CAPITOLO 25

Anteprima di Delphi per .NET: il linguaggio e la RTL

Nel capitolo precedente abbiamo introdotto l'architettura .NET di Microsoft. È ora giunto il momento di focalizzare l'attenzione sull'anteprima di Delphi per .NET fornita con Delphi 7. In questo capitolo illustrerò le modifiche apportate al linguaggio Delphi per renderlo compatibile con il Common Language Runtime. Ci sono state alcune aggiunte importanti al linguaggio, come i namespace e i nuovi specificatori di visibilità. Altre funzionalità classiche del linguaggio sono state eliminate, poiché non supportate nell'ambiente type safe di .NET.

In questo capitolo illustrerò diverse aree del nuovo compilatore e l'utilizzo di librerie Microsoft specifiche per il supporto a .NET e ASP.

Tenete presente che il compilatore Delphi per .NET, nel momento in cui scrivo, è in lavorazione. Il progetto originario di Borland prevedeva di rilasciare un'anteprima con Delphi 7 e far poi uscire aggiornamenti, per gli utenti registrati, nel corso del 2002 e nei primi mesi del 2003 (fino al completamento del progetto Galileo). Alcune delle funzionalità di cui vi parlerò in questo capitolo potrebbero risultare non disponibili o avere diversi livelli di completamento, a seconda dell'aggiornamento utilizzato. Al termine del capitolo, elencherò alcuni siti Web che potete visitare per le ultime notizie sul compilatore Delphi per .NET.

Come per il Capitolo 24, buona parte del materiale di questo capitolo proviene da John Bushakra.

In questo capitolo saranno trattati gli argomenti seguenti:

- ✔ modifiche al linguaggio Delphi;
- ✔ funzionalità deprecate e nuove;
- ✔ la libreria run time e la VCL;
- ✔ utilizzo delle librerie Microsoft;
- ✔ ASP.NET in linguaggio Delphi.

Funzionalità deprecate del linguaggio Delphi

Cominciamo con le funzionalità del linguaggio Delphi che è stato necessario dichiarare deprecate per ottenere la compatibilità con il CRL (*Common Language Runtime*). In seguito illustreremo le caratteristiche aggiunte (o programmate) in Delphi per .NET, che trasformeranno il linguaggio Delphi nel prossimo futuro, probabilmente anche per le piattaforme Win32 e Linux.

Tipi deprecati

Alcuni tipi di Delphi non sopravviveranno alla transizione verso l'ambiente gestito e virtuale di .NET. Al momento, i tipi elencati di seguito sono considerati deprecati, o comunque dal futuro incerto.

- ✔ **Puntatori:** i puntatori non sono considerati tipi sicuri dal CLR ed è vietata ogni forma di aritmetica dei puntatori. Il termine "non sicuro" indica che non è possibile verificare la sicurezza del codice. La versione finale di Delphi per .NET potrebbe supportare i puntatori non sicuri e non gestiti, ma in attesa si possono utilizzare gli array dinamici per recuperare parte delle funzionalità di `GetMem`, `FreeMem` e `ReallocMem` (anch'esse deprecate).
- ✔ **Tipi basati sul costrutto file of <tipo>:** i tipi basati sul tradizionale costrutto Pascal `file of <tipo>` non sono supportati, poiché il compilatore non ha modo di determinare la dimensione di un tipo dato nella piattaforma di destinazione.
- ✔ **Sintassi object precedente a Delphi:** la sintassi `object`, precedente a Delphi, non sarà supportata nella versione finale del compilatore. Questa sintassi venne introdotta ai tempi del Turbo Pascal e permetteva di dichiarare una nuova classe tramite la sintassi `type MyClass = object;`. Le variabili di questo tipo erano allocate sullo stack, a differenza degli oggetti di Delphi che sono allocati sull'heap.
- ✔ **Real48 e Comp:** questi tipi non saranno supportati. Real48 rappresenta un numero in virgola mobile a 6 byte, mentre Comp sarà sostituito da Int64 in futuro, come riporta la documentazione di Delphi 7.

Stringhe e altri tipi

I tipi seguenti, anche se non deprecati, saranno modificati nell'implementazione interna. Le modifiche saranno per lo più trasparenti, ma è lecito aspettarsi comportamenti leggermente diversi in alcune circostanze.

- ✔ **Stringhe:** in Delphi per .NET le stringhe corrispondono al tipo CLR `System.String` e per impostazione predefinita sono dimensionate come wide. Questo significa che le stringhe impiegano 16 bit per carattere, come il tipo WideString in Delphi 7. Allo stesso modo, tutti i caratteri sono wide di default.
- ✔ **Record:** i record corrispondono a tipi gestiti per valore. Nel Capitolo 24, abbiamo parlato delle due categorie di tipi del CTS (*Common Type System*): reference e value. Un record diventa, in .NET, un tipo value. In base alle specifiche del CLR, i tipi value non supportano l'ereditarietà, ma possono disporre di metodi (il che è una novità per il linguaggio Delphi). I

metodi definiti sui tipi value devono essere dichiarati come finali (parleremo della nuova parola chiave `final` nel paragrafo "Nuove funzionalità del linguaggio Delphi").

- ✔ **TDateTime:** in Delphi, questo tipo è implementato come il DATE Microsoft (per una spiegazione dettagliata si veda il Capitolo 2). La piattaforma .NET si serve di un'implementazione diversa. La struttura `System.DateTime` (derivata da `System.ValueType`) memorizza date e ore comprese tra la mezzanotte del 1 Gennaio 0001 C.E. (*Common Era*) e le 23.59.59 del 31 Dicembre 9999 C.E., con una precisione di 100 nanosecondi. Quando Delphi per .NET passerà al nuovo standard per la memorizzazione di date e ore, i calcoli su valori di questo tipo che dipendono dalla loro rappresentazione attuale in virgola mobile potranno provocare problemi. In particolare, se utilizzate le funzioni `Trunc` e `Frac` per estrarre la data e l'ora da un valore data/ora, in futuro potreste imbattervi in bug insidiosi.
- ✔ **Currency:** il tipo Currency verrà fatto corrispondere al tipo CLR `System.Decimal`.

Altre funzionalità deprecate

Come i tipi elencati nel paragrafo precedente, alcune funzionalità che sono state per lungo tempo parte del linguaggio Delphi non possono essere supportate in ambiente .NET.

- ✔ **Record varianti:** i record varianti, con campi sovrapposti, non sono supportati dal CLR. Generalmente, non è possibile formulare alcuna ipotesi sulla disposizione dei campi nella dichiarazione di un record, poiché il compilatore JIT (*Just In Time*) ha la possibilità di scegliere come ottimizzare la disposizione in funzione della piattaforma fisica sottostante.
- ✔ **ExitProc:** gli eventi non si verificano sempre quando vorremmo, o nell'ordine in cui li vorremmo. Problemi di questo genere relativi alle sezioni initialization e finalization delle unit sono stati superati (sussistono alcune differenze a cui prestare attenzione, come vedremo in seguito), ma `ExitProc` non è supportata.
- ✔ **Aggregazione dinamica delle interfacce:** il CLR non supporta l'aggregazione dinamica delle interfacce tramite la parola chiave `implements`, poiché in questo caso non è possibile la verifica di sicurezza dei tipi. Una classe deve dunque dichiarare tutte le interfacce che implementa.
- ✔ **Istruzioni assembly:** le istruzioni ASM e i blocchi di codice assembly non sono supportati dall'anteprima del compilatore di Delphi per .NET. Il futuro di `asm` nella versione finale è in dubbio. Il compilatore dovrebbe essere in grado di miscelare codice intermedio e codice nativo della CPU in uno stesso assembly, come è in grado di fare Microsoft Visual C++ .NET.
- ✔ **Parola chiave `automated`:** La parola chiave `automated` fu introdotta per supportare OLE Automation e non è necessaria in ambiente .NET. Lo stesso vale per la parola chiave `dispid`, utilizzata per chiamare i metodi COM per numero anziché per nome. Notate che, sebbene non siano più esplicitamente necessari, i GUID sono supportati e vengono memorizzati come attributi personalizzati di un tipo.
- ✔ **Funzioni di accesso diretto alla memoria:** le funzioni per l'accesso diretto alla memoria, come `BlockRead`, `BlockWrite`, `GetMem`, `FreeMem` e `ReallocMem`, così come `Absolute` e

`Addr`, si servono di puntatori non sicuri e quindi non possono essere utilizzate nel codice gestito e sicuro. L'operatore `@` è disponibile nella versione attuale del compilatore (sarà quasi certamente rimosso nella versione finale), ma non è consentita la conversione di tipo dei puntatori né alcuna forma di aritmetica su di essi.

Come abbiamo visto nel Capitolo 2, Delphi 7 è dotato di un nuovo insieme di messaggi di avviso (warning) del compilatore per fornire supporto nella conversione del codice, i quali evidenziano caratteristiche e costrutti del linguaggio non supportati in ambiente .NET e, di conseguenza, sono da evitare. I warning sono disattivati di default in un progetto Delphi 7 nuovo, ma sono attivi se si ricompila un progetto preesistente. Si possono anche attivare esplicitamente tramite la direttiva `{$WARN UNSAFE_CODE_ON}` *e altre simili. Per ulteriori dettagli, consultate il Capitolo 2.*

Nuove funzionalità del linguaggio Delphi

La prima versione del compilatore `dccil` era dotata di nuove caratteristiche richieste dal CLR e altre ne sono state aggiunte con gli aggiornamenti successivi.

Namespace delle unit

I namespace (spazi dei nomi) svolgono un ruolo importante in .NET, poiché consentono l'estensione della gerarchia di classi da parte di soggetti diversi, impedendo conflitti nei nomi utilizzati. Windows e COM si servono di un GUID di 16 byte per identificare in modo univoco i componenti: questo numero "magico" deve essere memorizzato nel Registry. In .NET il concetto di namespace, oltre ai metadati e alle regole rigide per la ricerca degli assembly da caricare, rende i GUID deprecati.

Ironicamente, il concetto di unit Delphi è simile a quello di namespace; la differenza non è notevole, se si pensa a una unit come a un contenitore di simboli e ad un namespace come a un contenitore di unit. In Delphi per .NET il namespace a cui una unit appartiene è dichiarato nella clausola `unit`:

```
unit NamespaceA.NamespaceB.UnitA;
```

I punti indicano che un namespace ne contiene un altro, il quale contiene la unit. I punti suddividono la dichiarazione in componenti e ciascun componente (tranne l'ultimo) è un namespace. L'intera dichiarazione, punti compresi, è il nome della unit. I punti svolgono unicamente la funzione di separatori: la dichiarazione non introduce alcun nuovo simbolo. In questo esempio, `NamespaceA.NamespaceB` è il namespace e `NamespaceA.NamespaceB.UnitA` è il nome della unit. Il file sorgente sarà chiamato `NamespaceA.NamespaceB.UnitA.pas` e la sua compilazione creerà un file di output chiamato `NamespaceA.NamespaceB.UnitA.dcuil`.

L'istruzione `program` (ed eventualmente `package` e `library`) può dichiarare un namespace predefinito per l'intero progetto; in mancanza di un namespace predefinito, il progetto è un *progetto generico* e il namespace è indicato dall'opzione `-ns` del compilatore. Se non si specifica un namespace nemmeno tramite le opzioni del compilatore, i namespace non vengono utilizzati (comportamento compatibile con Delphi 7 e versioni precedenti).

L'istruzione `uses` non deve necessariamente contenere l'indicazione esplicita di un namespace; può anche essere simile a un'istruzione `uses` tradizionale:

```
unit UnitA;
```

Una unit che non dichiara la propria appartenenza a un namespace è una *unit generica*. Le unit di questo tipo entrano automaticamente a far parte del namespace del progetto che le include. Questo aspetto, comunque, non influisce sul nome del file sorgente.

Nel momento in cui scrivo, il supporto per i namespace è piuttosto limitato; questo paragrafo, quindi, descrive più il funzionamento dei namespace nella versione futura che in quella attuale.

Il file di progetto può includere una clausola `namespaces` che elenca i namespace in cui il compilatore dovrà cercare le unit a cui si fa riferimento in modo generico. La clausola `namespaces`, se presente, deve trovar posto subito dopo l'istruzione `program` (o `package`, o `library`). Gli elementi dell'elenco devono essere separati da virgole e l'istruzione termina con l'usuale punto e virgola. Per esempio:

```
program NamespaceA.MyProgram
  namespaces Foo.Bar, Foo.Frob, Foo.Nitz;
```

In questo esempio i riferimenti a unit generiche sono risolti cercando nei namespace `Foo.Bar`, `Foo.Frob` e `Foo.Nitz`.

Vediamo ora, con un esempio pratico, come il compilatore risolve i riferimenti a unit generiche. Se si utilizza il nome completo di namespace, non ci sono problemi:

```
uses Foo.Frob.Gizmos;
```

In questo caso il compilatore sa già il nome del file `dcuil` (o `pas`). Supponiamo invece di scrivere il riferimento seguente:

```
uses Gizmos;
```

Questo è un *riferimento generico a una unit* e il compilatore deve essere in grado di reperire il file `dcuil` corrispondente. La ricerca viene effettuata nei namespace nell'ordine riportato di seguito.

1. Il namespace della unit corrente (se specificato).
2. Il namespace predefinito del progetto (se specificato).
3. I namespace indicati nella clausola `namespaces` (se presente).
4. I namespace specificati da opzioni del compilatore.

Se la unit corrente specifica un namespace, tutti i riferimenti generici presenti nella sua clausola `uses` sono risolti cercando prima in questo namespace (punto 1). Per esempio:

```
unit Foo.Frob.Gizmos;
uses doodads;
```

La unit `doodas` viene cercata prima nel namespace `Foo.Frob`, quindi il compilatore tenta di aprire il file `Foo.Frob.doodas.dcuil`. Se il file non esiste, il compilatore utilizzerà come prefisso per il nome il namespace predefinito del progetto (punto 2) e così via.

Lo stesso nome può comparire in diversi namespace; in questo caso, occorre fare riferimento al simbolo qualificandolo con il nome della unit (completo di namespace) in cui è definito. Se avete definito il simbolo `Hoozitz` nella unit `Foo.Frob.Gizmos`, potete farvi riferimento in uno dei due modi seguenti:

```
Hoozitz; // solo se il nome non è ambiguo
Foo.Frob.Gizmos.Hoozitz;
```

e non così:

```
Gizmos.Hoozitz;      // errore!
Frob.Gizmos.Hoozitz; // errore!
```

I nomi delle unit, se completi di namespace, possono essere lunghi e complessi; per questo motivo, è possibile creare un alias per il nome completo utilizzando la parola chiave `as` nella clausola `uses`:

```
uses Foo.Frob.DepartmentOfRedundancyDepartment.UIToys as ToyUnit;
```

Poiché un alias è di fatto un nuovo identificatore, il suo nome deve essere diverso da tutti gli identificatori della unit (gli alias sono locali alla unit). Anche dopo aver dichiarato un alias è consentito l'utilizzo del nome completo per fare riferimento alla unit.

I metadati dell'assembly contengono il nome di ciascun alias utilizzato, riportato esattamente come lo si digita nel sorgente (con le stesse maiuscole e minuscole). Per quanto riguarda Delphi, comunque, due namespace che differiscono solo per la mancata corrispondenza di maiuscole e minuscole sono equivalenti.

Identificatori estesi

L'integrazione tra CTS e CLR apre scenari interessanti per chi produce compilatori. Per esempio, come procedere se il nome di un identificatore in un assembly è una parola riservata in un certo linguaggio? Consideriamo la parola chiave `type` del linguaggio Delphi. `Type` è anche il nome di una classe del CLR. Poiché in Delphi si tratta di una parola riservata, `type` non può essere utilizzata come identificatore. Delphi per .NET consente di evitare questo problema in due modi (non disponibili in Delphi 7 e nelle versioni precedenti).

Innanzi tutto, si può utilizzare il nome completo dell'identificatore:

```
var
  T: System.Type;
```

In alternativa, si può adottare come prefisso per il nome il nuovo operatore `&`. L'effetto del codice seguente è identico a quello mostrato in precedenza:

```
var
  T: &Type;
```

In questa istruzione il compilatore tratta `Type` come simbolo e non come parola chiave, grazie all'operatore `&`. Il compilatore cercherà tra le unit disponibili e troverà il simbolo, in questo caso, nella unit `System` (ma il meccanismo, naturalmente, è indipendente dalla unit che definisce il simbolo).

Parole chiave final e sealed

Il compilatore Delphi per .NET introduce altri due concetti specificati dalla CLI (*Common Language Infrastructure*): l'attributo `sealed` per le classi e l'attributo `final` per i metodi. Una classe dotata dell'attributo `sealed` non può essere utilizzata come classe base. Un esempio è il seguente:

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

Una classe non può derivare da una che abbia questo attributo (la traduzione letterale del termine *sealed* è "sigillata"). Allo stesso modo, per i metodi identificati dall'attributo `final` non è ammesso l'override nelle classi derivate:

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

  TDeriv2 = class (TDeriv1)
    procedure A; override; // errore: "cannot override a final method"
  end;
```

Borland ha introdotto le parole chiave `sealed` e `final` poiché richieste da .NET, ma perché Microsoft ha ideato questi attributi? Questi attributi offrono a chi si serve di una classe suggerimenti importanti su come l'autore intende che la si utilizzi; inoltre, forniscono al compilatore suggerimenti per la generazione di codice CLI più efficiente.

Nuovi specificatori di visibilità e di accesso

La nozione di visibilità in Delphi (basata sulle parole chiave `private`, `protected` e `public`) non corrisponde perfettamente a quella della CLI. In linguaggi come C++ o Java, quando si specifica che un membro è privato, questo è visibile unicamente all'interno della classe; un membro protetto, invece, è visibile anche nelle classi derivate. Come abbiamo visto nel Capitolo 2, in Delphi questo è vero solo per classi definite in unit differenti, poiché tutti gli elementi definiti all'interno di una unit sono sempre visibili dalla unit stessa. La compatibilità con il CTS ha pertanto imposto l'introduzione di nuovi specificatori di visibilità:

- ✔ **class private:** un membro dichiarato `class private` segue le regole C++ e Java. Un membro di questo tipo è quindi visibile unicamente in metodi della classe che lo dichiara. Le procedure e funzioni dichiarate a livello di unit e i metodi di altre classe della stessa unit non hanno accesso al membro;
- ✔ **class protected:** i membri `class protected` sono visibili unicamente all'interno della classe che li dichiara e delle classi derivate. L'accesso da parte di altre classi definite nella stessa unit è possibile solo se si tratta di classi derivate.

L'esempio `ProtectedPrivate` nella cartella `LanguageTest` degli esempi mostra alcuni casi elementari.

Membri statici di classe

Delphi supporta da tempo i metodi di classe, cioè quelli applicabili a una classe nel suo insieme così come a una istanza specifica (nel primo caso, il metodo non ha accesso all'istanza in quando `Self` punta alla classe corrente e non a un oggetto). Delphi per .NET aggiunge lo specificatore `class static`, le proprietà di classe, i campi statici di classe e i costruttori di classe.

- **Metodi statici di classe:** come i metodi di classe in Delphi 7, i metodi statici di classe possono essere chiamati senza un'istanza, ma non è disponibile un parametro implicito `Self`. A differenza di quanto avviene in Delphi 7, infatti, non si può fare riferimento alla classe. Per esempio, non si può chiamare il metodo `ClassName`; un'ateriore differenza risiede nel fatto che i metodi di questo tipo non possono essere virtuali.
- **Proprietà statiche di classe:** come i metodi di classe, le proprietà sono accessibili senza utilizzare alcuna istanza. I metodi o i campi di accesso per queste proprietà devono essere metodi o campi statici di classe. Le proprietà statiche di classe non possono avere visibilità `published`, né clausole `stored` o `default`.
- **Campi statici di classe:** un campo statico di classe è accessibile senza un'istanza; insieme alle proprietà statiche di classe, questi campi sono generalmente utilizzati come strumenti per la progettazione. Permettono, in pratica, di dichiarare variabili e costanti all'interno del contesto di una classe.
- **Costruttori di classe:** un costruttore di classe è un costruttore privato (deve essere dichiarato con visibilità `class private`) chiamato prima del primo utilizzo della classe che lo dichiara. Il CLR non definisce in modo certo il momento in cui questo accade, ma si limita a definire questo momento come precedente al primo utilizzo della classe. In termini di CLR questo è complesso da stabilire, poiché il codice non è "utilizzato" finché non è eseguito. È ammesso dichiarare un solo costruttore di classe per ciascuna classe; ciascuna classe derivata può poi dichiarare un proprio costruttore di classe.

Non è possibile chiamare un costruttore di classe dal codice; il costruttore di classe è chiamato automaticamente per inizializzare i campi e le proprietà statiche di classe; anche la parola chiave `inherited` è proibita, poiché il compilatore se ne occupa automaticamente.

La dichiarazione di esempio seguente mostra la sintassi di queste nuove funzionalità:

```
TMyClass = class
class private // accessibile solo all'interno di TMyClass
  // I costruttori di classe devono avere visibilità class private
  class constructor Create;
class protected // accessibili in TMyClass e classi derivate
  // Metodi di accesso per la proprietà di classe P1
  class static function getP1 : Integer;
  class static procedure setP1(val : Integer);
public
  // Il metodo fx può essere chiamato senza un'istanza
  class static function fx(p : Integer) : Integer;
  // La proprietà statica di classe P1 deve avere metodi
  // di accesso statici di classe
  class static property P1 : Integer read getP1 write setP1;
end;
```

Tipi annidati

I tipi annidati sono simili ai membri di classe, nel senso che sono accessibili attraverso un class reference (non serve un'istanza). Un tipo annidato è dichiarato all'interno dell'intervallo di visibilità di una classe e consente di utilizzare la classe come una sorta di namespace per il tipo.

Eventi multipli (multicast)

La possibilità di impostare un metodo per l'ascolto di un evento (cioè un metodo che viene chiamato al verificarsi di un certo evento) esiste in Delphi da sempre. Il CLR supporta l'impostazione di diversi ascoltatori per ciascun evento, in un'architettura di eventi *multicast*. Delphi per .NET introduce due nuovi metodi di accesso per le proprietà, `add` e `remove`, a supporto degli eventi multicast. Questi metodi possono essere definiti unicamente per quelle proprietà che rappresentano eventi.

Per creare un evento multicast serve innanzi tutto un metodo per memorizzare tutte le funzioni che si registrano per l'ascolto. Come abbiamo visto nel Capitolo 24, questi eventi sono supportati grazie alla classe CLR `MulticastDelegate` e il compilatore nasconde gran parte della complessità insita nell'utilizzo di questa classe. Tramite le parole chiave `add` e `remove` si gestiscono l'aggiunta e la rimozione di ascoltatori, ma il meccanismo di memorizzazione è un dettaglio relativo all'implementazione di cui non è necessario occuparsi. Il compilatore genera i metodi `add` e `remove` automaticamente e questi implementano la memorizzazione delle funzioni da chiamare in modo efficiente.

Nella versione finale di Delphi per .NET i metodi `add` e `remove` dovrebbero funzionare insieme a due versioni overloaded delle funzioni standard `Include` e `Exclude`. Nel codice, volendo registrare un metodo come ascoltatore per un evento, si chiama `Include`. Per effettuare l'operazione inversa si utilizza invece `Exclude`. Per esempio:

```
Include(EventProp, eventHandler);
Exclude(EventProp, eventHandler);
```

Dietro le quinte, `Include` e `Exclude` chiameranno i metodi assegnati rispettivamente alle funzioni `add` e `remove` della proprietà. Nel momento in cui scrivo, questa parte del compilatore non è ancora stata completata, per cui non posso presentare esempi reali.

Per supportare il codice esistente, l'operatore di assegnamento di Delphi (`:=`) continua a funzionare nello stesso modo, cioè assegna un singolo event handler. Il compilatore, in questo caso, genererà codice per sostituire l'ultimo gestore assegnato tramite lo stesso operatore `:=` (e solo quello). In altre parole, l'operatore di assegnamento funziona in modo indipendente dal meccanismo `add/remove` (o `Include/Exclude`), cioè non influenza l'elenco di gestori memorizzato dalla classe `MulticastDelegate`.

Fate riferimento, come esempio, al programma `XmlDemo`. Il codice seguente (che si serve dell'unica sintassi funzionante al momento in cui scrivo) crea un pulsante a run time e ancora due gestori al suo evento `Click`:

```
MyButton := Button.Create;
MyButton.Location := Point.Create (
  Width div 2 - MyButton.Width div 2, 2);
```

```
MyButton.Text := 'Load';
MyButton.add_Click (OnButtonClick);
MyButton.add_click (OnButtonClick2);
Controls.Add (MyButton);
```

Attributi personalizzati

Come ricorderete dal Capitolo 24, una delle caratteristiche della CLI è un sistema di metadati estensibile. Tutti i compilatori .NET hanno l'obbligo di emettere metadati per i tipi definiti in un assembly. I metadati sono estensibili nel senso che i programmatori possono definire propri attributi da applicare pressoché a qualunque elemento: assembly, classi, metodi e altro. Il compilatore emetterà poi questi attributi personalizzati con i metadati dell'assembly. A run time, è possibile verificare quali attributi siano associati a una certa entità (un assembly, una classe, un metodo e così via) tramite metodi della classe CLR `System.Type`.

Gli attributi personalizzati sono tipi gestiti per riferimento derivati dalla classe CLR `System.Attribute`. La dichiarazione della classe per un attributo personalizzato è analoga a quella di qualunque altra (questa porzione di codice è tratta dall'esempio `NetAttributes` della cartella `LanguageTest`):

```
type
  TMyCustomAttribute = class(TCustomAttribute)
  private
    FAttr : Integer;
  public
    constructor Create(val: Integer);
    property customAttribute : Integer read FAttr write FAttr;
  end;
  ...
  constructor TMyCustomAttribute.Create(val: Integer)
  begin
    inherited Create;
    customAttribute := val;
  end;
```

La sintassi per l'applicazione dell'attributo è simile a quella adottata in C#:

```
type
  [TMyCustomAttribute(17)]
  TFoo = class
  public
    function P1(X : Integer) : Integer;
  end;
```

L'attributo è applicato al costrutto che lo segue. Nel nostro esempio, si tratta della classe `TFoo`. Avrete senza dubbio notato come la sintassi sia simile a quella dei GUID di Delphi. In questo frangente si presenta un problema: i GUID si applicano alle interfacce e devono *seguire* la dichiarazione dell'interfaccia. Gli attributi personalizzati, invece, devono *precedere* la dichiarazione. Come può il compilatore stabilire se quanto compare tra parentesi quadre (dopo una dichiarazione di interfaccia) è un tradizionale GUID o un attributo personalizzato da applicare al primo membro dell'interfaccia?

Occorre trattare come caso speciale gli attributi personalizzati nelle interfacce. Se si applica un GUID a un'interfaccia, questo deve immediatamente seguirne la dichiarazione e deve rispettare la sintassi Delphi tradizionale:

```
type
  interface IMyInterface
  ['(12345678-1234-1234-1234-1234567890ab)']
```

L'attributo personalizzato `GuidAttribute` del CLR è utilizzato per l'applicazione dei GUID ed è parte del namespace `System.Runtime.InteropServices`. Se si adotta questa tecnica per applicare un GUID, occorre seguire la sintassi CLR e applicarlo prima della dichiarazione dell'interfaccia.

Class helper

I class helper sono una funzionalità molto interessante aggiunta al linguaggio in Delphi per .NET. Il motivo principale della loro introduzione è il modo in cui Borland fa corrispondere le classi .NET con le classi della RTL (come vedremo nel paragrafo "Class helper per la RTL", nel prosieguo del capitolo). In questo paragrafo mi occuperò di tale funzionalità dal punto di vista del linguaggio.

Un class helper offre la possibilità di estendere una classe esistente senza utilizzare l'ereditarietà, aggiungendo nuovi metodi (ma non nuovi dati). L'aspetto inusuale, rispetto all'approccio basato sull'ereditarietà, risiede nel fatto che è possibile istanziare oggetti della classe originale, poiché l'estensione non ne modifica il nome. Questo significa che è possibile inserire metodi in una classe esistente. Un esempio chiarirà questo aspetto.

Supponiamo di avere una classe (non sviluppata da noi, altrimenti basterebbe l'ereditarietà per estenderla) definita nel modo seguente:

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

È possibile aggiungere un metodo `Show` agli oggetti di questa classe creando un class helper:

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

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

Notate che `Self`, nei metodi di un class helper, fa riferimento all'oggetto su cui il metodo è stato chiamato ed è del tipo della classe di tale oggetto. Il codice che utilizza il class helper ha un aspetto simile al seguente:

```
Obj := TMyObject.Create;
...
Obj.Show;
```

L'output mostra il nome della classe `TMyObject`; se si deriva una classe da `TMyObject`, tuttavia, il class helper è disponibile anche per la nuova classe (questo significa che si è aggiunto un metodo a un'intera gerarchia di classi) e tutto funziona correttamente. Per i vostri esperimenti, fate riferimento all'esempio `ClassHelperDemo` nella cartella `LanguageTest`.

Libreria run time e VCL

I sorgenti della libreria run time (RTL) sono memorizzati nella cartella `source\rtl` dell'installazione dell'anteprima di Delphi per .NET. La migrazione delle unit in namespace CLR è indicata già dai nomi dei file.

Borland ha seguito (nella maggior parte dei casi) l'approccio di mantenere i nomi tradizionali delle unit utilizzando come prefisso il namespace `Borland.Delphi`. Le caratteristiche specifiche del sistema operativo (come il supporto per il Registry e i file INI) fanno parte del namespace `Borland.Win32`, poiché si tratta di classi, procedure e funzioni Borland che incapsulano caratteristiche specifiche di Windows. La nomenclatura dovrebbe proseguire secondo questo stile, anche se non tutte le unit saranno portate sotto .NET e alcune verranno probabilmente riorganizzate in diversi namespace.

L'esame dei sorgenti è insieme educativo e consigliabile, purché si tenga conto che non si tratta ancora della versione finale del prodotto. Poiché il contenuto dei file della RTL è soggetto a modifiche, è consigliabile non formulare alcuna assunzione e non introdurre nel proprio codice dipendenze basate su quello che compare in questa versione dei sorgenti della RTL.

Class helper per la RTL

Detto questo, vediamo come è cambiata la RTL fino a questo momento. La modifica più interessante, probabilmente, è rappresentata dall'introduzione dei class helper.

La unit `Borland.Delphi.System.pas` contiene questa dichiarazione:

```
type
  TObject = System.Object;
```

Il significato è che la classe `TObject` di Delphi è un alias per la classe CLR `System.Object`. Questo è importante: `TObject` non è una classe derivata da `System.Object` ed è semanticamente equivalente. Che ne è dei metodi tradizionalmente definiti in `TObject`, come `ClassName` e `ClassParent`? In questo frangente risulta utile un class helper.

I metodi che erano dichiarati e implementati direttamente in `TObject` sono ora dichiarati e implementati in una classe chiamata `TObjectHelper`, che è un class helper di `TObject`. La dichiarazione, tratta da `Borland.Delphi.System.pas`, è la seguente:

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

Un class helper consente di estendere una classe senza utilizzare l'ereditarietà.

Potrebbe essere necessario estendere una classe CLR senza derivare da essa se, per esempio, si volesse consentire l'utilizzo della classe CLR a codice Delphi esistente. Come avrete sicuramente notato, il framework di Delphi e quello di .NET hanno molte funzionalità in comune. In alcuni casi esistono conflitti di nomi, come avviene per esempio tra la classe `Exception` di Borland e la classe CLR `System.Exception`. Le due classi dispongono di funzionalità equivalenti, ma le espongono in modi diversi. Poiché esiste un numero elevato di programmi Delphi che si servono della classe `Exception`, l'unica soluzione consisteva nel creare un meccanismo che consentisse agli sviluppatori (compresi quelli Borland) di sfruttare le classi CLR e di estenderle aggiungendo le caratteristiche storicamente supportate dalle equivalenti classi Delphi. Le stesse caratteristiche che il codice Delphi esistente si aspetta di trovare.

VCL

Le classi .NET del namespace `System.Windows.Forms` non sostituiscono le parti dell'API di Win32 dedicate all'interfaccia grafica. Lo stesso accade con la maggior parte del Framework .NET, il cui scopo è semplificare l'utilizzo dell'API sottostante tramite un'interfaccia a oggetti compatibile con i servizi di base nell'ambiente .NET. Il sottosistema grafico di Win32 è sempre al suo posto e svolge le operazioni che ha sempre effettuato. `System.Windows.Forms` lo riorganizza in forma object oriented e costruendoci sopra un sistema di eventi, ma le classi in `System.Windows.Forms` chiamano direttamente il codice nativo Win32. Quando si utilizza `System.Windows.Forms`, ci si serve sempre dell'API di Windows, ma utilizzando il CLR come (grande) intermediario.

Questo preambolo è importante per comprendere il motivo per cui la VCL adotta lo stesso approccio. La classe `TObject` è resa analoga a `System.Object` per mezzo di un class helper; `TPersistent` e `TComponent` sono ancora derivate da `TObject`. Quindi la classe `TForm` della VCL, per esempio, non deriva da `System.Windows.Forms.Form`. L'intera VCL resterà invece molto simile alla versione attuale: `TForm` deriva da `TWinControl`, che deriva da `TControl`, che deriva da `TComponent`.

Se si considera l'intero namespace System.Windows.Forms, la VCL è in qualche modo parallela a esso, piuttosto che soprastante. Entrambi i framework si affidano alle funzioni non gestite (dal CLR) dell'API di Windows per l'implementazione dei controlli dell'interfaccia utente.

Analisi dei sorgenti della VCL.NET

L'aggiornamento dell'anteprima di Delphi per .NET uscito nel Novembre 2002, per quanto ancora preliminare, mostra alcuni dettagli aggiuntivi sull'architettura che Borland sta preparando. Se aprite la unit Borland.Vcl.Controls, la sua somiglianza con la versione Win32 vi sorprenderà. Il codice è quasi identico e la differenza è nascosta dietro le quinte delle classi TObject e TComponent. Poiché ho già parlato di TObject, illustrerò i componenti, che sono definiti in tre passaggi:

```
type
  TComponent = System.ComponentModel.Component;
  TComponentHelper = class helper (TPersistentHelper) for TComponent
  TComponentSite = class(TObject, ISite, IServiceProvider)
```

La classe TComponent corrisponde alla classe .NET System.ComponentModel.Component ed è dotata di un helper che fornisce metodi e proprietà aggiuntivi, nonché di una classe aggiuntiva che contiene i dati necessari al class helper (che non può contenerne di propri). La situazione è complessa e non desidero addentrarmi nei dettagli, poiché la classe TComponent è a livello sperimentale, al momento, e potrebbe variare notevolmente in futuro.

Tornando alla VCL sono disponibili vari componenti, per cui è possibile cominciare la conversione del codice esistente verso .NET. L'unico problema, in merito, risiede nel fatto che l'aggiornamento del Novembre 2002 non supporta ancora lo streaming; per questo motivo, occorre fornire i componenti da codice alla creazione di ciascun form (nel costruttore). Questa operazione non sarà necessaria nella versione finale di Delphi per .NET.

Per presentare un esempio interessante che aiuti a esplorare l'architettura delle classi della VCL, ho convertito l'esempio ClassInfo presentato nel Capitolo 3. L'esempio NetClassInfo si serve del codice di progetto seguente (anche questa è una modifica che non sarà necessaria nella versione finale del compilatore):

```
  Application.Initialize;
  Form1 := TForm1.Create (Application);
  Application.MainForm := Form1;
```

Il codice del form, come ho anticipato, dispone di un metodo chiamato dal costruttore che si occupa di inizializzare i controlli. Questo è un metodo piuttosto esteso, per cui ne riporterò solo una porzione:

```
procedure TForm1.InitializeControls;
begin
  // creazione dei controlli...
  Label3:= TLabel.Create(Self);
  Panel1:= TPanel.Create(Self);
  Label1:= TLabel.Create(Self);
  Label2:= TLabel.Create(Self);
  ...

  // impostazione di proprietà ed eventi del form
```

```
    Left:= 217;
    Top:= 109;
    Caption:= 'Class Info';
    OnCreate:= FormCreate;

    // inizializzazione dei controlli (il listato ne comprende solo uno)
    with Label3 do
    begin
      Parent:= Self;
      Left:= 8;
      Top:= 8;
      Width:= 56;
      Height:= 13;
      Caption:= 'Class Name';
    end;
```

la parte restante del codice dell'applicazione è pressoché identica, il che è sorprendente se si considera che questo è un esempio di basso livello. Ho dovuto eliminare la chiamata a `InstanceSize` poiché il compilatore non è in grado di ricavare la dimensione di un oggetto in .NET, e ho dovuto effettuare il test sul nome della classe base rispetto a `Object` anziché `TObject`. Il codice responsabile dell'ouput mostrato nella Figura 25.1 è il seguente:

```
procedure TForm1.ListClassesClick(Sender: TObject);
var
  MyClass: TClass;
begin
  MyClass := ClassArray [ListClasses.ItemIndex];
  EditInfo.Text := Format ('Name: %s - Size: %d bytes',
    [MyClass.ClassName, 0 {MyClass.InstanceSize}]);
  with ListParent.Items do
  begin
    Clear;
    while MyClass.ClassName <> 'Object' do
    begin
      MyClass := MyClass.ClassParent;
      Add (MyClass.ClassName);
    end;
  end;
end;
```

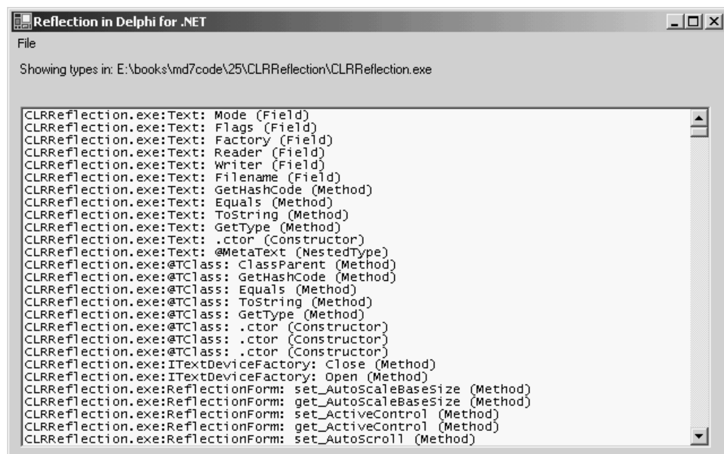

Figura 25.1 *L'esempio NetClassInfo mostra tutte le classi base di un componente.*

Altri esempi VCL

Come punto di partenza per le vostre sperimentazioni con la VCL sotto .NET ho creato altri due esempi. `NetEuroConv` è una versione .NET dell'esempio `EuroConv` del Capitolo 3, che si serve del motore di conversione della RTL; `NetLibSpeed` è invece il porting dell'esempio `LibSpeed` del Capitolo 5, che mette a confronto le prestazioni di VCL e CLX nella creazione di componenti visuali. I numeri che vedrete hanno poco senso in una versione preliminare della libreria come questa, anche se il fatto che la VCL.NET impieghi un tempo 4/5 volte superiore per creare i controlli rispetto alla VCL tradizionale potrebbe preoccuparvi.

Come ho detto, questi esempi non sono che punti di partenza per ulteriori esperimenti. Potrebbero anche non funzionare in versioni future dell'anteprima di Delphi per .NET.

Visitate spesso il mio sito Internet per aggiornamenti a questa parte del libro e nuovi esempi.

Utilizzo delle librerie Microsoft

La VCL non è ancora pronta, ma si può utilizzare la libreria di classi del Framework .NET per sperimentare con l'anteprima del compilatore di Delphi per .NET. Un'attività particolarmente educativa risiede nel compilare i programmi e in seguito controllarli con ILDASM. Il presente paragrafo affronterà questo argomento. Per un esempio più semplice che utilizza il supporto a XML, fate riferimento al programma `XmlDemo` menzionato in precedenza.

Il programma `CLRReflection` apre un assembly e si serve delle tecniche di reflection per ispezionare i moduli e i tipi che definisce. Il programma mostra l'utilizzo di una finestra di dialogo comune (`OpenFileDialog`), la creazione di menu, la gestione di eventi, gli array dinamici di Delphi e, naturalmente, le tecniche di reflection. Vediamo prima il file di progetto:

```
program CLRReflection;

uses
  System.Windows.Forms,
  ReflectionUnit;

var
  reflectForm : ReflectionForm;
begin
  reflectForm := ReflectionForm.Create;
  System.Windows.Forms.Application.Run(reflectForm);
end.
```

Il codice è simile a quello di un'applicazione VCL tradizionale. Si definisce una variabile per il form principale e poi lo si crea; quindi, si utilizza il metodo `Run` della classe `System.Windows.Forms.Application` per avviare l'applicazione in modo analogo (almeno come concetto) a quanto avviene con la VCL.

Notate che ho utilizzato i nomi completi delle classi .NET, in modo che ne risulti chiara la provenienza. Poiché la clausola `uses` include `System.Windows.Forms`, l'espressione:

```
System.Windows.Forms.Application.Run(reflectForm);
```

potrebbe essere abbreviata in:

```
Application.Run(reflectForm);
```

Vediamo ora il Listato 25.1, che riporta il codice della unit che definisce il form principale. Notate che questo codice è compilabile con la versione del novembre 2002 del compilatore, ma non con la versione distribuita inizialmente con Delphi 7.

Listato 25.1 *La unit ReflectionUnit dell'esempio CLRReflection*

```
unit ReflectionUnit;

interface

uses
    System.Windows.Forms,
    System.Reflection,
    System.Drawing,
    Borland.Delphi.SysUtils;

type
  ReflectionForm = class(System.Windows.Forms.Form)
  private
    mainMenu: System.Windows.Forms.MainMenu;
    fileMenu: System.Windows.Forms.MenuItem;
    separatorItem: System.Windows.Forms.MenuItem;
    openItem: System.Windows.Forms.MenuItem;
    exitItem: System.Windows.Forms.MenuItem;

    showFileLabel: System.Windows.Forms.Label;
    typesListBox: System.Windows.Forms.ListBox;
    openFileDialog: System.WIndows.Forms.OpenFileDialog;
  protected
    procedure InitializeMenu;
    procedure InitializeControls;
    procedure PopulateTypes(fileName: String);
    { gestori di eventi }
    procedure exitItemClick(sender: TObject; Args: System.EventArgs);
    procedure openItemClick(sender: TObject; Args: System.EventArgs);
  public
    constructor Create;
end;

implementation

constructor ReflectionForm.Create;
begin
  inherited Create;

  SuspendLayout;
  InitializeMenu;
  InitializeControls;

  { inizializza il form e altre variabili }
  openFileDialog := System.Windows.Forms.OpenFileDialog.Create;
  openFileDialog.Filter := 'Assemblies (*.dll;*.exe)¦*.dll;*.exe';
  openFileDialog.Title := 'Open an assembly';

  AutoScaleBaseSize := System.Drawing.Size.Create(5, 13);
```

```
    ClientSize := System.Drawing.Size.Create(631, 357);
    Menu := mainMenu;
    Name := 'reflectionForm';
    Text := 'Reflection in Delphi for .NET';

    { aggiunge i controlli al form }
    Controls.Add(showFileLabel);
    Controls.Add(typesListBox);
    ResumeLayout;
  end;

  { crea il menu principale }
  procedure ReflectionForm.InitializeMenu;
  var
    menuItemArray : array of System.Windows.Forms.MenuItem;
  begin
    mainMenu := System.Windows.Forms.MainMenu.Create;
    fileMenu := System.Windows.Forms.MenuItem.Create;
    openItem := System.Windows.Forms.MenuItem.Create;
    separatorItem := System.Windows.Forms.MenuItem.Create;
    exitItem := System.Windows.Forms.MenuItem.Create;

    { inizializza il menu principale }
    mainMenu.MenuItems.Add(fileMenu);

    { inizializza fileMenu }
    fileMenu.Index := 0;
    SetLength(menuItemArray, 3);
    menuItemArray[0] := openItem;
    menuItemArray[1] := separatorItem;
    menuItemArray[2] := exitItem;
    fileMenu.MenuItems.AddRange(menuItemArray);
    fileMenu.Text := '&File';

    // openItem
    openItem.Index := 0;
    openItem.Text := '&Open...';
    openItem.add_Click(openItemClick);

    // separatorItem
    separatorItem.Index := 1;
    separatorItem.Text := '-';

    // exitItem
    exitItem.Index := 2;
    exitItem.Text := 'E&xit';
    exitItem.add_Click(exitItemClick);
  end;

  { crea i controlli e popola il form }
  procedure ReflectionForm.InitializeControls;
  begin
    { inizializza showFileLabel }
    showFileLabel := System.Windows.Forms.Label.Create;
    showFileLabel.Location := System.Drawing.Point.Create(5, 6);
    showFileLabel.Name := 'showFileLabel';
    showFileLabel.Size := System.Drawing.Size.Create(616, 37);
    showFileLabel.TabIndex := 0;
    showFileLabel.Anchor := System.Windows.Forms.AnchorStyles.Top or
      System.Windows.Forms.AnchorStyles.Left or
```

```
    System.Windows.Forms.AnchorStyles.Right;
  showFileLabel.Text := 'Showing types in: ';

  { inizializza typesListBox }
  typesListBox := System.Windows.Forms.ListBox.Create;
  typesListBox.Anchor := System.Windows.Forms.AnchorStyles.Top or
    System.Windows.Forms.AnchorStyles.Bottom or
    System.Windows.Forms.AnchorStyles.Left or
    System.Windows.Forms.AnchorStyles.Right;
  typesListBox.Location := System.Drawing.Point.Create(8, 46);
  typesListBox.Name := 'typesListBox';
  typesListBox.Size := System.Drawing.Size.Create(610, 303);
  typesListBox.Font := System.Drawing.Font.Create('Lucida Console',
    8.25, System.Drawing.FontStyle.Regular,
    System.Drawing.GraphicsUnit.Point, 0);
  typesListBox.TabIndex  := 1;
end;

{ gestore per la voce di menu Exit }
procedure ReflectionForm.exitItemClick(sender: TObject;
  Args: System.EventArgs);
begin
  System.Windows.Forms.Application.Exit;
end;

{ gestore per la voce di menu Open }
procedure ReflectionForm.openItemClick(sender: TObject;
  Args: System.EventArgs);
begin
  if openFileDialog.ShowDialog = DialogResult.OK then
  begin
    showFileLabel.Text := 'Showing types in: ' + openFileDialog.FileName;
    PopulateTypes(openFileDialog.FileName);
  end;
end;

{ Apre l'assembly indicato e applica la reflection sui moduli e
  i tipi definiti in esso }
procedure ReflectionForm.PopulateTypes(fileName : String);
var
  assy: System.Reflection.Assembly;
  modules: array of System.Reflection.Module;
  module: System.Reflection.Module;
  types: array of System.Type;
  t: System.Type;
  members: array of System.Reflection.MemberInfo;
  m: System.Reflection.MemberInfo;
  i,j,k: Integer;
  s: String;
begin
  try
    { ripulisce la ListBox }
    typesListBox.BeginUpdate;
    typesListBox.Items.Clear;

    { Carica l'assembly e legge l'elenco dei moduli }
    assy    := System.Reflection.Assembly.LoadFrom(fileName);
    modules := assy.GetModules;

    { per ciascun modulo, legge l'elenco dei tipi }
```

```
      for i := 0 to High(modules) do
      begin
        module := modules[i];
        types  := module.GetTypes;

        { per ciascun tipo, legge l'elenco dei membri }
        for j := 0 to High(types) do
        begin
          t := types[j];
          members := t.GetMembers;

          { per ciascun membro, aggiunge alla ListBox dati sul tipo }
          for k := 0 to High(members) do
          begin
            m := members[k];
            s := module.Name + ':' + t.Name + ': ' + m.Name +
              ' (' + m.MemberType.ToString + ')';
            typesListBox.Items.Add(s);
          end;
        end;
      end;
      typesListBox.EndUpdate;
    except
      System.Windows.Forms.MessageBox.Show('Could not load the assembly');
    end;
  end;

  end.
```

La unit comincia dichiarando la propria dipendenza dai file `dcuil` del Framework .NET e dalla unit `Borland.Delphi.SysUtils`; in seguito dichiara la classe del form principale, derivata dalla classe .NET `System.Windows.Forms.Form`. La definizione della classe ha un aspetto familiare: esiste un campo per ciascun controllo e ciascun campo è del tipo corrispondente dichiarato nella libreria del Framework .NET.

Le funzioni `exitItemClick` e `openItemClick` sono gestori di eventi. L'elenco dei parametri passati a ciascun `evento` è quello definito dal CLR: l'oggetto che innesca l'evento (derivato da `System.Object`) e gli argomenti (incapsulati in un oggetto di classe `EventArgs` o classe derivata). Vedremo tra breve come ancorare i gestori agli eventi.

Passiamo al costruttore della classe. La prima riga, che chiama `inherited Create`, richiede attenzione.

In questo caso osserviamo una differenza sostanziale tra .NET e le abituali modalità operative della VCL e di Delphi in generale. In Delphi, un costruttore inizializza le variabili della classe e ha il compito di mettere l'oggetto in uno stato predeterminato e noto, ma non effettua alcuna allocazione di memoria. Per questo motivo, si può verificare il caso di un costruttore che effettua assegnamenti prima di chiamare la versione della classe base, o che non effettua alcuna chiamata. In Delphi per .NET questo approccio non è applicabile: nel costruttore, è obbligatorio chiamare `inherited Create`. *Se non lo si chiama, si ottiene un errore del compilatore il quale indica che* `Self` *non è inizializzata ed è necessario chiamare il costruttore ereditato per poter accedere ai campi della classe base.*

Il codice si serve di una gerarchia di classi nuova, ma dovrebbe risultare chiaro a qualunque programmatore Delphi. Si crea un'istanza di `System.Windows.Forms.OpenFileDialog` chiamando il suo costruttore `Create`; questo è l'approccio da utilizzare per istanziare qualunque classe .NET.

Le righe successive impostano proprietà dell'oggetto `OpenFileDialog` e del form. Per finire, si aggiungono due controlli (una Label per il nome del file e una ListBox per i dati dell'assembly) alla proprietà `Controls` del form, che è di tipo `Control.ControlCollection`.

La procedura `InitializeMenu` mostra la creazione e la configurazione di un'istanza di `System.Windows.Forms.MainMenu`. Quando s'inizializza il menu *File*, un array dinamico viene utilizzato per contenerne le singole voci e poi passato al metodo `AddRange`. Avrei potuto anche chiamare il metodo `Add` una volta per ciascuna voce di menu.

L'altro aspetto interessante di `InitializeMenu` è il collegamento dei gestori di eventi. Nel Capitolo 24 e nella prima parte di questo ho parlato della complessità insita nella gestione di eventi multicast e delegati e di come il compilatore la nasconda. In questo caso ne vediamo ricomparire una parte.

In Delphi per .NET non è ancora possibile, ma in altri linguaggi .NET (come C#) si può utilizzare la parola chiave `event` per inserire un delegato. La dichiarazione dell'evento specifica un delegato da utilizzare come meccanismo di callback. Poiché l'evento deriva da `System.MulticastDelegate` (nel caso specifico è un `System.EventHandler`), gli altri oggetti possono aggiungere (e rimuovere) gestori che verranno chiamati al verificarsi dell'evento.

Il C# adotta una sintassi semplificata per rendere meno complesso questo aspetto: gli operatori `+=` e `-=` possono essere utilizzati per aggiungere o rimuovere gestori da un evento, rispettivamente. Anche Delphi si muoverà in questa direzione, offrendo la possibilità di utilizzare `Include` e `Exclude` allo stesso modo (ne abbiamo parlato in precedenza). Il CTS obbliga tutti i compilatori a implementare metodi chiamati `add_<Evento>` e `remove_<Evento>`, che chiamino i metodi `Add` e `Remove` di `System.Delegate`.

Per ora, l'assegnazione dei gestori di eventi deve passare per questi metodi `add_` e `remove_`; normalmente non ce ne interesseremmo, poiché la complessità sarebbe nascosta dal compilatore. Nella dichiarazione della classe, ho introdotto due metodi compatibili con il delegato `System.EventHandler`: `openItemClick` e `exitItemClick`. Ho poi chiamato il metodo `add_Click` delle rispettive voci di menu passando i metodi come funzioni callback.

Ora che abbiamo concluso le impostazioni di base, vediamo il codice che applica la reflection ai tipi definiti in un assembly. Si può caricare qualunque assembly a partire dal nome del file, chiamando il metodo statico `LoadFrom` di `System.Reflection.Assembly` che crea e restituisce un'istanza. Una volta istanziato un oggetto di questo tipo, possiamo procedere senza problemi e applicare la reflection per esaminare l'assembly da qualunque angolazione.

L'elenco dei moduli contenuti nell'assembly è fornito dal metodo `GetModules`. A questo punto si può arrivare ai tipi definiti in ciascun modulo grazie a `GetTypes`. Come abbiamo visto nella procedura `InizializeMenu`, possiamo utilizzare gli array dinamici per le proprietà che espongono un elenco tramite `System.Array`.

Per finire, ciascun elemento degli elenchi di moduli e tipi dispone di una proprietà `Name`, utile per costruire una stringa da mostrare nella ListBox. L'effetto è mostrato nella Figura 25.2.

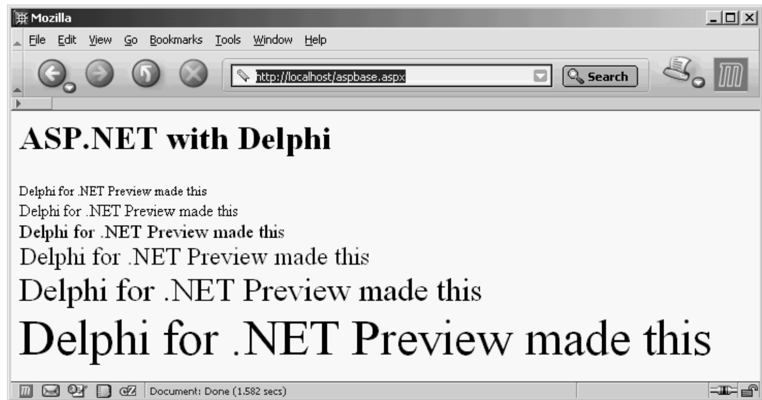

Figura 25.2
L'esempio CLRReflection mostra i dati di un assembly appena caricato.

ASP.NET in linguaggio Delphi

Potete apprezzarla o meno (per esempio, a me non soddisfa) ma la tecnologia ASP di Microsoft svolge un ruolo significativo nello sviluppo di applicazioni Web, almeno sotto Windows. Con la transizione verso ASP.NET, la tecnologia abbraccia completamente il Framework .NET; ora, con la disponibilità di un compilatore Delphi per .NET, si può utilizzare il linguaggio Delphi per lo sviluppo di applicazioni ASP.

Per effettuare una prova, configurate IIS per il supporto ad ASP.NET (non riporto in questa sede la procedura necessaria, che non rientra negli obiettivi del libro, ma potete trovare tutta la documentazione del caso all'indirizzo `www.asp.net`) e inserite nella cartella di destinazione il file `web.config` fornito da Borland con l'anteprima di Delphi per .NET (lo trovate nella cartella `aspx`). Questo file di configurazione definisce la corrispondenza tra un linguaggio e una libreria specifica, anch'essa fornita da Borland. Le impostazioni più importanti riportate nel file (che è in formato XML) sono le seguenti:

```
<compilation debug="true">
  <assemblies>
    <add assembly="DelphiProvider" />
  </assemblies>
  <compilers>
    <compiler language="Delphi" extension=".pas"
      type="Borland.Delphi.DelphiCodeProvider,DelphiProvider" />
  </compilers>
</compilation>
```

Per accertarsi che tutto sia configurato correttamente, è consigliabile prendere in considerazione un esempio. Create un nuovo file (io l'ho chiamato `aspbase.aspx` e lo trovate nella cartella `DelphiAsp` degli esempi) e scrivete il codice seguente:

```
<html>
<body>
```

```
<h1>ASP.NET with Delphi</h1>

<script language="Delphi" runat="server">
procedure HelloMessage(msg: string);
var
  i: Integer;
begin
  for i := 2 to 7 do
    Response.Write ('<font size=' + inttostr (i) +
      '>' + msg + '</font> <br>')
end;
</script>

<% HelloMessage('Delphi for .NET Preview made this'); %>

</body>
</html>
```

L'effetto è la trasformazione del codice in un sorgente .NET, la sua compilazione in linguaggio intermedio (ricordate che in .NET anche gli script sono compilati prima di essere eseguiti) con l'anteprima del compilatore Delphi e l'esecuzione del programma risultante. Se non si verificano problemi, il browser visualizzerà una finestra come quella mostrata nella Figura 25.3.

Figura 25.3
Il risultato dell'esempio aspbase.aspx in un browser.

Da questo momento in avanti, possiamo effettuare tutte le operazioni consentite in un'applicazione ASP.NET. L'ultimo esempio che mostrerò riguarda l'utilizzo di controlli con event handler, di cui ho già parlato in precedenza riguardo alle applicazioni .NET basate su form Windows.

Questo esempio, memorizzato nel file `aspui.aspx` della cartella `AspDelphi`, si serve di codice HTML per definire un form con una casella di testo, un pulsante e un'etichetta da utilizzare come output. Al pulsante è ancorato un event handler Delphi, che copia il testo inserito dall'utente dalla casella di testo all'etichetta (l'output del programma è mostrato nella Figura 25.4):

```
<html>
<body>

<h1>ASP.NET with Delphi</h1>

<script language="Delphi" runat="server">
  procedure ButtonClick(Sender: System.Object; E: EventArgs);
```

```
  begin
    Message.Text := Edit1.Text;
  end;
</script>

<form runat="server">
   <asp:textbox id="Edit1" runat="server"/>
   <asp:button text="Click Me!" OnClick="ButtonClick" runat="server"/>
</form>

<p><b><asp:label id="Message" runat="server" text="message"/></b></p>

</body>
</html>
```

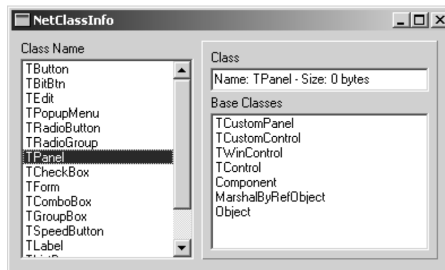

Figura 25.4 *L'output dell'esempio aspui.aspx, così come si presenta dopo aver digitato un testo e premuto il pulsante.*

Questa è un'introduzione limitata ad ASP.NET in Delphi. Tuttavia, dovrebbe risultare sufficiente a fornirvi un'idea delle possibilità che si presentano ai programmatori Delphi nel nuovo mondo di .NET.