KAPITOLA 3

Přehled syntaxe jazyka C#

Tato kapitola je úvodem do syntaxe jazyka C#. Předpokládá se, že již máte nějaké zkušenosti s jazykem C++ nebo Java, protože syntaxe jazyka C# je podobná. To není žádná náhoda. Návrháři jazyka C# zamýšleli rozšířit znalosti programátorů pracujících v jazycích Java a C++, což jsou bezesporu jazyky, které v objektově orientovaném vývoji softwaru dominují.

Určitě si všimnete drobných rozdílů, které jsou pro jazyk C# specifické. Ale pokud znáte C++ nebo Javu, budete se cítit v syntaxi jazyka C# hned jako doma.

C# je silně typový jazyk

Podobně jako C++ a Java je i jazyk C# silně typový, což znamená, že každá proměnná a instance objektu v systému je přesně definovaného typu. To umožňuje překladači kontrolovat, zda jsou operace, které se pokoušíte nad proměnnými a instancemi objektů provádět, platné. Předpokládejme například, že máte metodu, která počítá průměr dvou celých čísel a vrací výsledek. Tuto metodu můžete v jazyce C# deklarovat následujícím způsobem:

```
double ComputeAvg( int param1, int param2 )
{
   return (param1 + param2) / 2.0;
}
```

Tímto říkáte překladači, že metoda přijímá dvě čísla typu integer a vrací výsledek typu double. Když se překladač pokusí přeložit kód, ve kterém metodě nechtěně předáte instanci typu `Apple`, postěžuje si a překlad přeruší. Předpokládejme, že jste metodu napsali trochu jinak:

```
object ComputeAvg( object param1, object param2 )
{
   return ((int) param1 + (int) param2) / 2.0;
}
```

Druhá verze metody `ComputeAvg` je stále platná, ale nekompromisně jste z ní odstranili informaci o typu. Všechny typy v C# jsou implicitně odvozeny od třídy `System.Object`. Klíčové slovo `object` je v jazyce C# alias této třídy. Je tedy úplně v pořádku deklarovat tyto parametry jako `object`. Typ `object` však není numerický. Aby bylo možné výpočet provést, musíte nejprve objekty *přetypovat* na typ integer. Poté již můžete vrátit výsledek jako instanci typu `object`. Ačkoliv vám může tato verze metody připadat flexibilnější, jedná se o časovanou bombu. Co kdyby se nějaký kód vaší aplikace pokusil předat metodě `ComputeAvg` instanci typu `Apple`? Překladač si nebude stěžovat, protože typ `Apple` je odvozen od třídy `System.Object`, stejně jako každá jiná třída. Nemilé překvapení vás však bude čekat za běhu programu, když vaše aplikace vyvolá výjimku oznamující, že nemůže konvertovat instanci třídy `Apple` na typ integer. Metoda selže, a pokud nemáte pro výjimku příslušný ošetřující kód, může ukončit celou aplikaci. Jistě byste nechtěli, aby k tomu došlo při provádění vašeho zdrojového kódu na nějakém komerčním serveru.

Vždy je nejlepší nacházet chyby v programu při překladu, a ne za běhu. To je ponaučení, které vyplývá z tohoto příběhu. Kdybyste použili první verzi metody `ComputeAvg`, překladač by vám řekl, jak nemístné bylo pokusit se předat metodě instanci typu `Apple`. To je mnohem lepší, než když vám totéž řekne rozzuřený zákazník, jehož komerční server si právě na chvíli zdřímnul. Překladač je váš kamarád, takže si jej pusťte k tělu a předávejte mu tolik informací, kolik budete moci, abyste mohli přesně prosazovat své záměry.

Výrazy

Syntaxe výrazů je v C# téměř stejná jako v C++ a v Javě. Při jejich vytváření je důležité myslet hlavně na pořadí, ve kterém se vyhodnocují operátory. Výrazy se v jazyce C# budují z operandů, což jsou obvykle proměnné nebo typy ve vaší aplikaci, a z operátorů. Mnoho operátorů lze také přetěžovat. Přetěžování operátorů se věnuji v 6. kapitole. Tabulka 3.1 obsahuje přehled priorit operátorů. Skupiny operátorů jsou v tabulce řazeny podle priority odshora dolů a operátory v jednotlivých skupinách mají stejnou prioritu.

Tabulka 3.1 Priorita operátorů v jazyce C#

Skupina operátorů	Operátory ve skupině	Popis
Primární	`x.m`	Přístup ke členům
Primární	`x(...)`	Volání metody
Primární	`x[...]`	Přístup k elementům
Primární	`x++, x--`	Post-inkrementace a post-dekrementace
Primární	`new T(...), new T[...]`	Vytváření

Skupina operátorů	Operátory ve skupině	Popis
Vytváření objektů a polí	`typeof(T)`	Vrátí objekt typu `System.Type` pro T
Vytváření objektů a polí	`checked(x)`, `unchecked(x)`	Vyhodnotí výraz při současné kontrole přetečení
`default(T)`		Vytvoří výchozí hodnotu typu T*
`delegate {...}`		Anonymní funkce nebo metoda
Unární	`+x`, `-x`	Identita a opačné číslo
Unární	`!x`	Logická negace
Unární	`~x`	Bitová negace
Unární	`++x`, `--x`	Preinkrementace a predekrementace
Unární	`(T) x`	Přetypování
Násobení a dělení	`x*y`, `x/y`, `x%y`	Násobení, dělení a zbytek po dělení
Sčítání a odčítání	`x+y`, `x-y`	Sčítání a odčítání
Bitový posun	`x<<y`, `x>>y`	Bitový posun vlevo, bitový posun vpravo
Relační operátory a testování typu	`x<y`, `x>y`, `x<=y`, `x>=y`	Menší než, větší než, menší než nebo rovno, větší než nebo rovno
Relační operátory a testování typu	`x is T`	True, pokud lze x konvertovat na T; false v opačném případě
Relační operátory a testování typu	`x as T`	Vrací x konvertované na T, nebo null, když konverze není možná
Rovnost	`x == y`, `x != y`	Je rovno a není rovno
Logická konjunkce (AND)	`x & y`	Bitový operátor AND pro celočíselné hodnoty, Booleovský logický operátor AND
Logická nonekvivalence (XOR)	`x ^ y`	Bitový operátor XOR pro celočíselné hodnoty, Booleovský operátor XOR
Logický operátor disjunkce (OR)	`x | y`	Bitový operátor OR pro celočíselné hodnoty, Booleovský logický operátor OR
Podmínkový operátor konjunkce (AND)	`x && y`	Vyhodnotí y, pouze pokud x je true
Podmínkový operátor disjunkce (OR)	`x || y`	Vyhodnotí y, pouze pokud x je false
Koalescence	`x ?? y`	Jestliže x není null, je výsledkem x; jinak je výsledkem y

* O výrazech s výchozími hodnotami se dozvíte více v 11. kapitole v sekci „Výraz pro výchozí hodnotu“.

Skupina operátorů	Operátory ve skupině	Popis
Podmíněný výraz	`x ? y : z`	Pokud je x true, vyhodnotí y; pokud ne, vyhodnotí z
Přiřazení nebo anonymní fuknce	`x = y`	Jednoduché přiřazení
Přiřazení nebo anonymní fuknce	`x op= y`	Složené přiřazení; může být kterékoliv z následujících: *=, /=, %=, +=, -=, <<=, >>=, &=, ^= nebo |=
`(T x) => y`	Lambda-výraz (anonymní funkce, metoda)	

Poznámka

Tyto operátory mají v různém kontextu různé významy. Nezávisle na tom se však jejich priorita nikdy nemění. Pokud například jako operandy použijete řetězce, může operátor + tyto řetězce spojovat. Když použijete při definici vašich vlastních typů přetěžování operátorů, můžete dosáhnout jakéhokoliv jejich významu, který je pro dané typy smysluplný. Nikdy však nemůžete změnit jejich prioritu jiným způsobem, než organizací operací do skupin prostřednictvím kulatých závorek.

Ve výrazech určuje pořadí vyhodnocení operátorů se stejnou prioritou jejich asociativita. Všechny binární operátory, kromě operátorů přiřazení, jsou asociativní zleva doprava, zatímco operátory pro přiřazování a podmínkový (ternární) operátor jsou asociativní zprava doleva.

Příkazy a výrazy

Podoba příkazů je v jazyce C# stejná jako v C++ a v Javě. Jednoduché výrazy ukončuje středník. Avšak bloky kódu, jako například ty, které jsou uzavřené v závorkách příkazů `if` nebo `while`, středníkem ukončovat nemusíte.

Většinu příkazů, které jsou k dispozici v C++ a v Javě, lze použít také v jazyce C#, včetně příkazů pro deklaraci proměnných, podmínkových příkazů, příkazů řídících tok programu, příkazů `try`/`catch` atd. Jazyk C# má však i některé příkazy (podobně jako Java), které nejsou v C++ k dispozici. C# například obsahuje příkaz `try`/`finally`, který podrobně rozebírám v 7. kapitole. V kapitole 12. vám pak ukážu příkaz `lock`, který synchronizuje přístup k blokům zdrojového kódu. C# také přetěžuje klíčové slovo `using`, takže jej můžete použít buď jako direktivu, nebo jako příkaz. Příkaz `using` můžete použít společně s návrhovým vzorem „Disposable“, který popisuji v kapitolách 4 a 13. Za zmínku stojí také příkaz `foreach`, díky kterému můžete snáze procházet kolekcemi. Podrobněji se mu budu věnovat v 13. kapitole, ve které se budu věnovat kolekcím.

Typy a proměnné

Každá entita v programu v C# je objekt, který je buď v zásobníku, nebo ve spravované haldě. Každá metoda se definuje v deklaraci třídy (`class`) nebo struktury (`struct`). V C# neexistuje nic takového, jako jsou volné funkce definované mimo deklarace třídy nebo struktury jako v C++. Dokonce i vesta-

věné typy, jako jsou typy `int`, `long`, `double` atd., mají metody, které jsou s nimi implicitně sdruženy. To znamená, že v jazyce C# je naprosto v pořádku napsat obdobný příkaz:

```
System.Console.WriteLine( 42.ToString() );
```

Příkaz, ve kterém se volá přímo metoda hodnoty 42, vám bude připadat podivný, pokud jste zvyklí na C++ nebo na Javu. Ilustruje však, jak je v jazyce C# považováno za objekt úplně všechno, včetně těch nejzákladnějších typů. Ve skutečnosti se klíčová slova pro vestavěné typy v jazyce C# zobrazují přímo na typy ve jmenném prostoru `System`, které reprezentují. Můžete se dokonce rozhodnout, že nebudete používat vestavěné typy a že budete explicitně používat typy ze jmenného prostoru `System`, na které se vestavěné typy zobrazují (to se však nepovažuje za stylově čisté). Tabulka 3.2 obsahuje popis vestavěných typů - jejich velikost a odpovídající typy ze jmenného prostoru `System`.

Tabulka 3.2 Vestavěné typy jazyka C#

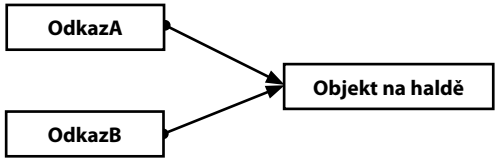
Typ v C#	Velikost v bitech	Systémový typ	Kompatibilní s CLS
`sbyte`	8	`System.SByte`	Ne
`short`	16	`System.Int16`	Ano
`int`	32	`System.Int32`	Ano
`long`	64	`System.Int64`	Ano
`byte`	8	`System.Byte`	Ano
`ushort`	16	`System.UInt16`	Ne
`uint`	32	`System.UInt32`	Ne
`ulong`	64	`System.UInt64`	Ne
`char`	16	`System.Char`	Ano
`bool`	8	`System.Boolean`	Ano
`float`	32	`System.Single`	Ano
`double`	64	`System.Double`	Ano
`decimal`	128	`System.Decimal`	Ano
`string`	N/A	`System.String`	Ano
`object`	N/A	`System.Object`	Ano
`dynamic`	N/A	`System.Object`	Ano

Údaj v posledním sloupci tabulky značí, zda je daný typ kompatibilní se specifikací CLS (Common Language Specification). Specifikace CLS je definována jako součást standardu CLI usnadňující interoperabilitu mezi různými jazyky a je podmnožinou standardu Common Type System (CTS). Dokonce i když CLR podporuje rozsáhlou množinu vestavěných typů, ne všechny jazyky, které se překládají do spravovaného kódu, všechny tyto typy podporují. Všechny spravované jazyky však musí podporovat typy definované v CLS. Visual Basic například tradičně nikdy nepodporoval bezznaménkové typy. Návrháři CLI definovali CLS za účelem standardizace typů , aby usnadnili interoperabilitu mezi jednotlivými jazyky. Bude-li celá vaše aplikace založená na jazyku C# a nebude vytvářet žádné komponenty převzaté z jiných jazyků, nebudete se muset CLS přesně řídit. Ale pokud pracu-

jete na projektu, jehož komponenty se budují v různých jazycích, bude pro vás přizpůsobení se CLS mnohem důležitější.

Ve světě spravovaného zdrojového kódu CLR existují dva druhy typů:

- *Hodnotové typy*: Jsou v jazyce C# definovány prostřednictvím klíčového slova `struct`. Instance hodnotových typů jsou jediným druhem instancí, které mohou být v zásobníku. Pokud jsou však členy referenčních typů, nebo jsou zapouzdřeny do objektové reprezentace (boxed), o čemž budu mluvit později, nacházejí se ve spravované haldě. Hodnotové typy jsou podobné strukturám v jazyce C++ v tom smyslu, že pokud je předáváte jako parametry nějaké metodě, nebo pokud je přiřazujete jiným proměnným, kopíruje se ve výchozím nastavení jejich hodnota. Ačkoliv zabudované typy jazyka C# reprezentují stejné druhy hodnot jako primitivní typy v jazyce Java, neexistují v C# pro primitivní typy jazyka Java odpovídající protějšky.
- *Referenční typy*: Referenční typy se v jazyce C# často definují prostřednictvím klíčového slova `class`. Referenční se jim říká proto, že proměnné, jejichž prostřednictvím s nimi manipulujete, jsou ve skutečnosti odkazy na objekty ve spravované haldě. Proměnné referenčních typů jsou v CLR něco jako proměnné hodnotových typů, které mají přiřazený typ a obsahují ukazatel na objekt na haldě. Vy, jako programátoři, používáte při definování těch objektů, které se vytvářejí na haldě, klíčové slovo `class`. V tomto ohledu se C# a Java nijak neliší. Programátoři pracující v jazyce C++ si mohou proměnné referenčních typů představovat jako ukazatele, které nemusí dereferencovat, když chtějí přistupovat k objektům, na které ukazují. Někteří programátoři pracující v C++ si je proto představují jako chytré ukazatele. Když někdo mluví o *referenčních typech* v jazyce C#, má často na mysli objekty, které jsou na spravované haldě. Avšak při interakci s objekty na spravované haldě se používá *proměnná referenčního typu*. Proměnné referenčních typů jsou nějakého typu a obsahují ukazatel na objekt, na který odkazují. Typ proměnné referenčního typu může být například stejný, jako je třída objektu, na který ukazuje, jako základní typ tohoto objektu, nebo se může jednat o typ rozhraní, které objekt implementuje. Na instanci stejného objektu může přirozeně odkazovat v jednu chvíli několik proměnných referenčních typů. Hodnotové typy oproti tomu obsahují přímo svá data, a ne pouze odkaz na tato data. Na obrázku 3.1 je diagram znázorňující situaci, kdy na stejný objekt na spravované haldě odkazují v jednu chvíli dvě proměnné referenčních typů. V diagramu je zvýrazněné, jak je každé proměnné referenčního typu přiřazený nějaký typ.



Obrázek 3.1 Proměnné referenčních typů

Hodnotové typy

Hodnotové typy mohou být uloženy buď v zásobníku, nebo na haldě. Používají se běžně v případě, kdy potřebujete reprezentovat nějaká neměnitelná data, která mají obecně vzato malé paměťové nároky. Uživatelsky definované hodnotové typy můžete v jazyce C# definovat prostřednictvím klíčového slova `struct`.

Poznámka

Následující podsekce této kapitoly se věnuje také výčtovým typům, které se deklarují prostřednictvím klíčového slova `enum`. Také prostřednictvím klíčového slova `enum` je možno deklarovat hodnotový typ.

Uživatelsky definované hodnotové typy se chovají úplně stejně jako vestavěné hodnotové typy. Když vytvoříte za běhu programu nějakou hodnotu, která není členem referenčního typu, vytvoří se obvykle v zásobníku. Podívejte se na tento úryvek zdrojového kódu:

```
int theAnswer = 42;
System.Console.WriteLine( theAnswer.ToString() );
```

Nejenže se proměnná `theAnswer` vytvoří v zásobníku, ale předáte-li ji nějaké metodě, obdrží tato metoda její kopii. Hodnotové typy se obvykle používají ve spravovaných aplikacích pro reprezentaci lehkých shluků anebo kolekcí dat podobně, jako se někdy používají vestavěné typy a struktury v C++ a primitivní typy v Javě.

Hodnoty mohou být uloženy ve spravované haldě, ale ne samy o sobě. K tomu může dojít, pouze pokud má referenční typ složku hodnotového typu. Ačkoli je hodnotový typ v objektu uložen v haldě, je-li předán nějaké metodě, bude se chovat stejně jako hodnotový typ uložený v zásobníku; to znamená, že metoda ve výchozím nastavení obdrží jeho kopii. Pokud nebyla hodnota předána odkazem, jsou všechny provedené změny instance hodnoty pouze změnami lokální kopie. Uvedené mšlenky ilustruje následující zdrojový kód:

```
public struct Coordinate //toto je hodnotový typ
{
    public int x;
    public int y;
}

public class EntryPoint //toto je referenční typ
{
    public static void AttemptToModifyCoord( Coordinate coord ) {
        coord.x = 1;
        coord.y = 3;
    }

    public static void ModifyCoord( ref Coordinate coord ) {
        coord.x = 10;
        coord.y = 10;
    }

    static void Main() {
        Coordinate location;
        location.x = 50;
        location.y = 50;

        AttemptToModifyCoord( location );
        System.Console.WriteLine( "( {0}, {1} )", location.x, location.y );

        ModifyCoord( ref location );
        System.Console.WriteLine( "( {0}, {1} )", location.x, location.y );
    }
}
```

Volání metody `AttemptToModifyCoord` v metodě `Main` ve skutečnosti hodnotu proměnné `location` nijak nemění, protože metoda `AttemptToModifyCoord` modifikuje pouze lokální kopii této hodnoty, která se vytvořila při jejím zavolání. Metodě `ModifyCoord` se naproti tomu hodnota proměnné `location` předává odkazem. To znamená, že jakékoliv změny této hodnoty provedené v metodě `ModifyCoord` se provedou přímo s předanou proměnnou. Princip je podobný, jako když v C++ předáte metodě hodnotu prostřednictvím ukazatele. Výstup tohoto příkladu vypadá takto:

```
( 50, 50 )
( 10, 10 )
```

VÝČTOVÉ TYPY

Výčtové typy (výčty, enumerace) jsou v C# podobné jako v C++ a syntaxe jejich definic je téměř identická. Ve chvíli, kdy je chcete použít, však musíte hodnoty enumerace plně kvalifikovat prostřednictvím názvu jejího typu. Všechny enumerace jsou založeny na celočíselném typu, kterým je, pokud nespecifikujete něco jiného, `int`. Proto jsou i výčty hodnotovými typy.

Poznámka

Typ, na kterém je enumerace založena, musí být jedním z následujících integrálních typů: `byte`, `sbyte`, `short`, `ushort`, `int`, `uint`, `long` nebo `ulong`.

Hodnota každé z konstant, definovaných ve výčtovém typu, musí být v rozsahu typu, na kterém je výčet založen. Pokud pro konstantu výčtového typu nespecifikujete žádnou hodnotu, bude mít výchozí hodnotu 0 (pokud se jedná o první konstantu v e výčtu), nebo 1 + hodnota předchozí konstanty. V následujícím příkladu je výčet založen na typu `long`:

```
public enum Color : long
{
   Red,
   Green = 50,
   Blue
}
```

Kdybych v tomto příkladu vynechali dvojtečku a klíčové slovo `long` za identifikátorem typu `Color`, byl by tento výčet typu `int`. Pamatujte, že hodnota konstanty `Red` bude 0, `Green` 50 a `Blue` 51.

Kdybyste chtěli tento výčtový typ použít, udělali byste to následujícím způsobem:

```
static void Main() {
   Color color = Color.Red;
   System.Console.WriteLine( "Barva je {0}", color.ToString() );
}
```

Když tento zdrojový kód přeložíte, bude ve výstupu název výčtu, nikoli číslovka 0. To má na svědomí implementace metody `ToString` typu `System.Enum`.

VÝČTY JAKO BITOVÉ PŘÍZNAKY

Konstanty výčtových typů můžete často použít za účelem reprezentace bitových příznaků (flags). Chcete-li tuto funkci výčtového typu explicitně deklarovat, můžete k němu připojit atribut `System.FlagsAttribute`. Atribut se uloží do metadat. Tento atribut použijte, když chcete určit, že budou členy výčtového typu bodou sloužit jako bitové příznaky.

Myslete na to, že atribut `System.FlagsAttribute` nemění chování hodnot definovaných v deklaraci tohoto typu. Za běhu programu však mohou některé jeho komponenty používat metadata vygenerovaná tímto atributem a zpracovávat hodnoty definované v tomto typu jiným způsobem. Jedná se o výborný příklad efektivního použití metadat způsobem typickým pro aspektově orientované programování (AOP).

Poznámka

AOP, kterému se někdy také říká aspektově orientovaný vývoj softwaru (AOSD), je koncepce, kterou původně vyvinul Gregor Kiczales a jeho tým ze společnosti Xerox PARC. Objektově orientované metodologie obecně vzato odvádí skvělou práci při rozdělení navrhované funkcionality nebo úkolů programu do kompaktních jednotek. Avšak některé úkoly, kterým se říká *roztroušené záležitosti* (*cross-cutting concerns*), nelze při standardním objektově orientovaném (OO) navrhování snadno do OO modelu aplikace zařadit. Představte si například, že potřebujete mít k dispozici výpis zahájení a ukončení provádění nejrůznějších metod. Modifikovat za tímto účelem zdrojový kód všech metod, které potřebujete do výpisu zařadit, by bylo velmi náročné. Mnohem jednodušší by bylo, kdybyste mohli jednoduše přiřadit metodě nějakou vlastnost, nebo v tomto případě atribut, takže by běhové prostředí metodu zařadilo do výpisu při každém jejím volání. Potom byste nemuseli upravovat zdrojový kód metody a požadavek zařazení jejího volání do výpisu by se vyřešil mimo okruh její implementace. Jedním z prvních rozšířených příkladů AOP byl transakční server Microsoft Transaction Server (MTS).

Díky metadům a skutečnosti, že můžete typům, metodám, vlastnostem atd. přiřazovat libovolné atributy, pak máte možnost AOP využít při navrhování vašich programů.

Následující výpis zdrojového kódu je příkladem výčtu obsahujícího bitové příznaky:

```
[Flags]
public enum AccessFlags

{
   NoAccess = 0x0,
   ReadAccess = 0x1,
   WriteAccess = 0x2,
   ExecuteAccess = 0x4
}
```

A tento výpis zdrojového kódu představuje příklad použití výčtu obsahujícího bitové příznaky:

```
static void Main() {
   AccessFlags access = AccessFlags.ReadAccess |
      AccessFlags.WriteAccess;

   System.Console.WriteLine( "Přístup je {0}", access );
}
```

Pokud tento příklad přeložíte a spustíte, uvidíte, že metoda `Enum.ToString` implicitně volaná metodou `WriteLine` vrací výpis všech příznaků z této hodnoty oddělených čárkami.

Referenční typy

Automatická správa paměti (garbage collector, GC) v CLR spravuje veškeré záležitosti týkající se umístění objektů a může objekty kdykoliv přesunout. Když to udělá, CLR se ujistí, zda proběhla

aktualizace všech proměnných, které na ně odkazují. Za normálních okolností se o přesné umístění objektů v haldě nestaráte a také se nemusíte starat o to, zda došlo k jejich přesunutí, či nikoliv. Existují však vzácné případy, například při propojení s dynamickými knihovnami, ve kterých byste mohli potřebovat získat přímý paměťový ukazatel na objekt v haldě. Můžete to udělat s pomocí *nebezpečných* (*unsafe*) neboli *nespravovaných* (*unmanaged*) programovacích technik, ale jejich popis je mimo oblast zájmu této knihy.

Poznámka

Podle konvencí pojem *objekt* označuje instanci referenčního typu, zatímco pojem *hodnota* označuje instanci hodnotového typu. Všechny typy (referenční nebo hodnotové) jsou však zároveň odvozeny od typu `object`.

Proměnné referenčních typů se inicializují buď prostřednictvím operátoru `new`, jehož prostřednictvím lze vytvořit objekt ve spravované haldě, nebo se inicializují přiřazením z jiné proměnné kompatibilního typu. V následujícím výpisu zdrojového kódu se inicializují dvě proměnné ukazující na stejný objekt:

```
object o1 = new object();
object o2 = o1;
```

Podobně jako běhové prostředí jazyka Java i CLR spravuje všechny odkazy na objekty v haldě. V C++ musíte objekty založené v haldě v určitou opatrně vybranou chvíli vždy odstranit. Ale ve spravovaném prostředí CLR to za vás udělá GC. Díky tomu se nemusíte starat o mazání objektů z paměti a minimalizují se její úniky. GC umí kdykoliv zjistit, kolik odkazů na konkrétní objekt v haldě zrovna existuje. Pokud zjistí, že žádný, může objekt z haldy odstranit. Detailní popis problémů spojených s tímto procesem a faktorů, které jej ovlivňují, najdete v 13. kapitole. Ve výše uvedeném příkladu se inicializují dva odkazy na stejný objekt. První z nich inicializujete tak, že vytvoříte nový objekt typu `object`. Druhý odkaz, `o2`, pak inicializujete pomocí prvního odkazu, `o1`. GC nezačne mazat objekt z haldy, dokud nebudou oba odkazy úplně nedostupné. Kdyby metoda obsahující tento zdrojový kód vrátila kopii odkazu čemukoliv, co by ji zavolalo, GC by tento odkaz stále sledoval, dokonce i když by metoda již nebyla nadále dostupná.

Poznámka

Pro ty z vás, kteří přecházíte k C# z C++, je základní způsob nakládání s objekty v C++ přesným opakem způsobu, jakým se s objekty nakládá v C#. V C++ se objekty, pokud je nevytvoříte explicitně prostřednictvím operátoru `new`, který vrací ukazatel na objekt v nativní haldě, alokují v zásobníku. V C# však objekty referenčních typů v zásobníku vytvářet nemůžete; mohou být pouze v haldě. To znamená, že je to skoro stejné, jako kdybyste psali zdrojový kód v jazyce C++, ve kterém byste vytvářeli všechny objekty v haldě, aniž byste se museli starat o jejich explicitní mazání při úklidu paměti.

Výchozí inicializace proměnných

Ve výchozím nastavení produkuje překladač jazyka C# tzv. *bezpečný kód*. Jedním ze zájmů bezpečnosti je zajistit, aby program nepoužíval neinicializovanou paměť. Překladač vyžaduje, aby byla každá

proměnná nastavena na nějakou hodnotu předtím, než ji použijete, takže se hodí vědět, kolik různých typů proměnných se inicializuje.

Výchozí hodnota odkazů na objekty je null. Ve chvíli deklarace můžete volitelně přiřazovat odkazům výsledek volání operátoru `new`; jinak se jejich hodnoty nastaví na null. Když vytváříte nějaký objekt, běhové prostředí inicializuje jeho interní složky. Složky, které jsou odkazy na jiné objekty, se samozřejmě inicializují hodnotou null. Složky hodnotových typů se inicializují tak, že se všechny bity nastaví na 0. V zásadě si to můžete představit tak, že běhové prostředí nastaví nepodkladové paměti na 0. V případě odkazů na objekty to znamená jejich nastavení na null a v případě hodnotových typů na hodnotu 0 (nebo false v případě typu Boolean).

V případě hodnotových typů, které deklarujete v zásobníku, překladač paměť nulami neinicializuje automaticky, avšak ujistí se, že jste ji před použitím hodnoty inicializovali vy.

Poznámka

Výčty jsou ve skutečnosti hodnotové typy, a dokonce i když lze 0 implicitně konvertovat na jakýkoli výčtový typ, je dobrým zvykem vždy deklarovat člen výčtového typu, který má hodnotu 0, dokonce i když má název `InvalidValue` nebo `None` a nemá jinak žádný význam. Pokud deklarujete výčtový typ jako složku třídy, nastaví se tato složka instance této třídy při výchozí inicializaci na 0. Když deklarujete člen výčtového typu, který má hodnotu 0, umožníte všem uživatelům tohoto typu, aby se s tímto případem snadno vyrovnali, a dalo by se říci, že výsledkem bude také čitelnější zdrojový kód.

Implicitně typované lokální proměnné

Protože je jazyk C# přísně typový, musí být každé proměnné ve zdrojovém kódu explicitně přidělen nějaký typ. Počínaje verzí 4.0 bylo toto pravidlo trochu zmírněno díky zavedení typu `dynamic`, kterému se věnuji v 17. kapitole. Když CLR uloží obsah proměnné na nějaké místo v paměti, přidělí tomuto místu zároveň nějaký typ. Někdy však může být psaní deklarací proměnných v silně typových jazycích velmi únavné, zejména pokud je typ proměnné generický. Názvy složitých typů, jako jsou například typy výsledků z dotazů vytvořených prostřednictvím jazyka LINQ, které rozebírám v 16. kapitole, mohou být vyloženě neohrabané. Zde vstupují na scénu implicitně typované proměnné.

Když deklarujete lokální proměnnou prostřednictvím nového klíčového slova `var`, požádáte vlastně překladač, aby rezervoval místo v lokální paměti a přidělil mu dovozený typ. Při překladu již pak má překladač ve chvíli inicializace proměnné k dispozici dost údajů na to, aby vydedukoval skutečný typ proměnné, aniž byste jej museli explicitně specifikovat. Pojďte se tedy podívat, jak to vypadá v praxi.

Zde je příklad deklarace implicitně typované proměnné:

```
using System;
using System.Collections.Generic;

public class EntryPoint
{
    static void Main() {
        var myList = new List<int>();

        myList.Add( 1 );
```

```
        myList.Add( 2 );
        myList.Add( 3 );

        foreach( var i in myList ) {
            Console.WriteLine( i );
        }
    }
}
```

Nejprve byste si měli všimnout klíčových slov vytištěných tučným písmem, která znázorňují způsob použití nového klíčového slova `var`. V případě jeho prvního použití deklaruji proměnnou `myList` a žádám překladač, aby nastavil její typ podle typu hodnoty, který se jí přiřadí. Zde je třeba zdůraznit, že v deklaraci implicitně typované proměnné musí být inicializátor. Kdybyste se pokusili použít následující zdrojový kód, počastoval by vás překladač varováním CS0818 „Implicitly typed locals must be initialized" („Implicitně typované lokální proměnné musí být inicializovány"):

```
var newValue; // výsledkem bude error CS0818
```

Podobně, když se pokusíte deklarovat implicitně typovanou proměnnou jako složku třídy, dokonce i když bude mít inicializátor, bude výsledkem chyba CS0825, „The contextual keyword 'var' may only appear within a local variable declaration." („Kontextové klíčové slovo 'var' lze použít pouze v deklaraci lokální proměnné.")

Možná jste také zvyklí deklarovat více různých proměnných v jediném řádku tak, že je od sebe oddělíte čárkami, jako v následujícím příkladu:

```
int a = 2, b = 1;
int x, y = 4;
```

Tímto způsobem však v případě implicitně typovaných proměnných postupovat nemůžete. Když se o to pokusíte, vrátí překladač chybu CS0819 „ Implicitly typed locals cannot have multiple declarators" („Implicitně typované lokální proměnné nemohou mít více deklarátorů").

KOMPLIKACE S PŘIDÁVÁNÍM NOVÝCH KLÍČOVÝCH SLOV DO PROGRAMOVACÍHO JAZYKA

Přidávání nových vlastností do překladače, jako jsou implicitně typované proměnné, není tak snadné, jak se může na první pohled zdát. Je to proto, že pokaždé, když přidáte do jazyka nové klíčové slovo, musíte vzít v úvahu případné narušení existujících programů a problémy zpětné kompatibility. Představte si například, co by se stalo, kdybyste měli rozsáhlý program napsaný v C# 1.0 nebo C# 2.0, který by pracoval s typem `var` (mohlo by se jednat třeba o třídu), a chtěli byste přejít na C# 3.0 nebo pozdější verzi jazyka a přeložili byste svou aplikaci novým překladačem. V programu bezpochyby budou nějaké deklarace proměnných, které budou vytvářet instance vaší třídy `var`. Co má překladač udělat? To je otázka, na kterou je velmi obtížné odpovědět.

Když jsem tuto situaci sám testoval, překladač neudělal nic. Ale je to v pořádku? Mohl také vrátit varování, které by mohlo znít nějak takhle: „Deklarovaný typ má stejný název jako klíčové slovo `var`." Jak si však za chvíli ukážeme, není to ve skutečnosti nutné. Opravdu je zřejmě lepší, když překladač varování nevrátí. Kvalitní vývojové týmy používají parametr překladače `/WARNASERRORS+`. Použijete-li tento parametr, překlad se při výskytu chyby přeruší. Když překladač vrátí varování, vaše aplikace se při migraci na verzi C# 3.0 nebo pozdější nepřeloží a na vině bude Microsoft a jeho do očí bijící ignorování problémů se zpětnou kompatibilitou.

Podstatné je, že když definujete ve zdrojovém kódu typ s názvem `var`, jednoduše nemůžete implicitně typované proměnné používat v žádném zdrojovém kódu jazyka C#, ve kterém se jmenný prostor tohoto typu importuje do oblasti lokální dostupnosti. Když to uděláte, bude výsledkem obvykle výskyt chyby překladače CS0029, která v podstatě říká, že typ, který se pokoušíte přidělit implicitně typované proměnné, nelze explicitně konvertovat na váš vlastní typ `var`. Dobrý jazykolam! Toto chování uvidíte například při pokusu o překlad následujícího kódu:

```
using System;
using System.Collections.Generic;
public class var
{
}

public class EntryPoint
{
    static void Main() {
        var myList = new List<int>(); // Nepřeloží se! Chyba CS0029
    }
}
```

Vývojáři překladače obvykle tento problém berou velmi vážně a někdy dokonce nové klíčové slovo do jazyka raději nezařadí, pokud by mohlo jeho zařazení narušit existující programy. Vývojáři překladače jazyka C# však mají také ještě eso v rukávu. Budete-li se řídit doporučenými konvencemi pro tvorbu názvů typů v .NET a budete v nich používat velká první písmena, nikdy se do této situace nedostanete. S tímto problémem se také nikdy nesetkáte, jestliže používáte při vývoji svých aplikací nástroj Visual Studia pro analýzu zdrojového kódu FxCop (jeho samostatnou verzi). Tato pravidla a návyky jsou detailně popsána také v knize Krzysztofa Cwaliny a Brada Abramse *Framework DesignGuidelines: Conventions, Idioms, and Patterns for Reusable .NET Libraries* (Boston, MA: Addison-Wesley Professional, 2005). Pokud jste tak ještě neučinili, rozhodně vám doporučuji, abyste si tuto knihu přečetli.

Konverze typů

Často je nezbytné konvertovat instance jednoho typu na instance jiného typu. V některých případech překladač provádí tuto konverzi implicitně, kdykoliv se přiřazuje jednomu typu hodnota jiného typu, která při konverzi na typ, kterému se přiřazuje, neztratí nic ze své přesnosti a rozsahu. Pokud může dojít ke snížení přesnosti, je třeba provést konverzi explicitní. Pravidla konvertování referenčních typů jsou v jazyce C# stejná jako v C++.

Sémantika konverzí je pak v C# podobná jako v C++ a v Javě. Explicitní konverze se vyjadřuje prostřednictvím obecně známé syntaxe přetypování, která je ve všech těchto jazycích odvozená ze syntaxe jazyka C - tj. typ, na který se konvertuje, se umístí do závorek před cokoliv, co je třeba konvertovat:

```
int defaultValue = 12345678;
long value = defaultValue;
int smallerValue = (int) value;
```

V tomto zdrojovém kódu je nutné uvádět přetypování `(int)` explicitně, protože velikost typu `int` je menší než velikost typu `long`. Je proto možné, že se hodnota typu `long` nevejde do prostoru, který má k dispozici typ `int`. Přiřazení hodnoty `defaultValue` proměnné `value` provedení konverze nevyžaduje, protože `long` má k dispozici větší úložiště, než `int`. Pokud při konverzi dojde ke ztrátě rozsahu, je možné, že konverze vyvolá za běhu programu výjimku. Obecné pravidlo je pak takové, že implicitní konverze výjimku vyvolat nemohou, ale explicitní mohou.

Jazyk C# vám umožňuje definovat pro vaše vlastní uživatelsky definované typy vaše vlastní operátory pro jejich implicitní a explicitní konverzi na nejrůznější jiné typy. Tomuto tématu se podrobně věnuji v 6. kapitole. Požadavky na ošetření výjimek, které se vztahují na zabudované operátory pro konverzi, platí i pro uživatelsky definované operátory. Konkrétně lze opět říci, že v případě implicitních konverzí nemůže k výskytům výjimek docházet.

Konverze z referenčních typů a na referenční typy je stejná jako v Javě a jako konverze mezi ukazateli v C++. Například odkaz na typ `DerivedType` lze implicitně konvertovat na odkaz na typ `BaseType` za předpokladu, že je typ `DerivedType` odvozen od typu `BaseType`. Opačnou konverzi však musíte provést explicitně. Jestliže CLR nemůže za běhu programu konverzi provést, vyvolá explicitní konverze výjimku typu `System.InvalidCastException`.

KOVARIANCE POLÍ

V jazyce C# lze také jednoduše provést jeden druh implicitní konverze, který není tak snadno proveditelný v C++, a to zejména kvůli sémantice výchozích hodnot. V C# je možné implicitně konvertovat pole jednoho referenčního typu na pole jiného referenčního typu, pokud lze zdrojový referenční typ implicitně konvertovat na cílový referenční typ a pokud mají obě pole stejný počet rozměrů. To se označuje jako kovariance polí. Platnou konverzi tohoto druhu provádí například následující zdrojový kód:

```
public class EntryPoint
{
    static void Main() {
        string[] names = new string[4];
        object[] objects = names; // příkaz pro implicitní konverzi

        string[] originalNames =
        (string[]) objects; // příkaz pro explicitní konverzi
    }
}
```

Protože je typ `System.String` odvozen od typu `System.Object`, je implicitně konvertovatelný na typ `System.Object` a tato konverze pole řetězců `names` na pole objektů `objects` je platná. Avšak kdybyste chtěli opačnou konverzi, museli byste použít explicitní přetypování, které může vyvolat výjimku za běhu programu, když konverze selže.

Myslete na to, že při volání metody může dojít k implicitní konverzi předaných parametrů, pokud je třeba je převést je na typy, se kterými metoda pracuje. Pokud nemůžete provést jejich konverzi implicitně, musíte je přetypovat explicitně.

BALENÍ HODNOTOVÝCH TYPŮ (BOXING)

Dalším typem konverze je *balení hodnotových typů* (*boxing conversion*). Zabalení hodnotového typu je nutné, když jej potřebujete předat metodě jako typ referenční nebo přiřadit proměnné referenčního typu. Při této konverzi dojde v haldě k dynamické alokaci objektu, který obsahuje složku příslušného hodnotového typu, a jeho hodnota se pak do této složky zkopíruje. Balení detailně probírám ve 4. kapitole. Prohlédnout si ho můžete na následujícím příkladu:

```
{
    static void Main() {
        int employeeID = 303;
```

```
        object boxedID = employeeID;

        employeeID = 404;
        int unboxedID = (int) boxedID;

        System.Console.WriteLine( employeeID.ToString() );
        System.Console.WriteLine( unboxedID.ToString() );
    }
}
```

K zabalení dojde ve chvíli, kdy je proměnné `boxedID` typu `object` přiřazena hodnota proměnné `employeeID` typu `int`. Tento druh konverze přemosťuje propast, která je v CLR mezi světem hodnotových a referenčních typů. Zabalený objekt `boxedID` ve skutečnosti obsahuje kopii hodnoty `employeeID`. Tento fakt zdůrazňuji změnou původní hodnoty proměnné `employeeID` poté, co proběhne operace zabalení. Předtím, než hodnoty zobrazím, vyjmu – vybalím – zabalenou hodnotu a zkopíruji hodnotu uloženou v objektu v haldě zpět do jiné proměnné typu int, která je v zásobníku. Chcete-li vybalit v jazyce C# zabalenou hodnotu, musíte použít explicitní přetypování.

Operátory as a is

Protože explicitní konverze mohou selhat a vyvolat výjimku, je občas potřeba před provedením konverze otestovat typ proměnné, aniž byste konverzi provedli, abyste mohli zjistit, zda selže, či nikoliv. Testování typů prostřednictvím konverzí je komplikované a neefektivní a výskyt výjimek způsobuje za běhu programu výkonové ztráty. Z tohoto důvodu má jazyk C# dva operátory, které můžete použít v obdobných situacích a které vám zaručí, že k vyvolání výjimky nedojde:

- is
- as

Výsledkem použití operátoru `is` je hodnota typu Boolean, která určuje, zda můžete daný výraz konvertovat na daný typ prostřednictvím konverze odkazů nebo konverze typu zabalení/vybalení. Pro příklad uvádím následující zdrojový kód:

```
using System;
public class EntryPoint
{
static void Main() {
String derivedObj = "Dummy";
Object baseObj1 = new Object();
Object baseObj2 = derivedObj;

        Console.WriteLine( "baseObj2 {0} String",
                           baseObj2 is String ? "je" : "není" );
        Console.WriteLine( "baseObj1 {0} String",
                           baseObj1 is String ? "je" : "není" );
        Console.WriteLine( "derivedObj {0} Object",
                           derivedObj is Object ? "je" : "není" );

        int j = 123;
        object boxed = j;
        object obj = new Object();

        Console.WriteLine( "boxed {0} int",
```

```
                        boxed is int ? "je" : "není" );
        Console.WriteLine( "obj {0} int",
                        obj is int ? "je" : "není" );
        Console.WriteLine( "boxed {0} System.ValueType",
                        boxed is ValueType ? "je" : "není" );
    }
}
```

Výstup tohoto zdrojového kódu vypadá takto:

```
baseObj2 je String
baseObj1 není String
derivedObj je Object
boxed je int
obj není int
boxed je System.ValueType
```

Jak jsem již zmínil výše, operátor `is` vyhodnocuje pouze konverze odkazů. To znamená, že nevyhodnotí žádné uživatelsky definované konverze, které jsou definované v typech.

Operátor `as` funguje podobně jako operátor `is`, ale vrací odkaz (referenci) na cílový typ. Protože nemůže nikdy vyvolat výjimku, jednoduše vrátí v případě, že konverzi nelze provést, prázdnou referenci (null). Podobně jako operátor `is` vyhodnocuje i operátor `as` pouze konverze odkazů a konverze typu zabalení/vybalení. Jako příklad uvádím následující zdrojový kód:

```
using System;

public class BaseType {}

public class DerivedType : BaseType {}

public class EntryPoint {
    static void Main() {
        DerivedType derivedObj = new DerivedType();
        BaseType baseObj1 = new BaseType();
        BaseType baseObj2 = derivedObj;

        DerivedType derivedObj2 = baseObj2 as DerivedType;
        if( derivedObj2 != null ) {
            Console.WriteLine( "Konverze provedena úspěšně" );
        } else {
            Console.WriteLine( "Konverzi nebylo možné provést" );
        }

        derivedObj2 = baseObj1 as DerivedType;
        if( derivedObj2 != null ) {
            Console.WriteLine( "Konverze provedena úspěšně" );
        } else {
            Console.WriteLine( "Konverzi nebylo možné provést" );
        }

        BaseType baseObj3 = derivedObj as BaseType;
        if( baseObj3 != null ) {
            Console.WriteLine( "Konverze provedena úspěšně" );
        } else {
            Console.WriteLine( "Konverzi nebylo možné provést" );
```

```
        }
    }
}
```

Jeho výstup vypadá takto:

```
Konverze provedena úspěšně
Konverzi nebylo možné provést
Konverze provedena úspěšně
```

Někdy také potřebujete otestovat, zda je proměnná určitého typu, a pokud ano, provést nad typem nějakou operaci. Testovat, zda je proměnná určitého typu, můžete prostřednictvím klíčového slova `is`, a jestliže je, můžete ji přetypovat na požadovaný typ. Tento postup je však velmi neefektivní. Lepší je postupovat tak, že prostřednictvím operátoru `as` získáte odkaz na proměnnou požadovaného typu a poté otestujete, zda je výsledný odkaz null. Jestliže není, znamená to, že konverze byla úspěšná. Tímto způsobem stačí kontrolovat typ proměnné pouze jednou.

Genericita

Podpora genericity je jednou z nejzajímavějších nových vlastností jazyka C#. Syntaxe generických konstrukcí vám umožňuje definovat typ závisející na jiném typu, který není znám ve chvíli definice generického typu, ale až ve chvíli jeho použití. Představte si například typ kolekce. Kolekci obvykle představují například seznamy, fronty a nebo zásobníky. Typy kolekcí, které se používají již od verze platformy .NET 1.0, mohou obsahovat jakýkoliv typ v CLR, protože obsahují odkazy na instance třídy `Object` a všechny ostatní typy jsou od této třídy odvozeny. Avšak všechny informace o obsažených typech se tak ztratí a překladač již není nadále schopný zachytávat chybné typy. Každý referenční typ obsažený v kolekci musíte přetypovat na referenční typ, který považujete za správný, a to může způsobit chyby za běhu programu. Původní typy kolekcí také mohou obsahovat heterogenní směs typů, místo aby nutily uživatele, aby do nich vkládal pouze instance určitého typu. Tomuto problému se můžete vyhnout například tak, že pro každý typ, který chcete do výčtu zahrnout, napíšete typy jako `ListOfIntegers` a `ListOfStrings`. Rychle však zjistíte, že většina zdrojového kódu spravujícího tyto seznamy je podobná, nebo *generická*, pro jakékoli typy ukládané do seznamu. Díky generickým typům můžete deklarovat otevřený (generický) typ a napsat společný zdrojový kód pouze jednou. Uživatelé vašeho typu poté mohou specifikovat, jaký typ bude kolekce obsahovat, a to ve chvíli, kdy se ji rozhodnou použít.

Díky generickým typům také dosáhnete větší efektivity. Koncepce genericity je tak rozsáhlá, že jsem rozboru jejich deklarace a použití věnoval celou 11. kapitolu. Věřím však, že bych vám měl o použití generických typů něco říci již teď, protože o nich budu několikrát mluvit ještě předtím, než se k 11. kapitole dostanete.

Poznámka

Syntaxe generických typů bude velmi povědomá těm z vás, kteří používají šablony v C++. Musím však zdůraznit, že mezi chováním šablon a generických typů v C# jsou podstatné rozdíly, které popíšu v kapitole 11.

Generické typy se nejčastěji používají při deklaraci typů kolekcí. Podívejte se například na následující zdrojový kód:

```
using System;
using System.Collections.Generic;
using System.Collections.ObjectModel;

class EntryPoint
{
    static void Main() {
        Collection<int> numbers =
            new Collection<int>();
        numbers.Add( 42 );
        numbers.Add( 409 );

        Collection<string> strings =
            new Collection<string>();
        strings.Add( "Pepa" );
        strings.Add( "Jirka" );

        Collection< Collection<int> > colNumbers
            = new Collection<Collection<int>>();
        colNumbers.Add( numbers );

         IList<int> theNumbers = numbers;
         foreach( int i in theNumbers ) {
             Console.WriteLine( i );
        }
    }
}
```

Tento příklad ilustruje použití generického typu `Collection`. O použití generického typu svědčí ostré závorky obklopující název typu. V tomto případě jsem deklaroval kolekci integerů, kolekci řetězců a, abych vám ukázal ještě komplexnější způsob použití generik, kolekci kolekcí integerů. Všimněte si také, že jsem použil i deklaraci generického rozhraní, konkrétně rozhraní `IList<>`.

Když specifikujete typové parametry generického typu tak, že jejich výčet umístíte do lomených závorek jako v případě kolekce `Collection<int>`, deklarujete uzavřený generický typ. V tomto případě předáváte typu `Collection<int>` pouze jeden typový parametr, ale kdyby jich bylo více, oddělili byste je čárkami. Když CLR na tento typ deklarace narazí poprvé, vygeneruje uzavřený typ založený na generickém typu a jeho předaných typových parametrech. Způsob použití uzavřených typů vytvořených z generických typů se nijak neliší od použití jiných typů až na to, že se v deklaraci typu používá k vytvoření uzavřeného typu zvláštní syntaxe s lomenými závorkami.

Nyní byste měli být připravení na občasné odkazy na generické typy, které v knize najdete předtím, než se dostanete k 11. kapitole.

Jmenné prostory

Jazyk C# podobně jako jazyk C++ podporuje jmenné prostory pro logické seskupování objektů. Můžeme se na ně dívat i jako na analogii balíků z Javy. Jmenné prostory vám pomáhají zabránit vzniku kolizí identifikátorů.

Prostřednictvím jmenných prostorů můžete definovat všechny své typy tak, že jejich identifikátory kvalifikuje jmenný prostor, do kterého náležejí. Použití jmenných prostorů jste již viděli v mnoha příkladech, které jsem až doposud v knize uvedl. V programu „Ahoj světe!" v první kapitole jsem například použil třídu `Console`, která je ve jmenném prostoru `System` knihovny tříd prostředí .NET (.NET Framework Class Library) a jejíž plně kvalifikované jméno zní `System.Console`. Za účelem organizace komponent můžete vytvářet své vlastní jmenné prostory. Obecné doporučení zní, abyste použili pro jméno kořenového jmenného prostoru nějaký identifikátor, například jméno vaší firmy, a následně používali jako jména vnořených jmenných prostorů konkrétnější identifikátory knihoven.

Jmenné prostory představují výborný mechanismus, který může usnadnit lokalizaci vašich typů, zejména pokud navrhujete knihovny určené pro jiné vývojáře. Můžete například vytvořit obecný jmenný prostor jako `MojeFirma.Nastroje`, do kterého umístíte nejčastěji používané typy nástrojů, a poté jmenný prostor `MojeFirma.Nastroje.Pokrocile`, do kterého umístíte méně často používané pokročilejší typy. Mohli byste je také samozřejmě umístit do jediného jmenného prostoru. Uživatele by však mohlo společné umístění často používaných a méně často používaných typů do jediného jmenného prostoru mást.

Poznámka

Když pro svůj jmenný prostor vybíráte název, obecně se doporučuje použít šablonu `<JménoFirmy>.<Technologie>.*`, takže první dvě součásti jména vašeho jmenného prostoru oddělené tečkami obsahují jméno vaší firmy a popis příslušné technologie. Poté můžete jmenný prostor konkrétněji klasifikovat. Příklady použití této šablony si můžete prohlédnout v prostředí .NET (šabloně odpovídá například název jmenného prostoru `Microsoft.Win32`).

Deklarace jmenných prostorů

Syntaxe deklarace jmenného prostoru je jednoduchá. Následující zdrojový kód je ukázkou deklarace jmenného prostoru `Acme`:

```
namespace Acme
{
   class Utility {}
}
```

Jmenné prostory nemusí být omezeny pouze na jednu jednotku překladu (tj. na jeden soubor se zdrojovým kódem v jazyce C#). Jinými slovy, deklaraci stejného jmenného prostoru můžete uvést ve více různých souborech se zdrojovým kódem v jazyce C#. Při překladu se identifikátory v jednotlivých deklaracích jmenných prostorů sjednotí do jedné sady identifikátorů příslušného jmenného prostoru. Toto sjednocení se ve skutečnosti týká všech sestavení. Pokud obsahuje typy definované v jednom jmenném prostoru více různých sestavení, budou kompletní jmenný prostor tvořit všechny identifikátory ve všech sestaveních, které typy definují.

Definice jmenných prostorů do sebe můžete také vnořovat. Použít můžete dva různé přístupy. První z nich je předvídatelný:

```
namespace Acme
{
   namespace Utilities
   {
```

```
      class SomeUtility {}
   }
}
```

Kdybyste použili tuto deklaraci a chtěli přistupovat ke třídě `SomeUtility` prostřednictvím jejího plně kvalifikovaného jména, museli byste ji identifikovat jako `Acme.Utilities.SomeUtility`. Následující příklad představuje alternativní způsob definování vnořených jmenných prostorů:

```
namespace Acme
{
}

namespace Acme.Utilities
{
   class SomeUtility {}
}
```

Tento zdrojový kód má úplně stejný efekt jako kód v předchozím příkladu. Ve skutečnosti můžete klidně vynechat i prázdnou deklaraci jmenného prostoru `Acme`. Ponechal jsem ji v kódu jenom proto, abych demonstroval, že deklarace jmenného prostoru `Utilities` není fyzicky vnořená v deklaraci jmenného prostoru `Acme`.

Jakékoliv typy, které deklarujete vně jmenného prostoru, se stanou součástí globálního jmenného prostoru.

Poznámka

Typy byste neměli definovat v globálním jmenném prostoru. Definování typů v globálním jmenném prostoru se říká „zamořování globálního jmenného prostoru" a obecně je považováno za špatnou programovací techniku. Proč, to by vám mělo být jasné. Tento postup nechrání typy definované různými entitami před potenciálními kolizemi jejich názvů.

Používání jmenných prostorů

V příkladu „Ahoj světe" v první kapitole jsem se letmo dotkl možností použití jmenných prostorů. Pojďte si nyní prohlédnout zdrojový kód, v němž se využívá třída `SomeUtility`, kterou jsem definoval v předchozí sekci:

```
public class EntryPoint
{
   static void Main() {
      Acme.Utilities.SomeUtility util =
         new Acme.Utilities.SomeUtility();
   }
}
```

Vždy plně kvalifikovat názvy je zbytečně neohrabané a může to eventuálně vést k vývoji ošklivého případu syndromu karpálního tunelu. Tomu se vyhnete díky direktivě `using`. Tato direktiva říká překladači, že v překladové jednotce nebo v jiném jmenném prostoru používáte jmenný prostor celý. Klíčové slovo `using` v podstatě importuje všechny názvy v daném jmenném prostoru do jmenného prostoru, do něhož je daný jmenný prostor vnořený, což může být globální jmenný prostor překladové jednotky. Podívejte se na následující příklad:

```
using Acme.Utilities;

public class EntryPoint
{
   static void Main() {
      SomeUtility util = new SomeUtility();
   }
}
```

Kód je mnohem jednodušší a o něco čitelnější. Direktiva `using` je na úrovni globálního jmenného prostoru, a proto importuje názvy typů ze jmenného prostoru `Acme.Utilities` do globálního jmenného prostoru. Někdy, když importujete názvy z více různých jmenných prostorů a oba importované jmenné prostory obsahují typy se stejnými názvy, může stále docházet ke konfliktům jmen. V takovém případě můžete ze jmenného prostoru importovat individuální typy a vytvořit pro ně alias. Toho dosáhnete přiřazením alternativního jména (aliasu) jmennému prostoru nebo třídě. Změňme použití třídy `SomeUtility` tak, že přiřadíte alias pouze jí, nikoli všemu ve jmenném prostoru `Acme.Utilities`:

```
namespace Acme.Utilities
{
   class AnotherUtility() {}
}

using SomeUtility = Acme.Utilities.SomeUtility;

public class EntryPoint
{
   static void Main() {
      SomeUtility util = new SomeUtility();
      Acme.Utilities.AnotherUtility =
            new Acme.Utilities.AnotherUtility();
   }
}
```

V tomto zdrojovém kódu je identifikátor `SomeUtility` alias třídy `Acme.Utilities.SomeUtility`. Abych toto tvrzení dokázal, rozšířil jsem jmenný prostor `Acme.Utilities` a přidal do něj novou třídu `AnotherUtility`. Když se však chci odvolat na tuto novou třídu, musím ji plně kvalifikovat, protože jsem pro ni nedeklaroval žádný alias. Předchozí alias může mít mimochodem také úplně jiný název než `SomeUtility`. Použití jiného názvu se může hodit při řešení konfliktu jmen, jinak je ale obecně vzato lepší přiřadit třídě alias odpovídající jejímu jménu, abyste zabránili záměnám při budoucí údržbě zdrojového kódu.

Poznámka

Budete-li se při definování svých jmenných prostorů řídit vhodnými zásadami, nemělo by se vám stát, že byste museli tento problém řešit. Není dobrým zvykem vytvářet jmenné prostory, které obsahují směs různých typů s různou funkcionalitou. Místo toho byste měli vytvářet jmenné prostory obsahující typy, které spolu nějak intuitivně souvisí, aby je mohli ostatní vývojáři snadno dohledat. Při práci s platformou .NET často narazíte na jmenný prostor obsahující některé obecné typy spolupracující s typy v jiném jmenném prostoru `Advanced`, který je do něj vnořený a který obsahuje typy specializovanější. Například jistě narazíte na jmenný prostor `System.Xml.Serialization.Advanced`. V mnoha ohledech tato pravidla tvoření knihoven reflektují zásady snadné lokalizace typů aplikované při vývoji intuitivních uživatelských rozhraní. Jinými slovy, dávejte vašim typům intuitivní názvy a seskupujte je tak, aby byly snadno dohledatelné.

Řízení toku programu

Podobně jako jazyky C++ a Java obsahuje i jazyk C# všechny obvyklé příkazy pro řízení toku programu. Jazyk C# implementuje dokonce i ničemný příkaz `goto`.

if-else, while, do-while a for

Příkazy `if-else` se v jazyce C# používají stejně jako v jazycích C++ a Java. Z hlediska programovacího stylu vždy propaguji používání bloků v příkazech `if` a všech ostatních příkazech pro kontrolu toku programu popsaných v následujících sekcích, a to dokonce i když obsahují pouze jeden příkaz jako v následujícím příkladu:

```
if( <test_podmínky> ) {
   Console.WriteLine( "Jsi tady." );
}
```

Příkazy `while`, `do` a `for` se používají stejně jako v jazycích Java a C++.

switch

Syntaxe příkazu `switch` je velmi podobná jeho syntaxi v C++ a Javě. Hlavní rozdíl spočívá v tom, že v jazyce C# tento příkaz neumožňuje propadnutí do další sekce. Každá sekce musí být zakončena příkazem `break` nebo jiným příkazem pro přenos řízení. Věřím tomu, že se jedná o velmi dobrou vlastnost. V programech napsaných v jazycích C++ a Java se vždy vyskytovaly nespočetné a těžko dohledatelné chyby způsobené tím, že vývojáři zapomněli použít příkaz `break` nebo přestavět pořadí sekcí příkazu `switch`, když jedna z nich předala řízení sekci následující. Naproti tomu překladač jazyka C# okamžitě ohlásí chybu, když najde sekci, která předává řízení sekci následující. Jedinou výjimkou z tohoto pravidla je, že můžete mít více různých návěští (definovaných pomocí klíčového slova `case`) v jedné sekci - viz následující příklad. Propadávání do jiných sekcí můžete také simulovat prostřednictvím příkazu `goto`:

```
switch( k ) {
   case 0:
      Console.WriteLine( "case 0" );
      goto case 1;
   case 1:
   case 2:
      Console.WriteLine( "case 1 or 2" );
      break;
}
```

Poznámka

Používání příkazu `goto` se všeobecně považuje za špatný programovací návyk, protože jeho výsledkem může být zdrojový kód, který je velmi náročný na údržbu. Jenom proto, že můžete, byste jej neměli lehkovážně používat. To znamená, že jsou vzácné případy, ve kterých je jeho použití vhodné. Obecně vzato vám však doporučuji, abyste pokaždé, když se nachytáte při jeho použití, prověřili návrh své aplikace a vyhodnotili, zda nutnost použití příkazu `goto` není výsledkem jeho chyby.

Všimněte si, že každá ze sekcí příkazu je ukončena příkazem pro přenos řízení, který je dokonce i v sekci poslední. Mnoho vývojářů pracujících s jazyky Java a C++ by příkaz `break` v poslední sekci vynechalo, protože program by pokračoval ukončením provádění příkazu `switch` tak jako tak. Krása myšlenky „zákazu propadnutí" však opět spočívá v tom, že i když se vývojář, který bude někdy v budoucnosti kód údržbovat, rozhodne pořadí sekcí příkazu změnit, nemohou vzniknout žádné chyby jako v jazycích C++ nebo Java. Obvykle budete sekce příkazu `switch` ukončovat příkazem `break`, ale můžete použít jakýkoliv jiný příkaz, který přesměruje tok programu ze sekce ven, včetně příkazů `throw`, `return` a nebo `continue`, pokud je příkaz `switch` vnořený do cyklu `for`, ve kterém použití příkazu `continue` dává smysl.

foreach

Příkaz `foreach` vám umožňuje procházet kolekce objektů za použití přirozené syntaxe. Stejnou funkcionalitu můžete implementovat také prostřednictvím příkazu `while`. To však může vypadat ošklivě a procházení kolekcí je tak časté, že je přidání příkazu `foreach` jistě příjemné. Máte-li například pole řetězců, můžete jimi procházet následujícím způsobem:

```
static void Main() {
   string[] strings = new string[5];
   strings[0] = "Pepa";
   strings[1] = "Jirka";
   foreach( string item in strings ) {
      Console.WriteLine( "{0}", item );
   }
}
```

V závorkách cyklu `foreach` můžete deklarovat typ iterační proměnné. V tomto případě to je proměnná typu `string`. Po deklaraci typu iterační proměnné následuje identifikátor kolekce, kterou procházíte. Můžete použít jakýkoliv objekt, který implementuje návrhový vzor „Kolekce".* Kolekcím se podrobně věnuji v 9. kapitole a rozebírám v ní i složky, které musí typ implementovat, aby jej bylo možné považovat za kolekci. Prvky kolekce, se kterými pracuje příkaz `foreach`, musí být přirozeně možno převést na typ iterační proměnné prostřednictvím explicitní konverze. Pokud ne, vyvolá příkaz `foreach` za běhu programu výjimku typu `InvalidCastException`. Chcete-li si to vyzkoušet na vlastní kůži, zkuste spustit tuto upravenou verzi předchozího příkladu:

```
static void Main() {
   object[] strings = new object[5];
   strings[0] = 1;
   strings[1] = 2;
   foreach( string item in strings ) {
      Console.WriteLine( "{0}", item );
   }
}
```

Zapamatujte si také, že zdrojový kód vnořený do příkazu `foreach` se nesmí pokoušet nijak upravovat iterační proměnnou. Tato proměnná je pouze ke čtení. To znamená, že iterační proměnnou

* Typ implementuje návrhový vzor Collection v případě, že implementuje rozhraní `IEnumerable` (nebo `IEnumerable<>`) nebo veřejné metody `GetEnumerator` a `MoveNext` spolu s veřejnou vlastností `Current`. Jinými slovy, abychom typ mohli použít v příkazu `foreach`, nemusí implementovat rozhraní `IEnumerable`, ačkoliv obvykle návrhový vzor Kolekce implementujete tak, že ve svých typech implementujete rozhraní `IEnumerable` (nebo `IEnumerable<>`).

nemůžete předat nějaké metodě jako parametr `out` nebo `ref`. Pokud se o to pokusíte, překladač vás na to rychle upozorní.

break, continue, goto, return a throw

Jazyk C# má také sadu známých příkazů, které předávají řízení programu, aniž by vyhodnocovaly nějakou podmínku. Jedná se o příkazy `break`, `continue`, `goto`, `return` a `throw`. Jejich syntaxe by měla být důvěrně známá každému vývojáři, který pracuje v jazycích C++ nebo Java (i když v Javě příkaz `goto` není). Tyto příkazy se používají v jazyce C# téměř stejně, jako v C++ a v Javě.

Shrnutí

Tato kapitola byla úvodem do syntaxe jazyka C#, ve kterém jsem kladl důraz na to, že je jazyk C# podobně jako jiné objektově orientované jazyky silně typový. Při programování v těchto jazycích se vždy pokoušíte využít nástroje překladače pro kontrolu typů a najít jeho prostřednictvím co nejvíce chyb už při překladu místo toho, abyste je posléze hledali za běhu programu. V CLR jsou typy klasifikovány buď jako hodnotové nebo jako referenční a každá z těchto kategorií má své charakteristické vlastnosti, které budu v této knize nadále rozebírat. Seznámil jsem vás také se jmennými prostory a ukázal vám, jakým způsobem se s jejich pomocí brání zamoření globálního prostoru příliš mnoha typy, jejichž názvy by mohly vzájemně kolidovat. Nakonec jsem vám předvedl funkci kontrolních příkazů jazyka C#, které fungují podobně jako v jazycích C++ a Java.

V následující kapitole se budu věnovat podrobnějšímu rozboru struktury tříd a struktur a současně upozorňovat na rozdíly v chování jejich instancí.