LEKCE 4: TEXT, DATA, NEKONEČNO

ZNAKY A ŘETĚZCE

ZNAKY

Znaky v .NET Frameworku jsou v paměti reprezentovány jednotně podle normy Unicode, přesněji v kódování UTF-16. Třída `System.Char` (alias je klíčové slovo `char`) reprezentuje jeden znak[1]. Třída `System.String` pak reprezentuje konečnou posloupnost znaků (až do délky 2 GB, tedy 1 miliardy znaků)

Třída `System.Char` má množství statickým metod, vesměs pro identifikaci Unicode kategorie znaku, např.: písmeno, číslice, oddělovač. Znakové literály jsou v C# stejné jako v rodině C jazyků, tedy symbol v jednoduchých uvozovkách: `'a'`. Také fungují obvyklé escape sekvence s pomocí zpětného lomítka:

`\'`	Apostrof	`\n`	Nový řádek (LF – ASCII 10)
`\"`	Dvojité uvozovky	`\r`	Návrat vozíku (CR – ASCII 13)
`\\`	Zpětné lomítko	`\t`	Tabulátor (ASCII 9)
`\0`	Znak NUL	`\u1234`	Unicode znak s hexadec. kódem 1234

Pro zápis nového řádku místo raději použije statickou vlastnost `Environment.NewLine`, která vrací řetězec odpovídající novému řádku na operačním systému: `"\r\n"` pro Windows a `"\n"` pro Unix.

ŘETĚZCE A DOSLOVNÉ LITERÁLY

Řetězce je posloupnost znaků a délka řetězce je uložena ve vnitřní proměnné. V C# může řetězec obsahovat jeden nebo více znaků NUL (`'\0'`), protože ty zde nemají význam konce řetězce.

Řetězce lze porovnávat pomocí operátoru ==, protože je přetížen a porovnává obsah řetězců. Pokud chcete porovnávat řetězce nezávisle na velikosti písmen, můžete použít jedno z přetížení metody `Equals`, kterému předáte hodnotu výčtu `StringComparison`.

Pro vytvoření řetězce se používají literály v podobně dvojitých uvozovek. C# má ale i takzvané *doslovné literály* (*verbatim literal*), které vzniknou umístěním znaku `@` před dvojité uvozovky řetězcového literálu. V doslovném

[1] Tedy většinou, viz specifikace Unicode a UTF-16.

literálu se bere text tak jak je zapsaný, včetně znaků pro nový řádek. Běžné escape sekvence se zpětným lomítkem nefungují, jediná escape sekvence je zápis dvojitých uvozovek jejich zdvojením.

```
string s = "Path to file:\n\r\"C:\\Windows\\calc.exe\"";
string t = @"Path to file:
""C:\Windows\calc.exe""";
```

Statická položka `string.Empty` reprezentuje prázdný řetězec, stejně jako literál `""`.

NEMĚNNOST ŘETĚZCŮ A TŘÍDA STRING BUILDER

Řetězce jsou sice třídy, ale jsou naprogramovány jako neměnné (*immutabilní*). Jednou vytvořený řetězec už nelze změnit. Všechny operace na řetězcích mají za následek vznik nového řetězce (a případné zahození původního). Při častých operacích s řetězci to ovšem znamená rychlé vytváření a zahazování objektů a tím snížení výkonu aplikace. Následující kód vytvoří a zahodí 1000 objektů typu řetězec:

```
string s = "";
for (int i = 0; i < 1000; i++) {
  s += "A";
}
```

Z tohoto důvodu existuje třída `StringBuilder`, která reprezentuje měnitelný řetězec. Tato třída má pro řetězec vyhrazeno určité množství paměti, kterou dokáže měnit a v případě potřeby i navýšit. Vlastnost `Capacity` udává, kolik znaků může `StringBuilder` pojmou, než bude muset zvýšit množství alokované paměti.

KÓDOVÁNÍ A SPOLUPRÁCE S VNĚJŠÍM SVĚTEM

Při komunikaci s vnějším prostředím se často setkáte s jiným kódováním než je UTF-16. Třída `System.Text.Encoding` slouží pro převod mezi různými kódováními, resp. primárně mezi jiným kódováním a Unicode. Instanci třídy `Encoding` pro konkrétní kódování lze získat pomocí statických vlastností této třídy (např. `Ascii` nebo `UTF8`) nebo pomocí statické metody `GetEncoding` na základě IANA názvu (webového názvu kódování: „utf-8“, „iso-8859-2“) nebo čísla kódové stránky.

Při práci s třídou `Encoding` se pak používají metody pro převod mezi textem a polem bytů (`GetString`, `GetChars` a `GetBytes`), statická metoda `Convert` pro převod mezi dvěma kódováními a další členy pro práci s kódováním.

FORMÁTOVÁNÍ ŘETĚZCŮ

Pro formátování řetězců slouží statická metoda `string.Format`, která nahrazuje formátovací položky v zadaném řetězci svými dalšími parametry. Formátovací položka má podobu čísla ve složených závorkách: `{0}`. Dále může obsahovat číslo udávající zarovnání na určitý počet znaků a formátovací řetězec pro dosazovaný objekt.

```
Console.WriteLine("|{0}|", 123);
Console.WriteLine("|{0,5}|", 123);    // zarovnání vpravo
Console.WriteLine("|{0,-5}|", 123);   // zarovnání vlevo
Console.WriteLine("|{0,5:F2}|", 123); // zformátování 123 jako desetinného čísla se 2 místy
```

```
|123|
|  123|
|123  |
```

```
|123,00|
```

Nejběžnějším převodem na řetězec je samozřejmě metoda `ToString` zděděná ze třídy `Object`. Třídy, které dovolují více formátování, ale obvykle mají i `ToString`, který bere jako parametr řetězec. Jak je vidět na příkladu, objekty některých tříd lze formátovat různými způsoby. Pro použití vlastních formátovacích řetězců (jako je „F2“ v příkladu) můžete implementovat rozhraní `IFormattable`.

REGULÁRNÍ VÝRAZY

Pro práci s regulárními výrazy slouží třída `Regex` ze jmenného prostoru `System.Text.RegularExpressions`. Na této třídě lze volat statické metody pro prohledávání a nahrazování řetězců. Nebo můžete vytvořit její instanci pro konkrétní regulární výraz a využít tuto instanci opakovaně.

SOUBORY A VSTUP/VÝSTUP

Pro práci se systémem souborů a soubory samotnými je v knihovně tříd .NET Frameworku vyhrazen jmenný prostor `System.IO`. Rozcestník v MSDN je zde a seznam častých operací pak zde.

SYSTÉM SOUBORŮ

Pro práci se souborovým systémem jsou určeny statické třídy `File` a `Dictionary`, nebo jejich instanciovatelné protějšky `FileInfo` a `DictionaryInfo`, vhodné pro opakované operace se souborem resp. složkou.

Práci s názvy souborů má pak na starosti třída `Path`. Statické metody této třídy umožňují vrácení části cesty (`GetFileName`, `GetExtension`) nebo vytvoření nové cesty skládáním existujících částí (metoda `Combine`). Použití třídy `Path` je bezpečnější i snadnější než práce s cestami jako s pouhými řetězci. Třída `Path` také dovoluje platformovou nezávislost operací s názvy souborů (např. `Combine` použije podle použitého operačního systému jako oddělovač znak `\` nebo `/`).

ČTENÍ A ZÁPIS

Pro zpřístupnění obsahu souborů slouží datové proudy (*streams*) a reprezentuje je třída `Stream`. S proudy lze provádět některé ze tří operací: čtení, zápis nebo přesun v proudu. Zda je operace na daném proudu dostupná prozrazují jeho vlastnosti `CanRead`, `CanWrite` a `CanSeek`.

Existuje více typů proudů, pro různé typy dat a pro různé operace: pro soubory je určen proud FileStream, síťovou komunikaci zase NetworkStream. Také lze mít proud jen v operační paměti MemoryStream, který se hodí při pro použití API proudů s polem bytů. Mezi proudy rozšiřující možnosti jiných proudů patří CryptoStream (zajišťuje šifrování) nebo DeflateStream a GZipStream, které provádějí průběžnou kompresi proudu.

Proudy samotné nenabízí příliš operací, jen zápis a čtení bytů. Pro čtení a zápis ale existují třídy `BinaryReader`, `BinaryWriter`, `StreamReader` a `StreamWriter`. Dvojice `Binary` tříd pracuje s otevřeným proudem jako s bitovým souborem a dovoluje z/do proudu přímo číst a zapisovat vestavěné datové typy a řetězce pevné délky.

Naopak dvojice `StreamReader` a `StreamWriter` pracuje s proudem na úrovni znaků textu v nějakém kódování (standardně UTF-8, není-li určeno jinak). Tyto třídy disponují hlavně metodami `Read`/`ReadLine` a `Write`/`WriteLine` pro

práci s obsahem jako textem. Tyto třídy sdílí bázovou třídu s dvojicí tříd `StringReader` a `StringWriter`, které dovolují pracovat s řetězcem stejným způsobem jako s proudem – proudy a řetězce pak lze libovolně zaměňovat.

UVOLŇOVÁNÍ NEŘÍZENÝCH ZDROJŮ: IDISPOSABLE

Soubory jsou příliš cenné zdroje a je potřeba je zavřít co nejdříve po dokončení práce s nimi. Třídy reprezentující soubory jsou naprogramovány tak, že zavření souboru proběhne při jejich odstranění z paměti Garbage Collectorem. Zde ovšem nastává problém, protože nelze přesně určit, kdy se Garbage Collector rozhodne objekt odstranit – jisté je jen to, že to udělá někdy poté, co se objekt stane nedostupným.

Soubory lze proto zavřít explicitně pomocí metody Close. Je ovšem také potřeba zajistit, aby se soubor zavřel za *všech* okolností a proto byste měli zcela automaticky použít blok try-finally.

Pro sjednocení způsobu uvolňování neřízených zdrojů existuje rozhraní `IDisposable`, které předepisuje jedinou metodu Dispose, která zdroj uvolní. V jazyce C# pak existuje konstrukce, které zjednodušuje použití tohoto rozhraní. Použití v bloku try-finally by mohlo vypadat takto:

```
FileStream fs;
try {
  fs = new FileStream("data.txt");
  fs.Write(…) // operace s proudem
}
finally {
  if( fs != null )
    ((IDisposable)fs).Dispose();
}
```

S použitím bloku `using` lze ten samý kód napsat zkráceně takto:

```
using (FileStream fs = new FileStream())
{
  fs.Write(…)// operace s proudem
}
```

Předchozí dva bloky kódu jsou po zkompilování identické, v obou případech je zaručeno zavolání metody Dispose na proudu `fs`, a to i v případě, že nastane výjimk.

Ovšem může stát, že programátor zapomene Dispose zavolat (ani nepoužije blok using). Je tedy potřeba zajistit uvolnění neřízeného zdroje před odstranění objektu z paměti Garbage Collectorem. Proto se rozhraní IDisposable kombinuje s destruktory (finalizací) ve vývojovém vzoru Disposable Pattern.

GENERÁTORY

Pro fungování `foreach` cyklu je potřeba implementovat iterátor reprezentovaný rozhraními `IEnumerable<T>` a `IEnumerator<T>` resp. jejich negenerickými variantami. Implementace iterátor je poměrně snadná, rozhraní `IEnumerator` předepisuje pouze 2 metody (přesun na další položku a vrácení se na začátek) a jednu vlastnost (aktuální položka), rozhraní `IEnumerable` pouze metodu vracející iterátor.

Veškerou logiku iterátorů pak lze zredukovat na vrácení dalšího prvku, zbytek je naprosto mechanický. Kompilátor jazyka poskytuje pro tvorbu iterátorů podporu v podobě klíčového slova `yield`, které lze použít v libovolné metodě nebo vlastnosti s návratovým typem `IEnumerable`, `IEnumerable<T>`, `IEnumerator`, nebo `IEnumerator<T>`. Klíčové slovo

`yield` se používá v kombinaci s `return` nebo `break` a způsobí, že kompilátor si „zapamatuje“, na kterém místě metodu opustil a při dalším volání pokračuje od tohoto místa dál. Ve skutečnosti je to jen další trik, protože kompilátor si na to postaví třídu iterátoru se stavovým automatem, tato složitost je ale před programátorem skryta.

```
public static IEnumerable<string> EnumeratNumbers()
{
  yield return 1;
  yield return 100;
  for(int i = 0; i < 3; i++)
  {
    yield return i;
  }
}

foreach(int n in EnumerateNumbres())
{
  Console.WriteLine(n);
}
```

```
1
100
0
1
2
3
```

Pokud chcete vygenerovat číselnou řadu, bude kód vypadat takto:

```
public static IEnumerable<int> Range(int start, int count)
{
  for(int i = start; i < (start+count); i++)
    yield return i;
}
```

Tato metoda již existuje ve třídě `Enumerable`, která je součástí BCL. Proč se ale omezovat jen na konečné řady? Nostalgicky si vzpomeňte na úvod do funkcionálního programování a líné vyhodnocování:

```
public static IEnumerable<long> EvenNumbers()
{
  int number = 0;
  while(true)
  {
    yield return number;
    number += 2;
  }
}
```

Nebo je libo něco složitějšího:

```
public static IEnumerable<long> GetPrimeNumbers()
{
  int number = 1;
  while(true)
  {
    if(IsPrime(number))
      yield return number;
    number++;
  }
}
```

```
// naivni test na prvocislo
private static bool IsPrime(int number)
{
  long numberSquareRoot = (long)Math.Sqrt(number);
  for (int i = 2; i <= numberSquareRoot; i++)
  {
    if (number % i == 0)
      return false;
  }
  return true;
}
```