PROGRAMOVÁNÍ

PAOLO PIALORSI, MARCO RUSSO

MICROSOFT LINQ

KOMPLETNÍ PRŮVODCE PROGRAMÁTORA

ARCHITEKTURA, SYNTAXE A TŘÍDY

JAK LINQ INTEGROVAT DO SVÝCH PROJEKTŮ

PARALELIZACE DOTAZŮ, MOŽNOSTI ROZŠÍŘENÍ

UKÁZKY V JAZYCÍCH C# A VISUAL BASIC

CPRESS Microsoft

Paolo Pialorsi, Marco Russo

Microsoft LINQ
Kompletní průvodce programátora

Computer Press, a. s.
Brno
2009

Microsoft LINQ
Kompletní průvodce programátora

Paolo Pialorsi, Marco Russo

Computer Press, a. s., 2009. Vydání první.

Překlad: Jiří Fadrný
Odborná korektura: Matěj Dusík
Jazyková korektura: Petra Láníčková
Vnitřní úprava: Jiří Matoušek
Sazba: Petr Klíma
Rejstřík: Daniel Štreit

Obálka: Martin Sodomka
Komentář na zadní straně obálky: Martin Herodek
Technická spolupráce: Dagmar Hajdajová
Odpovědný redaktor: Martin Herodek
Technický redaktor: Jiří Matoušek
Produkce: Petr Baláš

Computer Press, a. s.,
Holandská 8, 639 00 Brno

Objednávky knih:
http://knihy.cpress.cz
distribuce@cpress.cz
tel.: 800 555 513

ISBN 978-80-251-2735-3
Prodejní kód: K1695
Vydalo nakladatelství Computer Press, a. s., jako svou 3339. publikaci.

Stručný obsah

Část V
LINQ v praxi

Část VI
Přílohy

Obsah

Část II
LINQ pro relační data

Část VI
Přílohy

Úvodní slovo

LINQ mění styl psaní kódu. Alespoň u mě to tak bylo.

Nezpůsobuje to však dobře známý objektově relační aspekt technologie LINQ. Nechápejte mě špatně. Objektově relační stránku věci mám velice rád. Účastnil jsem se vytváření objektově relačních prostředí v Microsoftu po větší část osmi minulých let. Obdivuji tuto technologii a jsem nadšený, že jsme ji vytvořili. Jde o *velmi užitečné* prostředí. Ale nemění vaše smýšlení o kódu. Umožňuje používat objektově orientované postupy při interakci s relačními daty, ale objektové programování dozajista používáte ve svém programovacím jazyce již dlouho.

Ani LINQ pro XML nemění posun v uvažování. Ovšem, jde o skvělou knihovnu. Můžu zapsat kód XML a pochopit jej až další den. Ve Visual Basicu stačí jediný pohled; v C# je nutné bližší zkoumání. Ale stále se jedná o pouhou knihovnu, i když ji zdobí kouzla návrháře Anderse Hejlsberga. Napomáhá vám psát lepší kód XML, ale nemění vaše smýšlení o kódu.

Změna psaní kódu přišla s funkcionálním aspektem LINQ. A to není snadné. Píši programy již velmi dlouho a člověk ve svém uvažování trochu zkostnatí (v mém případě v čistě objektově orientovaném smyslu). Už je v tom dobrý. Nebo si to aspoň myslí...

Ale elegance operátorů LINQ a jejich skladba mě uchvátily. Velmi zřídka se mi v mém vlastním kódu podařilo něco podobného. V průběhu návrhu se stále zřetelněji ukazovalo, že vytváříme něco více než jen řadu dobrých knihoven a pěknou syntaxi na reprezentaci dotazů. Vytvářeli jsme náčrt toho, jak by mohly vypadat nové knihovny. Dávali jsme lidem nástroje na tvorbu těchto nových knihoven. Pracovali jsme na hraně integrace funkcionálního a objektově orientovaného programování. Způsob, jakým dnes píši kód, se změnil.

Jistě, k podobnému vývoji došlo již dříve ve vědecky orientovaných jazycích (jako například LISP). Ale toto je poprvé, alespoň pokud je mi známo, kdy se tato paradigmata spojila ve významném komerčním programovacím jazyce a vzniklo tak prostředí, které se umí vypořádat s naprosto fundamentálními scénáři, jako jsou databáze, XML a paralelní výpočty.

Využijte proto tuto velmi dobrou knihu, abyste se naučili LINQ používat. Nechť vás moji krajané Marco a Paolo provedou skrze všechna zákoutí LINQ pro objekty, LINQ pro SQL, LINQ pro XML a všemi dalšími vymoženostmi, které LINQ obsahuje. Neztraťte však ze zřetele celkový smysl; nechte si čas, abyste pochopili základní principy (tedy výrazy lambda, stromy výrazů, rozšiřující metody atd.). Ponořte se do kapitoly 12, pojednávající o rozšíření LINQ. Můžete být buď pasivními uživateli LINQ, nebo můžete pochopit jeho funkčnost do hloubky. Navrhuji vám druhou cestu, jež vás mnohem více odmění.

Luca Bolognese
hlavní programový manažer LINQ
Microsoft Corporation

Předmluva

Poprvé jsme integrovaný jazyk pro dotazování (Language Integrated Query, LINQ) viděli v září 2005, kdy byl projet LINQ oznámen během konference profesionálních vývojářů (Professional Developers Conference, PDC 2005). Ihned jsme pochopili význam a důsledky LINQ v dlouhodobém měřítku. Zároveň jsme viděli, že by bylo velkou chybou nahlížet na LINQ pouze jako na možnost vytvořit obálku pro přístup k datům. To by byl omyl, protože významným principem, který LINQ přináší, je narůst abstrakce kódu díky používání konzistentního zápisu, díky němuž je kód čitelnější, aniž bychom za to zaplatili ztrátou nadvlády nad programem. LINQ se nám zalíbil, viděli jsme široké pole použitelnosti, ale obávali jsme se možného chybného výkladu jeho klíčových bodů. Z tohoto důvodu jsme začali uvažovat o napsání knihy o LINQ.

Veliká příležitost napsat takovou knihu se nám otevřela, když náš záměr přijalo za svůj vydavatelství Microsoft Press. Napsali jsme počáteční krátkou verzi této knihy, *Introducing Microsoft LINQ*, která vycházela z kódu beta 1. Nesprávně jsme se domnívali, že napsáním první knihy jsme se dostali do půli cesty k sepsání této obsáhlé publikace, ale byli jsme ve skutečnosti pouze v jedné třetině (či spíše jedné čtvrtině). Dostalo se nám mnoha podnětů od čtenářů knihy *Introducing Microsoft LINQ*, a většina komentářů byla negativních. Dnes píšeme tuto předmluvu ke knize *Programming Microsoft LINQ* a myslíme si, že jde skutečně o publikaci, kterou bychom si koupili, kdybychom ji sami nenapsali!

Poté, co jsme prací na knize strávili téměř tři roky, nadešel pro nás okamžik dosažení velikého cíle, ale pro vás jde pouze o začátek. LINQ zavádí deklarativnější styl programování, který není současným trendem. Anders Hejlsberg, hlavní návrhář C#, prohlásil, že LINQ se pokouší řešit impedanční nesoulad mezi kódem a daty. Myslíme si, že LINQ je patrně o jeden krok vepředu před ostatními metodami, jež řeší toto dilema, neboť jej lze rovněž používat na psaní paralelních algoritmů, například pomocí implementace Parallel LINQ (PLINQ).

LINQ může pronikat do celé softwarové architektury, protože jej lze zapojit do libovolné vrstvy aplikace; nicméně, stejně jako u jakéhokoliv jiného nástroje, jej lze používat efektivně i neefektivně. V celé knize jsme se snažili popsat, jak používat LINQ tím nejužitečnějším způsobem. Ale přes všechnu tuto námahu stále cítíme, že LINQ je „nová" technologie. Myslíme si, že na začátku budete, podobně jako my, přirozeně používat LINQ tam, kde vstupuje do hry dotaz do relační databáze. Významným milníkem je psaní algoritmů operujících s daty v paměti prostřednictvím dotazu LINQ pro objekty. To by mělo být snadné. Vlastně již po třech kapitolách budete vědět, jak na to. Ale ve skutečnosti jde o neobtížnější část, protože musíte změnit své smýšlení o kódu. Potřebujete začít přemýšlet v kontextu LINQ. Nenašli jsme magickou formuli, jak vás to naučit. Patrně jako u každé velké změny budete potřebovat čas a praxi, abyste se s problematikou sžili.

Užijte si studium!

Poděkování

Kniha je vždy výsledkem práce mnoha lidí. Naneštěstí se na obálce objeví jen jména autorů. Tyto řádky jsou jen malou náplastí pro všechny, kdo nám pomohli.

Nejprve chceme vyjádřit dík Lucovi Bolognese za jeho pomoc se zdroji a kontakty, jež nám umožnily tuto knihu napsat. Luca nás rovněž poctil napsáním úvodního slova k této knize. Správná slova pro vyjádření vděčnosti nalézáme jen v našem mateřském jazyce: Grazie, Luca!

Chcete rovněž poděkovat všem lidem z Microsoftu, kteří neustále odpovídali na naše dotazy – především Mads Torgersen, Amanda Silver, Erick Thompson, Joe Duffy, Ed Essey, Yuan Yu, Dinesh Kulkarni a Luke Hoban. Speciální poděkování si rovněž zaslouží Charlie Calvert za svou velikou a cennou pomoc.

Dozajista si po přečtení výše uvedeného uvědomujete, že nám nepatří všechna sláva. Máme obrovské štěstí, že nám s redakční prací pomohli někteří skvělí pracovníci ve vydavatelství Microsoft Press: John Pierce a Roger LeBlanc. John pracoval na projektu od okamžiku našeho prvního nápadu, pomohl nám držet se vytyčeného kurzu, odpovídal na veškeré naše dotazy, zůstal shovívavý vůči zpožděním a vylepšil mnoho našich návrhů. Roger byl při redakční práci tak přesný a trpělivý, že vskutku nemáme dostatek slov, jak vyjádřit mimořádnost jeho práce.

Chceme rovněž poděkovat hlavnímu technickému korektorovi, Christophemu Nasarre, který našel chyby, kterých bychom si jinak nevšimli. Dále si náš vděk zaslouží mnoho lidí, kteří měli dostatek trpělivosti číst naše nápady a navrhovali zlepšení i opravy. Patří mezi ně Alberto Ferrari, Bill Ryan, Cristian Civera, Diego Colombo, Luca Regnicoli, Roberto Brunetti a Sergio Murru.

Na závěr děkujeme Francescovi Balenovi a Giovannimu Librandovi, kteří nás podpořili před třemi lety, kdy jsme se rozhodli napsat knihu v angličtině.

Úvod

Tato kniha široce a do hloubky pojednává o integrovaném jazyce pro dotazování (Language Integrated Query, LINQ). Hlavním cílem je poskytnout vám plné znalosti o tom, co je LINQ, a stejně tak předat poznatky, co se v LINQ dělat má a co nikoli. Cílovou skupinou této knihy jsou vývojáři .NET s dobrou znalostí Microsoft .NET 2.0, kteří se zajímají o přechod na úroveň Microsoft .NET 3.5.

Před zahájením prací s LINQ si musíte na svůj vývojový počítač nainstalovat Microsoft .NET Framework 3.5 a Microsoft Visual Studio 2008.

Tato kniha byla napsána v prostředí vydání LINQ a Microsoft .NET 3.5 pro prodej (RTM). Nabízíme vám webové stránky (*http://www.programminglinq.com/*), kde budeme udržovat seznam změn, historii revizí, opravy a blog o tom, co se s projektem LINQ a touto knihou děje. Máme také internetovou stránku (*http://www.programminglinq.com/booklinks.aspx*) se všemi adresami URL z této knihy, seřazenými podle stránek, takže tyto adresy nemusíte kopírovat ručně.

O této knize

Kniha se dělí na pět částí čítajících celkem 18 kapitol, za nimiž následují tři přílohy.

Pokud je pro vás C# 3.0, Visual Basic 2008 či oboje novinkou, doporučujeme vám začít přílohou B resp. C nebo oběma. Tyto přílohy zahrnují nové funkce v těchto jazycích, které vytvářejí plnou podporu LINQ. Jestliže tyto nové verze jazyků znáte, mohou vám uvedené přílohy posloužit jako referenční příručky, když budete mít pochybnosti o jazykové syntaxi při práci s LINQ. Jako hlavní jazyk v příkladech používáme C#, ale téměř všechny uváděné funkce LINQ máte k dispozici i ve Visual Basicu 2008. V případě potřeby používáme Visual Basic 2008, protože v něm existují určité funkce, jež v jazyce C# 3.0 nejsou dostupné.

První část knihy, „Základy LINQ", tvoří úvod do LINQ, vysvětluje jeho syntaxi a poskytuje veškeré informace, které potřebujete pro práci s LINQ a objekty v paměti. Naučit se LINQ pro objekty před ostatními implementacemi LINQ je důležité, protože mnoho z funkcí tohoto použití LINQ slouží i v dalších implementacích LINQ, popisovaných v této knize. Velice vám doporučujeme přečíst si první tři kapitoly této části knihy jako první.

Druhá část knihy nese název „LINQ pro relační data" a věnuje se veškeré implementaci LINQ, která nabízí přístup k relačním datovým úložištím. Implementace LINQ pro SQL je rozdělena na tři kapitoly. V kapitole 4, „LINQ pro SQL: Dotazování na data", se naučíte základy mapování relačních dat na entity LINQ a princip sestavování dotazů LINQ, které se posléze přetransformují na dotazy SQL. V kapitole 5, „LINQ pro SQL: Správa dat", se dozvíte, jak nakládat se změnami v datech načtených z databáze prostřednictvím entit LINQ pro SQL. Kapitola 6, „Nástroje LINQ pro SQL", je průvodcem pomůckami, které vám mohou napomoci definovat datové modely pro LINQ pro SQL. Uvažujete-li o použití těchto postupů ve své aplikaci, doporučujeme vám prostudovat všechny kapitoly věnované LINQ pro SQL.

Kapitola 7, „LINQ pro datovou sadu“, pojednává o implementaci LINQ, která je směřována na datové sady ADO.NET. Máte-li aplikaci využívající datové sady, tato kapitola vám poví, jak integrovat LINQ, či přinejmenším jak progresivně přejít z datových sad na doménový model obsluhovaný LINQ pro SQL či LINQ pro entity.

Kapitola 8, „LINQ pro entity“, nabízí popis implementace LINQ, která vytváří vrstvu pro přístup k ADO.NET Entity Frameworku. Navrhujeme vám přečíst si tuto kapitolu až po kapitolách věnovaných LINQ pro SQL, protože na principy, které jsou v těchto dvou implementacích obdobné, se tato kapitola často odkazuje. V této kapitole předpokládáme, že již znáte ADO.NET Entity Framework. Jestliže s ním nemáte dostatečné zkušenosti, nabízíme vám přílohu, kterou byste si měli přečíst jako první.

Třetí část knihy, „LINQ a XML“, obsahuje dvě kapitoly věnované LINQ pro XML: kapitola 9, „LINQ pro XML: správa informační sady“, a kapitola 10, „LINQ pro XML: dotazování do uzlů“. Doporučujeme vám přečíst si tyto kapitoly předtím, než začnete vyvíjet jakýkoliv program, který načítá či manipuluje s daty v XML.

Čtvrtá část publikace, „Pokročilé LINQ“, obsahuje nekomplexnější téma knihy. V kapitole 11, „Uvnitř stromů výrazů“, se dozvíte, jak ovládat, vytvářet a jednoduše načítat strom výrazů. Kapitola 12, „Rozšíření LINQ“, nabízí informace o tom, jak rozšířit LINQ pomocí svých vlastních datových struktur či obálky existující služby a také vytvořením vlastního poskytovatele LINQ. Kapitola 13, „Paralelní LINQ“, popisuje rozhraní LINQ pro Parallel Framework pro .NET. A poslední, čtrnáctá kapitola s názvem „Další implementace LINQ“ poskytuje přehled nejvýznamnějších komponent LINQ od dalších výrobců. Kteroukoliv kapitolu této části knihy můžete číst nezávisle na ostatních. Jediná kapitola, jež se odkazuje na další kapitolu v této sekci, je dvanáctá kapitola, v níž jsou některé odkazy na kapitolu 11.

Pátá část knihy, „Aplikovaný LINQ“, se věnuje použití LINQ v několika odlišných scénářích v distribuovaných aplikacích. Kapitola 15, „LINQ ve vícevrstvém řešení“, bude přínosná pro všechny, protože se výrazně věnuje architektuře a pomůže vám se správným rozhodováním při návrhu vašich aplikací. Kapitoly 16, 17 a 18 předkládají důležité informace o použití LINQ ve spolupráci s existujícími knihovnami jako ASP.NET, Windows Presentation Foundation, Silverlight a Windows Communication Foundation. Doporučujeme vám přečíst si nejprve kapitolu 15 a poté se zanořit do detailů jednotlivých knihoven. Kteroukoliv z kapitol 16, 17 a 18 můžete vynechat, pokud příslušnou technologii nepoužíváte.

Dodatečný obsah na Internetu

Nový či aktualizovaný materiál, který tvoří doplněk této knihy, naleznete na Internetu na stránkách Microsoft Press Online Develooper Tools. Mezi materiálem najdete aktualizace obsahu knihy, články, odkazy na doprovodný obsah, errata, ukázkové kapitoly atd. Stránka bude brzy dostupná na adrese *www.microsoft.com/learning/books/online/developer* a bude pravidelně aktualizována.

Systémové požadavky

Pro práci s LINQ a s ukázkovým kódem je nutné vyhovět těmto systémových požadavkům:

- Podporované operační systémy: Microsoft Windows Server 2003, Windows Server 2008, Windows Vista, Windows XP SP2
- Microsoft Visual Studio 2008

Doprovodná webová stránka

Tato publikace má doprovodnou internetovou stránku, která vám nabízí veškerý kód v knize. Kód je seřazen podle témat a můžete si jej stáhnout z adresy *http://www.microsoft.com/mspress/companion/9780735624009*.

Podpora knihy

Udělali jsme vše, co je v našich silách, aby kniha byla co nekvalitnější. Microsoft Press nabízí opravy této knihy na Internetu na následující stránce: *http://www.microsoft.com/mspress/support/*.

Máte-li komentáře, dotazy či nápady týkající se této knihy, napište je do vydavatelství Microsoft Press prostřednictvím následujících adres:

pošta:

Microsoft Press
attn: Editor, Programming Microsoft LINQ
One Microsoft Way
Redmond, WA 98052-6399

e-mail:

mspinput@microsoft.com

Nezapomeňte prosím, že na uvedené e-mailové adrese není dostupná podpora tohoto produktu. Informace o podpoře naleznete na internetové stránce společnosti Microsoft, *http://support.microsoft.com*.

Poznámka redakce českého vydání

Nakladatelství Computer Press, které pro vás tuto knihu přeložilo, stojí o zpětnou vazbu a bude na vaše podněty a dotazy reagovat. Můžete se obrátit na následující adresy:

Computer Press
redakce počítačové literatury
Holandská 8
639 00 Brno

nebo

knihy@cpress.cz.

Další informace a případné opravy českého vydání knihy najdete na internetové adrese *http://knihy.cpress.cz/K1695*. Prostřednictvím uvedené adresy můžete též naší redakci zaslat komentář nebo dotaz týkající se knihy. Na vaše reakce se srdečně těšíme.

ČÁST I

Základy LINQ

KAPITOLA 1

Úvod do LINQ

Při surfování po Internetu najdete několik popisů integrovaného jazyka pro dotazování (Language Integrated Query, LINQ), mezi nimi i tyto:

- LINQ je jednotný programovací model pro libovolný druh dat. LINQ vám umožňuje dotazovat se na data a pracovat s nimi v konzistentním modelu, nezávisle na datovém zdroji.
- LINQ je dalším nástrojem pro začlenění dotazů SQL do kódu.
- LINQ je další vrstvou pro abstrakci dat.

Všechny tyto definice jsou do určité míry správné, ale každá z nich se zaměřuje pouze na jeden aspekt LINQ. LINQ umí mnohem více, než jen začlenit dotazy SQL, používá se mnohem snáze než „jednotný programovací model" a zdaleka není jen další množinou pravidel pro modelování dat.

Co je LINQ?

LINQ je programovací model, který zavádí dotazy jako prvořadý princip do všech jazyků Microsoft .NET. Úplná podpora LINQ však vyžaduje určitá rozšíření vámi používaného jazyka. Tato rozšíření zvyšují efektivitu vývojářů a poskytují kratší, smysluplnější a jasnější syntaxi pro manipulaci s daty.

Další informace

Podrobnosti o rozšíření jazyků naleznete v příloze B, „C# 3.0: Nové funkce jazyka" a v příloze C, „Visual Basic 2008: Nové funkce jazyka".

LINQ nabízí metodologii, která zjednodušuje a sjednocuje implementaci libovolného typu přístupu k datům. LINQ vás nenutí používat specifickou architekturu; využívá implementaci několika existujících systémů pro přístup k datům, například:

- RAD/prototyp
- klient/server
- n-vrstev
- chytrý klient

LINQ se poprvé objevil v září 2005 jako technický náhled. Od té doby se vyvinul z rozšíření Microsoft Visual Studia 2005 do integrální součásti .NET Frameworku 3.5 a Visual Studia 2008, které byly vydány v listopadu 2007. První vydaná verze LINQ přímo podporuje několik datových zdrojů. Neobsahuje část LINQ pro entity, která bude vydána s ADO.NET Entity Frameworkem v průběhu roku 2008.

V této knize popisujeme současné a nadcházející implementace LINQ od Microsoftu, které slouží pro přístup k několika datovým zdrojům:

- LINQ pro objekty
- LINQ pro ADO.NET
  - LINQ pro SQL
  - LINQ pro datové sady
  - LINQ pro entity (viz poznámka níže)
- LINQ pro XML

Rozšíření LINQ

LINQ lze rozšířit o podporu dalších datových zdrojů. Mezi možná rozšíření může patřit kupříkladu LINQ pro SharePoint, LINQ pro Exchange či LINQ pro LDAP, máme-li jmenovat jen pár možností. Ve skutečnosti jsou již některé podobné implementace možné prostřednictvím LINQ pro objekty. Možný dotaz LINQ pro reflexi popisujeme v oddíle „LINQ pro objekty" v této kapitole. Pokročilejší rozšíření LINQ rozebíráme v kapitole 12, „Rozšíření LINQ". Některé existující implementace LINQ také obsahuje kapitola 14, „Další implementace LINQ".

LINQ bude mít pravděpodobně dopad na způsob psaní aplikací. Bylo by chybou se domnívat, že LINQ změní architekturu aplikací proto, že jeho cílem je poskytnout množinu nástrojů, které zlepšují implementaci kódu prostřednictvím úpravy několika různých architektur. Nicméně očekáváme, že LINQ ovlivní některé kritické části vrstev v *n*-vrstvé aplikaci. Umíme si například představit použití LINQ v uložené proceduře SQLCLR s přímým přesměrováním dotazu do stroje SQL namísto práce s příkazem SQL.

Z LINQ může vzejít mnoho vývojových větví, ale neměli bychom zapomínat na to, že SQL je široce přijímaným standardem, který nelze z pouhých výkonnostních důvodů tak snadno nahradit. LINQ nicméně představuje zajímavý krok ve vývoji současných běžných programovacích jazyků. Deklarativní povaha LINQ může být přitažlivá i při jiném použití než jen pro přístup k datům, například při paralelním programování prostřednictvím Parallel LINQ (PLINQ). Exekuční prostředí může nabídnout programu napsanému na vyšší úrovni abstrakce, na jaké se pohybuje například LINQ, mnoho dalších služeb. Dnes je významné tuto novou technologii dobře pochopit, ale zásadní vliv může získat až v budoucnu.

Další informace

O Parallel LINQ (PLINQ) pojednává kapitola 13, „Parallel LINQ".

Co je pro LINQ potřeba?

Data spravovaná programem dnes mohou pocházet z rozmanitých datových zdrojů: pole, graf objektů, dokument XML, databáze, textový soubor, klíč v registru, e-mailová zpráva, obsah zprávy z protokolu SOAP (Simple Object Access Protocol), soubor Microsoft Office Excel... Úplný výčet je dost dlouhý.

Každý datový zdroj má svůj vlastní model přístupu k datům. Když se dotazujete do databáze, používáte obvykle SQL. Daty XML se lze probírat prostřednictvím modelu DOM (Document Object Model) či XPath/XQuery. Polem se prochází v iteracích a pro navigaci v grafu objektů se vytvářejí algoritmy. Pro přístup k dalším datovým zdrojům lze použít specifická aplikační programová rozhraní (API), například pro soubory Excelu, e-mailové zprávy či registr systému Windows. Pro přístup k různým datovým zdrojům slouží odlišné programovací modely.

Již bylo učiněno mnoho pokusů o sjednocení metod přístupu k datům do jednoho vyčerpávajícího modelu. Například poskytovatele ODBC (Open Database Connectivity) umožňují dotazy do souboru Excelu stejně jako do úložiště Windows Management Instrumentation (WMI). Prostřednictvím ODBC přistupujete pomocí jazyka podobného SQL k datům reprezentovaným relačním modelem.

Někdy je však efektivnější reprezentovat data v hierarchickém či síťovém modelu než v relačním modelu. A dále, jestliže se datový model neváže na konkrétní jazyk, budete patrně muset spravovat různé typové systémy. Všechny tyto odlišnosti vytvářejí „impedanční nesoulad" mezi daty a kódem.

LINQ se s těmito problémy vyrovnává tak, že nabízí jednotný způsob přístupu a správy dat, aniž by vyžadoval přijetí „všeobjímajícího" modelu. LINQ používá běžné možnosti *operací* nad různými datovými modely namísto sjednocování různých *struktur* v těchto modelech. Jinými slovy, při práci s LINQ uchováváte existující nesourodé datové struktury, například třídy a tabulky, ale dostáváte jednotnou syntaxi na dotazování do všech těchto datových typů bez ohledu na jejich fyzickou reprezentaci. Zamyslete se nad rozdíly mezi grafem v objektech v paměti a relačními tabulkami se složitými vazbami. Prostřednictvím LINQ můžete pro oba modely použít stejnou syntaxi dotazů.

Zde vidíte jednoduchý dotaz LINQ v typickém softwarovém řešení, který vrací názvy zákazníků v Itálii. (Nestarejte se v tuto chvíli o syntaxi a klíčová slova jako *var*.)

```
var query =
    from c in Customers
    where c.Country == "Italy"
    select c.CompanyName;
```

Výsledkem je seznam řetězců. Tyto hodnoty lze vypsat v C# ve smyčce *foreach* takto:

```
foreach ( string name in query ) {
    Console.WriteLine( name );
}
```

Definice dotazu i smyčka *foreach* jsou běžnými výrazy v jazyce C# 3.0, ale co je to *Customers*? V tuto chvíli asi přemítáte, kam se to dotazujeme. Je tento dotaz novou formou vnořeného SQL? Ani v nejmenším. Tentýž dotaz (a smyčku *foreach*) můžete aplikovat na databázi SQL, objekt datové sady (*DataSet*), pole objektů v paměti, na vzdálenou službu i na mnoho dalších druhů dat.

Customers může být kolekce objektů, například:

```
Customer[] Customers;
```

Customers může být datová tabulka v datové sadě:

```
DataSet ds = GetDataSet();
DataTable Customers = ds.Tables["Customers"];
```

Customers může být třída entity popisující fyzickou tabulku v relační databázi:

```
DataContext db = new DataContext( ConnectionString );
Table<Customer> Customers = db.GetTable<Customer>();
```

Nebo může *Customers* být třída entity, která popisuje konceptuální model a mapuje se na relační databázi:

```
NorthwindModel dataModel = new NorthwindModel();
ObjectQuery<Customer> Customers = dataModel.Customers;
```

Jak LINQ pracuje

V kapitole 2, „Základy syntaxe LINQ“, se dozvíte, že syntaxe používaná v LINQ je podobná SQL a nazývá se *dotazovací výraz*. Nějaký dotaz, podobný SQL a smíchaný se syntaxí programu napsaného v jiném jazyce než SQL, se obvykle nazývá vnořené (Embedded) SQL, ale jazyky, které takové dotazy implementují, obvykle používají zjednodušenou syntaxi. Ve vnořeném SQL nebývají příkazy integrovány do nativní syntaxe jazyka a do typového systému, protože mají odlišnou syntaxi a některá omezení týkající se jejich interakce. A co víc, LINQ se oproti vnořenému SQL neomezuje pouze na dotazy do databáze. LINQ nabízí ve srovnání s vnořeným SQL mnohem více – syntaxi dotazování, která je integrovaná do jazyka. Ale jak tedy LINQ pracuje?

Když napíšete v LINQ následující kód:

```
Customer[] Customers = GetCustomers();
var query =
    from c in Customers
    where c.Country == "Italy"
    select c;
```

kompilátor vygeneruje tento kód:

```
Customer[] Customers = GetCustomers();
IEnumerable<Customer> query =
    Customers
    .Where( c => c.Country == "Italy" );
```

Když dotaz začne být složitější, jako například tento (od této chvíle budeme kvůli stručnosti vynechávat deklaraci *Customers*):

```
var query =
    from c in Customers
    where c.Country == "Italy"
    orderby c.Name
    select new { c.Name, c.City };
```

vygenerovaný kód bude rovněž komplikovanější:

```
var query =
    Customers
    .Where( c => c.Country == "Italy" );
    .OrderBy( c => c.Name )
    .Select( c => new { c.Name, c.City } );
```

Jak vidíte, kód zjevně volá členy instance objektu navráceného v předchozím volání: *Where* se volá na objektu *Customers*, *OrderBy* se volá na objektu navráceném klauzulí *Where* a na závěr, *Select* se volá na objektu navráceném klauzulí *OrderBy*. Uvidíte, že toto chování je regulováno tzv. *rozšiřujícími metodami* v hostujícím jazyce (v našem případě C#). Implementace metod *Where*, *OrderBy* a *Select* – které se volají v ukázce – závisí na typu objektu *Customers* a na jmenných prostorech specifikovaných v příslušných příkazech *using*. Rozšiřující metody jsou základním rysem syntaxe a slouží v LINQ pro práci s různými datovými zdroji prostřednictvím totožné syntaxe.

Další informace

Rozšiřující metody zdánlivě rozšiřují třídu (v naší ukázce třídu *Customers*), ale ve skutečnosti metoda externího typu přijímá instanci třídy, která je rozšiřována jako první parametr. Klíčové slovo *var*, které se používá v deklaraci dotazu – *query* – určuje typ deklarované proměnné z počátečního přiřazení; v našem případě jde o návratový typ *IEnumerable<T>*. Další popis těchto i jiných jazykových rozšíření naleznete v přílohách B a C.

Dalším významným principem je načasování operací nad daty. Obecně se dotaz LINQ neprovádí, dokud nedojde z nějakého důvodu k vyžádání výsledku dotazu. Dotaz popisuje množinu operací, které se provedou pouze v případě, kdy k výsledku dotazu přistoupí program. V následujícím příkladě dojde k tomuto přístupu až při vykonávání smyčky *foreach*:

```
var query = from c in Customers ...
foreach ( string name in query ) ...
```

Existují rovněž metody, které procházejí výsledky dotazu LINQ postupně, což vede k vytvoření trvalé kopie dat v paměti. Například metoda *ToList* vytváří typovou kolekci *List<T>*:

```
var query = from c in Customers ...
List<Customer> customers = query.ToList();
```

Když dotaz LINQ probíhá nad daty v relační databázi (například databáze Microsoft SQL Server), generuje namísto operací nad kopiemi datových tabulek v paměti odpovídající příkaz SQL. Vykonání příkazu v databázi je odloženo až do chvíle, kdy dojde k prvnímu přístupu k výsledkům dotazu. Kdyby proto v předchozích dvou příkladech představoval objekt *Customers* typ *Table<Customers>* (fyzická tabulka v relační databázi) nebo typ *ObjectQuery<Customer>* (konceptuální entita mapovaná na relační databázi), odpovídající dotaz SQL by se do databáze poslal až ve chvíli, kdy by došlo k provádění smyčky *foreach*

nebo k volání metody *ToList*. Dokud k těmto událostem nedojde, lze s dotazem LINQ různým způsobem manipulovat a sestavovat jej.

Další informace

Dotaz LINQ lze reprezentovat ve formě stromu výrazu. V kapitole 11, „Uvnitř stromů výrazů", popisujeme, jak procházet a dynamicky sestavovat strom výrazu, který je tedy rovněž dotazem LINQ.

Relační model versus hierarchický/síťový model

Na první pohled se LINQ může jevit jen jako další dialekt SQL. Tato podobnost má své kořeny ve způsobu, jakým LINQ popisuje vztahy mezi entitami, což dokládá následující kód:

```
var query =
    from c in Customers
    join o in Orders
         on c.CustomerID equals o.CustomerID
    select new { c.CustomerID, c.CompanyName, o.OrderID };
```

Tato syntaxe je podobná běžnému dotazování na data v relačním modelu prostřednictvím klauzule *join* v SQL. Nicméně LINQ se neomezuje pouze na jeden model reprezentace dat, například na relační, kde se vztahy mezi entitami vyjadřují uvnitř dotazu, avšak nikoli v datovém modelu. (Cizí klíče udržují referenční integritu, ale nejsou součástí dotazu.) V hierarchickém či síťovém modelu jsou vztahy podřízený/nadřízený součástí datové struktury. Předpokládejme například, že každý zákazník má svou množinu objednávek a každá objednávka má svůj seznam produktů. V LINQ načteme seznam objednaných produktů pro jednotlivé zákazníky následujícím způsobem:

```
var query =
    from c in Customers
    from o in c.Orders
    select new { c.Name, o.Quantity, o.Product.ProductName };
```

V tomto dotazu nejsou žádná spojení. Vztah mezi objekty *Customers* a *Orders* vyjadřuje druhá klauzule *from*, jež pomocí syntaxe *c.Orders* říká „načti všechny objednávky zákazníka *c*". Vztah mezi *Orders* a *Products* vyjadřuje člen *Product* instance *Order*. Výsledek vypisuje název produktu pro každý řádek pomocí výrazu *o.Product.ProductName*.

Hierarchické a síťové vztahy se vyjadřují v definicích typů prostřednictvím odkazů na další objekty. (Nadále budeme používat výraz „graf objektů", čímž obecně odkazujeme na hierarchické či síťové modely.) Pro podporu předchozího dotazu bychom založili třídy podle výpisu 1.1.

Výpis 1.1 Deklarace typů s jednoduchými vazbami

```
public class Customer {
    public string Name;
    public string City;
    public Order[] Orders;
}
public struct Order {
    public int Quantity;
    public Product Product;
}
```

```
public class Product {
    public int IdProduct;
    public decimal Price;
    public string ProductName;
}
```

Ale je možné, že budeme chtít používat tutéž instanci třídy *Product* pro mnoho odlišných objednávek téhož produktu. Patrně také budeme chtít filtrovat položky *Order* či *Product*, aniž bychom k nim přistupovali přes objekt typu *Customer*. Běžný postup by vypadal jako výpis 1.2.

Výpis 1.2 Deklarace typů s dvojcestnými vazbami

```
public class Customer {
    public string Name;
    public string City;
    public Order[] Orders;
}
public struct Order {
    public int Quantity;
    public Product Product;
    public Customer Customer;
}
public class Product {
    public int IdProduct;
    public decimal Price;
    public string ProductName;
    public Order[] Orders;
}
```

Pomocí pole všech produktů, definovaného jako:

```
Product[] products;
```

se můžeme dotazovat grafů objektů a vyžadovat seznam objednávek produktu s ID rovno 3:

```
var query =
    from p in products
    where p.IdProduct == 3
    from o in p.Orders
    select o;
```

Prostřednictvím téhož dotazovacího jazyka se lze dotazovat do různých datových modelů. Nemáte-li mezi entitami v dotazu LINQ definovány vztahy, vždy se můžete spoléhat na vnořené dotazy a spojení, které jsou v syntaxi LINQ dostupné právě tak jako v jazyce SQL. Ale jestliže váš datový model již obsahuje vztahy mezi entitami, můžete je použít a předejít tak zdvojení (a případným chybám) stejné informace.

Jestliže ve vašem datovém modelu již existují vazby mezi entitami, stále můžete v dotazu LINQ použít explicitní vztahy – když například chcete vynutit určitou podmínku nebo když zkrátka chcete dát do souvislosti entity, které nativní vztah nemají. Představme si kupříkladu, že chcete nalézt zákazníky a dodavatele, kteří žijí v tomtéž městě. Váš datový model explicitní vazbu mezi těmito atributy patrně neobsahuje, ale v LINQ můžete napsat následující:

```
var query =
    from c in Customers
    join s in Suppliers
         on c.City equals s.City
    select new { c.City, c.Name, SupplierName = s.Name };
```

Vrátí se data podobná těmto:

```
City=Torino     Name=Marco    SupplierName=Trucker
City=Dallas     Name=James    SupplierName=FastDelivery
City=Dallas     Name=James    SupplierName=Horizon
City=Seattle    Name=Frank    SupplierName=WayFaster
```

Máte-li zkušenosti s psaním dotazů SQL, patrně budete předpokládat, že návratová data budou vždy tvořit „pravoúhlou" tabulku, v níž se při spojení podobném předchozímu dotazu mnohokrát opakují data z určitých sloupců. Ale dotaz často obsahuje několik entit s jednou či více vazbami typu 1 ku mnoho (1-*). V LINQ můžete psát dotazy podobné následujícímu, který vrací graf objektů:

```
var query =
    from c in Customers
    join s in Suppliers
         on c.City equals s.City
         into customerSuppliers
    select new { c.City, c.Name, customerSuppliers };
```

Tento dotaz vrací řádek pro každého zákazníka a v každém řádku je seznam dodavatelů ze stejného města, jako bydlí zákazník. Dotaz se dá provádět opakovaně, stejně jako u jakéhokoliv jiného grafu objektů v LINQ. Takto by mohly vypadat *hierarchizované* výsledky:

```
City=Torino Name=Marco customerSuppliers=...
  customerSuppliers: Name=Trucker City=Torino
City=Dallas Name=James customerSuppliers=...
  customerSuppliers: Name=FastDelivery City=Dallas
  customerSuppliers: Name=Horizon City=Dallas
City=Seattle Name=Frank customerSuppliers=...
  customerSuppliers: Name=WayFaster City=Seattle
```

Chcete-li získat seznam zákazníků a dát každému zákazníkovi seznam produktů, které si alespoň jedenkrát objednal, a seznam dodavatelů ze stejného města, použijte následující dotaz:

```
var query =
    from c in Customers
    select new {
       c.City,
       c.Name,
       Products = (from o in c.Orders
                   select new { o.Product.IdProduct,
                                o.Product.Price }).Distinct(),
       CustomerSuppliers = from s in Suppliers
                           where s.City == c.City
                           select s };
```

Podívejte se na výsledky pro několik zákazníků a uvědomte si, jakým způsobem vrací jediný, výše uvedený dotaz data:

```
City=Torino Name=Marco Products=... CustomerSuppliers=...
  Products: IdProduct=1 Price=10
  Products: IdProduct=3 Price=30
  CustomerSuppliers: Name=Trucker City=Torino
City=Dallas Name=James Products=... CustomerSuppliers=...
  Products: IdProduct=3 Price=30
  CustomerSuppliers: Name=FastDelivery City=Dallas
  CustomerSuppliers: Name=Horizon City=Dallas
```

Tento typ výsledku jen stěží získáte s jedním či více dotazy SQL, protože by vyžadoval analýzu výsledků, aby bylo možné sestavit požadovaný graf objektů. LINQ nabízí snadný způsob, jak přesouvat data z jednoho modelu do druhého, a různé metody pro získání stejných výsledků.

LINQ vyžaduje, abyste svá data popisovali prostřednictvím entit, které zároveň tvoří typy v jazyce. Když vytváříte dotaz LINQ, jde vždy o množinu operací na instancích určitých tříd. Tyto objekty mohou být skutečným datovým kontejnerem nebo prostým popisem (v metadatech) externí entity, s níž chcete manipulovat. Dotaz do databáze lze prostřednictvím příkazu SQL poslat pouze tehdy, když jej aplikujete na množinu typů, které se mapují na tabulky a vztahy v databázi. Po nadefinování tříd entit můžete používat *oba* právě popsané přístupy (spojení i vztahy mezi entitami). Převod všech těchto operací na příkazy SQL je věcí LINQ.

Poznámka

Třídy entit můžete vytvářet pomocí nástrojů na generování kódu, například SQLMetal nebo LINQ to SQL Designer v Microsoft Visual Studiu. Tyto nástroje popisuje kapitola 6, „Nástroje LINQ pro SQL".

Ve výpisu 1.3 vidíte ukázku třídy *Product*, jež se mapuje na relační tabulku Products s pěti sloupci, které odpovídají veřejným datovým členům.

Výpis 1.3 Deklarace třídy mapované na tabulku v databázi

```
[Table("Products")]
public class Product {
   [Column(IsPrimaryKey=true)] public int IdProduct;
   [Column(Name="UnitPrice")] public decimal Price;
   [Column()] public string ProductName;
   [Column()] public bool Taxable;
   [Column()] public decimal Tax;
}
```

Když pracujete s entitami, které popisují externí data (například databázové tabulky), můžete vytvořit instance těchto tříd a manipulovat s objekty v paměti způsobem, jako kdyby do ní byla načtena data ze všech tabulek. Změny se do databáze promítnou prostřednictvím příkazů SQL ve chvíli, kdy zavoláte metodu *SubmitChanges*, což dokládá výpis 1.4.

Výpis 1.4 Aktualizace databáze voláním metody SubmitChanges

```
var taxableProducts =
    from p in db.Products
    where p.Taxable == true
    select p;
foreach( Product product in taxableProducts ) {
    RecalculateTaxes( product );
}
db.SubmitChanges();
```

Třída *Product* v předchozí ukázce představuje řádek v tabulce Products v externí databázi. Když zavoláte metodu *SubmitChanges*, všechny změněné objekty vygenerují příkaz SQL, který provede synchronizaci odpovídajících datových tabulek v databázi – v našem případě zaktualizuje příslušné řádky v tabulce Products.

Další informace

Třídy entit, které odpovídají tabulkám a vztahům v databázi, popisuje podrobněji kapitola 4, „LINQ pro SQL: Dotazování na data", kapitola 5, „LINQ pro SQL: Správa dat", a kapitola 8, „LINQ pro entity".

Manipulace s XML

LINQ nabízí pro podporu manipulace s daty XML samostatnou množinu tříd a rozšíření. Představte si, že vaši zákazníci mohou posílat objednávky v souborech XML, například ORDERS.XML, který vidíte ve výpisu 1.5.

Výpis 1.5 Fragment souboru XML s objednávkami

```
<?xml version="1.0" encoding="utf-8" ?>
<orders xmlns="http://schemas.devleap.com/Orders">
    <order idCustomer="ALFKI" idProduct="1" quantity="10" price="20.59"/>
    <order idCustomer="ANATR" idProduct="5" quantity="20" price="12.99"/>
    <order idCustomer="KOENE" idProduct="7" quantity="15" price="35.50"/>
</orders>
```

Pomocí běžných tříd Microsoft .NET 2.0 ze jmenného prostoru *System.Xml* lze načíst soubor v DOM nebo lze jeho obsah analyzovat pomocí třídy *XmlReader*, což dokládá výpis 1.6.

Výpis 1.6 Načtení souboru XML pomocí třídy XmlReader

```
String nsUri = "http://schemas.devleap.com/Orders";
XmlReader xmlOrders = XmlReader.Create( "Orders.xml" );

List<Order> orders = new List<Order>();
Order order = null;
while (xmlOrders.Read()) {
   switch (xmlOrders.NodeType) {
      case XmlNodeType.Element:
         if ((xmlOrders.Name == "order") &&
         (xmlOrders.NamespaceURI == nsUri)) {
            order = new Order();
            order.CustomerID = xmlOrders.GetAttribute( "idCustomer" );
            order.Product = new Product();
            order.Product.IdProduct =
               Int32.Parse( xmlOrders.GetAttribute( "idProduct" ) );
            order.Product.Price =
               Decimal.Parse( xmlOrders.GetAttribute( "price" ) );
            order.Quantity =
               Int32.Parse( xmlOrders.GetAttribute( "quantity" ) );
            orders.Add( order );
         }
         break;
   }
}
```

Pro načtení uzlů lze rovněž použít dotaz XQuery podobný následujícímu:

```
for $order in document("Orders.xml")/orders/order
return $order
```

Ale XQuery opět vyžaduje naučit se další jazyk a syntaxi. A co víc, výsledek předchozího dotazu XQuery je potřeba převést na množinu instancí třídy *Order*, aby bylo možné jej použít v našem kódu.

Bez ohledu na zvolené řešení musíte vždy brát v úvahu uzly, typy uzlů, jmenné prostory XML a vše, co souvisí se světem XML. Mnoho vývojářů s XML pracuje nerado, protože je potřeba znát další oblast datových struktur a manipulace s XML má svou vlastní syntaxi. Pro spoustu z nich není příliš intuitivní. Jak jsme si již řekli, LINQ nabízí nástroje pro všechny druhy zdrojů, dokonce i pro dokument XML. Pomocí dotazů LINQ můžete dosáhnout téhož výsledku s menší námahou a v jednotné programovací syntaxi. Výpis 1.7 ukazuje dotaz LINQ pro XML nad souborem s objednávkami.

Výpis 1.7 Načtení souboru XML pomocí LINQ pro XML

```
XDocument xmlOrders = XDocument.Load( "Orders.xml" );

XNamespace ns = "http://schemas.devleap.com/Orders";
var orders = from o in xmlOrders.Root.Elements( ns + "order" )
             select new Order {
                        CustomerID = (String)o.Attribute( "idCustomer" ),
                        Product = new Product {
                           IdProduct = (Int32)o.Attribute("idProduct"),
                           Price = (Decimal)o.Attribute("price") },
                        Quantity = (Int32)o.Attribute("quantity")
                     };
```

Pomocí nové syntaxe v jazyce Microsoft Visual Basic 2008 se můžete odkazovat na uzly XML ve svém kódu pomocí syntaxe podobné XPath, což ukazuje výpis 1.8.

Výpis 1.8 Načtení souboru XML pomocí LINQ pro XML a syntaxe Visual Basic 2008

```
Imports <xmlns:o="http://schemas.devleap.com/Orders">
' ...

Dim xmlOrders As XDocument = XDocument.Load("Orders.xml")
Dim orders = _
   From o In xmlOrders.<o:orders>.<o:order> _
   Select New Order With {
      .CustomerID = o.@idCustomer, _
      .Product = New Product With {
         .IdProduct = o.@idProduct,
         .Price = o.@price}, _
      .Quantity = o.@quantity}
```

Výsledek těchto dotazů LINQ pro XML lze použít pro jednoduché načtení seznamu entit *Order* do vlastnosti zákazníka *Orders* a pomocí LINQ pro SQL promítnout tyto změny do vrstvy fyzické databáze:

```
customer.Orders.AddRange(
   From o In xmlOrders.<o:orders>.<o:order> _
   Where o.@idCustomer = customer.CustomerID _
   Select New Order With {
      .CustomerID = o.@idCustomer, _
      .Product = New Product With {
         .IdProduct = o.@idProduct,
         .Price = o.@price}, _
      .Quantity = o.@quantity})
```

A pokud potřebujete generovat soubor ORDERS.XML na základě objednávek svého zákazníka, máte přinejmenším možnost využít přímé výrazy jazyka Visual Basic 2008 pro XML a definovat tak výstupní strukturu XML. Jde o výhradní funkci jazyka Visual Basic a nemá ekvivalent v C#. Ukázku vidíte ve výpisu 1.9.

Výpis 1.9 Vytvoření XML s objednávkami pomocí výrazů pro XML ve Visual Basicu 2008

```
Dim xmlOrders = <o:orders>
   <%= From o In orders _
      Select <o:order idCustomer=<%= o.CustomerID %>
                  idProduct=<%= o.Product.IdProduct %>
                  quantity=<%= o.Quantity %>
                  price=<%= o.Product.Price %>/> %>
   </o:orders>
```

Snad oceníte sílu tohoto řešení, které uchovává syntaxi XML, aniž by došlo ke ztrátě stability typového kódu, a transformuje množinu entit vybraných pomocí LINQ pro SQL do objektu XML *InfoSet*.

Další informace

Další informace o syntaxi LINQ pro XML a možnostech, které nabízí, naleznete v kapitole 9, „LINQ pro XML: Správa informačních množin", a v kapitole 10, „LINQ pro XML: Dotazování do uzlů".

Jazyková integrace

Jazyková integrace je základním aspektem LINQ. Nejviditelnější částí je funkce dotazovacího výrazu, která je součástí C# 3.0 a Visual Basicu 2008. Umožňuje psát kód podle dříve uvedených ukázek. Můžete například napsat kód:

```
var query =
    from c in Customers
    where c.Country == "Italy"
    orderby c.Name
    select new { c.Name, c.City };
```

namísto:

```
var query =
        Customers
        .Where( c => c.Country == "Italy" );
        .OrderBy( c => c.Name )
        .Select( c => new { c.Name, c.City } );
```

Mnoho lidí nazývá toto zjednodušení *syntaktickým cukrátkem*, protože jde zkrátka o jednodušší způsob zápisu kódu, který definuje dotaz nad daty. Ale jde o mnohem více. Je potřebných mnoho jazykových konstrukcí a syntaxí, aby fungovalo těchto pár řádků kódu, které se zdánlivě pouze ptají na data. V pozadí tohoto prostého dotazu stojí implikace lokálních typů, rozšiřující metody, výrazy lambda, výrazy inicializace objektů a anonymní typy. Všechno jsou to užitečné funkce samy o sobě, ale jestliže se podíváte na celkový obraz situace, spatříte významné kroky ve dvou směrech: jeden směrem k deklarativnímu stylu psaní kódu a druhý ke snížení impedančního nesouladu mezi daty a kódem.

Deklarativní programování

Jaké jsou rozdíly mezi dotazem SQL a odpovídajícím programem v C# či Visual Basicu 2005, který filtruje data v nativním úložišti (například tabulka v SQL či pole v C# nebo Visual Basicu)?

V SQL píšete následující dotaz:

```
SELECT * FROM Customers WHERE Country = 'Italy'
```

V C# byste patrně napsali toto:

```
public List<Customer> ItalianCustomers( Customer customers[] )
{
    List<Customer> result = new List<Customer>();
    foreach( Customer c in customers ) {
        if (c.Country == "Italy") result.Add( c );
    }
    return result;
}
```

Poznámka

Tuto specifickou ukázku bylo možné v C # zapsat pomocí predikátu *Find*, ale nám slouží pouze jako příklad odlišných programovacích postupů.

Psaní i čtení kódu C# trvá déle. Ale nejvýznamnějším faktorem je míra otevřenosti. V SQL popisujete, *co* vlastně chcete. V C# popisujete, *jak* získat potřebné výsledky. V SQL spočívá zodpovědnost za výběr nejlepšího algoritmu pro načtení výsledků (který je v C# vyjádřen více explicitně) na dotazovacím stroji. Dotaz SQL má větší volnost při aplikaci optimalizací oproti kompilátoru C#, jenž je při určování způsobu provádění operace mnohem svázanější.

LINQ umožňuje používat deklarativnější styl psaní kódu v C# i Visual Basicu. Dotaz LINQ popisuje operace na datech namísto iterativního přístupu prostřednictvím deklarativní konstrukce. LINQ umožňuje explicitněji vyjádřit záměry programátorů a vědomí těchto záměrů je zásadní při získávání lepší úrovně služeb od základního prostředí. Uvažme například paralelizaci. Dotaz SQL lze rozdělit na několik souběžných operací čistě proto, že SQL neuvaluje jakékoliv omezení na druh použitého algoritmu pro prohledávání tabulky. Smyčku *foreach* v C# je již obtížnější rozdělit na několik smyček nad různými částmi pole, které by pak bylo možné provádět souběžně na různých procesorech.

Další informace

Více informací o použití LINQ při paralelním vykonávání kódu naleznete v kapitole 13.

Deklarativní programování umí využít služby, které nabízejí kompilátory a exekuční prostředí, a je obecně snazší takový kód číst a udržovat. Tato prostá vlastnost LINQ je možná tou nejdůležitější, protože zvyšuje produktivitu programátorů. Předpokládejme například, že chcete získat seznam všech statických metod dostupných v aktuální aplikační doméně, které vracejí rozhraní *IEnumerable<T>*. Poslouží vám dotaz LINQ nad Reflection:

```
var query =
    from assembly in AppDomain.CurrentDomain.GetAssemblies()
```

```
from type in assembly.GetTypes()
from method in type.GetMethods()
where method.IsStatic
      && method.ReturnType.GetInterface( "IEnumerable`1" ) != null
orderby method.DeclaringType.Name, method.Name
group method by new { Class = method.DeclaringType.Name,
                      Method = method.Name };
```

Ekvivalentní kód C#, který zpracovává taková data, je delší, hůře se čte a patrně je náchylnější k chybám. Nepříliš optimalizovanou verzi vidíte ve výpisu 1.10.

Výpis 1.10 Kód C#, jenž odpovídá dotazu LINQ nad Reflection

```
List<String> results = new List<string>();
foreach( var assembly in AppDomain.CurrentDomain.GetAssemblies()) {
   foreach( var type in assembly.GetTypes() ) {
      foreach( var method in type.GetMethods()) {
         if (method.IsStatic &&
            method.ReturnType.GetInterface("IEnumerable`1") != null) {
            string fullName = String.Format( "{0}.{1}",
                                 method.DeclaringType.Name,
                                 method.Name );
            if (results.IndexOf( fullName ) < 0) {
               results.Add( fullName );
            }
         }
      }
   }
}
results.Sort();
```

Typová kontrola

Dalším významným aspektem jazykové integrace je typová kontrola. Kdykoliv v LINQ manipulujete s daty, není nutné žádné nezabezpečené přetypovávání. Krátká syntaxe dotazovacího výrazu nedělá v ověřování typů žádné kompromisy: data jsou vždy silně typová, včetně navrácených kolekcí i samostatných načítaných a vracených entit.

Typová kontrola v jazycích, které podporují LINQ (v současnosti C# a Visual Basic 2008), se zachovává i při použití specifických funkcí LINQ. To umožňuje používat funkce Visual Studia jako Microsoft IntelliSense a Refactoring dokonce i v dotazech LINQ. Tyto vymoženosti Visual Studia jsou dalším významným faktorem v produktivitě programátorů.

Přehlednost v různých typových systémech

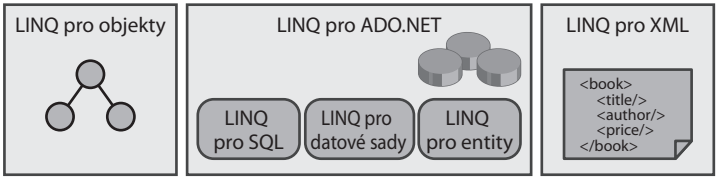
Když se podíváte na typový systém Microsoft .NET Frameworku a Microsoft SQL Serveru, zjistíte, že se odlišují. V LINQ dáváme přednost typovému systému .NET, protože tento systém podporují všechny jazyky umožňující dotazy LINQ. Ale většina vašich dat se bude ukládat do relační databáze a je potřebné převádět mnoho typů dat mezi těmito dvěma světy. LINQ tento převod provádí automaticky za vás, čímž se programátorovi téměř dokonale vyjasní rozdíly mezi typovými systémy.

Další informace

V možnostech převodu mezi různými typovými systémy a LINQ existují určitá omezení. Některé informace o tomto tématu naleznete v průběhu knihy a podrobnější tabulku kompatibilit typových systémů vám nabízí dokumentace produktu.

Implementace LINQ

LINQ je technologie, která zastřešuje mnoho datových zdrojů. Některé z těchto zdrojů jsou součástí implementací LINQ, které Microsoft poskytuje jako součást .NET Frameworku 3.5 (viz obrázek 1.1), jenž dále obsahuje LINQ pro entity (ten by měl být vydán v roce 2008).



Obrázek 1.1 Implementace LINQ, které Microsoft poskytuje jako součást .NET 3.5

Každá z těchto implementací je definována pomocí množiny rozšiřujících metod obsahujících operátory potřebné pro to, aby LINQ mohl pracovat s konkrétním datovým zdrojem. K těmto funkcím se přistupuje prostřednictvím významných jmenných prostorů.

LINQ pro objekty

LINQ pro objekty slouží pro manipulaci s kolekcemi objektů, které lze navzájem provázat a vytvořit tak graf. Z určitého úhlu pohledu je LINQ pro objekty výchozí implementací, kterou používá dotaz LINQ. LINQ pro objekty lze zpřístupnit pomocí jmenného prostoru *System.Linq*.

Další informace

V kapitole 2 jsou vysvětleny základní principy LINQ pomocí LINQ pro objekty jako referenční implementace.

Bylo by chybou se domnívat, že dotazy LINQ pro objekty se omezují na kolekce dat generovaných uživatelem. Nesprávnost tohoto předpokladu pochopíte při pohledu na výpis 1.11, který ukazuje dotaz LINQ nad informací získanou ze souborového systému. Seznam všech souborů v daném adresáři se načítá do paměti a poté filtruje v dotazu LINQ.

Výpis 1.11 Dotaz LINQ, který načítá dočasné soubory větší než 10 000 bajtů a řadí je podle velikosti

```
string tempPath = Path.GetTempPath();
DirectoryInfo dirInfo = new DirectoryInfo( tempPath );
var query =
    from f in dirInfo.GetFiles()
```

```
    where f.Length > 10000
    orderby f.Length descending
    select f;
```

LINQ pro ADO.NET

LINQ pro ADO.NET obsahuje různé implementace LINQ, které pracují s relačními daty. Obsahuje také další technologie, jež jsou specifické pro jednotlivé trvalé vrstvy:

- **LINQ pro SQL** Obsluhuje mapování mezi vlastními typy v .NET a schématem fyzické tabulky.
- **LINQ pro Entity** V mnoha směrech se podobá LINQ pro SQL. Ale namísto práce s fyzickou databází jako trvalou vrstvou používá konceptuální datový model entit (Entity Data Model, EDM). Výsledkem je abstraktní vrstva, která je nezávislá na fyzické datové vrstvě.
- **LINQ pro datové sady** Umožňuje v LINQ dotazy do objektu typu *DataSet*.

LINQ pro SQL a LINQ pro entity se v mnohém podobají, protože oba typy přistupují k relační databázi a pracují s entitami objektů v paměti, které reprezentují externí data. Hlavním rozdílem je, že operují na odlišné úrovni abstrakce. Zatímco LINQ pro SQL se váže na fyzickou strukturu databáze, LINQ pro entity pracuje nad konceptuálním modelem (obchodní entity), jenž se může od fyzické struktury (tabulky v databázi) výrazně odlišovat.

Důvodem pro tyto odlišné možnosti přístupu k relačním datům v LINQ je skutečnost, že pro přístup k databázi se dnes používají různé modely. Některé organizace provozují veškerý přístup přes uložené procedury včetně všech dotazů do databáze a vůbec nepoužívají dynamické dotazy. Mnoho dalších využívá uložené procedury na vkládání, aktualizace či mazání dat a pro dotazy vytváří dynamicky příkazy SELECT. Někteří chápou databázi jako jednoduchý objekt v trvalé vrstvě, zatímco další do databáze vkládají určitou obchodní logiku prostřednictvím spouští a uložených procedur. Jazyk LINQ se snaží nabídnout pomoc a vylepšení přístupu k databázi, aniž by někoho nutil přijímat jediný všeobsahující model.

Další informace

Použití jakékoliv implementace LINQ pro ADO.NET vyžaduje vložit do programu příslušný jmenný prostor. Jednotlivé implementace LINQ pro ADO.NET a související podrobnosti popisuje kapitola 4, kapitola 5, kapitola 7, „LINQ pro datové sady", a kapitola 8.

LINQ pro XML

LINQ pro XML obsahuje poněkud odlišnou syntaxi, která pracuje s daty XML, a umožňuje dotazy a práci s daty. Obzvláště významnou podporu LINQ pro XML nabízí Visual Basic 2008, jehož integrální součástí jsou výrazy XML. Tato rozšířená podpora zjednodušuje kód potřebný pro manipulaci s daty. Ve Visual Basicu 2008 můžete napsat takovýto dotaz:

```
Dim book = _
   <Book Title="Programování LINQ">
      <%= From person In team _
         Where person.Role = "Author" _
         Select <Author><%= person.Name %></Author> %>
   </Book>
```

Odpovídající dotaz v C# 3.0 by měl takovýto tvar:

```
dim book =
   new XElement( "Book",
      new XAttribute( "Title", "Programování LINQ" ),
      from person in team
      where person.Role == "Author"
      select new XElement( "Author", person.Name ) );
```

Další informace

Podrobné informace o LINQ pro XML naleznete v kapitole 9 a 10. Ostatní detaily o syntaxi Visual Basicu 2008 obsahuje příloha C.

Souhrn

V této kapitole jsme si představili jazyk LINQ a ukázali si, jak funguje. Probrali jsme, jak se lze dotazovat do rozmanitých datových zdrojů a jak s nimi lze pracovat prostřednictvím jednotné syntaxe, jež je integrální součástí současných hlavních programovacích jazyků, C# a Visual Basic. Podívali jsme se na výhody jazykové integrace, včetně deklarativního programování, typové kontroly a transparentnosti napříč různými typovými systémy. V krátkosti jsme se seznámili s implementacemi LINQ v prostředí .NET 3.5 – LINQ pro objekty, LINQ pro ADO.NET a LINQ pro XML – ve zbytku knihy se jimi budeme zabývat podrobněji.

KAPITOLA 2

Základy syntaxe LINQ

Integrovaný jazyk pro dotazování (Language Integrated Query, LINQ) umožňuje vývojářům dotazovat se a spravovat sekvence položek (objekty, entity, databázové záznamy, uzly XML atd.) v jejich softwarovém řešení pomocí běžné syntaxe a jednoho programovacího jazyka bez ohledu na charakter zpracovávaných položek. Klíčovou vlastností LINQ je jeho integrace v běžně používaných programovacích jazycích, kterou umožňuje používání společné syntaxe pro všechny druhy obsahu.

Jak jsme si řekli v kapitole 1, „Úvod do LINQ", LINQ poskytuje základní infrastrukturu pro mnoho odlišných implementací dotazovacích nástrojů, kupříkladu LINQ pro objekty, LINQ pro SQL, LINQ pro datové sady, LINQ pro entity, LINQ pro XML atd. Všechna tato dotazovací rozšíření vycházejí ze specializovaných rozšiřujících metod a sdílejí stejnou sadu klíčových slov pro dotazovací výrazy, které si probereme v této kapitole.

Než se podrobně podíváme na jednotlivá klíčová slova, projdeme si různé aspekty jednoduchého dotazu LINQ a ukážeme si základní elementy syntaxe LINQ.

Dotazy LINQ

LINQ vychází z množiny dotazovacích operátorů definovaných jako rozšiřující metody, které pracují s jakýmkoliv objektem implementujícím rozhraní *IEnumerable<T>* nebo *IQueryable<T>*.

Další informace

Více podrobností o rozšiřujících metodách naleznete v příloze B, „C# 3.0: Nové funkce jazyka" a v příloze C, „Visual Basic 2008: Nové funkce jazyka".

Tento přístup dělá z LINQ obecné dotazovací prostředí, protože rozhraní *IEnumerable<T>* či *IQueryable<T>* implementuje mnoho kolekcí či typů a jakýkoliv vývojář si může napsat svou vlastní implementací. Tato dotazovací infrastruktura je rovněž hojně rozšiřitelná, což si ukážeme v kapitole 12, „Rozšíření LINQ". S danou architekturou rozšiřujících metod mohou vývojáři specializovat chování metody na základě typu dat, na která se dotazují. Například LINQ pro SQL i LINQ pro XML mají specializované operátory LINQ, které umí zpracovávat relační data resp. uzly XML.

Syntaxe dotazu

Na začátku popisu syntaxe si uveďme krátký příklad. Přestavte si, že se pomocí LINQ pro objekty potřebujete zeptat do pole objektů typu *Developer* a získat jména vývojářů, kteří používají C# jako svůj hlavní programovací jazyk. Kód, který byste mohli použít, vidíte ve výpisu 2.1.

Výpis 2.1 Jednoduchý dotazovací výraz v C# 3.0

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

public class Developer {
  public string Name;
  public string Language;
  public int Age;
}

class App {
  static void Main() {
    Developer[] developers = new Developer[] {
      new Developer {Name = "Paolo", Language = "C#"},
      new Developer {Name = "Marco", Language = "C#"},
      new Developer {Name = "Frank", Language = "VB.NET"}};

    var developersUsingCSharp =
      from d in developers
      where d.Language == "C#"
      select d.Name;

    foreach (var item in developersUsingCSharp) {
      Console.WriteLine(item);
    }
  }
}
```

Výsledkem spuštění kódu budou jména *Paolo* a *Marco*.

Ve Visual Basicu 2008 bude mít tentýž dotaz nad stejným typem *Developer* tvar podle výpisu 2.2.

Výpis 2.2 Jednoduchý dotazovací výraz ve Visual Basicu 2008

```
Imports System
Imports System.Linq
Imports System.Collections.Generic

Public Class Developer
  Public Name As String
  Public Language As String
  Public Age As Integer
End Class
Module App
  Sub Main()

    Dim developers As New Developer() { _
      New Developer With {.Name = "Paolo", .Language = "C#"}, _
```

```
      New Developer With {.Name = "Marco", .Language = "C#"}, _
      New Developer With {.Name = "Frank", .Language = "VB.NET"}}

    Dim developersUsingCSharp = _
      From d In developers _
      Where d.Language = "C#" _
      Select d.Name

    For Each item in developersUsingCSharp
      Console.WriteLine(item)
    Next
  End Sub
End Module
```

Syntaxe dotazů (ve výpisech 2.1 a 2.2 tučně) se nazývá *dotazovací výraz*. V některých implementacích LINQ se reprezentaci těchto dotazů v paměti říká *strom výrazu*. Dotazovací výraz pracuje s jedním či více zdroji informací a aplikuje na ně jeden nebo více dotazovacích operátorů ze skupiny standardních dotazovacích operátorů nebo operátorů specifických pro danou doménu. Vyhodnocení dotazovacího výrazu obecně vrací řadu hodnot. Dotazovací výraz se vyhodnocuje až ve chvíli, kdy se vyčísluje jeho obsah. Další podrobnosti o dotazovacích výrazech a stromech výrazů naleznete v kapitole 11, „Uvnitř stromů výrazů".

Poznámka

V následujících ukázkách budeme kvůli zjednodušení používat pouze syntaxi v jazyce C# 3.0. Jak ale vidíte, verze výše uvedené ukázky v jazyce Visual Basic 2008 je velice podobná kódu v C# 3.0.

Uvedené dotazy vypadají podobně jako příkaz SQL, ale jejich styl se poněkud odlišuje. Ukázkový výraz, který jsme si definovali, se skládá z příkazu výběru:

```
select d.Name
```

aplikovaného na množinu položek:

```
from d in developers
```

kde klauzule *from* míří na jakoukoliv instanci třídy, která implementuje rozhraní *IEnumerable<T>*. Na výběr se aplikuje konkrétní podmínka filtrování:

```
where d.Language == "C#"
```

Tyto klauzule se překládají v kompilátorech jazyků do volání rozšiřujících metod, které se sekvenčním způsobem aplikují na cíl dotazu. Hlavní knihovna LINQ, definovaná v knihovně System.Core.dll, definuje množinu rozšiřujících metod seskupených podle cíle a účelu. Knihovna například obsahuje třídu s názvem *Enumerable* definovanou ve jmenném prostoru *System.Linq*, která definuje rozšiřující metody aplikovatelné na instance typů implementujících rozhraní *IEnumerable<T>*.

Podmínka filtru (*where*) z našeho ukázkového dotazu se překládá na volání rozšiřující metody *Where* třídy *Enumerable*. Tato metoda má dvě přetížení a obě přebírají delegáta pro funkci *predicate*, který popisuje filtrovací podmínku, jež se má ověřit během členění návratových dat. V našem případě je filtrovací predikát generickým delegátem, který přebírá element typu *T*,

jenž odpovídá typu instancí uložených ve filtrovaném výčtu. Delegát vrací logickou hodnotu, která určuje, zdali má být daná položka součástí filtrované výstupní množiny.

```
public static IEnumerable<T> Where<T>(
   this IEnumerable<T> source,
   Func<T, bool> predicate);
```

Jak je vidět ze záhlaví metody, lze ji volat s libovolným typem, který implementuje rozhraní *IEnumerable<T>*, a proto ji můžeme následujícím způsobem zavolat s naším polem *developers*:

```
var filteredDevelopers = developers.Where(delegate (Developer d) {
    return (d.Language == "C#");
});
```

Parametr *predicate*, předávaný do metody *Where*, zde reprezentuje anonymního delegáta funkce, který se volá pro každou položku typu *Developer* přebranou ze zdrojové datové množiny (*developers*). Výsledkem volání metody *Where* je podmnožina všech položek: všechny, které vyhovují podmínce predikátu.

V C# 3.0 a Visual Basicu 2008 lze anonymního delegáta definovat snáze pomocí výrazu lambda. S použitím výrazu lambda můžeme naši filtrovací podmínku přepsat do kompaktnější podoby:

```
var filteredDevelopers = developers.Where(d => d.Language == "C#");
```


Důležité

Podrobnosti o syntaxi rozšiřujících metod, výrazech lambda, anonymních delegátech i dalších tématech se dočtete v příloze B a příloze C.

Výraz *select* je také rozšiřující metoda (nazvaná *Select*) třídy *Enumerable*. Zde je záhlaví metody *Select*:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
   this IEnumerable<TSource> source,
   Func<TSource, TResult> selector);
```

Parametr *selector* je projekcí, která vrací výčet objektů typu *TResult* získaných z množiny zdrojových objektů typu *TSource*. Stejně jako dříve můžete tuto metodu pomocí výrazu lambda aplikovat na celou kolekci *developers*. Nebo ji lze volat pro kolekci filtrovanou v programovacím jazyce (s názvem *filteredDevelopers*), protože jde stále o typ disponující rozhraním *IEnumerable<T>*:

```
var csharpDevelopersNames = filteredDevelopers.Select(d => d.Name);
```

Na základě právě popsaných příkazů můžeme nyní přepsat ukázkový dotaz, aniž bychom použili syntaxi dotazovacího výrazu:

```
IEnumerable<string> developersUsingCSharp =
   developers
   .Where(d => d.Language == "C#")
   .Select(d => d.Name);
```

Metody *Where* i *Select* přebírají v parametrech výrazy lambda. Tyto výrazy lambda se překládají na predikáty a projekce vycházející z generických typů delegátů, definovaných ve jmenném prostoru *System* v sestavení System.Core.dll.

Uveďme si celou sestavu dostupných generických typů delegátů. Mnoho rozšiřujících metod třídy *Enumerable* může tyto delegáty přebírat v parametrech a budeme je používat v ukázkách v celé této kapitole.

```
public delegate TResult Func< TResult >();
public delegate TResult Func< T, TResult >( T arg );
public delegate TResult Func< T1, T2, TResult > (T1 arg1, T2 arg2 );
public delegate TResult Func< T1, T2, T3, TResult >
   ( T1 arg1, T2 arg2, T3 arg3 );
public delegate TResult Func< T1, T2, T3, T4, TResult >
   (T1 arg1, T2 arg2, T3 arg3, T4 arg4 );
```

Konečná verze našeho počátečního dotazu by mohla vypadat jako výpis 2.3.

Výpis 2.3 Počáteční dotazovací výraz přepsaný do základních elementů

```
Func<Developer, bool> filteringPredicate = d => d.Language == "C#";
Func<Developer, string> selectionPredicate = d => d.Name;
IEnumerable<string> developersUsingCSharp =
  developers
  .Where(filteringPredicate)
  .Select(selectionPredicate);
```

Kompilátor C# 3.0, podobně jako kompilátor Visual Basicu 2008, překládá dotazovací výrazy LINQ (výpis 2.1 a 2.2) do výrazu podobného příkazu z výpisu 2.3. Poté, co si na syntaxi dotazovacích výrazů (výpis 2.1 a 2.2) zvyknete, bude pro vás snazší a jednodušší psát a pracovat v ní, i když je nepovinná a stále máte možnost používat odpovídající rozvláčnější verzi z výpisu 2.3. Nicméně občas je nutné zavolat rozšiřující metodu přímo, protože syntaxe dotazovacích výrazů nepokrývá všechny rozšiřující metody.

> **Důležité**
>
> V kapitole 3, „LINQ pro objekty", si podrobněji popíšeme všechny rozšiřující metody, které jsou dostupné ve třídě *Enumerable* definované ve jmenném prostoru *System.Linq*.

Plná syntaxe dotazů

V předchozí části kapitoly jsme si ukázali jednoduchý dotaz nad seznamem objektů. Syntaxe dotazovacích výrazů je však složitější a členitější než uvedená ukázka a nabízí mnoho různých klíčových slov, které vyhoví většině běžných dotazovacích potřeb. Každý dotaz začíná klauzulí *from* a končí buď klauzulí *select*, nebo klauzulí *group*. Důvod, proč se začíná klauzulí *from* namísto *select*, jako je tomu v syntaxi SQL, souvisí (spolu s dalšími technickými důvody) s poskytováním funkčnosti Microsoft IntelliSense ve zbytku dotazu, která usnadňuje psaní podmínek, výběrů a dalších klauzulí výrazu. Klauzule *select* promítá výsledky výrazu do výčtového objektu. Klauzule *group* promítá výsledky výrazu do množiny skupin podle podmínky sdružování a každá skupina je výčtový objekt. Následující kód představuje vzor plné syntaxe dotazovacího výrazu:

```
query-expression ::= from-clause query-body

query-body ::=
join-clause*
(from-clause join-clause* | let-clause | where-clause)*
orderby-clause?
(select-clause | groupby-clause)
   query-continuation?

from-clause ::= from itemName in srcExpr

select-clause ::= select selExpr

groupby-clause ::= group selExpr by keyExpr
```

Za první klauzulí *from* mohou (ale nemusejí) být další klauzule *from*, *let* nebo *where*. Klauzule *let* přiřadí výsledku výrazu název, což může být užitečné v situaci, kdy se potřebujete v dotazu vícekrát odkazovat na tentýž výraz:

```
let-clause ::= let itemName = selExpr
```

Klauzule *where*, jak jsme si již řekli, definuje filtr, s jehož pomocí se do výsledku dostanou jen konkrétní řádky.

```
where-clause ::= where predExpr
```

Každá klauzule *from* generuje místní „proměnnou intervalu“, která odpovídá všem položkám ve zdrojové sekvenci, na něž se aplikují dotazovací operátory (například rozšiřující metody *System.Linq.Enumerable*).

Za klauzulí *from* může následovat libovolný počet klauzulí *join*. Před závěrečnou klauzulí *select* či *group* může stát klauzule *orderby*, která výsledky seřadí:

```
join-clause ::=
join itemName in srcExpr on keyExpr equals keyExpr
(into itemName)?

orderby-clause ::= orderby (keyExpr (ascending | descending)?)*

query-continuation ::= into itemName query-body
```

S ukázkami dotazovacích výrazů se budete setkávat v celé této knize. Kdykoliv si budete potřebovat ověřit syntaxi některého elementu, můžete se podívat sem do uvedeného popisu.

Klíčová slova v dotazech

V následujících pasážích si podrobněji popíšeme různá klíčová slova, která syntaxe dotazovacích výrazů nabízí.

Klauzule from

Prvním klíčovým slovem je *from*. Definuje datový zdroj dotazu či poddotazu a proměnnou intervalu, jež určuje všechny jednotlivé elementy ve zdroji, na něž má dotaz směřovat. Datovým zdrojem může být libovolná instance nějakého typu, jenž obsahuje rozhraní *IEnumerable*, *IEnumerable<T>* nebo *IQueryable<T>*, které implementuje rozhraní *IEnumerable<T>*. V následujícím kousku kódu vidíte ukázkový příkaz v C# 3.0, jenž tuto klauzuli používá:

```
from rangeVariable in dataSource
```

Kompilátor jazyka odvodí typ proměnné intervalu z typu datového zdroje. Jestliže je například datový zdroj typu *IEnumerable<Developer>*, proměnná intervalu bude typu *Developer*. V případech, kdy nepoužíváte silně typový datový zdroj – například pole typu *ArrayList* s objekty typu *Developer*, které obsahuje rozhraní *IEnumerable* – měli byste proměnné intervalu explicitně stanovit typ. Ve výpisu 2.4 vidíte ukázku takového dotazu s explicitní deklarací typu *Developer* pro proměnnou intervalu s názvem *d*.

Výpis 2.4 Dotazovací výraz pro negenerický datový zdroj, s deklarací typu pro proměnnou intervalu

```
ArrayList developers = new ArrayList();
developers.Add(new Developer { Name = "Paolo", Language = "C#" });
developers.Add(new Developer { Name = "Marco", Language = "C#" });
developers.Add(new Developer { Name = "Frank", Language = "VB.NET" });

var developersUsingCSharp =
   from Developer d in developers
   where d.Language == "C#"
   select d.Name;

foreach (string item in developersUsingCSharp) {
   Console.WriteLine(item);
}
```

V uvedeném příkladě je přetypování povinné, jinak se nepodaří dotaz zkompilovat, protože překladač nemůže automaticky odvodit typ proměnné intervalu a nebude moci ošetřit přístup ke členům *Language* a *Name* v tomtéž dotazu.

V dotazech lze použít vícero klauzulí *from* a spojovat tak různé datové zdroje. V C# 3.0 vyžaduje každý datový zdroj deklaraci v klauzuli *from*, což dokládá výpis 2.5, kde svazujeme zákazníky s jejich objednávkami. Všimněte si prosím, že vztah mezi třídami *Customer* a *Order* je fyzicky definován pomocí pole *Orders* typu *Order* v každé instanci třídy *Customer*.

Důležité

Když používáte více klauzulí *from*, „spojovací podmínku" určuje struktura dat a odlišuje se od principu propojování v relační databázi. (Proto je potřeba v dotazovacím výrazu použít klauzuli *join*, což si ukážeme později v této kapitole.)

Výpis 2.5 Dotazovací výraz v C# 3.0 s propojením dvou datových zdrojů

```
public class Customer {
  public String Name { get; set; }
  public String City { get; set; }
  public Order[] Orders { get; set; }
}

public class Order {
  public Int32 IdOrder { get; set; }
  public Decimal EuroAmount { get; set; }
  public String Description { get; set; }
}
```

```
// ... kód vynechán ...

static void queryWithJoin() {
  Customer[] customers = new Customer[] {
    new Customer { Name = "Paolo", City = "Brescia",
      Orders = new Order[] {
        new Order { IdOrder = 1, EuroAmount = 100, Description = "Pořadí 1" },
        new Order { IdOrder = 2, EuroAmount = 150, Description = "Pořadí 2"   },
        new Order { IdOrder = 3, EuroAmount = 230, Description = "Pořadí 3" },
}},
    new Customer { Name = "Marco", City = "Torino",
      Orders = new Order[] {
        new Order { IdOrder = 4, EuroAmount = 320, Description = "Pořadí 4" },
        new Order { IdOrder = 5, EuroAmount = 170, Description = "Pořadí 5" },
      }};

  var ordersQuery =
    from c in customers
    from o in c.Orders
    select new { c.Name, o.IdOrder, o.EuroAmount };

  foreach (var item in ordersQuery) {
    Console.WriteLine(item);
  }
}
```

Ve Visual Basicu 2008 lze v klauzuli *from* definovat více datových zdrojů oddělených čárkami, což vidíte ve výpisu 2.6.

Výpis 2.6 Dotazovací výraz ve Visual Basicu 2008 s propojením dvou datových zdrojů

```
Dim customers As Customer() = { _
  New Customer With {.Name = "Paolo", .City = "Brescia", _
    .Orders = New Order() { _
      New Order With {.IdOrder = 1, .EuroAmount = 100, .Description = "Pořadí 1"}, _
      New Order With {.IdOrder = 2, .EuroAmount = 150, .Description = "Pořadí 2"}, _
      New Order With {.IdOrder = 3, .EuroAmount = 230, .Description = "Pořadí 3"} _
}}, _
  New Customer With {.Name = "Marco", .City = "Torino", _
    .Orders = New Order() { _
      New Order With {.IdOrder = 4, .EuroAmount = 320, .Description = "Pořadí 4"}, _
      New Order With {.IdOrder = 5, .EuroAmount = 170, .Description = "Pořadí 5"} _
}}}

Dim ordersQuery = _
  From c In customers, _
      o In c.Orders _
  Select c.Name, o.IdOrder, o.EuroAmount

For Each item In ordersQuery
  Console.WriteLine(item)
Next
```

O propojování pohovoříme později v této kapitole.

Klauzule where

Jak již víte, klauzule *where* udává filtrovací podmínku, jež se aplikuje na datový zdroj. Predikát vyhodnotí logickou podmínku pro každou položku v datovém zdroji a vybere pouze ty

položky, kde má podmínka hodnotu *true*. V jednom dotazu lze mít více klauzulí *where* nebo klauzuli *where* s více predikáty, které je možné kombinovat pomocí logických operátorů (&&, || a *!* v C# 3.0 resp. *And*, *Or*, *AndAlso*, *OrElse*, *Is* a *IsNot* ve Visual Basicu 2008). Ve Visual Basicu 2008 může být v predikátu libovolný výraz vracející logickou hodnotu, takže lze použít i číselné výrazy, jež budou považovány za true v situaci, kdy nebudou rovny nule.

Podívejte se na výpis 2.7, v němž pomocí klauzule *where* vybíráme všechny objednávky s hodnotou *EuroAmount* větší než 200 Euro.

Výpis 2.7 Dotazovací výraz v C# 3.0 s klauzulí where

```
var ordersQuery =
  from c in customers
  from o in c.Orders
  where o.EuroAmount > 200
  select new { c.Name, o.IdOrder, o.EuroAmount };
```

Ve výpisu 2.8 vidíte odpovídající syntaxi v jazyce Visual Basic 2008.

Výpis 2.8 Dotazovací výraz ve Visual Basicu 2008 s klauzulí where

```
Dim ordersQuery = _
  From c In customers, _
           o In c.Orders _
  Where o.EuroAmount > 200 _
  Select c.Name, o.IdOrder, o.EuroAmount
```

Klauzule Select

Klauzule *select* určuje, jak bude vypadat výstup z dotazu. Jde o projekci, která určuje, co se má vybírat z výsledků všech předchozích klauzulí a výrazů. Ve Visual Basicu 2008 není klauzule *Select* povinná. Není-li zadána, vrací dotaz typ vycházející z proměnné intervalu stanovené pro aktuální oblast působnosti dotazu. Ve výpisech 2.7 a 2.8 jsme použili klauzuli *select* k projekci anonymních typů složených z vlastností či členů proměnných intervalů v dané oblasti. Při srovnání syntaxe C# 3.0 (výpis 2.7) a Visual Basicu 2008 (výpis 2.8) je vidět, že druhý zápis se v klauzuli *select* více podobá příkazu SQL, zatímco první vypadá spíše jako syntaxe programovacího jazyka. V C# 3.0 musíte explicitně deklarovat svůj záměr vytvořit instanci nového anonymního typu, zatímco ve Visual Basicu 2008 je jazyková syntaxe jednodušší a vnitřní procesy ukrývá.

Klauzule Group a Into

Klauzuli *group* lze použít na seskupení výsledků podle klíče. Je možné ji použít jako alternativu ke klauzuli *from* a umožňuje vám pracovat s klíči s jednou hodnotou i více hodnotami. Ve výpisu 2.9 vidíte ukázku dotazu, která seskupuje vývojáře podle programovacího jazyka.

Výpis 2.9 Dotazovací výraz v C# 3.0, který seskupuje vývojáře podle programovacího jazyka

```
Developer[] developers = new Developer[] {
  new Developer { Name = "Paolo", Language = "C#" },
  new Developer { Name = "Marco", Language = "C#" },
  new Developer { Name = "Frank", Language = "VB.NET" },
};
```

```
var developersGroupedByLanguage =
  from d in developers
  group d by d.Language;

foreach (var group in developersGroupedByLanguage) {
  Console.WriteLine("Jazyk: {0}", group.Key);
  foreach (var item in group) {
    Console.WriteLine("\t{0}", item.Name);
  }
}
```

Výstup ukázky kódu z výpisu 2.9 bude vypadat následovně:

```
Jazyk: C#
    Paolo
    Marco
Jazyk: VB.NET
    Frank
```

Jak v příkladu vidíte, výsledkem dotazu je výčet skupin označených klíčem a složených z podřízených položek. Ve skutečnosti vypisujeme pro každou skupinu ve výsledku dotazu na obrazovku vlastnost *Key* a poté procházíme položky v každé skupině a vypisujeme jejich hodnoty. Jak jsme si již řekli, položky lze sdružovat pomocí klíče s více hodnotami, který používá anonymní typy. Ukázku vidíte ve výpisu 2.10, kde sdružujeme vývojáře podle jazyka a věkové skupiny.

Výpis 2.10 Dotazovací výraz v C# 3.0, který seskupuje vývojáře podle programovacího jazyka a věkové skupiny

```
Developer[] developers = new Developer[] {
  new Developer { Name = "Paolo", Language = "C#", Age = 32 },
  new Developer { Name = "Marco", Language = "C#", Age = 37},
  new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
};

var developersGroupedByLanguage =
  from d in developers
  group d by new { d.Language, AgeCluster = (d.Age / 10) * 10 };

foreach (var group in developersGroupedByLanguage) {
  Console.WriteLine("Jazyk: {0}", group.Key);
  foreach (var item in group) {
    Console.WriteLine("\t{0}", item.Name);
  }
}
```

Tentokrát bude výstup z kódu ve výpisu 2.10 vypadat takto:

```
Jazyk: { Language = C#, AgeCluster = 30 }
    Paolo
    Marco
Jazyk: { Language = VB.NET, AgeCluster = 40 }
    Frank
```

V tomto příkladě je klíčem pro každou skupinu anonymní typ, definovaný dvěma vlastnostmi: *Language* a *AgeCluster*.

Visual Basic 2008 rovněž podporuje sdružování výsledků a používá k tomu klauzuli *Group By*. Ve výpisu 2.11 vidíte ukázku dotazu, který je ekvivalentem kódu ve výpisu 2.9.

Výpis 2.11 Dotazovací výraz ve Visual Basicu 2008, který seskupuje vývojáře podle programovacího jazyka

```
Dim developers As Developer() = { _
  New Developer With {.Name = "Paolo", .Language = "C#", .Age = 32}, _
  New Developer With {.Name = "Marco", .Language = "C#", .Age = 37}, _
  New Developer With {.Name = "Frank", .Language = "VB.NET", .Age = 48}}

Dim developersGroupedByLanguage = _
  From d In developers _
  Group d By d.Language Into Group _
  Select Language, Group

For Each group In developersGroupedByLanguage
  Console.WriteLine("Jazyk: {0}", group.Language)
  For Each item In group.Group
    Console.WriteLine(" {0}", item.Name)
  Next
Next
```


Syntaxe Visual Basicu 2008 je poněkud složitější než odpovídající syntaxe v C# 3.0. Ve Visual Basicu 2008 musíte sdružování promítat pomocí klauzule *Into* do nově vytvořeného objektu *Group* a poté explicitně deklarovat, co se má vybírat. Ale výsledek sdružování se vypisuje snáze, protože hodnota *Key* si uchovává svůj název (*Language*).

I C# 3.0 nabízí klauzuli *into*, jež je vhodná pro spolupráci s klíčovým slovem *group*, i když její použití není povinné. Klíčové slovo *into* se používá na ukládání výsledků příkazů *select*, *group* či *join* do dočasné proměnné. Tuto konstrukci můžete použít, když potřebujete provést nad výsledky dodatečné dotazy. Vzhledem k tomuto chování se tomuto klíčovému slovu říká rovněž klauzule *continuation*. Ve výpisu 2.12 máte ukázku dotazovacího výrazu v C# 3.0, který využívá klauzuli *into*.

Výpis 2.12 Dotazovací výraz v C# 3.0, který používá klauzuli into

```
var developersGroupedByLanguage =
  from d in developers
  group d by d.Language into developersGrouped
  select new {
    Language = developersGrouped.Key,
    DevelopersCount = developersGrouped.Count()
  };

foreach (var group in developersGroupedByLanguage) {
  Console.WriteLine ("Jazyk {0} obsahuje {1} vývojářů",
    group.Language, group.DevelopersCount);
}
```

Klauzule Orderby

Klauzule *orderby*, jak již název napovídá, umožňuje vzestupně či sestupně třídit výsledky dotazu. Řazení lze provádět pomocí jednoho či více klíčů, které kombinují různé směry řazení. Výpis 2.13 ukazuje příklad dotazu, který vrací objednávky zákazníků seřazené podle hodnoty *EuroAmount*.

Výpis 2.13 Dotazovací výraz v C# 3.0 s klauzulí orderby

```
var ordersSortedByEuroAmount =
  from c in customers
  from o in c.Orders
  orderby o.EuroAmount
  select new { c.Name, o.IdOrder, o.EuroAmount };
```

Výpis 2.14 ukazuje příklad dotazu, jenž vybírá objednávky a řadí je podle jména zákazníka a sestupně podle hodnoty *EuroAmount*.

Výpis 2.14 Dotazovací výraz v C# 3.0 s klauzulí orderby a vícenásobnou podmínkou řazení

```
var ordersSortedByCustomerAndEuroAmount =
  from c in customers
  from o in c.Orders
  orderby c.Name, o.EuroAmount descending
  select new { c.Name, o.IdOrder, o.EuroAmount };
```

Ve výpisu 2.15 vidíte odpovídající dotaz v jazyce Visual Basic 2008.

Výpis 2.15 Dotazovací výraz ve Visual Basicu 2008 s klauzulí orderby a vícenásobnou podmínkou řazení

```
Dim ordersSortedByCustomerAndEuroAmount = _
  From c In customers, _
      o In c.Orders _
  Order By c.Name, o.EuroAmount Descending _
Select c.Name, o.IdOrder, o.EuroAmount
```

V tomto případě mají oba jazyky velice podobnou syntaxi.

Klauzule Join

Klíčové slovo *join* umožňuje spojovat různé datové zdroje na základě členů zdrojů, u nichž lze zjišťovat rovnost. Pracuje to podobně jako vnitřní spojování tabulek v SQL. Nemůžete porovnávat položky pomocí operátorů „větší než“, „menší než“ či „není rovno“. Srovnávání lze provádět pouze pomocí speciálního slova *equals*, jež má odlišné chování od operátoru ==, protože záleží na pořadí operandů. U *equals* představuje levý klíč vnější sekvenci zdrojů a pravý klíč vnitřní zdroj. Vnější zdroj je platný pouze na levé straně výrazu *equals* a vnitřní zdroj pouze na pravé straně. Zde tento princip vidíte v pseudo-kódu:

```
join-clause ::= join innerItem in innerSequence on outerKey equals innerKey
```

Pomocí klauzule *join* můžete definovat vnitřní spojení, skupinové spojení a levé vnější spojení. Vnitřní spojení vrací prosté výsledné mapování elementů vnějšího datového zdroje na příslušný vnitřní datový zdroj. Vynechává elementy ve vnějším datovém zdroji, které nemají odpovídající elementy ve vnitřním zdroji. Výpis 2.16 ukazuje prostý dotaz s vnitřním spojením mezi kategoriemi produktů a odpovídajícími produkty.

Výpis 2.16 Dotazovací výraz v C# 3.0 s vnitřním spojením

```
public class Category {
  public Int32 IdCategory { get; set; }
  public String Name { get; set; }
}
```

```
public class Product {
  public String IdProduct { get; set; }
  public Int32 IdCategory { get; set; }
  public String Description { get; set; }
}

// ... kód vynechán ...

Category[] categories = new Category[] {
  new Category { IdCategory = 1, Name = "Pasta"},
  new Category { IdCategory = 2, Name = "Beverages"},
  new Category { IdCategory = 3, Name = "Other food"},
};

Product[] products = new Product[] {
  new Product { IdProduct = "PASTA01", IdCategory = 1, Description = "Tortellini" },
  new Product { IdProduct = "PASTA02", IdCategory = 1, Description = "Spaghetti" },
  new Product { IdProduct = "PASTA03", IdCategory = 1, Description = "Fusilli" },
  new Product { IdProduct = "BEV01", IdCategory = 2, Description = "Water" },
  new Product { IdProduct = "BEV02", IdCategory = 2, Description = "Orange Juice" },
};

var categoriesAndProducts =
  from c in categories
  join p in products on c.IdCategory equals p.IdCategory
  select new {
    c.IdCategory,
    CategoryName = c.Name,
    Product = p.Description
};

foreach (var item in categoriesAndProducts) {
  Console.WriteLine(item);
}
```

Výstup této ukázky kódu vypadá zhruba jako následující výpis. Všimněte si, že chybí kategorie „Other food“, protože v ní neexistují žádné produkty.

```
{ IdCategory = 1, CategoryName = Pasta, Product = Tortellini }
{ IdCategory = 1, CategoryName = Pasta, Product = Spaghetti }
{ IdCategory = 1, CategoryName = Pasta, Product = Fusilli }
{ IdCategory = 2, CategoryName = Beverages, Product = Water }
{ IdCategory = 2, CategoryName = Beverages, Product = Orange Juice }
```

Skupinové spojení produkuje hierarchickou výsledkovou množinu, v níž se sdružují elementy vnitřní sekvence s odpovídajícími elementy vnější sekvence. V případech, kdy element vnější sekvence postrádá odpovídající elementy vnitřní sekvence, propojí se vnější element na prázdné pole. Skupinové spojení nemá ekvivalentní syntaxi v relačním SQL, protože jde o hierarchický výstup. Ve výpisu 2.17 vidíte ukázku takového dotazu. (Rozšířenou formu tohoto typu dotazu uvidíte v kapitole 3.)

Výpis 2.17 Dotazovací výraz v C# 3.0 se skupinovým spojením

```
var categoriesAndProducts =
  from c in categories
  join p in products on c.IdCategory equals p.IdCategory
    into productsByCategory
  select new {
    c.IdCategory,
```

```
    CategoryName = c.Name,
    Products = productsByCategory
  };

foreach (var category in categoriesAndProducts) {
  Console.WriteLine("{0} - {1}", category.IdCategory,  category.CategoryName);
  foreach (var product in category.Products) {
    Console.WriteLine("\t{0}", product.Description);
  }
}
```

Všimněte si, že ve výstupu je i kategorie „Other food“, i když je prázdná:

```
1 - Pasta
    Tortellini
    Spaghetti
    Fusilli
2 - Beverages
    Water
    Orange Juice
3 - Other food
```

Visual Basic 2008 nabízí pro definici skupinového spojení v dotazech specifické klíčové slovo *Group Join*.

Levé vnější spojení vrací jednoduchou výsledkovou množinu obsahující všechny elementy ve vnějším zdroji, i když nemají odpovídající element ve vnitřním zdroji. Abyste získali takový výsledek, musíte použít rozšiřující metodu *DefaultEmpty*, která vrací v případě absence hodnoty v datovém zdroji výchozí hodnotu. O této i mnoha dalších rozšiřujících metodách si podrobně povíme v kapitole 3. Ve výpisu 2.18 vidíte ukázku podobné syntaxe.

Výpis 2.18 Dotazovací výraz v C# 3.0 s levým vnějším spojením

```
var categoriesAndProducts =
  from c in categories
  join p in products on c.IdCategory equals p.IdCategory
    into productsByCategory
  from pc in productsByCategory.DefaultIfEmpty(
    new Product {
      IdProduct = String.Empty,
      Description = String.Empty,
      IdCategory = 0})
  select new {
    c.IdCategory,
    CategoryName = c.Name,
    Product = pc.Description
  };

foreach (var item in categoriesAndProducts) {
  Console.WriteLine(item);
}
```

Tento příklad nám dává následující výstup na obrazovku:

```
{ IdCategory = 1, CategoryName = Pasta, Product = Tortellini }
{ IdCategory = 1, CategoryName = Pasta, Product = Spaghetti }
{ IdCategory = 1, CategoryName = Pasta, Product = Fusilli }
{ IdCategory = 2, CategoryName = Beverages, Product = Water }
{ IdCategory = 2, CategoryName = Beverages, Product = Orange Juice }
{ IdCategory = 3, CategoryName = Other food, Product = }
```

Všimněte si, že je zde kategorie „Other food“ s prázdným produktem, který je dílem metody *DefaultEmpty*.

Posledním bodem v otázce spojovací klauzule, který je nutno zdůraznit, je fakt, že můžete elementy porovnávat pomocí složených klíčů. Stačí zkrátka vytvořit anonymní typy, jak jsme si to ukázali u klíčového slova *group*. Jestliže máte například složený klíč u kategorie, jenž se skládá z položek *IdCategory* a *Year*, můžete psát následující příkaz s anonymním typem v podmínce *equals*:

```
from c in categories
join p in products
   on new { c.IdCategory, c.Year } equals new { p.IdCategory, p.Year }
   into productsByCategory
```


Jak jste již v této kapitole viděli, je také možné spojovat zdroje pomocí vícenásobných klauzulí *from*, což je užitečné v situaci, kdy potřebujete definovat dotazy se spojením, kde nestačí pouze rovnost.

Visual Basic 2008 nabízí syntaxi podobnou jazyku C# 3.0, ale má také některé zkratky pro rychlejší definici spojení. Implicitní spojení lze definovat pomocí vícenásobných klauzulí *In* v příkazu *From* a podmínek porovnávání v klauzuli *Where*. Ve výpisu 2.19 vidíte ukázku takové syntaxe.

Výpis 2.19 Implicitní spojení ve Visual Basicu 2008

```
Dim categoriesAndProducts = _
  From c In categories, p In products _
  Where c.IdCategory = p.IdCategory _
  Select c.IdCategory, CategoryName = c.Name, Product = p.Description

For Each item In categoriesAndProducts
  Console.WriteLine(item)
Next
```

Ve výpisu 2.20 vidíte tentýž dotaz zapsaný pomocí běžné syntaxe s *join*.

Výpis 2.20 Explicitní spojení ve Visual Basicu 2008

```
Dim categoriesAndProducts = _
  From c In categories Join p In products _
            On p.IdCategory Equals c.IdCategory _
  Select c.IdCategory, CategoryName = c.Name, Product = p.Description
```

Všimněte si, že ve Visual Basicu 2008 nehraje pořadí elementů v porovnávání roli, protože překladač si je uspořádá po svém, díky čemuž je syntaxe uvolněnější, podobně jako u klasického relačního SQL.

Klauzule Let

Klauzule *let* umožňuje ukládat výsledky poddotazu do proměnné, kterou pak můžete použít někde jinde v dotazu. Tato klauzule je užitečná ve chvíli, kdy potřebujete použít jeden výraz v dotazu vícekrát a nechcete jej psát samostatně pro každé místo, kde se má použít. Pomocí klauzule *let* nadefinujete pro tento výraz novou proměnnou intervalu, na niž se budete v dotazu odkazovat. Jakmile má proměnná intervalu přiřazenu hodnotu z klauzule *let*, není možné ji měnit. Ale jestliže je v proměnné intervalu typ, na nějž se lze dotazovat, můžete provádět

dotazy. Ve výpisu 2.21 vidíte ukázku použití této klauzule pro výběr stejných kategorií produktů s počtem produktů, s řazením podle počtu.

Výpis 2.21 Ukázka použití klauzule let v C# 3.0

```
var categoriesByProductsNumberQuery =
  from c in categories
  join p in products on c.IdCategory equals p.IdCategory
    into productsByCategory
  let ProductsCount = productsByCategory.Count()
  orderby ProductsCount
  select new { c.IdCategory, ProductsCount};

foreach (var item in categoriesByProductsNumberQuery) {
  Console.WriteLine(item);
}
```

Výstup z ukázky vypadá takto:

```
{ IdCategory = 3, ProductsCount = 0 }
{ IdCategory = 2, ProductsCount = 2 }
{ IdCategory = 1, ProductsCount = 3 }
```

Visual Basic 2008 používá velmi podobnou syntaxi jako C# 3.0 a umožňuje definovat v téže klauzuli *let* vícenásobná přejmenování oddělená čárkami.

Další klíčová slova ve Visual Basicu 2008

Visual Basic 2008 obsahuje i další klíčová slova pro dotazovací výrazy, jež jsou v C# 3.0 dostupná pouze prostřednictvím rozšiřujících metod. Tato klíčová slova popisuje následující seznam:

- *Aggregate*, vhodné pro aplikaci agregační funkce na datový zdroj. Je možné je použít namísto klauzule *From* na zahájení nového dotazu.
- *Distinct*, lze použít pro odstranění duplicitních hodnot ve výsledcích dotazu.
- *Skip*, slouží pro přeskočení prvních *N* elementů ve výsledku dotazu.
- *Skip While*, přeskočí první elementy dotazu, které vyhovují zadané podmínce.
- *Take*, slouží k převzetí prvních *N* elementů z výsledku dotazu.
- *Take While*, lze použít k převzetí prvních elementů výsledku dotazu, které odpovídají zadané podmínce.

Příkazy *Skip* a *Take* nebo *Skip While* a *Take While* lze použít společně na stránkování výsledků. K tomuto tématu se vrátíme v některých ukázkách v kapitole 3.

Další informace o syntaxi dotazů

Do této chvíle jsme si ukázali všechna klíčová slova, která jsou v programovacích jazycích dostupná. Ale nezapomeňte, že každý dotazovací výraz se při kompilaci mění na volání odpovídajících rozšiřujících metod. Kdykoliv se potřebujete pomocí LINQ ptát na datový zdroj a neexistuje klíčové slovo pro konkrétní operaci v dotazu, můžete použít nativní nebo vlastní rozšiřující metody přímo ve spojení se syntaxí dotazu. Jestliže používáte pouze rozšiřující metody (což ukazuje výpis 2.3), syntaxe se nazývá *syntaxe s metodami*. Když použijete syntaxi dotazů společně s rozšiřujícími metodami (viz výpis 2.17), výsledku se říká *smíšená syntaxe dotazu*.

Odložené vyhodnocení dotazu a rozeznávání rozšiřujících metod

V této části kapitoly se podíváme na dva aspekty chování dotazovacího výrazu: odložené vyhodnocení dotazu a rozeznávání rozšiřujících metod. Oba tyto principy jsou významné ve všech implementacích LINQ.

Odložené vyhodnocení dotazu

Dotazovací výraz se nevyhodnocuje ve chvíli, kdy jej definujete, ale v okamžiku použití. Podívejte se na výpis 2.22.

Výpis 2.22 Ukázkový dotaz LINQ nad množinou vývojářů

```
List<Developer> developers = new List<Developer>(new Developer[] {
  new Developer { Name = "Paolo", Language = "C#", Age = 32 },
  new Developer { Name = "Marco", Language = "C#", Age = 37},
  new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
});

var query =
  from d in developers
  where d.Language == "C#"
  select new { d.Name, d.Age };

Console.WriteLine("Je zde {0} vývojářů v C#.", query.Count());
```

Tento kód deklaruje velmi jednoduchý dotaz, jenž obsahuje pouze dvě odpovídající položky, což lze vidět buď v kódu, který deklaruje seznam vývojářů, nebo ve výstupu na obrazovce, kde se vypisuje výstupní hodnota rozšiřující metody *Count*.

```
Je zde 2 vývojářů v C#.
```

Nyní si představte, že chcete změnit obsah zdrojové sekvence a přidat novou instanci objektu *Developer* – po nadefinování proměnné *query* (viz výpis 2.23).

Výpis 2.23 Ukázka změny množiny vývojářů, na něž se ptáme

```
developers.Add(new Developer {
  Name = "Roberto", Language = "C#", Age = 35 });

Console.WriteLine("Je zde {0} vývojářů v C#.", query.Count());
```

Když nyní po přidání nového vývojáře vyhodnotíme opětovně dotaz nebo jen ověříme počet položek, což provádí výpis 2.23, bude výsledkem hodnota tři. Vývojář, kterého jsme přidali, bude součástí výsledku, i když jsme jej přidali až po nadefinování dotazu.

Důvodem pro toto chování je logická úvaha, že dotazovací výraz popisuje svého druhu „plán dotazování". Provede se až ve chvíli, kdy jej skutečně používáme, a bude se provádět stále znovu v okamžicích spuštění. Některé implementace LINQ – například LINQ pro objekty – implementují toto chování pomocí delegátů. Další – například LINQ pro SQL – mohou používat stromy výrazů, které využívají rozhraní *IQueryable<T>*. Tomuto způsobu chování

říkáme *odložené vyhodnocení dotazu* a jde o základní princip v LINQ, bez ohledu na typ používané implementace LINQ.

Odložené vyhodnocení výrazu je užitečnou věcí, neboť můžete nadefinovat dotazy jedenkrát a použít je vícekrát: jestliže se změní zdrojová sekvence, výsledek se vždy aktualizuje podle posledního obsahu. Ale zamyslete se nad situací, v níž potřebujete snímek výsledku v určitém „bezpečném bodě", chcete jej používat vícekrát a předejít opakovanému provádění z důvodů úspory výkonu či kvůli nezávislosti na změnách ve zdroji. Je potřeba založit kopii výsledku, k čemuž vám poslouží množina tzv. převodních operátorů (například *ToArray*, *ToList*, *ToDictionary*, *ToLookup*), které jsou speciálně vyhrazeny pro tyto účely. O převodních operátorech pohovoříme v kapitole 3.

Rozeznávání rozšiřujících metod

Rozeznávání rozšiřujících metod je jedním z nejdůležitějších principů, které je potřeba pochopit, máte-li zvládnout práci s LINQ. Podívejte se na kód ve výpisu 2.24, v němž definujeme vlastní seznam typu *Developer* (s názvem *Developers*) a třídu *DevelopersExtension*, která obsahuje rozšiřující metodu s názvem *Where*, jež slouží výhradně pro instance typu *Developers*.

Výpis 2.24 Ukázkový kód, který mění množinu vývojářů, na něž se ptáme.

```
public sealed class Developers : List<Developer> {
  public Developers(IEnumerable<Developer> items) : base(items) { }
}

public static class DevelopersExtension {
  public static IEnumerable<Developer> Where(
    this Developers source, Func<Developer, bool> predicate) {

    Console.WriteLine("Došlo k volání rozšiřující metody Where pro objekt typu
                      Developers");
    return (source.AsEnumerable().Where(predicate));
  }

  public static IEnumerable<Developer> Where(
    this Developers source,
    Func<Developer, int, bool> predicate) {

    Console.WriteLine("Došlo k volání rozšiřující metody pro objekt typu Developers");
    return (source.AsEnumerable().Where(predicate));
  }
}
```

Jedinou prací, kterou v našich rozšiřujících metodách *Where* provádíme, je výpis na konzolu, který nás informuje o tom, že daná metoda byla provedena. Poté předáme požadavek do rozšiřujících metod *Where* definovaných pro veškeré běžné instance typu *IEnumerable<T>* a zdroj převádíme pomocí metody *AsEnumerable*, o níž pojednává kapitola 3.

Použijeme-li naše obvyklé pole *developers*, chování dotazu ve výpisu 2.25 bude velmi zajímavé.

Výpis 2.25 Dotazovací výraz nad vlastním seznamem vývojářů

```
Developers developers = new Developers(new Developer[] {
  new Developer { Name = "Paolo", Language = "C#", Age = 32 },
```

```
  new Developer { Name = "Marco", Language = "C#", Age = 37},
  new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
});

var query =
  from d in developers
  where d.Language == "C#"
  select d;

Console.WriteLine("Je zde {0} vývojářů v C#.", query.Count());
```

Kompilátor převede dotazovací výraz na následující kód, což jsme viděli na začátku této kapitoly:

```
Var expert =
   developers
   .Where (d => d.Language == "C#")
   .Select(d => d);
```

Výsledkem přítomnosti třídy *DevelopersExtension* je, že rozšiřující metodou *Where* je metoda definovaná ve třídě *DevelopersExtension* namísto obecné metody ve třídě *System.Linq. Enumerable*. (Aby byla třída *DeveloperExtension* chápána jako kontejner pro rozšiřující metody, musí být deklarována jako statická a definována v aktuálním jmenném prostoru či v jakémkoliv jmenném prostoru načteném pomocí direktivy *using*.) Výsledný kód, který produkuje rozeznávání rozšiřujících metod v kompilátoru, má tento tvar:

```
var expr =
   Enumerable.Select(
      DevelopersExtension.Where(
         developers,
         d => d.Language == "C#"),
      d => d );
```

Ve výsledku nakonec vždy voláme statické metody statické třídy, ale syntaxe potřebná při práci s rozšiřujícími metodami je jednodušší a intuitivnější než při používání rozvláčného explicitního volání statických metod.

Nyní vidíme skutečnou sílu LINQ. Při používání rozšiřujících metod můžeme nadefinovat vlastní chování pro konkrétní typy dat. V následujících kapitolách probereme LINQ pro SQL, LINQ pro XML a další implementace LINQ. Tyto implementace jsou pouze konkrétními implementacemi dotazovacích operátorů, za což vděčíme rozeznávání rozšiřujících metod v kompilátoru.

V tuto chvíli vypadá vše skvěle. Ale představte si, že se potřebujete dotázat do svého seznamu vývojářů pomocí standardní rozšiřující metody *Where*, a nikoliv prostřednictvím specializované metody. K přesměrování rozeznávání rozšiřujících metod v kompilátoru je v takovém případě potřeba převést svůj seznam do obecnější podoby. Toto je další typ scénáře, který může těžit z převodních operátorů, což si ukážeme v kapitole 3.

Několik závěrečných úvah o dotazech LINQ

V poslední části kapitoly si ukážeme několik podrobností o degenerovaných dotazovacích výrazech a zpracování výjimek.

Degenerované dotazovací výrazy

Někdy je potřeba postupně procházet elementy datového zdroje bez jakéhokoliv filtrování, řazení, sdružování či vlastních projekcí. Podívejte se kupříkladu na dotaz ve výpisu 2.26.

Výpis 2.26 Degenerovaný dotazovací výraz nad seznamem vývojářů

```
Developer[] developers = new Developer[] {
...
};

var query =
  from d in developers
  select d;

foreach (var developer in query) {
  Console.WriteLine(developer.Name);
}
```

V tomto kousku kódu procházíme v cyklu datový zdroj, takže se patrně podivujete, proč nepoužijeme datový zdroj přímo jako ve výpisu 2.27.

Výpis 2.27 Procházení seznamu vývojářů

```
Developer[] developers = new Developer[] {
...
};

foreach (var developer in developers) {
  Console.WriteLine(developer.Name);
}
```

Výsledky z výpisů 2.26 a 2.27 budou evidentně totožné. Ale ve výpisu 2.26 nám dotazovací výraz zajistí, že jestliže pro daný datový zdroj existuje konkrétní rozšiřující metoda *Select*, dojde k jejímu zavolání a výsledek bude konzistentně výsledkem překladu dotazovacího výrazu na odpovídající volání metody.

Dotaz, který vrací stejný výsledek jako původní datový zdroj (tedy jeví se jako triviální či zbytečný), se nazývá *degenerovaný dotazovací výraz*. Na druhé straně cyklus přímo nad datovým zdrojem (výpis 2.27) překlenuje volání jakékoliv vlastní rozšiřující metody *Select* a negarantuje správné chování, pokud explicitně nepožadujete procházet data bez použití LINQ.

Zpracování výjimek

Dotazovací výrazy se mohou ve své definici odkazovat na externí metody. Někdy mohou tyto metody selhat. Prohlédněte si dotaz definovaný ve výpisu 2.28, kde pro každou položku v datovém zdroji voláme metodu *DoSomething*.

Výpis 2.28 Dotazovací výraz v C# 3.0 s externí metodou, která způsobuje fiktivní výjimku

```
static Boolean DoSomething(Developer dev) {
  if (dev.Age > 40)
    throw new ArgumentOutOfRangeException("dev");

  return (dev.Language == "C#");
}
```

```
static void Main() {
  Developer[] developers = new Developer[] {
    ...
    new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
  };

  var query =
    from d in developers
    let SomethingResult = DoSomething(d)
    select new { d.Name, SomethingResult };

  foreach (var item in query) {
    Console.WriteLine(item);
  }
}
```

Metoda *DoSomething* vrací fiktivní výjimku pro každého vývojáře staršího než 40 let. Tuto metodu voláme z dotazu. Během provádění dotazu dojde i na vývojáře Franka, kterému je 48 let, a naše vlastní metoda způsobí výjimku.

Nejprve byste měli pečlivě uvážit, zdali potřebujete v definicích dotazů volat vlastní metody, protože je to nebezpečný zvyk, což dokládá uvedená ukázka. Ale v případech, kdy se rozhodnete volat externí metody, nejlepším způsobem práce s nimi je zabalit volání dotazu do bloku *try ... catch*. Jak jste viděli v části zvané „Odložené vyhodnocení dotazu", dotaz se spouští pokaždé při jeho používání, a nikoli při definici. Správný způsob zápisu kódu z výpisu 2.28 je uveden ve výpisu 2.29.

Výpis 2.29 Dotazovací výraz v C# 3.0 s ošetřením výjimek

```
Developer[] developers = new Developer[] {
  ...
  new Developer { Name = "Frank", Language = "VB.NET", Age = 48 },
};

var query =
  from d in developers
  let SomethingResult = DoSomething(d)
  select new { d.Name, SomethingResult };

try {
  foreach (var item in query) {
    Console.WriteLine(item);
  }
}
catch (ArgumentOutOfRangeException e) {
  Console.WriteLine(e.Message);
}
```

Obecně je vkládání definici dotazovacího výrazu do bloku *try ... catch* zbytečné. Navíc byste se ze stejného důvodu měli vyhnout přímému používání výsledků metod či konstruktorů jako datových zdrojů pro dotazovací výrazy a namísto toho přiřadit jejich výsledky do proměnných a toto přiřazení vložit do bloku *try ... catch*, což dokládá výpis 2.30.

Výpis 2.30 Dotazovací výraz v C# 3.0 s ošetřením výjimek v deklaraci lokálních proměnných

```
static void queryWithExceptionHandledInDataSourceDefinition() {
  Developer[] developers = null;

  try {
    developers = createDevelopersDataSource();
  }
  catch (InvalidOperationException e) {
    // Představme si, že createDevelopersDataSource
    // vygeneruje v případě selhání výjimku InvalidOperationException

    // nějak ji ošetříme...
    Console.WriteLine(e.Message);
  }

  if (developers != null)
  {
    var query =
      from d in developers

      let SomethingResult = DoSomething(d)
      select new { d.Name, SomethingResult };

    try {
        foreach (var item in query) {
            Console.WriteLine(item);
        }
    }
    catch (ArgumentOutOfRangeException e) {
      Console.WriteLine(e.Message);
    }
  }
}
private static Developer[] createDevelopersDataSource() {
  // vygenerování fiktivní výjimky InvalidOperationException
  throw new InvalidOperationException();
}
```

Souhrn

V této kapitole jsme probrali principy dotazovacích výrazů a odlišné typy jejich syntaxe (syntaxe dotazů, syntaxe s metodami a smíšená syntaxe), stejně jako hlavní klíčová slova pro dotazy, která jsou dostupná v jazycích C# 3.0 a Visual Basic 2008. Probrali jsme dvě významné vlastnosti LINQ: odložené vyhodnocení dotazu a rozeznávání rozšiřujících metod. Ukázali jsme vám také degenerované dotazovací výrazy a způsob zpracování výjimek při vyčíslování dotazovacích výrazů. V následující kapitole se podíváme podrobně na LINQ pro objekty.

KAPITOLA 3

LINQ pro objekty

Moderní programovací jazyky a systémy vývoje softwaru stavějí stále více na objektově orientovaných principech a vývoji. Výsledkem je, že velmi často musíme spravovat a dotazovat se objektů či kolekcí namísto záznamů a tabulek. Potřebujeme rovněž nástroje a jazyky nezávislé na konkrétních datových zdrojích či vrstvách pro práci s perzistencí dat. LINQ pro objekty je hlavní implementací LINQ, kterou lze použít pro dotazování do kolekcí objektů, entit a položek v paměti.

V této kapitole si popíšeme hlavní třídy a operátory, na nichž je systém LINQ postaven, a využijeme je k pochopení architektury systému a k výuce syntaxe. Ukázky v této kapitole používají LINQ pro objekty takovým způsobem, abychom se mohli soustředit na dotazy a operátory.

Ukázková data pro příklady

Je nutné definovat si určitá data, která budeme používat v příkladech v této kapitole. Použijeme množinu *customers*, zákazníků, a každý z nich bude mít objednané produkty, *products*. Následující kód definuje tyto datové typy v kódu C# 3.0.

```
public enum Countries {
  USA,
  Italy,
}

public class Customer {
  public string Name;
  public string City;
  public Countries Country;
  public Order[] Orders;

  public override string ToString() {
    return String.Format("Name: {0} - City: {1} - Country: {2}",
      this.Name, this.City, this.Country );
  }
}

public class Order {
  public int IdOrder;
  public int Quantity;
  public bool Shipped;
  public string Month;
  public int IdProduct;
```

```
  public override string ToString() {
    return String.Format( "IdOrder: {0} - IdProduct: {1} - " +
      "Quantity: {2} - Shipped: {3} - " +
      "Month: {4}", this.IdOrder, this.IdProduct,
        this.Quantity, this.Shipped, this.Month);
  }
}

public class Product {
  public int IdProduct;
  public decimal Price;

  public override string ToString() {
    return String.Format("IdProduct: {0} - Price: {1}", this.IdProduct,
                         this.Price );
  }
}
```

Následující ukázka kódu inicializuje několik instancí těchto typů:

```
// ------------------------------------------------------
// Inicializace kolekce zákazníků a jejich objednávek:
// ------------------------------------------------------
customers = new Customer[] {
  new Customer {Name = "Paolo", City = "Brescia",
        Country = Countries.Italy, Orders = new Order[] {
          new Order { IdOrder = 1, Quantity = 3, IdProduct = 1 ,
                      Shipped = false, Month = "January"},
          new Order { IdOrder = 2, Quantity = 5, IdProduct = 2 ,
                      Shipped = true, Month = "May"}}},
  new Customer {Name = "Marco", City = "Torino",
        Country = Countries.Italy, Orders = new Order[] {
          new Order { IdOrder = 3, Quantity = 10, IdProduct = 1 ,
                      Shipped = false, Month = "July"},
          new Order { IdOrder = 4, Quantity = 20, IdProduct = 3 ,
                      Shipped = true, Month = "December"}}},
  new Customer {Name = "James", City = "Dallas",
        Country = Countries.USA, Orders = new Order[] {
          new Order { IdOrder = 5, Quantity = 20, IdProduct = 3 ,
                      Shipped = true, Month = "December"}}},
  new Customer {Name = "Frank", City = "Seattle",
        Country = Countries.USA, Orders = new Order[] {
          new Order { IdOrder = 6, Quantity = 20, IdProduct = 5 ,
                      Shipped = false, Month = "July"}}}};

products = new Product[] {
  new Product {IdProduct = 1, Price = 10 },
  new Product {IdProduct = 2, Price = 20 },
  new Product {IdProduct = 3, Price = 30 },
  new Product {IdProduct = 4, Price = 40 },
  new Product {IdProduct = 5, Price = 50 },
  new Product {IdProduct = 6, Price = 60 }};
```

Odpovídající pomocný kód ve Visual Basicu 2008 vypadá takto:

```
Public Enum Countries
  USA
  Italy
End Enum
```

```
Public Class Customer
  Public Name As String
  Public City As String
  Public Country As Countries
  Public Orders As Order()
  Public Overrides Function ToString() As String
    Return String.Format("Name: {0} - City: {1} - Country: {2}",
      Me.Name, Me.City, Me.Country)
  End Function
End Class

Public Class Order
  Public IdOrder As Integer
  Public Quantity As Integer
  Public Shipped As Boolean
  Public Month As String
  Public IdProduct As Integer

  Public Overrides Function ToString() As String
    Return String.Format ( _
      "IdOrder: {0} - IdProduct: {1} - " & _
      "Quantity: {2} - Shipped: {3} - " & _
      "Month: {4}", Me.IdOrder, Me.IdProduct, _
      Me.Quantity, Me.Shipped, Me.Month)
  End Function
End Class

Public Class Product
  Public IdProduct As Integer
  Public Price As Decimal

  Public Overrides Function ToString() As String
    Return String.Format("IdProduct: {0} - Price: {1}", Me.IdProduct,
      Me.Price)
  End Function
End Class
```

A zde máte odpovídající inicializační kód ve Visual Basicu 2008:

```
' ------------------------------------------------------
' Inicializace kolekce zákazníků a jejich objednávek:
' ------------------------------------------------------

customers = New Customer() { _
  New Customer With {.Name = "Paolo", .City = "Brescia", _
      .Country = Countries.Italy, .Orders = New Order() { _
        New Order With {.IdOrder = 1, .Quantity = 3, .IdProduct = 1, _
            .Shipped = False, .Month = "January"}, _
        New Order With {.IdOrder = 2, .Quantity = 5, .IdProduct = 2, _
            .Shipped = True, .Month = "May"}}}, _
  New Customer With {.Name = "Marco", .City = "Torino", _
      .Country = Countries.Italy, .Orders = New Order() { _
        New Order With {.IdOrder = 3, .Quantity = 10, .IdProduct = 1, _
            .Shipped = False, .Month = "July"}, _
        New Order With {.IdOrder = 4, .Quantity = 20, .IdProduct = 3, _
            .Shipped = True, .Month = "December"}}}, _
  New Customer With {.Name = "James", .City = "Dallas", _
      .Country = Countries.USA, .Orders = New Order() { _
```

```
          New Order With {.IdOrder = 5, .Quantity = 20, .IdProduct = 3, _
              .Shipped = True, .Month = "December"}}}, _
    New Customer With {.Name = "Frank", .City = "Seattle", _
        .Country = Countries.USA, .Orders = New Order() { _
          New Order With {.IdOrder = 6, .Quantity = 20, .IdProduct = 5, _
              .Shipped = False, .Month = "July"}}}}

  products = New Product() { _
    New Product With {.IdProduct = 1, .Price = 10}, _
    New Product With {.IdProduct = 2, .Price = 20}, _
    New Product With {.IdProduct = 3, .Price = 30}, _
    New Product With {.IdProduct = 4, .Price = 40}, _
    New Product With {.IdProduct = 5, .Price = 50}, _
    New Product With {.IdProduct = 6, .Price = 60}}
```

Dotazovací operátory

V této sekci si popíšeme hlavní metody a generické delegáty, které nabízí jmenný prostor *System.Linq*, jenž je součástí knihovny System.Core.dll a slouží k dotazování pomocí LINQ.

Operátor Where

Představte si, že potřebujete seznam jmen a měst zákazníků z Itálie. K filtrování množiny záznamů vám poslouží operátor *Where*, jenž se rovněž nazývá operátor restrikce, neboť omezuje množinu záznamů. Výpis 3.1 představuje jednoduchou ukázku.

Výpis 3.1 Dotaz s restrikcí

```
var expr =
  from c in customers
  where c.Country == Countries.Italy
  select new { c.Name, c.City };
```

Uveďme si záhlaví operátoru *Where*:

```
public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);

public static IEnumerable<TSource> Where<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Int32, Boolean> predicate);
```

Jak vidíte, jsou zde dvě možnosti. Ve výpisu 3.1 jsme použili první verzi, která vyčísluje položky ze *zdrojové* sekvence a vybírá ty, které odpovídají podmínce (*c.Country == Countries.Italy*). Druhá verze přebírá pro podmínku dodatečný parametr typu *Int32*. Tento parametr slouží jako index elementů ve zdrojové sekvenci, počínající nulou. Mějte na paměti, že jestliže do podmínek předáte prázdné argumenty, dojde k chybě *ArgumentNullException*. Index slouží k zahájení filtrování od určitého indexu, což dokládá výpis 3.2.

Výpis 3.2 Dotaz s restrikcí a filtrováním pomocí indexu

```
var expr =
  customers
  .Where((c, index) => (c.Country == Countries.Italy && index >= 1))
  .Select(c => c.Name);
```

> **Důležité**
>
> Ve výpisu 3.2 používáme syntaxi s metodou, protože verze *Where*, kterou chceme volat, nemá ekvivalentní výraz v dotazovacím výrazu. Od nynějška budeme používat obě syntaxe.

Výsledkem výpisu 3.2 bude seznam italských zákazníků s výjimkou prvního. Jak ukazuje následující výstup, dělení podle *indexu* se projeví nad datovým zdrojem, jenž je již filtrován podle země.

```
Marco
```

Možnost filtrování položek *zdrojové* sekvence pomocí indexu jejich pozice je užitečná v situaci, kdy potřebujete získat konkrétní stránku dat z velké řady položek. Výpis 3.3 je toho ukázkou.

Výpis 3.3 Dotaz se stránkováním

```
int start = 5;
int end = 10;

var expr =
  customers
  .Where((c, index) => ((index >= start) && (index < end)))
  .Select(c => c.Name);
```

Mějte na paměti, že obecně není dobré ukládat do paměti velké sekvence dat načtených z trvalé databázové vrstvy, takže v zásadě byste neměli muset data v paměti stránkovat. Obvykle je lepší stránkovat data na úrovni perzistentní vrstvy.

Projekční operátory

Následující pasáže popisují práci s projekčními operátory. Tyto operátory se používají na výběr („projekci“) obsahu zdroje dat do výsledné množiny.

Select

Ve výpisu 3.1 jste viděli ukázku definice výsledku dotazu pomocí operátoru *Select*. Záhlaví operátoru *Select* mají tento tvar:

```
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector);
public static IEnumerable<TResult> Select<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, Int32, TResult> selector);
```

Operátor *Select* je jedním z projekčních operátorů, neboť promítá výsledky dotazu a zpřístupňuje je prostřednictvím objektu s rozhraním *IEnumerable<TResult>*. Tento objekt vyčísluje položky nalezené predikátem *selector*. Podobně jako operátor *Where*, i *Select* vyhodnocuje *zdrojovou* sekvenci a získává výsledky z predikátu *selector*. Podívejte se na následující predikát:

```
var expr = customers.Select(c => c.Name);
```

Výsledkem tohoto zápisu bude sekvence jmen zákazníků (*IEnumerable*<*String*>). Nyní se podívejte na tento příklad:

```
var expr = customers.Select(c => new { c.Name, c.City });
```

Tento predikát promítá sekvenci instancí anonymního typu, definovaného jako dvojice *Name* a *City* pro každý objekt zákazníka. V druhém přetížení metody *Select* můžeme v predikátu také pracovat s parametrem typu *Int32*. Tento index, počínající nulou, slouží k definici pozice každé položky vložené do výsledné sekvence. Ve výpisu 3.4 máte ukázku použití této přetížené metody.

Výpis 3.4 Projekce s parametrem index v predikátu selector

```
var expr =
  customers
  .Select((c, index) => new { index, c.Name, c.Country } );

foreach (var item in expr) {
  Console.WriteLine(item);
}
```

Výsledek dotazu vypadá následovně:

```
{ index = 0, Name = Paolo, Country = Italy }
{ index = 1, Name = Marco, Country = Italy }
{ index = 2, Name = James, Country = USA }
{ index = 3, Name = Frank, Country = USA }
```

Stejně jako u operátoru *Where*, jednoduchá verze operátoru *Select* je dostupná jako klíčové slovo v dotazovacím výrazu, zatímco složitější verzi je nutné volat explicitně jako rozšiřující metodu.

Jak jste již viděli v kapitole 2, „Základy syntaxe LINQ“, syntaxe dotazovacího výrazu pro operátor *Select* se v jazycích C# 3.0 a Visual Basic 2008 trochu liší v otázce projekce anonymního typu. Ve Visual Basicu 2008 určuje vznik anonymního typu implicitně syntaxe dotazu, zatímco v C# 3.0 musíte explicitně deklarovat, že požadujete *nový* anonymní typ.

SelectMany

Představte si, že chcete vybrat všechny objednávky zákazníků z Itálie. Mohli byste napsat poněkud rozvláčný dotaz z výpisu 3.5.

Výpis 3.5 Seznam objednávek italských zákazníků

```
var orders =
  customers
  .Where(c => c.Country == Countries.Italy)
  .Select(c => c.Orders);

foreach(var item in orders) { Console.WriteLine(item); }
```

Vzhledem k chování operátoru *Select* bude výsledný typ dotazu roven *IEnumerable*<*Order[]*>, kde každá položka ve výsledné sekvenci reprezentuje pole objednávek jednoho zákazníka. Atribut *Orders* instance objektu *Customer* je totiž typu *Order[]*. Výstup kódu z výpisu 3.5 bude mít tuto podobu:

```
DevLeap.Linq.LinqToObjects.Operators.Order[]
DevLeap.Linq.LinqToObjects.Operators.Order[]
```

Chceme-li „jednoduchý“ typ výsledku, *IEnumerable<Order>*, musíme použít operátor *SelectMany*:

```
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TResult>> selector);
public static IEnumerable<TResult> SelectMany<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, Int32, IEnumerable<TResult>> selector);
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector);
public static IEnumerable<TResult> SelectMany<TSource, TCollection, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, Int32, IEnumerable<TCollection>> collectionSelector,
    Func<TSource, TCollection, TResult> resultSelector);
```

Tento operátor vyčísluje *zdrojovou* sekvenci a spojuje výsledky do jedné výčtové sekvence. Druhé přetížení tohoto operátoru, které máte k dispozici, je analogické k odpovídajícímu přetížení operátoru *Select* a umožňuje vkládat index začínající nulou. Ukázku vidíte ve výpisu 3.6.

Výpis 3.6 Jednoduchý seznam objednávek italských zákazníků

```
var orders =
  customers
  .Where(c => c.Country == Countries.Italy)
  .SelectMany(c => c.Orders);
```

S použitím syntaxe dotazovacích výrazů lze dotaz z výpisu 3.6 přepsat do podoby ve výpisu 3.7.

Výpis 3.7 Jednoduchý seznam objednávek italských zákazníků ve formě dotazovacího výrazu

```
var orders =
  from c in customers
  where c.Country == Countries.Italy
    from o in c.Orders
    select o;
```

Oba výstupy, 3.6 i 3.7, dávají následující výstup, v němž využíváme přepis metody *ToString* u typu *Order*.

```
IdOrder: 1 - IdProduct: 1 - Quantity: 3 - Shipped: False - Month: January
IdOrder: 2 - IdProduct: 2 - Quantity: 5 - Shipped: True - Month: May
IdOrder: 3 - IdProduct: 1 - Quantity: 10 - Shipped: False - Month: July
IdOrder: 4 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
```

Klíčové slovo *select* se v dotazovacím výrazu pro všechny klauzule *from* s výjimkou první překládá na volání metody *SelectMany*. Jiným slovy, kdykoliv uvidíte dotazovací výraz s více než jednou klauzulí *from*, můžete aplikovat toto pravidlo: *select* pro první klauzuli *from* se překládá na volání metody *Select*, další příkazy *Select* se překládají na volání metody *SelectMany*.

Třetí a čtvrté přetížení metody *SelectMany* je užitečné ve chvíli, kdy potřebujete vybírat výsledky ze zdrojové množiny sekvencí vlastním způsobem namísto prostého sloučení

položek, což dělají první dvě verze metody. Třetí a čtvrtá verze metody volá nad *zdrojovou* sekvencí projekci *collectionSelector* a vrací výsledek projekce *resultSelector*. Výsledek se aplikuje na každou položku v kolekci vybranou v projekci *collectionSelector* a v případě použití posledního uvedeného přetížení se nakonec použije index začínající nulou. Ve výpisu 3.8 vidíte ukázku použití třetího přetížení metody na výběr nového anonymního typu, vytvořeného z vlastností *Quantity* a *IdProduct* pro všechny objednávky italských zákazníků.

Výpis 3.8 Seznam Quantity a IdProduct pro objednávky od italských zákazníků

```
var items = customers
  .Where(c => c.Country == Countries.Italy)
  .SelectMany(c => c.Orders,
    (c, o) => new { o.Quantity, o.IdProduct });
```

Dotaz ve výpisu 3.8 lze přepsat do podoby dotazovacího výrazu, který vidíte ve výpisu 3.9.

Výpis 3.9 Seznam Quantity a IdProduct pro objednávky od italských zákazníků, zapsaný ve formě dotazovacího výrazu

```
var items =
  from c in customers
  where c.Country == Countries.Italy
    from o in c.Orders
    select new {o.Quantity, o.IdProduct};
```

Operátory řazení

Další užitečnou skupinu operátorů tvoří operátory řazení. Slouží k aplikaci řazení elementů a jeho směru ve výstupních sekvencích.

OrderBy a OrderByDescending

Někdy je vhodné na výsledky dotazu do databáze aplikovat řazení. LINQ umí výsledky dotazů řadit sestupně či vzestupně pomocí operátorů stejně jako syntaxe SQL. Jestliže například potřebujete vybrat jméno a město všech italských zákazníků a řadit je sestupně podle jména, můžete napsat příslušný dotazovací výraz, který vidíte ve výpisu 3.10.

Výpis 3.10 Dotazovací výraz se sestupným řazením

```
var expr =
  from c in customers
  where c.Country == Countries.Italy
  orderby c.Name descending
  select new { c.Name, c.City };
```

Syntaxe dotazovacího výrazu přeloží klíčové slovo *orderby* na jednu z následujících rozšiřujících metod řazení:

```
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);
public static IOrderedEnumerable<TSource> OrderBy<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer);
public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(
```

```
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);
public static IOrderedEnumerable<TSource> OrderByDescending<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer);
```

Jak vidíte, jsou zde dvě hlavní rozšiřující metody, *OrderBy* a *OrderByDescending*, které mají po dvou verzích. Názvy metod naznačují jejich použití: *OrderBy* slouží k vzestupnému řazení, *OrderByDescending* k sestupnému řazení. Parametr *keySelector* představuje funkci, která vybírá klíč typu *TKey* z každé položky typu *TSource*, načtené ze *zdrojové* sekvence. Vybraný klíč reprezentuje typový obsah, který se má během řazení porovnávat, a typ *TSource* představuje typ jednotlivých položek ve *zdrojové* sekvenci. Obě metody mají přetížení, které vám umožňuje vložit vlastní porovnávací třídu. Jestliže do parametru žádnou porovnávací třídu nevložíte nebo bude parametr *comparer* roven null, použije se vlastnost *Default* generického typu *Comparer<T>* (*Comparer<TKey>.Default*).

> **Důležité**
>
> **Výchozí porovnávací třída, *Comparer*, navrácený voláním *Comparer<T>.Default*, používá na porovnávání dvou objektů obecné rozhraní *IComparable<T>*. Jestliže typ *T* neobsahuje obecné rozhraní *System.IComparable<T>*, vrátí vlastnost *Default* typu *Comparer<T>* taková porovnávací třída, která používá rozhraní *System.IComparable*. Jestliže typ *T* neobsahuje implementaci ani jednoho z těchto dvou rozhraní, vyvolá volání metody *Compare* výchozí porovnávací třídy výjimku.**

Je nutné zdůraznit, že tyto metody řazení nevracejí pouze rozhraní *IEnumerable<TSource>*, ale *IOrderedEnumerable<TSource>*, což je rozhraní rozšiřující rozhraní *IEnumerable<T>*.

Dotazovací výraz ve výpisu 3.10 se přeloží na následující volání rozšiřujících metod:

```
var expr =
    customers
    .Where(c => c.Country == Countries.Italy)
    .OrderByDescending(c => c.Name)
    .Select(c => new { c.Name, c.City } );
```

Jak z této ukázky kódu vidíte, metoda *OrderByDescending*, stejně jako všechny ostatní metody řazení, přebírá výraz lambda, který vybírá hodnotu klíče z proměnné intervalu (*c*) v aktuálním kontextu. Selektor může vybrat libovolné pole pro řazení, které je dostupné v proměnné intervalu, dokonce i když jej metoda *Select* nepromítá do výstupu. Můžete například řadit zákazníky podle země a vybírat pouze jejich jména a města.

ThenBy a ThenByDescending

Kdykoliv potřebujete řadit data podle mnoha různých klíčů, máte k dispozici operátory *ThenBy* a *ThenByDescending*. Zde jsou jejich záhlaví:

```
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);
public static IOrderedEnumerable<TSource> ThenBy<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
```

```
    IComparer<TKey> comparer);
public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);
public static IOrderedEnumerable<TSource> ThenByDescending<TSource, TKey>(
    this IOrderedEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IComparer<TKey> comparer);
```

Forma těchto operátorů se podobá operátorům *OrderBy* a *OrderByDescending*. Rozdíl spočívá v tom, že *ThenBy* a *ThenByDescending* lze aplikovat pouze na rozhraní *IOrderedEnumerable<T>*, a nikoli na libovolné rozhraní *IEnumerable<T>*. Z tohoto důvodu lze operátory *ThenBy* a *ThenByDescending* používat až po aplikaci operátorů *OrderBy* a *OrderByDescending*. Ukázku máte zde:

```
var expr = customers
    .Where(c => c.Country == Countries.Italy)
    .OrderByDescending(c => c.Name)
    .ThenBy(c => c.City)
    .Select(c => new { c.Name, c.City } );
```

Ve výpisu 3.11 vidíte odpovídající dotazovací výraz.

Výpis 3.11 Dotazovací výraz s operátory orderby a thenby

```
var expr =
  from c in customers
  where c.Country == Countries.Italy
  orderby c.Name descending, c.City
  select new { c.Name, c.City };
```


Důležité

V případě, že se v sekvenci, která se má řadit, opakovaně objeví stejný klíč, není garantována „stabilita" výsledku. V takových případech nemusí porovnávací třída zachovat původní řazení.

V případě, že je potřeba položky ve vaší *zdrojové* sekvenci řadit podle vaší vlastní logiky, může do hry vstoupit vlastní porovnávací třída. Přestavte si například situaci, kdy chcete vybrat všechny objednávky vašich zákazníků seřazené podle měsíce, viz výpis 3.12:

Výpis 3.12 Dotazovací výraz s řazením pomocí porovnávače Comparer<T>.Default

```
var expr =
  from c in customers
    from o in c.Orders
    orderby o.Month
    select o;
```

Jestliže aplikujete výchozí porovnávací třídu na vlastnost *Month* vašich objednávek, dostanete výsledek seřazený podle abecedy, což je dáno chováním porovnávací třídy *Comparer<T>.Default*, jež jsme si popsali výše. Výsledek není správný, protože vlastnost *Month* je pouze řetězec, a nikoli číslo či datum:

```
IdOrder: 4 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
IdOrder: 5 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
```

```
IdOrder: 1 - IdProduct: 1 - Quantity: 3 - Shipped: False - Month: January
IdOrder: 3 - IdProduct: 1 - Quantity: 10 - Shipped: False - Month: July
IdOrder: 6 - IdProduct: 5 - Quantity: 20 - Shipped: False - Month: July
IdOrder: 2 - IdProduct: 2 - Quantity: 5 - Shipped: True - Month: May
```

Namísto toho je vhodné použít vlastní porovnávací třídu *MonthComparer*, jež měsíce řadí správně:

```
using System.Globalization;

class MonthComparer: IComparer<string> {
    public int Compare(string x, string y) {
        DateTime xDate = DateTime.ParseExact(x, "MMMM", new CultureInfo("en-US"));
        DateTime yDate = DateTime.ParseExact(y, "MMMM", new CultureInfo("en-US"));
        return(Comparer<DateTime>.Default.Compare(xDate, yDate)); } }
```

Nově definovaný porovnávač *MonthComparer* lze předat do volání rozšiřující metody *OrderBy*, což vidíte ve výpisu 3.13.

Výpis 3.13 Použití vlastního porovnávače v operátoru OrderBy

```
var orders =
  customers
  .SelectMany(c => c.Orders)
  .OrderBy(o => o.Month, new MonthComparer());
```

Nyní bude výsledek výpisu 3.13 následující, správně seřazený podle měsíců:

```
IdOrder: 1 - IdProduct: 1 - Quantity: 3 - Shipped: False - Month: January
IdOrder: 2 - IdProduct: 2 - Quantity: 5 - Shipped: True - Month: May
IdOrder: 3 - IdProduct: 1 - Quantity: 10 - Shipped: False - Month: July
IdOrder: 6 - IdProduct: 5 - Quantity: 20 - Shipped: False - Month: July
IdOrder: 4 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
IdOrder: 5 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
```

Operátor Reverse

Někdy je potřeba výsledek dotazu otočit a zobrazit poslední položku jako první. LINQ nabízí operátor *Reverse*, jenž tuto operaci umožňuje:

```
public static IEnumerable<TSource> Reverse<TSource>(
    this IEnumerable<TSource> source);
```

Implementace operátoru *Reverse* je velmi jednoduchá. Operátor pouze vypisuje všechny položky ze *zdrojové* sekvence v opačném pořadí. Ukázku předkládá výpis 3.14.

Výpis 3.14 Aplikace operátoru Reverse

```
var expr =
  customers
  .Where(c => c.Country == Countries.Italy)
  .OrderByDescending(c => c.Name)
  .ThenBy(c => c.City)
  .Select(c => new { c.Name, c.City } )
  .Reverse();
```

Operátor *Reverse*, podobně jako mnoho jiných operátorů, nemá odpovídající klíčové slovo v dotazovacích výrazech. Ale syntaxi dotazovacích výrazů lze s operátory míchat (jak jsme si ukázali v kapitole 2), což vidíte ve výpisu 3.15.

Výpis 3.15 Použití operátoru Reverse v dotazovacím výrazu s klauzulemi orderby a thenby

```
var expr =
  (from c in customers
  where c.Country == Countries.Italy
  orderby c.Name descending, c.City
  select new { c.Name, c.City }
  ).Reverse();
```

Jak vidíte, operátor *Reverse* aplikujeme na výraz z výpisu 3.11. Na pozadí se nejprve vnitřní dotazovací výraz přeloží na seznam rozšiřujících metod a poté se na konec řetězce rozšiřujících metod přidá metoda *Reverse*. Je to stejné jako výpis 3.14, ale dle našeho názoru snazší na zápis.

Sdružovací operátory

Nyní již víte, jak se vybírají, filtrují a řadí sekvence položek. Někdy je při dotazu na data také potřeba výsledky podle určitých kritérií sdružovat. Ke sdružování obsahu slouží sdružovací operátor.

Operátor *GroupBy*, nazývaný též sdružovací operátor, je jediným operátorem tohoto typu a nabízí kolekci osmi verzí. Zde máte první čtyři přetížení:

```
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
   this IEnumerable<TSource> source, Func<TSource, TKey> keySelector);
public static IEnumerable<IGrouping<TKey, TSource>> GroupBy<TSource, TKey>(
   this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
   IEqualityComparer<TKey> comparer);
public static IEnumerable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey, TElement>(
   this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector);
public static IEnumerable<IGrouping<TKey, TElement>> GroupBy<TSource, TKey, TElement>(
   this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector,
   IEqualityComparer<TKey> comparer);
```

Tyto verze metody *GroupBy* vybírají dvojice klíčů a položek pro každou položku ve *zdroji*. Na výběr hodnoty klíče, *Key*, z každé položky používají predikát *keySelector* a sdružují výsledky na základě odlišných hodnot klíče. Je-li přítomen parametr *elementSelector*, definuje funkci, která mapuje zdrojový element ve *zdrojové* sekvenci na cílový element výsledné sekvence. Jestliže *elementSelector* nezadáte, elementy se budou mapovat přímo ze zdroje do cíle. (Ukázku uvidíte později v této kapitole ve výpisu 3.18.) Poté se z nich získává sekvence objektů typu *IGrouping<TKey, TElement>*, v němž se každá skupina skládá ze sekvence položek s běžným klíčem.

Generické rozhraní *IGrouping<TKey, TElement>* je specializovaná implementace rozhraní *IEnumerable<TElement>*. Tato implementace vrací klíč typu *TKey* pro každou položku ve výčtu:

```
public interface IGrouping<TKey, TElement> : IEnumerable<TElement> {
   TKey Key { get; }
}
```

Z praktického pohledu je typ, který implementuje toto generické rozhraní, prostě typový výčet s identifikačním typem *Key* pro každou položku.

Další čtyři verze metody slouží k vlastní projekci výsledků.

```
public static IEnumerable<TResult> GroupBy<TSource, TKey, TResult>(
    this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector);
public static IEnumerable<TResult> GroupBy<TSource, TKey, TElement, TResult>(
    this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector);
public static IEnumerable<TResult> GroupBy<TSource, TKey, TResult>(
    this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer);
public static IEnumerable<TResult> GroupBy<TSource, TKey, Telement, TResult>(
    this IEnumerable<TSource> source, Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    Func<TKey, IEnumerable<TSource>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer);
```

Parametr *resultSelector* v těchto verzích sdružovací metody umožňuje definovat projekci operace *GroupBy*, díky níž lze vracet rozhraní *IEnumerable<TResult>*. Tato sada přetížení se hodí pro výběr prostého výčtu položek, vycházejícího z agregací nad výsledkovými množinami. Ukázku této syntaxe uvidíte později v této části kapitoly.

Posledním parametrem, který lze předat do některých verzí této metody, je *comparer*, který se hodí v situaci, kdy potřebujete porovnávat hodnoty klíče a definovat členy skupin. Jestliže nezadáte vlastní porovnávač, použije se *EqualityComparer<TKey>.Default*. Pořadí klíčů a položek v každé skupině odpovídá jejich výskytu ve *zdroji*. Výpis 3.16 ukazuje příklad použití operátoru *GroupBy*.

Výpis 3.16 Použití operátoru GroupBy pro seskupení zákazníků podle země

```
var expr = customers.GroupBy(c => c.Country);

foreach(IGrouping<Countries, Customer> customerGroup in expr) {
  Console.WriteLine("Zeme: {0}", customerGroup.Key);
  foreach(var item in customerGroup) {
      Console.WriteLine("\t{0}", item);
  }
}
```

Výstup výpisu 3.16 vypadá takto:

```
Zeme: Italy
    Name: Paolo - City: Brescia - Country: Italy
    Name: Marco - City: Torino - Country: Italy
Zeme: USA
    Name: James - City: Dallas - Country: USA
    Name: Frank - City: Seattle - Country: USA
```

Jak ukazuje výpis 3.16, před procházením položek v jednotlivých skupinách je potřeba vyhodnotit všechny klíče skupin. Každá skupina je instancí typu, který implementuje rozhraní *IGrouping<Countries, Customer>*, protože používáme výchozí *elementSelector*, jenž promítá zdrojové instance typu *Customer* přímo do výsledku. V dotazovacích výrazech slouží k definici operátoru *GroupBy* syntaxe *group...by...*, což dokládá výpis 3.17.

Výpis 3.17 Dotazovací výraz se syntaxí group by

```
var expr =
  from c in customers
  group c by c.Country;

foreach(IGrouping<Countries, Customer> customerGroup in expr) {
  Console.WriteLine("Zeme: {0}", customerGroup.Key);
  foreach(var item in customerGroup) {
    Console.WriteLine("\t{0}", item);
  }
}
```

Kód zapsaný ve výpisu 3.16 je sémanticky totožný s kódem z výpisu 3.17.

Výpis 3.18 tvoří další ukázku sdružování, tentokráte s vlastním predikátem *elementSelector*.

Výpis 3.18 Použití operátoru GroupBy pro seskupení zákazníků podle země

```
var expr =
  customers
  .GroupBy(c => c.Country, c => c.Name);
foreach(IGrouping<Countries, String> customerGroup in expr) {
  Console.WriteLine("Zeme: {0}", customerGroup.Key);
  foreach(var item in customerGroup) {
    Console.WriteLine("\t{0}", item);
  }
}
```

Výsledek kódu vypadá takto:

```
Zeme: Italy
   Paolo
   Marco
Zeme: USA
   James
   Frank
```

V posledním příkladu je výsledkem třída, která implementuje rozhraní *IGrouping<Countries, String>*, protože predikát *elementSelector* promítá do výstupní sekvence pouze jména zákazníků (typu *String*).

Ve výpisu 3.19 vidíte příklad použití operátoru *GroupBy* s parametrem *resultSelector*.

Výpis 3.19 Použití operátoru GroupBy pro seskupení zákazníků podle země

```
var expr = customers
  .GroupBy(c => c.Country,
     (k, c) => new { Key = k, Count = c.Count()});
foreach (var group in expr) {
  Console.WriteLine("Klíč: {0} - Počet: {1}", group.Key, group.Count);
}
```

V tomto posledním příkladu jsme promítli hodnotu *Key* každé skupiny a počet, *Count*, elementů v každé skupině. V případě potřeby máte k dispozici i taková přetížení metody *GroupBy*, která vám umožňují definovat *resultSelector* i vlastní *elementSelector*. Tyto verze jsou užitečné tehdy, když potřebujete promítat skupiny, počítat agregace na jednotlivých skupinách, ale také chcete prostřednictvím predikátu *elementSelector* získat výstup s jednoduchými položkami. Příkladem je výpis 3.20.

Výpis 3.20 Použití operátoru GroupBy pro seskupení zákazníků podle země, s vlastním predikátem resultSelector i elementSelector

```
var expr = customers
  .GroupBy(
    c => c.Country, // keySelector
    c => new { OrdersCount = c.Orders.Count() }, // elementSelector
   (key, elements) => new { // resultSelector
      Key = key,
      Count = elements.Count(),
      OrdersCount = elements.Sum(item => item.OrdersCount) });

foreach (var group in expr) {
  Console.WriteLine("Klíč: {0} - Počet: {1} - Počet objednávek: {2}",
    group.Key, group.Count , group.OrdersCount);
}
```

Kód ve výpisu 3.20 představuje ukázku dotazu, který vrací jednoduchý výčet položek, složený ze zákazníků seskupených podle země, z počtu zákazníků v jednotlivých skupinách a z celkového počtu objednávek učiněných zákazníky v příslušné skupině. Všimněte si, že výsledkem dotazu je typ *IEnumerable<TResult>*, a nikoli *IGrouping<TKey, TElement>*. Výstup kódu z výpisu 3.20 vypadá následovně:

```
Klíč: Italy - Počet: 2 - Počet objednávek: 4
Klíč: USA - Počet: 2 - Počet objednávek: 2
```


Spojovací operátory

Spojovací operátory slouží k definici vztahů mezi sekvencemi v dotazovacích výrazech. Z pohledu SQL a relačních systémů vyžaduje téměř každý dotaz spojení jedné či více tabulek. V LINQ existuje množina operátorů, která nabízí tuto *funkcionalitu.*

Join

Prvním operátorem této skupiny je pochopitelně metoda *Join*, definovaná následujícím způsobem:

```
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector);
public static IEnumerable<TResult> Join<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, TInner, TResult> resultSelector,
    IEqualityComparer<TKey> comparer);
```

Operátor *Join* vyžaduje čtyři obecné typy. Typ *TOuter* představuje typ *vnější* sekvence a *TInner* popisuje typ *vnitřní* zdrojové sekvence. Predikáty *outerKeySelector* a *innerKeySelector* definují, jak se mají vybírat identifikační klíče z *vnější* a *vnitřní* zdrojové sekvence položek. Tyto klíče jsou typu *TKey* a jejich rovnost definuje podmínku spojení. Predikát *resultSelector* určuje, co se má promítat do výsledné sekvence, jež je implementací typu *IEnumerable<TResult>*. *TResult* je posledním generickým typem, který operátor vyžaduje, a definuje typ jednotli-

vých položek ve výsledné sekvenci spojení. Druhé přetížení metody má dodatečnou vlastní porovnávací třídu, jež slouží k porovnávání klíčů. Jestliže je parametr *comparer* roven null nebo zavoláte-li první přetížení metody, použije se výchozí třída pro porovnání klíčů (*EqualityComparer<TKey>.Default*).

Uveďme si ukázku, jež nám práci s operátorem *Join* přiblíží. Zamysleme se nad našimi zákazníky a jejich objednávkami a produkty. Dotaz ve výpisu 3.21 propojuje objednávky s odpovídajícími produkty.

Výpis 3.21 Použití operátoru Join k mapování objednávek na produkty

```
var expr =
  customers
  .SelectMany(c => c.Orders)
  .Join( products,
    o => o.IdProduct,
    p => p.IdProduct,
    (o, p) => new {o.Month, o.Shipped, p.IdProduct, p.Price });
```

Výstup dotazu vypadá takto:

```
{Month = January, Shipped = False, IdProduct = 1, Price = 10}
{Month = May, Shipped = True, IdProduct = 2, Price = 20}
{Month = July, Shipped = False, IdProduct = 1, Price = 10}
{Month = December, Shipped = True, IdProduct = 3, Price = 30}
{Month = December, Shipped = True, IdProduct = 3, Price = 30}
{Month = July, Shipped = False, IdProduct = 5, Price = 50}
```

V této ukázce reprezentuje zápis *orders* vnější sekvenci a *products* vnitřní sekvenci. *o* a *p*, použité ve výrazu lambda, jsou typu *Order* a *Product*. Vnitřně operátor vybírá elementy *vnitřní* sekvence do hešovací tabulky pomocí klíčů vybraných prostřednictvím predikátu *innerKeySelector*. Poté se vyhodnocuje *vnější* sekvence a její elementy se podle hodnoty klíče získané predikátem *OuterKeySelector* mapují na hešovací tabulku. Vzhledem k tomuto chování zachovává výsledná sekvence operátoru *Join* nejprve pořadí *vnější* sekvence a poté pro jednotlivé elementy ve *vnější* sekvenci použije pořadí *vnitřní* sekvence.

Z pohledu SQL lze o ukázce ve výpisu 3.21 uvažovat jako o vnitřním ekvivalentním spojení, které by v dotazu SQL vypadalo asi takto:

```
SELECT o.Month, o.Shipped, p.IdProduct, p.Price
FROM Orders AS o
INNER JOIN Products AS p
   ON o.IdProduct = p.IdProduct
```

Chcete-li přeložit syntaxi SQL do syntaxe operátoru *Join*, můžete o výběru sloupců v SQL uvažovat jako o predikátu *resultSelector*, kdežto podmínka pro sloupce *IdProduct* (v tabulkách produktů a zákazníků) odpovídá dvojici predikátů *innerKeySelector* a *outerKeySelector*.

Operátor *Join* má i odpovídající syntaxi pro dotazovací výrazy, kterou vidíte ve výpisu 3.22.

Výpis 3.22 Syntaxe operátoru Join v dotazovacím výrazu

```
var expr =
  from c in customers
    from o in c.Orders
    join p in products
```

```
    on o.IdProduct equals p.IdProduct
  select new {o.Month, o.Shipped, p.IdProduct, p.Price };
```

Důležité

Jak jsme si řekli v kapitole 2, v pořadí porovnávaných položek (*o.IdProduct equals p.IdProduct*) v syntaxi dotazovacích výrazů musí být na prvním místě vnější sekvence a teprve poté může přijít vnitřní sekvence; v opačném případě kompilace dotazovacího výrazu selže. Tento požadavek v běžných dotazech SQL neplatí, neboť tam pořadí položek nehraje roli.

Ve výpisu 3.23 vidíte syntaxi ve Visual Basicu 2008, jež odpovídá dotazu z výpisu 3.22. Všimněte si podobnosti se syntaxí dotazu v SQL.

Výpis 3.23 Syntaxe operátoru Join v dotazovacím výrazu, zapsaná ve Visual Basicu 2008

```
Dim expr = _
  From c In customers _
  From o In c.Orders _
  Join p In products _
    On o.IdProduct Equals p.IdProduct _
  Select o.Month, o.Shipped, p.IdProduct, p.Price
```

GroupJoin

V případech, kdy potřebujete definovat něco jako LEFT OUTER JOIN nebo RIGHT OUTER JOIN, budete pracovat s operátorem *GroupJoin*. Jeho záhlaví je podobné operátoru *Join*:

```
public static IEnumerable<TResult>
    GroupJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector);
public static IEnumerable<TResult>
    GroupJoin<TOuter, TInner, TKey, TResult>(
    this IEnumerable<TOuter> outer,
    IEnumerable<TInner> inner,
    Func<TOuter, TKey> outerKeySelector,
    Func<TInner, TKey> innerKeySelector,
    Func<TOuter, IEnumerable<TInner>, TResult> resultSelector,
    IEqualityComparer<TKey> comparer);
```

Jediným rozdílem je definice projektoru *resultSelector*. Ten vyžaduje instanci typu *IEnumerable<TInner>* namísto jednoduchého objektu typu *TInner*, protože promítá hierarchický výsledek typu *IEnumerable<TResult>*. Každá položka typu *TResult* se skládá z položky získané z *vnější* sekvence a ze skupiny položek typu *TInner* připojených z *vnitřní* sekvence.

Výsledkem tohoto operátoru není prosté vnější spojení s rovností, které by nám dal operátor *Join*, ale hierarchická sekvence položek. Nicméně i s operátorem *GroupJoin* můžete definovat dotazy, které dávají výsledek obdobný operátoru *Join*, pokud jde v mapování o vztah jedna ku jedné. V případech, kdy ve *vnitřní* sekvenci chybí odpovídající skupina elementů, operátor *GroupJoin* vybere element *vnější* sekvence a k němu přidá prázdnou sekvenci (*Count=0*). Ve výpisu 3.24 vidíte ukázku použití tohoto operátoru.

Výpis 3.24 Operátor GroupJoin použitý pro mapování produktů na objednávky, jsou-li nějaké

```
var expr =
  products
  .GroupJoin(
    customers.SelectMany(c => c.Orders),
    p => p.IdProduct,
    o => o.IdProduct,
    (p, orders) => new { p.IdProduct, Orders = orders });

foreach(var item in expr) {
  Console.WriteLine("Produkt: {0}", item.IdProduct);
  foreach (var order in item.Orders) {
    Console.WriteLine("\t{0}", order); }}
```

Výsledek výpisu 3.24 vypadá takto:

```
Produkt: 1
    IdOrder: 1 - IdProduct: 1 - Quantity: 3 - Shipped: False - Month: January
    IdOrder: 3 - IdProduct: 1 - Quantity: 10 - Shipped: False - Month: July
Produkt: 2
    IdOrder: 2 - IdProduct: 2 - Quantity: 5 - Shipped: True - Month: May
Produkt: 3
    IdOrder: 4 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
    IdOrder: 5 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
Produkt: 4
Produkt: 5
    IdOrder: 6 - IdProduct: 5 - Quantity: 20 - Shipped: False - Month: July
Produkt: 6
```

Vidíte, že produkty 4 a 6 nemají žádné související objednávky, ale dotaz je přesto vrátí. O tomto operátoru můžete přemýšlet jako o dotazu typu SELECT ... FOR XML AUTO v jazyce Transact-SQL v Microsoft SQL Serveru 2000 a 2005. Vrací hierarchicky seskupené výsledky podobné množině uzlů XML vnořených do svých nadřízených uzlů, což odpovídá výchozímu typu výsledků dotazu FOR XML AUTO.

V dotazovacím výrazu se operátor *GroupJoin* zadává prostřednictvím klauzule *join ... into*. Dotazovací výraz z výpisu 3.24 je ekvivalentní výpisu 3.25.

Výpis 3.25 Dotazovací výraz s klauzulí join ... into

```
var customersOrders =
  from c in customers
    from o in c.Orders
    select o;

var expr =
  from p in products
  join o in customersOrders
          on p.IdProduct equals o.IdProduct
          into orders
  select new { p.IdProduct, Orders = orders };
```

V tomto příkladě nejprve definujeme výraz s názvem *customerOrders*, který vybírá jednoduchý seznam objednávek. (Tento výraz používá operátor *SelectMany*, protože zde vidíme dvě klauzule *from*.) Je také možné definovat jediný dotazovací výraz, který vkládá výraz *customerOrders* do hlavního dotazu. To vidíte ve výpisu 3.26.

Výpis 3.26 Dotazovací výraz z výpisu 3.25 v kompaktní podobě

```
var expr =
  from p in products
  join o in (
        from c in customers
          from o in c.Orders
          select o
        ) on p.IdProduct equals o.IdProduct
        into orders
  select new { p.IdProduct, Orders = orders };
```

Množinové operátory

Naše putování krajinou operátorů LINQ pokračuje skupinou metod, které slouží k práci s množinami dat, aplikují na ně běžné množinové operace (*sjednocení, průnik* a *doplněk* – *union, intersect* a *except*) a vybírají jedinečné výskyty jednotlivých položek (*distinct*).

Distinct

Představte si, že chcete vybrat všechny produkty, které odpovídají objednávkám, ale chcete předejít duplicitám. Tento požadavek ve standardním SQL splňuje v dotazu se spojením klauzule DISTINCT. LINQ nabízí rovněž operátor *Distinct*. Jeho záhlaví je prosté. Přebírá pouze *zdrojovou* sekvenci, z níž se získávají veškeré vzájemně odlišné položky, a nabízí přetížení s vlastním typem *IEqualityComparer<TSource>*, o němž pohovoříme později.

```
public static IEnumerable<TSource> Distinct<TSource>(
   this IEnumerable<TSource> source);
public static IEnumerable<TSource> Distinct<TSource>(
   this IEnumerable<TSource> source,
   IEqualityComparer<TSource> comparer);
```

Ukázku operátoru vidíte ve výpisu 3.27.

Výpis 3.27 Operátor Distinct aplikovaný na seznam produktů v objednávkách

```
var expr =
  customers
  .SelectMany(c => c.Orders)
  .Join(products,
      o => o.IdProduct,
      p => p.IdProduct,
     (o, p) => p)
  .Distinct();
```

Operátor *Distinct* nemá odpovídající klauzuli v dotazovacích výrazech, a proto můžeme – podobně jako ve výpisu 3.15 – aplikovat tento operátor na výsledky dotazovacího výrazu, což vidíte ve výpisu 3.28.

Výpis 3.28 Operátor Distinct aplikovaný na dotazovací výraz

```
var expr =
  (from c in customers
      from o in c.Orders
      join p in products
          on o.IdProduct equals p.IdProduct
      select p
  ).Distinct();
```

Operátor *Distinct* standardně porovnává a identifikuje elementy pomocí metod *GetHashCode* a *Equals*, protože vnitřně používá výchozí porovnávač typu *EqualityComparer<T>.Default*. V případě potřeby je možné tento způsob chování přepsat a změnit tak výsledky operátoru *Distinct*, případně použít druhé přetížení metody *Distinct*.

```
public static IEnumerable<TSource> Distinct<TSource>(
    this IEnumerable<TSource> source,
    IEqualityComparer<TSource> comparer);
```

Toto poslední přetížení přebírá parametr *comparer*, do nějž můžete vložit vlastní porovnávač instancí typu *TSource*.

Poznámka

Ukázku porovnávání referenčních typů v operátoru *Union* uvidíte ve výpisu 3.29.

Union, Intersect a Except

V množině operátorů existují ještě tři další operátory, jež slouží k běžným množinovým operacím. Jde o operátory *Union*, *Intersect* a *Except* a jejich definice je obdobná:

```
public static IEnumerable<TSource> Union<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second);
public static IEnumerable<TSource> Union<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer);
public static IEnumerable<TSource> Intersect<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second);
public static IEnumerable<TSource> Intersect<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer);
public static IEnumerable<TSource> Except<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second);
public static IEnumerable<TSource> Except<TSource>(
    this IEnumerable<TSource> first,
    IEnumerable<TSource> second,
    IEqualityComparer<TSource> comparer);
```

Operátor *Union* vyhodnocuje *první* sekvenci a *druhou* sekvenci a vkládá takový element, který zatím nebyl přidán. Například ve výpisu 3.29 vidíte, jak se spojí dvě množiny celých čísel.

Výpis 3.29 Operátor Union aplikovaný na dvě celočíselné množiny

```
Int32[] setOne = {1, 5, 6, 9};
Int32[] setTwo = {4, 5, 7, 11};

var union = setOne.Union(setTwo);
foreach (var i in union) {
  Console.Write(i + ", ");
}
```

Výsledkem výpisu 3.29 je tento výstup:

```
1, 5, 6, 9, 4, 7, 11,
```

Stejně jako u operátoru *Distinct*, i v případě operátorů *Union*, *Intersect* a *Except* se elementy v prvních verzích těchto operátorů porovnávají pomocí metod *GetHashCode* a *Equals* a ve druhých přetíženích se pracuje s vlastním porovnávačem. Podívejte se na kód ve výpisu 3.30.

Výpis 3.30 Operátor Union aplikovaný na dvě množiny produktů.

```
Product[] productSetOne = {
  new Product {IdProduct = 46, Price = 1000 },
  new Product {IdProduct = 27, Price = 2000 },
  new Product {IdProduct = 14, Price = 500}};
Product[] productSetTwo = {
  new Product {IdProduct = 11, Price = 350 },
  new Product {IdProduct = 46, Price = 1000 }};

var productsUnion = productSetOne.Union(productSetTwo);

foreach (var item in productsUnion) {
  Console.WriteLine(item);
}
```

Výstup z tohoto kódu má tento tvar:

```
IdProduct: 46 - Price: 1000
IdProduct: 27 - Price: 2000
IdProduct: 14 - Price: 500
IdProduct: 11 - Price: 350
IdProduct: 46 - Price: 1000
```

Možná vás výsledek překvapí, protože zde máme dva řádky, první a poslední, které se zdají být totožné. Ale pokud se podíváte na inicializační kód ve výpisu 3.30, uvidíte, že každý produkt je různou instancí referenčního typu *Product*. I když je druhý produkt množiny *productSetTwo* fakticky totožný s prvním produktem z množiny *productSetOne*, jde o odlišné objekty, jež mají různé hešovací kódy.

Pro referenční typ *Product* jsme nedefinovali sémantiku hodnoty typu. Abychom dostali očekávané výsledky, můžeme přepsat pro srovnávaný typ metody *GetHashCode* a *Equals* a zavést vlastní porovnávací logiku. V této situaci se to může hodit, což ukazuje nová implementace třídy *Product*:

```
public class Product {
    public int IdProduct;
    public decimal Price;

    public override string ToString(){
        return String.Format("IdProduct: {0} - Price: {1}",
            this.IdProduct, this.Price);
    }

    public override bool Equals(object obj) {
        if (!(obj is Product))
            return false;
        else {
          Product p = (Product)obj;
```

```
        return (p.IdProduct == this.IdProduct &&
          p.Price == this.Price);
      }
    }

    public override int GetHashCode(){
        return String.Format("{0}|{1}", this.IdProduct, this.Price)
            .GetHashCode();
    }
}
```

Jiným způsobem, jak dostat správný výsledek, je použít druhé přetížení metody *Union* a zadat pro typ *Product* vlastní porovnávač. Posledním způsobem, jak se dobrat požadovaných výsledků, je definovat typ *Product* jako hodnotový typ – pomocí výrazu *struct* namísto *class* v jeho deklaraci. Mimochodem, definice struktury, *struct*, není vždy možná, protože občas potřebujeme implementovat objektově orientovanou infrastrukturu s dědičností typů.

```
// Při použití struktury namísto třídy dostaneme hodnotový typ.
public struct Product {
    public int IdProduct;
    public decimal Price;
}
```

Pamatujte si, že anonymní typ se definuje jako referenční typ se sémantikou hodnotového typu. Jinými slovy, všechny anonymní typy se definují jako třída s přepisem metod *GetHashCode* a *Equals* zadaným kompilátorem, v implementaci využívající metody *GetHashCode* a *Equals* pro všechny vlastnosti příslušné instance anonymního typu.

Ve výpisu 3.31 naleznete ukázku použití operátorů *Intersect* a *Except*.

Výpis 3.31 Operátory Intersect a Except, aplikované na tutéž množinu produktů jako ve výpisu 3.30

```
var expr = productSetOne.Intersect(productSetTwo);
var expr = productSetOne.Except(productSetTwo);
```

Operátor *Intersect* vybírá pouze ty elementy, které se objevují v obou sekvencích, a operátor *Except* vybírá všechny elementy z *první* sekvence, které se nenacházejí ve *druhé* sekvenci. Opět neexistují kompaktní klauzule, které by umožnily zadat množinové operátory do dotazovacího výrazu, ale můžete je aplikovat na výsledky dotazovacího výrazu, což ukazuje výpis 3.32.

Výpis 3.32 Množinové operátory aplikované na dotazovací výrazy

```
var expr = (
  from c in customers
    from o in c.Orders
      join p in products on o.IdProduct equals p.IdProduct
  where c.Country == Countries.Italy
  select p)
  .Intersect(
  from c in customers
    from o in c.Orders
      join p in products on o.IdProduct equals p.IdProduct
  where c.Country == Countries.USA
  select p);
```

Sémantika hodnotového typu oproti referenčnímu typu

Pamatujte si, že veškeré úvahy pro operátory *Union* a *Distinct* platí též pro operátory *Intersect* a *Except*. Obecně platí pro všechny operace, které znamenají porovnávání dvou položek vytvořených v LINQ pro objekty. Výsledek operace *Intersect* ve výpisu 3.31 bude prázdný vždy, když typ *Product* bude referenčním typem, postrádajícím přepis metod *GetHashCode* a *Equals*. Jestliže nadefinujete *Product* jako hodnotový typ (pomocí *struct* namísto *class*), dostanete jako výsledek průniku jeden produkt (IdProduct: 46 – Price: 1000). Opět bychom chtěli zdůraznit, že při práci s LINQ je lepší používat typy se sémantikou hodnotových typů, i když se jedná o referenční typy, aby se prostředí chovalo konzistentně pro všechny běžné a anonymní typy.

Agregační operátory

Někdy je potřeba provést nad sekvencemi určité agregace, což ve výsledku znamená určité výpočty nad zdrojovými položkami. LINQ k tomu nabízí sadu agregačních operátorů, které zahrnují nejčastější agregační funkce: *Count*, *LongCount*, *Sum*, *Min*, *Max*, *Average* a *Aggregate*. Mnoho z těchto operátorů se používá snadno, neboť jejich chování není obtížné pochopit. Ale nezapomeňte, že LINQ pro objekty pracuje s instancemi typu *IEnumerable<T>* v paměti, a proto se mohou v kódu provádějícím podobnou operaci vyskytnout určité obtíže v situacích, kdy potřebujete procházet výsledky dotazu vícekrát.

Count a LongCount

Představte si situaci, že chcete vypsat všechny zákazníky a za každým uvést počet objednávek, které učinil. Ve výpisu 3.33 vidíte příslušnou syntaxi využívající operátoru *Count*.

Výpis 3.33 Operátor Count aplikovaný na objednávky zákazníků

```
var expr =
  from c in customers
  select new {c.Name, c.Country, OrdersCount = c.Orders.Count() };
```

Operátor *Count* má dvě verze, stejně jako operátor *LongCount*:

```
public static int Count<TSource>(
   this IEnumerable<TSource> source);
public static int Count<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, Boolean> predicate);
public static long LongCount<TSource>(
   this IEnumerable<TSource> source);
public static long LongCount<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, Boolean> predicate);
```

Metoda použitá ve výpisu 3.33 je běžnou a jednodušší verzí, která pouze spočítá položky ve *zdrojové* sekvenci. Druhé přetížení metody přebírá *predikát*, který se používá na filtrování počítaných položek. Varianta *LongCount* vrací namísto typu *integer* typ *long*.

Sum

Operátor *Sum* vyžaduje větší pozornost, neboť má více definicí:

```
public static Numeric Sum(
   this IEnumerable<Numeric> source);
public static Numeric Sum<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, Numeric> selector);
```

V syntaxi jsme použili typ *Numeric*, abychom návratový typ operátoru *Sum* zobecnili. V praxi existuje mnoho definicí, pro každý hlavní číselný typ jedna: *Int32, Nullable<Int32>, Int64, Nullable<Int64>, Single, Nullable<Single>, Double, Nullable<Double>, Decimal a Nullable<Decimal>*.

Důležité

Nezapomeňte, že v C# 2.0 a vyšším definuje otazník zapsaný za názvem hodnotového typu (*T?*) stejný typ, který však může nabývat hodnoty null (*Nullable<T>*). Můžete například psát *int?* namísto *Nullable<System.Int32>*.

První implementace sčítá položky ve *zdrojové* sekvenci, předpokládá, že všechny položky jsou téhož datového typu, a vrací výsledek. V případě, že je *zdrojová* sekvence prázdná, vrátí operátor hodnotu nula. Tuto implementaci lze použít, je-li možné sčítat položky přímo. Například pole celých hodnot můžeme sčítat tímto kódem:

```
int[] values = { 1, 3, 9, 29 };
int total = values.Sum();
```

Jestliže se sekvence neskládá z prostých typů *Numeric*, musíme nejprve pro každou položku ve *zdrojové* sekvenci získat hodnoty, které budeme sčítat. K tomu nám poslouží druhé přetížení, které přebírá parametr *selector*. Ukázku této syntaxe vidíte ve výpisu 3.34.

Výpis 3.34 Operátor Sum aplikovaný na objednávky zákazníků

```
var customersOrders =
  from c in customers
    from o in c.Orders
    join p in products
        on o.IdProduct equals p.IdProduct
    select new { c.Name, OrderAmount = o.Quantity * p.Price };

foreach (var o in customersOrders) {
  Console.WriteLine(o);
}

Console.WriteLine();

var expr =
  from c in customers
  join o in customersOrders
       on c.Name equals o.Name
       into customersWithOrders
  select new { c.Name,
           TotalAmount = customersWithOrders.Sum(o => o.OrderAmount) };
```

```
foreach (var item in expr) {
  Console.WriteLine(item);
}
```

Ve výpisu 3.34 spojujeme zákazníky do sekvence *customerOrders* a získáváme tak seznam jmen zákazníků spojený s celkovým počtem zadaných objednávek, což dává následující výstup:

```
{ Name = Paolo, OrderAmount = 30 }
{ Name = Paolo, OrderAmount = 100 }
{ Name = Marco, OrderAmount = 100 }
{ Name = Marco, OrderAmount = 600 }
{ Name = James, OrderAmount = 600 }
{ Name = Frank, OrderAmount = 1000 }
```

Další spojení se používá pro každého zákazníka a dává celkovou hodnotu jeho objednávek, vypočtenou operátorem *Sum*, viz následující výstup:

```
{ Name = Paolo, TotalAmount = 130 }
{ Name = Marco, TotalAmount = 700 }
{ Name = James, TotalAmount = 600 }
{ Name = Frank, TotalAmount = 1000 }
```

Jak obvykle můžeme předchozí kód sloučit pomocí vnořených dotazů, což nám ukazuje výpis 3.35.

Výpis 3.35 Operátor Sum aplikovaný na objednávky zákazníků ve vnořeném dotazu

```
var expr =
  from c in customers
  join o in (
      from c in customers
        from o in c.Orders
        join p in products
            on o.IdProduct equals p.IdProduct
        select new { c.Name, OrderAmount = o.Quantity * p.Price }
      ) on c.Name equals o.Name
      into customersWithOrders
  select new { c.Name,
           TotalAmount = customersWithOrders.Sum(o => o.OrderAmount) };
```

Syntaxe dotazovacích výrazů v SQL a v LINQ

Porovnejme nyní uváděné kódy se syntaxí SQL, protože jsou zde sice podobnosti, ovšem také významné odlišnosti. Následuje příkaz SQL, který je podobný dotazovacímu výrazu z výpisu 3.35 a předpokládá, že jména zákazníků jsou jedinečná:

```
SELECT c.Name, SUM(o.OrderAmount) AS OrderAmount
FROM customers AS c
INNER JOIN (
SELECT c.Name, o.Quantity * p.Price AS OrderAmount
FROM customers AS c
INNER JOIN orders AS o ON c.Name = o.Name
INNER JOIN products AS p ON o.IdProduct = p.IdProduct
) AS o
ON c.Name = o.Name
GROUP BY c.Name
```

Je vidět, že tato syntaxe SQL je zbytečně složitá. Téhož výsledku dosáhneme jednodušším dotazem:

```
SELECT c.Name, SUM(o.OrderAmount) AS OrderAmount
FROM customers AS c
INNER JOIN (
SELECT o.Name, o.Quantity * p.Price AS OrderAmount
FROM orders AS o
INNER JOIN products AS p ON o.IdProduct = p.IdProduct
) AS o
ON c.Name = o.Name
GROUP BY c.Name
```

Ale stále ještě můžeme dotaz zjednodušit a zkrátit, až nabude tuto podobu:

```
SELECT c.Name, SUM(o.Quantity * p.Price) AS OrderAmount
FROM customers AS c
INNER JOIN orders AS o ON c.Name = o.Name
INNER JOIN products AS p ON o.IdProduct = p.IdProduct
GROUP BY c.Name
```

Kdybychom začali tímto posledním dotazem a pokusili se napsat odpovídající dotazovací výraz v LINQ, pravděpodobně bychom narazili na určité obtíže. Důvodem je skutečnost, že SQL se dotazuje na data prostřednictvím vazeb, nicméně všechna data jsou uspořádaná jednoduše (v tabulkách), dokud se na ně nezeptáme. Oproti tomu LINQ zpracovává data, která mají přirozené hierarchické vztahy, například naše data Customer/Orders/Products. Tento rozdíl znamená, že v určitých situacích převládají výhody jednoho přístupu nad druhým, v závislosti na typu dotazu a druhu dat, s nimiž pracujeme.

Z těchto důvodů se nejlepší tvar výrazu může jevit odlišně v syntaxi SQL a LINQ, i když dotaz samotný dává ze stejných dat stejné výsledky.

Min a Max

Operátory *Min* a *Max* z agregační skupiny počítají nejmenší a největší hodnotu ze zdrojové sekvence. Obě tyto rozšiřující metody nabízejí širokou škálu přetížení:

```
public static Numeric Min/Max(
    this IEnumerable<Numeric> source);
public static TSource Min<TSource>/Max<TSource>(
    this IEnumerable<TSource> source);
public static Numeric Min<TSource>/Max<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Numeric> selector);
public static TResult Min<TSource, TResult>/Max<TSource, TResult>(
    this IEnumerable<TSource> source,
    Func<TSource, TResult> selector);
```

První verze nabízí stejně jako operátor *Sum* množství definicí pro hlavní číselné typy (*Int32, Nullable<Int32>, Int64, Nullable<Int64>, Single, Nullable<Single>, Double, Nullable<Double>, Decimal* a *Nullable<Decimal>*) a počítá nejnižší či nejvyšší hodnotu elementů ve *zdrojové* sekvenci aritmetickým způsobem. Tato verze metody je vhodná v situaci, kdy zdrojové elementy tvoří samotná čísla, jako je tomu ve výpisu 3.36.

Výpis 3.36 Operátor Min aplikovaný na množství objednávek

```
var expr =
  (from c in customers
    from o in c.Orders
    select o.Quantity
  ).Min();
```

Druhá verze metod počítá nejnižší či nejvyšší hodnotu zdrojových elementů bez ohledu na jejich typ. Srovnání se provádí prostřednictvím implementace rozhraní *IComparable<TSource>*, pokud je zdrojové elementy podporují, nebo pomocí negenerického rozhraní *IComparable*. Jestliže zdrojový typ neobsahuje ani jedno z těchto rozhraní, dojde k vyvolání výjimky *ArgumentException* a hlášení *Exception.Message* bude obsahovat text „At least one object must implement *IComparable*". Abyste situaci lépe porozuměli, podívejte se na výpis 3.37, v němž výsledný anonymní typ neobsahuje ani jedno z rozhraní požadovaných operátorem *Min*.

Výpis 3.37 Operátor Min aplikovaný na špatné typy (dojde k výjimce ArgumentException)

```
var expr =
  (from c in customers
    from o in c.Orders
    select new { o.IdProduct, o.Quantity }
  ).Min();
```

V případě prázdného zdroje či prázdných hodnot ve zdrojové sekvenci bude v situaci, kdy daný číselný typ může obsahovat hodnotu null, výsledkem hodnota null, v opačném případě dojde k výjimce *InvalidOperationException*. Predikát *selector*, jenž je součástí posledních dvou verzí metody, definuje funkci, s jejíž pomocí se získávají hodnoty z elementů ve *zdrojové* sekvenci. Tato přetížení můžete například použít k tomu, abyste se vyhnuli chybám pramenícím z chybějících rozhraní (*IComparable<T>*/*IComparable*), což předvádí výpis 3.38.

Výpis 3.38 Operátor Min aplikovaný na vlastní typy ve spolupráci se selektorem hodnot

```
var expr =
  (from c in customers
    from o in c.Orders
    select new { o.IdProduct, o.Quantity }
  ).Min(o => o.Quantity);
```

Average

Operátor *Average* počítá aritmetický průměr množiny hodnot vybraných ze zdrojové sekvence. Stejně jako předchozí operátory, i tato funkce pracuje se zdrojovými elementy samotnými či s hodnotami vybranými pomocí vlastního selektoru.

```
public static Result Average(
    this IEnumerable<Numeric> source);
public static Result Average<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Numeric> selector);
```

Uvedený typ *Numeric* může být *Int32, Nullable<Int32>, Int64, Nullable<Int64>, Single, Nullable<Single>, Double, Nullable<Double>, Decimal* a *Nullable<Decimal>*. Typ *Result* vždy odráží „nulovatelnost" daného číselného typu. Je-li daný typ *Numeric* roven *Nullable<Int32>*

či *Nullable<Int64>*, výsledný typ *Result* bude *Nullable<Double>*. V ostatních případech budou typy *Numeric* a *Result* totožné.

Je-li součet hodnot používaných pro výpočet aritmetického průměru příliš velký na datový typ výsledku, dojde k výjimce *OverflowException*. První verzi operátoru *Average* lze z definice použít pouze pro číselnou sekvenci. Chcete-li jej volat pro zdrojovou sekvenci s instancemi nečíselných typů, musíte zadat vlastní *selector*. Ve výpisu 3.39 vidíte ukázku obou typů přetížení.

Výpis 3.39 Obě verze operátoru Average aplikované na ceny produktu

```
var expr =
  (from p in products
  select p.Price
  ).Average();
var expr =
  (from p in products
  select new { p.IdProduct, p.Price }
  ).Average(p => p.Price);
```

Druhá verze metody je vhodná, když zadáváte dotaz, v němž průměr tvoří pouze jednu z vybíraných výsledných hodnot. Ukázku tvoří výpis 3.40, v němž vybíráme všechny zákazníky a průměrné výše jejich objednávek.

Výpis 3.40 Zákazníci a průměrná cena jejich objednávek

```
var expr =
  from c in customers
  join o in (
      from c in customers
          from o in c.Orders
          join p in products
                on o.IdProduct equals p.IdProduct
          select new { c.Name, OrderAmount = o.Quantity * p.Price }
      ) on c.Name equals o.Name
      into customersWithOrders
  select new { c.Name,
            AverageAmount = customersWithOrders.Average(o =>
o.OrderAmount) };
```

Výsledek bude vypadat asi takto:

```
{ Name = Paolo, AverageAmount = 65 }
{ Name = Marco, AverageAmount = 350 }
{ Name = James, AverageAmount = 600 }
{ Name = Frank, AverageAmount = 1000 }
```

Aggregate

Posledním operátorem z této množiny je *Aggregate*. Podívejte se na jeho definici:

```
public static T Aggregate<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, TSource, TSource> func);
public static TAccumulate Aggregate<TSource, TAccumulate>(
    this IEnumerable<TSource> source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func);
```

```
public static TResult Aggregate<TSource, TAccumulate, TResult>(
    this IEnumerable< TSource > source,
    TAccumulate seed,
    Func<TAccumulate, TSource, TAccumulate> func,
    Func<TAccumulate, TResult> resultSelector);
```

Tento operátor opakovaně volá funkci *func* a výsledek ukládá do akumulátoru. Každý krok volá funkci s aktuální hodnotou v akumulátoru v prvním parametru, počínaje hodnotou *seed*, a s aktuálním elementem ze *zdrojové* sekvence ve druhém parametru. Na konci iterace vrací operátor výslednou hodnotu v akumulátoru.

Jediným rozdílem mezi prvními dvěma verzemi je, že druhá vyžaduje explicitní zadání hodnoty *seed* typu *TAccumulator*. První verze používá jako hodnotu *seed* první element ve *zdrojové* sekvenci a typ hodnoty *seed* odvozuje ze samotné *zdrojové* sekvence. Třetí typ metody je podobný druhému, avšak vyžaduje predikát *resultSelector*, který se volá při výběru finálního výsledku.

Ve výpisu 3.41 použijeme operátor *Aggregate* na výběr nejnákladnější objednávky každého zákazníka.

Výpis 3.41 Zákazníci a jejich nejvyšší objednávky

```
var expr =
  from c in customers
  join o in (
      from c in customers
          from o in c.Orders
          join p in products
              on o.IdProduct equals p.IdProduct
          select new { c.Name, o.IdProduct,
                       OrderAmount = o.Quantity * p.Price }
      ) on c.Name equals o.Name
      into orders
  select new { c.Name,
               MaxOrderAmount =
                 orders
                 .Aggregate((a, o) => a.OrderAmount > o.OrderAmount ?
                                      a : o)
                 .OrderAmount };
```

Jak vidíte, funkce volaná operátorem *Aggregate* porovnává vlastnost *OrderAmount* každé objednávky aktuálního zákazníka a uchovává záznam o nejvyšší z nich v proměnné akumulátoru (*a*). Na konci agregace pro každého zákazníka bude akumulátor obsahovat nejvyšší objednávku a jeho vlastnost *OrderAmount* se promítne do závěrečného výsledku společně se jménem zákazníka. Výstup z dotazu vypadá následovně:

```
{ Name = Paolo, MaxOrderAmount = 100 }
{ Name = Marco, MaxOrderAmount = 600 }
{ Name = James, MaxOrderAmount = 600 }
{ Name = Frank, MaxOrderAmount = 1000 }
```

Ve výpisu 3.42 vidíte další ukázku agregace. Tento příklad počítá celkovou hodnotu objednávek jednotlivých produktů.

Výpis 3.42 Produkty a jejich celková objednaná hodnota

```
var expr =
  from p in products
  join o in (
      from c in customers
          from o in c.Orders
          join p in products
              on o.IdProduct equals p.IdProduct
          select new { p.IdProduct, OrderAmount = o.Quantity * p.Price }
      ) on p.IdProduct equals o.IdProduct
      into orders
  select new { p.IdProduct,
               TotalOrderedAmount =
                 orders
                 .Aggregate(0m, (a, o) => a += o.OrderAmount)};
```

Zde vidíme výstup dotazu:

```
{ IdProduct = 1, TotalOrderedAmount = 130 }
{ IdProduct = 2, TotalOrderedAmount = 100 }
{ IdProduct = 3, TotalOrderedAmount = 1200 }
{ IdProduct = 4, TotalOrderedAmount = 0 }
{ IdProduct = 5, TotalOrderedAmount = 1000 }
{ IdProduct = 6, TotalOrderedAmount = 0 }
```

V této druhé ukázce používá agregační funkce akumulátor typu *Decimal*. Je inicializován na hodnotu nula (*seed=0m*) a pro každý krok akumuluje hodnoty *OrderAmount*. Výsledek této funkce bude rovněž typu *Decimal*.

Oba předchozí příklady jsme mohli také zapsat pomocí operátorů *Max* či *Sum*. Uvedli jsme si je v této části kapitoly kvůli výkladu chování operátoru *Aggregate*. Obecně si pamatujte, že operátor *Aggregate* je vhodné použít v situaci, kdy nelze použít žádný jiný specializovaný agregační operátor; v opačném případě je dobré použít operátory jako *Min*, *Max*, *Sum* atd. Podívejte se kupříkladu na výpis 3.43.

Výpis 3.43 Zákazníci a jejich největší objednávky spárované s měsícem uskutečnění

```
var expr =
  from c in customers
  join o in (
        from c in customers
            from o in c.Orders
            join p in products
                on o.IdProduct equals p.IdProduct
            select new { c.Name, o.IdProduct, o.Month,
                      OrderAmount = o.Quantity * p.Price }
        ) on c.Name equals o.Name into orders
  select new { c.Name,
               MaxOrder =
                   orders
                   .Aggregate( new { Amount = 0m, Month = String.Empty },
                               (a, s) => a.Amount > s.OrderAmount
                                     ? a
                                     : new { Amount = s.OrderAmount,
                                         Month = s.Month })};
```

Výsledek výpisu 3.43 vypadá takto:

```
{ Name = Paolo, MaxOrder = { Amount = 100, Month = May } }
{ Name = Marco, MaxOrder = { Amount = 600, Month = December } }
{ Name = James, MaxOrder = { Amount = 600, Month = December } }
{ Name = Frank, MaxOrder = { Amount = 1000, Month = July } }
```

V uvedeném příkladě vrací operátor *Aggregate* nový anonymní typ s názvem *MaxOrder*: jde o dvojici složenou z hodnoty a měsíce, kdy jednotliví zákazníci uskutečnili svou nejvyšší objednávku. Zde nelze operátor *Aggregate* nahradit žádným předdefinovaným agregačním operátorem kvůli jeho specifickému chování a výslednému datovému typu.

Poznámka

Další informace o anonymních typech naleznete v příloze B, „C# 3.0: Nové funkce jazyka" či v příloze C, „Microsoft Visual Basic 2008: Nové funkce jazyka".

Jediným způsobem, jak dosáhnout obdobného výsledku pomocí standardních agregačních operátorů, je volat dva odlišné agregátory. To by znamenalo dvojí prohledávání *zdrojových* elementů: Jednou pro získání nejvyšší částky a podruhé pro získání měsíce. Dejte si pozor na definici hodnoty *seed*, která deklaruje výsledný anonymní typ, jenž se rovněž použije v agregační funkci.

Agregační operátory ve Visual Basicu 2008

Visual Basic 2008 zavádí množinu nových klíčových slov a klauzulí pro syntaxi dotazovacích výrazů LINQ, které umožňují lepší a snazší agregace nad datovými položkami. Především jde o klauzuli *Aggregate*, jež umožňuje agregovat funkce v dotazovacích výrazech. Syntaxe této klauzule vypadá následovně:

```
Aggregate element [As type] In collection _
   [, element2 [As type2] In collection2, [...]]
   [ clause ]
   Into expressionList
```

Výraz *element* je zde položka z iterace nad zdrojovou *kolekcí*, jež se má použít při provádění agregace. *Clause* (nepovinná část) představuje libovolný dotazovací výraz, který upřesňuje agregované položky. Může jít například o klauzuli *Where*. *ExpressionList* je povinná součást syntaxe a definuje jeden či více výrazů oddělených čárkami, které určují agregační funkci aplikovanou na kolekci. Standardní agregační funkce, které lze použít, jsou *All, Any, Average, Count, LongCount, Max, Min* a *Sum*.

Ve výpisu 3.44 vidíte ukázku klauzule *Aggregate*, jež počítá průměrnou cenu produktů objednaných zákazníky.

Výpis 3.44 Průměrná cena produktů objednaných zákazníky

```
Dim productsOrdered = _
  From c In customers _
  From o In c.Orders _
    Join p In products _
    On o.IdProduct Equals p.IdProduct _
  Select p
```

```
Dim expr = Aggregate p In productsOrdered _
  Into Average(p.Price)
```

Jak jste již viděli dříve v této kapitole v příkladech v jazyce C# 3.0, lze předchozí kód sloučit a napsat jediný dotazovací výraz. Ve výpisu 3.45 vidíte řešení s jediným dotazem.

Výpis 3.45 Průměrná cena produktů objednaných zákazníky, vypočtená v jediném dotazu

```
Dim expr = Aggregate p In ( _
  From c In customers _
  From o In c.Orders _
    Join p In products _
    On o.IdProduct Equals p.IdProduct _
  Select p) _
  Into Average(p.Price)
```

Nejzajímavější vlastností klauzule *Aggregate* je schopnost pracovat s libovolnou agregační funkcí, dokonce i s takovou, kterou si sami definujete. Jestliže si nadefinujete vlastní rozšiřující metodu, která rozšiřuje rozhraní *IEnumerable<T>*, můžete ji použít v seznamu *ExpressionList* v klauzuli *Aggregate*. Ve výpisu 3.46 uvádíme příklad vlastní agregační funkce, která počítá standardní odchylku množiny hodnot popisujících ceny produktů.

Výpis 3.46 Vlastní agregační funkce, která počítá standardní odchylku množiny hodnot typu Double

```
<Extension()> _
Function StandardDeviation( _
  ByVal source As IEnumerable(Of Double)) As Double

  If source Is Nothing Then
    Throw New ArgumentNullException("source")
  End If

  If source.Count = 0 Then
    Throw New InvalidOperationException("Nelze počítat standardní
odchylku pro prázdnou množinu.")
  End If

  Dim avg = Aggregate v In source Into Average(v)
  Dim accumulator As Double = 0

  For Each x In source
    accumulator += (x - avg) ^ 2
  Next

  Return Math.Sqrt(accumulator / (source.Count))

End Function

<Extension()> _
Function StandardDeviation(Of TSource)( _
    ByVal source As IEnumerable(Of TSource), _
    ByVal selector As Func(Of TSource, Double)) As Double

  Return (From element In source Select _
    selector(element)).StandardDeviation()

End Function
```

```
Sub Main()

  Dim expr = Aggregate p In products _
    Into StandardDeviation(p.Price)

End Sub
```

Operátory generování

Když pracujete s daty prostřednictvím agregátů, matematických operací a matematických funkcí, narazíte občas na potřebu procházení čísel či položek v kolekci v cyklu. Uvažte například dotaz, v němž chcete vybrat objednávky zadané pro konkrétní množinu let, mezi roky 2000 a 2007, či dotaz, který má opakovat jednu a tutéž operaci nad stejnými daty. Pro podobné typy operací se hodí operátory generování.

Range

Prvním operátorem v této množině je *Range*. Jde o jednoduchou rozšiřující metodu, která vrací množinu celých čísel vybraných ze zadaného intervalu hodnot, což dokládá její záhlaví:

```
public static IEnumerable<Int32> Range(
    Int32 start,
    Int32 count);
```

Kód ve výpisu 3.47 ilustruje způsob, jak filtrovat objednávky na měsíce v intervalu od ledna do června.

Důležité

Pamatujte si, že v následujícím příkladu by bylo vhodnější použít podmínku *where*, protože objednávky procházíme mnohokrát. Příklad z výpisu 3.47 uvádíme pouze pro demonstrační účely a není nejlepším řešením tohoto konkrétního dotazu.

Výpis 3.47 Množina měsíců generovaná operátorem Range a použitá na filtrování objednávek

```
var expr = Enumerable.Range(1, 6)
  .SelectMany(x => (
    from o in (
      from c in customers
      from o in c.Orders
      select o)
    where o.Month ==
      new CultureInfo("en-US").DateTimeFormat.GetMonthName(x)
    select new { o.Month, o.IdProduct }));
```

Operátor *Range* lze také použít na implementaci klasických matematických operací. Výpis 3.48 obsahuje příklad použití operátorů *Range* a *Aggregate* na výpočet faktoriálu.

Výpis 3.48 Faktoriál čísla počítaný pomocí operátoru Range

```
static int Factorial(int number) {
  return (Enumerable.Range(0, number + 1)
    .Aggregate(0, (s, t) => t == 0 ? 1 : s *= t)); }
```

Repeat

Dalším operátorem generování je *Repeat*, který vrací množinu *count* výskytů *elementu*. Jestliže je *element* instancí referenčního typu, každé opakování vrací odkaz na tutéž instanci, nikoli jeho kopii.

```
public static IEnumerable<TResult> Repeat<TResult>(
   TResult element,
   int count);
```

Operátor *Repeat* je užitečný pro inicializaci výčtů (s použitím téhož elementu pro všechny instance) nebo pro vícenásobné opakování téhož dotazu. Ve výpisu 3.49 dvakrát provádíme výběr jména zákazníka.

Výpis 3.49 Vícenásobné opakování téhož dotazu pomocí operátoru Repeat

```
var expr =
  Enumerable.Repeat( ( from c in customers
                       select c.Name), 2)
  .SelectMany(x => x);
```

Povšimněte si prosím, že v tomto příkladě vrací *Repeat* sekvenci sekvencí, kterou tvoří dva seznamy jmen zákazníků. Z tohoto důvodu jsme použili metodu *SelectMany*, abychom dostali jednoduchý seznam jmen.

Empty

Poslední operátor generování je *Empty*, který slouží k vytvoření prázdného výčtu konkrétního typu *TResult*. Tato operace se může hodit při inicializaci prázdných sekvencí.

```
public static IEnumerable<TResult> Empty<TResult>();
```

Výpis 3.50 nabízí příklad, který používá operátor *Empty*, aby naplnil prázdný výčet *Customer*.

Výpis 3.50 Operátor Empty inicializující prázdnou množinu zákazníků

```
IEnumerable<Customer> customers = Enumerable.Empty<Customer>();
```

Kvantifikační operátory

Představte si, že potřebujete ověřit existenci elementů v sekvenci pomocí podmínek či výběrových pravidel. Nejprve vyberete prvky pomocí operátoru *Restriction* a poté použijete agregační operátory, kupříkladu *Count*, abyste zjistili, zdali existuje nějaký prvek, který vyhovuje podmínce. Existuje ovšem skupina operátorů, zvaných kvantifikátory, jež jsou speciálně navrženy pro ověřování podmínek existence v sekvencích.

Any

První operátor v této skupině, který si popíšeme, je metoda *Any*. Má dvě přetížení:

```
public static Boolean Any<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, Boolean> predicate);
public static Boolean Any<TSource>(
   this IEnumerable<TSource> source);
```

Jak je ze záhlaví vidět, metoda nabízí přetížení přebírající predikát. Toto přetížení vrací *true*, jestliže ve *zdrojové* sekvenci existuje prvek, který vyhovuje zadanému predikátu. Existuje i druhé přetížení, které přebírá pouze *zdrojovou* sekvenci bez predikátu. Tato metoda vrací *true*, jestliže ve *zdrojové sekvenci* existuje alespoň jeden element, případně *false*, je-li *zdrojová* sekvence prázdná. Kvůli optimalizaci končí metoda *Any* ihned, jakmile je dostupný výsledek. Ve výpisu 3.51 vidíte ukázku, která zjišťuje, zdali mezi všemi objednávkami od zákazníků existuje nějaká objednávka produktu č. 1 (*IdProduct == 1*).

Výpis 3.51 Operátor Any aplikovaný na objednávky všech zákazníků, kde zjišťuje, zdali existují objednávky s IdProduct == 1

```
bool result =
  (from c in customers
    from o in c.Orders
    select o)
  .Any(o => o.IdProduct == 1);

result = Enumerable.Empty<Order>().Any();
```

V tomto příkladě operátor vyhodnocuje položky pouze do okamžiku, kdy dojde k nalezení první objednávky vyhovující podmínce (*IdProduct == 1*). Druhý příklad ve výpisu 3.51 ilustruje triviální použití operátoru *Any* s výsledkem *false*, neboť se zde používá operátor *Empty*, jejž jsme si ukázali dříve.

Důležité

Operátor *Any*, aplikovaný na prázdnou sekvenci, vrátí vždy hodnotu *false*. Vnitřní implementace operátoru v LINQ pro objekty vyhodnocuje všechny položky *zdrojové* sekvence. Jakmile dojde k nalezení elementu vyhovujícího predikátu, vrací operátor hodnotu *true*. Je-li sekvence prázdná, predikát se nikdy nezavolá a vrátí se hodnota *false*.

All

Chcete-li zjistit, zdali všechny položky v dané sekvenci vyhovují podmínce filtru, poslouží vám operátor *All*. Ten vrací hodnotu *true* pouze v situaci, kdy podmínce vyhovují všechny elementy ve *zdrojové* sekvenci:

```
public static Boolean All<TSource>(
   this IEnumerable<TSource> source,
   Func<TSource, Boolean> predicate);
```

Například ve výpisu 3.52 zjišťujeme, zdali má každá objednávka kladnou hodnotu množství.

Výpis 3.52 Operátor All aplikovaný na objednávky všech zákazníků, kde zjišťuje znaménko množství v objednávkách

```
bool result =
  (from c in customers
    from o in c.Orders
    select o)
  .All(o => o.Quantity > 0);

result = Enumerable.Empty<Order>().All(o => o.Quantity > 0);
```

Důležité

Operátor *All*, aplikovaný na prázdnou sekvenci, vrací vždy hodnotu *true*. Vnitřní implementace operátoru v LINQ pro objekty vyhodnocuje všechny položky *zdrojové* sekvence. Jakmile dojde k nalezení elementu nevyhovujícího predikátu, vrací operátor hodnotu *false*. Je-li sekvence prázdná, predikát se nikdy nezavolá a vrátí se hodnota *true*.

Contains

Poslední kvantifikační operátor je rozšiřující metoda *Contains*, která určuje, zdali *zdrojová* sekvence obsahuje konkrétní položku:

```
public static Boolean Contains<TSource>(
   this IEnumerable<TSource> source,
   TSource value);
public static Boolean Contains<TSource>(
   this IEnumerable<TSource> source,
   TSource value,
   IEqualityComparer<TSource> comparer)
```

V implementaci LINQ pro objekty zkouší tato metoda použít metodu *Contains* rozhraní *ICollection<T>*, pokud *zdrojová* sekvence toto rozhraní obsahuje. Jestliže není rozhraní *ICollection<T>* implementováno, vyhodnocuje metoda *Contains* všechny položky ve *zdrojové* sekvenci a každou z nich porovnává se zadanou hodnotou, *value*, typu *TSource*. Používá vlastní porovnávací třídu, *comparer*, pokud ji zadáte prostřednictvím atributu druhého přetížení metody. V opačném případě se použije standardní *EqualityComparer<T>.Default*.

Ve výpisu 3.53 vidíte ukázku použití metody *Contains* k ověření existence konkrétní objednávky v kolekci objednávek určitého zákazníka.

Výpis 3.53 Operátor Contains aplikovaný na objednávky prvního zákazníka

```
// první zákazník má objednávku s následujícími hodnotami
var orderOfProductOne = new Order {IdOrder = 1, Quantity = 3, IdProduct =
  1 , Shipped = false, Month = "January"};
bool result = customers[0].Orders.Contains(orderOfProductOne);
```

Narozdíl od očekávání bude po provedení kódu ve výpisu 3.53 výsledek roven *false*, i když u prvního zákazníka existuje objednávka, která obsahuje v jednotlivých polích odpovídající hodnoty. Metoda *Contains* vrací *true* pouze tehdy, když použijete tentýž objekt jako srovnávaný. V opačném případě byste potřebovali vlastní porovnávač nebo pro typ *Order* (referenční typ, který překrývá metody *GetHashCode* a *Equals*, nebo hodnotový typ, jak jsme si již řekli) použít sémantiku hodnotového typu a s jejich pomocí hledat odpovídající objednávku v sekvenci.

Dělicí operátory

Operace výběru a filtrování je někdy potřeba aplikovat pouze na podmnožinu elementů zdrojové sekvence. Můžete například chtít vybrat pouze prvních *n* elementů, které vyhovují dané podmínce. Lze použít operátory *Where* a *Select* s indexem v predikátu, ale tento přístup není

vždy užitečný a intuitivní. Je lepší pracovat se specializovanými operátory určenými pro tento druh operací, protože podobné úkoly se vyskytují velice často.

K těmto účelům slouží množina dělicích operátorů. Operátory *Take* a *TakeWhile* vybírají prvních *n* elementů či první elementy, které vyhovují zadanému predikátu. Operátory *Take* a *TakeWhile* doplňuje dvojice operátoru *Skip* a *SkipWhile*, které přeskakují prvních *n* elementů či první elementy, které vyhovují zadanému predikátu.

Take

Začněme operátory *Take* a *TakeWhile*:

```
public static IEnumerable<TSource> Take<TSource>(
    this IEnumerable<TSource> source,
    Int32 count);
```

Operátor *Take* má povinný parametr *count*, který představuje počet položek, jež se mají vybrat ze *zdrojové* sekvence. Záporné hodnoty *count* znamenají, že výsledek bude prázdný, hodnoty vyšší než velikost sekvence způsobí návrat celé *zdrojové* sekvence. Tato metoda je vhodná pro všechny dotazy, v nichž potřebujete vybrat horních *n* položek. Tuto metodu můžete například použít k výběru horních *n* zákazníků vybraných na základě velikosti jejich objednávek, viz výpis 3.54.

Výpis 3.54 Operátor Take, vybírající dva nejvýše stojící zákazníky v pořadí podle velikosti objednávek

```
var topTwoCustomers =
  (from c in customers
   join o in (
        from c in customers
          from o in c.Orders
          join p in products
               on o.IdProduct equals p.IdProduct
          select new { c.Name, OrderAmount = o.Quantity * p.Price }
        ) on c.Name equals o.Name
        into customersWithOrders
   let TotalAmount = customersWithOrders.Sum(o => o.OrderAmount)
   orderby TotalAmount descending
   select new { c.Name, TotalAmount }
  ).Take(2);
```

Je zjevné, že klauzule operátoru *Take* je velmi prostá, zatímco dotaz jako celek je složitější. Dotaz obsahuje několik základních elementů a operátorů, o nichž jsme mluvili dříve. Klauzule *let* je kromě *Take* jedinou klauzulí, kterou jsme dosud nepoužili v dotazu v LINQ pro objekty. Z kapitoly 2 víte, že pomocí klíčového slova *let* lze definovat nový název pro hodnotu či proměnnou obsahující vzorec. V této ukázce potřebujeme sečíst všechny hodnoty objednávek pro jednotlivé zákazníky, které promítneme do výsledného anonymního typu. Tutéž hodnotu zároveň použijeme pro podmínku třídění. Proto jsme definovali proměnnou *TotalAmount*, abychom vzorec nemuseli psát dvakrát.

TakeWhile

Operátor *TakeWhile* pracuje podobně jako operátor *Take*, ale namísto počítání vybírá položky na základě vzorce. Zde jsou verze metody:

```
public static IEnumerable<TSource> TakeWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
public static IEnumerable<TSource> TakeWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Int32, Boolean> predicate);
```

Metoda má dvě přetížení. První přebírá predikát, který se vyhodnotí pro každou položku ve *zdrojové* sekvenci. Metoda vyčísluje *zdrojovou* sekvenci a vybírá položky, pro něž je predikát roven *true;* vyhodnocování se zastaví ve chvíli, kdy začne být výsledek predikátu roven *false* nebo když metoda dojde na konec *zdroje*. Druhé přetížení vyžaduje dále pro predikát index počínající nulou, od nějž má dotaz začít vyhodnocovat *zdrojovou* sekvenci.

Představte si situaci, kdy chcete zjistit své nejlepší zákazníky a vygenerovat seznam, který určuje minimální agregovanou výši objednávek. Problém vypadá podobně jako ten, který jsme již řešili pomocí operátoru *Take* ve výpisu 3.54, ale nevíme, kolik zákazníků bychom měli prozkoumat. Tento problém nám pomůže vyřešit operátor *TakeWhile*, v němž použijeme predikát počítající agregovanou výši objednávek a využívající toto číslo k zastavení výpočtu při dosažení cílové hodnoty. Výsledný dotaz vidíte ve výpisu 3.55.

Výpis 3.55 Operátor TakeWhile vybírající nejlepší zákazníky, kteří generují 80 procent všech objednávek

```
// globalAmount je celková hodnota všech objednávek
var limitAmount = globalAmount * 0.8m;
var aggregated = 0m;
var topCustomers =
  (from c in customers
   join o in (
      from c in customers
         from o in c.Orders
         join p in products
              on o.IdProduct equals p.IdProduct
         select new { c.Name, OrderAmount = o.Quantity * p.Price }
      ) on c.Name equals o.Name
      into customersWithOrders
   let TotalAmount = customersWithOrders.Sum(o => o.OrderAmount)
   orderby TotalAmount descending
   select new { c.Name, TotalAmount }
  )
  .TakeWhile( X => {
                              bool result = aggregated < limitAmount;
                              aggregated += X.TotalAmount;
                              return result;
                         } );
```

Skip a SkipWhile

Záhlaví operátorů *Skip* a *SkipWhile* je velice podobné operátorům *Take* a *TakeWhile*:

```
public static IEnumerable<TSource> Skip<TSource>(
    this IEnumerable<TSource> source,
    Int32 count);
public static IEnumerable<TSource> SkipWhile<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
public static IEnumerable<TSource> SkipWhile<TSource>(
```

```
    this IEnumerable<TSource> source,
    Func<TSource, Int32, Boolean> predicate);
```

Jak jsme si již řekli, tyto operátory tvoří doplněk ke dvojici operátorů *Take* a *TakeWhile*. Následující kód vrací v podstatě celou sekvenci zákazníků:

```
var result = customers.Take(3).Union(customers.Skip(3));
var result = customers.TakeWhile(p).Union(customers.SkipWhile(p));
```

Jediným zajímavým bodem je, že *SkipWhile* přeskakuje položky ve *zdrojové* sekvenci, dokud se predikát rovná *true*, položky začíná vybírat ve chvíli, kdy se predikát začne rovnat *false*, a vyhodnocování predikátu pro všechny zbývající položky zruší.

Operátory pro elementy

Operátory pro elementy jsou určeny pro práci s jednotlivými elementy v sekvenci. Jsou navrženy tak, aby vybíraly konkrétní element podle pozice či predikátu, a nikoli pomocí výchozí hodnoty v případě chybějících elementů.

First

Začněme metodou *First*, která vybírá první element v sekvenci pomocí predikátu či pravidla pro pozici:

```
public static TSource First<TSource>(
    this IEnumerable<TSource> source);
public static TSource First<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

První přetížení vrací první element ve *zdrojové* sekvenci, zatímco druhé přetížení používá k určení prvního elementu, který se má vrátit, predikát. Jestliže neexistuje žádný element, který by predikátu vyhovoval, nebo ve *zdrojové* sekvenci nejsou vůbec žádné elementy, operátor způsobí výjimku *InvalidOperationException*. Výpis 3.56 představuje ukázku práce s operátorem *First*.

Výpis 3.56 Operátor First, vybírající prvního zákazníka z USA

```
var item = customers.First(c => c.Country == Countries.USA);
```

Tento dotaz bychom samozřejmě mohli zadat pomocí operátorů *Where* a *Take*. Nicméně metoda *First* lépe vyjadřuje záměr dotazu a také zajišťuje jediný (a částečný) průchod *zdrojovou* sekvencí.

FirstOrDefault

Potřebujete-li najít první element pouze tehdy, když existuje, aniž by v případě selhání došlo k výjimce, poslouží vám metoda *FirstOrDefault*. Tato metoda pracuje podobně jako *First*, avšak neexistuje-li žádný element, který by vyhovoval zadanému predikátu, nebo je-li *zdrojová* sekvence prázdná, vrátí metoda výchozí hodnotu:

```
public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source);
public static TSource FirstOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

Výchozí návratová hodnota pro případ prázdného *zdroje* je *default(TSource)*, která vrací *null* pro referenční typy a typy povolující hodnotu *null*. Není-li zadán parametr predikátu, metoda vrátí první element *zdroje*, pokud existuje. Příklady vidíte ve výpisu 3.57.

Výpis 3.57 Ukázky syntaxe operátoru FirstOrDefault

```
var item = customers.FirstOrDefault(c => c.City == "Las Vegas");
Console.WriteLine(item == null ? "null" : item.ToString()); // vrací null

IEnumerable<Customer> emptyCustomers = Enumerable.Empty<Customer>();
item = emptyCustomers.FirstOrDefault(c => c.City == "Las Vegas");
Console.WriteLine(item == null ? "null" : item.ToString()); // vrací null
```

Last a LastOrDefault

Operátory *Last* a *LastOrDefault* představují protějšek operátorů *First* a *FirstOrDefault*. Mají záhlaví uvedená níže, jejichž syntaxe odpovídá syntaxi operátorů *First* a *FirstOrDefault*:

```
public static TSource Last<TSource>(
    this IEnumerable<TSource> source);
public static TSource Last<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source);
public static TSource LastOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

Tyto metody fungují podobně jako operátory *First* a *FirstOrDefault*. Jediným rozdílem je, že vybírají poslední element ve *zdroji* namísto prvního.

Single

Kdykoliv potřebujete vybrat ze *zdrojové* sekvence jednu konkrétní položku, použijte operátory *Single* či *SingleOrDefault*:

```
public static TSource Single<TSource>(
    this IEnumerable<TSource> source);
public static TSource Single<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

Nezadáte-li predikát, operátor *Single* vybírá ze *zdrojové* sekvence první a jediný element. V opačném případě vybere jedinečný element, který vyhovuje podmínce predikátu. Jestliže není zadán predikát a zdrojová sekvence obsahuje více než jednu položku, dojde k vyvolání výjimky *InvalidOperationException*. Jestliže je zadán predikát a neexistuje žádný odpovídající element nebo je ve *zdroji* více než jeden odpovídající element, dojde opět k vyvolání výjimky *InvalidOperationException*. Ukázky vidíte ve výpisu 3.58.

Výpis 3.58 Ukázky syntaxe operátoru Single

```
// vrací produkt 1
var item = products.Single(p => p.IdProduct == 1);
Console.WriteLine(item == null ? "null" : item.ToString());

// InvalidOperationException
```

```
item = products.Single();
Console.WriteLine(item == null ? "null" : item.ToString());

// InvalidOperationException
IEnumerable<Product> emptyProducts = Enumerable.Empty<Product>();
item = emptyProducts.Single(p => p.IdProduct == 1);
Console.WriteLine(item == null ? "null" : item.ToString());
```

SingleOrDefault

Operátor *SingleOrDefault* vrací v případě, že *zdrojem* je prázdná sekvence nebo neexistuje element vyhovující podmínce, výchozí výstupní hodnotu. Jeho záhlaví jsou podobná operátoru *Single*:

```
public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source);
public static TSource SingleOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Func<TSource, Boolean> predicate);
```

Výchozí návratovou hodnotu metody je *default(TSource),* podobně jako u rozšiřujících metod *FirstOrDefault* a *LastOrDefault*.

Důležité

Výchozí hodnota se dostane do výsledku pouze tehdy, jestliže žádný element nevyhovuje predikátu. Jestliže *zdrojová* sekvence obsahuje více než jednu vyhovující položku, dojde k výjimce *InvalidOperationException*.

ElementAt a ElementAtOrDefault

Jestliže potřebujete ze sekvence získat položku na konkrétní pozici, poslouží vám metody *ElementAt* či *ElementAtOrDefault*:

```
public static TSource ElementAt<TSource>(
    this IEnumerable<TSource> source,
    Int32 index);
public static TSource ElementAtOrDefault<TSource>(
    this IEnumerable<TSource> source,
    Int32 index);
```

Metoda *ElementAt* má povinný parametr *index*, jenž reprezentuje pozici elementu, který se má vybrat. Parametr *index* začíná od nuly, chcete-li tedy vybrat třetí element, musíte vložit hodnotu 2. Je-li hodnota *indexu* záporná nebo vyšší než velikost *zdrojové* sekvence, dojde k výjimce *ArgumentOutOfRangeException*. Metoda *ElementAtOrDefault* se od metody *ElementAt* liší v tom, že pro referenční typy a typy povolující hodnotu *null* vrací v případě, že je zadán záporný *index* či *index* vyšší, než je velikost *zdrojové* sekvence, výchozí hodnotu – *default(TSource)*. Příklady použití těchto operátorů ukazuje výpis 3.59.

Výpis 3.59 Ukázky syntaxe operátorů ElementAt a ElementAtOrDefault

```
// vrací produkt 2
var item = products.ElementAt(2);
Console.WriteLine(item == null ? "null" : item.ToString());
```

```
// vrací null
item = Enumerable.Empty<Product>().ElementAtOrDefault(6);
Console.WriteLine(item == null ? "null" : item.ToString());

// vrací null
item = products.ElementAtOrDefault(6);
Console.WriteLine(item == null ? "null" : item.ToString());
```

DefaultIfEmpty

Operátor *DefaultIfEmpty* vrací výchozí element pro prázdnou sekvenci:

```
public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source);
public static IEnumerable<TSource> DefaultIfEmpty<TSource>(
    this IEnumerable<TSource> source,
    TSource defaultValue);
```

Ve výchozím nastavení vrací operátor seznam položek *zdrojové* sekvence. V případě, že je zdroj prázdný, vrátí operátor výchozí hodnotu, která je v případě první verze metody rovna *default(TSource)* a pro druhou verzi metody jde o hodnotu *defaultValue*.

Definice specifické výchozí hodnoty se může hodit za mnoha okolností. Představte si například situaci, kdy máte veřejnou statickou vlastnost s názvem *Empty*, která vrací prázdnou instanci objektu *Customer*, podobně jako následující ukázka:

```
private static volatile Customer empty;
private static Object emptySyncLock = new Object();

public static Customer Empty {
    get {
        // (navrhovy vzor singleton - bezpecny pro multithreading)
        if (empty == null) {
            lock (emptySyncLock) {
                if (empty == null) {
                    empty = new Customer();
                    empty.Name = String.Empty;
                    empty.Country = Countries.Italy;
                    empty.City = String.Empty;
                    empty.Orders = (new List<Order>(Enumerable.Empty<Order>())).
                    ToArray();
                }
            }
        }
        return (empty);
    }
}
```

Někdy je to užitečné, obzvláště v testovacím kódu modulu. Další taková situace nastane, když dotaz používá operátor *GroupJoin* a vytváří tak levé vnější spojení. Možné hodnoty *null* ve výsledku lze nahradit výchozí hodnotou, kterou si stanoví autor dotazu.

Ve výpisu 3.60 vidíte práci s operátorem *DefaultIfEmpty*, včetně použití vlastní výchozí hodnoty *Customer.Empty*.

Výpis 3.60 Ukázka syntaxe operátoru DefaultIfEmpty, s výchozí hodnotou default(T) i vlastní výchozí hodnotou.

```
var expr = customers.DefaultIfEmpty();

var customers = Enumerable.Empty<Customer>(); // prázdné pole
IEnumerable<Customer> customersEmpty =
  customers.DefaultIfEmpty(Customer.Empty);
```

Další operátory

Náš výklad ohledně operátorů LINQ pro objekty dokončíme popisem několika posledních rozšiřujících metod.

Concat

První je slučovací operátor s názvem *Concat*. Jak již název napovídá, připojuje zkrátka jednu sekvenci ke druhé, což naznačuje jeho záhlaví:

```
public static IEnumerable<TSource> Concat<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second);
```

Jediným požadavkem na parametry metody *Concat* je, aby jejich členy byly téhož typu, *TSource*. Tuto metodu lze použít na připojení libovolné sekvence typu *IEnumerable<T>* k jiné sekvenci téhož typu. Výpis 3.61 ukazuje příklad uživatelského spojení.

Výpis 3.61 Operátor Concat, spojující zákazníky z Itálie se zákazníky ze Spojených států

```
var italianCustomers =
   from c in customers
   where c.Country == Countries.Italy
   select c;

var americanCustomers =
   from c in customers
   where c.Country == Countries.USA
   select c;

var expr = italianCustomers.Concat(americanCustomers);
```

Sequence Equal

Dalším užitečným operátorem je operátor rovnosti, jejž poskytuje rozšiřující metoda *SequenceEqual*:

```
public static Boolean SequenceEqual<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second);
public static Boolean SequenceEqual<TSource>(
   this IEnumerable<TSource> first,
   IEnumerable<TSource> second,
   IEqualityComparer<TSource> comparer);
```

Tato metoda porovnává jednotlivé položky v první sekvenci s odpovídajícími položkami ve druhé sekvenci. Jestliže mají obě sekvence přesně stejný počet prvků a totožné položky na všech pozicích, považují se obě sekvence za identické. Nezapomeňte na možné problémy se

sémantikou referenčního typu, které mohou v tomto typu srovnávání vyvstat. Můžete zvážit přepis metod *GetHashCode* a *Equal* v typu *TSource*, abyste dosáhli požadovaných výsledků operátoru, nebo můžete použít druhou verzi metody, do níž se vkládá vlastní porovnávač *IEqualityComparer<T>*.

Převodní operátory

Metody v množině převodních operátorů zahrnují *AsEnumerable, ToArray, ToList, ToDictionary, ToLookup, OfType* a *Cast*. Převodní operátory mají za cíl především řešit problémy a potřeby související s odloženým prováděním dotazů LINQ. (Více o tomto tématu najdete v kapitole 2.) Někdy je potřeba získat stabilní a neměnné výsledky z dotazu nebo chcete použít obecnou rozšiřující metodu operátoru namísto speciální metody. V následujících pasážích si popíšeme převodní operátory podrobněji.

AsEnumerable

Záhlaví metody *AsEnumerable* vypadá následovně.

```
public static IEnumerable<TSource> AsEnumerable<TSource>(
   this IEnumerable<TSource> source);
```

Operátor *AsEnumerable* vrací *zdrojovou* sekvenci jako objekt typu *IEnumerable<TSource>*. Tento druh „letmého převodu" umožňuje volat nad *zdrojem* obecné rozšiřující metody, i když pro typ zdroje existují specifické verze těchto metod.

Mějme vlastní rozšiřující metodu *Where* pro typ *Customers*, obdobnou metodě definované ve výpisu 3.62.

Výpis 3.62 Vlastní rozšiřující metoda Where, definovaná pro typ Customers

```
public class Customers : List<Customer> {
  public Customers(IEnumerable<Customer> items): base(items) {
  }
}

public static class CustomersExtension {
  public static Customers Where(this Customers source,
    Func<Customer, Boolean> predicate) {
    Customers result = new Customers();

    Console.WriteLine("Vlastní rozšiřující metoda Where");
    foreach (var item in source) {
      if (predicate(item))
        result.Add(item);
    }
    return result;
  }
}
```

V tomto ukázkovém kódu si všimněte volání metody *Console.WriteLine*.

> **Důležité**
>
> Ve skutečných řešeních byste patrně pro zobrazení výsledků této rozšiřující metody namísto explicitního seznamu použili vlastní iterátor, ale kvůli jednoduchosti jsme se rozhodli jej v tomto krátkém příkladě nepoužívat.

Ve výpisu 3.63 vidíte ukázku dotazovacího výrazu spuštěného nad instancí typu *Customers*.

Výpis 3.63 Dotazovací výraz nad seznamem zákazníků

```
Customers customersList = new Customers(customers);

var expr =
  from c in customersList
  where c.Country == Countries.Italy
  select c;

foreach (var item in expr) {
  Console.WriteLine(item);
}
```

Výstup z tohoto ukázkového kódu by vypadal takto:

```
Name: Paolo - City: Brescia - Country: Italy - Orders Count: 2
Name: Marco - City: Torino - Country: Italy - Orders Count: 2
```

Jak vidíte, výstup začíná voláním metody *Console.WriteLine* v naší vlastní rozšiřující metodě *Where*. Ve skutečnost, jak jsme si již řekli v kapitole 2, dotazy LINQ se překládají na odpovídající rozšiřující metody a pro typ *Customers* je rozšiřující metodou ta, kterou jsme si sami nadefinovali.

Nyní si představte, že chcete napsat dotaz nad instancí typu *Customers* bez použití vlastní rozšiřující metody. Namísto toho chcete použít standardní operátor *Where*, definovaný pro typ *IEnumerable<T>*. Tento požadavek vám pomůže splnit rozšiřující metoda *AsEnumerable*, což vidíte ve výpisu 3.64.

Výpis 3.64 Dotazovací výraz nad seznamem zákazníků převedený pomocí operátoru AsEnumerable

```
Customers customersList = new Customers(customers);

var expr =
  from c in customersList.AsEnumerable()
  where c.City == "Brescia"
  select c;

foreach (var item in expr) {
  Console.WriteLine(item);
}
```

Kód ve výpisu 3.64 používá standardní operátor *Where*, definovaný pro typ *IEnumerable<T>* ve jmenném prostoru *System.Linq.Enumerable*.

ToArray a ToList

Dva další užitečné převodní operátory jsou *ToArray* a *ToList*. Převádějí zdrojovou sekvenci typu *IEnumerable<T>* na pole typu *TSource(TSource[])* resp. na generický seznam typu *TSource(List<TSource>)*:

```
public static TSource[] ToArray<TSource>(
   this IEnumerable<TSource> source);
public static List<TSource> ToList<TSource>(
   this IEnumerable<TSource> source);
```

Výsledkem těchto operátorů jsou snímky sekvence. Jestliže je použijete uvnitř dotazovacího výrazu, výsledky budou stabilní a neměnné, i když dojde ke změně ve *zdrojové* sekvenci. Ukázku použití operátoru *ToList* vidíte ve výpisu 3.65.

Výpis 3.65 Dotazovací výraz nad neměnným seznamem zákazníků získaným pomocí operátoru ToList

```
List<Customer> customersList = new List<Customer>(customers);

var expr = (
  from c in customersList
  where c.Country == Countries.Italy
  select c).ToList();

foreach (var item in expr) {
  Console.WriteLine(item);
}
```

Uvedené metody najdou rovněž uplatnění v situaci, kdy potřebujete vyčíslit výsledky dotazu vícekrát, neboť dotaz se provede pouze jednou a zvýší se tak výkon. Podívejte se na ukázku 3.66. Je patrně neefektivní obnovovat neustále seznam produktů pro spojení s objednávkami. Proto je možné vytvořit „kopii“ dotazu na produkty.

Výpis 3.66 Dotazovací výraz, který používá operátor ToList k vytvoření kopie výsledků dotazu do produktů

```
var productsQuery =
  (from p in products
  where p.Price >= 30
  select p)
  .ToList();

var ordersWithProducts =
  from c in customers
    from o in c.Orders
    join p in productsQuery
         on o.IdProduct equals p.IdProduct
    select new { p.IdProduct, o.Quantity, p.Price,
             TotalAmount = o.Quantity * p.Price};

foreach (var order in ordersWithProducts) {
  Console.WriteLine(order);
}
```

Při opakovaném výpočtu výrazu *ordersWithProducts* – například v bloku *foreach* – se výraz *productsQuery* nebude vyčíslovat znovu.

ToDictionary

Dalším operátorem z této množiny je rozšiřující metoda *ToDictionary*. Vytváří instanci typu *Dictionary<TKey, TSource>*. Predikát *keySelector* určuje klíč pro každou položku. Predikát *elementSelector* – je-li zadán – se používá k získávání jednotlivých položek. Uvedené predikáty se nacházejí v následujících verzích metody:

```
public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector);
public static Dictionary<TKey, TSource> ToDictionary<TSource, TKey>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    IEqualityComparer<TKey> comparer);
public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector);
public static Dictionary<TKey, TElement> ToDictionary<TSource, TKey, TElement>(
    this IEnumerable<TSource> source,
    Func<TSource, TKey> keySelector,
    Func<TSource, TElement> elementSelector,
    IEqualityComparer<TKey> comparer);
```

Když tato metoda sestavuje výsledný slovník, předpokládá, že všechny klíče získané voláním predikátu *keySelector* jsou jedinečné. V případě duplicit v klíčích dojde k vyvolání výjimky *ArgumentException*. Hodnoty klíčů se porovnávají pomocí porovnávače v parametru *comparer*, je-li zadán, nebo v opačném případě pomocí porovnávače *EqualityComparer<TKey>.Default*. Uvedený operátor použijeme ve výpisu 3.67 pro vytvoření slovníku se zákazníky.

Výpis 3.67 Ukázka aplikace operátoru ToDictionary na zákazníky

```
var customersDictionary =
  customers
  .ToDictionary(c => c.Name,
                c => new {c.Name, c.City});
```

První parametr operátoru je predikát *keySelector*, který vybírá do klíče jména zákazníků. Druhým parametrem je *elementSelector*, jenž vytváří anonymní typ skládající se ze jména zákazníka a z města. Výsledek dotazu z výpisu 3.67 vypadá takto:

```
[Paolo, { Name = Paolo, City = Brescia }]
[Marco, { Name = Marco, City = Torino }]
[James, { Name = James, City = Dallas }]
[Frank, { Name = Frank, City = Seattle }]
```


Důležité

Podobně jako operátory *ToList* a *ToArray*, i operátor *ToDictionary* se odkazuje na položky ve zdrojové *sekvenci*, jde-li o referenční typy. Metoda *ToDictionary* ve výpisu 3.67 efektivně řeší dotazovací výraz a vytváří výstupní slovník. Proto se výstup *customersDictionary* nechová jako dotazovací výrazy, jejichž vykonávání je odložené; výsledek se vytvoří při spuštění příkazu.

ToLookup

Dalším převodním operátorem je *ToLookup*, který lze použít na vytvoření výčtů typu *Lookup<K, T>*, jehož definice vypadá takto:

```
public class Lookup<K, T> : IEnumerable<IGrouping<K, T>> {
   public int Count { get; }
   public IEnumerable<T> this[K key] { get; }
   public bool Contains(K key);
   public IEnumerator<IGrouping<K, T>> GetEnumerator();
}
```

Každý objekt tohoto typu představuje slovník jedna-mnoho, který představuje dvojici klíčů a sekvenci položek, což je podobné výsledkům metody *GroupJoin*. Operátor má tyto verze:

```
public static Lookup<TKey, TSource> ToLookup<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector);
public static Lookup<TKey, TSource> ToLookup<TSource, TKey>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   IEqualityComparer<TKey> comparer);
public static Lookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector);
public static Lookup<TKey, TElement> ToLookup<TSource, TKey, TElement>(
   this IEnumerable<TSource> source,
   Func<TSource, TKey> keySelector,
   Func<TSource, TElement> elementSelector,
   IEqualityComparer<TKey> comparer);
```

Stejně jako u operátoru *ToDictionary*, i zde existují predikáty *keySelector* a *elementSelector* a *comparer*. Ukázka ve výpisu 3.68 demonstruje, jak se tato metoda používá na získání všech objednávek pro jednotlivé produkty.

Výpis 3.68 Ukázka operátoru ToLookup, jenž seskupí objednávky podle produktů

```
var ordersByProduct =
  (from c in customers
    from o in c.Orders
    select o)
  .ToLookup(o => o.IdProduct);

Console.WriteLine( "\n\nPočet objednávek pro produkt 1: {0}\n",
                   ordersByProduct[1].Count());

foreach (var product in ordersByProduct) {
  Console.WriteLine("Produkt: {0}", product.Key);
  foreach(var order in product) {
    Console.WriteLine(" {0}", order);
  }
}
```

Jak vidíte, s instancí typu *Lookup<K, T>* lze pracovat prostřednictvím klíče položky (*orders-ByProduct[1]*) nebo ve výčtu (smyčka *foreach*). Následuje výstup tohoto příkladu:

```
Počet objednávek pro produkt 1: 2
```

```
Produkt: 1
   IdOrder: 1 - IdProduct: 1 - Quantity: 3 - Shipped: False - Month: January
   IdOrder: 3 - IdProduct: 1 - Quantity: 10 - Shipped: False - Month: July
Produkt: 2
   IdOrder: 2 - IdProduct: 2 - Quantity: 5 - Shipped: True - Month: May
Produkt: 3
   IdOrder: 4 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
   IdOrder: 5 - IdProduct: 3 - Quantity: 20 - Shipped: True - Month: December
Produkt: 5
   IdOrder: 6 - IdProduct: 5 - Quantity: 20 - Shipped: False - Month: July
```

OfType a Cast

Poslední dva operátory této množiny jsou *OfType* a *Cast*. První filtruje zdrojovou sekvenci a vrací pouze položky typu *TResult*. Hodí se v situacích, kdy máme sekvence položek odlišných typů. Když například pracujete objektově, může se stát, že budete mít společnou základní třídu a konkrétní specializace v odvozených třídách:

```
public static IEnumerable<TResult> OfType<TResult>(
   this IEnumerable source);
```

Jestliže vložíte typ *TResult*, jenž není typem žádné zdrojové položky, vrátí operátor prázdnou sekvenci.

Operátor *Cast* prochází zdrojovou sekvenci a snaží se vrátit všechny položky převedené na typ *TResult*. V případě selhání dojde k vyvolání výjimky *InvalidCastException*. (Viz výpis 2.4, v němž se nachází ukázka použití tohoto operátoru.)

```
public static IEnumerable<TResult> Cast<TResult>(
   this IEnumerable source);
```

Vzhledem k záhlavím metod, které přebírají libovolnou sekvenci *IEnumerable*, lze tyto dvě metody použít na převod starých negenerických typů na novější typy *IEnumerable<T>*. Takový převod umožňuje dotazovat se v LINQ do těchto typů, i když tyto typy s LINQ nespolupracují.

> **Důležité**
>
> **Každá položka vrácená operátory *OfType* a *Cast* je odkazem na původní objekt, a nikoli jeho kopii. Operátor *OfType* nevytváří kopii zdroje, ale načítá zdroj pokaždé, když vypisujete výsledek operátoru. Tím se tyto operátory liší od ostatních převodních operátorů.**

Souhrn

V této kapitole jsme probrali principy dotazovacích výrazů LINQ a pravidla jejich syntaxe. Probrali jsme dotazovací operátory i převodní operátory. Jako referenční implementaci jsme používali LINQ pro objekty, ale veškeré probrané principy platí i pro další implementace LINQ, o nichž budeme hovořit v následujících kapitolách.

ČÁST II

LINQ pro relační data

KAPITOLA 4

LINQ pro SQL: Dotazování na data

První a nejzřejmější aplikací LINQ je dotazování do externí relační databáze. LINQ pro SQL je součást projektu LINQ, která nabízí možnost dotazování do relační databáze Microsoft SQL Serveru a také objektový model vycházející z dostupných entit. Jinými slovy, můžete definovat množinu objektů, které představují tenkou abstraktní vrstvu nad relačními daty, a do tohoto objektového modelu se dotazovat pomocí dotazů LINQ, které se ve stroji LINQ pro SQL převádějí na odpovídající dotazy SQL. LINQ pro SQL podporuje Microsoft SQL Server od verze SQL Serveru 2000 a Microsoft SQL Server Compact počínaje verzí 3.5.

V LINQ pro SQL lze napsat takovýto prostý dotaz:

```
var query =
   from c in Customers
   where c.Country == "USA"
         && c.State == "WA"
   select new {c.CustomerID, c.CompanyName, c.City };
```

Tento dotaz se převádí na dotaz SQL, jenž se pošle do relační databáze:

```
SELECT CustomerID, CompanyName, City
FROM Customers
WHERE Country = 'USA'
   AND Region = 'WA'
```


Důležité

Dotazy SQL generované v LINQ, které si budeme ukazovat v této kapitole, jsou pouze ilustrativní. Microsoft si vyhrazuje právo nezávisle definovat SQL generované v LINQ a občas v textu použijeme zjednodušené dotazy. Neměli byste se tedy na vypisované dotazy SQL spoléhat.

V tuto chvíli vás možná napadá několik otázek. Jednak, jak lze psát dotaz LINQ pomocí názvů objektů validovaných kompilátorem? Dále, kdy se generuje z dotazu LINQ dotaz SQL? A za třetí, kdy se dotaz SQL provádí? Abyste poznali odpovědi na tyto otázky, musíte pochopit model entit v LINQ pro SQL a také princip odloženého provádění dotazů.

Entity v LINQ pro SQL

Libovolná externí data je nutné popsat příslušnými metadaty, navázanými na definice tříd. Každá tabulka musí mít odpovídající třídu s příslušnými atributy. Tato třída odpovídá řádku dat a popisuje všechny sloupce jako datové členy definovaného typu. Typem může být úplný nebo částečný popis existující fyzické tabulky, pohledu či výstupu z uložené procedury. Uvnitř dotazu LINQ pro projekci či filtrování je možné používat pouze popsaná pole. Výpis 4.1 ukazuje definici krátké a jednoduché entity.

Důležité

Do svého projektu musíte vložit sestavení *System.Data.Linq*, abyste mohli používat třídy a atributy LINQ pro SQL. Atributy používané ve výpisu 4.1 mají své definice ve jmenném prostoru *System.Data.Linq.Mapping*.

Výpis 4.1 Definice entity pro LINQ pro SQL

```
using System.Data.Linq.Mapping;

[Table(Name="Customers")]
public class Customer {
    [Column] public string CustomerID;
    [Column] public string CompanyName;
    [Column] public string City;
    [Column(Name="Region")] public string State;
    [Column] public string Country;
}
```

Typ *Customer* stanovuje obsah řádku a každé pole či vlastnost s atributem *Column* odpovídá sloupci v relační tabulce. Parametr *Name* může obsahovat název sloupce, jenž se odlišuje od názvu datového členu. (V našem příkladě odpovídá název *State* sloupci Region v příslušné tabulce.) Atribut *Table* říká, že třída je entitou reprezentující data v databázové tabulce; vlastnost *Name* může obsahovat název tabulky, který se odlišuje od názvu entity. Pro název třídy se obvykle používá jednotné číslo (jeden řádek) a pro název tabulky (množina řádků) množné číslo.

Abyste mohli sestavovat dotazy LINQ pro SQL do dat zákazníků, potřebujete tabulku Customers. Správným způsobem, jak vytvořit takový typ, je použít obecnou třídu *Table*<*T*>:

```
Table<Customer> Customers = ...;
// ...
var query =
    from c in Customers
    // ...
```

Poznámka

K vytváření dotazu LINQ do tabulky *Customers* potřebujete třídu implementující rozhraní *IEnumerable*<*T*>, kde jako *T* bude figurovat typ *Customers*. Nicméně LINQ pro SQL potřebuje implementace rozšiřujících metod v jiné podobě, než v jaké je implementuje LINQ pro objekty, jež jsme viděli v kapitole 3, „LINQ pro objekty". Z tohoto důvodu musí-

te použít objekt s rozhraním *IQueryable<T>*, abyste mohli vytvářet dotazy LINQ pro SQL. Třída *Table<T>* rozhraní *IQueryable<T>* obsahuje. Aby bylo možné vložit rozšíření LINQ pro SQL, musí být ve zdrojovém kódu příkaz *using System.Data.Linq*.

Pro objekt tabulky *Customers* je potřeba založit instanci. K tomu potřebujete instanci třídy *DataContext*, která vytváří most mezi prostředím LINQ a externí relační databází. Principu práce s třídou *DataContext* je nejpodobnější spojení do databáze – ve skutečnosti je povinným parametrem při zakládání instance třídy *DataContext* připojovací řetězec či objekt *Connection*. Metoda *GetTable<T>* vrací odpovídající objekt typu *Table<T>* pro zadaný typ:

```
DataContext db = new DataContext("Database=Northwind");
Table<Customer> Customers = db.GetTable<Customer>();
```

Poznámka

Třída *DataContext* interně používá třídu *SqlConnection* z prostředí ADO.NET. Existující spojení *SqlConnection* můžete vložit do konstruktoru třídy *DataContext* a spojení, které používá instance třídy *DataContext*, můžete číst prostřednictvím vlastnosti *Connection*. Všechny služby související s databázovým spojením, například používání zásobníku spojení (connection pooling, standardně zapnuto), jsou dostupné na úrovni spojení *SqlConnection* a třída *DataContext* je přímo neimplementuje.

Výpis 4.2 ukazuje výsledný kód, když všechny uvedené prvky spojíte dohromady.

Výpis 4.2 Jednoduchý dotaz LINQ pro SQL

```
DataContext db = new DataContext( ConnectionString );
Table<Customer> Customers = db.GetTable<Customer>();

var query =
    from    c in Customers
    where   c.Country == "USA"
            && c.State == "WA"
    select new {c.CustomerID, c.CompanyName, c.City };

foreach( var row in query ) {
    Console.WriteLine( row );
}
```

Proměnná *query* se inicializuje pomocí dotazovacího výrazu, jenž vytváří strom výrazu. Strom výrazu představuje obraz výrazu v paměti, a nikoli odkaz na metodu pomocí delegáta. Když se ve smyčce *foreach* vypisují data získaná dotazem, používá se strom výrazu ke generování odpovídajícího dotazu SQL s pomocí všech metadat a informací, které se nacházejí ve třídách entit a v používané instanci třídy *DataContext*.

Poznámka

Metoda *odloženého provádění*, jež se používá v LINQ pro SQL, převádí strom výrazu na dotaz SQL, jenž je platný v příslušné relační databázi. Dotaz LINQ je funkčně ekvivalentní řetězci s dotazem SQL, přinejmenším se dvěma významnými rozdíly. Jednak je dotaz vázán na objektový model, a nikoli na databázovou strukturu. A dále, jeho reprezentace

je sémanticky logická bez nutnosti používání analyzátoru SQL a bez vazby na určitou konkrétní verzi jazyka SQL. Strom výrazu je také možné před použitím dynamicky vytvářet v paměti, což si ukážeme v kapitole 11, „Uvnitř stromů výrazů".

Návratová data z dotazu SQL, přistupující k řádku *row* ve smyčce *foreach*, se používají k naplnění promítaného anonymního typu zapsaného za klíčovým slovem *select*. V této ukázce se nikde nezakládá instance třídy *Customer*, která se v LINQ používá pouze pro analýzu metadat.

Vygenerovaný dotaz SQL si můžete prohlédnout pomocí metody *GetCommand* třídy *DataContext*. Ve vlastnosti *CommandText* navráceného objektu typu *DbCommand* se nachází vygenerovaný dotaz v jazyce SQL:

```
Console.WriteLine( db.GetCommand( query ).CommandText );
```

Jednodušší je zavolat pro dotaz LINQ pro SQL metodu *ToString*: přepsaná metoda *ToString* vrací tentýž výsledek jako příkaz *GetCommand(query).CommandText*.

```
Console.WriteLine( query );
```

Jednoduchý dotaz LINQ pro SQL ve výpisu 4.2 generuje následující dotaz SQL:

```
SELECT [t0].[CustomerID], [t0].[CompanyName], [t0].[City]
FROM [Customers] AS [t0]
WHERE ([t0].[Country] = @p0) AND ([t0].[Region] = @p1)
```

Jiným způsobem, jak sledovat všechny příkazy SQL posílané do databáze, je přiřadit hodnotu do vlastnosti *Log* třídy *DataContext*:

```
db.Log = Console.Out;
```

V následujících pasážích se podrobněji podíváme na generování tříd entit pro LINQ pro SQL.

Externí mapování

Mapování mezi entitami LINQ pro SQL a databázovou strukturou je nutné popsat prostřednictvím informací v metadatech. Ve výpisu 4.1 jste viděli atributy v definici entity, které splňovaly toto pravidlo. Ale je také možné použít externí mapovací soubor XML, který bude popisovat třídy entit a nebude tak nutné pracovat s atributy. Mapovací soubor XML vypadá podobně jako následující ukázka:

```
<Database Name="Northwind">
   <Table Name="Products">
      <Type Name="Product">
         <Column Name="ProductID" Member="ProductID"
                 Storage="_ProductID" DbType="Int NOT NULL IDENTITY"
                 IsPrimaryKey="True" IsDbGenerated="True" />
```

Element *Type* definuje vztah mezi třídou entity a atribut *Member* elementu *Column* udává odpovídající název členu ve třídě entity pro případ, že by byl odlišný od názvu sloupce v tabulce. Standardně není atribut *Member* povinný a předpokládá se, že je stejný jako atribut *Name* elementu *Column*. Tento soubor XML obvykle mívá rozšíření názvu souboru *dbml* a generují jej některé nástroje popsané v kapitole 6, „Nástroje LINQ pro SQL".

Soubor XML lze načíst pomocí instance třídy *XmlMappingSource* generované voláním její statické metody *FromXml* a předáním této instance do odvozeného konstruktoru třídy *DataContext*. Práci s touto syntaxí ukazuje následující příklad:

```
string path = "Northwind.dbml";
XmlMappingSource prodMapping =
   XmlMappingSource.FromXml(File.ReadAllText(path));
Northwind db = new Northwind(
       "Database=Test_Northwind;Trusted_Connection=yes",
       prodMapping
   );
```

Jedním z možných využití této techniky je situace, kdy je potřeba mapovat různé databáze na specifický datový model. Databáze se mohou lišit v tabulkách a názvech polí (například lokalizovaná verze databáze). Obecně byste měli o této možnosti uvažovat tehdy, když potřebujete trochu uvolnit mapovací vazbu mezi třídami entit a fyzickou datovou strukturou v databázi.

Další informace

Podrobný popis syntaxe XML v souboru .dbml je mimo možnosti této knihy. Syntaxi popisují soubory LinqToSqlMapping.xsd a DbmlSchema.xsd, jež se nacházejí v adresáři Microsoft Visual Studio 9.0\Xml\Schemas ve složce Program Files na vašem počítači, máte-li nainstalováno Visual Studio 2008. Nemáte-li ani jeden z těchto souborů, můžete si zkopírovat kód ze stránky dokumentace tohoto produktu na adrese *http://msdn2.microsoft.com/en-us/library/bb386907.aspx* a *http://msdn2.microsoft.com/en-us/library/bb399400.aspx*.

Modelování dat

Množina tříd entit, které LINQ pro SQL potřebuje, je tenká abstraktní vrstva nad relačním modelem. Každá třída entity definuje tabulku dat, do níž se lze dotazovat a kterou lze měnit. Instance měněných entit mohou promítat provedené změny do dat v relační databázi. Možnosti aktualizace dat uvidíte v kapitole 5, „LINQ pro SQL: Správa dat“. V následujících odstavcích se naučíte vytvářet datový model pro LINQ pro SQL.

DataContext

Třída *DataContext* obstarává komunikaci mezi LINQ a externími relačními datovými zdroji. Každá instance má vlastnost *Connection*, jež směřuje na relační databázi. Je typu *IDbConnection*, a proto by neměla být specificky zaměřena na konkrétní databázový produkt. Nicméně implementace LINQ pro SQL podporuje pouze databáze Microsoft SQL Server. Volba mezi konkrétními verzemi SQL Serveru závisí pouze na připojovacím řetězci, předaném do konstruktoru třídy *DataContext*.

Důležité

Architektura LINQ pro SQL podporuje mnoho poskytovatelů dat, aby bylo možné mapovat různorodé základní relační databáze. Poskytovatel je třída s rozhraním *System.Data.Linq.Provider.IProvider*. Toto rozhraní je však deklarováno jako vnitřní a není zdokumentováno. Microsoft podporuje pouze poskytovatele pro Microsoft SQL Server. .NET Framework 3.5 podporuje SQL Server 2000 a SQL Server 2005 ve 32bitové i 64bitové verzi. V čase psaní této knihy je podporován i SQL Server Compact 3.5, avšak pouze ve 32bitové verzi, protože SQL Server Compact 3.5 pro 64bitové platformy zatím není dostupný. (Pravděpodobně se dočká podpory v následujícím vydání.) Další verze SQL Serveru budou patrně rovněž podporovány.

Třída *DataContext* používá informaci v metadatech k mapování fyzické struktury relačních dat, z níž vychází generování kódu SQL. Třídu *DataContext* lze rovněž použít k volání uložené procedury a k trvalému uložení změn v instancích tříd entit do relační databáze.

Třídy pro specializovaný přístup do konkrétní databáze lze ze třídy *DataContext* odvodit. Takové třídy nabízejí snazší přístup k relačním datům, včetně členů, které představují dostupné tabulky. Pole odkazující se na existující tabulky v databázi lze definovat prostou deklarací bez nutnosti konkrétní inicializace, což dokládá následující kód:

```
public class SampleDb : DataContext {
    public SampleDb(IDbConnection connection)
           : base( connection ) {}
    public SampleDb(string fileOrServerOrConnection)
           : base( fileOrServerOrConnection ) {}
    public SampleDb(IDbConnection connection, MappingSource mapping)
           : base( connection, mapping ) {}

    public Table<Customer> Customers;
}
```

Poznámka

Členy tabulky se inicializují automaticky v základním konstruktoru třídy *DataContext*, jenž zkoumá typ za běhu pomocí Reflection, vyhledává příslušné členy a inicializuje je na základě mapovacích metadat.

Třídy entit

Třída entity má dvě role. První je poskytovat metadata pro dotazovací stroj LINQ: pro tyto účely se nezakládá instance třídy entity. Druhou rolí je vytvářet úložiště pro data načtená ze zdroje relačních dat, uchovávat případné změny a zařizovat jejich přenos zpět do zdroje relačních dat.

Třídu entity tvoří libovolná definice referenčního typu s atributem *Table*. Pro tyto účely nelze použít strukturu (hodnotový typ). Atribut *Table* může mít parametr *Name*, který specifikuje název odpovídající tabulky v databázi. Jestliže parametr *Name* neexistuje, použije se standardně název třídy:

```
[Table(Name="Products")] public class Product { ... }
```

Poznámka

I když se běžně používá pojem *tabulka*, nic vám nebrání použít v parametru *Name* namísto tabulky aktualizovatelný pohled. Pohled, jejž aktualizovat nelze, je možné použít také, přinejmenším do chvíle, dokud se nepokusíte aktualizovat data bez použití dané třídy entity.

Uvnitř třídy entity může být libovolný počet členů libovolného typu. Při definici mapování mezi třídou entity a odpovídající tabulkou v databázi hrají roli pouze datové členy či vlastnosti s atributem *Column*:

```
[Column] public int ProductID;
```

Třída entity by měla mít jedinečný klíč. Tento klíč je nezbytný pro jedinečnou identitu (více o tématu později), aby bylo možné identifikovat odpovídající řádky v databázových tabulkách a generovat příkazy SQL pro aktualizaci dat. Jestliže nemáte v tabulce primární klíč, lze zakládat instance tříd entit, avšak tyto instance nebudou modifikovatelné. Logická hodnota *IsPrimaryKey* v atributu *Column* nastavená na *true* říká, že sloupec přísluší primárnímu klíči v tabulce. Jestliže se jako primární klíč používá složený klíč, všechny sloupce vytvářející tento primární klíč budou mít ve svých parametrech nastaveno *IsPrimaryKey=true*:

```
[Column(IsPrimaryKey=true)] public int ProductID;
```

Standardně se sloupec mapuje pomocí stejného názvu, jaký má člen, v němž se používá daný atribut *Column*. Můžete použít i odlišný název a zadat hodnotu parametru *Name*. Například následující člen *Price* odpovídá poli *UnitPrice* v tabulce v databázi:

```
[Column(Name="UnitPrice")] public decimal Price;
```

Chcete-li filtrovat přístup k datům pomocí přístupů ve vlastnostech členů, musíte do parametru *Storage* zadat příslušný člen základního úložiště. Jestliže vložíte parametr *Storage*, LINQ pro SQL obejde veřejný přístup k vlastnosti a bude interagovat přímo se základní hodnotou. Pochopení tohoto principu je velice důležité především v situaci, kdy chcete sledovat pouze úpravy prováděné ve vašem kódu, a nikoli operace čtení/zápisu, které provádí prostředí LINQ. V následujícím kódu se přistupuje k vlastnosti *ProductName* při každé operaci čtení/zápis ve vašem kódu; při spuštění operace LINQ se provede přímé čtení/zápis do datového členu *_ProductName*:

```
[Column(Storage="_ProductName")]
public string ProductName {
    get { return this._ProductName; }
    set { this.OnPropertyChanging("ProductName");
          this._ProductName = value;
          this.OnPropertyChanged("ProductName");
    }
}
```

Vztah mezi relačním typem a typem .NET se zakládá s předpokladem použití výchozího relačního typu, odpovídajícího použitému typu .NET. Kdykoliv potřebujete stanovit jiný typ, poslouží vám parametr *DBType*, který určuje platný typ pomocí platné syntaxe SQL pro daný relační datový zdroj. Tato vlastnost se používá pouze tehdy, když chcete vytvořit databázové schéma počínaje definicemi tříd entit (tento proces si popíšeme v kapitole 5):

```
[Column(DBType="NVARCHAR(20)")] public string QuantityPerUnit;
```

Jestliže hodnotu ve sloupci automaticky generuje databáze (což v SQL Serveru nabízí klíčové slovo IDENTITY), bude patrně potřeba synchronizovat člen třídy entity s generovanou hodnotou, a to vždy, když vložíte instanci entity do databáze. K tomu musíte nastavit parametr *IsDBGenerated* na *true* a budete muset také příslušným způsobem upravit *DBType* – například přidáním modifikátoru *IDENTITY* pro tabulky SQL Serveru:

```
[Column(DBType="INT NOT NULL IDENTITY",
    IsPrimaryKey=true, IsDBGenerated=true)]
public int ProductID;
```

Stojí za zmínku, že existuje také parametr *CanBeNull*. Lze v něm stanovit, že daná hodnota může být null, ale je nutné podotknout, že pokud chcete stanovit podobnou podmínku v databázi vytvořené v LINQ pro SQL, je stále nutné v zadání *DBType* používat klauzuli *NOT NULL*:

```
[Column(DBType="INT NOT NULL IDENTITY", CanBeNull=false,
    IsPrimaryKey=true, IsDBGenerated=true)]
public int ProductID;
```

Další parametry týkající se aktualizace dat jsou *AutoSync*, *Expression*, *IsVersion* a *UpdateCheck*. Podrobnější výklad parametrů *IsDBGenerated*, *IsVersion* a *UpdateCheck* si uvedeme v kapitole 5.

Dědičnost entit

Někdy obsahuje jediná tabulka více typů entit. Představte si například seznam kontaktů – některé mohou být na zákazníky, další na dodavatele a ostatní na zaměstnance společnosti. Z datového pohledu může mít každá taková entita určitá specifická pole. (Například zákazník může mít pole pro slevu, které nehraje roli u zaměstnanců a dodavatelů.) Z pohledu obchodní logiky může každá entita pracovat s odlišnými obchodními pravidly. Nejlepší způsob, jak modelovat tento druh dat v objektově orientovaném prostředí, je využít dědičnost a vytvořit hierarchii specializovaných tříd. LINQ pro SQL umožňuje vytvořit množinu tříd odvozených z jedné základní třídy a mapovat je na tutéž relační tabulku.

Součástí základní třídy hierarchie je atribut *InheritanceMapping*, jenž obsahuje odpovídající odvozené třídy vycházející z hodnoty speciálního *sloupce diskriminátoru*. Parametr *Code* obsahuje možnou hodnotu a parametr *Typ* určuje odpovídající odvozený typ. Sloupec diskriminátoru se definuje nastavením parametru *IsDiscriminator* na *true* v atributech sloupce.

Výpis 4.3 nabízí ukázku hierarchie založené na tabulce Contacts v ukázkové databázi Northwind.

Výpis 4.3 Hierarchie tříd založená na kontaktech

```
[Table(Name="Contacts")]
[InheritanceMapping(Code = "Customer", Type = typeof(CustomerContact))]
[InheritanceMapping(Code = "Supplier", Type = typeof(SupplierContact))]
[InheritanceMapping(Code = "Shipper", Type = typeof(ShipperContact))]
[InheritanceMapping(Code = "Employee", Type = typeof(Contact), IsDefault = true)]
public class Contact {
    [Column(IsPrimaryKey=true)] public int ContactID;
    [Column(Name="ContactName")] public string Name;
    [Column] public string Phone;
    [Column(IsDiscriminator = true)] public string ContactType;
```