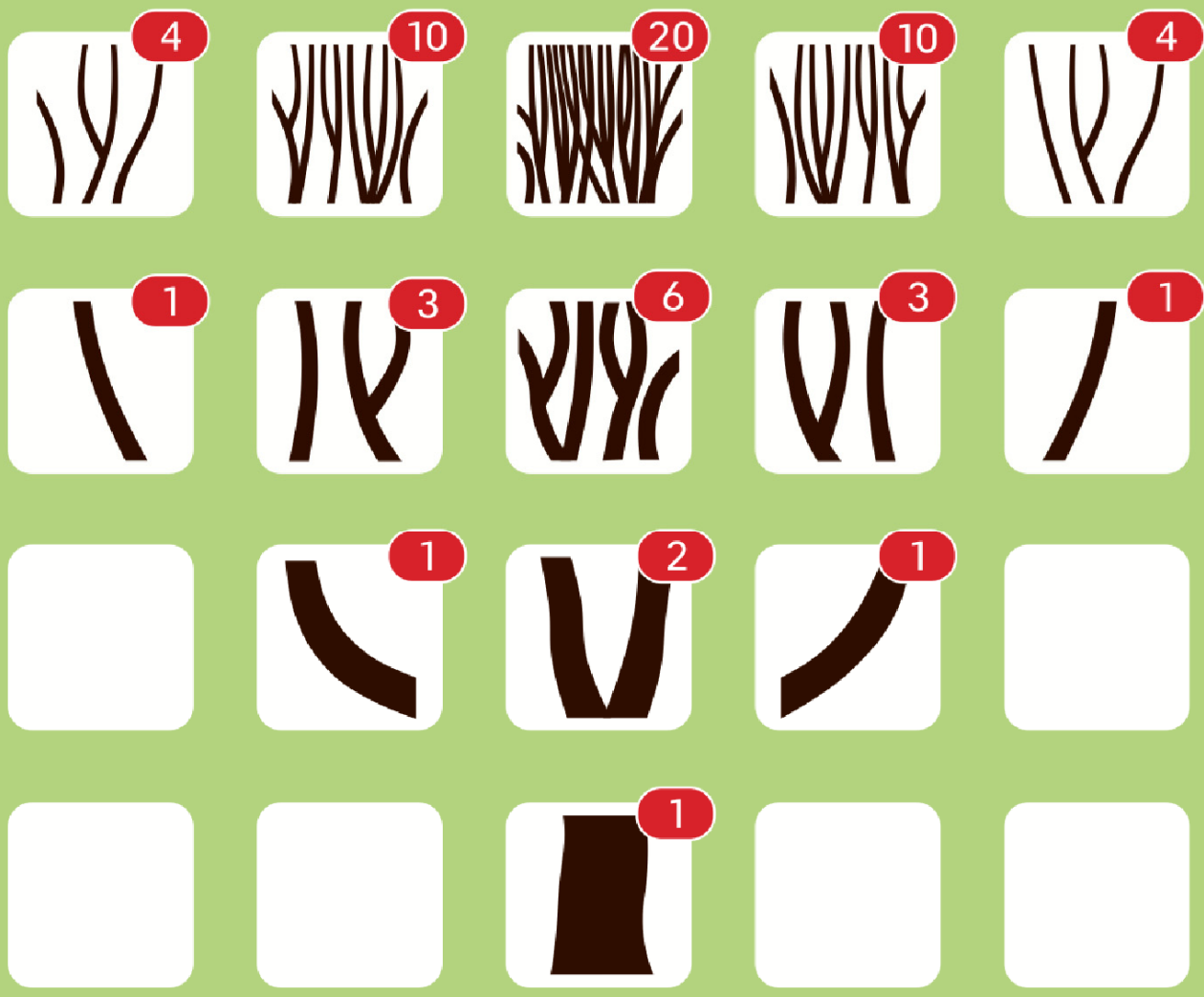

MARCO CANTÙ

OBJECT PASCAL HANDBOOK

Object Pascal プログラミング言語完全ガイド

第 III 部:高度な機能

さて、これからは言語の基礎部分とオブジェクト指向プログラミング パラダイムの詳細に踏み込んでいきます。Object Pascal 言語の最新機能やより高度な機能を確認するのによい機会でしょう。ジェネリクス、無名メソッド、リフレクションは、オブジェクト指向プログラミングを劇的に拡張する新しいパラダイムを用いたコーディングへの扉を開きます。

いくつかの高度な言語機能は、実際、型やコードの抽象化、リフレクションを用いて、その可能性を最大限活かしたよりダイナミックなアプローチを可能にし、開発者が新しいコーディング手法を実践できるようにします。

このセクションの最後のパートでは、Object Pascal 開発モデルの中核をなし、言語とライブラリの境界をあいまいにするコア ランタイム ライブラリ要素の概要を示すことで、これらの言語機能を広げていきます。例えば、前述したすべてのクラスの基底となる `TObject` クラスを詳細に見ていき、ライブラリ実装の詳細に関わる役割を明らかにしていこうと思います。

14: ジェネリクス

Object Pascal が提供する強い型チェックは、コードの正しさを強化するのに役立ちます。これは、しばしば本書で強調していることです。しかし強い型チェックは、複数の異なるデータ型を扱う手続きやクラスを使う場合には、迷惑かもしれません。この問題は、Object Pascal の言語機能によって解決されます。それは、C# や Java のような類似の言語でも利用できるジェネリクスと呼ばれる機能です。

ジェネリクスまたはテンプレート クラスの概念は、C++ 言語から来ています。以下は、私が1994年に C++ についての本で記述したものです。

> 今や、あなたは、1 つ以上のデータ メンバの型を指定することなく、クラスを宣言することができ、そのクラスのオブジェクトが実際に宣言されるまで、その操作は遅延させることができます。同様に、その関数が呼ばれるまで、あなたはその引数の 1 つ以上の型を指定することなく関数を定義することができます。

メモ この書籍は、私と Steve Tendo が90年代初めに執筆した「"Borland C++ 4.0 Object-Oriented Programming」です。

本章では、このトピックを掘り下げ、基本から高度な活用法までを紹介します。そして、標準的なビジュアル プログラミングに適用できる手法も紹介します。

4

ジェネリック Key-Value ペア

ジェネリック クラスの最初の例として、Key-Value のペアのデータ構造を実装しました。以下の最初のコード サンプルは、従来の手法で記述された、値を保持するために使用されるオブジェクトを持つデータ構造です。

```
type
  TKeyValue = class
  private
    FKey: string;
    FValue: TObject;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: TObject);
  public
    property Key: string read FKey write SetKey;
    property Value: TObject read FValue write SetValue;
  end;
```

このクラスを利用するには、KeyValueClassic アプリケーション プロジェクトのメインフォームにあるさまざまなメソッドにあるように、オブジェクトを作成し、Key と Value を設定して使います。

```
// FormCreate
kv := TKeyValue.Create;

// Button1Click
kv.Key := 'mykey';
kv.Value := Sender;

// Button2Click
kv.Value := self; // the form

// Button3Click
ShowMessage('[' + kv.Key + ',' + kv.Value.ClassName + ']');
```

オブジェクトではなく、整数を保持するようなクラスが必要な場合はどうすればよいでしょうか? 非常に不自然な (かつ危険な) 型キャストを行うか、数値とともに文字列キーを保持する別の新しいクラスを作成するかのいずれかです。元クラスをコピー & ペーストするのは、悪くないやり方かもしれませんが、ほとんど同じコードの複製を持つことになり、プログラミングのベスト プラクティスに反することになります。新しい機能を追加したりバグを修正するたびに、2 つのコードを修正しなければならず、その作業は 2 倍、3 倍、20 倍へと膨れ上がります。

ジェネリクスは、単一のジェネリック クラスを記述で、もっと広範な Value の定義を可能にします。いったん key-value ジェネリック クラスのインスタンスを作成してしまえば、指定したデータ型に結び付けられた特定のクラスになるということです。

ですから、依然として 2 つ、3 つ、あるいは 20 のクラスをアプリケーション向けにコンパイルしなければならないかもしれませんが、正しい文字列型チェックが適用され、実行時のオーバーヘッドもないにもかかわらず、ソースコードはそのすべてに対して 1 つでいいのです。ちょっと先走ってしまったようなので、ジェネリック クラスを定義する構文から始めましょう。

```
type
  TKeyValue<T> = class
  private
    FKey: string;
    FValue: T;
    procedure SetKey(const Value: string);
    procedure SetValue(const Value: T);
  public
    property Key: string read FKey write SetKey;
    property Value: T read FValue write SetValue;
  end;
```

このクラス定義には、山かっこ (< >) でくくられたプレースホールダー T によって表される「明示されていない型」が 1 つあります。シンボル T は慣例でしばしば利用されますが、コンパイラが認識する範囲では、どんなシンボルでも使うことができます。ジェネリック クラスが 1 つの型引数のみを使用するときには、一般的に T を使うことでコードの可読性が高まります。クラスが複数の型引数を必要とするときには、初期の C++ で以前行われていたように一連の文字 (T、U、V)を使うよりも、実際の役割に応じた名前を付けるほうが一般的です。

メモ C++ 言語が 1990 年代初期にテンプレートを導入した頃から、「T」はジェネリック型のための標準名ないしはプレースホールダーでした。著者に依りますが、「T」は「Type」または「Template type」のいずれかの略です。

ジェネリック クラス TKeyValue<T> は、2 つあるフィールドのうちの 1 つ (プロパティ値) の型、および Setter メソッドの引数の型として、明示されていない型を使います。メソッドは通常通りに定義されていますが、それらがジェネリック型を使わなければならないかどうかに関係なく、その定義では、クラス名は、ジェネリック型を含む完全名を使っていることが分かります。

```
procedure TKeyValue<T>.SetKey(const Value: string);
begin
  FKey := Value;
```

```
end;

procedure TKeyValue<T>.SetValue(const Value: T);
begin
  FValue := Value;
end;
```

そのかわり、このクラスを使用するには、ジェネリック型の実際の値を与えて、完全修飾しなければなりません。例えば、次のように記述して、Value として Button を保持する Key-Value オブジェクトを宣言することができます。

```
kv: TKeyValue<TButton>;
```

インスタンスを作成するときにも、実際の型名を示す完全名が必要です (インスタンス化されていないジェネリックな型名は、型構築メカニズムのように働きます)。

Key-Value ペアの Value に特定の型を使うことで、TButton (あるいはその派生) オブジェクトのみを Key-Value ペアに追加し、取り出したオブジェクトのさまざまなメソッドを利用できるようになり、コードの堅牢性が増します。以下は、KeyValueGeneric アプリケーション プロジェクトのメインフォームからのコード例です。

```
// FormCreate
kv := TKeyValue<TButton>.Create;

// Button1Click
kv.Key := 'mykey';
kv.Value := Sender as TButton;

// Button2Click
kv.Value := Sender as TButton; // was "self"

// Button3Click
ShowMessage ('[' + kv.Key + ',' + kv.Value.Name + ']');
```

前のバージョンのコードで汎用的なオブジェクトを代入する際、Button か Form のいずれかを追加することができましたが、いまや、コンパイラによって強制され、Button しか使用できません。同様に、その出力では、汎用的な kv.Value.ClassName よりも、TButton コンポーネントの Name や TButton クラスの任意のプロパティを使うことができます。

もちろん、次のように Key-Value ペアを宣言し、オリジナルのプログラムと類似のオブジェクトを作成することもできます。

```
kvo: TKeyValue<TObject>;
```

ジェネリック Key-Value ペアクラスのこのバージョンでは、任意のオブジェクトを Value に追加できます。しかし、取り出したオブジェクトに対して、特定の型にキャストしない限

り、そのオブジェクトの多くの機能を利用することはできません。よいバランスを取るために、Button と任意のオブジェクトの中間にしたいと思うかもしれません。例えば、Value が Component であることを要求します。

```
kvc: TKeyValue<TComponent>;
```

同じ KeyValueGeneric アプリケーション プロジェクトに、対応するコードサンプルがあります。次のように、オブジェクトではなく、単なる整数を保持するようなジェネリック Key-Value ペアクラスのインスタンスも作成できます。

```
var
  kvi: TKeyValue<Integer>;
begin
  kvi := TKeyValue<Integer>.Create;
  try
    kvi.Key := 'object';
    kvi.Value := 100;
    kvi.Value := Left;
    ShowMessage ('[' + kvi.Key + ',' +
      IntToStr (kvi.Value) + ']');
  finally
    kvi.Free;
  end;
```

ジェネリクスの型規則

ジェネリック型のインスタンスを宣言するとき、その型は特定のバージョンを得ます。それは、それ以降のすべての操作をコンパイラによって強制されます。次のようなジェネリック クラスがあるとします。

```
type
  TSimpleGeneric<T> = class
    Value: T;
  end;
```

型を指定して特定のオブジェクトを宣言すると、異なる型を Value フィールドに代入することはできません。以下の2つのオブジェクトでは、誤った代入を行っている箇所があります (このコードは TypeCompRules アプリケーション プロジェクトの一部です)。

```
var
  sg1: TSimpleGeneric<string>;
  sg2: TSimpleGeneric<Integer>;
begin
  sg1 := TSimpleGeneric<string>.Create;
  sg2 := TSimpleGeneric<Integer>.Create;
```

```
sg1.Value := 'foo';
sg1.Value := 10; // Error
// E2010 Incompatible types: 'string' and 'Integer'

sg2.Value := 'foo';  // Error
// E2010 Incompatible types: 'Integer' and 'string'
sg2.Value := 10;
```

いったん特定の型をジェネリック宣言で定義すれば、Object Pascal のような型付けの強い言語で期待されるのと同じように、コンパイラによって型が強制されます。型のチェックは、ジェネリック オブジェクト全体に対して行われます。例えば、あるオブジェクト用にジェネリック パラメータを指定すると、それに対して、異なる互換性のない型をベースとした同じようなジェネリック型のインスタンスを割り当てることはできません。混乱するようであれば、次の例が分かりやすいでしょう。

```
sg1 := TSimpleGeneric<Integer>.Create; // Error
// E2010 Incompatible types:
// 'TSimpleGeneric<System.string>'
// and 'TSimpleGeneric<System.Integer>'
```

「ジェネリック型の互換性規則」の箇所で特定のケースを説明しますが、型互換性規則は構造によるもので、型名によるものではありません。一度宣言されると、異なる互換性のない型をジェネリック型に割り当てることはできません。

Object Pascal におけるジェネリクス

前の例では、Object Pascal でどのようにジェネリッククラスを定義して使用するのかを見てきました。非常に複雑で重要な専門的なことがらに入る前に、ここでは、その特長をサンプルとともに紹介することにしましょう。言語的な観点からジェネリクスについて解説した後、ジェネリックコンテナクラスの使用と定義について、さらに、この技術がなぜ言語に追加されたのかという理由を含め、もっと多くの例に戻ろうと思います。

クラスを定義するときに、後で型を提供するための場所をキープするために、山かっこ (< >) の中に追加の「型引数」を加えることができるようになったことを学びました。

```
type
  TMyClass<T> = class
```

```
    ...
  end;
```

ジェネリック型は、フィールドの型として (前のサンプルで示したように)、プロパティの型、関数の引数や戻り値の型など、さまざまなところで使用できます。ジェネリック型がローカルフィールド (あるいは配列) で使用することを強制されていないことに注意してください。これは、ジェネリック型が、戻り値や引数だけで使われていたり、クラスの宣言では使われず、そのいくつかのメソッド定義の中だけで使われるケースがあるからです。

この拡張されたジェネリックな型宣言の書式は、クラスだけでなくレコードでも利用できます (第5章で説明したように、メソッド、プロパティ、オーバーロード演算子も定義可能です)。ジェネリッククラスでは、以下の例のように、メソッドの引数と戻り値で異なる型を定義できるように、複数の型引数を持つことができます。

```
type
  TPWGeneric<TInput,TReturn> = class
  public
    function AnyFunction (Value: TInput): TReturn;
  end;
```

Object Pascal におけるジェネリクスの実装は、他の静的言語のように実行時フレームワークをベースとしてはいません。それはコンパイラとリンカによって制御されており、ランタイム メカニズムに残しているものは、ほぼ何もありません。実行時に結び付けられる仮想関数呼び出しと異なり、テンプレートメソッドは、各テンプレート型をインスタンス化するごとに作られ、それはコンパイル時に生成されます！後ほどこのアプローチが持つ可能性がある欠点に触れますが、ジェネリック クラスのポジティブな面は、実行時チェックの必要性が減り、通常のクラスと同じくらいか、それ以上に効率的なことです。しかし、その内部を調べる前に、伝統的な Pascal 言語の型互換性規則を破るいくつかの非常に重要な規則に焦点を当てることにします。

ジェネリック型の互換性規則

伝統的な Pascal 言語と Object Pascal では、型互換性規則の核となる部分は、型名同等性を基盤にしています。いいかえれば、2 つの変数が型互換性を持つのは、2 つの型名が同じ場合のみで、両者の実際のデータ構造が同じかどうかは関係ありません。

以下は、配列を使った型非互換性の昔ながらの例です (`TypeCompRules` アプリケーションプロジェクト)。

```
type
```

```
  TArrayOf10 = array [1..10] of Integer;

procedure TForm30.Button1Click(Sender: TObject);
var
  array1: TArrayOf10;
  array2: TArrayOf10
  array3, array4: array [1..10] of Integer;
begin
  array1 := array2;
  array2 := array3; // Error
  // E2010 Incompatible types: 'TArrayOf10' and 'Array'

  array3 := array4;
  array4 := array1; // Error
  // E2010 Incompatible types: 'Array' and 'TArrayOf10'
end;
```

上記のコードで分かるように、4 つの配列すべては構造的に同一です。しかし、それらの型には、それぞれ同じ (TArrayOf10 のような) 明示的な名前か、同じ暗黙的な型名 (あるいは、1 つの文で 2 つの配列が宣言されているので、コンパイラが生成する型名) が付けられているため、型互換性を持つ配列同士でしか代入ができません。

この型互換性規則には、派生クラスに関する非常に限られた例外があります。この規則の新しい例外で、非常に重要性を帯びているのは、ジェネリック型の型互換性です。それは、いつジェネリッククラスから新しい型およびそのすべてのメソッドを生成するかを決定するために、コンパイラによって内部的にも恐らく利用されるものです。

新しい規則では、同じジェネリッククラス定義とインスタンスの型を共有していると、その定義に関連する型の名前に関係なく、ジェネリック型は互換性があります。いいかえれば、ジェネリック型のインスタンスの完全名は、ジェネリック型とインスタンスの型の組合せです。

次の例では、4 つの変数すべてが、互換性を持つ型です。

```
type
  TGenericArray<T> = class
    anArray: array [1..10] of T;
  end;

  TIntGenericArray = TGenericArray<Integer>;

procedure TForm30.Button2Click(Sender: TObject);
var
  array1: TIntGenericArray;
  array2: TIntGenericArray;
  array3, array4: TGenericArray<Integer>;
begin
  array1 := TIntGenericArray.Create;
  array2 := array1;
```

```
  array3 := array2;
  array4 := array3;
  array1 := array4;
end;
```

標準クラスのためのジェネリックメソッド

ジェネリック型は、クラスを定義するために用いるのが最も一般的なシナリオですが、非ジェネリッククラスにおいても用いることができます。いいかえれば、通常のクラスはジェネリックメソッドを持つことができます。この場合、クラスのインスタンスを作成するときだけでなく、メソッドを呼び出す際にも、ジェネリック プレース ホルダーのために特定の型を指定しません。こちらは、GenericMethod アプリケーション プロジェクトのジェネリックメソッドを使うクラスの例です。

```
type
  TGenericFunction = class
  public
    function WithParam <T> (t1: T): string;
  end;
```

メモ 私が最初にこのコードを記述したときには、おそらく C++を使っていたころの記憶が残っていてパラメータを (t: T) と書いたのだと思います。言うまでもなく、Object Pascal のような大文字／小文字を区別しない言語では、これはよいやり方ではありません。コンパイラは実際のところ、ありのままに処理しますが、ジェネリック型 T を参照するたびにエラーが発生します。

(少なくとも、この章のあとで触れる制約を使わない限り) 同じようなクラスメソッドの中でできることはそれほど多くはありません。そこで私は、特別なジェネリック型関数 (あとで触れます) と、ここでは触れていませんが、型を文字列に変換する特別な関数を使っているコードを記述しました。

```
function TGenericFunction.WithParam<T>(t1: T): string;
begin
  Result := GetTypeName (TypeInfo (T));
end;
```

ご覧のように、このメソッドは、パラメータとして渡した実際の値を使っておらず、型情報を扱うだけです。繰り返しますが、t1 の型をまったく知ることなく、コード上でかなり複雑な処理を行えます。

以下のように、「グローバルジェネリック関数」のさまざまなバージョンを呼び出すことができます。

```
var
```

```
  gf: TGenericFunction;
begin
  gf := TGenericFunction.Create;
  try
    Show (gf.WithParam<string>('foo'));
    Show (gf.WithParam<Integer> (122));
    Show (gf.WithParam('hello'));
    Show (gf.WithParam (122));
    Show (gf.WithParam(Button1));
    Show (gf.WithParam<TObject>(Button1));
  finally
    gf.Free;
  end;
```

上記のすべての呼び出しは、パラメータ型がこれらの呼び出しで暗黙的に扱われ、正しく動作します。ジェネリック型では、以下の出力例のように、特定の、あるいは推論として表示され、パラメータの実際の型が表示されるのではないということに注意してください。

```
string
Integer
string
ShortInt
TButton
TObject
```

もし、山かっこで囲った型指定なしにメソッドを呼び出した場合、実際の型は、パラメータの型から推論されます。型指定とパラメータを伴ってメソッドを呼び出す場合、パラメータ型は、ジェネリック型定義と合致していなければなりません。ですから、以下の 3 行はコンパイルできません。

```
    Show (gf.WithParam<Integer>('foo'));
    Show (gf.WithParam<string> (122));
    Show (gf.WithParam<TButton>(self));
```

ジェネリック型のインスタンス化

> *このセクションは、ジェネリクスの内部とその潜在的な最適化について解説した高度な内容である点に注意してください。ジェネリクスについて初めて学んでいる方は、一旦全部お読みいただいてから再読するとよいかもしれません。*

若干の最適化を除き、メソッド内でもクラス内であってもジェネリック型をインスタンス化するたびに、コンパイラによって新しい型が生成されます。この新しい型は、同じジェネリック型の異なるインスタンス (あるいは同じメソッドの異なるバージョン) とコードを共有していません。

GenericCodeGen アプリケーションプロジェクトの例を見てみましょう。ここでは、次のようなジェネリック クラスが定義されています。

```
type
  TSampleClass <T> = class
  private
    data: T;
  public
    procedure One;
    function ReadT: T;
    procedure SetT (value: T);
  end;
```

次のように、3 つのメソッドが実装されています (One メソッドがジェネリック型から完全に独立していることに注意してください)。

```
procedure TSampleClass<T>.One;
begin
  Form30.Show ('OneT');
end;

function TSampleClass<T>.ReadT: T;
begin
  Result := data;
end;

procedure TSampleClass<T>.SetT(value: T);
begin
  data := value;
end;
```

さて、メインプログラムでは、(コンパイラによって) インスタンスを生成した後、主にそのメソッドのメモリアドレスを計算するためにジェネリック型を使います。以下がそのコードです。

```
procedure TForm30.Button1Click(Sender: TObject);
var
  t1: TSampleClass<Integer>;
  t2: TSampleClass<string>;
begin
  t1 := TSampleClass<Integer>.Create;
  t1.SetT (10);
  t1.One;

  t2 := TSampleClass<string>.Create;
  t2.SetT ('hello');
  t2.One;

  Show ('t1.SetT: ' +
    IntToHex (PInteger(@TSampleClass<Integer>.SetT)^, 8));
  Show ('t2.SetT: ' +
```

```
    IntToHex (PInteger(@TSampleClass<string>.SetT)^, 8));

  Show ('t1.One: ' +
    IntToHex (PInteger(@TSampleClass<Integer>.One)^, 8));
  Show ('t2.One: ' +
    IntToHex (PInteger(@TSampleClass<string>.One)^, 8));
end;
```

結果は、次のようになります(実際の値は違ったものになるかもしれません)。

```
t1.SetT: C3045089
t2.SetT: 51EC8B55
t1.One: 4657F0BA
t2.One: 46581CBA
```

予測どおり、SetT メソッドで、使用されたデータ型ごとにメモリ上にコンパイラによって生成された異なるバージョンが得られただけでなく、One メソッドでも新しいバージョンが得られました。

さらに、もし同じジェネリクス型を再定義すると、新しい関数実装のセットを得ることになります。同様に、異なるユニットで使われるジェネリック型の同じインスタンスの場合も、コンパイラが何度も同じコードを生成するようなり、コードの肥大化を引き起こす可能性があります。この理由により、ジェネリック型に依存しない多くのメソッドを持つジェネリッククラスを作る場合、これらの共通メソッドを持つジェネリックではないベースクラスを定義して、それを継承してジェネリックメソッドを持つジェネリッククラスを定義することを推奨します。このメソッドで、ベースクラスのメソッドは、1 回だけコンパイルされ、実行モジュールに 1 つだけ含まれるようになります。

メモ この箇所で説明したようなシナリオでジェネリクスが引き起こすサイズ増加を抑えるための取り組みとして、コンパイラ、リンカ、低レベルの RTL に対して作業が加えられました。これに関する考察は、こちらのブログ (http://delphisorcery.blogspot.it/2014/10/new-language-feature-in-xe7.html) を参照してください。

ジェネリック型関数

ここまで見てきたジェネリック型の定義に関して最大の問題は、ジェネリッククラス型のオブジェクトによってできることがごくわずかしかないということです。この制限を克服するために使うことのできるテクニックが 2 つあります。1 つ目は、特にジェネリック型をサポートするランタイムライブラリのいくつかの特別な関数を利用することです。2 つ目は(より強力で)、使用できる型の制約を伴ったジェネリッククラスを定義することです。

ここではまず 1 つ目にフォーカスし、その後制約についてフォーカスします。先に触れたように、ジェネリック型定義の型引数 (T) を扱えるいくつかの RTL 関数があります。

- `Default (T)`: 現在の型の「空の値」、ないしは「ゼロ値」、「ヌル値」を返す新しい関数です。ゼロ、空の文字列、nil などを返します。
- `TypeInfo (T)`: ジェネリック型の現在のバージョンの実行時情報へのポインタを返します。詳細は、第16章をご覧ください。
- `SizeOf (T)`: 型のメモリサイズをバイト単位で返します (文字列やオブジェクトのような参照型の場合、その参照のサイズ、つまり 32-bit コンパイラでは 4 バイト、64-bit コンパイラでは 8 バイトになります)。
- `IsManagedType (T)`: 型がインメモリで管理されていることを示します。文字列および動的配列の場合。
- `HasWeakRef (T)`: ARC 対応のコンパイラで、ターゲットの型が特定のメモリ管理サポートを必要とする弱い参照を持っているかどうかを返します。
- `GetTypeKind (T)`: 型情報から型の種類にアクセスするためのショートカット。これは、`TypeInfo` の戻り値よりも、わずかにハイレベルな型定義です。

メモ これらのメソッドは、実行時に実際の関数を呼び出すのではなく、すべてコンパイラ評価の定数を返します。重要なのは、これらの操作が大変高速であるということではなく、コンパイラとリンカが、使用されないコードを除去して、生成コードを最適化できることです。もし、case 文や if 文があり、これらの関数の戻り値を評価に使っている場合、コンパイラは、指定した型に対して、分岐のうちひとつのコードブロックしか実行されないことが分かり、残りの使用されないコードを削除できます。同じメソッドに異なる型を指定してコンパイルしたときは、別のコードブロックが実行されるかもしれませんが、それもコンパイラは把握でき、メソッドサイズの最適化が可能です。

`GenericTypeFunc` アプリケーション プロジェクトには、3 つのジェネリック型関数の使用例を示すジェネリッククラスがあります。

```
type
  TSampleClass <T> = class
  private
    data: T;
  public
    procedure Zero;
    function GetDataSize: Integer;
    function GetDataName: string;
  end;

function TSampleClass<T>.GetDataSize: Integer;
begin
  Result := SizeOf (T);
```

```
end;

function TSampleClass<T>.GetDataName: string;
begin
  Result := GetTypeName (TypeInfo (T));
end;

procedure TSampleClass<T>.Zero;
begin
  data := Default (T);
end;
```

GetDataName メソッドでは、直接データ構造にアクセスするのではなく、GetTypeName 関数 (System.TypInfo ユニット) を使いました。これは、エンコードされた型名を保持した string 値から、適切な変換が行われるからです。

上記の宣言により、3 つの異なるジェネリック型のインスタンスに対し、3 回同じことを繰り返す以下のテストコードをコンパイルできます。繰り返しコードは省略しましたが、実際の型によって変わってくる data フィールドへのアクセスコードだけは記載しておきます。

```
var
  t1: TSampleClass<Integer>;
  t2: TSampleClass<string>;
  t3: TSampleClass<double>;
begin
  t1 := TSampleClass<Integer>.Create;
  t1.Zero;
  Show ('TSampleClass<Integer>');
  Show ('data: ' + IntToStr (t1.data));
  Show ('type: ' + t1.GetDataName);
  Show ('size: ' + IntToStr (t1.GetDataSize));

  t2 := TSampleClass<string>.Create;
  ...
  Show ('data: ' + t2.data);

  t3 := TSampleClass<double>.Create;
  ...
  Show ('data: ' + FloatToStr (t3.data));
```

この GenericTypeFunc アプリケーション プロジェクトのコードを実行すると、次のように出力されます。

```
TSampleClass<Integer>
data: 0
type: Integer
size: 4
TSampleClass<string>
data:
type: string
size: 4
```

```
TSampleClass<double>
data: 0
type: Double
size: 8
```

ジェネリック クラスのコンテキスト以外でも、特定の型に対して、ジェネリック型関数を使うことができることに注意してください。例えば、次のように記述できます。

```
var
  I: Integer;
  s: string;
begin
  I := Default (Integer);
  Show ('Default Integer': + IntToStr (I));

  s := Default (string);
  Show ('Default String': + s);

  Show ('TypeInfo String': +
    GetTypeName (TypeInfo (string));
```

次のように得られる情報はわずかです。

```
Default Integer: 0
Default String:
TypeInfo String: string
```

メモ 上記のコードでは、`TypeInfo(s)` のように、`TypeInfo` 呼び出しを変数に適用することはできず、型にだけ適用可能です。

ジェネリッククラスのための クラスコンストラクタ

ジェネリック クラスのためのクラス コンストラクタを定義するときに、大変興味深いことが明らかになります。実際、こうしたコンストラクタは、一般的にコンパイラによって生成され、ジェネリック テンプレートを使って定義されたそれぞれの実際の型に対し、各ジェネリック クラスのインスタンス用に呼び出されます。クラス コンストラクタなしに、プログラム上でジェネリック クラスの実際のインスタンスを生成する初期化コードを実行しようとすると、極めて複雑化してしまうため、これは大変興味深いことです

例として、いくつかのクラス データとともにジェネリック クラスについて考察しましょう。それぞれのジェネリック クラス インスタンスのために、このクラス データのインスタンスを取得するとします。このクラスデータに初期値を設定する必要があっても、ジェネリック クラ

スを定義しているユニットでは実際にどのクラスが必要になるかが分からないため、ユニットの初期化コードを使うことができません。

以下は、GenericClassCtor アプリケーション プロジェクトの、DataSize クラス フィールドの初期化にクラス コンストラクタを使ったジェネリック クラスのサンプルの骨子です。

```
type
  TGenericWithClassCtor <T> = class
  private
    FData: T;
    procedure SetData(const Value: T);
  public
    class constructor Create;
    property Data: T read FData write SetData;
    class var
      DataSize: Integer;
  end;
```

これは、どのクラス コンストラクタが実際に呼び出されたのかを記録しておくために、内部文字列リストを用いたジェネリック クラスコンストラクタのコードです (実装の詳細はフル ソースコードをご覧ください)。

```
class constructor TGenericWithClassCtor<T>.Create;
begin
  DataSize := SizeOf (T);
  ListSequence.Add(ClassName);
end;
```

デモプログラムでは、ジェネリック クラスの 2 つのインスタンスを作成して使っています。一方、3 番目に宣言したデータ型は、リンカによって削除されます。

```
var
  genInt: TGenericWithClassCtor <SmallInt>;
  genStr: TGenericWithClassCtor <string>;
type
  TGenDouble = TGenericWithClassCtor <Double>;
```

ListSequence 文字列リストの内容を表示するようにプログラムを書くと、実際に初期化された型のみが表示されます。

```
TGenericWithClassCtor<System.SmallInt>
TGenericWithClassCtor<System.string>
```

しかし、もし異なるユニットの同じデータ型に基づいてジェネリック インスタンスを作成すると、リンカは期待したようには動かず、同じジェネリック クラス コンストラクタを複数回 (より正確には、同じデータ型に対して 2 つのクラス コンストラクタ) 呼び出します。