Obsah

3

Stručný úvod do C#

Tato kapitola obsahuje stručný přehled jazyka C#. V této kapitole již předpokládáme určitou znalost programování a proto se zde nezabýváme přílišnými podrobnostmi. Jestliže některému vysvětlení zde nebudete rozumět, vyhledejte si podrobnější vysvětlení v příslušných dalších kapitolách v této knize.

Zdravíme vesmír

Jako spoluúčastník výzkumného programu SETI jsem si řekl, že by bylo vhodnější místo obligátního úvodního programu „Zdravíme svět" vytvořit spíše program „Zdravíme vesmír."

```
using System;
class Pozdrav
{
  public static void Main(string[] args)
  {
      Console.WriteLine("Zdravíme vesmír");
      // projdeme všechny argumenty z příkazového
      // řádku a vytiskneme je
      for (int arg = 0; arg < args.Length; arg++)
          Console.WriteLine("Arg {0}: {1}", arg,
args[arg]);
  }
}
```

Jak jsme se zmínili již dříve, prostředí fzRuntime poskytuje pro veškeré programové informace (neboli metadata) jednotný jmenný prostor. Klauzule using System znamená, že se na všech-

4

ZPRACOVÁNÍ VÝJIMEK

V mnoha knihách věnovaných programování najdete u zpracování výjimek odkaz na kapitolu někde ke konci knihy. V této knize jsme tuto kapitolu umístili hned na začátek, a to z několika důvodů.

Prvním důvodem je to, že zpracování výjimek je do prostředí .NET Runtime velmi hluboce integrováno, a proto se v kódu v C# velice často používá. Kód v C++ lze vždy napsat bez použití výjimek, ovšem v C# taková možnost nepřichází v úvahu.

Druhý důvod spočívá v tom, že uváděné příklady mohou být názornější. Pokud bychom se o zpracování výjimek zmínili až později v knize, v předchozích příkladech bychom je nemohli použít, což znamená, že bychom u příkladů nemohli využívat pokročilé programovací metody.

Bohužel to znamená, že budeme muset pracovat se třídami, aniž bychom je napřed vysvětlili. Proto si následující kapitolu přečtěte jen pro svoji informaci; třídy podrobněji probereme v následující kapitole.

Co je špatné na návratových kódech?

Patrně každý programátor během své kariéry použil následující konstrukci:

```
bool ok = VolaniFunkce();
if (!ok)
{
```

5

Třídy

Třídy jsou srdcem každé aplikace napsané v objektově orientovaném jazyce. Tato kapitola je rozdělena do několika částí. První část popisuje prvky jazyka C#, které se používají častěji, a další části popisují věci, které se nepoužívají tak často a závisejí na typu vytvářeného programu.

Jednoduchá třída

Třída v jazyce C# může být velmi jednoduchá:

```
class VelmiProsta
{
    int prostaHodnota = 0;
}
class Test
{
    public static void Main()
    {
        VelmiProsta vp = new VelmiProsta();
    }
}
```

Tato třída představuje kontejner pro jediné celé číslo. Protože proměnná je deklarována bez určení přístupnosti, je ve třídě `VelmiProsta` soukromá a nelze k ní přistupovat mimo tuto třídu. Tento stav lze specifikovat explicitně použitím modifikátoru `private`.

Celočíselná proměnná `prostaHodnota` je členskou proměnnou třídy; členské proměnné mohou být mnoha různých typů.

6

BÁZOVÉ TŘÍDY A DĚDIČNOST

Jak jsme uvedli již v kapitole 1, „Základy objektově orientovaného programování," někdy se může hodit možnost odvození jedné třídy z jiné třídy. To nastává tehdy, pokud je odvozená třída speciálním případem výchozí třídy.

Třída Inženýr

Následující třída implementuje třídu `Inženýr` s metodami pro zpracování účtování tohoto inženýra:

```
using System;
class Inzenyr
{
    // konstruktor
    public Inzenyr(string jmeno, float
hodinovaSazba)
    {
        this.jmeno = jmeno;
        this.hodinovaSazba = hodinovaSazba;
    }
    // výpočet platby odvozené z hodinové sazby
inženýra
    public float PocitejPlatbu(float hodin)
    {
        return(hodin * hodinovaSazba);
    }
    // vrátí název typu třídy
    public string NazevTypu()
    {
```

7

Řízení přístupu ke členům tříd

Jedním z důležitých rozhodnutí, která je třeba učinit při návrhu každého objektu, je rozhodnutí o stupni přístupnosti jednotlivých členů třídy. V jazyce C# lze přístupnost určovat několika způsoby.

Přístupnost třídy

Nejobecnější úrovní, na níž lze přístupnost řídit, je úroveň třídy. Ve většině případů jsou jedinými platnými modifikátory u třídy `public`, což znamená, že ke třídě může přistupovat každý, a `internal`. Výjimkou z tohoto pravidla jsou vnořené třídy uvnitř jiných tříd, což je o něco komplikovanější situace; popsaná je v kapitole 8, „Třídy podrobněji."

Přístup `internal` nabízí možnost, jak zpřístupnit třídu širší množině tříd a současně zamezit přístupu všem. Tento typ přístupu se nejčastěji používá při psaní pomocných tříd, které by měly zůstat skryty konečnému uživateli tříd. V prostředí .NET Runtime přístup `internal` znamená, že třída je zpřístupněna všem ostatním třídám, které jsou součástí stejného kompletu jako tato třída.

POZNÁMKA *V prostředí C++ je tento typ přístupu obvykle zajištěn použitím spřátelených tříd (friend), čímž se konkrétní třída zpřístupní jiným. Definice friend tříd umožňuje lépe specifikovat, kdo může ke třídě přistupovat, ovšem v praxi obvykle specifikace přístupu pomocí internal zcela postačuje.*

Obecně by všechny třídy měly být deklarovány jako `internal` s výjimkou těch tříd, které jsou určeny ke zpřístupnění uživateli.

8

TŘÍDY PODROBNĚJI

Tato kapitola probírá některá další témata týkající se tříd, včetně konstruktorů, vnořování tříd a pravidel pro přetěžování.

Vnořené třídy

Někdy může být užitečné vložit jednu třídu dovnitř jiné třídy, například jde-li o nějakou pomocnou třídu, která se používá výhradně v nadřazené třídě. Přístupnost vnořené třídy se řídí podobnými pravidly, jaká jsme uvedli u vztahů mezi modifikátory přístupnosti třídy a jejích členů. Stejně jako u členů tříd, modifikátor přístupnosti u vnořené třídy definuje, jak bude vnořená třída přístupná vně nadřazené třídy. Stejně jako privátní proměnná je viditelná uvnitř třídy, privátní vnořená třída je viditelná uvnitř třídy, v níž je obsažena.

V následujícím příkladu třída `Parser` obsahuje vnořenou třídu `Token`, kterou používá interně. Bez použití vnořených tříd by to mohlo být napsáno následovně:

```
public class Parser
{
    Token[] tokeny;
}
public class Token
{
    string nazev;
}
```

V tomto příkladu jsou obě třídy, `Parser` i `Token`, veřejně přístupné, což ovšem není ideální. Nejenže třída `Token` tvoří další z mnoha tříd a zabírá místo v seznamu tříd ve vývojovém prostředí, ale není ani navržena tak, aby byla všeobecně užitečná. Je proto vhodné z ní udělat vnořenou třídu, což nám

9

Struktury (hodnotové typy)

Pro implementaci většiny objektů se používají třídy. Někdy však může být žádoucí vytvořit objekt, který se chová jako vestavěný typ, tedy takový, který se rychle a nenáročně alokuje v paměti a zbytečně nepřináší režii, která je vlastní odkazovým typům. V takovém případě lze použít hodnotový typ, který se v jazyce C# deklaruje pomocí klíčového slova `struct`.

Struktury se chovají podobně jako třídy, mají však navíc několik omezení. Nemohou dědit z žádného jiného typu (ačkoliv implicitně dědí ze třídy `object`), a ze struktur nemohou dědit ostatní třídy.

Struktura Bod

V grafickém systému by se struktura mohla použít k zapouzdření definice obrazového bodu. Mohla by být deklarována například takto:

```
using System;
struct Bod
{
    public Bod(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
    public override string ToString()
    {
```

10

ROZHRANÍ

Rozhraní jsou ve velmi úzkém vztahu s abstraktními třídami. Připomínají abstraktní třídu, jejíž všechny členy jsou abstraktní.

Jednoduchý příklad

Následující program definuje rozhraní `Zvetsovaci` a třídu `ObjektText`, které toto rozhraní implementuje, což znamená, že definuje verze všech funkcí v tomto rozhraní deklarovaných.

```
public class ObjektDiagramu
{
    public ObjektDiagramu() {}
}

interface Zvetsovaci
{
    void ZvetseniX(float faktor);
    void ZvetseniY(float faktor);
}

// Objekt diagramu, který navíc implementuje
Zvětšovací rozhraní
public class ObjektText: ObjektDiagramu, Zvetsovaci
{
    public ObjektText(string text)
    {
        this.text = text;
    }
    // implementace Zvetsovaci.ZvetseniX()
    public void ZvetseniX(float faktor)
```

11

Řízení verzí pomocí NEW a OVERRIDE

Softwarové projekty jen výjimečně existují jako jediná verze kódu, která se nikdy neopravuje, jedině snad v případě softwaru, který nikdy nespatří světlo světa. Ve většině případů autoři softwarových knihoven vždy chtějí nějaké věci měnit a klienti se musí těmto změnám přizpůsobovat.

Práce s těmito záležitostmi se označuje jako řízení verzí a jde o jednu z nejnáročnějších věcí, s jakými se lze ve vývoji softwaru setkat. Jedním z důvodů je, že řízení verzí vyžaduje umění plánovat a předvídat; je třeba určit, které věci se mohou změnit, a návrh musí být upraven tak, aby změny umožňoval.

Dalším důvodem, proč je řízení verzí náročné, je to, že většina prováděcích prostředí nenabízí přílišnou podporu programátorům. V jazyce C++ zkompilovaný kód v sobě nese informace o velikosti a uspořádání všech tříd, které jsou v něm obsaženy. S určitou snahou sice lze do třídy zanést určité změny, aniž by si všichni uživatelé museli kód překompilovat, ovšem omezení jsou poměrně krutá. Při porušení kompatibility musí všichni uživatelé provést rekompilaci, aby mohli novou verzi používat. To ještě nemusí být takový problém, jenže instalace nové verze nějaké knihovny může vést k tomu, že aplikace, které používaly starší verzi knihovny, přestanou fungovat.

Řízená prostředí, která v metadatech nezveřejňují členy tříd nebo informace o vnitřním uspořádání, jsou pro řízení verzí mnohem lepší, stále je však možné napsat kód, který lze inovovat jen ztěží.

12

PŘÍKAZY A ŘÍZENÍ TOKU PROVÁDĚNÍ

Následující kapitola podrobně uvádí různé příkazy, kterými jazyk C# disponuje.

Podmínkové příkazy

Podmínkové příkazy se používají pro podmínečné vykonání operací na základě hodnoty nějakého výrazu.

Příkaz `if`

Příkaz `if` v jazyce C# vyžaduje, aby se podmínkový výraz uvnitř příkazu `if` vyhodnotil na typ `bool`. Jinými slovy, následující použití příkazu `if` je nesprávné:

```
// nefunkční příklad
using System;
class Test
{
    public static void Main()
    {
        int hodnota;

        if(hodnota)              // neplatné
            System.Console.WriteLine("true");

        if(hodnota == 0)         // musíme použít
toto
            System.Console.WriteLine("true");
    }
}
```

14

OPERÁTORY

Co se týče výrazů, syntaxe jazyka C# vychází ze syntaxe jazyka C++.

Přednost operátorů

Když nějaký výraz obsahuje více operátorů, pořadí, v němž jsou jednotlivé prvky výrazu vyhodnocovány, je určeno předností operátorů. Implicitní přednost operátorů lze změnit uzavřením prvků do závorek.

```
int hodnota = 1 + 2 * 3;       // 1 + (2 * 3) = 7
    hodnota = (1 + 2) * 3;     // (1 + 2) * 3 = 9
```

V jazyce C# jsou všechny binární operátory asociativní zleva, což znamená, že operace se provádějí ve směru zleva doprava, s výjimkou operátorů přiřazení a podmíněného výrazu (?:), které se vykonávají zprava doleva.

Všechny operátory seřazené od nejvyšší precedence po nejnižší jsou uvedeny v následující tabulce:

KATEGORIE	OPERÁTORY
Primární	`(x) x.y f(x) a[x] x++ x- new typeof sizeof checked unchecked`
Unární	`+ - ! ~ ++x --x (T)x`
Multiplikativní	`* / %`
Aditivní	`+ -`
Posuv	`<< >>`
Relační	`< > <= >= is`
Rovnost	`== !=`
Logické AND	`&`
Logické XOR	`^`
Logické OR	`|`

15

Převody mezi typy

V jazyce C# lze převody mezi typy rozdělit na převody implicitní a explicitní. Implicitní převody jsou takové, které vždy uspějí - převod může vždy proběhnout bez ztráty dat. U číselných typů to znamená, že cílový typ může plně reprezentovat celý rozsah výchozího typu. Například typ `short` lze implicitně převést na typ `int`, protože rozsah typu `short` je podmnožinou rozsahu typu `int`.

Číselné typy

U číselných typů je k dispozici široká škála implicitních konverzí pro všechny číselné typy, se znaménkem i bez znaménka. Převodní hierarchie je uvedena na obrázku 15-1. Jestliže existuje cesta (vyznačená šipkami) z výchozího typu do typu cílového, pak existuje i implicitní konverze z výchozího do cílového typu. Například existují implicitní konverze z typu `sbyte` na `short`, z typu `byte` na `decimal`, nebo z typu `ushort` na `long`.

Uvědomte si, že cesta z výchozího do cílového typu, tak jak je uvedena na obrázku, nepředstavuje skutečný průběh převodu, jak jej provádí kompilátor; obrázek pouze naznačuje, jaké konverze jsou přípustné. Jinými slovy, převod z typu `byte` na `long` se provede v jediné operaci a nikoliv převodem přes `ushort` a `uint`.

```
class Test
{
    public static void Main()
    {
        // všechny převody jsou implicitní
        sbyte h = 55;
        short h2 = h;
```

16

Pole

Pole jsou v jazyce C# objekty odkazového typu; znamená to, že paměť pro ně se vyhrazuje na hromadě a nikoliv na zásobníku. Jednotlivé prvky pole se ukládají podle toho, jakého jsou typu. Jestliže je prvek odkazového typu (například typu `string`), pole bude uchovávat odkazy na řetězce. Je-li prvek hodnotového typu (například číselného typu nebo typu `struct`), prvky budou uloženy přímo uvnitř pole. Jinými slovy, pole hodnotového typu neobsahuje prvky zabalené do odkazových typů.

Pole se deklarují za použití následující syntaxe:

```
<typ>[] identifikátor;
```

Výchozí hodnotou pole je hodnota null. Samotný objekt pole se vytvoří pomocí příkazu new:

```
int[] sklad = new int[50];
string[] nazvy = new string[50];
```

Bezprostředně po vytvoření pole obsahuje výchozí hodnoty příslušných typů, které jsou v poli obsaženy. U pole sklad je každý prvek typu int s výchozí hodnotou 0. U pole nazvy je každý prvek typu string s výchozí hodnotou null.

Inicializace pole

Pole lze současně s jejich vytvořením inicializovat. Při současné inicializaci lze vynechat příkaz `new int[x]`, přičemž kompilátor stanoví alokovanou velikost pole podle počtu prvků, které se nachází v inicializačním seznamu.

```
int[] sklad = {0, 1, 2, 3, 10, 12};
```

Předchozí řádek je ekvivalentní tomuto:

17

Řetězce

Všechny řetězce jsou v jazyce C# instancemi typu `System.String` z knihovny Common Language Runtime. Díky tomu je pro řetězce k dispozici spousta vestavěných operací. Třída `String` například definuje indexovací funkci, s jejíž pomocí lze procházet přes všechny znaky v řetězci:

```
using System;
class Test
{
    public static void Main()
    {
        string s = "Testovací řetězec";

        for (int index = 0; index < s.Length;
index++)
            Console.WriteLine("Znak: {0}",
s[index]);
    }
}
```

Operace

Třída `string` je příkladem finálního typu, což znamená, že znaky obsažené v řetězci nemohou uživatelé řetězce změnit. Všechny operace, které se nad třídou `string` provádějí, vracejí novou, modifikovanou verzi řetězce a nemodifikují tedy tu instanci, nad níž byla operace provedena.

Třída `String` podporuje následující porovnávací a vyhledávací metody:

18

VLASTNOSTI

Před několika měsíci jsem psal jakýsi program a narazil jsem na situaci, kde jedna členská proměnná třídy (`Filename`) mohla být zděděna z jiné proměnné (`name`). Proto jsem se rozhodl použít idiom (neboli návrhový vzor) jazyka C++ pro vlastnosti a napsat funkci `getFilename()` pro proměnnou, která je odvozena z té původní. Pak jsem musel projít celý program a všechny odkazy na tuto proměnnou nahradit voláním funkce `getFilename()`. To mi zabralo spoustu času, protože šlo o poměrně rozsáhlý projekt.

Navíc jsem si musel pamatovat, že pro zjištění názvu souboru nestačí pouze přečíst členskou proměnnou třídy `Filename`, ale musím k tomu použít volání členské funkce `getFilename()`. To znamená, že takový model je obtížněji pochopitelný; místo pouhého přístupu do proměnné `filename` si musím stále uvědomovat, že při každém přístupu ve skutečnosti volám funkci.

Jazyk C# proto nabízí takový koncept vlastností, v němž se stávají privilegovanými členy jazyka. Vlastnosti se uživateli třídy jeví jako proměnné, ovšem zjišťování aktuální a nastavování nových hodnot probíhá prostřednictvím členských funkcí. Proto lze takto oddělit uživatelský model (členskou proměnnou) od implementačního modelu (členské funkce), což snižuje nutnost přizpůsobení třídy uživatelům třídy a autorovi třídy dává větší volnost ohledně návrhu a údržby třídy.

V prostředí .NET Runtime jsou vlastnosti implementovány na základě pojmenování a několika dalších informací, které přiřazují názvy vlastností příslušným členským funkcím. To znamená, že vlastnosti se mohou v některých jazycích jevit skutečně jako vlastnosti, zatímco v jiných jazycích se jeví jako obyčejné členské funkce.

Vlastnosti se významně uplatňují v rámci knihovny tříd .NET Base Class Library; veřejně přístupné členské proměnné se zde ve skutečnosti používají jen minimálně (pokud vůbec).

19

INDEXERY

Někdy může být výhodné mít možnost indexovat nějaký objekt stejně, jako by to bylo pole. Toho lze dosáhnout tak, že pro daný objekt napíšeme tzv. *indexer*, který zprostředkuje stejnou funkčnost jako přístup do pole. Podobně jako vlastnost vypadá jako jednoduchá proměnná, jejíž čtení a modifikace však probíhá přes přístupové funkce, indexer vypadá jako pole, ale indexovací operace probíhají přes přístupové funkce.

Indexování celočíselným indexem

Třída, která obsahuje jeden řádek z databáze, může indexer implementovat například ke snadnějšímu přístupu k jednotlivým sloupcům v řádku:

```
using System;
using System.Collections;
class DatovaHodnota
{
    public DatovaHodnota(string nazev, object
data)
    {
        this.nazev = nazev;
        this.data  = data;
    }
    public string Nazev
    {
        get
        {
            return(nazev);
        }
```

20

Výčtové typy

Výčtové typy se používají tam, kde nějaká proměnná v programu může nabývat pouze určitých konkrétních hodnot. Příkladem může být nějaký prvek, který může mít pouze některou ze čtyř barev, nebo komunikační program, který podporuje pouze dva protokoly - ve všech podobných situacích může použití výčtů kód zjednodušit.

Výčet typů čar

V následujícím příkladu třída pro kreslení čar používá výčtový typ, jenž deklaruje styly čar, které třída dokáže nakreslit.

```
using System;
public class Kresleni
{
    public enum StylCary
    {
        Plna,
        Carkovana,
        Cerchovana
    }

    public void KresliCaru(int x1, int y1, int x2,
int y2, StylCary stylCary)
    {
        switch (stylCary)
        {
            case StylCary.Plna:
                // nakreslení plné čáry
```

21

ATRIBUTY

Ve většině programovacích jazyků se určitá část informací vyjadřuje prostřednictvím deklarací a jiná část prostřednictvím programového kódu. Například následující deklarace členské proměnné třídy

```
public int Test;
```

znamená, že kompilátor a běhové prostředí vyhradí prostor pro celočíselnou proměnnou a nastaví jeho přístupnost tak, aby proměnná byla přístupná odkudkoliv. Jde o případ deklarativní informace; její výhodnost spočívá v úsporném vyjádření a v tom, že o podrobnosti se postará kompilátor.

Typy deklarativních informací jsou obvykle předem definovány tvůrcem jazyka a uživatelé jazyka je nemohou dále rozšiřovat. Uživatel, který by chtěl například ukládat členskou proměnnou třídy do nějakého pole v databázi, musí sám vymyslet způsob, jak tento vztah vyjádřit prostředky jazyka, jak tento vztah uložit a jak k těmto informacím při běhu programu přistupovat. V jazycích stylu C++ lze definovat makro, jenž bude ukládat informace do proměnné, která je součástí objektu. Tyto přístupy sice fungují, ovšem jsou náchylné ke vzniku chyb a nejsou obecné. Navíc nejsou elegantní.

Prostředí .NET Runtime proto podporuje atributy, což jsou pouhé poznámky, umístěné u jednotlivých prvků zdrojového kódu, jako jsou třídy, členy tříd, parametry atd. Pomocí atributů lze změnit chování běhového prostředí, poskytnout transakční informace o objektech, případně jen předat nějaké informativní údaje uživateli tříd. Informace o atributech jsou uloženy spolu s metadaty příslušného prvku a při běhu programu je lze snadno přečíst prostřednictvím procesu, označovaného jako reflexe.

Jazyk C# obsahuje i podmíněný atribut, který umožňuje stanovit, kdy mají být členské funkce zavolány. Použití podmíněného atributu vypadá takto:

22

DELEGÁTY

Delegáty se podobají rozhraním, alespoň v tom smyslu, že specifikují jakousi dohodu mezi volající a implementující třídou. Na rozdíl od rozhraní však delegát specifikuje pouze tvar jedné jediné funkce. Dalším rozdílem je to, že rozhraní se vytvářejí již při překladu, zatímco delegáty se vytvářejí až za běhu programu.

Použití delegátů

Specifikace delegáta stanovuje tvar funkce a pro vytvoření instance delegáta je nutné použít jen takovou funkci, která odpovídá stanovenému tvaru. Delegáty se někdy označují jako "bezpečné ukazatele na funkce. Na rozdíl od ukazatelů na funkce však delegáty jazyka C# mohou volat i více než jednu funkci. Při spojení dvou delegátů dohromady je výsledkem delegát, který volá oba původní delegáty.

Díky své spíše dynamické povaze se delegáty hodí tam, kde chceme uživateli umožnit změnit chování naší třídy. Jestliže například nějaká třída kolekcí implementuje řazení, můžeme potřebovat, aby podporovala různé způsoby řazení. Řazení lze tedy založit na delegátu, který bude definovat porovnávací funkci.

```
using System;
public class Kontejner
{
    public delegate int PorovnaniPolozek(object
obj1, object obj2);
    public void Serad(PorovnaniPolozek porovnani)
    {
        // nejde o skutečné řazení, pouze o
ukázku.
```

23

Události

Prostřednictvím událostí může jedna třída upozornit jinou třídu (nebo třídy), že se něco stalo. Události využívají metodu označovanou jako „zveřejnění-předplacení" - třída zveřejní všechny události, které může vyvolat, a třídy, které nějaká konkrétní událost zajímá, si tuto událost mohou předplatit.

Události se často používají u grafických uživatelských rozhraní pro upozornění, že uživatel provedl nějakou akci, ovšem hodí se velmi dobře pro jakékoliv asynchronní operace, jako je například změna souborů nebo doručení elektronické zprávy.

Akce, která se zavolá v reakci na událost, se definuje pomocí delegáta. Aby se s událostmi lépe pracovalo, existuje pro ně konvence, která říká, že takový delegát vždy přijímá dva parametry. Prvním parametrem je objekt, který událost vyvolal, a druhým parametrem je objekt, který obsahuje informace o samotné události. Tento objekt je vždy odvozen ze třídy `EventArgs`.

Událost nová pošta

Zde je příklad události:

```
using System;
class UdalostNovaPosta: EventArgs
{
    public UdalostNovaPosta(string predmet, string
zprava)
    {
        this.predmet = predmet;
        this.zprava  = zprava;
    }
    public string Predmet
    {
```

24

Uživatelsky definované převody

Jazyk C# umožňuje uživatelům definovat vlastní převody mezi třídami či strukturami a jinými objekty systému. Uživatelsky definované konverzní funkce jsou vždy statické a musí vždy přijímat jako parametr nebo vracet jako návratovou hodnotu takový typ objektu, ve kterém jsou definovány. To mimo jiné znamená, že konverze nelze definovat mezi dvěma již existujícími typy, což jazyk zjednodušuje.

Jednoduchý příklad

Tento příklad implementuje strukturu, která pracuje s římskými číslicemi. Bylo by možné ji napsat i jako třídu.

```
using System;
using System.Text;
struct RimskeCislo
{
    public RimskeCislo(short hodnota)
    {
        if (hodnota > 5000)
            throw(new
ArgumentOutOfRangeException());

        this.hodnota = hodnota;
    }
```

25

Přetěžování operátorů

Prostřednictvím přetěžování operátorů je možné definovat novou funkčnost operátorů nad třídami nebo strukturami, takže lze určité funkce zapisovat pomocí operátorů. Nejužitečnější je tato možnost u takových datových typů, kde je zřejmé, co který operátor dělá. Uživateli se tak nabídne možnost úsporného vyjádření.

Přetěžování relačních operátorů (==, !=, >, <, >=, <=) je probráno v podkapitole, která se zabývá přetěžováním funkce `Equals()` z knihovny .NET Frameworks, což je v kapitole 27, „Objekty kompatibilní s .NET Frameworks."

Přetěžování operátoru konverze mezi typy je probráno v kapitole 24, „Uživatelsky definované převody."

Unární operátory

Všechny unární operátory se definují jako statické funkce, které přijímají jediný parametr typu příslušné třídy nebo struktury, a vracejí hodnotu stejného typu. Lze přetížit následující operátory:

```
+ - ! ~ ++ -- true false
```

Prvních šest unárních přetížených operátorů se zavolá vždy, když se nad třídou či strukturou provede příslušná operace. Operátory `true` a `false` jsou zde k dispozici pro logické výrazy, kde

```
if (a == true)
```

není ekvivalentní k

26

Další rysy jazyka C#

Tato kapitola se zabývá některými méně podstatnými rysy jazyka C#, například způsobem použití funkce `Main()`, způsobem činnosti preprocesoru, nebo tím, jak zapisovat literálové konstanty.

Funkce Main

Nejjednodušší varianta funkce `Main()` je nám již známa z předchozích příkladů:

```
using System;
class Test
{
    public static void Main()
    {
        Console.WriteLine("Zdravíme vesmír!");
    }
}
```

Vrácení celočíselného statusu

Často může být užitečná možnost předat z funkce `Main()` stavový kód. Toho lze dosáhnout tak, že u funkce `Main()`deklarujeme návratovou hodnotu typu `int`:

```
using System;
class Test
{
    public static int Main()
```

27

Objekty kompatibilní s .NET Frameworks

V předchozích kapitolách jsme vysvětlovali, jak napsat objekty, které budou fungovat v prostředí .NET Runtime. Funkčnost takovýchto samostatných objektů je však omezená oproti objektům, které jsou napsány s ohledem na použití ve složitějších systémech. Tato kapitola proto popisuje, jak napsat uživatelské objekty tak, aby se chovaly podobně jako objekty z knihoven .NET Frameworks.

Funkce podporované všemi objekty

Předefinování funkce `ToString()` ze třídy `object` nám umožňuje elegantně reprezentovat hodnotu objektu ve formě textového řetězce. Jestliže tak neučiníme, funkce `object.ToString()` vrací toliko název třídy.

Funkci `Equals()` ze třídy `object` volají třídy .NET Frameworks ke zjištění, zda jsou si dva objekty rovny.

Třída může rovněž předefinovat operátor == a operátor !=, což umožňuje uživateli používat nad instancemi objektů vestavěné operátory a uživatel tedy nemusí volat funkci `Equals()`.

28

Třída System.Array a třídy kolekcí

Tato kapitola nejprve uvádí přehled tříd kolekcí, které jsou k v .NET Frameworks dispozici. Poté jsou podrobněji probrány jednotlivé třídy a uvedeny příklady rozhraní a funkcí, které je nutné naprogramovat k dosažení požadované činnosti.

Řazení a vyhledávání

Třídy kolekcí z knihovny .NET Frameworks poskytují výhodnou podporu pro řazení a vyhledávání prvků, s vestavěnými funkcemi pro řazení a binární vyhledávání. Třída `Array` poskytuje tutéž funkčnost s tím rozdílem, že její funkce jsou statické a nikoliv členské.

Seřazení pole celých čísel je stejně snadné, jako tento příklad:

```
using System;
class Test
{
    public static void Main()
    {
        int[] pole = {5, 1, 10, 33, 100, 4};
        Array.Sort(pole);
        foreach (int h in pole)
            Console.WriteLine("Prvek: {0}", h);
    }
}
```

29

SPOLUPRÁCE S OKOLÍM

Jednou z nejdůležitějších vlastností jazyka C# je jeho schopnost spolupracovat s existujícími programy, ať již na základě COM nebo v nativních knihovnách DLL. Tato kapitola obsahuje stručný přehled, jak tato spolupráce s okolím funguje.

Použití objektů COM

Jestliže chceme volat nějaký objekt COM, jako první krok je třeba definovat zástupnou třídu (tzv. proxy třídu), která bude definovat funkce příslušného objektu COM a určité doplňující informace. Jedná se o poměrně velké množství nudné práce, které se však ve většině případů můžeme vyhnout tak, že k tomu použijeme nástroj `tlbimp`. Tento nástroj načte informace z typové knihovny COM a zástupnou třídu vytvoří automaticky. Většinou to funguje bez problémů, ovšem je-li třeba v zástupné třídě provádět nějaké složitější operace, je nutné napsat zástupnou třídu ručně. V takovém případě se způsob převádění objektu COM na zástupnou třídu definuje pomocí atributů.

Když máme zástupnou třídu k dispozici, je možné ji použít stejně jako kteroukoliv jinou třídu z .NET a o vše ostatní se postará běhové prostředí.

30

Přehled .NET Frameworks

.NET Frameworks obsahuje spoustu funkcí, které se obvykle ukládají do jazykově specifických běhových knihoven, a proto je důležité vědět, jaké třídy jsou v .NET Frameworks k dispozici.

Formáty čísel

Číselné typy se formátují pomocí členské funkce `Format()` příslušného datového typu. Tuto funkci lze zavolat buď přímo, prostřednictvím funkce `String.Format()`, která volá funkci `Format()` příslušného datového typu, nebo prostřednictvím funkce `Console.WriteLine()`, která volá `String.Format()`.

Definice formátování vlastních uživatelských objektů je probrána v odstavci „Uživatelské formátování objektů" později v této kapitole. Tato část se týká toho, jak formátování probíhá u vestavěných typů.

Existují dvě metody, jak lze specifikovat formátování čísel. Pomocí standardního formátovacího řetězce lze převést číselný typ na konkrétní řetězcovou reprezentaci. Je-li potřeba formátování nějak dále ovlivnit, je možné použít uživatelský formátovací řetězec.

Standardní formátovací řetězce

Standardní formátovací řetězec se skládá ze znaku specifikujícího formát, za nímž následuje posloupnost čísel, specifikujících přesnost. Podporovány jsou následující formáty:

31

Hlubší pohled na C#

Tato kapitola se podrobněji podívá na některé otázky, na něž můžete při práci s jazykem C# narazit. Zabývá se některými tématy, která mohou zajímat autory knihoven a složitějších systémů, jako například syntaktické konvence a dokumentace v XML. Kapitola rovněž popisuje, jak psát bezpečný kód a jak pracuje automatické přidělování paměti v běhovém prostředí .NET Runtime.

Konvence jazyka C#

Ve většině jazyků se k vyjádření nějakého záměru používají určité idiomy. Například v jazyce C se při práci se znakovými řetězci většinou používá ukazatelová aritmetika a nikoliv odkazy na pole. Jazyk C# zatím neexistuje tak dlouho, aby programátoři v této oblasti nabyli dostatek zkušeností, ovšem knihovny .NET Common Language Runtime vnášejí do jazyka určité konvence, které by se měly dodržovat.

Tyto konvence jsou podrobněji popsány v dokumentaci k .NET (konkrétně v „Class Library Design Guidelines") a jsou zvláště důležité pro autory knihoven nebo systémů určených pro další použití.

Těmito konvencemi se řídí i všechny příklady uvedené v této knize, a proto by vám již neměly být neznámé. Dalšími vzory mohou být třídy a ukázkové příklady z .NET Common Language Runtime.

32

Defenzivní programování

Prostředí .NET Runtime poskytuje několik prostředků, jejichž účelem je omezit chybovost v programování. Patří sem podmíněné metody a trasování, s jejichž pomocí lze aplikace rozšířit o kontrolu a logování běhu programu, zachytávat chyby během vývoje a diagnostikovat chyby v již distribuovaném kódu.

Podmíněné metody

Podmíněné metody se typicky používají pro psaní funkcí, které se budou vykonávat pouze v případě, že se program zkompiluje nějakým daným způsobem. Proto se často používají u funkcí, které se volají pouze v laděných verzích aplikace a z ostré verze jsou vynechány, většinou kvůli tomu, že doplňkové kontroly program zpomalují.

V C++ by se téhož dalo dosáhnout za použití makra z vkládaného souboru, které by v případě, že ladicí symbol není definován, změnilo volání funkce na prázdný příkaz.

V jazyce C# však tento přístup nelze použít, protože neumožňuje vkládání souborů ani použití maker. V C# lze funkci označit atributem `Conditional`, který stanovuje, kdy se má volání příslušné funkce vygenerovat a kdy nikoliv. Například:

```
using System;
using System.Diagnostics;

class MojeTrida
{
    public MojeTrida(int i)
```

33

Příkazový řádek

Tato kapitola popisuje přepínače příkazového řádku, které lze kompilátoru předávat. Přepínače, které lze zkrátit, jsou uvedeny tak, že nepovinná část je uzavřena do hranatých závorek ([]).

Přepínače `/out` a `/target` lze při jedné kompilaci použít i vícekrát; vztahují se vždy jen k těm zdrojovým souborům, které jsou za přepínačem uvedeny.

Jednoduché použití

Při jednoduchém použití stačí na příkazovém řádku uvést následující příkaz:

```
csc test.cs
```

Tím se zkompiluje soubor `test.cs` a vytvoří se komplet určený pro konzolu (`.exe`), který lze okamžitě spustit. Na jednom příkazovém řádku lze uvést i více souborů a lze používat i zástupné znaky.

Příkazový soubor

Kompilátor jazyka C# podporuje použití příkazového souboru, který obsahuje všechny přepínače, tak jak se píší na příkazový řádek. Jeho použití je užitečné zejména v situacích, kdy se kompiluje mnoho různých souborů, nebo při použití komplikovaných přepínačů.

Příkazový soubor se specifikuje prostým uvedením na příkazovém řádku:

```
csc @<příkazový_soubor>
```

34

C# v porovnání s jinými jazyky

Tato kapitola si klade za úkol porovnat jazyk C# s jinými jazyky. Jazyk C# sdílí společné kořeny s jazyky C++ a Java, proto se tyto jazyky vzájemně podobají více, než je tomu u mnoha ostatních jazyků. Visual Basic se sice jazyku C# nepodobá tolik, jako výše uvedené jazyky, ovšem přesto s ním sdílí mnoho syntaktických prvků.

Část této kapitoly se zabývá rovněž verzemi jazyků Visual C++ a Visual Basic, určenými pro platformu .NET, protože tyto verze se od svých předchůdců přece jen poněkud liší.

Rozdíly mezi C# a C/C++

Program v jazyce C# bude programátorům v C a C++ povědomý, je zde však několik podstatných rozdílů a mnoho rozdílů drobnějších. Tyto rozdíly jsou shrnuty v následujících odstavcích. Podrobnější pohled na tuto problematiku je uveden v dokumentu Microsoftu „C# pro programátory v C++."

Spravované prostředí

Jazyk C# běží v prostředí .NET Runtime. Z toho vyplývá nejen to, že programátor nemůže mnoho věcí ovlivňovat, ale i to, že se programátor musí připravit na zcela nový způsob práce se systémovým prostředím. Společně to znamená, že pár věcí je jinak:

- Odstraňování objektů provádí systém automatického přidělování paměti, a to až tehdy, když paměť po nepoužívaném objektu potřebuje. Pro určité ukončovací

1

Základy objektově orientovaného programování

Tato kapitola představuje úvod do objektově orientovaného programování. Ti z vás, kteří se s objektově orientovaným programováním již setkali, mohou tuto část přeskočit.

K objektově orientovanému návrhu existuje více přístupů, což dokazuje i množství knih, které byly o této problematice napsány. Následující úvod využívá docela pragmatický přístup a navrhováním se podrobněji nezabývá, ačkoliv i přístupy vycházející z návrhu mohou být pro začátečníky poměrně užitečné.

Co je to objekt?

Objekt je prostě jakési spojení určitých souvisejících informací spolu s funkčností. Objektem může být cokoliv, co má odpovídající obraz ve skutečném světě (například objekt zaměstnanec), něco, co má pouze nějaký abstraktní význam (například okno na obrazovce), či jakákoliv jiná užitečná abstrakce v rámci programu (například seznam úkolů, které je třeba udělat).

Objekt je složen z dat, která tento objekt popisují, a z operací, které lze nad tímto objektem vykonat. Informacemi uloženými v objektu zaměstnanec mohou být například různé identifikační údaje (jméno, adresa), pracovní údaje (pracovní zařazení, plat) a podobně. Mezi vykonatelné operace může patřit vygenerování výplatní pásky pro zaměstnance nebo přijetí nového zaměstnance.

2

Prostředí .NET Runtime

Tvorba modulů, které by bylo možno volat z různých jiných jazyků, byla dříve obtížná. Kód, který byl napsán ve Visual Basicu, není možné volat z Visual C++. Kód napsaný ve Visual C++ sice lze někdy volat z Visual Basicu, ovšem není to nikterak snadné. Visual C++ totiž využívá běhové knihovny C a C++, které mají velmi specifické chování, zatímco Visual Basic pro běh programů používá své vlastní prostředí, jenž má rovněž svoje specifické – jenže odlišné – chování.

To je důvodem, proč byla vyvinuta technologie COM a proč byla velmi úspěšná coby způsob pro psaní softwaru založeného na komponentách. V prostředí C++ je bohužel použití COM poměrně obtížné a v prostředí Visual Basicu nejsou k dispozici všechny její výhody. Proto se také začala široce využívat jen pro tvorbu COM komponent a méně pro tvorbu nativních aplikací. Jestliže tedy jeden programátor napsal kvalitní program v C++ a jiný napsal program ve Visual Basicu, neexistoval žádný jednoduchý způsob, jak by oba programy mohly spolupracovat.

Situace byla obtížná i pro autory knihoven, protože nebylo možné vytvořit jednu variantu, která by fungovala na všech platformách. Kdyby autor zamýšlel knihovnu používat v prostředí Visual Basicu, bylo by použití ve Visual Basicu snadné, avšak naopak taková varianta může znesnadňovat přístup z C++, případně může znamenat nepřijatelný handicap ve výkonnosti. Nebo opačně, knihovna by mohla být určena pro uživatele C++, kde by přinášela vysoký výkon a přístupnost na nízké úrovni, avšak programátory ve Visual Basicu by ignorovala.

Někdy může být taková knihovna napsána s ohledem na oba typy uživatelů, to však obvykle znamená, že byly učiněny určité kompromisy. Pro poslání elektronického dopisu ve Windows