DATOVÉ TYPY – POKRAČOVÁNÍ

Referenční typy – pokračování

Pole

Pole jsou v C# indexována od 0, stejně jako v C++. Jinak se pole v C# výrazně liší od polí v C++.

Hlavní rozdíly:

- Pole jsou dynamicky alokována pomocí operátu `new`. Jejich dealokaci provádí automatická správa paměti. Pracuje se s nimi pomocí referencí.
- S polem lze pracovat jako s celkem.
- Při přístupu k prvku pole se vždy kontroluje, zda požadovaný index prvku leží v rozsahu určeném deklarací. Pokud ne, vznikne výjimka typu `System.IndexOutOfRangeException`.

Pole jsou instance tříd odvozených od třídy `System.Array`. Kromě samotných prvků obsahují ještě další složky. Třída `System.Array` implementuje mj. rozhraní `IEnumerable`, takže pole lze použít v příkazu `foreach` (viz dále).

Jednorozměrná pole

Syntaxe specifikace typu:

typ_jednorozměrného_pole:

typ_ne_pole **[ ]**

Typ_ne_pole – typ prvků pole. Může se jednat o libovolný hodnotový nebo referenční typ kromě typu pole. Do hranatých závorek za typem se neuvádí počet prvků. To znamená, že dvě jednorozměrná pole prvků typu `int` jsou instance stejného typu, i když se liší počtem prvků.

Syntaxe deklarace instance:

deklarace_jednorozměrného_pole_s_inicializací:

typ_jednorozměrného_pole identifikátor = **new** *typ_ne_pole* **[** *počet_prvků*$_{\text{nep}}$ **]** *inicializátor_pole*$_{\text{nep}}$

typ_jednorozměrného_pole identifikátor = *inicializátor_pole*

Počet_prvků – celočíselný výraz, udávající počet prvků pole. Uvede-li se *inicializátor_pole*, může se *počet_prvků* vynechat. Uvede-li se *inicializátor_pole* a zároveň *počet_prvků*, musí *počet_prvků* představovat konstantní výraz.

Inicializátor_pole – podobný jako v C++. Ve složených závorkách počáteční hodnoty prvků pole, oddělené čárkami.

Následující deklarace

```
int[] pole = new int[10];
```

představuje pole 10 prvků typu `int` bez inicializace. Prvky se v takovém případě inicializují hodnotou 0 a nemusí se dále inicializovat.

Deklarace s inicializací:

```
int[] pole = new int[] { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

nebo

```
int[] pole = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Uvede-li se v hranatých závorkách počet prvků a zároveň se uvede inicializace, musí počet prvků v hranatých závorkách souhlasit s počtem prvků v inicializaci (v C++ lze inicializovat menší počet prvků, zbývající se inicializují na hodnotu 0).

Proměnná `pole` je reference, takže lze někde dále napsat příkaz

```
pole = new int[5];
```

Předchozí pole zanikne (automatická správa paměti jej později dealokuje) a vytvoří se nové.

Chybné zápisy deklarace pole:

```
int[] xpole = new int[3] { 0, 1 }; // nesouhlasí počet prvků
int n = 4;
int[] xpole = new int[n] { 0, 1, 2, 3 }; // nekonstantní počet prvků
```

Při vytvoření nového pole mimo deklaraci lze taktéž uvést inicializační část, ale pouze s operátorem `new`:

```
pole = new int[] { 0, 1, 2 }; // OK
pole = { 0, 2, 3 }; // Chyba
```

Počet prvků pole poskytuje vlastnost `Length`. Lze ji použít např. pro výpis prvků pole

```
static void VypisPole(int[] pole)
{
   for (int i = 0; i < pole.Length; i++) {
      Console.Write(pole[i] + " ");
   }
   Console.WriteLine();
}
```

Pro zkopírování pole slouží metoda `Clone()`, která vrací referenci na třídu `object`, takže se musí přetypovat:

```
int[] kopiePole = (int[])pole.Clone();
```

Pro zkopírování obsahu jednoho pole do jiného pole stejného typu s jiným počtem prvků slouží metoda `CopyTo`:

```
int[] pole2 = new int[15];
pole.CopyTo(pole2, 3);
```

Uvedený příkaz zkopíruje všechny prvky pole `pole` do pole `pole2`, ve kterém budou zkopírované prvky začínat na indexu 3. Cílové pole musí být dostatečně velké, jinak se vyvolá výjimka `System.ArgumentException`.

Další metody třídy `System.Array`:

- Statická metoda `BinarySearch` – binární hledání hodnoty v utříděném poli.
- Statická metoda `Clear` – do prvků pole uloží hodnotu 0, `false` nebo `null` podle typu prvků.
- Statická metoda `IndexOf` – vrací index prvního výskytu prvku pole mající zadanou hodnotu.
- Statická metoda `LastIndexOf` – vrací index posledního výskytu prvku pole mající zadanou hodnotu.

- Statická metoda `Sort` – setřídí prvky pole. Existuje několik verzí umožňujících utřídění části pole, určení způsobu porovnávání prvků pole aj.

Metody a vlastnosti třídy `System.Array` od verze .NET 2.0:

- Statická metoda `ConstrainedCopy` – zkopíruje zadaný počet prvků jednoho pole od zadaného indexu do jiného pole od zadaného indexu.
- Statická metoda `Find` – vrací hodnotu prvního prvku, pro nějž je pravdivý zadaný predikát.
- Statická metoda `FindLast` – vrací hodnotu posledního prvku, pro nějž je pravdivý zadaný predikát.
- Statická metoda `FindIndex` – vrací index prvního prvku, pro nějž je pravdivý zadaný predikát.
- Statická metoda `FindLastIndex` – vrací index posledního prvku, pro nějž je pravdivý zadaný predikát.
- Statická metoda `Resize` – zvětší nebe zmenší velikost pole. Do nově alokovaného pole zkopíruje prvky původního pole.

Vícerozměrná pole

Syntaxe specifikace typu:

typ_vícerozměrného_pole:

typ_ne_pole **[** *oddělovače_dimenzí* **]**

Oddělovače dimenzí – jedna nebo více čárek podle počtu rozměrů pole. Pro *k*-rozměrné pole je uvedeno *k* – 1 čárek.

Příklady deklarací bez inicializace – počet prvků pole může být v libovolné dimenzi nekonstantní celočíselný výraz:

```
int[,] matice = new int[2, 3];
int n = 2;
int[,,] matice3D = new int[2, 3, n*2];
```

Při deklaraci pole lze provést i inicializaci, podobně jako v C++ s uvedením počtu prvků (musí se jednat o konstantní výrazy – viz jednorozměrné pole):

```
int[,] matice2 = new int[2, 3] { { 0, 1, 2 }, { 3, 4, 5 } };
```

nebo bez uvedení počtu prvků:

```
int[,] matice2 = new int[,] { { 0, 1, 2 }, { 3, 4, 5 } };
```

Přístup k prvku matice na řádku `i` a ve sloupci `j` se provede výrazem

```
matice[i, j]
```

Nelze použít výraz známý z jazyka C++:

```
matice[i][j] // chyba
```

Vlastnosti a metody třídy `System.Array` pro práci s vícerozměrnými poli:

- Vlastnost `Rank` – počet dimenzí pole.
- Vlastnost `Length` – celkový počet prvků pole. Např. pro matici velikosti 2 x 3 má hodnotu 6.
- Metoda `GetLength` – vrací počet prvků v zadané dimenzi pole.

Výpis matice lze provést např. následující metodou:

```
static void VypisMatici(int[,] matice)
{
   for (int i = 0; i < matice.GetLength(0); i++) {
      for (int j = 0; j < matice.GetLength(1); j++) {
         Console.Write("{0,5}", matice[i,j]);
      }
      Console.WriteLine();
   }
}
```

Nepravidelná pole

Vícerozměrná pole jsou „obdélníkového“ tvaru. Jazyk C# nabízí kromě nich i *nepravidelná* pole (angl. jagged arrays).

Syntaxe specifikace typu:

typ_nepravidelného_pole:

typ_ne_pole specifikátory_řad

specifikátory_řad:

[ ]

[ ] *specifikátory_řad*

Specifikátory řad – dvě a více prázdných hranatých závorek. Specifikace *k*-rozměrného pole obsahuje *k* prázdných hranatých závorek.

Dvourozměrné nepravidelné pole je pole odkazů na pole.

Nepravidelné pole se vytváří postupně. Nejprve se vytvoří pole 1. rozměru, potom pole 2. rozměru atd.

Např. dvourozměrné pole:

```
int[][] neprPole = new int[2][];
neprPole[0] = new int[2] { 0, 1 };
neprPole[1] = new int[3] { 2, 3, 4 };
```

Přístup k prvku uvedeného pole na indexu `i`, `j` se provede výrazem

```
neprPole[i][j]
```

Jazyk C# umožňuje deklarovat i nepravidelná pole složená z pravidelných polí, např.

```
int[][,] smisenePole;
```

Delegáty

Delegát (angl. delegate) je něco jako objektově zapouzdřený ukazatel na metodu. Delegát může obsahovat:

- adresu statické metody,
- adresu nestatické metody spolu s adresou instance, pro níž má být metoda volána.

Metoda, na níž delegát ukazuje, se zavolá podobně jako se v C++ zavolá metoda pomocí ukazatele. Za identifikátor instance delegátu se zapíše do kulatých závorek seznam skutečných parametrů metody.

Syntaxe deklarace delegátu:

deklarace_typu_delegátu:

modifikátor$_{nep}$ **delegate** *typ jméno* **(** *seznam_parametrů*$_{nep}$ **) ;**

Modifikátor – přístupová práva.

Typ – typ návratové hodnoty metody, na níž budou delegáty tohoto typu ukazovat. Může to být i typ `void`.

Jméno – identifikátor deklarovaného typu delegátu. Pokud delegát není použit jako typ události, doporučuje se, aby měl příponu `Callback`. Neměl by mít příponu `Delegate`.

Seznam_parametrů – seznam parametrů metody, na níž budou delegáty ukazovat. Seznam musí obsahovat i jména parametrů. Jména parametrů delegátu mohou být odlišná od jmen parametrů metody, na níž delegát ukazuje. Seznam může být i prázdný – metoda bez parametrů.

Deklarace typu delegátu se může objevit v prostoru jmen (včetně globálního) nebo uvnitř třídy nebo struktury (vnořený typ). Nelze deklarovat lokální typ delegátu uvnitř metody.

Instance delegátu se vytvoří pomocí operátoru `new`, za nímž se uvede jméno delegátu a v závorkách jméno metody. Jedná se vlastně o volání konstruktoru delegátu, kterému se jako parametr předá jméno metody, které může být v případě potřeby kvalifikované jménem instance třídy nebo jménem třídy. Od verze .NET 2.0 lze vytvořit instanci delegátu přiřazením pouze jména metody.

Příklad

V uvedeném příkladu se vypíše tabulka hodnot funkce sinus a druhé mocniny.

```
// deklarace typu delegátu:
public delegate double FunkceCallback(double x);

class Tabelace
{
   public void Tabulka(FunkceCallback f, double od, double @do,
      double krok)
   {
      Console.WriteLine("x     f(x)");
      for (double x = od; x <= @do; x += krok) {
         Console.WriteLine("{0:F1}   {1:F5}", x, f(x));
      }
   }
   public double Mocnina(double x)
   {
      return x * x;
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      Tabelace t = new Tabelace();
      FunkceCallback f = new FunkceCallback(Math.Sin);
         // Funkce Sin je statická metoda třídy Math
      // FunkceCallback f = Math.Sin; // dtto předchozí příkaz
      Console.WriteLine("Tabulka funkce sinus");
      t.Tabulka(f, 0, 1, 0.2);
      Console.WriteLine("\nTabulka funkce druhé mocniny");
      t.Tabulka(new FunkceCallback(t.Mocnina), 0, 5, 1);
      // t.Tabulka(t.Mocnina, 0, 5, 1); // dtto předchozí příkaz
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
Tabulka funkce sinus
x     f(x)
0,0   0,00000
0,2   0,19867
0,4   0,38942
0,6   0,56464
0,8   0,71736
1,0   0,84147

Tabulka funkce druhé mocniny
x     f(x)
0,0   0,00000
1,0   1,00000
2,0   4,00000
3,0   9,00000
4,0   16,00000
5,0   25,00000
```

V příkazu #1 je použit výpis textu `"{0:F1}  {1:F5}"`, v němž 0 a 1 představují pořadová čísla parametrů funkce `WriteLine`, tj. 0 představuje parametr `x` a 1 parametr `f(x)`. Texty `F1` a `F5` představují formát výpisu reálných čísel na 1 a 5 desetinných míst.

Pro horní mez funkce `Tabulka` nebylo možné použít identifikátor `do`, protože se jedná o klíčové slovo. Proto se použil identifikátor `@do`.

Vícenásobný delegát (angl. *multicast delegate*) – delegát ukazující na metody, vracející typ `void`. Jedna instance vícenásobného delegátu může obsahovat ukazatele na několik metod. Nové metody lze do delegátu připojit pomocí operátoru += (resp. = a +) a odebrat pomocí operátoru -= (resp. = a -). Při volání vícenásobného delegátu se zavolají všechny metody, na něž delegát ukazuje, a to v pořadí, v jakém do něj byly vloženy. Delegát může ukazovat i na více stejných metod.

Příklad

```
delegate void FunkceCallback(int x, int y);

class MatOper
{
   public static void Součet(int x, int y)
   {
      Console.WriteLine("{0} + {1} = {2}", x, y, x + y);
   }
   public static void Násobení(int x, int y)
   {
      Console.WriteLine("{0} * {1} = {2}", x, y, x * y);
   }
   public static void Dělení(int x, int y)
   {
      Console.WriteLine("{0} / {1} = {2}", x, y, (double)x / y);
   }
}

class Program
{
   static void Main(string[] args)
   {
      FunkceCallback f = new FunkceCallback(MatOper.Součet);
      f += new FunkceCallback(MatOper.Dělení);
      //f = f + new FunkceCallback(MatOper.Dělení); dtto předchozí příkaz
      FunkceCallback f1 = new FunkceCallback(MatOper.Násobení);
      f += f1;
      f(5, 2);
      f -= f1;
      f(10, 2);
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
5 + 2 = 7
5 / 2 = 2,5
5 * 2 = 10
10 + 2 = 12
10 / 2 = 5
```

Delegát je zvláštní druh referenčního typu. Přiřazovací operátorem sice dojde pouze ke kopii reference, stejně jako u třídy, ale pokud později prostřednictvím jedné reference dojde k modifikaci delegátu přidáním nebo odebráním metody, druhá reference se bude odkazovat na původní delegát.

Příklad

Je dán stejný delegát a třída `MatOper` jako v předchozím příkladu. Odlišná je pouze metoda `Main`.

```
static void Main(string[] args)
{
   FunkceCallback f = new FunkceCallback(MatOper.Součet);
   FunkceCallback g = f; // #1
   f += MatOper.Dělení;
   Console.WriteLine("Volání delegátu f");
   f(5, 2);
   Console.WriteLine("Volání delegátu g");
   g(5, 2);
   Console.ReadKey();
}
```

Po provedení příkazu #1 se reference `g` odkazuje na stejnou instanci delegátu jako reference `f`. Avšak přidáním další metody do instance delegátu pomocí reference `f` dojde k vytvoření nové instance delegátu. Na původní instanci bude odkazovat reference g a na novou instanci reference `f`. Výstup programu bude tudíž následující:

```
Volání delegátu f
5 + 2 = 7
5 / 2 = 2,5
Volání delegátu g
5 + 2 = 7
```

Od verze .NET 2.0 byla zavedena vlastnost *kovariance* (angl. *covariance*) pro návratový typ delegátu a *kontra-variance* (angl. *contra-variance*) pro typ parametru delegátu. Návratovým typem metody, na kterou instance delegátu ukazuje, může být potomek návratového typu delegátu. Parametrem metody, na kterou instance delegátu ukazuje, může být předek parametru delegátu. V obou případech musí být předek a potomek referenčního typu. To znamená, že vlastnost kovariance neplatí pro konverzi z hodnotového typu na typ `object` a kontra-variance z typu `object` na hodnotový typ.

Příklad

```
delegate object FunkceCallback(string s);
delegate object Funkce2Callback(int s);

class Program
{
   static string Retezec(object s)
   {
      return "Objekt " + s;
   }

   static void Main(string[] args)
   {
      FunkceCallback f = new FunkceCallback(Retezec);
      //Funkce2Callback f2 = new Funkce2Callback(Retezec); // Chyba
      Console.WriteLine(f("text"));
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
Objekt text
```

Třída `string` je nepřímým potomkem třídy `objekt` a reprezentuje řetězec znaků.

Druhý příkaz metody `Main` je chybný, protože parametr delegátu `Funkce2Callback` je hodnotového typu.

Od verze .NET 2.0 byly zavedeny *anonymní metody* (angl. *anonymous methods*). Jedná se o blok příkazů, který se přímo přiřazuje instanci delegátu.

Syntaxe:

výraz_anonymní_metody:

delegate *signatura_anonymní_metody*$_{nep}$ *blok*

Signatura_anonymní_metody – seznam formálních parametrů anonymní metody v kulatých závorkách. Může se vynechat, pokud formální parametry nejsou v anonymní metodě využity, nebo pokud delegát je bez parametrů.

Blok – složený příkaz. V bloku se mohou vyskytovat i lokální proměnné metody, v níž je anonymní metoda deklarována – tzv. *vnější proměnné* (angl. *outer variables*).

Příklad

```
delegate int OperaceCallback(int x, int y);
delegate void VypisCallback();

class Program
{
   static void VypisOperace(int a, int b, OperaceCallback f)
   {
      Console.WriteLine("f({0}, {1}) = {2}", a, b, f(a, b));
   }
   static void Main(string[] args)
   {
      OperaceCallback f;
      f = delegate(int x, int y) { return x + y; };
      VypisOperace(5, 10, f);
      // dtto předchozí 3 příkazy:
      // VypisOperace(5, 10, delegate(int x, int y) { return x + y; });
      int a = 20;
      VypisCallback v = delegate
      {
         Console.WriteLine("a = " + a); // a - vnější proměnná
      };
      v();
      Console.ReadKey();
   }
}
```

Výstup programu bude následující:

```
f(5, 10) = 15
f(5, 10) = 15
a = 20
```

Každý delegát je třídou odvozenou od třídy `System.MulticastDelegate`, která je potomkem třídy `System.Delegate`. Vícenásobný delegát využívá všechny možnosti třídy `System.MulticastDelegate`, obyčejný delegát využívá jen možnosti třídy `System.Delegate`. Volání metody prostřednictvím delegátu je synchronní a provádí jej metoda `Invoke`. Asynchronní volání lze provést pomocí metod `BeginInvoke` a `EndInvoke`.

Rozhraní

Rozhraní obsahuje pouze seznam metod, indexerů (přetížených operátorů indexování), vlastností a událostí bez jejich definice (těla). Třída, která implementuje dané rozhraní, musí definovat všechny složky daného rozhraní. Bližší informace o rozhraních – viz samostatná kapitola dále.

Některé systémové třídy

Třída object

Slovo `object` je v jazyce C# klíčovým slovem a je synonymem pro třídu `System.Object`.

Tato třída je společným předkem všech datových typů v jazyce C#. To znamená, že její metody lze volat i pro instance hodnotových typů.

Má jeden veřejně přístupný konstruktor bez parametrů a následující veřejné metody:

- Virtuální metoda `bool Equals(object obj)` – vrací `true`, pokud instance `this` je rovna instanci, na kterou se odkazuje `obj`.
- Statická metoda `bool Equals(object objA, object objB)` – dtto virtuální metoda `Equals`.
- Virtuální metoda `int GetHashCode()` – vrací hešovací kód instance. Využívá se pro ukládání instance do hešovací tabulky (`System.Collections.Hashtable`).
- Metoda `Type GetType()` – vrací instanci třídy `Type` poskytující informace o skutečném typu instance. Lze ji využít pro dynamické určování typu nebo pro tzv. reflexi (viz dále).
- Statická metoda `bool ReferenceEquals(object objA, object objB)` – vrací `true`, pokud `objA` a `objB` se odkazují na stejnou instanci.
- Virtuální metoda `string ToString()` – vrací řetězec znaků reprezentující instanci. Implementace v třídě `object` vrací skutečné jméno třídy.

Dále obsahuje dvě chráněné metody:

- Virtuální metoda `void Finalize()` – je volána automatickou správou paměti před vlastní dealokací instance.
- Metoda `object MemberwiseClone()` – vytvoří mělkou kopii instance, kterou vrací. Pracuje se skutečným typem instance (viz dále).

Odkazy (reference) na třídu `object` lze v C# použít podobně jako ukazatel typu `void*` v C++.

Protože potomek může vždy zastoupit předka, lze proměnné typu `object` přiřadit hodnotu představující odkaz na instanci jakékoliv třídy v C# včetně hodnotového typu (viz dále kapitola „Zabalení“).

Pokud je potřebné pracovat s instancí, na níž se odkazuje proměnná typu reference na `object`, musí se přetypovat na její skutečný typ. Program v C# vždy kontroluje, zda má takové přetypování smysl a pokud ne, vznikne výjimka typu `System.InvalidCastException`.

Třída string

Pro práci s řetězci znaků slouží třída `string`. Jedná se o klíčové slovo a je synonymem pro třídu `System.String`.

Novou instanci třídy lze vytvořit např. příkazem

```
string s1 = "Česká Třebová";
```

Konstanta `"Česká Třebová"` je také instancí třídy `string`, nikoli polem znaků, jak je tomu v C++. Např. lze napsat příkaz, který do proměnné `n` uloží délku řetězce:

```
int n = "Česká Třebová".Length;
```

Vybrané veřejné složky třídy `string`:

- Konstruktor `string(char[] s)` – vytvoří řetězec z pole znaků.
- Konstruktor `string(char c, int n)` – vytvoří řetězec, v němž se bude `n` krát opakovat znak `c`.
- Statická datová složka `Empty` – prázdný řetězec.
- Vlastnost `Length` – počet znaků řetězce.
- Statické metody `Compare` – slouží k porovnání dvou řetězců podle aktuální nebo zadané kultury, bez ohledu nebo s ohledem na velikost znaků aj.
- Statické metody `Concat` – slouží ke spojení několika řetězců. Jako parametry lze zadat i jiné objekty, pro které se nejprve zavolá metoda `ToString`.
- Metody `EndsWith` – vrací `true`, pokud řetězec končí zadaným řetězcem. Porovnání řetězců lze provést pomocí aktuální nebo zadané kultury s nebo bez ignorování velikostí písmen.
- Metoda `int IndexOf(char c)` – vrací index prvního výskytu znaku `c` v řetězci.
- Metoda `string Insert(int index, string s)` – vrací řetězec, který vznikne vložením řetězce `s` do řetězce `this` na pozici `index`. Řetězec `this` se nezmění.
- Metoda `string Remove(int index, int n)` – vrací řetězec, který vznikne z řetězce `this` odstranění `n` znaků od pozice `index`. Řetězec `this` se nezmění.
- Metody `StartWith` – vrací `true`, pokud řetězec začíná zadaným řetězcem. Způsob porovnání řetězců je stejný jako u metod `EndsWith`.
- Metody `ToLower` resp. `ToUpper` – vrací řetězec, který vznikne převodem řetězce `this` na malá resp. velká písmena podle aktuální nebo zadané kultury. Řetězec `this` se nezmění.
- Metoda `string Trim()` – vrací řetězec, který vznikne odstraněním bílých znaků z počátku a konce řetězce `this`. Řetězec `this` se nezmění. Přetížená verze umožňuje odstranit zadané znaky.
- Metoda `bool Equals(object obj)` – vrací `true`, pokud řetězec `this` je shodný s řetězcem `obj`.
- Operátory == resp. `!=` – vrací výsledek metody `Equals` resp. negaci jejího výsledku.
- Operátory + a += – slouží k spojení dvou řetězců.
- Operátor indexování `[]` – slouží k přístupu ke znaku na zadaném indexu pouze pro čtení.

Řetězcový literál (konstanta) se zapisuje do uvozovek. Nezobrazitelné znaky mohou být uvedeny pomocí řídících posloupností. Např. text rozdělený ve výpisech na dva řádky:

```
string s2 = "Nějaký delší text A\nNějaký delší text B";
```

Narozdíl od C++ nelze v C# zapsat uvedený text na dva řádky zdrojového programu, přičemž první řádek by končil uvozovkami a druhý řádek jimi začínal. Texty na dvou řádcích lze v C# ale spojit operátorem +:

```
string s2 = "Nějaký delší text A\n" +
            "Nějaký delší text B";
```

Vedle toho nabízí jazyk C# ještě tzv. *doslovné řetězcové konstanty* (angl. *verbatim string literal*). Ty jsou také uzavřeny v uvozovkách a před ně se ještě zapisuje znak `@`. Doslovný řetězcový literál může obsahovat libovolné zobrazitelné znaky kromě uvozovek, a to včetně řídících posloupností a

přechodů na nový řádek. Tyto znaky budou chápány doslovně – to znamená, že přechod na nový řádek se stane součástí literálu, znaky řídící posloupnosti budou chápany jako jednotlivé znaky (nebudou interpretovány jako řídící posloupnost). Pokud mají být uvozovky součástí doslovné řetězcové konstanty, musí se zdvojit. Např. příkaz

```
Console.WriteLine(@"Text na prvním řádku
""Text na druhém řádku v uvozovkách"".
Řídící posloupnost \n způsobí přechod na nový řádek");
```

provede následující výpis

```
Text na prvním řádku
"Text na druhém řádku v uvozovkách".
Řídící posloupnost \n způsobí přechod na nový řádek
```

Znakový řetězec, uložený v instanci `string`, nelze měnit. Pro úpravu řetězce slouží třída `System.Text.StringBuilder`. Ta obsahuje např. metody:

- Metody `Append` – přidá na konec řetězce text, číslo převedené na řetězec nebo `object` převedený na řetězec.
- Metoda `Remove` – odstraní z řetězce znaky od do zadaného indexu.
- Metody `Replace` – nahradí řetězec jiným řetězcem nebo znak jiným znakem.
- Operátor indexování `[]` – slouží k získání znaku na zadaném indexu nebo k uložení znaku na zadaný index.

Třída `StringBuilder` alokuje pro řetězec větší pole znaků než kolik řetězec zabírá – má kapacitu, podobně jako šablona třídy `vector` v C++. Kapacitu je možné získat nebo nastavit prostřednictvím vlastnosti `Capacity`.

Zabalení

Někdy se stane, že na místě, kde se očekává instance referenčního typu, je potřebné použít hodnotový typ. Typickým příkladem jsou předdefinované kolekce, které jsou deklarovány v prostoru jmen `System.Collections`. Ty pracují s prvky typu `object`, což je referenční typ.

Pro převod instance hodnotového typu na instanci referenčního typu nabízí C# operaci *zabalení* (angl. *boxing*) a pro opačný převod *vybalení* (angl. *unboxing*).

Pokud se přiřadí odkazu na typ `object` hodnota hodnotového typu,

```
object o = 11;
```

vytvoří překladač instanci „obalové“ třídy, do níž uloží hodnotu uvedenou na pravé straně. Tuto hodnotu lze kdykoliv získat zpět přetypováním

```
int i = (int)o;
```

Přitom se za běhu programu kontroluje, zda má toto přetypování smysl, a pokud nemá, vyvolá se výjimka typu `System.InvalidCasException`.