Essential Delphi 8 for .NET
by Marco Cantù

(Free) Chapter 3:
The Delphi Language

Version 0.03 – April 2nd, 2004

Chapter 3: The Delphi Language in Delphi 8 for .NET

The transition from Delphi 7 to Delphi 8 marks the most relevant set of changes Borland has made to the Object Pascal (or Delphi) language probably since Delphi 1 came to light. There are several reasons for this change, but the most relevant is certainly the need to obtain a high degree of compatibility with the underlying architecture of .NET, which in turn makes the Delphi language compatible with other .NET languages.

On one hand, obtaining full language-features compatibility with other .NET languages is critical to be fully interoperable, so that programmers using other languages can use assemblies and components written in Delphi while Delphi programmers have all of the features of the .NET libraries immediately available. On the other hand, having an existing implementation (at the CIL level) of a number of core language features has made Borland's job of updating the language somewhat easier (although this was more the theory than it has been the practice, according to insiders).

Notice that I tend to use the terms Delphi Language and Object Pascal Language almost interchangeably. For a number of years, the language of the Delphi IDE was officially called Object Pascal by Borland. Starting with Delphi 7, the company formally announced the new name for the language itself, to highlight the core ancestry of the tool with other, like the Kylix for Linux (of which there is a Delphi version) and Borland's .NET IDE, often indicated with the codename "Galileo", which has a Delphi and a C# personality.

This chapter is focused on the changes Borland has made to the language since Delphi 7, it is not a complete exploration of the language itself. You'll only find an introductory chapter that summarizes the core features of the Delphi language, explaining why it is a good choice for .NET development (and not only for .NET).

The Delphi Language: An Assessment

To put it shortly, the Delphi language is a modern OOP language based on the Pascal language syntax. Its Pascal roots convey to Delphi a few relevant features:

- Delphi code is quite verbose (for example you type begin instead of an open brace) but this tends to make the code highly readable
- By spelling out your intentions more clearly (compared to the C-family of languages) there are less chances the compiler will misunderstand your program, while at the same time it is more likely you'll receive an error or a warning message indicating your code is not clear enough.
- The language is type safe, not only at the OOP level but even at the level of the base types. Enumerations, characters, integers, sets, floating point numbers cannot be freely mixed as they represent different types. Implicit type conversions are very limited.

You can learn more about the key features of the base Pascal language (the non-OOP subset of Delphi) by reading my free ebook Essential Pascal, available on http://www.marcocantu.com/epascal.

Focusing on OOP features, Delphi provides a very modern language, much closer to C# and Java than C++. In Delphi objects are invariably allocated on the heap, there is a single base class for all of the classes you define, features like properties and interfaces are built into the language, and RAD capabilities are based on the simple idea that components are classes.

Delphi actually did introduce some of features that were alter ported to other languages. In particular, Turbo Pascal and Delphi 1 chief architect was Anders Hejlsberg, who later moved to Microsoft to become a key architect of the .NET framework and the C# language (his title at is something like "Distinguished Engineer and Chief C# Language Architect "). His influence (and the Delphi influence) is clearly visible. The reason this is important is that considering this fact it should come to no surprise that the Delphi language has been moved to .NET more easily than most other languages (including VB and C++), and your OOP code maintains a high degree of compatibility from Delphi 7 for Win32 to Delphi 8 for .NET.

One of the advantages you have right using Delphi, in fact, is that you can compile your code (often the same source code) on different platforms: Win32, .NET and (with more limitations) even Linux.

The other relevant thing to mentions is that both traditional Delphi and its new .NET incarnations lack some language features of C# or Java, but also provide many constructs not found in those other languages. I'll shortly cover those features later in this chapter. First, I need to cover how the Delphi language adapts to .NET.

Good Ol' Units

One of the relevant differences between the Delphi language and the C# language is that the latter is a pure OOP language, a term used to indicate that all the code must be inside methods, rooting out global functions (or procedures). The Delphi language, instead, allows for both programming paradigms, like C++.

Here I don't want to enter a technical dispute whether a pure OOP language is any better than an OOP language supporting procedural programming. I certainly do find quite awkward seeing libraries with classes not meant to be instantiated but used only to access a plethora of class functions. In these case the classes are containers of global routines, exactly like modules in advanced procedural languages.

Globals and the Fake Unit Class

Moving Delphi to .NET, Borland had to find a way to maintain the existing model with global functions and procedures, and global data. This effect is obtained by creating a "fake" class for each unit, and adding to it global procedures and functions like they were class methods and global data like it was class data. For example, the UnitTestDemo program has a unit with a single global function, Foo:

```
unit Marco.Test;

interface
```

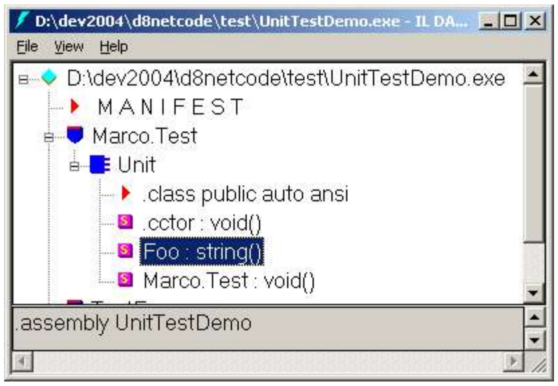
```
function Foo: string;

implementation

function Foo: string;
begin
  Result := 'foo';
end;

end.
```

By adding this function to a program, compiling it, and inspecting it with IL DASM, you can see (in the figure below) that the Marco.Test unit has a Unit fake class with the Foo global function.



By looking at Foo, in fact, you'll get the following signature, with static indicating this is a class method:

```
.method public static string  Foo() cil managed
```

Units as Namespaces

The other relevant way in which units are used, is that they become namespaces. A Delphi compiled assembly has as many namespaces as there are units in its source code. What's brand new (well, almost as this technically works also in Delphi 7) is that you can have units with long names, using the dot notation. See for example the code of the unit Marco.Test a few paragraphs above. To be consistent with the Delphi notation, the filename of such a unit is called Marco.Test.pas.

Similarly, from Delphi code you can access to CLR namespaces and namespaces defined by other assemblies exactly as if they were native units, with the uses notation:

```
uses
  System.IO.Text, System.Web;

uses
  Marco.Test;
```

Notice that this can be confusing as the system namespaces contains large numbers of classes while a single Delphi unit should be limited in size to remain manageable. Again technically a unit defines a namespace so that the uses statements above are functionally equivalent. However I wish Delphi for .NET had an option to "condense" multiple units in a single namespace. As far as I know this feature is not available out of the box.

Regarding namespaces and units, it's important to notice that Delphi classic Win32 library units, like SysUtils or Classes have been prefixed with the Borland.VCL name. So in theory when porting an existing project you should modify lines like:

```
uses
  Classes, SysUtils;
```

into

```
uses
  Borland.Vcl.Classes, Borland.Vcl.SysUtils;
```

However, Delphi 8 for .NET allows you to indicate a namespace search path (or a list of project namespaces), like Borland.Vcl, so you can omit the first portion of the declaration and the compiler will pick up the correct unit anyway. In practice you can add a list of namespace prefixes in the Directories/Conditionals page of the Project Options dialog box (the same effect can be obtained with the -ns compiler flag, like -nsBorland.Vcl). This approach eliminates the tedious use of IFDEFs Delphi programmers had to deal with for VCL/CLX compatibility between Delphi and Kylix, or the two libraries on Windows alone.

The "Non-Existing" Unit Aliases

The Delphi for .NET compiler suggests the possibility of defining an alias for a unit having a long name (apparently only Delphi internal units, not system namespaces) with a syntax like:

```
uses
  Marco.Components.FunSide.Labels as FunLabels;
```

Not only I've not been able to make this work, but R&D members have confirmed that the documentation is wrong and this feature is in fact not avaialble.

Unit Initialization and Class Constructors

In the context of how unit are adapted to a pure OOP structure, a relevant change relates to the initialization and finalization sections of a unit, which are "global" potions of code executed (in Delphi for Win32) at the start and termination of a program in a deterministic order determined by the sequence of inclusion (uses statements).

In Delphi for .NET, units become fake classes and the initialization code becomes a class static method (see later) that is invoked by the class constructor (see later, again). As class constructors are automatically invoked by the CLR before each class is used, the resulting behavior is similar to what we were used to. The only difference, which is very relevant, is that there is no deterministic order of execution for the various class constructors in a program. That is, the order of execution of the various initialization sections is not known and can change for different executions of the program.

As as example consider this initialization section that sets a value for a global variable (that is, global within a unit):

```
initialization
  startTime := Now;
```

Compiling this code adds to the unit class a static method with same name of the unit (Marco.Test) and a call to this method from the class constructor, as the following IL snippets demonstrate:

```
.method public static void  Marco.Test() cil managed
{
  // Code size       7 (0x7)
  .maxstack  1
  .locals init ([0] valuetype [Borland.Delphi]
    Borland.Delphi.System.TDateTime startTime)
  IL_0000:  call       valuetype [Borland.Delphi]
    Borland.Delphi.System.TDateTime [Borland.VclRtl]
    Borland.Vcl.SysUtils.Unit::Now()
  IL_0005:  stloc.0
  IL_0006:  ret
} // end of method Unit::Marco.Test
.method private hidebysig specialname rtspecialname static
        void  .cctor() cil managed
{
  // Code size       36 (0x24)
  .maxstack  1
  ... // more IL code omitted
  IL_001e:  call       void Marco.Test.Unit::Marco.Test()
  IL_0023:  ret
} // end of method Unit::.cctor
```

Identifiers

With multiple languages and a single CLR, chances are that a reserved symbol of a language will be legitimately used by programmers using another language. In the past, this would have made it very hard to refer to that symbol from the code. Delphi 8 introduces a special symbol (&) you can use as a prefix for any reserved word, to legitimately use it as an identifier. This is called a "qualified identifier".

To be more precise, the & allows you to access CLR defined symbols (or other symbols in general, whenever they are defined), but won't allow you to declare identifiers within Delphi code that violate Delphi's own syntax rules ("& is for access only, not declaration" as Borland says). For example, you can use & when referring to the base class of a class, but not when defining the name of a new class. I'd say, this makes a lot of sense.

The most classic example is the use of the Label class of the WinForms library. As the label word is reserved in the Delphi language you can either use the full name space to it or prefix it with the &. The following two declarations are identical:

```
    Label1: System.Windows.Forms.Label;
    Label2: &Label;
```

An issue that Delphi 8 doesn't address, instead, is the use of Unicode (UTF8) identifiers that the CLR allows and Delphi currently doesn't support. R&D members have told they've looked into it with the trouble being that a change to the string type of the identifier tables of the compiler could badly affect the compiler performance, so we'll have to wait for a future version to see this issue tackled. By the way, the & symbol will also work for UTF8 characters in the future.

Base Data Types

Contrary to what happened in the past, when the Delphi language base data types had to partially adapt to the underlying CPU (for example in relation to the handling of floating point numbers), now the language must comply with the Common Type System specification indicated in the CLR. Of course most of the base data types are still there and others have been built by Borland on top of what's available.

The CLR makes a clear distinction between two different families of data types, value types and reference types:

- Value types are allocated directly on the stack and when you copy or assign a value type the system will make a complete copy of all of its data. Value types include the primitive data types (integer and floating point numbers, characters, boolean values) and records.
- Reference types are allocated on the heap and garbage collected. Reference types include anything else, from objects to strings, from dynamic arrays to class metadata.

Primitive Types

The .NET CLR defines quite a few "primitive types", which are not natively mapped to objects but a direct representation (not predefined, as it can change with the target CPU and operating system version, for example on a 32biit or 64bit CPU). The CLR primitive types are mapped to corresponding Delphi types, as the following table highlights.

Category	**Delphi Type**	**CLR Type**
Generic size	Integer	System.Int32
	Cardinal (equals LongWord)	System.UInt32
Signed	ShortInt	System.SByte
	SmallInt	System.Int16
	Integer	System.Int32
	Int64	System.Int64
Unsigned	Byte	System.Byte
	Word	System.UInt16
	LongWord	System.UInt32
Floating point	Single	System.Single
	Double	System.Double
	Extended	Borland.Delphi.System.Extended
	Comp (equals Int64, deprecated)	System.Int64
	Currency	Borland.Delphi.System.Currency (a record based on Decimal)

Category	Delphi Type	CLR Type
	Decimal	System.Decimal
	Real (equals Double)	System.Double
char	Char	System.Char
	WideChar (equals Char)	System.Char
boolean	Boolean	System.Boolean
	ByteBool	Borland.Delphi.System.ByteBool
	WordBool	Borland.Delphi.System.WordBool
	LongBool	Borland.Delphi.System.LongBool

What you can notice is that not all Delphi types have a CLR equivalent. You are recommended not to use those types if you want your classes to be usable by other .NET languages and fully CLS-compliant. Feel free to use them inside a program and for compatibility with existing Delphi code, but whenever possible stick to the types with a direct CLR mapping.

If you don't trust me, the data here is taken from the output of the PrimitiveList program. The program passes as parameters to the PrintDelphiType function a long list of Delphi types, occasionally with a comment. The function uses Delphi's RTTI and the system reflection to print out to the console the Delphi and CLR type names. Here is the relevant code:

```
procedure PrintDelphiType (tInfo: TTypeInfo;
  strComment: string = '');
begin
  write (tInfo.DelphiTypeName);
  write (' - ');
  write (tInfo.ToString);
  writeln (' ' + strComment);
end;

// snippet from main program
begin
  writeln ('');
  writeln ('generic size');
  PrintDelphiType (typeInfo (Integer));
  PrintDelphiType (typeInfo (Cardinal),
    'declared as Cardinal');

  writeln ('');
  writeln ('specific size: signed ');
  PrintDelphiType (typeInfo (ShortInt));
  PrintDelphiType (typeInfo (SmallInt));
  PrintDelphiType (typeInfo (LongInt));
  PrintDelphiType (typeInfo (Int64));
```

The old Real48 types, representing six-byte floats and already deprecated in the last few versions of Delphi, is not available any more. Not being directly mapped to a supported FPU type of the Pentium-class CPUs, its implementation was slow in Delphi for Win32 and is not part of the CLR. The Real type, instead, is directly mapped by the compiler to Double.

Boxing Primitive Types

Primitive types can be boxed into object wrappers to convert them into numbers:

```
var
  n1: Integer;
  o1: TObject;
begin
  n1 := 12;
  o1 := TObject (n1);
```

This is very handy in a number of cases, including the possibility of using object container classes to hold primitive types along with objects, the notion that an object reference can host anything (including a primitive value), the and ability to apply some of the predefined methods of the base Object class (like the ToString method) to any value:

```
var
  n1: Integer;
  str: string;
begin
  n1 := 12;
  str := TObject (n1).ToString;
```

Of course, one could argue that a pure object-oriented language should root out the use of primitive types and make use of objects for everything (as Smalltalk, the father-of-all-OOP-languages, does), but efficiency reasons demand to keep primitive types close to the system. As a simple test, the BoxingDemo application saves the intermediary value of a computation in an integer boxed in an object, with the time take to box and unbox it far exceeding the time taken for the computation (which, in this extreme case, involves only adding numbers). Similarly, I'd expect an IntToStr call to be much faster than boxing the value into an object and than applying to ToString method to it.

The other operation demonstrated by the application is the definition of a list of objects accepting boxed elements and based on Delphi own container classes:

```
var
  list: TObjectList;
  i: Integer;
begin
  list := TObjectList.Create;
  list.Add(TObject (100));
  list.Add(TObject ('hello'));
  list.Add (button3);

  for i := 0 to list.Count - 1 do
  begin
    Memo1.Lines.Add (list[i].ToString);
  end;
```

Running this code the program adds to the memo the following output:

```
100
hello
Button3 [Borland.Vcl.StdCtrls.TButton]
```

Delphi Specific Ordinal Types

Enumerators and sets are ordinal types of the Pascal language, commonly used in Delphi. The idea of enumeration is not strictly part of the C language, which expresses a similar idea with the use of a sequence of numeric constants, thus making the enumerated values equivalent to numbers. A form of strictly-typed enumerators has been added to C# language and is part of the CLR types system. This means that Delphi enumerations are mapped to a corresponding CLR features, as you can see by inspecting compiled code.

For example, the definition of this enumeration from the Borland.Delphi.System unit:

```
  TTextLineBreakStyle = (tlbsLF, tlbsCRLF);
```

gets transformed in the following "compiled" definition (visible with ILDAMS), which looks like an enumeration class (marked as sealed) that inherits from System.Enum and has a single value (a short unsigned integer) and two literal constants:

```
[ENU] TTextLineBreakStyle
 .class public auto ansi sealed
 extends [mscorlib]System.Enum
 [STF] tlbsCRLF : public static literal valuetype
   Borland.Delphi.System.TTextLineBreakStyle
 [STF] tlbsLF : public static literal valuetype
   Borland.Delphi.System.TTextLineBreakStyle
 [FLD] value__ : public specialname rtspecialname unsigned int8
```

The Delphi set type, instead, is not commonly found in many other programming languages. This is why the CLR has no clue about sets and why they are not CLS-compliant. Delphi for .NET implementation of sets is based on the following declaration, plus some compiler magic:

```
type
  _TSet = Array of Byte;
```

Records on Steroids

Another relevant family of value types is represented by structures in C# jargon, or records as they are called in Delphi. If records have always been part of the language, in this version they gain a lot of new ground as records can now have methods associated with them (and even operators, as we'll see later on in this chapter).

A record with methods is somewhat similar to a class: the most relevant difference (beside the lack of inheritance and polymorphisms) is that record type variables use local memory (of the stack frame they are declared onto or the object they are part of), are passed as parameters to functions by value, making a copy, and have a "value copy" behavior on assignments. This contrasts with class type variables that must be allocated on the dynamic memory heap, are passed by references, and have a "reference copy" behavior on assignments (thus copying the reference to the same object in memory).

For example, when you declare a record variable on the stack you can start using it right away, without having to call its constructor. This means record variables are leaner (and more efficient) on the memory manager and garbage collector than regular objects, as they do not participate in the management of the dynamic memory. These are they key reasons for using records instead of object for small and simple data structures.

```
type
  TMyRecord = record
  private
    one: string;
    two: Integer;
    three: Char;
  public
    procedure Print;
    constructor Create (aString: string);
    procedure Init (aValue: Integer);
  end;
```

A record can also have a constructor, but the record constructors must have parameters (if you try with Create(); you'll get the error message "Parameterless constructors not allowed on record types". This behavior is far from clear to me, as you still have to manually call the constructor, optionally passing parameters to it (I mean it seems you cannot use the constructor as a type conversion, something you can use the Implicit and Explicit operators for, as discussed later). Here is a sample snippet:

```
var
  myrec: TMyRecord;
begin
  myrec := TMyRecord.Create ('hello');
  myrec.Print;
```

Unless there is another way to call a record constructor, for example to initialize a global variable or a field, I'm far from sure why this constructor syntax has been added. Notice, in fact, that using the plain syntax above doesn't affect the initialization portion of the CIL code. In this respect, it seems a better idea to use a plain initialization method rather than a constructor to assign multiple (initial) values to a record structure.

From Borland Object Pascal days, Delphi for .NET has left out the *object* type definition, which predates Delphi as it was introduced in the days of Turbo Pascal with Objects. The reason is that .NET provides extended records (with methods) that are value types and sit on the stack or in the container type exactly like objects defined with the *object* keyword in past versions of Delphi. This is another deprecated language feature that few Delphi programmers use, so its absence should not constitute a big roadblock.

Delphi for .NET still allows you to define either a record or a packet record. The difference is traditionally related to 16-bit or 32-bit alignment of the various fields, so that a byte followed by an integer might end up taking up 32 bits even if only 8 are used. The reason is that accessing the following integer value on the 32-bit boundary makes the code faster to execute.

In Delphi for .NET two syntaxes produce structures marked as auto (for a plain record) or sequential (for a packed record). How this data ends up being mapped by the system is probably an implementation specific feature of the CLR so that different platforms can use different approaches. But you can give the CLR a hint...

Records or Classes

Having classes in the language, someone might wonder what's the rationale for having also records with methods. Beside the lack of many advanced features of classes, the key difference between records and classes relates to the way the use memory.

The bytes needed for the storage of the fields of a record comes from local memory: (i) the stack if the record variable is a local variable or (ii) the memory of the hosting data structure if the record is inside another type (an array, another record, a class...). At the opposite, a class variable or field is just a reference to the memory location where the class is held. This means classes need a specific memory allocation, their data blocks participate in the memory management (including the garbage collector), and they must eventually be disposed. Records just sit there on their own, and cost much less to allocate, manage, and free.

Another relevant difference is in the way records are copied or passed as parameters. In both cases the default is to make a full copy of the memory block representing the record. Of course, you can use var or const record parameters to modify this default behavior. On the contrary assigning or passing an object is invariably an operation on the references to the objects: It's the reference that is copies or passed around.

To see somewhat the performance differences among using objects and records, I've added to the RecordsDemo example, already mentioned earlier, two units which are extremely similar but use the two approaches. The rather dumb code I've written uses a temporary record or object inside a routine that is called a few million times... although the difference depends on weight of the actual algorithm (I had to remove the calls to random() I originally used as they slow down things too much) and the amount of data of the structures, the current result of the RecordsDemo test on my computer gives a relative speed of 360ms for records against 540ms for classes. The difference is relevant, but you save 180ms when you allocate memory 2 million times: this means that is computing-intensive operations the difference is certainly worth, while in most other cases it will be almost unnoticeable.

So the difference from records to classes boils down to the fact you might save one line of code of memory allocation, that you might still need to replace with a call to an initialization method.

Delphi New Predefined Records

The presence of a sophisticated record type definition and of the operators overloading has led Borland to change a lot of predefined Delphi types turning them into records. Notable examples are the types variant, datetime, and currency.

Variants in particular represent another data type not part of the CLR foundations that Borland has been able to redefine on top of the available CLR features without affecting the syntax and the semantic of the existing code. The implementation of variants on .NET differs heavily from that of Win32, but your code (at least the higher level code) won't be affected.

I'll cover these Borland predefined data types in Chapter 4, which is devoted to Delphi 8 RTL.

> Considering the relevant changes on the variant data type it should come to no surprise that the TVarData type has disappeared and the VarUtils unit is no longer present. Notice also that in a few circumstances you'll need to add a reference to the Variants unit for some existing Delphi programs to recompile under Delphi for .NET.

Reference Types

As mentioned earlier, the key different between value types and reference types is that reference types are allocated on the heap and garbage collected. The class type is the most obvious example of a reference type, but also strings and Objects are the most obvious examples of a reference type, but strings and dynamic arrays are part of the same category as well (which is not much different from what Delphi programmers were used to, a part from the garbage collection support).

More details on the Garbage Collector were already in Chapter 1, while a discussion on how this affects Delphi objects destruction will take place later in this chapter.

Strings

Strings are ... just strings and considering that Delphi long strings are allocated on the heap, reference counted and use the copy-on-write technique, .NET strings will be quite familiar. There are a few differences, though. The first is that strings on .NET use UTF16 Unicode characters, that is each character is represented with 16 bits of data (2 bytes). This is transparent, as when you index into a string looking for a given character, the index will be that of the character, not that of the byte (the two concepts are usually identical on Delphi for Win32). Of course, using UTF16 means that the strings will use on average twice as much memory than in Delphi for Win32 (that is, unless you used the widestring type on Win32).

The PrimitiveList program tells us, again, that the available Delphi string types have the following mappings to CLR types:

Category	**Delphi Type**	**CLR Type**
strings	string	System.String
	AnsiChar	Borland.Delphi.System.AnsiChar
	ShortString	Borland.Delphi.System.ShortString
	AnsiString	Borland.Delphi.System.AnsiString
	WideString	System.String

Another relevant issues is that strings in .NET are immutable. This means that string concatenation is slow when done with the classic + sign (and the already obsolete AppendStr routine is now gone). In fact, a new string has to be created in memory copying the content of the two strings being added, even if the effective result is adding some more characters to one of the strings. To overcome this slow implementation, the .NET framework provides a specific class for string concatenation, called StringBuilder.

For example, if you have a procedure like PlainStringConcat below that creates a string with the first 20,000 numbers, you should rather re-implement it using a StringBuilder object like in the following UseStringBuilder function:

```
function PlainStringConcat: Integer;
var
  str: string;
  i, ms: Integer;
  t: tDateTime;
begin
  t := Now;
  str := '';
  for I := 1 to 20000 do
  begin
    str := str + IntToStr (i) + ' - ';
  end;
  ms := trunc (MilliSecondSpan(Now, t));
  writeln (str[100]);
  Result := ms;
end;

function UseStringBuilder: Integer;
var
  st: StringBuilder;
  str: string;
  i, ms: Integer;
  t: tDateTime;
begin
  t := Now;
  st := StringBuilder.Create;
  for I := 1 to 20000 do
  begin
    st.Append (IntToStr (i) + ' - ');
  end;
  str := st.ToString;
  ms := trunc (MilliSecondSpan(Now, t));
  writeln (str[100]);
  Result := ms;
end;
```

If you run the StringConcatSpeed demo that includes the two functions above you'll see the time taken by each of the two appraoches, and the difference will be striking! This is the output (you should try out with different ranges of the for loop counter):

```
String builder: 19
Plain concat: 10235
```

This means that the string builder takes a few milliseconds while the string concatenation takes about 10 seconds. The morale of this story is that you have to get rid of all of the lengthy string concatenations in your code using either a StringBuilder or a string list. The second approach is a little slower (takes almost twice as much as the StringBuilder, which seems acceptable for most tasks) but has the advantage of maintaining your code compatible with Delphi 7. You can certainly obtain the same compatibility by writing a StringBuilder class for Delphi 7, but using a string list seems a better approach to me if you need backward compatibility.

[TODO: short strings performance test]

[TODO: cover dynamic arrays shortly?]

[TODO: using a const parameter when passing strings in Delphi 8: is it still worth as in Delphi 7?]

In the .NET FCL (and as a consequence also in Delphi's RTL) every object has a string representation, available by calling the *ToString* virtual method. System library classes already define a string representation, while you can plug in your own code for your custom classes. This is covered in more details and with an example in Chapter 4, focusing on the RTL.

Using Unsafe Types

In Delphi for .NET the use of pointers and other unsafe types is not totally ruled out. However, you'll need to mark the code as unsafe and have to go along a lot of troubles to obtain soemthing you could easily and better achieve on the Win32 platform. In other words, although I'm going to show you a few tricks in this section, I'm not recommending at all the use of these techniques, quite the contrary.

A second disclaimer, at least for now, is that these features are mostly undocumented and I'm having a hard time to figure out how to make them work, so this section is still mostly incomplete and is just a description of some hardly working experiments.

As a general rule, notice that you can ask the Delphi for .NET compiler to permit the generation of unsafe code with the directive:

```
{$UNSAFECODE ON}
```

After you've used this directive you can mark global routines or methods with the unsafe directive. The application you'll produce will be a legitimate .NET application, possibly managed, but certainly not safe. Running it through PEVerify should fail.

When declaring an *unsafe* method, you mark the method with this directive in the method definition in the implementation section, not in the method declaration (within the class definition) in the interface section. That is, unless the method takes unsafe types as parameters, in which case the unsafe directive goes in the method declaration.

Variant Records

Let's start with variant records. These are data structure with fields that can assume different data types either depending on a given field (as in the example below, taken from the UnsafeTest project) or in an undetermined way:

```
type
  TFlexi = record
  public
    value: integer;
    case test: Boolean of
      true: (c1: Char; c2: Char);
      false: (n: Integer);
  end;
```

If you compile this code the compiler issues the warning "Unsafe type TFlexi", which is certainly correct. Anyway, you can use it in a method like the following, which saves data using a format (the 'a' and 'b' characters) and retrieves it with another (the 6,422,625 number):

```
procedure UseVariantRecord;
var
  flexi: TFlexi;
begin
  writeln ('using variant record');
  flexi.test := true;
  flexi.c1 := 'a';
  flexi.c2 := 'b';
  flexi.test := false;
  writeln (IntToStr (flexi.n));
end;
```

Notice that this works even without marking the procedure unsafe or marking the module as such. The most relevant rule for variant records in Delphi 8 for .NET is that you cannot use reference types (managed data) in the overlapping portion of the type. So if you try the following data structure:

```
type
  TFlexi2 = record
  public
    value: integer;
    case test: Boolean of
      true: (s: string);
      false: (n: Integer);
  end;
```

as soon as you try to use a local variable of this type, at runtime you'll get a CLR exception like: "Could not load type UnsafeTest.TFlexi2 from assembly UnsafeTest (...) because it contains an object field at offset 5 that is incorrectly aligned or overlapped by a non-object field".

Untyped Parameters

Another risky technique is the use of untyped parameters in procedures. Here is an example (again from the UnsafeTest project) that works:

```
procedure UnsafeParam (var param);
begin
  writeln (tObject(param).ToString);
end;

var
  test: string;
  n: Integer;
  obj: TObject;
begin
  test := 'foo';
  UnsafeParam (test);
  n := 23;
  UnsafeParam (n);
  obj := TObject.Create;
  UnsafeParam (obj);
```

In this case you'll get the output you'd expect (and no compile time warning or runtime error):

```
foo
23
System.Object
```

before you get too surprise by this behavior, notice that the compiler generates a method with the following signature, which has an object passed by reference (& in C is like var in Pascal), which means the compiler user objects and boxing to simulate an undefined type:

```
.method public static void UnsafeParam(object& param) cil
managed
```

Allocating Memory with New

Finally, you cannot use GetMem, FreeMem, ReallocMem any more. They have been removed from the standard routines. You can still use New (someone says also Dispose, but I wasn't able to find it) but only when allocating a dynamic array. In fact if you try to use New in another context, you'll get the compiler error message "NEW standard function expects a dynamic array type identifier". Here is an example (again from the UnsafeTest project) of this usage:

```
type
  arrayofchar = array of Char;
var
  ptest2: arrayofchar;
begin
  ptest2 := New (arrayofchar, 100);
```

Using the PChar type

Another issue relates to the use of PChar pointers. If you simply try using the PChar type, the compiler will stop with an error indicating that: "Unsafe pointer variables, parameters or consts only allowed in unsafe procedure". This is solved by using the unsafe directive to mark the procedure or method.

Bu the actual question is, what exactly can you do with a PChar? I don't know for sure, but my impression is that you can do very little. This is an example (again in the UnsafeTest project), which basically takes uses a PChar to refer to a character in the array declared above and modify it:

```
procedure testpchar; unsafe;
var
  ptest: PChar;
begin
  ptest := @ptest2 [5];
  ptest^ := 'd';
  writeln (ptest2 [5]);
end;
```

The code does work as expected, but it is hard to tell why one might want to use it rather than accessing the dynamic array directly.

A related technique is the use of the GCHandle class of .NET, a sort of parent of the Object class, which allows you to "pin down" an object and get a pointer to its memory location to work with, while the CLR guarantees that the object won't be moved in memory (for example, as a result of a garbage collection). There is some code introducing this technique in the UnsafeTest project, but nothing really interesting for now.

[TODO: Finish exploring the topic here or delay coverage of GCHandle and pointers to the RTL chapter.]

The file of Type is Gone

Very few of the traditional Pascal and Delphi types are missing in Delphi for .NET. A notable absence, albeit seldom used, is the file of <type> construct of the traditional Pascal language.

Type Casts On the Safe Side

The Delphi language tends to force the developers to use the type system in an appropriate way. By treating the base data types as different entities (if you compare it, for example, with the C language) programs tends to be better written and more readable. For example, an enumeration is not the same as an integer, an integer is not type-compatible with a character, and so on.

Still the Delphi language allows you to code a direct type cast, imposing your own rule on top of what the compiler generally allows you to do. When there is direct cast, the Delphi compiler for Win32 gives up. So you can cast an object into an object of another type, cast an object reference to an integer (holding the address of the memory location of the object, not the object's data), and the like. These are unsuggested and unsafe practices, but they are somewhat frequent in Delphi.

As .NET CLR has a high regard for type safety (a precondition for having a safe application) some of the direct type cast capabilities we are used to don't apply any more, or apply in a different way. The first example is when you cast an object to a class type different than its own. While in Delphi 7 the cast was always allowed (optionally ending up with a nonsense piece of code), Delphi 8 for .NET treats any cast among class types like they were as casts. This means that any cast among classes is checked for type compatibility, with the rule that an object of an inherited class is compatible with each of its base class types.

This is slower than a direct cast, but it certainly safer. Where Delphi on Win32 has a syntax to express the two types of casts, in Delphi for .NET they are both converted into the same code (the as cast):

```
anotherObject := TAnotherClass (anObject);  // direct
anotherObject := anObject as TAnotherClass; // safe
```

There is only an exception to this rules, which is the case the compiler spots the use of the protected hack , as discussed later in a specific section on this technique.

A totally different case is the cast of a primitive type (let's say an integer) to a class type. Instead of a cast involving the value of the reference (as on Win32), you end up with the boxing of the native value:

```
anObject := TObject (aNumber);
```

Thus you obtain an actual object (not just a fake reference), which is a new object containing the value you are casting. This value can be cast back to the original native type, with a notation like:

```
aNumber := Integer (anObject);
```

As this is a controlled cast, casting to integer a regular object (not a boxed integer) will cause an error. This breaks the existing Delphi code, which formally allows you to cast any object to an integer to extract the value of the reference. Of course, this was not a good way to write code in the first place, so you shouldn't really complain!

I won't even touch on the idea of casting object refereces to pointers and possibly even manipulating those pointers, as almost none of the pointer operations are allowed in a safe and managed .NET application.

Finally, notice that you can define custom type cats, or custom data type conversions, with the Implicit and Explicit operators (see later in the section devoted to Operator Overloading).

Classes Gain New Ground

To fully support the class model of the CLR, Borland had to tweak the Delphi language in a few ways, including making changes to classes. With most of the traditional features of classes left unchanged, there are few relevant new features. The most notables are the definition of visibility rules more on line with other languages, the support for nested types, and concept of class helpers.

When Private is Really Private

Differently from most other OOP languages, Delphi had a more relaxed rule for private visibility. In fact, access specifiers where enforced only across units and not for classes within the same unit. In other words, a class could access to private fields and methods of another class defined in the same unit. The same happened with protected members.

The first important thing to consider is that to maintain source code compatibility the behavior of the private and protected keywords has not changed. This means your existing code taking advantage of this feature will keep working. The only change is that protected symbols defined in another unit are accessible only within the same assembly/package, but not across package boundary (as they used to be in Delphi 7). In fact, Delphi's protected access specifier is mapped to CLR assembly access specifier. In other words, Delphi protected specifier reverts to strict protected across assembly boundaries.

For CLR compatibility, Borland has added two more access specifiers, strict private and strict protected. These correspond to the CLR private and protected specifiers and behave like you'd expect, which means other classes within the same unit cannot access strict private symbols of a class and can access strict protected symbols only if they inherit from that class.

As an example of the syntax and error messages you'll get see the ProtectedPrivate demo in the LanguageTest folder of this chapter's code. The demo has a couple of classes and code using them, and you can experiment with it by changing the access specifiers. The base class code is the following (listed here only to show the exact syntax):

```
type
  TBase = class
  strict private
    one: Integer;
  strict protected
    two: integer;
  end;
```

There are a couple of things to notice. The first is that the error message you'll get when you try to access a non-visible member is quite readable ("Cannot access protected symbol TBase.two") while in the past it used to be very cryptic ("Undeclared identifier"). The second important element is that the syntax of this feature has changed since the original Delphi for .NET preview distributed with Delphi 7. The preliminary syntax was class private, now replaced with strict private.

To summarize, the following table lists correspondences between Delphi and the CLR.

Delphi	CLR
private	assembly
strict private	private
protected	famorassem (family or assembly)
strict protected	family
public	public
published	public

The Protected Hack Still Works!

Longtime Delphi programmer's certainly know that there is a trick allowing you to access to any protected data of another class, even if this class is declared in a different unit. This trick, often called the protected hack is described as follows in Mastering Delphi 7.

[TODO: add text here from MD7, use special formatting]

In Delphi for .NET the protected keyword has kept the same meaning, but one of the key element of the protected hack is the ability to cast another of a class to its fake subclass, something the CLR won't allow you to do (see the section "Cast on the Safe Side" above). However, in case the compiler recognizes you are using the protected hack (that is, when it notices a weird typecast to an empty subclass to access a protected member) it ignores the cast but lets you access the protected member anyway.

This is demonstrated by the ProtectedHack demo, which has a class with a protected member:

```
type
  TTest = class
  protected
    ProtectedData: Integer;
  public
    PublicData: Integer;
    function GetValue: string;
  end;
```

The main form uses the protected hack to access to the data, as follows:

```
type
  TFake = class (TTest);

procedure TForm1.Button2Click(Sender: TObject);
var
  Obj: TTest;
begin
  Obj := TTest.Create;
  Obj.PublicData := 10;
  TFake(Obj).ProtectedData := 20;
  ShowMessage (Obj.GetValue);
```

The generated IL code (in the following listing) shows that the compiler skips the type cast and produces the same code for the access to the public and protected data:

```
.method public instance void  Button2Click(object Sender)
            cil managed
{
  // Code size       40 (0x28)
  .maxstack  2
  .locals init ([0] class TestClass.TTest Obj)
  IL_0000:  newobj     instance void TestClass.TTest::.ctor()
  IL_0005:  stloc.0
  IL_0006:  ldloc.0
  IL_0007:  ldc.i4.s   10
  IL_0009:  stfld      int32 TestClass.TTest::PublicData
  IL_000e:  ldloc.0
  IL_000f:  ldc.i4.s   20
  IL_0011:  stfld      int32 TestClass.TTest::ProtectedData
  IL_0016:  ldloc.0
  IL_0017:  call       instance string
            TestClass.TTest::GetValue()
  IL_001c:  call       void
            Borland.Vcl.Dialogs.Unit::ShowMessage(string)
  IL_0021:  ldloc.0
  IL_0022:  call       instance void TestClass.TTest::Free()
  IL_0027:  ret
} // end of method TForm1::Button2Click
```

The only limitation is that, according to the rules mentioned in the previous section, the use of the protected hack is limited only within a single assembly. If the unit that defines the base class is contained by a different assembly the code will not work.

Class Data and Statics Class Methods

Another long-awaited feature of the Delphi language relates to the introduction of class data. Delphi has always allowed the declaration of class methods, that is methods not bound to a specific object but to a class as a whole. By the way, not all Delphi programmers know that within class methods you can indeed use the self keyword, but that refers to the class itself, not an instance as it happens with regular methods.

As I mentioned, Delphi 8 introduces class data, that is data shared among all object of the class. It was possible to simulate this construct in past versions of Delphi by using global variables hidden in the implementation section of a unit, but this is far form a neat way of writing the code and could cause troubles in case of derived classes. The introduction of class data adds to Delphi a feature that most other OOP languages share. How do you declare class data? Simply by pre-pending the class var keywords to the declaration, as you do with class methods:

```
type
  TMyData = class
  private
    class var
      CommonCount: Integer;
  public
    class procedure GetCommon: Integer;
```

Actually class var introduces a block of one or more declarations. Beside declaring class data, you can also define class properties and static class procedures. Class static methods have been introduced for compatibility to the CLS, as Delphi code could use class methods to express a very similar concept. As there are relevant implementation differences Borland decided to have two separate features in the language. The differences are that class static methods have no references to their own class (no self referring to the class itself), and cannot be virtual. On the positive side, they can be used to define class properties.

Here is some sample code to highlight the syntax, taken from the ClassStatic example:

```
type
  TBase = class
  private
    class var
      fMyName: string;
  public
    class procedure One;
    class procedure Two; static;

    class function GetMyName: string; static;
    class procedure SetMyName (Value: string); static;
    class property MyName: string
      read GetMyName write SetMyName;
  end;
```

Class properties in .NET are properties mapped to class static methods, but in Delphi you can also map them to class data (with the compiler automatically generating the missing method as it happens with plain properties), like in:

```
    class property MyName: string
      read FMyName write SetMyName;
```

This second declaration is mapped to the following CLS compliant metadata:

```
.property instance string MyName()
{
  .custom instance void [System]
    System.ComponentModel.BrowsableAttribute::.ctor(bool) =
    ( 01 00 00 00 00 )
  .set void ClassStatic.TBase::set_MyName(string)
  .get string ClassStatic.TBase::get_MyName()
} // end of property TBase::MyName
```

Class Constructors

Beside standard constructors used to allocate an object of a class and optionally initialize its data, .NET introduces the idea of a class constructor, a sort of class static method called automatically by the CLR to initialize a class. In fact, the class constructor executes before the class is referenced or used in the program.

A class can have only one class constructor, declared as strict private:

```
type
   TMyTestClass = class
   strict private
     class constructor Create;
```

The effect of this code, seen with ILDAMS, is a special method called .cctor (class constructor, as apposed to plain constructors internally called .ctor), as the following IL declaration shows:

```
.method private specialname rtspecialname static
        void  .cctor() cil managed
```

As mentioned earlier (in the section about units) the class constructors in .NET can be considered as a replacement for Delphi initialization sections, and are generated by the Delphi compiler for the fake unit class when an initialization section is used.

Again, as mentioned, the key difference is that the sequence of call of the various class constructors within a program is non-deterministic, so you cannot rely on a class constructor being called before another one gets executed. If you have similar dependencies you'll have to move them off to initial code of the project itself or some other global place in which you can control the sequence of execution. At times, though, since class constructors are guaranteed to be called before a class is used, you can fine tune your existing code to guarantee that the sequence you are looking for is followed. For example, is an initialization section uses a class from another unit you know that the class constructor of that other unit is executed first.

[TODO: test this last behavior with a demo]

Sealed Classes and Final Methods

In short, sealed classes and classes you cannot further inherit from, while final methods are virtual methods you cannot further override in inherited classes. This is the syntax of a sealed class (taken from the SealedAndFinal example):

```
type
  TDeriv1 = class sealed (TBase)
    procedure A; override;
  end;
```

Trying to inherit form it causes the error: "Cannot extend sealed class TDeriv1". This is the syntax of a final method:

```
type
  TDeriv1 = class (TBase)
    procedure A; override; final;
  end;
```

Inheriting from this class and overriding the A method causes the compiler error: "Cannot override a final method".

Now, again we can ask ourselves what is the rationale behind these related concepts. It seems that the more relevant issue is protection. You might want to disallow other to inherit from your classes in general, and in particular you'll want to disallow others from inheriting from security/cryptography classes and possibly hamper them. However, by looking at Microsoft's class library for .NET (the FCL) it seems there are too many sealed classes and too many virtual methods short-cutted with final.

By looking at the .NET literature you'll see references to the fact that a virtual final class can be make very efficient by the system, as it basically boils down to a (slightly faster) non virtual call. This is the same reason behind the same features offered by the Java language. However, in Java this efficiency consideration makes perfect sense as all methods are virtual by default. In C# or Delphi, instead, you can write plain (non-virtual) methods, which seems a better idea than declaring a virtual method and disabling its key capability.

Nested Types and Nested Constants

Delphi traditionally allows you to declare new classes in the interface section of a unit, allowing other units of the program to references them, or in the implementation section, where they are accessible only from methods of other classes of the same unit. Delphi for .NET adds another possibility, namely the declaration of a class within another class. As any other member of the class, the nested class can have a restricted visibility (say, private or protected).

As an example, consider the following declaration (extracted form the NestedClass demo):

```
type
  TOne = class
    // nested constant
    const foo = 12;
    // nested type
    type TInside = class
      {type TInsideInside = class
      end;}
    public
      procedure InsideHello;
    private
      Msg: string;
    end;
  public
    procedure Hello;
  end;

procedure TOne.Hello;
var
  ins: TInside;
begin
  ins := TInside.Create;
  ins.InsideHello;
  writeln ('constant is ' + IntToStr (foo));
end;

procedure TOne.TInside.InsideHello;
begin
   writeln ('internal call');
end;
```

In the listing above you can notice a few things. First, the nested class can be used directly within the class. Second, the same syntax allows you to define a nested constant, a constant value associated with the class (again usable only internally if private or from the rest of the program if public). Third, the definition of the method of the nested class uses the full name of the class, in this case TOne.TInside.

At first sight it is far from clear how you would benefit from using a nested class in the Delphi language. The concept is commonly used in Java to implement event handlers, and makes sense in C# where you cannot have a class inside a unit. If it is important to have this feature in the language for compatibility with the .NET world, I'm not sure if your code design could benefit from their usage.

As a final note consider that you can declare a field of the nested class right after you've declared the nested class (see the complete code of the NestedClass demo for an example).

Class Helpers

When Borland started working on Delphi for .NET, one the problems that surfaced was the need to somewhat reconcile Delphi own base classes (like TObject, Exception) with the corresponding classes of the .NET Framework. After some research, it came out with a somewhat astonishing trick, called class helpers. A class helper is nothing but a way to pretend that some methods and properties are part of a class you have no power to modify, while in fact they are hosted by a different class. In other words, you can add a special class, the helper, that adds methods to an existing one (methods only, but no data). This way you'll be able to apply the new method to an object of that other class, even if that class has no clue about the existence of the method.

If this is not clear, and it is probably not, let's look at an example (taken from the ClassHelperDemo project):

```
type
  TMyObject = class
  private
    Value: Integer;
    Text: string;
  public
    procedure Increase;
  end;

  TMyObjectHelper = class helper for TMyObject
  public
    procedure Show;
  end;
```

The code above declares a class and a helper for this class. This means that on an object of type TMyObject you can call the method(s) of the class but also each of the methods of the class helper:

```
  Obj := TMyObject.Create;
  Obj.Text := 'foo';
  Obj.Show;
```

The helper method becomes part of the class and can use self as any other method to refer to the current object (of the class it helps, as class helpers are not instantiated), as this code demonstrates:

```
procedure TMyObjectHelper.Show;
begin
  WriteLn (Text + ' ' + IntToStr (Value) + ' -- ' +
    self.ClassType.ClassName + ' -- ' + ToString);
end;
```

Finally notice that a helper method can "override" the original method. In the code I've added a Show method both to the class and to the helper, but only the one of the helper gets called!

Of course, it makes very little sense to declare a class and an extension to the same class using the class helper syntax in the same unit or even in the same program. What can be interesting, instead, is the ability to extend a class defined in an external assembly (and possibly even written in another language). Borland itself uses class helpers heavily in Delphi's RTL to extend standard .NET classes and integrate with .NET RTL support.

For example, Delphi traditional TObject class has a ClassName method. In Delphi for .NET TObject is an alias of System.Object, which would prevent you from calling the ClassName method on your objects. However, by defining a class helper for the TObject class, Delphi makes it possible to you to call ClassName on any .NET objects, even those not originating in Delphi itself. This is the code you can find in System.pas:

```
TObjectHelper = class helper for TObject
public
  procedure Free;
  function ClassType: TClass;
  class function ClassName: string;
  class function ClassNameIs(const Name: string): Boolean;
  class function ClassParent: TClass;
  class function ClassInfo: System.Type;
  class function InheritsFrom(AClass: TClass): Boolean;
  class function MethodAddress
    (const AName: string): TMethodCode;
  class function MethodName(ACode: TMethodCode): string;
  function FieldAddress(const AName: string): TObject;
  procedure Dispatch(var Message);
end;
```

(More details about TObjectHelper implementation in Chapter 4, which covers Delphi RTL.) Given this declaration, not only you can call the helper methods on any object of classes you compile with Delphi, but also to any object written in any language and created by assemblies you hook to your code. This includes containers and contained objects, WinForms and ASP.NET controls and just about any object in the system.

Just in case you are thinking of mimicking this behavior in another language, consider that (as far as I know) Borland has filed for patents on the class helper technology.

According to Delphi's R&D members, the core rule is that class helpers should be used to bind the core classes of the library to new platforms while maintaining compatibility with existing code. Class helpers should not be used as a general language construct for developing applications or components. However, it happens that not only RTL libraries but also the higher level ECO framework uses class helper extensively.

There are a few more rules that apply to class helpers. Class helpers methods can have different access specifiers, class methods, virtual methods (which can be overridden – the compiler adds an interface behind the scenes), extra constructors, class variables, properties, class operators . The only features they lack is instance data.

Properties

Properties in Delphi for .NET maintain the core features whey used to have since Delphi 1. The only relevant difference is in how they are translated to .NET code and how you can use them from other .NET languages. In fact in the CLS properties must be mapped to getter and setter methods and not directly to fields, as in Delphi. For this reason the Delphi for .NET compiler creates the methods for you if your code maps the properties directly to data, but also maps the properties to the proper methods if you use non-complaint property names. This is a sample class with two properties.

The first property (One) was completed automatically by the Delphi editor after writing the declaration property One: string and typing Ctrl-Shift+C. In this case Delphi for .NET uses the set_One method name instead of the traditional Delphi name SetOne. The second property was written manually with a getter and a setter method (again generated by class completion). Here is the class code (from the PropertyDemo example):

```
type
  TMyClass = class
  public
    FOne, FTwo: &string;
    procedure set_One(const Value: &string);
    function GetTwo: string;
    procedure SetTwo(const Value: &string);
  public
    property One: string read FOne write set_One;
  published
    property Two: string read GetTwo write SetTwo;
  end;
```

Notice that Delphi erroneously prefixes the string type name with an & (in the declaration and parameters). Although string is a reserved word it can be legitimately used in these contexts, as it happens to be a type name! I find this behavior of the editor quite annoying as the code is far less readable with the extra &s (although both versions do compile).

If we now compile this code and inspect it with ILDASM we can find out a couple of interesting things. First the property One is mapped to two methods, one we have written in the code and the other generated by the compiler:

```
.property instance string One()
{
  .custom instance void [System]
    System.ComponentModel.BrowsableAttribute::.ctor(bool) =
    ( 01 00 00 00 00 )
  .get instance string PropertyDemo.TMyClass::get_One()
  .set instance void PropertyDemo.TMyClass::set_One(string)
} // end of property TMyClass::One
```

Now let's look at the method synthetized by the compiler, which is almost identical to one generated by hand:

```
.method public hidebysig specialname instance string
        get_One() cil managed
{
  // Code size       9 (0x9)
  .maxstack  1
  .locals init ([0] string Result)
  IL_0000:  ldarg.0
  IL_0001:  ldfld      string PropertyDemo.TMyClass::FOne
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  ret
} // end of method TMyClass::get_One
```

Now let's look a the second property. A first different is that the BrowsableAttribute is set to true as an effect of the published declaration (more on this as we discuss attributes later in this chapter) but what I want to point out is the methods the property is mapped to are not those in the declaration!

```
.property instance string Two()
{
  .custom instance void [System]
    System.ComponentModel.BrowsableAttribute::.ctor(bool) =
    ( 01 00 01 00 00 )
  .get instance string PropertyDemo.TMyClass::get_Two()
  .set instance void PropertyDemo.TMyClass::set_Two(string)
} // end of property TMyClass::Two
```

Again the compiler does a few things behind the scenes. It doesn't change the method names, but it generates another method, get_Two, that calls GetTwo:

```
.method public hidebysig specialname instance string
        get_Two() cil managed
{
  // Code size       9 (0x9)
  .maxstack  1
  .locals init ([0] string Result)
  IL_0000:  ldarg.0
  IL_0001:  call       instance string
                       PropertyDemo.TMyClass::GetTwo()
  IL_0006:  stloc.0
  IL_0007:  ldloc.0
  IL_0008:  ret
} // end of method TMyClass::get_Two
```

Indexers or Array Properties

Since its first release, Delphi has always supported the idea of array properties that receive as parameter a value passed among square brackets. And we've always enjoyed the possibility of marking one of the array properties of a class as default, so that it can be referenced applying the square brackets right to the object, omitting the property name. So for example...

The C# language has a very similar idea, called indexer, with a significant difference: you can have multiple indexers (that is default array properties) for a single class, based on a different types of index. To match this feature, Delphi for .NET adds support for overloading the default array property. Notice that you cannot really have multiple array properties (it you try, only the last default array property will be considered) but you can have multiple definitions of a single property (as in this snippet from the PropertyDemo example):

```
type
 TMySecondClass = class
  private
    strList: TStringList;
  public
    constructor Create;
    function get_One(I: Integer): string; overload;
    function get_One(Id: string): string; overload;
    procedure set_One(I: Integer;
      const Value: string); overload;
    procedure set_One(Id: string;
      const Value: string); overload;
  public
    property One [I: Integer]: string
      read get_One write set_One; default;
    property One [Id: string]: string
      read get_One write set_One; default;
  end;
```

This means you can use both versions of the One default property simply by specifying different types of parameters among square brackets:

```
  sc := TMySecondClass.Create;
  sc ['text'] := 'hello';
  writeln (sc [0]);
```

Applying Constructors to Instances

In past versions of Delphi, constructors could be used in two different scenarios: (i) you could use them in the traditional object creation mode, by applying them to a class type, or (ii) you could use them as initializer mode, by applying them to existing objects:

```
anObject := aClass.Create; // (i) object creation
anObject.Create; // (ii) object (re)initialization
```

In Delphi 8 for .NET you cannot apply a constructor to an instance any more. The reason is that the underlying execution environment doesn't support this feature. The effect of this change is quite positive, though. Applying a constructor to an instance is a classic error in Delphi, an error for which the compiler might at most issue a warning, because there is a legitimate use of the syntax. Now with that use being illegal, the compiler will issue an error, and will almost always be correct! In fact, 99% of the times you apply a constructor to an instance you are typing the wrong code (this percentage is true for myself at least, and I keep mistyping constructor calls after so many years of Delphi coding).

Calling Inherited Constructors

Delphi 8 for .NET marks a clear departure from the traditional Delphi implementation of constructors. Delphi used to be one of the few OOP programming languages that didn't require to initialize base classes in the constructor of an inherited class, opening up the possibility for errors and odd behaviors. Now Delphi 8 enforces this rule. In the constructor of an inherited class you must call the base class constructor and you must call it before touching any field or calling any method of the base class.

For example, the following trivial code doesn't compile any more:

```
 type
  TMyClass = class
  private
    fValue: Integer;
  public
    constructor Create;
  end;

constructor TMyClass.Create;
begin
  fValue := 10;
end;
```

Instead, is gives the error message: "'Self' is uninitialized. An inherited constructor must be called". In fact, the TMyClass class inherits from TObject, so it is bound to the same rule of any other class. This is quite unfortunate, as the TObject.Create constructor is basically useless...

```
constructor TMyClass.Create;
begin
  fValue := 10;
  inherited Create;
end;
```

Considering that there are restrictions on what you can do before initializing the base class, the preferred way of coding the constructor above is certainly:

```
constructor TMyClass.Create;
begin
  inherited Create;
  fValue := 10;
end;
```

[TODO: cover more complex cases]

Free and Destroy in the Garbage Collected World

One of the features of the .NET platform (like the Java platform) that gets more enthusiastic supporter and angry opponents is garbage collection. Here I don't want to get into this discussion, but only address a specific issue related to the management of external resources. In fact, although the garbage collector will automatically reclaim unused memory, no one will magically free external resources (like files, database connections, bitmaps and GDI resources...) unless you write and execute specific code.

Now the problem is that with the garbage collector in action you cannot simply put this code in the class destructor, as the object can be destroyed a lot of time after it is not referenced any more, keeping resources locked, or might even not be called at all!

To address this issue .NET has a specific patterns you must implement: any class needing resource clean-up should implement the Dispose method of the IDisposable interface. So you'll have to define the method in your classes, if needed, and than also call the method before all of the references to the object go out of scope. Only in some specific cases, for example when the object is placed in a container, a library class might call Dispose for you.

In Delphi for .NET, all objects implicitly implement IDisposable and redirect calls to Dispose to the Destroy destructor. That is, the destructor code doesn't compile to a real destructor (very seldom needed) but to Dispose. This means that most of the times you existing Delphi code will still do.

If you write a *destructor Destroy* and forget the override keyword you'll get an odd error message saying *"Unsupported language feature: 'destructor'"*. Of course this feature is supported but you must override the *TObject* destructor.

So if you write this code:

```
type
  TMyClass = class
  public
    destructor Destroy; override;
```

You'll end up with the following IL, indicating the the class implements System.IDisposable:

```
.class public auto ansi beforefieldinit TMyClass
  extends [mscorlib]System.Object
  implements [mscorlib]System.IDisposable
```

In turn the Destroy method becomes an implementation to Dispose:

```
.method public newslot virtual instance void
  Destroy() cil managed
{
  .override [mscorlib]System.IDisposable::Dispose
```

If you look with more care into the code of the destructor, you'll notice that upon termination it sets a Disposed_ boolean field added to the class by the compiler. This same field is checked upon entering the destructor to avoid re-executing it, thus blocking a double destroy operation.

If you still use Destroy to define resource deallocation, you never call Destroy (as in Delphi 1 to 7) but call Free to invoke it. In Delphi for .NET the call to Free has been redirected to a call to the Dispose method of the IDisposable interface:

```
procedure TObjectHelper.Free;
begin
  if (Self <> nil) and (Self is IDisposable) then
  begin
    if Assigned(VCLFreeNotify) then
      VCLFreeNotify(Self);
    (Self as IDisposable).Dispose;
  end;
end;
```

Consider that thanks to class helpers you can indeed call free also on FCL objects or other objects written in C#.

Again, this means that to properly dispose external resources (and since you generally don't know how on object is implemented, better doing this anyway) you can keep using try-finally blocks like:

```
var
  MyObj: TMyObj;
begin
  MyObj := TMyObj.Create;
  try
    // use MyObj
  finally
    MyObj.Free;
  end;
```

To sum things up, you keep writing destructors (unless all you have to free is memory) and you keep calling Free on objects. Your Delphi code remains the same, but the behavior changes considerably. You need to know this, but than keep your habits. The garbage collector will help you removing unreferenced objects (but you should not rely on this because of the resource issue) and won't really get into your way (with extra stuff like Dispose to remember). In other words, keep writing your code as you are used to and stop reading discussions on the garbage collector benefits and drawbacks...

Class References and MetaClasses

Class references are a specific feature of the Delphi language, so it shouldn't surprise you that the .NET framework doesn't have the same concept. As usual, Delphi 8 retains the syntax for class references and most of their behavior stays the same as well (including the call of virtual constructors on them and their support for virtual class methods). However behind the scenes, the implementation changes considerably.

In Delphi 8 for .NET, for each class (let's call one TMyClass) the compiler creates both a class and a metaclass (called by default something like @MetaTMyClass) inherited from the generic TClass metaclass. The compiler defines also a constant static instance of the metaclass.

Delphi class references (or metacalsses) and not CLS compliant: they are not intended for use by other .NET languages. The same holds for virtual constructors.

However the Delphi 8 compiler cannot impose the presence of a specific metaclass for classes not compiled by Delphi but imported from assemblies written in other languages. In this case the compiler creates an instance of a generic TClass metaclass, passing the CLR type to the constructor. In this way Delphi 8 can simulate Delphi metaclass behaviors for any .NET class, although the code is not as efficient as with the class-specific metaclasses generated by the compiler.

By the way, notice that internally TClass uses an instance of the type System.RuntimeTypeHandle rather than the more obvious (but less memory efficient) System.Type.

[TODO: add an example and inspect the IL code]

Interfaces are now "Pure"

At the time of its introduction in the early versions of Delphi, the interface type was considered by most programmers strictly as a COM-related technique. This was also due to some implementation decisions withing Delphi's RTL. Along the versions of Delphi the relationship between the concept of interface and COM has been slowly but increasingly reduced, with the introduction of the IInterface base interface (along with IUnknown), with the availability of helper classes and routines in general RTL units (instead of the COM/ActiveX units) and so on.

For sure I'm a big fan of the use of interfaces ad a sounds OOP technology. My Mastering Delphi 7 books shows some ways to benefit from interfaces, but nothing like my Design Patterns papers (not publicly available, sorry) shows the power of this technique.

In Delphi 8 COM is basically gone, so it should some to no surprise that some relevant implementation details of interfaces have changes as well. First of all, the definition of IInterface is still available but quite different, as it now has an empty definition (in Borland.Delphi.System):

```
type
  IInterface = interface
  end;
```

This means that the reference counting for interfaces is gone, something that should come at no surprise with a runtime that uses garbage collection, but also the the type checking is not based on QueryInterface any more but on specific compiler/runtime features. The side-effect of this change is that you don't need to decorate interfaces with GUIDs any more for the type checking to work properly as it happens in Delphi 7 (and previous versions).

Correspondingly the TInterfacedObject class has an empty implementation:

```
type
  TInterfacedObject = TObject;
```

These changes imply that now interfaces are 100% a language feature, with no connection whatsoever to COM or anything else. The .NET runtime fully supports interfaces (and the runtime and the FCL uses them quite extensively) so Delphi for .NET embraces this appraoch.

Does it mean that interfaces work better in Delphi 8 than they used to do in Delphi 7? For sure, having garbage collection on interfaces object is very handy, as it is far form trivial to free objects in Delphi 7 when you access them exclusively via interfaces.

However, there are also some interesting features of interfaces in Delphi 7 that didn't make over to .NET. In particular, dynamic aggregation of interfaces (that is the use of the implements keyword for interfaces) is not supported in Delphi for .NET. This is quite bad, as dynamic aggregation allows you to share a common implementation of the interface methods between separate classes implementing the same interface.

[TODO: cover interface methods resolution]

As a sample of the use of interfaces in Delphi 8 you can look at the InterfaceTest demo. Its secondary unit has an interface with a few methods and a property. When declaring the property I found out that for .NET compatibility you should follow very precise rules, even more than on regular properties (it seems, but I'm not 100% sure). This is how Delphi 8 likes it:

```
type
  ISimple = interface (IInterface)
    procedure ShowMessage;
    function Compute (a, b: Integer): Integer;
    function get_Value: Integer;
    procedure set_Value (Value: Integer);
    property Value: Integer read get_Value write set_Value;
  end;
```

Notice the lowercase names for the get and set methods and the underscore before the property name. If you don't follow this convention you'll get hints like:

```
[Hint] Property accessor GetValue should be get_Value
[Hint] Case of property accessor method ISimple.Get_Value
should be ISimple.get_Value
```

These hints, which can be suppressed, help you write CLS-compatible code. If you fail do do so other .NET languages won't be able to use the property. Some of them (like VB and C#) will still allow you to use the property or call the getter and setter accessor methods directly, but some . NET languages might not allows any access at all.

By looking at the interfaces used by .NET, however, seems that they avoid the problem altogether by *not* using properties inside interfaces, but only custom set and get methods written in many inconsistent ways...

Notice that the behavior of the Delphi compiler with properties inside interfaces (you'll get the a compiler hint) is quite different from properties inside classes. In this case, as discussed earlier in the section "Properties", the compiler adds to your code the CLS-compliant methods generating code for them. This ensures CLS compatibility, at the cost of extra code (and generally also slower code). With interface the automatic generation would have made no sense, as you'd had to implement multiple versions of the same method.

As an experiment I've added a GUID (or to be more precise an IID, an interface ID) to the interface as follows:

```
type
  ISimple = interface (IInterface)
    ['{6F1B5589-3987-4665-9C4F-630287760BE9}']
```

This is apparently ignored by the compiler. I was expecting to see the class decorated with a Guid attribute, but if you want to obtain one you'll need to write code like in C#:

```
type
  [Guid('6F1B5589-3987-4665-9C4F-630287760BE9')]
  ISimple = interface (IInterface)
```

This effectively adds the attribute in the code, which is what you might have to do for COM interoperability in .NET (notice, in fact, you'll need to use the System.Runtime.InteropServices namespace for the Guid attribute to work).

Operators Gain New Ground

Another brand new addition to the Delphi language is the concept of operators overloading, that is the ability to define you own implementation for doing standard operations (sum, multiply, compare...) on your data types. For example, you can implement an add operator (a special Add method) and than use the + sign to call it.

To define an operator you use the directive class operator (with a directive Borland managed to have no impact on existing code, while adding a new reserved word could have caused troubles). The term class here relates to class methods, as operators like class methods have no self parameter, no current object. After the directive you write the operator's name, like Add:

```
type
  TPointRecord = record
  public
    class operator Add (a, b: TPointRecord): TPointRecord;
```

The operator Add is than called with the + symbol, as you'd expect. So which are the available operators? Basically the entire set of operators of the language, as you cannot define brand new language operators:

- **Cast Operators**: Implicit, Explicit
- **Unary Operators**: Positive, Negative, Inc, Dec, LogicalNot, BitwiseNot, Trunc, Round
- **Comparison Operators**: Equal, NotEqual, GreaterThan, GraterThanOrEqual, LessThan, LessThenOrEqual
- **Binary Operators**: Add, Subtract, Multiply, Divide, IntDivide, Modulus, ShiftLeft, ShiftRight, LogicalAnd, LogicalOr, LogicalXor, BitwiseAnd, BitwiseOr, BitwiseXor

In the code calling the operator, you do not use these names but the corresponding symbol. This way an existing field of method within your code having a similar name (say your record has an Add method or an Inc field) won't conflict.

When you define these operators you spell out the parameters, and the operator will be applied only if the parameters match exactly. To add two values of different types you'll probably have to specify two Add operations, as each operand could be the first or second entry of the expression. In fact, the definition of operators provides no automatic commutativity. Moreover, you have to indicate the type very precisely, as automatic type conversions don't apply. Many times this implies overloading the overloaded operator providing multiple versions with different types of parameters.

There are two further special operators you can define, Implicit and Explicit. The first is used to define an implicit type cast (or silent conversions), which should be perfect and not lossy. The second, Explicit, can be invoked only with an explicit type cast from the record structure to a given type. Together these two operators define the casts that are allowed to and from the given data type. Notice that both the Implicit and the Explicit operators can be overloaded based on the function return type, which is generally not possible for overloaded methods. In case of a type cast, in fact, the compiler knows the expected resulting type, and can figure out which is the typecast operation to apply.

As an example, the OperatorsOver demo includes both a record with a few operators and a class with similar ones:

```
type
  TPointRecord = record
  private
    x, y: Integer;
  public
    procedure SetValue (x1, y1: Integer);
    class operator Add (a, b: TPointRecord): TPointRecord;
    class operator Explicit (a: TPointRecord): string;
    class operator Implicit (x1: Integer): TPointRecord;
  end;

 type
  TPointClass = class
  private
    x, y: Integer;
  public
    procedure SetValue (x1, y1: Integer);
    class operator Add (a, b: TPointClass): TPointClass;
    class operator Explicit (a: TPointClass): string;
  end;
```

Here is the trivial implementation of the methods of the record:

```
class operator TPointRecord.Add(a, b:
  TPointRecord): TPointRecord;
begin
  Result.x := a.x + b.x;
  Result.y := a.y + b.y;
end;

class operator TPointRecord.Explicit(a: TPointRecord): string;
begin
```

```
  Result := Format('(%d:%d)', [a.x, a.y]);
end;

class operator TPointRecord.Implicit(x1: Integer):
  TPointRecord;
begin
  Result.x := x1;
  Result.y := 0;
end;
```

Using such a record is quite straightforward, as you can write code like this (remember that record variables don't need an explicit allocation):

```
procedure TForm1.Button1Click(Sender: TObject);
var
  a, b, c: TPointRecord;
begin
  a.SetValue(10, 10);
  b := 30;
  c := a + b;
  ShowMessage (string(c));
end;
```

The second assignment (b) is done using the implicit operators, in fact there is no cast, while the ShowMessage call uses the cast notation to activate an explicit type conversion. Consider also that the operator Add doesn't modify its parameters, rather it returns a brand new value. This is a general rule of operators overloading in Delphi for .NET, which applies also to classes. In this second case, however, you'll have to create – allocate – a new object:

```
class operator TPointClass.Add(a, b: TPointClass): TPointClass;
begin
  Result := TPointClass.Create;
  Result.x := a.x + b.x;
  Result.y := a.y + b.y;
end;
```

Danny Thorpe (in a presentation I attended) suggested that "while it is valid syntax to define operators on class types, it seems significant that there is not one class in the entire .NET framework that implements operators. Stick with records until we find out why."

Delphi's RTL has been rewritten to take advantage of records with methods and operators. You'll see examples of operators overloading in the Currency type and DateTime type (in Borland.Delphi.System, that is the good old System.pas) and in the complex numbers implementation you can find in the Borland.Vcl.Complex unit.

Operators Resolution Rules

The rules related to the resolution of calls involving operators are different than the traditional rules involving methods, as particularly with automatic type conversion there are chances for a single expression to end up calling different versions of an overloaded operator and to cause ambiguous calls.

[TODO: More details might follow]

Attributes, or RTTI to the Extreme

The concept of attributes represent probably the single most relevant innovation of the .NET runtime and the C# language (being also one of the few ideas not coming from Java).

Attributes in .NET represent RTTI to the extreme. In fact, like in Delphi you can declare a property as published to be able to access it at runtime, using RTTI techniques, in .NET you can decorate properties, methods, classes and any other entity with attributes you can later query for at runtime. The huge differences between the two approaches is that while a concept like published is rooted into the system, attributes are totally open: you can define the attributes you like, even with parameters, and decorate symbols even with multiple attributes.

Technically is Delphi for .NET (as in other .NET languages) attributes are listed within square brackets, like this (where the attribute is applied to a class):

```
type
  [anAttribute]
  TFoo = class
    ...
  end;
```

In case an attribute has parameters the code becomes like (in this case the attribute is applied to a method):

```
type
  TFoo = class
    [anotherAttribute (22)]
    procedure Test;
  end;
```

Declaring Custom Attributes

As I mentioned, you can define a new type of attribute, that is a new attributes class. This has to be a class inheriting from TCustomAttribute (which in turn is an alias of System.Attribute). The following is a simple code snippet:

```
type
  TMyCustomAttribute = class(TCustomAttribute)
  private
    FAttrValue : Integer;
  public
    constructor Create(AttrValue: Integer);
    property CustomValue : Integer
      read FAttrValue write FAttrValue;
  end;

  constructor TMyCustomAttribute.Create(AttrValue: Integer);
  begin
    inherited Create;
    CustomValue := AttrValue;
  end;
```

This is how you can use this attribute to mark a class and a method. Notice that you can use the short form (the class name without the final "Attribute") or the complete form:

```
type
  [TMyCustom(17)]
  TFoo = class
  public
    [TMyCustomAttribute(22)]
    Data : Integer;
  end;
```

The instance data and the class definition are now marked with the attribute:

```
.class public auto ansi beforefieldinit TFoo
  extends [mscorlib]System.Object
{
  .custom instance void
     NetAttributes.TMyCustomAttribute::.ctor(int32) = (...)
  ...
  .field public int32 Data
  .custom instance void
     NetAttributes.TMyCustomAttribute::.ctor(int32) = (...)
```

Unit attributes are achieved by placing the attribute immediately before the begin..end block of the unit or by using the *[unit:]* attribute scope modifier. Another “global” scope modifier is [assembly:], which is placed anywhere in the code (but usually in the project source file) and applies to the compiled assembly.

Inspecting Attributes with Reflection

By itself adding attributes to declarations, as we've done earlier, is completely useless. It becomes interesting as soon as there is some other code you have written or part of the .NET libraries that looks after those specific attributes and behaves accordingly.

This means that other code typically acts only on classes or methods marked with a given attribute, eventually considering the attribute parameters. In the NetAttributes example this is accomplished by two routines, ShowCustomAttributes and ShowAttribs, shown below. The first routine receives as parameter a type, outputs the type name and than extracts from the type the list of attributes of type TMyCustomAttribute (or a compatible derived class). This list is passed to the second routine, which displays the attribute name (only of the first attribute of the list) and grabs the value of its parameter. As we've asked for TMyCustomAttribute attributes the cast to this type is indeed correct. Back to the ShowCustomAttributes routine, it repeats the process of displaying the type name, the member type, and the eventual attribute for each of the type members, methods and data.

```
procedure ShowCustomAttributes (aType: System.Type);
var
  members : array of System.Reflection.MemberInfo;
  I: Integer;
  mtypes: System.Reflection.MemberTypes;
begin
  write (aType.Name);
  ShowAttribs (aType.GetCustomAttributes
    (TMyCustomAttribute.ClassInfo, True));
  writeln;

  members := aType.GetMembers;
  for I := 0 to High(members) do
  begin
    mtypes := members[i].MemberType;
    write (aType.Name + ':' + members[i].Name +
      ' (' + TObject(mtypes).ToString + ')');
    ShowAttribs (members[i].GetCustomAttributes
      (TMyCustomAttribute.ClassInfo, True));
    writeln; // new line
  end;
end;

procedure ShowAttribs (attribs: array of System.Object);
begin
  // show only the first one...
  if Length (attribs) > 0 then
  begin
    write (' ---> ' + attribs[0].ToString);
    write ('(' + IntToStr ((attribs[0] as TMyCustomAttribute).
      CustomValue) + ')' );
  end;
end;
```

The effect of this code is that a call in the main module over an object of the TFoo type , like:

```
ShowCustomAttributes (Foo.GetType);
```

(or over the TFoo type itself) produces an output like:

```
TFoo ---> CustomAttribute.TMyCustomAttribute(17)
TFoo:Data (Field) ---> CustomAttribute.TMyCustomAttribute(22)
TFoo:GetHashCode (Method)
TFoo:Equals (Method)
TFoo:ToString (Method)
TFoo:Free (Method)
(more output omitted)
```

Relevant Predefined Attributes

[TODO: List of interesting predefined attributes in .NET. Here or in a later chapter.]

Events for Everybody

In its first incarnation, Delphi introduced the idea of events as most development tool use it nowadays. An event in Delphi provides a way to hook an external method to an object, thus modifying the object's behavior through delegation (instead of customizing its class through inheritance). For example, the code a button executes when it is clicked in not written in the button class, but the button delegates to a method of another object, usually the form hosting the button.

Technically, in Delphi an event is a property with a method pointer type, that is a reference to a method of an object. Java took a different approach, but .NET uses an architecture similar to the Delphi one, with only a relevant extension: an event can have multiple handlers attached to it. The term used to indicate this behavior is generally multicast events.

Delphi for .NET actually supports both traditional unicast and the new multicast events, depending on the components you are working with. The classic event semantics is still supported through := assignments; the new multicast semantics uses the Include() and Exclude() standard procedures, overloaded, to operate on events (these functions were used in the past to operate on sets). As a comparison, C# uses the += and -= operators of the C language.

```
Include (Button1.Click, Button1Click)
```

In general you'll stick to the traditional approach when working with VCL.NET, while the multicast technique is necessary when integrating with .NET native framework. To better support interoperability, though, the standard Delphi read/write events support now also the add/remove semantic of .NET for compatibility with CLR (for example to let C# code use traditional Delphi objects), although the actual behavior will be a single assignment.

[TODO: Sample code missing, defer it to WinForms chapter?]

Back to Windows

[TODO: this section still missing]

DllInvoke

Example

Speed considerations

Inverse P/Invoke (Unmanaged Exports)

Introducing unmanaged exports (other related topics to come later), as this is a language extension

"functions in a managed Delphi for .NET code assembly can be called directly by unmanaged Win32 code with no COM interop or .NET awareness."

[add Demo code I have]

Three simple steps:

```
library

{$UNSAFECODE ON}

exports function_name;
```

Notes on the dccil Compiler

[TODO: section still missing]

Documentation: Compiler switch -DOC to produce XML documentation

Conditional defines of dccil compiler

```
{$IFDEF CLR}
{$IFDEF CIL}
{$IFDEF MANAGEDCODE}
```

Update 1

The Update 1 for Delphi 8 for .NET released in February 2004 provides among (few) other things fixes to the compiler (see http://bdn.borland.com/article/0,1410,31971,00.html).

The compiler fixes relate to the Variant to AnsiString conversion, help dealing with a few code browsing errors, cause an error when compiling a package twice, and (in particular) fix a code generation bug that will consider signed some unsigned values when they are promoted to Int64, thus altering their content. As the readme file suggests:

> A large positive 32 bit value such as 2952790015 ($AFFFFFFF) would turn into -1342177281 when assigned to an Int64 variable or parameter.

This Update 1 change causes a few troubles when rebuilding the VCL, as the library code must be updated for some of the compiler changes. You have two solutions: (i) avoid rebuilding it, or (ii) download the updated VCL source from Code Central (id=21403). The BDN article http://bdn.borland.com/article/0,1410,31968,00.html explains the details.

Delphi 8 for .NET Update 2

[TODO: Section still missing, don't know if it is really needed]

Summary

At end of this chapter devoted to the Delphi language in .NET there are two relevant ideas I discussed at length and want to underline as a conclusion of this chapter.

Delphi Language Extended as Never Before

First, Delphi 8 for .NET is a relevant milestone for the Delphi language, with fixes and improvements on long standing issues (private that really works, inherited constructors that initialize base classes, class data) and a number of relevant new features (records with methods, operators overloading, attributes). Waiting for at least some of these features in a future Win32 version, we Delphi programmers will take some time to adjust our coding style to a heavily updated and revived language (based on a rock solid language foundation).

Delphi Language Above and Beyond the CLR

The other relevant thought is that the Delphi language was indeed extended to match features in the .NET CLR and for CLS-compliance, but it is nice to notice that Delphi has CLS-compliant features that other .NET languages lack (like class helpers and Inverse P/Invoke) and also a number of non CLS-compliant features the language always has (like class references and virtual constructors, named constructors, virtual calls from class methods, a safe use of unsafe types, short strings, and many more).