EMBARCADERO
TECHNOLOGIES®

White Paper

Using New Delphi Coding Styles and Architectures

A Review of the Language Features in Delphi 2009

By Marco Cantù

December 2008

Corporate Headquarters
100 California Street, 12th Floor
San Francisco, California 94111

EMEA Headquarters
York House
18 York Road
Maidenhead, Berkshire
SL6 1SF, United Kingdom

Asia-Pacific Headquarters
L7. 313 La Trobe Street
Melbourne VIC 3000
Australia

Introduction: The Delphi Language

The Delphi language, better known as Object Pascal, is a modern strong type-checked and object-oriented language, featuring single inheritance and an object reference model. In recent years, the language has been augmented with record with methods, operators overloading for records, class data, nested types, sealed classes, final methods and many other relevant features. The most surprising extension was probably the introduction of class helpers, a technique used to add new methods to an existing class or replace some of the existing methods.

But in Delphi 2009 the new features added to the compiler are even more relevant. Besides the extensions to the string type to support Unicode, the last version of Delphi introduces generic data types, anonymous methods, and a number of other "minor" but very interesting features.

Introducing Generics

As a first example of a generic class, I've implemented a key-value pair data structure. The first code snippet below shows the data structure as it is traditionally written, with an object used to hold the value:

```
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;
```

To use this class you can create an object, set its key and value, and use it, shown in the following snippets:

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

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

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

// Button3Click
ShowMessage ('[' + kv.Key + ',' +
```

```
  kv.Value.ClassName + ']');
```

Generics make it possible to use a much broader definition for the value, but that's not the key point. What's totally different (as we'll see) is that once you've instantiated the key-value generic class, it becomes a specific class tied to a given data type. This makes your code type safer, but I'm getting ahead of myself. Let's start with the syntax used to define the generic class:

```
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;
```

In this class definition, there is one unspecified type that is indicated by the placeholder `T`. The generic `TKeyValue<T>` class uses the unspecified type as the type of the property value field and the setter method parameter. The methods are defined as usual; however, even though they have to do with the generic type, their definition contains the complete name of the class, including the generic type:

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

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

To use the class, instead, you have to fully qualify it, providing the actual value of the generic type. For example, you can now declare a key-value object hosting button as value by writing:

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

The full name is also required when creating an instance because this is the actual type name (the generic, uninstantiated type name is like a type construction mechanism).

Using a specific type of the value for the key-value pair makes the code much more robust, as you can now only add `TButton` (or derived) objects to the key-value pair and can use the various methods of the extracted object. These are some snippets:

```
// 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 + ']');
```

When assigning a generic object in the previous version of the code we could add either a button or a form. Now we can use only a button, a rule enforced by the compiler. Likewise, rather than a generic `kv.Value.ClassName` in the output, we can use the component `Name`, or any other property of `TButton`.

Of course, we can also mimic the original program by declaring the key-value pair as:

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

In this version of the generic key-value pair class, we can add any object as value. However, we won't be able to do much on the extracted objects unless we cast them to a more specific type. To find a good balance, you might want to go for something in between specific buttons and any object and request the value to be a component:

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

Finally, we can create an instance of the generic key-value pair class that doesn't store object values, but rather plain integers, as shown:

```
kvi: TKeyValue<Integer>;
```

TYPE RULES ON GENERICS

When you declare an instance of a generic type, this type gets a specific version, which is enforced by the compiler in all subsequent operations. So, if you have a generic class like:

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

as you declare a specific object with a given type, you cannot assign a different type to the `Value` field. Given the following two objects, some of the assignments below are incorrect:

```
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;
```

Once you define a specific type the generic declaration, this is enforced by the compiler, as you should expect by a strongly-typed language like Object Pascal. The type checking is also in place for the generic objects as a whole. As you specify the generic parameter for an object, you cannot assign to it a similar generic type based on a different and incompatible type instance. If this seems confusing, an example should help clarify:

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

The type compatibility rule is by structure and not by type name but you cannot assign to a generic type instance a different and incompatible one.

GENERICS IN DELPHI

In the previous example, we saw how you can define and use a generic class which is one of the most relevant extensions to the Object Pascal language since Delphi 3 introduced interfaces. I decided to introduce the feature with an example before delving into the technicalities, which are quite complex and very relevant at the same time. After covering generics from a language perspective we'll get back to more examples, including the use and definition of generic container classes, one of the main reasons this technique was added to the language.
We have seen that when you define a class in Delphi 2009, you can now add an extra "parameter" within angle brackets to hold the place of a type to be provided later:

```
type
  TMyClass <T> = class
    ...
  end;
```

The generic type can be used as the type of a field (as I did in the previous example), as the type of a property, as the type of a parameter or return value of a function and more. Notice that it is not compulsory to use the type for a local field (or array) as there are cases in which the generic type is used only as a result, a parameter, or is not used in the declaration of the class, but only in the definition of some of its methods.

This form of extended or *generic* type declaration is available for classes and also for records (in the most recent versions of Delphi records can also have methods and overloaded operators, in case you didn't notice). You cannot declare a generic global function unlike C++, but you can declare a generic class with a single class method, which is almost the same thing.

The implementation of generics in Delphi, like in other static languages, is not based on a runtime framework, but is handled by the compiler and the linker, and leaves almost nothing to runtime mechanism. Unlike virtual function calls that are bound at runtime, template methods are generated once for each template type you instantiate, and are generated at compile time! We'll see the possible drawbacks of this approach, but, on the positive side, it implies that the generic classes are as efficient as plain classes, or even more efficient as the need for runtime cats is reduced.

GENERIC TYPE FUNCTIONS

The biggest problem with the generic type definitions we've seen so far is that you can do very little with objects of the generic type. There are two techniques you can use to overcome this limitation. The first is to make use of the few special functions of the runtime library that specifically support generic types. The second (and much more powerful) is to define generic classes with constraints on the types you can use.

I'll focus on the first part in this section and the constraints in the next section. As I mentioned, there is a brand new function and two classic ones specifically modified to work on the parametric type (**T**) of generic type definition:

- `Default(T)` is a brand new function that returns the empty or "zero value" or null value for the current type; this can be zero, an empty string, `nil`, and so on;
- `TypeInfo (T)` returns the pointer to the runtime information for the current version of the generic type;
- `SizeOf (T)` returns memory size of the type in bytes.

The following example has a generic class showing the three generic type functions in action:

```
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;
```

In the `GetDataName` method, I used the `GetTypeName` function (or the TypInfo unit) rather than directly accessing the data structure because it performs the proper UTF-8 conversion from the encoded ShortString value holding the type name.

Given the declaration above, you can compile the following test code that repeats itself three times on three different generic type instances. I've omitted the repeated code, but kept the statements used to access the `data` field, as they change depending on the actual type:

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

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

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

Running this code produces the following output:

```
TSampleClass<Integer>
data: 0
type: Integer
size: 4
TSampleClass<string>
data:
type: string
size: 4
TSampleClass<double>
data: 0
type: Double
size: 8
```

Notice that, oddly enough, you can use the generic type functions also on specific types outside of the contest of generic classes. For example, you can write:

```
var
```

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

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

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

While the calls to `Default` are brand new in Delphi 2009 (although not terribly useful outside of templates), the call to `TypeInfo` at the end was already possible in past versions of Delphi. This is the trivial output:

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

GENERIC CONSTRAINTS

As we have seen, there is very little you can do with the methods of your generic class over the generic type value. You can pass it around (that is, assign it) and perform the limited operations allowed by the generic type functions I've just covered.

To be able to perform some actual operations of the generic type of class, you generally place a constraint on it. For example, when you limit the generic type to be a class, the compiler will let you call all of the `TObject` methods on it. You can also further constrain the class to be part of a given hierarchy or implement a specific interface.

CLASS CONSTRAINTS

The simplest constraint you can adopt is a class constraint. To use it, you declare generic type as:

```
type
  TSampleClass <T: class> = class
```

By specifying a class constraint, you indicate that you can use only object types as generic types.

With the following declaration:

```
type
  TSampleClass <T: class> = class
  private
    data: T;
  public
    procedure One;
```

```
    function ReadT: T;
    procedure SetT (t: T);
  end;
```

you can create the first two instances but not the third:

```
  sample1: TSampleClass<TButton>;
  sample2: TSampleClass<TStrings>;
  sample3: TSampleClass<Integer>; // Error
```

The compiler error caused by this last declaration would be:

```
E2511 Type parameter 'T' must be a class type
```

What's the advantage of indicating this constraint? In the generic class methods you can now call any `TObject` method, including virtual ones! This is the `One` method of the `TSampleClass` generic class:

```
procedure TSampleClass<T>.One;
begin
  if Assigned (data) then
  begin
    Form30.Log('ClassName: ' + data.ClassName);
    Form30.Log('Size: ' + IntToStr (data.InstanceSize));
    Form30.Log('ToString: ' + data.ToString);
  end;
end;
```

You can play with the program to see its actual effect as it defines and uses a few instances of the generic type, as in the following code snippet:

```
var
  sample1: TSampleClass<TButton>;
begin
  sample1 := TSampleClass<TButton>.Create;
  try
    sample1.SetT (Sender as TButton);
    sample1.One;
  finally
    sample1.Free;
  end;
```

Notice that by declaring a class with a customized `ToString` method, this custom version will get called when the data object is of the specific type, regardless of the actual type provided to the generic type. In other words, if you have a `TButton` descendant such as:

```
type
  TMyButton = class (TButton)
  public
    function ToString: string; override;
```

```
  end;
```

You can pass this object as value of a `TSampleClass<TButton>` or define a specific instance of the generic type, and in both cases calling `One` ends up executing the specific version of `ToString`:

```
var
  sample1: TSampleClass<TButton>;
  sample2: TSampleClass<TMyButton>;
  mb: TMyButton;
begin
  ...
  sample1.SetT (mb);
  sample1.One;
  sample2.SetT (mb);
  sample2.One;
```

Similar to a class constraint, you can have a record constraint, declared as:

```
type
  TSampleRec <T: record> = class
```

However, there is very little that different records have in common (there is no common ancestor), so this declaration is somewhat limited.

SPECIFIC CLASS CONSTRAINTS

If your generic class needs to work with a specific subset of classes (a specific hierarchy), you might want to resort to specifying a constraint based on a given base class. For example, if you declare:

```
type
  TCompClass <T: TComponent> = class
```

instances of this generic class can be applied only to component classes; that is, any `Tcomponent` descendant class. This lets you have a very specific generic type (yes it sounds odd, but that's what it really is) and the compiler will let you use all of the methods of the `TComponent` class while working on the generic type.

If this seems extremely powerful, think twice. If you consider what you can achieve with inheritance and type compatibly rules, you might be able to address the same problem using traditional, object-oriented techniques rather than having to use generic classes. I'm not saying that a specific class constraint is never useful, but it is certainly not as powerful as a higher-level class constraint or (something I find very interesting) an interface-based constraint.

INTERFACE CONSTRAINTS

Rather than constraining a generic class to a given class, it is generally more flexible to accept only classes implementing a given interface as the type parameter. This makes it possible to call the interface on instances of the generic type.

That said, the use of interface constraints for generics is also very common in the .NET framework. Let me start by showing you an example.

First, we need to declare an interface:

```
type
  IGetValue = interface
    ['{60700EC4-2CDA-4CD1-A1A2-07973D9D2444}']
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    property Value: Integer
      read GetValue write SetValue;
  end;
```

Next, we can define a class that implements it:

```
type
  TGetValue = class (TSingletonImplementation, IGetValue)
  private
    fValue: Integer;
  public
    constructor Create (Value: Integer = 0);
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
  end;
```

Things start to get interesting when you define a generic class limited to types that implement the given interface:

```
type
  TInftClass <T: IGetValue> = class
  private
    val1, val2: T; // or IGetValue
  public
    procedure Set1 (val: T);
    procedure Set2 (val: T);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
  end;
```

Notice that in the code for the generic methods of this class we can write:

```
function TInftClass<T>.GetMin: Integer;
begin
  Result := min (val1.GetValue, val2.GetValue);
end;

procedure TInftClass<T>.IncreaseByTen;
begin
```

```
  val1.SetValue (val1.GetValue + 10);
  val2.Value := val2.Value + 10;
end;
```

With all these definitions, we can now use the generic class as follows:

```
procedure TFormIntfConstraint.btnValueClick(
  Sender: TObject);
var
  iClass: TInftClass<TGetValue>;
begin
  iClass := TInftClass<TGetValue>.Create;
  iClass.Set1 (TGetValue.Create (5));
  iClass.Set2 (TGetValue.Create (25));
  Log ('Average: ' + IntToStr (iClass.GetAverage));
  iClass.IncreaseByTen;
  Log ('Min: ' + IntToStr (iClass.GetMin));
end;
```

To show the flexibility of this generic class, I've created another totally different implementation for the interface:

```
  TButtonValue = class (TButton, IGetValue)
  public
    function GetValue: Integer;
    procedure SetValue (Value: Integer);
    class function MakeTButtonValue (Owner: TComponent;
      Parent: TWinControl): TButtonValue;
  end;

{ TButtonValue }

function TButtonValue.GetValue: Integer;
begin
  Result := Left;
end;

procedure TButtonValue.SetValue(Value: Integer);
begin
  Left := Value;
end;
```

The class function creates a button within a `Parent` control in a random position and is used in the following sample code:

```
procedure TFormIntfConstraint.btnValueButtonClick(
  Sender: TObject);
var
  iClass: TInftClass<TButtonValue>;
begin
  iClass := TInftClass<TButtonValue>.Create;
```

```
  iClass.Set1 (TButtonValue.MakeTButtonValue (
    self, ScrollBox1));
  iClass.Set2 (TButtonValue.MakeTButtonValue (
    self, ScrollBox1));
  Log ('Average: ' + IntToStr (iClass.GetAverage));
  Log ('Min: ' + IntToStr (iClass.GetMin));
  iClass.IncreaseByTen;
  Log ('New Average: ' + IntToStr (iClass.GetAverage));
end;
```

Interface References vs. Generic Interface Constraints

In the last example I defined a generic class that works with any object implementing a given interface. I could have obtained a similar effect by creating a standard (non-generic) class based on interface references. In fact, I could have defined a class like:

```
type
  TPlainInftClass = class
  private
    val1, val2: IGetValue;
  public
    procedure Set1 (val: IGetValue);
    procedure Set2 (val: IGetValue);
    function GetMin: Integer;
    function GetAverage: Integer;
    procedure IncreaseByTen;
  end;
```

What is the difference between these two approaches? One difference is that in the class above you can pass two objects of different types to the setter methods provided their classes both implement the given interface. While in the generic version you can pass (to any given instance of the generic class) only objects of the given type. So, the generic version is more *conservative* and strict in terms of type checking.

In my opinion, the key difference is that using the interface-based version means using the Delphi reference counting mechanism, while using the generic version the class deals with plain objects of a given type and reference counting is not involved. Moreover, the generic version could have multiple constraints (like a constructor constraint) and lets you use the various generic-functions (like asking for the actual type of the generic type). This is something you cannot do when using an interface. (When you are working with an interface, in fact, you have no access to the base `TObject` method).

In other words, using a generic class with interface constraints makes it possible to have the benefits of interfaces without their nuisances. Still, it is relevant to notice that in most cases the two approaches are equivalent, and in others, the interface-based solution is more flexible.

Using Predefined Generic Containers

Since the early days of templates in the C++ language, one of the most obvious uses of generic classes has been the definition of generic containers, lists, or containers. When you define a list

of objects, like Delphi's own `TObjectList`, in fact, you have a list that can potentially hold objects of any kind. Using either inheritance or composition you can indeed define custom container for a specific type, but this is a tedious (and potentially error-prone) approach.

Delphi 2009 defines a small set of generic container classes you can find in the new Generics.Collections unit. The four core container classes are all implemented in an independent way (they don't inherit from the other), all implemented in a similar fashion (using a dynamic array), and all mapped to the corresponding non-generic container class of the Contnrs unit:

```
type
  TList<T> = class
  TQueue<T> = class
  TStack<T> = class
  TDictionary<TKey,TValue> = class
```

The logical difference among these classes should be quite obvious considering their names. A good way to test them is to figure out how many changes you have to perform on existing code that uses a non-generic container class. As an example, I've taken an actual sample program of the Mastering Delphi 2005 book and converted it to use generics.

USING TLIST<T>

The sample program has a unit that defines a `TDate` class and the main form is used to refer to a `TList` of dates. As a starting point, I added a uses clause referring to Generics.Collections, and then I changed the declaration of the main form field to:

```
  private
    ListDate: TList <TDate>;
```

Of course, the main form `OnCreate` event handler that creates the list needs to be updated as well, becoming:

```
procedure TForm1.FormCreate(Sender: TObject);
begin
  ListDate := TList<TDate>.Create;
end;
```

Now we can try to compile the rest of the code as it is. The program has a “wanted” bug, trying to add a `TButton` object to the list. The corresponding code used to compile, now fails:

```
procedure TForm1.ButtonWrongClick(Sender: TObject);
begin
  // add a button to the list
  ListDate.Add (Sender);  // Error:
    // E2010 Incompatible types: 'TDate' and 'TObject'
end;
```

The new list of dates is more robust in terms of type-checking than the original generic list pointers. Having removed that line the program compiles and works. Still, it can be improved.

This is the original code used to display all of the dates of the list in a ListBox control:

```
var
  I: Integer;
begin
  ListBox1.Clear;
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add (
      (TObject(ListDate [I]) as TDate).Text);
```

Notice the rather ugly cast, due to the fact that the program was using a list of pointers (`TList`), and not a list of objects (`TObjectList`). The reason might as well be that the original demo predates the `TObjectList` class! One can easily improve the program by writing:

```
  for I := 0 to ListDate.Count - 1 do
    Listbox1.Items.Add (ListDate [I].Text);
```

Another improvement to this snippet could come from using an enumeration (something the predefined generic lists fully support) rather than a plain `for` loop:

```
var
  aDate: TDate;
begin
  for aDate in ListDate do
  begin
    Listbox1.Items.Add (aDate.Text);
  end;
```

Finally, the program could be improved by using a generic `TObjectList` owning the `TDate` objects, but that's a topic for the next section.

As I mentioned earlier, the `TList<T>` generic class has a high degree of compatibility. There are all the classic methods, like `Add`, `Insert`, `Remove`, and `IndexOf`. The `Capacity` and `Count` properties are there as well. Oddly, `Items` become `Item`, but being the default property you seldom explicitly refer to it anyway.

SORTING A TLIST<T>

What is interesting to understand is how sorting works (my goal here is to add sorting support to the previous example). The `Sort` method is defined as:

```
procedure Sort; overload;
procedure Sort(const AComparer: IComparer<T>); overload;
```

where the `IComparer<T>` interface is declared in the Generics.Defaults unit. If you call the first version the program it will use the default comparer, initialized by the default constructor of `TList<T>`. In our case, this will be useless.

What we need to do instead, is to define a proper implementation of the `IComparer<T>` interface. For type compatibility, we need to define an implementation that works on the specific `TDate` class. There are multiple ways to accomplish this, including using anonymous methods (covered in the next section).It is also an interesting technique because it gives me the opportunity to show several usage patterns of generics and takes advantage of a *structural* class that is part of the unit Generics.Defaults that is called `TComparer`. The class is defined as an abstract and generic implementation of the interface, as follows:

```
type
  TComparer<T> = class(TInterfacedObject, IComparer<T>)
  public
    class function Default: IComparer<T>;
    class function Construct(
      const Comparison: TComparison<T>): IComparer<T>;
    function Compare(
      const Left, Right: T): Integer; virtual; abstract;
  end;
```

What we have to do is instantiate this generic class for the specific data type (`TDate`, in the example) and also inherit a concrete class that implements the `Compare` method for the specific type. The two operations can be done at once, using a coding idiom that takes a while to digest:

```
type
  TDateComparer = class (TComparer<TDate>)
    function Compare(
      const Left, Right: TDate): Integer; override;
  end;
```

If this code looks very unusual to you, you're not alone. The new class is inherited from a specific instance of the generic class, something you could express in two separate steps as:

```
type
  TAnyDateComparer = TComparer<TDate>;
  TMyDateComparer = class (TAnyDateComparer)
    function Compare(
      const Left, Right: TDate): Integer; override;
  end;
```

You can find the actual implementation of the `Compare` function in the source code, though that's not the key point I want to stress here. Keep in mind, though, that even if you sort the list its `IndexOf` method won't take advantage of it (unlike the `TStringList` class).

SORTING WITH AN ANONYMOUS METHOD

The sorting code presented in the previous section looks quite complicated and it really is. It would be much easier and cleaner to pass the sorting function to the `Sort` method directly. In the past, this was generally achieved by passing a function pointer. In Delphi 2009, this can be obtained by passing an anonymous method.

The `IComparer<T>` parameter of the `Sort` method of the `TList<T>` class, in fact, can be used by calling the `Construct` method of `TComparer<T>`, passing as parameter an anonymous method defined as:

```
type
  TComparison<T> = reference to function(
    const Left, Right: T): Integer;
```

In practice you can write a type-compatible function and pass it as parameter:

```
function DoCompare (const Left, Right: TDate): Integer;
var
  ldate, rDate: TDateTime;
begin
  lDate := EncodeDate(Left.Year, Left.Month, Left.Day);
  rDate := EncodeDate(Right.Year, Right.Month, Right.Day);
  if lDate = rDate then
    Result := 0
  else if lDate < rDate then
    Result := -1
  else
    Result := 1;
end;

procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct (DoCompare));
end;
```

If this looks too traditional, consider you could also avoid the declaration of a separate function and pass it (its source code) as parameter to the `Construct` method, as follows:

```
procedure TForm1.ButtonAnonSortClick(Sender: TObject);
begin
  ListDate.Sort (TComparer<TDate>.Construct (
    function (const Left, Right: TDate): Integer
    var
      ldate, rDate: TDateTime;
    begin
      lDate := EncodeDate(Left.Year,
        Left.Month, Left.Day);
      rDate := EncodeDate(Right.Year,
        Right.Month, Right.Day);
      if lDate = rDate then
        Result := 0
      else if lDate < rDate then
        Result := -1
      else
        Result := 1;
    end));
end;
```

This example should have whetted your appetite for learning more about anonymous methods! For sure, this last version is much simpler to write than the original covered in the previous section. Although, having a derived class might look cleaner and be easier to understand for many Delphi developers.

ANONYMOUS METHODS (OR CLOSURES)

The Delphi language has had procedural types (types declaring pointers to procedures and functions) and method pointers (types declaring pointers to methods) for a long time. Although you seldom use them directly, these are key features of Delphi that every developer works with. In fact, method pointers types are the foundation for event handlers in the VCL: every time you declare an event handler, even a pure `Button1Click` you are declaring a method that will be connected to an event (the `OnClick` event, in this case) using a method pointer.

Anonymous methods extend this feature by letting you pass the actual code of a method as a parameter, rather than the name of a method defined elsewhere. This is not the only difference, though. What makes anonymous methods very different from other techniques is the way they manage the lifetime of local variables.

Anonymous methods are a brand new feature for Delphi, but they've been around in different forms and with different names for many years in other programming languages--most notably dynamic languages. I've had extensive experience with closures in JavaScript, particularly with the jQuery (www.jquery.org) library and AJAX calls. The corresponding feature in C# is anonymous delegate.

But I don't want to devote time comparing closures and related techniques in the various programming languages, but instead describe in detail how they work in Delphi 2009.

SYNTAX AND SEMANTIC OF ANONYMOUS METHODS

An anonymous method in Delphi is a mechanism to *create a method value in an expression context*. A rather cryptic definition, but one that underlines the key difference from method pointers: the *expression context*. Before we get to this, let me start from the beginning with a very simple code example.

This is the declaration of an anonymous method type, something you need as Delphi remains a strongly-typed language:

```
type
  TIntProc = reference to procedure (n: Integer);
```

This differs from a reference method only in the keywords being used for the declaration:

```
type
  TIntMethod = procedure (n: Integer) of object;
```

AN ANONYMOUS METHOD VARIABLE

Once you have an anonymous method type you can declare a variable of this type, assign a type-compatible anonymous method, and call the method through the variable:

```
procedure TFormAnonymFirst.btnSimpleVarClick(
  Sender: TObject);
var
  anIntProc: TIntProc;
begin
  anIntProc :=
    procedure (n: Integer)
    begin
      Memo1.Lines.Add (IntToStr (n));
    end;
  anIntProc (22);
end;
```

Notice the syntax used to assign an actual procedure with in-place code to the variable. This is something never seen in Pascal in the past.

AN ANONYMOUS METHOD PARAMETER

As a more interesting example (with an even more surprising syntax), we can pass an anonymous method as parameter to a function. Suppose you have a function taking an anonymous method parameter:

```
procedure CallTwice (value: Integer;
  anIntProc: TIntProc);
begin
  anIntProc (value);
  Inc (value);
  anIntProc (value);
end;
```

The function calls the method passed as parameter twice with two consecutive integer values, the one passed as parameter and the following one. You call the function by passing an actual anonymous method to it, with directly in-place code that looks surprising:

```
procedure TFormAnonymFirst.btnProcParamClick(
  Sender: TObject);
begin
  CallTwice (48,
    procedure (n: Integer)
    begin
      Memo1.Lines.Add (IntToHex (n, 4));
    end);
  CallTwice (100,
    procedure (n: Integer)
    begin
      Memo1.Lines.Add (FloatToStr(Sqrt(n)));
```

```
    end);
end;
```

From the syntax point of view, notice the procedure passed as parameter with parentheses and not terminated by a semicolon. The actual effect of the code is to call the `IntToHex` with 48 and 49 and the `FloatToStr` on the square root of 100 and 101, producing the following output:

```
0030
0031
10
10.0498756211209
```

USING LOCAL VARIABLES

Even with a different and "less nice" syntax, we could have achieved the same effect using method pointers. What makes the anonymous methods clearly different is the way they can refer to local variables of the calling method. Consider the following code:

```
procedure TFormAnonymFirst.btnLocalValClick(
  Sender: TObject);
var
  aNumber: Integer;
begin
  aNumber := 0;
  CallTwice (10,
    procedure (n: Integer)
    begin
      Inc (aNumber, n);
    end);
  Memo1.Lines.Add (IntToStr (aNumber));
end;
```

Here the method (still passed to the `CallTwice` procedure) uses the local parameter `n`, but also a local variable in the calling context, `aNumber`. What's the effect? The two calls of the anonymous method will modify the local variable, adding the parameter to it, 10 the first time and 11 the second. The final value of `aNumber` will be 21.

EXTENDING THE LIFETIME OF LOCAL VARIABLES

The previous example shows an interesting effect, but with a sequence of nested function call, and the fact you can use the local variable isn't that surprising. The power of anonymous methods, however, lies in the fact they can use a local variable and also extend its lifetime as needed. An example will prove the point more than a lengthy explanation.

I've added (using class completion) to the `TFormAnonymFirst` form class a property of an anonymous method pointer type (well, actually the same anonymous method pointer type I've used in all of the code for the project):

```
  private
    FAnonMeth: TIntProc;
    procedure SetAnonMeth(const Value: TIntProc);
  public
    property AnonMeth: TIntProc
      read FAnonMeth write SetAnonMeth;
```

Then, I've added two more buttons to the form. The first saves an anonymous method in the property that uses a local variable (more or less like in the previous `btnLocalValClick` method):

```
procedure TFormAnonymFirst.btnStoreClick(
  Sender: TObject);
var
  aNumber: Integer;
begin
  aNumber := 3;
  AnonMeth :=
    procedure (n: Integer)
    begin
      Inc (aNumber, n);
      Memo1.Lines.Add (IntToStr (aNumber));
    end;
end;
```

When this method executes, the anonymous method is not executed, only stored. The local variable `aNumber` is initialized to zero, is not modified, goes out of local scope (as the method terminates), and is displaced. At least, that is what you'd expect from a standard Delphi code.

The second button I added to the form for this specific step is called the anonymous method and is stored in the `AnonMeth` property:

```
procedure TFormAnonymFirst.btnCallClick(Sender: TObject);
begin
  if Assigned (AnonMeth) then
  begin
    CallTwice (2, AnonMeth);
  end;
end;
```

When this code is executed, it calls on an anonymous method that uses the local variable `aNumber` of a method that's not on the stack any more. However, since anonymous methods *capture* their execution context, the variable is still there and can be used as long as that given instance of the anonymous method (that is, a reference to the method) is around.

As further proof, do the following: press the Store button once, the Call button two times, and you'll see that same *captured* variable being used:

```
5
8
10
13
```

Now press Store once more and press Call again. Why is the value of the local variable reset? By assigning a new anonymous methods instance, the old one is deleted (along with its own execution context) and a new execution context is captured, including a new instance of the local variable. The full sequence *Store – Call – Call – Store – Call* produces:

```
5
8
10
13
5
8
```

It is the implication of this behavior, resembling what some other languages do, that makes anonymous methods an extremely powerful language feature that you can use to implement something literally impossible in the past.

Other New Language Features

With so many new relevant features in the Object Pascal language, it is easy to miss some of the minor ones.

A Commented Deprecated Directive

The `deprecated` directive (used to indicate a symbol) is still available for compatibility reasons only but can now be followed by a string that will be displayed as part of the compiler warning. If you define a procedure and call it as in the following code snippet:

```
procedure DoNothing;
  deprecated 'use DoSomething instead';
begin
end;

procedure TFormMinorLang.btnDepracatedClick(
  Sender: TObject);
begin
  DoNothing;
end;
```

At the call location (in the `btnDepracatedClick` method) you'll get the following warning:

```
W1000 Symbol 'DoNothing' is deprecated: 'use DoSomething
instead'
```

This is much better than the previous practice of adding a comment to the declaration of the deprecated symbol: having a to click on the error message to get to the source code line in which this is used, jump to the declaration location, and find the comment. Needless to say, the code above won't compile in Delphi 2007, where you get the error:

```
E2029 Declaration expected but string constant found
```

The new feature of `deprecated` is used rather heavily in the Delphi 2009 RTL and VCL, while I'm expecting that third party vendors will have to refrain from using it because of the incompatibility with past versions of the compiler.

EXIT WITH A VALUE

Traditionally, Pascal functions used to assign a result by using the function name, as in:

```
function ComputeValue: Integer;
begin
  ...
  ComputeValue := 10;
end;
```

Delphi has long provided an alternate coding, using the `Result` identifier to assign a return value to a function:

```
function ComputeValue: Integer;
begin
  ...
  Result := 10;
end;
```

The two approaches are identical and do not alter the flow of the code. If you need to assign the function result and stop the current execution you can use two separate statements, assign the result and then call `Exit`. The following code snippet (looking for a string containing a given number in a string list) shows a classic example of this approach:

```
function FindExit (sl: TStringList; n: Integer): string;
var
  I: Integer;
begin
  for I := 0 to sl.Count do
    if Pos (IntToStr (n), sl[I]) > 0 then
    begin
      Result := sl[I];
      Exit;
    end;
end;
```

In Delphi 2009, you can replace the two statements with a new special call to `Exit,` passing to it the return value of the function, in a way resembling the C language `return` statement. So

you can write the code above in a more compact version (also because with a single statement you can avoid the `begin`/`end`):

```
function FindExitValue (
  sl: TStringList; n: Integer): string;
var
  I: Integer;
begin
  for I := 0 to sl.Count do
    if Pos (IntToStr (n), sl[I]) > 0 then
      Exit (sl[I]);
end;
```

NEW AND ALIASED INTEGRAL TYPES

Although this is not strictly a compiler change, but rather, an addition in the System unit, you can now use a set of easier-to-remember aliases for signed and unsigned integral data types. These are the signed and unsigned predefined types in the compiler:

`ShortInt`	`Byte`
`SmallInt`	`Word`
`Integer`	`Cardinal`
`NativeInt`	`NativeUInt`
`Int64`	`UInt64`

These types were already in Delphi 2007 and previous versions, but the 64bit ones date back only a few versions of the compiler. The `NativeInt` and `NativeUInt` types, which should depend on the compiler version (32 bit and future 64 bit) were already in Delphi 2007, but, they were not documented.

If you need a data type that will match the CPU native integer size, these are the types to use. The Integer type, in fact, is expected to remain unchanged when moving from 32-bit to 64-bit compilers.

The following set of predefined aliases added by System unit is brand new in Delphi 2009:

```
type
  Int8   = ShortInt;
  Int16  = SmallInt;
  Int32  = Integer;
  UInt8  = Byte;
  UInt16 = Word;
  UInt32 = Cardinal;
```

Although they don't add anything new, they are probably easier to use because it is generally hard to remember if a ShortInt is smaller than a SmallInt, and it is easy to remember the actual implementation of Int16 or Int8.

CONCLUSION

I've outlined a few interesting things that were added to the Delphi language, but what makes a huge difference in this version of the compiler is the support for generics, for anonymous methods, and the combination of the two. These features don't merely extend the Delphi language, but open it up for new programming paradigms beside the classic object-oriented programming and event-driven programming approaches Delphi traditionally featured. The ability of having classes parameterized on one or more data types and that of passing routines as parameters open up new coding styles and new ways of architecting Delphi applications. The language power is here in Delphi 2009, but it will take a while before libraries and components start taking full advantage of these features. Still, with Delphi 2009, you can start working out new coding techniques today.

ABOUT THE AUTHOR

This white paper has been written for Embarcadero Technologies by Marco Cantù, author of the best-selling series, Mastering Delphi. The content has been extracted from his latest book, "*Delphi 2009 Handbook*", http://www.marcocantu.com/dh2009. You can read about Marco on his blog (http://blog.marcocantu.com) and reach him at his e-mail address: marco.cantu@gmail.com.

Embarcadero Technologies, Inc. empowers application developers and database professionals with award-winning tools to design, build and run software applications in the environment they choose. With the acquisition of CodeGear from Borland® Software Inc. in 2008, Embarcadero now serves more than three million professionals worldwide with tools that are both interoperable and integrated. From individual software vendors (ISVs) and developers to DBAs, database professionals and large enterprise teams, Embarcadero's tools are used in the most demanding vertical industries in 29 countries and by 90 of the Fortune 100. The company's flagship tools include: Embarcadero® Change Manager™, CodeGear™ RAD Studio, DBArtisan®, Delphi®, ER/Studio®, JBuilder® and Rapid SQL®. Founded in 1993, Embarcadero is headquartered in San Francisco, with offices located around the world. For more information, visit www.embarcadero.com.