、TOCLocal を使って実装します(OC = Objective-C)

Android と同様に iOS もダイアログは
非同期なので Delegate が必要です

```
type
  {$IFDEF IOS}
  TAlertViewDelegate = class(TOCLocal, UIAlertViewDelegate)
  public
    procedure alertView(
      alertView: UIAlertView;
      clickedButtonAtIndex: NSInteger); cdecl;
    procedure alertViewCancel(alertView: UIAlertView); cdecl;
    procedure didPresentAlertView(alertView: UIAlertView); cdecl;
    [MethodName('alertView:didDismissWithButtonIndex:')]
    procedure alertViewDidDismissWithButtonIndex(
      alertView: UIAlertView;
      didDismissWithButtonIndex: NSInteger); cdecl;
  end;
  {$ENDIF}
```

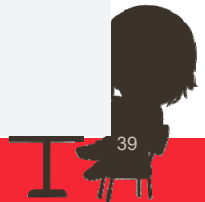
MethodName Attribute を使うと
OS が求める名前を指定できます
Object Pascal の制約から名前の指定が
難しい時に使います。
（Objective-C は名前引きのため）



API について

- Dialog の表示 - iOS 編
  - Android と同様に自動的に破棄されないようにメンバ変数に確保します

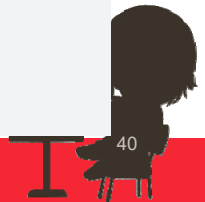
```
TfrmSample = class(TForm)
  Button1: TButton;
  Label1: TLabel;
  procedure Button1Click(Sender: TObject);
private
  {$IFDEF IOS}
  FAlertViewDelegate: TAlertViewDelegate;
  FDialog: UIAlertView;
  {$ENDIF}
end;
```



API について

- Dialog の表示 - iOS 編
  - Android と同様にヘルパ関数群を implementation 部で uses します

```
implementation

uses
  FMX.Platform
  {$IFDEF IOS}
  , Macapi.Helpers
  , FMX.Helpers.iOS
  {$ENDIF}
  ;
```



API について

- Dialog の表示 - iOS 編
  - Delegate を作って呼び出します

```
procedure TfrmSample.Button1Click(Sender: TObject);
begin
  Label1.Text := 'Showing';

  {$IFDEF IOS}
  FAlertViewDelegate := TAlertViewDelegate.Create;

  FDialog := TUIAlertView.Alloc;
  FDialog.initWithTitle(
    StrToNSStr('Dialog Sample'),
    StrToNSStr('Hello, iOS !'),
    FAlertViewDelegate.GetObjectID,
    StrToNSStr('OK'),
    nil);

  FDialog.show;
  {$ENDIF}
end;
```

Objective-C だと…
[[UIAlertView alloc] init]
のように書いている部分
Delphi でも同じように書く

Delphi 界のポインタと
Objective-C 界のポインタは別！
Objective-C 界のポインタは
GetObjectID で取得できる
(TOCLocal が提供するメソッド)

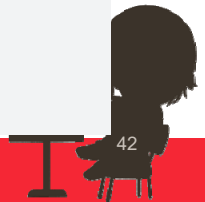
API について

- Dialog の表示 - iOS 編
  - Delegete の実装

```
{$IFDEF IOS}
{ TAlertViewDelegate }

procedure TAlertViewDelegate.alertView(
  alertView: UIAlertView;
  clickedButtonAtIndex: NSInteger);
procedure TAlertViewDelegate.didPresentAlertView(alertView: UIAlertView);
procedure TAlertViewDelegate.alertViewCancel(alertView: UIAlertView);

procedure TAlertViewDelegate.alertViewDidDismissWithButtonIndex(
  alertView: UIAlertView;
  didDismissWithButtonIndex: NSInteger);
begin
  frmSample.Label1.Text := '';
end;
{$ENDIF}
```

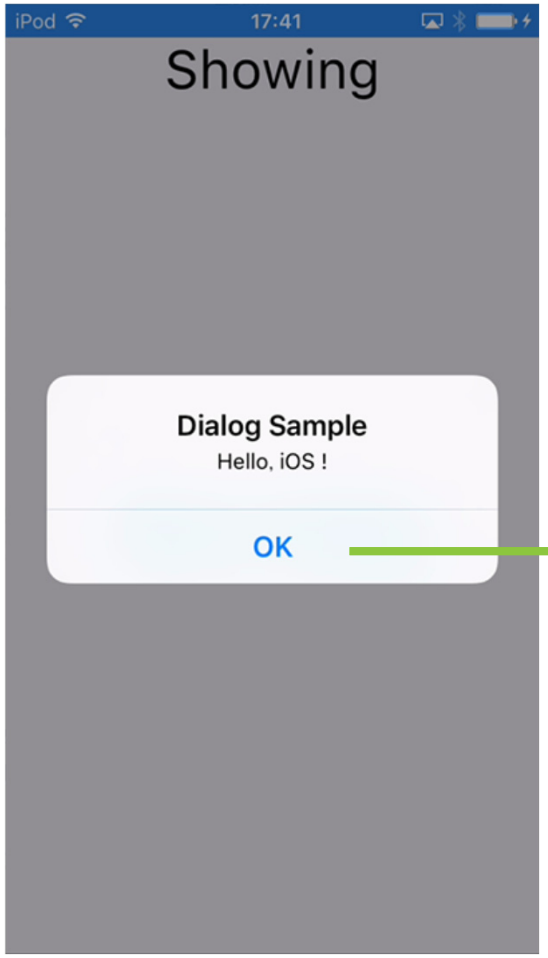
実装がない（begin end; があるだけ）ので省略

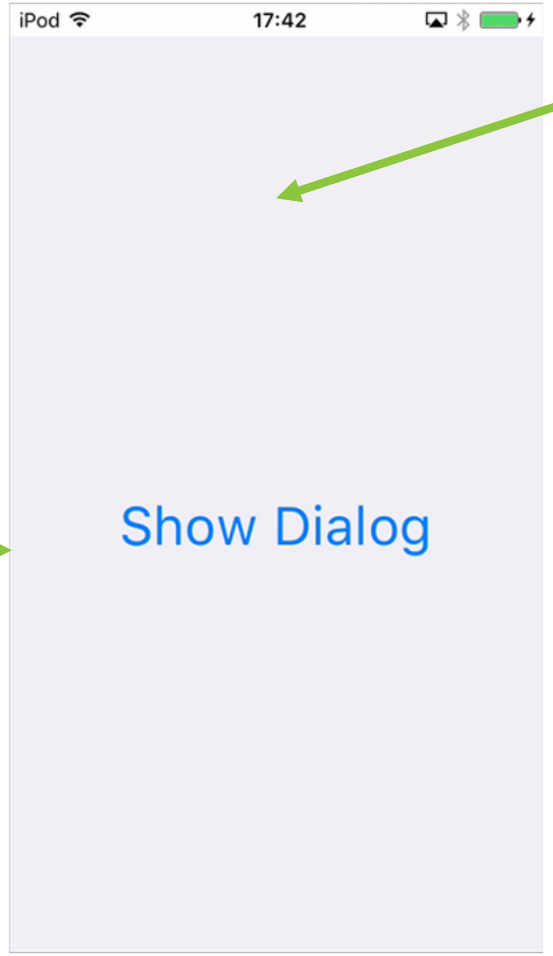
Android と違って同じバイナリになるので基本的にスレッドは考えなくてよい



API について

- Dialog の表示 - iOS 編





UIAlertViewDelegate の alertViewDidDismissWithButtonIndex が呼ばれて Label がクリアされている

API について

- Dialog の表示
  - OS の API は結構気軽に呼べました ※ macOS 版は割愛
  - そして、お気づきの方もいらっしゃると思いますが IFDEF によって処理を分岐していました
  - ソースの全体像（次ページ参照）は、各 OS 毎に分けたもののワンソースになっています

```
85  procedure TfrmSample.Button1Click(Sender: TObject);
    begin
      Label1.Text := 'Showing';

      {$IFDEF MSWINDOWS}
90    MessageBox(FormToHWND(Self), 'Hello, Windows !', 'Dialog Sample', MB_OK);
      Label1.Text := '';
      {$ENDIF}

      {$IFDEF ANDROID}
      CallInUIThread(
        procedure
        var
          Builder: JAlertDialog_Builder;
        begin
100       FClickListener := TClickListener.Create;

          Builder := TJAlertDialog_Builder.JavaClass.init(TAndroidHelper.Context);
          Builder.setTitle(StrToJCharSequence('Dialog Sample'));
          Builder.setMessage(StrToJCharSequence('Hello, Android !'));
          Builder.setPositiveButton(StrToJCharSequence('OK'), FClickListener);

          FDialog := Builder.create;
          FDialog.show;
        end
110   );
      {$ENDIF}

      {$IFDEF IOS}
      FAlertViewDelegate := TAlertViewDelegate.Create;

      FDialog := TUIAlertView.Alloc;
      FDialog.initWithTitle(
        StrToNSStr('Dialog Sample'),
        StrToNSStr('Hello, iOS !'),
120     FAlertViewDelegate.GetObjectID,
        StrToNSStr('OK'),
        nil);

      FDialog.show;
      {$ENDIF}
    end;
```

Windows

Android

iOS

API について

- IFDEF で分ければ確かにワンソースを実現できるものの…
  - １ファイルの記述量が増えて見づらい
  - 各 OS のソースが入り乱れるので処理を追いづらい

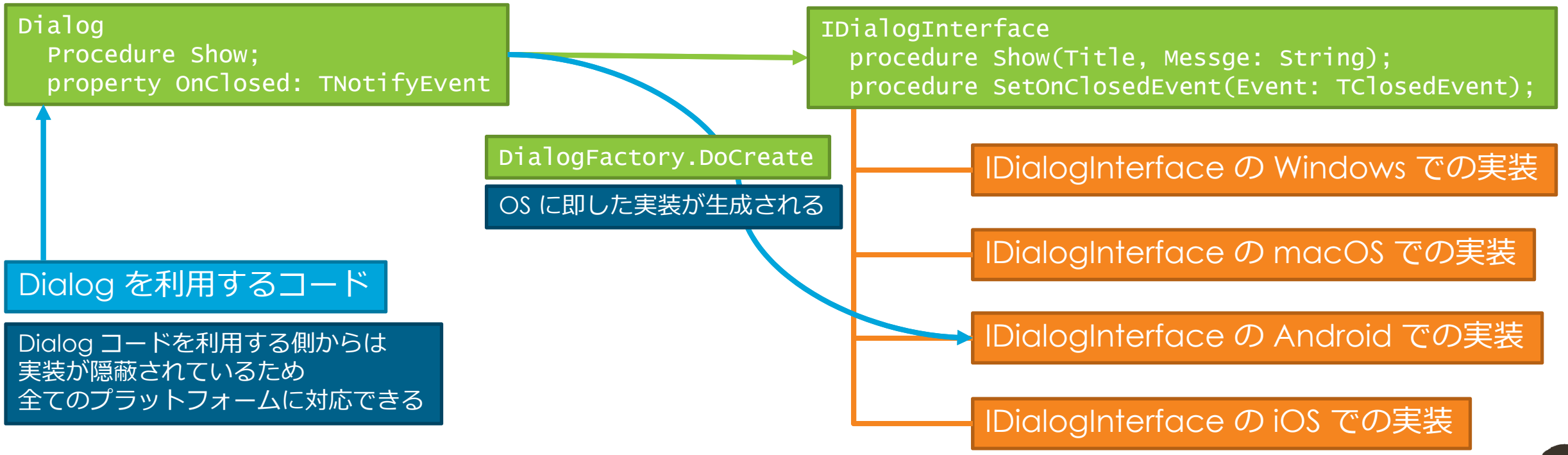
  などの問題があります

- FireMonkey では、この問題を避けるために Inteface を使った実装をしています

- API をラップする

API をラップする

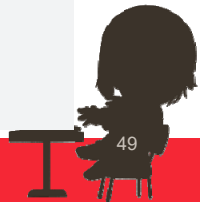
- 概念
  - 各 OS で共通する処理を Interface としてまとめます
  - その Interface を各 OS 毎に実装します



API をラップする

- 共通 Interface

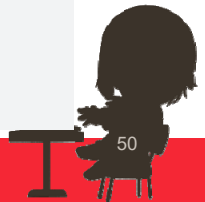
```
unit Dialog.Types;

interface

type
  TDialogClosedEvent = procedure of object;

  ICustomDialog = interface
  ['{41DA9368-2CBB-40D4-9F19-05FFD72B080A}']
    procedure Show(const iTitle, iMessage: String);
    procedure SetOnClosedEvent(const iEvent: TDialogClosedEvent);
  end;

  IDialogFactoryService = interface
  ['{CEF85443-419C-4F64-B6AC-EABA5F6FB32C}']
    function DoCreateDialog: ICustomDialog;
  end;

  TDialogFactory = class(TInterfacedObject, IDialogFactoryService)
  public
    function DoCreateDialog: ICustomDialog; virtual; abstract;
  end;

implementation

end.
```



API をラップする

- Dialog

```
unit Dialog;

interface

uses
  System.Classes
  , System.SysUtils
  , Dialog.Types
  ;

type
  TDialog = class
  private var
    FDialog: ICustomDialog;
    FClosedProc: TProc;
  private
    procedure ClosedHandler;
  public
    constructor Create; reintroduce;
    procedure Show(
      const iTitle, iMessage: String;
      const iOnClosedProc: TProc);
  end;
```



API をラップする

- Dialog

```
implementation

uses
  FMX.Platform

{$IFDEF MSWINDOWS}
  , Dialog.Win
{$ENDIF}
{$IFDEF ANDROID}
  , Dialog.Android
{$ENDIF}
{$IFDEF iOS}
  , Dialog.iOS
{$ENDIF}
;

{ TDialog }

procedure TDialog.ClosedHandler;
begin
  if Assigned(FClosedProc) then
    FClosedProc;
end;
```

使う実装を IFDEF で分ける
IFDEF はここにしか出てこない
（ユーザーからは見えない）

API をラップする

- Dialog

```
constructor TDialog.Create;
var
  DialogFactory: IDialogFactoryService;
begin
  inherited Create;

  if
    TPlatformServices.Current.SupportsPlatformService(
      IDialogFactoryService,
      IInterface(DialogFactory))
  then
    FDialog := DialogFactory.DoCreateDialog;
end;

procedure TDialog.Show(
  const iTitle, iMessage: String;
  const iOnClosedProc: TProc);
begin
  FClosedProc := iOnClosedProc;

  if FDialog <> nil then
  begin
    FDialog.SetOnClosedEvent(ClosedHandler);
    FDialog.Show(iTitle, iMessage);
  end;
end;
```

実装を生成する
Factory を取り出す

API をラップする

- 使う側

```
interface
  uses ...(中略), Dialog;

implementation

procedure TfrmSample.Button1Click(Sender: TObject);
begin
  FDialog.Show(
    'Dialog Sample',
    'Hello, Native API !',
    procedure
    begin
      Label1.Text := '';
    end
  );
end;

procedure TfrmSample.FormCreate(Sender: TObject);
begin
  FDialog := TDialog.Create;
end;

procedure TfrmSample.FormDestroy(Sender: TObject);
begin
  FDialog.DisposeOf;
end;
```

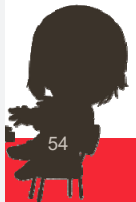
使う側は Dialog を uses するだけ
各実装の詳細は隠蔽される

API をラップする

- 一例として Windows での実装

```
unit Dialog.Win;

interface

implementation

uses
 Winapi.Windows
  , Winapi.Messages
  , FMX.Platform
  , FMX.Platform.Win
  , FMX.Forms
  , Dialog.Types
  ;

type
  TWindowsDialogFactory = class(TDialogFactory)
  public
    function DoCreateDialog: ICustomDialog; override;
  end;

  TWindowsDialog = class(TInterfacedObject, ICustomDialog)
  private var
    FOnClosedEvent: TDialogClosedEvent;
  public
    procedure Show(const iTitle, iMessage: String);
    procedure SetOnClosedEvent(const iEvent: TDialogClosedEvent);
  end;
```

interface 部には何も無し
FMX の場合は、ここに Register / Unregister がある
今回は、この Unit 自身が登録するので無し



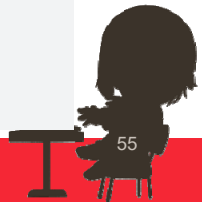
API をラップする

- 一例として Windows での実装

```
var
  GDialogFactory: TWindowsDialogFactory;

procedure RegisterDialogService;
begin
  GDialogFactory := TWindowsDialogFactory.Create;
  TPlatformServices.Current.AddPlatformService(
    IDialogFactoryService,
    GDialogFactory);
end;

procedure UnRegisterDialogService;
begin
  TPlatformServices.Current.RemovePlatformService(IDialogFactoryService);
end;

{ TWindowsDialogFactory }

function TWindowsDialogFactory.DoCreateDialog: ICustomDialog;
begin
  Result := TWindowsDialog.Create as ICustomDialog;
end;
```

Factory は各実装のインスタンスを作るクラス

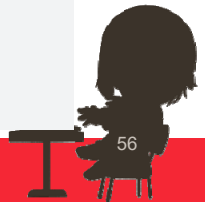
Factory を生成して PlatformService に登録
こうすると、Factory の Inteface を取り出せば
OS に適したインスタンスが作られる



API をラップする

- 一例として Windows での実装

```
{ TWindowsDialog }

procedure TWindowsDialog.SetOnClosedEvent(const iEvent: TDialogClosedEvent);
begin
  FOnClosedEvent := iEvent;
end;

procedure TWindowsDialog.Show(const iTitle, iMessage: String);
var
  Wnd: HWND;
begin
  if Screen.ActiveForm = nil then
    Wnd := 0
  else
    Wnd := FormToHWND(Screen.ActiveForm);

  MessageBox(Wnd, PChar(IMessage), PChar(iTitle), MB_OK);

  if Assigned(FOnClosedEvent) then
    FOnClosedEvent;
end;
```



API をラップする

- 一例として Windows での実装

```
initialization
  RegisterDialogService;

finalization
  UnRegisterDialogService;

end.
```

initialization で登録
finalization で削除
uses されると自動的に
この実装が使われるようになる

API をラップする

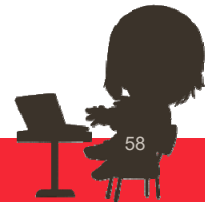
- Inteface による実装
  このような実装をすると

  - 実装が隠蔽される

  - もしも対応すべき OS が増えてもファイルを一個追加するだけで済む
    - 今まで稼働していた部分へ影響しない

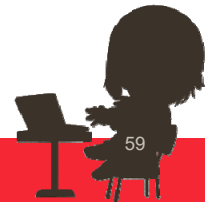
  などの利点があります

- 今回は Dialog を例にしましたが、もちろん NativeControl をラップできます！



まとめ

- Delphi + FireMonkey なら、簡単に OS の API を呼び出せる
- フレームワーク化するなら、Interface を使った実装に！
- OS の API を呼び出すので、各 OS の動作を知っておくべき！
  - 一度は Android Studio / Xcode を使った開発をしておくべき！



THANKS!

www.embarcadero.com/jp

第33回 エンバカデロ・デベロッパーキャンプ