Sadržaj

Predgovor ..... **xi**

1. Uvod u C# i .NET Framework **1**
Objektna orijentisanost 1
Bezbednost tipova 2
Upravljanje memorijom 2
Podrška za različite platforme 3
Odnos između jezika C# i izvršnog okruženja CLR 3
CLR i .NET Framework 3
C# i Windows Runtime 4
Novo u C# 5.0 6
Novo u C# 4.0 6
Novo u C# 3.0 7

2. Osnove jezika C# **9**
Prvi C# program 9
Sintaksa 11
Osnove tipova 14
Numerički tipovi 21
Logički tip i operatori 27
Znakovni nizovi i pojedinačni znaci 29
Nizovi 31
Promenljive i parametri 34
Izrazi i operatori 42
Naredbe 45
Imenski prostori 53

3. Definisanje tipova u jeziku C# **59**
Klase 59
Nasleđivanje 71
Tip object 79
Strukture 82
Modifikatori pristupa 83
Interfejsi 85
Nabrajanje (tip enum) 90

Ugnežđeni tipovi 93
Generičke komponente 94

4. Napredni C# 107
Delegati 107
Događaji 115
Lambda izrazi 121
Anonimne metode 125
Naredbe try i izuzeci 125
Nabrajanje i iteratori 132
Tipovi koji prihvataju null 137
Preklapanje operatora 142
Proširene metode 145
Anonimni tipovi 147
Dinamičko povezivanje 148
Atributi 155
Atributi za podatke od pozivaoca (C# 5) 157
Nebezbedan kôd i pokazivači 159
Pretprocesorske direktive 162
Dokumentovanje programa u formatu XML 164

5. Uvod u .NET Framework 169
CLR i osnovni Framework 171
Primenjene tehnologije 175

6. Osnove .NET Frameworka 181
Obrada znakovnih nizova i teksta 181
Datumi i vremena 193
Datumi i vremenske zone 200
Formatiranje i raščlanjivanje znakovnih nizova 205
Standardni opisi formata i indikatori za raščlanjivanje podataka 210
Ostali mehanizmi za konverzije 217
Globalizacija 220
Rad s brojevima 221
Nabrajanja 225
N-torke 228
Struktura Guid 229
Utvrđivanje jednakosti 229
Utvrđivanje relativnog redosleda 239
Pomoćne klase 242

7. Kolekcije 245
Nabrajanje članova kolekcije 245
Interfejsi ICollection i IList 252

Klasa Array 255
Liste, redovi čekanja, stekovi i skupovi 263
Rečnici 271
Namenske kolekcije i posrednici 276
Zamenjiva logika za utvrđivanje jednakosti i relativnog redosleda 282

8. LINQ upiti 289
Uvod 289
Tečna sintaksa 291
Izrazi upita 296
Odloženo izvršavanje 300
Podupiti 306
Strategije za sastavljanje upita 309
Strategije za projekcije 312
Interpretirani upiti 314
LINQ to SQL i Entity Framework 320
Formiranje izraza upita 332

9. LINQ operatori 339
Pregled operatora 340
Filtriranje ulazne sekvence 343
Projekcije 346
Spajanje 356
Menjanje redosleda elemenata 363
Grupisanje elemenata ulazne sekvence 366
Operatori za skupove 369
Metode za konverziju 370
Operatori za pojedinačne elemente 372
Metode za agregiranje elemenata 374
Kvantifikatori 378
Metode za generisanje sekvenci 379

10. Model LINQ to XML 381
Opis arhitekture modela 381
Opis X-DOM-a 382
Instanciranje X-DOM stabla 385
Navigacija po stablu i pretraživanje sadržaja 387
Ažuriranje sadržaja X-DOM stabla 392
Upotreba svojstva Value 394
Dokumenti i deklaracije 397
Imena i imenski prostori 400
Napomene 405
Projektovanje u X-DOM stablo 406

11. Ostale XML tehnologije 411
Klasa XmlReader 411
Klasa XmlWriter 420
Šabloni za upotrebu klasa XmLReader/XmlWriter 421
Klasa XmlDocument 425
XPath upiti 429
XSD i provera ispravnosti dokumenta pomoću šeme 432
XSLT 435

12. Oslobađanje resursa i sakupljanje smeća 437
Interfejs IDisposable i njegove metode Dispose i Close 437
Automatsko sakupljanje smeća 442
Finalizatori 444
Kako radi sakupljač smeća 448
Curenje upravljane memorije 452
Slabe reference 455

13. Dijagnostika i kodni ugovori 461
Uslovno prevođenje koda 461
Klase Debug i Trace 464
Kodni ugovori 467
Preduslovi 471
Konačna stanja 475
Pretpostavke i objektne invarijante 477
Kodni ugovori u interfejsima i apstraktnim metodama 478
Obrada slučaja kršenja ugovora 479
Selektivno proveravanje ugovora 481
Statičko proveravanje ugovora 483
Integrisanje aplikacije sa spoljnim dibagerom 484
Procesi i procesne niti 485
Klase StackTrace i StackFrame 486
Windowsovi dnevnici događaja 488
Brojači performansi 490
Klasa Stopwatch 494

14. Istovremenost i asinhroni način rada 495
Uvod 495
Višenitni način rada programa 496
Poslovi 511
Principi asinhronog rada 518
Asinhrone funkcije jezika C# 5.0 523
Modeli asinhronog programiranja 537
Zastareli modeli 544

15. Tokovi i ulaz/izlaz podataka 547
Arhitektura tokova 547
Upotreba tokova 549
Adapteri za tokove 561
Tokovi za komprimovanje podataka 568
Rad sa zip datotekama 570
Operacije s datotekama i direktorijumima 571
Rad s datotekama u WinRT-u 581
Memorijske datoteke 583
Izolovano skladištenje podataka 585

16. Rad u umreženom okruženju 591
Arhitektura mreže 591
Mrežne adrese i priključci 593
URI 594
Klase za klijentsku stranu 596
Upotreba protokola HTTP 608
Pisanje vlastitog HTTP servera 613
Upotreba protokola FTP 615
Upotreba protokola DNS 617
Slanje e-pošte pomoću klase SmtpClient 618
Upotreba protokola TCP 619
Primanje POP3 e-pošte pomoću TCP-a 622
Upotreba TCP-a u WinRT-u 623

17. Serijalizovanje 625
Teorijski koncepti serijalizovanja 625
Serijalizator ugovora za podatke 628
Ugovori za podatke i kolekcije 637
Proširivanje ugovora za podatke 639
Binarni serijalizator 642
Atributi za binarno serijalizovanje 644
Binarno serijalizovanje pomoću interfejsa ISerializable 647
Serijalizovanje u XML format 650

18. Sklopovi 659
Šta se sve nalazi u jednom sklopu 659
Jaka imena i potpisivanje sklopova 663
Imena sklopova 666
Potpisivanje po sistemu Authenticode 668
Globalni keš za sklopove 671
Resursi i zavisni sklopovi 673
Razrešavanje i učitavanje sklopova 681
Instaliranje sklopova izvan osnovnog direktorijuma aplikacije 686
Priprema jedne izvršive datoteke 687
Rad sa nereferenciranim sklopovima 688

19. Refleksija i metapodaci 691
Reflektovanje i aktiviranje tipova 691
Reflektovanje i pozivanje članova tipa 698
Reflektovanje sklopova 709
Rad sa atributima 710
Dinamičko generisanje koda 715
Dinamičko generisanje sklopova i tipova 722
Dinamičko generisanje članova tipa 725
Dinamičko generisanje generičkih metoda i tipova 730
Slučajevi koji su „nezgodni“ za dinamičko generisanje 732
Raščlanjivanje IL koda 735

20. Dinamičko programiranje 741
DLR (Dynamic Language Runtime) 741
Unifikacija numeričkih tipova 743
Dinamičko razrešavanje preklopljenih verzija argumenata 744
Implementiranje dinamičkih objekata 749
Interoperabilnost s dinamičkim jezicima 752

21. Zaštita podataka 755
Ovlašćenja 755
Zaštita pomoću ovlašćenja koja se dodeljuju kodu 759
Prihvatanje poziva od delimično pouzdanih pozivalaca 762
Model transparentnosti 764
Ograničavanje drugog sklopa na rad u karantinu 772
Zaštita na nivou operativnog sistema 775
Zaštita pomoću identiteta i uloga korisnika 777
Uvod u kriptografiju 778
API Windows Data Protection 779
Heširanje 780
Simetrično šifrovanje 781
Šifrovanje i potpisivanje javnim ključem 786

22. Napredne mogućnosti za višenitni rad 789
Uvod u sinhronizaciju 789
Ekskluzivne blokade 790
Blokade i bezbednost koda za višenitni rad 797
Neekskluzivne blokade 802
Signaliziranje pomoću čekača događaja 807
Klasa Barrier 814
Odloženo inicijalizovanje 815
Skladištenje podataka lokalno u nitima 817
Metode Interrupt i Abort 820
Metode Suspend i Resume 821
Tajmeri 821

23. Paralelno programiranje 825
Čemu PFX? 825
PLINQ 828
Klasa Parallel 840
Paralelizam poslova 846
Izuzetak AggregateException 854
Kolekcije za istovremeni pristup 856
Klasa BlockingCollection<T> 859

24. Domeni aplikacija 863
Arhitektura domena aplikacije 863
Formiranje i uklanjanje domena aplikacija 864
Rad s više domena aplikacija u istom procesu 866
Upotreba metode DoCallBack 867
Nadgledanje domena aplikacija 868
Domeni aplikacija i niti izvršavanja 869
Deljenje podataka između domena 870

25. Interoperabilnost sa C i COM komponentama 875
Pozivanje DLL datoteka napisanih na jeziku C 875
Razmena tipova 876
Povratne funkcije iz neupravljanog koda 878
Simuliranje unije jezika C 879
Deljena memorija 880
Preslikavanje strukture u neupravljanu memoriju 882
Interoperabilnost sa COM komponentama 886
Pozivanje COM komponente iz C# koda 888
Ugrađivanje interop tipova 891
Primarni interop sklopovi 892
Izlaganje C# objekata COM objektima 892

26. Regularni izrazi 893
Osnove regularnih izraza 893
Kvantifikatori 897
Pravila dužine nula 898
Grupe 901
Zamenjivanje i razdvajanje teksta 902
Gotovi regularni izrazi 903
Referentni pregled jezika za regularne izraze 907

Spisak termina korišćenih u knjizi 911

Indeks 915



Predgovor

C# 5.0 je četvrta značajna nadogradnja Microsoftovog vrhunskog programskog jezika; zahvaljujući tome, odlikuju ga neuobičajena fleksibilnost i širina. S jedne strane, nudi apstrakcije visokog nivoa kao što su izrazi upita i asinhroni nastavci, dok s druge strane, pruža moć na niskom nivou u obliku konstrukata kao što su namenski vrednosni tipovi i opciona upotreba memorijskih pokazivača.

Cena tog porasta je to što sada treba naučiti više stvari nego dosad. Mada su alatke kao što je Microsoftov IntelliSense – i reference na internetu – odlične i od velike pomoći u tom poslu, one pretpostavljaju postojanje određenog konceptualnog znanja. Ova knjiga vam pruža tačno to znanje u sažetom i ujednačenom stilu – bez pretrpanosti i dugačkih uvoda.

Organizacija knjige *C# 5.0 za programere* zasniva se na konceptima i primerima upotrebe, što je čini pogodnom i za čitanje redom, ali i za nasumično pregledanje. Objašnjenja su detaljna, ali se ipak pretpostavlja osnovno predznanje – što knjigu čini pogodnom za čitaoce sa srednjim nivoom znanja isto koliko i za one napredne.

U knjizi su objašnjeni C#, CLR i osnovni sklopovi Frameworka. Usredsredili smo se na te oblasti da bismo dobili prostor za teže teme – kao što su istovremenost, zaštita koda i domeni aplikacija – bez žrtvovanja dubine ili razumljivosti. Nove mogućnosti koje uvode C# 5.0 i odgovarajuća verzija Frameworka posebno su obeležene kako biste ovu knjigu mogli da koristite i kao referentni priručnik za C# 4.0.

Kome je knjiga namenjena

Ciljna grupa ove knjige jesu čitaoci sa srednjim do naprednim nivoom znanja. Nije potrebno predznanje o jeziku C#, ali je neophodno određeno opšte znanje o programiranju. U slučaju početnika, ova knjiga pre dopunjuje nego što zamenjuje udžbenike za programiranje.

Ako već poznajete verziju C# 4.0, ovde ćete naći reorganizovan odeljak o istovremenosti, s detaljnim objašnjenjima asinhronih funkcija i odgovarajućih tipova jezika C# u verziji 5.0. Uz to, opisujemo i principe asinhronog programiranja i kako ono doprinosi efikasnosti i bezbednom višenitnom radu programskog koda.

Ova knjiga je savršena dopuna brojnih knjiga o primenjenim tehnologijama kao što su WPF, ASP.NET ili WCF. Oblasti jezika i .NET Frameworka koje su u takvim knjigama izostavljene, *C# 5.0 za programere* detaljno objašnjava – i obrnuto.

Ako tražite knjigu koja ukratko objašnjava sve postojeće tehnologije .NET Frameworka, ova nije za vas. Neće vam odgovarati ni ako želite da savladate API-je koji su specifični za razvijanje aplikacija za tablične računare ili za razvijanje Windows Phone aplikacija.

Kako je knjiga organizovana

Prva tri poglavlja iza uvoda usredsređena su na čisti C# i kreću sa osnovama sintakse, tipovima i promenljivama, da bi se završila s naprednijim temama kao što su nebezbedan kôd i pretprocesorske direktive. Ako ste početnik u jeziku C#, trebalo bi da ta poglavlja pročitate jedno za drugim.

Ostala poglavlja objašnjavaju osnove .NET Frameworka i pokrivaju teme kao što su LINQ, XML, kolekcije, kodni ugovori, istovremenost, ulazno/izlazne operacije i rad u mreži, upravljanje memorijom, refleksija, dinamičko programiranje, atributi, bezbednost, domeni aplikacija i interoperabilnost sa C kodom. Najveći deo tih poglavlja možete čitati proizvoljnim redosledom, osim poglavlja 6 i 7, koja polažu temelje za teme koje slede. Najbolje je da jedno za drugim pročitate i tri poglavlja o LINQ-u, a u nekim poglavljima se pretpostavlja da imate određeno predznanje o istovremenosti, koju objašnjavamo u poglavlju 14.

Šta vam je potrebno da biste koristili knjigu

Za primere iz knjige treba vam kompajler C# 5.0 koda i Microsoftov .NET Framework 4.5. Korisna je i Microsoftova dokumentacija za .NET (dostupna je i na internetu), u kojoj ćete naći informacije o pojedinačnim tipovima i njihovim članovima.

Iako je moguće pisati izvorni kôd u Notepadu i koristiti kompajler s komandne linije, bićete znatno produktivniji ako koristite neko *pomagalo za pisanje koda* da biste odmah testirali blokove koda, kao i neko *integrisano razvojno okruženje* (IDE) za generisanje izvršnih datoteka i biblioteka.

Kao pomagalo za pisanje koda, preuzmite (besplatno) LINQPad 4.40 ili noviji sa adrese www.linqpad.net. LINQPad potpuno podržava C# 5.0 a održava ga jedan od autora ove knjige.

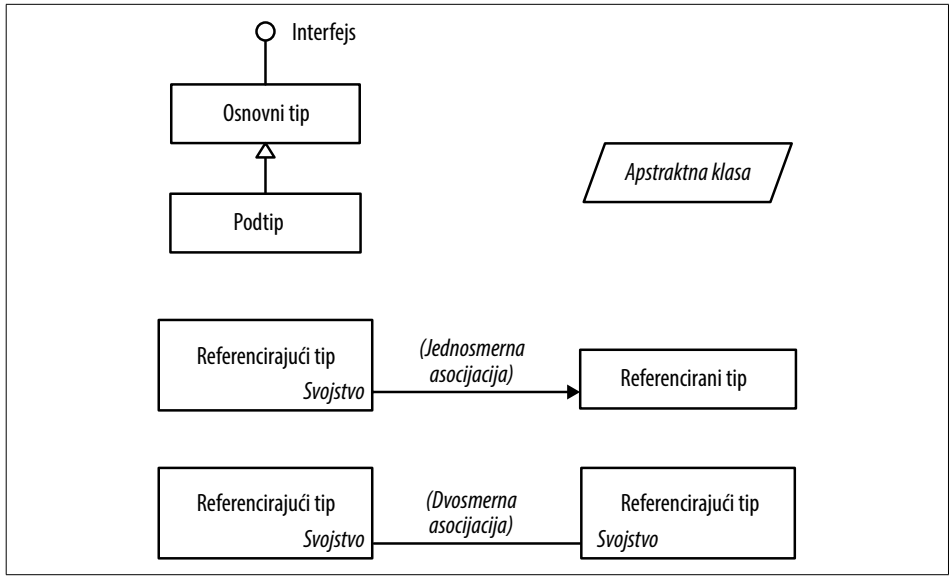
Kao integrisano razvojno okruženje, preuzmite Microsoftov Visual Studio 2012: za gradivo iz ove knjige odgovara svako izdanje osim besplatne verzije Express.



Svi listinzi primera iz poglavlja 2–11, kao i oni iz poglavlja o istovremenosti, paralelnom programiranju i dinamičkom programiranju, na raspolaganju su u obliku interaktivnih (koji se mogu menjati) LINQPad primera. Možete ih odjednom preuzeti sve zajedno: pomoću jezička dole levo pređite u LINQPadov odeljak Samples, mišem pritisnite opciju „Download more samples“, a zatim izaberite „C# 5.0 in a Nutshell“.

Konvencije korišćene u knjizi

U ovoj knjizi koristimo UML notaciju da bismo ilustrovali odnose između tipova, kao što je prikazano na slici P-1. Pravougaonik sa iskošenim stranicama predstavlja apstraktnu klasu; krug predstavlja interfejs. Linija koja se završava praznim trouglom predstavlja nasleđivanje, pri čemu je trougao usmeren ka osnovnom tipu. Linija završena strelicom predstavlja jednosmernu asocijaciju; linija bez strelica predstavlja dvosmernu asocijaciju.



Slika P-1. Jednostavan dijagram.

U knjizi koristimo sledeće tipografske konvencije:

Kurziv
: Predstavlja nove termine, URI-je, imena datoteka i direktorijuma.

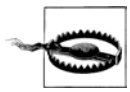
`Neproporcionalan font`
: Predstavlja C# kôd, rezervisane reči i identifikatore, kao i rezultate izvršavanja programa.

`Podebljan neproporcionalan font`
: Prikazuje istaknute delove programskog koda.

`Kurzivni neproporcionalni font`
: Prikazuje tekst koji korisnik programa treba da zameni svojim vrednostima.



Označava savet, predlog ili napomenu opšteg tipa.



Označava upozorenje.

Upotreba primera koda

Svrha ove knjige je da vam pomogne da obavite posao. Kôd iz knjige možete koristiti u svojim programima i dokumentaciji. Nije potrebno da tražite našu dozvolu za to ukoliko ne preuzimate značajan deo koda. Na primer, ako pišete program u kome ćete upotrebiti nekoliko manjih delova koda iz ove knjige, ne treba vam naša dozvola. Ali, za preprodaju ili distribuciju kompakt diska s primerima iz knjige, naša *dozvola je neophodna*. Za odgovaranje na pitanja uz korišćenje citata iz ove knjige i navođenje primera koda nije vam potrebna naša dozvola, što *ne važi* za ugrađivanje značajnog dela primera koda iz ove knjige u vašu dokumentaciju.

Ne želimo da vas obavezujemo, ali cenili bismo ako biste nas pomenuli kada koristite materijal iz ove knjige. Pod tim podrazumevamo navođenje naslova, autora, izdavača i ISBN broja. Na primer: „*C# 5.0 in a Nutshell*, Joseph Albahari i Ben Albahari. Copyright 2012 Joseph Albahari i Ben Albahari, 978-1-449-32010-2“.

Ako smatrate da način na koji želite da upotrebite primere iz ove knjige ne odgovara nijednom od pomenutih slučajeva, pišite nam na adresu *permissions@oreilly.com*.

Zahvalnica

Džozef Albahari

Prvo želim da se zahvalim svom bratu, Benu Albahariju, što me je ubedio napišem *C# 3.0 in a Nutshell*, čiji uspeh je doveo do naknadnih izdanja. Ben deli moju volju za dovođenje u pitanje konvencionalnih istina, kao i istrajnost da stvari rastavljam na deliće dok ne postane jasno kako one *zaista* rade.

Bila mi je čast da u timu sarađujem s vrhunskim recenzentima. Ovo izdanje mnogo duguje dvojici legendarnih Microsoftovih stručnjaka: Eriku Lipertu (tim za razvoj C# kompajlera) i Stivenu Taubu (tim za paralelno programiranje). Ne mogu vam dovoljno zahvaliti na obimnoj i korisnoj podršci – niti na strpljivom odgovaranju na sva moja pitanja. Takođe sam neizmerno zahvalan C# MVP-u Nikolasu Paldinou, čiji su oštro oko i sposobnost da zapazi stvari koje su drugima promakle uobličili ovu knjigu i oba prethodna izdanja.

Ova knjiga nadograđuje izdanje *C# 4.0 in a Nutshell*, čijim recenzentima dugujem sličnu zahvalnost. Kris Burous (tim za razvoj C# kompajlera) značajno je „ispolirao“ poglavlja o istovremenosti, dinamičkom programiranju i jeziku C#. Neprocenjive informacije o bezbednosti i upravljanju memorijom dobijao sam od Šona Farkasa, Brajana Grankemejera, Maonija Stivensa i Dejvida DeVintera iz CLR tima. A u vezi s temom kodnih ugovora, informacije od Brajana Grankemejera, Maka Barneta i Melite Andersen podigle su to poglavlje na viši nivo.

Izuzetno cenim doprinos Jona Sketa (autora knjige *C# in Depth* i vrlo uglednog na veb lokaciji Stack Overflow), čiji su oštroumni predlozi doprineli prethodnom izdanju, zatim doprinose C# MVP-ova Miča Vita i Brajana Peka, kao i zasluge recenzenata izdanja za verziju 3.0, među kojima su Kžištof Kvalina, Met Voren, Džoel Pobar, Glin Grifits, Jon Vasilian, Bred Abrams, Sem Gentajl i Adam Nejtan.

I najzad, želim da zahvalim izdavačkom timu kuće O'Reilly – urednici, Rejčel Rumeliotis (bila je radost raditi zajedno) i odličnoj lektorki Nensi Rajnhart – kao i članovima moje porodice, Majri i Sonji.

Ben Albahari

Budući da je moj brat prvi napisao svoju zahvalnicu, možete naslutiti najveći deo onog što želim da kažem. :) Mi se obojica bavimo programiranjem zapravo još od detinjstva (delili smo jedan Apple IIe; on je razvijao vlastiti operativni sistem dok sam ja pisao igricu Hangman), pa je zasad baš „kul“ što zajedno pišemo knjige. Nadam se da će se bogato iskustvo koje smo stekli pisanjem ove knjige pretočiti u bogato iskustvo za vas dok je čitate.

Želeo bih takođe da se zahvalim svojim bivšim kolegama u Microsoftu. Tamo radi mnogo vrednih ljudi, ne samo u intelektualnom pogledu, nego i u širem emotivnom smislu, a zajednički rad s njima mi nedostaje. Pre svega, mnogo sam naučio od Brajana Bekmana, čiji sam večiti dužnik.



2

Osnove jezika C#

U ovom poglavlju predstavljamo osnove jezika C#.



Svi programi i blokovi koda iz ovog i dva naredna poglavlja na raspolaganju su u obliku interaktivnih primera u LINQPadu. Proučavanje tih primara u kombinaciji s ovom knjigom ubrzava učenje jer možete menjati primere i odmah videti rezultat bez izrade projekata i rešenja u Visual Studiju.

Da biste preuzeli primere, mišem pritisnite jezičak u LINQPadu a zatim izaberite „Download more samples.“ LINQPad je besplatan – idite na adresu *http://www.linqpad.net*.

Prvi C# program

Sledeći program množi 12 sa 30 i prikazuje rezultat, 360, na ekranu. Dvostruka kosa crta znači da je preostali deo reda *komentar*.

```
using System;                          // Uvoženje imenskog prostora

class Test                             // Deklaracija klase
{
  static void Main()                   //   Deklaracija metode
  {
    int x = 12 * 30;                   //     Naredba 1
    Console.WriteLine (x);             //     Naredba 2
  }                                    //   Kraj metode
}                                      // Kraj klase
```

Srce ovog programa čine dve *naredbe* (engl. *statements*):

```
int x = 12 * 30;
Console.WriteLine (x);
```

Naredbe se u jeziku C# izvršavaju jedna za drugom, a završavaju se znakom tačka i zarez (ili *blokom koda*, kao što ćemo videti u nastavku). Prva naredba izračunava vrednost *izraza* `12 * 30` i rezultat smešta u *lokalnu promenljivu*, čije je ime `x` a tip celobrojni (engl. *integer*). Druga naredba poziva *metodu* `WriteLine` klase `Console`, koja prikazuje vrednost promenljive `x` u obliku teksta na ekranu.

Metoda izvršava određenu akciju pomoću grupe naredaba, koja se zove *blok naredaba* (engl. *statement block*) – što je par vitičastih zagrada između kojih se nalazi nula, jedna ili više naredaba. Definisali smo samo jednu metodu, po imenu `Main`:

```
static void Main()
{
  ...
}
```

Pisanje funkcija višeg nivoa koje pozivaju druge funkcije nižeg nivoa pojednostavljuje program. Evo kako ćemo *refaktorisati* (tj. poboljšati) program pomoću višekratno upotrebljive metode koja množi zadatu celobrojnu vrednost sa 12:

```
using System;

class Test
{
  static void Main()
  {
    Console.WriteLine (FeetToInches (30));      // 360
    Console.WriteLine (FeetToInches (100));     // 1200
  }

  static int FeetToInches (int feet)
  {
    int inches = feet * 12;
    return inches;
  }
}
```

Metoda može da prima *ulazne podatke* (engl. *input*) od svog pozivaoca tako što definiše *parametre*, a može i da mu vraća *izlazne podatke* (engl. *output*) ako definiše *povratni tip* (engl. *return type*). Definisali smo metodu nazvanu `FeetToInches` (pretvaranje stopa u inče) koja ima ulazni parametar za stope i povratni tip za izračunate inče:

```
static int FeetToInches (int feet ) {...}
```

Literali `30` i `100` su *argumenti* koje pozivalac prosleđuje metodi `FeetToInches`. Metoda `Main` iz našeg primera ima prazne zagrade iza imena zato što nema ulazne parametre, a njen povratni tip je `void` jer svom pozivaocu ne vraća nikakvu vrednost:

```
static void Main()
```

Metodu čije je ime `Main` C# prepoznaje kao podrazumevanu ulaznu tačku u programu, odakle počinje izvršavanje programa. Metoda `Main` može po potrebi imati povratnu vrednost celobrojnog tipa (umesto `void`), koju vraća izvršnom okruženju (gde je uobičajeno da vrednost različita od nule znači grešku). Metoda `Main` može opciono da prihvata kao ulazni parametar niz znakovnh vrednosti (pomoću kojeg se programu prosleđuju argumenti).

Na primer:

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



Niz (kao što je `string[]`) predstavlja fiksan broj elemenata određenog tipa. Nizove definišete tako što napišete uglaste zagrade iza tipa elemenata niza. Nizovi su opisani u odeljku „Nizovi“ na strani 31.

Metode su jedna od više vrsta funkcija u jeziku C#. Druga vrsta funkcije koju smo pozvali jeste *operator* *, koji obavlja množenje. Postoje još i *konstruktori*, *svojstva* (engl. *properties*), *događaji* (engl. *events*), *indekseri* i *finalizatori*.

U našem primeru, navedene dve metode su grupisane u jednu klasu. *Klasa* grupiše funkcije članice i podatke članove tako da svi zajedno čine celinu i objektno orijentisan gradivni blok jezika. Klasa `Console` grupiše članove koji obezbeđuju funkcionalnost učitavanja

podataka sa komandne linije i prikazivanje podataka na njoj, kao što je metoda `WriteLine`. Naša klasa `Test` grupiše dve metode – metodu `Main` i metodu `FeetToInches`. Klasa je jedna vrsta *tipa*, što ćemo razmatrati u odeljku „Osnove tipova“, na strani 14.

Na najvišem nivou programa, tipovi su organizovani u *imenske prostore* (engl. *namespaces*). Direktiva `using` uključuje imenski prostor `System` u našu aplikaciju, kako bismo mogli da koristimo klasu `Console`. Sve svoje klase možemo da definišemo unutar imenskog prostora `TestPrograms`, na sledeći način:

```
using System;

namespace TestPrograms
{
  class Test  {...}
  class Test2 {...}
}
```

.NET Framework je organizovan tako da su njegovi imenski prostori ugnežđeni jedan u drugi. Na primer, sledeći imenski prostor sadrži tipove za rad s tekstom:

```
using System.Text;
```

Upotreba direktive `using` samo je pogodniji oblik za programera; svaki tip možete referencirati i pomoću njegovog potpuno kvalifikovanog imena, koje se sastoji od imena tipa, kojem prethodi ime imenskog prostora (kojem tip pripada). Na primer: `System.Text.StringBuilder`.

Prevođenje programskog koda

Kompajler jezika C# prevodi izvorni kôd, raspodeljen u grupu datoteka s nastavkom *.cs*, u oblik koji se zove *sklop* (engl. *assembly*). Sklop je jedinica pakovanja i distribuiranja .NET softvera. Sklop može biti *aplikacija* ili *biblioteka*. Standardna konzolska ili Windows aplikacija ima metodu `Main` i pakuje se u *.exe* datoteku. Biblioteka ima nastavak *.dll* i ekvivalentna je *.exe* aplikaciji ali nema ulaznu tačku. Njena svrha je da je druge aplikacije ili biblioteke pozivaju (*referenciraju*). Sam .NET je veliki skup biblioteka.

Ime kompajlera C# koda je *csc.exe*. Da biste prevodili svoje programe, možete ga pozivati unutar nekog integrisanog razvojnog okruženja, kao što je Visual Studio, ili pozovite `csc` ručno, s komandne linije. Da biste program preveli ručno, prvo ga snimite u datoteku kao što je *MyFirstProgram.cs*, a zatim pređite na komandnu liniju i pokrenite `csc` (nalazi se ispod *%SystemRoot%\Microsoft.NET\Framework\<framework-version>*, gde je *%SystemRoot%* direktorijum u kome je instaliran vaš primerak Windowsa), na sledeći način:

```
csc MyFirstProgram.cs
```

Rezultat je aplikacija čije je ime *MyFirstProgram.exe*.

Da biste napravili biblioteku (*.dll*), uradite sledeće:

```
csc /target:library MyFirstProgram.cs
```



Sklopove detaljno razmatramo u poglavlju 18.

Sintaksa

Za sintaksu jezika C# uzor je bila sintaksa jezika C i C++. U ovom odeljku opisaćemo elemente sintakse jezika C# pomoću sledećeg primera programa:

```
using System;

class Test
{
  static void Main()
  {
    int x = 12 * 30;
    Console.WriteLine (x);
  }
}
```

Identifikatori i rezervisane reči

Identifikatori su imena koje programeri izaberu za svoje klase, metode, promenljive itd. U našem primeru programa imamo sledeće identifikatore, redosledom kojim se pojavljuju:

```
System   Test   Main   x   Console   WriteLine
```

Identifikator mora da se sastoji od jedne reči, uglavnom od Unicode znakova, a počinje slovom ili znakom za podvlačenje. C# pravi razliku između malih i velikih slova u identifikatorima. Uobičajena je konvencija prema kojoj se imena parametara, lokalnih promenljivih i privatnih polja pišu s malim početnim slovom i mešanim ostalim slovima (npr., `ovoJePromenljiva`), a sve ostale identifikatore treba pisati u tzv. Pascal formatu (npr., `OvoJeMetoda`), s velikim početnim slovom.

Rezervisane reči (engl. *keywords*) jesu imena koje je rezervisao kompajler i koje zbog toga ne možete koristiti kao identifikatore u svojim programima. U našem primeru programa, imamo sledeće rezervisane reči:

```
using   class   static   void   int
```

Evo i kompletnog spiska rezervisanih reči jezika C#:

abstract	const	extern	int	out	short	typeof
as	continue	false	interface	override	sizeof	uint
base	decimal	finally	internal	params	stackalloc	ulong
bool	default	fixed	is	private	static	unchecked
break	delegate	float	lock	protected	string	unsafe
byte	do	for	long	public	struct	ushort
case	double	foreach	namespace	readonly	switch	using
catch	else	goto	new	ref	this	virtual
char	enum	if	null	return	throw	void
checked	event	implicit	object	sbyte	true	volatile
class	explicit	in	operator	sealed	try	while

Izbegavanje sukoba

Ako zaista želite da koristite identifikator koji je isti kao jedna od rezervisanih reči, to možete uraditi tako što mu pridružite prefiks `@`. Na primer:

```
class class  {...}      // Nije dozvoljeno
class @class {...}      // Ispravno
```

Pošto simbol `@` nije sastavni deo samog identifikatora, `@imePromenljive` je isto što i `imePromenljive`.



Prefiks `@` može biti koristan kada radite s bibliotekama napisanim na drugim .NET jezicima koji imaju druge rezervisane reči.

Kontekstne rezervisane reči

Neke rezervisane reči su *kontekstne*, što znači da se mogu koristiti kao identifikatori – bez prefiksa `@`. To su sledeće:

```
add         by          from        in          on          select      where
ascending   descending  get         into        orderby     set         yield
async       dynamic     global      join        partial     value
await       equals      group       let         remove      var
```

Pri upotrebi kontekstnih rezervisanih reči, ne može se pojaviti dvosmislenost u kontekstu u kojem se one koriste.

Literali, graničnici i operatori

Literali su jedinice podataka umetnute u program. U našem primeru programa, literali su `12` i `30`.

Graničnici služe za razgraničavanje pojedinih delova strukture programa. U našem primeru programa imamo sledeće graničnike:

```
{   }   ;
```

Vitičaste zagrade grupišu više naredaba u jedan *blok naredaba*.

Svaka naredba se završava znakom tačka i zarez. (Međutim, taj znak nije obavezan iza blokova naredaba.) Jedna naredba može se prostirati na više redova:

```
Console.WriteLine
  (1 + 2 + 3 + 4 + 5 + 6 + 7 + 8 + 9 + 10);
```

Operator transformiše i kombinuje izraze. Većina operatora jezika C# pišu se u obliku simbola, kao što je operator množenja, `*`. Operatore razmatramo detaljnije u nastavku ovog poglavlja. U našem primeru programa, koristimo sledeće operatore:

```
.  ()   *   =
```

Tačkom se obeležava član nečega (ili je decimalna tačka u numeričkim literalima). Zagrade se koriste u pozivima i deklaracijama metoda; prazne zagrade se navode kada metoda ne prihvata nijedan argument. Znak jednakosti obavlja operaciju *dodeljivanja vrednosti*. (Dvostruki znak jednakosti, `==`, ispituje jednakost, kao što ćemo videti u nastavku poglavlja.)

Komentari

U jeziku C# postoje dve vrste komentara za dokumentovanje izvornog koda: *jednoredni komentari* i *višeredni komentari*. Jednoredni komentar počinje dvostrukom kosom crtom i nastavlja se do kraja reda. Na primer:

```
int x = 3;   // Komentar o dodeli vrednosti 3 promenljivoj x
```

Višeredni komentar počinje znacima `/*` i završava se sa `*/`. Na primer:

```
int x = 3;   /* Ovak komentar se prostire
                u dva reda */
```

U komentare se mogu umetati dokumentacione XML oznake, što je objašnjeno na strani 164 u poglavlju 4.

Osnove tipova

Tip (engl. *type*) definiše prirodu date vrednosti. U našem primeru, koristimo dva literala tipa `int` čije su vrednosti 12 i 30. Deklarisali smo i *promenljivu* tipa `int` čije je ime `x`:

```
static void Main()
{
  int x = 12 * 30;
  Console.WriteLine (x);
}
```

Promenljiva (engl. *variable*) opisuje mesto u memoriji gde se nalaze vrednosti koje se mogu menjati tokom vremena. Za razliku od toga, *konstanta* uvek predstavlja istu vrednost (više o tome u nastavku ovog poglavlja):

```
const int y = 360;
```

Sve vrednosti u C# programu jesu *instance* određenog tipa. Tip promenljive određuje značenje njene vrednosti, kao i skup vrednosti koje promenljiva može imati.

Primeri unapred definisanih tipova

Unapred definisani (standardni, ugrađeni) tipovi jesu tipovi koje kompajler koda podržava bez eksplicitne definicije tipa. Tip `int` je unapred definisan tip za celobrojne vrednosti koje se mogu smestiti u 32 bita memorije, u opsegu od -2^{31} do $2^{31}-1$. Nad instancama tipa `int` možemo obavljati operacije kao što su aritmetičke, na sledeći način:

```
int x = 12 * 30;
```

Drugi unapred definisan tip u jeziku C# jeste `string`. Tip `string` predstavlja niz alfanumeričkih znakova, kao što je „.NET“ ili „*http://www.mikroknjiga.rs*“. Znakovne nizove možemo obrađivati pomoću funkcija kao što su sledeće:

```
string message = "Hello world";
string upperMessage = message.ToUpper();
Console.WriteLine (upperMessage);              // HELLO WORLD

int x = 2015;
message = message + x.ToString();
Console.WriteLine (message);                   // Hello world2015
```

Unapred definisan tip `bool` može imati jednu od dve moguće vrednosti: `true` ili `false`. Tip `bool` se uglavnom koristi za uslovno usmeravanje toka izvršavanja programa pomoću naredbe `if`. Na primer:

```
bool simpleVar = false;
if (simpleVar)
  Console.WriteLine ("This will not print");

int x = 5000;
bool lessThanAMile = x < 5280;
if (lessThanAMile)
  Console.WriteLine ("This will print");
```



Jezik C# prepoznaje unapred definisane tipove (koji se nazivaju i ugrađeni tipovi) na osnovu odgovarajućih rezervisanih reči jezika. Imenski prostor `System` .NET Frameworka sadrži brojne važne tipove koji nisu unapred definisani u jeziku C# (na primer, `DateTime`).

Primeri namenskih tipova

Isto kao što možemo graditi složene funkcije kombinujući jednostavne, možemo graditi i složene tipove kombinujući osnovne ili osnovne tipove. U narednom primeru, definišemo namenski (nestandardni) tip (engl. *custom type*) čije je ime `UnitConverter` – klasu koja omogućava konverziju mernih jedinica:

```
using System;

public class UnitConverter
{
  int ratio;                                                  // Polje
  public UnitConverter (int unitRatio) {ratio = unitRatio; } // Konstruktor
  public int Convert   (int unit)    {return unit * ratio; } // Metoda
}

class Test
{
  static void Main()
  {
    UnitConverter feetToInchesConverter = new UnitConverter (12);
    UnitConverter milesToFeetConverter  = new UnitConverter (5280);

    Console.WriteLine (feetToInchesConverter.Convert(30));    // 360
    Console.WriteLine (feetToInchesConverter.Convert(100));   // 1200
    Console.WriteLine (feetToInchesConverter.Convert(
                         milesToFeetConverter.Convert(1)));   // 63360
  }
}
```

Članovi tipa

Tip sadrži *podatke članove* (engl. *data members*) i *funkcije članice* (engl. *function members*). Podatak član klase `UnitConverter` jeste *polje* (engl. *field*) čije je ime `ratio` (faktor konverzije). Funkcije članice klase `UnitConverter` jesu metoda `Convert` i *konstruktor* klase `UnitConverter`.

Simetrija unapred definisanih i namenskih tipova

Jedna od lepota jezika C# jeste činjenica da ima malo razlika između unapred definisanih i namenskih tipova. Unapred definisan tip `int` opisuje celobrojne vrednosti. On čuva podatke – 32 bita – i ima funkcije članice za rad s tim podacima, kao što je metoda `ToString`. Slično tome, naš namenski tip `UnitConverter` opisuje konverzije mernih jedinica. On čuva podatke – faktor konverzije – i ima funkcije članice koji rade s tim podacima.

Konstruktori i instanciranje tipova

Podaci članovi tipa dobijaju vrednosti izradom instance, tj. *instanciranjem*, tipa. Unapred definisani tipovi mogu se instancirati jednostavnim navođenjem literala, kao što su `12` ili `"Zdravo svima"`. Operator `new` pravi instance namenskog tipa. Instancu tipa `UnitConverter` napravili smo i definisali pomoću sledeće naredbe:

```
UnitConverter feetToInchesConverter = new UnitConverter (12);
```

Odmah pošto operator `new` instancira objekat, poziva se *konstruktor* tipa radi inicijalizovanja stanja objekta. Konstruktor se definiše u istom obliku kao metoda, s tom razlikom što su ime metode i njen povratni tip jednaki imenu tipa:

```
public class UnitConverter
{
  ...
```

```
  public UnitConverter (int unitRatio) { ratio = unitRatio; }
  ...
}
```

Statički članovi i članovi instance

Podaci članovi i funkcije članice koje deluju na konkretnu *instancu* tipa, zovu se *članovi instance*. Metoda `Convert` klase (tipa) `UnitConverter` i metoda `ToString` tipa `int` primeri su članova instance. Članovi tipa su standardno članovi instance.

Podaci članovi i funkcije članice koje ne deluju ni na jednu određenu instancu tipa, nego na sam tip, moraju se označiti kao `static`. Metode `Test.Main` i `Console.WriteLine` jesu statičke metode. Klasa `Console` je u stvari *statička klasa*, što znači da su *svi* njeni članovi statički. Nećete nikad napraviti instancu klase `Console` – istu konzolu dele svi delovi aplikacije.

Da bismo ilustrovali razliku između članova instance i statičkih članova, u narednom kodu polje instance `Name` sadrži vrednost koja pripada jednoj od postojećih instanci klase `Panda` (može biti različita u drugim instancama), dok polje `Population` sadrži vrednost koja je uvek jednaka u svim postojećim instancama klase `Panda`:

```
public class Panda
{
  public string Name;              // Polje instance
  public static int Population;    // Statičko polje

  public Panda (string n)          // Konstruktor
  {
    Name = n;                      // Dodeljuje vrednost polju instance
    Population = Population + 1;   // Inkrementira statičko polje Population
  }
}
```

Naredni kôd pravi dve instance klase `Panda`, prikazuje njihova imena a zatim prikazuje i ukupan broj pandi:

```
using System;

class Test
{
  static void Main()
  {
    Panda p1 = new Panda ("Pan Dee");
    Panda p2 = new Panda ("Pan Dah");

    Console.WriteLine (p1.Name);      // Pan Dee
    Console.WriteLine (p2.Name);      // Pan Dah

    Console.WriteLine (Panda.Population);   // 2
  }
}
```

Rezervisana reč public

Rezervisana reč `public` izlaže (stavlja na raspolaganje) članove jedne klase drugim klasama. U našem primeru, pošto polje `Name` klase `Panda` nije deklarisano kao `public` (javno), klasa `Test` ne može da mu pristupa. Deklarisanjem svog člana kao `public`, tip poručuje sledeće: „Ovo hoću da drugi tipovi vide – sve ostalo su moji privatni detalji implementacije“. U objektno orijentisanom modelu kaže se da javni članovi *kapsuliraju* (engl. *encapsulate*) privatne članove tipa.

Konverzije tipova

C# može da pretvara instance jednog tipa u drugi, kompatibilan tip. U postupku konverzije uvek nastaje nova vrednost od postojeće. Konverzije mogu biti *implicitne* ili *eksplicitne*: implicitne konverzije se odvijaju automatski, dok je za eksplicitne konverzije potrebno eksplicitno *pretvaranje* (engl. *casting*). U primeru koji sledi, *implicitno* pretvaramo instancu tipa `int` u tip `long` (koji ima dvostruko veći broj bitova od tipa `int`), ali *eksplicitno* pretvaramo `int` u tip `short` (koji ima upola manji kapacitet od tipa `int`):

```
int x = 12345;       // int je 32-bitni ceo broj
long y = x;          // Implicitna konverzija u 64-bitni ceo broj
short z = (short)x;  // Eksplicitna konverzija u 16-bit ceo broj
```

Implicitne konverzije su dozvoljene kada su ispunjena oba sledeća uslova:

- Kompajler može garantovati da će operacija biti uspešna.
- Pri konverziji se ne gube informacije.[1]

Nasuprot tome, *ekplicitne* konverzije su obavezne u jednom od sledećih slučajeva:

- Prevodilac ne može da garantuje uspešnu konverziju.
- Pri konverziji može doći do gubljenja informacija.

(Ako kompajler može da utvrdi da će konverzija *uvek* biti neuspešna, zabranjene su obe vrste konverzija. Konverzije u kojima učestvuju generički tipovi, u određenim uslovima mogu se takođe završiti neuspehom – videti odeljak „Parametri tipa i konverzije“ u poglavlju 3.)



Numeričke konverzije koje smo upravo razmotrili, ugrađene su u jezik. C# podržava još i *konverzije referenci* i *konverzije pri pakovanju/raspakivanju* (videti poglavlje 3), kao i *namenske konverzije* (videti odeljak „Preklapanje operatora“ na strani 142 u poglavlju 4). Pošto pri namenskim konverzijama kompajler ne nameće pravila koja smo naveli, moguće je da se loše projektovani tipovi drugačije ponašaju.

Vrednosni tipovi i referentni tipovi

Svaki tip u jeziku C# pripada jednoj od sledećih kategorija:

- Vrednosni tipovi
- Referentni tipovi
- Generički parametri tipa
- Pokazivački tipovi



U ovom odeljku razmotrićemo vrednosne tipove i referentne tipove. Generičke parametre tipa razmotrićemo u odeljku „Generičke komponente“ na strani 94 poglavlja 3, a pokazivačkim tipovima bavićemo se na strani 159 poglavlja 4.

Vrednosnim tipovima (engl. *value types*) pripada većina ugrađenih tipova (a to su svi numerički tipovi, tip `char` i tip `bool`), te namenske strukture (tip `struct`) i nabrajanja (tip `enum`).

Referentni tipovi (engl. *reference types*) jesu sve klase i svi nizovi, delegati i interfejsi. (Toj kategoriji pripada i ugrađen tip `string`.)

1. Treba imati u vidu da se pri konverziji vrlo velikih vrednosti tipa `long` u `double` gubi deo preciznosti.

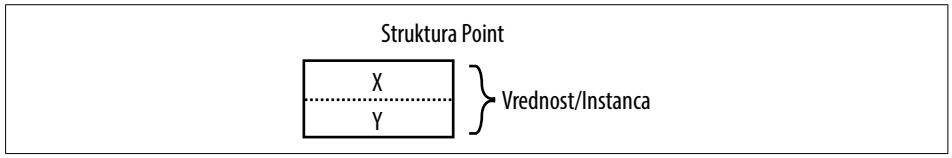
Suštinsku razliku između vrednosnih i referentnih tipova čini način na koji se oni obrađuju u memoriji.

Vrednosni tipovi

Sadržaj promenljive *vrednosnog tipa* jeste određena konstanta ili vrednost. Na primer, sadržaj ugrađenog vrednosnog tipa `int` čini 32 bita podataka.

Namenski vrednosni tip možete definisati pomoću rezervisane reči `struct` (slika 2-1):

```
public struct Point { public int X, Y; }
```



Slika 2-1. Izgled instance vrednosnog tipa u memoriji.

Dodeljivanjem instance vrednosnog tipa drugoj promenljivoj, izvorna instanca se uvek *kopira*.

Na primer:

```
static void Main()
{
  Point p1 = new Point();
  p1.X = 7;

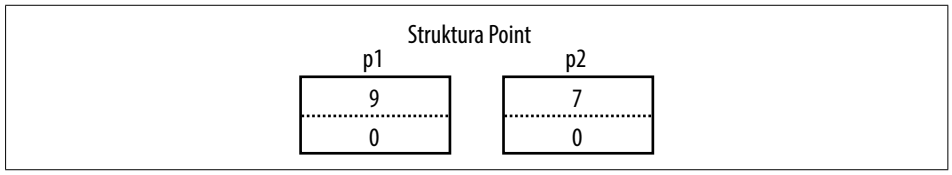
  Point p2 = p1;             // Dodeljivanje prouzrokuje kopiranje

  Console.WriteLine (p1.X);  // 7
  Console.WriteLine (p2.X);  // 7

  p1.X = 9;                  // Nova vrednost za p1.X

  Console.WriteLine (p1.X);  // 9
  Console.WriteLine (p2.X);  // 7
}
```

Slika 2-2 prikazuje da su promenljive `p1` i `p2` smeštene u memoriju nezavisno jedna od druge.



Slika 2-2. Pri operaciji dodeljivanja, instanca vrednosnog tipa se kopira.

Referentni tipovi

Referentni tip je složeniji od vrednosnog i čine ga dva dela: *objekat* i *referenca* na taj objekat. Sadržaj promenljive ili konstante referentnog tipa jeste referenca na objekat koji sadrži vrednost. Evo kako izgleda tip `Point` iz prethodnog primera kada ga napišemo u obliku klase, a ne kao tip `struct` (slika 2-3):

```
public class Point { public int X, Y; }
```



8
LINQ upiti

LINQ (Language Integrated Query – upit integrisan u jezik) jeste skup mogućnosti koje jezik C# i Framework pružaju za pisanje strukturiranih upita za pretraživanje lokalnih kolekcija objekata i udaljenih izvora podataka, na način koji ne narušava bezbednost tipova. LINQ je uveden u verziji C# 3.0 i Framework 3.5.

LINQ omogućava pretraživanje svake kolekcije koja implementira interfejs `IEnumerable<T>`, bez obzira na to da li je u pitanju niz vrednosti, lista ili XML DOM dokument, kao i udaljene izvore podataka, kao što su tabele na SQL Serveru. LINQ pruža prednosti i proveravanja ispravne upotrebe tipova u vreme prevođenja i dinamičkog sastavljanja upita.

Ovo poglavlje opisuje arhitekturu LINQ-a i osnove pisanja upita. Svi osnovni tipovi su definisani u imenskim prostorima `System.Linq` i `System.Linq.Expressions`.



Primeri u ovom i u naredna dva poglavlja ugrađeni su u interaktivnu alatku za izradu upita koja se zove LINQPad; možete je preuzeti na *http://www.linqpad.net*.

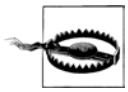
Uvod

Osnovne jedinice podataka s kojima LINQ radi jesu *sekvence* i *elementi*. Sekvenca je svaki objekat koji implementira interfejs `IEnumerable<T>` a element je svaka stavka sekvence. U primeru koji sledi, niz `names` je sekvenca, a `"Tom"`, `"Dick"` i `"Harry"` su njeni elementi:

```
string[] names = { "Tom", "Dick", "Harry" };
```

Ovakvu sekvencu zovemo *lokalna sekvenca* zato što predstavlja lokalnu kolekciju objekata u memoriji.

Operator za upit je metoda koja transformiše sekvencu. Tipičan operator za upit prihvata *ulaznu sekvencu* koju transformiše u *izlaznu sekvencu*. U klasi `Enumerable` u imenskom prostoru `System.Linq`, postoji oko 40 operatora za upite – a svi su implementirani u obliku statičkih proširenih metoda. To su *standardni operatori za upite*.



Upiti koji pretražuju lokalne sekvence zovu se lokalni upiti ili *LINQ-to-object* upiti.

LINQ podržava i sekvence koje se mogu dinamički popunjavati iz određenog udaljenog izvora podataka kao što SQL Server. Te sekvence dodatno implementiraju interfejs `IQueryable<T>` i podržane su pomoću odgovarajućeg standardnog skupa operatora za upite u klasi `Queryable`. To ćemo detaljnije razmotriti u odeljku „Interpretirani upiti“, na strani 314 ovog poglavlja.

Upit je izraz koji, kada ga nabrojite, transformiše sekvence pomoću operatora za upite. Najjednostavniji upit se sastoji od jedne ulazne sekvence i jednog operatora. Na primer, operator `Where` možemo primeniti na jednostavan niz da bismo iz njega izdvojili elemente koji sadrže barem četiri znaka, na sledeći način:

```
string[] names = { "Tom", "Dick", "Harry" };
IEnumerable<string> filteredNames = System.Linq.Enumerable.Where
                                    (names, n => n.Length >= 4);
foreach (string n in filteredNames)
  Console.WriteLine (n);

Dick
Harry
```

Pošto su standardni operatori za upite implementirani u obliku proširenih metoda, metodu `Where` možemo pozvati direktno za objekat `names` – isto kao da je to metoda instance:

```
IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
```

Da bi to moglo da se prevede, morate uvesti imenski prostor `System.Linq`. Evo kako izgleda kompletan primer:

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

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry" };

    IEnumerable<string> filteredNames = names.Where (n => n.Length >= 4);
    foreach (string name in filteredNames) Console.WriteLine (name);
  }
}
```



Kôd bismo mogli još skratiti ako za promenljivu `filteredNames` zadamo implicitno određivanje tipa:

```
var filteredNames = names.Where (n => n.Length >= 4);
```

Međutim, time smo umanjili razumljivost koda, naročito izvan integrisanog razvojnog okruženja, gde nema kratkih opisa koji bi nam pomogli.

U ovom poglavlju, izbegavaćemo implicitno određivanje tipa rezultata upita osim na mestima gde je to obavezno (kao što ćemo videti u nastavku poglavlja, u odeljku „Strategije za projekcije“, na strani 312), ili kad je tip upita nebitan za sam primer.

Većina operatora za upite prihvata lambda izraz kao argument. Lambda izraz olakšava razumevanje i formiranje upita. U našem primeru, lambda izraz je sledeći:

```
n => n.Length >= 4
```

Ulazni argument odgovara jednom ulaznom elementu. U ovom primeru, ulazni argument `n` predstavlja svako ime u nizu a tip mu je `string`. Operator `Where` zahteva da lambda izraz vraća vrednost tipa `bool`, koja ako je `true`, znači da taj element treba uključiti u izlaznu sekvencu. Potpis operatora `Where` izgleda ovako:

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

Sledeći upit izdvaja sva imena koja sadrže slovo „a“:

```
IEnumerable<string> filteredNames = names.Where (n => n.Contains ("a"));

foreach (string name in filteredNames)
  Console.WriteLine (name);             // Harry
```

Dosad smo upite sastavljali pomoću proširenih metoda i lambda izraza. Kao što ćemo uskoro videti, ta strategija pruža velike mogućnosti kombinovanja jer omogućava ulančavanje operatora za upite. U ovoj knjizi, to zovemo *tečna sintaksa* (engl. *fluent syntax*).[1] C# podržava i drugi oblik sintakse za pisanje upita, nazvan sintaksa *izraza upita*. Ovako izgleda naš prethodni upit napisan u obliku izraza upita:

```
IEnumerable<string> filteredNames = from n in names
                                    where n.Contains ("a")
                                    select n;
```

Tečna sintaksa i sintaksa za upite međusobno su komplementarne. U naredna dva odeljka detaljno razmatramo oba oblika.

Tečna sintaksa

Tečna sintaksa je fleksibilna i fundamentalna. U ovom odeljku opisujemo kako se operatori za upite mogu ulančavati da bi se formirali složeniji upiti – i pokazati zbog čega su proširene metode važne za taj postupak. Opisujemo i kako treba formulisati lambda izraze za operator upita i uvodimo više novih operatora za upite.

Ulančavanje operatora za upite

U prethodom odeljku prikazali smo dva jednostavna upita, svaki s po jednim operatorom. Da biste sastavili složeniji upit, izrazu treba da dodate druge operatore, čime formirate lanac. Ilustracije radi, upit koji sledi izdvaja sva imena koja sadrže slovo „a“, sortira ih po dužini, a rezultat zatim pretvara u velika slova:

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

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
```

1. Izraz potiče iz rada autora Erika Evansa i Martina Faulera na temu tečnih interfejsa.

```
    IEnumerable<string> query = names
      .Where   (n => n.Contains ("a"))
      .OrderBy (n => n.Length)
      .Select  (n => n.ToUpper());

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

JAY
MARY
HARRY
```



Promenljiva, `n`, u našem primeru, privatna je u svakom lambda izrazu. Ime `n` možemo upotrebiti višekratno iz istih razloga zbog kojih možemo višekratmo upotrebiti `c` u sledećoj metodi:

```
void Test()
{
  foreach (char c in "string1") Console.Write (c);
  foreach (char c in "string2") Console.Write (c);
  foreach (char c in "string3") Console.Write (c);
}
```

`Where`, `OrderBy` i `Select` su standardni operatori za upite koji se prevode u proširene metode klase `Enumerable` (ako uvezete imenski prostor `System.Linq`).

Već smo uveli operator `Where`, koji formira filtriranu verziju ulazne sekvence. Operator `OrderBy` vraća sortiranu verziju svoje ulazne sekvence; metoda `Select` vraća izlaznu sekvencu u kojoj je svaki ulazni element transformisan ili *projektovan* pomoću zadatog lambda izraza (`n.ToUpper()`, u ovom slučaju). Podaci teku lancem operatora sleva nadesno, tako da se podaci prvo filtriraju, zatim sortiraju, a onda projektuju.



Operator upita nikad ne menja ulaznu sekvencu, nego uvek vraća novu sekvencu. To je u skladu s modelom *funkcionalnog programiranja*, kojim je LINQ inspirisan.

Ovako izgledaju potpisi pomenutih proširenih metoda (potpis metode `OrderBy` je neznatno pojednostavljen):

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

public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)

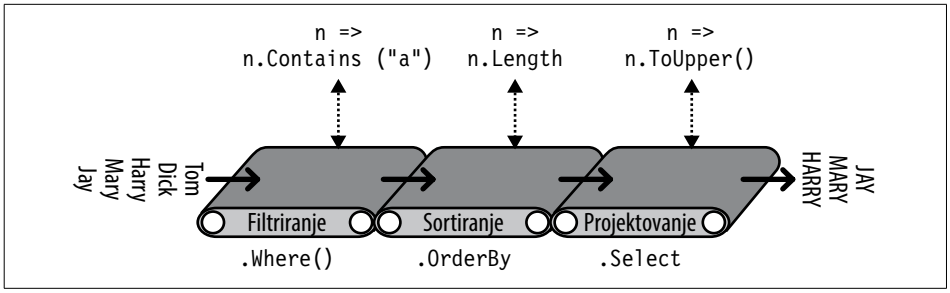
public static IEnumerable<TResult> Select<TSource,TResult>
  (this IEnumerable<TSource> source, Func<TSource,TResult> selector)
```

Kada se operatori za upite ulančavaju kao u ovom primeru, izlazna sekvenca jednog operatora postaje ulazna sekvenca operatora koji mu sledi. Konačan rezultat liči na proizvodnu liniju s transportnim trakama, kao što ilustruje slika 8-1.

Identičan upit možemo zadati i u obliku *korak po korak*, kao što sledi:

```
// Da bi se ovo prevelo, morate uvesti imenski prostor System.Linq:

IEnumerable<string> filtered   = names   .Where   (n => n.Contains ("a"));
IEnumerable<string> sorted     = filtered.OrderBy (n => n.Length);
IEnumerable<string> finalQuery = sorted  .Select  (n => n.ToUpper());
```



Slika 8-1. Ulančavanje operatora upita.

Upit `finalQuery` je po sadržaju identičan upitu `query` koji smo ranije napisali. Osim toga, i svaki međukorak je ispravan upit koji možemo izvršiti:

```
foreach (string name in filtered)
  Console.Write (name + "|");        // Harry|Mary|Jay|

Console.WriteLine();
foreach (string name in sorted)
  Console.Write (name + "|");        // Jay|Mary|Harry|

Console.WriteLine();
foreach (string name in finalQuery)
  Console.Write (name + "|");        // JAY|MARY|HARRY|
```

Zbog čega su važne proširene metode

Umesto sintakse s proširenim metodama, operatore za upite možete pozivati i pomoću standardne sintakse za statičke metode. Na primer:

```
IEnumerable<string> filtered = Enumerable.Where (names,
                                                 n => n.Contains ("a"));
IEnumerable<string> sorted = Enumerable.OrderBy (filtered, n => n.Length);
IEnumerable<string> finalQuery = Enumerable.Select (sorted,
                                                    n => n.ToUpper());
```

To je, zapravo, oblik u koji kompajler prevodi pozive proširenim metodama. Međutim, izbegavanje proširenih metoda ima svoju cenu, ako pokušavate da napišete upit koji se sastoji od samo jedne naredbe, kao što smo to ranije uradili. Vratimo se na upit u jednoj naredbi – prvo sa sintaksom pomoću proširene metode:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                 .OrderBy (n => n.Length)
                                 .Select  (n => n.ToUpper());
```

Njegov prirodan linearni oblik odražava tok podataka sleva nadesno i postavlja lambda izraze neposredno pored odgovarajućih operatora u upitu (*infiksna* notacija). Bez proširenih metoda, upit više nije *tečan*:

```
IEnumerable<string> query =
  Enumerable.Select (
    Enumerable.OrderBy (
      Enumerable.Where (
        names, n => n.Contains ("a")
      ), n => n.Length
    ), n => n.ToUpper()

  );
```

Kombinovanje lambda izraza

U prethodnim primerima, operatoru `Where` prosledili smo sledeći lambda izraz:

```
n => n.Contains ("a")      // Ulazni tip=string, povratni tip=bool.
```



Lambda izraz koji prihvata ulaznu vrednost i vraća `bool` zove se *predikat*.

Svrha lambda izraza zavisi od konkretnog operatora za upite. Uz operator `Where`, lambda izraz određuje da li element treba uključiti u izlaznu sekvencu. U slučaju operatora `OrderBy`, lambda izraz preslikava svaki element u ulaznoj sekvenci na njegovo mesto po ključu za sortiranje. Uz operator `Select`, lambda izraz određuje kako se svaki element ulazne sekvence transformiše pre nego što se umetne u izlaznu sekvencu.



Lambda izraz pridružen operatoru za upite uvek deluje na pojedinačne elemente ulazne sekvence – a ne na sekvencu kao celinu.

Operator za upite izračunava vrednost lambda izraza na zahtev – najčešće jedanput po elementu ulazne sekvence. Lambda izrazi omogućavaju da operatorima za upite prosleđujete vlastitu logiku. To čini operatore za upite vrlo raznovrsnim – kao i jednostavnim „ispod haube“. Ovako izgleda kompletna implementacija metode `Enumerable.Where`, bez koda za obradu izuzetaka:

```
public static IEnumerable<TSource> Where<TSource>
  (this IEnumerable<TSource> source, Func<TSource,bool> predicate)
{
  foreach (TSource element in source)
    if (predicate (element))
      yield return element;

}
```

Lambda izrazi i potpisi delegata Func

Standardni operatori za upite koriste generičke delegate `Func`. `Func` je porodica generičkih delegata opšte namene u imenskom prostoru `System.Linq`, koji su definisani za sledeću svrhu:

> Argumenti tipa u delegatu `Func` zadaju se istim redosledom kao i u lambda izrazima.

Stoga `Func<TSource,bool>` odgovara lambda izrazu `TSource=>bool`: onom koji prihvata argument `TSource` i vraća vrednost `bool`.

Slično tome, `Func<TSource,TResult>` odgovara lambda izrazu `TSource=>TResult`.

`Func` delegati su nabrojani u odeljku „Lambda izrazi“, na strani 121 poglavlja 4.

Lambda izrazi i tip elemenata

Standardni operatori za upite koriste sledeća generička imena tipova:

Generički tip	Značenje
TSource	Tip elemenata ulazne sekvence
TResult	Tip elemenata izlazne sekvence – različit od TSource
TKey	Tip elemenata za *ključ* po kojem se elementi sortiraju, grupišu ili spajaju

Tip TSource određuje ulazna sekvenca. Tipovi TResult i TKey *izvode se iz lambda izraza*.

Na primer, pogledajte potpis operatora Select:

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

Delegat Func<TSource,TResult> odgovara lambda izrazu TSource=>TResult: onom koji preslikava jedan *ulazni element* u jedan *izlazni element*. Pošto su TSource i TResult različiti tipovi, lambda izraz može da menja tip svakog elementa. Osim toga, lambda izraz *određuje tip izlazne sekvence*. Naredni upit pomoću operatora Select transformiše ulazne elemente tipa string u izlazne elemente ceobrojnog tipa:

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<int> query = names.Select (n => n.Length);

foreach (int length in query)
  Console.Write (length + "|");    // 3|4|5|4|3|
```

Kompajler *izvodi* tip TResult iz povratne vrednosti lambda izraza. U ovom slučaju, za TResult je izveden tip int.

Operator za upite Where je jednostavniji i ne zahteva izvođenje tipa izlazne sekvence, pošto su ulazni i izlazni elementi istog tipa. To je logično jer operator samo filtrira elemente; on ih ne *transformiše*:

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

I najzad, razmotrite potpis operatora OrderBy:

```
// Malo pojednostavljen:
public static IEnumerable<TSource> OrderBy<TSource,TKey>
  (this IEnumerable<TSource> source, Func<TSource,TKey> keySelector)
```

Func<TSource,TKey> preslikava jedan ulazni element u *ključ za sortiranje*. Tip TKey se izvodi iz lambda izraza nezavisno od tipova ulaznih i izlaznih elemenata. Na primer, listu imena mogli smo da sortiramo po dužini imena (ključ tipa int) ili alfabetskim redosledom (ključ tipa string):

```
string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };
IEnumerable<string> sortedByLength, sortedAlphabetically;
sortedByLength       = names.OrderBy (n => n.Length);   // ključ tipa int

sortedAlphabetically = names.OrderBy (n => n);          // ključ tipa string
```



Operatore za upite u klasi Enumerable možete pozivati i pomoću tradicionalnih delegata koji referenciraju metode umesto lambda izraza. Takvo rešenje je dobro za pojednostavljivanje određenih vrsta lokalnih upita – naročito u slučaju LINQ u XML – što je opisano u poglavlju 10. Međutim, nije upotrebljivo za sekvence koje implementiraju interfejs IQueryable<T> (na primer, kada pretražujete bazu podataka), zato što su operatorima definisani u interfejsu Queryable potrebni lambda izrazi da bi formirali stabla izraza. To ćemo razmotriti u nastavku poglavlja, u odeljku „Interpretirani upiti“, na strani 314.

Prirodni redosled

U LINQ-u je značajan redosled elemenata ulazne sekvence. Neki operatori za upite, kao što su `Take`, `Skip` i `Reverse`, uzimaju u obzir taj redosled.

Operator `Take` izdvaja prvih `x` elemenata ulazne sekvence, a ostale odbacuje:

```
int[] numbers  = { 10, 9, 8, 7, 6 };
IEnumerable<int> firstThree = numbers.Take (3);     // { 10, 9, 8 }
```

Operator `Skip` preskače prvih `x` elemenata i vraća sve preostale:

```
IEnumerable<int> lastTwo    = numbers.Skip (3);     // { 7, 6 }
```

`Reverse` radi tačno ono što biste očekivali:

```
IEnumerable<int> reversed   = numbers.Reverse();    // { 6, 7, 8, 9, 10 }
```

U lokalnim upitima (LINQ-to-objects), operatori kao što su `Where` i `Select` zadržavaju izvorni redosled elemenata u ulaznoj sekvenci (kao što to čine i drugi operatori za upite, osim onih koji izričito menjaju redosled).

Drugi operatori

Ne vraćaju svi operatori za upite izlaznu sekvencu. Operatori nad *elementima* izdvajaju određeni element iz ulazne sekvence; primeri toga su operatori `First`, `Last` i `ElementAt`:

```
int[] numbers    = { 10, 9, 8, 7, 6 };
int firstNumber  = numbers.First();                       // 10
int lastNumber   = numbers.Last();                        // 6
int secondNumber = numbers.ElementAt(1);                  // 9
int secondLowest = numbers.OrderBy(n=>n).Skip(1).First(); // 7
```

Operatori za agregiranje (grupni) vraćaju skalarnu vrednost, najčešće numeričkog tipa:

```
int count = numbers.Count();          // 5;
int min = numbers.Min();              // 6;
```

Kvantifikatori vraćaju vrednost tipa `bool`:

```
bool hasTheNumberNine = numbers.Contains (9);          // true
bool hasMoreThanZeroElements = numbers.Any();          // true
bool hasAnOddElement = numbers.Any (n => n % 2 != 0);  // true
```

Pošto ti operatori ne vraćaju kolekciju, s njihovim rezultatom ne možete dalje pozivati druge operatore za upite. Drugim rečima, oni mogu biti samo poslednji operator u upitu.

Neki operatori za upite prihvataju dve ulazne sekvence. Primeri toga su operator `Concat`, koji nadovezuje jednu sekvencu na drugu, i operator `Union`, koji radi to isto, ali uklanja duplikate:

```
int[] seq1 = { 1, 2, 3 };
int[] seq2 = { 3, 4, 5 };
IEnumerable<int> concat = seq1.Concat (seq2);    //  { 1, 2, 3, 3, 4, 5 }
IEnumerable<int> union  = seq1.Union (seq2);     //  { 1, 2, 3, 4, 5 }
```

Operatori za spajanje sekvenci takođe pripadaju ovoj kategoriji. U poglavlju 9 detaljno su objašnjeni svi operatori za upite.

Izrazi upita

C# ima sintaksnu prečicu za pisanje LINQ upita, što se zove *izraz upita*. Nasuprot raširenom verovanju, izraz upita nije način da se SQL ugradi u C# kôd. U stvari, dizajn izraza

upita bio je inspirisan prvenstveno konceptom *kombinovanja lista* (engl. *list comprehension*) iz jezika za funkcionalno programiranje kao što su LISP i Haskell, mada je i SQL imao kozmetički uticaj.



U ovoj knjizi, sintaksu za izraze upita nazivamo jednostavno „sintaksa za upite“.

U prethodnom odeljku, napisali smo upit s tečnom sintaksom koji izdvaja znakovne nizove koji sadrže slovo „a“, sortira ih po dužini i pretvara u velika slova. Ovako izgleda isti upit u obliku sintakse za upite:

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

class LinqDemo
{
  static void Main()
  {
    string[] names = { "Tom", "Dick", "Harry", "Mary", "Jay" };

    IEnumerable<string> query =
      from    n in names
      where   n.Contains ("a")     // Filtrira elemente
      orderby n.Length             // Sortira elemente
      select  n.ToUpper();         // Pretvara svaki element (projekcija)

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

JAY
MARY
HARRY
```

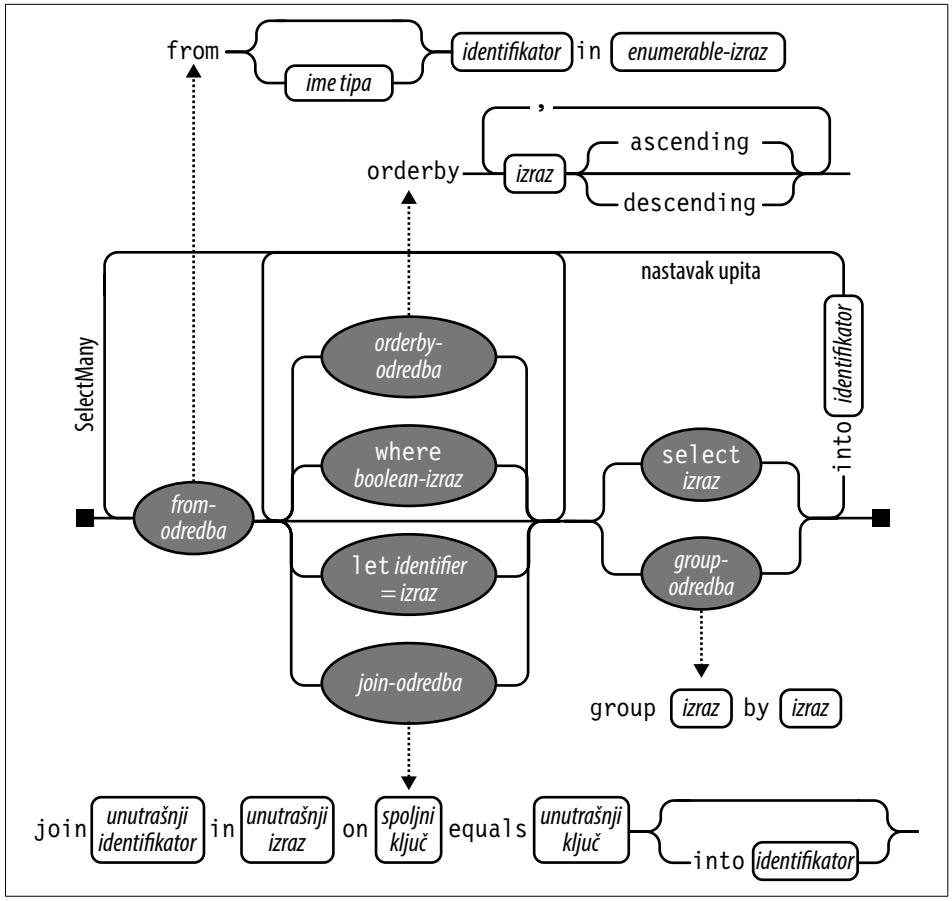
Izrazi upita uvek počinju odredbom `from` a završavaju se odredbom `select` ili `group`. Odredba `from` deklariše *promenljivu opsega* (u ovom slučaju, `n`), koju možete zamisliti kao način za pristupanje jednom po jednom elementu sekvence – otprilike kao `foreach`. Slika 8-2 ilustruje kompletnu sintaksu u obliku „šinskog“ dijagrama.



Da biste čitali taj dijagram, krenite od levog kraja a zatim nastavite po određenom „koloseku“ kao da ste voz. Na primer, iza obavezne odredbe `from`, možete opciono dodati odredbu `orderby`, `where`, `let` ili `join`. Iza toga, možete nastaviti do odredbe `select` ili `group`, ili se vratiti da biste dodali novu odredbu `from`, `orderby`, `where`, `let` ili `join`.

Kompajler obrađuje izraz upita tako što ga prevodi u tečnu sintaksu. To radi na sasvim mehanički način – vrlo slično načinu na koji naredbe `foreach` prevodi u pozive metoda `GetEnumerator` i `MoveNext`. To znači da sve što možete napisati u obliku izraza upita, možete napisati i u obliku tečne sintakse. Naš primer upita kompajler prevodi (prvo) u sledeći oblik:

```
IEnumerable<string> query = names.Where   (n => n.Contains ("a"))
                                 .OrderBy (n => n.Length)
                                 .Select  (n => n.ToUpper());
```



Slika 8-2. Sintaksa za upite.

Zatim razrešava operatore `Where`, `OrderBy` i `Select` prema istim pravilima koja bi važila kada bi upit bio napisan u tečnoj sintaksi. U ovom slučaju, oni se pretvaraju u pozive odgovarajućih proširenih metoda klase `Enumerable` jer je uvezen imenski prostor `System.Linq` i sekvenca `names` implementira interfejs `IEnumerable<string>`. Međutim, kada prevodi izraze upita, kompajler nije posebno sklon klasi `Enumerable`. Možete ga zamisliti kao da u naredbu prvo mehanički umeće reči „Where“, „OrderBy“ i „Select“, a zatim prevodi rezultat kao da ste ta imena metoda vi napisali. To pruža veliku fleksibilnost u pogledu načina razrešavanja tih metoda. Na primer, upiti u baze podataka koje ćemo pisati u odeljcima koji slede, povezivaće se s proširenim metodama klase `Queryable`.



Ako iz našeg programa uklonimo direktivu `using System.Linq`, upit se neće prevesti zato što metode `Where`, `OrderBy` i `Select` neće imati s čim da se povežu. Izrazi upita *ne mogu se prevesti* dok ne uvezete odgovarajući imenski prostor (ili za svaki operator napišete odgovarajuću metodu instance!).



13

Dijagnostika i kodni ugovori

Kada stvari krenu naopako, važno je imati informacije koje će vam pomoći da postavite dijagnozu problema. Integrisano razvojno okruženje ili alatka za otkrivanje grešaka mogu biti od velike pomoći ka tom cilju – ali su uglavnom na raspolaganju samo tokom razvoja aplikacije. Pošto aplikaciju otpremite korisnicima, sama aplikacija mora da prikuplja i beleži dijagnostičke podatke. Da bi zadovoljio taj zahtev, .NET Framework nudi skup alatki za beleženje dijagnostičkih podataka, nadziranje ponašanja aplikacije, otkrivanje grešaka pri izvršavanju i integrisanje s alatkama za otkrivanje grešaka, ako su one na raspolaganju.

.NET Framework omogućava i da koristite *kodne ugovore* (engl. *code contracts*). Uvedeni u verziji Frameworka 4.0, kodni ugovori omogućavaju metodama interakciju kroz skup međusobnih obaveza koje čine da metode *rano* prekidaju rad ako se te obaveze ne ispunjavaju.

Tipovi koje pominjemo u ovom poglavlju definisani su prevashodno u imenskim prostorima `System.Diagnostics` i `System.Diagnostics.Contracts`.

Uslovno prevođenje koda

Pretprocesorske direktive omogućavaju da uslovno prevedete proizvoljan odeljak C# koda. To su specijalne naredbe namenjene kompajleru koje počinju simbolom # (i, za razliku od ostalih konstrukata jezika C#, moraju se zadavati svaka u zasebnom redu). Logički gledano, one se izvršavaju pre glavng postupka prevođenja koda (mada, u praksi, kompajler ih obrađuje tokom faze leksičke analize koda). Pretprocesorske direktive za uslovno prevođenje koda jesu: `#if`, `#else`, `#endif` i `#elif`.

Direktive `#if` nalažu kompajleru da zanemari određeni odeljak koda ako nije definisan zadati *simbol*. Simbol možete definisati pomoću pretprocesorske direktive `#define` ili odgovarajuće opcije koju zadate kompajleru. Direktiva `#define` važi samo za sadržaj *datoteke* u kojoj je zadata, dok opcija kompajlera važi za ceo *sklop*:

```
#define TESTMODE            // direktive #define morate zadati na početku datoteke
                            // Imena simbola se po konvenciji pišu velikim slovima.
using System;

class Program
{
  static void Main()
  {
#if TESTMODE
```

```
      Console.WriteLine ("in test mode!");     // REZULTAT: in test mode!
#endif
  }
}
```

Kada bismo izbrisali prvi red, program bi se preveo u izvršnu verziju iz koje je bi bila uklonjena naredba `Console.WriteLine`, kao kada bismo je u izvornoj verziji pretvorili u komentar.

Direktiva `#else` je analogna naredbi `else` jezika C#, a `#elif` je ekvivalentna direktivi `#else` kojoj sledi `#if`. Pomoću operatora `||`, `&&` i `!` možete obavljati logičke operacije *disjunkcije*, *konjukcije* i *negacije*:

```
#if TESTMODE && !PLAYMODE      // if TESTMODE and not PLAYMODE
  ...
```

Međutim, imajte u vidu da ne sastavljate običan C# izraz, a simboli s kojima radite nemaju apsolutno nikakve veze s *promenljivama* – ni statičkim, ni drugačijim.

Da biste definisali simbol koji važi za ceo sklop, upotrebite opciju kompajlera `/define` kada kôd prevodite s komandne linije:

```
csc Program.cs /define:TESTMODE,PLAYMODE
```

Visual Studio omogućava da zadate opcije za uslovno prevođenje, u odeljku Project Properties.

Ako ste određeni simbol definisali na nivou celog sklopa, ali želite da se on u određenoj datoteci zanemari, to možete zadati pomoću direktive `#undef`.

Uslovno prevođenje u poređenju sa statičkim opcijama

Prethodni primer se može implementirati i pomoću jednostavnog statičkog polja:

```
static internal bool TestMode = true;

static void Main()
{
  if (TestMode) Console.WriteLine ("in test mode!");
}
```

Prednost ovog rešenja je to što omogućava konfigurisanje u vreme izvršavanja koda. Zašto bismo se onda opredelili za uslovno prevođenje? Razlog je to što uslovno prevođenje omogućava da radite i stvari koje nisu moguće pomoću statičkih opcija, kao što su:

- Uslovno uključivanje atributa
- Menjanje deklarisanog tipa promenljive
- Menjanje imenskog prostora ili alijasa tipa unutar direktive `using` – na primer:

```
using TestType =
  #if V2
     MyCompany.Widgets.GadgetV2;
  #else
     MyCompany.Widgets.Gadget;
  #endif
```

Unutar direktive za uslovno prevođenje možete zadati i vrlo složene promene, što vam omogućava da birate između starih i novih verzija, da pišete biblioteke koje se prevode za razne verzije Frameworka ili da koristite mogućnosti najnovije verzije Frameworka kada su na raspolaganju.

Još jedna prednost uslovnog prevođenja jeste to što se kôd za otkrivanje grešaka može odnositi na tipove koji se nalaze u sklopovima koji neće biti sastavni deo konačne verzije aplikacije koju ćete distribuirati.

Atribut Conditional

Atribut `Conditional` nalaže kompajleru da zanemari pozive određenoj klasi ili metodi, ako zadati simbol nije definisan.

Da biste videli koliko je to korisno, pretpostavimo da ste napisali sledeću metodu za evidentiranje satusnih podataka u datoteku:

```
static void LogStatus (string msg)
{
  string logFilePath = ...
  System.IO.File.AppendAllText (logFilePath, msg + "\r\n");
}
```

A sada zamislite da želite da se ova metoda izvršava samo ako je definisan simbol `LOGGINGMODE`. Prvo rešenje je da pozive metodi `LogStatus` umetnete unutar direktive `#if`:

```
#if LOGGINGMODE
LogStatus ("Message Headers: " + GetMsgHeaders());
#endif
```

To daje savršen rezultat, ali je nezgrapno. Drugo rešenje je da direktivu `#if` umetnete unutar koda metode `LogStatus`. Međutim, to pravi problem ako se metoda `LogStatus` poziva na sledeći način:

```
LogStatus ("Message Headers: " + GetComplexMessageHeaders());
```

U tom obliku se metoda `GetComplexMessageHeaders` uvek poziva – što može biti uzrok slabijih performansi.

Funkcionalnost prvog rešenja možemo kombinovati s pogodnošću drugog ako metodi `LogStatus` pridružimo atribut `Conditional` (definisan u imenskom prostoru `System.Diagnostics`):

```
[Conditional ("LOGGINGMODE")]
static void LogStatus (string msg)
{
  ...
}
```

Time nalažemo kompajleru da pozive metodi `LogStatus` obrađuje kao da se nalaze unutar direktive `#if LOGGINGMODE`. Ako taj simbol nije definisan, svi pozivi metodi `LogStatus` uklanjaju se iz prevedene verzije – uključujući i iz izraza koji se metodi zadaju kao argument. (Iz tog razloga, izbegavaju se i svi sporedni efekti.) Ovo je izvodljivo čak i kad se metoda `LogStatus` i njen pozivalac nalaze u različitim sklopovima.



Još jedna korist od atributa `[Conditional]` jeste to što se ispunjavanje zadatog uslova proverava kada se prevodi *pozivalac* metode, a ne kada se prevodi *pozvana* metoda. To je dobro zato što omogućava da napišete biblioteku metoda kao što je `LogStatus` – i da prevedete samo jednu verziju te biblioteke.

U vreme izvršavanja koda, atribut `Conditional` se zanemaruje – to je naredba koja važi isključivo za kompajler.

Alternative za atribut Conditional

Atribut `Conditional` je beskoristan ako vam treba da određenu funkcionalnost uključujete ili isključujete u vreme izvršavanja koda: umesto atributa, morate primeniti neko rešenje sa statičkom promenljivom. To nameće pitanje kako elegantno zaobići izračunavanje vrednosti argumenata kad pozivate metode za uslovno beleženje podataka u dnevnik. Rešenje pomoću funkcije izgleda ovako:

```
using System;
using System.Linq;

class Program
{
  public static bool EnableLogging;

  static void LogStatus (Func<string> message)
  {
    string logFilePath = ...
    if (EnableLogging)
      System.IO.File.AppendAllText (logFilePath, message() + "\r\n");
  }
}
```

Lambda izraz omogućava pozivanje metode bez složene sintakse:

```
LogStatus ( () => "Message Headers: " + GetComplexMessageHeaders() );
```

Ako polje `EnableLogging` ima vrednost `false`, metoda `GetComplexMessageHeaders` se ne poziva.

Klase Debug i Trace

Klase `Debug` i `Trace` su statičke klase koje pružaju osnovne mogućnosti za beleženje podataka i ispitivanje da li su dati uslovi ispunjeni. Klase su međusobno vrlo slične; glavna razlika je njihova namena. Klasa `Debug` je namenjena upotrebi u verzijama koda koje se prevode u režimu `Debug`; klasa `Trace` je namenjena upotrebi i u režimu `Debug` i u režimu `Release`. U tom cilju:

Sve metode u klasi `Debug` definisane su s atributom `[Conditional("DEBUG")]`.
Sve metode u klasi `Trace` definisane su s atributom `[Conditional("TRACE")]`.

To znači da kompajler uklanja sve pozive metoda klasa `Debug` ili `Trace` osim kada definišete simbole `DEBUG` odnosno `TRACE`. Visual Studio standardno definiše oba simbola (`DEBUG` i `TRACE`) kada za projekat odaberete konfiguraciju *debug* – ali samo simbol `TRACE` u konfiguraciji *release*.

Obe klase, `Debug` i `Trace`, imaju metode `Write`, `WriteLine` i `WriteIf`. One standardno ispisuju poruke u prozor za rezultate dibagera:

```
Debug.Write     ("Data");
Debug.WriteLine (23 * 34);
int x = 5, y = 3;
Debug.WriteIf   (x > y, "x is greater than y");
```

Klasa `Trace` ima i metode `TraceInformation`, `TraceWarning` i `TraceError`. Razlika u ponašanju između njih i metoda `Write` zavisi od aktivnih osluškivača (objekti tipa `TraceListener`), koje opisujemo u odeljku „Klasa TraceListener“.

Metode Fail i Assert

Klase `Debug` i `Trace` obe imaju metodu `Fail` i `Assert`. Metoda `Fail` šalje poruku svakom objektu tipa `TraceListener` u kolekciji `Listeners` (videti naredni odeljak) klase `Debug` ili `Trace`. Svaki takav objekat standardno ispisuje poruku u prozor za rezultate alatke za otkrivanje grešaka (dibagera) i u okviru za dijalog:

```
Debug.Fail ("File data.txt does not exist!");
```

Dijalog koji se otvara nudi opcije ignore (zanemariti), abort (prekinuti) i retry (pokušati ponovo). Ova poslednja opcija omogućava da program povežete s nekim dibagerom, što je korisno za brzo dijagnostikovanje problema.

Metoda `Assert` samo poziva metodu `Fail` ako njen argument `bool` ima vrednost `false` – to se zove *zadavanje pretpostavke* (engl. *assertion)* i pokazuje da u kodu postoji greška ako pretpostavka nije potvrđena. Za taj slučaj možete opciono zadati odgovarajuću poruku:

```
Debug.Assert (File.Exists ("data.txt"), "Datoteka data.txt ne postoji!");
var result = ...
Debug.Assert (result != null);
```

Metode `Write`, `Fail` i `Assert` imaju i preklopljene verzije koje, osim teksta poruke, prihvataju i argument za opis kategorije greške tipa `string`, što može biti korisno pri obradi rezultata.

Umesto da zadate pretpostavku, možete generisati izuzetak ako je ispunjen suprotan uslov. To je uobičajena praksa kada se ispituje ispravnost argumenata metode:

```
public void ShowMessage (string message)
{
  if (message == null) throw new ArgumentNullException ("message");
  ...
}
```


Takve „pretpostavke“ se bezuslovno prevode i manje su fleksibilne jer rezultate nepotvrđene pretpostavke ne možete obraditi pomoću objekata `TraceListener`. Osim toga, tehnički govoreći, to nisu pretpostavke. Pretpostavka je nešto što, ako nije potvrđeno, pokazuje grešku u kodu tekuće metode. Pojava izuzetka pri ispitivanju ispravnosti argumenta metode znak je greške u kodu *pozivaoca* metode.



Uskoro ćemo videti kako *kodni ugovori* proširuju principe metoda `Fail` i `Assert` da bi pružili moćnija i fleksibilnija rešenja.

Klasa TraceListener

Obe klase – `Debug` i `Trace` – imaju svojstvo `Listeners`, čija je vrednost statička kolekcija osluškivača, tj. instanci klase `TraceListener`. Ti objekti su odgovorni za dalju obradu sadržaja koji proizvode metode `Write`, `Fail` i `Trace`.

Kolekcija `Listeners` svake klase standardno sadrži samo jedan osluškivač (`DefaultTraceListener`). Taj podrazumevani osluškivač ima dve ključne odlike:

- Kada se poveže s nekom alatkom za otkrivanje grešaka, kao što je Visual Studio, poruke osluškivača se ispisuju u prozoru za rezultate te alatke; u suprotnom, sadržaj poruka se zanemaruje.
- Kada se pozove metoda `Fail` (ili pretpostavka ne bude potvrđena), pojavljuje se okvir za dijalog koji korisniku nudi opcije da nastavi program (continue), da ga prekine (abort) ili da ponovo pokuša (retry) ili program poveže sa alatkom za otkrivanje grešaka (attach/debug) – bez obzira na to da li je programu već pridružen dibager.

To ponašanje možete promeniti ako iz kolekcije (opciono) uklonite podrazumevani osluškivač, a zatim kolekciji dodate jedan ili više vlastitih osluškivača. Osluškivač možete napisati od početka (tako što nalsedite klasu `TraceListener`) ili upotrebite neki od postojećih tipova:

- `TextWriterTraceListener` upisuje u tok, u objekat `TextWriter` ili u datoteku.
- `EventLogTraceListener` upisuje u Windowsov sistemskii dnevnik događaja.
- `EventProviderTraceListener` upisuje u podsistem ETW (Event Tracing for Windows) pod Windowsom Vista i novijim.
- `WebPageTraceListener` upisuje na ASP.NET veb stranicu.

Klasu `TextWriterTraceListener` dalje nasleđuju klase `ConsoleTraceListener`, `DelimitedListTraceListener`, `XmlWriterTraceListener` i `EventSchemaTraceListener`.



Nijedan od ovih osluškivača ne otvara okvir za dijalog kada se pozove metoda `Fail` – tako se ponaša samo `DefaultTraceListener`.

Primer koji sledi uklanja podrazumevani osluškivač klase `Trace`, zatim dodaje tri osluškivača – jedan koji podatke upisuje u datoteku, drugi koji ih ispisuje na konzolu i treći koji ih upisuje u Windowsov sistemski dnevnik događaja:

```
// Uklanja podrazumevani osluškivač:
Trace.Listeners.Clear();

// Dodaje pisač koji podatke dopisuje na kraj datoteke trace.txt:
Trace.Listeners.Add (new TextWriterTraceListener ("trace.txt"));

// Pribavlja izlazni tok konzole, koji zatim dodaje kao osluškivač:
System.IO.TextWriter tw = Console.Out;
Trace.Listeners.Add (new TextWriterTraceListener (tw));

// Definiše izvor za Windowsov dnevnik događaja a zatim dodaje osluškivač.
// Pošto CreateEventSource zahteva administratorska ovlašćenja, ovaj deo
// bi se obično obavio tokom instaliranja aplikacije.
if (!EventLog.SourceExists ("DemoApp"))
  EventLog.CreateEventSource ("DemoApp", "Application");

Trace.Listeners.Add (new EventLogTraceListener ("DemoApp"));
```

(Osluškivače možete dodati i pomoću konfiguracione datoteke aplikacije; to je zgodno jer ispitivačima aplikacije omogućava da konfigurišu osluškivač nakon prevođenja i sklapanja aplikacije – MSDN članak o tome nalazi se na *http://albahari.com/traceconfig*.)

U slučaju Windowsovog dnevnika događaja, poruke koje šaljete iz metoda `Write`, `Fail` ili `Assert` uvek se prikazuju u kategoriji „Information“, u prozoru prikaza Windowsovih događaja. Međutim, poruke koje šaljete pomoću metoda `TraceWarning` i `TraceError`, prikazuju se u kategoriji upozorenja (warning) ili u kategoriji grešaka (errors).

Klasa `TraceListener` ima i svojstvo `Filter` tipa `TraceFilter` koje se može podesiti radi filtriranja poruka koje osluškivač prima. Da biste to postigli, treba da napravite instancu jedne od unapred definisanih potklasa (`EventTypeFilter` ili `SourceFilter`), ili da nasledite klasu `TraceFilter` i redefinišete njenu metodu `ShouldTrace`. To možete iskoristiti da biste, na primer, filtrirali poruke po kategoriji.

Klasa `TraceListener` definiše i svojstva `IndentLevel` i `IndentSize` za zadavanje nivoa i veličine uvlaka, kao i svojstvo `TraceOutputOptions` za upisivanje dodatnih podataka:

```
TextWriterTraceListener tl = new TextWriterTraceListener (Console.Out);
tl.TraceOutputOptions = TraceOptions.DateTime | TraceOptions.Callstack;
```

Opcije zadate pomoću objekata `TraceOutputOptions` primenjuju se kada pozivate metode `Trace`:

```
Trace.TraceWarning ("Orange alert");

DiagTest.vshost.exe Warning: 0 : Orange alert
     DateTime=2007-03-08T05:57:13.6250000Z
     Callstack=   at System.Environment.GetStackTrace(Exception e, Boolean
needFileInfo)
     at System.Environment.get_StackTrace()     at ...
```

Pražnjenje i zatvaranje osluškivača

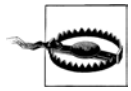
Neki osluškivači, kao što je `TextWriterTraceListener`, upisuju svoje podatke u tok, koji je podložan keširanju. To ima dve posledice:

- Poruka se možda neće odmah pojaviti u izlaznom toku ili datoteci.
- Osluškivač morate zatvoriti – ili barem isprazniti – pre nego što aplikacija završi rad jer ćete inače izgubiti ono što se nalazi u kešu (standardne veličine najviše 4 KB, ako upisujete u datoteku).

Klase `Trace` i `Debug` imaju statičke metode `Close` i `Flush` koje pozivaju metodu `Close` odnosno `Flush` svih osluškivača (koji pak pozivaju metodu `Close` odnosno `Flush` internog pisača ili toka s kojim rade). Metoda `Close` implicitno poziva metodu `Flush`, zatvara resurse operativnog sistema i sprečava dalje upisivanje podataka.

Opšte pravilo preporučuje da metodu `Close` pozivate na završetku aplikacije, a metodu `Flush` kad god vam zatreba da upišete tekuće podatke. To važi u slučajevima kada radite s pisačima čije odredište je tok ili datoteka.

Klase `Trace` i `Debug` imaju i svojstvo `AutoFlush`, koje, ako mu zadate vrednost `true`, pokreće izvršavanje metode `Flush` posle svake poruke.



Preporučuje se da svojstvo `AutoFlush` objekata `Debug` i `Trace` podesite na `true` ako radite s osluškivačima koji upisuju u tok ili datoteku. Ako ne uradite tako, kada se pojavi neobrađeni izuzetak ili katastrofalna greška, možete izgubiti poslednjih 4 KB dijagnostičkih podataka.

Kodni ugovori

Već smo ranije pomenuli koncept *pretpostavke*, pomoću koje utvrđujete da li su u programu ispunjeni određeni uslovi. Ako data pretpostavka nije potvrđena, to je znak da postoji greška, koja se obrađuje najčešće tako što se pokrene alatka za otkrivanje grešaka (kada je projekat u režimu Debug) odnosno generiše izuzetak (kada je projekat sklopljen u varijanti Release).

Pretpostavke slede princip da ako nešto krene naopako, najbolje je da to saznamo rano i blizu izvora greške. To je obično bolje nego da pokušamo da nastavimo s lošim podacima – što kasnije u programu može postati uzrok pogrešnih rezultata, neželjnih sporednih efekata ili izuzetka (a sve je to teže dijagnostikovati).

Tokom vremena, pretpostavke su se zadavale u dva oblika:

- Pozivanjem metode `Assert` klase `Debug` ili `Trace`
- Izazivanjem izuzetaka (kao što je `ArgumentNullException`)

Framework 4.0 je uveo novu mogućnost nazvanu *kodni ugovor* (engl. *code contract*), koji oba oblika zamenjuje jednoobraznim sistemom. Taj sistem omogućava da zadajete ne samo jednostavne pretpostavke, nego i moćnije pretpostavke u obliku *ugovora*.

Kodni ugovori potiču od principa „Design by Contract“ (dizajn zasnovan na ugovorima) programskog jezika Eiffel, gde se interakcija između funkcija odvija kroz sistem međusobnih obaveza i koristi. Ukratko, funkcija definiše *preduslove* (engl. *preconditions*) koje treba da ispuni klijent (pozivalac) funkcije, a zauzvrat ona klijentu garantuje pouzdano *konačno stanje* (engl. *postcondition*) kada funkcija završi svoj rad.

U imenskom prostoru `System.Diagnostics.Contracts` postoje dve vrste kodnih ugovora.



Iako su tipovi koji podržavaju kodne ugovore ugrađeni u .NET Framework, binarni pisač i alatke za statičku proveru ugovora treba zasebno preuzeti sa veb lokacije Microsoft DevLabs (*http://msdn.microsoft.com/devlabs*). Te alatke morate instalirati pre nego što počnete da koristite kodne ugovore u Visual Studiju.

Zbog čega biste koristili kodne ugovore?

Ilustracije radi, napisaćemo metodu koja dodaje novu stavku na listu samo ako ona ne postoji na listi – s dva *preduslova* (engl. *precondition*) i jednim *konačnim stanjem* (engl. *postcondition*):

```
public static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  Contract.Requires (list != null);          // Preduslov
  Contract.Requires (!list.IsReadOnly);      // Preduslov
  Contract.Ensures (list.Contains (item));   // Konačno stanje
  if (list.Contains(item)) return false;
  list.Add (item);
  return true;
}
```

Preduslovi se definišu pomoću metode `Contract.Requires`, a da li su ispunjeni ispituje se kada se ciljna metoda pokrene. Konačno stanje se definiše pomoću metode `Contract.Ensures` a ispituje se, ne na mestu gde je definisano u kodu, nego *kada se metoda završi*.

Preduslovi i konačna stanja nalik su pretpostavkama i, u ovom primeru, otkrivaju sledeće greške:

- Metoda je pozvana s argumentom koji je `null` ili imamo listu koja se može samo čitati
- U kodu metode smo zaboravili da dodamo stavku na listu



Preduslove i (očekivana) konačna stanja morate navesti na početku metode. To olakšava dobar dizajn: ako u kodu metode koji sledi ne ispunite uslove ugovora, greška će biti otkrivena.

Osim toga, ti uslovi i završna stanja čine *ugovor* za tu metodu koji se može čitati. Metoda `AddIfNotPresent` poručuje svojim korisnicima:

- „Morate me pozvati sa listom koja nije `null` i koja dozvoljava dodavanje nove stavke“.
- „Kada završim svoj posao, lista će sadržati stavku koju ste mi zadali“.

Te činjenice mogu se preneti u XML datoteku dokumentacije sklopa (u Visual Studiju, to možete uraditi ako u prozoru Project Properties izaberete jezičak Code Contracts, dozvolite izradu referentnog sklopa za ugovore i potvrdite opciju „Emit Contracts into XML doc file“). Alatke, kao što je SandCastle, mogu onda da umeću uslove ugovora u dokumentacione datoteke.

Kodni ugovori omogućavaju i da ispravnost programa analiziraju alatke za statičko ispitivanje ispunjavanja ugovora. Na primer, ako metodu `AddIfNotPresent` pozovete s argumentom `list` čija vrednost može biti `null`, takva alatka za statičko ispitivanje ispravnosti mogla bi da vas upozori čak i pre nego što pokrenete program.

Još jedna prednost kodnih ugovora jeste lakoća upotrebe. U našem primeru, lakše je zadati konačno stanje na samom početku metode nego na obe njene izlazne tačke. Kodni ugovori podržavaju i *objektne invarijante* – koje dalje smanjuju potrebu za ponavljanjem koda i obezbeđuju pouzdanije poštovanje pravila.

Preduslovi i konačna stanja mogu se zadati i za članove interfejsa i apstraktne metode, što je nemoguće kod standardnih rešenja za utvrđivanje ispravnosti koda. Osim toga, ne može se dogoditi da nasleđene klase greškom zaobiđu preduslove i konačna stanja koji su definisani u virtuelnim metodama.

Još jedna prednost kodnih ugovora jeste to što se željeno ponašanje u slučaju kršenja ugovora može zadati lakše i na više načina nego kad se za tu namenu koristi klasa metoda `Debug.Assert` ili generišu izuzeci. Osim toga, može se obezbediti da se kršenje ugovora uvek registruje – čak i kad izuzetke zbog kršenja ugovora „progutaju" blokovi za obradu izuzetaka koji su na višem nivou stabla pozivanja.

Nedostatak upotrebe kodnih ugovora jeste to što .NET-ova implementacija koristi *binarni pisač* – alatku koja menja sadržaj sklopa posle prevođenja izvornog koda. To usporava postupak sklapanja aplikacije i komplikuje servise koji pri svom radu pozivaju kompajler C# koda (eksplicitno ili pomoću klase `CSharpCodeProvider`).

Upotreba kodnih ugovora može biti i uzrok nešto slabijih performansi u vreme izvršavanja koda, ali se to lako ispravlja tako što se u varijanti Release aplikacije isključi proveravanje da li se ugovori krše.



Još jedno ograničenje kodnih ugovora jeste to da se ne mogu koristiti za ispitivanje bezbednonih mera, zato što se u vreme izvršavanja aplikacije ugovori mogu zaobići (obradom događaja `ContractFailed`).

Principi kodnih ugovora

Kodni ugovori obuhvataju *preduslove*, *konačna stanja*, *pretpostavke* i *objektne invarijante*. To su sve elementi ugovora koji se nalaze u sklopu i mogu se čitati od spolja. Razlikuju se po trenutku u kojem se ispituje da li su ispunjeni:

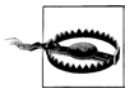
- *Preduslovi* se ispituju kada se funkcija pokrene.
- *Konačna stanja* se proveravaju kada funkcija završi rad.
- *Pretpostavke* se ispituju na mestima gde se pojavljuju u kodu.
- *Objektne invarijante* se ispituju nakon izvršavanja svake javne funkcije u klasi.

Kodni ugovori se definišu isključivo pozivanjem (statičkih) metoda klase `Contract`. To kodne ugovore čini *nezavisnim od jezika*.

Kodni ugovori se mogu definisati ne samo u metodama, nego i u drugim vrstama funkcija, kao što su konstruktori, svojstva, indekseri i operatori.

Prevođenje

Gotovo sve metode klase `Contract` definisane su s atributom `[Conditional("CONTRACTS_FULL")]`. To znači da sve dok ne definišete simbol `CONTRACTS_FULL`, (najveći) deo koda koji se tiče kodnih ugovora nema nikakvog efekta. Visual Studio automatski definiše simbol `CONTRACTS_FULL` ako u odeljku Code Contracts prozora Project Properties uključite proveravanje kodnih ugovora. (Da bi se pojavio jezičak Code Contracts, morate preuzeti i instalirati alatku Contracts sa Microsoftove veb lokacije DevLabs.)



Uklanjanje simbola `CONTRACTS_FULL` može se činiti kao jednostavan način za isključivanje provere ugovora. Međutim, on ne važi za uslove `Requires<TException>` (koje ćemo uskoro detaljno opisati).

Jedini način da u kodu koji koristi `Requires<TException>` isključite proveravanje kodnih ugovora jeste da definišete simbol `CONTRACTS_FULL` i da zatim naložite binarnom pisaču da ukloni kôd koji se tiče kodnih ugovora, tako što izabrete nivo provere „none“ (ništa).

Binarni pisač

Pošto prevedete kôd koji sadrži ugovore, morate pozvati binarni pisač, *ccrewrite.exe* (Visual Studio to čini automatski ako je uključeno proveravanje ugovora). Binarni pisač premešta definicije konačnih stanja (i objektne invarijante) na odgovarajuće mesto, poziva preduslove i objektne invarijante u redefinisanim metodama i zamenjuje pozive članova klase `Contract` pozivima članova *klase za ugovore izvršnog okruženja*. Evo kako bi izgledala (pojednostavljena) verzija našeg ranijeg primera posle prepravljanja pomoću binarnog pisača:

```
static bool AddIfNotPresent<T> (IList<T> list, T item)
{
  __ContractsRuntime.Requires (list != null);
  __ContractsRuntime.Requires (!list.IsReadOnly);
  bool result;
  if (list.Contains (item))
    result = false;
  else
  {
    list.Add (item);
    result = true;
  }
  __ContractsRuntime.Ensures (list.Contains (item));   // Konačno stanje
  return result;
}
```

Ako ne pozovete binarni pisač, klasa `Contract` neće biti zamenjena klasom `__ContractsRuntime` i zato će se generisati izuzeci.



Tip `__ContractsRuntime` je podrazumevana klasa za ugovore u izvršnom okruženju. U složenijim slučajevima, možete zadati vlastitu klasu za ugovore u izvršnom okruženju, pomoću opcije `/rw` ili u Visual Studju, u odeljku Code Contracts prozora Project Properties.

Pošto se klasa `__ContractsRuntime` isporučuje uz binarni pisač (koji nije standardni deo .NET Frameworka), binarni pisač zapravo umeće klasu `__ContractsRuntime` u vaš preveden sklop. Kôd te klase možete videti ako pomoću neke alatke za pregled sklopa (disasembler) ispitate sklop u kojem se koriste kodni ugovori.

Binarni pisač ima opcije za isključivanje nekih ili svih nivoa provera ugovora: to opisujemo u odeljku „Selektivno proveravanje ugovora“, na strani 481. Uobičajeno je da se kompletno proveravanje zadaje kada se aplikacija prevodi u varijantu Debug, a u varijanti Release zadaje se samo određeni podskup provera.

Pretpostavka ili izuzetak u slučaju kršenja ugovora

Binarni pisač omogućava i da birate da li da se u slučaju kršenja ugovora prikaže okvir za dijalog, kao kada pretpostavka nije potvrđena, ili da se generiše izuzetak `ContractException`. Prva mogućnost je uobičajena u varijanti Debug; druga u varijanti Release. Da biste izabrali ovu drugu mogućnost, zadajte opciju `/throwonfailure` kada pozovete binarni pisač, ili uklonite znak potvrde pored polja „Assert on contract failure“ u odeljku Code Contracts prozora Project Properties u Visual Studiju.



14

Istovremenost i asinhroni način rada

Većina aplikacija mora da obrađuje više stvari u isto vreme (*istovremenost*, engl. *concurrency*). U ovom poglavlju, počinjemo od ključnog predznanja, a to su osnove poslova i višenitnog načina rada, a zatim detaljno opisujemo principe asinhronizma i asinhrone funkcije jezika C# 5.0.

U poglavlju 22 detaljno razmatramo rad u višenitnom okruženju, a u poglavlju 23 – srodnu temu paralelnog programiranja.

Uvod

Najuobičajeniji slučajevi kada je potrebna istovremenost jesu sledeći:

Pisanje korisničkog interfejsa s brzim odzivom
: U WPF, Metro i Windows Forms aplikacijama, dugotrajniji poslovi moraju se izvršavati istovremeno s kodom koji održava korisnički interfejs aplikacije da bi se ona mogla brzo odazivati na akcije korisnika.

Obrada više zahteva istovremeno
: Na server može stići više klijentskih zahteva istovremeno i oni se moraju obraditi paralelno da bi se održala skalabilnost sistema. Ako koristite ASP.NET, WCF ili veb servise, .NET Framework vam to obezbeđuje automatski. Međutim, i dalje treba da imate u vidu deljeno stanje (na primer, efekat upotrebe statičkih promenljivih za keširanje podataka.)

Paralelno programiranje
: Kôd koji obavlja mnogo složenih proračuna može se izvršavati brže na računaru s višejezgarnim procesorom ili s više procesora ako se opterećenje celog sistema rasporedi između više jezgara (tome je posvećeno poglavlje 23).

Spekulativno izvršavanje
: Na mašinama s više jezgara možete ponekad poboljšati performanse ako predviđate da će određeni deo možda trebati da se uradi i uradite ga unapred, pre vremena. LINQPad koristi tu tehnologiju da bi ubrzao formiranje novih upita. Varijanta toga je paralelno izvršavanje više algoritama koji rade isti posao. Prvi koji ga završi je „pobednik“ – to je efikasna strategija kada ne možete unapred znati koji će se algoritam najbrže izvršiti.

Opšti mehanizam koji omogućava da program istovremeno izvršava više delova koda zove se *višenitni* rad (engl. *multithreading*). Višenitni način rada programa, koji podržavaju i CLR i operativni sistem, predstavlja suštinski koncept istovremenosti. Zato je ključno da shvatite osnove višenitnog rada, a prvenstveno suštinu niti u *deljenom stanju*.

Višenitni način rada programa

Nit (engl. *thread*) jeste jedna putanja izvršavanja u programu koja se može obraditi nezavisno od drugih putanja.

Svaka nit se izvršava unutar jednog procesa operativnog sistema, koji obezbeđuje izolovano okruženje u kojem radi program. U *jednonitnom* programu, samo se jedna nit izvršava unutar izolovanog okruženja programa i ta nit ima neograničen pristup tom okruženju. U *višenitnom* programu, u jednom procesu se izvršava više niti koje dele isto izvršno okruženje (prvenstveno memoriju). To je, delimično, razlog zbog kojeg je višenitni rad koristan: na primer, jedna nit može učitavati podatke u pozadini, a druga nit prikazuje podatke kako koji pristigne. Takvi podaci se zovu *deljeno stanje* (engl. *shared state*).

Pokretanje nove niti



Windowsov Metro profil ne omogućava direktno formiranje i pokretanje novih niti; to morate raditi pokretanjem odgovarajućih poslova (videti odeljak „Poslovi“, na strani 511). Pošto poslovi (engl. *tasks*) dodaju sloj preusmeravanja koji komplikuje učenje, najbolje je da počnete s konzolskim aplikacijama (ili LINQPad) i da nove niti pokrećete direktno dok ne shvatite kako one rade.

Operativni sistem automatski pokreće *klijentski* program (konzolski, WPF, Metro ili Windows Forms) u jednoj niti („glavna“ nit). U toj niti program proživi svoj životni ciklus kao jednonitna aplikacija, osim vi promenite to stanje, tako što pokrenete (neposredno ili posredno) jednu ili više novih niti.[1]

Novu nit možete formirati i pokrenuti tako što instancirate objekat klase `Thread` i pozovete njegovu metodu `Start`. Najjednostavniji konstruktor klase `Thread` preuzima delegat tipa `ThreadStart`: predstavlja metodu bez parametara koja pokazuje gde bi trebalo da počne izvršavanje. Na primer:

```
// NB: U svim primerima u ovom kodu pretpostavlja se da se uvozi sledeći imenski
// prostor:
using System;
using System.Threading;

class ThreadTest
{
  static void Main()
  {
    Thread t = new Thread (WriteY);          // Pokreće novu nit u kojoj
    t.Start();                               // se izvršava metoda WriteY()

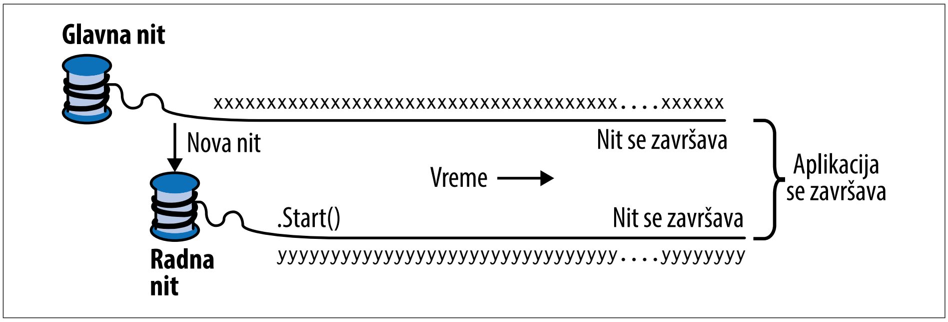
    // U isto vreme, nešto radimo i u glavnoj niti.
    for (int i = 0; i < 1000; i++) Console.Write ("x");
  }

  static void WriteY()
  {
    for (int i = 0; i < 1000; i++) Console.Write ("y");
  }
}
```

1. CLR pokreće u pozadini nove niti koje sakupljaju smeće i finalizuju objekte.

```
// Primer tipičnog rezultata:
xxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
xxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyy
yyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxx
xxxxxxxxxxxxxxxxxxxxxxyyyyyyyyyyyyyyyyyyyyyyyyyyyyyyy
yyyyyyyyyyyyyxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxxx
...
```

Glavna nit pokreće novu nit `t` u kojoj izvršava metodu koja u petlji ispisuje znak *y*. U isto vreme, glavna nit ispisuje u petlji znak *x*, što je prikazano na slici 14-1. Na računaru s jednojezgarnim procesorom, operativni sistem mora da svakoj niti dodeljuje „isečke" procesorskog vremena (pod Windowsom, tipične dužine 20) kako bi simulirao istovremeni rad obe niti, a rezultat su blokovi znakova *x* i *y*. Na računaru s višejezgarnim procesorom, te dve niti se mogu izvršavati zaista paralelno (koliko im to omogućavaju drugi aktivni procesi na računaru), mada u ovom primeru i dalje dobijate blokove znakova *x* i *y* zbog suptilnosti mehanizma po kome klasa `Console` obrađuje istovremene zahteve.



Slika 14-1. Pokretanje nove niti.

> Kaže se da je nit *predupređena* (engl. *preempted*) na mestima gde je njeno izvršavanje prekinuto i nastavljeno izvršavanjem koda iz druge niti. Taj izraz se često pominje u objašnjenjima zašto je nešto krenulo naopako!

Nakon pokretanja, svojstvo `IsAlive` niti ima vrednost `true`, do tačke kada se završi. Nit se završava kada prestane izvršavanje delegata koji ste prosledili konstruktoru objekta niti. Pošto se završi, nit se više ne može ponovo pokrenuti.

Svaka nit ima svojstvo `Name` koje možete zadati radi otkrivanja grešaka. To je naročito korisno u Visual Studiju jer se to ime niti prikazuje u prozoru Threads i na paleti Debug Location. Ime niti možete zadati samo jedanput; ako pokušate da ga naknadno izmenite, prouzrokovaćete izuzetak.

Statičko svojstvo `Thread.CurrentThread` daje nit koja se trenutno izvršava:

```
Console.WriteLine (Thread.CurrentThread.Name);
```

Metode Join i Sleep

Pozivanjem metode `Join` tekuće niti možete sačekati da se druga nit završi:

```
static void Main()
{
  Thread t = new Thread (Go);
```

```
  t.Start();
  t.Join();
  Console.WriteLine ("Thread t has ended!");
}

static void Go() { for (int i = 0; i < 1000; i++) Console.Write ("y"); }
```

Ovaj odlomak koda ispisuje znak „y“ 1000 puta, a odmah iza toga i rečenicu „Thread t has ended!“. Kada pozivate metodu `Join`, možete zadati interval čekanja na završetak druge niti, izražen u milisekundama ili kao objekat tipa `TimeSpan`. Metoda onda vraća `true` ako se ciljna nit završila, odnosno `false` ako je istekao interval čekanja.

Metoda `Thread.Sleep` pauzira („uspavljuje“) tekuću nit tokom zadatog perioda:

```
Thread.Sleep (TimeSpan.FromHours (1));  // Pauzira 1 sat
Thread.Sleep (500);                     // Pauzira 500 milisekundi
```

Metoda `Thread.Sleep(0)` odmah oslobađa tekući isečak svog procesorskog vremena, koji dobrovoljno vraća procesoru radi dodele drugim nitima. Metoda `Thread.Yield()` radi isto – s tom razlikom što svoj deo vremena procesora prepušta samo nitima koje rade na *istom* procesoru.



Metoda `Sleep(0)` ili `Yield` ponekad je korisna u produkcijskoj verziji koda za napredna podešavanja performansi. Ona je i odlična dijagnostička alatka koja olakšava otkrivanje problema u vezi s bezbednim višenitnim radom: ako umetanje poziva `Thread.Yield()` bilo gde u kôd programa ruši taj program, gotovo sigurno imate grešku u njemu.

Dok je nit u stanju čekanja zato što je pozvana metoda `Sleep` ili `Join`, nit je *blokirana*.

Blokiranje niti

Kaže se da je nit *blokirana* kada je njeno izvršavanje pauzirano zbog određenog razloga, kao što je pozivanje metode `Sleep` ili nit čeka da se druga nit završi jer je pozvana metoda `Join`. Blokirana nit odmah *prepušta* svoj komadić procesorskog vremena i od tog trenutka nadalje ne troši vreme procesora dok ne bude ispunjen uslov zbog kojeg je bila blokirana. Da li je nit blokirana, možete saznati iz njenog svojstva `ThreadState`:

```
bool blocked = (someThread.ThreadState & ThreadState.WaitSleepJoin) != 0;
```



Tip svojstva `ThreadState` je nabrajanje čije su vrednosti bitovi koji omogućavaju kombinovanje tri „sloja“ podataka. Međutim, većina tih vrednosti je suvišna, ne koristi se ili je zastarela. Sledeća proširena metoda svodi nabrajanje `ThreadState` na jednu od četiri zaista korisne vrednosti: `Unstarted`, `Running`, `WaitSleepJoin` i `Stopped`:

```
public static ThreadState Simplify (this ThreadState ts)
{
  return ts & (ThreadState.Unstarted |
               ThreadState.WaitSleepJoin |
               ThreadState.Stopped);
}
```

Svojstvo `ThreadState` je korisno za dijagnostičke svrhe, ali nije pogodno za sinhronizovanje niti zato što se stanje niti može promeniti između ispitivanja vrednosti svojstva `ThreadState` i preduzimanja mera na osnovu te informacije.

Kada se nit blokira ili odblokira, operativni sistem obavlja *promenu konteksta* (engl. *context switch*), za šta je potrebna jedna do dve mikrosekunde.

U/I zavisne operacije i operacije zavisne od proračuna

Ako operacija najveći deo svog vremena provodi *čekajući* da se nešto dogodi, kaže se da je ta operacija *U/I zavisna* (engl. I/O-*bound*) – primer toga preuzimanje veb stranice ili pozivanje metode `Console.ReadLine`. (Tipično za U/I zavisne operacije jeste učitavanje ili upisivanje podataka, ali to nije obavezno: metoda `Thread.Sleep` takođe se smatra U/I zavisnom.) Nasuprot tome, ako operacija tokom najvećeg dela svog vremena drži zauzet procesor jer obavlja proračune, kaže se da je *zavisna od proračuna* (engl. *compute-bound*).

Blokiranje i vrtenje niti

U/I zavisna operacija odvija se na jedan od sledeća dva načina: operacija *sinhrono* čeka u tekućoj niti da se određena spoljna operacija završi (primer toga je pozivanje metoda `Console.ReadLine`, `Thread.Sleep` ili `Thread.Join`) ili se operacija odvija *asinhrono*, a kada se završi, pokreće povratnu metodu (više o tome u nastavku ovog poglavlja).

U/I zavisne operacije koje sinhrono čekaju, provode najveći deo svog vremena blokirajući neku nit.

Niti mogu i da se periodično „vrte“ u petlji:

```
while (DateTime.Now < nextStartTime)
  Thread.Sleep (100);
```

Ako zanemarimo činjenicu da postoje bolji načini da se postigne isto (kao što su tajmeri ili signalizacioni konstrukti), još jedna mogućnost je da se nit vrti beskonačno:

```
while (DateTime.Now < nextStartTime);
```

To je uglavnom nepotrebno traćenje vremena procesora: što se CLR-a i operativnog sistema tiče, nit obavlja važan proračun i zato joj se u skladu s time dodeljuju i odgovarajući resursi. U ovom slučaju, ono što bi trebalo da bude U/I zavisna operacija pretvorili smo u operaciju zavisnu od proračuna.



Postoji nekoliko detalja u vezi s vrtenjem i blokiranjem niti. Prvo, *vrlo kratko* vrtenje niti može biti efikasno kada očekujete da će uskoro biti ispunjen određeni uslov (možda u sledećih nekoliko mikrosekundi) jer se onda vrtenjem niti izbegava čekanje dok operativni sistem menja kontekst. Za takve slučajeve .NET Framework ima specijalne metode i klase – videti odeljak „SpinLock and SpinWait“ na *http://albahari.com/threading/*.

Drugo, nije tačno da blokiranje niti ne košta *baš ništa*. Tokom celog svog životnog veka, svaka nit zauzima približno 1MB memorije, a za CLR i operativni sistem to znači dodatan posao administriranja niti. Iz tog razloga, blokiranje niti može predstavljati veliko opterećenje u slučaju U/I zavisnih programa koji treba da izvršavaju stotine ili hiljade istovremenih U/I operacija. Umesto sinhronog blokiranja niti, takvi programi treba da koriste rešenje zasnovano na povratnim funkcijama i da svoju nit potpuno oslobađaju dok čekaju. To je (delimično) svrha asinhronih modela programiranja koje ćemo razmotriti u nastavku ovog poglavlja.

Lokalno i deljeno stanje

CLR dodeljuje svakoj niti zaseban stek u memoriji, što omogućava da se lokalne promenljive ne mešaju. U narednom primeru, definišemo metodu s lokalnom promenljivom, a zatim tu metodu pozivamo istovremeno i iz glavne niti i iz novopokrenute niti:

```
static void Main()
{
  new Thread (Go).Start();      // Poziva metodu Go() iz nove niti
  Go();                         // Poziva metodu Go() iz glavne niti
}

static void Go()
{
  // Deklaracija i upotreba lokalne promenljive – 'cycles'
  for (int cycles = 0; cycles < 5; cycles++) Console.Write ('?');
}
```

Pošto se na memorijskom steku svake niti pravi zasebna kopija promenljive `cycles`, rezultat je, kao što bismo i očekivali, deset znakova pitanja.

Niti dele iste podatke ako imaju zajedničku referencu na istu instancu objekta:

```
class ThreadTest
{
  bool _done;

  static void Main()
  {
    ThreadTest tt = new ThreadTest();   // Pravi zajedničku instancu
    new Thread (tt.Go).Start();
    tt.Go();
  }

  void Go()   // Imajte u vidu da je ovo metoda instance
  {
     if (!_done) { _done = true; Console.WriteLine ("Done"); }
  }
}
```

Budući da obe niti pozivaju metodu `Go()` iste instance klase `ThreadTest`, one dele polje `_done`. Zbog toga se reč „Done“ ispisuje samo jedanput umesto dvaput.

Lokalne promenljive preuzete unutar lambda izraza ili anonimnog delegata, kompajler pretvara u polja, pa se zato i one mogu deliti:

```
class ThreadTest
{
  static void Main()
  {
    bool done = false;
    ThreadStart action = () =>
    {
      if (!done) { done = true; Console.WriteLine ("Done"); }
    };
    new Thread (action).Start();
    action();
  }
}
```

Statička polja pružaju još jedan način za deljenje istih podataka između više niti:

```
class ThreadTest
{
  static bool _done;    // Statička polja dele sve niti
                        // u istom domenu aplikacije.
  static void Main()
  {
    new Thread (Go).Start();
```

```
    Go();
  }

  static void Go()
  {
    if (!_done) { _done = true; Console.WriteLine ("Done"); }
  }
}
```

Sva tri primera ilustruju još jedan ključan koncept, a to je bezbednost niti (ili, bolje rečeno, nedostatak bezbednosti!). Rezultat metode je zapravo neodređen: moguće je (mada malo verovatno) da reč „Done“ bude ispisana dvaput. Međutim, ako u metodi `Go` zamenimo redosled naredaba, dramatično se povećava verovatnoća da će reč „Done“ biti ispisana dvaput:

```
static void Go()
{
  if (!_done) { Console.WriteLine ("Done"); _done = true; }
}
```

Problem je to što jedna nit može izvršavati naredbu `if` tačno u trenutku kada druga nit izvršava naredbu `WriteLine` – pre nego što dobije priliku da promenljivu `done` podesi na `true`.



Naš primer ilustruje jedan od brojnih načina na koji *deljeno upisivo stanje* može uvesti vrstu povremenih grešaka koje su veoma poznata pojava u višenitnom načinu rada. U odeljku koji sledi videćemo kako možemo ispraviti program pomoću blokada; međutim, najbolje je da od samog početka izbegavate deljena stanja, gde god je to moguće. U nastavku ovog poglavlja videćemo kako se to postiže pomoću asinhronih modela programiranja.

Blokade i bezbednost niti



Blokade i bezbednost niti veoma su opsežne teme. Detaljan opis naći ćete u odeljcima „Ekskluzivne blokade“, na strani 790, i „Blokade i bezbednost koda za višenitni rad“, na stani 797 poglavlja 22.

Prethodni primer možemo ispraviti ako postavimo *ekskluzivnu blokadu* dok čitamo deljeno polje i upisujemo u njega. Za tu namenu C# ima naredbu `lock`:

```
class ThreadSafe
{
  static bool _done;
  static readonly object _locker = new object();

  static void Main()
  {
    new Thread (Go).Start();
    Go();
  }

  static void Go()
  {
    lock (_locker){
      if (!_done) { Console.WriteLine ("Done"); _done = true; }
    }
  }
}
```

Kada dve niti istovremeno pokušavaju da postave blokadu (koja se može odnositi na bilo kakav objekat referentnog tipa, kao u ovom primeru, objekat `_locker`), jedna nit čeka, ili se blokira, dok se blokada ne oslobodi. U ovom slučaju, time obezbeđujemo da u svakom datom trenutku samo jedna nit može izvršavati deo koda zaštićen blokadom, a reč „Done“ biće ispisana samo jedanput. Za programski kôd koji je zaštićen na opisani način – od neodređenosti u višenitnom kontekstu – kaže se da je *bezbedan za višenitni rad* (engl. *thread-safe*).



Čak ni operacija automatskog inkrementiranja promenljive nije bezbedna u višenitnom radu: izraz `x++` procesor izvršava kao tri zasebne operacije čitanja, povećavanja vrednosti i dodeljivanja te nove vrednosti prvobitnoj promenljivoj. Iz tog razloga, ako dve niti istovremeno izvršavaju operaciju `x++` bez blokade, može se dogoditi da se vrednost promenljive inkrementira samo jedanput umesto dvaput (ili, još gore, pod određenim uslovima `x` se može *pocepati*, tj. dobiti neku vrednost koja je mešavina bitova starog i novog sadržaja).

Blokade nisu garancija za bezbedan višenitni rad – lako se zaboravlja da treba postaviti blokadu kada se pristupa polju, a postavljanje blokada može uvesti i svoje probleme (kao što je uzajamno blokiranje).

Dobar primer kada bi trebalo koristiti blokade jeste pristupanje memorijskom kešu (ostavi) za objekte iz baze podataka kojima se često pristupa iz ASP.NET aplikacije. Tu vrstu aplikacija je lako napraviti kako treba i ne postoji način da dođe do uzajamnog blokiranja. Primer toga naći ćete u odeljku „Bezbednost za višenitni rad servera za aplikacije“, na strani 800 poglavlja 22.

Prosleđivanje podataka drugoj niti

Ponekad ćete poželeti da metodi koja pokreće novu nit prosledite određene argumente. To ćete najlakše uraditi pomoću lambda izraza koji poziva metodu s odgovarajućim argumentima:

```
static void Main()
{
  Thread t = new Thread ( () => Print ("Hello from t!") );
  t.Start();
}

static void Print (string message) { Console.WriteLine (message); }
```

U ovom obliku, metodi možete proslediti proizvoljan broj argumenata. Možete čak i celu implementaciju zadati u obliku lambda izraza s više naredaba:

```
new Thread (() =>
{
  Console.WriteLine ("I'm running on another thread!");
  Console.WriteLine ("This is so easy!");
}).Start();
```

Pošto lambda izrazi nisu postojali pre verzije C# 3.0, možda ćete naići na tehniku iz stare škole, a to je prosleđivanje argumenta metodi `Start` klase `Thread`:

```
static void Main()
{
  Thread t = new Thread (Print);
  t.Start ("Hello from t!");
}

static void Print (object messageObj)
{
```

```
  string message = (string) messageObj;   // Ovde nam treba eksplicitno pretvaranje
                                          // tipa
  Console.WriteLine (message);
}
```

Ovo radi zato što konstruktor klase `Thread` ima preklopljene verzije koje prihvataju jedan od sledeća dva delegata:

```
public delegate void ThreadStart();
public delegate void ParameterizedThreadStart (object obj);
```

Ograničenje delegata `ParameterizedThreadStart` jeste to što prihvata samo jedan argument. A pošto je taj argument tipa `object`, najčešće je potrebno još i eksplicitno pretvaranje u odgovarajući ciljni tip.

Lambda izrazi i preuzete promenljive

Kao što smo videli, upotreba lambda izraza je najzgodniji i najmoćniji način da prosledite podatke drugoj niti. Međutim, morate voditi računa da greškom ne izmenite *preuzete promenljive* nakon pokretanja niti. Na primer, razmotrite sledeće:

```
for (int i = 0; i < 10; i++)
  new Thread (() => Console.Write (i)).Start();
```

Rezultat je nedeterministički! Ovako izgleda tipičan rezultat:

```
0223557799
```

Problem je u tome što promenljiva `i` upućuje uvek na *isto* mesto u memoriji tokom celog životnog ciklusa petlje. Iz tog razloga, svaka nit poziva metodu `Console.Write` s promenljivom čija se vrednost može menjati tokom izvršavanja pozivajuće niti! Rešenje je upotreba privremene promenljive, na sledeći način:

```
for (int i = 0; i < 10; i++)
{
  int temp = i;
  new Thread (() => Console.Write (temp)).Start();
}
```

Svaka od cifara od 0 do 9 ispisuje se samo jedanput. (*Redosled* cifara je i dalje nepredvidljiv jer niti mogu započinjati u nepredvidljivim trenucima.)



Ovo je analogno problemu koji smo opisali u odeljku „Preuzete promenljive“, na strani 302 poglavlja 8. Problem se odnosi na pravila jezika C# za preuzimanje promenljivih u petljama `for` u istoj meri kao i na višenitni rad.

Problem se pojavljivao i u petljama `foreach` pre verzije C# 5.

Promenljiva `temp` je sada lokalna za svaku iteraciju petlje. Iz tog razloga, svaka nit preuzima različitu memorijsku lokaciju i zato se problem više ne pojavljuje. Problem u ranijem kodu možemo jednostavnije ilustrovati pomoću sledećeg primera:

```
string text = "t1";
Thread t1 = new Thread ( () => Console.WriteLine (text) );

text = "t2";
Thread t2 = new Thread ( () => Console.WriteLine (text) );

t1.Start(); t2.Start();
```

Pošto oba lambda izraza preuzimaju istu tekstualnu promenljivu, `t2` se ispisuje dvaput.

Obrada izuzetaka

Nijedan blok `try/catch/finally` ne važi u novoj niti i ništa ne znači u njoj kada počne izvršavanje niti. Razmotrite sledeći program:

```
public static void Main()
{
  try
  {
    new Thread (Go).Start();
  }
  catch (Exception ex)
  {
    // Ovde nećemo nikad stići!
    Console.WriteLine ("Izuzetak!");
  }
}

static void Go() { throw null; }   // Generiše izuzetak NullReferenceException
```

Naredba `try/catch` u ovom primeru ne služi ničemu, a novopokrenuta nit je opterećena neobrađenim izuzetkom `NullReferenceException`. To ponašanje postaje razumljivo ako uzmete u obzir da svaka nit ima vlastitu nezavisnu putanju izvršavanja.

Rešenje je da blok za obradu grešaka premestimo u metodu `Go`:

```
public static void Main()
{
   new Thread (Go).Start();
}

static void Go()
{
  try
  {
    ...
    throw null;    // izuzetak NullReferenceException biće presretnut niže
    ...
  }
  catch (Exception ex)
  {
    Ovde možemo ubeležiti izuzetak u dnevnik i/ili signalizirati drugoj niti
    da smo se dočekali na noge
    ...
  }
}
```

U verzijama aplikacija koje su namenjene tržištu, potreban vam je blok za obradu izuzetaka u svakoj početnoj metodi niti – isto kao što ga imate (obično na višem nivou, na stablu izvršavanja metoda) u vašoj glavnoj niti. Neobrađen izuzetak ruši celu aplikaciju. Uz prikazivanje ružnog okvira za dijalog pride!



Kada pišete takve blokove za obradu izuzetaka, grešku ćete retko *zanemariti*: obično ćete podatke o grešci zabeležiti u neki dnevnik, a zatim možda prikazati korisniku okvir za dijalog koji mu omogućava da te podatke automatski prosledi na vaš veb server. Nakon toga možete izabrati da aplikaciju ponovo pokrenete, zato što je moguće da neobrađen izuzetak ostavi program u neispravnom stanju.



23

Paralelno programiranje

U ovom poglavlju razmatramo API-je i konstrukte za višenitno programiranje pomoću kojih možete iskoristiti procesor s više jezgara:

- Paralelni LINQ ili *PLINQ*
- Klasa `Parallel`
- Konstrukti za paralelno izvršavanje poslova
- Kolekcije koje podržavaju istovremenost

Navedene mogućnosti su uvedene s Frameworkom 4.0 a poznate su pod zajedničkim (veoma opštim) imenom PFX (Parallel Framework). Klasa `Parallel`, zajedno s konstruktima za paralelno izvršavanje poslova, zove se *Task Parallel Library* (biblioteka za paralelno izvršavanje poslova) ili TPL.

Pre čitanja ovog poglavlja potrebno je da dobro shvatite osnove opisane u poglavlju 14 – pre svega blokade, bezbedan višenitni rad i klasu `Task`.

Čemu PFX?

Relativno nedavno, proizvođači procesora su prešli sa jednog jezgra, na višejezgarne procesore. To je nama programerima prouzrokovalo problem jer naš standardni kôd napisan za rad u jednoprocesorskom okruženju neće automatski raditi brže samo zbog postojanja tih dodatnih jezgara.

Prednosti postojanja više jezgara mogu se na jednostavan način iskoristiti u većini serverskih aplikacija, gde svaka nit može nezavisno od drugih niti preuzeti obradu klijentskog zahteva, ali to je teže u slučaju klijentskih aplikacija – zato što onda svoj kôd koji teže opterećuje procesor morate najčešće prepraviti na sledeći način:

1. *Raspodelite* ga u manje celine.
2. Te celine izvršavajte paralelno primenom višenitnih tehnika.
3. Rezultate delimičnih obrada (međurezultate) *spajajte* čim postanu raspoloživi, na način koji je bezbedan u višenitnom okruženju i koji omogućava dobre performanse.

Mada sve navedeno možete postići i pomoću klasičnih konstrukata za višenitni rad, postupak je nezgrapan – naročito oni njegovi delovi koji se tiču raspodele obrade i spajanja međurezultata. Dodatan problem je to što uobičajena strategija postavljanja blokada radi bezbednog višenitnog rada prouzrokuje mnogo sukoba kada veći broj niti radi sa istim podacima u isto vreme.

PFX biblioteke su projektovane kako bi olakšale rešavanje problema u takvim slučajevima.



Programiranje s ciljem da se iskoristre prednosti postojanja više procesorskih jezgara ili više procesora, zove se *paralelno programiranje*. To je podskup šire teme višenitnog programiranja.

PFX koncepti

Postoje dve strategije za raspoređivanje opterećanja između više niti: *paralelizam podataka* (engl. *data parallelism*) i *paralelizam poslova* (engl. *task parallelism*).

Kada određen skup poslova treba izvršiti s velikim brojem vrednosti podataka, to možemo paralelizovati tako što svaka nit obavlja (isti) skup poslova s drugim podskupom izvornih vrednosti. To se zove *paralelizam podataka* zato što između niti raspodeljujemo *podatke*. Nasuprot tome, pri *paralelizmu poslova* raspodeljujemo *poslove*; drugim rečima, svaka nit obavlja različit posao.

Paralelizam podataka je uglavnom jednostavniji i bolje se skalira na hardveru s visokim stepenom paralelizma, zato što umanjuje ili potpuno uklanja dupliranje podataka (što smanjuje broj sukoba i probleme bezbednog izvršavanja niti). Osim toga, paralelizam podataka iskorišćava činjenicu da je broj vrednosti podataka znatno veći od broja različitih poslova, što povećava potencijal za paralelizam.

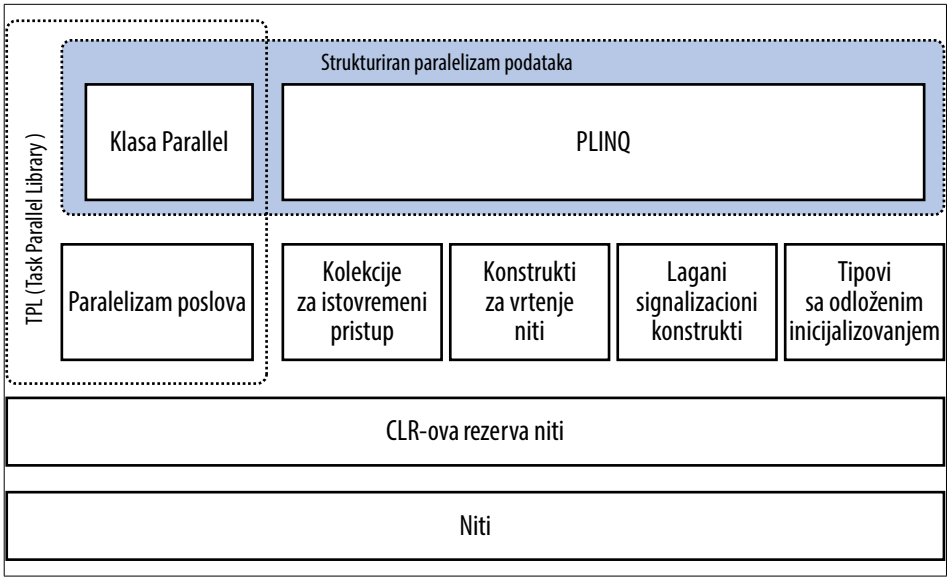
Paralelizam podataka dovodi i do *strukturiranog paralelizma*, što znači da jedinice paralelne obrade počinju i završavaju se na istim mestima u programu. Nasuprot tome, paralelizam poslova češće je nestrukturiran, što znači da jedinice paralelne obrade mogu počinjati i završavati se na nasumičnim mestima širom programa. Strukturiran paralelizam je jednostavniji, manje podložan greškama i omogućava da složenije poslove raspodele i koordinisanja niti (pa čak i spajanja međurezultata) grupišete i prepustite specijalizovanim bibliotekama.

PFX komponente

PFX se sastoji od dva sloja funkcionalnosti, kao što se vidi na slici 23-1. Gornji sloj čine dva API-ja za *strukturiran paralelizam podataka*: PLINQ i klasa `Parallel`. Donji sloj čine klase za paralelizam poslova – zajedno sa skupom dodatnih konstrukata koji olakšavaju aktivnosti u vezi s paralelnim programiranjem.

PLINQ pruža najbogatiju funkcionalnost: automatizuje sve korake postupka paralelizovanja obrade – uključujući i raspodelu obrade na poslove, izvršavanje tih poslova u nitima i spajanje međurezultata u jednu izlaznu sekvencu. To se zove *deklarativno* paralelizovanje – zato što samo deklarišete da želite paralelizovanje obrade (što strukturirate u obliku LINQ upita), a Frameworku prepuštate detalje izvedbe. Nasuprot tome, drugi pristupi su *imperativni* jer morate sami napisati dodatan kôd za raspodelu poslova ili spajanje međurezultata. Kada koristite klasu `Parallel`, morate sami spajati međurezultate; kada koristite konstrukte za paralelizam poslova, te poslove morate sami i raspodeliti:

	Raspodeljuje obradu	Spaja međurezultate
PLINQ	Da	Da
Klasa `Parallel`	Da	Ne
PFX-ov paralelizam poslova	Ne	Ne



Slika 23-1. PFX komponente.

Kolekcije za istovremeni pristup i konstrukti za vrtenje niti olakšavaju programiranje aktivnosti nižeg nivoa za paralelan rad. To su važni elementi zato što je PFX projektovan tako da radi ne samo na današnjem hardveru, nego i na budućim generacijama procesora koje će imati znatno veći broj jezgara. Ako želite da premestite gomilu iscepanih drva i to treba da obavi 32 radnika, najteži deo posla je da pri premeštanju drva radnici ne ometaju jedan drugog. Isti je slučaj i s raspodelom izvršavanja algoritma između 32 jezgra: ako pristup zajedničkim resursima štitite pomoću običnih blokada, rezultat može biti da samo deo svih tih jezgara bude zaista uposlen u isto vreme. Kolekcije za istovremeni pristup optimizovane su tako da omogućavaju vrlo veliki broj istovremenih pristupa, pri čemu nastoje da minimizuju ili eliminišu blokiranje niti. PLINQ i klasa `Parallel` koriste mogućnosti kolekcija za istovremeni pristup i konstrukata za vrtenje niti da bi efikasno upravljali obradom.

Druge upotrebe PFX-a

Konstrukti za paralelno programiranje korisni su ne samo kada imate više jezgara, nego i u drugim slučajevima:

- Kolekcije za istovremeni pristup ponekad su pogodne kad želite red čekanja, stek ili rečnik koji je bezbedan za višenitni rad.
- Klasa `BlockingCollection` pruža jednostavan način za implementiranje struktura tipa proizvođač/potrošač i dobar je način za *ograničavanje* broja istovremenih pristupa.
- Poslovi (objekti tipa `Task`) predstavljaju osnovu asinhronog programiranja, kao što smo videli u poglavlju 14.

Kada se koristi PFX

Primarni slučaj za upotrebu PFX-a jeste *paralelno programiranje*: iskorišćavanje mogućnosti višejezgarnih procesora radi bržeg izvršavanja koda koji teško opterećuje sistem.

Pri upošljavanju više jezgara, problem je Amdalov (Amdahl) zakon, prema kojem maksimalno poboljšanje performansi primenom paralelizovanja zavisi od dela koda koji mora da se izvršava sekvencijalno. Na primer, ako se može paralelizovati samo dve trećine vremena izvršavanja određenog algoritma, poboljšanje performansi ne može nikad biti veće od tri puta – čak i s beskonačnim brojem jezgara.

Stoga, pre nego što nastavite, vredi ispitati da li je usko grlo baš deo koda koji se može paralelizovati. Vredi proveriti i da li vaš kôd baš *mora* da obavlja toliki broj proračuna – optimizovanje koda je često najlakše i najefikasnije rešenje. Međutim, „cena“ nekih tehnika optimizovanja koda može biti teže paralelizovanje koda.

Dobici se najlakše postižu u slučaju onog što se ponekad naziva *ponižavajuće paralelni* (engl. *embarrassingly parallel*) problemi – gde se obrada lako može podeliti na poslove koji se samostalno efikasno izvršavaju (strukturiran paralelizam je veoma pogodan za tu vrstu problema). Primeri toga su poslovi obrade slika, iscrtavanja putanja svetlosnih zraka i rešavanja matematičkih ili kriptografskih problema primenom metode grube sile. Primer neponižavajuće paralelnog problema jeste implementiranje optimizovane verzije algoritma quicksort – za dobar rezultat potrebno je razmišljanje, a može zahtevati i nestrukturiran paralelizam.

PLINQ

PLINQ automatski paralelizuje lokalne LINQ upite. PLINQ pruža prednost jednostavne upotrebe jer posao raspodele obrade i spajanja rezultata prebacuje na Framework.

Da biste iskoristili PLINQ, samo dodajte izlaznoj sekvenci `AsParallel()` a zatim nastavite LINQ upit na uobičajen način. Sledeći upit izračunava proste brojeve u opsegu između 3 i 100.000 – pri čemu upošljava sva jezgra na ciljnoj mašini:

```
// Izračunava proste brojeve pomoću jednostavnog (neoptimizovanog) algoritma.

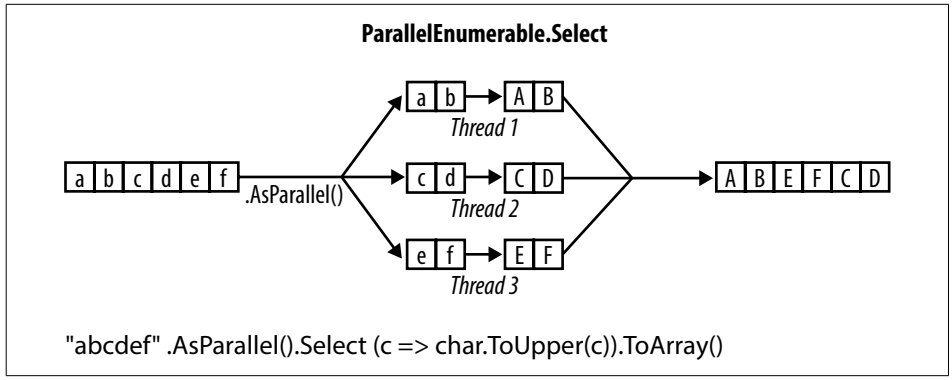
IEnumerable<int> numbers = Enumerable.Range (3, 100000-3);

var parallelQuery =
  from n in numbers.AsParallel()
  where Enumerable.Range (2, (int) Math.Sqrt (n)).All (i => n % i > 0)
  select n;


int[] primes = parallelQuery.ToArray();
```

`AsParallel` je proširena metoda klase `System.Linq.ParallelEnumerable`. Ta metoda „umeće“ izvornu ulaznu sekvencu u sekvencu tipa `ParallelQuery<TSource>`, koja čini da se LINQ operatori koje zatim pozivate preslikavaju u alternativni skup proširenih metoda definisanih u klasi `ParallelEnumerable`. Te metode su paralelne implementacije svih standardnih operatora za upite, a rade tako što u suštini dele ulaznu sekvencu na delove koji se obrađuju u različitim nitima, a rezultate spajaju u jednu rezultujuću sekvencu za dalju obradu (slika 23-2).

Pozivanjem metode `AsSequential()`, sekvenca tipa `ParallelQuery` se „raspakiva“ u običnu sekvencu da bi se operatori za upite koje zatim pozivate razrešili u standardne operatore za upite i izvršili sekvencijalno. To je potrebno pre pozivanja metoda koje imaju sporedne efekte ili koje nisu bezbedne za višenitni rad.



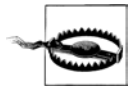
Slika 23-2. PLINQ-ov model izvršavanja.

U slučaju operatora za upite koji prihvataju dve ulazne sekvence (`Join`, `GroupJoin`, `Concat`, `Union`, `Intersect`, `Except` i `Zip`), metodu `AsParallel()` morate primeniti na obe ulazne sekvence (inače ćete izazvati izuzetak). Međutim, nema potrebe da metodu `AsParallel` neprekidno pozivate u upitu dok napreduje njegovo izvršavanje, zato što PLINQ-ovi operatori za upite proizvode druge izlazne sekvence tipa `ParallelQuery`. U stvari, višekratno pozivanje metode `AsParallel` ponovo uvodi neefikasnost jer izaziva spajanje i ponovno deljenje upita:

```
mySequence.AsParallel()              // Pretvara sekvencu u drugu
                                     // tipa ParallelQuery<int>
          .Where (n => n > 100)      // Vraća novu sekvencu ParallelQuery<int>
          .AsParallel()              // Nepotrebno – i neefikasno!
          .Select (n => n * n)
```


Ne mogu se svi operatori za upite efikasno paralelizovati. U slučaju onih za koje to nije moguće (videti odeljak „Ograničenja PLINQ-a“, na strani 831), PLINQ implementira operator sekvencijalno. PLINQ može raditi sekvencijalno i ako proceni da bi režijski deo postupka paralelizovanja zapravo usporio dati upit.

PLINQ je upotrebljiv samo za lokalne upite i ne radi za LINQ to SQL ili Entity Framework upite zato što se u tim slučajevima LINQ upit prevodi u SQL upit, koji se zatim izvršava na serveru baze podataka. Međutim, PLINQ *možete* iskoristiti za dodatno pretraživanje lokalnog skupa rezultata koji ste dobili pomoću upita u baze podataka.



Ako u PLINQ upitu dođe do izuzetka, on se ponovo generiše u obliku izuzetka tipa `AggregateException` čije svojstvo `InnerExceptions` sadrži stvarni izuzetak (ili izuzetke). Više informacija o tome naći ćete u odeljku „Izuzetak AggregateException“ na strani 854.

Zašto se metoda AsParallel ne poziva automatski?

Pošto se pozivanjem metode `AsParallel` LINQ upiti automatski paralelizuju, nameće se pitanje „Zašto Microsoft ne paralelizuje automatski standardne operatore za upite da bi PLINQ bio podrazumevani način obrade?“

To *opciono* rešenje je odabrano iz više razloga. Prvo, da bi PLINQ bio od koristi, treba da postoji razumna količina složene obrade koja bi se raspodelila na radne niti. Budući da se većina upita tipa LINQ to Objects izvršava vrlo brzo, paralelizovanje bi bilo ne samo nepotrebno, nego bi i režijski deo posla raspodele obrade, spajanja međurezultata i koordinisanja dodatnih niti zapravo usporio celu stvar.

Osim toga:

- Rezultat PLINQ upita (standardno) može biti različit od rezultata LINQ upita po pitanju redosleda elemenata (videti odeljak „PLINQ i redosled elemenata izlazne sekvence“, na ovoj strani).
- PLINQ pakuje sve izuzetke u izuzetak tipa `AggregateException` (da bi podržao mogućnost generisanja izuzetaka više vrsta).
- PLINQ vraća nepouzdane rezultate ako upit poziva metode koji nisu bezbedne za višenitni rad.

I najzad, PLINQ pruža priličan broj mogućnosti za fino podešavanje. Opterećivanje standardnog API-ja LINQ to Objects detaljima te vrste samo bi odvlačilo pažnju.

Postupak paralelnog izvršavanja upita

Isto kao i obični LINQ upiti, PLINQ upiti se izvršavaju odloženo. To znači da se izvršavanje pokreće samo kada vam zatreba rezultat upita – najčešće unutar petlje `foreach` (mada to može biti i u slučaju operatora za konverziju kao što je `ToArray` ili drugog operatora koji vraća pojedinačni element ili vrednost).

Međutim, dok nabrajate rezultate, PLINQ upiti se izvršavaju donekle drugačije od običnog sekvencijalnog upita. Sekvencijalni upit korisnik u potpunosti izvršava na „potezni“ način: svaki element ulazne sekvence povlači se i obrađuje tek kada to korisnik izričito zatraži. Paralelni upit obično koristi nezavisne niti za povlačenje elemenata iz ulazne sekvence neznatno *pre* nego što oni zatrebaju korisniku (slično aplikacijama za prikazivanje vesti koje prikazuju naslov sledeće vesti, ili slično baferu za sprečavanje preskakanja sadržaja u CD plejerima). Upit zatim paralelno obrađuje elemente u lancu, pri čemu rezultate čuva u malom baferu da bi bili spremni kada ih korisnik zahteva. Ako korisnik pauzira ili ranije prekine nabrajanje rezultata upita, obrada upita se takođe zaustavlja ili otkazuje da se ne bi bez potrebe tratilo vreme procesora ili memorija.



Ponašanje PLINQ-a možete finije podesiti ako iza metode `AsParallel` pozovete metodu `WithMergeOptions`. Podrazumevana vrednost parametra te metode, opcija `AutoBuffered`, uglavnom daje najbolje ukupne rezultate. Opcija `NotBuffered` isključuje bafer i korisna je ako želite da vidite rezultat čim to bude moguće; opcija `FullyBuffered` smešta ceo skup rezultata u keš pre nego što ga prosledi korisniku (operatori `OrderBy` i `Reverse` prirodno rade na taj način, što važi i za operatore nad pojedinačnim elementima i operatore za agregiranje i konverziju elemenata ulazne sekvence)

PLINQ i redosled elemenata izlazne sekvence

Sporedni efekat paralelizovanja operatora upita jeste sledeći: kada se spajaju međurezultati, redosled nije obavezno isti kao pri povlačenju ulaznih elemenata (slika 23-2). Drugim rečima, više ne važi LINQ-ova standardna garancija očuvanja redosleda elemenata u sekvencama.

Ako redosled treba da ostane sačuvan, iza metode `AsParallel()` pozovite metodu `AsOrdered()`:

```
myCollection.AsParallel().AsOrdered()...
```

Pozivanje metode `AsOrdered` dovodi do slabljenja performasi u slučaju sekvenci sa velikim brojem elemenata zato što PLINQ mora da pamti izvorni položaj svakog ulaznog elementa.

Efekat pozivanja metode `AsOrdered` možete poništiti kasnije u upitu ako pozovete metodu `AsUnordered`: to uvodi „tačku nasumičnog mešanja“ što omogućava da se upit efikasnije izvršava od tog mesta nadalje. Na primer, ako želite da sačuvate isti redosled rezultata kao elemenata ulazne sekvence, ali da to važi samo za prva dva operatora upita, uradite sledeće:

```
inputSequence.AsParallel().AsOrdered().QueryOperator1()
  .QueryOperator2()
  .AsUnordered()       // Odavde nadalje, redosled je nebitan
  .QueryOperator3()
  ...
```

`AsOrdered` nije podrazumevana opcija zato što je u većini upita izvorni redosled elemenata nevažan. Drugim rečima, kada bi `AsOrdered` bila podrazumevana opcija, u većini svojih paralelnih upita morali biste eksplicitno zadavati opciju `AsUnordered` da biste postigli najbolje performanse, što bi bilo nezgrapno.

Ograničenja PLINQ-a

Zasad postoji nekoliko praktičnih ograničenja toga šta sve PLINQ može da paralelizuje. Ta ograničenja će možda biti ublažena u budućim servisnim paketima i verzijama Frameworka.

Sledeći operatori upita onemogućavaju paralelizovanje upita, osim kada su elementi izvorne sekvence na svojim izvornim indeksnim položajima:

- `Take`, `TakeWhile`, `Skip` i `SkipWhile`
- Indeksirane verzije `Select`, `SelectMany` i `ElementAt`

Većina operatora upita menja indeksne položaje elemenata (što važi i za operatore koji uklanjaju elemente iz ulazne sekvence, kao što je operator `Where`). To znači sledeće: ako želite da koristite prethodne operatore, morate ih postaviti najčešće na početak upita.

Sledeći operatori upita mogu se paralelizovati, ali primenjuju „skupu“ strategiju raspodele obrade koja ponekad može biti sporija od sekvencijalnog načina obrade:

- `Join`, `GroupBy`, `GroupJoin`, `Distinct`, `Union`, `Intersect` i `Except`

Preklopljene verzije operatora za agregiranje elemenata sa *početnim zrnom*, u svojim standardnim varijantama ne mogu se paralelizovati – PLINQ ima specijalne verzije svojih operatora za te slučajeve (videti odeljak „Optimizovanje PLINQ-a“, na strani 835).

Svi ostali operatori se mogu paralelizovati, mada upotreba paralelizovanih operatora ne garantuje da će upit biti paralelizovan. PLINQ može izvršiti upit na sekvencijalan način ako proceni da bi režijski deo postupka paralelizovanja usporio dati upit. To ponašanje možete isključiti i nametnuti paralelizam ako iza metode `AsParallel()` zadate sledeće:

```
.WithExecutionMode (ParallelExecutionMode.ForceParallelism)
```

Primer: paralelni program za proveru pravopisne ispravnosti

Pretpostavimo da nam treba program za proveru engleskog pravopisa koji veoma brzo obrađuje obimne dokumente tako što upošljava sva procesorska jezgra s kojima raspolaže. Ako algoritam formulišemo u obliku LINQ upita, možemo ga vrlo lako paralelizovati.

Prvi korak je priprema rečnika engleskih reči u obliku objekta tipa `HashSet` radi efikasnog pretraživanja:

```
if (!File.Exists ("WordLookup.txt"))    // Sadrži približno 150.000 reči
  new WebClient().DownloadFile (
    "http://www.albahari.com/ispell/allwords.txt", "WordLookup.txt");

var wordLookup = new HashSet<string> (
  File.ReadAllLines ("WordLookup.txt"),
  StringComparer.InvariantCultureIgnoreCase);
```

Zatim pomoću našeg rečnika (objekta `wordLookup`) pravimo probni „dokument“ čiji je sadržaj niz od milion nasumično izabranih reči. Pošto formiramo taj niz, uvodimo nekoliko pravopisnih grešaka:

```
var random = new Random();
string[] wordList = wordLookup.ToArray();

string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();

wordsToTest [12345] = "woozsh";     // Uvodimo nekoliko
wordsToTest [23456] = "wubsie";     // pravopisnih grešaka.
```

Sada možemo paralelno ispitivati tačnost pravopisa tako što poredimo sadržaj niza `wordsToTest` sa sadržajem rečnika `wordLookup`. PLINQ čini taj postupak veoma jednostavnim:

```
var query = wordsToTest
  .AsParallel()
  .Select  ((word, index) => new IndexedWord { Word=word, Index=index })
  .Where   (iword => !wordLookup.Contains (iword.Word))
  .OrderBy (iword => iword.Index);

foreach (var mistake in query)
  Console.WriteLine (mistake.Word + " - index = " + mistake.Index);

// REZULTAT:
// woozsh - index = 12345
// wubsie - index = 23456
```

`IndexedWord` je namenska struktura definisana kao što sledi:

```
struct IndexedWord { public string Word; public int Index; }
```

Metoda `wordLookup.Contains` u predikatu dodaje upitu malo „mesa“ zbog kojeg ga vredi paralelizovati.



Upit bi bio nešto jednostavniji da smo upotrebili anonimni tip umesto strukture `IndexedWord`. Međutim, to bi pogoršalo performanse zato što je za anonimne tipove (pošto su to klase, to su referentni tipovi) potreban dodatan režijski rad dodeljivanja memorije na hipu a kasnije i sakupljanje smeća (kad ta memorija više ne bude potrebna).

Razlika možda nije velika u slučaju sekvencijalnih upita, ali kod paralelnih upita, rešenje s dodeljivanjem memorije na steku može može pružiti prilično veliku prednost. Razlog je to što se postupak dodeljivanja memorije na steku može u velikoj meri paralelizovati (zato što svaka nit ima vlastiti stek), dok se nasuprot tome sve niti takmiče za isti hip – kojim upravljaju isti upravljač memorijom i isti sakupljač smeća.

Upotreba klase ThreadLocal<T>

Naš primer možemo proširiti paralelizovanjem same liste nasumično izabranih probnih reči. Pošto smo njeno sastavljanje strukturirali u obliku LINQ upita, to bi trebalo da bude lako. Ovako izgleda sekvencijalna verzija upita:

```
string[] wordsToTest = Enumerable.Range (0, 1000000)
  .Select (i => wordList [random.Next (0, wordList.Length)])
  .ToArray();
```

Nažalost, pozivanje `random.Next` nije bezbedno za višenitni rad, pa zato postupak nije tako jednostavan kao umetanje operatora `AsParallel()` u upit. Moguće rešenje je da napišemo funkciju koja postavlja blokadu oko `random.Next`; međutim, to ograničava mogućnost istovremenog pristupa iz više niti. Bolje rešenje je upotreba objekta tipa `ThreadLocal<Random>` (videti odeljak „Skladištenje podataka lokalno u nitima“, na strani 817, u prethodnom poglavlju) da bismo u svakoj niti napravili zaseban objekat `Random`. Upit možemo zatim paralelizovati na sledeći način:

```
var localRandom = new ThreadLocal<Random>
 ( () => new Random (Guid.NewGuid().GetHashCode()) );

string[] wordsToTest = Enumerable.Range (0, 1000000).AsParallel()
  .Select (i => wordList [localRandom.Value.Next (0, wordList.Length)])
  .ToArray();
```

Našoj fabričkoj funkciji koja instancira objekat `Random`, prosleđujemo heš vrednosti `Guid` kako bismo obezbedili da ako se unutar vrlo kratkog razdoblja generišu dva `Random` objekta, oni proizvode različite sekvence nasumičnih brojeva.

Kada se može koristiti PLINQ

Možete pasti u iskušenje da u svojim postojećim aplikacijama prepravite LINQ upite i eksperimentišete s njihovim paralelizovanjem. To je najčešće pogrešno, zato što se većina problema za koje je LINQ očigledno najbolje rešenje, obično izvršava vrlo brzo i zato paralelizovanje ne bi ništa poboljšalo. Bolje rešenje je da pronađete uska grla koja teško opterećuju procesor i da u tim slučajevima razmislite „Može li se ovo izraziti u obliku LINQ upita?“ (Dobrodošao sporedni efekat takvog restrukturiranja jeste to što LINQ obično čini programski kôd sažetijim i razumljivijim.)

PLINQ je pogodan za ponižavajuće paralelne probleme. Međutim, to može biti loš izbor za obradu slika, zato što spajanje međurezultata obrade piksela u jednu izlaznu sekvencu formira usko grlo. Umesto toga, bolje je da piksele upisujete direktno u niz ili u blok neupravljane memorije i da višenitnom obradom upravljate pomoću klase `Parallel` ili pomoću paralelizma poslova. (Međutim, moguće je isključivanje spajanja delimičnih rezultata pomoću operatora `ForAll` – što razmatramo u odeljku „Optimizovanje PLINQ-a“, na strani 835. To je logičan postupak ako se algoritam za obradu slika prirodno može izraziti u obliku LINQ upita.)

Funkcionalna čistoća

Budući da PLINQ izvršava upit u više paralelnih niti, morate voditi računa o tome da ne pokrećete operacije koje nisu bezbedne za višenitni rad. Pre svega, upisivanje vrednosti u promenljive proizvodi *sporedne efekte* i zato nije bezbedno u višenitnom radu:

```
// Sledeći upit množi svaki element njegovim indeksom.
// Ako krenemo od opsega vrednosti tipa Enumerable.Range(0,999),
// trebalo bi da rezultat budu kvadrati tih vrednosti.
int i = 0;
var query = from n in Enumerable.Range(0,999).AsParallel() select n * i++;
```

Operaciju inkrementiranja tekuće vrednosti `i` mogli smo da načinimo bezbednom za višenitni rad pomoću blokada, ali bi nam i dalje ostao problem da vrednost `i` neće uvek odgovarati indeksu tekućeg ulaznog elementa koji se obrađuje. Dodavanje operatora `AsOrdered` upitu ne bi rešilo taj problem, zato što `AsOrdered` obezbeđuje da se obrađeni elementi slažu redosledom kojim bi bili kada bi se obrađivali sekvencijalno – ali ne garantuje da se oni *zaista* obrađuju sekvencijalno.

Umesto toga, ovaj upit treba napisati tako da koristi indeksiranu verziju operatora `Select`:

```
var query = Enumerable.Range(0,999).AsParallel().Select ((n, i) => n * i);
```

Za najbolje performanse, trebalo bi da svaka metoda koja se poziva iz operatora bude bezbedna za višenitni rad zato što je projektovana tako da ne upisuje ništa ni u polja, ni u svojstva (nema sporedne efekte, tj. metoda je *funkcionalno čista*). Ako je metoda bezbedna za višenitni rad zato što sadrži *blokade*, mogućnost paralelizovanja upita biće biće ograničena – na trajanje blokade podeljeno s ukupnim trajanjem izvršavanja te funkcije.

Podešavanje stepena paralelizma

PLINQ standardno sam određuje optimalan stepen za raspoloživ procesor. Možete zadati drugačije ako pozovete metodu `WithDegreeOfParallelism` iza `AsParallel`:

```
...AsParallel().WithDegreeOfPallelism(4)...
```

Primer slučaja kada biste stepen paralelizma možda povisili na vrednost veću od broja jezgara jeste pri ulazno/izlaznim operacijama (recimo, preuzimanje više veb stranica istovremeno). Međutim, u C# 5 i Frameworku 4.5, kombinatori poslova i asinhrone funkcije pružaju rešenje koje je slično po jednostavnosti i *efikasnosti* (videti odeljak „Kombinatori poslova“, na strani 541 poglavlja 14. Za razliku od objekata `Task`, PLINQ ne može da obavlja ulazno/izlazne operacije bez blokiranja niti (i to niti *iz rezerve*, da bi stvari bile još gore).

Menjanje stepena paralelizma

Metodu `WithDegreeOfParallelism` možete pozvati samo jedanput unutar PLINQ upita. Ako treba da je pozovete ponovo, morate pokrenuti spajanje rezultata i ponovnu raspodelu obrade u upitu tako što unutar upita ponovo pozovete `AsParallel()`:

```
"The Quick Brown Fox"
  .AsParallel().WithDegreeOfParallelism (2)
  .Where (c => !char.IsWhiteSpace (c))
  .AsParallel().WithDegreeOfParallelism (3)   // Pokreće spajanje rezultata i novu
                                              // raspodelu obrade
  .Select (c => char.ToUpper (c))
```

Otkazivanje upita

Otkazivanje PLINQ upita čije rezultate preuzimate u petlji `foreach` lako je: samo izađite iz petlje `foreach` i izvršavanje upita se automatski otkazuje jer se enumerator implicitlno oslobađa.