SOUBORY, VSTUPY A VÝSTUPY – POKRAČOVÁNÍ

Serializace objektů – pokračování

Serializace v binárním nebo SOAP formátu – pokračování

Přizpůsobení serializace – pokračování

Příklad

Program provádí serializaci a deserializaci třídy `Student`. Tato třída obsahuje datovou složku `fakulta` výčtového typu a `obor` typu třída `StudijniObor`. Implicitně se hodnota výčtového typu serializuje ve formátu SOAP v podobě názvu výčtové konstanty. Pokud se při deserializaci načte název neodpovídající žádné z výčtových konstant, vyvolá se výjimka. Aby k tomu nedošlo, datová složka `fakulta` se serializuje jako celé číslo. Při deserializaci se kontroluje, zda celé číslo odpovídá některé z výčtových konstant. Pokud ano, převede se na odpovídající výčtovou konstantu a uloží se do datové složky `fakulta`. Pokud ne, do datové složky `fakulta` se uloží implicitní výčtová konstanta.

```
public enum Fakulta { DFJP, FES, FCHT, FEI };

[Serializable]
public class StudijniObor
{
   string nazev;
   int cislo;
   public StudijniObor(string nazev, int cislo)
   {
      this.nazev = nazev;
      this.cislo = cislo;
   }
   public override string ToString()
   {
      return "obor: " + nazev + ", " + cislo;
   }
}
[Serializable]
public class Student : ISerializable
{
   private Fakulta fakulta;
   private StudijniObor obor;
   public Student(Fakulta fakulta, StudijniObor obor)
   {
      this.fakulta = fakulta;
      this.obor = obor;
   }
```

```
   protected Student(SerializationInfo info, StreamingContext context)
   {
      obor = (StudijniObor)info.GetValue("Obor", typeof(StudijniObor));
      int f = info.GetInt32("Fakulta");
      foreach (int item in Enum.GetValues(typeof(Fakulta))) {
         if (item == f) {
            fakulta = (Fakulta)f;
            return;
         }
      }
      fakulta = Fakulta.DFJP;
      Console.WriteLine("Byla nastavena implictní fakulta");
   }
   public void GetObjectData(SerializationInfo info,
                             StreamingContext context)
   {
      info.AddValue("Obor", obor);
      info.AddValue("Fakulta", (int)fakulta);
   }
   public void Vypis()
   {
      Console.WriteLine("fakulta: " + fakulta + ", " + obor);
   }
}
class Program
{
   static void Serializuj(IFormatter f, Student userConfig)
   {
      using (FileStream fs = File.Create("data.bin")) {
         f.Serialize(fs, userConfig);
      }
   }
   static Student Deserializuj(IFormatter f)
   {
      using (FileStream fs = File.OpenRead("data.bin")) {
         return (Student)f.Deserialize(fs);
      }
   }
   static void Main(string[] args)
   {
      Student student = new Student(Fakulta.FEI,
                                    new StudijniObor("AID", 10));
      //BinaryFormatter f = new BinaryFormatter();
      SoapFormatter f = new SoapFormatter();
      Serializuj(f, student);
      student.Vypis();
      student = Deserializuj(f);
      student.Vypis();
      Console.ReadKey();
   }
}
```

Proces serializace lze od verze .NET 2.0 také ovlivňovat metodami, které jsou deklarovány s následujícími atributy:

- `OnDeserialized` – metoda s tímto atributem se zavolá bezprostředně poté, co byl objekt deserializován.

- `OnDeserializing` – metoda s tímto atributem se zavolá bezprostředně před deserializací objektu.
- `OnSerialized` – metoda s tímto atributem se zavolá bezprostředně poté, co byl objekt serializován.
- `OnSerializing` – metoda s tímto atributem se zavolá bezprostředně před serializací objektu.

Metody s uvedenými atributy musí mít jeden parametr typu `StreamingContext` a vracet `void`.

Příklad

Program serializuje pole tříd `Bod`. Třída `Bod` má metodu `OnSerialized`, která je volána po serializaci bodu. Tato metoda vypíše na obrazovku pořadové číslo bodu, který byl serializován a jeho údaje.

Pořadové číslo bodu je uchováno ve vlastnosti `Pocet` třídy `Pocitadlo`, jejíž instance poskytuje vlastnost `Context` třídy `StreamingContext`, která je parametrem metody `OnSerialized`.

```
class Pocitadlo
{
   private int pocet;
   public int Pocet
   {
      get { return pocet; }
      set { pocet = value; }
   }
}
[Serializable]
class Bod
{
   int x;
   int y;
   public Bod(int x, int y)
   {
      this.x = x;
      this.y = y;
   }
   [OnSerialized]
   private void OnSerialized(StreamingContext context)
   {
      int pocet = ++((Pocitadlo)context.Context).Pocet;
      Console.WriteLine("{0}. bod ({1}, {2}) byl serializován",
        pocet, x, y);
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      const int n = 3;
      Bod[] body = new Bod[n];
      for (int i = 0; i < n; i++) {
         body[i] = new Bod(i, i * 10);
      }
      BinaryFormatter bf = new BinaryFormatter(null,
         new StreamingContext(StreamingContextStates.File,
                              new Pocitadlo()));
      using (FileStream fs = new FileStream("data.bin",
             FileMode.Create)){
         bf.Serialize(fs, body);
      }
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
1. bod (0, 0) byl serializován
2. bod (1, 10) byl serializován
3. bod (2, 20) byl serializován
```

Serializace v XML formátu

Pro serializaci objektů ve formátu XML slouží třída `System.Xml.Serialization.XmlSerializer`.

Serializují se nejen datové složky, ale i vlastnosti třídy (struktury), které obsahují obě přístupové metody (`get` i `set`).

Ukládané objekty musí splňovat následující požadavky:

- Třída (struktura), jejíž instance se má serializovat, musí obsahovat konstruktor bez parametrů a musí být veřejná. Konstruktor bez parametrů může být deklarován s libovolným modifikátorem přístupových práv, tj. může být i soukromý.
- Datové složky a vlastnosti, které se mají serializovat musí být deklarovány jako veřejné a musí být typu, který vyhovuje podmínkám uvedeným v první odrážce. To se týká i datových složek a vlastností objektu, na který se datová složka této třídy přímo nebo nepřímo odkazuje.
- Datové složky a vlastnosti, které se nemají serializovat, musí být buď neveřejné nebo se musí deklarovat s atributem `XmlIgnore`.

Instance třídy `XmlSerializer` se zpravidla vytváří pomocí konstruktoru

```
XmlSerializer(Type type)
```

Parametr `type` reprezentuje informace o objektu, který se má serializovat, získané např. operátorem `typeof`. Konstruktor kontroluje, zda typ, který se má serializovat, splňuje podmínky XML serializace. Pokud zjistí chybu, vyvolá výjimku typu `System.InvalidOperationException`.

Serializaci lze provést následujícími metodami:

- `void Serialize(Stream stream, object o)` – zapíše objekt `o` do datového proudu `stream`.

- `object Deserialize(Stream stream)` – vrací objekt, který načte z datového proudu `stream`.

Pokud během serializace vznikne chyba, uvedené metody vyvolají výjimku typu `System.InvalidOperationException`, obsahující případně vnitřní výjimku, popisující skutečnou chybu.

Do jednoho XML souboru (datového proudu) lze sice za sebou uložit více objektů samostatným voláním metod `Serialize`, ale při pozdějším čtení prvního objektu metodou `Deserialize` vznikne výjimka, protože soubor obsahuje dvakrát hlavičku XML.

Při deserializaci se objekt nejprve vytvoří voláním konstruktoru bez parametrů a potom se načtou jeho složky. Pokud některá složka, která se má serializovat, se nenačte, žádná výjimka nevznikne.

Všechny složky třídy jsou do XML souboru implicitně ukládány jako prvky XML, jejichž název odpovídá názvu složky třídy. Tento stav lze změnit pomocí dvou atributů:

- `XmlElement` – serializuje složku třídy jako prvek XML. Jedna z verzí tohoto atributu obsahuje název, pod kterým se má prvek serializovat.
- `XmlAttribute` – serializuje složku třídy jako atribut XML. Jedna z verzí tohoto atributu obsahuje název, pod kterým se má prvek serializovat.

Příklad

V příkladu se provádí serializace třídy `Osoby`, která obsahuje pole tříd `Osoba`. Třída `Osoba` obsahuje složky:

- datové složky `rodneCislo`, `jmeno` a `interniCislo`,
- vlastnosti `RodneCislo` a `JeMuz`.

Z uvedených složek se nebudou serializovat složky `rodneCislo`, `interniCislo` a `JeMuz`.

Vlastnost `RodneCislo` a jí odpovídající datová složka jsou typu struktura `RodneCislo`, jež obsahuje dvě datové složky `castA` a `castB`, které se budou serializovat.

```
public struct RodneCislo
{
   public int castA;
   public int castB;
   public RodneCislo(int castA, int castB)
   {
      this.castA = castA; this.castB = castB;
   }
   public override string ToString() { return castA + "/" + castB; }
}
public class Osoba
{
   private RodneCislo rodneCislo;
   [XmlAttribute]
   public string jmeno;
   [XmlIgnore]
   public int interniCislo = -1;
   [XmlElement("BirthNumber")]
   public RodneCislo RodneCislo
   {
      get { return rodneCislo; }
      set { rodneCislo = value; }
   }
   public bool JeMuz
   { get { return (rodneCislo.castA / 1000) % 10 <= 1; } }
```

```
   public Osoba() { }
   public Osoba(string jmeno, RodneCislo rodneCislo)
   {
      this.jmeno = jmeno; this.rodneCislo = rodneCislo;
   }
   public override string ToString()
   {
      return jmeno + ", rodné číslo: " + rodneCislo.ToString() +
             ", interní číslo: " + interniCislo;
   }
}
public class Osoby
{
   public Osoba[] pole;
   public Osoby() { }
   public Osoby(Osoba[] pole) { this.pole = pole; }
   public void Vypis()
   {
      foreach (Osoba item in pole) {
         Console.WriteLine(item.ToString());
      }
   }
}
class Program
{
   static void Main(string[] args)
   {
      Osoby osoby = new Osoby(new Osoba[] {
         new Osoba("Karel", new RodneCislo(661222, 1111)),
         new Osoba("Jana", new RodneCislo(665222, 2222)) });
      XmlSerializer xs = new XmlSerializer(typeof(Osoby));
      using (FileStream fs = new FileStream("data.xml",FileMode.Create)){
         xs.Serialize(fs, osoby);
      }
      osoby.Vypis();
      using (FileStream fs = new FileStream("data.xml", FileMode.Open)) {
         osoby = (Osoby)xs.Deserialize(fs);
      }
      Console.WriteLine("Po načtení objektu ze souboru");
      osoby.Vypis();
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
Karel, rodné číslo: 661222/1111, interní číslo: -1
Jana, rodné číslo: 665222/2222, interní číslo: -1
Po načtení objektu ze souboru
Karel, rodné číslo: 661222/1111, interní číslo: -1
Jana, rodné číslo: 665222/2222, interní číslo: -1
```

Soubor `data.xml` bude mít následující obsah:

```
<?xml version="1.0"?>
<Osoby xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <pole>
    <Osoba jmeno="Karel">
      <BirthNumber>
        <castA>661222</castA>
        <castB>1111</castB>
      </BirthNumber>
    </Osoba>
    <Osoba jmeno="Jana">
      <BirthNumber>
        <castA>665222</castA>
        <castB>2222</castB>
      </BirthNumber>
    </Osoba>
  </pole>
</Osoby>
```

Pokud se má serializovat kolekce prvků typu `object` (např. `object[]`, `ArrayList`), musí se použít konstruktor

```
XmlSerializer(Type type, Type[] extraTypes)
```

Parametr `extraTypes` reprezentuje pole informací o objektech, které mohou obsahovat prvky kolekce.

Příklad

Příklad vychází z předchozího příkladu. Liší se v tom, že místo třídy `Osoby` se používá pole `object[]`, které obsahuje prvky typu `Osoba`.

```
class Program
{
   static void Vypis(object[] osoby)
   {
      foreach (Osoba item in osoby) {
         Console.WriteLine(item.ToString());
      }
   }
```

```
   static void Main(string[] args)
   {
      object[] osoby = new object[] {
         new Osoba("Karel", new RodneCislo(661222, 1111)),
         new Osoba("Jana", new RodneCislo(665222, 2222)) };
      XmlSerializer xs = new XmlSerializer
         (typeof(object[]), new Type[] { typeof(Osoba) });
      using (FileStream fs = new FileStream("data.xml",FileMode.Create)){
         xs.Serialize(fs, osoby);
      }
      Vypis(osoby);
      osoby = null;
      using (FileStream fs = new FileStream("data.xml", FileMode.Open)) {
         osoby = (object[])xs.Deserialize(fs);
      }
      Console.WriteLine("Po načtení objektu ze souboru");
      Vypis(osoby);
      Console.ReadKey();
   }
}
```

FORMÁTOVÁNÍ

Formátování řetězců

Pro formátování řetězců znaků slouží statická metoda `Format` třídy `string`

```
string Format(string format, params object[] args)
```

případně její přetížené verze. Její tvar odpovídá jedné z verzí metod `Write` a `WriteLine` používaných v textových výstupech do konzoly nebo jiného datového proudu. Tvar uvedené metody `Format` mají i jiné metody, např. metoda `AppendFormat` třídy `StringBuilder`.

Parametr `format` představuje *složený formátovací řetězec* (angl. *composite format string*), v němž se vyskytují složené závorky s pořadovými čísly parametrů pole `args`. Tyto složené závorky se nazývají tzv. *formátovací položky* (angl *format items*). První parametr pole `args` má pořadové číslo 0. Parametr pole `args` se v dalším textu označuje jako *formátovaný parametr*. Metoda vrací řetězec, který vznikne z řetězce `format` nahrazením formátovacích položek textovou podobou formátovaných parametrů.

Příklad

```
class Program
{
   static void Main(string[] args)
   {
      int a = 10;
      double b = 2.5;
      string s = string.Format("a = {0}, b = {1}", a, b);
      Console.WriteLine(s);
   }
}
```

V uvedeném příkladu má složený formátovací řetězec formátovací položky `{0}` a `{1}`.

Výstup programu bude následující:

```
a = 10, b = 2,5
```

Pokud se má ve výsledném textu objevit levá nebo pravá složená závorka, musí se ve složeném formátovacím řetězci uvést dvě složené závorky za sebou. Např. následující příkaz

```
string s = string.Format("a = {{{0}}}, b = {{{1}}}", a, b);
```

uloží do řetězce `s` text

```
a = {10}, b = {2,5}
```

Formátovací položka

Formátovací položka vyskytující se ve složeném formátovacím řetězci má následující syntaxi.

Syntaxe:

formátovací položka:

{*index část_zarovnání*$_{\text{nep}}$ *část_formátovací_řetězec*$_{\text{nep}}$ **}**

část zarovnání:

, *zarovnání*

část formátovací řetězec:

: *formátovací řetězec*

Mezi levou složenou závorkou a částí *index* nesmí být mezera. Mezi jednotlivými částmi formátovací položky, jakož i před pravou složenou závorkou může být mezera (ale nemusí).

Index je povinná část, která reprezentuje pořadové číslo formátovaného parametru, počínaje nulou. Více formátovacích položek se může odkazovat na stejný formátovaný parametr. Formátovací řetězec se může odkazovat na pořadová čísla v libovolném pořadí, např. `"{b = {1}, a = {0}"`.

Část *zarovnání* je nepovinná a představuje minimální šířku vyhrazeného prostoru zadanou jako celé číslo se znaménkem. Je-li uvedena, musí být od části *index* oddělena čárkou. Hodnota *zarovnání* se ignoruje, pokud skutečná délka výsledného textu formátovací položky je větší než absolutní hodnota *zarovnání*. Způsob zarovnání je dán znaménkem hodnoty *zarovnání*:

- kladná hodnota – výsledný text je zarovnán napravo vyhrazeného prostoru,
- záporná hodnota – výsledný text je zarovnán nalevo vyhrazeného prostoru.

Zbývající prostor vyhrazeného prostoru je vyplněn mezerami.

Např. příkazy

```
Console.WriteLine("{0,-5}{1,5}", 100, 200);
Console.WriteLine("{0,-5}{1,5}", -100, 20);
```

produkují následující výstup

```
100    200
-100    20
```

Část *formátovací řetězec* je nepovinná a představuje řetězec určující formát výsledného textu pro daný formátovaný parametr. Možnosti závisí na typu formátovaného parametru (číslo, datum a čas, výčtový typ apod.) a jsou popsány dále. Je-li formátovací řetězec uveden, musí začínat dvojtečkou. Není-li uveden, použije se všeobecný formát `"G"`.

Při předání formátovaného parametru do formátovací položky se postupuje podle následujícího pořadí:

1. Je-li hodnota formátovaného parametru `null`, výsledkem je prázdný řetězec (`""`).
2. Jinak, jestliže typ formátovaného parametru implementuje rozhraní `ICustomFormatter`, je volána metoda `Format` tohoto rozhraní, mající tvar

   ```
   string Format(string format, object arg, IFormatProvider fp)
   ```

   kde: `format` reprezentuje formátovací řetězec, `arg` formátovaný parametr a `fp` formátovací informace závislé na kultuře.
3. Jinak, jestliže typ formátovaného parametru implementuje rozhraní `IFormattable`, je volána metoda `ToString` tohoto rozhraní, mající tvar

   ```
   string ToString(string format, IFormatProvider fp)
   ```

   `kde` parametry mají stejný význam jako ve 2. variantě.
4. Jinak je pro formátovaný parametr volána metoda `ToString` třídy `object` (může být předefinována), mající tvar

   ```
   string ToString()
   ```

Způsob zarovnání je aplikován až pro hodnotu, která je výsledkem jednoho z uvedených kroků.

Pro základní číselné datové typy (`int`, `float`, `decimal` apod.), výčtové typy a typ datum a čas se použije 3. varianta uvedeného postupu, protože tyto typy implementují rozhraní `IFormattable`.

Typy `char` a `string` neimplementují žádné z uvedených rozhraní, a proto se pro ně použije 4. varianta.

Pokud *index* má hodnotu neexistujícího pořadového čísla formátovaného parametru, nebo pokud je uveden jinak nesprávný zápis formátovací položky, vyvolá se výjimka typu `System.FormatException`.

Standardní formátovací řetězec pro čísla

Standardní formátovací řetězec pro čísla má tvar

Axx

kde:

A – formátovací specifikátor – jeden znak,

xx – specifikátor přesnosti – celé číslo v rozsahu 0 až 99.

Standardní formát vychází z formátu čísel třídy `System.Globalization.NumberFormatInfo` pro aktuální kulturu, implicitně podle nastavení Windows (menu **Start | Ovládací panely | Místní a jazykové nastavení**).

Přehled formátovacích specifikátorů je uveden v následující tabulce. Uváděná jména vlastností jsou součástí třídy `NumberFormatInfo`.

Formátovací specifikátor	Význam
`C` nebo `c`	Formát měny specifikovaný v třídě `NumberFormatInfo`. Specifikátor přesnosti udává počet desetinných míst. Je-li vynechán, použije se přesnost, kterou poskytuje vlastnost `CurrencyDecimalDigits`.
`D` nebo `d`	Číslo v desítkové soustavě. Lze použít pouze pro celočíselného typy. Specifikátor přesnosti udává minimální počet číslic. Je-li počet číslic formátovaného parametru menší než specifikátor přesnosti, číslo je doplněno zleva nulami.
`E` nebo `e`	Semilogaritmický tvar. Specifikátor přesnosti udává počet desetinných míst. Je-li vynechán, číslo se zobrazí s přesností na 6 desetinných míst. Velikost písmene exponentu odpovídá velikosti formátovacího specifikátoru.
`F` nebo `f`	Tvar s pevnou řádovou čárkou. Specifikátor přesnosti udává počet desetinných míst. Je-li vynechán, použije se přesnost, kterou poskytuje vlastnost `NumberDecimalDigits`.
`G` nebo `g`	Všeobecný formát. Je-li specifikátor přesnosti vynechán nebo má hodnotu nula, použije se implicitní přesnost pro daný datový typ (viz nápověda).
`N` nebo `n`	Číslo s oddělovačem tisíců specifikovaným hodnotou vlastnosti `NumberGroupSeparator`. Počet číslic ve skupině udává vlastnost `NumberGroupSizes` (implicitně 3). Specifikátor přesnosti udává počet desetinných míst. Je-li vynechán, použije se přesnost, kterou poskytuje vlastnost `NumberDecimalDigits`.

Formátovací specifikátor	Význam
`P` nebo `p`	Číslo se symbolem procent podle vzoru, který poskytují vlastnosti `PercentNegativePattern` a `PercentPositivePattern` pro kladné a záporné číslo. Vstupní hodnota je vynásobena 100. Specifikátor přesnosti udává počet desetinných míst. Je-li vynechán, použije se přesnost, kterou udává vlastnost `PercentDecimalDigits`. Výsledný formát ovlivňují další vlastnosti: `PercentGroupSeparator`, `PercentGroupSizes`, `PercentSymbol`.
`R` **nebo** `r`	Tvar „cesta tam a zpět“ (angl. „round-trip“). Lze použít pouze pro typy `double` a `float`. Tento tvar garantuje, že reálné číslo zkonvertované na řetězec bude možné zkonvertovat zpět na stejnou číselnou hodnotu. Nejprve se testuje převod čísla na text s použitím implicitní přesnosti (15 desetinných míst pro typ `double`, 7 míst pro typ `float`). Jestliže takto zkonvertované číslo lze převést zpět na stejnou číselnou hodnotu, použije se pro číslo všeobecný formát `G`. Jinak se číslo převede na text s přesností 17 desetinných míst pro typ `double` a 9 míst pro typ `float`. Případný specifikátor přesnosti se ignoruje.
`X` nebo `x`	Hexadecimální tvar. Lze použít pouze pro celočíselné typy. Velikost písmen A až F odpovídá velikosti formátovacího specifikátoru. Specifikátor přesnosti udává minimální počet číslic. Je-li počet číslic formátovaného parametru menší než specifikátor přesnosti, číslo je doplněno zleva nulami.

Oddělovač desetinných míst poskytují vlastnosti `CurrencyDecimalSeparator`, `NumberDecimalSeparator`, `PercentDecimalSeparator` třídy `NumberFormatInfo` podle použitého formátovacího specifikátoru.

Jestliže hodnota formátovaného parametru typu `double` nebo `float` je kladné nebo záporné nekonečno nebo NaN, bez ohledu na použitý formátovací specifikátor je výsledkem text, který poskytují vlastnosti `PositiveInfinitySymbol`, `NegativeInfinitySymbol` a `NaNSymbol` třídy `NumberFormatInfo`.

Příklady pro českou kulturu jsou uvedeny v následující tabulce.

Formátovací položka	Hodnota	Výstup
`{0:C}`	`1234`	`1 234,00 Kč`
`{0:C1}`	`123.456`	`1 234,5 Kč`
`{0:D}`	`1234`	`1234`
`{0:D6}`	`1234`	`001234`
`{0:E}`	`1234`	`1,234000E+003`
`{0:e2}`	`123.456`	`1,23e+002`
`{0:F}`	`1234`	`1234,00`
`{0:F1}`	`123.456`	`123,5`
`{0:G}`	`123.456`	`123,456`

Formátovací položka	Hodnota	Výstup
{0:G1}	123.456	1E+02
{0:G2}	123.456	1,2E+02
{0:N}	1234	1 234,00
{0:P}	1234	123 400,00%
{0:P0}	123.456	12 346%
{0:R}	123.456	123,456
{0:R1}	123.456	123,456

Uživatelem definovaný formátovací řetězec pro čísla

Uživatelem definovaný formátovací řetězec pro čísla se skládá z jednoho nebo více formátovacích specifikátorů, které jsou uvedeny v následující tabulce.

Formátovací specifikátor	Význam
0	Jestliže hodnota formátovaného parametru má číslici (včetně nevýznamné nuly) na pozici, na které je ve formátovacím řetězci uvedena tato nula, potom se tato číslice zobrazí ve výsledném textu. Pozice nuly nejvíce vlevo a nuly nejvíce vpravo ve formátovacím řetězci určují rozsah číslic, které se vždy zobrazí ve výsledném textu (včetně nevýznamných nul).
#	Jestliže hodnota formátovaného parametru má číslici jinou než nevýznamnou nulu na pozici, na které je ve formátovacím řetězci uveden symbol #, potom se tato číslice zobrazí ve výsledném textu.
.	Pozice desetinné čárky.
,	Je-li tento specifikátor umístěn jinde než bezprostředně před specifikátorem desetinné čárky, představuje oddělovač tisíců. Příslušné vlastnosti třídy NumberFormatInfo určují symbol oddělovače a počet číslic ve skupině. Počet číslic ve skupině nezávisí na počtu specifikátorů 0 nebo # uvedených před nebo za specifikátorem oddělovače tisíců. Je-li tento specifikátor uveden bezprostředně před specifikátorem desetinné čárky resp. před implicitně zobrazenou desetinnou čárkou, hodnota formátovaného parametru je podělena 1000 pro každý výskyt tohoto specifikátoru.
%	Pro každý výskyt tohoto specifikátoru se nejprve hodnota formátovaného parametru vynásobí 100 a na pozici, na které je ve formátovacím řetězci uveden tento specifikátor, se ve výsledném textu zobrazí znak procent daného vlastností PercentNegativePattern resp. PercentPositivePattern.
E0 nebo e0 E+0 nebo e+0 E-0 nebo e-0	Je-li ve formátovacím řetězci uveden některý z těchto specifikátorů, hodnota se zobrazí v semilogaritmickém tvaru. Počet nul za symbolem E resp. e specifikuje počet číslic, které se zobrazí v exponentu. Je-li ve specifikátoru uveden znak +, zobrazí se znaménko + u kladné hodnoty exponentu, jinak se u kladné hodnoty exponentu znaménko nezobrazí. U záporné hodnoty exponentu se zobrazí znaménko − vždy bez ohledu na použitý typ tohoto specifikátoru.
\	Znak řídící posloupnosti (angl. escape sequence). Např. \t způsobí vložení znaku tabulátoru do výsledného textu.

Formátovací specifikátor	Význam
'ABC' nebo "ABC"	Znaky v apostrofech nebo uvozovkách se zobrazí ve výsledném textu ve stejném tvaru jako ve formátovacím řetězci.
;	Oddělovač oddílů pro kladná, záporná a nulová čísla. Formátovací řetězec může obsahovat: • jeden oddíl – použije se pro všechny hodnoty. • dva oddíly – první oddíl se použije pro kladné a nulové hodnoty, druhý oddíl pro záporné hodnoty. • tři oddíly – první oddíl se použije pro kladné, druhý pro záporné a třetí pro nulové hodnoty.
jiný znak	Jakýkoli jiný znak se zobrazí ve výsledném textu ve stejném tvaru jako ve formátovacím řetězci.

Příklady pro českou kulturu jsou uvedeny v následující tabulce.

Formátovací položka	Hodnota	Výstup
{0:00000.00}	1234	01234,00
{0:00000.00}	123.456	00123,46
{0:##.##}	1234	1234
{0:##.##}	123.456	123,46
{0:#,#}	1234	1 234
{0:#,0.00}	12345.678	12 345,68
{0:0,.0}	123456.78	123,5
{0:0,,.0}	123456.78	0,1
{0:0.0%}	12345	123400,0%
{0:0.0%}	123.456	12345,6%
{0:0.0E+00}	123.456	1,2E+02
{0:0.0E-00}	123.456	1,2E02
{0:00\t00}	1234	12 34
{0:0.0' km'}	123.456	123,5 km
{0:0;(0);'nula'}	12	12
{0:0;(0);'nula'}	-12	(12)
{0:0;(0);'nula'}	0	nula

Datový typ data a času

Pro práci s datem a časem slouží struktura `DateTime`, která umožňuje uchovat čas od půlnoci 1. ledna roku 1 do 31. prosince roku 9999, 23:59:59.

Časové hodnoty jsou měřeny ve stovkách nanosekund nazývaných v anglickém jazyce „ticks“, česky tiky, tikoty. Konkrétní datum je interně reprezentován jako počet tiků od 1.1.0001 00:00:00 v gregoriánském kalendáři. Gregoriánský kalendář je používán ve většině zemí světa, včetně České republiky.

Od verze .NET 2.0 struktura `DateTime` obsahuje 64-bitovou celočíselnou datovou složku (typu `ulong`) složenou ze 62-bitové hodnoty počtu tiků a 2-bitové hodnoty vlastnosti `Kind` výčtového typu `DateTimeKind`, který má následující konstanty:

- `Local` – čas reprezentuje lokální čas počítače.
- `Utc` – čas reprezentuje koordinovaný světový čas (angl. Coordinated Universal Time).
- `Unspecified` – druh času není specifikován.

Vlastnost `Kind` je používána k převodu mezi UTC a lokálním časem.

Před verzí .NET 2.0 neobsahovala struktura `DateTime` 2-bitovou hodnotu `Kind`.

Čas UTC je také někdy nazýván „Zulu time“ a je označován písmenem Z za časovým údajem. Jednotlivá časová pásma jsou definována svými odchylkami od UTC. UTC je nástupcem greenwichského středního času GMT (angl. Greenwich Mean Time). GMT udává čas platný v časovém pásmu vůči nultému (základnímu) poledníku a je založen na rotaci Země. Zatímco UTC je založen na atomových hodinách. Protože se rotace Země mírně zpomaluje, GMT se oproti UTC postupně zpožďuje. Aby se UTC dal používat v praktickém životě, který je s rotací Země spjatý, je udržován v rozmezí ±0,9 sekund od přesného atomového času. Pokud je tato odchylka překročena, je přidána přestupná sekunda. K tomu dochází průměrně jednou za rok až rok a půl.

UTC čas je vhodný pro výpočty, porovnání a ukládání data a času do souboru. Lokální čas je vhodný pro zobrazení uživateli. UTC čas se nemění při přechodu na letní nebo zimní čas, zatímco lokální čas ano.

Struktura `DateTime` neumožňuje změnit hodnotu její datové složky, jedná se o tzv. „*immutable type*“. Metody pouze vrací novou hodnotu po provedené operaci.

Výsledkem nebo parametrem některých metod, např. výsledkem rozdílu dvou časů je instance struktury `TimeSpan`.

Struktura `TimeSpan` reprezentuje časový interval nebo dobu trvání, měřenou jako kladný nebo záporný počet dnů, hodin, minut, sekund a zlomků sekund. Interně je tento interval uložen jako počet tiků ve stovkách nanosekund v datové složce typu `long`. Časový interval pokrývá rozsah od –10 675 199 do 10 675 199 dnů.

Struktura `DateTime` nabízí celou řadu konstruktorů s následujícími parametry:

- počet tiků se zadaným nebo nespecifikovaným druhem času,
- rok, měsíc, den v gregoriánském kalendáři nebo v zadaném kalendáři (např. čínský, juliánský),
- rok, měsíc, den, hodina, minuta, sekunda v gregoriánském kalendáři nebo v zadaném kalendáři se zadaným nebo nespecifikovaným druhem času,
- rok, měsíc, den, hodin, minuta, sekunda, milisekunda v gregoriánském kalendáři nebo v zadaném kalendáři se zadaným nebo nespecifikovaným druhem času.

Vlastnosti poskytují jednotlivé části data a času (rok, měsíc, den v měsíci, den v týdnu, den v roce, hodina, minuta, sekunda, milisekunda, počet tiků), druh času a dále vlastnosti:

- `Date` – poskytuje datum s nulovým časem,
- `TimeOfDay` – poskytuje čas v rámci dne typu `TimeSpan`,
- `Today` – poskytuje aktuální datum (s nulovým časem) v počítači v lokálním čase,
- `Now` – poskytuje aktuální datum a čas počítače v lokálním čase,
- `UtcNow` – poskytuje aktuální datum a čas počítače v UTC čase,

Některé metody struktury `DateTime` jsou uvedeny v následujícím seznamu.

```
DateTime Add(TimeSpan value)
```

Přičte hodnotu časového intervalu k hodnotě `this`.

```
DateTime Add<i>Jednotky</i>(value)
```

Skupina metod, které přičtou k hodnotě `this` zápornou nebo kladnou hodnotu zadaných jednotek (roky až milisekundy nebo tiky).

metody `CompareTo`, `Compare` a `Equals`
relační operátory

Metody a operátory porovnávají dva časy podle počtu tiků bez ohledu na druh času.

```
static bool IsLeapYear(int year)
```

Vrací `true`, pokud zadaný rok je přestupný.

```
bool IsDaylightSavingTime()
```

Vrací `true`, pokud `this` datum a čas patří do období letního času.

```
static DateTime operator + (DateTime d, TimeSpan t)
static DateTime operator - (DateTime d, TimeSpan t)
DateTime Subtract(TimeSpan value)
```

Přičte nebo odečte hodnotu časového intervalu k resp. od hodnoty `this`.

```
static TimeSpan operator - (DateTime d1, DateTime d2)
TimeSpan Subtract(DateTime value)
```

Odečte od sebe dvě data.

```
DateTime ToLocalTime()
```

Převádí `this` hodnotu na lokální čas. Je-li `this` hodnota v nespecifikovaném čase, chápe se jako v čase UTC.

```
DateTime ToUniversalTime()
```

Převádí `this` hodnotu na UTC čas. Je-li `this` hodnota v nespecifikovaném čase, chápe se jako v lokálním čase.

```
string ToLongDateString()
string ToShortDateString()
```

Převádí datum a čas na řetězec znaků reprezentující dlouhý resp. krátký formát data aktuální kultury. Odpovídá formátovacímu specifikátoru `D` resp. `d` (viz dále).

```
string ToLongTimeString()
string ToShortTimeString()
```

Převádí datum a čas na řetězec znaků reprezentující dlouhý resp. krátký formát času aktuální kultury. Odpovídá formátovacímu specifikátoru `T` resp. `t` (viz dále).

```
string ToString()
```

Převádí datum a čas na řetězec znaků reprezentující krátký formát data a dlouhý formát času. Odpovídá formátovacímu specifikátoru `G` (viz dále).

Příklad

```
class Program
{
   static void Main(string[] args)
   {
      DateTime d = new DateTime();
      Console.WriteLine("Nulová hodnota reprezentuje čas {0}", d);
      DateTime d2 = new DateTime(2008, 1, 31, 12, 0, 0,
                                 DateTimeKind.Utc);
      Console.WriteLine("Čas v UTC: {0} ", d2);
      DateTime d3 = d2.ToLocalTime();
      Console.WriteLine("Čas po převodu na lokální čas: {0}", d3);
      d2 = DateTime.UtcNow;
      Console.WriteLine("Aktuální UTC čas: {0} ", d2);
      d3 = DateTime.Now;
      Console.WriteLine("Aktuální lokální čas: {0} ", d3);
      TimeSpan ts = d3 - d2;
      Console.WriteLine("Počet hodin rozdílu mezi lokálním a UTC časem:"
         +" {0}", ts.Hours);
      Console.WriteLine("Aktuální UTC čas po přičtení 1,5 dne: {0}",
         d2.AddDays(1.5));
      Console.WriteLine("Aktuální UTC čas po přičtení 1 dne a 2 h "+
         +"a 3 s: {0}", d2.Add(new TimeSpan(1, 2, 3, 0)));
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
Nulová hodnota reprezentuje čas 1.1.0001 0:00:00
Čas v UTC: 31.1.2008 12:00:00
Čas po převodu na lokální čas: 31.1.2008 13:00:00
Aktuální UTC čas: 11.3.2008 12:31:10
Aktuální lokální čas: 11.3.2008 13:31:10
Počet hodin rozdílu mezi lokálním a UTC časem: 1
Aktuální UTC čas po přičtení 1,5 dne: 13.3.2008 0:31:10
Aktuální UTC čas po přičtení 1 dne a 2 h a 3 s: 12.3.2008 14:34:10
```