embarcadero Developer Camp

【B1】Delphiチュートリアルセッション

Delphi で学ぶ
楽しいプログラミング基礎

デキるプログラマになる第一歩

株式会社シリアルゲームズ
細川　淳

アジェンダ

- オブジェクト指向について
- Class について
- 付録:Interface について

embarcadero Developer Camp

1

OOP について

OOP とは？

- OOP = Object Oriented Program
  - オブジェクト指向プログラム
  - 今となっては OOP ではない言語の方が少ない？
    - つまり、比較する物がないので、説明しづらい……

- 原義
  - オブジェクトに注目したプログラム技法
    - オブジェクト＝データとそれを扱うプログラムの集合
      - オブジェクトは自律的に動く
      - 与えられた入力に対し、自律的に出力を導く
    - 言語として OOP を実現したのが Smalltalk や Delphi や C++ など

OOP とは？

- OOP が目指していた物
  - バグの少ないプログラム
    - スパゲティプログラム
      - ソフトウェア・クライシス
      - 構造化プログラム
  - 再利用可能なプログラム
    - 生産性を上げる
      - プログラムのモジュール化
- つまり、OOP が目指したのは、高品質・高生産性

OOP とは

- Delphi での OOP といえば！
  - コンポーネント
    - 部品化（モジュール化）
      - より高度なプログラマが部品を作る：高品質化
      - 資産化され再利用可能：高生産性
    - カプセル化（ブラックボックス化）
      - 中身を知らずとも誰でも使える：高生産性
    - 拡張性
      - コードを一部変更しただけで動作を変更できる：高生産性

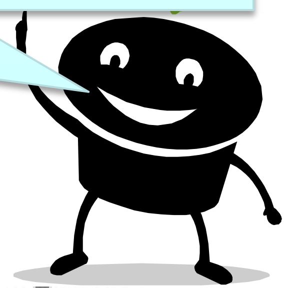
OOP とは

- コンポーネントとは、結局 Class である
  - Delphi における Class とは
    - 部品化の最小単位
      - 内部にデータを持ち、それを扱うプログラムの集合体
      - 入力に対して、自律的に出力を導く
    - カプセル化を可能にする「アクセス制御」と「プロパティ」
    - 拡張性を提供する「継承」

デキる！

プログラマー・ポイント

- OOP
  - 高品質化・高生産性（資産性）の為に考えられたシステムの総称
    - モジュール化・カプセル化・拡張性（継承）
- Delphi の OOP 機能
  - Component の高い資産性
  - Class による OOP の実現



Developer Camp

2

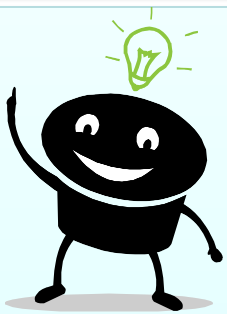
Class とは

登場人物

ブルジョアジー

資産家

プロレタリアート



プロジェクト・マネージャ

上級のプログラマ

プログラマ

Class とは

入力値が2つあり、それを加算するプログラムを書いてくれ。
金は払う。

入力	出力
整数型の値A	AとBの加算
整数型の値B	

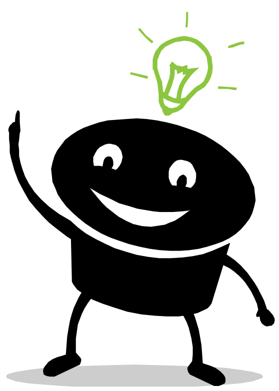
embarcadero Developer Camp

Class とは

コード A

```
function Add(i1, i2: Integer); // 入力
begin
  Result := i1 + i2;           // 出力
end;
```

できた！



Class とは

仕様変更発生！

入力	出力
整数型の値A	AとBの加算
整数型の値B	

新規1：入力値が０の場合は前回の値を使う
新規2：初めての呼び出し時、入力値が0の場合は、0を出力する

Class とは

コード A'

```
var
  G1: Integer;
  G2: Integer;

// 新規2に対応するために初期化
procedure Init;
begin
  G1 := 0;
  G2 := 0;
end;

function Add(i1, i2: Integer);
begin
  // 新規1に対応
  if (i1 = 0) then
    i1 := G1
  else
    G1 := i2;

  if (i2 = 0) then
    i2 := G2
  else
    G2 := i2;

  Result := i1 + i2;
end;
```

Class とは

- 実際のコーディング・シチュエーション

```
// 仕様を熟知しているプログラマのコード
procedure OutputAdd;
begin
  Init;
  Writeln(Foo(1, 2));
  Writeln(Foo(0, 0));
end;
```

最初に init を1回だけ
呼ぶんだな

```
// 仕様をよく知らないプログラマのコード
procedure OutputAdd;
begin
  Init;
  Writeln(Foo(1, 2));
  Init;
  Writeln(Foo(0, 0));
end;
```

毎回 init を
呼ぶんじゃね？

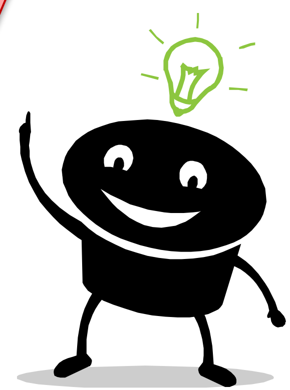
プログラムの品質が、
プログラマに大きく依存している！

Class とは

優秀なプログラマ
そうではないプログラマ
どちらも、正しく動作するコードを
書かせるにはどうすれば良いか？

Class とは

その1つの回答が OOP であり
OOP を実現した Class という
構造！



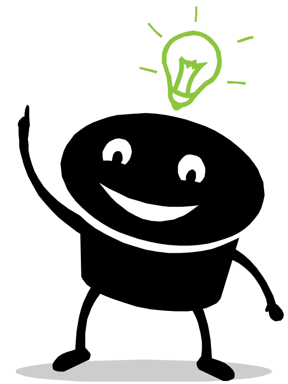
Developer Camp

Class の実際

3

Class の実際

コード A’ をクラスで書き換えてね
よろしく頼むよ



……わかりました

Class の実際

コード B クラスによる書き換え

```
type
  TFoo2 = class
  private
    F1, F2: Integer;
  public
    constructor Create;
    function Output(i1, i2: Integer): Integer;
  end;

{ TFoo2 }

constructor TFoo2.Create;
begin
  // クラスは生成時にフィールドを初期化するため実際は 0 で初期化する必要は無い
  F1 := 0;
  F2 := 0;
end;

function TFoo2.Output(i1, i2: Integer): Integer;
begin
  if (i1 = 0) then
    i1 := F1
  else
    F1 := i1;

  if (i2 = 0) then
    i2 := F2
  else
    F2 := i2;

  Result := i1 + i2;
end;
```

Class の実際

- クラスを使った場合のシチュエーション

```
// 仕様を熟知しているプログラマのコード
procedure Baz;
begin
  with TFoo2.Create do
    try
      Writeln(Output(1, 2));
      Writeln(Output(0, 0));
    finally
      Free;
    end;
end;
```

TFoo2 クラスを使う

```
// 仕様をよく知らないプログラマのコード
procedure Dame;
var
  Foo2: TFoo2;
begin
  Foo2 := TFoo2.Create;
  Writeln(Foo2.Output(1, 2));
  Writeln(Foo2.Output(0, 0));
end;
```

TFoo2 を使うのか……

解放忘れてるけど

コードの均質化による高品質化

Class の実際

- Class とは、データと、それを扱うメソッドを纏めた「型」
  - Integer や Boolean といったほかの型と同じように使える
    - 変数の定義や、引数の定義として使える
  - フィールド変数（データ）
  - メソッド（データを扱う）従来の「関数」と、ほぼ同じ意味

```
type
  TFoo2 = class
  private
    F1, F2: Integer;
  public
    constructor Create;
    function Output(i1, i2: Integer): Integer;
  end;
```

データ

メソッド

Class の実際

- Class の宣言
  - Class は型なので、type 部で宣言する

```
type
  // 慣例として型には接頭辞 "T" を付ける
  T名前 = class(T継承するクラス名) // 継承するクラスが無い場合は省略可
  private
    // 変数やメソッドの定義
  protected
    // 変数やメソッドやプロパティの定義
  public
    // メソッドやプロパティの定義
  published
    // プロパティの定義
  end;
```

Delphi では全ての Class が TObject から派生する
省略した場合は TObject を継承する

アクセス制御(private, protected etc)については後ほど

Class の実際

- Class は使うときに**生成**する必要がある！
  - クラスを「生成」するメソッドを「コンストラクタ」と呼ぶ
  - Delphi 言語では、**constructor** という予約語で宣言する
    - 名前は、慣例的に「Create」という名前をつける

- Class は使い終わったら**破棄**する必要がある！
  - クラスを「破棄」するメソッドを「デストラクタ」と呼ぶ
  - Delphi 言語では、**destructor** という予約語で宣言する
    - 名前は、慣例的に「Destory」という名前をつける

- 生成された値を「インスタンス」と呼ぶ！

Class の実際

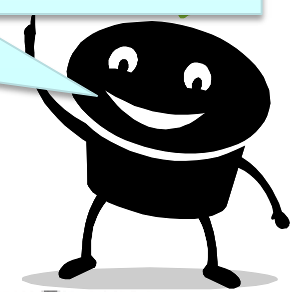
- Constructor と Destructor、クラス生成の例

```
type
  TFoo = class
  private
    FList: TList; // TList というクラスの型で変数を宣言している！
  public
    constructor Create;           // コンストラクタの定義
    destructor Destroy; override; // デストラクタの定義(TObject.Destroyはvirtual)
end;

// 最初に呼ばれる→初期化できる！
constructor TFoo.Create;
begin
  inherited;                // inherited については、後のページで
  FList := TList.Create; // 使うクラスを生成 FList には TList のインスタンスが入っている
end;

// 最後に呼ばれる→終了処理できる！
destructor TFoo.Destroy;
begin
  FList.Free; // 使うクラスを破棄(Free メソッドは、Destroy を呼び出す)
  inherited;  // inherited については、後のページで
end;
```

デキる！

プログラマー・ポイント

- Class
  - データと、それを扱うメソッドを纏めた「型」
  - Class には初期化・終了の為の「コンストラクタ」「デストラクタ」がある
    - 高品質化に寄与する
  - 生成された値を「インスタンス」と呼ぶ



embarcadero Developer Camp

4

Class の OOP 機能 - 継承

Class の OOP 機能 - 継承

また！仕様変更発生！

処理1	入力	出力
	整数型の値A	AとBの加算
	整数型の値B	

処理2	入力	出力
	整数型の値A	AとBの乗算
	整数型の値B	

・入力値が 0 の場合は前回の値を使う
・初めての呼び出し時、入力値が0の場合は、0を出力する

※処理1と処理2は、プログラム生成時に決まり、同時には起こらない

Class の OOP 機能 - 継承

- 前提
  - 処理1と処理2は、同時に発生しない
- 設計方針
  - クラスを2つ作る
    - 処理1のクラス名を TAdd とする
    - 処理2のクラス名を TMul とする

Class の OOP 機能 - 継承

宣言部や、その他の部分についてはコードBとほぼ同じため割愛

```
function TAdd.Output(i1, i2: Integer): Integer;
begin
  if (i1 = 0) then
    i1 := F1
  else
    F1 := i1;

  if (i2 = 0) then
    i2 := F2
  else
    F2 := i2;

  Result := i1 + i2;
end;

function TMul.Output(i1, i2: Integer): Integer;
begin
 if (i1 = 0) then
    i1 := F1
  else
    F1 := i1;

  if (i2 = 0) then
    i2 := F2
  else
    F2 := i2;

  Result := i1 * i2;
end;
```

同じコードだ！
同一化できないかな…？

コードを同一化する意義

コードを同一化すると

- バグがあったとき
- 改修が発生したとき

1箇所を直せば良い。
コピー＆ペーストで、複数の所に同じコードがあると何カ所も変更があるため、修正が困難になる。

Class の OOP 機能 - 継承

- Class には継承という考え方がある！
  - 基本的な処理は同じだけど、一部だけ変えたい！
  - その要望を叶える機構

- 抽出を使った SuperClass 化
  - 同じ処理を「抽出」してクラスとして纏める方法
  - 概念としては
    - (X * A) + (X * B) ⇒ X * (A + B)
    みたいな……

Class の OOP 機能 - 継承

まず、同じ処理を抜き出してみる

```
type
  TSuperClass = class
  private
    F1, F2: Integer;
  protected
    function Proc(i1, i2: Integer): Integer; virtual; abstract;
  public
    constructor Create;
    function Output(i1, i2: Integer): Integer;
  end;

{ TSuperClass }

function TSuperClass.Output(i1, i2: Integer): Integer;
begin
  if (i1 = 0) then
    i1 := F1
  else
    F1 := i1;

  if (i2 = 0) then
    i2 := F2
  else
    F2 := i2;

  Result := Proc(i1, i2);
end;
```

先ほどのコード！

Proc の処理内容は継承先で決定される！

virtual とは
継承先のクラスで変更されるかもしれないよ！とコンパイラに指示する指令。

abstract とは
このクラスでは、宣言はするけど、実装しないよ！たぶん、継承先で実装されるよ！とコンパイラに指示する指令。

→ abstract を指定すると、継承先で変更するため、virtual も一緒に指定しなくてはならない。

Class の OOP 機能 - 継承

次に異なった処理を書いてみる

```
type
  TAdd = class(TSuperClass)
  protected
    function Proc(i1, i2: Integer): Integer; override;
  end;

  TMul = class(TSuperClass)
  protected
    function Proc(i1, i2: Integer): Integer; override;
  end;

{ TAdd }

function TAdd.Proc(i1, i2: Integer): Integer;
begin
  Result := i1 + i2;
end;

{ TMul }

function TMul.Proc(i1, i2: Integer): Integer;
begin
  Result := i1 * i2;
end;
```

加算！

乗算！

TSubClass = class(TSuperClass)
と書くと TSubClass は TSuperClass を継承している
という意味になる

override とは
継承元のクラスが提供しているメソッドを変更しますよ！
という指令。
元のメソッドは、virtual か dynamic が指定されている必要がある。

それぞれのクラスでは
異なった処理だけ記述

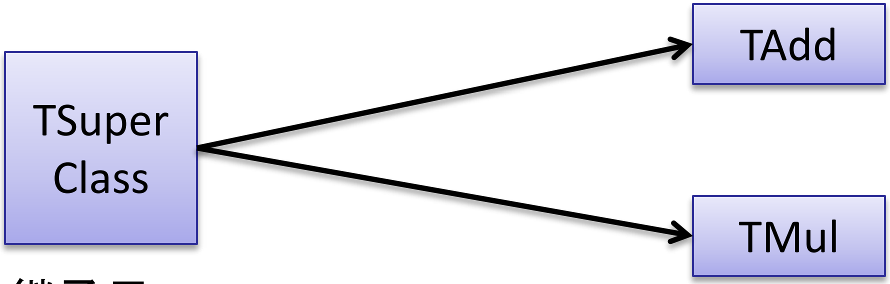
Class の OOP 機能 - 継承

TAdd, TMul を使用する

```
begin
  with TAdd.Create do
    try
      // 加算の結果が出る
      Writeln('TAdd ', Output(1, 2));
      Writeln('TAdd ', Output(0, 0));
    finally
      Free;
    end;

  with TMul.Create do
    try
      // 乗算の結果が出る
      Writeln('TMul ', Output(1, 2));
      Writeln('TMul ', Output(0, 0));
    finally
      Free;
    end;

  Readln;
end.
```

Class の OOP 機能 - 継承

- 継承
  - 一部の処理を変更する事で、元の処理を変化させる
    - TAdd, TMul は TSuperClass を「継承」して「Proc」という「一部の処理を変更」した
      - 継承を TSuperClass から考えると、TAdd, TMul は TSuperClass から「派生」した、とも言う



継承元
SuperClass という言い方以外に
親クラス、
Ancestor(祖先)とも言う

継承先
SubClass,
子クラス、
派生クラスともいう

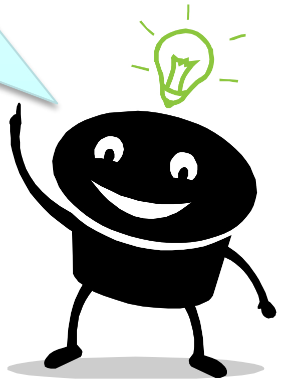
Class の OOP 機能 - 継承

継承元の型の変数に代入できる！
逆はできない！（機能がリッチになる方向へは代入できない）

```
var
  Foo: TFoo2;
  Bar: TAdd;
  Baz: TMul;
begin
  Foo := TAdd.Create;   // 代入可能

× Bar := TMul.Create;   // もちろん代入不可
× Baz := TFoo2.Create;  // これも代入不可
end.
```

Class の OOP 機能 - 継承

代入できるなら、TAdd, TMul を使う部分、
こんな風に改修できないかな？



Class の OOP 機能 - 継承

TAdd, TMul を使用する2

```
procedure OutToConsole(const iFoo: TFoo2); // TAdd, TMul のインスタンスを受け取れる
begin
   Writeln(iFoo.ClassName + ' ', iFoo.Output(1, 2));
   Writeln(iFoo.ClassName + ' ', iFoo.Output(0, 0));
end;

var
  Add: TAdd;
  Mul: TMul;
begin
  Add := TAdd.Create;
  try
    OutToConsole(Add);
  finally
    Add.Free;
  end;

  Mul := TMul.Create;
  try
    OutToConsole(Mul);
  finally
    Mul.Free;
  end;
end.
```

最初のコードの同一処理部分を纏められた！

でも、まだ似たような
コードがあるなあ……

Class の OOP 機能 - 継承

TAdd, TMul を使用する3

```
type
  TFooClass = class of TFoo2; // クラスを表す「クラス型」（メタクラス）

procedure OutToConsole(const iFooClass: TFooClass);
var
  Foo: TFoo2;
begin
  Foo := iFooClass.Create; // 渡されたクラスは、そのまま生成できる！
  try
    Writeln(iFoo.ClassName + ' ', iFoo.Output(1, 2));
    Writeln(iFoo.ClassName + ' ', iFoo.Output(0, 0));
  finally
    Foo.Free;
  end;
end;

begin
  OutToConsole(TAdd); // クラスそのものを渡せる！
  OutToConsole(TMul); // クラスそのものを渡せる！
end.
```

Class の OOP 機能 - 継承

またまた！仕様変更発生！

	入力	出力
処理1	整数型の値A	AとBの加算
	整数型の値B	

	入力	出力
処理2	整数型の値A	AとBの乗算
	整数型の値B	

	入力	出力
処理3	整数型の値A	A が 1 の時、処理1を呼び出す それ以外の時、A + A + B とする
	整数型の値B	

・入力値が 0 の場合は前回の値を使う
・初めての呼び出し時、入力値が0の場合は、0を出力する
※処理1と処理2は、プログラム生成時に決まり、同時には起こらない

Class の OOP 機能 - 継承

処理3は
新しくクラスを作らないと
実装できないかな？

TAdd から継承すればいいんですよ

Class の OOP 機能 - 継承

TAdd を継承した TAdd2 を作る

```
type
  TAdd2 = class(TAdd)
  protected
    function Proc(i1, i2: Integer): Integer; override;
  end;

{ TAdd2 }

function TAdd2.Proc(i1, i2: Integer): Integer;
begin
  if (i1 = 1) then
    Result := inherited Proc(i1, i2);
  else
    Result := i1 + i1 + i2;
end;
```

継承元のメソッドを呼び出す機能！

Class の OOP 機能 - 継承

- inherited
  - 継承元のメソッドを呼び出す機能
    - inherited メソッド名
      - inherited Proc(1, 2); // 先ほどの例
  - 上書きしたメソッドの元の処理を呼び出す場合、メソッド名は省略できる

上書きしたメソッド

元のメソッド

```
{ TAdd2 }

function TAdd2.Proc(i1, i2: Integer): Integer;
begin
  if (i1 = 1) then
    Result := inherited // Result := inherited Proc(i1, i2); と同じ
  else
    Result := i1 + i1 + i2;
end;
```

embarcadero Developer Camp

ClassのOOP機能 - 継承

- コンストラクタとデストラクタでは、初期化・終了処理を行う
  - ということは、継承元クラスも初期化・終了処理をやっている可能性が高い
  - constructor, destructor を override した場合は、絶対に inherited を呼ぶ！

```
// 初期化ができる
constructor TFoo.Create;
begin
  inherited;
  FList := TList.Create;
end;

// 終了処理ができる
destructor TFoo.Destroy;
begin
  FList.Free;
  inherited;
end;
```

継承元の初期化と終了処理
inherited を呼ぶ順番に注意！
constructor では最初に呼ぶ
destructor では最後に呼ぶ

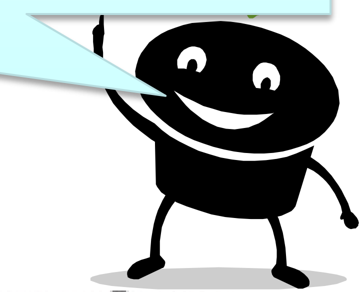
デキる！

プログラマー・ポイント

- 継承

  override

  – 親クラスのメソッドを「上書き」して、処理を変える機構
  – 親クラス型の変数に、子クラスのインスタンスを代入可能
    - 逆はできない
  – inherited は親クラスのメソッドを呼ぶ機能
    - 上書きしてしまうと変更後のメソッドしか呼べないため
- メタクラス
  – クラスそのものを扱える機能



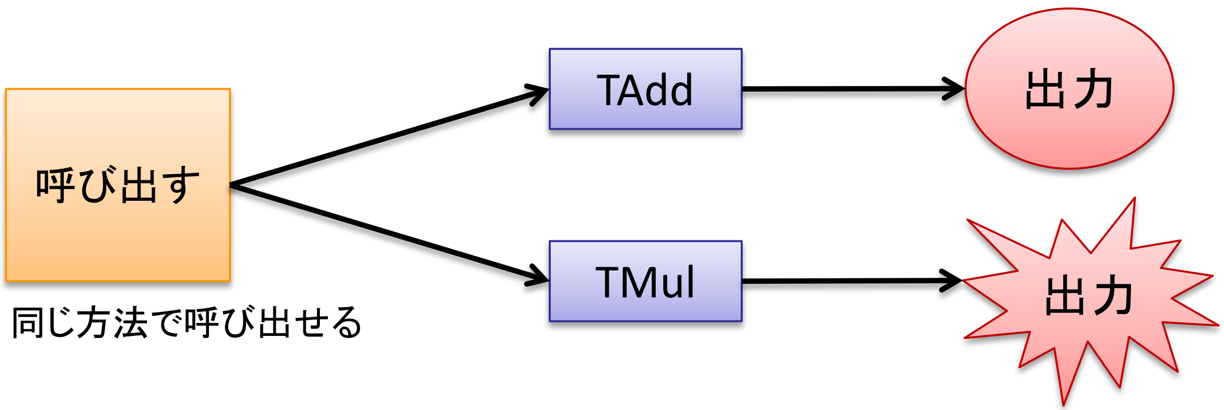
Developer Camp

5

Class の OOP 機能 - 多態性

Class の OOP 機能 - 多態性

- 多態性（ポリモーフィズム）
  - Class や型に寄って、入力や出力「形式」は同じなのに、別の結果を得られるようにする
    - TAdd, TMul は同じ呼び出しなのに、出てきた「値」は別の値になっていた

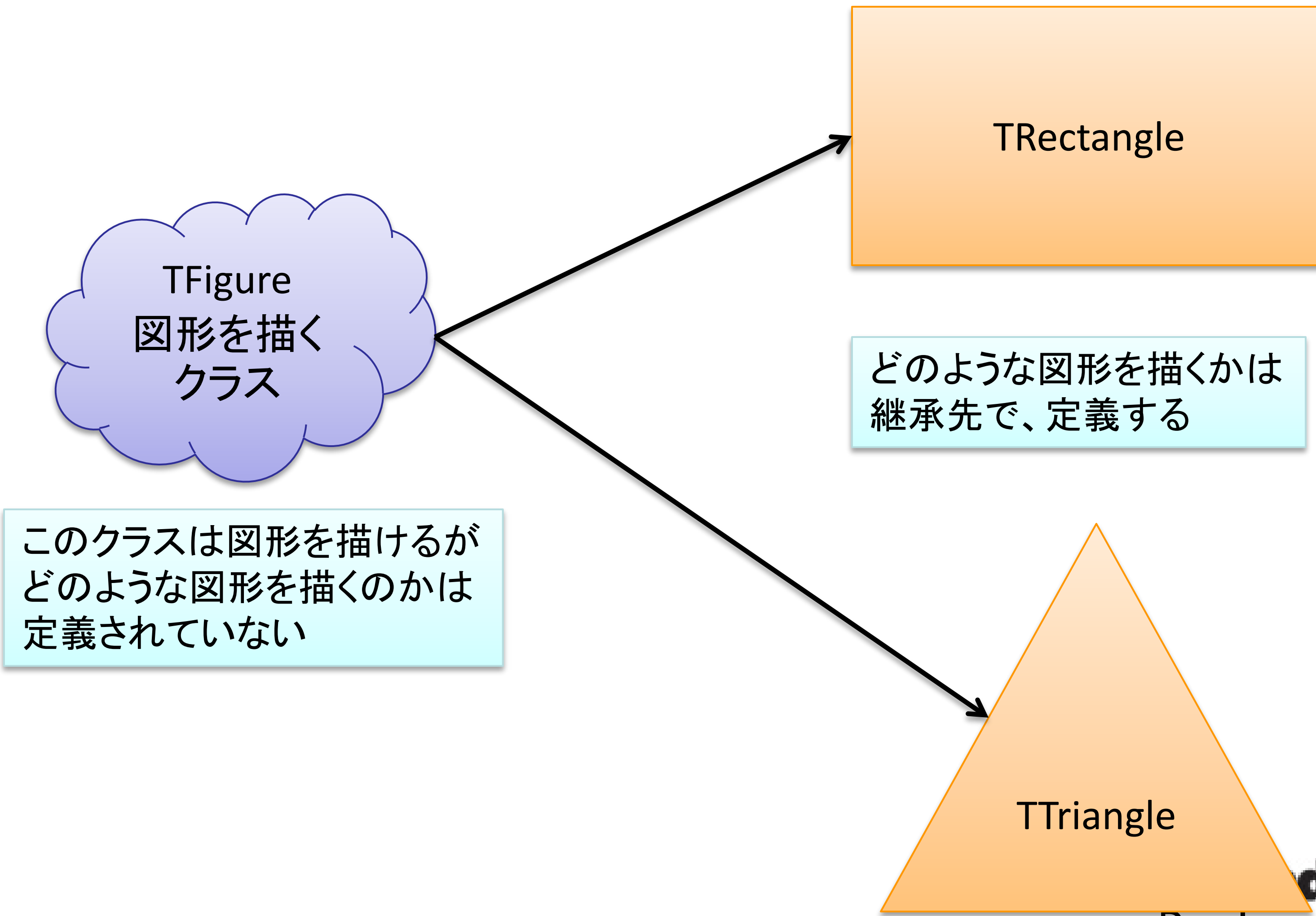


同じ方法で呼び出せる

結果は異なる！

Class の OOP 機能 - 多態性

- Class 継承する事で、多態性を表現できる



Class の OOP 機能 - 多態性

TFigure の実装

```
type
  TFigure = class
  protected
    FPoints: TList<TPoint>;
    procedure GetPoints; virtual; abstract; // 継承先で実装する！
  public
    constructor Create;
    destructor Destroy; override;
    procedure Draw;
  end;

// 実装
constructor TFigure.Create;
begin
  inherited;

  FPoints := TList<TPoint>.Create;
end;

destructor TFigure.Destroy;
begin
  FPoints.Free;

  inherited;
end;
```

ジェネリクス
コードの同一化という観点から
有用な技術だがここでは解説しない

Class の OOP 機能 - 多態性

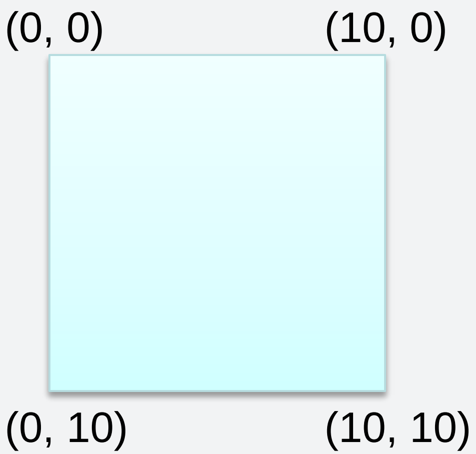
TFigure の実装

```
procedure TFigure.Draw;
var
  Point: TPoint;
  i: Integer;
begin
  GetPoints;

  if (FPoints.Count < 1) then
    Exit;

  // FPoints に与えられた点を元に線を描く
  Point := FPoints[0];
  MoveTo(Point.X, Point.Y);

  for i := 1 to FPoints.Count - 1 do
    LineTo(Point.X, Point.Y);
end;
```

Class の OOP 機能 - 多態性

四角形を描画する TRectangle の実装

```
type
  TRectangle = class
  protected
    procedure GetPoints; override; // 実装する！
  end;

procedure TRectangle.GetPoints;
begin
  FPoints.Clear;
  FPoints.Add(TPoint.Create( 0,  0));
  FPoints.Add(TPoint.Create(10,  0));
  FPoints.Add(TPoint.Create(10, 10));
  FPoints.Add(TPoint.Create( 0, 10));
end;
```



Class の OOP 機能 - 多態性

```
procedure DrawRectangle; // 四角形を描く！
var
  Figure: TFigure;
begin
  Figure := TRectangle.Create;
  try
    Figure.Draw;
  finally
    Figure.Free;
  end;
end;
```

TFigure は何を描くか知らないのに
Draw を呼び出すと、四角形が描ける！

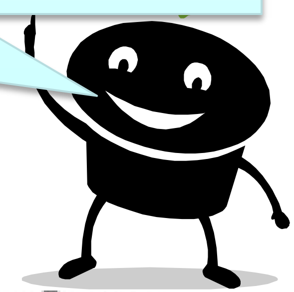
Class の OOP 機能 - 多態性

- 多態性の具体例
  - 複数のアルゴリズムがある場合
    - ランダム関数
      - 線形合同法、メルセンヌツイスターなど
    - 暗号化関数
      - RSA, DSA など
    - FireMonkey の TCanvas
      - Windows では DirectX を使い、MacOS では OpenGL を使う

デキる！

プログラマー・ポイント

- 多態性（ポリモーフィズム）
  - Delphi では Class や Interface で実現されている
  - 派生クラスの扱い方は同じなのに、派生クラス毎に異なる処理を実装していること



embarcadero Developer Camp

6

Class の OOP 機能 - アクセス制御

Class の OOP 機能 -アクセス制御

- アクセス制御
  - privte
    - 自分！
    - 同一ユニット内！(strict 指令をつけると見えなくなる; strict private)
  - protected
    - 自分！
    - 継承先！
    - 同一ユニット内！ (strict 指令をつけると見えなくなる; strict protected)
  - public
    - 誰でも！
  - published
    - むしろ積極的に見せていく！
      - public までは名前を知らないと呼び出せないが、published の場合、名前を知らなくても呼び出せる(RTTI が生成される)

Class の OOP 機能 -アクセス制御

先ほどの例で Proc が private になっていると、変更不可能！

```
type
  TSuperClass = class
  private
    F1, F2: Integer;
  private
    function Proc(i1, i2: Integer): Integer; virtual; abstract;
  public
    constructor Create;
    function Output(i1, i2: Integer): Integer;
  end;
```

もしも private だったら、継承先では見えない為
override できない！！

Class の OOP 機能 -アクセス制御

- アクセス制御
  - 処理の隠蔽
    - 最初の例(コードA’)では、Init 関数をユーザーに見せて、処理の流れを任せてしまった
      - しかも、G1, G2 という変数は誰もが触ることができた。
    - クラスとアクセス制御を使えば、ユーザーに触らせたくない処理を作成可能！

Class の OOP 機能 -アクセス制御

先ほどの例で F1, F2 が public になっていると、変更されてしまう！

```
type
  TSuperClass = class
  public
    F1, F2: Integer;
  protected
    function Proc(i1, i2: Integer): Integer; virtual; abstract;
  public
    constructor Create;
    function Output(i1, i2: Integer): Integer;
  end;
```

もしも public だと、誰からも見える！
つまり、変更されてしまう！

あれ？ F1, F2 に触れるじゃん！
変えちゃお

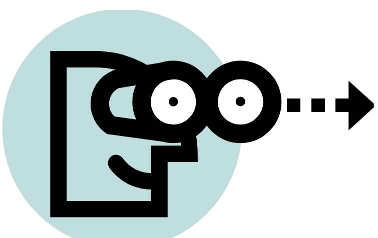
Class の OOP 機能 -アクセス制御

- 高度なカプセル化を可能にする Property
  - Delphi 言語に特有の Property
    - 変数でも、メソッドでもない、その間の存在
    - ユーザーからは変数に見える
    - 実体はメソッド

```
type
  TFoo = class
  private
    FFoo: Integer;
    procedure SetFoo(const iFoo: Integer);
  public
    Bar: Integer;
    property Foo read FFoo write SetFoo;
  end;
```

実装は private にあり、クラス外からは触れない

値を設定される時に実行される



ユーザーからは、両方とも変数に見える
その実、Propety はメソッドとして実装できるため、値を設定される時に範囲チェックなど、値の参照・代入を契機に処理を実行できる。

Class の OOP 機能 -アクセス制御

- たとえば、TControl.Visible プロパティ

```
type
  TControl = class(TComponent)
  private
    FVisible: Boolean;
    procedure SetVisible(Value : Boolean);
  public
    property Visible: Boolean
      read FVisible
      write SetVisible
      stored IsVisibleStored // true のときコンポーネントストリームに格納される
      default True;          // デフォルト値と同じときは格納されない
end;
```

```
procedure TControl.SetVisible(Value: Boolean);
begin
  if FVisible <> Value then
  begin
    VisibleChanging;
    FVisible := Value;
    Perform(CM_VISIBLECHANGED, Ord(Value), 0);
    RequestAlign;
  end;
end;
```

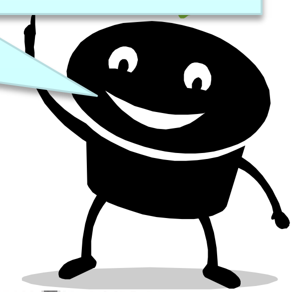
値を代入すると、それを契機にメソッドが実行される

デキる！

プログラマー・ポイント

- アクセス制御
  - 見せるべきデータと、見せてはならないデータを定義する機能
    - プログラムの高品質化に寄与

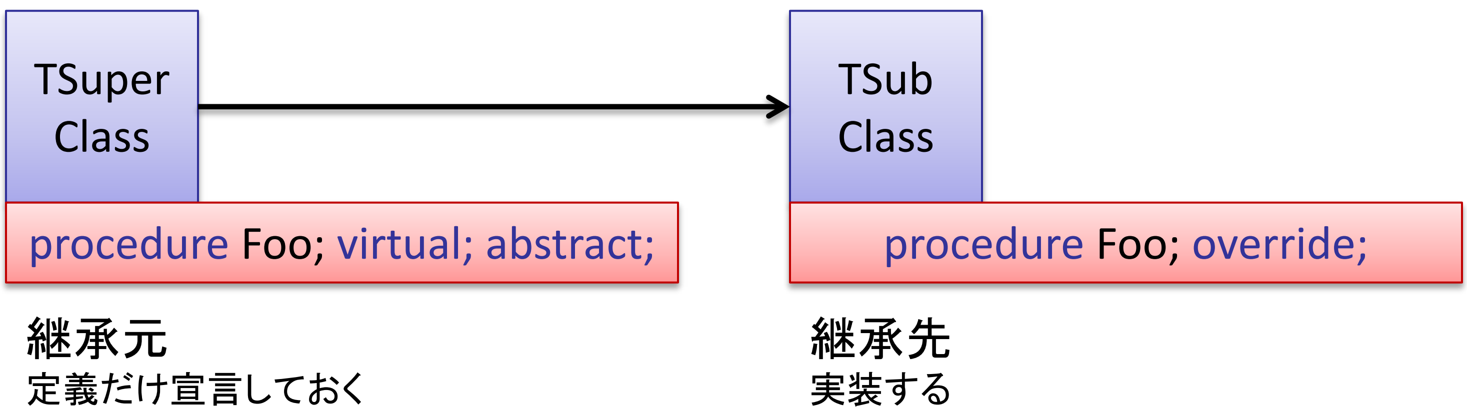
- プロパティ
  - カプセル化、モジュール化を促進する機能
    - アクセス制御の一形態と見なせる



Apendix: Interface

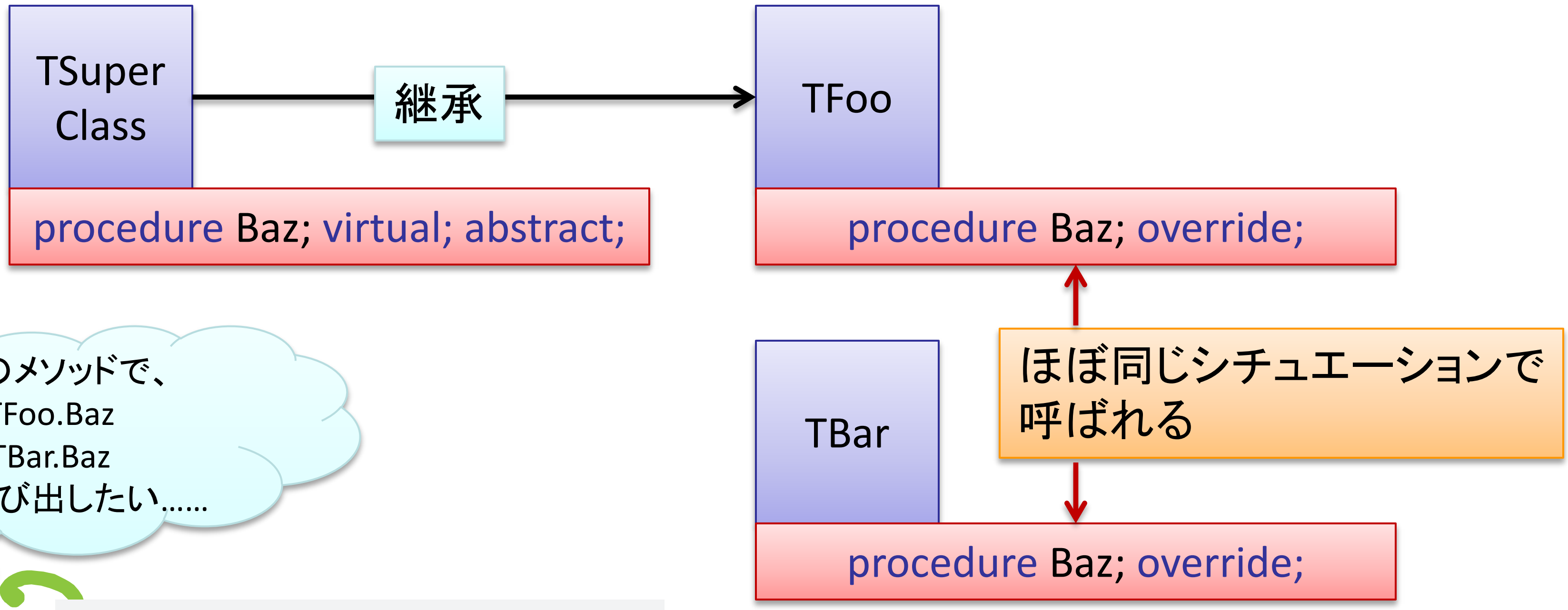
Interface

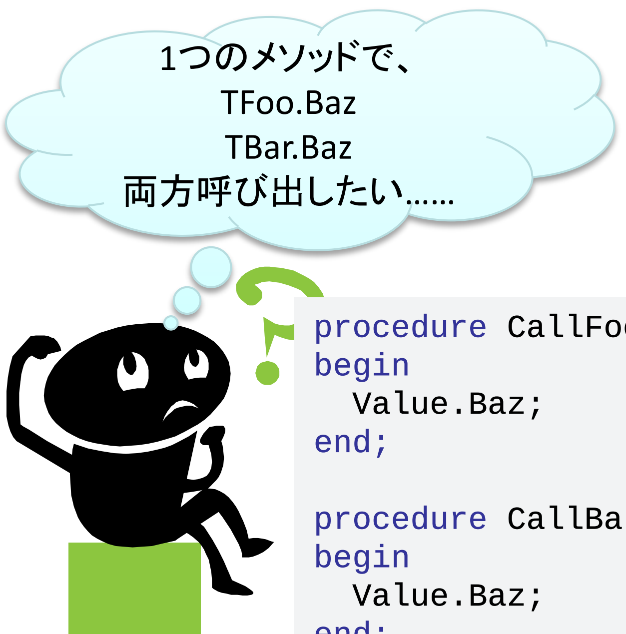
- Class の abstract 指令
  - 継承先で実装されることを期待するメソッドにつける
  - クラスと関連付いているため、親クラスに宣言されていないと使えない



Interface

- 別のクラスが同じような関数を実装して呼び出したい。どうする？





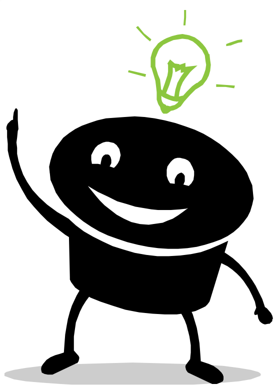

```
procedure CallFoo(Value: TFoo);
begin
  Value.Baz;
end;

procedure CallBar(Value: TBar);
begin
  Value.Baz;
end;
```

クラス（型）が違う為
2つメソッドを作らなくてはならない

Interface

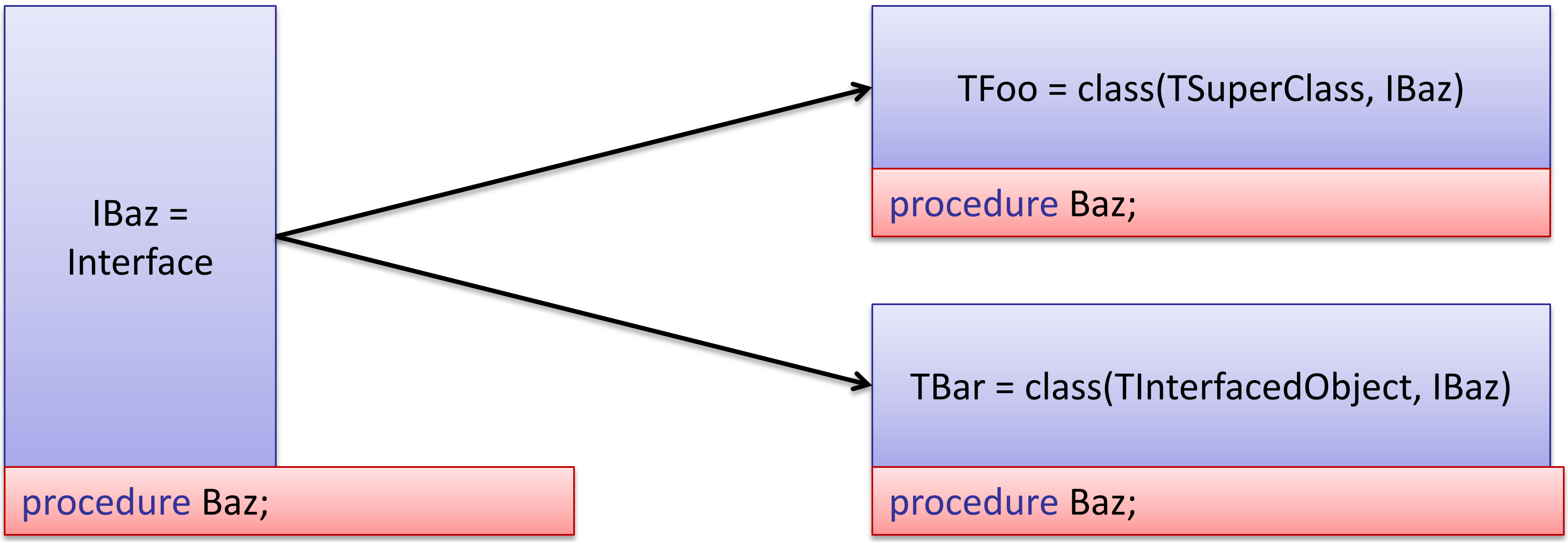
入出力部分を切り出して
それを、各クラスで実装すればいい
んじゃね！！

その機構を Interface と呼ぼう



Interface

- 宣言だけする機構を Interface と呼ぶ



Interface は慣例として接頭辞 "I" を付ける

Interface を実装する場合
class(継承元, インターフェース)
と宣言する。
なお、インターフェースはカンマで区切り複数宣言できる
TBar = class(TObject, IBaz, ITest, ITestTest)

Interface

Interface の例

```
type
  // Interface の定義にはアクセス制御はない
  ITest = interface(IInterface)
    procedure Show;
  end;

  // 歴史的な経緯で TInterfacedObject から継承する
  TTest = class(TInterfacedObject, ITest)
  strict private // ITest として見る場合には意味が無い
    procedure Show;
  end;
```

Delphi の Interface は COM 対応のために導入されたので、
参照カウンタによる自動破棄が必須になる。
そのため、参照カウンタを実装している TInterfacedObject から継承する。

TComponent から派生するコンポーネントは TComponent が参照カウンタを実装しているため、気にする必要は無い。

Interface

Interface の例

```
procedure TTest.Show;
begin
  Writeln('TTest');
end;

procedure OutToConsole(const iT: ITest);
begin
  iT.show;
end;

var
  T: TTest;
begin
  T := TTest.Create;
  try
    OutToConsole(T); // Interface を継承していると、そのまま渡せる
  finally
    T.Free;
  end;
end.
```

Interface

Interface の例

```
type
  // TTest とは異なり TComponent から生成した、全く別のクラス
  TTest2 = class(TComponent, ITest)
  public
    procedure Show;
  end;
```

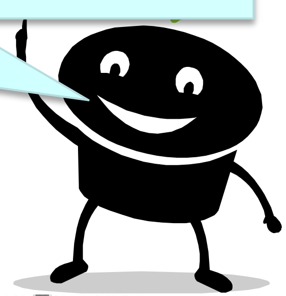
```
var
  T2: TTest2;
begin
  T2 := TTest2.Create;
  try
    OutToConsole(T2); // 別の型だが、同じインターフェースを実装してさえいれば渡せる！
  finally
    T.Free;
  end;
end.
```

Developer Camp

デキる！

プログラマー・ポイント

- Interface
  - メソッドやプロパティの宣言部だけを定義する機構
  - インターフェースを実装していれば、どんな型のインスタンスでも同じインターフェース型の変数に代入できる
    - 利用できるメソッド・プロパティはインターフェースに宣言されたもの



デキる！

プログラマー・ポイント

- 参考文献
  - docwiki
    - http://docwiki.embarcadero.com/RADStudio/XE3/ja/