Programowanie zaawansowane

13

Serializacja

Waldemar Bartyna
wbartyna@gmail.com

Wprowadzenie (1)

- W przestrzeni nazw *System.IO* znajduje się wiele typów związanych z odczytywaniem i zapisywaniem danych.
- Mogą one być wykorzystane do utrwalania danych we wskazanej lokalizacji i formacie.

- Podczas tego wykładu omówiony zostanie pokrewny temat, a mianowicie serializacja obiektów.
- Pozwala ona na zapisywanie i odczytanie stanu danego obiektu do/z dowolnego strumienia (typu dziedziczącego po *System.IO.Stream)*.

Wprowadzenie (2)

- Możliwość serializacji obiektu jest kluczowa w przypadku potrzeby skopiowania obiektu na zdalną maszynę za pomocą różnego rodzaju technologii zdalnego dostępu, takich jak:
  - Warstwa zdalnego dostępu .NET (ang. *remoting layer*),
  - XML-owe usługi sieciowe,
  - Windows Communication Foundation.

- Seirailzacja może również znaleźć zastosowanie w innych rodzajach aplikacji.
- Poznamy mechanizm jej działania, różne metody serializacji oraz sposoby jej konfigurowania.

Serializacja obiektów

Serializacja

- Termin **serializacja** oznacza proces zapisywania (i ewentualnie transformacji) stanu obiektu w postaci serii bajtów do strumienia (strumienia do pliku, strumienia do pamięci, itd.).

- Sekwencja zapisanych danych zawiera wszystkie informacje niezbędne do rekonstrukcji (**deserializacji**) danego stanu obiektu.

- Przy pomocy tej technologii, bardzo prostym staje się zapisanie bardzo dużej ilości danych (w różnych formatach) przy minimalnym zaangażowaniu programisty (mniejszym niż w przypadku używania typów w przestrzeni *System.IO*).

Przykład zastosowania serializacji (1)

- Dla przykładu załóżmy, że stworzyliśmy desktopową aplikację i chcemy zapewnić sposób pozwalający na zapisywanie preferencji każdego z jej użytkowników (kolor okna, rozmiar czcionki, itd.).
- W tym celu możemy zdefiniować odpowiednią klasę (np., *UserPrefs*), która będzie hermetyzować około 20 właściwości.

- Gdybyśmy chcieli skorzystać z typu *System.IO.BinaryWriter*, musielibyśmy **ręcznie** zapisać każde pole danych z obiektu typu *UserPrefs*.
- To samo dotyczy wczytania tych danych za pomocą typu *System.IO.BinaryReader*.

Przykład zastosowania serializacji (2)

- To oczywiście jest do zrobienia, ale używając mechanizmu serializacji możemy zaoszczędzić sobie wiele pracy.
- Wystarczy naszą klasę oznaczyć atrybutem [Serializable].

```
[Serializable]
public class UserPrefs
{
    // różne pola danych...
}
```

- Dzięki temu drobnemu zabiegowi, cały stan obiektu może być utrwalony w kilku liniach kodu (co pokazuje następny slajd).

Przykład zastosowania serializacji (3)


```
static void Main(string[] args)
{
    // Załóżmy, że w klasie UserProfile zostały
    // zdefiniowane następujące właściwości.
    UserPrefs userData = new UserPrefs();
    userData.WindowColor = "Yellow";
    userData.FontSize = "50";

    // BinaryFormatter utwala dene w binarnym formacie.
    BinaryFormatter binFormat = new BinaryFormatter();

    // Zachowanie obiektów w lokalnym pliku.
    using (Stream fStream = new FileStream("user.dat",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, userData);
    }
    Console.ReadLine();
}
```

Wewnętrzne mechanizmy

- Podczas gdy sam proces wykorzystania serializacji jest bardzo prosty, mechanizmy, który za nią odpowiadają od strony implementacji są dużo bardziej złożone.

- Na przykład, podczas utrwalenia obiektu do strumienia, wszystkie powiązane z nim dane (klasy bazowe, zagregowane obiekty, itd.) są również automatycznie serializowane.
- Dlatego też , w przypadku utrwalania obiektu klasy potomnej, wszystkie dane w łańcuchu dziedziczenia też są utrwalane.

- Zbiór powiązanych ze sobą obiektów reprezentowany jest za pomocą grafów obiektów.

Rożne formaty serializacji

- Usługi serializujące platformy .NET pozwalają na utrwalanie grafu obiektu w kilku różnych formatach.

- Przykładowy kod przedstawiony wcześniej pokazywał użycie typu *BinaryFormater*, który powoduje utrwalanie stanu obiektu w spakowanym binarnym formacie.

- Możemy również utrwalić graf obiektów w formacie koperty SOAP lub dokumentu XML za pomocą dwóch innych typów serializerów.
- Formaty te są pomocne, gdy chcemy zapewnić łatwe przenoszenie naszego obiektu między różnymi systemami operacyjnymi, językami i architekturami.

Różne strumienie

- Należy pamiętać, że graf obiektów możemy zapisywać do dowolnego strumienia (obiektu klasy dziedziczącej po *System.IO.Stream*).

- W przedstawionym przykładzie utrwaliliśmy stan obiektu w lokalnym pliku.

- Moglibyśmy równie dobrze zapisać go w określonym obszarze pamięci (za pomocą typu *MemoryStream*) lub w strumieniu sieciowym.

- To co się liczy, to fakt, że pewna sekwencja bajtów poprawnie reprezentuje stan obiektów w grafie.

Rola grafu obiektów

- Podczas serializacji obiektu, CLR uwzględnia również wszystkie powiązane obiekty w celu zapewnienia prawidłowego utrwalenia danych.

- Taki zbiór powiązanych ze sobą obiektów nazywa się **grafem obiektów**.

- Graf obiektów stanowi prosty sposób udokumentowania powiązań między obiektami niekoniecznie odwzorowując tradycyjne relacje z programowania obiektowego („*is-a*”, „*has-a*”). Jednakże pozwalają na odpowiednie modelowanie potrzebne w tym podejściu.

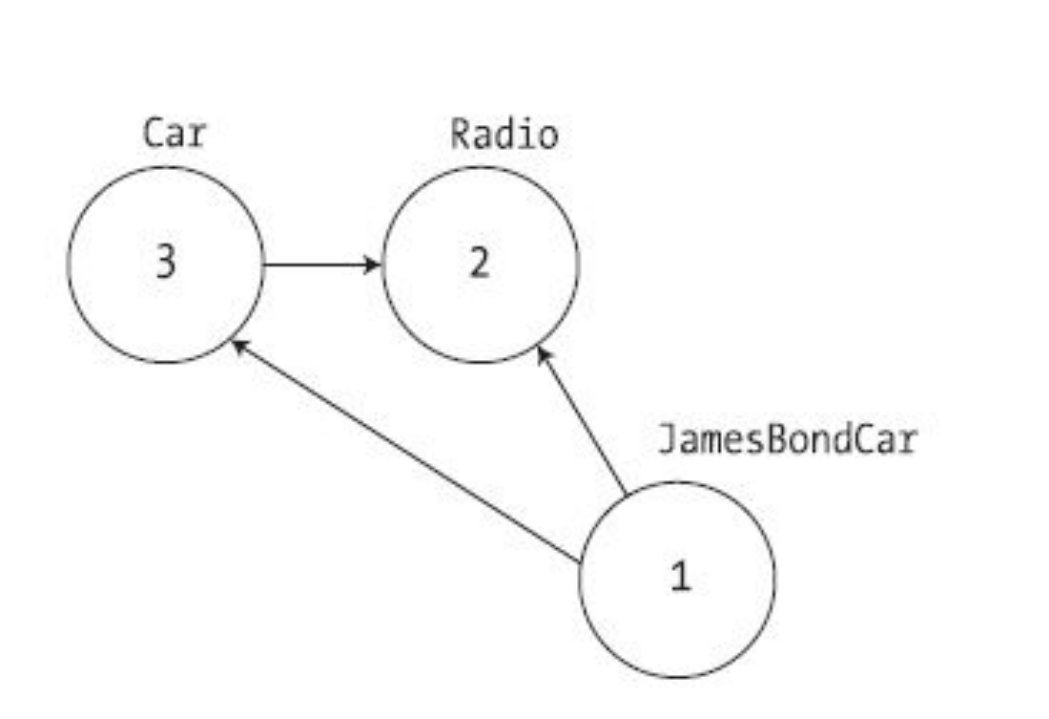
Grafy obiektów

- Do każdego obiektu w grafie obiektów przypisywana jest unikalna wartość liczbowa.
- Wartości te są generowane automatycznie i nie mają żadnego znaczenia poza grafem.
- Po przypisaniu unikalnych wartości (identyfikatorów), możliwe jest zapisanie w grafie zbioru zależności dla każdego z obiektów.

Przykład grafu obiektów (1)

- Dla przykładu załóżmy, że stworzyliśmy następujący zbiór klas:
  - bazową klasę *Car,* która posiada („*has-a*”) *Radio*,
  - potomna klasa, *JamesBondCar*, rozszerza klasę („*is-a*”) *Car*.

- Graf obiektów dla takiego zbioru klas będzie wyglądał następująco:



Przykład grafu obiektów (2)

- Klasa *JamesBosnCar* posiada referencję do klasy *Radio*, ponieważ odziedziczyła ją po klasie bazowej *Car*.

- Oczywiście, CLR nie przechowuje grafu obiektów w postaci obrazka. Relacje udokumentowane na diagramie reprezentowane są w postaci matematycznej formuły

```
[Car 3, ref 2], [Radio 2], [JamesBondCar 1, ref 3, ref 2]
```

- W momencie serializacji lub deserializacji instancji klasy *JamesBondCar*, graf obiektów zapewnia to, że obiekty klas *Radio* i *Car* również zostaną uwzględnione w tym procesie.

Automatyczne generowanie grafu obiektów

- Zaletą procesu serializacji jest to, że graf reprezentujący relacje między obiektami jest tworzony automatycznie „za kurtyną”.
- Jeżeli jednak chcemy sami wziąć udział w jego tworzeniu (dokonać pewnych modyfikacji), możemy tego dokonać poprzez konfigurację procesu serializacji za pomocą odpowiednich atrybutów i interfejsów.

- Typ *XmlSerializer* nie utrwala stanu przy pomocy grafu obiektów, jednakże typ ten również dokonuje serializacji i deserializacji powiązanych obiektów w przewidywalny sposób.

Oznaczanie obiektów do serializacji

- Aby dany obiekt był dostępny dla usług serializacji platformy .NET, wystarczy oznaczyć każdą z powiązanych klas (lub struktur) atrybutem **[Serializable]**.
- Jeżeli zdecydujemy, że dany typ posiada dane składowe, które nie powinny (lub nie mogą) brać udziału w serializacji, oznaczamy takie pola atrybutem **[NonSerialized]**.
- Może nam to pomóc wtedy, gdy w danym typie mamy dane, które nie musza być „zapamiętane”, np. pola o stałych wartościach, wartości losowe itd., a zależy nam na zmniejszeniu rozmiaru utrwalanych danych.

Przykładowe klasy (1)

- Dla przykładu stwórzmy kilka powiązanych klas.

```
[Serializable]
public class Radio
{
    public bool hasTweeters;
    public bool hasSubWoofers;
    public double[] stationPresets;

    [NonSerialized]
    public string radioID = "XF-552RR6";
}
```

- Pole *radioID* zostało oznaczone atrybutem [NonSerialized] i dlatego nie będzie utrwalone.

Przykładowe klasy (2)

- Dodajemy klasę bazową posiadająca *Radio* oraz klasę po niej dziedziczącą.

```
[Serializable]
public class Car
{
    public Radio theRadio = new Radio();
    public bool isHatchBack;
}
```

```
[Serializable]
public class JamesBondCar : Car
{
    public bool canFly;
    public bool canSubmerge;
}
```

Własności atrybutu [Serializable] (1)

- Atrybut [Serializable] nie jest odziedziczony po klasie bazowej.
- Dlatego też, w przypadku tworzenia klasy dziedziczącej po klasie oznaczonej przez [Serializable], musimy ją również oznaczyć tym atrybutem. W przeciwnym razie nie będzie mogła zostać poddana procesowi serializacji.
- Wszystkie obiekty w grafie obiektów muszą być oznaczone atrybutem [Serializable].

Własności atrybutu [Serializable] (2)

- Jeżeli spróbujemy serializować obiekty nie oznaczone tym atrybutem używając typów *BinaryFormater* i *SoapFormater*, zostanie zgłoszony wyjątek *SerializationException*.

- Ponieważ typ *XmlSerializer* nie korzysta z grafów obiektów, teoretycznie nie jest wymagane oznaczanie utrwalanych typów atrybutem [Serializable].
- Jednakże, w celu zapewnienia możliwości serializacji danego typu w dowolnym formacie, najczęściej dany typ (i typy z nim powiązane) oznacza się odpowiednim atrybutem.

Pola prywatne, publiczne i właściwości (1)

- Typy *BinaryFormater* i *SoapFormater* są zaprogramowane do utrwalania **wszystkich** serializowalnych pól danych (bez względu na to, czy są nimi pola prywatne, pola publiczne lub pola prywatne widoczne poprzez publiczne właściwości).

- Typ *XmlSerializer* serializuje wyłącznie pola publiczne i pola prywatne widoczne poprzez publiczne właściwości.
- Zwykłe pola prywatne są pomijane.

Pola prywatne, publiczne i właściwości (2)


```
[Serializable]
public class Person
{
    // Pole publiczne.
    public bool isAlive = true;

    // Pole prywatne.
    private int personAge = 21;

    // Pole prywatne i publiczna właściwość.
    private string fName = string.Empty;
    public string FirstName
    {
        get { return fName; }
        set { fName = value; }
    }
}
```

Pola prywatne, publiczne i właściwości (3)

- W przypadku, gdy obiekt klasy *Person* będziemy serializować za pomocą typów *BinaryFormater* i *SoapFormater*, zobaczymy, że pola *isAlive*, *personAge* i *fName* zostaną zapisane we wskazanym strumieniu.

- W przypadku zastosowania typu *XmlSerializer*, pole *personAge* nie zostanie zapisane (ponieważ to prywatne pole nie jest udostępniane za pomocą publicznej właściwości typu).
- Jeżeli chcemy, aby zostało ono uwzględniono musimy oznaczyć je jako publiczne lub dodać odpowiednią właściwość.

Formatery serializacji

Formatery serializacji

- Po skonfigurowaniu typów do możliwości ich serializoacji na platformie .NET poprzez oznaczenie ich niezbędnymi atrybutami, kolejnym krokiem jest wybór formatu (binarny, SOAP lub XML), w jakim powinny być utrwalone stany obiektów.
- Każda z możliwości jest reprezentowana przez jedną z klas:
  - *BinaryFormater*,
  - *SoapFormater*,
  - *XmlSerializer*.

Klasa *BinaryFormater*

- Typ *BinaryFormater* zapisuje stan obiektu do wskazanego strumienia używając formatu binarnego.
- Zdefiniowany jest w przestrzeni nazw *System.Runtime.Serialization.Formaters.Binary*, która jest częścią pakietu *mscorlib.dll*.
- Dlatego też, aby skorzystać z tego formatera wystarczy wpisać odpowiednią dyrektywę *using*.

```
using System.Runtime.Serialization.Formatters.Binary;
```

Klasa *SoapFormater*

- Typ *SospFormater* utrwala stan obiektu jako wiadomość w formacie SOAP.
- Zdefiniowany jest on w przestrzeni nazw *System.Runtime.Serialization.Formaters.Soap*.
- Funkcjonalność ta znajduje się w osobnym pakiecie.
- Dlatego też, aby skorzystać z tego typu należy dodać referencję do pakietu *System.Runtime.Serialization.Formatters.Soap.dll,* a następnie dodać do programu odpowiednią dyrektywę *using.*

```
using System.Runtime.Serialization.Formatters.Soap;
```

Klasa *XmlSerializer*

- Typ *SospSerializer* utrwala stan obiektu w postaci dokumentu XML.
- Aby użyć tego typu musimy dopisać dyrektywę *using* z przestrzenią nazw *System.Xml.Serialization* i dodać referencję do pakietu *System.Xml.dll*.
- Wszystkie projekty w Visual Studio 2015 mają automatycznie dołączoną referencję do wspomnianego pakietu. Zatem wystarczy dopisać:

```
using System.Xml.Serialization;
```

Wspólne interfejsy

- Wszystkie wymienione formatery dziedziczą bezpośrednio po klasie *System.Object*. Nie dzielą zatem wspólnego zbioru składowych związanych z serializacją.

- Jednakże, *BinaryFormater* i *SoapFormater* wspierają wspólne składowe poprzez implementację interfejsów *IFormater* i *IRemotingFormater*

- XmlSerializer nie implementuje żadnego z nich.

Interfejs *IFormater*

- Interfejs *System.Runtime.Serialization.IFormater* definiuje podstawowe metody *Serialize() i Deserialize()* oraz kilka właściwości wykorzystywanych „po cichu” przez implementujące go typy.

```
public interface IFormatter
{
    SerializationBinder Binder { get; set; }
    StreamingContext Context { get; set; }
    ISurrogateSelector SurrogateSelector { get; set; }
    object Deserialize(System.IO.Stream serializationStream);
    void Serialize(System.IO.Stream serializationStream, object graph);
}
```

Interfejs *IRemotingFormater*

- Interfejs *System.Runtime.Remoting.Messaging.IRemotingFormater* (wykorzystywany przez warstwę zdalnego dostępu .NET) przeciąża metody *Serialize()* i *Deserialize()* do postaci bardziej odpowiadającej rozproszonemu utrwalaniu.

- Interfejs ten rozszerza bardziej ogólny interfejs *IFormater*.

```
public interface IRemotingFormatter : IFormatter
{
    object Deserialize(Stream serializationStream, HeaderHandler handler);
    void Serialize(Stream serializationStream, object graph,
    Header[] headers);
}
```

Przykładowe wykorzystanie interfejsu *IFormater*

- Mimo, że raczej nie korzysta się bezpośrednio z wymienionych interfejsów, ich przykładowe zastosowanie może wynikać z bazującego na interfejsach polimorfizmu.
- Instancję typy *BinaryFormater* i *SoapFormater* możemy przypisać do referencji do interfejsu *IFormater*.

```
static void SerializeObjectGraph(IFormatter itfFormat,
                                 Stream destStream, object graph)
{
    itfFormat.Serialize(destStream, graph);
}
```

Wierność zapisywanych nazw (1)

- Najbardziej oczywistą różnicą między trzema poznanymi formaterami jest sposób, w jaki graf obiektów jest utrwalany w strumieniu (binarny, SOAP lub XML).
- Istnieje jeszcze kilka innych różnic między formaterami.

- Podczas korzystania z typu *BinaryFormater*, utrwalone zostaną nie tylko pola danych, ale również pełna kwalifikowana nazwa każdego z typów i pełna nazwa definiującego je pakietu (nazwa, wersja, klucz publiczny, i lokalizacja).
- Utrwalanie tych dodatkowych danych czynią typ *BinaryFormater* idealnym wyborem, jeżeli chcemy przenosić stan obiektu (pełną jego kopię) między maszynami pracującymi w oparciu o platformę .NET.

Wierność zapisywanych nazw (2)

- Typ *SoapFormater* utrwala informację o pakiecie w postaci XML-owych przestrzeni nazw.
- Na przykład, w przypadku serializacji klasy *Person* jako wiadomości SOAP, otrzymalibyśmy następujący wynik:

```
<a1:Person id="ref-1" xmlns:a1=
  "http://schemas.microsoft.com/clr/nsassem/SimpleSerialize/MyApp%2C%20
  Version%3D1.0.0.0%2C%20Culture%3Dneutral%2C%20PublicKeyToken%3Dnull">
  <isAlive>true</isAlive>
  <personAge>21</personAge>
  <fName id="ref-3"></fName>
</a1:Person>
```

Wierność zapisywanych nazw (3)

- Typ *XmlSerializer* nie próbuje zachowywać całkowitej zgodności typów i dlatego nie utrwala pełnej kwalifikowanej nazwy typu ani nazwy pakietu.

- Powodem tego jest otwarta natura XML-owej reprezentacji danych.

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <isAlive>true</isAlive>
  <PersonAge>21</PersonAge>
  <FirstName />
</Person>
```

Wierność zapisywanych nazw (4)

- Jeżeli chcemy utrwalić stan obiektu w postaci, która może być wykorzystana przez dowolny system operacyjny (Windows XP, Mac OS X i różne dystrybucje Linuxa), dowolną platformę (.NET, J2EE, COM, itd.) lub dowolny język programistyczny, nie musimy dołączać informacji o pełnych nazwach typów.
- Nie możemy być pewni, że wszyscy potencjalni odbiorcy będą w stanie zrozumieć specyficzne dla platformy .NET typy danych.

- Typy *SoapFormater* i *XmlSerializer* są idealnym wyborem, jeżeli chcemy zapewnić jak najszerszy dostęp do utrwalanych danych.

Serializacja za pomocą typu *BinaryFormater* (1)

- Dwie kluczowe metody to:
  - *Serialize()* – utrwalenie grafu obiektów do wskazanego strumienia jako sekwencji bajtów,
  - *Deserialize()* – skonwertowania utrwalonej sekwencji bajtów do grafu obiektów.

```
static void Main(string[] args)
{
    JamesBondCar jbc = new JamesBondCar();
    jbc.canFly = true;
    jbc.canSubmerge = false;
    jbc.theRadio.stationPresets = new double[] { 89.3, 105.1, 97.1 };
    jbc.theRadio.hasTweeters = true;

    // Przeniesienie obiektu do wskazanego pliku w binarnym formacie.
    SaveAsBinaryFormat(jbc, "CarData.dat");
    Console.ReadLine();
}
```

Serializacja za pomocą typu *BinaryFormater* (2)


```
static void SaveAsBinaryFormat(object objGraph, string fileName)
{
    BinaryFormatter binFormat = new BinaryFormatter();
    using (Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in binary format!");
}
```

- Metoda *Serialize()* odpowiedzialna jest za skomponowanie grafu obiektów i przeniesienie sekwencji bajtów do wskazanego strumienia.

Serializacja za pomocą typu *BinaryFormater* (3)

- Wynikiem użycia typu *BinaryFormater* jest następujący plik binarny przechowujący stan wskazanego obiektu:

CarData.dat | Cars.cs | Radio.cs | Program.cs* | Start Page

```
00000000  00 01 00 00 00 FF FF FF  FF 01 00 00 00 00 00 00  ................
00000010  00 0C 02 00 00 00 46 53  69 6D 70 6C 65 53 65 72  ......FSimpleSer
00000020  69 61 6C 69 7A 65 2C 20  56 65 72 73 69 6F 6E 3D  ialize, Version=
00000030  31 2E 30 2E 30 2E 30 2C  20 43 75 6C 74 75 72 65  1.0.0.0, Culture
00000040  3D 6E 65 75 74 72 61 6C  2C 20 50 75 62 6C 69 63  =neutral, Public
00000050  4B 65 79 54 6F 6B 65 6E  3D 6E 75 6C 6C 05 01 00  KeyToken=null...
00000060  00 00 1C 53 69 6D 70 6C  65 53 65 72 69 61 6C 69  ...SimpleSeriali
00000070  7A 65 2E 4A 61 6D 65 73  42 6F 6E 64 43 61 72 04  ze.JamesBondCar.
00000080  00 00 00 06 63 61 6E 46  6C 79 0B 63 61 6E 53 75  ....canFly.canSu
00000090  62 6D 65 72 67 65 08 74  68 65 52 61 64 69 6F 0B  bmerge.theRadio.
000000a0  69 73 48 61 74 63 68 42  61 63 6B 00 00 04 00 01  isHatchBack.....
000000b0  01 15 53 69 6D 70 6C 65  53 65 72 69 61 6C 69 7A  ..SimpleSerializ
000000c0  65 2E 52 61 64 69 6F 02  00 00 00 01 02 00 00 00  e.Radio.........
000000d0  01 00 09 03 00 00 00 00  05 03 00 00 00 15 53 69  ..............Si
000000e0  6D 70 6C 65 53 65 72 69  61 6C 69 7A 65 2E 52 61  mpleSerialize.Ra
000000f0  64 69 6F 03 00 00 00 0B  68 61 73 54 77 65 65 74  dio.....hasTweet
00000100  65 72 73 0D 68 61 73 53  75 62 57 6F 6F 66 65 72  ers.hasSubWoofer
00000110  73 0E 73 74 61 74 69 6F  6E 50 72 65 73 65 74 73  s.stationPresets
00000120  00 00 07 01 01 06 02 00  00 00 01 00 09 04 00 00  ................
00000130  00 0F 04 00 00 00 03 00  00 00 06 33 33 33 33 33  ...........33333
00000140  53 56 40 66 66 66 66 66  46 5A 40 66 66 66 66 66  SV@fffffFZ@fffff
00000150  46 58 40 0B                                       FX@.
```

Serializacja za pomocą typu *BinaryFormater* (4)

- Deserializacja stanu obiektu z pliku binarnego.

```
static void LoadFromBinaryFile(string fileName)
{
    BinaryFormatter binFormat = new BinaryFormatter();

    // Odczytanie obiektu z binarnego pliku.
    using (Stream fStream = File.OpenRead(fileName))
    {
        JamesBondCar carFromDisk =
            (JamesBondCar)binFormat.Deserialize(fStream);
        Console.WriteLine("Can this car fly? : {0}", carFromDisk.canFly);
    }
}
```

- Metoda *Deserialize()* zwraca obiekty ogólnej klasy *System.Obiect* , dlatego musimy użyć odpowiedniego jawnego rzutowania.

Serializacja za pomocą typu *SoapFormater* (1)

- Specyfikacja SOAP (ang. *Simple Object Access Protocol*) definiuje standardowy proces, w jaki mogą być wywoływane metody w sposób niezależny od platformy i systemu operacyjnego.
- Programowo, serializacjia przebiega identycznie, jak w przypadku typy *BinaryFormater*.

```
static void SaveAsSoapFormat(object objGraph, string fileName)
{
    // Zapisanie obiektu do pliku w formacie wiadomości SOAP.
    SoapFormatter soapFormat = new SoapFormatter();

    using (Stream fStream = new FileStream(fileName,
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        soapFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in SOAP format!");
}
```

Serializacja za pomocą typu *SoapFormater* (2)

- Wynikowy plik zawiera stan obiektu oraz relacje między jego poszczególnymi podobiektami (atrybut *#ref*).

CarData.soap* | Cars.cs | Radio.cs | Program.cs | Start Page

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:xsd="h
  <SOAP-ENV:Body>
    <a1:JamesBondCar id="ref-1" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/Si
      <canFly>true</canFly>
      <canSubmerge>false</canSubmerge>
      <theRadio href="#ref-3"/>
      <isHatchBack>false</isHatchBack>
    </a1:JamesBondCar>
    <a1:Radio id="ref-3" xmlns:a1="http://schemas.microsoft.com/clr/nsassem/SimpleSer
      <hasTweeters>true</hasTweeters>
      <hasSubWoofers>false</hasSubWoofers>
      <stationPresets href="#ref-4"/>
    </a1:Radio>
    <SOAP-ENC:Array id="ref-4" SOAP-ENC:arrayType="xsd:double[3]">
      <item>89.3</item>
      <item>105.1</item>
      <item>97.1</item>
    </SOAP-ENC:Array>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Serializacja za pomocą typu *XmlSerializer* (1)

- Typ XmlSerializer służy do zapisania publicznego stanu obiektu w postaci czystego XML-a.

```
static void SaveAsXmlFormat(object objGraph, string fileName)
{
    // Zapisanie stanu obiektu do pliku w formacie dokumentu XML.
    XmlSerializer xmlFormat = new XmlSerializer(typeof(JamesBondCar),
         new Type[] { typeof(Radio), typeof(Car) });

    using (Stream fStream = new FileStream(fileName,
    FileMode.Create, FileAccess.Write, FileShare.None))
    {
        xmlFormat.Serialize(fStream, objGraph);
    }
    Console.WriteLine("=> Saved car in XML format!");
}
```

- Serializacja ta różni się tym od wcześniejszych, że możemy podać informacje o serializowanych typach.
- Pierwszy argument konstruktora to informacje o typie, który jest korzeniem dokumenty.
- Drugi argument to tablica z informacjami (metadanymi) o pod-elementach dokumentu (typach powiązanych z głównym).

Serializacja za pomocą typu *XmlSerializer* (2)

- Wynikiem serializacji naszego obiektu jest nastepujący dokument XML.

CarData.xml | Cars.cs | Radio.cs | Program.cs | Start Page

```
<?xml version="1.0"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
              xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <theRadio>
    <hasTweeters>true</hasTweeters>
    <hasSubWoofers>false</hasSubWoofers>
    <stationPresets>
      <double>89.3</double>
      <double>105.1</double>
      <double>97.1</double>
    </stationPresets>
    <radioID>XF-552RR6</radioID>
  </theRadio>
  <isHatchBack>false</isHatchBack>
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

Kontrola sposobu generowania dokumentu XML

- *XmlSerializer* wymaga zdefiniowania dla każdej z klas biorącej udział w serializacji domyślnego konstruktora.

- Dokumenty XML powinny być zgodne ze zdefiniowanymi wcześniej schematami (XML schema, DTD).
- Domyślnie, każde z pól przekształcane jest do elementu dokumentu XML.
- Jeżeli chcemy zmienić sposób tworzenia dokumentu reprezentującego stan naszego obiektu (np. jedno z pól zdefiniować jako atrybut głównego elementu) możemy skorzystać ze zbioru atrybutów zdefiniowanych w przestrzeni nazw *System.Xml.Serialization*.

Atrybuty przestrzeni *System.Xml.Serialization*

Atrybut	Znaczenie
XmlAttribute	Ta składowa będzie utrwalona jako atrybut.
XmlElement	Ta składowa będzie utrwalona jako element.
XmlEnum	Nazwa elementu przechowującego enumerację.
XmlRoot	Decyduje o tym, jak będzie konstruowany korzeń dokumentu (przestrzeń nazw , nazwa elementu).
XmlText	Ta składowa będzie serializowana jako XML-owy text.
XmlType	Nazwa i przestrzeń nazw XML-owego typu.

Przykładowe zastosowanie atrybutów


```
<?xml version="1.0" encoding="utf-8"?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
   xmlns:xsd="http://www.w3.org/2001/XMLSchema">
...
  <canFly>true</canFly>
  <canSubmerge>false</canSubmerge>
</JamesBondCar>
```

Wynikowy dokument XML.

```
[Serializable,
XmlRoot(Namespace = "http://www.intertech.com")]
public class JamesBondCar : Car
{
     [XmlAttribute]
     public bool canFly;
     [XmlAttribute]
     public bool canSubmerge;
}
```

Zmiana w definicji klasy (zastosowanie atrybutów)

```
<?xml version="1.0"""?>
<JamesBondCar xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    canFly="true" canSubmerge="false"
    xmlns="http://www.intertechtraining.com">
...
</JamesBondCar>
```

Nowy wynikowy dokument XML

Serializacja kolekcji obiektów (1)

- Ponieważ kolekcje z przestrzeni nazw *System.Collections* i *System.Collection.Generic* są serializowalne (oznaczone atrybutem [Serializable]), serilaizacja kolekcji nie różni się niczym od serializacji zwykłych grafów obiektów.
- Przykład serializacji kolekcji za pomocą typu *BinaryFormater*.

```
static void SaveListOfCarsAsBinary()
{
    List<JamesBondCar> myCars = new List<JamesBondCar>();
    BinaryFormatter binFormat = new BinaryFormatter();

    using (Stream fStream = new FileStream("AllMyCars.dat",
        FileMode.Create, FileAccess.Write, FileShare.None))
    {
        binFormat.Serialize(fStream, myCars);
    }
    Console.WriteLine("=> Saved list of cars in binary!");
}
```

Serializacja kolekcji obiektów (2)

- Serializacja za pomocą typu *XmlSerializer* wymaga podania metadanych o odpowiednich typach. W typ przyapdku głównym typem będzie kolekcja generyczna *List<JamesBondCar>*.

```
static void SaveListOfCars()
{
    List<JamesBondCar> myCars = new List<JamesBondCar>();
    myCars.Add(new JamesBondCar(true, true));
    myCars.Add(new JamesBondCar(true, false));
    myCars.Add(new JamesBondCar(false, true));
    myCars.Add(new JamesBondCar(false, false));

    using (Stream fStream = new FileStream("CarCollection.xml",
    FileMode.Create, FileAccess.Write, FileShare.None))
    {
        XmlSerializer xmlFormat = new XmlSerializer(typeof(List<JamesBondCar>),
            new Type[] { typeof(JamesBondCar), typeof(Car), typeof(Radio) });
        xmlFormat.Serialize(fStream, myCars);
    }
    Console.WriteLine("=> Saved list of cars!");
}
```

Konfiguracja procesu serializacji

Konfiguracja procesu serializacji

- W niektórych przypadkach, domyślne działanie serializacji może okazać się niewystarczające.
- Możemy chcieć w pewien sposób zmodyfikować proces serializacji, np.:
  - Pewna reguła biznesowa mówi nam, że pola musza być utrwalone w pewnym zadanym formacie,
  - Możemy chcieć dodać dodatkową informację, której nie ma w polach obiektu (czas utrwalenia, unikalny identyfikator, itd.).

- Przestrzeń nazw *System.Runtime.Serialization* dostarcza kilku typów, które pozwalają na konfigurację procesu serializacji.

Typy związane z konfiguracją serializacji

Typ	Znaczenie
ISerializable	Ten interfejs może być zaiplementowany przez typ oznaczony jako [Serializable] w celu kontroli procesu serializacji i deserializacji.
ObjectIGenerator	Typ ten generuje identyfikatory dla składowych w grafie obitków.
[OnDeserialized]	Metoda oznaczona tym atrybutem będzie wywołana natychmiast po deserializacji obiektu.
[OnDeserializing]	Metoda oznaczona tym atrybutem będzie wywołana przed deserializacją obiektu.
[OnSerialized]	Metoda oznaczona tym atrybutem będzie wywołana natychmiast po serializacjiobiektu.
[OnSerializing]	Metoda oznaczona tym atrybutem będzie wywołana przed serializacją obiektu.
[OptionalFiled]	Pozwala na zdefiniowanie brakującego pola.
SerializationInfo	Typ ten jest zbiorem par nazwa/wartość reprezentujących stan obiektu podczas serializacji.

Kolejne fazy procesu serializacji (1)

- W momencie serializacji obiektu, typ *BinaryFormater*, przesyła następujące informacje do danego strumienia:
  - pełną kwalifikowaną nazwę obiektów w grafie obiektów,
  - nazwę pakietu definiującego graf obiektów,
  - instancję klasy *SerializationInfo* zawierającą informację o stanach każdego z obiektów w utrwalanym grafie obiektów.
- Podczas procesu deserializacji, typ *BinaryFormater* wykorzystuje te informacje do odtworzenia identycznej kopi obiektu (jak przed zapisem).
- Typ *SoapFormater* zachowuje się podobnie.

Kolejne fazy procesu serializacji (2)

- Poza przenoszeniem danych do i ze strumienia, formatery analizują składowe obiektów w grafie szukając następujących elementów infrastruktury:
  - Sprawdzane jest, czy typ danego obiektu oznaczony jest atrybutem [Serializable].
  - Jeżeli jest oznaczony tym atrybutem, sprawdza się, czy typ danego obiektu implementuje interfejs *ISerializable*. Jeżeli tak, wywoływana jest metoda *GetObjectData()* danego obietku.
  - Jeżeli dany obiekt nie implementuje tego interfejsu, uruchomiany jest domyślny proces serializujący wszystkie pola nieoznaczone atrybutem [NonSerialized].

- Dodatkowo, oprócz sprawdzania implementowania interfejsu ISerializable, sprawdzane jest czy dany typ obiektu posiada metody oznaczone atrybutami [OnSerializing], [OnSerialized], [OnDeserializing], [OnDeserialized].

Implementacja interfejsu *ISerializable*

- Obiekty oznaczone atrybutem [Serializable] mogą implementować interfejs *ISerializable*. Pozwala to na zaangażowanie się w proces serializacji i wykonywanie formatowania danych w wybranym momencie (przed lub po).

- Od wersji platformy .NET 2.0, preferowane jest stosowanie metod oznaczonych odpowiednimi atrybutami.

- Interfejs *ISerializable* definiuje jedną metodę.

```
public interface ISerializable
{
    void GetObjectData(SerializationInfo info,
                       StreamingContext context);
}
```

Klasa *SerializationInfo*

- Metoda *GetObjectData()* jest wywoływana automatycznie przez formater w czasie serializacji.
- Implementacja tej metody wypełnia parametr (*SerializationInfo*) zbiorem par nazwa/wartość reprezentujących pola danych obiektu.

- Typ *SerializationInfo* definiuje wiele wariantów przeładowanej metody *Addvalue()* oraz kilka właściwości pozwalających na pobranie i ustawienie nazwy typu, pakietu i liczby składowych.

```
public sealed class SerializationInfo : object
{
    public SerializationInfo(Type type, IFormatterConverter converter);
    public string AssemblyName { get; set; }
    public string FullTypeName { get; set; }
    public int MemberCount { get; }
    public void AddValue(string name, short value);
    public void AddValue(string name, UInt16 value);
    public void AddValue(string name, int value);
    //...
}
```

Specjalny konstruktor (1)

- Klasa implementująca interfejs *ISerializable* musi również definiować specjalny konstruktor o następujących parametrach:

```
[Serializable]
class SomeClass : ISerializable
{
    protected SomeClass(SerializationInfo si, StreamingContext ctx) {
        //...
    }

    //...
}
```

- Konstruktor jest deklarowany jako chroniony po to, aby nie mogli go wywoływać „zwykli” użytkownicy.
- Pierwszym parametrem konstruktora jest instancja typu *SerializationInfo*, zawierająca zbiór parametrów odczytanych ze strumienia, które następnie zostają przypisane odpowiednim polom danych odtwarzanego obiektu.

Specjalny konstruktor (2)

- Drugim parametrem tego specjalnego konstruktora jest typ *StreamingContext*, który zawiera informacje odnoście źródła lub miejsca przeznaczenia danych.
- Najwięcej mówiącą składową tego typu jest właściwość *State* zwracająca wartość z z enumeracji *StreamingContextStates*.

```
public enum StreamingContextStates
{
    CrossProcess,
    CrossMachine,
    File,
    Persistence,
    Remoting,
    Other,
    Clone,
    CrossAppDomain,
    All
}
```

Przykład implementacji interfejsu *ISerializable*


```
[Serializable]
class StringData : ISerializable
{
    public string dataItemOne = "First data block";
    public string dataItemTwo = "More data";

    public StringData() { }
    protected StringData(SerializationInfo si, StreamingContext ctx)
    {
        // RWydobycie skadowych ze strumienia.
        dataItemOne = si.GetString("First_Item").ToLower();
        dataItemTwo = si.GetString("dataItemTwo").ToLower();
    }

    void ISerializable.GetObjectData(SerializationInfo info, StreamingContext ctx)
    {
        // Wypelnienie obiektu SerializationInfo.
        info.AddValue("First_Item", dataItemOne.ToUpper());
        info.AddValue("dataItemTwo", dataItemTwo.ToUpper());
    }
}
```

Wynik przykładowej konfiguracji serializacji

- Wynikiem przykładowej konfiguracji procesu serializacji będzie następujący plik.

MyData.soap | StringData.cs | Program.cs

```
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema
  <SOAP-ENV:Body>
    <a1:StringData id="ref-1" xmlns:a1="http://schemas.microso
      <First_Item id="ref-3">FIRST DATA BLOCK</First_Item>
      <dataItemTwo id="ref-4">MORE DATA</dataItemTwo>
    </a1:StringData>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

- Nazwy jego elementów nie wynikają bezpośrednio z pól danych obiektu, ale zostały odpowiednio skonfigurowane w metodzie *GetObiectDate()* (podczas serializacji) i specjalnym konstruktorze (podczas deserializacji).

Konfiguracja za pomocą atrybutów (1)

- Drugim sposobem konfiguracji procesu serializacji (preferowanym sposobem) jest definiowanie metod oznaczanych odpowiednimi atrybutami mówiącymi o momencie w procesie serializcji, w którym mają być wywołane.

- Atrybuty te to [OnSerializing], [OnSerialized], [OnDeserializing] i [OnDeserialized].

- Użycie atrybutów (w porównaniu z implementacją interfejsu ISerializable) jest wygodniejsze, ponieważ nie wymaga bezpośredniej interakcji z typem *SerializationInfo*.

Konfiguracja za pomocą atrybutów (2)

- Atrybuty te są zdefiniowane w przestrzeni nazw *System.Runtime.Serialization*.
- Metody oznaczone tymi atrybutami muszą przyjmować jako parametr typ *StreamingContext* i nie mogą nic zwracać.
- Nie jest wymagane uwzględnianie wszystkich atrybutów związanych z serializacją. Możemy zastosować tylko te, które nas w danej chwili interesują.

Przykładowa konfiguracji przy użyciu atrybutów


```
[Serializable]
class MoreData
{
    public string dataItemOne = "First data block";
    public string dataItemTwo = "More data";

    [OnSerializing]
    private void OnSerializing(StreamingContext context)
    {
        // Wywyolywany podczas procesu serializacji.
        dataItemOne = dataItemOne.ToUpper();
        dataItemTwo = dataItemTwo.ToUpper();
    }

    [OnDeserialized]
    private void OnDeserialized(StreamingContext context)
    {
        // Wywolywany po zakonczeniu deserializacji.
        dataItemOne = dataItemOne.ToLower();
        dataItemTwo = dataItemTwo.ToLower();
    }
}
```

Dziękuję za uwagę

Waldemar Bartyna
wbartyna@gmail.com