

IDŹ DO

PRZYKŁADOWY ROZDZIAŁ



SPIS TREŚCI

KATALOG KSIĄŻEK

KATALOG ONLINE

ZAMÓW DRUKOWANY KATALOG

TWÓJ KOSZYK

DODAJ DO KOSZYKA

CENNIK I INFORMACJE

ZAMÓW INFORMACJE
O NOWOŚCIACH

ZAMÓW CENNIK

CZYTELNIA

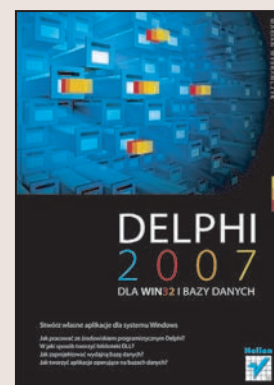
FRAGMENTY KSIĄŻEK ONLINE

Delphi 2007 dla WIN32 i bazy danych

Autor: Marian Wybrańczyk

ISBN: 978-83-246-1972-6

Stron: 608



Stwórz własne aplikacje dla systemu Windows

- Jak pracować ze środowiskiem programistycznym Delphi?
- W jaki sposób tworzyć biblioteki DLL?
- Jak zaprojektować wydajną bazę danych?
- Jak tworzyć aplikacje operujące na bazach danych?

Wśród wszystkich środowisk programistycznych umożliwiających tworzenie aplikacji Delphi jest jednym z najbardziej znanych i popularnych. To narzędzie, obecne na rynku od ponad dwunastu lat, cieszy się zasłużonym uznaniem twórców oprogramowania — dzięki sporym możliwościom, ogromnej bibliotece komponentów i czytelnej składni języka Object Pascal, będącego podstawą tego środowiska. Najnowsza wersja Delphi, oznaczona symbolem RAD Studio 2007, nie tylko umożliwia tworzenie „klasycznych” aplikacji dla Windows, opartych o Windows API, ale także udostępnia kontrolki platformy .NET.

Książka „Delphi 2007 dla WIN32 i bazy danych” to podręcznik opisujący zasady tworzenia aplikacji dla systemu Windows w najnowszej wersji Delphi. Przedstawia ona techniki tworzenia aplikacji bazodanowych w oparciu o mechanizmy Windows API i kontrolki VCL. Czytając ją, poznasz komponenty, jakie Delphi oferuje programiście, i dowiesz się, jak korzystać z nich we własnych aplikacjach. Opanujesz mechanizmy komunikacji z niemal wszystkimi systemami zarządzania bazami danych dostępnymi na rynku. Przeczytasz także o tworzeniu wersji instalacyjnych napisanych przez siebie aplikacji.

- Interfejs użytkownika Delphi 2007
- Komponenty dostępne w Delphi
- Przetwarzanie grafiki
- Korzystanie z komponentów VCL
- Aplikacje wielowątkowe
- Tworzenie bibliotek DLL
- Operacje na plikach
- Obsługa dokumentów XML
- Projektowanie bazy danych i struktury tabel
- Komunikacja z bazami danych
- Mechanizmy blokowania rekordów
- Tworzenie wersji instalacyjnych aplikacji

**Wykorzystaj możliwości najnowszej wersji środowiska programistycznego,
które zrewolucjonizowało proces tworzenia aplikacji!**

Wydawnictwo Helion
ul. Kościuszki 1c
44-100 Gliwice
tel. 032 230 98 63
e-mail: helion@helion.pl



Spis treści

Podziękowania	9
Wstęp	11
Rozdział 1. Środowisko Delphi — szybki start	15
Wstęp	15
Pierwszy projekt	15
Kompilujemy program	22
Najważniejsze ustawienia aplikacji	22
Sprawdzamy i wylapujemy błędy	24
Ustawiamy środowisko pracy	27
Popularne skróty klawiszowe	29
Podsumowanie	30
Rozdział 2. Wybrane komponenty środowiska	31
Wstęp	31
Uwaga na temat stosowanego nazewnictwa	31
Ogólne wskazówki dotyczące korzystania z komponentów	32
Zakładka Standard	35
MainMenu	36
Label	37
Edit	37
Memo	41
Button	43
CheckBox i GroupBox	44
RadioGroup i RadioButton	45
ListBox	46
ComboBox	48
ActionList	49
Zakładka Additional	51
BitBtn	51
SpeedButton	53
MaskEdit	54
StringGrid	54
DrawGrid	79
Image	80
Shape	80
Bevel	81
ScrollBar	81

CheckBox	84
Splitter	85
StaticText	85
ControlBar	85
ApplicationEvents	86
ValueListEditor	86
LabeledEdit	87
ColorBox	87
Chart	87
TFlowPanel, TGridPanel	89
Zakładka Win32	89
TabControl i PageControl	89
ImageList	89
RichEdit	90
TrackBar i ProgressBar	92
UpDown	92
HotKey	93
Animate	94
DateTimePicker i MonthCalendar	95
TreeView	95
ListView	97
StatusBar	98
ToolBar	99
Zakładka System	100
Timer	100
PaintBox	100
Zakładka Dialogs	102
Zakładka Win 3.1	103
Podsumowanie	104
Rozdział 3. Niewizualne środki programowania	105
Plik ini	105
Lista stringów — TStringList	112
Lista obiektów TList	117
Lista obiektów — klasa TObjectList	120
Schowek Windows	125
Rejestr Windows	134
Podsumowanie	137
Rozdział 4. Elementy grafiki	139
Podstawowe operacje na obrazach	171
Save Screen	179
Podstawy operacji graficznych w systemie Windows	181
Drukowanie grafiki	192
Podsumowanie	199
Rozdział 5. Wykorzystujemy bibliotekę VCL	201
Wstęp	201
Rozpoczynamy pracę z VCL	201
Zarządzanie formularzami	205
Okno modalne	206
Okno niemodalne	208
Ręczna kontrola życia formatki	210
Obsługa zdarzeń formularza	212
Modyfikacja możliwości istniejących komponentów	213
Tworzenie i instalacja nowego komponentu	220

Usuwanie komponentu	222
Instalowanie kilku komponentów	223
Obsługa wyjątków	224
Podsumowanie	235
Rozdział 6. Wielowątkowość	237
Wstęp	237
Klasa TThread	240
Funkcje oczekujące	246
Semafor	247
Sekcja krytyczna	252
Priorytet wątku	253
Mutex	254
Podsumowanie	256
Rozdział 7. Biblioteki DLL	257
Wstęp	257
Budujemy pierwszą bibliotekę DLL	258
Wykorzystanie kodu biblioteki DLL	261
Ładowanie statyczne	261
Ładowanie dynamiczne	263
Formularz w bibliotece DLL	265
Eksportowanie klas?	267
Podsumowanie	270
Rozdział 8. Pliki tekstowe	271
Wstęp	271
Przetwarzamy pliki tekstowe	271
Czytanie z pliku wierszami	272
Czytanie pliku znak po znaku	274
Zapis do pliku wierszami	275
Podsumowanie	277
Rozdział 9. Strumienie plikowe	279
Wstęp	279
Klasa TFileStream	279
Zapis rekordu do strumienia plikowego	281
Odczyt rekordu ze strumienia plikowego	285
Zapis dużych porcji danych w strumieniu plikowym	286
Korzystanie z TMemoryStream	289
Podsumowanie	291
Rozdział 10. Pliki typowane	293
Wstęp	293
Utworzenie pliku	295
Otwarcie pliku	295
Zapis do pliku	296
Odczyt z pliku	297
Przeszukiwanie pliku	297
Zapis na końcu pliku	299
Podsumowanie	300
Rozdział 11. XML i DOM	301
Wstęp	301
Budowa pliku XML	302
Analiza dokumentów XML	306

Delphi a XML	306
XML jak plik INI	316
Podsumowanie	321
Rozdział 12. Planujemy bazę danych	323
Wstęp	323
Analiza problemu	323
Model bazy danych	324
Uwagi na temat implementacji	329
Podsumowanie	330
Rozdział 13. FireBird — elementy języka SQL	333
Wstęp	333
Instalacja programu FireBird	333
SQL Manager 2008	335
Rejestracja istniejącej bazy danych	335
Wykonywanie poleceń SQL	337
SQL — co to jest?	338
Baza danych	339
Tabele	341
Tworzenie tabel	344
Select	347
Złączenia	351
Klucz główny (primary key)	354
Klucz obcy (foreign key) i integralność referencyjna	355
Wartość NULL	358
Domena	359
Indeksy	360
Widoki (perspektywy)	362
Wyzwalacze i generatory	363
Procedury	365
Transakcje	366
Podsumowanie	366
Rozdział 14. ODBC i MS Access, DBF	367
Tworzymy bazę danych w MS Access	367
Tabele	367
Relacje	370
Kwerendy	371
Formularze	373
ODBC i MS Access	374
Łączymy się z MS Access poprzez ODBC	375
ODBC i XBase	377
Podsumowanie	378
Rozdział 15. InterBase	379
Wstęp	379
Instalacja serwera bazy danych	380
Pierwsze uruchomienie	380
Praca z InterBase	382
IBConsole	383
Interactive SQL	391
Backup	396
Restore	398
Użytkownicy i uprawnienia	400

Komponenty InterBase	405
Połączenie z serwerem InterBase	406
IBDatabase	406
IBTransaction	408
IBQuery	411
Wykonywanie polecenia SQL	423
Polecenia SQL z parametrami	423
OnGetText, OnSetText, OnValidate	426
IBTable	429
IBStoredProc	431
Monitorowanie bazy danych InterBase	433
Usunięcie instalacji serwera InterBase	433
Podsumowanie	433
Rozdział 16. MySQL i dbExpress	435
Wstęp	435
Instalacja MySQL	435
Zmiana hasła administratora	439
Uzyskiwanie podstawowych informacji	440
Tworzenie bazy danych	441
Baza danych a polskie litery	442
Tworzenie nowego użytkownika	442
Minimum uprawnień	443
Usuwanie bazy danych	443
Tworzenie tabel	444
dbExpress	446
SQLConnection	447
SQLDataSet	449
Transakcje	461
ClientDataSet	465
Komunikacja dwukierunkowa	471
Informacje na temat bazy danych	474
SQLMonitor	475
Podsumowanie	476
Rozdział 17. MySQL i Zeos	477
Instalacja komponentów Zeos	477
Wykorzystanie komponentów Zeos	478
Podsumowanie	480
Rozdział 18. SQL Server 2005 i dbGo	481
Wstęp	481
Ograniczenia wersji Express	482
Instalacja serwera bazy danych	482
Instalacja SQL Server Management Studio Express	483
Praca z serwerem bazy danych	483
Tworzymy bazę danych	484
Polskie znaki	487
Tworzymy tabele	488
Komponenty z zakładki dbGo	490
ADOConnection	490
ADOCommand	494
ADOTable, ADOQuery, ADOSToredProc	496
ADODataset	497
ADO i transakcje	500
Motor JET	501

Połączenie z plikiem Excel	501
Połączenie z plikiem tekstowym	505
Podsumowanie	506
Rozdział 19. Rave Reports — drukujemy	507
Wstęp	507
Zbieramy dane	507
Drukujemy	509
Podsumowanie	512
Rozdział 20. Interfejs bazodanowy	513
Wstęp	513
Abstrakcja rekordu tabeli	514
Abstrakcja tabeli bazy danych	519
Wykorzystywanie interfejsu bazodanowego	529
Podsumowanie	537
Rozdział 21. BDE	539
Wstęp	539
Database	542
Query	544
Table	545
Filtrowanie i lokalizowanie rekordów	556
UpdateSQL	561
StoredProc	565
Podsumowanie	567
Rozdział 22. Logiczne blokowanie rekordu	569
Wstęp	569
Logika blokowania rekordu	570
Implementacja blokady	570
Podsumowanie	574
Rozdział 23. Instalowanie programów — InnoSetup	575
Wstęp	575
Instalacja programu InnoSetup	575
Przygotowujemy pliki	576
Tworzymy skrypt instalacyjny	576
Podsumowanie	583
Zakończenie	585
Skorowidz	587

Rozdział 11.

XML i DOM

Wstęp

XML to skrót od angielskiej nazwy *Extensible Markup Language*, co można przetłumaczyć na rozszerzalny język znaczników. Zapewne większość czytelników miała już okazję zetknąć się z plikami w formacie *XML*. Użycie słowa „format” nie jest tutaj przypadkowe. Pliki tego typu posiadają swój własny format, a ściślej rzecz biorąc, są zbudowane między innymi ze znaczników (ang. *tags*). Z pewnością wiele osób ze znacznikami również się spotkało, przykładowo niezwykle popularny format plików HTML oparty jest o znaczniki. Skoro już użyłem takiego porównania, śpieszę nadmienić, że znaczniki HTML określają wygląd, a znaczniki XML —znaczenie. Poniżej przedstawiam przykładową zawartość pewnego pliku w formacie *XML*.

```
<?xml version="1.0" encoding="UTF-8"?>
<HelpError>
<html>
<head>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8"/>
<title>parsing error</title>
<link type="text/css" href="../../../jbdocs.css" rel="stylesheet"/>
</head>
<body>
<a name="_top_"></a>
<a name="cannotload"></a>
<h1>Can not parse file</h1>
URL:<MessageHolder/>
<p>Can not load or parse the file </p>
<MessageHolder/>
</body>
</html>
</HelpError>
```

Budowa pliku XML

W dokumentach XML najczęściej spotkamy się ze znacznikami otwierającymi, na przykład:

```
<html>
<title>
```

oraz zamykającymi, przy czym dla wymienionych powyżej znacznik zamykający ma odpowiednio postać:

```
</html>
</title>
```

Widząc wiersz o treści:

```
<title>parsing error</title>
```

będziemy mówili, że występuje tutaj element (węzeł) o nazwie `title` oraz wartości `parsing error`.

Spotykamy się również ze znacznikami pustymi w postaci:

```
<nazwa/>
```

Element pusty rozpoznamy po tym, że **nie posiada** odpowiadającej mu pary znaczników otwierającego i zamykającego, a znak `/` znajduje się za nazwą znacznika, a nie przed nią. Można więc powiedzieć, że znacznik ten jest jakby samodzielnym węzłem i wartością jednocześnie. Istotne z punktu widzenia analizy pliku o formacie *XML* jest poszukiwanie odpowiednich par znaczników otwierających i zamykających oraz treści zawartej pomiędzy nimi. Wobec tego łatwo zbudować plik oparty o znaczniki, na przykład taki, jak widoczny na listingu poniżej.

```
Plik rodzina.xml
<?xml version = "1.0" encoding = "Windows-1250"?>
<rodzina>
  <ojciec>
    <pali>tak</pali>
    <pracuje>tak</pracuje>
    <uczy_sie>nie</uczy_sie>
    <posiada_prawo_jazdy>tak</posiada_prawo_jazdy>
    <imie>Piotr</imie>
  </ojciec>

  <matka>
    <pali>nie</pali>
    <pracuje>tak</pracuje>
    <uczy_sie>nie</uczy_sie>
    <posiada_prawo_jazdy>nie</posiada_prawo_jazdy>
    <imie>Barbara</imie>
  </matka>
```

```

<syn>
  <pali>nie</pali>
  <pracuje>nie</pracuje>
  <uczy_sie>nie</uczy_sie>
  <posiada_prawo_jazdy>nie</posiada_prawo_jazdy>
  <imie>Grzegorz</imie>
</syn>
</rodzina>

```

Dla zwiększenia czytelności w przykładzie zastosowano wcięcia. Podobnie jak w HTML-u, brak wcięć nie ma wpływu na znaczenie poszczególnych znaczników. Łatwo rozróżnić, że mamy do czynienia z węzłami, takimi jak rodzina, ojciec, matka i syn. W skład węzła rodziny wchodzi poszczególni jej członkowie. Wśród członków rodziny rozróżniamy węzły dla ojca, matki i syna. Dodatkowo została wykorzystana konstrukcja z atrybutami — dla każdej osoby umieszczono informację o tym, czy osoba ta pali papierosy, pracuje, uczy się, czy posiada prawo jazdy oraz jakie ma imię. Atrybuty mogą przyjmować dowolne wartości. Już teraz można zauważyć, że XML — umożliwiając tworzenie własnych znaczników — staje się nośnikiem danych. W ten sposób dochodzimy do cechy rozszerzalności wymienionej w pełnej nazwie. Rozszerzalność ta sprawia również, że XML staje się metajęzykiem, czyli językiem do tworzenia języków. Skoro panuje tutaj taka — wydawałoby się — dowolność, z pewnością muszą istnieć jakieś reguły, według których należy konstruować pliki *XML*. I tak oczywiście jest — gdy prześlemy komuś nasz plik *XML*, wypadałoby go poinformować o sposobie interpretacji poszczególnych znaczników. Ale po kolei. Oto kilka zasad (reguł), jakie należy stosować przy budowie plików *XML*.

1. Wielkość liter

Wielkość liter w nazwach znaczników jest istotna. Krótko mówiąc, znacznik:

```
<rodzina>
```

powinien być zamknięty znacznikiem:

```
</rodzina>
```

2. Ignorowane znaki

Domyślnie znaki, takie jak spacje, tabulatory (wcięcia), znaki końca linii, są ignorowane. Znaczy to ni mniej, ni więcej tylko tyle, że tekst możemy dość swobodnie formatować według własnych potrzeb, kierując się względami czytelności i estetyki.

3. Możliwość stosowania komentarzy

Istnieje możliwość umieszczania komentarzy. Do tego celu służą znaczniki:

```
<!-- oraz -->.
```

Oto przykład:

```
<!-- to jest komentarz -->
```

4. Parowanie znaczników

Znaczniki — o ile nie jest to znacznik pusty — powinny być otwierane i zamykane. Wykorzystujemy je na takich samych zasadach, jakie stosujemy dla sekcji `begin`, `end` lub `{, }` (początek i koniec bloku) w programowaniu. Oznacza to również, że znaczniki zamykające muszą wystąpić w odpowiedniej kolejności. Dla wiersza o treści:

```
<znacznik 1> <znacznik 2> <znacznik 3> Przykładowa treść
```

zamykamy znaczniki w sposób zaprezentowany poniżej.

Tak jest dobrze:

```
<znacznik 1> <znacznik 2> <znacznik 3> Przykładowa treść </znacznik 3>  
↳</znacznik 2> </znacznik 1>
```

A tak błędnie:

```
<znacznik 1> <znacznik 2> <znacznik 3> Przykładowa treść </znacznik 1>  
↳</znacznik 2> </znacznik 3>
```

5. Obecność elementu głównego

Te osoby, które pisały już skrypty HTML, pamiętają, że głównym znacznikiem dokumentu jest znacznik:

```
<html> </html>
```

W jednym z poprzednich przykładów była to para znaczników:

```
<rodzina> </rodzina>
```

Krótko mówiąc, w pliku *XML* wyróżniamy jeden główny element, który zawiera wszystkie powstałe elementy. Element taki może być tylko jeden.

6. Znaki zastrzeżone

Zapewne zastanawiasz się, w jaki sposób użyć na przykład znaków `<` lub `>` wewnątrz znaczników, skoro stanowią one istotną część składni znacznika. Okazuje się, że:

- ◆ znak (ampersand) `&` zastępujemy przez `&`;
- ◆ znak `<` zastępujemy przez `<`;
- ◆ znak `>` zastępujemy przez `>`.

Oto przykłady.

Dla znaku `&`:

```
<nazwa firmy> Lee & Son </nazwa firmy>
```

gdy chodzi o treść `Lee & Son`.

Dla znaku `<`:

```
<jeśli mniejszy> if(a &lt; b) </jeśli mniejszy>
```

gdy chodzi o treść `if (a < b)`.

Dla znaku >:

```
<jeśli większy> if(a > b) </jeśli większy>
```

gdy chodzi o treść if (a > b).

Oczywiście, znaki te bez żadnych problemów możemy stosować w sekcji komentarza.

7. Stosowanie sekcji CDATA

W pliku XML mogą znaleźć się dane binarne, które również zawierają znaki zastrzeżone. Stosujemy wówczas znaczniki `<![CDATA[` oraz `]]>`. Oto przykład:

```
<![CDATA[ Lee & Son, a id(a < b), może if(b > c) ]]>
```

8. Stosowanie atrybutów

Dla pary znaczników może wystąpić dowolna ilość atrybutów — tak jak w przykładzie widocznym niżej.

```
<ojciec>
  <pali>tak</pali>
  <pracuje>tak</pracuje>
  <uczy_sie>nie</uczy_sie>
  <posiada_prawo_jazdy>tak</posiada_prawo_jazdy>
  <imie>Piotr</imie>
</ojciec>
```

9. Nagłówek dokumentu XML

Zaleca się rozpoczynanie dokumentu XML od nagłówka (ang. *header*) zawierającego informację, że jest to dokument XML, oraz numer wersji czy strony kodowej, na przykład:

```
<?xml version="1.0" encoding = "Windows-1250"?>
```

10. Deklaracja typu dokumentu

Jeżeli chcemy podkreślić fakt występowania typu dokumentu, należy to uczynić przed jego pierwszym elementem. Deklaracja typu definiuje element główny dokumentu oraz plik *dtd* zawierający definicję dokumentu. Plik XML, ze względu na swoją konstrukcję, jest analizowany przez specjalny program interpretujący (parser), który w informacji umieszczonej w pliku *dtd* sprawdza, czy dokument jest zbudowany według reguł określonych w tymże pliku. Poniżej umieszczono stosowny przykład.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE propertylist SYSTEM "http://www.sigames.com/dtds/propertylist.dtd">
<properties version="1.0">
  <comment>image borders</comment>
  <integer id="brd1" value="2"/>
  <integer id="brdt" value="2"/>
  <integer id="brdr" value="2"/>
  <integer id="brdb" value="2"/>
</properties>
```

Oczywiście, aby powyższy przykład został prawidłowo zinterpretowany, konieczny jest dostęp do wymienionego tutaj pliku <http://www.sigames.com/dtds/propertylist.dtd>.

Analiza dokumentów XML

Istnieją dwie popularne techniki przetwarzania dokumentów XML:

SAX — *Simple API for XML* — prosty interfejs API dla XML

oraz

DOM — *Document Object Model* — obiektowy model dokumentu.

Warto wspomnieć o powstałym niedawno formacie *Open XML* utworzonym przez firmę Microsoft. Format ten z założenia ma służyć głównie do obsługi pakietu MS Office (stąd również nazwa *Open Office XML*). Zainteresowanych tym kierunkiem rozwoju odsyłam do stron poświęconych temu formatowi, na przykład: <http://www.microsoft.com/poland/developer/openxml/default.msp>.

Dalej skoncentruję się na opisie interfejsu DOM.

Obiektowy model dokumentu znany pod nazwą DOM oferuje mechanizmy, które umożliwiają dostęp do elementów dokumentu XML w oparciu o strukturę, która odwzorowuje strukturę tego dokumentu. Struktura ma w tej interpretacji budowę drzewiastą. Po wczytaniu dokumentu XML element główny zostaje ustawiony jako korzeń drzewa, a kolejne węzły stają się gałęziami (elementami) drzewa.

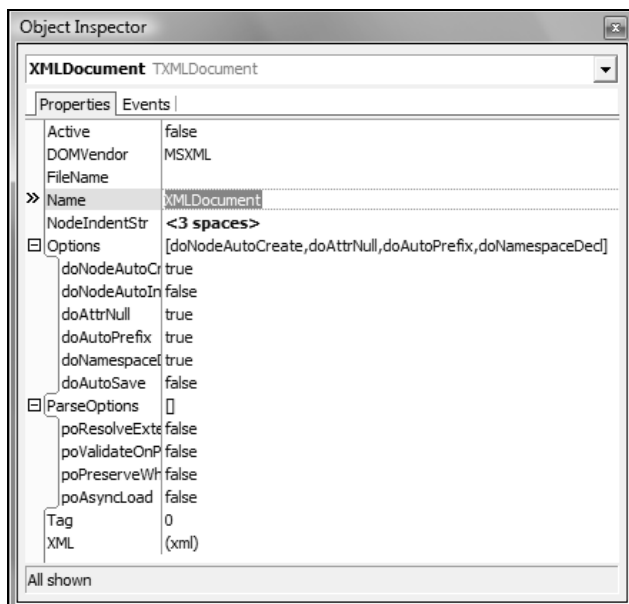
Z grubsza rzecz biorąc, analiza dokumentu XML polega na odczytywaniu kolejnych elementów (węzłów) drzewa i skojarzonych z nimi wartości. Aby z kolei zmodyfikować dokument, trzeba wprowadzić zmianę w ramach węzła drzewa, usunąć węzeł lub dodać nowy. Interfejs DOM pozwala na pominięcie nieistotnych fragmentów dokumentu XML, a w razie potrzeby — na szybki powrót do tych fragmentów. W ramach systemu Windows mamy do dyspozycji wersję DOM w postaci MSXML firmy Microsoft (wykorzystywanej między innymi przez Internet Explorer).

Delphi a XML

Środowisko Delphi na zakładce *Internet (Tool Palette/Internet)* oferuje komponent o nazwie *TXMLDocument*. Na rysunku 11.1 widzimy właściwości tego komponentu.

Komponent *XMLDocument* stanowi obudowę dla interfejsu DOM, a jego głównym zadaniem jest tworzenie w pamięci odpowiedniej struktury dokumentu XML wraz z jego weryfikacją. Aby wyświetlić całą zawartość pliku XML w komponencie *Memo*, wystarczy kilka instrukcji.

Rysunek 11.1.
Okno *Object Inspector*
— właściwości
komponentu
XMLDocument



Przykład w XML\XMLBasic01

```

procedure TMainForm.btnLoad1Click(Sender: TObject);
var
  fn : String;

begin
  Memo1.Clear();

  fn := '';
  fn := ExtractFilePath(Application.ExeName) + 'customers.xml';

  XMLDocument.FileName := fn;
  XMLDocument.Active := True;
  Memo1.Lines.Add(XMLDocument.XML.Text);
  XMLDocument.Active := False;
end;

```

Jak widać na przykładzie, należy wskazać nazwę pliku *XML* (zmienna *fn*), sprawić, by stał się aktywny, pobrać jego zawartość, korzystając z właściwości komponentu *XMLDocument* o nazwie *XML*, a na koniec wstawić uzyskaną zawartość do *Memo* oraz zamknąć komponent *XMLDocument*. Oczywiście, takie działanie nie zawsze będzie nam na rękę. Plik *XML* może być zbyt duży, może posiadać określoną strukturę, w ramach której będzie nas interesować wybrany fragment tego pliku. Aby uzyskać efekt w postaci fragmentu pliku *XML*, wykorzystamy następujące cechy komponentu *XMLDocument*:

- ♦ nazwę węzła *NodeName*,
- ♦ typ węzła *NodeType*,
- ♦ wartość węzła *NodeValue*,
- ♦ atrybuty *AttributeNodes*.

NodeType jest typem wyliczeniowym, zadeklarowanym jako:

```
TNodeType = (
  ntReserved,
  ntElement,
  ntAttribute,
  ntText,
  ntCDATA,
  ntEntityRef,
  ntEntity,
  ntProcessingInstr,
  ntComment,
  ntDocument,
  ntDocType,
  ntDocFragment,
  ntNotation);
```

NodeType identyfikuje typ węzła w dokumencie XML. Poniższe zestawienie demonstruje listę możliwych wartości.

ntReserved	Nie używane.
ntElement	Węzeł reprezentuje element. Element reprezentuje zwykły znacznik, który posiada węzły-dzieci. Należy zauważyć, że czasami węzły-dzieci nie są widoczne, gdy używamy IXMLNode. Na przykład węzły-dzieci typu ntText są typowo ukryte przez IXMLNode i pojawiają się tylko jako wartość właściwości Text. Węzły-dzieci węzła elementu mogą przyjmować następujące typy: ntElement, ntText, ntCDATA, ntEntityRef, ntProcessingInstr, i ntComment. Elementy-węzły mogą również posiadać atrybuty (ntAttribute). Mogą być dzieckiem węzła typu ntDocument, ntDocFragment, ntEntityRef i ntElement
ntReserved	Nie używane.
ntAttribute	Węzeł reprezentuje atrybut elementu. Nie jest dzieckiem innego węzła, ale jego wartość może być dostępna poprzez użycie właściwości Attribute węzła elementu dla interfejsu IXMLNode. Węzeł atrybutu może mieć węzeł-dziecko typu ntText i ntEntityRef.
ntText	Węzeł reprezentuje tekst zawierający znacznik. Węzeł tego typu nie może mieć innych węzłów-dzieci, ale może się pojawić jako węzeł-dziecko węzła typu ntAttribute, ntDocFragment, ntElement lub ntEntityRef. Węzeł ntCDATA reprezentuje sekcję CDATA w dokumencie XML. Sekcja CDATA identyfikuje bloki tekstu. Węzeł ntCDATA nie może mieć węzłów-dzieci. Może się pojawiać jako węzły-dzieci dla ntDocFragment, ntEntityRef lub węzła ntElement.

ntEntityRef	Węzeł reprezentuje referencję do jednostki w dokumencie XML. Może być dowolnym typem jednostki, zawierającym referencję znakowej jednostki. Dzieci-jednostki reprezentowane przez węzły mogą być następującego typu: ntElement, ntProcessingInstr, ntComment, ntText, ntCDATA i ntEntityRef. Może pojawiać się jako dziecko węzłów typu ntAttribute, ntDocFragment, ntElement lub ntEntityRef.
ntEntity	Węzeł reprezentuje rozszerzoną jednostkę. Węzły-jednostki mogą mieć węzły-dzieci, które reprezentują rozszerzone jednostki (na przykład ntText, ntEntityRef). Węzły te pojawiają się wyłącznie jako dzieci węzła ntDocType.
ntProcessingInstr	Węzeł reprezentuje instrukcję przetwarzania (ang. <i>Processing instruction</i>) PI z dokumentu XML. Węzeł PI nie może mieć żadnych węzłów-dzieci, ale może pojawiać się jako dziecko węzła typu ntDocument, ntDocFragment, ntElement lub ntEntityRef.
ntComment	Węzeł reprezentuje komentarz w dokumencie XML. Węzeł tego typu nie może posiadać węzłów potomnych (węzłów-dzieci). Pojawia się jako węzeł-dziecko dla typów węzłów ntDocument, ntDocFragment, ntElement lub ntEntityRef.
ntDocument	Węzeł reprezentuje obiekt dokumentu, który jest korzeniem (ang. <i>root</i>) całego dokumentu XML. Węzeł ntDocument posiada pojedynczy węzeł ntElement jako węzeł potomny. Dodatkowo węzeł taki może posiadać węzły typu ntProcessingInstr, ntComment i ntDocType. Ponieważ węzeł ten jest korzeniem dokumentu XML, nigdy nie pojawia się jako węzeł potomny.
ntDocType	Węzeł reprezentuje deklarację typu dokumentu wyróżnionego przez znacznik <code><!DOCTYPE ></code> .
ntDocFragment	Węzeł reprezentuje fragment dokumentu. Kojarzy węzeł lub poddrzewo z dokumentem niezawartym aktualnie w dokumencie. Może mieć potomne węzły typu ntElement, ntProcessingInstr, ntComment, ntText, ntCDATA oraz ntEntityRef. Nie pojawia się nigdy jako węzeł potomny innego węzła.
ntNotation	Węzeł reprezentuje zapis w deklaracji typu dokumentu. Zawsze pojawia się jako dziecko węzła ntDocType i sam nigdy nie posiada węzłów potomnych.

Wiemy już, w jaki sposób wczytać zawartość pliku *XML* do komponentu Memo. Następny fragment kodu prezentuje uzyskanie efektu identycznego z efektem uzyskanym przed chwilą:

```
procedure TMainForm.btnLoad2Click(Sender: TObject);
var
  fn      : String;
  element : IXMLNode;

begin
  Memo2.Clear();
  Memo2.Visible := False;

  fn := ExtractFilePath(Application.ExeName) + 'customers.xml';

  XMLDocument.FileName := fn;
  XMLDocument.Active := True;

  element := XMLDocument.DocumentElement;
  Narysuj(element);

  XMLDocument.Active := False;
  Memo2.Visible      := True;
end;
```

Tym razem jednak do umieszczenia zawartości pliku *XML* w Memo2 wykorzystamy metodę Narysuj:

```
procedure TMainForm.Narysuj(wezel: IXMLNode);
var
  nazwa_wezla : String;
  lp           : Integer;

begin
  if wezel.NodeType <> ntElement then
  begin
    exit;
  end;

  nazwa_wezla := wezel.NodeName;
  if wezel.IsTextElement then
  begin
    nazwa_wezla := nazwa_wezla + ' = ' + wezel.NodeValue;
  end;

  if wezel.HasChildNodes then
  begin
    for lp := 0 to wezel.ChildNodes.Count-1 do
    begin
      Narysuj(wezel.ChildNodes.Nodes[lp]);
    end;
  end;

  Memo2.Lines.Add(nazwa_wezla);
end;
```

Jest to typowa procedura wykorzystująca rekurencję. Szukając węzła, wędrujemy po zawartości każdego węzła. Niestety, oba podane sposoby niewiele dają, poza zwykłym wrzuceniem zawartości pliku *XML* do komponentu Memo. Nie widać na ekranie struktury pliku *XML*, jego węzłów ani elementów. Kolejna procedura pomoże rozwiązać ten problem. Tym razem do wizualizacji zawartości pliku *XML* wykorzystamy komponent *TreeView*, który w znacznym stopniu ułatwi prezentację budowy wewnętrznej pliku *XML*. Do wskazania pliku *XML* przyda się również komponent *OpenDialog*:

```
procedure TMainForm.Start();
begin
    OpenDialog.InitialDir := ExtractFilePath(Application.ExeName);

    if OpenDialog.Execute() then
    begin
        TreeView.Items.Clear();

        XMLDocument.LoadFromFile(OpenDialog.FileName);

        DoTreeView(XMLDocument.DocumentElement, Nil);

        TreeView.FullExpand();
    end;
end;
```

Zadanie umieszczenia zawartości pliku *XML* w komponencie *TreeView* wykonuje również metoda rekurencyjna o nazwie *DoTreeView*:

```
procedure TMainForm.DoTreeView(ixmlNode: IXMLNode; TreeNode: TTreeNode);
var
    NewTreeNode : TTreeNode;
    lp           : Integer;
    NodeText     : String;
    txt          : String;

begin
    // przeskocz węzły tekstu i inne specjalne przypadki
    if ixmlNode.NodeType <> ntElement then
    begin
        exit;
    end;

    // dodaj sam węzeł
    NodeText := ixmlNode.NodeName;
    if ixmlNode.IsTextElement then
    begin
        NodeText := NodeText + ' = ' + ixmlNode.NodeValue;
    end;

    NewTreeNode := TreeView.Items.AddChild(TreeNode, NodeText);

    // dodaj atrybuty
    for lp := 0 to ixmlNode.AttributeNodes.Count-1 do
    begin
        AttrNode := ixmlNode.AttributeNodes.Nodes[lp];
        txt      := Trim(AttrNode.Text);
        txt      := ' ' + txt;
        TreeView.Items.AddChild(NewTreeNode, AttrNode.NodeName + txt);
    end;
```

```
// dodaj każdy węzeł podrzędny
if ixmlNode.HasChildNodes then
begin
  for lp := 0 to ixmlNode.ChildNodes.Count-1 do
  begin
    DoTreeView(ixmlNode.ChildNodes.Nodes[lp], NewTreeNode);
  end;
end;
end;
```

Po wskazaniu na przykład pliku *rodzina.xml* na ekranie uzyskamy widok, taki jak na rysunku 11.2.

Rysunek 11.2.

*Budowa pliku
rodzina.xml
zaprezentowana
w postaci drzewa*



Teraz dla porównania możemy uruchomić przeglądarkę, przykładowo Internet Explorer, i otworzyć ten sam plik *rodzina.xml*, aby się przekonać, jak poradzi sobie z wyświetlaniem zawartości pliku XML. Na rysunku 11.3 przedstawiam efekt wczytania wymienionego pliku do przeglądarki.

Zbudujemy teraz następny przykładowy program (projekt o nazwie *XMLBasic02*), który zademonstruje dalsze możliwości komponentu *XMLDocument*. W tym celu po utworzeniu nowego projektu kładziemy na nim dwa komponenty — *Memo* oraz *XMLDocument* — następnie wybieramy *File/New Other*, w *Item Categories* wskazujemy *XML* i wybieramy *XML Data Binding*. Pojawi się okno *XML Data Binding Wizard*, takie jak na rysunku 11.4.

W linii *Schema or XML Data File* wskazujemy dowolny plik — na przykład *rodzina.xml*. Klikamy przycisk *Next* i przechodzimy do kolejnego okna, takiego jak na rysunku 11.5.

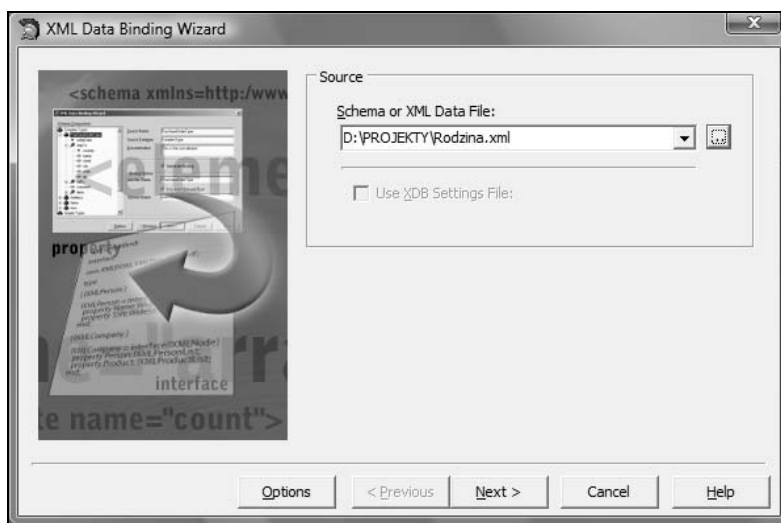
Rysunek 11.3.

Plik *rodzina.xml*
wyświetlony
w oknie przeglądarki
Internet Explorer

```
<?xml version="1.0" encoding="Windows-1250" ?>
- <rodzina>
-   <ojciec>
      <pali>tak</pali>
      <pracuje>tak</pracuje>
      <uczy_sie>nie</uczy_sie>
      <posiada_prawo_jazdy>tak</posiada_prawo_jazdy>
      <imie>Piotr</imie>
    </ojciec>
-   <matka>
      <pali>nie</pali>
      <pracuje>tak</pracuje>
      <uczy_sie>nie</uczy_sie>
      <posiada_prawo_jazdy>nie</posiada_prawo_jazdy>
      <imie>Barbara</imie>
    </matka>
-   <syn>
      <pali>nie</pali>
      <pracuje>nie</pracuje>
      <uczy_sie>nie</uczy_sie>
      <posiada_prawo_jazdy>nie</posiada_prawo_jazdy>
      <imie>Grzegorz</imie>
    </syn>
  </rodzina>
```

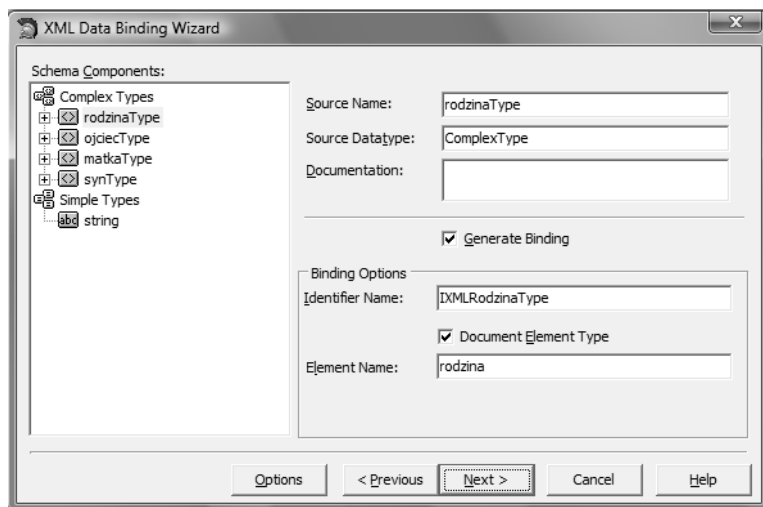
Rysunek 11.4.

Okno XML Data
Binding Wizard



Na rysunku 11.5 z lewej strony w oknie *Schema Components* widzimy strukturę pliku XML w postaci drzewa. Poszczególne gałęzie można rozwinąć, aby uzyskać bardziej szczegółowy obraz całości. Klikamy kolejne przyciski *Next*, aż do pojawienia się przycisku *Finish*. Po wybraniu przycisku *Finish* środowisko Delphi utworzy moduł *rodzina.pas*. Co właściwie uzyskaliśmy? Otóż kreator XML Data Binding Wizard utworzył zbiór interfejsów do operowania na pliku *rodzina.xml*. Proponuję teraz przynajmniej pójrzeć na zawartość pliku *rodzina.pas*. Mamy tam do dyspozycji między innymi globalne funkcje:

Rysunek 11.5.
Okno XML Data Binding Wizard
 — 2. okno



```
function Getrodzina(Doc: IXMLDocument): IXMLRodzinaType;
function Loadrodzina(const FileName: WideString): IXMLRodzinaType;
function Newrodzina: IXMLRodzinaType;
```

Przyjrzyjmy się na przykład mechanizmom dostępu do klasy związanej z ojcem:

```
IXML0ojciecType = interface(IXMLNode)
  ['{D0BEC75B-4A30-4DBB-BF9B-DA03C07A4A4A}']
  { Property Accessors }
  function Get_Pali: WideString;
  function Get_Pracuje: WideString;
  function Get_Uczy_sie: WideString;
  function Get_Posiada_prawo_jazdy: WideString;
  function Get_Imie: WideString;
  procedure Set_Pali(Value: WideString);
  procedure Set_Pracuje(Value: WideString);
  procedure Set_Uczy_sie(Value: WideString);
  procedure Set_Posiada_prawo_jazdy(Value: WideString);
  procedure Set_Imie(Value: WideString);
  { Methods & Properties }
  property Pali: WideString read Get_Pali write Set_Pali;
  property Pracuje: WideString read Get_Pracuje write Set_Pracuje;
  property Uczy_sie: WideString read Get_Uczy_sie write Set_Uczy_sie;
  property Posiada_prawo_jazdy: WideString read Get_Posiada_prawo_jazdy write
    Set_Posiada_prawo_jazdy;
  property Imie: WideString read Get_Imie write Set_Imie;
end;
```

W pliku *rodzina.xml* węzeł ojciec wygląda następująco:

```
<ojciec>
  <pali>tak</pali>
  <pracuje>tak</pracuje>
  <uczy_sie>nie</uczy_sie>
  <posiada_prawo_jazdy>tak</posiada_prawo_jazdy>
  <imie>Piotr</imie>
</ojciec>
```

Spróbujemy teraz wykorzystać oferowane mechanizmy do wyciągnięcia z pliku informacji wyłącznie z tego węzła. Czyścimy zawartość bufora komponentu `XMLDocument`:

```
XMLDocument.XML.Clear();
```

Przygotowujemy zmienne do przechowywania informacji o rodzinie i ojcu:

```
var  
    rodzina : IXMLRodzinaType;  
    ojciec   : IXMLOjciecType;
```

Wczytujemy zawartość węzła `rodzina`:

```
rodzina := Getrodzina(XMLDocument);
```

oraz umieszczamy ją w buforze komponentu `XMLDocument`:

```
XMLDocument.Active := True;
```

Pobieramy dane związane z ojcem:

```
ojciec := rodzina.Get_ojciec();
```

Teraz możemy wyciągnąć na przykład tylko informację w postaci jego imienia:

```
var kom : String;  
kom := ojciec.Get_imie();  
kom := 'Ojciec imię: ' + kom;  
Memo1.Lines.Add(kom);
```

Aby natomiast zmienić wartość elementu imię, wywołujemy:

```
ojciec.Set_imie('Wacek');
```

Oto cała procedura wykonująca wczytanie węzła `ojciec` oraz dokonująca zmiany wszystkich elementów na inne.

Przykład w XML\XMLBasic02

```
procedure TMainForm.btnSTARTClick(Sender: TObject);  
var  
    rodzina : IXMLRodzinaType;  
    ojciec   : IXMLOjciecType;  
  
    kom      : String;  
  
begin  
    XMLDocument.XML.Clear();  
  
    rodzina := Getrodzina(XMLDocument);  
    XMLDocument.Active := True;  
  
    ojciec := rodzina.Get_ojciec();  
  
    kom := '';
```

```

kom := ojciec.Get_imie();
kom := 'Ojciec imie: ' + kom;
Memo1.Lines.Add(kom);

kom := ojciec.Get_pali();
kom := 'Ojciec pali: ' + kom;
Memo1.Lines.Add(kom);

kom := ojciec.Get_posiada_prawo_jazdy();
kom := 'Ojciec posiada prawo jazdy: ' + kom;
Memo1.Lines.Add(kom);

kom := ojciec.Get_pracuje();
kom := 'Ojciec pracuje: ' + kom;
Memo1.Lines.Add(kom);

kom := ojciec.Get_uczy_sie();
kom := 'Ojciec uczy się: ' + kom;
Memo1.Lines.Add(kom);

// zmieniamy zawartość
ojciec.Set_imie('Wacek');
ojciec.Set_pali('Nie');
ojciec.Set_posiada_prawo_jazdy('Nie');
ojciec.Set_pracuje('Nie');
ojciec.Set_uczy_sie('Nie');

Memo1.Lines.Add(' ');

Memo2.Lines.Add('----- 2 -----');
Memo2.Lines.Add(ojciec.GetXML());
end;
```

XML jak plik INI

Przykład w XML\XML_AS_INI

Na koniec rozdziału związanego z przetwarzaniem pliku *XML* utworzymy prosty, mały program umożliwiający traktowanie pliku *XML* jak pliku *INI* (projekt *XML_AS_INI*). W tym celu zdefiniujemy następującą klasę:

```

MXML = class(TObject)
public
    XMLFileName : String;
    Wiersze      : TStringList;

    procedure ReadSections(var Lista : TStringList);
    function  SectionIndex(const Section : String) : Integer;
    procedure DeleteKey(const Section : String; const Name : String);
    function  ValueExist(const Section : String; const Name : String) : Boolean;
    procedure WriteString(const Section : String; const Name : String; const Value :
↳String);
```

```

procedure AddSection(const Section : String);
function SectionExist(const Section : String) : Boolean;
procedure EraseSection(const Section : String);

constructor Create(const FileName : String); virtual;
destructor Destroy(); override;
end;
```

Zmienna Wiersze typu TStringList posłuży do przechowywania zawartości pliku *XML*. Będziemy mogli sprawdzić, czy sekcja istnieje, dodać ją i usunąć. A tak należy zrealizować podstawowe operacje.

Na początek tworzymy pusty plik:

```

procedure TMainForm.btnPUSTY_PLIKClick(Sender: TObject);
var
    ini : MXML;

begin
    fn := ExtractFilePath(Application.ExeName) + 'nowy.xml';

    if FileExists(fn) then
        begin
            DeleteFile(fn);
        end;

    try
        ini := MXML.Create(fn);
    finally
        FreeAndNil(ini);
    end;
    // pokaż zawartość w Memo
    Wczytaj();
end;
```

Tworzeniem pustego pliku *XML* zajmuje się konstruktor klasy MXML:

```

constructor MXML.Create(const FileName : String);
begin
    XMLFileName := FileName;
    Wiersze := TStringList.Create();

    if FileExists(FileName) then
        begin
            Wiersze.LoadFromFile(XMLFileName);
        end
    else begin
        Wiersze.Add('<?xml version = "1.0" encoding = "Windows-1250"?>');
        Wiersze.Add('<xmlini>'); // root
        Wiersze.Add('<info>');
        Wiersze.Add('<program>' + Version + '</program>');
        Wiersze.Add('</info>');
        Wiersze.Add('</xmlini>');

        Wiersze.SaveToFile(XMLFileName);
    end;
```

```

// jeżeli plik był pusty
if Wiersze.Count = 0 then
begin
  Wiersze.Add('<?xml version = "1.0" encoding = "Windows-1250"?>');
  Wiersze.Add('<xmlini>');
  Wiersze.Add('<info>');
  Wiersze.Add('<program>' + Version + '</program>');
  Wiersze.Add('</info>');
  Wiersze.Add('</xmlini>');

  Wiersze.SaveToFile(XMLFileName);
end;
end;

```

Po utworzeniu plik *nowy.xml* ma postać zaprezentowaną poniżej.

```

<?xml version = "1.0" encoding = "Windows-1250"?>
<xmlini>
  <info>
    <program>XMLINI 1.0</program>
  </info>
</xmlini>

```

Występuje w nim węzeł główny *xmlini*:

```
<xmlini> </xmlini>
```

oraz węzły *info* i *program*.

Dodajemy do pliku przykładową sekcję adres:

```

procedure TMainForm.btnADD_ADRESClick(Sender: TObject);
var
  ini : MXML;

begin
  Memo.Lines.Clear();

  fn := ExtractFilePath(Application.ExeName) + 'nowy.xml';

  if not FileExists(fn) then
  begin
    ShowMessage('Brak pliku: ' + fn);
    exit;
  end;

  try
    ini := MXML.Create(fn);

    if not ini.SectionExist('adres') then
    begin
      ini.AddSection('adres');
    end;
  end;
end;

```

```

        ini.WriteString('adres', 'ulica' , '3 Maja 132');
        ini.WriteString('adres', 'miasto', 'Pszczyna');
        ini.WriteString('adres', 'kodp' , '43-200');
    end;
finally
    FreeAndNil(ini);
end;

Memo.Lines.Clear();
Wczytaj();
Memo.Visible := True;
end;
```

A procedura dodająca sekcję (w pliku *XML* — węzeł) wygląda następująco:

```

procedure MXML.AddSection(const Section : String);
var
    iLast : Integer;

begin
    iLast := Wiersze.Count-1;
    Wiersze.Delete(iLast);

    Wiersze.Add('<' + Section + '>');
    Wiersze.Add('</' + Section + '>');
    Wiersze.Add('</xmlini>');
end;
```

Konieczne jest usunięcie z niej ostatniego wiersza zawierającego znacznik zamykający dla głównego znacznika pliku *XML*. Zapis pojedynczego elementu uzyskaliśmy poprzez `WriteString`:

```

procedure MXML.WriteString(const Section : String;
                           const Name   : String;
                           const Value  : String);
var
    txt : String;
    idx : Integer;

begin
    // jeżeli sekcja ma już taki węzeł, to go usuwam
    if ValueExist(Section, Name) then
        begin
            DeleteKey(Section, Name);
        end;

    txt := '';
    txt := txt + '<' + Name + '>' + Value + '</' + Name + '>';
    idx := SectionIndex(Section);
    Wiersze.Insert(idx+1, txt);
end;
```

Sprawdzenie, czy sekcja istnieje, wykonuje funkcja:

```
function MXML.SectionExist(const Section : String) : Boolean;
begin
  if Wiersze.IndexOf('<' + Section + '>') >= 0 then
    begin
      Result := True;
    end
  else begin
    Result := False;
  end;
end;
```

Na koniec jeszcze możliwość usunięcia sekcji:

```
procedure MXML.EraseSection(const Section: String);
var
  lp      : Integer;
  start   : Integer;
  koniec  : Integer;

begin
  start := Wiersze.IndexOf('<' + Section + '>');
  koniec := Wiersze.IndexOf('</' + Section + '>');

  lp := koniec;
  while lp >= start do
    begin
      Wiersze.Delete(lp);
      Dec(lp);
    end;
  end;
```

Likwidując obiekt klasy MXML, trzeba pamiętać o zwolnieniu pamięci zajmowanej przez wiersze:

```
destructor MXML.Destroy();
begin
  if Assigned(Wiersze) then
    begin
      // zrzut aktualnej zawartości do pliku
      Wiersze.SaveToFile(XMLFileName);

      Wiersze.Clear();
      FreeAndNil(Wiersze);
    end;

  inherited;
end;
```

Mimo nieco przewrotnego traktowania pliku *XML* jak pliku *INI*, zachowuje on nadal cechy pliku *XML*, o czym można się przekonać, wczytując zawartość pliku *XML* do przeglądarki. Poniżej mamy przykładową zawartość pliku po dodaniu sekcji adres oraz dzieci.

```
<?xml version = "1.0" encoding = "Windows-1250"?>
  <xmlini>
    <info>
      <program>1.0</program>
    </info>
    <adres>
      <kodp>43-200</kodp>
      <miasto>Pszczyna</miasto>
      <ulica>3 Maja 132</ulica>
    </adres>
    <dzieci>
      <córka>Marzena</córka>
      <syn>Piotr</syn>
    </dzieci>
  </xmlini>
```

Podsumowanie

Przedstawiłem tylko bardzo wąski fragment dotyczący elementów przetwarzania plików *XML* z wykorzystaniem środowiska Delphi. Zachęcam do poznawania dalszych tajników budowy pliku *XML* i własnych eksperymentów.