14

Istovremenost i asinhroni način rada

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

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

Uvod

Najuobičajeniji slučajevi kada je potrebna istovremenost jesu sledeći:

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

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

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

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

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

Višenitni način rada programa

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

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

Pokretanje nove niti

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

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

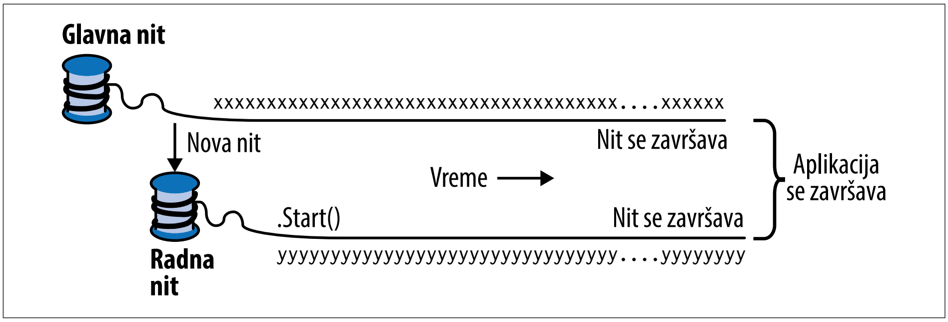
Novu nit možete formirati i pokrenuti tako što instancirate objekat klase `Thread` i pozovete njegovu metodu `Start`. Najjednostavniji konstruktor klase `Thread` preuzima delegat tipa `ThreadStart`: predstavlja metodu bez parametara koja pokazuje gde bi trebalo da počne izvršavanje. Na primer:

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

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

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

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

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

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

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



Slika 14-1. Pokretanje nove niti.

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

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

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

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

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

Metode Join i Sleep

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

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

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

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

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

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

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

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

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

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

Blokiranje niti

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

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

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

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

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

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

U/I zavisne operacije i operacije zavisne od proračuna

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

Blokiranje i vrtenje niti

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

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

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

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

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

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

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

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

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

Lokalno i deljeno stanje

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

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

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

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

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

```
class ThreadTest
{
  bool _done;

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

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

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

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

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

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

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

```
    Go();
  }

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

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

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

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

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

Blokade i bezbednost niti

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

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

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

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

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

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

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

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

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

Prosleđivanje podataka drugoj niti

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

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

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

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

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

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

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

static void Print (object messageObj)
{
```

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

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

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

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

Lambda izrazi i preuzete promenljive

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

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

Rezultat je nedeterministički! Ovako izgleda tipičan rezultat:

```
0223557799
```

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

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

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

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

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

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

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

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

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

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

Obrada izuzetaka

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

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

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

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

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

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

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

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

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