TŘÍDY – POKRAČOVÁNÍ

Metody – pokračování

Externí metody

V programu C# lze volat i funkci naprogramovanou v jazyce C/C++ nebo v jiném programovacím jazyce a uloženou v klasické dynamické knihovně pro neřízený kód. Tato funkce se musí v C# deklarovat jako externí statická metoda s atributem `DllImport`, v němž se uvede jméno souboru dynamické knihovny, v níž se funkce nachází. Takto lze volat i funkce Windows API:

Syntaxe:

deklarace externí metody:

[DllImport(*knihovna* **)]** *modifikátory*$_{\text{nep}}$ **static extern** *typ identifikátor* **(***seznam_parametrů*$_{\text{nep}}$**);**

Modifikátory – modifikátory přístupových práv.

Typ, identifikátor a seznam_parametrů – dtto ostatní metody.

Knihovna – řetězec znaků představující jméno souboru s dynamickou knihovnou.

Volání externí metody se nijak neliší od volání jiných statických metod.

Poznámky:

- Atribut `DllImport` je deklarován v prostoru jmen `System.Runtime.InteropServices`.
- Jazyk C# nepředepisuje, že deklarace externí metody se musí uvádět s atributem `DllImport`, ale současná implementace však jinou možnost nenabízí.
- Atribut `DllImport` může mít i další parametry, které umožňují použít pro externí metodu jiný identifikátor, než jaký má v dynamické knihovně aj.

Příklad

```
class Program
{
   [DllImport("user32.dll")]
   static extern int MessageBox(int windowHandle, string text,
      string caption, uint type);

   static void Main(string[] args)
   {
      MessageBox(0, "Ukázka volání funkce MessageBox z Windows API",
         "Hlášení", 0);
   }
}
```

Funkce `MessageBox` z dynamické knihovny user32.dll má druhý a třetí parametr typu `LPCSTR`, tedy `const char*` resp. `const wchar_t*`. Prostředí .NET se postará o automatickou konverzi (ang. *marshalling)* parametru typu `string` na tento typ, jakož i o další konverze.

Metoda Main

Metoda `Main` představuje vstupní bod konzolové i okenní aplikace. Má čtyři podoby:

```
static int Main() {}
static int Main(string[] args) {}
static void Main() {}
static void Main(string[] args) {}
```

Metoda `Main` může být deklarována s jakýmkoli specifikátorem přístupových práv. Je-li návratový typ metody `Main` typu `int`, vrací kód ukončení programu, podobně jako v C++. Parametr `args` představuje parametry, které jsou zadány na příkazovém řádku za názvem programu při jeho spuštění. Na rozdíl od C++ není jméno programu nultým parametrem.

Metodu `Main` lze přetěžovat, avšak jako vstupní bod programu může sloužit jen metoda s jednou z uvedených signatur.

Metodu `Main` může obsahovat i více tříd v programu. V takovém případě se musí určit, kterou z nich má překladač označit jako vstupní bod programu. To lze provést pomocí přepínače `/main:`*třída*, kde *třída* je jméno třídy, jejíž metoda `Main` se má zavolat při spuštění programu. Ekvivalentem k uvedenému přepínači je menu **Project | Properties**, část **Application**, rozevírací seznam **Startup object**.

Podmíněné metody

Metodu s návratovým typem `void` lze deklarovat jako podmíněnou. Slouží k tomu atribut `Conditional`, definovaný v prostoru jmen `System.Diagnostic`.

Syntaxe:

deklarace podmíněné metody:

[Conditional ("*jméno*" **)]** *deklarace_metody_typu_void*

Jméno – identifikátor podmíněného symbolu deklarovaného pomocí direktivy `#define`.

Deklarace_metody_typu_void – deklarace metody s návratovým typem `void`.

Je-li identifikátor *jméno* definován, přeloží se tato metoda i všechna její volání stejně, jako kdyby atribut `Conditional` nebyl u metody uveden.

Pokud identifikátor *jméno* není definován, překladač odstraní všechna volání této metody ze zdrojových textů programu, ale metodu přeloží.

Příklad

```
#define LADĚNÍ

using System;
using System.Diagnostics;

class Ladění
{
   [Conditional("LADĚNÍ")]
   public static void Výpis(string s, int x)
   {
      Console.WriteLine("Proměnná {0} má hodnotu {1}", s, x);
   }
}
class Program
{
   static void Main(string[] args)
   {
      int a = 1, b = 2;
      Ladění.Výpis("a", a); // #1
      Ladění.Výpis("b", a); // #2
      Console.WriteLine("Součet proměnných je {0}", a + b);
      Console.ReadKey();
   }
}
```

Pokud se uvedeném příkladu odstraní direktiva `#define`, odstraní se i příkazy #1 a #2.

Konstruktory a destruktory

Podobně jako v C++ slouží i v C# konstruktor k vytvoření a inicializaci instance a destruktor se volá při jejím zániku.

Hlavní rozdíly mezi pojetím těchto metod:

- V C# lze deklarovat také statický konstruktor.
- V C# může konstruktor volat jiný konstruktor téže třídy.
- Konstruktor v C# nelze použít k implicitní konverzi.
- V C# neexistuje obdoba kopírovacích konstruktorů.
- V C# je třeba volat konstruktor vždy explicitně, a to pomocí klíčového slova `new`.
- Role destruktorů je v C# výrazně menší než v C++. Destruktory nelze explicitně volat.

Konstruktor instance

Konstruktor instance (instanční konstruktor) (angl. *instance constructor*) je konstruktor známý z C++.

Syntaxe:

deklarace konstruktoru instance:

modifikátory$_{\text{nep}}$ *identifikátor* **(** *seznam_parametrů*$_{\text{nep}}$ **)** *inicializátor*$_{\text{nep}}$ *tělo*

inicializátor:

: this (*seznam_parametrů*$_{\text{nep}}$ **)**

: base (*seznam_parametrů*$_{\text{nep}}$)

Modifikátory – modifikátory přístupových práv.

Identifikátor – shodný s identifikátorem třídy, jejíž složkou je deklarovaný konstruktor.

Seznam_parametrů – stejný význam jako u jiných metod.

Tělo – blok příkazů.

Inicializátor s klíčovým slovem `this` slouží k volání jiného konstruktoru téže třídy.

Inicializátor s klíčovým slovem `base` slouží k volání konstruktoru bezprostředního předka.

Konstruktory se nedědí.

Jestliže v třídě není žádný uživatelem deklarovaný konstruktor, vytvoří překladač *implicitní konstruktor* (angl. *default constructor*), který je veřejně přístupný, bez parametrů a s prázdným tělem. Jestliže třída má uživatelem deklarovaný konstruktor, překladač žádný další konstruktor nevytvoří.

Přiklad

```
class Matice
{
   double[,] a;
   public Matice(int m, int n)
   {
      a = new double[m, n];
   }
   public Matice(int m, int n, double hodnota) : this(m, n)
   {
      for (int i = 0; i < a.GetLength(0); i++) {
         for (int j = 0; j < a.GetLength(1); j++ ) a[i, j] = hodnota;
      }
   }
   public void Výpis() { ... }
}
class Program
{
   static void Main(string[] args)
   {
      Matice matice = new Matice(3, 2);
      matice.Výpis();
      matice = new Matice(3, 2, 10.54);
      matice.Výpis();
      // Matice matice2 = new Matice();
      // Chyba - třída nemá implicitní konstruktor
   }
}
```

Doporučení pro konstruktory

Doporučuje se, aby parametr konstruktoru, kterým se inicializuje určitá vlastnost třídy (a jí případně odpovídající datová složka), měl stejný název jako jemu odpovídající vlastnost. Rozdíl je pouze ve velikosti písmen – pro parametr se používá velbloudí styl a pro veřejnou a chráněnou vlastnost pascalovský styl.

Statický konstruktor

Statický konstruktor slouží k inicializaci statických složek třídy. V C++ nemá obdobu. Statické datové složky lze inicializovat v deklaraci nebo ve statickém konstruktoru.

Syntaxe:

deklarace statického konstruktoru:

static *identifikátor* **( )** *tělo*

Identifikátor – identifikátor třídy, v němž je konstruktor definován.

Tělo – blok příkazů. V něm lze pracovat pouze se statickými složkami této třídy – volat statické metody a používat statické datové složky.

V deklaraci statického konstruktoru nelze použít modifikátory specifikující přístupová práva. Statický konstruktor nemůže mít parametry. Nelze jej volat, program jej zavolá sám při zavedení třídy do paměti. Statický konstruktor se nedědí.

Příklad

```
class Matice
{
   double[,] a;
   static readonly int rozměr;
   static Matice()
   {
      Console.Write("Zadej rozměr matice: ");
      rozměr = Convert.ToInt32(Console.ReadLine());
   }
   public Matice()
   {
      // rozměr = 3; // Chyba
      a = new double[rozměr, rozměr];
   }
   public Matice(double hodnota)
      : this()
   {
      for (int i = 0; i < rozměr; i++) {
         for (int j = 0; j < rozměr; j++) a[i, j] = hodnota;
      }
   }
   // ...
}
```

Destruktor

Před verzí 2.0 normy C# se destruktor nazýval *destructor*, ale nyní se nazývá *finalizer*, aby programátoři jazyka C++ nepokládali destruktor C# za obdobu destruktoru v C++. V české literatuře se vyskytuje jak pojem *destruktor* tak i *finalizér*. V těchto přednáškách je použit pojem *destruktor*.

Úloha destruktoru v C# je jiná než v C++. Destruktor v C# představuje syntaktickou zkratku pro předefinovanou virtuální metodu `Finalize()` třídy `object`. Tuto metodu zavolá automatická správa paměti při uvolnění instance z paměti. To ale nemusí nastat bezprostředně po zániku posledního odkazu na danou instanci, nýbrž kdykoli později, nejpozději při ukončení programu.

Destruktor v C# lze využít k uvolňování jiných prostředků než je paměť. Může jít o otevřené soubory, databázová a síťová spojení atd.

Syntaxe:

deklarace destruktoru:

~ identifikátor **( )** *tělo*

Identifikátor – identifikátor třídy, v němž je destruktor deklarován.

V deklaraci destruktoru nelze použít modifikátory specifikující přístupová práva. Destruktor nemůže mít parametry, nelze jej explicitně volat a nedědí se.

V C# se destruktory používají poměrně zřídka. V rozsáhlejších programech mohou způsobit zdržení při automatickém úklidu paměti, protože objekty, které nemají destruktor, se odstraňují okamžitě, zatímco objekty s destruktorem se nejprve odkládají stranou, potom se pro ně volá jejich destruktor (metoda `Finalize`) a teprve pak jsou odstraněny.

Destruktory se zpravidla kombinují s metodou `Dispose()` rozhraní `IDisposable` – viz popis rozhraní `IDisposable`.

Vlastnosti

Vlastnosti v normě jazyka C++ neexistují. Lze se s nimi setkat v některých rozšířeních jazyka C++, jako je např. Borland C++ Builder.

Vlastnost (angl. *property*) je syntaktická zkratka pro dvojici přístupových metod.

V programu se používá vlastnost prostřednictvím jejího identifikátoru. Přiřazením hodnoty do vlastnosti se zavolá jedna z přístupových metod. Ta může např. uložit danou hodnotu do soukromé datové složky a provést i řadu jiných operací.

Při použití hodnoty vlastnosti se zavolá druhá z dvojice metod. Ta může vrátit hodnotu uloženou v soukromé datové složce nebo vrátit hodnotu zjištěnou nějakým výpočtem aj.

Deklarace vlastnosti

Syntaxe:

deklarace vlastnosti:

modifikátory$_{\text{nep}}$ *typ jméno* **{** *část_set část_get*$_{\text{nep}}$ **}**

modifikátory$_{\text{nep}}$ *typ jméno* **{** *část_get část_set*$_{\text{nep}}$ **}**

Modifikátory – modifikátory přístupových práv a modifikátory vyjadřující, zda jde o statickou, virtuální, abstraktní, zapečetěnou, předefinovanou nebo zastiňující vlastnost.

Typ – typ hodnoty vlastnosti.

Jméno – jméno (identifikátor) vlastnosti.

Část_set – přístupová metoda pro nastavení hodnoty vlastnosti.

Část_get – přístupová metoda pro zjištění hodnoty.

Na pořadí uvedení *části get* a *části set* nezáleží. Jednu z nich lze vynechat. Vynecháním *části set* se získá vlastnost „jen pro čtení“, jejíž hodnotu nelze měnit. Vynecháním *části set* se získá vlastnost „jen pro zápis“, jejíž hodnotu lze měnit, nikoli zjišťovat.

Syntaxe těchto částí:

část_get:

modifikátory$_{\text{nep}}$ **get** *tělo*

část_set:

modifikátory$_{\text{nep}}$ **set** *tělo*

Tělo – blok příkazů. U abstraktní vlastnosti je uveden místo bloku středník.

Modifikátory – modifikátory přístupových práv. Jsou zavedeny od verze .NET 2.0. Jsou-li modifikátory u dané části uvedeny, musí reprezentovat přístupové právo, které více omezuje přístup k dané části než modifikátory přístupových práv pro vlastnost jako celek – viz následující tabulka. Modifikátory lze specifikovat jen u jedné z částí vlastnosti. Pokud vlastnost obsahuje pouze *část get* nebo *část set*, nelze modifikátory u této části specifikovat. Modifikátory nelze použít u vlastnosti deklarované v rozhraní nebo v explicitní implementaci rozhraní (viz kapitola o rozhraních).

Modifikátory ve vlastnosti	Povolené modifikátory u get a set

`public`	`protected internal`
	`internal`
	`protected`
	`private`

`protected internal`	`internal`
	`protected`
	`private`

`internal`	`private`
`protected`	`private`
`private`	

V těle *části set* je definována proměnná `value` typu shodného s typem vlastnosti. Tato proměnná obsahuje hodnotu přiřazovanou vlastnosti.

Tělo *části get* musí obsahovat příkaz

`return` *výraz;*

který poskytuje vrácenou hodnotu vlastnosti.

Příklad

```
class KomplexCislo
{
   private int real;
   private int imag;

   public KomplexCislo(int real, int imag)
   {
      this.real = real;
      this.imag = imag;
   }
   public int Real
   {
      get { return real; }
      set { real = value; }
   }
   public int Imag
   {
      get { return imag; }
      set { imag = value; }
   }
   public double Abs
   {
      get { return Math.Sqrt(Real*Real + Imag*Imag); }
   }
}
```

```
class Program
{
   static void Main(string[] args)
   {
      KomplexCislo kc = new KomplexCislo(10, 20);
      Console.WriteLine("Absolutní hodnota čísla ({0} + {1}i) je {2:F1}",
         kc.Real, kc.Imag, kc.Abs);
      kc.Real = 3;
      kc.Imag = 4;
      Console.WriteLine("Absolutní hodnota čísla ({0} + {1}i) je {2:F1}",
         kc.Real, kc.Imag, kc.Abs);
      Console.ReadKey();
   }
}
```

Statické vlastnosti

Statická vlastnost je deklarována s modifikátorem `static`. Lze ji použít, i když neexistuje žádná instance dané třídy. Při použití statické vlastnosti mimo její třídu se kvalifikuje jménem její třídy.

Část get a *část set* statické vlastnosti může pracovat pouze se statickými složkami dané třídy.

Doporučení pro vlastnosti

Vlastnost by měla mít stejný název jako jí odpovídající datová složka, rozdíl je pouze ve velikosti písmen. Pro soukromou datovou složku se používá velbloudí styl a pro veřejnou a chráněnou vlastnost pascalovský styl.

Doporučuje se použít metodu místo vlastnosti, jestliže se jedná o řádově pomalejší operaci než je nastavení hodnoty do datové složky třídy, např. přístup k síti nebo k souboru.

Nedoporučuje se deklarovat vlastnosti, které umožňují pouze nastavení hodnoty. V takovém případě se má nastavení hodnoty realizovat metodou.

Nastavení vlastností dané třídy by nemělo být závislé na pořadí, i kdyby instance třídy byla dočasně v neplatném stavu.

Pokud při nastavení vlastnosti dojde k vyvolání výjimky, měla by si vlastnost uchovat předchozí hodnotu.

Část get by neměla vyvolávat výjimku.

Události

Jazyk C++ zpracování událostí neřeší ani na úrovni syntaxe, ani na úrovni knihoven, specifikovaných normou. Nadstavbové knihovny jednotlivých vývojových prostředí (MFC, VCL apod.) si řeší problematiku událostí po svém.

Jazyk C# naproti tomu nabízí jednotný mechanismus zpracování událostí. Je založen na návrhovém vzoru *vydavatel – předplatitel* (*publisher – subscriber*), známém také z jazyka Java:

- Instance, v níž může událost vzniknout (*vydavatel*), si vede seznam instancí, jež mají zájem na tuto událost reagovat (*předplatitelů*).
- Instance, jež má zájem reagovat na určitou událost, tedy předplatitel, si u vydavatele registruje metodu, která se o reakci postará. K tomu použije delegát.
- Nestane-li událost, vyrozumí o ní vydavatel své předplatitele tím, že zavolá jejich registrované metody.

Handler a údaje o události

Metody, které reagují na události, se označují jako handlery. Jsou vždy typu `void` a měly by mít dva parametry. První parametr by měl být typu `object` a měl by představovat odkaz na instanci,

jež událost vyvolala. Druhý by měl být typu `System.EventArgs` nebo typu, který je jeho potomkem. Signatura handleru může vypadat např. takto

```
void Stisk(object sender, EventArgs e) {}
```

Knihovna BCL obsahuje řadu potomků třídy `EventArgs` pro běžné typy událostí, jako je stisknutí tlačítka myši, stisknutí klávesy apod.

Pokud událost umožňuje stornování činnosti, druhý parametr handleru by měl být typu `CancelEventArgs` nebo typu, který je jeho potomkem. Typ `CancelEventArgs` obsahuje jednu vlastnost `Cancel`, která poskytuje nebo nastavuje hodnotu, udávající, zda činnost, která je s událostí spojená, by měla být stornována.

Pro zpracování událostí nabízí knihovna BCL celou řadu předdefinovaných delegátů. Lze si však deklarovat i vlastní. Jméno delegátu by mělo obsahovat příponu `EventHandler`. Potomek třídy `EventArgs` by měl obsahovat příponu `EventArgs`. Signatura vlastního delegátu by mohla vypadat např. takto:

```
delegate void NecoSeStaloEventHandler(object sender, MojeEventArgs e);
```

Vzhledem k tomu, že handler je typu `void`, může odpovídající delegát obsahovat ukazatele na více handlerů.

Deklarace události

Událost se deklaruje jako složka třídy.

Syntaxe:

deklarace události:

modifikátory$_{\text{nep}}$ **event** *typ jméno* **;**

modifikátory$_{\text{nep}}$ **event** *typ jméno* **{** *části_add_remove* **}**

Modifikátory – modifikátory přístupových práv a modifikátory vyjadřující, zda jde o statickou, virtuální, abstraktní, zapečetěnou, předefinovanou nebo zastiňující událost.

Typ – identifikátor delegátu.

Jméno – jméno (identifikátor) události. Zpravidla odpovídá identifikátoru *typ* bez přípony `EventHandler`.

Druhá varianta události představuje deklaraci události jako vlastnosti – viz dále.

Klíčové slovo `event` zamezí vyvolání události mimo třídu, v níž je událost deklarována. Mimo třídu události lze používat pouze operátory += a −= pro registraci handleru a zrušení jeho registrace.

Událost je tedy vícenásobný delegát, který nelze zavolat mimo jeho třídu.

Událost by měla vyvolávat chráněná virtuální metoda třídy, v níž je událost deklarována. Tato metoda má mít název ve tvaru *OnJmeno,* kde *Jmeno* je jméno události, a má mít jeden parametr odpovídající druhému parametru události, tj. typu `EventArgs` nebo typu, který je jeho potomkem.

První parametr vyvolané události by neměl mít hodnotu `null`, kromě statické události. Druhý parametr by neměl být nikdy `null`. Třída `EventArgs` nabízí statickou datovou složku jen pro čtení `Empty`, která by se měla použít místo vytvoření nové instance této třídy.