C# 5.0 • Mali referentni priručnik

C# je objektno orijentisan programski jezik opšte namene, s obaveznom proverom tipova podataka. Cilj jezika je produktivnost programera, pa je stoga on istovremeno jednostavan, izražajan i efikasan. C# je neutralan u pogledu platforme, ali je napisan tako da odlično radi u okruženju Microsoft .NET Framework. Verzija C# 5.0 razvijena je za .NET Framework 4.5.

Napomena

Programi i odlomci koda u ovoj knjizi prate one iz poglavlja 2–4 iz knjige *C# 5.0 za programere* (*C# 5.0 in a Nutshell*) i svi su na raspolaganju kao interaktivni primeri u aplikaciji LINQPad. Rad na tim primerima uz ovaj mali priručnik ubrzava učenje jer ih možete menjati i odmah videti rezultate a da pritom ne morate postavljati projekte i rešenja u integrisano razvojno okruženje Visual Studio.

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

Konvencije korišćene u knjizi

U knjizi su korišćene sledeće tipografske konvencije:

Kurziv
: Koristi se za nove pojmove, URL-ove, adrese e-pošte, imena i oznake tipova (tj. nastavke imena) datoteka.

`Font konstantne širine`
: Koristi se za listinge programa, a unutar pasusa za programske elemente kao što su imena promenljivih i funkcija, baze podataka, tipove podataka, te promenljive okruženja, naredbe i rezervisane reči.

`Podebljan font konstantne širine`
: Koristi se za komande ili drugi tekst koji treba da unese korisnik.

`Kurzivni font konstantne širine`
: Koristi se za tekst koji treba da se zameni vrednostima koje unosi korisnik ili vrednostima koje zavise od konteksta.

Savet

Ukazuje na savet, predlog ili opštu napomenu.

Upozorenje

Ukazuje na oprez ili upozorenje.

Korišćenje primera koda

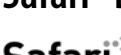
Namena ove knjige je da vam pomogne da obavite posao. U opštem slučaju, kôd iz knjige možete koristiti u svojim programima i dokumentaciji. Ne morate tražiti dozvolu od nas, osim ako reprodukujete značajan deo koda. Na primer, za pisanje programa u kome se koristi nekoliko segmenata koda iz ove knjige, ne treba vam dozvola. Za prodavanje ili distribuciju CD-ROM-ova s primerima iz knjiga izdavačke

kuće O'Reilly, dozvola je neophodna. Dozvola vam ne treba kada odgovarate na pitanja tako što citirate ovu knjigu i navodite primer koda iz nje. S druge strane, obavezno nabavite dozvolu ukoliko u dokumentaciji svog proizvoda citirate značajan deo primera koda iz ove knjige.

Ne zahtevamo, ali svakako cenimo navođenje izvora. To obično uključuje sledeće podatke: autor, izdavač i ISBN. Na primer: „C# 5.0 Pocket Reference, Joseph Albahari i Ben Albahari (O'Reilly). Copyright 2012 Joseph Albahari i Ben Albahari, 978-1-449-3201-71."

Ako smatrate da se upotreba primera koda u vašem slučaju ne uklapa u gore navedene uslove fer korišćenja ili dozvole, slobodno nam pišite na adresu permissions@oreilly.com.

Safari® Books Online



Safari Books Online (*www.safaribooksoline.com*) digitalna je biblioteka koja funkcioniše na zahtev i korisnicima obezbeđuje ekspertski sadržaj u obliku knjiga ili videa; autori sadržaja su vodeći svetski autori u oblastima tehnologije i biznisa.

Inženjeri, projektanti softvera, veb dizajneri i kreativci koriste Safari Books Online kao svoj primarni izvor za istraživanje, rešavanje problema, učenje i obuku za sertifikate.

Safari Books Online nudi niz kompleta proizvoda i cenovnih kategorija za organizacije, vladine agencije i pojedince. Pretplatnici imaju pristup hiljadama knjiga, videa za obuku i ranih rukopisa publikacija u jednoj potpuno pretraživoj bazi podataka, i to od izdavača kao što su O'Reilly Media, Prentice Hall Professional, Addison-Wesley Professional, Microsoft Press, Sams, Que, Peachpit Press, Focal Press, Cisco Press, John Wiley & Sons, Syngress, Morgan Kaufmann, IBM Redbooks, Packt, Adobe Press, FT Press, Apress, Manning, New Riders, McGraw-Hill, Jones & Bartlett, Course Technology i desetina drugih. Ako želite više informacija o servisu Safari Books Online, molimo vas da nas posetite na vebu.

Kako da stupite u kontakt s nama

Komentare i pitanja koji se odnose na izvorno izdanje ove knjige šaljite na adresu:

O'Reilly Media, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
800-998-9938 (u SAD ili u Kanadi)
707-829-0515 (međunarodni ili lokalni pristup)
707-829-0104 (faks)

Na veb stranicama ove knjige objavljuju se ispravke, primeri i sve dodatne informacije. Možete joj pristupiti na adresi:

http://oreil.ly/CSharp5_PR

Komentare ili tehnička pitanja o ovoj knjizi, pošaljite na sledeću adresu:

bookquestions@oreilly.com

Više informacija o knjigama, kursevima, konferencijama i novostima kompanije O'Reilly, naći ćete na adresi *http://www.oreilly.com*.

Potražite ih na Facebooku: *http://facebook.com/oreilly*

Pratite ih na Twitteru: *http://twitter.com/oreillymedia*

Gledajte ih na YouTubeu: *http://www.youtube.com/oreillymedia*

Informacije o izdanjima Mikro knjige potražite na adresi:

http://www.mikroknjiga.rs

Prvi program na jeziku C#

Evo programa koji množi 12 sa 30 i ispisuje rezultat, 360, na ekranu. Dupla kosa crta ukazuje na to da je ostatak reda *komentar*.

```
using System;                   // Učitavanje 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
```

U srcu ovog programa su dve naredbe (engl. *statements*). Naredbe se u jeziku C# izvršavaju sekvencijalno i završavaju znakom tačka i zarez. Prva naredba izračunava *izraz* `12 * 30` i rezultat smešta u *lokalnu promenljivu* po imenu `x`, celobrojnog tipa (engl. *integer*). Druga naredba poziva *metodu* `WriteLine` klase `Console` da ispiše promenljivu `x` u tekstualnom prozoru na ekranu.

Metoda izvršava akciju u obliku niza naredaba koji se naziva *blok naredaba* (engl. *statement block*) – par vitičastih zagrada koje sadrže nula ili više naredaba. Definisali smo samo jednu metodu, po imenu `Main`.

Pisanje funkcija višeg nivoa koje pozivaju funkcije nižeg nivoa pojednostavljuje program. Evo kako ćemo *refaktorisati* program pomoću metode koja se može ponovo upotrebiti i koja množi ceo broj 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 primi *ulazne podatke* (engl. *input*) od pozivaoca tako što joj se zadaju parametri i može da vrati *rezultat* (engl. *output*) pozivaocu tako što joj se zada *povratni tip* (engl. *return type*). Definisali smo

metodu `FeetToInches` koja ima parametar za unošenje stopa (`feet`), i povratni tip za rezultat u inčima, pri čemu su oba tipa `int` (celi brojevi).

Literali `30` i `100` su *argumenti* koji su prosleđeni metodi `FeetToInches`. U našem primeru, metoda `Main` ima prazne zagrade zato što nema parametre, i tipa je `void` zato što svom pozivaocu ne vraća nikakvu vrednost. C# prepoznaje metodu po imenu `Main` kao mesto podrazumevane ulazne tačke izvršavanja. Metoda `Main` može opciono da vrati tip `int` (umesto `void`) kako bi vratila neku vrednost izvršnom okruženju. Uz to, metoda `Main` može opciono da prihvati niz znakovnih nizova (stringova) kao parametar (niz će sačinjavati argumenti prosleđeni izvršnom programu). Na primer:

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

NAPOMENA

Niz (kao što je `string[]`) predstavlja fiksan broj elemenata određenog tipa (više informacija naći ćete u odeljku „Nizovi“, na strani 33).

Metode su jedna od nekoliko vrsta funkcija u jeziku C#. Koristili smo i drugu vrstu funkcija, *operator* `*`, koji služi za množenje. Osim toga, postoje još i *konstruktori*, *svojstva* (engl. *properties*), *događaji* (engl. *events*), *indekseri* i *finalizatori*.

U našem primeru, navedene dve metode grupisane su u klasu. *Klasa* grupiše funkcije članice i podatke članove da bi se formirao objektno orijentisan gradivni blok. Klasa `Console` grupiše članove zadužene za funkcionisanje ulaza/izlaza s komandne linije, kao što je metoda `WriteLine`. Naša klasa `Test` grupiše dve metode – `Main` i `FeetToInches`. Klasa je neka vrsta *tipa*, o čemu će biti reči u odeljku „Osnove tipova“ na strani 11.

Na krajnjem spoljnom nivou svakog programa, tipovi su organizovani u *imenske prostore* (engl. *namespaces*). Direktiva `using` je zadata da bi imenski prostor `System` bio dostupan našoj aplikaciji, kako bi kori-

stila klasu `Console`. Sve klase unutar imenskog prostora `TestPrograms` možemo definisati na sledeći način:

```
using System;

namespace TestPrograms
{

  class Test {...}
  class Test2 {...}
}
```

.NET Framework je organizovan u obliku ugnežđenih (engl. *nested*) imenskih prostora. Na primer, evo imenskog prostora koji sadrži tipove za rad sa tekstom:

```
using System.Text;
```

Direktiva `using` je tu radi pogodnosti; tip možete referencirati i navođenjem njegovog potpuno kvalifikovanog imena, tj. imena tipa kojem prethodi njegov imenski prostor – na primer, `System.Text.String Builder`.

Prevođenje programa

C# kompajler (prevodilac programa) pakuje izvorni kôd, zadat kao skup datoteka s nastavkom *.cs*, u *sklop* (engl. *assembly*). Sklop je jedinica pakovanja i primene u okruženju .NET. Sklop može biti ili *aplikacija* ili *biblioteka*. Standardna konzolna ili Windows aplikacija jeste datoteka tipa *.exe* i ima metodu `Main`. Biblioteka je datoteka tipa *.dll* i ekvivalentna je *.exe* datoteci, s tim što nema ulaznu tačku. Nju poziva (*referencira*) neka aplikacija ili druge biblioteke. .NET Framework je skup biblioteka.

C# kompajler se zove *csc.exe*. Za kompajliranje (prevođenje) možete koristiti neko integrisano razvojno okruženje (IDE), kao što je Visual Studio, ili ručno pozvati kompajler `csc` s komandne linije. Da biste program preveli ručno, prvo ga snimite u datoteku, npr. *MyFirstProgram.cs*, a zatim idite na komandnu liniju i pozovite `csc` (smešten u *%SystemRoot%\Microsoft.NET\Framework\<framework-version>* gde je *%SystemRoot%* vaš Windows direktorijum) na sledeći način:

```
csc MyFirstProgram.cs
```

Dobija se aplikacija pod imenom *MyFirstProgram.exe*. Da biste napravili biblioteku (*.dll*), uradite sledeće:

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

Sintaksa

Sintaksa jezika C# inspirisana je sintaksom programskih jezika C i C++. U ovom odeljku opisaćemo elemente sintakse jezika C# na primeru sledećeg programa:

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

Identifikatori i rezervisane reči

Identifikatori su imena koja programeri biraju za svoje klase, metode, promenljive itd. U našem primeru programa, identifikatori su:

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

Identifikator mora biti cela reč, sastavljena isključivo od Unicode znakova, i mora počinjati slovom ili znakom za podvlačenje. U C# identifikatorima pravi se razlika između malih i velikih slova. Po konvenciji, parametri, lokalne promenljive i privatna polja pišu se tzv. kamiljom notacijom – CamelCase – (npr., `mojaPromenljiva`), a svi drugi identifikatori – Pascal notacijom (npr., `MojaMetoda`).

Rezervisane reči (engl. *keywords*) jesu imena koja je rezervisao kompajler i koja ne možete koristiti kao identifikatore. U našem primeru, to su sledeće reči:

```
using   class   static   void   int
```

Evo i kompletne liste rezervisanih reči u jeziku C#:

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

Izbegavanje konflikata

Ako ipak želite da koristite rezervisanu reč kao identifikator, morate ispred nje staviti znak `@`. Na primer:

```
class class {...}   // Neispravno
class @class {...}  // Ispravno
```

Simbol `@` nije deo identifikatora. Znači, `@mojaPromenljiva` je isto što i `mojaPromenljiva`.

Kontekstualne rezervisane reči

Neke rezervisane reči su *kontekstualne*, što znači da se mogu koristiti i kao identifikatori – bez znaka `@`. To su:

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

Kontekstualne rezervisane reči ne mogu biti dvosmislene u kontekstu u kome su upotrebljene.

Literali, znakovi interpunkcije i operatori

Literali su prosti delovi podataka, leksički ugrađeni u program. U našem primeru, literali su `12` i `30`. *Znakovi interpunkcije* pomažu da se označi struktura programa. U našem programu, to su {, } i `;`.

Vitičaste zagrade grupišu više naredaba u blok naredaba. Znak tačka i zarez završava (pojedinačnu) naredbu. Naredbe se mogu prelamati u više redova:

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

Operator transformiše i kombinuje izraze. Većina operatora u jeziku C# pišu se pomoću simbola; takav je, na primer, i operator množenja, `*`. U našem programu, operatori su:

```
. () * =
```

Tačka označava pristupanje članu (ili decimalnu tačku u numeričkim literalima). Zagrade se koriste pri deklarisanju ili pozivanju metode; prazne zagrade se koriste kada metoda ne prihvata nijedan argument.

Znak jednakosti obavlja operaciju *dodele* (engl. *assignment*), dok dvostruki znak jednakosti, ==, poredi dve vrednosti kako bi se utvrdilo da li su jednake.

Komentari

C# podržava dva stila dokumentovanja izvornog koda: *jednoredne komentare* i *višeredne komentare*. Jednoredni komentar počinje sa dve kose crte i nastavlja se do kraja reda. Na primer:

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

Višeredni komentar počinje znakovima /* a završava znakovima */. Na primer:

```
int x = 3; /* ovo je komentar koji se proteže
u dva reda */
```

Komentari mogu da sadrže XML dokumentacione oznake (videti „XML dokumentacija“ na strani 202).

Osnove tipova

Tip definiše prirodu date vrednosti. U našem primeru koristili smo dva literala tipa `int` s vrednostima 12 i 30. Uz to, deklarisali smo i *promenljivu* tipa `int` po imenu `x`.

Promenljiva (engl. *variable*) označava lokaciju u memoriji koja može da sadrži različite vrednosti u različito vreme. Nasuprot tome, *konstanta* uvek predstavlja istu vrednost (više o tome kasnije).

U jeziku C#, svaka vrednost je *instanca* određenog tipa. Značenje same vrednosti i skup mogućih vrednosti date promenljive, određeni su njenim tipom.

Primeri unapred definisanih tipova

Unapred definisani tipovi, engl. *predefined types* (zovu se i ugrađeni tipovi, engl. *built-in types*) jesu tipovi koje kompajler standardno podržava. Tip `int` je unapred definisan tip namenjen za predstavljanje

skupa celih brojeva (engl. *integers*) koji staju u 32 bita memorije, od -2^{31} do $2^{31}-1$. Evo kako možemo da izvršavamo aritmetičke funkcije sa instancama tipa `int`:

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

Još jedan od unapred definisanih tipova u jeziku C# jeste `string`. Tip `string` predstavlja znakovne nizove, tj. sekvence znakova – na primer, „.NET“ ili „*http://oreilly.com*“. Sa znakovnim nizovima možemo raditi tako što ih pozivamo pomoću funkcija:

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

int x = 2014;
message = message + x.ToString();
Console.WriteLine (message);             // Hello world2014
```

Unapred definisan tip `bool` ima samo dve moguće vrednosti: `true` i `false`. Tip `bool` se obično koristi za uslovno grananje izvršnog toka programa s naredbom `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");
```

Napomena

Imenski prostor `System` u .NET Frameworku sadrži mnoge važne tipove koji u jeziku C# nisu unapred definisani (npr., `DateTime`).

Primeri namenskih tipova

Kao što složene funkcije možemo da gradimo od onih jednostavnih, tako i složene tipove možemo praviti od osnovnih. U ovom primeru definisaćemo namenski tip (engl. *custom type*) po imenu `UnitConverter` – klasu koja služi za konverziju mernih jedinica:

```
using System;
public class UnitConverter
{
  int ratio;                                  // Polje

  public UnitConverter (int unitRatio)        // Konstruktor
  {
    ratio = unitRatio;
  }

  public int Convert (int unit)               // Metoda
  {
    return unit * ratio;
  }
}

class Test
{
  static void Main()
 {
    UnitConverter feetToInches = new UnitConverter(12);
    UnitConverter milesToFeet = new UnitConverter(5280);

    Console.Write (feetToInches.Convert(30));   // 360
    Console.Write (feetToInches.Convert(100));  // 1200
    Console.Write (feetToInches.Convert
                    (milesToFeet.Convert(1)));  // 63360
  }
}
```

Članovi tipa

Tip sadrži *podatke članove* (engl. *data members*) i *funkcije članice* (engl. *function members*). Podatak član namenskog tipa `UnitConverter` jeste *polje* (engl. *field*) po imenu `ratio`. Funkcije članice tipa `UnitConverter` jesu metoda `Convert` i konstruktor `UnitConvertera`.

Simetrija između unapred definisanih tipova i namenskih tipova

U jeziku C# zgodno je to što se unapred definisani tipovi i namenski tipovi ne razlikuju mnogo. Unapred definisan tip `int` služi za cele brojeve. On sadrži podatke – 32 bita – i stavlja ih na raspolaganje funkcijama članicama, npr. funkciji `ToString`. Slično tome, naš namenski tip `UnitConverter` služi za konverzije mernih jedinica. On sadrži podatke – u ovom slučaju, `ratio` – i stavlja ih na raspolaganje funkcijama članicama koje ih koriste.

Konstruktori i instanciranje

Podaci nastaju *instanciranjem* tipa. Unapred definisani tipovi instanciraju se jednostavno – korišćenjem literala kao što je `12` ili `"Hello, world"`.

Instance namenskog tipa pravi operator `new`. Metodu `Main` smo započeli tako što smo napravili dve instance tipa `UnitConverter`. Odmah nakon što operator `new` instancira objekat, poziva se *konstruktor* tog objekta da ga inicijalizuje. Konstruktor se definiše kao metoda, s tim što su ime metode i tip rezultata (povratne vrednosti) ime samog tipa:

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

Poređenje članova instanci i statičkih članova

Podaci članovi i funkcije članice koji deluju na *instancu* datog tipa zovu se članovi instance (engl. *instance members*). Metoda `Convert` tipa `UnitConverter` i metoda `ToString` tipa `int` primeri su članova instance. Članovi tipa su podrazumevano članovi instance.

Podaci članovi i funkcije članice koji ne deluju na instancu tipa nego na sam tip, moraju biti označeni 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. Nikad se ne prave instance klase `Console` – jednu konzolu deli cela aplikacija.

Da bismo istakli razlike između članova instance i statičkih članova, reći ćemo da se polje `Name` instance odnosi na jednu određenu instancu klase `Panda`, dok se `Population` odnosi na skup svih instanci klase `Panda`:

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

  public Panda (string n)       // Konstruktor
  {
    Name = n;                   // Dodeljivanje vrednosti polju
                                // instance
     Population = Population+1; // Inkrementiranje statičkog
                                // polja
  }
}
```

Naredni kôd pravi dve instance klase `Panda`, ispisuje njihova imena, a zatim ispisuje ukupan broj instanci klase:

```
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 članove klase drugim klasama. U ovom primeru, kada polje `Name` klase `Panda` ne bi bilo javno, klasa `Test` ne bi mogla da mu pristupi. Označavanjem svog člana rezervisanom rečju

`public`, tip kaže: „Evo šta želim da vide ostali tipovi – sve ostalo su moji privatni detalji implementacije.“ U terminologiji objektno orijentisanog programiranja, kažemo da javni članovi *kapsuliraju* (engl. *encapsulate*) privatne članove klase.

Konverzije

C# može da konvertuje instance kompatibilnih tipova jedne u druge. Konverzijom uvek nastaje nova vrednost od postojeće. Konverzije mogu biti *implicitne* ili *eksplicitne*: implicitne konverzije se obavljaju automatski, dok je za eksplicitne konverzije potrebno *pretvaranje* (engl. *cast*). U sledećem primeru, *implicitno* konvertujemo tip `int` u tip `long` (koji ima dvaput veći kapacitet u bitovima od `int`) i *eksplicitno* konvertujemo tip `int` u tip `short` (koji ima upola manji kapacitet u bitovima od `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-bitni
                     // ceo broj
```

U opštem slučaju, implicitne konverzije su dozvoljene kada kompajler može da garantuje da će one uvek uspeti bez gubljenja informacija. U suprotnom, morate obaviti eksplicitnu konverziju između kompatibilnih tipova.

Poređenje vrednosnih i referentnih tipova

U jeziku C#, tipovi se mogu podeliti na *vrednosne* (engl. *value types*) i *referentne* (engl. *reference types*).

U *vrednosne tipove* spada većina ugrađenih tipova (konkretno, svi numerički tipovi, tip `char` i tip `bool`) kao i namenski tipovi `struct` i `enum`. U *referentne tipove* spadaju svi tipovi klasa, nizova, delegata i interfejsa.

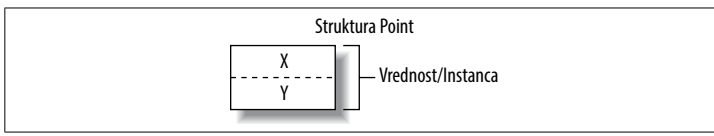
Osnovna razlika između vrednosnih i referentnih tipova jeste način rada s njima u memoriji.

Vrednosni tipovi

Sadržaj pomenljive ili konstante *vrednosnog tipa* jeste samo neka vrednost. Na primer, ugrađen vrednosni tip `int` sadrži 32 bita podataka.

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

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



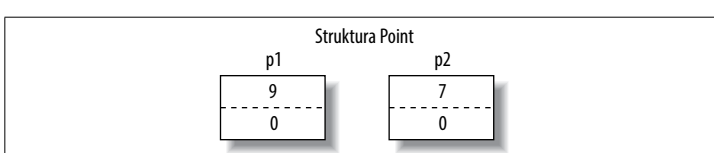
Slika 1. Instanca vrednosnog tipa u memoriji

Kada se instanca vrednosnog tipa dodeljuje drugoj instanci, postojeća instanca se uvek kopira. Na primer:

```
Point p1 = new Point();
p1.X = 7;

Point p2 = p1;             // Dodeljivanje uzrokuje kopiranje
Console.WriteLine (p1.X);  // 7
Console.WriteLine (p2.X);  // 7
p1.X = 9; // Promena p1.X
Console.WriteLine (p1.X);  // 9
Console.WriteLine (p2.X);  // 7
```

Slika 2 prikazuje da su `p1` i `p2` uskladišteni nezavisno.

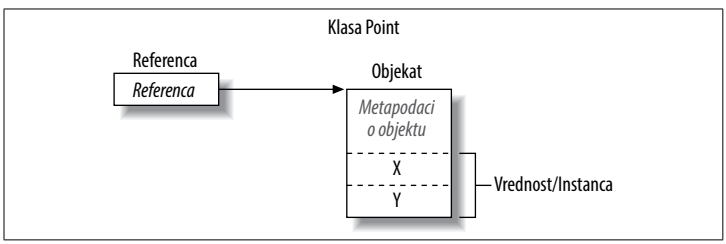


Slika 2. Pri dodeljivanju se instanca vrednosnog tipa kopira

Referentni tipovi

Referentni tip je složeniji od vrednosnog, i ima dva dela: *objekat* i *referencu* na taj objekat. Promenljiva ili konstanta referentnog tipa sadrži referencu na objekat u kome se nalazi vrednost. Evo tipa `Point` iz prethodnog primera, s tim što je sada napisan kao klasa (slika 3):

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



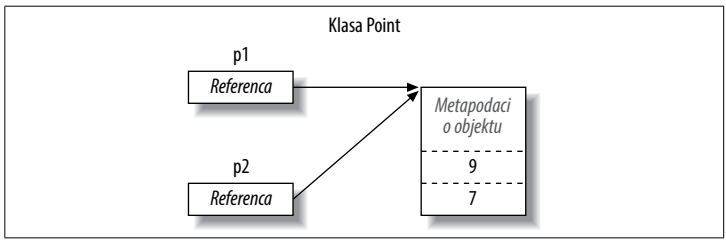
Slika 3. Instanca referentnog tipa u memoriji

Kada se instanca referentnog tipa dodeljuje drugoj instanci, kopira se referenca instance a ne objekat. To omogućava da više promenljivih referencira isti objekat – što obično nije moguće postići s vrednosnim tipovima. Ako ponovimo prethodni primer, ali s promenljivom `Point` koja je sada klasa, operacija nad `p1` utiče i na `p2`:

```
Point p1 = new Point();
p1.X = 7;

Point p2 = p1;             // Kopira referencu na p1
Console.WriteLine (p1.X);  // 7
Console.WriteLine (p2.X);  // 7
p1.X = 9;                  // Promena p1.X
Console.WriteLine (p1.X);  // 9
Console.WriteLine (p2.X);  // 9
```

Slika 4 prikazuje da su `p1` i `p2` dve reference koje upućuju na isti objekat.



Slika 4. Dodeljivanjem se kopira referenca

Vrednost null

Referentnom tipu se može dodeliti literal `null`, što znači da ta referenca ne upućuje ni na jedan objekat. Pod pretpostavkom da je `Point` klasa:

```
Point p = null;
Console.WriteLine (p == null); // true
```

Pristupanje članu s referencom `null`, generiše grešku pri izvršavanju (engl. *runtime error*):

```
Console.WriteLine (p.X); // NullReferenceException
```

Suprotno tome, vrednosni tip obično ne može da ima vrednost `null`:

```
struct Point {...}
...
Point p = null; // Greška pri prevođenju
int x = null;   // Greška pri prevođenju
```

NAPOMENA

C# ima specijalan konstrukt pod imenom *tip koji prihvata vrednost null* (engl. *nullable type*) za predstavljanje praznih elemenata vrednosnog tipa (više informacija u odeljku „Tipovi koji prihvataju null“ na strani 135).

Taksonomija unapred definisanih tipova

U jeziku C#, unapred definisani tipovi su:

Vrednosni tipovi

- Numerički
  - Ceo broj s predznakom (`sbyte`, `short`, `int`, `long`)
  - Ceo broj bez predznaka (`byte`, `ushort`, `uint`, `ulong`)
  - Realan broj (`float`, `double`, `decimal`)
- Logički izraz (`bool`)
- Znak (`char`)

Referentni tipovi

- Znakovni niz (`string`)
- Objekat (`object`)

Unapred definisani tipovi u jeziku C# alijasi su Framework tipova u imenskom prostoru `System`. Sledeće dve naredbe razlikuju se samo sintaksički:

```
int i = 5;
System.Int32 i = 5;
```

Unapred definisani *vrednosni tipovi* – izuzev `decimal` – poznati su kao CLR (Common Language Runtime) *osnovni tipovi* (engl. *primitive types*). Osnovni tipovi se tako zovu zato što su podržani direktno preko instrukcija u kompajliranom kodu, što obično znači da se prevode direktno u naredbe koje podržava odgovarajući procesor.

Numerički tipovi

C# sadrži sledeće unapred definisane numeričke tipove:

C# tip	Sistemski tip	Sufiks	Veličina	Opseg
Celobrojni s predznakom				
`sbyte`	`SByte`		8 bitova	od -2^7 do 2^7-1
`short`	`Int16`		16 bitova	od -2^{15} do $2^{15}-1$
`int`	`Int32`		32 bita	od -2^{31} do $2^{31}-1$
`long`	`Int64`	`L`	64 bita	od -2^{63} do $2^{63}-1$

C# tip	Sistemski tip	Sufiks	Veličina	Opseg
Celobrojni bez predznaka				
`byte`	`Byte`		8 bitova	od 0 do 2^8-1
`ushort`	`UInt16`		16 bitova	od 0 do $2^{16}-1$
`uint`	`UInt32`	`U`	32 bita	od 0 do $2^{32}-1$
`ulong`	`UInt64`	`UL`	64 bita	od 0 do $2^{64}-1$
Realan broj				
`float`	`Single`	`F`	32 bita	± (od ~10^{-45} do 10^{38})
`double`	`Double`	`D`	64 bita	± (od ~10^{-324} do 10^{308})
`decimal`	`Decimal`	`M`	128 bitova	± (od ~10^{-28} do 10^{28})

Među *celobrojnim* tipovima, `int` i `long` su građani prvog reda i favorizuju ih i C# i CLR. Ostali celobrojni tipovi se obično koriste radi postizanja interoperabilnosti ili kada je najvažnije efikasno korišćenje memorije.

Među tipovima *realnih* brojeva, `float` i `double` se zovu *tipovi s pokretnim zarezom* (engl. *floating-point types*) i obično se koriste za naučna izračunavanja. Tip `decimal` se najčešće upotrebljava za finansijske proračune, gde su potrebni aritmetika brojeva sa osnovom 10 i velika preciznost. (Tehnički, `decimal` je takođe tip s pokretnim zarezom, mada se retko tako naziva.)

Numerički literali

Za *celobrojne literale* može da se koristi decimalna ili heksadecimalna notacija; heksadecimalni literal se označava prefiksom `0x` (na primer, `0x7f` je isto što i `127`). *Realni literali* mogu se pisati pomoću decimalne ili eksponencijalne notacije – recimo, `1E06`.

Pretpostavke o tipu numeričkog literala

Kompajler podrazumevano zaključuje da je numerički literal ili tipa `double` ili celobrojnog tipa:

- Ako literal sadrži decimalnu tačku ili eksponencijalni simbol (`E`), onda je tipa `double`.
- U suprotnom, tip literala je prvi tip iz sledeće liste u koji može da stane vrednost literala: `int`, `uint`, `long` i `ulong`.

Na primer:

```
Console.Write (       1.0.GetType());  // Double (double)
Console.Write (      1E06.GetType());  // Double (double)
Console.Write (         1.GetType());  // Int32 (int)
Console.Write (0xF0000000.GetType());  // UInt32 (uint)
```

Numerički sufiksi

Numerički sufiksi navedeni u prethodnoj tabeli eksplicitno definišu tip literala:

```
decimal d = 3.5M;   // M = decimal (svejedno da li je malo
                    // ili veliko slovo)
```

Sufiksi `U` i `L` su retko kada potrebni, pošto se tipovi `uint`, `long` i `ulong` gotovo uvek mogu ili *pogoditi* ili *implicitno konvertovati* iz `int`:

```
long i = 5; // Implicitna konverzija iz tipa int u tip long
```

Tehnički, sufiks `D` je suvišan, zato što se pretpostavlja da su svi literali s decimalnom tačkom tipa `double` (a uvek možete dodati decimalnu tačku numeričkom literalu). Sufiksi `F` i `M` su najkorisniji i obavezni su pri navođenju frakcionih literala tipa `float` ili `decimal`. Bez sufiksa, sledeći iskaz se ne bi preveo, jer bi se smatralo da je vrednost 4.5 tipa `double`, koja se ne konvertuje implicitno u `float` ili `decimal`:

```
float f = 4.5F;       // Neće se prevesti bez sufiksa
decimal d = -1.23M;   // Neće se prevesti bez sufiksa
```

Konverzije brojeva

Konverzije celih brojeva u cele brojeve

Konverzije celih brojeva su implicitne kada odredišni tip može da predstavlja svaku moguću vrednost izvornog tipa. U svim drugim slučajevima, neophodna je *eksplicitna* konverzija. Na primer:

```
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-bitni
                     // ceo broj
```

Konverzije realnih brojeva u realne

Realan broj, tj. broj s pokretnim zarezom, može se implicitno konvertovati u tip `double`, pošto `double` može da predstavlja svaku moguću vrednost tipa `float`. Obrnuta konverzija mora biti eksplicitna.

Konverzije između tipa `decimal` i ostalih tipova realnih brojeva moraju biti eksplicitne.

Konverzije realnih brojeva u cele brojeve

Konverzije celobrojnih tipova u realne implicitne su, dok konverzije u obrnutom smeru moraju biti eksplicitne. Pri konvertovanju broja s pokretnim zarezom u ceo broj, odseca se decimalni deo – ukoliko postoji; za konverzije sa zaokruživanjem, koristite statičku klasu `System.Convert`.

Začkoljica je to što se pri implicitnom konvertovanju velikog celog broja u broj s pokretnim zarezom zadržava *red veličine* ali se ponekad može izgubiti *preciznost*:

```
int i1 = 100000001;
float f = i1;     // Sačuvan red veličine ali izgubljena
                  // preciznost
int i2 = (int)f;  // 100000000
```

Aritmetički operatori

Aritmetički operatori (`+`, `-`, `*`, `/`, `%`) definisani su za sve numeričke tipove osim za 8-bitne i 16-bitne celobrojne tipove. Operator `%` izračunava ostatak deljenja.

Operatori za inkrementiranje i dekrementiranje

Operatori za inkrementiranje i dekrementiranje (`++`, `--`) povećavaju odnosno smanjuju vrednosti numeričkih tipova za 1. Operator može da bude ili ispred ili iza promenljive, zavisno od toga da li želite da se vrednost promenljive ažurira *pre* ili *posle* izračunavanja izraza. Na primer:

```
int x = 0;
Console.WriteLine (x++);    // Rezultat je 0; x je sada 1
Console.WriteLine (++x);    // Rezultat je 2; x je sada 2
Console.WriteLine (--x);    // Rezultat je 1; x je sada 1
```

Specijalizovane operacije s celim brojevima

Deljenje celih brojeva

U operacijama deljenja celih brojeva uvek se odsecaju ostaci deljenja (brojevi se zaokružuju ka nuli). Deljenje promenljivom čija je vrednost nula izaziva grešku pri izvršavanju, engl. *runtime error* (`DivideByZeroException`). Deljenje *literalom* ili *konstantom* 0 izaziva grešku pri prevođenju, engl. *compile-time error*.

Prekoračenje u operacijama s celobrojnim tipovima

U vreme izvršavanja aritmetičkih operacija nad celobrojnim tipovima, može doći do prekoračenja (engl. *overflow*). To se standardno odvija u tišini – ne generiše se izuzetak a rezultat je takav kao da je izračunavanje obavljeno na većem celobrojnom tipu pri čemu je višak značajnih bitova odbačen. Na primer, dekrementiranjem minimalne moguće vrednosti `int` dobija se maksimalna moguća vrednost `int`:

```
int a = int.MinValue;
a--;
Console.WriteLine (a == int.MaxValue);  // true
```

Operatori checked i unchecked

Operator `checked` nalaže izvršnom okruženju da generiše izuzetak `OverflowException` umesto tihog prekoračenja kada celobrojni izraz ili naredba prekorači aritmetičke granice za taj tip podataka. Opera-

tor `checked` deluje na izraze s operatorima ++, --, (unarni) –, +, –, *, / i operatorima za eksplicitnu konverziju celobrojnih tipova.

Operator `checked` možete primeniti na jedan izraz ili na blok naredaba. Na primer:

```
int a = 1000000, b = 1000000;

int c = checked (a * b);   // Proverava samo dati izraz

checked                    // Proverava sve izraze
{                          // u bloku naredaba.
  c = a * b;
  ...
}
```

Možete učiniti da se aritmetičko prekoračenje podrazumevano proverava za sve izraze u programu tako što ćete za kompajliranje koristiti opciju komandne linije `/checked+` (u okruženju Visual Studio, idite na Advanced Build Settings). Ukoliko nakon toga bude trebalo da isključite proveru prekoračenja samo za određene izraze ili naredbe, to možete postići pomoću operatora `unchecked`.

Operatori nad bitovima

C# podržava sledeće operatore nad bitovima (engl. *bitwise operators*):

Operator	Značenje	Primer izraza	Rezultat
~	Komplement	`~0xfU`	`0xfffffff0U`
&	I	`0xf0 & 0x33`	`0x30`
|	Ili	`0xf0 | 0x33`	`0xf3`
^	Ekskluzivno ili	`0xff00 ^ 0x0ff0`	`0xf0f0`
<<	Pomeranje ulevo	`0x20 << 2`	`0x80`
>>	Pomeranje udesno	`0x20 >> 1`	`0x10`

8-bitni i 16-bitni celobrojni tipovi

8-bitni i 16-bitni celobrojni tipovi su `byte`, `sbyte`, `short` i `ushort`. Ovi tipovi nemaju sopstvene aritmetičke operatore, pa ih C# implicitno

konvertuje u obimnije tipove, po potrebi. To može izazvati grešku pri prevođenju kada kompajler pokuša da dobijeni rezultat dodeli nekom manjem celobrojnom tipu:

```
short x = 1, y = 1;
short z = x + y;           // Greška pri prevođenju
```

U ovom slučaju, `x` i `y` se implicitno konvertuju u `int` da bi se mogli sabrati. To znači da je rezultat takođe tipa `int`, koji se ne može implicitno vratiti u tip `short` (jer bi se mogli izgubiti podaci). Da bi se ovaj kôd preveo, moramo dodati eksplicitno konvertovanje:

```
short z = (short) (x + y);     // OK
```

Specijalne vrednosti tipova float i double

Za razliku od celobrojnih tipova, tipovi s pokretnim zarezom imaju vrednosti koje se u određenim operacijama tretiraju na poseban način. Te specijalne vrednosti su NaN (Not a Number – nije broj), +∞, –∞ i –0. Klase `float` i `double` imaju konstante za NaN, +∞ i –∞ (kao i za druge vrednosti, uključujući `MaxValue`, `MinValue` i `Epsilon`). Na primer:

```
Console.Write (double.NegativeInfinity); // Minus beskonačno
```

Deljenjem broja različitog od nule nulom, dobija se beskonačna vrednost:

```
Console.WriteLine ( 1.0 / 0.0);   // Beskonačno
Console.WriteLine (–1.0 / 0.0);   // Minus beskonačno
Console.WriteLine ( 1.0 / –0.0);  // Minus beskonačno
Console.WriteLine (–1.0 / –0.0);  // Beskonačno
```

Deljenjem nule nulom ili oduzimanjem jedne beskonačne vrednosti od druge, dobija se NaN:

```
Console.Write ( 0.0 / 0.0);                 // NaN
Console.Write ((1.0 / 0.0) – (1.0 / 0.0));  // NaN
```

Kada se koristi operator `==`, NaN vrednost nikada nije jednaka drugoj vrednosti, čak ni drugoj NaN vrednosti. Da biste proverili da li je neka vrednost NaN, morate koristiti metodu `float.IsNaN` ili `double.IsNaN`:

```
Console.WriteLine (0.0 / 0.0 == double.NaN);   // False
Console.WriteLine (double.IsNaN (0.0 / 0.0));  // True
```

Međutim, kada se koristi metoda `object.Equals`, dve NaN vrednosti su jednake:

```
bool isTrue = object.Equals (0.0/0.0, double.NaN);
```

Poređenje tipova double i decimal

Tip `double` je koristan u naučnim proračunima (npr. za izračunavanje prostornih koordinata). Tip `decimal` je koristan za finansijske proračune i za vrednosti koje je smislio čovek a ne za one koje su rezultat stvarnih merenja. Evo sažetka razlika između ova dva tipa podataka:

Osobina	`double`	`decimal`
Interno predstavljanje	Osnova 2	Osnova 10
Preciznost	15–16 značajnih cifara	28–29 značajnih cifara
Opseg	±(od ~10^{-324} do ~10^{308})	±(od ~10^{-28} do ~10^{28})
Specijalne vrednosti	+0, –0, +∞, –∞ i NaN	Nema
Brzina	Zavisi od procesora	Ne zavisi od procesora (oko 10 puta sporije od tipa `double`)

Greške zaokruživanja realnih brojeva

Pošto tipovi `float` i `double` interno predstavljaju brojeve sa osnovom 2, većina literala s decimalnim delom (sa osnovom 10) neće biti predstavljena precizno:

```
float tenth = 0.1f;                    // Nije baš 0.1
float one = 1f;
Console.WriteLine (one - tenth * 10f);  // -1.490116E-08
```

Zbog toga tipovi `float` i `double` nisu pogodni za finansijska izračunavanja. Za razliku od njih, tip `decimal` koristi osnovu 10 pa može precizno da predstavi decimalne brojeve kao što je 0.1 (u čijem prikazu sa osnovom 10 nema periodičnog ponavljanja cifara).

Logički tipovi i operatori

Logički tip u jeziku C# (alijas tipa `System.Boolean`) jeste logička vrednost kojoj se može dodeliti literal `true` ili `false`.

Mada je za skladištenje logičke vrednosti potreban samo jedan bit, izvršno okruženje će koristiti jedan bajt memorije, pošto je to najmanja jedinica s kojom izvršno okruženje i procesor mogu efikasno da rade. Da bi se izbeglo neefikasno korišćenje prostora pri radu s nizovima, u imenskom prostoru `System.Collections`, Framework sadrži klasu `BitArray` koja koristi samo jedan bit po logičkoj vrednosti.

Operatori jednakosti i poređenja

Operatori `==` i `!=` ispituju jednakost odnosno nejednakost svih tipova podataka, i uvek vraćaju logičku vrednost. Jednakost vrednosnih tipova obično se izražava veoma jednostavno:

```
int x = 1, y = 2, z = 1;
Console.WriteLine (x == y);   // False
Console.WriteLine (x == z);   // True
```

Jednakost referentnih tipova se, podrazumevano, zasniva na jednakosti *referenci*, a ne na jednakosti stvarnih vrednosti objekata. Prema tome, dve instance objekta sa identičnim podacima ne smatraju se jednakim osim kada je operator `==` posebno preklopljen da bi se to postiglo (više informacija potražite u odeljcima „Tip object“ na strani 81 i „Preklapanje operatora“ na strani 140).

Operatori jednakosti i poređenja, `==`, `!=`, `<`, `>`, `>=` i `<=`, rade sa svim numeričkim tipovima, ali ih treba pažljivo koristiti s realnim brojevima (videti odeljak „Greške zaokruživanja realnih brojeva“ na strani 27). Operatori poređenja rade i sa nabrajanjima, tj. sa članovima tipa `enum`, tako što porede njihove pripadajuće celobrojne vrednosti.

Uslovni operatori

Operatori `&&` i `||` ispituju uslove *and* i *or*. Često se koriste zajedno s operatorom `!`, koji znači *not* (nije). U ovom primeru, metoda `UseUm-`

`brella` vraća `true` ako je kišovito ili sunčano (da bismo se zaštitili od kiše ili sunca), pod uslovom da nije i vetrovito (pošto su kišobrani neupotrebljivi po vetru):

```
static bool UseUmbrella (bool rainy, bool sunny,
                         bool windy)
{
  return !windy && (rainy || sunny);
}
```

Operatori `&&` i `||` *zaobilaze* izračunavanje kad god je to moguće. U prethodnom primeru, ako je vetrovito, vrednost izraza `(rainy || sunny)` uopšte se i ne izračunava. Takvo zaobilaženje je važno jer omogućava da se izrazi poput sledećeg izvršavaju a da se pritom ne generiše izuzetak `NullReferenceException`:

```
if (sb != null && sb.Length > 0) ...
```

Operatori `&` i `|` takođe ispituju uslove *and* i *or*:

```
return !windy & (rainy | sunny);
```

Razlika je to što oni ne zaobilaze izraze, pa se stoga retko koriste umesto uslovnih operatora. Ternarni uslovni operator (koji se jednostavno zove uslovni operator, engl. *conditional operator*) ima oblik `q ? a : b`, pri čemu važi sledeće: ako je uslov `q` ispunjen, izračunava se izraz `a`, inače se izračunava `b`. Na primer:

```
static int Max (int a, int b)
{
  return (a > b) ? a : b;
}
```

Uslovni operator je naročito koristan u LINQ upitima.

Znakovni nizovi i znakovi

U jeziku C#, tip `char` (alijas tipa `System.Char`) predstavlja Unicode znak i zauzima dva bajta. Literal `char` se zadaje unutar polunavodnika:

```
char c = 'A';   // Jednostavan znak
```

Pomoću *izlaznih sekvenci* (engl. *escape sequences*) pišu se znakovi koje ne treba izraziti ili tumačiti doslovno. Izlaznu sekvencu čine obrnuta kosa crta praćena znakom specijalnog značenja. Na primer:

```
char newLine = '\n';
char backSlash = '\\';
```

U izlaznim sekvencama koriste se sledeći znakovi:

Znak	Značenje	Vrednost
`\'`	Polunavodnik	0x0027
`\"`	Navodnik	0x0022
`\\`	Obrnuta kosa crta	0x005C
`\0`	Null (prazan znak)	0x0000
`\a`	Alarm	0x0007
`\b`	Brisanje prethodnog znaka (Backspace)	0x0008
`\f`	Prelazak na sledeću stranicu (Form feed, FF)	0x000C
`\n`	Novi red (New line, NL)	0x000A
`\r`	Vraćanje na početak reda (Carriage return, CR)	0x000D
`\t`	Horizontalni tabulator	0x0009
`\v`	Vertikalni tabulator	0x000B

Izlazna sekvenca `\u` (ili `\x`) omogućava da zadate bilo koji Unicode znak pomoću njegovog četvorocifrenog heksadecimalnog koda:

```
char copyrightSymbol = '\u00A9';
char omegaSymbol = '\u03A9';
char newLine = '\u000A';
```

Implicitna konverzija tipa `char` u numerički tip moguća je za numeričke tipove koji mogu da prime vrednost tipa `short` bez predznaka. Za ostale numeričke tipove neophodna je eksplicitna konverzija.

Tip string

U jeziku C#, tip `string` (alijas tipa `System.String`) predstavlja nepromenljivu sekvencu Unicode znakova. Literal tipa `string` zadaje se unutar navodnika:

```
string a = "Heat";
```

NAPOMENA

`string` je referentni tip a ne vrednosni. Međutim, njegovi operatori jednakosti slede semantiku vrednosnih tipova:

```
string a = "test", b = "test";
Console.Write (a == b); // True
```

Izlazne sekvence koje važe za literale tipa `char`, funkcionišu i unutar znakovnih nizova:

```
string a = "Here's a tab:\t";
```

Cena toga je sledeća: kad god vam treba baš obrnuta kosa crta, morate je napisati dvaput:

```
string a1 = "\\\\server\\fileshare\\helloworld.cs";
```

Da bi se izbegao taj problem, C# dozvoljava *doslovne* (engl. *verbatim*) znakovne literale. Doslovni znakovni literal ima prefiks `@` i ne podržava izlazne sekvence. Sledeći doslovni znakovni niz identičan je prethodnom:

```
string a2 = @"\\server\fileshare\helloworld.cs";
```

Doslovni znakovni literal može se protezati i na više redova. Znak navoda ćete uvrstiti u doslovni literal tako što ćete ga napisati dvaput.

Nadovezivanje znakovnih nizova

Operator `+` nadovezuje dva znakovna niza:

```
string s = "a" + "b";
```

Jedan od operanada ne mora biti znakovni niz; u tom slučaju, metoda `ToString` se poziva za tu vrednost. Na primer:

```
string s = "a" + 5; // a5
```

Višekratno korišćenje operatora `+` da bi se napravio znakovni niz često je neefikasno: bolje je koristiti tip `System.Text.String Builder` –

on predstavlja izmenljiv znakovni niz i ima metode za efikasno dodavanje (`Append`), umetanje (`Insert`), uklanjanje (`Remove`) i zamenu (`Replace`) podnizova.

Poređenje znakovnih nizova

Tip `string` ne podržava operatore poređenja `<` i `>`. Umesto njih morate koristiti metodu `CompareTo` tipa `string`, koja vraća pozitivan broj, negativan broj ili nulu, zavisno od toga da li je prva vrednost po redosledu iza, ispred ili na mestu druge vrednosti:

```
Console.Write ("Boston".CompareTo ("Austin")); // 1
Console.Write ("Boston".CompareTo ("Boston")); // 0
Console.Write ("Boston".CompareTo ("Chicago")); // -1
```

Pretraživanje znakovnih nizova

Indekser podataka tipa `string` vraća znak koji se nalazi na zadatoj poziciji:

```
Console.Write ("word"[2]); // r
```

Metode `IndexOf` i `LastIndexOf` traže znak unutar zadatog znakovnog niza. Metode `Contains`, `StartsWith` i `EndsWith` traže podniz u zadatom znakovnom nizu.

Obrada znakovnih nizova

Pošto je tip `string` nepromenljiv, sve metode koje „manipulišu“ znakovnim nizom vraćaju nov niz, ostavljajući originalni nedirnut:

- `Substring` izdvaja deo znakovnog niza.
- `Insert` umeće znakove na zadato mesto, a `Remove` ih uklanja odatle.
- `PadLeft` i `PadRight` dodaju beline.
- `TrimStart`, `TrimEnd` i `Trim` uklanjaju beline.

Klasa `string` definiše i metode `ToUpper` i `ToLower` za promenu malih slova u velika i obrnuto, metodu `Split` koja deli znakovni niz na podnizove (na osnovu zadatih graničnika), i statičku metodu `Join` – za ponovno spajanje podnizova u znakovni niz.

Nizovi

Niz (engl. *array*) predstavlja fiksan broj elemenata određenog tipa. Elementi niza se uvek čuvaju u kontinualnom bloku memorije, pa im se može veoma efikasno pristupati.

Niz se označava uglastim zagradama iza tipa elemenata. Sledećim izrazom deklariše se niz od pet znakova:

```
char[] vowels = new char[5];
```

Uglaste zagrade služe i za *indeksiranje* niza; određenom elementu se pristupa preko njegovog položaja:

```
vowels[0] = 'a'; vowels[1] = 'e'; vowels[2] = 'i';
vowels[3] = 'o'; vowels[4] = 'u';

Console.WriteLine (vowels [1]);      // e
```

Rezultat je „e“ zato što indeksi niza počinju od 0. Možemo koristiti petlju s naredbom `for` da bismo prošli kroz svaki element niza – to se zove *iteracija*. U ovom primeru, petlja `for` menja vrednost celog broja `i` od `0` do `4`:

```
for (int i = 0; i < vowels.Length; i++)
  Console.Write (vowels [i]);                // aeiou
```

Nizovi implementiraju i interfejs `IEnumerable<T>` (videti „Nabrajanje i iteratori“ na strani 130), pa možete i nabrojiti članove pomoću naredbe `foreach`:

```
foreach (char c in vowels) Console.Write (c);    // aeiou
```

Tokom izvršavanja se proverava da li su svi indeksi niza u zadatom opsegu. Ukoliko upotrebite nevažeći indeks, generiše se izuzetak `IndexOutOfRangeException`:

```
vowels[5] = 'y';   // Greška pri izvršavanju
```

Svojstvo `Length` niza vraća broj elemenata niza. Kada je niz napravljen, ne može mu se menjati broj elemenata. Imenski prostor `System.Collection` i njemu podređeni imenski prostori obezbeđuju strukture podataka višeg nivoa, kao što su dinamički dimenzionirani nizovi i rečnici.

Izraz za inicijalizovanje niza omogućava da deklarišete i popunite niz u jednom koraku:

```
char[] vowels = new char[] {'a','e','i','o','u'};
```

ili, jednostavno:

```
char[] vowels = {'a','e','i','o','u'};
```

Svi nizovi nasleđuju klasu `System.Array`, koja definiše uobičajene metode i svojstva za sve nizove. To uključuje i svojstva instance kao što su `Length` i `Rank`, te statičke metode za:

- dinamičku izradu niza (`CreateInstance`)
- učitavanje i zadavanje elemenata nezavisno od tipa niza (`GetValue`/`SetValue`)
- pretraživanje sortiranog niza (`BinarySearch`) ili nesortiranog niza (`IndexOf`, `LastIndexOf`, `Find`, `FindIndex`, `FindLastIndex`)
- sortiranje niza (`Sort`)
- kopiranje niza (`Copy`)

Inicijalizacija elemenata niza podrazumevanim vrednostima

Pri pravljenju niza, elementima se uvek dodeljuju podrazumevane vrednosti. Podrazumevana vrednost za određeni tip podataka rezultat je dodeljivanja nule svim bitovima instance u memoriji. Na primer, razmotrimo izradu niza celih brojeva. Pošto je `int` vrednosni tip, sledeći primer koda smešta 1000 celih brojeva u jedan kontinualan blok memorije. Podrazumevana vrednost svakog elementa biće 0:

```
int[] a = new int[1000];
Console.Write (a[123]);                    // 0
```

Kada su elementi referentnog tipa, podrazumevana vrednost je `null`.

Sam niz je uvek objekat referentnog tipa, nezavisno od toga kojeg su tipa elementi. Na primer, sledeća naredba je ispravna:

```
int[] a = null;
```

Višedimenzioni nizovi

Postoje dve varijante višedimenzionih nizova: *pravougaoni* i *nazubljeni*. Pravougaoni (engl. *rectangular*) nizovi predstavljaju *n*-dimenzioni blok memorije, dok su nazubljeni (engl. *jagged*) nizovi – nizovi nizova.

Pravougaoni nizovi

U deklaracijama pravougaonih nizova, zarezi razdvajaju svaku dimenziju. Sledećom naredbom deklariše se pravougaoni dvodimenzioni niz, dimenzija 3 × 3:

```
int[,] matrix = new int [3, 3];
```

Metoda `GetLength` niza vraća dužinu zadate dimenzije (počevši od 0):

```
for (int i = 0; i < matrix.GetLength(0); i++)
  for (int j = 0; j < matrix.GetLength(1); j++)
    matrix [i, j] = i * 3 + j;
```

Evo kako se može inicijalizovati pravougaoni niz (da bi se napravio niz identičan onom u prethodnom primeru):

```
int[,] matrix = new int[,]
{
  {0,1,2},
  {3,4,5},
  {6,7,8}
};
```

(U ovakvim deklarativnim naredbama može se izostaviti kôd prikazan podebljano.)

Nazubljeni nizovi

Nazubljeni nizovi se deklarišu pomoću uzastopnih ugaonih zagrada koje predstavljaju svaku dimenziju niza. U sledećem primeru deklariše se nazubljen dvodimenzioni niz čija je najveća spoljna dimenzija 3:

```
int[][] matrix = new int[3][];
```

Unutrašnje dimenzije nisu zadate u deklaraciji jer – za razliku od pravougaonog niza – svaki unutrašnji niz može biti proizvoljne dužine. Svaki unutrašnji niz se implicitno inicijalizuje na vrednost `null` a ne na prazan niz. Svaki unutrašnji niz mora se napraviti ručno:

```
for (int i = 0; i < matrix.Length; i++)
{
  matrix[i] = new int [3]; // Izrada unutrašnjeg niza
  for (int j = 0; j < matrix[i].Length; j++)
    matrix[i][j] = i * 3 + j;
}
```

Evo kako se može inicijalizovati nazubljen niz (da bi se napravio niz identičan onom u prethodnom primeru, ali s dodatnim elementom na kraju):

```
int[][] matrix = new int[][]
{
  new int[] {0,1,2},
  new int[] {3,4,5},
  new int[] {6,7,8,9}
};
```

(U ovakvim deklarativnim naredbama može se izostaviti kôd prikazan podebljano.)

Pojednostavljeni izrazi za inicijalizovanje niza

Već smo videli kako da pojednostavimo izraze za inicijalizaciju nizova – izostavljanjem rezervisane reči `new` i deklaracije tipa:

```
char[] vowels = new char[] {'a','e','i','o','u'};
char[] vowels = {'a','e','i','o','u'};
```

Druga mogućnost je da se izostavi ime tipa iza rezervisane reči `new`, i da se prepusti kompajleru da zaključi kog je tipa dati niz. To je korisna prečica kada se nizovi prosleđuju kao argumenti. Na primer, razmotrite sledeću metodu:

```
void Foo (char[] data) { ... }
```

Evo kako ćemo ovu metodu pozvati pomoću niza koji pravimo na licu mesta:

```
Foo ( new char[] {'a','e','i','o','u'} );    // Pun oblik
Foo ( new[] {'a','e','i','o','u'} );         // Prečica
```

Ova prečica je neophodna za izradu nizova anonimnih tipova, kao što ćemo videti kasnije.

Promenljive i parametri

Promenljiva predstavlja memorijsku lokaciju čija je vrednost izmenljiva. Promenljiva može da bude *lokalna promenljiva*, *parametar* (*po vrednosti*, *po referenci* ili *izlazni*), *polje* (*instance* ili *statičko*), ili *element niza*.

Stek i hip

Stek i hip su mesta gde se smeštaju promenljive i konstante. Svako od njih ima veoma različitu semantiku tokom životnog veka.

Stek

Stek je memorijski blok u koji se smeštaju lokalne promenljive i parametri. On logički raste i sažima se kako funkcija započne odnosno završi rad. Razmotrite sledeću metodu (radi jednostavnosti, zanemaruje se provera ulaznih argumenata):

```
static int Factorial (int x)
{
  if (x == 0) return 1;
  return x * Factorial (x-1);
}
```

Ova metoda je rekurzivna, što znači da poziva samu sebe. Kad god se pokrene metoda, na steku se alocira (dodeljuje) nov `int`, a svaki put kad se izađe iz metode, `int` se dealocira.

Hip

Hip je memorijski blok u koji se smeštaju *objekti* (tj., instance referentnog tipa). Kad god se napravi nov objekat, on se alocira na hipu, i vraća se referenca na taj objekat. Tokom izvršavanja programa, hip počinje da se puni kako se stvaraju novi objekti. U izvršnom okruženju postoji *sakupljač smeća* (engl. *garbage collector*) koji povremeno dealocira objekte sa hipa, da računar ne bi ostao bez memorije. Objekat je podložan dealociranju čim ga više ne referencira ništa što je i dalje „živo".

Instance vrednosnog tipa (i reference objekata) žive na mestima gde je promenljiva deklarisana. Ukoliko je instanca deklarisana kao polje unutar objekta ili kao element niza, ta instanca živi na hipu.

Napomena

U jeziku C# ne možete eksplicitno brisati objekte kao što možete u jeziku C++. Nereferenciran objekat na kraju pokupi sakupljač smeća.

Na hip se takođe smeštaju statička polja i konstante. Za razliku od objekata na hipu (koje može pokupiti sakupljač smeća), statička polja i konstante žive sve dok živi domen aplikacije.

Izričita dodela

C# nameće politiku izričite dodele (engl. *definite assignment*). U praksi, to znači da je van `unsafe` konteksta nemoguće pristupiti neinicijalizovanoj memoriji. Izričita dodela ima tri posledice:

- Lokalnim promenljivama se mora dodeliti vrednost da bi se mogle učitati.
- Funkciji moraju biti predati argumenti pre nego što se pozove metoda (osim ukoliko su označeni kao opcioni – više informacija u odeljku „Opcioni parametri" na strani 42).
- Sve ostale promenljive (kao što su polja i elementi niza), automatski inicijalizuje izvršno okruženje.

Na primer, sledeći kôd generiše grešku pri prevođenju:

```
static void Main()
{
  int x;
  Console.WriteLine (x);      // Greška pri prevođenju
}
```

Međutim, kada bi `x` bilo polje klase kojoj funkcija pripada, kôd bi bio ispravan i rezultat bi bio `0`.

Podrazumevane vrednosti

Sve instance tipa imaju podrazumevanu vrednost. Podrazumevana vrednost unapred definisanih (ugrađenih) tipova rezultat je postavljanja svih bitova instance u memoriji na nulu, i za referentne tipove je `null`, `0` za numeričke i nabrojane tipove (`enum`), `'\0'` za tip `char`, i `false` za tip `bool`.

Podrazumevanu vrednost svakog tipa podataka saznaćete pomoću rezervisane reči `default` (u praksi, to je korisno za generičke tipove, kao što ćemo videti kasnije). Podrazumevana vrednost u namenskom vrednosnom tipu (tj., `struct`) jednaka je podrazumevanoj vrednosti svakog polja definisanog u tom namenskom tipu.

Parametri

Metoda ima niz parametara. Parametri definišu skup argumenata koji se moraju proslediti metodi. U ovom primeru, metoda `Foo` ima samo jedan parametar, po imenu `p`, tipa `int`:

```
static void Foo (int p)  // p je parametar
{
  ...
}
static void Main() { Foo (8); }  // 8 je argument
```

Pomoću modifikatora `ref` i `out` možete upravljati načinom prosleđivanja parametara:

Modifikator parametra	Prosleđuje se po	Promenljiva mora biti izričito dodeljena
Nema	Vrednosti	Ide *u* metodu
`ref`	Referenci	Ide *u* metodu
`out`	Referenci	Vraća se *iz* metode

Prosleđivanje argumenata po vrednosti

U jeziku C#, argumenti se podrazumevano prosleđuju po vrednosti, što je daleko najuobičajenije. To znači da se pravi kopija vrednosti kada se vrednost prosledi metodi:

```
static void Foo (int p)
{
  p = p + 1;                    // Inkrementira p za 1
  Console.WriteLine (p);        // Ispisuje p na ekranu
}
static void Main()
{
  int x = 8;
  Foo (x);                     // Pravi kopiju x
  Console.WriteLine (x);       // x će i dalje biti 8
}
```

Dodeljivanjem nove vrednosti parametru `p` ne menja se sadržaj promenljive `x`, pošto se `p` i `x` nalaze na različitim lokacijama u memoriji.

Pri prosleđivanju argumenta referentnog tipa po vrednosti, kopira se *referenca*, ali ne i objekat. U sledećem primeru, `Foo` vidi isti onaj objekat `StringBuilder` koji je instancirala metoda `Main`, ali ima nezavisnu *referencu* na njega. Drugim rečima, `sb` i `fooSB` su posebne promenljive koje referenciraju isti objekat `StringBuilder`:

```
static void Foo (StringBuilder fooSB)
{
  fooSB.Append ("test");Ž
  fooSB = null;
}
static void Main()
{
```

```
  StringBuilder sb = new StringBuilder();
  Foo (sb);
  Console.WriteLine (sb.ToString()); // test
}
```

Pošto je `fooSB` *kopija* reference, njenim postavljanjem na vrednost `null` ne postiže se da `sb` bude `null`. (Međutim, da je promenljiva `fooSB` deklarisana i pozvana pomoću modifikatora `ref`, `sb` *bi* postala `null`.)

Modifikator ref

Za *prosleđivanje po referenci*, C# ima modifikator parametara `ref`. U narednom primeru, `p` i `x` referenciraju iste memorijske lokacije:

```
static void Foo (ref int p)
{
  p = p + 1;
  Console.WriteLine (p);
}
static void Main()
{
  int x = 8;
  Foo (ref x);                // Prosleđuje x po referenci
  Console.WriteLine (x);      // x je sada 9
}
```

Dodeljivanje nove vrednosti parametru `p` sada menja sadržaj promenljive `x`. Obratite pažnju na to da je modifikator `ref` neophodan i pri pisanju i pri pozivanju metode. Zahvaljujući tome, potpuno je jasno šta se dešava.

Napomena

Parametar se može proslediti po referenci ili po vrednosti, nezavisno od toga da li je referentnog ili vrednosnog tipa.

Modifikator out

Argument `out` sličan je argumentu `ref`, osim što:

- ne mora imati dodeljenu vrednost pre ulaska u funkciju
- mora imati dodeljenu vrednost pre nego što se *vrati* iz funkcije

Modifikator `out` se najčešće koristi da bi se kao rezultat metode dobilo više izlaznih vrednosti.

Modifikator params

Modifikator `params` može se zadati uz poslednji parametar metode, tako da ona prihvata proizvoljan broj parametara određenog tipa. Tip parametra se mora deklarisati kao niz. Na primer:

```
static int Sum (params int[] ints)
{
  int sum = 0;
  for (int i = 0; i < ints.Length; i++) sum += ints[i];
  return sum;
}
```

Evo kako će izgledati poziv:

```
Console.WriteLine (Sum (1, 2, 3, 4));    // 10
```

Argument s modifikatorom `params` možete proslediti i kao običan niz. Prethodni poziv je semantički ekvivalentan sledećem:

```
Console.WriteLine (new int[] { 1, 2, 3, 4 } );
```

Opcioni parametri

Počevši od verzije C# 4.0, metode, konstruktori i indekseri mogu da deklarišu *opcione parametre*. Parametar je opcioni ako je u njegovoj deklaraciji navedena *podrazumevana vrednost*:

```
void Foo (int x = 23) { Console.WriteLine (x); }
```

Opcioni parametri se mogu izostaviti pri pozivanju date metode:

```
Foo();      // 23
```

Podrazumevani argument `23` u stvari se *prosleđuje* opcionom parametru `x` – kompajler upisuje vrednost `23` u prevedeni kôd na *pozivajućoj* strani. Prethodni poziv funkcije `Foo` semantički je identičan sledećem:

```
Foo (23);
```

zato što kompajler samo zamenjuje podrazumevanu vrednost opcionog parametra gde god se koristi.

Upozorenje

Dodavanje opcionog parametra javnoj metodi koja se poziva iz drugog sklopa zahteva ponovno kompajliranje oba sklopa – isto kao da su parametri obavezni.

Podrazumevana vrednost opcionog parametra mora da se navede pomoću konstantnog izraza, ili konstruktora vrednosnog tipa bez parametara. Opcioni parametri se ne mogu označiti modifikatorom `ref` ili `out`.

Obavezni parametri moraju biti *ispred* opcionih i u deklaraciji i u pozivu metode (izuzetak su `params` argumenti, koji su i dalje uvek na kraju). U sledećem primeru, eksplicitna vrednost `1` prosleđuje se promenljivoj `x`, a podrazumevana vrednost `0` – promenljivoj `y`:

```
void Foo (int x = 0, int y = 0)
{
  Console.WriteLine (x + ", " + y);
}
void Test()
{
  Foo(1);   // 1, 0
}
```

Da biste uradili suprotno (prosledili podrazumevanu vrednost promenljivoj `x` a eksplicitnu `y`), morate kombinovati opcione parametre sa *imenovanim argumentima*.

Imenovani argumenti

Umesto da identifikujete argument po položaju, možete ga identifikovati po imenu. Na primer:

```
void Foo (int x, int y)
{
  Console.WriteLine (x + ", " + y);
}
void Test()
{
  Foo (x:1, y:2); // 1, 2}
```

Imenovani argumenti mogu biti navedeni bilo kojim redosledom. Naredni pozivi funkcije `Foo` semantički su identični:

```
Foo (x:1, y:2);
Foo (y:2, x:1);
```

Možete mešati imenovane i pozicione parametre, pod uslovom da su imenovani argumenti na kraju:

```
Foo (1, y:2);
```

Imenovani argumenti su posebno korisni kada se upotrebljavaju zajedno sa opcionim parametrima. Na primer, razmotrite sledeću metodu:

```
void Bar (int a=0, int b=0, int c=0, int d=0) { ... }
```

Možemo je pozvati tako što ćemo obezbediti samo vrednost za `d`:

```
Bar (d:3);
```

To je naročito korisno pri pozivanju COM API-ja.

var – implicitno zadavanje tipa lokalnih promenljivih

Često se dešava da deklarišete i inicijalizujete promenljivu u jednom koraku. Ako je kompajler u stanju da na osnovu inicijalizacionog izraza zaključi o kom se tipu radi, možete koristiti reč `var` umesto deklaracije tipa. Na primer:

```
var x = "hello";
var y = new System.Text.StringBuilder();
var z = (float)Math.PI;
```

Ovo je potpuno ekvivalentno sledećem:

```
string x = "hello";
System.Text.StringBuilder y =
  new System.Text.StringBuilder();
float z = (float)Math.PI;
```

Zbog ove direktne ekvivalencije, promenljive kojima se tip zadaje implicitno (automatski), postaju statičke. Recimo, sledeći kôd generiše grešku pri prevođenju:

```
var x = 5;
x = "hello";   // Greška pri prevođenju; x je tipa int
```

U odeljku „Anonimni tipovi“ na strani 145, opisujemo scenario u kome je upotreba reči `var` obavezna.

Izrazi i operatori

U osnovi, *izraz* označava određenu vrednost. Najjednostavnije vrste izraza su konstante (npr. `123`) i promenljive (npr. `x`). Izrazi se mogu transformisati i kombinovati pomoću operatora. *Operatori* deluju na jedan ili više ulaznih *operanada* i kao rezultat daju nov izraz:

```
12 * 30   // * je operator; 12 i 30 su operandi.
```

Složeni izrazi se mogu formirati zato što i sam operand može da bude izraz, kao što je operand `(12 * 30)` u sledećem primeru:

```
1 + (12 * 30)
```

U jeziku C#, operatori se mogu klasifikovati kao *unarni*, *binarni* ili *ternarni* – zavisno od broja operanada s kojima rade (jedan, dva ili tri). Binarni operatori uvek koriste *infix* notaciju, gde je operator smešten *između* dva operanda.

Operatori ugrađeni u osnove jezika nazivaju se *primarni*; primer takvog operatora je operator tačka, za pozivanje metode. Izraz koji nema vrednost zove se *prazan izraz* (engl. *void expression*):

```
Console.WriteLine (1)
```

Pošto prazan izraz nema vrednost, ne može se koristiti kao operand za formiranje složenijih izraza:

```
1 + Console.WriteLine (1)   // Greška pri prevođenju
```

Izrazi dodele

U izrazu dodele (engl. *assignment expression*) koristi se operator `=` kako bi se rezultat drugog izraza dodelio promenljivoj. Na primer:

```
x = x * 5
```

Izraz dodele nije prazan izraz. On ima vrednost dodele, pa se može uvrstiti u drugi izraz. U sledećem primeru, izraz dodeljuje vrednost 2 promenljivoj `x` i vrednost 10 promenljivoj `y`:

```
y = 5 * (x = 2)
```

Ovaj stil izraza može se koristiti za inicijalizovanje više vrednosti:

```
a = b = c = d = 0
```

Složeni operatori dodele (engl. *compound assignment operators*) sintaksičke su prečice u kojima se kombinuje dodela s nekim drugim operatorom. Na primer:

```
x *= 2    // isto što i x = x * 2
x <<= 1   // isto što i x = x << 1
```

(Suptilan izuzetak od ovog pravila jesu događaji (engl. *events*), koje opisujemo kasnije: tu se operatori `+=` i `-=` posebno tretiraju i preslikavaju u pristupne metode `add` i `remove`.)

Prioritet i asocijativnost operatora

Kada izraz sadrži više operatora, *prioritet* (engl. *precedence*) i *asocijativnost* (engl. *associativity*) određuju redosled njihovog izračunavanja. Operatori višeg prioriteta izvršavaju se pre operatora nižeg prioriteta. Ako su operatori istog prioriteta, redosled izračunavanja određen je asocijativnošću operatora.

Prioritet

Izraz `1 + 2 * 3` izračunava se kao `1 + (2 * 3)` pošto * ima viši prioritet od +.

Levo asocijativni operatori

Binarni operatori (osim operatora dodele, lambda i zamene za `null`) spadaju u *levo asocijativne*; drugim rečima, izračunavaju se sleva nadesno. Na primer, izraz `8/4/2` izračunava se kao `(8/4)/2` zbog leve asocijativnosti. Naravno, uvek možete dodati sopstvene zagrade da biste promenili redosled izračunavanja.

Desno asocijativni operatori

Operatori dodele i lambda, zamene za `null` i (ternarni) uslovni operator spadaju u *desno asocijativne*; drugim rečima, izračunavaju se zdesna nalevo. Desna asocijativnost omogućava kompajliranje višestrukih dodela, kao što je `x=y=3`: funkcioniše tako što se prvo dodeljuje `3` promenljivoj `y`, a zatim se rezultat tog izraza (`3`) dodeljuje promenljivoj `x`.

Tabela operatora

U sledećoj tabeli, operatori iz jezika C# navedeni su po redosledu prioriteta. Operatori navedeni pod istim podnaslovom imaju isti prioritet. Operatore koje korisnici mogu preklapati objašnjavamo u odeljku „Preklapanje operatora“ na strani 140.

Operator	Ime operatora	Primer	Preklopiv
Primarni (najviši prioritet)			
`.`	Pristup članu	`x.y`	Ne
`->`	Pokazivač na strukturu (nebezbedno)	`x->y`	Ne
`()`	Poziv funkcije	`x()`	Ne
`[]`	Niz/indeks	`a[x]`	Preko indeksera
`++`	Post-inkrement	`x++`	Da
	Post-dekrement	`x-`	Da
`new`	Pravljenje instance	`new Foo()`	Ne

Operator	Ime operatora	Primer	Preklopiv
`stackalloc`	Nebezbedna alokacija steka	`stackalloc(10)`	Ne
`typeof`	Pribavljanje tipa identifikatora	`typeof(int)`	Ne
`checked`	Uključena provera prekoračenja u celobrojnoj aritmetici	`checked(x)`	Ne
`unchecked`	Isključena provera prekoračenja u celobrojnoj aritmetici	`unchecked(x)`	Ne
`default`	Podrazumevana vrednost	`default(char)`	Ne
`await`	Čekanje	`await mytask`	Ne
Unarni			
`sizeof`	Pribavljanje veličine strukture	`sizeof(int)`	Ne
+	Pozitivna vrednost	`+x`	Da
-	Negativna vrednost	`-x`	Da
!	Nije	`!x`	Da
~	Komplement bit po bit	`~x`	Da
++	Pre-inkrement	`++x`	Da
--	Post-inkrement	`--x`	Da
()	Eksplicitna konverzija	`(int)x`	Ne
*	Vrednost na adresi (nebezbedno)	`*x`	Ne
&	Adresa vrednosti (nebezbedno)	`&x`	Ne
Multiplikativni			
*	Množenje	`x * y`	Da
/	Deljenje	`x / y`	Da
%	Ostatak	`x % y`	Da
Aditivni			
+	Sabiranje	`x + y`	Da
-	Oduzimanje	`x - y`	Da
Pomeranja			
<<	Pomeranje ulevo	`x << 1`	Da
>>	Pomeranje udesno	`x >> 1`	Da

Operator	Ime operatora	Primer	Preklopiv
Relacioni			
<	Manje od	x < y	Da
>	Veće od	x > y	Da
<=	Manje od ili jednako	x <= y	Da
>=	Veće od ili jednako	x >= y	Da
is	Tip je ili je podklasa od	x is y	Ne
as	Konverzija tipa	x as y	Ne
Jednakosti			
==	Jednako	x == y	Da
!=	Nije jednako	x != y	Da
Logičko And			
&	I	x & y	Da
Logičko Xor			
^	Ekskluzivno ili	x ^ y	Da
Logičko Or			
|	Ili	x | y	Da
Uslovno And			
&&	Uslovno i	x && y	Preko *&*
Uslovno Or			
||	Uslovno ili	x || y	Preko |
Uslovni (ternarni)			
? :	Uslovni	isTrue ? then This : elseThis	Ne
Dodele, složene dodele i lambda (najniži prioritet)			
=	Dodela	x = y	Ne
*=	Množenje i dodela	x *= 2	Preko *
/=	Deljenje i dodela	x /= 2	Preko /
+=	Sabiranje i dodela	x += 2	Preko +

Operator	Ime operatora	Primer	Preklopiv
-=	Oduzimanje i dodela	x -= 2	Preko -
<<=	Pomeranje ulevo i dodela	x <<= 2	Preko <<
>>=	Pomeranje udesno i dodela	x >>= 2	Preko >>
&=	Operacija And i dodela	x &= 2	Preko &
^=	Operacija Exclusive-Or i dodela	x ^= 2	Preko ^
|=	Operacija Or i dodela	x |= 2	Preko |
=>	Lambda	x => x + 1	Ne

Naredbe

Funkcije se sastoje od naredaba koje se izvršavaju sekvencijalno, po redosledu kojim se pojavljuju. *Blok naredaba* je niz naredaba navedenih između vitičastih zagrada (simbola {}).

Naredbe za deklarisanje

Naredba za deklarisanje deklariše novu promenljivu, opciono je inicijalizujući pomoću nekog izraza. Naredba za deklarisanje završava se znakom tačka i zarez. Više promenljivih istog tipa možete deklarisati pomoću liste u kojoj su te promenljive razdvojene zarezima. Na primer:

```
bool rich = true, famous = false;
```

Konstanta se deklariše isto kao promenljiva, osim što joj se vrednost ne može promeniti nakon što je deklarisana, i što se mora inicijalizovati u deklaraciji (više o tome u odeljku „Konstante“ na strani 70):

```
const double c = 2.99792458E08;
```

Vidljivost lokalne promenljive

Vidljivost (engl. *scope*) lokalne promenljive ili lokalne konstante proteže se kroz ceo tekući blok. U tekućem bloku ili u njegovim ugnežđenim blokovima (ako postoje), ne možete deklarisati drugu lokalnu promenljivu istog imena.

Naredbe za izraze

Naredbe za izraze (engl. *expression statements*) jesu izrazi koji su istovremeno i ispravne naredbe. U praksi, to znači da ti izrazi nešto „rade“; drugim rečima, to su izrazi koji:

- dodeljuju vrednost promenljivoj ili modifikuju promenljivu
- instanciraju objekat
- pozivaju metodu

Izrazi koji ne rade ništa od toga, nisu ispravne naredbe:

```
string s = "foo";
s.Length;            // Neispravna naredba: ne radi ništa!
```

Kada pozovete konstruktor ili metodu koja vraća vrednost, ne morate koristiti taj rezultat. Međutim, ukoliko konstruktor ili metoda ne promene stanje, naredba je beskorisna:

```
new StringBuilder();     // Ispravno, ali beskorisno
x.Equals (y);            // Ispravno, ali beskorisno
```

Naredbe za biranje

Naredbe za biranje (engl. *selection statements*) upravljaju tokom izvršavanja programa.

Naredba if

Naredba `if` izvršava određenu naredbu ako je logički (`bool`) izraz istinit. Na primer:

```
if (5 < 2 * 3)
  Console.WriteLine ("true");           // true
```

Naredba može biti i blok koda:

```
if (5 < 2 * 3)
{
  Console.WriteLine ("true");           // true
  Console.WriteLine ("...")
}
```

Odredba else

Naredba `if` može da sadrži i odredbu `else`:

```
if (2 + 2 == 5)
  Console.WriteLine ("Does not compute");
else
  Console.WriteLine ("False");           // False
```

Unutar odredbe `else` možete da ugnezdite još jednu naredbu `if`:

```
if (2 + 2 == 5)
  Console.WriteLine ("Does not compute");
else
  if (2 + 2 == 4)
    Console.WriteLine ("Computes");      // Computes
```

Promena toka izvršavanja pomoću vitičastih zagrada

Odredba `else` se uvek primenjuje na naredbu `if` koja joj neposredno prethodi u istom bloku naredaba. Na primer:

```
if (true)
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");
```

Ovo je semantički identično sledećem kodu:

```
if (true)
{
  if (false)
    Console.WriteLine();
  else
    Console.WriteLine ("executes");
}
```

Tok izvršavanja promenićemo tako što ćemo premestiti vitičaste zagrade:

```
if (true)
{
  if (false)
```

```
    Console.WriteLine();
}
else
  Console.WriteLine ("does not execute");
```

C# nema rezervisanu reč „elseif“; međutim, pomoću sledećeg šablona postiže se isti rezultat:

```
static void TellMeWhatICanDo (int age)
{
  if (age >= 35)
    Console.WriteLine ("You can be president!");
  else if (age >= 21)
    Console.WriteLine ("You can drink!");
  else if (age >= 18)
    Console.WriteLine ("You can vote!");
  else
    Console.WriteLine ("You can wait!");
}
```

Naredba switch

Naredbe `switch` omogućavaju da se izvršavanje programa grana na osnovu mogućih vrednosti promenljive. Naredbe `switch` mogu dati čistiji kôd nego kada se koristi više naredaba `if`, pošto s naredbama `switch` izraz mora da se izračunava samo jednom. Na primer:

```
static void ShowCard (int cardNumber)
{
  switch (cardNumber)
  {
    case 13:
      Console.WriteLine ("King");
      break;
    case 12:
      Console.WriteLine ("Queen");
      break;
    case 11:
      Console.WriteLine ("Jack");
      break;
    default: // Bilo koji drugi cardNumber
```

```
        Console.WriteLine (cardNumber);
        break;
    }
}
```

Naredbom `switch` može upravljati samo izraz tipa koji se može izračunati statički, što je ograničava na tip `string`, ugrađene celobrojne tipove, `enum` tipove, i verzije tih tipova koje prihvataju null (videti „Tipovi koji prihvataju null“, na strani 135). Na kraju svake odredbe `case`, morate eksplicitno navesti – pomoću neke naredbe za skok – odakle se nastavlja izvršavanje. Evo raspoloživih opcija:

- `break` (skače na kraj naredbe `switch`)
- `goto case` *x* (skače na neki drugu odredbu `case`)
- `goto default` (skače na odredbu `default`)
- Bilo koja druga naredba za skok – konkretno, `return`, `throw`, `continue` ili `goto` *oznaka* (engl. *label*)

Kada isti kôd treba izvršiti za više od jedne vrednosti, odredbe `case` možete navesti u obliku liste:

```
switch (cardNumber)
{
  case 13:
  case 12:
  case 11:
    Console.WriteLine ("Face card");
    break;
  default:
    Console.WriteLine ("Plain card");
    break;
}
```

Ova osobina naredbe `switch` može biti ključna za dobijanje čistijeg koda od onog koji se dobija kada se koristi više naredaba `if-else`.

Naredbe za iteraciju

C# omogućava da se ponavlja izvršavanje grupe naredaba; za to služe naredbe `while`, `do-while`, `for` i `foreach`.

Petlje while i do-while

Petlje `while` ponovo izvršavaju telo koda sve dok je logički izraz istinit. Izraz se ispituje *pre* nego što se izvrši telo petlje. Na primer, sledeći kôd ispisuje 012:

```
int i = 0;
while (i < 3)
{                          // Vitičaste zagrade su ovde opcione
  Console.Write (i++);
}
```

Petlje `do-while` razlikuju se od petlji `while` samo po tome što se u njima izraz ispituje *nakon* izvršavanja bloka naredaba (što obezbeđuje da se blok uvek izvrši bar jednom). Evo prethodnog primera napisanog uz korišćenje petlje `do-while`:

```
int i = 0;
do
{
  Console.WriteLine (i++);
}
while (i < 3);
```

Petlje for

Petlje `for` su slične petljama `while` s posebnim odredbama za inicijalizaciju i iteraciju promenljive petlje. Petlja `for` sadrži sledeće tri odredbe:

```
for (odredba_za_inicijalizaciju; odredba_uslov;
     odredba_za_iteraciju)
  naredbe ili blok naredaba
```

Odredba_za_inicijalizaciju se izvršava pre početka petlje, i obično inicijalizuje jednu ili više promenljivih iteracije.

Odredba_uslov je logički izraz koji se ispituje pre svake iteracije petlje. Telo programa se izvršava sve dok je taj uslov istinit.

Odredba_za_iteraciju se izvršava *nakon* svake iteracije tela programa. Obično se koristi za ažuriranje iteracione promenljive.

Na primer, sledeći kôd ispisuje brojeve od 0 do 2:

```
for (int i = 0; i < 3; i++)
  Console.WriteLine (i);
```

Sledeći program ispisuje prvih 10 Fibonačijevih brojeva (gde je svaki broj zbir dva prethodna):

```
for (int i = 0, prevFib = 1, curFib = 1; i < 10; i++)
{
  Console.WriteLine (prevFib);
  int newFib = prevFib + curFib;
  prevFib = curFib; curFib = newFib;
}
```

Svaki od tri dela naredbe `for` može se izostaviti. Moguće je implementirati beskonačnu petlju kao što je sledeća (mada se umesto toga može koristiti `while(true)`):

```
for (;;) Console.WriteLine ("interrupt me");
```

Petlje foreach

Naredba `foreach` prolazi (iterira) kroz svaki element u nabrojivom objektu. U jeziku C# i.NET Frameworku, većina tipova koji predstavljaju skup ili listu elemenata nabrojivi su. Recimo, i niz i znakovni niz su nabrojivi. Evo primera nabrajanja znakova iz niza, od prvog do poslednjeg znaka:

```
foreach (char c in "beer")
  Console.WriteLine (c + " ");     // b e e r
```

Nabrojive objekte definišemo u odeljku „Nabrajanje i iteratori“ na strani 130.

Naredbe za skok

U jeziku C#, naredbe za skok su `break`, `continue`, `goto`, `return` i `throw`. Rezervisanu reč `throw` opisujemo u odeljku „Naredbe try i izuzeci“ na strani 122.

Naredba break

Naredba `break` prekida izvršavanje tela iteracione petlje ili naredbe `switch`:

```
int x = 0;
while (true)
{
  if (x++ > 5) break;                    // izlazak iz petlje
}
// izvršavanje se nastavlja ovde nakon prekida
...
```

Naredba continue

Naredba `continue` zanemaruje ostale naredbe u petlji i započinje sledeću iteraciju. Naredna petlja preskače parne brojeve:

```
for (int i = 0; i < 10; i++)
{
  if ((i % 2) == 0) continue;
    Console.Write (i + " ");             // 1 3 5 7 9
}
```

Naredba goto

Naredba `goto` prenosi izvršavanje na oznaku (obeleženu dvotačkom na kraju) unutar bloka naredaba. Sledeći kôd prolazi kroz brojeve od 1 do 5, imitirajući petlju `for`:

```
int i = 1;
startLoop:
if (i <= 5)
{
  Console.Write (i + " "); // 1 2 3 4 5
  i++;
  goto startLoop;
}
```

Naredba return

Naredba `return` služi za izlazak iz metode i – ukoliko metoda nije `void` – mora da vrati izraz s povratnim tipom te metode:

```
static decimal AsPercentage (decimal d)
{
  decimal p = d * 100m;
  return p;   // Povratak na pozivajuću metodu s vrednošću
}
```

Naredba `return` može da se nalazi bilo gde u metodi (osim u bloku `finally`).

Imenski prostori

Imenski prostor je domen unutar koga imena tipova moraju biti jedinstvena. Tipovi su obično organizovani u hijerarhijske imenske prostore – i da bi se izbegli konflikti pri imenovanju i da bi se imena tipova lakše pronašla. Na primer, tip `RSA` koji služi za šifrovanje javnim ključem definisan je u sledećem imenskom prostoru:

```
System.Security.Cryptography
```

Imenski prostor je sastavni deo imena tipa. Sledeći kôd poziva metodu `Create` tipa `RSA`:

```
System.Security.Cryptography.RSA rsa =
  System.Security.Cryptography.RSA.Create();
```

Napomena

Imenski prostori ne zavise od sklopova, koji su jedinice primene koda, kao što je jedna *.exe* ili *.dll* datoteka.

Imenski prostori ne utiču ni na mogućnost pristupanja članovima – `public`, `internal`, `private` itd.

Rezervisana reč `namespace` definiše imenski prostor za tipove unutar datog bloka. Na primer:

```
namespace Outer.Middle.Inner
{
  class Class1 {}
```

```
  class Class2 {}
}
```

Tačke u imenskom prostoru predstavljaju hijerarhiju ugnežđenih imenskih prostora. Sledeći kôd je semantički identičan prethodnom primeru.

```
namespace Outer
{
  namespace Middle
  {
    namespace Inner
    {
      class Class1 {}
      class Class2 {}
    }
  }
}
```

Tip možete navesti pomoću njegovog *potpuno kvalifikovanog imena*, koje uključuje sve imenske prostore – od krajnjeg spoljnog do krajnjeg unutrašnjeg. Recimo, `Class1` iz prethodnog primera možemo navesti u obliku `Outer.Middle.Inner.Class1`.

Za tipove koji nisu definisani ni u jednom imenskom prostoru kaže se da su u *globalnom imenskom prostoru*. Globalni imenski prostor takođe sadrži imenske prostore najvišeg nivoa, kao što je `Outer` u našem primeru.

Direktiva using

Direktiva `using` *uvozi* imenski prostor i pogodna je za referenciranje tipova bez izričitog navođenja njihovih potpuno kvalifikovanih imena. Recimo, `Class1` iz prethodnog primera možemo referencirati ovako:

```
using Outer.Middle.Inner;

class Test     // Test je u globalnom imenskom prostoru
{
  static void Main()
  {
```

```
    Class1 c;   // Nije potrebno potpuno kvalifikovano ime
    ...
  }
}
```

Direktiva `using` se može ugnezditi unutar samog imenskog prostora kako bi joj se ograničila vidljivost.

Pravila unutar imenskog prostora

Vidljivost imena

Imena deklarisana u spoljnim imenskim prostorima mogu se koristiti nekvalifikovana unutar unutrašnjih imenskih prostora. U ovom primeru, imenu `Class1` nije potrebno kvalifikovanje unutar prostora `Inner`:

```
namespace Outer
{
  class Class1 {}

  namespace Inner
  {
    class Class2 : Class1 {}
  }
}
```

Ukoliko želite da referencirate tip u drugoj grani date hijerarhije imenskih prostora, možete koristiti delimično kvalifikovano ime. U narednom primeru, `SalesReport` zasnivamo na `Common.ReportBase`:

```
namespace MyTradingCompany
{
  namespace Common
  {
    class ReportBase {}
  }
  namespace ManagementReporting
  {
    class SalesReport : Common.ReportBase {}
  }
}
```

Sakrivanje imena

Ako se isto ime tipa pojavljuje i u unutrašnjem i u spoljašnjem imenskom prostoru, važi ono iz unutrašnjeg. Da biste referencirali tip iz spoljašnjeg imenskog prostora, morate mu kvalifikovati ime.

NAPOMENA

Sva imena tipova konvertuju se u potpuno kvalifikovana imena tokom kompajliranja. Međukod (engl. *Intermediate Language code*, *IL code*) ne sadrži ni nekvalifikovana ni delimično kvalifikovana imena.

Ponavljanje imenskih prostora

Deklaraciju imenskog prostora možete da ponavljate sve dok nema sukoba imena tipova unutar tog prostora:

```
namespace Outer.Middle.Inner { class Class1 {} }
namespace Outer.Middle.Inner { class Class2 {} }
```

Klase mogu čak da se protežu na više izvornih datoteka i sklopova.

Kvalifikator global::

Ponekad i potpuno kvalifikovano ime tipa može da bude u sukobu sa unutrašnjim imenom. C# će koristiti potpuno kvalifikovano ime ako ispred njega napišete `global::`:

```
global::System.Text.StringBuilder sb;
```

Alijasi tipova i imenskih prostora

Uvoženje imenskog prostora može da rezultuje sukobom imena tipova. Umesto da uvezete ceo imenski prostor, možete uvesti samo određene tipove koji vam trebaju, dodeljujući svakom tipu alijas. Na primer:

```
using PropertyInfo2 = System.Reflection.PropertyInfo;
class Program { PropertyInfo2 p; }
```

Alijas se može napraviti i od celog imenskog prostora:

```
using R = System.Reflection;
class Program { R.PropertyInfo p; }
```

Klase

Među referentnim tipovima, najčešće se koristi klasa. Evo najjednostavnije moguće deklaracije klase:

```
class Foo
{
}
```

Složenija klasa može da sadrži i sledeće:

Ispred rezervisane reči `class`	*Atribute* i *modifikatore klase*. Neugnežđeni modifikatori klase su `public`, `internal`, `abstract`, `sealed`, `static`, `unsafe` i `partial`
Iza imena klase `YourClassName`	*Generičke parametre tipa, osnovnu klasu* i *interfejse*
Unutar vitičastih zagrada	*Članove klase* (to su *metode*, *svojstva*, *indekseri*, *događaji*, *polja*, *konstruktori*, *preklopljeni operatori*, *ugnežđeni tipovi* i *finalizator*)

Polja

Polje je promenljiva koja je članica klase ili strukture. Na primer:

```
class Octopus
{
  string name;
  public int Age = 10;
}
```

Polje može da ima modifikator `readonly` kako bi se sprečilo da mu se menja vrednost nakon izrade. Polje koje je namenjeno samo za čitanje može da se dodeli isključivo u njegovoj deklaraciji ili unutar konstruktora onog tipa podataka koji se nalazi u polju.

Inicijalizovanje polja je opciono. Neinicijalizovano polje ima podrazumevanu vrednost (`0`, `\0`, `null`, `false`). Inicijalizatori polja se izvršavaju pre konstruktora, redosledom kojim se pojavljuju.

Iz praktičnih razloga, više polja istog tipa možete deklarisati u obliku liste polja, razdvojenih zarezima. To je pogodno kada sva polja treba da dele iste atribute i modifikatore. Na primer:

```
static readonly int legs = 8, eyes = 2;
```

Metode

Metoda izvršava određenu operaciju definisanu kao grupa naredaba. Može da primi ulazne podatke od pozivaoca tako što joj se zadaju parametri i da vrati rezultat pozivaocu tako što joj se zada povratni tip. Povratni tip metode može da bude `void`, što znači da ne vraća nikakvu vrednost svom pozivaocu. Metoda može takođe da vraća pozivaocu izlazne podatke (rezultat) preko parametara `ref` i `out`.

Potpis (engl. *signature*) metode mora biti jedinstven unutar datog tipa. Potpis metode se sastoji od tipova njenog imena i parametara (ali ne od *imena* parametara, niti od povratnog tipa).

Preklapanje metoda

Tip može da preklapa (engl. *overloads*) svoje metode (tj. da ima više metoda istog imena), sve dok su tipovi parametara metode različiti. Recimo, sve sledeće metode mogu da postoje istovremeno u istom tipu:

```
void Foo (int x);
void Foo (double x);
void Foo (int x, float y);
void Foo (float x, int y);
```

Konstruktori instanci

Konstruktori izvršavaju inicijalizacioni kôd nad klasom ili strukturom. Konstruktor se definiše kao metoda, s tim što su ime metode i povratni tip isti kao ime sadržanog tipa:

```
public class Panda
{
```

```
  string name;                    // Definiše polje
  public Panda (string n)         // Definiše konstruktor
  {
    name = n;                     // Inicijalizacioni kôd
  }
}
...
Panda p = new Panda ("Petey");    // Poziva konstruktor
```

Klasa ili struktura mogu da preklapaju konstruktore. Jedno preklapanje može da poziva drugo, pomoću rezervisane reči `this`:

```
public class Wine
{
  public Wine (decimal price) {...}

  public Wine (decimal price, int year)
              : this (price) {...}
}
```

Kada jedan konstruktor poziva drugi, prvo se izvršava *pozvani konstruktor*.

Evo kako ćete proslediti *izraz* drugom konstruktoru:

```
public Wine (decimal price, DateTime year)
            : this (price, year.Year) {...}
```

Sam izraz ne može da koristi referencu `this`, na primer, da bi pozvao metodu instance. Međutim, može da poziva statičke metode.

Implicitni besparametarski konstruktori

Za klase, C# kompajler automatski generiše javni konstruktor bez parametara ako i samo ako ne definišete nijedan drugi konstruktor. Međutim, čim definišete bar jedan konstruktor, besparametarski konstruktor se više ne pravi automatski.

S druge strane, besparametarski konstruktor je integralni deo strukture; prema tome, ne možete definisati sopstveni. Uloga implicitnog besparametarskog konstruktora strukture jeste da inicijalizuje svako polje tako što im dodeli podrazumevane vrednosti.

Konstruktori koji nisu javni

Konstruktori ne moraju da budu javni. Takav konstruktor se najčešće koristi za izradu instanci pomoću statičke metode. Statička metoda se može koristiti za vraćanje postojećeg objekta iz rezerve (engl. *pool*) umesto izrade novog objekta, ili za vraćanje specijalizovane potklase izabrane na osnovu ulaznih argumenata.

Inicijalizatori objekata

Da bi se objekti lakše inicijalizovali, dostupna polja ili svojstva objekta mogu se inicijalizovati pomoću inicijalizatora objekata, odmah nakon konstruisanja. Na primer, razmotrite sledeću klasu:

```
public class Bunny
{
  public string Name;
  public bool LikesCarrots, LikesHumans;

  public Bunny () {}
  public Bunny (string n) { Name = n; }
}
```

Pomoću inicijalizatora objekata, objekte `Bunny` instanciraćete na sledeći način:

```
Bunny b1 = new Bunny {
                       Name="Bo",
                       LikesCarrots = true,
                       LikesHumans = false
                     };
Bunny b2 = new Bunny ("Bo") {
                              LikesCarrots = true,
                              LikesHumans = false
                            };
```

Referenca this

Referenca `this` upućuje na samu instancu. U sledećem primeru, metoda `Marry` koristi `this` za inicijalizovanje polja `mate` promenljive `partner`:

```
public class Panda
{
  public Panda Mate;

  public void Marry (Panda partner)
  {
    Mate = partner;
    partner.Mate = this;
  }
}
```

Uz to, zahvaljujući referenci `this` razlikuju se lokalna promenljiva ili parametar od polja. Recimo:

```
public class Test
{
  string name;
  public Test (string name) { this.name = name; }
}
```

Referenca `this` je upotrebljiva samo unutar nestatičkih članova klase ili strukture.

Svojstva

Svojstva (engl. *properties*) spolja izgledaju kao polja, ali sadrže unutrašnju logiku, kao metode. Na primer, gledajući sledeći kôd ne možete zaključiti da li je `CurrentPrice` polje ili svojstvo:

```
Stock msft = new Stock();
msft.CurrentPrice = 30;

msft.CurrentPrice -= 3;
Console.WriteLine (msft.CurrentPrice);
```

Svojstvo se deklariše kao polje, ali mu se dodaje blok `get/set`. Evo kako da implementirate `CurrentPrice` kao svojstvo:

```
public class Stock
{
  decimal currentPrice; // Privatno pozadinsko polje
```

```
    public decimal CurrentPrice // Javno svojstvo
    {
      get { return currentPrice; }
      set { currentPrice = value; }
    }
}
```

`get` i `set` su metode za pristupanje svojstvima (engl. *property accessors*). `get` se izvršava pri učitavanju vrednosti svojstva i mora da vrati vrednost istog tipa kao svojstvo. `set` se izvršava pri dodeljivanju vrednosti svojstva. `set` ima implicitan parametar po imenu `value`, istog tipa kao svojstvo, koji obično dodeljujete nekom privatnom polju (u ovom slučaju, `currentPrice`).

Mada se svojstvima pristupa na isti način kao poljima, ona se razlikuju po tome što onome ko ih implementira daju potpunu kontrolu nad učitavanjem i zadavanjem njihove vrednosti. Zahvaljujući tome, osoba koja implementira svojstva može da izabere potreban način internog predstavljanja a da pritom ne otkriva interne detalje korisniku svojstva. U ovom primeru, metoda `set` bi generisala izuzetak kada bi `value` bila izvan važećeg opsega vrednosti.

NAPOMENA

Kroz celu knjigu koristimo javna polja da bi primeri bili jednostavniji. U stvarnoj aplikaciji, najčešće biste davali prednost javnim svojstvima nad javnim poljima kako biste podstakli kapsuliranje.

Svojstvo može samo da se čita ukoliko je zadata samo metoda `get`, a samo da se upisuje ako je zadata samo metoda `set`. Svojstva koja se mogu samo upisivati koriste se retko. Svojstvo obično ima namensko pozadinsko polje za smeštaj pripadajućih podataka. Međutim, ne mora uvek da ga ima – umesto toga, može da vraća vrednost izračunatu na osnovu drugih podataka.

Automatska svojstva

Najčešća implementacija svojstva je metoda `get` i/ili `set` koja samo čita iz privatnog polja istog tipa kao svojstvo, odnosno upisuje u njega. Deklaracija *automatskog svojstva* nalaže kompajleru da omogući tu implementaciju. Evo kako ćemo redeklarisati prvi primer iz ovog odeljka:

```
public class Stock
{
  public decimal CurrentPrice { get; set; }
}
```

Kompajler automatski generiše privatno pozadinsko polje sa imenom koje je sam odredio i koje se ne može referencirati. Pristupna metoda `set` može se označiti kao `private` ako želite da drugi tipovi mogu samo da čitaju vrednost svojstva.

Pristupanje pomoću metoda get i set

Metode `get` i `set` mogu imati različite nivoe pristupa. Uobičajen slučaj je imati svojstvo tipa `public` s modifikatorom pristupa `internal` ili `private` uz `set`:

```
private decimal x;
public decimal X
{
  get         { return x; }
  private set { x = Math.Round (value, 2); }
}
```

Zapazite da sâmo svojstvo deklarišete sa slobodnijim nivoom pristupa (u ovom slučaju, `public`), a modifikator dodajete onoj pristupnoj metodi koju želite da učinite manje pristupačnom.

Indekseri

Indekseri obezbeđuju prirodnu sintaksu za pojedinačno pristupanje elementima klase ili strukture koja kapsulira listu ili rečnik vrednosti. Indekseri su slični svojstvima, ali im se pristupa preko argumenta indeksa umesto preko imena svojstva. Klasa `string` ima indekser koji omogućava da pristupite svakoj od njenih `char` vrednosti preko `int` indeksa:

```
string s = "hello";
Console.WriteLine (s[0]); // 'h'
Console.WriteLine (s[3]); // 'l'
```

Sintaksa za upotrebu indeksera ista je kao ona za nizove, s tim što argument(i) indeksa može biti bilo kog tipa.

Implementiranje indeksera

Da biste napisali indekser, definišite svojstvo po imenu `this`, zadajući argumente u uglastim zagradama. Na primer:

```
class Sentence
{
  string[] words = "The quick brown fox".Split();

  public string this [int wordNum]       // indekser
  {
    get { return words [wordNum]; }
    set { words [wordNum] = value; }
  }
}
```

Evo kako možemo koristiti ovaj indekser:

```
Sentence s = new Sentence();
Console.WriteLine (s[3]);       // fox
s[3] = "kangaroo";
Console.WriteLine (s[3]);       // kangaroo
```

Jedan tip može da deklariše više indeksera, od kojih svaki ima parametre različitog tipa. Indekser takođe može da ima više od jednog parametra:

```
public string this [int arg1, string arg2]
{
  get { ... } set { ... }
}
```

Ako izostavite reč `set`, indekser postaje dostupan samo za čitanje.

Konstante

Konstanta je statičko polje čija se vrednost nikada ne može promeniti. Konstanta se izračunava statički u vreme prevođenja i kompajler doslovno zamenjuje njenu vrednost gde god je nađe (slično kao makro u jeziku C++). Konstanta može da bude bilo kog od ugrađenih numeričkih tipova – `bool`, `char`, `string` ili neki nabrojivi tip.

Konstanta se deklariše pomoću rezervisane reči `const` i mora da se inicijalizuje nekom vrednošću. Na primer:

```
public class Test
{
  public const string Message = "Hello World";
}
```

Konstanta je mnogo restriktivnija od polja tipa `static readonly` – i po tipovima koje možete koristiti i po semantici inicijalizacije polja. Konstanta se razlikuje od polja `static readonly` i po tome što se konstanta izračunava u vreme prevođenja. Konstante se mogu deklarisati i tako da budu lokalne za određenu metodu:

```
static void Main()
{
  const double twoPI = 2 * System.Math.PI;
  ...
}
```

Statički konstruktori

Statički konstruktor se izvršava jednom po *tipu* umesto jednom po svakoj *instanci* tipa. Jedan tip može da definiše samo jedan statički konstruktor, koji ne sme imati parametre i mora imati isto ime kao tip:

```
class Test
{
  static Test() { Console.Write ("Type Initialized"); }
}
```

Izvršno okruženje automatski poziva statički konstruktor neposredno pre upotrebe tipa. To se dešava u dva slučaja: instanciranje tipa i pristupanje statičkom članu u tipu.

Upozorenje

Ako statički konstruktor generiše neobrađen izuzetak, taj tip postaje *neupotrebljiv* tokom života aplikacije.

Inicijalizatori statičkih polja izvršavaju se neposredno *pre* nego što se pozove statički konstruktor. Ako tip nema statički konstruktor, inicijalizatori polja će se izvršiti neposredno pre tipa koji se koristi – ili u *nekom ranijem trenutku* po nahođenju izvršnog okruženja. (To znači da postojanje statičkog konstruktora može dovesti do toga da se inicijalizatori polja izvrše kasnije u programu nego što bi inače.)

Statičke klase

Klasa može biti označena kao `static`, što znači da mora sadržati samo statičke članove i ne može imati potklase. Klase `System.Console` i `System.Math` dobri su primeri statičkih klasa.

Finalizatori

Finalizatori su metode koje se mogu deklarisati samo unutar klasa i izvršavaju se pre nego što sakupljač smeća ponovo zatraži memoriju za nereferenciran objekat. Sintaksu finalizatora čini ime klase kojem prethodi simbol ~:

```
class Class1
{
  ~Class1() { ... }
}
```

C# prevodi finalizator u metodu koja redefiniše (engl. *overrides*) metodu `Finalize` u klasi `object`. Sakupljanje smeća i finalizatori detaljno su opisani u poglavlju 12 knjige *C# 5.0 za programere*.

Parcijalni tipovi i metode

Parcijalni tipovi omogućavaju podelu definicije tipa – obično u više datoteka. Uobičajen scenario je da se parcijalna klasa automatski generiše iz nekog drugog izvora (npr., šablona iz razvojnog okruženja Visual Studio), i da se ta klasa nadogradi metodama dodatim ručno. Na primer:

```
// PaymentFormGen.cs - automatski generisano
partial class PaymentForm { ... }

// PaymentForm.cs - ručno napisano
partial class PaymentForm { ... }
```

Svaki deo definicije tipa mora da ima deklaraciju `partial`.

Pojedini delovi definicije ne smeju da imaju sukobljene članove. Recimo, ne sme se ponoviti konstruktor sa istim parametrima. Parcijalne tipove potpuno razrešava kompajler, što znači da svaki deo definicije mora biti dostupan u vreme prevođenja i mora se nalaziti u istom sklopu.

Osnovna klasa mora biti zadata za jedan deo ili za sve delove. Osim toga, svaki deo definicije može nezavisno zadati interfejse koje treba implementirati. Osnovne klase i interfejse razmatramo detaljnije u odeljcima „Nasleđivanje“ na strani 73 i „Interfejsi“ na strani 88.

Parcijalne metode

Parcijalni tip može da sadrži parcijalne metode. One omogućavaju da automatski generisan parcijalni tip obezbedi prilagodljive šablone (engl. *hooks*) za ručno dopunjavanje. Na primer:

```
partial class PaymentForm  // U automatski generisanoj datoteci
{
  partial void ValidatePayment (decimal amount);
}

partial class PaymentForm  // U datoteci napisanoj ručno
{
  partial void ValidatePayment (decimal amount)
  {
    if (amount > 100) Console.Write ("Expensive!");
  }
}
```

Parcijalna metoda se sastoji od dva dela: *definicije* i *implementacije*. Definiciju obično piše generator koda, dok se implementacija najčešće dopunjava ručno. Ako nije obezbeđena implementacija, kompajlira se definicija parcijalne metode (i kôd koji je poziva). To daje autogenerisanom kodu slobodu da obezbedi samo šablon, a da ne treba da se brine o nepotrebnom narastanju koda (engl. *code bloat*). Parcijalne metode moraju biti `void` i implicitno su `private`.

Nasleđivanje

Klasa može da *nasleđuje* drugu klasu kako bi se originalna klasa proširila ili prilagodila. Nasleđivanje (engl. *inheriting*) klase omogućava da ponovo koristite funkcionalnost te klase umesto da je pišete iz početka. Klasa može da nasleđuje samo jednu klasu, ali nju mogu da naslede mnoge druge klase i da se tako formira hijerarhija klasa. U ovom primeru počinjemo tako što definišemo klasu po imenu `Asset`:

```
public class Asset { public string Name; }
```

Nakon toga, definišemo klase `Stock` i `House`, koje će nasleđivati klasu `Asset`. Klase `Stock` i `House` dobijaju sve što ima `Asset`, plus – eventualno – dodatne članove koje same definišu:

```
public class Stock : Asset   // nasleđuje od Asset
{
 public long SharesOwned;
}

public class House : Asset   // nasleđuje od Asset
{
 public decimal Mortgage;
}
```

Evo kako možemo koristiti ove klase:

```
Stock msft = new Stock { Name="MSFT",
                         SharesOwned=1000 };

Console.WriteLine (msft.Name);         // MSFT
```

```
Console.WriteLine (msft.SharesOwned);  // 1000

House mansion = new House { Name="Mansion",
                            Mortgage=250000 };

Console.WriteLine (mansion.Name);      // Mansion
Console.WriteLine (mansion.Mortgage);  // 250000
```

Potklase, `Stock` i `House`, nasleđuju svojstvo `Name` od *osnovne klase* (engl. *base class*), `Asset`.

Potklase se zovu još i *izvedene klase* (engl. *derived classes*).

Polimorfizam

Reference su polimorfne. To znači da promenljiva tipa *x* može referencirati objekat koji je potklasa od *x*. Na primer, razmotrite sledeću metodu:

```
public static void Display (Asset asset)
{
  System.Console.WriteLine (asset.Name);
}
```

Ova metoda može da prikaže i `Stock` i `House`, pošto su obe `Asset`. Polimorfizam funkcioniše na osnovu toga što potklase (`Stock` i `House`) imaju sve osobine svoje osnovne klase (`Asset`). Međutim, obrnuto ne važi. Kada bi metoda `Display` bila prepravljena da prihvati `House`, ne biste mogli da joj prosledite `Asset`.

Konvertovanje referenci

Referenca na objekat može da bude:

- implicitno *konvertovana naviše* (engl. *upcast*) u referencu osnovne klase
- eksplicitno *konvertovana naniže* (engl. *downcast*) u referencu potklase

Konvertovanjem naviše i naniže između kompatibilnih referentnih tipova obavljaju se *konverzije referenci*: pravi se nova referenca koja pokazuje na *isti* objekat. Konvertovanje naviše uvek uspeva; konvertovanje naniže uspeva samo ako je objekat odgovarajućeg tipa.

Konvertovanje naviše

Pomoću operacije konvertovanja naviše pravi se referenca na osnovnu klase od reference potklase. Na primer:

```
Stock msft = new Stock();    // Iz prethodnog primera
Asset a = msft;              // Konvertovanje naviše
```

Nakon konvertovanja naviše, promenljiva `a` i dalje referencira isti objekat `Stock` kao promenljiva `msft`. Sâm referencirani objekat se ne menja niti se konvertuje:

```
Console.WriteLine (a == msft);      // True
```

Mada `a` i `msft` referenciraju isti objekat, `a` ima restriktivniji pogled na njega:

```
Console.WriteLine (a.Name);         // OK
Console.WriteLine (a.SharesOwned);  // Greška
```

Poslednji red koda generiše grešku pri prevođenju zato što je promenljiva `a` tipa `Asset`, bez obzira na to što referencira objekat tipa `Stock`. Da biste došli do njenog polja `SharesOwned`, morate `Asset` konvertovati naniže u `Stock`.

Konvertovanje naniže

Pomoću operacije konvertovanja naniže pravi se referenca potklase od reference osnovne klase. Na primer:

```
Stock msft = new Stock();
Asset a = msft;                      // Konvertovanje naviše
Stock s = (Stock)a;                  // Konvertovanje naniže
Console.WriteLine (s.SharesOwned);   // <Nema greške>
Console.WriteLine (s == a);          // True
Console.WriteLine (s == msft);       // True
```

Kao i u slučaju konvertovanja naviše, konvertuje se samo referenca a ne i pripadajući objekat. Za konvertovanje naniže potrebna je ekspli-

citna konverzija zato što, tokom izvršavanja, ta operacija se može završiti i neuspehom:

```
House h = new House();
Asset a = h;           // Konvertovanje naviše uvek uspeva
Stock s = (Stock)a;    // Konvertovanje naniže nije uspelo:
                       // a nije Stock
```

Ako konvertovanje naniže ne uspe, generiše se izuzetak `InvalidCast`. To je primer provere tipa tokom izvršavanja, engl. *runtime type checking* (videti „Proveravanje tipova – statičko i u vreme izvršavanja“ na strani 83).

Operator as

Operator `as` obavlja konvertovanje naniže u kome se, u slučaju neuspeha, dobija `null` (a ne izuzetak):

```
Asset a = new Asset();
Stock s = a as Stock;   // s je null; ne generiše se izuzetak
```

Ovo je korisno kada ćete naknadno ispitivati da li je rezultat `null`:

```
if (s != null) Console.WriteLine (s.SharesOwned);
```

Operator `as` ne može da obavlja *namenske konverzije* (videti odeljak „Preklapanje operatora“ na strani 140) niti numeričke konverzije.

Operator is

Operator `is` ispituje da li će konverzija reference uspeti; drugim rečima, da li se objekat izvodi iz zadate klase (ili implementira interfejs). Često se koristi za ispitivanje pre konvertovanja naniže:

```
if (a is Stock) Console.Write (((Stock)a).SharesOwned);
```

Operator `is` ne razmatra namenske i numeričke konverzije, ali razmatra konverzije raspakivanja, engl. *unboxing conversions* (videti „Tip object“ na strani 81).

Virtuelne funkcije članice

Funkcija označena rezervisanom rečju `virtual` može biti *redefinisana* u potklasama koje treba da obezbede specijalizovanu implementaciju. Metode, svojstva, indekseri i događaji mogu se deklarisati kao virtuelni:

```
public class Asset
{
  public string Name;
  public virtual decimal Liability { get { return 0; } }
}
```

Potklasa redefiniše virtuelnu metodu primenom modifikatora `override`:

```
public class House : Asset
{
  public decimal Mortgage;


  public override decimal Liability
  { get { return Mortgage; } }
}
```

Svojstvo `Liability` objekta `Asset` podrazumevano ima vrednost `0`. Klasa `Stock` ne mora da to izričito zadaje. Međutim, `House` zadaje svojstvo `Liability` tako da vraća vrednost `Mortgage`:

```
House mansion = new House { Name="Mansion",
                            Mortgage=250000 };
Asset a = mansion;
Console.WriteLine (mansion.Liability);   // 250000
Console.WriteLine (a.Liability);         // 250000
```

Potpisi, povratni tipovi i dostupnost virtuelnih i redefinisanih metoda moraju biti identični. Redefinisana metoda može da pozove implementaciju svoje osnovne klase pomoću rezervisane reči `base` (videti „Rezervisana reč base“ na strani 79).

Apstraktne klase i apstraktni članovi

Klasa deklarisana kao *apstraktna* nikad ne može da se instancira – mogu da se instanciraju samo njene *potklase*.

Apstraktne klase mogu da definišu *apstraktne članove*, koji su slični virtuelnim članovima, s tim što ne obezbeđuju podrazumevanu implementaciju. Tu implementaciju mora da obezbedi potklasa, osim ako je i ona deklarisana kao apstraktna:

```
public abstract class Asset
{
  // Obratite pažnju na praznu implementaciju
  public abstract decimal NetValue { get; }
}
```

Potklase redefinišu apstraktne članove isto kao da su virtuelni.

Sakrivanje nasleđenih članova

Osnovna klasa i potklasa mogu da definišu iste članove. Na primer:

```
public class A       { public int Counter = 1; }
public class B : A   { public int Counter = 2; }
```

Kaže se da polje `Counter` u klasi `B` *krije* polje `Counter` u klasi `A`. To se obično dešava slučajno, kada se osnovnom tipu doda član *nakon* što je identičan član dodat podtipu. Zbog toga kompajler generiše upozorenje, a zatim rešava dvosmislenost na sledeći način:

- Reference na **`A`** (u vreme prevođenja) vezuje za **`A.Counter`**.
- Reference na **`B`** (u vreme prevođenja) vezuje za **`B.Counter`**.

Ponekad želite namerno da sakrijete član, pa možete primeniti modifikator `new` na taj član u potklasi. Modifikator `new` samo sprečava da se pojavi upozorenje kompajlera:

```
public class A     { public int Counter = 1; }
public class B : A { public new int Counter = 2; }
```

Modifikator `new` saopštava vašu nameru kompajleru – i drugim programerima – da duplirani član nije slučajnost.

Zatvaranje funkcija i klasa

Redefinisana funkcija članica može da *zatvori* (engl. *seal*) svoju implementaciju pomoću rezervisane reči `sealed` kako bi sprečila da je potklase redefinišu. U našem ranijem primeru virtuelne funkcije članice, mogli smo da zatvorimo implementaciju svojstva `Liability` potklase `House`, kako bismo sprečili da klasa koja se izvodi iz `House` redefiniše `Liability`. Evo kako:

```
public sealed override decimal Liability { get { ... } }
```

Možete zatvoriti i samu klasu, čime će se implicitno zatvoriti sve virtuelne funkcije, tako što ćete modifikator `sealed` primeniti na samu klasu.

Rezervisana reč base

Rezervisana reč `base` slična je rezervisanoj reči `this`. Ona ima dve osnovne svrhe: pristupanje redefinisanoj funkciji članici iz potklase, i pozivanje konstruktora osnovne klase (videti naredni odeljak).

U ovom primeru, potklasa `House` koristi rezervisanu reč `base` da bi pristupila implementaciji svojstva `Liability` klase `Asset`:

```
public class House : Asset
{
  ...
  public override decimal Liability
  {
    get { return base.Liability + Mortgage; }
  }
}
```

Pomoću rezervisane reči `base`, svojstvu `Liability` klase `Asset` pristupamo *nevirtuelno*. To znači da ćemo uvek pristupati `Asset`-ovoj verziji ovog svojstva – bez obzira na stvarni tip instance u vreme izvršavanja.

Isti pristup važi i kada je svojstvo `Liability` *skriveno* a ne *redefinisano*. (Skrivenim članovima možete pristupiti i tako što ćete ih eksplicitno konvertovati u osnovnu klasu pre pozivanja funkcije.)

Konstruktori i nasleđivanje

Potklasa mora da deklariše sopstvene konstruktore. Recimo, ako definišemo `Baseclass` i `Subclass` ovako:

```
public class Baseclass
{
  public int X;
  public Baseclass () { }
  public Baseclass (int x) { this.X = x; }
```

```
}
public class Subclass : Baseclass { }
```

sledeći kôd nije ispravan:

```
Subclass s = new Subclass (123);
```

`Subclass` mora da „redefiniše“ sve konstruktore koje želi da izloži – ako takvih ima. Kada to uradi, može da poziva sve konstruktore osnovne klase pomoću rezervisane reči `base`:

```
public class Subclass : Baseclass
{
  public Subclass (int x) : base (x) { ... }
}
```

Rezervisana reč `base` funkcioniše slično kao rezervisana reč `this`, s tim što poziva konstruktor osnovne klase. Konstruktori osnovne klase uvek se izvršavaju prvi; zahvaljujući tome, *osnovna* inicijalizacija se odvija pre *specijalizovane*.

Ako se iz konstruktora u potklasi izostavi rezervisana reč `base`, automatski se poziva *besparametarski* konstruktor osnovne klase (ukoliko osnovna klasa nema dostupan besparametarski konstruktor, kompajler generiše grešku).

Konstruktor i redosled inicijalizovanja polja

Nakon što se objekat instancira, inicijalizacija se odvija sledećim redosledom:

1. Od potklase ka osnovnoj klasi:
   a. Inicijalizuju se polja.
   b. Izračunavaju se argumenti poziva konstruktora osnovne klase.
2. Od osnovne klase ka potklasi:
   a. Izvršavaju se tela konstruktora.

Preklapanje i razrešavanje

Nasleđivanje ima zanimljiv uticaj na preklapanje metoda. Razmotrite sledeća dva preklapanja:

```
static void Foo (Asset a) { }
static void Foo (House h) { }
```

Kada se pozove preklapanje, prednost ima najspecifičniji tip:

```
House h = new House (...);
Foo(h);                          // Poziva Foo(House)
```

Koje će se preklapanje pozvati, određuje se statički (u vreme prevođenja) a ne u vreme izvršavanja. Sledeći kôd poziva `Foo(Asset)`, mada je `a` tipa `House` u vreme izvršavanja:

```
Asset a = new House (...);
Foo(a);                          // Poziva Foo(Asset)
```

NAPOMENA

Ako konvertujete Asset u `dynamic` (videti „Dinamičko povezivanje“ na strani 171), odluka o tome koje preklapanje pozvati odlaže se do vremena izvršavanja i zasniva se na stvarnom tipu objekta.

Tip object

`object` (`System.Object`) predstavlja vrhovnu osnovnu klasu za sve tipove. Svaki tip se može implicitno konvertovati naviše u tip `object`.

Da bismo ilustrovali kakva je korist od toga, razmotrimo *stek* opšte namene. Stek je struktura podataka zasnovana na principu „poslednji unutra, prvi napolje“ (engl.„*last in, first out*“, LIFO). Stek ima dve operacije: *stavljanje* (engl. *push*) objekta na stek, i *skidanje* (engl. *pop*) objekta sa steka. Evo jednostavne implementacije koja može da sadrži do 10 objekata:

```
public class Stack
{
  int position;
  object[] data = new object[10];
  public void Push (object o) { data[position++] = o; }
  public object Pop() { return data[--position]; }
}
```

Pošto `Stack` radi s tipom `object`, možemo koristiti komande `Push` i `Pop` da bismo instance bilo kog tipa stavljali na stek, odnosno da bismo ih skidali sa steka:

```
Stack stack = new Stack();
stack.Push ("sausage");
string s = (string) stack.Pop();      // Konvertovanje naniže
Console.WriteLine (s);                // sausage
```

`object` je referentni tip, zahvaljujući tome što je klasa. Uprkos tome, vrednosni tipovi, kao što je `int`, takođe mogu da se konvertuju u tip `object` ili iz njega. Da bi to omogućio, CLR mora da obavi specijalan posao kako bi se premostile razlike između vrednosnih i referentnih tipova. Taj proces se zove *pakovanje* (engl. *boxing*) i *raspakivanje* (engl. *unboxing*).

Napomena

U odeljku „Generičke komponente“ na strani 95, opisujemo kako poboljšati klasu `Stack` da bi bolje radila sa stekovima koji sadrže elemente istog tipa.

Pakovanje i raspakivanje

Pakovanje je operacija konvertovanja instance vrednosnog tipa u instancu referentnog tipa. Referentni tip može biti klasa `object` ili interfejs (videti „Interfejsi“ na strani 88). U ovom primeru, pakujemo `int` u objekat:

```
int x = 9;
object obj = x;          // Pakovanje tipa int
```

Raspakivanje obrće operaciju, tako što konvertuje objekat nazad u originalni vrednosni tip:

```
int y = (int)obj;        // Raspakivanje tipa int
```

Za raspakivanje je neophodna eksplicitna konverzija. Izvršno okruženje proverava da li zadati vrednosni tip odgovara stvarnom tipu objekta, i – ako provera ne uspe – generiše izuzetak `InvalidCastException`.

Na primer, sledeći kôd generiše izuzetak pošto tip `long` ne odgovara tipu `int`:

```
object obj = 9;          // zaključuje se da je 9 tipa int
long x = (long) obj;     // InvalidCastException
```

Međutim, sledeći kôd se izvršava uspešno:

```
object obj = 9;
long x = (int) obj;
```

Kao i ovaj:

```
object obj = 3.5;        // zaključuje se da je 3.5 tipa double
int x = (int) (double) obj;   // x je sada 3
```

U poslednjem primeru, `(double)` obavlja *raspakivanje* a zatim `(int)` obavlja *numeričku konverziju*.

Pakovanjem se instanca vrednosnog tipa *kopira* u novi objekat, dok se raspakivanjem sadržaj objekta *kopira* nazad u instancu vrednosnog tipa:

```
int i = 3;
object boxed = i;
i = 5;
Console.WriteLine (boxed);   // 3
```

Proveravanje tipova – statičko i u vreme izvršavanja

C# proverava tipove i statički (u vreme prevođenja) i u vreme izvršavanja.

Statička provera tipa omogućava kompajleru da utvrdi ispravnost programa a da ga ne izvrši. Sledeći kôd neće uspeti zato što kompajler nameće statičku proveru tipa:

```
int x = "5";
```

U vreme izvršavanja, tipove proverava CLR kada obavite konverziju naniže preko konverzije reference, tj. raspakivanja:

```
object y = "5";
int z = (int) y;     // Greška pri izvršavanju, neuspela
                     // konverzija naniže
```

Provera tipova u vreme izvršavanja moguća je zato što svaki objekat na hipu interno čuva kratak opis tipa. Taj opis se učitava tako što se pozove metoda `GetType` za objekat.

Metoda GetType i operator typeof

U jeziku C#, svi tipovi su u vreme izvršavanja predstavljeni instancom `System.Type`. Dva su osnovna načina za dobijanje objekta `System.Type`: pozivanje metode `GetType` za instancu, ili primena operatora `typeof` na ime tipa. `GetType` se izračunava u vreme izvršavanja; `typeof` se izračunava statički, u vreme prevođenja.

`System.Type` ima svojstva za elemente kao što su ime tipa, sklop, osnovni tip itd. Na primer:

```
int x = 3;

Console.Write (x.GetType().Name);                    // Int32
Console.Write (typeof(int).Name);                    // Int32
Console.Write (x.GetType().FullName);          // System.Int32
Console.Write (x.GetType() == typeof(int));          // true
```

`System.Type` takođe ima metode koje služe kao prolaz ka modelu refleksije s kojim radi izvršno okruženje. Detaljnije informacije date su u poglavlju 19 knjige *C# 5.0 za programere*.

Lista članova objekta

Evo svih članova klase `object`:

```
public extern Type GetType();
public virtual bool Equals (object obj);
public static bool Equals (object objA, object objB);
public static bool ReferenceEquals (object objA,
                                    object objB);
public virtual int GetHashCode();
public virtual string ToString();
protected override void Finalize();
protected extern object MemberwiseClone();
```

Metode Equals, ReferenceEquals i GetHashCode

Metoda `Equals` klase `object` slična je operatoru `==`, osim što je `Equals` virtuelna, dok je `==` statički. Sledeći primer ilustruje tu razliku:

```
object x = 3;
object y = 3;
Console.WriteLine (x == y);         // false
Console.WriteLine (x.Equals (y));   // true
```

Pošto su `x` i `y` konvertovani u tip `object`, kompajler se statički vezuje za operator `==` klase `object`, koji koristi semantiku *referentnog tipa* da bi uporedio dve instance. (A budući da su `x` i `y` spakovani, oni su predstavljeni na različitim memorijskim lokacijama, pa nisu jednaki.) Međutim, virtuelna metoda `Equals` potčinjava se metodi `Equals` za tip `Int32`, u kojoj se za poređenje dve vrednosti koristi semantika *vrednosnog tipa*.

Statička metoda `object.Equals` jednostavno poziva virtuelnu metodu `Equals` za prvi argument – nakon što proveri da argumenti nisu `null`:

```
object x = null, y = 3;
bool error = x.Equals (y);       // Greška pri izvršavanju!
bool ok = object.Equals (x, y);  // OK (false)
```

Metoda `ReferenceEquals` nameće poređenje jednakosti referentnog tipa (to je ponekad korisno za referentne tipove gde je operator `==` preklopljen da radi drugačije).

Metoda `GetHashCode` pravi heš kôd pogodan za upotrebu s rečnicima zasnovanim na heš tabelama, konkretno `System.Collections.Generic.Dictionary` i `System.Collections.Hashtable`.

Da biste prilagodili semantiku za ispitivanje jednakosti nekog tipa, morate redefinisati barem metode `Equals` i `GetHashCode`. Uz to, obično biste i preklopili operatore `==` i `!=`. Primer kako da uradite i jedno i drugo, dat je u odeljku „Preklapanje operatora“ na strani 140.

Metoda ToString

Metoda `ToString` vraća podrazumevan tekstualni prikaz instance tipa. Ovu metodu redefinišu svi ugrađeni tipovi:

```
string s1 = 1.ToString();      // s1 je "1"
string s2 = true.ToString();   // s2 je "True"
```

Metodu `ToString` za namenske tipove možete redefinisati ovako:

```
public override string ToString() { return "Foo"; }
```

Strukture

Struktura je slična klasi, uz sledeće ključne razlike:

- Struktura je vrednosni tip, dok je klasa referentni.
- Struktura ne podržava nasleđivanje (osim implicitnog izvođenja iz tipa `object`, tj., preciznije, iz tipa `System.Value`).

Struktura može da ima sve članove kao i klasa, osim besparametarskog konstruktora, finalizatora i virtuelnih članova.

Struktura se koristi umesto klase kada je poželjna semantika vrednosnog tipa. Dobri primeri su numerički tipovi, gde je prirodnije da operacija dodeljivanja kopira vrednost nego referencu. Pošto je struktura vrednosni tip, nije potrebno instanciranje objekta na hipu za svaku instancu; to može da bude korisna ušteda pri izradi velikog broja instanci tipa. Recimo, za izradu niza vrednosnog tipa potrebna je samo jedna operacija dodele na hipu.

Semantika konstruisanja strukture

Evo semantike konstruisanja strukture:

- Postoji besparametarski konstruktor koji ne možete redefinisati implicitno. Zbog toga se svi bitovi svih polja strukture postavljaju na nulu.
- Kada definišete parametarski kontruktor strukture, morate eksplicitno dodeliti vrednosti svakom polju.
- U strukturi ne možete imati inicijalizatore polja.

Modifikatori pristupa

Radi podsticanja kapsulacije, može se ograničiti dostupnost tipa ili člana tipa drugim tipovima i sklopovima tako što će se deklaraciji dodati jedan od pet modifikatora pristupa:

`public`
: Potpuno dostupan. Implicitna dostupnost za članove nabrajanja ili interfejsa.

`internal`
: Dostupan samo unutar sadržanog sklopa ili prijateljskih sklopova. Podrazumevana dostupnost za neugnežđene tipove.

`private`
: Dostupan samo unutar sadržanog tipa. Podrazumevana dostupnost za članove klase ili strukture.

`protected`
: Dostupan samo unutar sadržanog tipa ili potklasa.

`protected internal`
: *Unija* dostupnosti `protected` i `internal` (omogućava veću dostupnost od pojedinačnih modifikatora `protected` ili `internal`, zahvaljujući tome što je član dostupniji na dva načina)

U sledećem primeru, klasi `Class2` može se pristupiti izvan njenog sklopa, a klasi `Class1` ne može:

```
class Class1 {}          // Class1 je interna (podrazumevano)
public class Class2 {}
```

`ClassB` izlaže polje `x` drugim tipovima u istom sklopu, a `ClassA` ne:

```
class ClassA { int x;          } // x je privatna
class ClassB { internal int x; }
```

Pri redefinisanju osnovne funkcije klase, dostupnost mora biti ista i za redefinisanu funkciju. Kompajler sprečava svaku nekonzistentnu upotrebu modifikatora pristupa – na primer, sama potklasa može biti manje dostupna od osnovne klase, ali ne i dostupnija od nje.

Prijateljski sklopovi

U naprednim scenarijima, `internal` članove možete izložiti drugim *prijateljskim* sklopovima tako što ćete dodati atribut sklopa `System.Runtime.CompilerServices.InternalsVisibleTo`, zadajući ime prijateljskog sklopa na sledeći način:

```
[assembly: InternalsVisibleTo ("Friend")]
```

Ako je prijateljski sklop potpisan jakim imenom, morate zadati njegov *pun* 160-bajtni javni ključ. Taj ključ možete dobiti pomoću LINQ upita – interaktivan primer je dat u LINQPadovoj besplatnoj biblioteci primera za knjigu *C# 5.0 za programere* (*C# 5.0 in a Nutshell*).

Nadjačavanje dostupnosti

Tip nadjačava dostupnost svojih deklarisanih članova. Najčešći primer nadjačavanja je kada imate `internal` tip sa `public` članovima. Na primer:

```
class C { public void Foo() {} }
```

Podrazumevana dostupnost klase `C` – `internal` – nadjačava dostupnost metode `Foo`, pa i `Foo` postaje `internal`. `Foo` se najčešće označava pomoću modifikatora `public` da bi refaktorisanje bilo lakše ako se i C kasnije promeni u `public`.

Interfejsi

Interfejs je sličan klasi, ali sadrži samo specifikaciju a ne i implementaciju svojih članova. Evo po čemu je interfejs poseban:

- *Svi članovi interfejsa* su implicitno apstraktni. Sutrotno tome, klasa može da obezbedi implementaciju i apstraktnih i konkretnih članova.
- Klasa (ili struktura) može da implementira *više* interfejsa. Suprotno tome, klasa može da nasleđuje samo *jednu* klasu, a struktura ne može da nasleđuje ništa (ali može da se izvodi iz `System.ValueType`).

Deklaracija interfejsa ista je kao deklaracija klase, ali ne sadrži implementaciju svojih članova pošto su svi oni implicitno apstraktni. Te članove će implementirati klase i strukture koje implementiraju interfejs. Interfejs može da sadrži samo metode, svojstva, događaje i indeksere, koji su – ne slučajno – baš članovi klase koja može da bude apstraktna.

Evo malo pojednostavljene verzije interfejsa `IEnumerator`, definisanog u `System.Collections`:

```
public interface IEnumerator
{
  bool MoveNext();
  object Current { get; }
}
```

Članovi interfejsa su uvek implicitno javni i ne mogu deklarisati modifikator pristupa. Implementiranje interfejsa znači obezbeđivanje `public` implementacije za sve njegove članove:

```
internal class Countdown : IEnumerator
{
  int count = 11;
  public bool MoveNext()  { return count-- > 0 ; }
  public object Current   { get { return count; } }
}
```

Objekat možete implicitno konvertovati u svaki interfejs koji ga implementira:

```
IEnumerator e = new Countdown();
while (e.MoveNext())
  Console.Write (e.Current);            // 109876543210
```

Proširivanje interfejsa

Interfejsi se mogu izvesti iz drugih interfejsa. Na primer:

```
public interface IUndoable             { void Undo(); }
public interface IRedoable : IUndoable  { void Redo(); }
```

`IRedoable` „nasleđuje“ sve članove interfejsa `IUndoable`.

Eksplicitna implementacija interfejsa

Implementiranje više interfejsa može ponekad da rezultira sukobom između potpisa članova. Takve sukobe možete da razrešite tako što ćete *eksplicitno implementirati* člana interfejsa. Na primer:

```
interface I1 { void Foo(); }
interface I2 { int Foo(); }

public class Widget : I1, I2
{
  public void Foo() // Implicitna implementacija
  {
    Console.Write ("Widget's implementation of I1.Foo");
  }
  int I2.Foo() // Eksplicitna implementacija člana I2.Foo
  {
    Console.Write ("Widget's implementation of I2.Foo");
    return 42;
   }
}
```

Pošto i `I1` i `I2` imaju sukobljene potpise za `Foo`, `Widget` eksplicitno implementira metodu `Foo` člana `I2`. To omogućava da te dve metode postoje u istoj klasi. Jedini način da pozovete eksplicitno implementiran član jeste da ga konvertujete u njegov interfejs:

```
Widget w = new Widget();
w.Foo();          // Klasa Widget implementira I1.Foo
((I1)w).Foo();    // Klasa Widget implementira I1.Foo
((I2)w).Foo();    // Klasa Widget implementira I2.Foo
```

Drugi razlog za eksplicitno implementiranje članova interfejsa jeste sakrivanje članova koji su visokospecijalizovani i ometaju uobičajenu primenu datog tipa. Recimo, obično je poželjno da tip koji implementira `ISerializable` izbegne izlaganje svojih `ISerializable` članova osim ako su eksplicitno konvertovani u taj interfejs.

Virtuelno implementiranje članova interfejsa

Implicitno implementiran član interfejsa je, podrazumevano, zatvoren. U osnovnoj klasi mora da bude označen sa `virtual` ili `abstract` da bi bio redefinisan: pozivanje člana interfejsa preko osnovne klase ili interfejsa rezultuje pozivom implementacije potklase.

Eksplicitno implementiran član interfejsa ne može se označiti pomoću `virtual`, niti se može redefinisati na uobičajen način. Međutim, može se *reimplementirati*.

Reimplementiranje interfejsa u potklasi

Potklasa može da *reimplementira* svaki član interfejsa koji je već implementirala osnovna klasa. Reimplementacija otima implementaciju člana (kada se pozove preko interfejsa) i funkcioniše bez obzira na to da li je član označen kao `virtual` u osnovnoj klasi.

U sledećem primeru, `TextBox` implementira `IUndo.Undo` eksplicitno, pa se ne može označiti sa `virtual`. Da bi ga „redefinisao“, `RichTextBox` mora da reimplementira metodu `Undo` interfejsa `IUndo`:

```
public interface IUndoable { void Undo(); }

public class TextBox : IUndoable
{
  void IUndoable.Undo()
  { Console.WriteLine ("TextBox.Undo"); }
}

public class RichTextBox : TextBox, IUndoable
{
  public new void Undo()
  { Console.WriteLine ("RichTextBox.Undo"); }
}
```

Pozivanjem reimplementiranog člana preko interfejsa, poziva se implementacija potklase:

```
RichTextBox r = new RichTextBox();
r.Undo();                  // RichTextBox.Undo
((IUndoable)r).Undo();     // RichTextBox.Undo
```

U ovom slučaju, metoda `Undo` je implementirana eksplicitno. Implicitno implementirani članovi takođe se mogu reimplementirati, ali efekat nije potpun pošto osnovna klasa pokreće osnovnu implementaciju.

Nabrajanja

`enum` je specijalan vrednosni tip (nabrojivi tip) koji omogućava da zadate grupu imenovanih numeričkih konstanti. Na primer:

```
public enum BorderSide { Left, Right, Top, Bottom }
```

Evo kako možemo koristiti ovaj nabrojivi tip:

```
BorderSide topSide = BorderSide.Top;
bool isTop = (topSide == BorderSide.Top);    // true
```

Svaki član nabrajanja ima pridruženu celobrojnu vrednost. Te vrednosti su podrazumevano tipa `int`, a članovima nabrajanja su dodeljene konstante `0`, `1`, `2`... (po redosledu deklarisanja članova). Alternativan celobrojni tip možete zadati na sledeći način:

```
public enum BorderSide : byte { Left,Right,Top,Bottom }
```

Možete zadati i eksplicitnu celobrojnu vrednost za svaki član:

```
public enum BorderSide : byte
  { Left=1, Right=2, Top=10, Bottom=11 }
```

Kompajler vam takođe omogućava da eksplicitno dodelite vrednosti nekim članovima nabrajanja. Članovi nabrajanja kojima nije dodeljena vrednost nastavljaju da se inkrementiraju od poslednje eksplicitno dodeljene vrednosti. Prethodni primer je ekvivalentan sledećem:

```
public enum BorderSide : byte
  { Left=1, Right, Top=10, Bottom }
```

Konverzije nabrajanja

Instancu nabrojivog tipa možete eksplicitno konvertovati u pridruženu celobrojnu vrednost i iz nje:

```
int i = (int) BorderSide.Left;
BorderSide side = (BorderSide) i;
bool leftOrRight = (int) side <= 2;
```

Možete, takođe, eksplicitno konvertovati jedan nabrojivi tip u drugi; tada se koriste celobrojne vrednosti pridružene članovima.

Numerički literal 0 tretira se na poseban način – ne treba mu eksplicitna konverzija:

```
BorderSide b = 0;      // Nije potrebna eksplicitna konverzija
if (b == 0) ...
```

U ovom konkretnom primeru, `BorderSide` nema člana s celobrojnom vrednošću `0`. Ipak, ne javlja se greška: ograničenje nabrojivih tipova jeste to što kompajler i CLR ne sprečavaju dodeljivanje celih brojeva čije su vrednosti van opsega članova:

```
BorderSide b = (BorderSide) 12345;
Console.WriteLine (b);   // 12345
```

Kombinovanje nabrajanja i atribut Flags

Članove nabrajanja možete kombinovati. Da bi se sprečile nejasnoće, članovi nabrajanja koje se može kombinovati moraju imati eksplicitno dodeljene vrednosti, obično stepene broja dva. Na primer:

```
[Flags]
public enum BorderSides
  { None=0, Left=1, Right=2, Top=4, Bottom=8 }
```

Po konvenciji, nabrojivom tipu koji se može kombinovati daje se ime u množini umesto u jednini. Da biste radili s kombinovanim enum vrednostima, koristite operatore nad bitovima, kao što su `|` i `&`. Oni deluju na pridružene celobrojne vrednosti:

```
BorderSides leftRight =
  BorderSides.Left | BorderSides.Right;
```

```
if ((leftRight & BorderSides.Left) != 0)
  Console.WriteLine ("Includes Left");      // Includes Left

string formatted = leftRight.ToString();    // "Left, Right"

BorderSides s = BorderSides.Left;
s |= BorderSides.Right;
Console.WriteLine (s == leftRight);         // True
```

Na nabrojive tipove koji se mogu kombinovati treba primeniti atribut `Flags`; ako to ne uradite, pozivanjem metode `ToString` za instancu `enum` dobija se broj umesto niza imena.

Pogodnosti radi, kombinovane članove možete uključiti u deklaraciju samog nabrajanja:

```
[Flags] public enum BorderSides
{
   None=0,
  Left=1, Right=2, Top=4, Bottom=8,
  LeftRight = Left | Right,
  TopBottom = Top  | Bottom,
  All       = LeftRight | TopBottom
}
```

Operatori za nabrajanja

S nabrojivim tipovima koriste se sledeći operatori:

```
= == != < > <= >= + -^  &  | ~
+= -= ++ -sizeof
```

Operatori nad bitovima, aritmetički operatori i operatori poređenja vraćaju rezultat obrade pridruženih celobrojnih vrednosti. Mogu se sabirati nabrojivi tip i celobrojni tip, ali ne i dva nabrojiva tipa.

Ugnežđeni tipovi

Ugnežđeni tip (engl. *nested type*) deklariše se unutar opsega nekog drugog tipa. Na primer:

```
public class TopLevel
{
  public class Nested { }              // Ugnežđena klasa
  public enum Color { Red, Blue, Tan } // Ugnežđeno nabrajanje
}
```

Ugnežđeni tip ima sledeće osobine:

- Može pristupati privatnim članovima nadređenog tipa (tj. tipa u koji je ugnežđen) i svemu drugome čemu taj tip može da pristupa.
- Može se deklarisati sa svim modifikatorima pristupa, a ne samo sa `public` i `internal`.
- Podrazumevana vidljivost ugnežđenog tipa je `private` a ne `internal`.
- Za pristupanje ugnežđenom tipu izvan nadređenog tipa potrebna je kvalifikacija pomoću imena nadređenog tipa (kao za pristupanje statičkim članovima).

Na primer, da bismo pristupili članu `Color.Red` izvan klase `TopLevel`, morali bismo da postupimo ovako:

```
TopLevel.Color color = TopLevel.Color.Red;
```

Svi tipovi se mogu ugnežđivati u druge tipove; međutim, samo klase i strukture mogu *ugnežđivati* druge tipove.

Generičke komponente

C# ima dva zasebna mehanizma za pisanje koda koji se može višekratno koristiti s različitim tipovima: *nasleđivanje* i *generičke komponente*. Dok nasleđivanje izražava mogućnost ponovne upotrebe pomoću osnovnog tipa, generički mehanizam je izražava pomoću „šablona“ koji sadrži „uzorke“ (engl. *placeholders*). Generički mehanizam, u poređenju s nasleđivanjem, može da *poveća bezbednost tipova* i *smanji konvertovanje i pakovanje*.

Generički tipovi

Generički tip deklariše *parametre tipa* – zamenske tipove koje korisnik generičkog tipa zamenjuje stvarnim tipovima tako što zadaje *argumente tipa*. Evo generičkog tipa, `Stack<T>`, koji skladišti instance tipa `T`. `Stack<T>` deklariše samo jedan parametar tipa, `T`:

```
public class Stack<T>
{
  int position;
  T[] data = new T[100];
  public void Push (T obj) { data[position++] = obj; }
  public T Pop()           { return data[--position]; }
}
```

Klasu `Stack<T>` možemo koristiti na sledeći način:

```
Stack<int> stack = new Stack<int>();
stack.Push(5);
stack.Push(10);
int x = stack.Pop();          // x je 10
int y = stack.Pop();          // y je 5
```

Napomena

Obratite pažnju na to da u poslednja dva reda nije potrebno konvertovanje naniže, čime se izbegava greška pri izvršavanju i eliminiše potreba za pakovanjem/raspakivanjem. Zahvaljujući tome, naš generički stek je bolji od negeneričkog steka koji koristi tip `object` umesto `T` (videti primer u odeljku „Tip object“ na strani 81).

`Stack<int>` zamenjuje parametar tipa `T` argumentom tipa `int`, implicitno određujući tip u hodu (do sinteze dolazi u vreme izvršavanja).U suštini, `Stack<int>` ima sledeću definiciju (supstitucija je označena podebljanim slovima, a ime klase je zamenjeno znakovima tarabe da bi se izbegla zabuna):

```
public class ###
{
  int position;
  int[] data;
  public void Push (int obj) { data[position++] = obj; }
  public int Pop()           { return data[--position]; }
}
```

Tehnički, kaže se da je `Stack<T>` *otvoren tip*, dok je `Stack<int>` *zatvoren tip*. U vreme izvršavanja, sve instance generičkog tipa su zatvorene – s popunjenim zamenama za argumente.

Generičke metode

Generička metoda deklariše parametre tipa u okviru potpisa metode. Pomoću generičkih metoda, mnogi algoritmi se mogu implementirati kao opšti, bez određenih tipova. Evo generičke metode koja međusobno zamenjuje sadržaje dve promenljive generičkog (opšteg) tipa `T`:

```
static void Swap<T> (ref T a, ref T b)
{
  T temp = a; a = b; b = temp;
}
```

`Swap<T>` se može upotrebiti na sledeći način:

```
int x = 5, y = 10;
Swap (ref x, ref y);
```

Generičkoj metodi uglavnom ne treba prosleđivati argumente tipa, pošto kompajler može implicitno zaključiti o kom tipu je reč. Ako postoji mogućnost zabune, generičke metode se mogu pozvati i sa izričito zadatim argumentima tipa:

```
Swap<int> (ref x, ref y);
```

U okviru generičkog *tipa*, metoda se ne svrstava u generičke ukoliko ne *uvodi* svoje parametre tipa (pomoću sintakse sa uglastim zagradama).

Metoda `Pop` u našem generičkom steku samo koristi postojeći parametar tipa, `T`, tako da nije svrstana u generičke.

Metode i tipovi su jedini konstrukti koji mogu da uvedu parametre tipa. Svojstva, indekseri, događaji, polja, konstruktori, operatori itd., ne mogu da deklarišu parametre tipa, mada mogu da učestvuju u svim parametrima tipa koji su već deklarisani u tipu koji ih obuhvata. Recimo, u našem primeru generičkog steka, mogli bismo da napišemo indekser koji vraća generički objekat:

```
public T this [int index] { get { return data[index]; } }
```

Slično tome, konstruktori mogu da učestvuju u postojećim parametrima tipa, ali ne mogu da ih *uvedu*.

Deklarisanje parametara tipa

Parametri tipa se mogu uvesti kroz deklaraciju klasa, struktura, interfejsa, delegata (videti odeljak „Delegati“ na strani 104) i metoda. Generički tip ili metoda može imati više parametara:

```
class Dictionary<TKey, TValue> {...}
```

Za instanciranje:

```
var myDic = new Dictionary<int,string>();
```

Imena generičkih tipova i imena metoda mogu se preklapati sve dok im se broj parametara tipa razlikuje. Na primer, sledeća dva imena tipa nisu sukobljena:

```
class A<T> {}
class A<T1,T2> {}
```

Napomena

Po konvenciji, generički tipovi i metode sa samo *jednim* parametrom tipa, daju svom parametru ime `T`, pod uslovom da je namena tog parametra jasna. Kada ima *više* parametara tipa, svaki parametar ima opisnije ime (s prefiksom `T`).

Operator typeof i nevezani generički tipovi

Otvoreni generički tipovi ne postoje u vreme izvršavanja: zatvaranje otvorenih generičkih tipova deo je kompajliranja. Međutim, *nevezani* (engl. *unbound*) generički tip može da postoji u vreme izvršavanja – samo kao objekat `Type`. Nevezani generički tip u jeziku C# može se zadati isključivo kao operand operatora `typeof`:

```
class A<T> {}
class A<T1,T2> {}
...

Type a1 = typeof (A<>);   // Nevezani tip
Type a2 = typeof (A<,>);  // Označava 2 argumenta tipa
Console.Write (a2.GetGenericArguments().Count()); // 2
```

Operator `typeof` možete koristiti za zadavanje zatvorenog tipa:

```
Type a3 = typeof (A<int,int>);
```

ili otvorenog tipa (koji se zatvara u vreme izvršavanja):

```
class B<T> { void X() { Type t = typeof (T); } }
```

Generička vrednost default

Rezervisana reč `default` može se koristiti za dobijanje podrazumevane vrednosti parametra generičkog tipa. Za referentni tip, podrazumevana vrednost je `null`, a za vrednosni tip – rezultat postavljanja svih bitova u polju na nulu:

```
static void Zap<T> (T[] array)
{
  for (int i = 0; i < array.Length; i++)
    array[i] = default(T);
}
```

Ograničenja generičkih tipova

Parametar tipa se može zameniti bilo kojim stvarnim tipom. Na parametar tipa mogu se primeniti *ograničenja* (engl. *constraints*) kako bi se zahtevali specifičniji argumenti tipa. Postoji šest vrsta ograničenja:

```
where T : osnovna_klasa    // Ograničenje na osnovnu klasu
where T : interfejs        // Ograničenje na interfejs
where T : class            // Ograničenje na referentni tip
where T : struct           // Ograničenje na vrednosni tip
where T : new()            // Ograničenje na besparametarski
                           // konstruktor
where U : T                // Ograničenje na generički tip
```

U sledećem primeru, klasa `GenericClass<T,U>` zahteva da `T` bude objekat izveden iz klase `SomeClass` (ili identične klase) i da implementira `Interface1`, a da `U` ima besparametarski konstruktor:

```
class     SomeClass {}
interface Interface1 {}

class GenericClass<T,U> where T : SomeClass, Interface1
                        where U : new()
{ ... }
```

Ograničenja se mogu primeniti gde god se definišu parametri tipa, i to ili u metodama ili u definicijama tipova.

Ograničenje na osnovnu klasu specificira da parametar tipa mora biti potklasa određene klase (ili joj odgovarati); *ograničenje na interfejs* specificira da parametar tipa mora implementirati taj interfejs. Ova ograničenja omogućavaju da se instance parametra tipa implicitno konvertuju u tu klasu, odnosno interfejs.

Ograničenje class i *ograničenje struct* specificiraju da `T` mora biti referentnog tipa odnosno vrednosnog tipa koji ne prihvata `null`. *Ograničenje na besparametarski konstruktor* zahteva da `T` ima javni besparametarski konstruktor i omogućava vam da pozovete metodu `new()` za `T`:

```
static void Initialize<T> (T[] array) where T : new()
{
  for (int i = 0; i < array.Length; i++)
```

```
        array[i] = new T();
    }
```

Ograničenje na generički tip zahteva da se jedan parametar tipa izvodi iz drugog parametra tipa (ili da mu odgovara).

Potklase generičkih tipova

Generičkoj klasi se može dodati potklasa – baš kao i negeneričkoj. Potklasa može da ostavi otvorene parametre tipa osnovne klase, kao u sledećem primeru:

```
class Stack<T> {...}
class SpecialStack<T> : Stack<T> {...}
```

Ili, potklasa može da zatvori parametre generičkog tipa pomoću konkretnog tipa:

```
class IntStack : Stack<int> {...}
```

Podtip takođe može da uvede i nove argumente tipa:

```
class List<T> {...}
class KeyedList<T,TKey> : List<T> {...}
```

Samoreferenciranje generičkih tipova

Tip može da imenuje *sebe* kao konkretan tip koda koji zatvara argument tipa:

```
public interface IEquatable<T> { bool Equals (T obj); }
public class Balloon : IEquatable<Balloon>
{
  public bool Equals (Balloon b) { ... }
}
```

I ovo je ispravno:

```
class Foo<T> where T : IComparable<T> { ... }
class Bar<T> where T : Bar<T> { ... }
```

Statički podaci

Statički podaci su jedinstveni za svaki zatvoren tip:

```
class Bob<T> { public static int Count; }...
Console.WriteLine (++Bob<int>.Count);      // 1
Console.WriteLine (++Bob<int>.Count);      // 2
Console.WriteLine (++Bob<string>.Count);   // 1
Console.WriteLine (++Bob<object>.Count);   // 1
```

Kovarijansa

Pod pretpostavkom da se `A` može konvertovati u `B`, `X` je kovarijantno ako se `X<A>` može konvertovati u `X<B>`.

NAPOMENA

Kovarijansa i kontravarijansa su napredni koncepti. Motiv za njihovo uvođenje u C# bio je omogućavanje da generički interfejsi i druge generičke komponente (naročito one definisane u Frameworku, kao što je `IEnumerable<T>`) rade *više u skladu s očekivanjima*. To vam može koristiti čak i ako ne razumete detalje koji stoje iza kovarijanse i kontravarijanse.

(U jeziku C# i njegovom poimanju varijanse, „može se konvertovati“ znači može se konvertovati pomoću *implicitne konverzije reference* – npr. `A je` *potklasa* klase `B`, ili `A` *implementira* `B`. To ne važi za numeričke konverzije, konverzije pri pakovanju (engl. *boxing conversions*) i namenske konverzije.)

Na primer, tip `IFoo<T>` je kovarijantan za `T` ukoliko je sledeći kôd ispravan:

```
IFoo<string> s = ...;
IFoo<object> b = s;
```

Od verzije C# 4.0, generički interfejsi dozvoljavaju kovarijansu za parametre tipa označene modifikatorom `out` (poput generičkih delegata).

Radi ilustracije, pretpostavite da klasa `Stack<T>` koju smo napisali na početku ovog odeljka implementira sledeći interfejs:

```
public interface IPoppable<out T> { T Pop(); }
```

Modifikator `out` primenjen na `T` ukazuje na to da se `T` koristi samo na izlaznim pozicijama (npr., povratni tipovi metoda). Modifikator `out` označava interfejs kao *kovarijantan* i omogućava nam da uradimo sledeće:

```
// Pod pretpostavkom da je Bear potklasa klase Animal:
var bears = new Stack<Bear>();
bears.Push (new Bear());

// Pošto bears implementira IPoppable<Bear>,
// možemo je konvertovati u IPoppable<Animal>:
IPoppable<Animal> animals = bears;   // Ispravno
Animal a = animals.Pop();
```

Konverziju `bears` u `animals` dozvoljava kompajler – budući da je interfejs kovarijantan.

Napomena

Interfejsi `IEnumerator<T>` i `IEnumerable<T>` (videti „Nabrajanja i iteratori“ na strani 130) označeni su kao kovarijantni od verzije Framework 4.0. To vam omogućava da konvertujete `IEnumerable<string>` u `IEnumerable<object>`, na primer.

Kompajler će generisati grešku ako kovarijantni parametar tipa upotrebite na *ulaznoj* poziciji (npr., kao parametar metode ili upisivo svojstvo). Namena ovog ograničenja je garantovanje bezbednosti tipa u vreme prevođenja. Na primer, ono nas sprečava da interfejsu dodamo metodu `Push(T)` koju bi korisnici mogli da zloupotrebe pomoću naizgled bezazlene operacije dodeljivanja objekta tipa Camel kao vrednost objekta `IPoppable<Animal>` (ne zaboravite da je u našem primeru pridružen tip stek objekata tipa Bear). Da bismo definisali metodu `Push(T)`, `T` mora biti kontravarijantan.

Napomena

C# podržava kovarijansu (i kontravarijansu) samo za elemente s *konverzijama referenci* – ne i *konverzijama pri pakovanju*. Prema tome, ako ste napisali metodu koja prihvata parametar tipa `IPoppable<object>`, možete da je pozovete sa `IPoppable<string>`, ali ne i sa `IPoppable<int>`.

Kontravarijansa

Prethodno smo videli sledeće: pod uslovom da `A` dozvoljava implicitnu konverziju reference u `B`, tip `X` je kovarijansa ako `X<A>` dozvoljava konverziju reference u `X<B>`. Tip je *kontravarijantan* kada možete da obavite konverziju u suprotnom smeru – iz `X<B>` u `X<A>`. Ovo je podržano za interfejse i delegate kada se parametar tipa pojavljuje samo na *ulaznim pozicijama*, koje su označene modifikatorom `in`. Proširujući prethodni primer, ako klasa `Stack<T>` implementira sledeći interfejs:

```
public interface IPushable<in T> { void Push (T obj); }
```

možemo uraditi i ovo:

```
IPushable<Animal> animals = new Stack<Animal>();
IPushable<Bear> bears = animals;   // Ispravno
bears.Push (new Bear());
```

Preslikavajući kovarjansu, kompajler će prijaviti grešku ako pokušate da upotrebite kontravarijantni parametar tipa na izlaznoj poziciji (npr., kao povratnu vrednost, ili u čitljivom svojstvu).

Delegati

Delegat povezuje pozivajuću metodu s pozvanom metodom u vreme izvršavanja. Postoje dva aspekta delegata: *tip* i *instanca*. *Tip delegata* definiše *protokol* kome se pokoravaju pozivalac i pozvani, koji obuhvata listu parametara tipa i povratni tip. *Instanca delegata* je objekat koji referencira jednu (ili više) pozvanih metoda usklađenih s tim protokolom.

Instanca delegata bukvalno služi pozivaocu kao delegat: pozivalac poziva delegata, a zatim taj delegat poziva odredišnu metodu. To preusmeravanje razdvaja pozivaoca od odredišne metode.

Deklaraciji tipa delegata prethodi rezervisana reč `delegate`, ali je inače ona slična deklaraciji (apstraktne) metode. Na primer:

```
delegate int Transformer (int x);
```

Da biste napravili instancu delegata, dodelite metodu promenljivoj delegata:

```
class Test
{
  static void Main()
  {
    Transformer t = Square;  // Pravi instancu delegata
    int result = t(3);       // Pokreće delegata
    Console.Write (result);  // 9
  }
  static int Square (int x) { return x * x; }
}
```

Pokretanje delegata je kao pokretanje metode (pošto je namena delegata samo da obezbedi peusmeravanje):

```
t(3);
```

Naredba `Transformer t = Square` skraćeni je način pisanja naredbe:

```
Transformer t = new Transformer (Square);
```

A `t(3)` je skraćeno od:

```
t.Invoke (3);
```

Delegat je sličan *povratnoj funkciji* (engl. *callback*), opštem terminu koji obuhvata konstrukte kao što su pokazivači na C funkcije.

Pisanje modularnih metoda pomoću delegata

Promenljivoj delegata metoda se dodeljuje u vreme izvršavanja. To je korisno za pisanje modularnih metoda (engl. *plug-in methods*). U ovom primeru, imamo uslužnu metodu `Transform` koja primenjuje

neku transformaciju na svaki element u celobrojnom nizu. Metoda `Transform` ima parametar delegata, pomoću kojeg se zadaje modularna transformacija.

```
public delegate int Transformer (int x);

class Test
{
  static void Main()
  {
    int[] values = { 1, 2, 3 };
    Transform (values, Square);
    foreach (int i in values)
      Console.Write (i + " ");   // 1 4 9
  }

  static void Transform (int[] values, Transformer t)
  {
    for (int i = 0; i < values.Length; i++)
      values[i] = t (values[i]);
  }

  static int Square (int x) { return x * x; }
}
```

Višeznačni delegati

Sve instance delegata karakteriše *višeznačnost* (engl. *multicast*). To znači da jedna instanca delegata može da referencira ne samo jednu odredišnu metodu već i listu takvih metoda. Operatori + i += kombinuju instance delegata. Na primer:

```
SomeDelegate d = SomeMethod1;
d += SomeMethod2;
```

Poslednji red je funkcionalno isti kao:

```
d = d + SomeMethod2;
```

Kada se pozove d, time se sada pozivaju i metoda SomeMethod1 i metoda SomeMethod2. Delegati se pozivaju redosledom kojim su dodati.

Operatori - i -= uklanjaju desni operand delegata iz levog. Na primer:

```
d -= SomeMethod1;
```

Pokretanjem d sada će se pokrenuti samo SomeMethod2.

Primena operatora + ili += na promenljivu delegata čija je vrednost null dozvoljena je, pošto poziva -= za promenljivu delegata s jednim odredištem (što će rezultovati time da instanca delegata bude null).

NAPOMENA

Delegati su *nepromenljivi*, pa kada pozovete += ili -=, vi – u stvari – pravite *novu* instancu delegata i dodeljujete je postojećoj promenljivoj.

Ako povratni tip višeznačnog delegata (engl. *multicast delegate*) nije void, pozivalac prima povratnu vrednost od poslednje metode koja se poziva. Prethodne metode se i dalje pozivaju, ali se njihove povratne vrednosti odbacuju. U većini scenarija korišćenja višeznačnih delegata, njihovi povratni tipovi će biti void, pa je ovaj detalj nebitan.

Svi tipovi delegata implicitno se izvode iz System.MulticastDelegate, koji nasleđuje klasu System.Delegate. C# prevodi operacije +, -, += i -= nad delegatom u statičke metode Combine i Remove klase System.Delegate.

Poređenje metode instance sa statičkom metodom kao odredištem

Kada se metoda *instance* dodeli delegatskom objektu, on mora da održi referencu ne samo ka toj metodi već i ka instanci kojoj ta metoda pripada. Svojstvo Target klase System.Delegate predstavlja tu *instancu* (i biće null za delegata koji referencira statičku metodu).

Generički tipovi delegata

Tip delegata može da sadrži parametre generičkog tipa. Na primer:

```
public delegate T Transformer<T> (T arg);
```

Evo kako možemo koristiti ovaj tip delegata:

```
static double Square (double x) { return x * x; }
static void Main()
{
  Transformer<double> s = Square;
  Console.WriteLine (s (3.3));          // 10.89
}
```

Delegati Func i Action

S generičkim delegatima postaje moguće napisati mali skup delegatskih tipova koji su toliko opšti da funkcionišu s metodama koje vraćaju proizvoljan tip rezultata i prihvataju proizvoljan broj argumenata (u razumnim granicama). Radi se o delegatima `Func` i `Action`, definisanim u imenskom prostoru `System` (`in` i `out` označavaju *varijansu*, koju ćemo uskoro objasniti):

```
delegate TResult Func <out TResult> ();
delegate TResult Func <in T, out TResult> (T arg);
delegate TResult Func <in T1, in T2, out TResult>
  (T1 arg1, T2 arg2);
  ... i tako dalje, do T16

delegate void Action ();
delegate void Action <in T> (T arg);
delegate void Action <in T1, in T2> (T1 arg1, T2 arg2);
... i tako dalje, do T16
```

Ovi delegati su krajnje opšti. Delegat `Transformer` iz našeg prethodnog primera može se zameniti delegatom `Func` koji prima jedan argument tipa `T` i vraća vrednost istog tipa:

```
public static void Transform<T> (
  T[] values, Func<T,T> transformer)
{
```

```
  for (int i = 0; i < values.Length; i++)
  values[i] = transformer (values[i]);
}
```

Jedini praktični slučajevi koje ovi delegati ne pokrivaju jesu `ref`/`out` i parametri tipa pokazivač.

Kompatibilnost delegata

Svi delegatski tipovi međusobno su nekompatibilni, čak i ako su im potpisi isti:

```
delegate void D1(); delegate void D2();
...
D1 d1 = Method1;
D2 d2 = d1;              // Greška pri prevođenju
```

Međutim, sledeće je dozvoljeno:

```
D2 d2 = new D2 (d1);
```

Instance delegata se smatraju jednakim ako su istog tipa i sa istim odredištem (odredištima) metode. Za višeznačne delegate, važan je redosled odredišnih metoda.

Varijansa povratnog tipa

Kada pozovete metodu, možda ćete kao rezultat dobiti tip koji je specifičniji nego što ste želeli. To je uobičajeno polimorfno ponašanje. U skladu s tim, odredišna metoda delegata može da vrati tip koji je specifičniji od onog opisanog u deklaraciji delegata. To je kovarijansa, i podržana je počevši od verzije C# 2.0:

```
delegate object ObjectRetriever();
...
static void Main()
{
  ObjectRetriever o = new ObjectRetriever (GetString);
  object result = o();
  Console.WriteLine (result);     // hello
}
static string GetString() { return "hello"; }
```

`ObjectRetriever` očekuje da mu se vrati `object`, ali će mu odgovarati i *potklasa* tipa `object` pošto su povratni tipovi delegata *kovarijantni*.

Varijansa parametra

Kada pozovete metodu, možete joj proslediti argumente čiji su tipovi specifičniji od parametara te metode. To je uobičajeno polimorfno ponašanje. U skladu s tim, odredišna metoda delegata može imati manje specifične tipove parametara od onih koje opisuje delegat. To se zove *kontravarijansa*:

```
delegate void StringAction (string s);
...
static void Main()
{
  StringAction sa = new StringAction (ActOnObject);
  sa ("hello");
}
static void ActOnObject (object o)
{
  Console.WriteLine (o); // hello
}
```

Napomena

Standardan šablon događaja zamišljen je tako da vam omogući da upotrebom osnovne klase `EventArgs` iskoristite kontravarijansu parametara delegata. Na primer, jednu metodu mogu pozvati dva različita delegata, od kojih joj jedan prosleđuje argumente tipa `MouseEventArgs` a drugi tipa `KeyEventArgs`.

Varijansa parametara tipa za generičke delegate

U odeljku „Generičke komponente“ na strani 95, videli smo kako parametri tipa mogu biti kovarijantni i kontravarijantni za generičke interfejse. Ista mogućnost postoji i za generičke delegate od verzije C# 4.0. Ako za delegat definišete generički tip, preporučljivo je sledeće:

- Parametar tipa koji se koristi samo za povratnu vrednost označiti kao kovarijantni (`out`).
- Svaki parametar tipa koji se koristi samo za ulazne parametre označiti kao kontravarijantni (`in`).

Tako omogućavate prirodno obavljanje konverzija uz poštovanje naslednih veza između tipova. Sledeći delegat (definisan u imenskom prostoru `System`) kovarijantan je za `TResult`:

```
delegate TResult Func<out TResult>();
```

što omogućava da se napiše:

```
Func<string> x = ...;
Func<object> y = x;
```

Sledeći delegat (definisan u imenskom prostoru `System`) kontravarijantan je za `T`:

```
delegate void Action<in T> (T arg);
```

što omogućava da se napiše:

```
Action<object> x = ...;
Action<string> y = x;
```

Događaji

Pri korišćenju delegata obično se javljaju dve važne uloge: *emiter* ili *izvor* (engl. *broadcaster*) i pretplatnik (engl. *subscriber*). *Emiter* je tip koji sadrži polje tipa delegat. Emiter odlučuje kada će emitovati, tako što poziva delegata. *Pretplatnici* su primaoci čije metode obrađuju događaje. Pretplatnik odlučuje kada da započne i prekine osluškivanje, tako što primenjuje operatore `+=` i `-=` na delegata datog emitera. Jedan pretplatnik ne zna za druge pretplatnike, niti se meša u njihov rad.

Događaji su komponenta programskog jezika pomoću koje se formalizuje ovakav način rada. `event` je konstrukt koji eksponira samo onaj podskup osobina delegata koji je potreban modelu emiter/pretplatnik. Glavna namena događaja je da *spreče pretplatnike da ometaju jedni druge*.

Događaj ćete najlakše deklarisati tako što stavite rezervisanu reč `event` ispred delegatskog člana:

```
public class Broadcaster
{
  public event ProgressReporter Progress;
}
```

Kôd unutar tipa `Broadcaster` ima pun pristup događaju `Progress` i može ga tretirati kao delegata. Kôd izvan tipa `Broadcaster` može da primenjuje samo operacije `+=` i `-=` na događaj `Progress`.

U narednom primeru, klasa `Stock` pokreće svoj događaj `Price.Changed` kad god se promeni cena (`Price`) robe (`Stock`):

```
public delegate void PriceChangedHandler
  (decimal oldPrice, decimal newPrice);

public class Stock
{
  string symbol; decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event PriceChangedHandler PriceChanged;

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      // Pokreće događaj ako lista pokretanja nije prazna:
      if (PriceChanged != null)
        PriceChanged (price, value);
      price = value;
    }
  }
}
```

Ako iz našeg primera uklonimo rezervisanu reč `event` tako da `PriceChanged` postane obično polje tipa delegat, dobićemo isti rezultat. Međutim, klasa `Stock` neće više biti tako robusna jer će pretplatnici moći da urade sledeće kako bi uticali jedan na drugog:

- da zamene druge pretplatnike tako što će dodeliti drugu metodu delegatu `PriceChanged` (umesto da koriste operator `+=`).
- da obrišu sve pretplatnike klase (tako što će delegat `PriceChanged` postaviti na vrednost `null`).
- da emituju ka drugim pretplatnicima pokretanjem delegata.

Događaji mogu biti virtuelni, redefinisani, apstraktni ili zatvoreni. Mogu takođe biti i statički.

Standardni šablon za događaje

.NET Framework definiše standardni šablon za pisanje događaja. Namena mu je da obezbedi doslednost između Frameworka i korisničkog koda. Evo prethodnog primera napisanog pomoću ovog šablona:

```
public class PriceChangedEventArgs : EventArgs
{
  public readonly decimal LastPrice, NewPrice;

  public PriceChangedEventArgs (decimal lastPrice,
                                decimal newPrice)
  {
    LastPrice = lastPrice; NewPrice = newPrice;
  }
}
public class Stock
{
  string symbol; decimal price;

  public Stock (string symbol) { this.symbol = symbol; }

  public event EventHandler<PriceChangedEventArgs>
               PriceChanged;
```

```
  protected virtual void OnPriceChanged
                         (PriceChangedEventArgs e)
  {
    if (PriceChanged != null) PriceChanged (this, e);
  }

  public decimal Price
  {
    get { return price; }
    set
    {
      if (price == value) return;
      OnPriceChanged (new PriceChangedEventArgs (price,
                                                 value));
      price = value;
    }
  }
}
```

U središtu standardnog šablona za događaje nalazi se `System.EventArgs`: unapred definisana Framework klasa bez članova (izuzev statičkog svojstva `Empty`). `EventArgs` je osnovna klasa za prenos informacija događaju. U ovom primeru, pravimo potklasu `EventArgs` da bismo preneli staru i novu cenu kada se pokrene događaj `PriceChanged`.

Generički delegat `System.EventHandler` takođe je deo.NET Frameworka i definisan je ovako:

```
public delegate void EventHandler<TEventArgs>
  (object source, TEventArgs e)
  where TEventArgs : EventArgs;
```

Napomena

Pre verzije C# 2.0 (kada su jeziku dodate generičke komponente) rešenje je bilo da se napiše namenski delegat za obradu događaja za svaki `EventArgs` tip, na sledeći način:

```
delegate void PriceChangedHandler
  (object sender,
   PriceChangedEventArgs e);
```

Iz istorijskih razloga, većina događaja unutar Frameworka koristi ovako definisane delegate.

Zaštićena virtuelna metoda, po imenu `On-ime-događaja`, centralizuje pokretanje tog događaja. To omogućava da potklase pokrenu događaj (što je najčešće poželjno), a takođe i da umetnu kôd pre i posle pokretanja događaja.

Evo kako bismo mogli upotrebiti klasu `Stock`:

```
static void Main()
{
  Stock stock = new Stock ("THPW");
  stock.Price = 27.10M;

  stock.PriceChanged += stock_PriceChanged;
  stock.Price = 31.59M;
}

static void stock_PriceChanged
  (object sender, PriceChangedEventArgs e)
{
  if ((e.NewPrice - e.LastPrice) / e.LastPrice > 0.1M)
    Console.WriteLine ("Alert, 10% price increase!");
}
```

Za događaje koji ne nose dodatne informacije, Framework obezbeđuje negenerički delegat `EventHandler`. Da bismo to pokazali, ponovo ćemo napisati klasu `Stock` tako da se događaj `PriceChanged` pokreće *nakon* promene cene. To znači da nikakve dodatne informacije ne treba preneti s događajem:

```
public class Stock
{
  string symbol; decimal price;
```

```
    public Stock (string symbol) {this.symbol = symbol;}

    public event EventHandler PriceChanged;

    protected virtual void OnPriceChanged (EventArgs e)
    {
      if (PriceChanged != null) PriceChanged (this, e);
    }

    public decimal Price
    {
      get { return price; }
      set
      {
        if (price == value) return;
        price = value;
        OnPriceChanged (EventArgs.Empty);
      }
    }
  }
```

Obratite pažnju na to da smo koristili i svojstvo `EventArgs.Empty` – zahvaljujući tome ne moramo da pravimo instancu klase `EventArgs`.

Metode za pristupanje događajima

Za pristupanje događaju koriste se implementacije njegovih funkcija `+=` i `-=`. Podrazumevane *metode za pristupanje* (engl. *accessors*) implicitno implementira kompajler. Razmotrite sledeću deklaraciju događaja:

```
public event EventHandler PriceChanged;
```

Kompajler konvertuje ovaj kôd u sledeće:

- Polje privatnog delegata
- Par javnih funkcija za pristupanje događaju, čije implementacije prosleđuju operacije `+=` i `-=` privatnom delegatu

Ovaj proces možete da preuzmete tako što ćete definisati *eksplicitne* elemente za pristup događajima. Evo ručne implementacije događaja `PriceChanged` iz prethodnog primera:

```
EventHandler _priceChanged;   // Privatni delegat
public event EventHandler PriceChanged
{
  add { _priceChanged += value; }
  remove { _priceChanged -= value; }
}
```

Ovaj primer je funkcionalno identičan podrazumevanoj implementaciji metoda za pristupanje događajima u jeziku C# (osim što se C# stara za bezbedne višenitne operacije pri ažuriranju delegata). Time što sami definišemo metode za pristupanje događaju, nalažemo jeziku C# da ne generiše podrazumevanu logiku polja i metodu za pristupanje.

Uz eksplicitno definisane metode za pristupanje događajima, možete primeniti složenije strategije za skladištenje pripadajućeg delegata i pristup njemu. To je korisno kada su metode za pristup događaju samo releji za drugu klasu koja emituje taj događaj, ili kada se eksplicitno implementira interfejs koji deklariše događaj:

```
public interface IFoo { event EventHandler Ev; }
class Foo : IFoo
{
  EventHandler ev;
  event EventHandler IFoo.Ev
  {
    add { ev += value; } remove { ev -= value; }
  }
}
```

Lambda izrazi

Lambda izraz je bezimena metoda napisana umesto instance delegata. Kompajler odmah konvertuje lambda izraz u jedno od sledećeg:

- instancu delegata ili
- *stablo izraza*, tipa `Expression<TDelegate>`, koje predstavlja kôd unutar lambda izraza u sekvencijalnom objektnom modelu. To omogućava da se lambda izraz interpretira kasnije u vreme izvršavanja (taj proces opisujemo u poglavlju 8 knjige *C# 5.0 za programere*).

Ako imamo sledeći tip delegata:

```
delegate int Transformer (int i);
```

evo kako ćemo dodeliti i pokrenuti lambda izraz `x => x * x`:

```
Transformer sqr = x => x * x;
Console.WriteLine (sqr(3)); // 9
```

Napomena

Interno, kompajler pretvara lambda izraze ovog tipa tako što piše privatnu metodu i u nju premešta kôd datog izraza.

Lambda izraz ima sledeći oblik:

```
(parametri) => izraz-ili-blok-naredaba
```

Pogodnosti radi, možete da izostavite zagrade, ali samo ako postoji samo jedan parametar i ako se može implicitno zaključiti kog je tipa.

U našem primeru postoji samo jedan parametar, `x`, a izraz je `x * x`:

```
x => x * x;
```

Svaki parametar lambda izraza odgovara jednom parametru delegata, a tip izraza (koji može biti i `void`) odgovara povratnom tipu delegata.

U našem primeru, `x` odgovara parametru `i`, a izraz `x * x` odgovara povratnom tipu `int`, pa je stoga kompatibilan s delegatom `Transformer`.

Kôd lambda izraza može da bude *blok naredaba* umesto izraz. Evo kako ćemo prepraviti primer:

```
x => { return x * x; };
```

Lambda izrazi se obično koriste s delegatima `Func` i `Action`, pa ćete naš raniji izraz najčešće videti napisan ovako:

```
Func<int,int> sqr = x => x * x;
```

Kompajler obično može iz konteksta da zaključi kog su tipa lambda parametri. Kada to nije slučaj, kompajleru možete navesti tipove parametara:

```
Func<int,int> sqr = (int x) => x * x;
```

Evo primera izraza koji prihvata dva parametra:

```
Func<string,string,int> totalLength =
  (s1, s2) => s1.Length + s2.Length;

int total = totalLength ("hello", "world"); // total=10;
```

Pod pretpostavkom da je `Clicked` događaj tipa `EventHandler`, sledeći kôd priključuje događaju proceduru za obradu događaja (engl. *event handler*) u obliku lambda izraza:

```
obj.Clicked += (sender,args) => Console.Write ("Click");
```

Preuzimanje vrednosti spoljnih promenljivih

Lambda izraz može da referencira lokalne promenljive i parametre metode u kojoj je definisan (*spoljne promenljive*). Na primer:

```
static void Main()
{
  int factor = 2;
  Func<int, int> multiplier = n => n * factor;
  Console.WriteLine (multiplier (3)); // 6
}
```

Spoljne promenljive koje referencira lambda izraz zovu se *preuzete promenljive* (engl. *captured variables*). Lambda izraz koji preuzima promenljive zove se *zatvarajući izraz* (engl. *closure*). Preuzete promenljive se izračunavaju kada se dati delegat *pokrene*, a ne onda kada su promenljive *preuzete*:

```
int factor = 2;
Func<int, int> multiplier = n => n * factor;
factor = 10;
Console.WriteLine (multiplier (3));             // 30
```

Lambda izrazi mogu i sami da ažuriraju preuzete promenljive:

```
int seed = 0;Func<int> natural = () => seed++;
Console.WriteLine (natural());              // 0
```

```
Console.WriteLine (natural());              // 1
Console.WriteLine (seed);                   // 2
```

Životni vek preuzetih promenljivih produžava se na životni vek datog delegata. U narednom primeru, seme lokalne promenljive (`seed`) obično bi nestalo iz opsega čim se izvrši delegat `Natural`. Međutim, pošto je promenljiva `seed` *preuzeta*, životni vek joj se produžava do kraja životnog veka delegata `natural`:

```
static Func<int> Natural()
{
  int seed = 0;
  return () => seed++;      // Vraća zatvarajući izraz
}
static void Main()
{
  Func<int> natural = Natural();
  Console.WriteLine (natural());        // 0
  Console.WriteLine (natural());        // 1
}
```

Preuzimanje iteracionih promenljivih

Kada preuzmete iteracionu promenljivu u petlji `for`, C# je tretira kao da je deklarisana *izvan* te petlje. To znači da se *ista* promenljiva preuzima u svakoj iteraciji. Sledeći program ispisuje `333` umesto `012`:

```
Action[] actions = new Action[3];

for (int i = 0; i < 3; i++)
  actions [i] = () => Console.Write (i);
foreach (Action a in actions) a(); // 333
```

Svaki zatvarajući izraz (prikazan podebljano) preuzima istu promenljivu, `i`. (To u stvari ima smisla kada imate u vidu da je `i` promenljiva čija vrednost ostaje ista između iteracija petlje; ako želite, možete čak eksplicitno izmeniti `i` u telu petlje.) Posledica je sledeća: kada se kasnije pokrenu delegati, svaki od njih vidi vrednost `i` u vreme *pokretanja* – a to je 3. Ukoliko želimo da program ispiše `012`, rešenje je da iteracionu promenljivu dodelimo lokalnoj promenljivoj koja se prati *unutar* petlje:

```
Action[] actions = new Action[3];
for (int i = 0; i < 3; i++)
{
  int loopScopedi = i;
  actions [i] = () => Console.Write (loopScopedi);
}
foreach (Action a in actions) a();    // 012
```

To uslovljava da lambda izraz preuzme *različitu* promenljivu pri svakoj iteraciji.

Upozorenje

Petlje `foreach` su ranije funkcionisale na isti način, ali su se od tada promenila pravila. Počevši od verzije C# 5.0, možete u lambda izrazu bezbedno zatvoriti iteracionu promenljivu petlje `foreach` a da vam ne treba neka privremena promenljiva.

Anonimne metode

Anonimne metode su mogućnost iz verzije C# 2.0 koju sada zamenjuju lambda izrazi. Anonimna metoda je kao lambda izraz, a razlikuje se po sledećem: tip njenih parametara se ne određuje implicitno, sintaksi izraza (anonimna metoda mora uvek da bude blok naredaba) i sposobnosti da se prevede u stablo izraza.

Da biste napisali anonimnu metodu, dodajte rezervisanu reč `delegate` praćenu (opciono) deklaracijom parametra iza koje sledi telo metode. Na primer, ako je dat ovaj delegat:

```
delegate int Transformer (int i);
```

evo kako bismo mogli da napišemo i pozovemo anonimnu metodu:

```
Transformer sqr = delegate (int x) {return x * x;};
Console.WriteLine (sqr(3)); // 9
```

Prvi red je semantički ekvivalentan sledećem lambda izrazu:

```
Transformer sqr =        (int x) => {return x * x;};
```

Ili samo:

```
Transformer sqr =              x => x * x;
```

Jedinstvena karakteristika anonimnih metoda jeste to što možete potpuno izostaviti deklaraciju parametara – čak i ako je delegat očekuje. To može biti korisno pri deklarisanju događaja s podrazumevanom, praznom procedurom za obradu događaja:

```
public event EventHandler Clicked = delegate { };
```

Tako se izbegava provera da li je delegat `null` pre pokretanja događaja. Sledeće je takođe ispravno (zapazite da nema parametara):

```
Clicked += delegate { Console.Write ("clicked"); };
```

Anonimne metode preuzimaju spoljne promenljive na isti način kao što to rade lambda izrazi.

Naredbe try i izuzeci

Naredba `try` specificira blok koda za obradu grešaka, tj. za operacije čišćenja. Iza *bloka* `try` mora da se nalazi *blok* `catch`, *blok* `finally`, ili i jedan i drugi. Blok `catch` se izvršava kada se u bloku `try` pojavi greška. Blok `finally` se izvršava nakon što tok izvršavanja napusti blok `try` (ili, ukoliko postoji, blok `catch`), kako bi se izvršio kôd za čišćenje, bez obzira na to da li se javila greška ili nije.

Blok `catch` ima pristup objektu `Exception` koji sadrži informaciju o grešci. Blok `catch` koristite ili da obradite grešku ili da ponovo generišete taj izuzetak. Ponovo generišete izuzetak ukoliko želite samo da evidentirate problem, ili ako hoćete da generišete nov tip izuzetka, višeg nivoa.

Blok `finally` dodaje determinizam vašem programu tako što se uvek izvršava, bez obzira na sve. Koristan je za poslove čišćenja kao što je zatvaranje mrežnih veza.

Evo kako izgleda naredba try:

```
try {
  ... // izuzetak se može generisati tokom izvršavanja
      // ovog bloka
}
catch (ExceptionA ex)
{
  ... // obrada izuzetka tipa ExceptionA
}
catch (ExceptionB ex)
{
  ... // obrada izuzetka tipa ExceptionB
}
finally
{
  ... // kôd za čišćenje
}
```

Razmotrite sledeći kôd:

```
int x = 3, y = 0;
Console.WriteLine (x / y);
```

Pošto je y nula, izvršno okruženje generiše izuzetak DivideByZero, i program se završava. To možemo da sprečimo tako što ćemo presresti izuzetak na sledeći način:

```
try
{
  int x = 3, y = 0;
  Console.WriteLine (x / y);
}
catch (DivideByZeroException ex)
{
  Console.Write ("y cannot be zero. ");
}
// Izvršavanje se nastavlja ovde, nakon izuzetka...
```

Napomena

Ovo je jednostavan primer koji ilustruje obradu izuzetaka. U praksi se sa ovim konkretnim slučajem možemo bolje izboriti tako što ćemo eksplicitno proveriti da li je delilac nula pre nego što pozovemo `Calc`.

Obrada izuzetaka je relativno „skupa“ jer su potrebne stotine ciklusa radnog takta.

Kada se generiše izuzetak, CLR obavlja test:

Da li je tok izvršavanja trenutno unutar neke naredbe `try` *koja može da presretne izuzetak?*

- Ako jeste, izvršavanje se prosleđuje odgovarajućem bloku `catch`. Ukoliko se blok `catch` uspešno izvrši, izvršavanje se pomera na sledeću naredbu iza naredbe `try` (ako postoji, pri čemu se prvo izvršava blok `finally`).
- U suprotnom, tok izvršavanja se vraća pozivaocu funkcije, a test se ponavlja (nakon što se izvrše eventualni blokovi `finally` koji omotavaju datu naredbu).

U slučaju da nijedna funkcija na steku poziva ne preuzima odgovornost za izuzetak, korisniku se prikazuje poruka o grešci i program se završava.

Odredba catch

Odredba `catch` specificira koju vrstu izuzetka treba obraditi. To mora biti ili `System.Exception` ili neka njegova potklasa. Obradom izuzetaka `System.Exception` obrađuju se sve moguće greške. To je korisno u sledećim slučajevima:

- Kada se program možda može oporaviti, bez obzira na konkretnu vrstu izuzetka.
- Kada planirate da ponovo generišete izuzetak (možda nakon što ste ga evidentirali).

- Kada vam je procedura za obradu izuzetaka poslednje rešenje, pre izlaska iz programa.

Međutim, češće ćete obrađivati *određene vrste izuzetaka*, kako ne biste morali da se bavite slučajevima za koje vaša procedura za obradu izuzetaka nije namenjena (npr., `OutOfMemoryException`).

Za obradu više vrsta izuzetaka koristićete više odredaba `catch`:

```
try
{
  DoSomething();
}

catch (IndexOutOfRangeException ex) { ... }
catch (FormatException ex)          { ... }
catch (OverflowException ex)        { ... }
```

Pri obradi izuzetka, izvršava se samo jedna odredba `catch`. Ako želite da u kôd uvrstite i sigurnosnu mrežu kako biste presretali i opštije izuzetke (kao što je `System.Exception`) morate *prvo* navesti procedure za obradu specifičnijih izuzetaka.

Izuzetak se može presresti i bez zadavanja promenljive, ukoliko ne treba da pristupate njenim svojstvima:

```
catch (StackOverflowException)   // bez promenljive
  { ... }
```

Štaviše, možete izostaviti i promenljivu i tip (što znači da će biti obrađeni svi izuzeci):

```
catch { ... }
```

Blok finally

Blok `finally` se uvek izvršava – bez obzira na to da li je izuzetak generisan i da li se blok `try` izvršava do kraja. Blokovi `finally` se obično koriste za kôd za čišćenje.

Blok `finally` se izvršava ili:

- posle završetka bloka `catch`
- nakon što upravljanje tokom izvršavanja napusti blok `try` zbog naredbe za skok (npr., `return` ili `goto`)
- posle završetka bloka `try`

Blok `finally` pomaže da se programu doda determinisanost. U sledećem primeru, datoteka koju otvorimo *uvek* se zatvori, nezavisno od toga da li se:

- blok `try` završi normalno.
- izvršavanje prekine ranije jer je datoteka prazna (`EndOf Stream`).
- generiše izuzetak `IOException` tokom čitanja datoteke.

Na primer:

```
static void ReadFile()
{
  StreamReader reader = null; // U imenskom prostoru System.IO
  try
  {
    reader = File.OpenText ("file.txt");
    if (reader.EndOfStream) return;
    Console.WriteLine (reader.ReadToEnd());
 }
  finally
  {
    if (reader != null) reader.Dispose();
  }
}
```

U ovom primeru zatvorili smo datoteku tako što smo pozvali metodu `Dispose` objekta `StreamReader`. Pozivanje metode `Dispose` objekta unutar bloka `finally`, standardno je pravilo u celom .NET Frameworku i C# ga eksplicitno podržava pomoću naredbe `using`.

Naredba using

Mnoge klase kapsuliraju neupravljane resurse, kao što su identifikatori datoteka (engl. *file handles*), identifikatori grafičkih objekata (engl. *graphics handles*), ili veze s bazama podataka. Te klase implementiraju

interfejs `System.IDisposable`, koji definiše samo jednu, besparametarsku metodu po imenu `Dispose` za brisanje tih resursa. Naredba `using` obezbeđuje elegantnu sintaksu za pozivanje metode `Dispose` za `IDisposable` objekat unutar bloka `finally`.

Sledeći kôd:

```
using (StreamReader reader = File.OpenText ("file.txt"))
{
  ...
}
```

potpuno je identičan kodu:

```
StreamReader reader = File.OpenText ("file.txt");
try
{
  ...
}
finally
{
  if (reader != null) ((IDisposable)reader).Dispose();
}
```

Generisanje izuzetaka

Izuzetke može generisati ili izvršno okruženje ili korisnički kôd. Ovde metoda `Display` generiše izuzetak `System.ArgumentNullException`:

```
static void Display (string name)
{
  if (name == null)
    throw new ArgumentNullException ("name");

  Console.WriteLine (name);
}
```

Ponovno generisanje izuzetka

Evo kako možete preuzeti i ponovo generisati izuzetak:

```
try { ... }
catch (Exception ex)
{
  // Evidentiranje greške
  ...
  throw;            // Ponovno generisanje istog izuzetka
}
```

Ponovno generisanje na ovaj način omogućava da evidentirate grešku a da je ne *progutate*. Uz to, omogućava vam i da izbegnete obradu neke greške ukoliko se ispostavi da su okolnosti izvan očekivanih.

Napomena

Kada bismo zamenili `throw` sa `throw ex`, primer bi i dalje funkcionisao, ali svojstvo `StackTrace` više ne bi odražavalo prvobitnu grešku.

Drugi uobičajen scenario jeste ponovno generisanje specifičnije ili smislenije vrste izuzetka:

```
try
{
  ... // učitavanje datuma rođenja iz XML podataka
}
catch (FormatException ex)
{
  throw new XmlException ("Invalid date of birth", ex);
}
```

Pri ponovnom generisanju drugačijeg izuzetka, u svojstvo `InnerException` možete upisati prvobitni izuzetak da bi se olakšalo pronalaženje i otklanjanje grešaka. Gotovo sve vrste izuzetaka imaju konstruktor za tu namenu (kao u našem primeru).

Najvažnija svojstva izuzetka System.Exception

Evo najvažnijih svojstava izuzetka `System.Exception`:

`StackTrace`
: Znakovni niz koji predstavlja sve metode koje se pozivaju od mesta nastanka greške do bloka `catch`.

`Message`
: Znakovni niz sa opisom greške.

`InnerException`
: Unutrašnja greška (ako postoji) koja je izazvala spoljnu grešku. Ona i sama može da ima još jedan `InnerException`.

Uobičajene vrste izuzetaka

Ovde su navedene vrste izuzetaka koje se standardno koriste u celom izvršnom okruženju (CLR) i .NET Frameworku. Možete ih generisati sami ili ih koristiti kao osnovne klase za izvođenje namenskih vrsta izuzetaka.

`System.ArgumentException`
: Generiše se kada se funkcija pozove s pogrešnim argumentom. Obično znači grešku u programu.

`System.ArgumentNullException`
: Potklasa izuzetka `ArgumentException` koja se generiše kada je argument funkcije (neočekivano) `null`.

`System.ArgumentOutOfRangeException`
: Potklasa izuzetka `ArgumentException` koja se generiše kada je (obično numerički) argument prevelik ili premali. Na primer, generiše se kada se negativan broj prosledi funkciji koja prihvata samo pozitivne vrednosti.

`System.InvalidOperationException`
: Generiše se kada stanje objekta nije pogodno da bi se metoda uspešno izvršila, bez obzira na konkretne vrednosti argumenata. Primeri toga su čitanje neotvorene datoteke ili pribavljanje sledećeg elementa od enumeratora kada je pripadajuća lista izmenjena tokom iteracije.

`System.NotSupportedException`
: Generiše se kada određena funkcionalnost nije podržana. Dobar primer je pozivanje metode `Add` za kolekciju za koju `IsReadOnly` vraća `true`.

`System.NotImplementedException`
: Generiše se kada određena funkcija još nije implementirana.

`System.ObjectDisposedException`
: Generiše se kada je iz memorije uklonjen objekat za koji je pozvana funkcija.

Napomena

Potreba za izuzetkom `ArgumentException` (i njegovim potklasama) eliminisana je zahvaljujući tzv. kodnim ugovorima (engl. *code contracts*), što je opisano u poglavlju 13 knjige *C# 5.0 za programere*.

Nabrajanje i iteratori

Nabrajanje

Enumerator je kursor za sekvencu vrednosti koje se mogu samo čitati od početka ka kraju sekvence; to je objekat koji implementira interfejs `System.Collections.IEnumerator` ili `System.Collections.Generic.IEnumerator<T>`.

Naredba `foreach` obavlja iteraciju nad *nabrojivim* (engl. *enumerable*) objektom. Nabrojiv objekat je logički oblik predstavljanja sekvence. On sâm nije kursor, već objekat koji formira kursor za svoje vrednosti. Nabrojiv objekat implementira interfejs `IEnumerable`/`IEnumerable<T>` ili ima metodu `GetEnumerator` koja vraća *enumerator*.

Sintaksa nabrajanja izgleda ovako:

```
class Enumerator    // Obično implementira IEnumerator<T>
{
  public IteratorVariableType Current { get {...} }
```

```
  public bool MoveNext() {...}
}
class Enumerable    // Obično implementira IEnumerable<T>
{
  public Enumerator GetEnumerator() {...}
}
```

Evo kako se na visokom nivou iterira kroz znakove u reči *beer* pomoću naredbe `foreach`:

```
foreach (char c in "beer") Console.WriteLine (c);
```

Evo kako se na niskom nivou iterira kroz znakove u reči *beer* bez korišćenja naredbe `foreach`:

```
using (var enumerator = "beer".GetEnumerator())
  while (enumerator.MoveNext())
  {
    var element = enumerator.Current;
    Console.WriteLine (element);
}
```

Ako enumerator implementira interfejs `IDisposable`, naredba `foreach` služi i kao naredba `using`, implicitno uklanjajući nabrojivi objekat.

Inicijalizatori kolekcija

Nabrojiv objekat možete instancirati i popuniti u jednom koraku. Na primer:

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

List<int> list = new List<int> {1, 2, 3};
```

Kompajler prevodi poslednji red u sledeći kôd:

```
List<int> list = new List<int>();
list.Add (1); list.Add (2); list.Add (3);
```

Za to je potrebno da nabrojivi objekat implementira interfejs `System.Collections.IEnumerable`, i da ima metodu `Add` s odgovarajućim brojem parametara za poziv.

Iteratori

Dok je naredba `foreach` *korisnik* enumeratora, iterator je *proizvođač* enumeratora. U ovom primeru koristimo iterator da bismo dobili niz Fibonačijevih brojeva (u kome je svaki broj zbir prethodna dva):

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

class Test
{
  static void Main()
  {
    foreach (int fib in Fibs(6))
      Console.Write (fib + " ");
  }

  static IEnumerable<int> Fibs(int fibCount)
  {
    for (int i = 0, prevFib = 1, curFib = 1; i < fibCount; i++)
    {
      yield return prevFib;
      int newFib = prevFib+curFib;
      prevFib = curFib;
      curFib = newFib;
    }
  }
}
REZULTAT: 1  1  2  3  5  8
```

Dok naredba `return` saopštava: „Evo vrednosti koju ste tražili da vratim iz ove metode," naredba `yield return` saopštava: „Evo sledećeg elementa koji ste tražili da učitam iz ovog nabrajanja." Posle svake naredbe `yield`, kontrola se vraća pozivaocu, ali se stanje pozvanog održava, tako da metoda može nastaviti da se izvršava čim pozivalac pređe na sledeći element. Životni vek ovog stanja vezan je za enumerator, i ono se može osloboditi kada pozivalac završi nabrajanje.

Napomena

Kompajler konvertuje metode iteratora u privatne klase koje implementiraju interfejs `IEnumerable<T>` i/ili `IEnumerator<T>`. Logika unutar bloka iteratora je „obrnuta“ i raspodeljena između metode `MoveNext` i svojstva `Current` klase enumeratora koju generiše kompajler i koja postaje mašina s konačnim brojem stanja. To znači sledeće: kada pozovete metodu iteratora, samo instancirate klasu koju je generisao kompajler; nikakav vaš kôd se ne izvršava! Vaš kôd se izvršava samo kada počnete nabrajanje nad rezultujućom sekvencom vrednosti, obično pomoću naredbe `foreach`.

Semantika iteratora

Iterator je metoda, svojstvo ili indekser koji sadrži jednu ili više naredaba `yield`. Iterator mora da vrati jedan od sledeća četiri interfejsa (u suprotnom, kompajler će generisati grešku):

```
System.Collections.IEnumerable
System.Collections.IEnumerator
System.Collections.Generic.IEnumerable<T>
System.Collections.Generic.IEnumerator<T>
```

Iteratori koji vraćaju interfejs tipa *enumerator* koriste se ređe. Korisni su pri pisanju namenske klase za kolekcije: iteratoru obično date ime `GetEnumerator`, u svojoj klasi implementirate `IEnumerable<T>`.

Iteratori koji vraćaju interfejs tipa *enumerable* češći su i lakše se koriste pošto ne morate da pišete klasu za kolekcije. U pozadini, kompajler generiše privatnu klasu koja implementira `IEnumerable<T>` (kao i `IEnumerator<T>`).

Više naredaba yield

Iterator može da sadrži više naredaba `yield`:

```
static void Main()
{
  foreach (string s in Foo())
```

```
    Console.Write (s + " "); // One Two Three
}

static IEnumerable<string> Foo()
{
  yield return "One";
  yield return "Two";
  yield return "Three";
}
```

Naredba yield break

Naredba `yield break` ukazuje na to da iz bloka iteratora treba da se izađe brzo, bez vraćanja još elemenata. Da bismo to pokazali, izmenićemo `Foo`:

```
static IEnumerable<string> Foo (bool breakEarly)
{
  yield return "One";
  yield return "Two";
  if (breakEarly) yield break;
  yield return "Three";
}
```

Upozorenje

U bloku iteratora nije dozvoljena naredba `return` – morate koristiti naredbu `yield break`.

Formiranje sekvenci

Iteratori su veoma fleksibilni. Primer s Fibonačijevim nizom proširićemo tako što ćemo klasi dodati sledeću metodu:

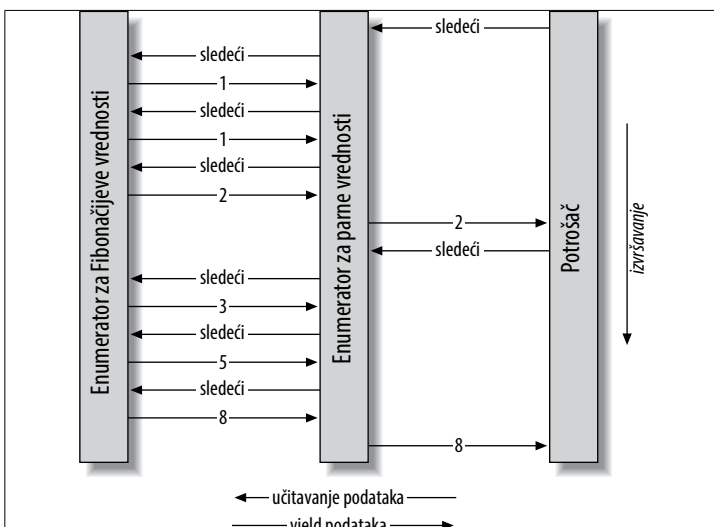
```
static IEnumerable<int> EvenNumbersOnly (
  IEnumerable<int> sequence)
  {
    foreach (int x in sequence)
      if ((x % 2) == 0)
        yield return x;
  }
}
```

Parne Fibonačijeve brojeve sada ćemo prikazati ovako:

```
foreach (int fib in EvenNumbersOnly (Fibs (6)))
  Console.Write (fib + " "); // 2 8
```

Element sekvence se ne izračunava do poslednjeg trenutka – dok ga ne zatraži operacija `MoveNext()`. Na slici 5 prikazani su zahtevi za podacima i rezultati prosleđeni tokom vremena.

Mogućnost kombinovanja modela iteratora ključna je za sastavljanje LINQ upita.



Slika 5. Formiranje sekvenci

Tipovi koji prihvataju null

Referentni tipovi mogu da predstave nepostojeću vrednost pomoću reference na `null`. S druge strane, vrednosni tipovi ne mogu da predstave vrednost `null`. Na primer:

```
string s = null;  // OK - referentni tip
.int i = null;    // Greška pri prevođenju - int ne može da
                  // bude null.
```

Da biste za vrednosni tip predstavili vrednost `null`, morate koristiti specijalan konstrukt koji se zove tip koji prihvata `null` (engl. *nullable type*). Taj tip se piše kao vrednosni tip praćen znakom pitanja:

```
int? i = null;                  // OK - tip koji prihvata
                                // null
Console.WriteLine (i == null);  // true
```

Struktura Nullable<T>

`T?` se prevodi u `System.Nullable<T>`. `Nullable<T>` je lagana, nepromenljiva struktura sa samo dva polja koja predstavljaju `Value` i `HasValue`. Suština strukture `System.Nullable<T>` veoma je jednostavna:

```
public struct Nullable<T> where T : struct
{
 public T Value {get;}
 public bool HasValue {get;}
 public T GetValueOrDefault();
 public T GetValueOrDefault (T defaultValue);
 ...
}
```

Kôd:

```
int? i = null;
Console.WriteLine (i == null);          // True
```

prevodi se u:

```
Nullable<int> i = new Nullable<int>();
Console.WriteLine (! i.HasValue);       // True
```

Pri pokušaju da se učita `Value` kada je `HasValue` `false`, generiše se `InvalidOperationException`. Metoda `GetValueOrDefault()` vraća `Value` ako je `HasValue` `true`; u suprotnom, vraća `new T()` ili zadatu podrazumevanu vrednost.

Podrazumevana vrednost `T?` je `null`.

Konverzije tipova koji prihvataju null

Konverzija `T` u `T?` je implicitna, a `T?` u `T` eksplicitna. Na primer:

```
int? x = 5;      // implicitna
int y = (int)x;  // eksplicitna
```

Eksplicitna konverzija je direktno ekvivalentna pozivanju svojstva `Value` datog objekta koji prihvata `null`. Znači, generiše se `InvalidOperationException` ukoliko je `HasValue false`.

Pakovanje/raspakivanje tipova koji prihvataju null

Kada je `T?` spakovano, zapakovana vrednost na hipu sadrži `T`, a ne `T?`. Ta optimizacija je moguća zato što je zapakovana vrednost referentni tip koji može da izrazi vrednost `null`.

C# takođe dopušta raspakivanje tipova koji prihvataju `null` pomoću operatora `as`. Ako konvertovanje ne uspe, rezultat će biti `null`:

```
object o = "string";
int? x = o as int?;
Console.WriteLine (x.HasValue);   // false
```

Krađa operatora

Struktura `Nullable<T>` ne definiše operatore kao što su `<`, `>`, pa čak ni `==`. Uprkos tome, sledeći kôd se prevodi i izvršava ispravno:

```
int? x = 5;
int? y = 10;
bool b = x < y;   // true
```

Ovo funkcioniše zahvaljujući tome što kompajler pozajmljuje, tj. „krade“ operator „manje od“ od pripadajućeg vrednosnog tipa. Semantički, on prevodi prethodni poredbeni izraz u sledeći:

```
bool b = (x.HasValue && y.HasValue)
          ? (x.Value < y.Value)
          : false;
```

Drugim rečima, ako i `x` i `y` imaju vrednosti, one se porede preko operatora „manje od“ promenljive tipa `int`; u suprotnom, rezultat je `false`.

Krađa operatora znači da možete implicitno primenjivati `T`-ove operatore na `T?`. Možete definisati operatore za `T?` da biste obezbedili posebno ponašanje za `null`, ali je u velikoj većini slučajeva najbolje da se oslonite na to da će kompajler automatski primeniti logiku za `null` umesto vas.

Kompajler primenjuje različitu logiku za `null`, zavisno od kategorije operatora.

Operatori jednakosti (==, !=)

Ukradeni operatori jednakosti obrađuju vrednosti `null` isto kao referentni tipovi. To znači da su dve vrednosti `null` jednake:

```
Console.WriteLine (       null ==        null);  // true
Console.WriteLine ((bool?)null == (bool?)null);  // true
```

Dalje:

- Ako jedan operand ima vrednost `null`, operandi nisu jednaki.
- Ako su oba operanda različita od `null`, porede se njihove vrednosti (`Value`).

Relacioni operatori (<, <=, >=, >)

Princip rada relacionih operatora jeste da je besmisleno porediti operande čije su vrednosti `null`. To znači da poređenje `null` vrednosti s vrednošću `null` ili onom koja nije `null` daje rezultat `false`.

```
bool b = x < y; // prevođenje vrednosti

bool b = (x == null || y == null)
  ? false
  : (x.Value < y.Value);

// b je false (pod uslovom da x ima vrednost 5 a y null)
```

Svi ostali operatori (+, –, *, /, %, &, |, ^, <<, >>, +, ++, --, !, ~)

Navedeni operatori vraćaju `null` kada bilo koji operand ima vrednost `null`. Trebalo bi da je ovaj šablon poznat korisnicima SQL-a.

```
int? c = x + y; // Prevod:
int? c = (x == null || y == null)
         ? null
         : (int?) (x.Value + y.Value);

// c je null (pod uslovom da x ima vrednost 5 a y null)
```

Izuzetak je kada se operatori `&` i `|` primene na `bool?`, o čemu će uskoro biti reči.

Kombinovanje operatora s tipovima koji prihvataju null i onima koji ga ne prihvataju

Možete mešati i kombinovati tipove koji prihvataju `null` i one koji ga ne prihvataju (zahvaljujući tome što postoji implicitna konverzija iz `T` u `T?`):

```
int? a = null;
int b = 2;
int? c = a + b;  // c je null - ekvivalentno a + (int?)b
```

bool? sa operatorima & i |

Kada im se proslede operandi tipa `bool?`, operatori `&` i `|` tretiraju `null` kao *nepoznatu vrednost*. Znači, `null | true` je true, zato što:

- Ako je nepoznata vrednost false, rezultat je true.
- Ako je nepoznata vrednost true, rezultat je true.

Slično tome, `null & false` je false. Ovakvo ponašanje je verovatno poznato korisnicima SQL-a. U sledećem primeru nabrojane su ostale kombinacije:

```
bool? n = null, f = false, t = true;
Console.WriteLine (n | n); // (null)
Console.WriteLine (n | f); // (null)
Console.WriteLine (n | t); // True
Console.WriteLine (n & n); // (null)
Console.WriteLine (n & f); // False
Console.WriteLine (n & t); // (null)
```

Operator koji zamenjuje vrednost null

Operator `??` zamenjuje vrednost `null` (engl. *null coalescing operator*), i može se koristiti i s tipovima koji prihvataju `null` i s referentnim tipovima. On znači: „Ako operand nije `null`, daj mi ga; u suprotnom, daj mi podrazumevanu vrednost.“ Na primer:

```
int? x = null;
int y = x ?? 5;    // y je 5

int? a = null, b = 1, c = 2;
Console.Write (a ?? b ?? c); // 1
                             // (prva vrednost koja nije null)
```

Operator `??` ekvivalentan je pozivanju funkcije/metode `GetValueOrDefault` sa eksplicitno zadatom podrazumevanom vrednošću, osim što se izraz prosleđen funkciji `GetValueOrDefault` nikada ne izračunava ukoliko promenljiva nije `null`.

Preklapanje operatora

Operatori se mogu preklapati (engl. *overload*) da bi se dobila prirodnija sintaksa za namenske tipove. Preklapanje operatora je najsvrsishodnije kada se koristi za implementiranje namenskih struktura koje predstavljaju relativno proste tipove podataka. Na primer, namenski numerički tip je odličan kandidat za preklapanje operatora.

Sledeći simbolički operatori mogu se preklopiti:

```
+   -   *   /   ++   --   !   ~   %   &   |   ^   ==   !=   <   <<   >>   >
```

Uz to, implicitne i eksplicitne konverzije takođe se mogu redefinisati (pomoću rezervisanih reči `implicit` i `explicit`), kao što mogu i literali `true` i `false`, te unarni operatori `+` i `-`.

Složeni operatori dodele (npr., `+=`, `/=`) automatski se redefinišu kada redefinišete osnovne operatore (npr., `+`, `/`).

Operatorske funkcije

Operator se preklapa tako što se deklariše *operatorska funkcija*. Operatorska funkcija mora da bude statička, i bar jedan operand mora da bude onog tipa u kojem je operatorska funkcija deklarisana.

U narednom primeru definišemo strukturu po imenu `Note`, koja predstavlja muzičku notu, a zatim preklapamo operator `+`:

```
public struct Note
{
  int value;

  public Note (int semitonesFromA)
  { value = semitonesFromA; }

  public static Note operator + (Note x, int semitones)
  {
    return new Note (x.value + semitones);
  }
}
```

Preklapanje nam omogućava da strukturi `Note` dodamo `int`:

```
Note B = new Note (2);
Note CSharp = B + 2;
```

Pošto smo preklopili `+`, možemo koristiti i operator `+=`:

```
CSharp += 2;
```

Preklapanje operatora jednakosti i poređenja

Operatori jednakosti i poređenja često se preklapaju pri pisanju struktura i, u retkim slučajevima, klasa. Za preklapanje ovih operatora postoje posebna pravila i obaveze:

Uparivanje
: C# kompajler nameće da budu definisana oba operatora koji čine logički par. To su operatori (`== !=`), (`< >`), i (`<= >=`).

Metode `Equals` *i* `GetHashCode`
: Ako preklopite `==` i `!=`, obično ćete morati da redefinišete metode `Equals` i `GetHashCode` datog objekta, kako bi kolekcije i heš-tabele pouzdano funkcionisale s tim tipom podataka.

Interfejsi `IComparable` *i* `IComparable<T>`
: Ako preklopite `<` i `>`, obično ćete upotrebiti i `IComparable` i `IComparable<T>`.

Proširujući prethodni primer, evo kako bismo preklopili operatore jednakosti za strukturu `Note`:

```
public static bool operator == (Note n1, Note n2)
{
  return n1.value == n2.value;
}
public static bool operator != (Note n1, Note n2)
{
  return !(n1.value == n2.value);
}
public override bool Equals (object otherNote)
{
  if (!(otherNote is Note)) return false;
  return this == (Note)otherNote;
}
public override int GetHashCode()
{
  return value.GetHashCode(); // Koristi se heš-kod vrednosti
}
```

Namenske implicitne i eksplicitne konverzije

Implicitne i eksplicitne konverzije su preklopivi operatori. One se obično preklapaju da bi konverzije između tesno povezanih tipova (kao što su numerički tipovi) bile koncizne i prirodne.

Kao što je objašnjeno pri razmatranju tipova, razlog korišćenja implicitnih konverzija jeste to što one treba da uvek uspevaju i što se informacije ne gube tokom konverzije. Inače, treba definisati eksplicitne konverzije.

U narednom primeru definišemo konverzije između našeg tipa `Note` i tipa `double` (koji predstavlja frekvenciju te note u hercima):

```
...
// Konvertovanje u herce
public static implicit operator double (Note x)
{
  return 440 * Math.Pow (2,(double) x.value / 12 );
}

// Konvertovanje iz herca (tačno do prvog sledećeg polutona)
public static explicit operator Note (double x)
{
  return new Note ((int) (0.5 + 12 * (Math.Log(x/440)
                                    / Math.Log(2)) ));
}
...

Note n =(Note)554.37;  // eksplicitna konverzija
double x = n;          // implicitna konverzija
```

NAPOMENA

Ovaj primer je pomalo nategnut: u stvarnoj situaciji, navedene konverzije bi se možda bolje implementirale pomoću metode `ToFrequency` i (statičke) metode `FromFrequency`.

Operatori `as` i `is` ignorišu namenske konverzije.

Proširene metode

Proširene metode (engl. *extension methods*) omogućavaju da se postojeći tip proširi novim metodama a da se ne menja definicija originalnog tipa. Proširena metoda je statička metoda statičke klase, gde je modifikator `this` primenjen na prvi parametar. Tip prvog parametra biće onaj koji je proširen. Na primer:

```
public static class StringHelper
{
  public static bool IsCapitalized (this string s)
  {
    if (string.IsNullOrEmpty (s)) return false;
    return char.IsUpper (s[0]);
  }
}
```

Proširena metoda `IsCapitalized` može se pozvati kao da je to metoda instance primenjena na znakovni niz, na sledeći način:

```
Console.Write ("Perth".IsCapitalized());
```

Kada se poziv proširene metode kompajlira, prevodi se u poziv obične, statičke metode:

```
Console.Write (StringHelper.IsCapitalized ("Perth"));
```

Interfejsi se takođe mogu proširiti:

```
public static T First<T> (this IEnumerable<T> sequence)
{
  foreach (T element in sequence)
    return element;

  throw new InvalidOperationException ("No elements!");
}

...
Console.WriteLine ("Seattle".First());   // S
```

Ulančavanje proširenih metoda

Proširene metode, poput metoda instance, pružaju način za uredno ulančavanje funkcija. Razmotrite sledeće dve funkcije:

```
public static class StringHelper
{
  public static string Pluralize (this string s) {...}
  public static string Capitalize (this string s) {...}
}
```

Promenljive `x` i `y` su ekvivalentne i obe daju rezultat `"Sausages"`, ali `x` koristi metode proširenja, a `y` – statičke metode:

```
string x = "sausage".Pluralize().Capitalize();

string y = StringHelper.Capitalize
           (StringHelper.Pluralize ("sausage"));
```

Razjašnjavanje dvosmislenosti

U imenskim prostorima

Da bi se moglo pristupiti proširenoj metodi, njen imenski prostor mora biti vidljiv (obično uvezen pomoću naredbe `using`).

Između proširene metode i metode instance

Svaka kompatibilna metoda instance uvek ima prednost nad proširenom metodom – čak i kada parametri proširene metode više odgovaraju po tipu.

Između proširene metode i proširene metode

Ako dve proširene metode imaju isti potpis, potrebna proširena metoda mora da se pozove kao obična statička metoda kako bi bilo jasno koja metoda se poziva. Međutim, ukoliko jedna proširena metoda ima specifičnije argumente, prednost ima specifičnija metoda.

Anonimni tipovi

Anonimni tip je jednostavna klasa napravljena u hodu, za smeštaj određenog skupa vrednosti. Da biste definisali anonimni tip, zadajte rezervisanu reč `new` praćenu inicijalizatorom objekta koji određuje svojstva i vrednosti koje će tip sadržati. Na primer:

```
var dude = new { Name = "Bob", Age = 1 };
```

Kompajler to rešava tako što formira privatan ugnežđen tip sa svojstvima samo za čitanje, za `Name` (tipa `string`) i `Age` (tipa `int`). Za opis anonimnog tipa morate navesti rezervisanu reč `var`, zato što ime tipa generiše kompajler.

Ime svojstva anonimnog tipa može se izvesti na osnovu izraza koji je sam po sebi identifikator. Na primer, izraz:

```
int Age = 1;var dude = new { Name = "Bob", Age };
```

ekvivalentan je:

```
var dude = new { Name = "Bob", Age = Age };
```

Evo kako ćete napraviti nizove anonimnih tipova:

```
var dudes = new[]
{
  new { Name = "Bob", Age = 30 },
  new { Name = "Mary", Age = 40 }
```

LINQ

LINQ, tj. Language Integrated Query, omogućava da pišete strukturirane upite sa sigurnim tipovima za lokalne kolekcije objekata i udaljene izvore podataka.

LINQ omogućava da izvršite upit nad svakom kolekcijom koja implementira interfejs `IEnumerable<>`, bez obzira na to da li je reč o nizu, listi, XML DOM-u, ili udaljenom izvoru podataka (kao što je tabela na SQL Serveru). Prednosti LINQ-a su i provera tipova u vreme prevođenja i dinamičko sastavljanje upita.

Napomena

Dobar način za eksperimentisanje sa LINQ-om jeste preuzimanje LINQPada sa lokacije *http://www.linqpad.net*. LINQPad omogućava da interaktivno izvršavate LINQ upite nad lokalnim kolekcijama i SQL bazama podataka, a da ne morate ništa da podešavate; uz to, u LINQ su ugrađeni brojni primeri.

Osnove LINQ-a

U LINQ-u, osnovne jedinice podataka su *sekvence* i *elementi*. Sekvenca je svaki objekat koji implementira generički interfejs `IEnumerable`,

a element je svaka stavka u toj sekvenci. U sledećem primeru, `names` je sekvenca, a `Tom`, `Dick` i `Harry` su elementi:

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

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

Operator upita (engl. *query operator*) jeste metoda koja transformiše sekvencu. Tipičan operator upita prihvata *ulaznu sekvencu* i proizvodi transformisanu *izlaznu sekvencu*. U klasi `Enumerable`, u imenskom prostoru `System.Linq` ima oko 40 operatora upita; svi su implementirani kao statičke proširene metode. Oni se zovu *standardni operatori za upite*.

NAPOMENA

LINQ podržava i sekvence koje se mogu dinamički proslediti iz udaljenog izvora podataka kao što je SQL Server. Te sekvence dodatno implementiraju interfejs `IQueryable<>` i podržava ih odgovarajući skup standardnih operatora za upite u klasi `Queryable`.

Jednostavan upit

Upit je izraz koji transformiše sekvence pomoću jednog ili više operatora za upite. Najjednostavniji upit se sastoji od jedne ulazne sekvence i jednog operatora. Na primer, evo kako možemo primeniti operator `Where` na jednostavan niz da bismo izdvojili imena dugačka najmanje četiri znaka:

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

IEnumerable<string> filteredNames =
  System.Linq.Enumerable.Where (
    names, n => n.Length >= 4);

foreach (string n in filteredNames)
  Console.Write (n + "|");              // Dick|Harry|
```

Pošto su standardni operatori za upite implementirani kao proširene metode, operator `Where` možemo pozvati direktno za `names` – kao da je to metoda instance:

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

(Da bi se ovaj kôd preveo, morate učitati imenski prostor `System.Linq` pomoću direktive `using`.) Metoda `Where` u imenskom prostoru `System.Linq.Enumerable` ima sledeći potpis:

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

`source` je ulazna *sekvenca*; `predicate` je delegat koji se izvršava za svaki ulazni *element*. Metoda `Where` obuhvata sve elemente u *izlaznoj sekvenci* za koje taj delegat vraća `true`. Interno, on je implementiran pomoću iteratora – evo njegovog izvornog koda:

```
foreach (TSource element in source)
  if (predicate (element))
    yield return element;
```

Projektovanje

Još jedan od osnovnih operatora za upite jeste metoda `Select`. Ona transformiše (*projektuje*) svaki element iz ulazne sekvence pomoću zadatog lambda izraza:

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

IEnumerable<string> upperNames =
  names.Select (n => n.ToUpper());

foreach (string n in upperNames)
  Console.Write (n + "|"); // TOM|DICK|HARRY|
```

Upit može da projektuje elemente u anonimni tip:

```
var query = names.Select (n => new {
                                    Name = n,
                                    Length = n.Length
                                  });
foreach (var row in query)
  Console.WriteLine (row);
```

Evo rezultata:

```
{ Name = Tom, Length = 3 }
{ Name = Dick, Length = 4 }
{ Name = Harry, Length = 5 }
```

Operatori Take i Skip

LINQ-u je važan prvobitni redosled elemenata u ulaznoj sekvenci. Neki operatori za upite podrazumevaju takvo ponašanje; takvi su operatori `Take`, `Skip` i `Reverse`. Operator `Take` daje prvih *x* elemenata, odbacujući ostale:

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

Operator `Skip` ignoriše prvih *x* elemenata, i kao rezultat daje ostale:

```
IEnumerable<int> lastTwo = numbers.Skip (3);
```

Operatori koji vraćaju elemente

Ne vraćaju svi operatori upita sekvencu. Operatori koji vraćaju elemente (engl. *element operators*) izdvajaju jedan element iz ulazne sekvence; primeri su `First`, `Last`, `Single` i `ElementAt`:

```
int[] numbers = { 10, 9, 8, 7, 6 };
int firstNumber = numbers.First();                  // 10
int lastNumber = numbers.Last();                    // 6
int secondNumber = numbers.ElementAt (2);           // 8
int firstOddNum = numbers.First (n => n%2 == 1);    // 9
```

Ako nema elemenata, svi ovi operatori generišu izuzetak. Da biste umesto izuzetka dobili povratnu vrednost koja je null/prazno, upotrebite metode `FirstOrDefault`, `LastOrDefault`, `SingleOrDefault` ili `ElementAtOrDefault`.

Metode `Single` i `SingleOrDefault` ekvivalentne su metodama `First` i `FirstOrDefault`, s tim što generišu izuzetak ukoliko postoji više od jednog poklapanja. Takvo ponašanje je korisno kada pomoću upita tražite neki red u tabeli baze podataka preko primarnog ključa.

Agregatni operatori

Agregatni operatori vraćaju skalarnu vrednost – obično numeričkog tipa. Najčešće se koriste agregatni operatori `Count`, `Min`, `Max` i `Average`:

```
int[] numbers = { 10, 9, 8, 7, 6 };
int count = numbers.Count();       // 5
int min = numbers.Min();           // 6
int max = numbers.Max();           // 10
double avg = numbers.Average();    // 8
```

`Count` prihvata opcioni predikat, koji pokazuje da li uvrstiti dati element. Sledeći kôd broji sve parne brojeve:

```
int evenNums = numbers.Count (n => n % 2 == 0); // 3
```

Operatori `Min`, `Max` i `Average` prihvataju opcioni argument koji transformiše svaki element pre njegove agregacije:

```
int maxRemainderAfterDivBy5 = numbers.Max (n => n % 5);   // 4
```

Sledeći kôd izračunava srednju vrednost kvadratnog korena promenljive `numbers`:

```
double rms = Math.Sqrt (numbers.Average (n => n * n));
```

Kvantifikatori

Kvantifikatori vraćaju logičku vrednost (`bool`). Kvantifikatori su `Contains`, `Any`, `All` i `SequenceEquals` (poredi dve sekvence):

```
int[] numbers = { 10, 9, 8, 7, 6 };

bool hasTheNumberNine = numbers.Contains (9);    // true
bool hasMoreThanZeroElements = numbers.Any();    // true
bool hasOddNum = numbers.Any (n => n % 2 == 1);  // true
bool allOddNums = numbers.All (n => n % 2 == 1); // false
```

Operatori za skupove

Operatori za skupove (engl. *set operators*) prihvataju dve ulazne sekvence istog tipa. `Concat` nadovezuje jednu sekvencu na drugu; `Union` radi isto to, ali uz uklanjanje duplikata:

```
int[] seq1 = { 1, 2, 3 }, seq2 = { 3, 4, 5 };

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

Preostala dva operatora iz ove kategorije jesu `Intersect` i `Except`:

```
IEnumerable<int>
  commonality = seq1.Intersect (seq2),   // { 3 }
  difference1 = seq1.Except (seq2),      // { 1, 2 }
  difference2 = seq2.Except (seq1);      // { 4, 5 }
```

Odloženo izvršavanje

Važna osobina mnogih operatora za upite jeste to što se ne izvršavaju onda kad su napravljeni, već kad su *nabrojani* (drugim rečima, kada se metoda `MoveNext` pozove za svoj enumerator). Razmotrite sledeći upit:

```
var numbers = new List<int> { 1 };
numbers.Add (1);

IEnumerable<int> query = numbers.Select (n => n * 10);
numbers.Add (2);    // Krišom ubacuje još jedan element

foreach (int n in query)
  Console.Write (n + "|");    // 10|20|
```

Dodatni broj koji smo ubacili u listu nakon sastavljanja upita obuhvaćen je rezultatom, zato što se operacije filtriranja ili sortiranja obavljaju tek kada se izvrši naredba `foreach`. To se naziva *odloženo* (engl. *deferred*) ili *lenjo* (engl. *lazy*) izvršavanje. Odloženo izvršavanje razdvaja formiranje upita od njegovog izvršavanja, omogućavajući vam da sastavite upit u nekoliko koraka, kao i da izvršite upit nad bazom podataka a da klijentu ne prikazujete sve redove. Svi standardni operatori za upite omogućavaju odloženo izvršavanje, osim sledećih:

- Operatori koji vraćaju samo jedan element ili skalarnu vrednost (operatori koji vraćaju *elemente*, *agregatni operatori* i *kvantifikatori*)
- Sledeći operatori za *konverzije*: `ToArray`, `ToList`, `ToDictionary`, `ToLookup`

Operatori za konverzije su zgodni, delimično i zato što sprečavaju odloženo izvršavanje. To može biti korisno za „zamrzavanje“ ili keširanje rezultata u određenom trenutku, da bi se izbeglo ponovno izvršavanje računski zahtevnog upita ili upita iz udaljenog izvora, kao što je LINQ upit za SQL tabelu. (Prateći efekat odloženog izvršavanja jeste to što se upit ponovo izvršava ukoliko ga kasnije renumerišete.)

Sledeći primer ilustruje upotrebu operatora `ToList`:

```
var numbers = new List<int>() { 1, 2 };

List<int> timesTen = numbers
  .Select (n => n * 10)
  .ToList();   // Odmah se izvršava u List<int>

numbers.Clear();
Console.WriteLine (timesTen.Count);    // I dalje 2
```

Upozorenje

Podupiti obezbeđuju još jedan nivo preusmeravanja. U podupitu je sve podložno odloženom izvršavanju – uključujući i metode agregacije i konverzije, zato što se i sâm podupit izvršava isključivo odloženo, na zahtev. Pod pretpostavkom da je `names` niz znakovnih nizova, podupit izgleda ovako:

```
names.Where (
  n => n.Length ==
    names.Min (n2 => n2.Length))
```

Standardni operatori za upite

Standardne operatore za upite (onakve kakvi su implementirani u klasi `System.Linq.Enumerable`) možemo svrstati u 12 kategorija, ukratko opisanih u tabeli 1.

Tabela 1. Kategorije operatora za upite

Kategorija	Opis	Odloženo izvršavanje?
Filtriranje	Vraća podskup elemenata koji zadovoljavaju zadati uslov	Da
Projektovanje	Transformiše svaki element pomoću lambda funkcije, uz opciono širenje podsekvenci	Da
Spajanje	Kombinuje elemente jedne kolekcije s drugom, primenom strategije koja efikasno troši raspoloživo vreme	Da
Promena redosleda	Vraća drugačiji redosled sekvence	Da
Grupisanje	Grupiše sekvencu u podsekvence	Da
Rad sa skupovima	Prihvata dve sekvence istog tipa, i vraća ono što im je zajedničko, zbir ili razliku	Da
Elementi	Izdvaja samo jedan element iz sekvence	Ne
Agregiranje	Obavlja izračunavanje nad sekvencom, vraćajući skalarnu vrednost (najčešće broj)	Ne
Kvantifikacija	Obavlja izračunavanje nad sekvencom, vraćajući `true` ili `false`	Ne
Konverzija: uvoženje	Konvertuje negeneričku sekvencu u generičku sekvencu (nad kojom se mogu izvršavati upiti)	Da
Konverzija: izvoženje	Konvertuje sekvencu u niz, listu, rečnik ili pretraživanje, namećući momentalno izvršavanje	Ne
Generisanje	Proizvodi jednostavnu sekvencu	Da

U tabelama od 2 do 13 dat je kratak pregled svih operatora upita. Operatori ispisani podebljano posebno su podržani u jeziku C# (videti „Izrazi upita“ na strani 158).

Tabela 2. Operatori za filtriranje

Metoda	Opis
`Where`	Vraća podskup elemenata koji zadovoljava dati uslov
`Take`	Vraća prvih *x* elemenata i odbacuje ostale
`Skip`	Ignoriše prvih *x* elemenata i vraća ostale
`TakeWhile`	Preuzima elemente od ulazne sekvence sve dok dati predikat ne postane true
`SkipWhile`	Ignoriše elemente iz ulazne sekvence sve dok dati predikat ne postane true, a onda preuzima ostale
`Distinct`	Vraća kolekciju bez duplikata

Tabela 3. Operatori projekcije

Metoda	Opis
`Select`	Transformiše svaki ulazni element pomoću datog lambda izraza
`SelectMany`	Transformiše svaki ulazni element, a zatim spaja dobijene podsekvence u jednu sekvencu

Tabela 4. Operatori za spajanje

Metoda	Opis
`Join`	Primenjuje strategiju pretraživanja da bi upario elemente iz dve kolekcije, dajući kao rezultat ravan skup
`GroupJoin`	Isto kao gore, ali daje *hijerarhijski* skup
`Zip`	Nabraja dve sekvence element po element, vraćajući sekvencu koja primenjuje funkciju na svaki par elemenata

Tabela 5. Operatori za promenu redosleda

Metoda	Opis
`OrderBy, ThenBy`	Vraća date elemente sortirane po rastućem redosledu
`OrderByDescending, ThenByDescending`	Vraća date elemente sortirane po opadajućem redosledu
`Reverse`	Vraća date elemente obrnutim redosledom

Tabela 6. Operatori za grupisanje

Metoda	Opis
`GroupBy`	Grupiše sekvencu u podsekvence

Tabela 7. Operatori za skupove

Metoda	Opis
`Concat`	Nadovezuje dve sekvence
`Union`	Nadovezuje dve sekvence uz uklanjanje duplikata
`Intersect`	Vraća elemente koji postoje u obe sekvence
`Except`	Vraća elemente koji postoje u prvoj sekvenci, ali ne u drugoj

Tabela 8. Operatori koji vraćaju pojedinačne elemente

Metoda	Opis
`First, FirstOrDefault`	Vraća prvi element sekvence, ili prvi element koji zadovoljava dati predikat
`Last, LastOrDefault`	Vraća poslednji element sekvence, ili poslednji element koji zadovoljava dati predikat
`Single, SingleOrDefault`	Ekvivalentno metodi `First`/`FirstOrDefault`, ali generiše izuzetak ukoliko ima više od jednog poklapanja
`ElementAt, ElementAtOrDefault`	Vraća element koji se nalazi na zadatoj poziciji
`DefaultIfEmpty`	Vraća sekvencu s jednom vrednošću, čija je vrednost `null` odnosno `default(TSource)` ako data sekvenca nema nijedan element

Tabela 9. Agregatni operatori

Metoda	Opis
`Count, LongCount`	Vraća ukupan broj elemenata u ulaznoj sekvenci, ili broj elemenata koji zadovoljavaju dati predikat
`Min, Max`	Vraća najmanji odnosno najveći element u sekvenci
`Sum, Average`	Izračunava numerički zbir odnosno prosečnu vrednost svih elemenata u sekvenci
`Aggregate`	Obavlja namensko agregiranje

Tabela 10. Kvalifikatori

Metoda	Opis
`Contains`	Vraća `true` ako ulazna sekvenca sadrži dati element
`Any`	Vraća `true` ako bilo koji element zadovoljava dati predikat
`All`	Vraća `true` ako svi elementi zadovoljavaju dati predikat
`SequenceEqual`	Vraća `true` ako druga sekvenca ima identične elemente kao ulazna sekvenca

Tabela 11. Operatori konverzije (uvoženje)

Metoda	Opis
`OfType`	Konvertuje `IEnumerable` u `IEnumerable<T>`, odbacujući elemente pogrešnog tipa
`Cast`	Konvertuje `IEnumerable` u `IEnumerable<T>`, generišući izuzetak ukoliko ima elemenata pogrešnog tipa

Tabela 12. Operatori konverzije (izvoženje)

Metoda	Opis
`ToArray`	Konvertuje `IEnumerable<T>` u `T[]`
`ToList`	Konvertuje `IEnumerable<T>` u `List<T>`
`ToDictionary`	Konvertuje `IEnumerable<T>` u `Dictionary<TKey,TValue>`
`ToLookup`	Konvertuje `IEnumerable<T>` u `ILookup<TKey,TElement>`
`AsEnumerable`	Konvertuje naniže u `IEnumerable<T>`
`AsQueryable`	Eksplicitno konvertuje ili konvertuje u `IQueryable<T>`

Tabela 13. Operatori za generisanje

Metoda	Opis
`Empty`	Pravi praznu sekvencu
`Repeat`	Pravi sekvencu sa elementima koji se ponavljaju
`Range`	Pravi sekvencu celih brojeva

Ulančavanje operatora za upite

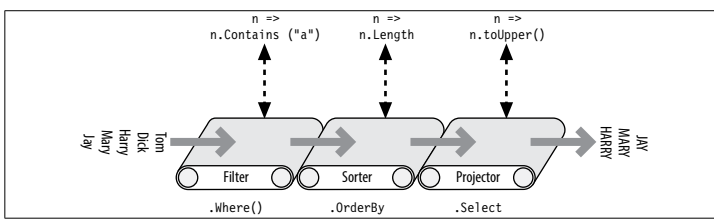
Da biste napravili složenije upite, možete ulančati operatore za upite. Na primer, sledeći upit izdvaja sve znakovne nizove koji sadrže slovo *a*, sortira ih po dužini, a zatim konvertuje rezultate u velika slova:

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

IEnumerable<string> query = names
  .Where (n => n.Contains ("a"))
  .OrderBy (n => n.Length)
  .Select (n => n.ToUpper());
foreach (string name in query)
  Console.Write (name + "|");

// REZULTAT: JAY|MARY|HARRY|
```

`Where`, `OrderBy` i `Select` su standardni operatori za upite koji se preslikavaju u proširene metode klase `Enumerable`. Operator `Where` daje filtriranu verziju ulazne sekvence; `OrderBy` daje sortiranu verziju svoje ulazne sekvence; `Select` daje sekvencu u kojoj se svaki ulazni element transformiše ili *projektuje* pomoću datog lambda izraza (`n.ToUpper()`, u ovom slučaju). Podaci teku sleva nadesno kroz lanac operatora, tako da se prvo filtriraju, zatim sortiraju i, na kraju, projektuju. Krajnji rezultat podseća na pokretnu proizvodnu traku, kao što je prikazano na slici 6.



Slika 6. Ulančavanje operatora za upite

Operatori celim tokom obezbeđuju odloženo izvršavanje, pa nema nikakvog filtriranja, sortiranja ni projektovanja sve dok se ne učitaju svi elementi za obradu u upitu.

Izrazi upita

Dosad smo pisali upite tako što smo pozivali proširene metode u klasi `Enumerable`. U ovoj knjizi to opisujemo kao fluentnu sintaksu. C# nudi i specijalnu jezičku podršku za pisanje upita, a to su izrazi koji čine upite, tj. *izrazi upita* (engl. *query expressions*). Evo prethodnog upita napisanog u obliku izraza:

```
IEnumerable<string> query =
  from n in names
  where n.Contains ("a")
  orderby n.Length
  select n.ToUpper();
```

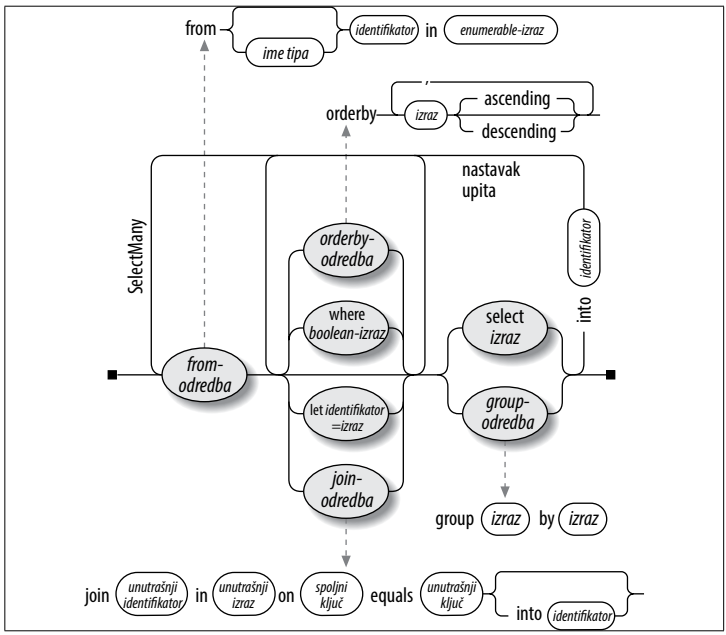
Izraz upita uvek počinje odredbom `from`, a završava se odredbom `select` ili `group`. Pomoću odredbe `from` deklariše se *promenljiva skupa*, engl. *range variable* (u ovom slučaju, `n`), koju možete smatrati promenljivom koja redom preuzima jedan po jedan element iz kolekcije – slično kao `foreach`. Slika 7 ilustruje kompletnu sintaksu.

Napomena

Ako poznajete sintaksu SQL-a, sintaksa izraza za LINQ upite – sa odredbom `from` na početku i odredbom `select` na kraju – možda će vam delovati čudno. U stvari, sintaksa izraza za upite je logičnija jer se navedene odredbe pojavljuju *redosledom kojim se izvršavaju*. To omogućava da vam Visual Studio pomogne pomoću sistema IntelliSense dok kucate, a i pojednostavljuje pravila vidljivosti za podupite.

Kompajler obrađuje izraze upita tako što ih prevodi u fluentnu sintaksu. On to radi prilično mehanički – slično kao što prevodi naredbe `foreach` u pozive metoda `GetEnumerator` i `MoveNext`:

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



Slika 7. Sintaksa upita napisanog u obliku izraza

Zatim se razrešavaju operatori `Where`, `OrderBy` i `Select`, pri čemu se primenjuju ista pravila kao što su ona koja bi važila da je upit bio napisan uz korišćenje fluentne sintakse. U ovom slučaju, oni se vezuju za proširene metode u klasi `Enumerable` (pod uslovom da ste učitali imenski prostor `System.Linq`) zato što `names` implementira `IEnumerable<string>`. Međutim, kompajler ne daje prednost klasi `Enumerable` kada prevodi sintaksu upita. O kompajleru možete da razmišljate kao da mehanički ubacuje reči *Where*, *OrderBy* i *Select* u naredbu, a zatim je prevodi kao da ste sami otkucali imena metoda. Zahvaljujući tome, razrešavanje metoda je fleksibilno – operatori u LINQ upitima za SQL i Entity Framework, na primer, preslikavaju se u proširene metode klase `Queryable`.

Poređenje upita sa izrazima i fluentnih upita

I upiti sa izrazima i fluentni upiti imaju svoje prednosti.

Izrazi koji se koriste za upite podržavaju samo mali podskup operatora za upite, i to:

```
Where, Select, SelectMany
OrderBy, ThenBy, OrderByDescending, ThenByDescending
GroupBy, Join, GroupJoin
```

Upite u kojima se koriste drugi operatori, morate ili cele napisati uz primenu fluentne sintakse ili ih sastaviti kombinovanjem sintaksi; na primer:

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

IEnumerable<string> query =
  from n in names
  where n.Length == names.Min (n2 => n2.Length)
  select n;
```

Ovaj upit vraća imena čije dužine odgovaraju najkraćem imenu („Tom“ i „Jay“). Podupit (ispisan podebljano) izračunava najmanju dužinu imêna, i daje rezultat 3. Za podupit moramo koristiti fluentnu sintaksu pošto sintaksa izraza za upite ne podržava operator `Min`. Međutim, i dalje možemo koristiti sintaksu izraza za spoljni upit.

Osnovna prednost sintakse sa izrazima jeste to što može značajno pojednostaviti upite koji sadrže sledeće:

- odredbu `let` za uvođenje nove promenljive uz promenljivu skupa;
- više generatora (`SelectMany`) praćenih referencom na spoljnu promenljivu skupa;
- ekvivalent naredbe `Join` ili `GroupJoin`, praćen referencom na spoljnu promenljivu skupa.

Rezervisana reč let

Rezervisana reč `let` uvodi novu promenljivu uz promenljivu skupa. Na primer, pretpostavimo da želimo da izlistamo sva imena koja su, ne računajući samoglasnike, duža od dva znaka:

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

IEnumerable<string> query =
  from n in names
  let vowelless = Regex.Replace (n, "[aeiou]", "")
  where vowelless.Length > 2
  orderby vowelless
  select n + " - " + vowelless;
```

Rezultat ovog upita je:

```
Dick - Dck
Harry - Hrry
Mary - Mry
```

Odredba `let` obavlja izračunavanja nad svakim elementom, ne menjajući originalni element. U našem upitu, odredbe koje slede (`where`, `orderby` i `select`) mogu da pristupe i promenljivoj `n` i promenljivoj `vowelless`. Jedan upit može da sadrži proizvoljan broj odredaba `let`, i one mogu biti proizvoljno raspoređene s dodatnim odredbama `where` i `join`.

Kompajler prevodi rezervisanu reč `let` tako što projektuje u privremen anonimni tip koji sadrži i originalne i transformisane elemente:

```
IEnumerable<string> query = names
  .Select (n => new
    {
      n = n,
      vowelless = Regex.Replace (n, "[aeiou]", "")
    }
  )
  .Where (temp0 => (temp0.vowelless.Length > 2))
  .OrderBy (temp0 => temp0.vowelless)
  .Select (temp0 => ((temp0.n + " - ") + temp0.vowelless))
```

Nastavljanje upita

Ukoliko želite da dodate odredbe iza odredbe `select` ili `group`, morate upotrebiti rezervisanu reč `into` kako biste „nastavili“ upit. Na primer:

```
from c in "The quick brown tiger".Split()
select c.ToUpper() into upper
where upper.StartsWith ("T")select upper

// REZULTAT: "THE", "TIGER"
```

Iza odredbe `into`, promenljiva skupa koja joj prethodi nalazi se van opsega vidljivosti.

Upite s rezervisanom rečju `into`, kompajler samo prevodi u duži lanac operatora:

```
"The quick brown tiger".Split()
  .Select (c => c.ToUpper())
  .Where (upper => upper.StartsWith ("T"))
```

(Upit izostavlja završnu naredbu `Select(upper=>upper)` zato što je redundantna.)

Više generatora elemenata

Upit može da sadrži više generatora elemenata (odredaba `from`). Recimo:

```
int[] numbers = { 1, 2, 3 };
string[] letters = { "a", "b" };

IEnumerable<string> query = from n in numbers
                            from l in letters
                            select n.ToString() + l;
```

Rezultat je unakrsni proizvod, poput onog koji biste dobili pomoću ugnežđenih petlji `foreach`:

```
"1a", "1b", "2a", "2b", "3a", "3b"
```

Kada u upitu ima više od jedne odredbe `from`, kompajler poziva `SelectMany`:

```
IEnumerable<string> query = numbers.SelectMany (
  n => letters,
  (n, l) => (n.ToString() + l));
```

`SelectMany` funkcioniše kao ugnežđena petlja. Nabraja svaki element u izvornoj kolekciji (`numbers`), transformišući svaki od njih pomoću prvog lambda izraza (`letters`). To generiše niz podsekvenci koje `SelectMany` zatim nabraja. Konačni izlazni elementi određuju se pomoću drugog lambda izraza (`n.ToString()+l`).

Ako naknadno primenite odredbu `where`, možete filtrirati unakrsni proizvod i projektovati rezultat sličan onom koji se dobija kada se koristi odredba `join`:

```
string[] players = { "Tom", "Jay", "Mary" };
IEnumerable<string> query =
  from name1 in players
  from name2 in players
  where name1.CompareTo (name2) < 0
  orderby name1, name2
  select name1 + " vs " + name2;

REZULTAT: { "Jay vs Mary", "Jay vs Tom", "Mary vs Tom" }
```

Prevođenje ovog upita u fluentnu sintaksu složenije je – potrebna je privremena anonimna projekcija. Mogućnost da se ovo prevođenje obavi automatski jedna je od ključnih prednosti izraza za upite.

Izraz u drugom generatoru može da koristi prvu promenljivu opsega:

```
string[] fullNames =
  { "Anne Williams", "John Fred Smith", "Sue Green" };

IEnumerable<string> query =
   from fullName in fullNames
   from name in fullName.Split()
   select name + " came from " + fullName;

Anne came from Anne Williams
Williams came from Anne Williams
John came from John Fred Smith
```

Ovo funkcioniše zato što izraz `fullName.Split` pravi sekvencu (niz znakovnih nizova).

Više generatora intenzivno se koriste u upitima u baze podataka, za uprošćavanje odnosa roditelj–dete i za ručna spajanja.

Spajanje

LINQ ima tri operatora za *spajanje*, a najvažniji su `Join` i `GroupJoin` koji spajaju elemente na osnovu pretraživanja po ključevima. `Join` i `GroupJoin` podržavaju samo podskup funkcionalnosti koje dobijate pomoću više generatora/`SelectMany`, ali su pogodniji za lokalne upite pošto koriste strategiju pretraživanja zasnovanu na heš-tabelama umesto da izvršavaju ugnežđene petlje.

(Sa upitima tipa LINQ to SQL i Entity Framework, operatori za spajanje nisu u prednosti nad više generatora.)

`Join` i `GroupJoin` podržavaju samo *jednakovredne spojeve*, engl. *equi-joins* (to jest, u uslovu spajanja mora da se koristi operator jednakosti). Postoje dve metode: `Join` i `GroupJoin`. `Join` daje ravan skup rezultata, a `GroupJoin` hijerarhijski.

Evo sintakse izraza za upite, za ravno spajanje:

```
from outer-var in outer-sequence
join inner-var in inner-sequence
  on outer-key-expr equals inner-key-expr
```

Na primer, ako je data sledeća kolekcija:

```
var customers = new[]
{
  new { ID = 1, Name = "Tom" },
  new { ID = 2, Name = "Dick" },
  new { ID = 3, Name = "Harry" }
};
var purchases = new[]
{
  new { CustomerID = 1, Product = "House" },
  new { CustomerID = 2, Product = "Boat" },
```

```
    new { CustomerID = 2, Product = "Car" },
    new { CustomerID = 3, Product = "Holiday" }
};
```

spajanje možemo obaviti ovako:

```
IEnumerable<string> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  select c.Name + " bought a " + p.Product;
```

Kompajler prevodi ovaj kôd u:

```
customers.Join (              // spoljna kolekcija
  purchases,                  // unutrašnja kolekcija
  c => c.ID,                  // birač spoljnog ključa
  p => p.CustomerID,          // birač unutrašnjeg ključa
  (c, p) =>                   // birač rezultata
    c.Name + " bought a " + p.Product
);
```

Evo rezultata:

```
Tom bought a House
Dick bought a Boat
Dick bought a Car
Harry bought a Holiday
```

S lokalnim sekvencama, operatori `Join` i `GroupJoin` efikasnije obrađuju velike kolekcije od `SelectMany` zato što prvo učitaju unutrašnju sekvencu u heširanu tabelu ključeva. Međutim, kad je reč o upitima u baze podataka, isti rezultat možete podjednako efikasno dobiti na sledeći način:

```
from c in customers
from p in purchases
where c.ID == p.CustomerID
select c.Name + " bought a " + p.Product;
```

Operator GroupJoin

`GroupJoin` obavlja isto što i `Join`, ali – umesto ravnog – daje hijerarhijski rezultat, grupisan po svakom spoljnom elementu.

U slučaju operatora `GroupJoin`, sintaksa izraza za upite ista je kao za `Join`, ali se iza izraza koristi rezervisana reč `into`. Evo osnovnog primera, u kome se koriste kolekcije `customers` i `purchases` koje smo napravili u prethodnom odeljku:

```
IEnumerable<IEnumerable<Purchase>> query =
  from c in customers
  join p in purchases on c.ID equals p.CustomerID
  into custPurchases
  select custPurchases;   // custPurchases je sekvenca
```

Napomena

Odredba `into` se prevodi u `GroupJoin` samo kada se pojavljuje neposredno iza odredbe `join`. Ukoliko se nalazi iza odredbe `select` ili `group`, označava *nastavak upita*. Ova dva načina primene rezervisane reči `into` prilično su različiti, mada imaju jednu zajedničku osobinu: oba uvode novu promenljivu upita.

Rezultat je sekvenca sekvenci, koju možemo nabrojati ovako:

```
foreach (IEnumerable<Purchase> purchaseSequence in query)
  foreach (Purchase p in purchaseSequence)
    Console.WriteLine (p.Description);
```

Međutim, to nije mnogo korisno pošto `outerSeq` nema referencu ka spoljnom kupcu. U projekciji ćete češće referencirati promenljivu spoljnog skupa:

```
from c in customers
join p in purchases on c.ID equals p.CustomerID
into custPurchases
select new { CustName = c.Name, custPurchases };
```

Isti rezultat bismo mogli dobiti (ali manje efikasno, za lokalne upite) projektovanjem u anoniman tip koji uključuje podupit:

```
from c in customers
select new
{
```

```
    CustName = c.Name,
    custPurchases =
      purchases.Where (p => c.ID == p.CustomerID)
}
```

Operator Zip

`Zip` je najjednostavniji operator za spajanje. On nabraja dve sekvence element po element (kao patent-zatvarač), vraćajući sekvencu zasnovanu na primeni neke funkcije na svaki par elemenata. Na primer:

```
int[] numbers = { 3, 5, 7 };
string[] words = { "three", "five", "seven", "ignored" };
IEnumerable<string> zip =
  numbers.Zip (words, (n, w) => n + "=" + w);
```

daje sekvencu sa sledećim elementima:

```
3=three
5=five
7=seven
```

Višak elemenata u bilo kojoj od dve ulazne sekvence ignoriše se. Upiti u baze podataka ne podržavaju `Zip`.

Sortiranje

Rezervisana reč `orderby` sortira sekvencu. Možete zadati proizvoljan broj izraza po kojima će se sekvenca sortirati:

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

IEnumerable<string> query = from n in names
                            orderby n.Length, n
                            select n;
```

Ovaj kôd sortira prvo po dužini a zatim po imenu, pa je rezultat:

```
Jay, Tom, Dick, Mary, Harry
```

Kompajler prevodi prvi `orderby` izraz u poziv funkcije `OrderBy`, a dalje izraze u poziv funkcije `ThenBy`:

```
IEnumerable<string> query = names
  .OrderBy (n => n.Length)
  .ThenBy (n => n)
```

Operator `ThenBy` ne zamenjuje prethodno sortiranje već ga pročišćava.

Iza svakog `orderby` izraza možete dodati rezervisanu reč `descending`:

```
orderby n.Length descending, n
```

To se prevodi u:

```
.OrderByDescending (n => n.Length).ThenBy (n => n)
```

NAPOMENA

Operatori za sortiranje vraćaju prošireni oblik interfejsa `IEnumerable<T>`, zvan `IOrderedEnumerable<T>`. Taj interfejs definiše dodatnu funkcionalnost potrebnu operatoru `ThenBy`.

Grupisanje

`GroupBy` organizuje ravnu ulaznu sekvencu u sekvencu koju čini više *grupa*. Recimo, sledeći kôd grupiše sekvencu imena po njihovim dužinama:

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

var query = from name in names
            group name by name.Length;
```

Kompajler prevodi ovaj upit u:

```
IEnumerable<IGrouping<int,string>> query =
  names.GroupBy (name => name.Length);
```

Evo kako da prikažete rezultate ovog upita:

```
foreach (IGrouping<int,string> grouping in query)
{
  Console.Write ("\r\n Length=" + grouping.Key + ":");
```

```
    foreach (string name in grouping)
      Console.Write (" " + name);
}

  Length=3: Tom Jay
  Length=4: Dick Mary
  Length=5: Harry
```

`Enumerable.GroupBy` radi tako što učitava ulazne elemente u privremeni rečnik lista, pa svi elementi sa istim ključem završavaju u istoj podlisti. Zatim formira sekvencu čiji su elementi *grupe*. Grupa je sekvenca koja ima ključ (svojstvo `Key`):

```
public interface IGrouping <TKey,TElement>
 : IEnumerable<TElement>, IEnumerable
{
  // Ključ važi za podsekvencu kao celinu
  TKey Key { get; }
}
```

Elementi svake grupe standardno su netransformisani ulazni elementi, osim kada zadate argument `elementSelector`. Sledeći kôd projektuje svaki ulazni element u velika slova:

```
from name in names
group name.ToUpper() by name.Length
```

što se prevodi u:

```
names.GroupBy (
  name => name.Length,
  name => name.ToUpper() )
```

Potkolekcije se ne formiraju po redosledu ključa. Operator `GroupBy` *ne sortira* ništa (u stvari, on čuva originalni redosled). Da biste sortirali, morate dodati operator `OrderBy` (što znači da prvo morate dodati odredbu `into`, pošto se sa `group by` obično završava upit):

```
from name in names
group name.ToUpper() by name.Length into grouping
orderby grouping.Key
select grouping
```

Nastavljanje upita često se koristi u upitima tipa `group by`. Sledeći upit filtrira grupe u kojima postoje tačno dva poklapanja:

```
from name in names
group name.ToUpper() by name.Length into grouping
where grouping.Count() == 2
select grouping
```

Napomena

Odredba `where` iza `group by` ekvivalentna je odredbi `HAVING` u SQL-u. Primenjuje se na svaku podsekvencu ili na celu grupu, a ne na pojedinačne elemente.

Operatori OfType i Cast

Operatori `OfType` i `Cast` prihvataju negeneričku kolekciju `IEnumerable` i daju generičku sekvencu `IEnumerable<T>` nad kojom možete naknadno izvršiti upit:

```
var classicList = new System.Collections.ArrayList();
classicList.AddRange ( new int[] { 3, 4, 5 } );
IEnumerable<int> sequence1 = classicList.Cast<int>();
```

Ovo je korisno jer vam omogućava da izvršite upite nad kolekcijama napisanim pre pojave verzije C# 2.0 (kada je uveden `IEnumerable<T>`), kao što je `ControlCollection` u `System.Windows.Forms`.

`Cast` i `OfType` razlikuju se po ponašanju kada naiđu na ulazni element nekompatibilnog tipa: `Cast` generiše izuzetak, a `OfType` ignoriše nekompatibilni element.

Pravila za kompatibilnost elemenata slede ona za operator `is` u jeziku C#. Evo interne implementacije operatora `Cast`:

```
public static IEnumerable<TSource> Cast <TSource>
             (IEnumerable source)
{
  foreach (object element in source)
    yield return (TSource)element;
}
```

C# podržava operator `Cast` u izrazima za upite – samo umetnite tip elementa odmah iza rezervisane reči `from`:

```
from int x in classicList ...
```

Ovo se prevodi u:

```
from x in classicList.Cast <int>() ...
```

Dinamičko povezivanje

Dinamičko povezivanje odlaže *povezivanje* (engl. *binding*) – proces razrešavanja tipova, članova i operacija – od trenutka prevođenja do trenutka izvršavanja. Dinamičko povezivanje je uvedeno u verziju C# 4.0, i korisno je kada u vreme prevođenja *vi* znate da postoji određena funkcija, član ili operacija, ali *kompajler* ne zna. To se obično dešava kada radite i sa dinamičkim jezicima (kao što je IronPython) i COM, i u slučajevima kada biste inače možda koristili refleksiju.

Dinamički tip se deklariše pomoću kontekstualne rezervisane reči `dynamic`:

```
dynamic d = GetSomeObject();
d.Quack();
```

Dinamički tip saopštava kompajleru da se opusti. Očekujemo da u vreme izvršavanja `d` ima metodu `Quack` – ali to ne možemo da dokažemo statički. Pošto je `d` dinamički tip, kompajler odlaže povezivanje metode `Quack` za `d` do trenutka izvršavanja. Da bismo razumeli šta to znači, moramo razlikovati *statičko povezivanje* i *dinamičko povezivanje*.

Poređenje statičkog i dinamičkog povezivanja

Uobičajeni primer povezivanja je preslikavanje (mapiranje) nekog imena u određenu funkciju prilikom prevođenja izraza. Da bi preveo sledeći izraz, kompajler mora da pronađe implementaciju metode po imenu `Quack`:

```
d.Quack();
```

Pretpostavimo da je `Duck` statički tip od `d`:

```
Duck d = ...
d.Quack();
```

U najjednostavnijem slučaju, kompajler obavlja povezivanje tako što traži besparametarsku metodu `Quack` u klasi `Duck`. Ukoliko to ne uspe, kompajler širi potragu na metode koje prihvataju opcione parametre, metode osnovne klase koju klasa `Duck` nasleđuje, i proširene metode koje prihvataju `Duck` kao svoj prvi parametar. Ako ništa od toga ne pronađe, dobićete grešku pri prevođenju. Bez obzira na to koja metoda se poveže, ključno je to da povezivanje obavlja kompajler, a ono potpuno zavisi od statičkog poznavanja tipova operanada (u ovom slučaju, `d`). Zbog toga je to *statičko povezivanje*.

Promenimo sada statički tip promenljive `d` u `object`:

```
object d = ...
d.Quack();
```

Kada pozovemo metodu `Quack` dobijamo grešku pri prevođenju, a razlog je sledeći: mada vrednost sačuvana u `d` može da sadrži metodu po imenu `Quack`, kompajler to ne može da zna pošto je jedina informacija koju ima tip promenljive – u ovom slučaju, `object`. Ali, sada ćemo promeniti statički tip promenljive `d` u `dynamic`:

```
dynamic d = ...
d.Quack();
```

Tip `dynamic` je kao tip `object` – ništa ne govori o tipu. Razlika je to što vam omogućava da ga koristite na načine koji nisu poznati u vreme prevođenja. Dinamički objekat se povezuje u trenutku izvršavanja na osnovu svog tipa u trenutku izvršavanja a ne onog u trenutku prevođenja. Kada kompajler vidi dinamički povezan izraz (što je, u opštem slučaju, izraz koji sadrži bilo koju vrednost tipa `dynamic`), on ga samo pakuje tako da se povezivanje može obaviti kasnije, u trenutku izvršavanja.

U trenutku izvršavanja, ako dinamički objekat implementira interfejs `IDynamicMetaObjectProvider`, taj interfejs se koristi za povezivanje. U suprotnom, povezivanje se obavlja na gotovo isti način kao što bi se

obavljalo kada bi kompajler znao tip dinamičkog objekta u trenutku izvršavanja. Pomenute dve alternative zovu se *namensko povezivanje* (engl. *custom binding*) i *jezičko povezivanje* (engl. *language binding*).

Namensko povezivanje

Namensko povezivanje se obavlja kada dinamički objekat implementira interfejs `IDynamicMetaObjectProvider` (IDMOP). Mada možete implementirati IDMOP na tipove koje pišete na jeziku C# – i to je korisno – češće ćete koristiti IDMOP objekat iz dinamičkog jezika implementiranog u .NET za Dynamic Language Runtime (DLR), kao što su IronPython ili IronRuby. Objekti iz tih jezika implicitno implementiraju IDMOP kao sredstvo za direktno određivanje značenja operacija koje se obavljaju nad njima. Evo jednostavnog primera:

```
using System;
using System.Dynamic;

public class Test
{
  static void Main()
  {
    dynamic d = new Duck();
    d.Quack();    // Pozvana je metoda Quack
    d.Waddle();   // Pozvana je metoda Waddle
    }
  }
  public class Duck : DynamicObject
  {
    public override bool TryInvokeMember (
      InvokeMemberBinder binder, object[] args,
      out object result)
     {
       Console.WriteLine (binder.Name + " was called");
       result = null;
       return true;
  }
}
```

Klasa `Duck` u stvari nema metodu `Quack`. Umesto nje, ona koristi namensko povezivanje da bi presrela i protumačila sve pozive te metode.

Detaljnije razmatranje namenskog povezivanja dato je u poglavlju 20 knjige *C# 5.0 za programere*.

Jezičko povezivanje

Do jezičkog povezivanja dolazi kada dinamički objekat ne implementira interfejs `IDynamicMetaObjectProvider`. Ono je korisno kada radite sa loše definisanim tipovima ili ograničenjima svojstvenim .NET-ovom sistemu tipova. Pri korišćenju numeričkih tipova, tipičan problem je to što oni nemaju zajednički interfejs. Već smo videli da se metode mogu povezati dinamički; isto važi i za operatore:

```
static dynamic Mean (dynamic x, dynamic y)
{
  return (x + y) / 2;
}
static void Main()
{
  int x = 3, y = 4;
  Console.WriteLine (Mean (x, y));
}
```

Prednost je očigledna – ne morate duplirati kôd posebno za svaki numerički tip. Međutim, gubite bezbednost statičkih tipova, rizikujući da se generišu izuzeci pri izvršavanju a ne greške pri prevođenju.

Napomena

Dinamičko povezivanje zanemaruje statičku bezbednost tipova, ali ne i bezbednost tipova pri izvršavanju. Za razliku od refleksije, dinamičkim povezivanjem ne možete zaobići pravila pristupanja elementima tipova.

Jezičko povezivanje u vreme izvršavanja projektovano je tako da se ponaša najsličnije moguće statičkom povezivanju kada bi tipovi dinamičkih objekata u vreme izvršavanja bili poznati u vreme prevođenja. U našem prethodnom primeru, program bi se ponašao isto i da smo eksplicitno napisali kôd da `Mean` radi s tipom `int`. Najznačajniji izuzetak u sličnosti statičkog i dinamičkog povezivanja odnosi se na proširene metode, koje razmatramo u odeljku „Funkcije koje se ne mogu pozivati“, na strani 179.

Napomena

Dinamičko povezivanje loše utiče i na performanse. Međutim, zahvaljujući DLR-ovim mehanizmima keširanja, optimizuju se ponovljeni pozivi istih dinamičkih izraza – što vam omogućava da efikasno pozivate dinamičke izraze u petlji. Zahvaljujući toj optimizaciji, uobičajeno opterećenje za jednostavan dinamički izraz na današnjem hardveru manje je od 100 ns.

Izuzetak RuntimeBinderException

Ako neki član ne uspe da se poveže, generiše se `RuntimeBinderException`, koji možete smatrati greškom pri prevođenju u vreme izvršavanja:

```
dynamic d = 5;
d.Hello();        // generiše RuntimeBinderException
```

Ovaj izuzetak se generiše zato što tip `int` nema metodu `Hello`.

Predstavljanje dinamičkog tipa u izvršnom okruženju

Postoji jaka ekvivalencija između tipova `dynamic` i `object`. Izvršno okruženje tretira sledeći izraz kao istinit:

```
typeof (dynamic) == typeof (object)
```

Ovaj princip važi i za konstruisane tipove i tipove niza:

```
typeof (List<dynamic>) == typeof (List<object>)
typeof (dynamic[]) == typeof (object[])
```

Kao i referenca na objekat, i dinamička referenca može da upućuje na objekat bilo kog tipa (izuzev na pokazivačke tipove, engl. *pointer types*):

```
dynamic x = "hello";
Console.WriteLine (x.GetType().Name); // Znakovni niz

x = 123; // Nema greške (bez obzira na istu promenljivu)
Console.WriteLine (x.GetType().Name); // Int32
```

Strukturno, nema razlike između reference na objekat i dinamičke reference. Dinamička referenca samo omogućava da se nad objektom na koji upućuje obavljaju dinamičke operacije. Tip `object` možete konvertovati u `dynamic` da biste obavili bilo koju dinamičku operaciju na tipu `object`:

```
object o = new System.Text.StringBuilder();
dynamic d = o;
d.Append ("hello");
Console.WriteLine (o);   // hello
```

Dinamičke konverzije

Tip `dynamic` može se implicitno konvertovati u druge tipove i iz njih. Da bi konverzija uspela, tip dinamičkog objekta u vreme izvršavanja mora biti u stanju da se implicitno konvertuje u odredišni statički tip.

Sledeći primer generiše izuzetak `RuntimeBinderException` pošto se `int` ne može implicitno konvertovati u `short`:

```
int i = 7;
dynamic d = i;
long l = d;        // OK - implicitna konverzija funkcioniše
short j = d;       // generiše RuntimeBinderException
```

Poređenje tipova var i dynamic

Tipovi `var` i `dynamic` su na prvi pogled slični, ali je razlika velika:

```
var kaže, "Neka kompajler ustanovi tip."
dynamic kaže, "Neka izvršno okruženje ustanovi tip."
```

Evo i primera:

```
dynamic x = "hello";   // Statički tip je dynamic
var y = "hello";       // Statički tip je string
int i = x;             // Greška pri izvršavanju
int j = y;             // Greška pri prevođenju
```

Dinamički izrazi

Dinamički se mogu pozivati polja, svojstva, metode, događaji, konstruktori, indekseri, operatori i konverzije.

Nije dozvoljeno da se upotrebi rezultat dinamičkog izraza povratnog tipa `void` – kao što je slučaj i sa izrazom statičkog tipa. Razlika je to što se greška javlja u vreme izvršavanja.

Izrazi s dinamičkim operandima obično su i sami dinamički, pošto se efekat izostanka informacije o tipu prenosi dalje:

```
dynamic x = 2;
var y = x * 3;          // Statički tip promenljive y je
                        // dynamic
```

Postoji nekoliko očiglednih izuzetaka od ovog pravila. Prvo, eksplicitnim konvertovanjem dinamičkog izraza u statički tip dobija se statički izraz. Drugo, pozivanje konstruktora uvek rezultuje statičkim izrazima – čak i kada se konstruktor poziva s dinamičkim argumentima.

Uz to, postoji i nekoliko graničnih slučajeva gde je izraz koji sadrži dinamički argument statičan, među kojima su prosleđivanje indeksa nizu i izrazi koji generišu delegate.

Razrešavanje preklapanja dinamičkog člana

Školski primer upotrebe tipa `dynamic` jeste i dinamički *prijemnik* (engl. *receiver*). To znači da je dinamički objekat prijemnik poziva dinamičke funkcije:

```
dynamic x = ...;
x.Foo (123);            // x je prijemnik
```

Međutim, dinamičko povezivanje nije ograničeno na prijemnike: argumenti metode se također mogu dinamički povezati. Funkcija s dinamičkim argumentima poziva se da bi se razrešavanje preklapanja odložilo od trenutka prevođenja na trenutak izvršavanja:

```
class Program
{
  static void Foo (int x) { Console.WriteLine ("1"); }
  static void Foo (string x) { Console.WriteLine ("2"); }

  static void Main()
  {
    dynamic x = 5;
    dynamic y = "watermelon";

    Foo (x);     // 1
    Foo (y);     // 2
  }
}
```

Razrešavanje preklapanja u vreme izvršavanja zove se i višestruko raspoređivanje (engl. *multiple dispatch*) i korisno je pri implementaciji projektnih obrazaca (engl. *design patterns*) kao što je *visitor*.

Ako u proces nije uključen dinamički prijemnik, kompajler može da obavi osnovne provere statički kako bi ustanovio da li će dinamički poziv uspeti: on proverava postoji li funkcija ispravnog imena i sa odgovarajućim brojem parametara. Ukoliko nema takve, dobićete grešku pri prevođenju.

Ako se funkcija pozove s mešavinom dinamičkih i statičkih argumenata, konačan izbor metode zavisiće od mešavine dinamičkih i statičkih odluka o povezivanju:

```
static void X(object x, object y) {Console.Write("oo");}
static void X(object x, string y) {Console.Write("os");}
static void X(string x, object y) {Console.Write("so");}
static void X(string x, string y) {Console.Write("ss");}
static void Main()
{
```

```
    object o = "hello";
    dynamic d = "goodbye";
    X (o, d);                  // os
}
```

Poziv metode `X(o,d)` povezan je dinamički zato što je jedan njen argument, `d`, tipa `dynamic`. Ali, pošto je argument `o` statički poznat, povezivanje – bez obzira na to što se odvija dinamički – iskoristiće tu činjenicu. U ovom slučaju, razrešavanje preklapanja će izabrati drugu implementaciju metode `X` zbog statičkog tipa `o` i tipa `d` koji je poznat tek u trenutku izvršavanja. Drugim rečima, kompajler je „onoliko statički koliko uopšte može biti."

Funkcije koje se ne mogu pozivati

Neke funkcije se ne mogu pozivati dinamički. Ne možete pozvati:

- proširene metode (preko sintakse proširenih metoda)
- članove interfejsa (preko interfejsa)
- osnovne članove koje sakriva potklasa

Razlog je to što su za dinamičko povezivanje potrebne dve informacije: ime funkcije koja se poziva i objekat za koji se ona poziva. Međutim, u sva tri slučaja funkcija koje se ne mogu pozivati, učestvuje i *dodatni tip*, koji je poznat samo u vreme prevođenja. Počevši od verzije C# 5.0, nema načina da se ti dodatni tipovi zadaju dinamički.

Kada se pozivaju proširene metode, taj dodatni tip je proširena klasa, izabrana implicitno zahvaljujući direktivama `using` u vašem izvornom kodu (koje nestaju nakon prevođenja). Kada se članovi pozivaju preko interfejsa, dodatni tip se saopštava preko implicitne ili eksplicitne konverzije. (Kada je implementacija eksplicitna, u stvari je nemoguće pozvati člana a da se on ne konvertuje u člana interfejsa.) Slična situacija nastaje i pri pozivanju skrivenog osnovnog člana: morate zadati dodatni tip ili pomoću konverzije ili pomoću rezervisane reči `base` – i taj dodatni tip se gubi u vreme izvršavanja.

Atributi

Već vam je poznat pojam dodeljivanja atributa elementima programskog koda pomoću modifikatora, kao što su `virtual` ili `ref`. Ti konstrukti su ugrađeni u jezik. *Atributi* su proširiv mehanizam za dodavanje namenskih informacija elementima koda (sklopovima, tipovima, članovima, povratnim vrednostima i parametrima). Ta proširivost je korisna za servise koji su duboko integrisani u sistem tipova, a da im ne trebaju specijalne rezervisane reči ili konstrukti jezika C#.

Dobar scenario za atribute predstavlja serijalizacija – proces konvertovanja proizvoljnih objekata u određen format ili iz njega. U ovom scenariju, pomoću atributa polja može se zadati prevođenje između načina predstavljanja tog polja u jeziku C# i u izabranom formatu.

Klase za atribute

Atribut definiše klasa koja nasleđuje (direktno ili indirektno) apstraktnu klasu `System.Attribute`. Da biste atribut priključili elementu koda, zadajte ime tipa atributa u uglastim zagradama, ispred elementa koda. Na primer, evo kako ćete atributom `ObsoleteAttribute` obeležiti klasu `Foo`:

```
[ObsoleteAttribute]
public class Foo {...}
```

Kompajler prepoznaje ovaj atribut i prikazaće upozorenje ukoliko se referencira tip ili član označen kao zastareo. Po konvenciji, imena svih tipova atributa završavaju se rečju *Attribute*. C# to prepoznaje i omogućava vam da izostavite taj sufiks kada dodajete atribut:

```
[Obsolete]
public class Foo {...}
```

`ObsoleteAttribute` je tip deklarisan u imenskom prostoru `System` na sledeći način (pojednostavljeno da bi bilo kraće):

```
public sealed class ObsoleteAttribute : Attribute {...}
```

Imenovani i pozicioni parametri atributa

Atributi mogu da imaju parametre. U narednom primeru primenjujemo atribut `XmlElementAttribute` na klasu. Ovaj atribut saopštava klasi `XmlSerializer` (u imenskom prostoru `System.Xml.Serialization`) kako je objekat predstavljen u XML-u i prihvata nekoliko *parametara atributa*. Sledeći atribut preslikava klasu `CustomerEntity` u XML element po imenu `Customer`, koji pripada imenskom prostoru `http://oreilly.com`:

```
[XmlElement ("Customer", Namespace="http://oreilly.com")]
public class CustomerEntity { ... }
```

Parametri atributa mogu biti pozicioni ili imenovani. U prethodnom primeru, prvi argument je *pozicioni parametar* (engl. *positional parameter*), a drugi je *imenovani parametar* (engl. *named parameter*). Pozicioni parametri odgovaraju parametrima javnih konstruktora tipa atributa. Imenovani parametri odgovaraju javnim poljima ili javnim svojstvima tipa atributa.

Kada zadajete atribut, morate navesti pozicione parametre koji odgovaraju jednom od konstruktora atributa. Imenovani parametri nisu obavezni.

Odredišta atributa

Implicitno, odredište atributa je element koda kome atribut neposredno prethodi, što je obično tip ili član tipa. Međutim, atribute možete dodati i sklopu. Da biste to uradili, morate eksplicitno zadati odredište atributa.

Evo primera upotrebe atributa `CLSCompliant` za zadavanje usklađenosti celog bloka sa specifikacijom CLS (Common Language Specification):

```
[assembly:CLSCompliant(true)]
```

Zadavanje više atributa

Za jedan element koda može se zadati više atributa. Svaki atribut se može navesti ili unutar istog para uglastih zagrada (razdvojen zarezom od drugih atributa) ili u zasebnim parovima uglastih zagrada (a može se koristiti i kombinacija ovih načina). Sledeća dva primera su semantički identična:

```
[Serializable, Obsolete, CLSCompliant(false)]
public class Bar {...}

[Serializable] [Obsolete] [CLSCompliant(false)]
public class Bar {...}
```

Pisanje namenskih atributa

Sopstvene atribute definišete tako što napravite potklasu od `System.Attribute`. Na primer, mogli bismo da koristimo sledeći namenski atribut za obeležavanje određene metode za jedinično testiranje:

```
[AttributeUsage (AttributeTargets.Method)]
public sealed class TestAttribute : Attribute
{
  public int Repetitions;
  public string FailureMessage;


  public TestAttribute () : this (1) { }
  public TestAttribute (int repetitions)
  {
    Repetitions = repetitions;
  }
}
```

Evo kako bismo mogli da upotrebimo taj atribut:

```
class Foo

{
  [Test]
  public void Method1() { ... }
```

```
  [Test(20)]
  public void Method2() { ... }

  [Test(20, FailureMessage="Debugging Time!")]
  public void Method3() { ... }
}
```

`AttributeUsage` je i sâm atribut koji ukazuje na konstrukt (ili kombinaciju konstrukata) na koji može da se primeni namenski atribut. Nabrajanje `AttributeTargets` ima članove kao što su `Class`, `Method`, `Parameter` i `Constructor` (kao i `All`, koji kombinuje sva odredišta).

Učitavanje atributa u vreme izvršavanja

Dva su standardna načina za učitavanje atributa u vreme izvršavanja:

- Pozovite `GetCustomAttributes` za bilo koji `Type` ili `MemberInfo` objekat.
- Pozovite `Attribute.GetCustomAttribute` ili `Attribute.GetCustomAttributes`.

Poslednje dve metode su preklopljene tako da prihvataju svaki objekat dobijen postupkom refleksije koji odgovara ispravnom odredištu za atribut (`Type`, `Assembly`, `Module`, `MemberInfo` ili `ParameterInfo`).

Evo kako možemo nabrojati sve metode u prethodnoj klasi `Foo` koje imaju atribut `TestAttribute`:

```
foreach (MethodInfo mi in typeof (Foo).GetMethods()) {
  TestAttribute att = (TestAttribute)
    Attribute.GetCustomAttribute (mi, typeof (TestAttribute));

  if (att != null)
    Console.WriteLine (
      "{0} will be tested; reps={1}; msg={2}",
      mi.Name, att.Repetitions, att.FailureMessage);
}
```

Evo rezultata:

```
Method1 will be tested; reps=1; msg=
Method2 will be tested; reps=20; msg=
Method3 will be tested; reps=20; msg=Debugging Time!
```

Atributi za informacije od pozivaoca (C# 5.0)

Počevši od verzije C# 5.0, opcione parametre možete označiti pomoću jednog od tri *atributa za informacije od pozivaoca* (engl. *caller info attribute*), koji nalažu kompajleru da informacije dobijene od izvornog koda pozivaoca dodeli kao podrazumevanu vrednost datog parametra:

- `[CallerMemberName]` zadaje ime člana pozivaoca.
- `[CallerFilePath]` zadaje putanju datoteke sa izvornim kodom pozivaoca.
- `[CallerLineNumber]` zadaje odgovarajući broj reda datoteke sa izvornim kodom pozivaoca.

Metoda `Foo` u sledećem programu ilustruje upotrebu sva tri atributa:

```
using System;
using System.Runtime.CompilerServices;

class Program

{
  static void Main()
  {
    Foo();
  }

  static void Foo (
    [CallerMemberName] string memberName = null,
    [CallerFilePath] string filePath = null,
    [CallerLineNumber] int lineNumber = 0)
  {
    Console.WriteLine (memberName);
    Console.WriteLine (filePath);
```

```
    Console.WriteLine (lineNumber);
  }
}
```

Pod pretpostavkom da se naš program nalazi u datoteci *c:\source\test\Program.cs*, rezultat bi bio:

```
Main
c:\source\test\Program.cs
8
```

Kao i u slučaju standardnih opcionih parametara, zamena se obavlja na *mestu poziva*. Prema tome, naša metoda `Main` je sintaksno ekvivalentna sledećem:

```
static void Main()
{
  Foo ("Main", @"c:\source\test\Program.cs", 8);
}
```

Atributi za informacije od pozivaoca korisni su za pisanje funkcija za upisivanje podataka u sistemske dnevnike i za implementiranje šablona za obaveštavanje o promenama. Recimo, metoda poput sledeće može se pozvati iz metode `set` koja zadaje vrednost svojstva – a da se ne mora navesti ime tog svojstva:

```
void RaisePropertyChanged (
  [CallerMemberName] string propertyName = null)
  {
    ...
  }
```

Asinhrone funkcije (C# 5.0)

U C# 5.0 uvedene su rezervisane reči `await` i `async` da bi se podržalo *asinhrono programiranje* – stil programiranja u kome funkcije koje se dugo izvršavaju obavljaju većinu ili sav svoj posao *nakon* što vrate kontrolu pozivaocu. To je suprotno *sinhronom* programiranju, u kome funkcije koje se dugo izvršavaju *blokiraju* pozivaoca sve dok se operacija ne završi. Asinhrono programiranje podrazumeva *uporedan rad*

(engl. *concurrency*), pošto se dugotrajne operacije izvršavaju *uporedo* s operacijama pozivaoca. Kôd koji implementira asinhronu funkciju započinje paralelan rad ili u obliku višenitnog rada, engl. *multithreading* (za operacije izračunavanja), ili putem mehanizma povratnog pozivanja (za I/O operacije).

Napomena

Višenitni rad, paralelni rad i asinhrono programiranje opsežne su teme. Posvećena su im dva poglavlja u knjizi *C# 5.0 za programere*, kao i razmatranje na adresi *http://albahari.com/threading*.

Na primer, razmotrite sledeću sinhronu metodu koja se dugo izvršava i obavlja dugotrajno izračunavanje:

```
int ComplexCalculation()
{
  double x = 2;
  for (int i = 1; i < 100000000; i++)
    x += Math.Sqrt (x) / i;
  return (int)x;
}
```

Ova metoda blokira pozivaoca nekoliko sekundi dok se ona izvršava. Zatim se rezultat izračunavanja vraća pozivaocu:

```
int result = ComplexCalculation();
// Malo kasnije:
Console.WriteLine (result); // 116
```

U okruženju CLR definisana je klasa `Task<TResult>` (u imenskom prostoru `System.Threading.Tasks`) da bi se obuhvatio koncept operacije koja se završava u budućnosti. Objekat tipa `Task<TResult>` možete generisati za operaciju izračunavanja tako što ćete pozvati metodu `Task.Run`; ona nalaže CLR-u da zadati delegat izvrši u zasebnoj niti koja se izvršava paralelno s pozivaocem:

```
Task<int> ComplexCalculationAsync()
{
  return Task.Run (() => ComplexCalculation());
}
```

Ova metoda je *asinhrona* zato što odmah vraća kontrolu pozivaocu dok se ona izvršava paralelno. Međutim, treba nam neki mehanizam koji bi pozivaocu omogućio da zada šta bi trebalo da se desi kada se operacija završi i rezultat postane dostupan. `Task<TResult>` to rešava tako što izlaže metodu `GetAwaiter`, koja omogućava da pozivalac definiše *nastavak* (engl. *continuation*):

```
Task<int> task = ComplexCalculationAsync();
var awaiter = task.GetAwaiter();
awaiter.OnCompleted (() =>           // Nastavak
{
  int result = awaiter.GetResult();
   Console.WriteLine (result);       // 116
});
```

Ovaj kôd saopštava operaciji: „Kada završiš, neka se izvrši zadati delegat.“ U našem nastavku, prvo se poziva metoda `GetResult` koja vraća rezultat izračunavanja. (Ili, ako operacija *nije uspela* [generisan je izuzetak], pozivanjem metode `GetResult` ponovo se generiše taj izuzetak.) Naš nastavak tada ispisuje rezultat pomoću naredbe `Console.WriteLine`.

Rezervisane reči await i async

Rezervisana reč `await` pojednostavljuje dodavanje nastavaka. Počevši od osnovnog scenarija, kompajler proširuje kôd:

```
var result = await izraz;
naredbe;
```

u nešto što je po funkciji slično ovome:

```
var awaiter = izraz.GetAwaiter();
awaiter.OnCompleted (() =>
{
```

```
  var result = awaiter.GetResult();
  naredbe;
);
```

Napomena

Kompajler takođe generiše kôd da bi optimizovao situacije u kojima se operacije završavaju sinhrono (odmah). Asinhrona operacija se završava odmah najčešće onda kada implementira unutrašnji mehanizam za keširanje a rezultat je već u kešu.

Stoga, evo kako možemo pozvati metodu `ComplexCalculationAsync` koju smo ranije definisali:

```
int result = await ComplexCalculationAsync();
Console.WriteLine (result);
```

Da bi se kôd preveo, sadržanoj metodi moramo dodati modifikator `async`:

```
async void Test()
{
  int result = await ComplexCalculationAsync();
  Console.WriteLine (result);
}
```

Modifikator `async` nalaže kompajleru da `await` tretira kao rezervisanu reč a ne kao identifikator ukoliko se u toj metodi pojave nejasnoće (tako se obezbeđuje da se kôd napisan pre pojave verzije C# 5.0, a u kome bi `await` mogao da se koristi kao identifikator, i dalje kompajlira bez greške). Modifikator `async` je primenljiv samo na metode (i lambda izraze) koji vraćaju `void` ili (kao što ćemo videti kasnije) `Task` ili `Task<TResult>`.

Napomena

Modifikator `async` sličan je modifikatoru `unsafe` po tome što ne utiče na potpis metode niti na javne metapodatke; utiče samo na ono što se dešava unutar date metode.

Metode s modifikatorom `async` zovu se *asinhrone funkcije* zato što su i same obično asinhrone. Da bismo videli zašto, pogledajmo kako se izvršavanje nastavlja putem asinhrone funkcije.

Posle nailaska na izraz `await`, tok izvršavanja se (standardno) vraća pozivaocu – slično kao u slučaju naredbe `yield return` u iteratoru. Ali, pre vraćanja, izvršno okruženje pridružuje nastavak poslu koji čeka, obezbeđujući tako da se po završetku asinhronog posla izvršavanje vrati u pozivajuću metodu i nastavi od mesta na kome je zaustavljeno. Ako posao ne uspe, ponovo se generiše njegov izuzetak (tako što se pozove `GetResult`); u suprotnom, njegova povratna vrednost se dodeljuje izrazu `await`.

Napomena

CLR-ova implementacija metode `OnCompleted` „čekalice" (nastavka), omogućava da se nastavci standardno prosleđuju kroz tekući *sinhronizacioni kontekst*, ako takav postoji. U praksi, to znači da u slučajevima sa korisničkim interfejsima složenih klijenata (WPF, Metro, Silverlight i Windows Forms), ako zadate `await` u niti korisničkog interfejsa, vaš kôd će nastaviti da se izvršava u istoj toj niti. To pojednostavljuje probleme bezbednosti niti.

Izraz na koji se primenjuje modifikator `await` obično je posao; međutim, kompajler će biti zadovoljan svakim objektom s metodom `GetAwaiter` koji vraća *objekat za čekanje* – koji implementira interfejs `INotifyCompletion.OnCompleted` i s metodom `GetResult` ispravnog tipa (i logičkim svojstvom `IsCompleted` pomoću kojeg se ispituje sinhroni završetak).

Rezultat našeg `await` izraza je tipa `int`; razlog je to što je izraz koji smo čekali bio tipa `Task<int>` (čija metoda `GetAwaiter().GetResult()` vraća tip `int`).

Čekanje na završetak negeneričkog posla dozvoljeno je i daje prazan izraz:

```
await Task.Delay (5000);
Console.WriteLine ("Five seconds passed!");
```

`Task.Delay` je statička metoda koja vraća posao koji se završava u zadatom broju milisekundi. *Sinhroni* ekvivalent metode `Task.Delay` jeste `Thread.Sleep`.

`Task` je negenerička osnovna varijanta klase `Task<TResult>` i funkcionalno joj je ekvivalentna, osim što nema rezultat.

Beleženje lokalnog stanja

Stvarna snaga `await` izraza leži u tome što se mogu pojaviti gotovo bilo gde u kodu. Konkretno, `await` izraz može biti na mestu bilo kog izraza (unutar asinhrone funkcije) osim unutar bloka `catch` ili `finally`, `lock` izraza, `unsafe` konteksta, ili u ulaznoj tački izvršnog koda (glavnoj metodi).

U sledećem primeru, pomoću modifikatora `await` čekamo unutar petlje:

```
async void Test()
{
  for (int i = 0; i < 10; i++)
  {
    int result = await ComplexCalculationAsync();
    Console.WriteLine (result);
  }
}
```

Nakon prvog izvršavanja metode `ComplexCalculationAsync`, tok izvršavanja se vraća pozivaocu zbog izraza `await`. Kada se metoda završi (ili ne uspe), izvršavanje se nastavlja od mesta gde je bilo prekinuto, sa očuvanim vrednostima promenljivih i brojača u petljama. Kompajler to postiže tako što prevodi takav kôd u mašinu s konačnim brojem stanja (engl. *state machine*), kao i u slučaju iteratora.

Bez rezervisane reči `await`, ručna upotreba nastavaka bi značila da morate napisati nešto ekvivalentno mašini s konačnim brojem stanja. To je oduvek bio razlog zbog kojeg je asinhrono programiranje teško.

Pisanje asinhronih funkcija

U svakoj asinhronoj funkciji, povratni tip `void` možete zameniti tipom `Task` da bi sama metoda postala *korisno* asinhrona (i da bi uz nju bila upotrebljena rezervisana reč `await`). Druge izmene nisu potrebne:

```
async Task PrintAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  Console.WriteLine (answer);
}
```

Obratite pažnju na to da posao ne vraćamo eksplicitno u telo metode. Kompajler generiše posao koji obaveštava o završetku metode ili u slučaju neobrađenog izuzetka. To olakšava izradu lanaca asinhronih poziva:

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}
```

(A pošto metoda `Go` vraća objekat tipa `Task`, `Go` je samu po sebi moguće čekati.) Kompajler proširuje asinhrone funkcije koje vraćaju poslove u kôd koji (indirektno) koristi objekat klase `TaskCompletionSource` da bi formirao posao koji zatim izvršava do kraja ili prekida.

Napomena

`TaskCompletionSource` je CLR tip koji vam omogućava da pravite poslove kojima upravljate ručno, obeležavajući ih kao završene s rezultatom ili prekinute zbog izuzetka. Za razliku od `Task.Run`, `TaskCompletionSource` ne blokira nit tokom trajanja operacije. Koristi se i za pisanje I/O metoda koje vraćaju poslove (kao što je `Task.Delay`).

Cilj je osigurati sledeće: kada se završi asinhrona metoda koja vraća kontrolu pozivajućem poslu, tok izvršavanja može da se vrati – putem nastavka – bilo kom elementu koda koji je čekao na izvršavanje.

Vraćanje objekta tipa Task<TResult>

Možete vratiti `Task<TResult>` ako telo date metode vraća tip `TResult`:

```
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  // odgovor je int, pa naša metoda vraća Task<int>
  return answer;
}
```

Rad metode `GetAnswerToLife` možemo prikazati tako što ćemo je pozvati iz metode `PrintAnswerToLife` (koja se poziva iz metode `Go`):

```
async Task Go()
{
  await PrintAnswerToLife();
  Console.WriteLine ("Done");
}
async Task PrintAnswerToLife()
{
  int answer = await GetAnswerToLife();
  Console.WriteLine (answer);
}
async Task<int> GetAnswerToLife()
{
  await Task.Delay (5000);
  int answer = 21 * 2;
  return answer;
}
```

Asinhrone funkcije čine asinhrono programiranje sličnim sinhronom programiranju. Evo sinhronog ekvivalenta naše šeme poziva, u kome se pozivanjem metode `Go()` dobija isti rezultat, nakon blokade od pet sekundi:

```
void Go()
{
  PrintAnswerToLife();
  Console.WriteLine ("Done");
}
void PrintAnswerToLife()
{
  int answer = GetAnswerToLife();
  Console.WriteLine (answer);
}
int GetAnswerToLife()
{
  Thread.Sleep (5000);
  int answer = 21 * 2;
  return answer;
}
```

Ovaj primer takođe ilustruje osnovni princip upotrebe asinhronih funkcija u jeziku C#, a to je: pišite sinhrone metode, a zatim *sinhrone* pozive metoda zamenite *asinhronim* pozivima metoda, i čekajte na njih (koristeći rezervisanu reč `await`).

Paralelizam

Upravo smo prikazali najčešće korišćen šablon, a to je `await` s funkcijama koje vraćaju kontrolu pozivajućem poslu odmah nakon pozivanja. Rezultat toga je sekvencijalni tok programa koji je logički sličan sinhronom ekvivalentu.

Pozivanje asinhrone metode bez čekanja da se ona završi, omogućava da se kôd koji sledi izvršava paralelno. Na primer, u sledećem kodu, metoda `PrintAnswerToLife` izvršava se dvaput, paralelno:

```
var task1 = PrintAnswerToLife();
var task2 = PrintAnswerToLife();
await task1; await task2;
```

Pošto nakon pozivanja čekamo na završetak obe operacije, „okončavamo“ paralelizam u toj tački (i ponovo generišemo eventualne izuzetke generisane u tim metodama). Klasa `Task` obezbeđuje statičku

metodu po imenu `WhenAll` da bi se isti rezultat ostvario malo efikasnije. `WhenAll` vraća zadatak koji se završava kada se završe svi poslovi koje ste prosledili toj metodi:

```
await Task.WhenAll (PrintAnswerToLife(),
                    PrintAnswerToLife());
```

`WhenAll` je pozvani *kombinator poslova*. (Klasa `Task` obezbeđuje i kombinator poslova `WhenAny`, koji se završava kada se završi bilo koji od poslova koji mu je prosleđen.) Kombinatori poslova su detaljno opisani u knjizi *C# 5.0 za programere*.

Asinhroni lambda izrazi

Kao što obične *imenovane* metode mogu biti asinhrone:

```
async Task NamedMethod()
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
}
```

tako mogu i *neimenovane* metode (lambda izrazi i anonimne metode), ukoliko im prethodi rezervisana reč `async`:

```
Func<Task> unnamed = async () =>
{
  await Task.Delay (1000);
  Console.WriteLine ("Foo");
};
```

Možemo ih pozivati i čekati na isti način:

```
await NamedMethod();
await unnamed();
```

Asinhroni lambda izrazi mogu se koristiti kao procedure za obradu događaja:

```
myButton.Click += async (sender, args) =>
{
```

```
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

Ovo je sažetije od sledećeg koda, koji ima isti efekat:

```
myButton.Click += ButtonHandler;
...
async void ButtonHander (object sender, EventArgs args)
{
  await Task.Delay (1000);
  myButton.Content = "Done";
};
```

Asinhroni lambda izrazi takođe mogu da vrate objekat tipa `Task<TResult>`:

```
Func<Task<int>> unnamed = async () =>
{
  await Task.Delay (1000);
  return 123;
};
int answer = await unnamed();
```

Nebezbedan kôd i pokazivači

C# podržava direktan rad s memorijom putem pokazivača unutar blokova koda označenih kao nebezbedni i prevođenja koda s opcijom `/unsafe` kompajlera. Pokazivački tipovi su korisni, pre svega, za interoperabilnost sa API-jima na jeziku C, ali se mogu koristiti i za direktno pristupanje memoriji izvan upravljanog hipa ili tačkama koje su kritične za performanse.

Osnove pokazivača

Za svaki vrednosni ili pokazivački tip V postoji odgovarajući pokazivački tip V*. Instanca pokazivača čuva adresu promenljive. Pokazivački tipovi se mogu (nebezbedno) konvertovati u bilo koji drugi pokazivački tip. Evo glavnih operatora za rad s pokazivačima:

Operator	Značenje
&	Operator *adrese* vraća pokazivač na adresu promenljive.
*	Operator *dereferenciranja* vraća promenljivu koja se nalazi na adresi pokazivača.
->	Operator *pokazivač člana* je sintaksna prečica, u kojoj je `x->y` ekvivalentno `(*x).y`.

Nebezbedan kôd

Ako označite tip, član tipa ili blok naredaba pomoću rezervisane reči `unsafe`, možete koristiti pokazivačke tipove i obavljati operacije u stilu jezika C++ nad memorijom unutar njegovog opsega. Evo primera u kome se pokazivači koriste za brzu obradu bit-mape:

```
unsafe void BlueFilter (int[,] bitmap)
{
  int length = bitmap.Length;
  fixed (int* b = bitmap)
  {
    int* p = b;
    for (int i = 0; i < length; i++)
    *p++ &= 0xFF;
  }
}
```

Nebezbedan kôd može da se izvršava brže od iste bezbedne verzije koda. U ovom slučaju, bila bi potrebna ugnežđena petlja s indeksiranjem niza i proverom granica. Nebezbedna C# metoda mogla bi, takođe, da bude brža od pozivanja spoljne C funkcije, pošto nema opterećenja zbog napuštanja upravljanog izvršnog okruženja.

Naredba fixed

Naredba `fixed` je potrebna da bi se fiksirao objekat koji se obrađuje, kao što je bit-mapa u prethodnom primeru. Tokom izvršavanja programa, mnogim objektima se dodeljuje i oduzima memorija sa hipa. Da bi se izbeglo nepotrebno zauzimanje ili fragmentiranje memorije, skupljač smeća premešta objekte unaokolo. Održavanje pokazivača na objekat je uzaludno ako se njegova adresa može promeniti dok

se on referencira, pa naredba `fixed` nalaže skupljaču smeća da „zakuca“ taj objekat i da ga ne pomera. Pošto to može loše uticati na efikasnost izvršnog okruženja, fiksirane blokove treba koristiti samo kratko, i ne treba im dodeljivati memoriju sa hipa.

Unutar naredbe `fixed` možete dobiti pokazivač na vrednosni tip, niz vrednosnih tipova, ili znakovni niz. U slučaju nizova i znakovnih nizova, pokazivač će – u stvari – pokazivati na prvi element, koji je vrednosnog tipa.

Kada su vrednosni tipovi deklarisani u istom redu s referentnim tipovima, mora se fiksirati referentni tip:

```
class Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    fixed (int* p = &test.x)   // Fiksira test
    {
      *p = 9;
    }
    System.Console.WriteLine (test.x);
  }
}
```

Operator pokazivač člana

Pored operatora `&` i `*`, C# ima i operator `->` u stilu jezika C++, primenljiv na strukture:

```
struct Test
{
  int x;
  unsafe static void Main()
  {
    Test test = new Test();
    Test* p = &test;
    p->x = 9;
```

```
    System.Console.WriteLine (test.x);
  }
}
```

Nizovi

Rezervisana reč stackalloc

Memorija se može eksplicitno dodeliti u obliku bloka na steku pomoću rezervisane reči `stackalloc`. Pošto je dodeljena na steku, životni vek joj je ograničen na izvršavanje metode, baš kao i u slučaju svake druge lokalne promenljive. Blok može da koristi operator `[]` za indeksiranje memorije:

```
int* a = stackalloc int [10];
for (int i = 0; i < 10; ++i)
  Console.WriteLine (a[i]);   // Ispisuje sirovu memoriju
```

Baferi fiksne veličine

Memorija se može dodeliti u bloku unutar strukture pomoću rezervisane reči `fixed`:

```
unsafe struct UnsafeUnicodeString
{
  public short Length;
  public fixed byte Buffer[30];
}

unsafe class UnsafeClass
{
  UnsafeUnicodeString uus;

  public UnsafeClass (string s)
  {
    uus.Length = (short)s.Length;
    fixed (byte* p = uus.Buffer)
      for (int i = 0; i < s.Length; i++)
        p[i] = (byte) s[i];
  }
}
```

U ovom primeru, rezervisana reč `fixed` koristi se i da bi se na hipu fiksirao objekat koji čini bafer (što će biti instanca klase `UnsafeClass`).

void*

Prazan pokazivač (`void*`) ne pretpostavlja ništa o tipu pridruženog podatka i koristan je za funkcije koje rade direktno s memorijom. Svaki pokazivački tip može se implicitno konvertovati u `void*`. `void*` se ne može dereferencirati, niti se nad praznim pokazivačima mogu obavljati aritmetičke operacije. Na primer:

```
unsafe static void Main()
{
  short[] a = {1,1,2,3,5,8,13,21,34,55};
  fixed (short* p = a)
  {
    // sizeof vraća veličinu vrednosnih tipova u bajtovima
    Zap (p, a.Length * sizeof (short));
  }
  foreach (short x in a)
    System.Console.WriteLine (x); // Ispisuje sve nule
}

unsafe static void Zap (void* memory, int byteCount)
{
  byte* b = (byte*) memory;
    for (int i = 0; i < byteCount; i++)
      *b++ = 0;
}
```

Pretprocesorske direktive

Pretprocesorske direktive pružaju prevodi dodatne informacije o pojedinim delovima koda. Najčešće pretprocesorske direktive su uslovne direktive, koje obezbeđuju način za uključivanje ili izostavljanje delova koda pri prevođenju. Na primer:

```
#define DEBUG
class MyClass
{
  int x;
  void Foo()
  {
    # if DEBUG
    Console.WriteLine ("Testing: x = {0}", x);
    # endif }
  ...
}
```

U ovoj klasi, naredba u metodi `Foo` prevodi se kao uslovno zavisna od toga postoji li simbol `DEBUG`. Ako uklonimo simbol `DEBUG`, naredba se ne prevodi. Pretprocesorski simboli se mogu definisati u datoteci sa izvornim kodom (kao što smo mi uradili), ili se mogu proslediti kompajleru pomoću opcije komandne linije `/ define:`*simbol*.

(S direktivama `#if` i `#elif`, operatore `||`, `&&` i `!` možete koristiti za obavljanje operacija *or*, *and* i *not* nad više simbola. Sledeća direktiva nalaže kompajleru da prevede kôd koji sledi ukoliko je simbol `TESTMODE` definisan a simbol `DEBUG` nije:

```
#if TESTMODE && !DEBUG
  ...
```

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

Simboli `#error` i `#warning` sprečavaju nehotičnu zloupotrebu uslovnih direktiva tako što nalažu kompajleru da generiše upozorenje ili grešku ukoliko mu se prosledi nepoželjan skup simbola.

Evo kompletne liste pretprocesorskih direktiva:

Pretprocesorska direktiva	Akcija
`#define` *simbol*	Definiše *simbol*
`#undef` *simbol*	Ukida definiciju *simbola*
`#if` *simbol*`[`*operator* `simbol2]...`	Uslovno prevođenje (*operatori* su `==`, `!=`, `&&` i `||`)

Pretprocesorska direktiva	Akcija
`#else`	Izvršava kôd do sledeće direktive `#endif`
`#elif` *`simbol`*`[`*`operator`* `simbol2]`	Kombinuje granu `#else i test #if`
`#endif`	Završava uslovne direktive
`#warning` *`tekst`*	*`tekst`* upozorenja koje se pojavljuje kao rezultat prevođenja
`#error` *`tekst`*	*`tekst`* poruke o grešci koja se pojavljuje kao rezultat prevođenja
`#line [`*`number`*`["`*`file`*`"] | hidden]`	*`number`* je broj reda u izvornom kodu; *`file`* je ime datoteke koje će se pojaviti u rezultatu; `hidden` nalaže modulima za pronalaženje i otklanjanje grešaka da preskoče kôd od ove tačke do naredne direktive `#line`
`#region` *`name`*	Označava početak celine u kodu
`#endregion`	Označava kraj celine
`#pragma warning`	Videti sledeći odeljak

Direktiva #pragma warning

Kompajler generiše upozorenje kada u kodu uoči nešto što deluje da nije tu namerno. Za razliku od grešaka, upozorenja obično ne sprečavaju prevođenje aplikacije.

Upozorenja kompajlera mogu biti izuzetno korisna za uočavanje grešaka u kodu. Međutim, njihovu korisnost podrivaju *lažna* upozorenja. U velikoj aplikaciji, održavanje dobrog odnosa signal/šum od ključnog je značaja ukoliko želite da se primete „stvarna“ upozorenja.

U tom smislu, kompajler vam omogućava da selektivno sprečite pojavu određenih upozorenja, pomoću direktive `#pragma warning`. U ovom primeru, nalažemo kompajleru da nas ne upozorava na to da se polje `Message` ne koristi:

```
public class Foo
{
  static void Main() { }
  #pragma warning disable 414
  static string Message = "Hello";
```

```
    #pragma warning restore 414
}
```

Izostavljanjem broja iz direktive `#pragma warning`, deaktivirate odnosno ponovo aktivirate sve upozoravajuće kodove.

Ukoliko dosledno primenjujete ovu direktivu, kôd možete prevesti uz opciju `/warnaserror` – ona nalaže kompajleru da eventualna zaostala upozorenja tretira kao greške.

XML dokumentacija

Dokumentacioni komentar je blok ugrađenog XML-a kojim se dokumentuje neki tip ili član. Dokumentacioni komentar se piše neposredno ispred deklaracije tipa ili člana, i počinje s tri kose crte:

```
/// <summary>Otkazuje izvršavanje upita.</summary>
public void Cancel() { ... }
```

Višeredni komentari se pišu ili ovako:

```
/// <summary>
/// Otkazuje izvršavanje upita
/// </summary>
public void Cancel() { ... }
```

ili ovako (zapazite dodatnu zvezdicu na početku):

```
/**
  <summary> Otkazuje izvršavanje upita. </summary>
*/
public void Cancel() { ... }
```

Ukoliko prevodite kôd pomoću direktive `/doc`, kompajler izdvaja i skuplja dokumentacione komentare u jednu XML datoteku. Ona se koristi na dva osnovna načina:

- Ako se smesti u isti direktorijum kao prevedeni sklop, Visual Studio automatski čita XML datoteku i koristi informacije iz nje kako bi preko mehanizma IntelliSense obezbedio liste članova spoljnim korisnicima sklopa istog imena.

- Alatke drugih proizvođača (kao što su Sandcastle i NDoc) mogu da pretvore XML datoteku u HTML datoteku pomoći.

Standardne XML oznake u dokumentaciji

Evo standardnih XML oznaka (engl. *tags*) koje prepoznaju Visual Studio i generatori dokumentacije:

```
<summary>
        <summary>...</summary>
```

Označava kratak opis (engl. *tool tip*) koji IntelliSense treba da prikaže za dati tip ili član. To je obično jedan izraz ili rečenica.

```
<remarks>
        <remarks>...</remarks>
```

Dodatni tekst kojim se opisuje dati tip ili član. Preuzimaju ga generatori dokumentacije i kombinuju u opširniji opis tipa ili člana.

```
<param>
        <param name="name">...</param>
```

Objašnjava parametar metode.

```
<returns>
        <returns>...</returns>
```

Objašnjava povratnu vrednost metode.

```
<exception>
        <exception [cref="type"]>...</exception>
```

Navodi izuzetak koji može da generiše neka metoda (`cref` ukazuje na tip izuzetka).

```
<permission>
        <permission [cref="type"]>...</permission>
```

Upućuje na tip `IPermission` koji je potreban dokumentovanom tipu ili članu.

`<example>`

```
<example>...</example>
```

Označava primer (koji koriste generatori dokumentacije). Obično sadrži i opisni tekst i izvorni kôd (izvorni kôd je najčešće unutar oznaka `<c>` ili `<code>`).

`<c>`

```
<c>...</c>
```

Označava jednoredni odlomak koda. Ova oznaka se obično koristi unutar bloka `<example>`.

`<code>`

```
<code>...</code>
```

Označava odlomak koda. Ova oznaka se obično koristi unutar bloka `<example>`.

`<see>`

```
<see cref="member">...</see>
```

Umeće referencu na drugi tip ili član, u istom redu. Generatori HTML dokumentacije obično pretvaraju ovu oznaku u hipervezu. Kompajler daje upozorenje ukoliko je ime tipa ili člana neispravno.

`<seealso>`

```
<seealso cref="member">...</seealso>
```

Referencira drugi tip ili član. Generatori dokumentacije ovo obično upisuju u poseban odeljak „See Also“ („Videti i“) na dnu strane.

`<paramref>`

```
<paramref name="name"/>
```

Referencira parametar iz oznake `<summary>` ili `<remarks>`.

`<list>`

```
<list type=[ bullet | number | table ]>
  <listheader>
    <term>...</term>
    <description>...</description>
  </listheader>
```

```
        <item>
          <term>...</term>
          <description>...</description>
        </item>
      </list>
```

Nalaže generatorima dokumentacije da naprave listu s bulitima, numerisanu listu ili listu u stilu tabele.

```
<para>
      <para>...</para>
```

Nalaže generatorima dokumentacije da formatiraju sadržaj kao zasebne pasuse.

```
<include>
      <include file='filename' path='tagpath[@name="id"]'>...
      </para>
```

Kombinuje spoljnu XML datoteku koja sadrži dokumentaciju. Atribut `path` označava XPath upit za određeni element u toj datoteci.

O autorima

Džozef (Joseph) Albahari je autor knjiga *C# 4.0 in a Nutshell*, *LINQ Pocket Reference* i *C# 4.0 Pocket Reference*. Bavi se razvojem obimnih poslovnih aplikacija duže od 15 godina, i autor je LINQPad-a – popularne alatke za izradu LINQ upita namenjenih bazama podataka. Trenutno radi honorarno, kao konsultant. Više informacija o njemu naći ćete na adresi *http://albahari.net/*.

Ben Albahari je osnivač veb lokacije *TakeOnIt.com*. Pet godina je upravljao razvojem programa u Microsoftu, gde je radio na nekoliko projekata, uključujući .NET Compact Framework i ADO.NET. Ben je suosnivač kompanije Genamics, koja pravi alatke za C# i J++ programere, kao i softver za analizu DNA i proteinskih sekvenci. Koautor je knjige *C# Essentials,* prve knjige o jeziku C# koju je objavila izdavačka kuća O'Reilly, kao i prethodnih izdanja knjige *C# in a Nutshell*.