26

Serializacja

W tym rozdziale zajmiemy się serializacją i deserializacją, czyli mechanizmami pozwalającymi na przedstawienie obiektów w pliku jednorodnym bądź w postaci binarnej. Jeśli nie zaznaczono inaczej, wszystkie typy wymienione w rozdziale pochodzą z następujących przestrzeni nazw:

```
System.Runtime.Serialization
System.Xml.Serialization
System.Text.Json
```

Koncepcje serializacji

Serializacja to akt pobrania obiektu z pamięci lub **drzewa obiektów** (zbioru obiektów wzajemnie odwołujących się do siebie) i umieszczenia ich w strumieniu bajtów lub węzłów w formacie XML, JSON lub innym, które następnie można przechowywać lub transportować. Z kolei **deserializacja** działa w odwrotną stronę, czyli polega na pobraniu strumienia danych i utworzeniu na jego podstawie obiektu w pamięci lub drzewa obiektów.

Serializacja i deserializacja są z reguły używane do następujących celów:

- transmisja obiektów przez sieć lub poza aplikację;
- przechowywanie reprezentacji obiektów w pliku bądź w bazie danych.

Innym, rzadziej spotykanym użyciem serializacji jest utworzenie tzw. głębokiej kopii obiektów. Kontrakt danych i narzędzia do serializacji mogą być wykorzystane jako narzędzia ogólnego przeznaczenia do wczytywania oraz zapisywania plików XML o znanej strukturze, a serializator JSON pełni tę samą funkcję w odniesieniu do plików JSON.

Platforma .NET Framework obsługuje serializację i deserializację z perspektywy zarówno klienta, który chce przeprowadzać serializację i deserializację obiektów, jak i typów, które chcą zachować kontrolę nad sposobami ich serializacji.

Mechanizmy serializacji

Platformy .NET 5 i .NET Core oferują cztery mechanizmy serializacji:

- `XmlSerializer` (XML);
- `JsonSerializer` (JSON);
- raczej zbędny serializator kontraktów danych (XML i JSON);
- serializator binarny.

Serializator binarny jest wycofywany z powodu problemów z bezpieczeństwem. Szczegółowe informacje znajdują się na stronie *https://aka.ms/binaryformatter*.

W przypadku serializacji do formatu XML do wyboru są `XmlSerializer` i serializator kontraktów danych. `XmlSerializer` jest elastyczniejszy w sposobie traktowania struktury kodu XML, a serializator kontraktów danych ma wyjątkową zdolność zachowywania odwołań do wspólnych obiektów.

Jeśli wybierzesz serializację do formatu JSON, to także masz wybór. `JsonSerializer` charakteryzuje się najlepszą wydajnością i (od .NET 5) niewiele jest argumentów przemawiających na korzyść serializatora kontraktów. Jeżeli `JsonSerializer` nie będzie spełniał wymagań, dobrym wyborem może być zewnętrzna biblioteka Json.NET.

Serializator kontraktów danych jest najlepszym wyborem w sytuacji, gdy trzeba współpracować ze starymi usługami sieciowymi opartymi na SOAP.

A jeśli nie zależy Ci na formacie, to największe możliwości i najłatwiejszą obsługę daje binarny mechanizm serializacji. Jednak dane, które on zwraca, są nieczytelne dla człowieka i jest on mniej tolerancyjny w odniesieniu do wersji niż inne serializatory.

Jeśli format nie ma znaczenia, to zważywszy, że serializator binarny jest wycofywany, obecnie najlepszym wyborem jest `JsonSerializer`.

W poniższej tabeli znajduje się porównanie poszczególnych silników serializacji. Im więcej gwiazdek, tym lepszy wynik.

Funkcja	XmlSerializer	JsonSerializer	Serializacja kontraktu danych	Serializator binarny
Poziom automatyzacji	****	*****	***	*****
Wynik	XML	JSON	XML lub JSON	Binarny
Sprzężenie typów	Luźne	Luźne	Luźne	Ścisłe
Tolerancja wersji	*****	*****	*****	***
Możliwość deserializacji podtypów	Z pomocą	Nie	Z pomocą	Tak
Zachowanie odwołań do obiektów	Nie	Od .NET 5	Przy użyciu XML	Tak
Możliwość serializacji niepublicznych składowych	Nie	Od .NET 5	Tak	Tak

Funkcja	XmlSerializer	JsonSerializer	Serializacja kontraktu danych	Serializator binarny
Zdolność do wymiany komunikatów	Tak	Tak	Tak	Nie
Elastyczność w zakresie formatu wyjściowego	****	***	**	—
Kompaktowe dane wyjściowe	**	***	**	****
Wydajność	Od * do ***	****	***	***

Mechanizm serializacji XML wymaga wykorzystania tego samego obiektu `XmlSerializer`, aby zapewnić dobrą wydajność.

Dlaczego istnieją cztery mechanizmy?

Powody istnienia czterech mechanizmów serializacji są po części historyczne. Na początku platforma została wydana wraz z dwoma oddzielnymi celami w zakresie serializacji:

- serializacja drzewa obiektów .NET wraz z zachowaniem pełnej wierności pod względem typu i odwołań;
- współpraca ze standardami komunikacji XML i SOAP.

Pierwszy doprowadził do powstania serializatora binarnego (który był wykorzystywany przez .NET Remoting), a drugi — do powstania serializatora `XmlSerializer` (który był wykorzystywany przez usługi sieciowe ASMX).

Gdy w 2006 r. powstała biblioteka Windows Communication Foundation (WCF), potrzebny stał się nowy mechanizm serializacji — **serializator kontraktów danych**. Liczono, że zastąpi on w znacznym stopniu dwa poprzednie. Tego celu jednak nie udało się w pełni osiągnąć, ponieważ nowe narzędzie skupiało się głównie na interoperacyjnej wymianie komunikatów, a pozostałe dwa nadal były przydatne.

Biblioteka WCF z definicji była neutralna pod względem formatu, ale w praktyce na jej kształt miały wpływ wymogi skomplikowanego protokołu SOAP, który później stracił popularność na rzecz REST i JSON. W efekcie najpierw firma Microsoft dodała obsługę formatu JSON do serializatora kontraktów danych, a następnie biblioteka WCF straciła dotychczasowe zainteresowanie programistów i została usunięta z .NET Core 3. Serializator kontraktów danych pozostał w .NET Core, choć brak biblioteki WCF zmniejszył jego rolę, podobnie jak dodanie przez Microsoft serializatora `JsonSerializer` do .NET Core 3 i .NET 5.

XmlSerializer

Mechanizm serializacji XML generuje tylko dane w formacie XML oraz ma uboższą funkcjonalność w zakresie zapisywania i odzyskiwania złożonych drzew obiektów (nie przywraca wspólnych odwołań do obiektów) niż serializator binarny i kontraktów danych. Z drugiej strony jest najbardziej elastyczny z wszystkich czterech pod względem struktury wynikowej. Programista może np. wybrać, czy własności mają stać się elementami czy atrybutami oraz sposób postępowania z elementem zewnętrznym kolekcji. Ponadto mechanizm XML jest bardzo liberalny pod względem wersji. `XmlSerializer` był używany przez stare usługi sieciowe ASMX.

JsonSerializer

Serializator JSON jest szybki i efektywny. Ponadto charakteryzuje się dobrą tolerancją wersji i pozwala na używanie niestandardowych konwerterów, co zwiększa jego elastyczność. `JsonSerializer` jest używany przez ASP.NET Core 3, co eliminuje zależność od biblioteki Json.NET, choć z łatwością można się do niej zwrócić z powrotem, jeśli zajdzie taka potrzeba.

Od .NET 5 serializator JSON zachowuje referencje do obiektów.

Serializacja kontraktu danych

Serializacja kontraktu danych obsługuje model **kontraktu danych** pomagający w usunięciu powiązania między niskiego poziomu szczegółami dotyczącymi typu przeznaczonego do serializacji i strukturą serializowanych danych. Ponadto oferuje doskonałą tolerancję na wersję, co oznacza możliwość deserializacji danych, które były serializowane z wcześniejszej lub późniejszej wersji typu. Istnieje nawet możliwość zmiany nazwy bądź przeniesienia deserializowanych typów do zupełnie innego zestawu.

Serializacja kontraktu danych może sobie poradzić z większością drzew obiektów, choć jednocześnie będzie wymagała większej pomocy niż serializacja binarna. Można ją także wykorzystać w charakterze ogólnego przeznaczenia narzędzia do odczytu i zapisu plików XML, jeżeli nie mamy problemów w pracy ze strukturą dokumentów XML. (Jeśli zachodzi potrzeba przechowywania danych w atrybutach lub radzenia sobie z elementami XML przedstawionymi w dowolnej kolejności, wówczas nie można zastosować omawianej tutaj serializacji kontraktu danych).

Serializator binarny

Mechanizm serializacji binarnej jest łatwy w użyciu, wysoce zautomatyzowany oraz doskonale obsługiwany przez platformy .NET 5 i .NET Core 3 (i jeszcze lepiej przez .NET Framework). Bardzo często pojedynczy atrybut to wszystko, co jest wymagane, aby skomplikowany typ stał się w pełni możliwy do serializacji. Ponadto serializacja binarna jest szybsza niż serializacja kontraktu danych, gdy trzeba zapewnić wierne zachowanie typu. Jednak jej ścisłe powiązanie wewnętrznej struktury z formatem serializowanych danych oznacza dość słabą tolerancję na wersję (choć toleruje dodanie pola). Mechanizm binarny generuje tylko dane binarne, a więc nie uzyskamy za jego pomocą danych w formacie XML ani JSON na platformach .NET 5 i .NET Core. (Na platformie .NET Framework dostępny jest formater dla komunikatów opartych na SOAP, zapewniający ograniczoną obsługę formatu XML).

IXmlSerializable

Do skomplikowanych serializacji można zaimplementować interfejs `IXmlSerializable`, aby samodzielnie wykonywać serializację za pomocą klas `XmlReader` i `XmlWriter`. Omawiany interfejs jest rozpoznawany zarówno przez `XmlSerializer`, jak i serializator kontraktów danych, więc można go użyć do selektywnej obsługi znacznie bardziej skomplikowanych typów. Dokładne omówienie klas `XmlReader` i `XmlWriter` znajduje się w rozdziale 11.

Formatery

Dane wyjściowe serializacji kontraktu danych i serializacji binarnej przyjmują postać możliwego do wstawienia **formatera**. Rola formatera jest taka sama w obu silnikach serializacji, choć wykorzystują one całkowicie inne klasy do wykonania zadania.

Formater kształtuje ostateczną postać prezentacji w taki sposób, aby pasowała do określonego medium lub kontekstu serializacji. Na platformach .NET 5 i .NET Core serializator kontraktów danych pozwala wybrać między formaterem XML i JSON, a na platformie .NET Framework do wyboru jest jeszcze formater binarny. Formater binarny został zaprojektowany do pracy w kontekście, w którym będzie użyty dowolny strumień bajtów — zwykle strumień lub plik bądź też pakiet komunikacyjny. Binarne dane mają zwykle mniejszą objętość niż dane w formacie XML lub JSON.

Na platformach .NET 5 i .NET Core serializator binarny udostępnia tylko formater binarny (na platformie .NET Framework dostępny jest także formater SOAP dla wiadomości w formacie XML).

Serializacja jawna kontra serializacja niejawna

Serializacja i deserializacja mogą być zainicjowane na dwa sposoby.

Pierwszy to **jawny** — przez żądanie serializacji lub deserializacji określonego obiektu. Podczas przeprowadzania jawnej serializacji lub deserializacji można wybrać zarówno silnik serializacji, jak i formater.

Drugi to **niejawny** — w tym przypadku serializacja jest zainicjowana przez .NET. Z tym rodzajem serializacji mamy do czynienia w następujących sytuacjach:

- serializacja powoduje rekurencyjną serializację obiektu potomnego;
- wykorzystujemy funkcję opierającą się na serializacji, np.: API sieciowe.

API sieciowy obsługuje zarówno serializację XML, jak i JSON.

Niejawna serializacja jest rzadziej spotykana w .NET 5 i .NET Core niż na platformie .NET Framework, która zawiera bibliotekę WCF (niejawnie wykorzystującą serializator kontraktów danych), Remoting (niejawnie wykorzystującą serializator binarny) oraz usługi sieciowe ASMX (niejawnie wykorzystujące `XmlSerializer`).

Serializator XML

Klasa `XMLSerializer` z przestrzeni nazw `System.Xml.Serialization` służy do serializacji i deserializacji danych na podstawie atrybutów znajdujących się w danej klasie.

Podstawy serializacji na podstawie atrybutów

W celu użycia klasy `XmlSerializer` należy utworzyć egzemplarz i wywołać metodę `Serialize()` lub `Deserialize()` wraz z klasą `Stream` i egzemplarzem obiektu. Aby to zilustrować, przyjmujemy założenie o zdefiniowaniu następującej klasy:

```
public class Person
{
  public string Name;
  public int Age;
}
```

Poniższy fragment kodu powoduje najpierw zapisanie obiektu `Person` w pliku XML, a następnie jego przywrócenie:

```
Person p = new Person();
p.Name = "Staszek"; p.Age = 30;

var xs = new XmlSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  xs.Serialize (s, p);

Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) xs.Deserialize (s);

Console.WriteLine (p2.Name + " " + p2.Age);    // Staszek 30
```

Metody `Serialize()` i `Deserialize()` mogą działać razem z klasami `Stream`, `XmlWriter`/`XmlReader` oraz `TextWriter`/`TextReader`. Poniżej przedstawiono wygenerowany plik w formacie XML:

```
<?xml version="1.0"?>
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Name>Staszek</Name>
  <Age>30</Age>
</Person>
```

Klasa `XmlSerializer` może serializować typy (np. `Person`) bez jakichkolwiek atrybutów. Domyślnie serializuje wszystkie *publiczne elementy składowe i właściwości* w danym typie. Za pomocą atrybutu `[XmlIgnore]` można wskazać elementy składowe, które nie mają być serializowane:

```
public class Person
{
  ...
  [XmlIgnore] public DateTime DateOfBirth;
}
```

`XmlSerializer` opiera się na pozbawionym parametrów konstruktorze deserializacji. Jeżeli nie będzie on dostępny, wówczas nastąpi zgłoszenie wyjątku. (W omawianym przykładzie klasa `Person` ma *niejawny* konstruktor pozbawiony parametrów). To oczywiście oznacza wykonanie metod inicjalizacyjnych elementów składowych przed deserializacją:

```
public class Person
{
  public bool Valid = true;     // wykonanie przed deserializacją
}
```

`XmlSerializer` może serializować w zasadzie dowolny typ. Rozpoznaje typy wymienione poniżej i traktuje je w sposób specjalny:

- typy proste, `DateTime`, `TimeSpan`, `Guid` i ich wersje akceptujące wartość `null`;
- `byte[]` (serializacja XML na postać base64);

- `XmlAttribute` lub `XmlElement` (ich zawartość będzie wstrzykiwana do strumienia);
- dowolny typ implementujący `IXmlSerializable`;
- dowolny typ kolekcji.

Deserializator jest odporny na wersjonowanie i nie zgłasza zastrzeżeń, gdy elementy bądź atrybuty są niedostępne, lub w przypadku istnienia zbędnych danych.

Atrybuty, nazwy i przestrzenie nazw

Domyślnie elementy składowe i właściwości są serializowane do elementu XML. Istnieje możliwość żądania użycia atrybutu XML w pokazany poniżej sposób:

```
[XmlAttribute] public int Age;
```

Kontrola w zakresie nazwy elementu lub atrybutu odbywa się w sposób pokazany w poniższym fragmencie kodu:

```
public class Person
{
  [XmlElement ("FirstName")] public string Name;
  [XmlAttribute ("RoughAge")] public int Age;
}
```

Oto wynikowe dane w formacie XML:

```
<Person RoughAge="30" ...>
  <FirstName>Staszek</FirstName>
</Person>
```

Domyślnie przestrzeń nazw XML jest pusta. Jeżeli chcemy podać przestrzeń nazw XML, atrybuty `[XmlElement]` i `[XmlAttribute]` akceptują argument `Namespace`. Istnieje również możliwość przypisania nazwy i przestrzeni nazw do samego typu za pomocą atrybutu `[XmlRoot]`:

```
[XmlRoot ("Candidate", Namespace = "http://mynamespace/test/")]
public class Person { ... }
```

W ten sposób element `person` otrzymuje nazwę `Candidate`, a ponadto przypisaliśmy przestrzeń nazw wymienionemu elementowi oraz jego elementom potomnym.

Kolejność elementów XML

Klasa `XmlSerializer` zapisuje elementy w kolejności, w jakiej zostały zdefiniowane w klasie. Tę kolejność można zmienić przez podanie w atrybucie `[XmlElement]` argumentu `Order`:

```
public class Person
{
  [XmlElement (Order = 2)] public string Name;
  [XmlElement (Order = 1)] public int Age;
}
```

Jeżeli zdecydujemy się na użycie argumentu `Order`, musimy go stosować dla wszystkich elementów.

Deserializator nie jest wybredny pod względem kolejności elementów — mogą się pojawiać w dowolnej sekwencji, a typ i tak zostanie poprawnie deserializowany.

Podklasy i obiekty potomne

Podklasy typu głównego

Przyjmujemy założenie, że typ główny ma dwie podklasy, jak w poniższym fragmencie kodu:

```
public class Person { public string Name; }

public class Student : Person { }
public class Teacher : Person { }
```

Następnie tworzymy metodę wielokrotnego użycia przeznaczoną do serializacji typu głównego:

```
public void SerializePerson (Person p, string path)
{
  XmlSerializer xs = new XmlSerializer (typeof (Person));
  using (Stream s = File.Create (path))
    xs.Serialize (s, p);
}
```

Aby ta metoda działała z klasą `Student` lub `Teacher`, konieczne jest poinformowanie `XmlSerializer` o podklasach. Mamy w tym zakresie dwa sposoby. Pierwszy polega na zarejestrowaniu każdej podklasy za pomocą atrybutu `[XmlInclude]`:

```
[XmlInclude (typeof (Student))]
[XmlInclude (typeof (Teacher))]
public class Person { public string Name; }
```

Natomiast drugi polega na podaniu wszystkich podtypów podczas tworzenia egzemplarza `XmlSerializer`:

```
XmlSerializer xs = new XmlSerializer (typeof (Person),
                    new Type[] { typeof (Student), typeof (Teacher) } );
```

Niezależnie od wybranego sposobu serializator reaguje przez zarejestrowanie podtypu w atrybucie `type`:

```
<Person xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:type="Student">
  <Name>Staszek</Name>
</Person>
```

Teraz tak przygotowany deserializator wie, że na podstawie podanego atrybutu ma utworzyć egzemplarz klasy `Student`, a nie `Person`.

Istnieje możliwość podania nazwy wyświetlanej w atrybucie XML `type`. Wymaga to zastosowania w podklasie atrybutu `[XmlType]`:

```
[XmlType ("Candidate")]
public class Student : Person { }
```

Poniżej przedstawiono wygenerowane dane w formacie XML:

```
<Person xmlns:xsi="..."
        xsi:type="Candidate">
```

Serializacja obiektów potomnych

Klasa `XmlSerializer` przeprowadza automatyczną rekurencję po odwołaniach obiektu, takich jak element składowy `HomeAddress` w `Person`:

```
public class Person
{
  public string Name;
  public Address HomeAddress = new Address();
}

public class Address { public string Street, PostCode; }
```

Oto przykład:

```
Person p = new Person { Name = "Staszek" };
p.HomeAddress.Street = "Odo St";
p.HomeAddress.PostCode = "6020";
```

Poniżej przedstawiono dane w formacie XML otrzymane na skutek serializacji:

```
<Person ... >
  <Name>Staszek</Name>
  <HomeAddress>
    <Street>Odo St</Street>
    <PostCode>6020</PostCode>
  </HomeAddress>
</Person>
```

Jeżeli masz dwa elementy składowe lub właściwości odwołujące się do tego samego obiektu, wówczas taki obiekt zostanie serializowany dwukrotnie. Gdy zachodzi potrzeba zachowania odwołań obiektu, musisz użyć innego silnika serializacji.

Podklasy obiektów potomnych

Przyjmujemy założenie o konieczności serializacji obiektu `Person`, który może zawierać odwołania do *podklas* `Address` w pokazanej poniżej postaci:

```
public class Address { public string Street, PostCode; }
public class USAddress : Address {  }
public class AUAddress : Address {  }

public class Person
{
  public string Name;
  public Address HomeAddress = new USAddress();
}
```

W tym zakresie mamy dwa odmienne rozwiązania w zależności od sposobu, w jaki chcemy przygotować strukturę dokumentu XML. Jeżeli nazwa elementu ma zawsze odpowiadać nazwie elementu składowego lub właściwości wraz z podtypem zapisanym w atrybucie `type`, jak w poniższym fragmencie kodu:

```
<Person ...>
  ...
  <HomeAddress xsi:type="USAddress">
    ...
  </HomeAddress>
</Person>
```

to należy użyć atrybutu [XmlInclude] do zarejestrowania wszystkich podklas wraz z Address, jak pokazano poniżej:

```
[XmlInclude (typeof (AUAddress))]
[XmlInclude (typeof (USAddress))]
public class Address
{
  public string Street, PostCode;
}
```

Natomiast jeśli nazwa elementu ma odzwierciedlać nazwę podtypu, aby osiągnąć przedstawiony poniżej efekt:

```
<Person ...>
  ...
  <USAddress>
    ...
  </USAddress>
</Person>
```

to należy użyć wielu atrybutów [XmlElement] w elemencie składowym lub we właściwości typu nadrzędnego, jak w poniższym fragmencie kodu:

```
public class Person
{
  public string Name;
  [XmlElement ("Address", typeof (Address))]
  [XmlElement ("AUAddress", typeof (AUAddress))]
  [XmlElement ("USAddress", typeof (USAddress))]
  public Address HomeAddress = new USAddress();
}
```

Każdy atrybut [XmlElement] mapuje nazwę elementu na typ. Po zastosowaniu tego rodzaju podejścia nie jest wymagane użycie atrybutów [XmlInclude] w typie Address (choć ich obecność nie spowoduje zakłócenia serializacji).

Jeżeli pominiesz nazwę elementu w [XmlElement] i po prostu podasz typ, wówczas zostanie użyta domyślna nazwa typu, na którą wpływ ma atrybut [XmlType], ale nie [XmlRoot].

Serializacja kolekcji

Klasa XmlSerializer rozpoznaje i serializuje konkretne typy kolekcji bez konieczności interwencji programisty:

```
public class Person
{
  public string Name;
  public List<Address> Addresses = new List<Address>();
}
public class Address { public string Street, PostCode; }
```

Poniżej przedstawiono dane w formacie XML otrzymane na skutek serializacji:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
```

```
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    <Address>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Address>
    ...
  </Addresses>
</Person>
```

Atrybut [XmlArray] pozwala na zmianę nazwy elementu *zewnętrznego*, np. Addresses.

Atrybut [XmlArrayItem] pozwala na zmianę nazwy elementów *wewnętrznych*, np. Address.

Spójrz na poniższą klasę:

```
public class Person
{
  public string Name;
  [XmlArray ("PreviousAddresses")]
  [XmlArrayItem ("Location")]
  public List<Address> Addresses = new List<Address>();
}
```

Poniżej przedstawiono dane w formacie XML otrzymane na skutek serializacji:

```
<Person ... >
  <Name>...</Name>
  <PreviousAddresses>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    <Location>
      <Street>...</Street>
      <Postcode>...</Postcode>
    </Location>
    ...
  </PreviousAddresses>
</Person>
```

Atrybuty [XmlArray] i [XmlArrayItem] pozwalają na zdefiniowanie przestrzeni nazw XML.

Aby przeprowadzić serializację kolekcji *bez* zewnętrznego elementu, np. jak w poniższym fragmencie kodu:

```
<Person ... >
  <Name>...</Name>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
  <Address>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </Address>
</Person>
```

należy dodać atrybut [XmlElement] do elementu składowego kolekcji lub właściwości:

```
public class Person
{
  ...
  [XmlElement ("Address")]
  public List<Address> Addresses = new List<Address>();
}
```

Praca z elementami podklas kolekcji

Reguły dotyczące elementów podklas kolekcji są takie same jak w przypadku innych reguł podklas. Aby zakodować elementy podklasy w atrybucie type, np.:

```
<Person ... >
  <Name>...</Name>
  <Addresses>
    <Address xsi:type="AUAddress">
    ...
```

należy dodać atrybuty [XmlInclude] do typu bazowego (Address), czyli podobnie jak to zrobiliśmy już wcześniej. Takie rozwiązanie działa niezależnie od tego, czy serializowany będzie również element zewnętrzny.

Aby elementy podklas miały nazwy zgodne z ich typami, np.:

```
<Person ... >
  <Name>...</Name>
  <!—początek opcjonalnego elementu zewnętrznego—>
  <AUAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </AUAddress>
  <USAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </USAddress>
  <!—koniec opcjonalnego elementu zewnętrznego—>
</Person>
```

należy użyć wielu atrybutów [XmlArrayItem] lub [XmlElement] w elemencie składowym lub we właściwości kolekcji.

Aby *dołączyć* zewnętrzny element kolekcji, należy zastosować wiele atrybutów [XmlArrayItem]:

```
[XmlArrayItem ("Address",   typeof (Address))]
[XmlArrayItem ("AUAddress", typeof (AUAddress))]
[XmlArrayItem ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

Natomiast aby *pominąć* zewnętrzny element kolekcji, należy zastosować stos elementów [XmlElement]:

```
[XmlElement ("Address",   typeof (Address))]
[XmlElement ("AUAddress", typeof (AUAddress))]
[XmlElement ("USAddress", typeof (USAddress))]
public List<Address> Addresses = new List<Address>();
```

Interfejs IXmlSerializable

Wprawdzie serializacja XML na podstawie atrybutów jest elastyczna, ale ma również pewne ograniczenia. Na przykład nie można dołączyć zaczepów serializacji, a także nie można serializować niepublicznych elementów składowych. Ponadto staje się niewygodna w użyciu, jeżeli XML może przedstawić ten sam element bądź atrybut na wiele różnych sposobów.

W przypadku tego ostatniego ograniczenia można nieco przesunąć granicę przez przekazanie obiektu `XmlAttributeOverrides` do konstruktora `XmlSerializer`. Jednak docieramy do punktu, w którym łatwiejsze będzie zastosowanie podejścia imperatywnego. To zadanie interfejsu `IXmlSerializable`:

```
public interface IXmlSerializable
{
  XmlSchema GetSchema();
  void ReadXml (XmlReader reader);
  void WriteXml (XmlWriter writer);
}
```

Implementacja tego interfejsu daje pełną kontrolę nad danymi w formacie XML, które są odczytywane lub zapisywane.

Klasa kolekcji implementująca interfejs `IXmlSerializable` pomija reguły `XmlSerializer` dotyczące serializacji kolekcji. Może to być użyteczne, gdy zachodzi konieczność serializacji kolekcji wraz z dodatkami, innymi słowy: z dodatkowymi elementami składowymi lub właściwościami, które w przeciwnym razie byłyby zignorowane.

Poniżej wymieniono reguły dotyczące implementacji interfejsu `IXmlSerializable`:

- Wywołanie `ReadXml()` powinno rozpocząć odczyt danych od zewnętrznego elementu początkowego, następnie przejść do treści i później do zewnętrznego elementu końcowego.
- Wywołanie `WriteXML()` powinno jedynie zapisać treść.

Spójrz na poniższy fragment kodu:

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

public class Address : IXmlSerializable
{
  public string Street, PostCode;

  public XmlSchema GetSchema() { return null; }

  public void ReadXml(XmlReader reader)
  {
    reader.ReadStartElement();
    Street   = reader.ReadElementContentAsString ("Street", "");
    PostCode = reader.ReadElementContentAsString ("PostCode", "");
    reader.ReadEndElement();
  }
```

```
  public void WriteXml (XmlWriter writer)
  {
    writer.WriteElementString ("Street", Street);
    writer.WriteElementString ("PostCode", PostCode);
  }
}
```

Serializacja i deserializacja egzemplarza `Address` za pomocą `XmlSerializer` automatycznie wywołują metody `WriteXml()` i `ReadXml()`. Co więcej, jeżeli klasa `Person` została zdefiniowana w poniższy sposób:

```
public class Person
{
  public string Name;
  public Address HomeAddress;
}
```

wówczas interfejs `IXmlSerializable` będzie wywołany selektywnie do serializacji elementu składowego `HomeAddress`.

Dokładne omówienie klas `XmlReader` i `XmlWriter` znajduje się w pierwszej części rozdziału 11. W wymienionym rozdziale, a dokładnie w sekcji „Typowe zastosowania klas XmlReader i XmlWriter", przedstawiliśmy przykłady klas implementujących interfejs `IXmlSerializable`.

Serializator JSON

Klasa `JsonSerializer` (z przestrzeni nazw `System.Text.Json`) jest łatwa w użyciu dzięki prostocie formatu JSON. Elementem głównym dokumentu JSON jest tablica lub obiekt. Element ten ma własności, którymi mogą być obiekty, tablice, łańcuchy, liczby oraz wartości `"true"`, `"false"` i `"null"`. Serializator JSON bezpośrednio odwzorowuje nazwy własności klas na nazwy własności w JSON.

Podstawy

Mamy klasę `Person` o następującej definicji:

```
public class Person
{
  public string Name { get; set; }
}
```

Możemy dokonać jej serializacji do postaci łańcucha JSON za pomocą wywołania metody `JsonSerializer.Serialize`:

```
var p = new Person { Name = "Jan" };
string json = JsonSerializer.Serialize (p,
                new JsonSerializerOptions { WriteIndented = true });
```

Wynik:

```
{
Name: "Jan"
}
```

Metoda `JsonSerializer.Deserialize` wykonuje odwrotną operację, czyli deserializuje dane:

```
Person p2 = JsonSerializer.Deserialize<Person> (json);
```

Deserializacja typów niezmiennych

Deserializator może wypełniać typy własnościami tylko do odczytu — pod warunkiem że istnieje publiczny konstruktor z parametrami o nazwach odpowiadających nazwom deserializowanych własności (wielkość liter jest bez znaczenia):

```
var p = new Person ("Joe", "Bloggs");
string json = JsonSerializer.Serialize (p);
Person p2 = JsonSerializer.Deserialize<Person> (json);

public class Person
{
    public string FirstName { get; }
    public string LastName { get; }
    public Person (string firstName, string lastName)
      => (FirstName, LastName) = (firstName, lastName);
}
```

Jeśli typ definiuje więcej niż jeden konstruktor z parametrami, należy poinformować deserializator, którego ma użyć, przez opatrzenie go atrybutem `[JsonConstructor]`.

Rekordy (wprowadzone w C# 9) domyślnie bardzo dobrze współpracują z deserializatorem.

Serializacja obiektów potomnych

Do naszej klasy `Person` postanowiliśmy dodać adres domowy i do pracy:

```
public class Address
{
  public string Street { get; set; }
  public string PostCode { get; set; }
}

public class Person
{
  public string Name { get; set; }
  public Address HomeAddress { get; set; }
  public Address WorkAddress { get; set; }
}
```

Serializacja tej klasy nie wymaga od nas żadnej dodatkowej pracy:

```
var home = new Address { Street = "1 Main St.", PostCode="11235" };
var work = new Address { Street = "4 Elm Ln.", PostCode="31415" };
var p = new Person { Name = "Ian", HomeAddress = home, WorkAddress = work };

Console.WriteLine (JsonSerializer.Serialize (p,
                 new JsonSerializerOptions { WriteIndented = true } ));
```

Kiedy serializator JSON napotka `HomeAddress` i `WorkAddress`, utworzy obiekty JSON:

```
{
  "Name": "Ian",
  "HomeAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  },
  "WorkAddress": {
    "Street": "4 Elm Ln.",
    "PostCode": "31415"
  }
}
```

Zwróć jednak uwagę, co się stanie, gdy ustawimy `HomeAddress` i `WorkAddress` na ten sam obiekt:

```
var p = new Person { Name = "Ian", HomeAddress = home, WorkAddress = home };
```

Wynik:

```
{
  "Name": "Ian",
  "HomeAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  },
  "WorkAddress": {
    "Street": "1 Main St.",
    "PostCode": "11235"
  }
}
```

W JSON nie ma informacji wskazującej, że `HomeAddress` i `WorkAddress` pierwotnie były tym samym obiektem. W trakcie deserializacji zostaną utworzone dwa osobne egzemplarze `Address`, które zostaną przypisane do odpowiednich własności.

Zachowywanie referencji do obiektów

Ten problem można także rozwiązać (od .NET 5) przez nakazanie serializatorowi, aby zachował referencje do obiektów. Służy do tego następująca opcja:

```
new JsonSerializerOptions
{
   ReferenceHandler = ReferenceHandler.Preserve,
    WriteIndented = true
}
```

Teraz wynik wygląda tak:

```
{
    "$id": "1",
    "Name": "Ian",
    "HomeAddress": {
    "$id": "2",
    "Street": "1 Main St.",
    "PostCode": "11235"
},
```

```
    "WorkAddress": {
        "$ref": "2"
    }
}
```

To spowoduje deserializację bez duplikowania obiektu adresu — pod warunkiem że przekażemy tę samą opcję do deserializatora.

Ponadto zachowywanie referencji do obiektów umożliwia serializatorowi `JsonSerializer` obsługę cykli w grafie obiektów.

Serializacja kolekcji

`JsonSerializer` automatycznie serializuje kolekcje, które mogą znajdować się we własnościach obiektów, jak i w samym obiekcie głównym. Ten drugi przypadek możemy zilustrować na przykładzie klas `Person` i `Address`, które zdefiniowaliśmy na początku poprzedniego podrozdziału:

```
var sara = new Person { Name = "Sara" };
var ian = new Person { Name = "Jan" };
Console.WriteLine (JsonSerializer.Serialize (new[] { sara, ian },
  new JsonSerializerOptions { WriteIndented = true }));
```

Wynik:

```
[
  {
    "Name": "Sara"
  },
  {
    "Name": "Jan"
  }
]
```

Poniższy kod dokonuje deserializacji:

```
Person[] people = JsonSerializer.Deserialize<Person[]> (json);
```

Można też zserializować kolekcję zawierającą obiekty innych typów:

```
var sara = new Person { Name = "Sara" };
var addr = new Address { Street = "1 Main St.", PostCode = "11235" };

Console.WriteLine (JsonSerializer.Serialize (new object[] { sara, addr },
  new JsonSerializerOptions { WriteIndented = true }));
```

Wynik:

```
[
  {
    "Name": "Sara"
  },
  {
    "Street": "1 Main St.",
    "PostCode": "11235"
  }
]
```

Deserializacja takich kolekcji jest kłopotliwa, ponieważ w formacie JSON nie zostały zapisane typy wszystkich elementów. Należy zastosować niskopoziomowe podejście, polegające na deserializacji do `JsonElement[]` i przejrzeniu wszystkich własności po kolei:

```
var deserialized = JsonSerializer.Deserialize<JsonElement[]>(json);
foreach (var element in deserialized)
{
  foreach (var prop in element.EnumerateObject())
    Console.WriteLine ($"{prop.Name}: {prop.Value}");
  Console.WriteLine ("---");
}

// Wynik:
Name: Sara
---
Street: 1 Main St.
PostCode: 11235
```

Sposób użycia obiektów klasy `JsonElement` opisaliśmy w rozdziale 11.

Kontrolowanie serializacji za pomocą atrybutów

Proces serializacji można kontrolować za pomocą atrybutów zdefiniowanych w przestrzeni nazw `System.Text.Json.Serialization`.

JsonIncludeAttribute (od .NET 5)

Domyślnie serializator/deserializator ignoruje pola, chyba że programista doda atrybut `[JsonInclude]`:

```
class Person
{
    [JsonInclude]
    public string Phone; // to teraz będzie uwzględnione
}
```

Atrybut ten można przypisać także własnościom z niepublicznym akcesorem `set`, co stanowi instrukcję dla deserializatora, aby wywoływać go za pośrednictwem refleksji:

```
class Person
{
    [JsonInclude]
    public string Phone { get; private set; }
}
```

JsonIgnoreAttribute

Domyślnie serializator JSON serializuje wszystkie własności, chyba że programista nakaże mu tego nie robić za pomocą atrybutu `JsonIgnore`:

```
public class Person
{
  public string Name { get; set; }

  [JsonIgnore]
  public decimal NetWorth { get; set; } // wykluczony z serializacji
}
```

JsonPropertyNameAttribute

Jeśli nazwa własności JSON różni się od nazwy własności C#, to można je ze sobą powiązać za pomocą atrybutu `[JsonPropertyName]`. Jeśli np. własność JSON ma nazwę `"FullName"`, a własność C# ma nazwę `Name`, możemy je ze sobą powiązać w następujący sposób:

```
public class Person
{
  [JsonPropertyName("FullName")]
  public string Name { get; set; }
}
```

Wynik serializacji tego będzie następujący:

```
{
  "FullName":"...",
}
```

JsonExtensionDataAttribute

Wyobraź sobie sieciowy interfejs API zwracający obiekty klasy `Person` i klienta, który używa tego interfejsu. Każdy należy do innej organizacji. Jeśli twórca API doda nową własność do klasy `Person` (np. `Age`), to klient nadal będzie mógł zdeserializować dane JSON przy użyciu swojej starej klasy `Person`, ponieważ nieznana własność `Age` zostanie po prostu pominięta. A teraz powiedzmy, że klient aktualizuje swój obiekt klasy `Person`, serializuje go i wysyła do API. Oryginalna wartość własności `Age` zostanie zgubiona.

Aby pokazać to na przykładzie, w sieciowym interfejsie API zdefiniujemy klasę `Person` w następujący sposób:

```
public class Person_// v2

{
  public int Id { get; set; }
  public string Name { get; set; }
  public int Age { get; set; } // nowa własność
}
```

Na jej podstawie zostałby wygenerowany następujący kod JSON:

```
{
  "Id": 27182,
  "Name": "Sara",
  "Age": 35
}
```

Jeśli zdeserializujemy go do starszej wersji klasy (bez własności `Age`):

```
public class Person_// v1
{
  public int Id { get; set; }
  public string Name { get; set; }
}
```

nie będzie miejsca dla informacji na temat wieku.

Jeśli później zserializujemy naszą wersję i wyślemy ją do API, to ten kod JSON nie będzie zawierał własności `Age` i API uzna, że własność ta ma wartość zero (domyślna wartość całkowitoliczbowa).

Problem ten rozwiązuje atrybut `JsonExtensionDataAttribute`, który umożliwia zapisanie wszystkich nierozpoznanych własności, aby można było użyć ich wartości podczas ponownej serializacji. Jeśli atrybut ten zostanie zastosowany do własności typu `IDictionary<string,TValue>` (`TValue` musi być obiektem lub elementem `JsonElement`), serializator użyje jej do zapisania nierozpoznanych własności JSON i żadne informacje nie zostaną utracone:

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }

  [JsonExtensionData]
  public IDictionary<string, JsonElement> Storage { get; set; } =
    new Dictionary<string, JsonElement>();
}
```

JsonConverterAttribute

Ten atrybut służy do określenia typu użytego przy konwersji danych na format JSON i z powrotem. Dokładniej opisujemy go w następnym podrozdziale.

Dostosowywanie parametrów konwersji danych

Powiedzmy, że chcesz skorzystać z usług dostawcy API, który koduje daty w postaci uniksowego znacznika czasu (liczby sekund, jaka upłynęła od 1 stycznia 1970 r.):

```
{
  "Id":27182,
  "Name":"Sara",
  "Born":464572800 // liczba sekund, jaka upłynęła od 1 stycznia 1970 r.
}
```

Chcemy te dane zdeserializować do postaci klasy używającej klasy .NET `DateTime`:

```
public class Person
{
  public int Id { get; set; }
  public string Name { get; set; }
  public DateTime Born { get; set; }
}
```

W tym celu możemy napisać specjalny konwerter danych:

```
public class UnixTimestampConverter : JsonConverter<DateTime>
{
  public override DateTime Read (ref Utf8JsonReader reader, Type type,
                                 JsonSerializerOptions options)
  {
    if (reader.TryGetInt32(out int timestamp))
      return new DateTime (1970, 1, 1).AddSeconds (timestamp);

    throw new Exception ("Expected the timestamp as a number.");
  }
```

```
  public override void Write (Utf8JsonWriter writer, DateTime value,
                              JsonSerializerOptions options)
  {
    int timestamp = (int)(value - new DateTime(1970, 1, 1)).TotalSeconds;
    writer.WriteNumberValue(timestamp);
  }
}
```

Teraz możemy dodać atrybut [JsonConverter] do własności, które chcemy przekonwertować:

```
[JsonConverter(typeof(UnixTimestampConverter))]
public DateTime Born { get; set; }
```

Albo, jeśli API utrzymuje konsekwencję w zakresie reprezentacji typów danych, możemy pozwolić konwerterowi działać w sposób domyślny:

```
JsonSerializerOptions opts = new JsonSerializerOptions();
opts.Converters.Add (new UnixTimestampConverter());
var sara = JsonSerializer.Deserialize<Person> (json, opts);
```

W drugim przypadku poleciliśmy serializatorowi używać obiektu `UnixTimestampConverter` za każdym razem, gdy napotka obiekt `DateTime`.

Obsługa wartości null (od .NET 5)

W ramach optymalizacji serializator i deserializator normalnie omijają niestandardowe konwertery, kiedy napotykają wartości `null`. Jeśli niestandardowy konwerter ma je obsługiwać, należy przesłonić własność `HandleNull`:

```
public class UnixTimestampConverter : JsonConverter<DateTime?>
{
    public override bool HandleNull => true;
...
```

Technika ta działa w przypadku zarówno wartości `null` typów referencyjnych (np. łańcuchów), jak i typów wartościowych dopuszczających `null` (jak w przytoczonym przykładzie).

Opcje serializacji JSON

Serializator przyjmuje opcjonalny parametr `JsonSerializationOptions`, zapewniający dodatkową kontrolę nad procesem serializacji i deserializacji. Poniżej opisujemy najbardziej przydatne opcje.

Od .NET 5 domyślne opcje serializacji różnią się w zależności od tego, czy dana aplikacja jest sieciowa, czy nie.

W poniższej tabeli zebrano opcje, które różnią się w zależności od rodzaju aplikacji:

Opcja	Normalna domyślna	Domyślna dla aplikacji sieciowej
`PropertyNameCaseInsensitive`	`False`	`True`
`PropertyNamingPolicy`	(brak)	`CamelCase`
`NumberHandling`	`Strict`	`AllowReadingFromString`

Niezależnie od rodzaju aplikacji poniższe statyczne własności dają dostęp do każdego zestawu ustawień domyślnych:

```
JsonSerializerDefaults.Default
JsonSerializerDefaults.Web
```

Każdy z nich można sklonować za pomocą konstruktora przyjmującego inny obiekt `JsonSeria
↪lizationOptions`:

```
var options = new JsonSerializerOptions (JsonSerializerDefaults.Web);
```

WriteIndented

We wszystkich przykładach w tym podrozdziale opcję `WriteIndented` ustawialiśmy na `true`, aby serializator generował białe znaki, które zwiększają czytelność danych w formacie JSON dla człowieka. Domyślne ustawienie to `false`, czyli wszystko znajduje się w jednym wierszu.

AllowTrailingCommas

Specyfikacja formatu JSON wymaga, aby własności i elementy tablicy były oddzielone przecinkami, ale zabrania stawiania przecinka na samym końcu:

```
{
  "Name":"Dylan",
  "LuckyNumbers": [10, 7, ],
  "Age":46,
}
```

Domyślnie przecinki za liczbami `7` i `46` są niedozwolone. Aby można było je zostawić, należy zrobić to:

```
var commaTolerant = JsonSerializer.Deserialize<Person> (brokenJson,
new JsonSerializerOptions { AllowTrailingCommas = true });
```

ReadCommentHandling

Domyślnie deserializator zgłasza wyjątek, gdy napotka komentarz (ponieważ komentarze nie należą do oficjalnego standardu JSON). Ustawienie parametru `ReadCommentHandling` na `JsonCommentHandling.Skip` nakazuje deserializatorowi pomijać je, dzięki czemu poniższe dane zostaną prawidłowo przetworzone:

```
{
  "Name":"Dylan" // komentarz
  /* To jest kolejny komentarz. */
}
```

PropertyNameCaseInsensitive

Domyślnie deserializator rozróżnia wielkość liter przy dopasowywaniu nazw własności JSON do nazw własności C#. To znaczy, że poniższe dane:

```
{ "name":"Dylan" }
```

nie zostałyby wstawione do własności `Name` naszej klasy `Person` (ta własność JSON zostałaby zignorowana).

Ustawienie parametru PropertyNameCaseInsensitive na true rozwiązuje ten problem przez nakazanie deserializatorowi ignorowania wielkości liter przy dopasowywaniu nazw (niewielkim kosztem w zakresie wydajności):

```
var dylan = JsonSerializer.Deserialize<Person> (json,
new JsonSerializerOptions { PropertyNameCaseInsensitive = true });
```

Jeśli wielkość liter w danych wejściowych jest przewidywalna, to można także użyć atrybutu JsonPropertyName (opisanego wcześniej) lub opcji PropertyNamingPolicy (opisanej w następnej kolejności).

Od .NET 5 domyślną wartością ustawienia jest true w przypadku aplikacji sieciowych (i false w pozostałych przypadkach).

PropertyNamingPolicy

Aby udoskonalić obsługę nazw w popularnej notacji wielbłądziej, w .NET Core 3 wprowadzono atrybut PropertyNamingPolicy, który jest lepszy pod względem wydajności od poprzednio opisanej opcji PropertyNameCaseInsensitive i może być stosowany zarówno do serializacji, jak i deserializacji. W efekcie poniższy kod:

```
var dylan = new Person { Name = "Dylan" };

var json = JsonSerializer.Serialize (dylan,
  new JsonSerializerOptions
  {
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
  });
```

da następujący wynik:

```
{"name": "Dylan"}
```

W ten sam sposób można go zdeserializować:

```
var dylan2 = JsonSerializer.Deserialize<Person> (json,
  new JsonSerializerOptions
  {
    PropertyNamingPolicy = JsonNamingPolicy.CamelCase
  });
```

Od .NET 5 opcja CamelCase jest domyślna dla aplikacji sieciowych.

DictionaryKeyPolicy

Za pomocą opcji DictionaryKeyPolicy można wymusić serializację i deserializację kluczy słownika w notacji wielbłądziej:

```
var dict = new Dictionary<string, string>
{
  { "BookName", "Nutshell" }
  { "BookVersion", "9.0" },
};

Console.WriteLine (JsonSerializer.Serialize (dict,
```

```
  new JsonSerializerOptions
  {
    WriteIndented = true,
    DictionaryKeyPolicy = JsonNamingPolicy.CamelCase
  }));
```

Wynik działania tego kodu jest następujący:

```
{
  "bookName": "Nutshell"
  "bookVersion": "9.0",
}
```

Koder

Domyślny koder tekstu stosuje ostre zasady zastępowania znaków specjalnych kodami zastępczymi, aby wynik jego działania można było wprowadzić do dokumentu HTML bez dodatkowej obróbki:

```
string dylan = "<b>Dylan & Friends</b>";
Console.WriteLine (JsonSerializer.Serialize (dylan));
```

Wynik:

```
"\u003Cb\u003EDylan \u0026 Friends\u003C/b\u003E"
```

Można to zmienić przez wybranie innego kodera:

```
Console.WriteLine (JsonSerializer.Serialize (dylan,
  new JsonSerializerOptions {
    Encoder = JavaScriptEncoder.UnsafeRelaxedJsonEscaping
}));
```

Teraz wynik będzie następujący:

```
"<b>Dylan & Friends</b>"
```

`UnsafeRelaxedJsonEscaping` to podklasa klasy `System.Text.Encodings.Web.JavaScriptEncoder`. W razie potrzeby możesz zaimplementować własną podklasę, aby mieć pełną kontrolę nad procesem kodowania.

IgnoreNullValues

Domyślnie wartości `null` własności są uwzględniane w danych JSON:

```
var person = new Person { Name = null };
```

Wynik:

```
{
  "Name": null
}
```

Ustawienie opcji `IgnoreNullValues` na `true` powoduje, że własności o wartości `null` są ignorowane:

```
Console.WriteLine (JsonSerializer.Serialize (person),
  new JsonSerializerOptions { IgnoreNullValues = true } ));
```

Wynik:

```
{}
```

IgnoreReadOnlyProperties

Domyślnie własności tylko do odczytu są serializowane (ale nie deserializowane z powodu braku metody dostępowej `set`). Przez ustawienie `IgnoreReadOnlyProperties` na `true` można nakazać serializatorowi ich ignorowanie.

NumberHandling (od .NET 5)

W JSON normalnie liczb nie ujmuje się w cudzysłów, ale nie wszystkie formatery JSON przestrzegają tej zasady. Dlatego w .NET 5 wprowadzono opcję włączającą obsługę odczytu i zapisu liczb w cudzysłowach:

```
var options = new JsonSerializerOptions
{
    WriteIndented = true,
    NumberHandling = JsonNumberHandling.AllowReadingFromString |
                     JsonNumberHandling.WriteAsString
};

string json = JsonSerializer.Serialize (new Point { X = 2, Y = 3}, options);
Console.WriteLine (json);
var p2 = (Point) JsonSerializer.Deserialize (json, typeof (Point), options);
public class Point
{
    public int X { get; set; }
    public int Y { get; set; }
}

Wynik:
{
    "X": "2",
    "Y": "3"
}
```

W aplikacjach sieciowych domyślnym ustawieniem jest `JsonNumberHandling.AllowReading` ↪`FromString`, co znaczy, że liczby można wczytywać bezpiecznie bez względu na to, czy znajdują się w cudzysłowie, czy nie.

Serializacja kontraktu danych

Serializacja kontraktu danych obsługuje model **kontraktu danych** pomagający w usunięciu powiązania między niskiego poziomu szczegółami dotyczącymi typu przeznaczonego do serializacji i strukturą serializowanych danych. Ponadto oferuje doskonałą tolerancję na wersję, co oznacza możliwość deserializacji danych, które były serializowane z wcześniejszej lub późniejszej wersji typu. Istnieje nawet możliwość zmiany nazwy bądź przeniesienia deserializowanych typów do zupełnie innego zestawu.

Serializacja kontraktu danych może sobie poradzić z większością drzew obiektów, choć jednocześnie będzie wymagała większej pomocy niż serializacja binarna. Można ją także wykorzystać w charakterze ogólnego przeznaczenia narzędzia do odczytu i zapisu plików XML, jeżeli nie mamy problemów w pracy ze strukturą dokumentów XML. (Jeśli zachodzi potrzeba przechowywania danych w atrybutach lub radzenia sobie z elementami XML przedstawionymi w dowolnej kolejności, wówczas nie można zastosować omawianej tutaj serializacji kontraktu danych).

Podstawy

Poniżej przedstawiliśmy podstawowe kroki pozwalające na użycie serializacji kontraktu danych:

1. Wybór klasy: `DataContractSerializer` lub `DataContractJsonSerializer` (na platformie .NET Framework dostępna jest też klasa `NetDataContractSerializer`).
2. Udekorowanie typów i elementów składowych, które mają być serializowane. W tym celu wykorzystujemy atrybuty odpowiednio `[DataContract]` i `[DataMember]`.
3. Utworzenie egzemplarza serializatora oraz wywołanie jego metod `WriteObject()` i `ReadObject()`.

Jeżeli wybierzemy `DataContractSerializer`, konieczne będzie również zarejestrowanie „znanych typów" (podtypów, które też mogą być serializowane) i ustalenie, czy zachowywać odwołania do obiektów.

Konieczne może być także podjęcie specjalnej akcji w celu zapewnienia prawidłowej serializacji kolekcji.

Typy dla serializacji kontraktu danych zostały zdefiniowane w przestrzeni nazw `System.Runtime.Serialization`, w zestawie o takiej samej nazwie.

Wybór serializatora

Mamy trzy rodzaje serializatorów używanych w kontraktach danych:

`DataContractSerializer`

: Luźne połączenie typów .NET i typów kontraktu danych przez XML.

`DataContractJsonSerializer`

: Luźne połączenie typów .NET i typów kontraktu danych przez JSON.

`NetDataContractSerializer`

: Ścisłe połączenie typów .NET i typów kontraktu danych (tylko platforma .NET).

Dwa pierwsze wymagają wcześniejszego zarejestrowania podtypów możliwych do serializacji, aby można było mapować nazwę kontraktu danych, taką jak „Person", na odpowiedni typ .NET. Egzemplarz `NetDataContractSerializer` nie wymaga takiej operacji, ponieważ zapisuje pełne nazwy typu i zestawu podczas serializacji, podobnie jak silnik serializacji binarnej:

```
<Person z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
  ...
</Person>
```

To rozwiązanie opiera się na istnieniu określonego typu .NET w konkretnej przestrzeni danych i zestawie, aby można było przeprowadzić deserializację.

Jeżeli zapiszemy drzewo obiektu do czarnego pudełka, możemy wybrać serializator na podstawie korzyści, które uznamy za najważniejsze. W przypadku komunikacji za pomocą WCF bądź też odczytu i zapisu danych w pliku XML prawdopodobnie będziemy chcieli użyć `DataContractSerializer`.

Poruszonymi powyżej tematami znacznie dokładniej zajmiemy się nieco dalej w rozdziale.

Użycie serializatorów

Po wyborze serializatora kolejnym krokiem jest dołączenie atrybutów do typów oraz elementów składowych, które mają być serializowane. Absolutne minimum przedstawia się następująco:

- dodanie atrybutu `[DataContract]` do każdego typu;
- dodanie atrybutu `[DataMember]` do każdego elementu składowego, który ma zostać uwzględniony.

Oto przykład:

```
namespace SerialTest
{
  [DataContract] public class Person
  {
    [DataMember] public string Name;
    [DataMember] public int Age;
  }
}
```

Te atrybuty są wystarczające do przeprowadzenia *niejawnej* serializacji za pomocą silnika kontraktów danych.

Istnieje możliwość przeprowadzenia *jawnej* serializacji lub deserializacji egzemplarza przez utworzenie obiektu serializatora i wywołanie metody `WriteObject()` lub `ReadObject()`:

```
Person p = new Person { Name = "Staszek", Age = 30 };

var ds = new DataContractSerializer (typeof (Person));

using (Stream s = File.Create ("person.xml"))
  ds.WriteObject (s, p);                          // serializacja

Person p2;
using (Stream s = File.OpenRead ("person.xml"))
  p2 = (Person) ds.ReadObject (s);                // deserializacja

Console.WriteLine (p2.Name + " " + p2.Age);       // Staszek 30
```

Konstruktor obiektu `DataContractSerializer` wymaga typu **obiektu głównego** (typu obiektu jawnie serializowanego). Z kolei `NetDataContractSerializer` tego nie wymaga:

```
var ns = new NetDataContractSerializer();

// NetDataContractSerializer jest taki sam w użyciu
// jak DataContractSerializer
...
```

Domyślnie oba typy serializatorów używają formatera XML. W przypadku `XmlWriter` można żądać wcięcia danych wyjściowych, co poprawia ich czytelność:

```
Person p = new Person { Name = "Staszek", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

XmlWriterSettings settings = new XmlWriterSettings() { Indent = true };
using (XmlWriter w = XmlWriter.Create ("person.xml", settings))
  ds.WriteObject (w, p);

System.Diagnostics.Process.Start ("person.xml");
```

Poniżej przedstawiono wygenerowane dane wyjściowe:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Age>30</Age>
  <Name>Staszek</Name>
</Person>
```

Nazwa elementu XML `<Person>` odzwierciedla **nazwę kontraktu danych**, którą domyślnie jest nazwa typu .NET. Istnieje możliwość zmiany tej nazwy i jej jawnego podania w pokazany poniżej sposób:

```
[DataContract (Name="Candidate")]
public class Person { ... }
```

Przestrzeń nazw XML odzwierciedla **przestrzeń nazw kontraktu danych**, którą domyślnie jest `http://schemas.datacontract.org/2004/07/` plus przestrzeń nazw typu .NET. Także w tym przypadku można wprowadzić zmianę w pokazany poniżej sposób:

```
[DataContract (Namespace="http://oreilly.com/nutshell")]
public class Person { ... }
```

Podanie nazwy i przestrzeni nazw powoduje zerwanie powiązania tożsamości kontraktu z nazwą typu .NET. Dzięki temu jeśli później będziemy musieli przeprowadzić refaktoryzację i zmienić nazwę lub przestrzeń nazw, serializacja na tym nie ucierpi.

Oczywiście można również zmienić nazwy elementów składowych danych, jak pokazano poniżej:

```
[DataContract (Name="Candidate", Namespace="http://oreilly.com/nutshell")]
public class Person
{
  [DataMember (Name="FirstName")]  public string Name;
  [DataMember (Name="ClaimedAge")] public int Age;
}
```

Poniżej przedstawiono wygenerowane dane wyjściowe:

```
<?xml version="1.0" encoding="utf-8"?>
<Candidate xmlns="http://oreilly.com/nutshell"
           xmlns:i="http://www.w3.org/2001/XMLSchema-instance" >
  <ClaimedAge>30</ClaimedAge>
  <FirstName>Staszek</FirstName>
</Candidate>
```

Atrybut `[DataMember]` obsługuje zarówno elementy składowe, jak i właściwości publiczne i prywatne. Typ danych właściwości lub elementu składowego może być którymś z wymienionych poniżej:

- dowolny typ prosty;
- wartość `DateTime`, `TimeSpan`, `Guid`, `Uri` i `Enum`;
- akceptujące wartości `null` wersje powyższych typów;
- `byte[]` (serializacja XML na postać base64);
- dowolny „znany” typ udekorowany przez `DataContract`;
- dowolny typ `IEnumerable` (zob. sekcję „Serializacja kolekcji” w dalszej części rozdziału);

- dowolny typ wraz z atrybutem [Serializable] lub implementujący interfejs ISerializable (zob. podrozdział „Rozszerzenie kontraktu danych” w dalszej części rozdziału);
- dowolny typ implementujący IXmlSerializable.

Określenie formatera binarnego (tylko .NET Framework)

Formatera binarnego można użyć wraz z DataContractSerializer lub NetDataContractSerializer. Procedura jest taka sama i wygląda następująco:

```
Person p = new Person { Name = "Staszek", Age = 30 };
var ds = new DataContractSerializer (typeof (Person));

var s = new MemoryStream();
using (XmlDictionaryWriter w = XmlDictionaryWriter.CreateBinaryWriter (s))
  ds.WriteObject (w, p);

var s2 = new MemoryStream (s.ToArray());
Person p2;
using (XmlDictionaryReader r = XmlDictionaryReader.CreateBinaryReader (s2,
                               XmlDictionaryReaderQuotas.Max))
  p2 = (Person) ds.ReadObject (r);
```

Dane wyjściowe mogą się trochę różnić — będą nieco mniejsze od wygenerowanych przez formater XML i znacząco mniejsze, jeśli typy zawierają ogromne tablice.

Serializacja podklas

Nie trzeba podejmować żadnych specjalnych kroków, aby przeprowadzić serializację podklas za pomocą NetDataContractSerializer. Jedynym wymaganiem jest dodanie w podklasie atrybutu [DataContract]. Serializator zapisze w pełni kwalifikowane nazwy rzeczywistych typów poddawanych serializacji w przedstawiony poniżej sposób:

```
<Person ... z:Type="SerialTest.Person" z:Assembly=
  "SerialTest, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null">
```

Jednak egzemplarz DataContractSerializer trzeba poinformować o wszystkich podtypach, które mają być serializowane lub deserializowane. Aby to zilustrować, przyjmujemy założenie o utworzeniu podklasy klasy Person w przedstawiony poniżej sposób:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
}
[DataContract] public class Student : Person { }
[DataContract] public class Teacher : Person { }
```

Następnie przygotowujemy metodę klonującą klasę Person:

```
static Person DeepClone (Person p)
{
  var ds = new DataContractSerializer (typeof (Person));
  MemoryStream stream = new MemoryStream();
  ds.WriteObject (stream, p);
  stream.Position = 0;
  return (Person) ds.ReadObject (stream);
}
```

Nową metodę możemy wywołać w pokazany poniżej sposób:

```
Person  person  = new Person  { Name = "Staszek", Age = 30 };
Student student = new Student { Name = "Staszek", Age = 30 };
Teacher teacher = new Teacher { Name = "Staszek", Age = 30 };

Person  p2 =           DeepClone (person);     // OK
Student s2 = (Student) DeepClone (student);    // zgłoszenie wyjątku SerializationException
Teacher t2 = (Teacher) DeepClone (teacher);    // zgłoszenie wyjątku SerializationException
```

Metoda `DeepClone()` działa w przypadku wywołania jej z klasą `Person`, natomiast zgłasza wyjątek po wywołaniu z klasą `Student` lub `Teacher`, ponieważ deserializator w żaden sposób nie wie, jak zinterpretować typy .NET (zestawy) o nazwach `Student` i `Teacher`. Takie rozwiązanie zapewnia większe bezpieczeństwo, uniemożliwiając deserializację nieoczekiwanych typów.

Właściwym rozwiązaniem będzie podanie wszystkich dozwolonych (inaczej: „znanych") podtypów. Można to zrobić podczas tworzenia egzemplarza `DataContractSerializer`:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (Student), typeof (Teacher) } );
```

lub w samym typie za pomocą atrybutu `KnowType`:

```
[DataContract, KnownType (typeof (Student)), KnownType (typeof (Teacher))]
public class Person
...
```

Poniżej pokazano wynik serializacji klasy `Student`:

```
<Person xmlns="..."
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        i:type="Student" >
  ...
<Person>
```

Ponieważ klasa `Person` została podana jako typ główny, element główny nadal ma tę nazwę. Rzeczywista podklasa została opisana oddzielnie w atrybucie `type`.

Klasa `NetDataContractSerializer` będzie miała mniejszą wydajność podczas serializacji podklas, niezależnie od rodzaju użytego formatera. Wydaje się, że po napotkaniu podtypu klasa musi się na chwilę zatrzymać i ustalić, co zrobić dalej!

Wydajność działania serializacji ma znaczenie w aplikacji serwera obsługującej wiele jednoczesnych żądań.

Odwołania do obiektu

Odwołania do innych obiektów również są serializowane. Spójrz na poniższe klasy:

```
[DataContract] public class Person
{
  [DataMember] public string Name;
  [DataMember] public int Age;
  [DataMember] public Address HomeAddress;
}
```

```
[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Poniżej pokazano przykład serializacji tych klas na format XML z użyciem `DataContractSerializer`:

```
<Person...>
  <Age>...</Age>
  <HomeAddress>
    <Street>...</Street>
    <Postcode>...</Postcode>
  </HomeAddress>
  <Name>...</Name>
</Person>
```

Utworzona nieco wcześniej w rozdziale metoda `DeepClone()` będzie klonowała również `HomeAddress`. Należy odróżnić ją od prostej `MemberwiseClone()`.

Jeżeli używamy `DataContractSerializer`, wówczas podczas tworzenia podklas klasy `Address` mają zastosowanie te same reguły co w przypadku typu głównego. Dlatego też po zdefiniowaniu klasy `USAddress` np. w poniższy sposób:

```
[DataContract]
public class USAddress : Address { }
```

i przypisaniu jej egzemplarza do klasy `Person`:

```
Person p = new Person { Name = "Janek", Age = 30 };
p.HomeAddress = new USAddress { Street="Fawcett St", Postcode="02138" };
```

egzemplarz `p` nie może być serializowany. Rozwiązaniem jest użycie w egzemplarzu `Address` atrybutu `KnownType`:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}
```

lub poinformowanie klasy `DataContractSerializer` w jej konstruktorze o istnieniu `USAddress`:

```
var ds = new DataContractSerializer (typeof (Person),
  new Type[] { typeof (USAddress) } );
```

(Nie ma konieczności informowania o klasie `Address`, ponieważ jest to zadeklarowany typ elementu składowego `HomeAddress`).

Zachowanie odwołań do obiektu

Klasa `NetDataContractSerializer` zawsze sprawdza odwołania. Z kolei `DataContractSerializer` nie sprawdza odwołań, chyba że zlecimy przeprowadzenie takiej operacji.

Oznacza to, że jeśli do tego samego obiektu istnieją odwołania w dwóch różnych miejscach, egzemplarz `DataContractSerializer` po prostu zapisze go dwukrotnie. Jeżeli więc wcześniejszy przykład zmodyfikujemy tak, aby klasa `Person` przechowywała także adres firmy:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address HomeAddress, WorkAddress;
}
```

a następnie przeprowadzimy serializację w poniższy sposób:

```
Person p = new Person { Name = "Staszek", Age = 30 };
p.HomeAddress = new Address { Street = "Odo St", Postcode = "6020" };
p.WorkAddress = p.HomeAddress;
```

to te same informacje szczegółowe o adresie będą dwukrotnie umieszczone w pliku XML, jak widać w poniższym fragmencie kodu:

```
...
<HomeAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</HomeAddress>
...
<WorkAddress>
  <Postcode>6020</Postcode>
  <Street>Odo St</Street>
</WorkAddress>
```

Podczas późniejszej deserializacji `WorkAddress` i `HomeAddress` będą zupełnie odmiennymi obiektami. Zalety przedstawionego systemu to zachowanie prostoty danych w formacie XML oraz zgodność ze standardami. Natomiast wady to: większa ilość danych XML, utrata spójności odwołań oraz brak możliwości radzenia sobie z odwołaniami cyklicznymi.

Możesz zażądać integralności referencyjnej przez utworzenie obiektu klasy `DataContractSerializer`:

```
var settings = new DataContractSerializerSettings { PreserveObjectReferences = true };
var ds = new DataContractSerializer (typeof (Person), settings);
```

Możesz też określić maksymalną liczbę referencji do obiektów, jaką serializator ma monitorować, przez przypisanie własności `MaxItemsInObjectGraph` obiektu `settings`. Serializator zgłasza wyjątek, kiedy ta liczba zostaje przekroczona (to uniemożliwia przeprowadzenie ataku DoS za pomocą specjalnie spreparowanego strumienia).

Poniżej przedstawiono kod XML dla klasy `Person`, gdy adresy domowy i firmowy są takie same:

```
<Person xmlns="http://schemas.datacontract.org/2004/07/SerialTest"
        xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:z="http://schemas.microsoft.com/2003/10/Serialization/"
        z:Id="1">
  <Age>30</Age>
  <HomeAddress z:Id="2">
    <Postcode z:Id="3">6020</Postcode>
    <Street z:Id="4">Odo St</Street>
  </HomeAddress>
  <Name z:Id="5">Staszek</Name>
  <WorkAddress z:Ref="2" i:nil="true" />
</Person>
```

Kosztem jest zmniejszona interoperacyjność (zwróć uwagę na własnościową przestrzeń nazw atrybutów `Id` i `Ref`).

Tolerancja wersji

Elementy składowe można dodawać i usuwać bez obaw o niezachowanie zgodności. Domyślnie serializacja kontraktu danych działa w wymienione poniżej sposoby:

- pominięcie danych, które nie mają podanego w typie atrybutu `[DataMember]`;
- brak jakichkolwiek komunikatów ostrzeżeń lub błędów, gdy atrybutu `[DataMember]` zabraknie w strumieniu serializacji.

Zamiast pomijać nierozpoznane dane, można nakazać deserializatorowi przechowywanie ich w czarnym pudełku, a następnie ich zastosowanie podczas późniejszej ponownej serializacji typu. Tym samym można będzie prawidłowo obsłużyć dane serializowane przez nowszą wersję typu. Aby aktywować tę funkcję, należy zaimplementować interfejs `IExtensibleDataObject`. Ten interfejs tak naprawdę można odczytać jako „IDostawcaCzarnegoPudełka". Wymaga on implementacji pojedynczej właściwości pozwalającej na definiowanie i pobieranie wartości czarnego pudełka:

```
[DataContract] public class Person : IExtensibleDataObject{
  [DataMember] public string Name;
  [DataMember] public int Age;

  ExtensionDataObject IExtensibleDataObject.ExtensionData { get; set; }
}
```

Wymagane elementy składowe

Jeżeli element składowy ma ważne znaczenie dla typu, wówczas można się domagać jego obecności za pomocą `IsRequired`:

```
[DataMember (IsRequired=true)] public int ID;
```

W przypadku braku tak oznaczonego elementu składowego podczas deserializacji nastąpi zgłoszenie wyjątku.

Kolejność elementów składowych

W trakcie serializacji kontraktu danych ogromna uwaga jest zwracana na kolejność elementów składowych. Tak naprawdę deserializator *pomija wszystkie elementy składowe uznane za niezgodne z kolejnością*.

Podczas serializacji elementy składowe są zapisywane w następującej kolejności:

1. Od klasy bazowej do podklas.
2. Od najmniejszej do największej wartości `Order` (w przypadku elementów składowych, dla których zdefiniowano wartość `Order`).
3. W kolejności alfabetycznej (za pomocą *zwykłego* porównywania ciągów tekstowych).

Dlatego też w przedstawionym powyżej przykładzie element `Age` będzie się znajdował przed `Name`. Z kolei w poniższym fragmencie kodu element `Name` będzie serializowany przed `Age`:

```
[DataContract] public class Person
{
  [DataMember (Order=0)] public string Name;
  [DataMember (Order=1)] public int Age;
}
```

Jeżeli `Person` ma klasę bazową, wówczas wszystkie elementy składowe klasy bazowej będą serializowane jako pierwsze.

Podstawowym powodem podania kolejności jest zachowanie zgodności z określonym schematem XML. Kolejność elementu XML odpowiada kolejności elementu składowego danych.

Jeżeli nie trzeba zapewnić możliwości współdziałania z jakimkolwiek innym komponentem, najłatwiejsze podejście polega na *uniknięciu* definiowania atrybutu `Order` dla elementu składowego i pozostaniu wyłącznie przy kolejności alfabetycznej. W takim przypadku między serializowanymi i deserializowanymi danymi nigdy nie pojawi się jakakolwiek rozbieżność podczas dodawania i usuwania elementów składowych. Jedyny problem może wystąpić po przeniesieniu elementu składowego między klasą bazową i podklasą.

Wartości null i wartości puste

Mamy dwa sposoby pracy z elementami składowymi danych o wartości `null` lub wartości pustej:

1. Jawny zapis wartości `null` lub wartości pustej (podejście domyślne).
2. Pominięcie danego elementu składowego podczas serializacji.

W danych XML jawnie zapisana wartość `null` wygląda następująco:

```
<Person xmlns="..."
          xmlns:i="http://www.w3.org/2001/XMLSchema-instance">
  <Name i:nil="true" />
</Person>
```

Zapis elementów składowych o wartości `null` lub wartości pustej jest marnowaniem miejsca, zwłaszcza w przypadku typów o dużej liczbie elementów składowych bądź właściwości, które najczęściej są puste. Co więcej, być może trzeba będzie stosować się do schematu XML oczekującego użycia elementów opcjonalnych (np. `minOccurs="0"`) zamiast wartości `nil`.

Istnieje możliwość zakazania serializatorowi emisji elementów składowych danych zawierających wartości `null` lub wartości puste, co wymaga zastosowania poniższego podejścia:

```
[DataContract] public class Person
{
  [DataMember (EmitDefaultValue=false)] public string Name;
  [DataMember (EmitDefaultValue=false)] public int Age;
}
```

Element `Name` będzie pominięty, jeśli jego wartość wynosi `null`. Z kolei element `Age` będzie pominięty, jeśli jego wartość wynosi `0` (wartość domyślna dla typu `int`). Jeżeli `Age` to typ `int` akceptujący wartości `null`, wówczas ten element będzie pominięty tylko wtedy, gdy jego wartość wynosi `null`.

Podczas przywracania obiektu deserializator kontraktu danych pomija konstruktory typu oraz metody inicjalizacyjne elementów składowych. Dzięki temu można pominąć elementy składowe zgodnie z przedstawionym powyżej opisem, bez obaw o uszkodzenie elementów składowych, które mają niestandardowe wartości przypisane przez metodę inicjalizacyjną lub przez konstruktor. Aby to zilustrować, przyjmujemy założenie, że właściwości `Age` klasy `Person` przypisujemy wartość `30` w pokazany poniżej sposób:

```
[DataMember (EmitDefaultValue=false)]
public int Age = 30;
```

Teraz przyjmujemy założenia o utworzeniu egzemplarza klasy `Person`, jawnym przypisaniu jego właściwości `Age` wartości `0` i przeprowadzeniu serializacji. Wygenerowane dane wyjściowe nie będą zawierały właściwości `Age`, ponieważ wartość `0` jest domyślna dla typu `int`. Oznacza to, że podczas deserializacji właściwość `Age` zostanie zignorowana i ten element składowy będzie miał wartość domyślną, która na szczęście wynosi `0`. Metoda inicjalizacyjna i konstruktor tego elementu składowego zostały pominięte.

Kontrakty danych i kolekcje

Serializator kontraktu danych może zapisywać i przywracać kolekcje. Na przykład przyjmujemy założenie o zdefiniowaniu egzemplarza `Person` zawierającego listę adresów (`List<>`):

```
[DataContract] public class Person
{
  ...
  [DataMember] public List<Address> Addresses;
}

[DataContract] public class Address
{
  [DataMember] public string Street, Postcode;
}
```

Poniżej przedstawiono wynik serializacji klasy `Person` wraz z dwoma adresami:

```
<Person ...>
  ...
  <Addresses>
    <Address>
      <Postcode>6020</Postcode>
      <Street>Odo St</Street>
    </Address>
    <Address>
      <Postcode>6152</Postcode>
      <Street>Comer St</Street>
    </Address>
  </Addresses>
  ...
</Person>
```

Zwróć uwagę na fakt, że serializator nie zakodował żadnych informacji o konkretnym *typie* serializowanej kolekcji. Jeżeli element składowy `Addresses` byłby typu `Address[]`, wówczas otrzymalibyśmy dokładnie takie same dane wyjściowe. Pozwala to na zmianę typu kolekcji między serializacją i deserializacją bez spowodowania jakiegokolwiek problemu.

Jednak czasami zachodzi potrzeba, aby kolekcja była konkretnego typu. Ekstremalny przykład dotyczy interfejsów:

```
[DataMember] public IList<Address> Addresses;
```

Podobnie jak wcześniej serializacja powyższego obiektu przebiegnie prawidłowo, ale problemy pojawią się w trakcie deserializacji. Nie ma sposobu, aby deserializator mógł ustalić konkretny typ, więc wybiera najprostszą opcję, czyli tablicę. Deserializator pozostaje przy tej strategii nawet po zainicjalizowaniu elementu składowego z innym konkretnym typem:

```
[DataMember] public IList<Address> Addresses = new List<Address>();
```

(Pamiętaj, że deserializator pomija metody inicjalizacyjne elementów składowych). Rozwiązaniem jest zdefiniowanie elementu składowego danych jako prywatnego oraz dodanie publicznej właściwości pozwalającej na uzyskanie do niego dostępu:

```
[DataMember (Name="Addresses")] List<Address> _addresses;

public IList<Address> Addresses { get { return _addresses; } }
```

W nieco bardziej skomplikowanych aplikacjach właściwości i tak prawdopodobnie będą używane w powyższy sposób. Jedynym nietypowym rozwiązaniem jest tutaj oznaczenie jako prywatnego elementu składowego danych, a nie właściwości.

Elementy podklasy kolekcji

Elementy podklasy kolekcji są przez serializator obsługiwane w sposób niewidoczny. Konieczne jest zadeklarowanie prawidłowych podtypów, jakby były używane gdziekolwiek indziej:

```
[DataContract, KnownType (typeof (USAddress))]
public class Address
{
  [DataMember] public string Street, Postcode;
}

public class USAddress : Address { }
```

Dodajemy `USAddress` do listy adresów `Person`, co powoduje wygenerowanie danych XML podobnych do pokazanych poniżej:

```
...
  <Addresses>
    <Address i:type="USAddress">
      <Postcode>02138</Postcode>
      <Street>Fawcett St</Street>
    </Address>
  </Addresses>
```

Dostosowanie kolekcji i nazw elementów do własnych potrzeb

Jeżeli tworzymy podklasę samej klasy kolekcji, wówczas można dostosować do własnych potrzeb nazwę XML opisującą poszczególne elementy. W tym celu wystarczy skorzystać z atrybutu `CollectionDataContract`:

```
[CollectionDataContract (ItemName="Rezydencja")]
public class AddressList : Collection<Address> { }

[DataContract] public class Person
{
  ...
  [DataMember] public AddressList Addresses;
}
```

Oto wygenerowane dane wyjściowe:

```
...
  <Addresses>
    <Rezydencja>
      <Postcode>6020</Postcode>
      <Street>Odo St</Street>
    </Rezydencja>
    ...
```

Obiekt `CollectionDataContract` pozwala na zdefiniowanie właściwości `Namespace` i `Name`. Druga z wymienionych nie jest używana, gdy kolekcja będzie serializowana jako właściwość innego obiektu (jak w omawianym przykładzie), natomiast jest używana podczas serializacji kolekcji jako obiektu głównego.

Istnieje również możliwość użycia `CollectionDataContract` do kontrolowania serializacji słowników:

```
[CollectionDataContract (ItemName="Entry",
                         KeyName="Kind",
                       ValueName="Number")]
public class PhoneNumberList : Dictionary <string, string> { }

[DataContract] public class Person
{
  ...
  [DataMember] public PhoneNumberList PhoneNumbers;
}
```

Poniżej pokazano sformatowane dane wyjściowe:

```
...
  <PhoneNumbers>
    <Entry>
      <Kind>Home</Kind>
      <Number>08 1234 5678</Number>
    </Entry>
    <Entry>
      <Kind>Mobile</Kind>
      <Number>040 8765 4321</Number>
    </Entry>
  </PhoneNumbers>
```

Rozszerzenie kontraktu danych

W tym rozdziale dowiesz się, jak rozbudowywać możliwości serializacji kontraktu danych przez dodanie zaczepów serializacji, czyli `[Serializable]` i `IXmlSerializable`.

Zaczepy serializacji i deserializacji

Można zdefiniować, aby niestandardowa metoda została wykonana przed serializacją lub po serializacji. W tym celu metodę należy oznaczyć jednym z poniższych atrybutów:

`[OnSerializing]`

Wskazuje, że metoda powinna zostać wywołana tuż *przed* serializacją.

`[OnSerialized]`

Wskazuje, że metoda powinna zostać wywołana tuż *po* serializacji.

Podobne atrybuty są obsługiwane przez deserializację:

`[OnDeserializing]`

Wskazuje, że metoda powinna zostać wywołana tuż *przed* deserializacją.

`[OnDeserialized]`

Wskazuje, że metoda powinna zostać wywołana tuż *po* deserializacji.

Niestandardowa metoda musi mieć pojedynczy parametr typu `StreamingContext`. Ten parametr jest wymagany do zachowania spójności z silnikiem binarnym i nie jest używany przez serializację kontraktu danych.

Atrybuty `[OnSerializing]` i `[OnDeserialized]` są użyteczne podczas obsługi elementów składowych, które są poza możliwościami silnika kontraktu danych, np. kolekcji posiadających dodatkowe dane lub nieimplementujących interfejsów standardowych. Poniżej pokazano najprostsze podejście:

```
[DataContract] public class Person
{
  public TypNieprzyjaznySerializacji Addresses;

  [DataMember (Name="Addresses")]
  TypPrzyjaznySerializacji _serializationFriendlyAddresses;

  [OnSerializing]
  void PrepareForSerialization (StreamingContext sc)
  {
    // kopiowanie Addresses—> _serializationFriendlyAddresses
    // ...
  }

  [OnDeserialized]
  void CompleteDeserialization (StreamingContext sc)
  {
    // kopiowanie _serializationFriendlyAddresses—> Addresses
    // ...
  }
}
```

Metoda oznaczona atrybutem `[OnSerializing]` może być również używana w celu warunkowej serializacji elementów składowych, jak w poniższym fragmencie kodu:

```
public DateTime DateOfBirth;

[DataMember] public bool Confidential;

[DataMember (Name="DateOfBirth", EmitDefaultValue=false)]
DateTime? _tempDateOfBirth;

[OnSerializing]
void PrepareForSerialization (StreamingContext sc)
{
  if (Confidential)
    _tempDateOfBirth = DateOfBirth;
  else
    _tempDateOfBirth = null;
}
```

Przypomnij sobie, że deserializatory kontraktów danych pomijają konstruktory i metody inicjalizacyjne elementów składowych. Metoda oznaczona atrybutem `[OnDeserializing]` działa w charakterze pseudokonstruktora dla deserializacji i okazuje się użyteczna podczas inicjalizacji elementów składowych, które zostały wykluczone z serializacji:

```
[DataContract] public class Test
{
  bool _editable = true;

  public Test() { _editable = true; }

  [OnDeserializing]
  void Init (StreamingContext sc)
  {
    _editable = true;
  }
}
```

Jeżeli nie byłoby takiego rozwiązania dla metody `Init()`, wówczas `_editable` przyjmie wartość `false` w deserializowanym egzemplarzu `Test` pomimo dwóch innych prób przypisania tej właściwości wartości `true`.

Metody udekorowane czterema omówionymi powyżej atrybutami mogą być prywatne. Jeżeli podtypy wymagają ich użycia, mogą zdefiniować własne metody z tymi samymi atrybutami i one również zostaną wykonane.

Współpraca z atrybutem [Serializable]

Serializator kontraktu danych może również serializować typy oznaczone interfejsami i atrybutami silnika serializacji binarnej. Tego rodzaju możliwość jest ważna, ponieważ obsługa silnika binarnego została bardzo mocno wpleciona w to, co zostało utworzone przed wydaniem .NET Framework 3.0, włączając w to samą platformę .NET!

Poniższe konstrukcje powodują oznaczenie typu jako możliwego do serializacji za pomocą silnika binarnego:

- atrybut [Serializable];
- implementacja interfejsu `ISerializable`.

Binarna interoperacyjność jest użyteczna podczas serializacji istniejących typów, jak również dla nowych typów wymagających obsługi obu silników serializacji. Oferuje także inny aspekt rozszerzenia możliwości serializacji kontraktu danych, ponieważ interfejs `ISerializable` silnika binarnego jest znacznie elastyczniejszy niż interfejs atrybutu kontraktu danych. Niestety, serializer kontraktu danych jest nieefektywny pod względem formatowania danych dodawanych za pomocą `ISerializable`.

Typ, który chce korzystać z najlepszych możliwości obu światów, nie może definiować atrybutów dla obu silników. Stanowi to problem dla typów takich jak `string` i `DateTime`, które z powodów historycznych nie mogą porzucić atrybutów silnika binarnego. Serializator kontraktu danych radzi sobie z tym przez filtrowanie tego rodzaju typów prostych i przetwarzanie ich w specjalny sposób. W przypadku wszystkich typów oznaczonych do serializacji binarnej serializator kontraktu danych stosuje podobne reguły, jakie byłyby zastosowane przez silnik binarny. Oznacza to honorowanie atrybutów takich jak `NonSerialized` lub użycie interfejsu `ISerializable`, jeśli jest zaimplementowany. Nie jest to *podprocedura* samego silnika binarnego, ale gwarantuje, że dane wyjściowe będą sformatowane w takim samym stylu jak dane w przypadku użycia atrybutów kontraktu danych.

Typy zaprojektowane do serializacji za pomocą silnika binarnego oczekują zachowania odwołań do obiektu. Tę opcję można włączyć za pomocą klasy `DataContract` ↪`Serializer` (lub przez użycie egzemplarza klasy `NetDataContractSerializer`).

Reguły dotyczące rejestracji znanych typów mają również zastosowanie do obiektów i podobiektów serializowanych za pomocą interfejsów binarnych.

W poniższym fragmencie kodu zaprezentowano klasę wraz z elementem składowym danych udekorowanym przez atrybut `[Serializable]`:

```
[DataContract] public class Person
{
  ...
  [DataMember] public Address MailingAddress;
}
[Serializable] public class Address
{
  public string Postcode, Street;
}
```

Oto wynik serializacji wspomnianego obiektu:

```
<Person ...>
  ...
  <MailingAddress>
    <Postcode>6020</Postcode>
    <Street>Odo St</Street>
  </MailingAddress>
  ...
```

Jeżeli `Address` zaimplementuje interfejs `ISerializable`, wówczas wynik będzie sformatowany nieco mniej efektywnie:

```
<MailingAddress>
  <Street xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">str</Street>
```

```
  <Postcode xmlns:d3p1="http://www.w3.org/2001/XMLSchema"
    i:type="d3p1:string" xmlns="">pcode</Postcode>
</MailingAddress>
```

Interoperacyjność z IXmlSerializable

Ograniczeniem serializacji kontraktu danych jest niewielka kontrola nad strukturą danych w formacie XML. W aplikacji WCF może to być zaletą, ponieważ znacznie ułatwia infrastrukturze zachowanie zgodności z protokołami komunikatów.

Aby zachować większą kontrolę nad danymi XML, można zaimplementować `IXmlSerializable`, a następnie wykorzystać `XmlReader` i `XmlWriter` w celu ręcznego wczytania i zapisu danych w formacie XML. Serializator kontraktu danych pozwala na zastosowanie takiego rozwiązania względem typów wymagających większego poziomu kontroli. Interfejs `IXmlSerializable` zostanie dokładnie omówiony na końcu rozdziału.

Serializator binarny

Mechanizm serializacji binarnej jest niejawnie używany przez interfejs Remoting. Można go wykorzystać do przeprowadzania operacji takich jak zapis obiektów na dysku oraz przywracanie ich. Serializacja binarna jest wysoce zautomatyzowana i potrafi obsługiwać skomplikowane drzewa obiektów przy jedynie minimalnej interwencji. Jednak pozostaje niedostępna dla aplikacji przeznaczonych do umieszczenia w sklepie Windows Store.

Mamy dwa sposoby zapewnienia serializacji binarnej dla typu. Pierwszy jest oparty na atrybutach, natomiast drugi obejmuje implementację interfejsu `ISerializable`. Dodanie atrybutów jest prostsze, za to implementacja `ISerializable` charakteryzuje się większą elastycznością. Wymieniony interfejs najczęściej implementujemy w poniższych celach:

- dynamicznej kontroli tego, co będzie serializowane;
- ułatwienia zadania tworzenia przez inne komponenty podklas serializowanego typu.

Z korzystaniem z serializatora binarnego wiążą się poważne zastrzeżenia dotyczące bezpieczeństwa. Szczegółowe informacje znajdują się na stronie *https://aka.ms/binaryformatter*.

Podstawy

Typ można serializować po dodaniu do niego zaledwie jednego argumentu:

```
[Serializable] public sealed class Person
{
  public string Name;
  public int Age;
}
```

Atrybut `[Serializable]` nakazuje serializatorowi uwzględnienie wszystkich elementów składowych w danym typie. Obejmuje to elementy składowe zarówno prywatne, jak i publiczne (z wyłączeniem właściwości). Każdy element składowy sam musi być możliwy do serializacji, w przeciwnym razie nastąpi zgłoszenie wyjątku. Typy proste na platformie .NET, takie jak `string` i `int`, obsługują serializację (podobnie jak wiele innych typów .NET).

Atrybut [Serializable] nie jest dziedziczony, więc podklasy nie będą automatycznie umożliwiały serializacji, jeżeli nie zostaną oznaczone za pomocą omawianego atrybutu.

Aby przeprowadzić serializację egzemplarza Person, konieczne jest utworzenie egzemplarza formatera BinaryFormatter (z przestrzeni nazw System.Runtime.Serialization.Formatters.Binary) i wywołanie metody Serialize.

Na platformie .NET Framework dostępny jest także formater SoapFormatter, przy użyciu którego w taki sam sposób można wygenerować dane w formacie XML zgodnym z SOAP. Jest mniej funkcjonalny niż BinaryFormatter i nie obsługuje typów generycznych ani filtrowania zewnętrznych danych niezbędnego do tolerancji wersji przy serializacji.

W poniższym fragmencie kodu pokazano serializację obiektu Person za pomocą formatera BinaryFormatter:

```
Person p = new Person() { Name = "Grzegorz", Age = 25 };

IFormatter formatter = new BinaryFormatter();

using (FileStream s = File.Create ("serialized.bin"))
  formatter.Serialize (s, p);
```

Wszystkie dane niezbędne do przywrócenia obiektu Person zostały zapisane w pliku *serialized.bin*. Metoda Deserialize() powoduje przywrócenie obiektu:

```
using (FileStream s = File.OpenRead ("serialized.bin"))
{
  Person p2 = (Person) formatter.Deserialize (s);
  Console.WriteLine (p2.Name + " " + p.Age);      // Grzegorz 25
}
```

Deserializator pomija wszystkie konstruktory i inicjalizatory pól podczas odtwarzania obiektów. W tle następuje wywołanie metody FormatterServices.GetUni
↪nitializedObject w celu wykonania tego zadania. Wymienioną metodę można wywołać samodzielnie i tym samym zaimplementować inne wzorce projektowe!

Serializowane dane zawierają pełne informacje o typie i zestawie. Dlatego też próba rzutowania wyniku serializacji na dopasowany typ Person w innym zestawie spowoduje błąd. Deserializator w pełni przywraca odwołania do obiektu w ich pierwotnym stanie po deserializacji. Dotyczy to również kolekcji, które są traktowane jako możliwe do serializacji obiekty, podobnie jak każde inne (wszystkie typy kolekcji zdefiniowane w przestrzeniach nazw System.Collections.* są oznaczone jako możliwe do serializacji).

Silnik binarny może obsługiwać ogromne, skomplikowane drzewa obiektów bez konieczności specjalnej pomocy (oczywiście poza zagwarantowaniem, że wszystkie obiekty składowe są możliwe do serializacji). Trzeba jedynie mieć świadomość, że wydajność serializatora spada proporcjonalnie do liczby odwołań w drzewie obiektu. Może to być problemem w serwerze Remoting, który musi obsługiwać wiele jednoczesnych żądań.

Atrybuty serializacji binarnej

Atrybut [NonSerialized]

Domyślnie *wszystkie* pola są serializowane. Jeśli chcesz któreś pominąć w tym procesie, np. pola służące do wykonywania tymczasowych obliczeń lub przechowywania uchwytów do plików albo okien, musisz je oznaczyć za pomocą atrybutu `[NonSerialized]`:

```
[Serializable] public sealed class Person
{
  public string Name;
  [NonSerialized] public int Age;
}
```

W powyższym fragmencie kodu nakazaliśmy serializatorowi zignorowanie elementu składowego `Age`.

Nieserializowane elementy składowe zawsze są puste lub mają wartość `null` po deserializacji, nawet jeśli metody inicjalizacyjne lub konstruktory powodują przypisanie innej wartości.

Atrybut [OnDeserializing]

Metoda z atrybutem `[OnDeserializing]` jest wykonywana tuż przed deserializacją i stanowi coś w rodzaju konstruktora. Jej działanie może być bardzo istotne, ponieważ deserializator binarny omija wszystkie normalne konstruktory i inicjalizatory pól.

W poniższym przykładzie definiujemy pole o nazwie `Valid`, które możemy wykluczyć z serializacji za pomocą atrybutu `[NonSerialized]`:

```
public sealed class Person
{
  public string Name;
  [NonSerialized] public bool Valid = true;

  public Person() => Valid = true;
}
```

Deserializowany obiekt `Person` nie będzie miał własności `Valid` o wartości `true` pomimo istnienia konstruktora i metody inicjalizacyjnej przypisujących tę wartość. Rozwiązaniem tego problemu jest napisanie specjalnego **konstruktora** deserializacji:

```
[OnDeserializing]
void OnDeserializing (StreamingContext context) => Valid = true;
```

Atrybut [OnDeserialized]

Metoda z atrybutem `[OnDeserialized]` jest uruchamiana od razu *po* deserializacji. Przydaje się do aktualizowania obliczonych pól i w połączeniu z atrybutem `[OnSerializing]`, który opisujemy w następnej kolejności.

Atrybuty [OnSerializing] i [OnSerialized]

Atrybuty `[OnSerializing]` i `[OnSerialized]` oznaczają metodę jako przeznaczoną do wykonania, odpowiednio, przed serializacją i po serializacji.

Atrybut `[OnSerializing]` umożliwia nadanie wartości polu, które jest używane *tylko* do serializacji. Wyobraź sobie na przykład, że chcesz umożliwić serializację następującej klasy:

```
class Foo
{
  public XDocument Xml;
}
```

Trudność polega na tym, że klasa `XDocument` (z przestrzeni nazw `System.Xml.Linq`) nie jest serializowalna. Z problemem tym możemy sobie poradzić przez dodanie atrybutu `[NonSerialized]` do pola `Xml` i zdefiniowanie metody z atrybutem `[OnSerializing]` zapisującej zawartość obiektu `XDocument` do łańcucha (który można zserializować):

```
[Serializable]
class Foo
{
  [NonSerialized]
  public XDocument Xml;

  string _xmlString; // używane tylko do serializacji

  [OnSerializing]
  void OnSerializing (StreamingContext context)
    => _xmlString = Xml.ToString();
}
```

Ostatnim krokiem jest odtworzenie obiektu `XDocument` w trakcie deserializacji. Aby to zrobić, możemy posłużyć się metodą z atrybutem `[OnDeserialized]`:

```
[OnDeserialized]
void OnDeserialized (StreamingContext context)
  => Xml = XDocument.Parse (_xmlString);
```

Atrybut [OptionalField] i wersjonowanie

Dodanie lub usunięcie pola nie powoduje utraty zgodności z danymi już poddanymi serializacji, ponieważ serializator pominie dane, dla których nie znajdzie pasującego pola. Dodając pole, możesz zastosować następujący atrybut, który będzie przypominał, że może ono być nieobecne w danych zserializowanych przez starszą wersję programu.

```
[Serializable] public sealed class Person
{
  public string Name;
  [OptionalField (VersionAdded = 2)] public DateTime DateOfBirth;
}
```

Atrybut ten pełni funkcję dokumentacyjną i nie ma wpływu na semantykę serializacji.

Jeżeli niezawodność wersjonowania jest ważna, unikaj zmiany nazw i usuwania elementów składowych, a także retrospektywnego dodawania atrybutu `[NonSerialized]`. Nigdy nie zmieniaj typu pola.

Serializacja binarna przy użyciu interfejsu ISerializable

Po zaimplementowaniu interfejsu `ISerializable` otrzymujemy pełną kontrolę nad binarną serializacją i deserializacją typu.

Poniżej przedstawiono definicję interfejsu `ISerializable`:

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

Metoda `GetObjectData()` jest wywoływana przed serializacją. Do jej zadań należy m.in. wypełnienie obiektu `SerializationInfo` (słownik przechowujący pary nazwa-wartość) danymi pochodzącymi z wszystkich elementów składowych przeznaczonych do serializacji. Poniżej przedstawiliśmy wersję metody `GetObjectData()` przeznaczoną do serializacji dwóch elementów składowych: `Name` i `DateOfBirth`:

```
public virtual void GetObjectData (SerializationInfo info,
                                   StreamingContext context)
 {
   info.AddValue ("Name", Name);
   info.AddValue ("DateOfBirth", DateOfBirth);
 }
```

W omawianym przykładzie wybraliśmy dla każdego elementu nazwę zgodną z odpowiadającym jej elementem składowym. To nie jest wymóg, można użyć dowolnej nazwy, jeśli tylko ta sama nazwa będzie zastosowana także w deserializacji. Same wartości mogą być dowolnego typu możliwego do serializacji. Framework będzie rekurencyjnie prowadzić serializację, jeśli wystąpi potrzeba. Całkowicie poprawne jest przechowywanie w słowniku wartości `null`.

Dobrym rozwiązaniem będzie zdefiniowanie metody `GetObjectData()` jako wirtualnej (`virtual`), pod warunkiem że klasa nie jest zdefiniowana jako zapieczętowana (`sealed`). Umożliwia to podklasom rozszerzenie serializacji bez konieczności ponownego implementowania interfejsu.

Klasa `SerializationInfo` zawiera również właściwości, które można wykorzystać do kontrolowania typu oraz zestawu deserializowanych przez egzemplarz.

Poza implementacją interfejsu `ISerializable` typ kontroluje własne potrzeby w zakresie serializacji przez dostarczenie konstruktora deserializacji, który pobiera dwa takie same parametry jak metoda `GetObjectData()`. Konstruktor może być zadeklarowany wraz z dowolnym poziomem dostępności, a środowisko uruchomieniowe nadal będzie w stanie go odnaleźć. Jednak zwykle jest deklarowany jako chroniony (`protected`), aby mógł być wykonywany przez podklasy.

W poniższym przykładzie definiujemy klasy `Player` i `Team` zgodnie z zasadami niezmienności (wszystko jest tylko do odczytu). Ponieważ jednak niezmienne kolekcje nie mogą być serializowane, musimy przejąć kontrolę nad procesem serializacji przez zaimplementowanie interfejsu `ISerializable`:

```
[Serializable] public class Player
{
  public readonly string Name;
  public Player (string name) => Name = name;
}

[Serializable] public class Team : ISerializable
{
  public readonly string Name;
  public readonly ImmutableList<Player> Players; // bez możliwości serializacji!

  public Team (string name, params Player[] players)
  {
    Name = name;
    Players = players.ToImmutableList();
  }

  // serializacja obiektu
  public virtual void GetObjectData (SerializationInfo si,
                                     StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    // Konwertuje strukturę Players na zwykłą tablicę z możliwością serializacji:
    si.AddValue ("PlayerData", Players.ToArray());
  }

  // deserializacja obiektu
  protected Team (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");

    // deserializacja Players do tablicy pasującej do naszej serializacji:
    Player[] p = (Player[]) si.GetValue ("PlayerData", typeof (Player[]));

    // utworzenie nowej niezmiennej listy przy użyciu tej tablicy
    Players = p.ToImmutableList();
  }
}
```

(Problem ten można by było też rozwiązać przy użyciu opisanych wcześniej atrybutów `[OnSerializing]` i `[OnDeserialized]`).

W przypadku najczęściej używanych typów klasa `SerializationInfo` ma typowane metody `Get*()`, np. `GetString()` w celu ułatwienia tworzenia konstruktorów deserializacji. Jeżeli podamy nazwę dla nieistniejących danych, wówczas nastąpi zgłoszenie wyjątku. Taka sytuacja najczęściej zdarza się w przypadku niedopasowania między kodem odpowiedzialnym za serializację i deserializację. Przykładem może być dodanie nowego elementu składowego i zapomnienie o implikacjach związanych z deserializacją starego egzemplarza. W celu rozwiązania tego problemu można:

- dodać obsługę wyjątków dla kodu pobierającego element składowy dołączony w najnowszej wersji;
- zaimplementować własny system numerowania, jak w poniższym fragmencie kodu:

```
  public string MyNewField;

  public virtual void GetObjectData (SerializationInfo si,
                                     StreamingContext sc)
  {
    si.AddValue ("_version", 2);
    si.AddValue ("MyNewField", MyNewField);
    ...
  }

  protected Team (SerializationInfo si, StreamingContext sc)
  {
    int version = si.GetInt32 ("_version");
    if (version >= 2) MyNewField = si.GetString ("MyNewField");
    ...
  }
```

Tworzenie podklas klasy pozwalającej na serializację

W poprzednich przykładach zapieczętowaliśmy (`sealed`) klasy, które w zakresie serializacji opierały się na atrybutach. Jeżeli chcesz się dowiedzieć dlaczego, spójrz na poniższą hierarchię klas:

```
[Serializable] public class Person
{
  public string Name;
  public int Age;
}

[Serializable] public sealed class Student : Person
{
  public string Course;
}
```

W tym przykładzie klasy `Person` i `Student` umożliwiają serializację. Ponadto w zakresie serializacji obie wykorzystują domyślne zachowanie środowiska uruchomieniowego, ponieważ żadna z nich nie implementuje interfejsu `ISerializable`.

Teraz wyobraź sobie, że twórca klasy `Person` zadecydował z pewnych powodów o implementacji przez nią interfejsu `ISerializable` oraz dostarczył konstruktor deserializacji przeznaczony do kontrolowania serializacji klasy `Person`. Nowa wersja klasy `Person` może wyglądać następująco:

```
[Serializable] public class Person : ISerializable
{
  public string Name;
  public int Age;

  public virtual void GetObjectData (SerializationInfo si,
                                     StreamingContext sc)
  {
    si.AddValue ("Name", Name);
    si.AddValue ("Age", Age);
  }

  protected Person (SerializationInfo si, StreamingContext sc)
  {
    Name = si.GetString ("Name");
    Age = si.GetInt32 ("Age");
  }
  public Person() {}
}
```

Wprawdzie przedstawiona wersja działa dla egzemplarzy klasy `Person`, ale zmiana powoduje zepsucie serializacji egzemplarzy klasy `Student`. Serializacja egzemplarza klasy `Student` może się wydawać zakończona powodzeniem, choć element składowy `Course` tego obiektu nie będzie zapisany w strumieniu, ponieważ implementacja `ISerializable.GetObjectData()` w klasie `Person` nie ma żadnych informacji o elementach typu pochodnego `Student`. Ponadto deserializacja egzemplarzy klasy `Student` powoduje zgłoszenie wyjątku ze względu na to, że środowisko uruchomieniowe szuka (nieskutecznie) konstruktora deserializacji obiektu `Student`.

Rozwiązaniem problemu jest implementacja `ISerializable` od początku dla możliwych do serializacji klas, które są publiczne i niezapieczętowane. (W przypadku klas definiowanych jako `internal` nie ma to aż takiego znaczenia, ponieważ podklasy można później łatwo modyfikować, jeśli zachodzi potrzeba).

Jeżeli zaczęlibyśmy tworzyć klasę `Person` jak we wcześniejszym przykładzie, wówczas klasa `Student` byłaby utworzona w następujący sposób:

```
[Serializable]
public class Student : Person
{
  public string Course;

  public override void GetObjectData (SerializationInfo si,
                                      StreamingContext sc)
  {
    base.GetObjectData (si, sc);
    si.AddValue ("Course", Course);
  }

  protected Student (SerializationInfo si, StreamingContext sc)
    : base (si, sc)
  {
    Course = si.GetString ("Course");
  }
  public Student() {}
}
```