尾崎 浩司

株式会社ミガロ.

RAD事業部 営業・営業推進課

[Delphi/400]

Delphi/400最新プログラム文法の活用法

略歴
1973年8月16日生まれ
1996年 三重大学工学部卒業
1999年10月 株式会社ミガロ. 入社
1999年10月 システム事業部配属
2013年4月 RAD事業部配属

現在の仕事内容
ミガロ. 製品の営業を担当。これまでのシステム開発経験を活かして、IBM iをご利用のお客様に対して、GUI化、Web化、モバイル化などを提案している。

1.はじめに

Delphi/400は、ビジュアルプログラミングと呼ばれる開発手法でアプリケーションを作成する。ビジュアルプログラミングとは、コンポーネントをフォームに配置し、プロパティを定義したうえで、必要に応じたユーザーのアクション（マウスをクリックする、キーボードで入力するといった操作）に対し、イベントハンドラと呼ばれるプログラムをコーディングしていく手法である。

このDelphi/400のコーディングに使用するのが、Object Pascalである。Object Pascalは、もともと教育用として開発されたPascal言語をオブジェクト指向プログラミングが行えるように拡張したもので、シンプルな文法やデータ型の厳格な型チェックを採用しているのが特徴である。

Delphi/400は、これまでのバージョンアップでさまざまな機能拡張を実施しているが、Ver.2009以降では、文法についても多くの新しい記述方法が追加されている。

本稿ではVer.2009以降に追加された文法について、具体例とともに説明する。

2.文法の高度な機能（Delphi/400 Ver.2009以降）

Delphi/400 Ver.2009では、それ以前のバージョンまでのShift-JISベースであった文字コード体系がUnicodeベースに大きく変更された。

このバージョンでは文字コード体系の変更とともに、ジェネリクスならびに無名メソッドという大きな文法の進化も見られる。以下に、この2つの概要ならびに活用例を説明する。

2-1. ジェネリクスとは

ジェネリクスとは一言でいうと、特定の型に依存しない実装を行うプログラミングスタイルのことである。具体例として、たとえばInteger型の変数A、BとChangeメソッドをもつTIntChangeクラスを考えてみる。【図1】

Changeメソッドは、変数AとBの値をひっくり返すだけの簡単な処理である。このクラスを使用するプログラムの実装例は、【図2】のとおりである。

このプログラムを実行し、ボタンをクリックすると、初期セットされたA＝100、B＝200の値がひっくり返り、画面上には結果として、A＝200、B＝100が表示される。

ここでは変数A、BにInteger型の整数値を使用したが、もしこれをDouble型やString型にしたい場合、それぞれのデータ型用のクラスを追加する。【図3】

【図1】と【図3】を比べると、データ型が異なる以外はまったく同じ処理であるとわかる。このように処理自体は変わらないのに、データ型が異なるためにそれぞれのクラスを作成するとなると、あらゆるデータ型への対応が必要になる。

こうしたケースで便利なのが、ジェネリクスである。【図1】のプログラムを

図1

【宣言部】

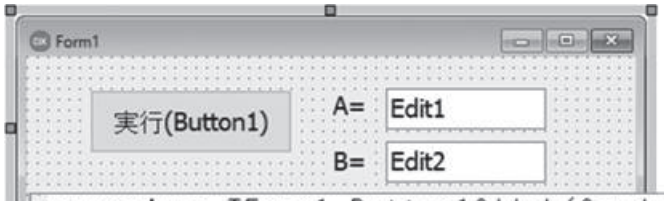
```
type
  TIntChange = class
    A: Integer;
    B: Integer;
    procedure Change;
  end;
```

【宣言部】

```
{ TIntChange }

procedure TIntChange.Change;
var
  Temp: Integer;
begin
  //AとBの値をひっくり返す
  Temp := B;
  B := A;
  A := Temp;
end;
```

図2




```
procedure TForm1.Button1Click(Sender: TObject);
var
  obj: TIntChange;
begin
  //オブジェクト生成
  obj := TIntChange.Create;

  //値をセット
  obj.A := 100;
  obj.B := 200;

  //値の入れ替えを実施
  obj.Change;

  //結果を出力
  Edit1.Text := IntToStr(obj.A);  // <--- A=200
  Edit2.Text := IntToStr(obj.B);  // <--- B=100

  //オブジェクト破棄
  obj.Free;
end;
```

元にジェネリクスを使用したのが、【図4】である。

宣言部を見ると“＜T＞”となっているが、これが仮のデータ型を表しており、Tという名前で宣言されている（このシンボルTは慣例としてよく利用されるが、シンボルとして有効な名称であれば制限はない）。

このようにクラス上では仮のデータ型で宣言や実装を行い、クラスを使用するプログラム側で使用したいときに、実際のデータ型を指定できる。【図5】

この仕組みを使用すれば、どんなデータ型で処理が必要となっても同じクラスが利用できる。これがジェネリクスと呼ばれるものである。

2-2. ジェネリクス活用例

以下に、活用方法について考察する。最もよく使用されるのがコレクションである。

コレクションとは、複数の要素の集まりのことである。Delphi/400でおそらく一番よく使用されているコレクションは、TStringListである。TStringListは、文字列のリストを扱うクラスである。このコレクションを使用すると、文字列を動的配列として扱える。【図6】

ここではリストに対してAddメソッドを使用することで、文字列をリストに追加している。追加されたリストは、配列と同じように要素番号を指定することで、各要素値が取得できることがわかる。

ジェネリクスコレクションクラスであるTList＜T＞を使用すると、これと同じようなことが任意のデータ型で行える。ジェネリクスのコレクションを使用するには、uses節にGenerics.Collectionsを追加すればよい。このユニットには、TList＜T＞をはじめとするいくつかのジェネリクスコレクションクラスが定義されているので、これらを使用できる。

TList＜T＞を使用したリストのプログラムは、【図7】のとおりである。

このプログラムではTList＜Integer＞と定義しているので、Integer型の値をリストとして扱える。【図6】と比較すれば明らかだが、TStringListの場合とまったく同じ手法で、数値に対する動的配列が実現できる。Addメソッドで、Integer型の値を直接リストにセットしている。

便利なのは、Addメソッドには Integer型の値以外はセットできないことである。ジェネリクスで実際の型が決定すると、その型のみが使用できる。

さらに配列に対してリストを使用するメリットの1つとして、ソートが簡単に行える点がある。【図7】で、リストに値を追加した後に、「iList.Sort;」と1行追加すると、整数値の昇順にリストが並び替わる。ソートを配列で実現しようとすると、ロジックを作成せねばならないので、こうした場合にリストを使用するメリットがある。

もう1つの例は、TDictionary＜TKey,TValue＞である。このクラスはキーと値のセットをコレクションとして扱う。こちらも実装例を、【図8】で説明する。

このプログラムでは、キーにはString型を、値にはTCustomer型（Record型）を指定している。TDictionaryの場合も、コレクションの追加はAddメソッドで可能である。要素へのアクセスは、キー値を指定すればよい。

またこのコレクションクラスは、データの検索も容易である。たとえばTryGetValueメソッドを使用すると、存在しないキーを指定した場合、結果がFalseとなるので、入力妥当性チェックにも活用できる。リストの場合と違い、ディクショナリはキーを指定して任意のデータにアクセスできるので、応用範囲が広い。

ここまで、TList＜T＞ならびにTDictionary＜TKey,TValue＞を説明したが、これらのコレクションは使用時のデータ型に、値型を想定したものである。クラス型オブジェクトを想定したTObjectList＜T: class＞やTObjectDictionary＜TKey,TValue＞も用意されているので、用途に合わせて使い分けると効果的である。

2-3. 無名メソッドとは

無名メソッドとはその名のとおり、名前がついていないprocedureやfunctionのことである。通常変数には、値をセットできるが、無名メソッドを使用すると、手続きや関数自体を変数にセットできる。無名メソッドの簡単な使用例を、【図9】で説明する。

まず宣言部を確認する。ここでは、1つのString型引数をもつ手続き型の無名メソッドが保持できるデータ型として、TStrProc型を宣言している。

次に実装部を見ると、変数宣言部分（var）で、TStrProc型の変数pProcを宣言しているのがわかる。そして、この無名メソッド変数pProcに対して、名前のない手続き（procedure）を代入している。こう記述することで、通常の値を変数に代入するのとまったく同じ記述方法により、手続きや関数を変数に代入できる。

なお、無名メソッド変数に手続きや関数を代入した時点では、まだ無名メソッドは実行されない。実際に無名メソッドが実行されるのは、「pProc（‘テスト’）;」のように変数を使用したときである。

このように、無名メソッドは変数に代入できるのだが、それだけではなく、手続きや関数の引数に無名メソッドを渡すこともできる。具体例を、【図10】で説明する。

ここでの宣言部では、Integer型の引数を2つもつ関数型の無名メソッドが保持できるデータ型として、TCalcFunc型を宣言している。

実装部には、Calculate手続きを作成しているが、このサブルーチンは、2つの整数の引数とともにTCalcFunc型の引数を使用しているのがわかる。

つまりこのサブルーチンは、呼び出し側で定義された無名メソッドを受け取って処理を実行する。Button2のOnClickイベントでは、Calculate手続きを2回呼び出している。それぞれ引数として、異なる2つの値とともに、異なる無名メソッドを渡している。

このプログラムを実行して、Button2をクリックすると、メッセージボックスに計算式の異なる2つの処理結果が表示される。【図11】

このように無名メソッドを使用すると、通常の変数等と同じように手続きや関数を引数として渡せる。

2-4. 無名メソッド活用例

次に、名前をもたない無名メソッドについて、サブルーチンに対して無名メソッドを渡す仕組みの活用例を説明する。

図3

【宣言部】

```
type
  // Double型用
  TFloatChange = class
    A: Double;
    B: Double;
    procedure Change;
  end;

  // String型用
  TStringChange = class
    A: String;
    B: String;
    procedure Change;
  end;
```

【実装部】

```
{ TFloatChange }

procedure TFloatChange.Change;
var
  Temp: Double;
begin
  //AとBの値をひっくり返す
  Temp := B;
  B := A;
  A := Temp;
end;

{ TStringChange }

procedure TStringChange.Change;
var
  Temp: String;
begin
  //AとBの値をひっくり返す
  Temp := B;
  B := A;
  A := Temp;
end;
```

図4

【宣言部】

```
type
  // ジェネリッククラス
  TChange<T> = class
    A: T;
    B: T;
    procedure Change;
  end;
```

【実装部】

```
{ TChange<T> }

procedure TChange<T>.Change;
var
  Temp: T;
begin
  //AとBの値をひっくり返す
  Temp := B;
  B := A;
  A := Temp;
end;
```

IBM i（AS/400）をはじめ、各種データベースに対して更新処理を行うような場合、たとえばdbExpress接続では、【図12】のような処理を記述することが多い。

このプログラムのように、データベースへの更新処理は大きく次の3つから構成される。

①トランザクションの開始
②データの登録／変更／削除等の更新処理
③トランザクションのコミット（②でエラーの場合ロールバック）

たとえば受注と売上の各更新処理がある場合、一般にそれぞれの更新処理で①②③を記述する。しかしデータベースへの更新内容が異なっても、②が異なるだけで、①と③は共通の処理となる。【図13】

この場合に役立つのが、無名メソッドである。②の部分を無名メソッドとして、データベース更新処理の共通サブルーチンの引数とすればよい。【図12】のプログラムを修正し、無名メソッドを使用する例を、【図14】で説明する

このプログラム例では、引数のデータ型をTProc型としているが、これは引数をもたない手続き型の無名メソッド用にあらかじめ用意されたデータ型なので、これを使用すれば、とくに型の宣言をせずに無名メソッドが使用できる。

【図14】で定義したDataUpdateメソッドを呼び出すプログラムは、【図15】のとおりである。ここでは更新処理自体の無名メソッドを引数にセットして、DataUpdateメソッドを呼び出しているのがわかる。

以上、無名メソッドの使用例として、データベースの更新処理を説明したが、ほかによく使用される無名メソッドの活用方法として、TThread.CreateAnonymousThreadと無名メソッドを使用したスレッド（並列）処理がある。

これについては、2015年版ミガロ.テクニカルレポートにある『マルチスレッドを使用したレスポンスタイム向上』で詳しく説明しているので、そちらを参照してほしい。

3.最新文法活用TIPS（Delphi/400 Ver.2010以降）

ここからは、Delphi/400でプログラミングする際に便利な2つの文法活用TIPSを説明する。どちらもプログラム開発で非常に有用なので、ぜひ参考にしてほしい。

3-1. レコードヘルパ、クラスヘルパによる既存機能の拡張（Delphi/400 Ver.2010以降）

Delphi/400は、データ型の取り扱いが厳格である。

たとえば、String型とInteger型とで相互代入はできない。Integer型の変数iに、演算結果として整数値123がセットされていると、「ShowMessage（i）;」という手続きは、コンパイルエラーとなる。

これは、ShowMessage手続きの引数がString型を要求しているにもかかわらず、Integer型の変数をセットしているから発生するエラーである。では、プログラムのなかで演算された結果をメッセージボックスに表示するには、どうすればよいだろうか。

この場合、データ変換関数を使用するのが一般的である。先の例では、「ShowMessage（IntToStr（i））;」と記述すれば、演算結果をメッセージボックスに出力できる。このようにデータ変換の機能がサブルーチンとして定義されているので、それを利用する。しかしデータ変換を行うのに、その都度サブルーチンを使用するのは、いささかプログラミングが面倒である。

そこでDelphi/400 Ver.2010以降には、レコードヘルパという機能が用意されている。これは特定のレコードに対して、機能拡張をサポートする。通常、既存機能の拡張というと、オブジェクトクラスに対して「継承」を利用するのが一般的だが、レコードヘルパを使用すると、String型やInteger型といった組み込みデータ型に対しても機能を拡張できる。

とくにDelphi/400 Ver.XE5以降のObject Pascalでは、TIntegerHelperやTStringHelperといった定義済みのレコードヘルパクラスが用意されているので、既存機能の拡張を意識することなく、そのまま使用できる。

たとえばInteger型のレコードヘルパであるTIntegerHelperを使用すると、変数iに対して、「ShowMessage（i.ToString）;」のように記述できる。このようにレコードヘルパを使用すると、変数などに対して直接メソッドが記述できるので、コードの見通しがよくなる。

なお、このようにすぐに使用できるレコードヘルパは、SysUtilsユニットに定義されている。Delphiの開発元であるエンバカデロ・テクノロジーズ社が提供するオンラインヘルプ（DocWiki）のSysUtilsユニットページを参照し、「レコードヘルパ」でページ検索すれば、定義済みのレコードヘルパを確認できる。【図16】

このレコードヘルパは、独自の定義も可能である。

IBM i（AS/400）を活用するアプリケーションでは、日付値を示すデータベースのフィールドとして数値8桁を定義することが多い。しかしDelphi/400では、日付値はTDate型を使用するのが一般的である。そこで以下に、TDate型の日付値をInteger型に変換するレコードヘルパの作成手順を説明する。

まず、宣言部にTDate型のレコードヘルパクラスと、そのなかに機能となるメソッド（ToIntegerメソッド）を宣言する【図17】。宣言が完了したら、［Ctrl］＋［Shift］＋［C］を押下し、実装部のテンプレートを作成のうえ、メソッド内に実装を記述する。【図18】

【図18】の実装例を見ると、Selfというキーワードがあるのがわかる。このSelfには、メソッドが実行される際のTDate型の日付値がセットされる。ここではSelfで指定された日付値に対し、FormatDateTime関数を使用して、いったん8桁の文字列に変換したのち、StrToInt関数で整数値に変換している。

レコードヘルパが完成すれば、使用方法は簡単である。たとえばフォーム上にあるTDateTimePicker（日付入力コンポーネント）にセットされたTDate型の値を、Integer型の値として取得するのは、【図19】のようなコードで記述できる。

TDate型の値に対し、直接ToInteger

図5

```
procedure TForm1.Button1Click(Sender: TObject);
var
  IntObj: TChange<Integer>;
  StrObj: TChange<String>;
begin
  //オブジェクト生成
  IntObj := TChange<Integer>.Create;
  StrObj := TChange<String>.Create;

  //値をセット
  IntObj.A := 100;
  IntObj.B := 200;

  StrObj.A := 'ABC';
  StrObj.B := 'DEF';

  //値の入れ替えを実施
  IntObj.Change;
  StrObj.Change;

  //結果を出力
  Edit1.Text := IntToStr(IntObj.A);  // <--- A=200
  Edit2.Text := IntToStr(IntObj.B);  // <--- B=100

  Edit3.Text := StrObj.A;            // <--- A='DEF'
  Edit4.Text := StrObj.B;            // <--- B='ABC'

  //オブジェクト破棄
  IntObj.Free;
  StrObj.Free;
end;
```

図6

```
procedure TForm1.Button1Click(Sender: TObject);
var
  sList: TStringList;   //文字列リスト
  i: Integer;
begin
  //文字列リストを生成
  sList := TStringList.Create;

  //文字列のリストを追加
  sList.Add('文字列１');
  sList.Add('文字列２');
  sList.Add('文字列３');

  //配列の要素にアクセス
  for i := 0 to sList.Count - 1 do
    ShowMessage(sList[i]);

  //リストの解放
  sList.Free;
end;
```

メソッドを記述してInteger型の値に変換できている。もちろん同様のことは、TDate型の値をInteger型に変換するためのデータ変換関数（function）を作成し、その関数を使用しても実装できるが、レコードヘルパを使用したコードのほうが読みやすいのは一目瞭然である。

ここで説明した例は、TDate型をInteger型に変換するレコードヘルパだが、もちろんInteger型をTDate型に変換するレコードヘルパも作成可能である。その場合、【図20】のような処理が考えられる。

ただし、レコードヘルパは1つのデータ型に対して1つしか使用できない点に注意が必要である。Delphi/400 Ver.XE5以降には、あらかじめ定義済みのTIntegerHelperが存在するので、【図20】の宣言を参照するプログラムでは、TIntegerHelperに定義されたメソッドが使用できなくなる。

すでに存在するデータ型のレコードヘルパと共用したい場合は、Integer型に対するエリアス（別名）を定義すればよい【図21】。ここでは、Integer型のエリアスとしてTDateInt型を定義している。それにより、独自に定義した【図21】のレコードヘルパを使用する場合には、TDateInt型でキャストすればよい。

たとえば、日付整数値が格納されたInteger型の変数iに対しては、「TDateInt（i）.ToDate」のように記述できる。【図22】

説明したのは組み込みデータ型に対するレコードヘルパだが、クラスに対してもクラスヘルパが使用可能である。クラスヘルパを使用すると、たとえば標準のコンポーネントに対して簡単に機能を追加できる。つまり独自の継承コンポーネントを作成することなく、機能拡張できるわけだ。

TEditの親クラスであるTCustomEditに対して、データ型の変換機能を実装した例を、【図23】で説明する。

このクラスヘルパを参照するプログラムでは、TCustomEditを継承したTEdit等の入出力コンポーネントに対し、直接TDate型やInteger型で値の取得ならびに代入が可能になる。【図24】

このようにレコードヘルパやクラスヘルパを作成すると、元のレコードやクラスに一切手を加えることなく、新しい機能が追加できるので、汎用ユニットとして定義できる。

3-2. ランタイムライブラリ（RTL）を活用したプログラム作成法

Delphi/400でプログラムを記述する際、前述したIntToStr関数などのデータ変換関数を使用することが多い。では、なぜ作成するプログラムで、IntToStr関数が使用できるのだろうか。

VCLとFireMonkeyのそれぞれで、新規プロジェクトを作成し、作成直後のForm1ユニット（Unit1.pas）を見ると、どちらもほぼ同じ構成であるのがわかる。【図25】【図26】

構成のなかで異なるのは、uses節の部分である。Object Pascalでuses節は、プログラムの実行に必要なほかの参照ユニットを表している。VCLかFireMonkeyかで、使用するビジュアルコンポーネントのフレームワークが異なるので内容も違っているのだが、よく見るとSystem.SysUtils、System.Variants、System.Classesの各ユニットはどちらのプロジェクトにも含まれているのがわかる。

冒頭のIntToStr関数は、System.SysUtilsユニットに定義された関数である。つまり、IntToStr関数が使用できるのは、ユニット参照されているからである。このIntToStr関数のようなアプリケーション開発で一般に使用されるサブルーチンは、ライブラリとして提供されており、Delphiランタイムライブラリ（RTL）と呼ばれている。

このRTLには多彩な機能が実装されており、プログラムで多様な機能を実現できる。RTLの多くはSystemユニットスコープに定義されており、DocWikiを参照しても多数のユニットが用意されている（http://docwiki.embarcadero.com/Libraries/Seattle/ja/System）。【図27】

このなかで、知っておくと役立つRTLを以下に説明する。

（1）System.IOUtils（Delphi/400Ver.2010以降）

System.IOUtilsは、Delphi/400 Ver.2010以降に追加されたRTLである。以前はディレクトリやファイル操作のプログラミングが少し面倒であったが、このユニットが追加されたことで扱いが簡単になった。

まず、TDirectoryクラスについて説明しよう。TDirectoryはディレクトリを操作するクラスである。たとえば、このクラスにはクラスメソッドDeleteが用意されており、これを使用すると特定フォルダを簡単に削除できる。【図28】

System.IOUtilsは標準でuses節に含まれていないので、個別に追加する。こうすれば、あとはクラスメソッドを呼び出すだけで使用できる。

このメソッドが便利な点は、フォルダ内にサブフォルダやファイルが存在していたとしても、一括削除できることだ。Delphi/400 Ver.2009以前の場合、同じ処理を実現するのに次のようなサブルーチンを作成する必要があった。

［フォルダ削除サブルーチンの処理ロジック］

①削除しようとするフォルダ内のすべてのファイルおよびフォルダを検索する
②ファイルならばDeleteFileを用いて削除し、フォルダならば再帰的に自身の関数処理を呼び出す
③フォルダの中身が空になったところで、RemoveDirectoryを用いてフォルダを削除する

TDirectoryクラスの追加により、簡単にフォルダ削除ができるようになった。ほかにもフォルダのコピー（TDirectory.Copy（）メソッド）や移動（TDirectory.Move（）メソッド）も用意されている。

次に、フォルダ内に含まれるファイルを一覧取得する処理を考えよう。これもDelphi/400 Ver.2009以前では、FindFirst関数やFindNext関数を使用しながらファイル名を取得し、サブフォルダについては、再帰処理を行う必要があった。しかしSystem.IOUtilsを使用すると、TDirectory.GetFilesメソッドで容易に取得できる。【図29】

TDirectory.GetFilesメソッドの引数に検索オプション（soAllDirectories）を付与するだけで、サブフォルダまで含めた一括検索ができる。

また【図29】のソースでは、for in

図7

```
uses Generics.Collections;

procedure TForm1.Button1Click(Sender: TObject); ...

procedure TForm1.Button2Click(Sender: TObject);
var
  iList: TList<Integer>; //ジェネリクスリスト
  i: Integer;
begin
  //整数値用リストを作成
  iList := TList<Integer>.Create;

  //整数値のリストを追加
  iList.Add(234);
  iList.Add(567);
  iList.Add(111);

  //配列の要素にアクセス
  for i := 0 to iList.Count - 1 do
    ShowMessage(IntToStr(iList[i]));

  //リストの解放
  iList.Free;
end;
```

図8

【宣言部】

```
type
  //顧客レコード
  TCustomer = record
    sName: String;  //顧客名
    sAddr: String;  //住所
    sTel : String;  //電話番号
  end;
```

【実装部】

```
uses Generics.Collections;

procedure TForm1.Button3Click(Sender: TObject);
var
  sDict: TDictionary<String, TCustomer>;
  rCust: TCustomer;
begin
  //ディクショナリを作成
  sDict := TDictionary<String, TCustomer>.Create;

  //ディクショナリに追加
  rCust.sName := '株式会社ミガロ.';
  rCust.sAddr := '大阪市浪速区湊町';
  rCust.sTel  := '06-1234-5678';
  sDict.Add('00001', rCust);

  rCust.sName := 'エンバカデロ';
  rCust.sAddr := '東京都文京区';
  rCust.sTel  := '03-1111-2222';
  sDict.Add('00002', rCust);

  rCust.sName := '日本アイ・ビー・エム株式会社';
  rCust.sAddr := '東京都中央区日本橋';
  rCust.sTel  := '03-3333-4444';
  sDict.Add('00003', rCust);

  //キーを指定してデータにアクセス
  with sDict['00002'] do
  begin
    ShowMessage(sName + ' ' + sAddr + ' ' + sTel);
  end;

  //データの検索
  if sDict.TryGetValue(Edit1.Text, rCust) then
    ShowMessage(rCust.sName)
  else
    ShowMessage('顧客コードが存在しません');

  //ディクショナリの解放
  sDict.Free;
end;
```

do ループを使用している点にも注目してほしい。従来からのカウンタ変数を使用した for ループだけでなく、このような配列などを使用した for ループ処理も記述できる。

ほかにもパス名、フォルダ名、ファイル名を操作する TPath クラスや、ファイルを操作する TFile クラスが用意されている。Delphi/400 Ver.2009 以前では、ファイルをコピーする関数が用意されておらず、Win32API を使用する必要があったが、TFile.Copy（）メソッドを使用すれば、API を意識せず簡単に実装できる。

（2）System.RegularExpressions（Delphi/400 Ver.XE 以降）

次に説明する System.RegularExpressions は、Delphi/400 Ver.XE 以降で使用可能な RTL で、いわゆる正規表現を実現する。

正規表現とは、文字列の集合を 1 つの文字列で表現する方法で、たとえば郵便番号やメールアドレスなど、特定の文字列パターンで表せるものをチェックするのに利用することが多い。これを使用したプログラムの例を、【図 30】で説明する。

ここでは TRegEx クラスの IsMatch メソッドを使用すると、文字列が指定された正規表現とマッチするかを確認できる。OnChange イベントなどで比較すると、入力途中の整合性チェックに活用できる。

RTL はほかにもいろいろあるが、知っていると便利なユニットを以下にいくつか説明する。

System.StrUtils は文字列処理関数が含まれており、たとえば LeftStr、MidStr、RightStr 関数を使用すると、Copy 関数を使わなくても、任意の位置の文字列を容易に取得できる。

System.DateUtils は、日付処理関数が含まれている。月末日を取得するのに、従来は翌月 1 日の日付 -1 という取得方法が一般的であったが、EndOfAMonth 関数を使用すると容易に取得できる。

System.Math は数値演算関数が含まれており、たとえば四捨五入は SimpleRoundTo 関数で容易に実行できる。

さらに Delphi/400 Ver.XE3 では、ZIP ファイルを扱うための System.Zip が、Delphi/400 Ver.XE7 では、JSON 文字列を扱うための System.JSON やインターネットエンコード、デコード処理を行うための System.NetEncoding が追加されており、バージョンアップのたびに便利な RTL が拡充されている。

4.まとめ

本稿では、Delphi/400 のコーディングで使用される Object Pascal の新しい文法に関するテクニックを取り上げて説明した。

Delphi/400 のコーディングに普段から使用している Object Pascal だが、本稿執筆に際してあらためて文法を調べてみると、Delphi/400 Ver.2009 以降で文法が大きく強化されていることがわかった。

本稿で説明した各文法は、いろいろな局面で活用できるので、ぜひ今後のアプリケーション開発時のコーディング技法としてチャレンジし、開発の幅を広げていただきたい。

M

図9

【宣言部】

```
type
  TStrProc = reference to procedure(sStr: String);
```

【実装部】

```
procedure TForm1.Button1Click(Sender: TObject);
var
  pProc: TStrProc;
begin
  //無名メソッドを変数pProcに代入
  pProc := procedure(sStr: String)
  begin
    ShowMessage(sStr);
  end;

  //変数pProcの使用
  pProc('テスト');
end;
```

図10

【宣言部】

```
type
  TCalcFunc = reference to function(a, b: Integer): Integer;
```

【実装部】

```
procedure Calculate(iValue1, iValue2: Integer; Calc: TCalcFunc);
var
  iRet: Integer;
begin
  //受け取った無名メソッドを実行
  iRet := Calc(iValue1, iValue2);

  //処理結果を出力
  ShowMessage(IntToStr(iRet));
end;

procedure TForm1.Button2Click(Sender: TObject);
begin

  //足し算を行う無名メソッドを引数にセット
  Calculate(3, 4, function(a, b: Integer): Integer
                  begin
                    Result := a + b
                  end);

  //掛け算を行う無名メソッドを引数にセット
  Calculate(6, 8, function(a, b: Integer): Integer
                  begin
                    Result := a * b
                  end);
end;
```

無名メソッドが引数となっている

無名メソッドを引数として
サブルーチンを実行
(異なるメソッド(条件)で
プログラムが実行できる)

図11

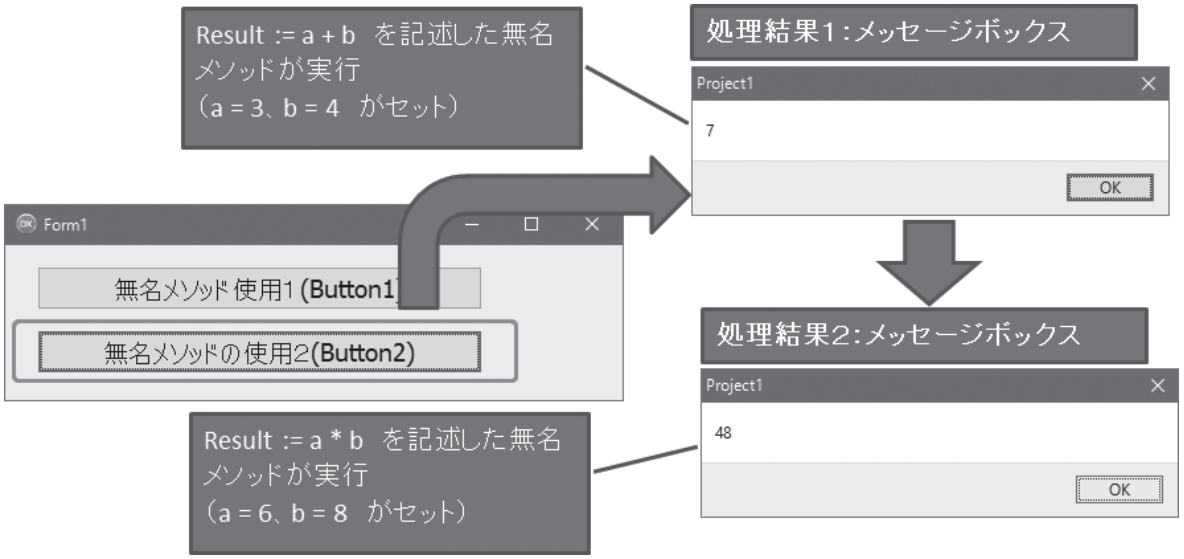


図12

```
procedure TdmMain.DataUpdate;
var
  dbTran: TDBXTransaction; //トランザクション変数
begin

  //①トランザクションの開始
  dbTran := SQLConnection1.BeginTransaction;
  try
    //②データの登録/変更/削除等の更新処理
    SQLQuery1.SQL.Clear;
    SQLQuery1.SQL.Add('UPDATE PNAME SET PNAME = ''テスト'' ');
    SQLQuery1.SQL.Add('WHERE PCODE = 1 ');

    SQLQuery1.ExecSQL;

    //③トランザクションのコミット
    SQLConnection1.CommitFreeAndNil(dbTran);

  except
    //例外処理
    //ロールバックを行う
    SQLConnection1.RollbackFreeAndNil(dbTran);
    raise;
  end;
end;
```

図13

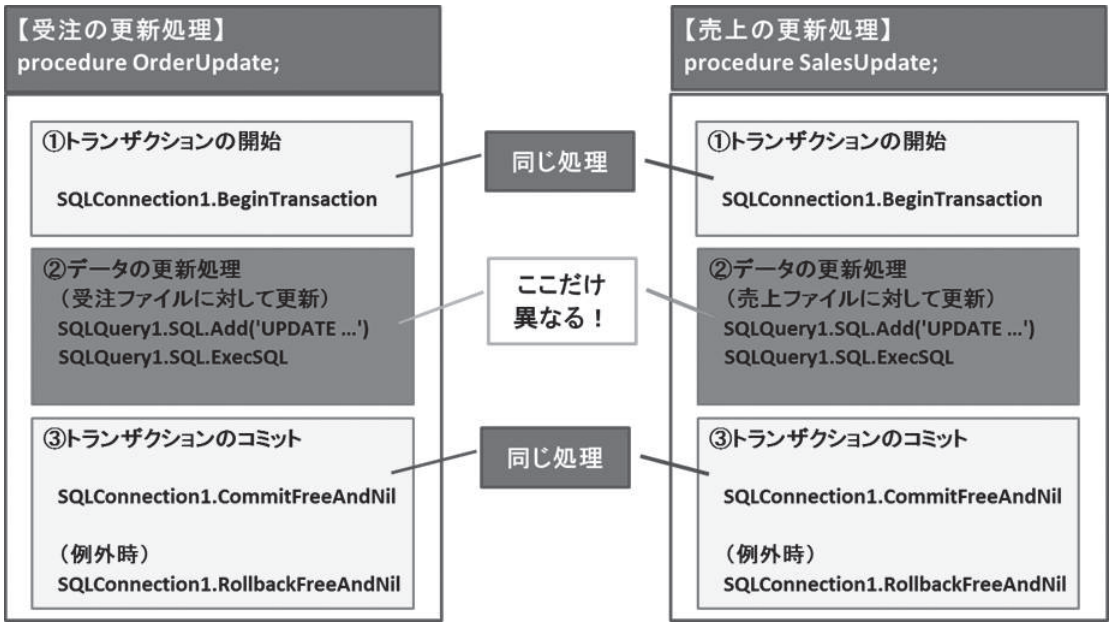


図14

```
procedure TdmMain.DataUpdate(SQLProc: TProc);
var
  dbTran: TDBXTransaction; //トランザクション変数
begin
  //①トランザクションの開始
  dbTran := SQLConnection1.BeginTransaction;
  try
    //②データの登録/変更/削除等の更新処理
    SQLProc;

    //③トランザクションのコミット
    SQLConnection1.CommitFreeAndNil(dbTran);

  except
    //例外処理
    //ロールバックを行う
    SQLConnection1.RollbackFreeAndNil(dbTran);
    raise
  end;
end;
```

図15

```
procedure TdmMain.OrderUpdate;
begin
  //データの更新部分を無名メソッドで指定
  DataUpdate(procedure ()
             begin
               //受注データの更新処理
               SQLQuery1.SQL.Clear;
               SQLQuery1.SQL.Add('UPDATE ... ');
                        :
               SQLQuery1.ExecSQL;
             end
             );
end;
```

図16

図17

【宣言部】
TDate型のレコードヘルパクラスとして、TDateToIntHelper型を宣言。
メソッドとして、整数値に変換する関数を定義。

```
type
  TDateToIntHelper = record helper for TDate
    function ToInteger: Integer;
  end;
```

図18

【実装部】
TDateToIntHelper型のメソッドであるToIntegerのロジックを記述。
元のレコード値(Self)に対して、整数値に変換した値をResultにセット。

```
{ TDateToIntHelper }

function TDateToIntHelper.ToInteger: Integer;
begin
  if Self <> 0 then
    Result := StrToInt(FormatDateTime('YYYYMMDD', Self))
  else
    Result := 0;
end;
```

図19

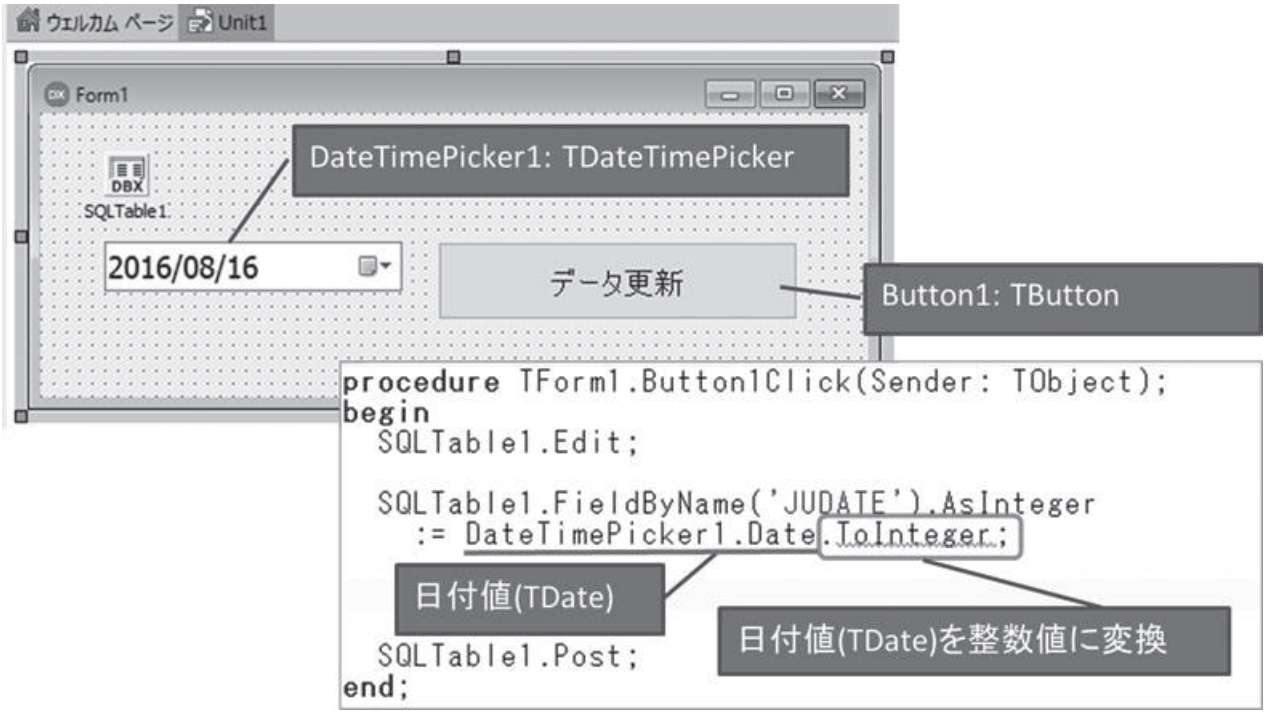


図20

【宣言部】
Integer型のレコードヘルパクラスとして、TIntToDateHelper型を宣言。

```
type
  TIntToDateHelper = record helper for Integer
    function ToDate: TDate;
  end;
```

【実装部】
TIntToDateHelper型のToDateメソッドを記述。

```
{ TIntToDateHelper }

function TIntToDateHelper.ToDate: TDate;
begin
  if Self <> 0 then
    Result := StrToDate(FormatFloat('0000/00/00', Self))
  else
    Result := 0;
end;
```

図21

【宣言部】
Integer型のエリアスとしてTDateInt型を宣言し、TDateInt型のレコードヘルパとして、TIntToDateHelper型を宣言。

```
type
  TDateInt = type Integer;     //Integer型のエリアス

  TIntToDateHelper = record helper for TDateInt
    function ToDate: TDate;
  end;
```

図22

【使用例】

```
procedure TForm1.Button4Click(Sender: TObject);
var
  i: Integer;
  d: TDate;
  s: String;
begin
  i := 20160816;  // 日付整数値

  d := TDateInt(i).ToDate;    //TDateToIntHelper のToDateメソッド
  s := i.ToString;            //TIntegerHelper のToStringメソッド
end;
```

図23

【宣言部】
TCustomEditのクラスヘルパとして、TCustomEditHelper型を宣言。
整数値と、日付値に対するプロパティを定義。

```
type
  TCustomEditHelper = class helper for TCustomEdit
  private
    function GetAsInteger: Integer;
    procedure SetAsInteger(const Value: Integer);
    function GetAsDate: TDate;
    procedure SetAsDate(const Value: TDate);
  public
    property AsDate: TDate read GetAsDate write SetAsDate;
    property AsInteger: Integer read GetAsInteger write SetAsInteger;
  end;
```

【実装部】
各プロパティに対する取得(Get)、書込み(Set)メソッドを実装。

```
{ TCustomEditHelper }

function TCustomEditHelper.GetAsDate: TDate;
begin
  Result := StrToDate(Self.Text);
end;

function TCustomEditHelper.GetAsInteger: Integer;
begin
  Result := StrToInt(Self.Text);
end;

procedure TCustomEditHelper.SetAsDate(const Value: TDate);
begin
  Self.Text := DateToStr(Value);
end;

procedure TCustomEditHelper.SetAsInteger(const Value: Integer);
begin
  Self.Text := IntToStr(Value);
end;
```

図24

【使用例】

```
procedure TForm1.Button1Click(Sender: TObject);
begin
  Edit1.AsInteger := 12345678;    // Editに数値をセット
  Edit2.AsDate := Date;           // Editに日付値をセット
end;
```

図25

【VCLフォームアプリケーション】

```
unit Unit1;

interface

uses
  Winapi.Windows, Winapi.Messages, System.SysUtils, System.Variants,
  System.Classes, Vcl.Graphics, Vcl.Controls, Vcl.Forms, Vcl.Dialogs;

type
  TForm1 = class(TForm)
  private
    { Private 宣言 }
  public
    { Public 宣言 }
  end;

var
  Form1: TForm1;

implementation

{$R *.dfm}

end.
```

図26

【マルチデバイスアプリケーション】
(FireMonkey)

```
unit Unit1;

interface

uses
  System.SysUtils, System.Types, System.UITypes, System.Classes, System.Variants,
  FMX.Types, FMX.Controls, FMX.Forms, FMX.Graphics, FMX.Dialogs;

type
  TForm1 = class(TForm)
  private
    { private 宣言 }
  public
    { public 宣言 }
  end;

var
  Form1: TForm1;

implementation

{$R *.fmx}

end.
```

図27


http://docwiki.embarcadero.c...
System - RAD Studio API D...
Trend ツールバー
アカウント作成
ログイン
embarcadero
ページ
ソースを表示
履歴
表示設定
検索
表示
検索
RAD STUDIO SEATTLE
メイン ページ - Seattle ライブラリ リファレンス
ユニット リスト
Seattle のトピック
Seattle のコード例
新しいバージョン
Berlin ライブラリ
以前のバージョン
XE8 ライブラリ
XE7 ライブラリ
XE6 ライブラリ
XE5 ライブラリ
XE4 ライブラリ
XE3 ライブラリ
XE2 ライブラリ
System
親: ユニット リスト
Delphi ランタイム ライブラリ (RTL) のユニット、クラス、関数、型、変数、定数が含まれています。また、System.Bindings、System.Generics、System.Win の各ユニット スコープも含まれています。System ユニット スコープと System ユニットは同じである点に注意してください。System ユニットで公開されている Delphi 専用組み込みルーチンの一覧については、「Delphi 組み込みルーチン」を参照してください。
目次 [非表示]
1 ユニット スコープ
2 ユニット
3 クラス
4 ルーチン
5 型
6 変数
7 定数
ユニット スコープ
System.Analytics
System.Beacon
System.Bindings
LiveBinding フレームワーク エンジンのユニット スコープです。LiveBinding エンジンを実装しているユニットが含まれています。
System.Bluetooth
アプリケーションの動作デバイスの Bluetooth 機能を使用して、リモート デバイスで動作するアプリケーションに接続するためのクラスが用意されています。
ジェネリックス (後でインスタンス化されてパラメータとして使われる一般化された型を含むアルゴリズム) を使用するため

図28

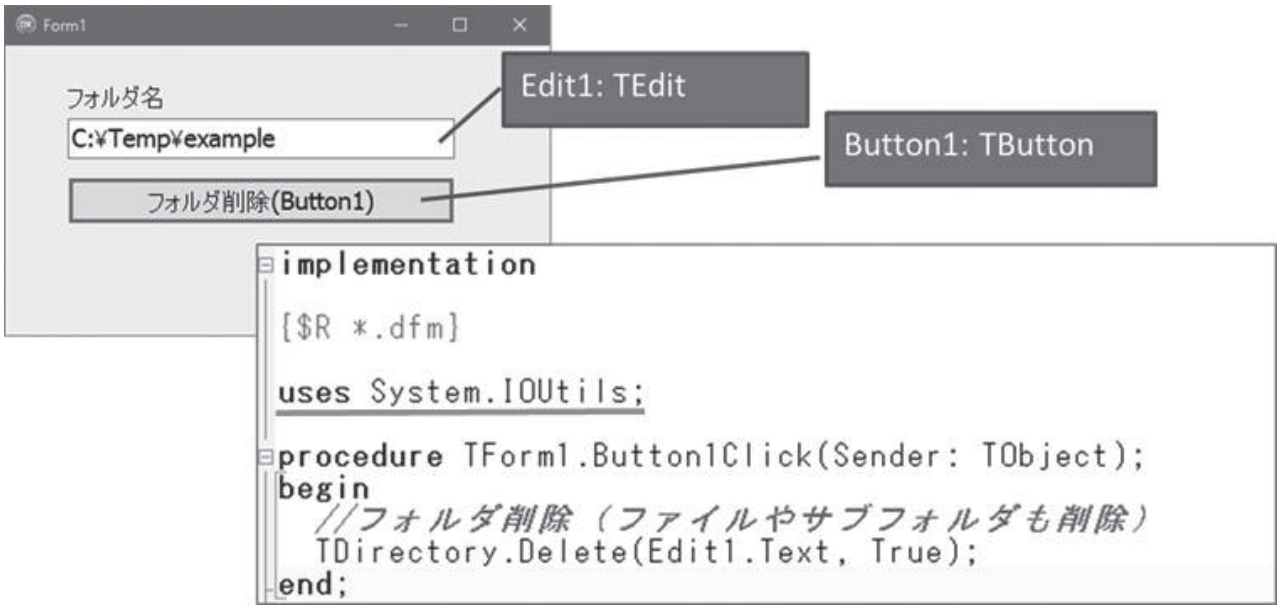

Form1
フォルダ名
C:¥Temp¥example
フォルダ削除(Button1)
Edit1: TEdit
Button1: TButton
implementation
{$R *.dfm}
uses System.IOUtils;
procedure TForm1.Button1Click(Sender: TObject);
begin
//フォルダ削除（ファイルやサブフォルダも削除）
TDirectory.Delete(Edit1.Text, True);
end;

図29

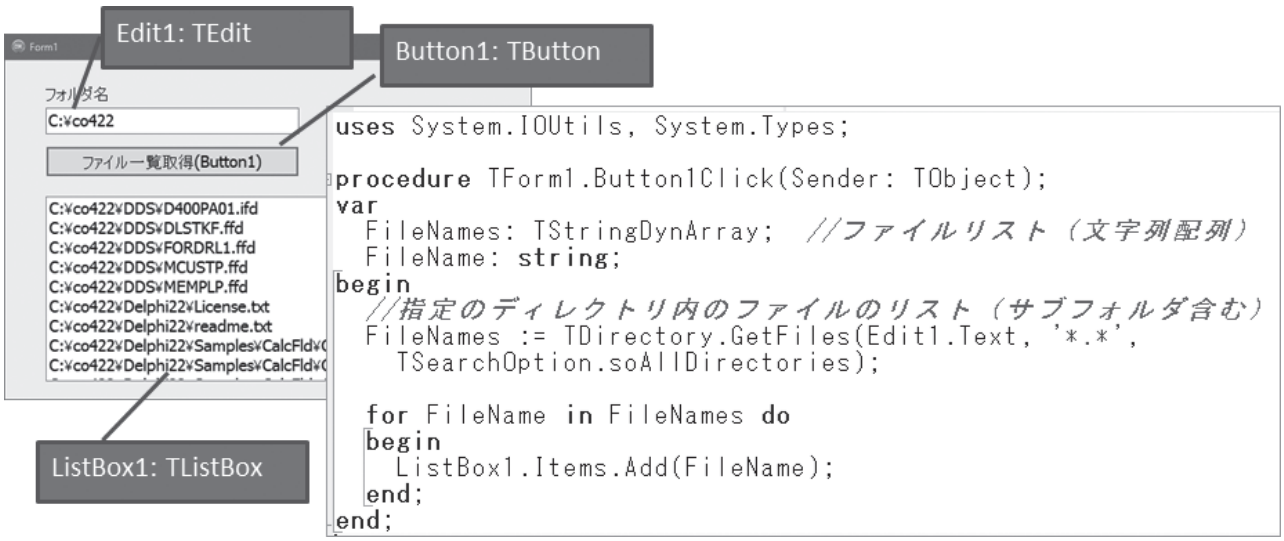
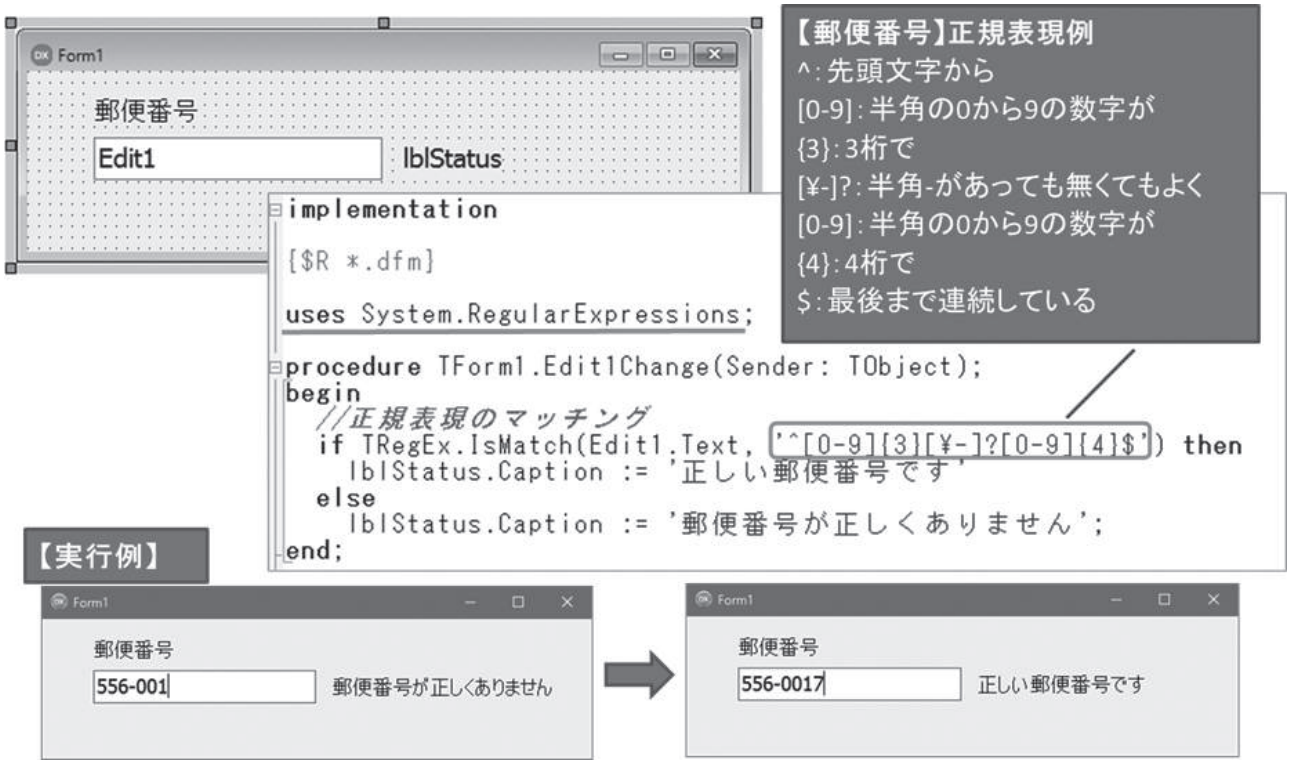

Edit1: TEdit
Button1: TButton
ListBox1: TListBox
フォルダ名
C:¥co422
ファイル一覧取得(Button1)
C:¥co422¥DDS¥D400PA01.ifd
C:¥co422¥DDS¥DLSTKF.ffd
C:¥co422¥DDS¥FORDRL1.ffd
C:¥co422¥DDS¥MCUSTP.ffd
C:¥co422¥DDS¥MEMPLP.ffd
C:¥co422¥Delphi22¥License.txt
C:¥co422¥Delphi22¥readme.txt
C:¥co422¥Delphi22¥Samples¥CalcFld¥
C:¥co422¥Delphi22¥Samples¥CalcFld¥
uses System.IOUtils, System.Types;
procedure TForm1.Button1Click(Sender: TObject);
var
FileNames: TStringDynArray; //ファイルリスト（文字列配列）
FileName: string;
begin
//指定のディレクトリ内のファイルのリスト（サブフォルダ含む）
FileNames := TDirectory.GetFiles(Edit1.Text, '*.*',
TSearchOption.soAllDirectories);
for FileName in FileNames do
begin
ListBox1.Items.Add(FileName);
end;
end;

図30


Form1
郵便番号
Edit1
lblStatus
【郵便番号】正規表現例
^:先頭文字から
[0-9]:半角の0から9の数字が
{3}:3桁で
[¥-]?:半角-があっても無くてもよく
[0-9]:半角の0から9の数字が
{4}:4桁で
$:最後まで連続している
implementation
{$R *.dfm}
uses System.RegularExpressions;
procedure TForm1.Edit1Change(Sender: TObject);
begin
//正規表現のマッチング
if TRegEx.IsMatch(Edit1.Text, '^[0-9]{3}[¥-]?[0-9]{4}$') then
lblStatus.Caption := '正しい郵便番号です'
else
lblStatus.Caption := '郵便番号が正しくありません';
end;
【実行例】
Form1
郵便番号
556-001
郵便番号が正しくありません
Form1
郵便番号
556-0017
正しい郵便番号です