PREPROCESOR – POKRAČOVÁNÍ

Chybová hlášení

V C# podobně jako v C++ existuje direktiva `#error`, která způsobí vypsání chybového hlášení překladačem a zastavení překladu. jazyk C# navíc nabízí direktivu `#warning`, která způsobí vypsání varovného hlášení překladačem, ale překlad se nezastaví.

Syntaxe:

direktiva_error:

#error *zpráva*

direktiva_warning:

#warning *zpráva*

Zpráva – znakový řetězec bez uvozovek. Pokud se uzavře do uvozovek, uvozovky budou součástí *zprávy*.

Obě direktivy lze použít pouze v zdrojovém textu direktivy `#if`.

Oblast zdrojového kódu

K vymezení oblasti ve zdrojovém souboru slouží direktivy `#region` a `#endregion`, které v C++ nemají obdobu.

Syntaxe:

direktiva_region:

#region $zpráva_{nep}$

direktiva_endregion:

#endregion $zpráva_{nep}$

Zpráva – znakový řetězec bez uvozovek.

Tyto direktivy nijak nemění význam programu, slouží pouze k označení oblasti ve zdrojovém kódu. Direktiva `#region` označuje začátek takové oblasti a `#endregion` její konec.

Např. Visual Studio 2005 označuje dvojicí

```
#region Windows Form Designer generated code

#endregion
```

oblast, v níž ukládá automaticky generovaný kód, který nedoporučuje měnit.

Editor v prostředí Visual Studio navíc umožňuje takovéto oblasti „zabalit“ (skrýt) nebo „rozbalit“ (zobrazit).

Direktivy `#region` se mohou do sebe vnořovat, nesmějí se ale křížit. Nesmějí se také křížit s oblastmi podmíněného překladu, tj. direktiva `#if` musí ležet ve stejném regionu jako jí odpovídající direktiva `#endif`.

Řádkování

Direktiva `#line` umožňuje změnit počítání řádků a jméno souboru při překladu. Její tvar i význam je stejný jako v C++.

Syntaxe:

direktiva_line:

#line *celé_číslo jméno_souboru*$_{\text{nep}}$

Celé_číslo – udává číslo, které má překladač považovat za číslo aktuálního řádku.

Jméno_souboru – jméno souboru bez uvozovek, který má překladač považovat za jméno aktuálního souboru. Je nepovinné. Obrácené lomítko se nezdvojuje.

Direktivy pragma

Direktivy `#pragma` mají být podle normy C# používány k specifikaci kontextových informací pro překladač. Jejich význam se může lišit podle použitého překladače.

Překladač Visual Studio 2005 obsahuje dvě direktivy `#pragma`:

- `#pragma warning` – vypnutí/obnovení určitého varování
- `#pragma checksum` – generování kontrolního součtu pro daný soubor – podrobnosti viz nápověda.

Syntaxe direktivy `#pragma warning`:

direktiva_pragma_warning:

#pragma warning disable *seznam_varování*

#pragma warning restore *seznam_varování*

Seznam_varování – seznam čísel varování oddělených čárkou.

První verze vypne zadaná varování, druhá verze je obnoví.

Pokud např. nechcete popisovat XML komentářem jednotlivé výčtové konstanty veřejného výčtového typu v sestavení, v němž se provádí překlad XML komentářů, lze deklaraci výčtových konstant uvést mezi dvě direktivy, z nichž první vypne varování číslo 1591 a druhá jej obnoví:

```
/// <summary>
/// Základní barvy.
/// </summary>
public enum Barva
{
#pragma warning disable 1591
   červená, zelená, modrá
#pragma warning restore 1591
}
```

TŘÍDY

Hlavní rozdíly v pojetí tříd mezi C# a C++:

- V C# není vícenásobná a s tím související virtuální dědičnost.
- Všechny třídy v C# mají společného předka, třídu `object`.
- Lze vytvářet pouze dynamické instance, o jejichž rušení se stará automatická správa paměti.
- Třídy mohou sice obsahovat destruktor, ale jeho význam je podstatně menší než v C++.
- Třídy mohou kromě datových složek, metod, konstruktorů a destruktorů obsahovat také vlastnosti, události, přetížené operátory a vnořené typy.
- Třídy mohou implementovat rozhraní.
- U tříd lze zakázat dědění.

První informace

Deklarace třídy

Syntaxe:

deklarace třídy:

modifikátory$_{\text{nep}}$ **partial**$_{\text{nep}}$ **class** *identifikátor předek*$_{\text{nep}}$ *tělo* **;**$_{\text{nep}}$

Modifikátory – vyjadřují přístupová práva. Jejich možnosti a význam závisí na tom, zda jde o vnořenou třídu nebo o třídu deklarovanou v prostoru jmen. Přístupová práva pro typy deklarované v prostoru jmen – viz dříve kapitola „Prostory jmen“. Na rozdíl od C++ se uvádí přístupová práva pro každou složku zvlášť. *Modifikátory* mohou dále obsahovat specifikaci `abstract`, `sealed`, `new` nebo `static`.

Identifikátor – jméno právě deklarované třídy.

Předek – předek třídy a implementovaná rozhraní.

Tělo – skupina deklarací ve složených závorkách. Tělo může být i prázdné.

Je-li uveden modifikátor `partial`, jedná se o deklaraci částečné třídy.

Za deklarací třídy může, ale nemusí být středník. Zpravidla se nepíše.

Deklarace třídy se na rozdíl od C++ nemůže objevit v metodě (lokální typ), ale jen v prostoru jmen (včetně globálního) nebo uvnitř třídy nebo struktury (vnořený typ).

Přístupová práva

Jazyk C# nabízí pět stupňů přístupových práv pro složky třídy. Určují přístupnost složek v rámci třídy, v potomcích třídy, v sestavení této třídy a v ostatních sestaveních. V následující tabulce je uveden seznam přístupových práv a oblasti, pro jaké je složka třídy s daným přístupovým právem přístupná.

Stupeň	Třída	Sestavení	Potomci	Ostatní
`public`	x	x	x	x
`protected internal`	x	x	x	
`protected`	x		x	
`internal`	x	x		
`private`	x			

Popis přístupových práv:

- `private` (soukromé složky) a `protected` (chráněné složky) – význam stejný jako v C++.
- `public` (veřejné, veřejně přístupné složky) – složka je dostupná všem součástem aktuálního sestavení i všech dalších sestavení za podmínky, že je přístupná třída jako celek (má přístupové právo `public`).
- `internal` (interní, vnitřní složky) – složka je dostupná všem součástem sestavení, v němž je třída definována.
- `protected internal` – je sjednocením přístupového práva `internal` a `protected`. Složka je tedy přístupná jak v třídě, v níž je deklarována, tak ve všech jejích potomcích bez ohledu na sestavení, a kromě toho je přístupná všem součástem sestavení, v němž je definována třída obsahující tuto složku.

Není-li specifikátor přístupových práv u dané složky uveden, jedná se o složku soukromou (se specifikátorem `private`).

Datový typ nebo složka nesmí mít širší přístupová práva než kterýkoli z dalších typů uvedených v její deklaraci. To např. znamená, že potomek nesmí mít širší přístupová práva než předek, metoda nesmí mít širší přístupová práva než typ, který vrací apod. Následující deklarace je tedy nesprávná:

```
class A {} // je internal

public class B {
   public A f() { /* ... */ } // Chyba
}
```

Metoda `f()` veřejné třídy `B` vrací typ `A`, který je interní v tomto sestavení. Proto tato metoda nemůže být veřejná. Pokud by třída `B` byla interní v tomto sestavení, žádnou chybu by překladač neoznámil.

Datové složky

Deklarace datových složek se podobají deklaracím proměnných. Podobně jako deklarace proměnných mohou obsahovat inicializace. V deklaraci datové složky lze navíc použít ještě atributy a modifikátory `new` a `readonly`.

Podobně jako v C++ jsou v C# datové složky statické a nestatické.

Nestatické datové složky

Nestatické datové složky (angl. *instance fields*) jsou složky jednotlivých instancí, stejně jako v C++. Třída může obsahovat datovou složku svého vlastního typu. Např.

```
public class Prvek
{
   private Prvek další; // OK
   // ...
}
```

Jedná se vlastně o odkaz (ukazatel) na instanci třídy, což je legální i v C++.

Deklarace datové složky může obsahovat i inicializaci. Následující příklad deklarace datových složek je v C# správný:

```
public class A
{
   private int x = 10;
   private string s = "Text";
   // ...
}
```

Přiřazení hodnot takovýmto složkám proběhne při vytváření instance ještě před voláním konstruktoru.

Ve strukturách lze uvést inicializaci jen v deklaraci statických nebo konstantních datových složek (deklarovaných s modifikátorem `const`).

Datové složky lze také inicializovat v těle konstruktoru. Datové složky, které nejsou inicializovány v deklaraci, překladač před voláním konstruktoru inicializuje hodnotou 0, `false` nebo `null`.

Statické datové složky

Statické datové složky (angl. *static fields*) jsou, podobně jako v C++, společné pro všechny instance dané třídy. Existují od zavedení třídy do paměti nezávisle na tom, zda existuje instance dané třídy nebo ne. Deklarují se pomocí modifikátoru `static`.

Také statické datové složky lze inicializovat v deklaraci nebo v tzv. statickém konstruktoru. Tato inicializace proběhne v okamžiku zavedení třídy do paměti (tedy typicky těsně před jejím prvním použitím).

Statické datové složky, které nejsou inicializovány v deklaraci, překladač inicializuje hodnotou 0, `false` nebo `null`.

Statické datové složky se při použití mimo třídu kvalifikují jménem třídy.

Neměnné datové složky

Datové složky, které se nebudou měnit a budou stejné ve všech instancích, lze deklarovat jako konstantní pomocí modifikátoru `const`. Konstantní datová složka se musí inicializovat přímo v deklaraci, např.

```
class A
{
   const int x = 12;
   // ...
}
```

Výraz, sloužící k inicializaci konstantní složky, musí být konstantní, tj. musí být známý již v době překladu. Konstantní datová složka je automaticky statická. Modifikátor `static` nelze v její deklaraci použít.

Modifikátor `const` lze použít pro předdefinované hodnotové typy (základní datové typy), výčtové typy a typ `string`. Pro referenční typy (třídy apod.) lze modifikátor `const` použít pouze k deklaraci proměnné, která je inicializována hodnotou `null`.

S modifikátorem `readonly` lze deklarovat datovou složku, kterou stejně jako konstantní datovou složku nelze měnit, ale lze ji inicializovat nekonstantním výrazem v její deklaraci nebo v konstruktoru nebo na obou místech. Po provedení konstruktoru ji již nelze měnit. Může se jednat o statickou nebo nestatickou datovou složku.

Např.:

```
class A
{
   const int x = 10;
   readonly int y = 20;
   public A(int y)
   {
      this.y = y; // OK
      x = 20; // Chyba
   }
   public void Set(int y)
   {
      this.y = y; // Chyba
   }
}
```

Modifikátorem `readonly` u datové složky se ale nezabrání modifikaci jejích složek prostřednictvím jejích metod.

Příklad

```
struct Struktura
{
   public int x;
   public void SetX(int x) { this.x = x; }
}
class Trida
{
   public readonly int[] pole = new int[] {1, 2};
   public readonly Struktura b;
}

class Program
{
   static void Main(string[] args)
   {
      Trida a = new Trida();
      a.pole[0] = 5; // OK
      //a.b.x = 20; // Chyba
      a.b.SetX(20); // OK
   }
}
```

Doporučení pro datové složky

Doporučuje se, aby nestatické datové složky nebyly deklarovány jako veřejné nebo chráněné. Přístup k nim z jiných tříd se má realizovat pomocí vlastností.

Pro předdefinovanou instanci třídy nebo struktury, která neobsahuje metody, jež umožňují modifikovat její datové složky (angl. *immutable type*), se má deklarovat veřejná statická datová složka s modifikátorem `readonly`. Např. třída `string` má veřejnou statickou „readonly“ datovou složku `Empty` typu `string`, obsahující prázdný řetězec `""`.

Metody

Hlavní odlišnosti od C++:

- Deklarace metody se musí zapsat vždy v těle třídy. Nelze v těle třídy uvést pouze prototyp a definici zapsat samostatně mimo tělo třídy.
- V C# neexistuje obdoba specifikace `inline` z C++.
- Parametry lze předávat podobně jako v C++ hodnotou nebo odkazem. Vedle toho zná C# také tzv. výstupní parametry.
- Metody v C# lze přetěžovat i na základě rozdílu ve způsobu předávání parametru.
- Nelze předepsat implicitní hodnoty parametrů.

Syntaxe deklarace metody:

deklarace_metody:

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

Modifikátory – modifikátory přístupových práv a podle okolností modifikátory vyjadřující, zda jde o statickou, virtuální, abstraktní, zapečetěnou, předefinovanou, zastiňující metodu nebo externí metodu z klasické dynamické knihovny. Dále lze zde uvést klíčové slovo `unsafe`, vyjadřující, že jde o metodu, jejíž tělo tvoří nezabezpečený kód.

Typ – typ vrácené hodnoty. Může být i `void`.

Identifikátor – identifikátor deklarované metody. Protože C# umožňuje přetěžování metod, nemusí se lišit od identifikátorů ostatních metod, pokud se liší signatura (pořadí a typ parametrů a způsob jejich předávání).

Seznam_parametrů – seznam formálních parametrů metody. Nemá-li metoda žádné parametry, její deklarace obsahuje prázdné kulaté závorky.

Tělo – zpravidla blok příkazů. U abstraktních a externích metod je tělo tvořeno středníkem.

Nestatické metody

Nestatické metody (angl. *instance methods*) implementují operace s jednotlivými instancemi, a proto se volají pro konkrétní instanci. Jméno nestatické metody volané mimo její třídu se kvalifikuje jménem instance její třídy.

Nestatické metody jsou metody, v jejichž deklaraci není uveden modifikátor `static`.

V těle metody je k dispozici odkaz na aktuální instanci vyjádřený klíčovým slovem `this`, stejně jako v C++. Odkazem `this` lze kvalifikovat pouze nestatické složky třídy, např.:

```
class TridaA
{
   int a;

   TridaA(int a)
   {
      this.a = a;
   }
}
```

Statické metody

Statická metoda se deklaruje s modifikátorem `static`.

Statické metody implementují operace s třídou jako celkem. Lze je volat i v době, kdy ještě (nebo už) neexistuje žádná instance třídy. Při volání statické metody mimo její třídu se kvalifikuje jménem její třídy.

V těle statické metody není k dispozici odkaz `this` a nelze pracovat s nestatickými složkami třídy bez kvalifikace instancí této třídy.

Statickou metodou je např. metoda `Main`, která se volá při spuštění konzolové aplikace.

Příklad

```
class Program
{
   void Run() // nestatická metoda
   {
      Console.WriteLine("Zavolána nestatická metoda Run");
   }
   static void Main(string[] args) // statická metoda
   {
      Program program = new Program();
      program.Run();
   }
}
```

Předávání parametrů

V C# lze předávat parametry metody hodnotou nebo odkazem a používat výstupní parametry. Vedle toho lze deklarovat metodu s proměnným počtem parametrů, což je analogie výpustky v C++.

Parametry předávané hodnotou

Syntaxe:

deklarace parametru předávaného hodnotou:

typ identifikátor

Typ – typ parametru.

Identifikátor – jméno parametru.

Význam parametru předávaného hodnotou je stejný jako v C++. V těle metody se vytvoří lokální proměnná, do níž se uloží hodnota skutečného parametru. V případě hodnotových parametrů proto operace s formálním parametrem v těle metody nezmění hodnotu skutečného parametru.

V případě referenčních typů se ovšem hodnotou předává odkaz na instanci, a proto operace s instancí prostřednictvím formálního parametru může změnit datové složky odpovídající instance.

Skutečným parametrem může být jakýkoli výraz, který lze implicitní konverzí převést na *typ* parametru.

Příklad

```
class TridaA
{
   public int x, y;
   public TridaA(int x, int y) { this.x = x; this.y = y; }
}
struct StrukturaB
{
   public int x, y;
   public StrukturaB(int x, int y) { this.x = x; this.y = y; }
}
```

```
class Program
{
   static void f(TridaA a1, TridaA a2, StrukturaB b)
   {
      a1.x = b.x; // složka x se ve skutečném parametru změní
      a2 = a1; // skutečný parametr se nezmění
      b.y = a1.y; // skutečný parametr se nezmění
   }
   static void Main(string[] args)
   {
      TridaA a1 = new TridaA(10, 20), a2 = new TridaA(30, 40);
      StrukturaB b = new StrukturaB(50, 60);
      f(a1, a2, b);
      // změna pouze u složky a1.x = 50
   }
}
```

Parametry předávané odkazem

Význam předávání parametru odkazem je stejný jako v C++, syntaxe je ovšem jiná – před typ se zapisuje modifikátor `ref`.

Syntaxe:

deklarace parametru předávaného odkazem:

ref *typ identifikátor*

Formální parametr v metodě představuje vlastně jen jiné jméno pro skutečný parametr, takže veškeré operace prováděné s formálním parametrem se ihned projeví v hodnotě skutečného parametru.

Při volání metody se před skutečný parametr musí zapsat modifikátor `ref`.

Skutečným parametrem musí být proměnná stejného typu, jaký je uveden v deklaraci formálního parametru nebo typu, který je potomkem typu formálního parametru.

Příklad

```
static void Výměna(ref int a, ref int b)
{
   int c = a;
   a = b;
   b = c;
}
static void Main(string[] args)
{
   int x = 10, y = 20;
   Výměna(ref x, ref y);
   // výměna hodnoty x a y
}
```

Příklad

Příklad využívá třídu a strukturu z předchozího příkladu.

```
class Program
{
   static void g(ref TridaA a1, ref TridaA a2, ref StrukturaB b)
   {
      a1.x = b.x;
      a2 = a1; // a2 se bude odkazovat na a1
      b.y = a1.y;
   }
   static void Main(string[] args)
   {
      TridaA a1 = new TridaA(10, 20), a2 = new TridaA(30, 40);
      StrukturaB b = new StrukturaB(50, 60);
      g(ref a1, ref a2, ref b);
      // všechny instance mají složky x = 50, y = 20
   }
}
```

Výstupní parametry

Výstupní parametry se liší od parametrů předávaných hodnotou tím, že jako skutečný výstupní parametr lze předat proměnnou, která nebyla inicializována. V metodě se musí výstupní parametr inicializovat, i když byl skutečný parametr před voláním této metody inicializován.

Před formálním i skutečným parametrem se musí zapsat modifikátor `out`.

Syntaxe:

deklarace výstupního parametru:

out *typ identifikátor*

Příklad

```
class Program
{
   static void f(int x, out int y)
   {
      // Console.WriteLine("y = " + y); // Chyba - y není inicializována
      y = x;
      Console.WriteLine("y = " + y); // OK
   }
   static void Main(string[] args)
   {
      int x = 20, y;
      f(x, out y); // proměnná x musí být inicializována
   }
}
```

Neznámý počet parametrů

Jestliže předem není známý počet a typ parametrů metody, lze v C# použít modifikátor `params`. Tento modifikátor lze použít u posledního z formálních parametrů metody. V normě C# se tento parametr nazývá *parameter array*.

Syntaxe:

specifikace proměnného počtu parametrů:

params *typ_pole identifikátor*

Typ pole – typ jednorozměrného pole. Nelze v něm použít modifikátory `ref` a `out`.

Skutečným parametrem může být:

- Pole odpovídajícího typu – parametr s modifikátorem `params` se chová stejně jako parametr předávaný hodnotou.
- Seznam výrazů oddělených čárkou, jejichž hodnoty lze implicitními konverzemi převést na typ prvků pole formálního parametru.

V těle metody se zachází s formálním parametrem s modifikátorem `params` stejně jako s polem odpovídajícího typu.

Příklad

```
class Matematika
{
   public static int Maximum(params int[] data)
   {
      if (data.Length == 0) return 0;
      int max = data[0];
      foreach (int a in data) {
         if (a > max) max = a;
      }
      return max;
   }
}
class Program
{
   static void Main(string[] args)
   {
      Console.WriteLine("Maximum = {0}", Matematika.Maximum(5, 88, -3));
      int[] pole = { 9, 8, 7 };
      int max = Matematika.Maximum(pole);
      Console.WriteLine("Maximum = {0}", max);
   }
}
```

Přetěžování metod

Jazyk C# umožňuje deklarovat v jedné třídě několik metod se stejným identifikátorem, pokud se liší v počtu, typu, pořadí nebo způsobu předávání parametrů. To znamená, že lze v jedné třídě deklarovat metody

```
void f() {}
void f(int x) {}
```

ale také metody

```
void f(char x) {}
void f(ref char x) {}
```

nebo

```
void f(char x) {}
void f(out char x) {}
```

Nelze však deklarovat metody lišící se pouze v modifikátorech `ref` a `out`:

```
void g(ref int x) {}
void g(out int x) { ... } // Chyba – existuje metoda g(ref int x)
```

Přetížené metody nelze rozlišit pouze podle typu návratové hodnoty nebo podle modifikátoru `params`.

Pokud skutečné parametry neodpovídají žádné z přetížených metod, použijí se pravidla pro nalezení „nejlepší shody“. To znamená, že se vybere metoda se stejným počtem parametrů, pro níž platí, že skutečné parametry lze v nějakém smyslu nejsnáze převést implicitními konverzemi na typy formálních parametrů. Přesné znění těchto pravidel – viz norma C#.

Doporučení pro metody

Doporučuje se, aby výstupní parametry byly vždy uvedeny jako poslední v seznamu parametrů metody kromě parametru s modifikátorem `params`, který musí být vždy na konci.

Přetížené metody by měly mít stejně pojmenované parametry stejného typu ve stejném pořadí. Výjimku tvoří parametr s modifikátorem `params` a výstupní parametry, které mají být uvedeny jako poslední.

Pokud mají být přetížené metody virtuální, má být deklarována jako virtuální pouze jedna verze přetížené metody, která má největší počet parametrů. Ostatní verze metody, by měly volat tuto verzi. Např.

```
public void Write(string message, FileStream stream)
{
   this.Write(message, stream, false);
}
public virtual void Write(string message, FileStream stream,
                          bool closeStream)
{
   // vlastní činnost
}
```

Pokud některý parametr metody je volitelný, měly by se vytvořit dvě přetížené verze této metody, jedna s tímto parametrem a druhá bez tohoto parametru. Parametr referenčního typu, pro který je povolena hodnota `null` by se neměl chápat jako volitelný parametr, ale jako parametr, pro nějž se má použít nějaká implicitní hodnota.