XML

Elaborazione di file XML.

Anno 2017/2018

Indice generale

1 Introduzione

Il tutorial rappresenta un'introduzione al linguaggio XML (*e**X**tensible **M**arkup **L**anguage*) e alle funzionalità fornite dal C# per l'elaborazione di documenti XML.

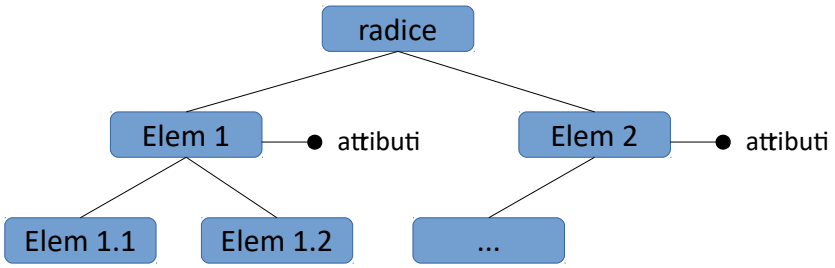
Qui non considero la natura di XML e il suo legame con i linguaggi che ne derivano (HTML, XAML, etc); mi limito a fornire le basi per la creazione di semplici documenti XML e la realizzazione di programmi in grado di processarli.

1.1 Documenti XML

XML definisce delle regole sintattiche per modellare la struttura di documenti e dati in modo da poter essere facilmente elaborati.

Un documento XML è un file di testo contenente **tag**, **attributi** e testo. Il documento è caratterizzato da una struttura di **elementi** (i tag); ciascun elemento può contenere altri elementi, oppure del semplice testo, che rappresenta l'informazione da memorizzare. Ad ogni elemento è possibile associare altre informazioni mediante degli **attributi**.

Gli elementi sono organizzati in una *struttura ad albero*, che prevede un elemento principale: ***root element*** o **radice**. L'intera struttura viene chiamata ***document tree*** e può essere schematizzata come in figura:



1.1.1 Esempio di un documento XML

Si vuole memorizzare i dati sui team e i piloti iscritti al campionato di Formula 1:

- i nomi dei team
- i nomi dei piloti
- il numero di macchina dei piloti.

Limitandoci ai primi due team del campionato, questi sono i dati da memorizzare:

> **Mercedes** – 1° pilota: **Hamilton, Lewis** (**44**); 2° pilota: **Rosberg, Nico** (**6**)
>
> **Ferrari** – 1° pilota: **Vettel, Sebastian** (**5**); 2° pilota: **Raikkonen, Kimi** (**7**)

Si vuole inoltre rappresentare la relazione di appartenenza dei piloti ai rispettivi team:

- esiste un elenco di team;
- ogni team ha due piloti.

XML consente facilmente di rappresentare questi dati nel seguente modo[1]:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Team iscritti al campionato di Formula 1 2015 -->
<Formula1>
  <Team nome="Ferrari">
    <Pilota numero="5">Vettel, Sebastian</Pilota>
    <Pilota numero="7">Raikkonen, Kimi</Pilota>
  </Team>
  <Team nome="Mercedes">
    <Pilota numero="44">Hamilton, Lewis</Pilota>
    <Pilota numero="6">Rosberg, Nico</Pilota>
  </Team>
</Formula1>
```

(I dati sono stati riquadrati allo scopo di evidenziarli.)

Nota bene:

- La prima riga stabilisce che si tratta di un documento XML: **dichiarazione XML**.
- La seconda riga è un commento.
- Esiste un elemento radice, `<Formula1>`, che contiene tutti gli altri.
- Esiste un elemento, `<Team>`, per ogni team. Il nome del team è specificato dall'attributo `nome`. Il valore dell'attributo è tra virgolette.
- Ogni elemento `<Team>` contiene due elementi `<Pilota>`.
- Il contenuto degli elementi `<Pilota>` è il nome del pilota. L'attributo `numero` specifica il numero di macchina del pilota.

1.2 Sintassi di un documento XML

Perché un documento XML sia ben formato deve rispettare determinate regole:

- Deve avere un solo elemento radice, il quale conterrà tutti gli altri elementi. All'esterno della radice sono ammessi soltanto **commenti** e **direttive** (per esempio, la **dichiarazione della versione di XML**).

```
<!-- questo è un commento -->
```

```
<?xml version="1.0" encoding="utf-8" ?>
```

- Un elemento è caratterizzato da un tag di apertura e un tag di chiusura. Il contenuto è collocato tra i due tag e può essere una stringa di testo oppure altri elementi.

```
<Elemento>CONTENUTO</Elemento>
```

- Un elemento può essere privo di contenuto (vuoto); in questo caso non ha bisogno del tag di chiusura e può utilizzare il suffisso `/` nel tag di apertura:

1 Quella utilizzata è soltanto una possibile organizzazione dei dati.

```
<ElementoVuoto/>
```

- Gli elementi possono avere degli attributi, specificati nel tag di apertura. I valori degli attributi devono essere tra virgolette:

```
<Dipendente nome="Pippo" cognome="Spada"/>
```

- XML fa distinzione tra maiuscole e minuscole, sia per i tag che per gli attributi. Il nome del tag di chiusura deve coincidere esattamente con il tag di apertura.
- Gli elementi devono essere nidificati correttamente. I tag di apertura e di chiusura non devono sovrapporsi.

Corretto

```
<Elemento>
    <SottoElemento>

    </SottoElemento>
  </Elemento>
```

Errato

```
<Elemento>
    <SottoElemento>

    </Elemento>     <-errore!
  </SottoElemento>
```

- Sia l'indentazione e che gli spazi tra gli elementi sono ininfluenti. Entrambi sono utili se il documento si presta ad essere scritto e/o letto da una persona.

Quelle sopraelencate sono le regole di base di XML. Il linguaggio non stabilisce il nome di elementi e attributi, o quanti debbano essere; né stabilisce quale struttura dare al documento per rappresentare correttamente i dati.

Ad esempio, in questa versione, entrambe le informazioni associate ai piloti sono rappresentate mediante attributi all'interno di elementi vuoti:

```
<?xml version="1.0" encoding="utf-8" ?>
<!-- Team iscritti al campionato di Formula 1 2015 -->
<Formula1>
  <Team nome="Ferrari">
    <Pilota nome="Vettel, Sebastian" numero="5"/>
    <Pilota nome="Raikkonen, Kimi" numero="7"/>
  </Team>
  <Team nome="Mercedes">
    <Pilota nome="Hamilton, Lewis" numero="44"/>
    <Pilota nome="Rosberg, Nico" numero="6"/>
  </Team>
</Formula1>
```

2 Elaborazione di file XML

.NET fornisce svariate classi per elaborare file XML; le più semplici da utilizzare sono definite nel namespace `System.Xml.Linq`. Queste sono state progettare per operare all'interno di **LINQ** (**L**anguage **In**tegrated **Q**uery), una funzionalità creata appositamente per interrogare collezioni di dati.

> **Linq to XML**
>
> Linq esiste in più varianti, in base alla provenienza dei dati da elaborare:
>
> - Collezione di oggetti (List, vettori, etc): **Linq to Object**.
> - Dati memorizzati in un database: **Linq to SQL / Linq to Entities**
> - Dati memorizzati in formato XML: **Linq to XML**.

Qui mi limito a introdurre i fondamenti necessari per elaborare file XML, senza prendere in esame le caratteristiche di LINQ.

2.1 Accedere al contenuto di un file XML

Le classi fondamentali per elaborare dati XML sono `XDocument`, `XElement` e `XAttibute`. La prima rappresenta un intero documento, la seconda un elemento (tag) XML, la terza un attributo.

Tipicamente, l'elaborazione del documento avviene in due fasi:

- Caricamento del documento.
- Accesso ai dati contenuti negli elementi.

2.1.1 Caricamento di un documento XML

Supponiamo di aver aggiunto al progetto il file **Formula1.xml** e di averne modificato le proprietà in modo che, durante la compilazione del programma, venga copiato nella cartella dell'eseguibile. Il caricamento del file in memoria si riduce al seguente codice:

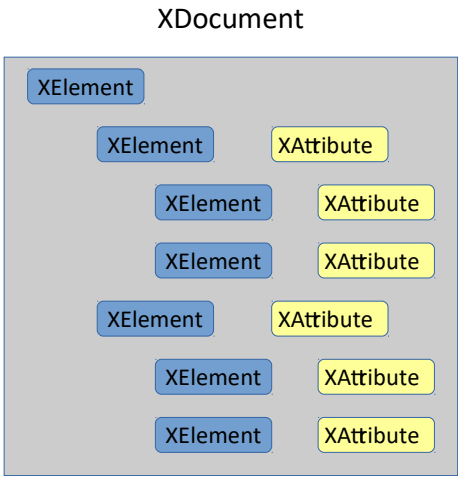
```
using System.Xml.Linq;
...
static void Main(string[] args)
{
    XDocument doc = XDocument.Load("Formula1.xml");
   ...
}
```

2.1.2 Struttura del documento

Una variabile `XDocument` memorizza l'intero contenuto del file e riflette la struttura gerarchica del documento[2]:

2 Quello in figura è uno schema semplificato della struttura.

```xml
<?xml version="1.0" encoding="utf-8" ?>
<Formula1>
  <Team nome="Ferrari">
    <Pilota numero="5">Vettel, Sebastian</Pilota>
    <Pilota numero="7">Raikkonen, Kimi</Pilota>
  </Team>
  <Team nome="Mercedes">
    <Pilota numero="44">Hamilton, Lewis</Pilota>
    <Pilota numero="6">Rosberg, Nico</Pilota>
  </Team>
</Formula1>
```



La struttura è simile a quella di un albero genealogico:

- Ogni elemento può avere elementi **figli** (*childs*).
- Ogni elemento, eccetto la radice, ha un **padre** (*parent*)
- Ogni elemento può avere **fratelli** (*siblings*), cioè elementi che hanno lo stesso padre.
- Ogni elemento può avere dei **discendenti** (*descendants*), e cioè elementi figli, nipoti, pronipoti, etc (Ad esempio: `<Pilota>` è discendente di `<Formula1>`).
- Ogni elemento può avere degli attributi.
- Nell'accesso a figli, discendenti, fratelli e attributi viene mantenuto l'ordine che essi hanno nel documento. (Esiste il primo figlio, il primo discendente, il primo attributo, etc).

2.1.3 Accesso agli elementi

`XDocument` e `XElement` forniscono i metodi necessari per accedere agli elementi della struttura, i quali sfruttano le "relazioni di parentela" sopraelencate. Tra i più comunemente utilizzati ci sono:

Metodo	Descrizione
Descendants()	Ritorna l'elenco di tutti i discendenti. Se viene specificato un nome, ritorna soltanto i discendenti con quel nome (o una lista vuota se non esistono)
Element()	Ritorna il primo figlio. Se viene specificato un nome, ritorna il primo figlio con quel nome, oppure `null` se non esiste
Elements()	Ritorna tutti i figli. Se viene specificato un nome, ritorna soltanto i figli con quel nome (o una lista vuota se non esistono)
Attribute()	Ritorna l'attributo corrispondente al nome specificato.
Attributes()	Ritorna tutti gli attributi.

Il codice seguente mostra come ottenere un elenco dei piloti iscritti al campionato (nota bene: il nome dell'elemento non comprende le parentesi angolari):

```
XDocument doc = XDocument.Load("Formula1.xml");
var piloti = doc.Descendants("Pilota");
```

`Descendants()` restituisce una variabile di tipo `Ienumerable<XElement>`, e cioè una sequenza di elementi che può essere scandita con `foreach`.

2.1.4 Accesso al contenuto di elementi e attributi

In XML i dati sono memorizzati negli elementi (tra il tag di apertura e quello di chiusura) e negli attributi e sono accessibili mediante la proprietà `Value` degli oggetti `Xelement`. Il codice seguente mostra come visualizzare i nomi dei piloti:

```
XDocument doc = XDocument.Load("Formula1.xml");
var list = doc.Descendants("Pilota");
foreach (var e in list)
{
    Console.WriteLine(e.Value);
}
```

Nota bene: se un elemento ne contiene degli altri, la proprietà `Value` memorizza tutti i sotto elementi, così come sono scritti nel documento, con tag e attributi compresi.

2.1.5 Accesso al valore degli attributi

Il metodo `Attribute()` restituisce un attributo, `XAttribute`, di un elemento. La proprietà `Value` memorizza il valore dell'attributo.

Il codice seguente mostra come visualizzare il numero di macchina del pilota oltre al suo nome:

```
XDocument doc = XDocument.Load("Formula1.xml");
var piloti = doc.Descendants("Pilota");
foreach (var e in piloti)   // "e" è di tipo XElement
{
    var att = e.Attribute("numero");   // "att" è di tipo XAttribute
    Console.WriteLine("{0,-20} N° {1}", e.Value, att.Value;);
}
```

2.2 Modifica dei dati associati agli elementi

Attraverso le classi `XDocument`, `XElement` e `XAttribute` è possibile modificare sia la struttura che i contenuti di un documento. Le modifiche effettuate possono essere salvate su disco mediante il metodo `Save()` dell'oggetto `Xdocument`:

```
XDocument doc = XDocument.Load("Formula1.xml");
//...modifiche al documento..
doc.Save("Formula1.xml");
```

Di seguito mi limito a considerare la modifica dei dati, memorizzati nelle proprietà `Value` di elementi e attributi.

2.2.1 Modifica del contenuto di un elemento: SetValue()

Il metodo `SetValue()` di `XElement` consente di sostituire il vecchio contenuto con uno nuovo, oppure di aggiungere il contenuto a un elemento vuoto. Il codice seguente modifica il nome del pilota con la macchina numero 7.

```
XDocument doc = XDocument.Load("Formula1.xml");
var piloti = doc.Descendants("Pilota");
foreach (var e in piloti)
{
    var att = e.Attribute("numero");
    if (att.Value == "7")
        e.SetValue("Perez, Sergio");
}

 doc.Save("Formula1.xml");
```

2.2.2 Modifica di un attributo: SetAttributeValue() e SetValue()

Anche la classe `XAttribute` definisce il metodo `SetValue()`, ma è possibile impostare un attributo direttamente dall'elemento usando `SetAttributeValue()`. Il seguente codice modifica il numero di macchina di “Sergio Perez”:

```
XDocument doc = XDocument.Load("Formula1.xml");
var piloti = doc.Descendants("Pilota");
foreach (var e in piloti)
{
    var att = e.Attribute("numero");
    if (att.Value == "7")
    {
        e.SetValue("Perez, Sergio");
        e.SetAttributeValue("numero", 77);
    }
}
doc.Save("Formula1.xml");
```

Nota bene: `SetAttributeValue()` richiede il nome dell'attributo e il nuovo valore. Se l'attributo non esiste, viene creato con il valore specificato.

3 Un esempio completo

In alcuni scenari, elaborare un file XML significa caricare in memoria i dati e costruire con essi un *modello a oggetti*. L'esempio seguente mostra come ottenere un modello a oggetti che rappresenti i team e i piloti memorizzati in **Formula1.xml**.

3.1 Progettare il modello a oggetti

Il disegno di un modello prescinde dalla loro memorizzazione in un file XML, o qualsiasi altro *storage*; occorre chiedersi soltanto in che modo rappresentare le informazioni di team e piloti, e le relazioni che intercorrono tra esse.

Nel modello utilizzato di seguito, a ogni tipo di elemento XML corrisponde una classe:

```
public class Formula1
{
    public Formula1()
    {
        ListTeam = new List<Team>();
    }
    public List<Team> ListTeam { get; private set; }
    public void AggiungiTeam(Team team)
    {
        ListTeam.Add(team);
    }
}

public class Team
{
    public string Nome { get; set; }
    public Pilota PrimoPilota { get; set; }
    public Pilota SecondoPilota { get; set; }
}

public class Pilota
{
    public string Nome { get; set; }
    public int NumeroMacchina { get; set; }
}
```

Alla radice del modello c'è un oggetto di tipo `Formula1`, dal quale è possibile ottenere le informazioni sui team e conseguentemente sui piloti.

3.2 Caricamento dei dati

L'obiettivo è quello di realizzare un metodo che produca un oggetto di tipo `Formula1`, contenente tutti i dati del file.

```
static Formula1 CaricaDatiFormula1(string nomeFile)
{
    Formula1 formula1 = new Formula1();
    XDocument xdoc = XDocument.Load("Formula1.xml");
    var xElencoTeam = xdoc.Descendants("Team");
    foreach (var xteam in xElencoTeam)
    {
        var team = new Team();
        team.Nome = xteam.FirstAttribute.Value;
        team.PrimoPilota = CaricaPilota(xteam.Element("Pilota"));
        team.SecondoPilota = CaricaPilota(xteam.Elements("Pilota").Last());
        formula1.AggiungiTeam(team);
    }
    return formula1;
}

static Pilota CaricaPilota(XElement xPilota)
{
    var pilota = new Pilota();
    pilota.Nome = xPilota.Value;
    pilota.NumeroMacchina = int.Parse(xPilota.FirstAttribute.Value);
    return pilota;
}
```

Il metodo `CaricaDatiFormula1()` ottiene innanzitutto la lista dei team; per ogni team crea l'oggetto corrispondente, valorizza le sue proprietà e lo aggiunge all'elenco. Il caricamento dei dati associati al pilota viene eseguito su un metodo a parte, `CaricaPilota()`, che riceve come argomento l'elemento XML corrispondente.

Il codice utilizza due funzionalità ancora non introdotte. La proprietà `FirstAttribute` contiene il primo attributo di un elemento. Il metodo `Last()` si applica a una lista (prodotta dal metodo `Elements()`) e ne ritorna l'ultimo elemento.

4 Creare un documento XML

La creazione di un documento XML si svolge in due fasi: si crea la struttura degli elementi in memoria (*XML tree*), quindi la si salva invocando il metodo `Save()` della classe `Xdocument`.

Alla base della creazione dell'*XML tree* c'è la classe `XElement`; questa definisce svariati costruttori, che possono essere impiegati per creare strutture anche complesse. Alternativamente, si può creare degli elementi vuoti e aggiungere successivamente valore, attributi e sotto elementi.

4.1 Creare una struttura predefinita

Se il documento deve contenere un numero predefinito di elementi, la struttura XML può essere prodotta facilmente mediante una singola istruzione. Supponiamo di voler memorizzare i dati di un record:

```
public class Settings
{
    public string HomePage { get; set; }
    public bool ShowBookmars { get; set; }
}
```

```
<?xml version="1.0" encoding="utf-8"?>
<Settings>
  <HomePage>www.isisfermi.it</HomePage>
  <ShowBookmarks>true</ShowBookmarks>
</Settings>
```

La struttura XML è composta da un elemento radice contenente due elementi figli; è possibile crearla mediante il seguente codice:

```
var xsettings = new XElement("Settings",
                    new XElement("HomePage", settings.HomePage),
                    new XElement("ShowBookmarks", settings.ShowBookmars));

var xdoc = new XDocument();
xdoc.Add(xsettings);  // aggiunge elemento radice al documento
xdoc.Save("Settings.xml");
```

Nell'esempio viene usato il costruttore di `Xelement`:

```
public XElement(XName name, params object[] content);
```

Questo accetta il nome dell'elemento (anche in forma di stringa) e un sequenza di uno o più sotto elementi e/o attributi.

Il precedente codice può essere riscritto in forma procedurale:

```
xsettings = new XElement("Settings");
xsettings.Add(new XElement("HomePage", settings.HomePage));
xsettings.Add(new XElement("ShowBookmarks", settings.ShowBookmars));
```

4.2 Creare una struttura corrispondente a un elenco

La memorizzazione di un elenco di dati non richiede funzionalità diverse da quelle introdotte. In questo caso, comunque, l'approccio procedurale può risultare più naturale.

Supponiamo di voler memorizzare un elenco di autori, ognuno dei quali definisce il nominativo e il numero di libri scritti. Vogliamo riprodurre la struttura XML a destra:

```
public class Autore
{
    public string Nominativo { get; set; }
    public int NumeroLibri { get; set; }
}
...
var autori = new List<Autore>()
{
    new Autore() {...},
    new Autore() {...},
    new Autore() {...},
    new Autore() {...}
};
```

```
<?xml version="1.0" encoding="utf-8"?>
<Autori>
  <Autore numeroLibri="30">
          Asimov, Isaac</Autore>
  <Autore numeroLibri="25">
          Van Vogt, Alfred </Autore>
  <Autore numeroLibri="18">
          Heinlein, Robert</Autore>
  <Autore numeroLibri="15">
          Clarke, Arthur</Autore>
</Autori>
```

Nota bene: ogni elemento **Autore** memorizza il nominativo come valore e il numero di libri come attributo. Segue il codice in grado di produrre il documento:

```
var xAutori = new XElement("Autori");
foreach (var autore in autori)
{
    var xa = new XElement("Autore");
    xa.SetValue(autore.Nominativo);
    xa.SetAttributeValue("numeroLibri", autore.NumeroLibri);
    xAutori.Add(xa);
}

XDocument xdoc = new XDocument();
xdoc.Add(xAutori);
xdoc.Save(fileName);
```

Nota bene: il metodo `SetAttributeValue()` crea l'attributo se questo non esiste.

4.2.1 Usare LINQ

Il codice precedente può essere riscritto in modo più conciso usando LINQ. L'obiettivo è creare una lista di elementi **Autore** a partire dalla lista degli autori; dopodiché si crea l'elemento radice **Autori** passandogli la lista appena creata:

```
var xListaAutori = autori.Select(a => new XElement("Autore", a.Nominativo,
                                      new XAttribute("numeroLibri", a.NumeroLibri)));

var xAutori = new XElement("Autori", xListaAutori);

XDocument xdoc = new XDocument();
xdoc.Add(xAutori);
xdoc.Save(fileName);
```