SOUBORY, VSTUPY A VÝSTUPY – POKRAČOVÁNÍ

Vstupy a výstupy – pokračování

Kódování textů

Texty (řetězce nebo znaky) v jazyce C# jsou v paměti uloženy v kódování označovaném běžně Unicode (kódová stránka 1200).

Texty vypisované na obrazovku konzolové aplikace se implicitně převádí do kódování Latin 2 (kódová stránka 852). Texty zapisované do datového proudu se implicitně převádí do kódování UTF-8 (kódová stránka 65001). Při čtení se provádí opačný převod.

UTF je zkratka slov „Unicode Transformation Format“ a určuje způsob zakódování Unicode kódů (znaků – 16-ti bitových kódů) bez ztráty informací. Existují následující typy:

- UTF-7 – zastaralý způsob kódování,
- UTF-8 – převádí Unicode kód na posloupnost jednoho až čtyř bytů,
- UTF-16 – převádí Unicode kód na posloupnost jednoho až dvou 16-bitových celých čísel,
- UTF-32 – převádí Unicode kód na jedno 32-bitové celé číslo.

Kódování UTF-16 a UTF-32 mají dva podtypy:

- little-endian byte order – např. znak A `'\u0041'` je zapsán v UTF-16 ve tvaru 00 41,
- big-endian byte order – např. znak A `'\u0041'` je zapsán v UTF-16 ve tvaru 41 00.

Běžné označení Unicode kódování je kódování UTF-16 little-endian byte order. Jeden Unicode kód umožňuje uchovat 65 535 znaků. Tento počet nestačí pro uchování všech znaků jazyků světa. Např. čínský jazyk vyžaduje více než 80 000 znaků. Tyto a další znaky lze však také uchovat, a to jako dvojici znaků – základní znak a doprovodný znak. Doprovodné znaky jsou definovány v rozsahu `'\u0328'` až `'\u0345'` (podle [2]). Např. znak ǫ (malé o s ogonkem) lze zapsat jako dvojici znaků `"\u006F\u0328"`.

Typ kódování textů lze zadat v parametru konstruktoru datových proudů `BinaryReader`, `BinaryWriter`, `StreamReader` a `StreamWriter`. Parametr je typu `System.Text.Encoding`, což je abstraktní třída, která je předkem tříd, jež představují jednotlivá kódování.

Většina potřebných kódování je k dispozici pomocí statických složek třídy `Encoding`:

- vlastnost `Default` – poskytuje implicitní kódování kódové stránky ANSI nastavené v operačním systému. Pro česká Windows je to 1250.
- vlastnost `Unicode` – poskytuje kódování UTF-16 little-endian byte order.
- vlastnost `UTF8` – poskytuje kódování UTF-8.
- vlastnost `UTF32` – poskytuje kódování UTF-32 little-endian byte order.
- metoda `GetEncoding(int codepage)` – vrací kódování pro zadané číslo kódové stránky.

Třída `Console` umožňuje změnit kódování nebo zjistit aktuální kódování pro čtení a výpis pomocí dvou statických vlastností `InputEncoding` a `OutputEncoding`.

Příklad

Program uloží zadaný text do textového souboru v implicitním kódování ANSI, což v českých Windows bude kódová stránka 1250.

```
class Program
{
   static void UlozText(string jmeno, string text, Encoding encoding)
   {
      using (StreamWriter sw = new StreamWriter(File.Create(jmeno),
             encoding)) {
         sw.WriteLine(text);
      }
   }
   static void Main(string[] args)
   {
      string text = "Nějaký český text";
      UlozText("data.txt", text, Encoding.Default);
   }
}
```

Paměťové datové proudy

Paměťové datové proudy jsou v jazyce C# orientovány na pole bytů, a ne na řetězce znaků jako je tomu v C++.

Paměťový datový proud reprezentuje třída `MemoryStream`, která je potomkem třídy `Stream`. Obsahuje mj. tyto konstruktory:

- `MemoryStream()` – vytvoří prázdný proud.
- `MemoryStream(byte[] buffer)` – vytvoří proud svázaný s externím polem bytů `buffer`. Pole bytů nelze později zvětšit.
- `MemoryStream(int capacity)` – vytvoří proud s vnitřním polem bytů velikosti `capacity`. Pole bytů lze později zvětšit.

Pro čtení a zápis z/do paměťového proudu lze využít metody třídy `Stream`, ale zpravidla se využívají třídy `BinaryReader`, `BinaryWriter`, `StreamReader` a `StreamWriter`.

Třída `MemoryStream` mj. obsahuje metodu `byte[] ToArray()`, která vrací kopii vnitřního pole bytů paměťového proudu.

Příklad

Program provádí konverzi řetězce znaků z jednoho kódování do jiného. Text určený ke konverzi se zapíše do pole bytů pomocí paměťového datového proudu v zadaném kódování.

Pole bytů se převede z jednoho kódování na jiné pomocí statické metody `Convert` třídy `Encoding`:

```
byte[] Convert(Encoding srcEncoding, Encoding dstEncoding, byte[] bytes)
```

Metoda `Convert` převede pole bytů `bytes` z kódování `srcEncoding` na kódování `dstEncoding` a vrací převedené pole bytů.

Převod pole bytů na řetězec provádí statická metoda `GetString` příslušné třídy kódování. V příkladu je to třída, kterou poskytuje vlastnost `Encoding.Unicode`.

```
class Program
{
   static void Main(string[] args)
   {
      string text = "Nějaký český text";
      MemoryStream ms = new MemoryStream();
      using (StreamWriter sw = new StreamWriter(ms, Encoding.Default)) {
         sw.Write(text);
      }
      byte[] buffer = ms.ToArray();
      buffer = Encoding.Convert(Encoding.Default, Encoding.Unicode,
         buffer);
      string text2 = Encoding.Unicode.GetString(buffer);
      Console.WriteLine(text2);
   }
}
```

Pro převod řetězce znaků na pole bytů kódové stránky odpovídající danému potomkovi třídy `Encoding` slouží virtuální metoda `GetBytes` třídy `Encoding`:

```
virtual byte[] GetBytes(string s)
```

Např. převod řetězce znaků `s` na pole bytů kódové stránky 1250 lze provést příkazem

```
byte[] b = Encoding.Default.GetBytes(s);
```

Příklad

Příklad demonstruje uložení pole tříd `Osoba` do binárního souboru a jeho načtení. Záznam osoby v souboru má pevnou délku, což umožňuje případné načtení jen konkrétního záznamu v souboru. Při ukládání údajů osoby se vytvoří pole bytů odpovídající délce záznamu osoby, které se spojí s paměťovým proudem. Do paměťového proudu se binárně uloží údaje osoby. Potom se pole bytů uloží do binárního souboru.

Při čtení údajů osoby se z binárního souboru načte pole bytů odpovídající délce záznamu osoby. Toto pole se spojí se paměťovým proudem, ze kterého se binárně načtou údaje osoby.

```
class Osoba
{
   public const int Delka = 100;
   public int cislo;
   public string jmeno;
   public Osoba() { }
   public Osoba(int cislo, string jmeno)
   {
      this.cislo = cislo;
      this.jmeno = jmeno;
   }
   public void NactiPevnaDelka(BinaryReader br)
   {
      byte[] buffer = br.ReadBytes(Delka);
      MemoryStream ms = new MemoryStream(buffer);
      using (BinaryReader br2 = new BinaryReader(ms)) {
         cislo = br2.ReadInt32();
         jmeno = br2.ReadString();
      }
   }
```

```
   public void UlozPevnaDelka(BinaryWriter bw)
   {
      byte[] buffer = new byte[Delka];
      MemoryStream ms = new MemoryStream(buffer);
      using (BinaryWriter bw2 = new BinaryWriter(ms)) {
         bw2.Write(cislo);
         bw2.Write(jmeno);
      }
      bw.Write(buffer);
   }
}
class Program
{
   static void UlozPevnaDelka(string jmeno, Osoba[] osoby)
   {
      using (BinaryWriter bw = new BinaryWriter(File.Create(jmeno))) {
         foreach (Osoba osoba in osoby) {
            osoba.UlozPevnaDelka(bw);
         }
      }
   }
   static void NactiPevnaDelka(string jmeno, out Osoba[] osoby)
   {
      FileStream fs = File.OpenRead(jmeno);
      int pocet = (int)(fs.Length / Osoba.Delka);
      osoby = new Osoba[pocet];
      using (BinaryReader br = new BinaryReader(fs)) {
         for (int i = 0; i < pocet; i++) {
            osoby[i] = new Osoba();
            osoby[i].NactiPevnaDelka(br);
         }
      }
   }
   static void Main(string[] args)
   {
      Osoba[] osoby = new Osoba[3]
         { new Osoba(10, "Čejka"), new Osoba(20, "Dvořák"),
           new Osoba(30, "Mojžíš") };
      try {
         UlozPevnaDelka("data.bin", osoby);
         NactiPevnaDelka("data.bin", out osoby);
         foreach (Osoba osoba in osoby) {
            Console.WriteLine("{0} {1}", osoba.cislo, osoba.jmeno);
         }
      }
      catch (Exception e) {
         Console.WriteLine("Chyba: " + e.Message);
      }
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
10 Čejka
20 Dvořák
30 Mojžíš
```

Serializace objektů

Serializace představuje mechanizmus, umožňující uložit do datového proudu jakýkoli objekt. Objekt může obsahovat odkazy na jiné objekty, které se mohou odkazovat zase na další objekty atd. Spolu s ukládaným objektem se uloží i objekty, na které se objekt přímo nebo nepřímo odkazuje a informace o jejich vzájemných vztazích. Ukládá se tzv. objektový graf.

Deserializace reprezentuje opačný proces než je serializace, tj. z datového proudu načte a vytvoří objekt včetně objektů, na které se přímo nebo nepřímo odkazuje.

Serializaci objektů lze provést v následujících formátech:

- binární,
- SOAP (Simple Object Access Protocol) – protokol založený na XML určený pro výměnu informací na webu,
- XML.

Serializace v prvních dvou formátech se liší od XML serializace.

Serializace v binárním nebo SOAP formátu

Ukládané objekty musí splňovat následující požadavky:

- Třída (struktura), jejíž instance se má serializovat, musí být deklarována s atributem `Serializable`.
- Datové složky, které se nemají serializovat, se deklarují s atributem `Nonserialized`. Při načtení objektu z datového proudu se provede jejich inicializace v závislosti na jejich typu:
  - jsou-li hodnotového typu, volá se pro ně implicitní konstruktor,
  - jsou-li referenčního typu, mají hodnotu `null`.
- Datové složky, které se mají serializovat, musí být typu, který je deklarován s atributem `Serializable`, jinak při serializaci vznikne výjimka typu `System.Runtime.Serialization.SerializationException`. To se týká i datových složek objektu, na který se datová složka této třídy přímo nebo nepřímo odkazuje.

Datové složky se serializují bez ohledu na jejich přístupová práva. Vlastnosti se neserializují.

S atributem `Serializable` jsou deklarovány všechny základní datové typy, typ `string` i třídy kolekcí `Array`, `ArrayList` aj.

Pro serializaci slouží následující třídy:

- `System.Runtime.Serialization.Formatters.Binary.BinaryFormatter` – serializace v binárním formátu,
- `System.Runtime.Serialization.Formatters.Soap.SoapFormatter` – serializace ve formátu SOAP. Třída je součástí sestavení `System.Runtime.Serialization.Formatters.Soap`, které se v prostředí Visual Studio musí přidat do části **References** daného projektu.

Běžně se používá konstruktor bez parametrů. Z metod se používají následující dvě:

- `void Serialize(Stream serializationStream, object graph)` – zapíše objekt `graph` do datového proudu `serializationStream`.
- `object Deserialize(Stream serializationStream)` – vrací objekt, který načte z datového proudu `serializationStream`.

Obě třídy formátovačů implementují rozhraní `IFormatter`, které obsahuje metody `Serialize` a `Deserialize`. Pokud tedy chceme provádět serializaci nějakého objektu nezávislého na formátu, lze pracovat s tímto rozhraním.

Do jednoho datového proudu lze serializovat i více objektů samostatným voláním metod `Serialize` a později je deserializovat metodami `Deserialize` ve stejném pořadí, v jakém byly serializovány.

Pokud některá složka objektu, která se má serializovat, se metodou `Deserialize` nenačte, chování závisí na zvoleném formátovači:

- Pro binární formát výjimka nevznikne a taková složka se inicializuje stejnou hodnotou jako složka, která se nemá serializovat.
- Pro formát SOAP se vyvolá výjimka typu `SerializationException`. Pokud je však složka deklarována s atributem `OptionalField`, inicializuje se stejnou hodnotou jako složka, která se nemá serializovat a výjimka nevznikne.

Příklad

Příklad demonstruje serializaci třídy `C`, která obsahuje pole tříd `B`. Třída `B` obsahuje složky `a`, `a2` typu struktury `A` a složku `y` typu `int`. Složka `a2` se nebude serializovat. Jak třídy `B` a `C`, tak i struktura `A` musí být deklarovány s atributem `Serializable`, jinak při serializaci vznikne výjimka.

```
[Serializable]
struct A
{
   public int x;
   public A(int x) { this.x = x; }
}
[Serializable]
class B
{
   A a;
   [NonSerialized]
   A a2 = new A(-1);
   int y;
   public B(int x, int y) { a = new A(x); this.y = y; }
   public override string ToString()
   {
      return "a.x = " + a.x + ", a2.x = " + a2.x + ", y = " + y;
   }
}
[Serializable]
class C
{
   B[] b;
   public C(B[] b) { this.b = b; }
   public void Vypis()
   {
      foreach (B x in b) {
         Console.WriteLine(x.ToString());
      }
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      C c = new C(new B[] { new B(10, 20), new B(30, 40) });
      BinaryFormatter bf = new BinaryFormatter();
      // SoapFormatter bf = new SoapFormatter();
      using (FileStream fs =
             new FileStream("data.bin", FileMode.Create)) {
         bf.Serialize(fs, c);
      }
      c.Vypis();
      c = null;
      using (FileStream fs = new FileStream("data.bin", FileMode.Open)) {
         c = (C)bf.Deserialize(fs);
      }
      Console.WriteLine("Po načtení objektu ze souboru");
      c.Vypis();
   }
}
```

Výstup programu bude následující:

```
a.x = 10, a2.x = -1, y = 20
a.x = 30, a2.x = -1, y = 40
Po načtení objektu ze souboru
a.x = 10, a2.x = 0, y = 20
a.x = 30, a2.x = 0, y = 40
```

Přizpůsobení serializace

Proces serializace lze přizpůsobit např. za účelem určité transformace hodnot složek některé třídy, která je součástí objektového grafu. K tomuto účelu slouží rozhraní `ISerializable`, které musí třída, jež se má serializovat specifickým způsobem, implementovat. Třída musí i tak být deklarována s atributem `Serializable`. Rozhraní `ISerializable` je deklarováno následovně:

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

Metodu `GetObjectData` volá automaticky daný formátovač před uložením dané třídy do datového proudu. Třída, implementující uvedené rozhraní, musí naplnit parametr `info` kolekcí dvojic název/hodnota. Kde název zpravidla reprezentuje název datové složky, která se má serializovat, a hodnota její hodnotu.

Třída `SerializationInfo` obsahuje pro účely serializace přetížené metody `AddValue`, které mají tvar:

```
void AddValue(string name, typ value)
```

kde *typ* reprezentuje jednotlivé základní datové typy.

Pokud se má do kolekce přidat dílčí objekt objektového grafu (včetně objektů, na které odkazuje), na který se odkazuje příslušná datové složka třídy, lze volat metodu

```
void AddValue(string name, object value)
```

Typ `StreamingContext` je struktura obsahující dvě vlastnosti:

- `Context` – poskytuje objekt typu `object`, který může obsahovat libovolné dodatečné informace.
- `State` – poskytuje příznaky, udávající o jaký typ serializace se jedná, např. serializace do souboru, mezi aplikačními doménami, mezi počítači apod.

Obě tyto vlastnosti jsou inicializovány konstruktorem této třídy. Instanci této třídy lze předat konstruktoru příslušného formátovače. Třídy `BinaryFormatter` a `SoapFormatter` obsahují k tomuto účelu konstruktor se dvěma parametry:

```
XxxFormatter(ISurrogateSelector selector, StreamingContext context)
```

Parametr `selector` se používá při serializaci po síti pomocí technologie .NET Remoting (podrobnosti viz nápověda). Pro jinou serializaci může být `null`.

Kromě složek rozhraní `ISerializable` musí třída obsahovat chráněný konstruktor ve tvaru:

```
protected NejakaTrida(SerializationInfo info, StreamingContext context)
```

Je-li třída zapečetěná, může být konstruktor soukromý.

Tento konstruktor volá daný formátovač po načtení hodnot dané třídy z datového proudu. Pro získání prvku z kolekce název/hodnota z třídy `SerializationInfo` se používají metody ve tvaru:

```
typ GetTyp(string name)
```

kde *typ* a *Typ* reprezentují jednotlivé základní datové typy.

Pro deserializaci dílčího objektu objektového grafu lze volat metodu

```
object GetValue(string name, Type type)
```

Do parametru `type` se musí předat informace o skutečném typu dílčího objektu, který metoda vrací.