13

Dijagnostika i kodni ugovori

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

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

Kodni ugovori su objašnjeni (na engleskom jeziku) u posebnom dodatku, koji možete preuzeti na *http://www.albahari.com/nutshell*.

API Code Contracts je razvio Microsoft Research i zahteva zasebno preuzimaje. Uprkos obećavajućem početku, ta tehnologija se nije nikad zaista „primila“ i u nju je poslednjih godina ugrađen minimalan broj ispravki i dopuna. Glavni nedostatak joj je možda to što ne postoji direktna podrška u jeziku C#. Posledica toga je sporiji ciklus sklapanja aplikacija zato se nakon kompajliranja, rezultujući sklop mora „ponovo“ napisati.

Tipovi koje pominjemo u ovom poglavlju definisani su prevashodno u imenskom prostorima `System.Diagnostics`.

Uslovno prevođenje koda

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

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

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

class Program
{
  static void Main()
  {
#if TESTMODE
    Console.WriteLine ("ovo je test!");     // REZULTAT: ovo je test!
#endif
  }
}
```

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

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

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

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

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

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

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

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

Uslovno prevođenje u poređenju sa statičkim promenljivama

Prethodni primer se može implementirati i pomoću jednostavnog statičkog polja:

```
static internal bool TestMode = true;

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

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

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

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

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

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

Atribut Conditional

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

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

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

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

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

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

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

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

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

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

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

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

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

Alternative za atribut Conditional

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

```
using System;
using System.Linq;

class Program
{
  public static bool EnableLogging;

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

Lambda izraz omogućava pozivanje metode bez složene sintakse:

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

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

Klase Debug i Trace

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

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

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

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

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

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

Metode Fail i Assert

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

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

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

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

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

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

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

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

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

Klasa TraceListener

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

Pražnjenje i zatvaranje osluškivača

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

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

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

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

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

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

Integrisanje aplikacije sa spoljnim dibagerom

Ponekad je korisno da aplikacija sarađuje s nekom alatkom za otkrivanje grešaka (dibager), ako je takva na raspolaganju. Tokom razvoja aplikacije, dibager je najčešće sastavni deo vašeg integrisanog razvojnog okruženja (na primer, Visual Studio). Za instaliranu verziju aplikacije, dibager će verovatnije biti:

- DbgCLR
- Neka od alatki za otkrivanje grešaka nižeg nivoa kao što su WinDbg, Cordbg ili Mdbg

DbgCLR je Visual Studio iz kojeg je „skinuto“ sve osim dibagera, a može se besplatno preuzeti u paketu .NET Framework SDK. To je najjednostavnija opcija za otkrivanje grešaka kada nemate na raspolaganju integrisano razvojno okruženje, mada zahteva da preuzmete ceo SDK.

Povezivanje dibagera sa aplikacijom i raskidanje veze između njih

Statička klasa `Debugger` u imenskom prostoru `System.Diagnostics` obezbeđuje osnovne funkcije za interakciju s dibagerom. To su `Break`, `Launch`, `Log` i `IsAttached`.

Dibager morate prvo povezati sa aplikacijom kako biste mogli da tražite greške u njoj. Ako aplikaciju pokrećete unutar integrisanog razvojnog okruženja, to se obavlja automatski, osim kada zahtevate drugačije (biranjem opcije „Start without debugging“). Međutim, ponekad nije zgodno ili nije ni moguće pokrenuti aplikaciju u režimu otklanjanja grešaka unutar integrisanog razvojnog okruženja. Primer toga je aplikacija koja je Windowsov servis ili (ironično) Visual Studiov dizajner. Jedno od rešenja je da aplikaciju pokrenete na uobičajen način i da, zatim, u svom razvojnom okruženju izaberete opciju Debug Process. Međutim, to vam ne omogućava da postavite prekidne tačke na početku izvršavanja programa.

Problem možete zaobići ako pozovete metodu `Debugger.Break` iz aplikacije. Ta metoda pokreće dibager, povezuje ga sa aplikacijom i zaustavlja izvršavanje aplikacije na tom mestu. (metoda `Launch` radi isto, ali ne zaustavlja izvršavanje aplikacije.) Pošto aplikaciju povežete s dibagerom, pomoću metode `Log` možete ispisivati poruke direktno u izlazni prozor dibagera. Da li je aplikacija povezana s dibagerom, možete saznati pomoću svojstva `IsAttached`.

Atributi Debugger

Atributi `DebuggerStepThrough` i `DebuggerHidden` služe za zadavanje uputstava dibageru kako da korak po korak prati izvršavanje metode, konstruktora ili klase.

Atribut `DebuggerStepThrough` nalaže da dibager izvrši kôd funkcije bez interakcije s korisnikom. Taj atribut je koristan za automatski generisane metode i posredničke metode koje posao samo prosleđuju nekoj drugoj metodi. U ovom poslednjem slučaju, dibager će i dalje prikazivati posredničku metodu na stablu pozivanja metoda ako prekidnu tačku postavite unutar „stvarne“ metode – osim kada joj pridružite atribut `DebuggerHidden`. Ta dva atributa možete kombinovati za posredničke metode da bi korisiku bilo lakše da se usredsredi na greške u logici aplikacije umesto da se bavi infrastrukturom:

```
[DebuggerStepThrough, DebuggerHidden]
void DoWorkProxy()
{
  // priprema...
  DoWork();
  // završne operacije...
}


void DoWork() {...}   // Stvarna metoda...
```

Procesi i procesne niti

U poslednjem odeljku poglavlja 6 opisali smo kako se pomoću metode `Process.Start` pokreće nov proces. Klasa `Process` omogućava još i propitivanje i interakciju s procesima koji su aktivni na istom ili na drugom računaru. Klasa `Process` je definisana u .NET Standardu 2.0, ali je ograničena samo na platformu UWP.

Ispitivanje tekućih aktivnih procesa

Metode `Process.GetProcessXXX` učitavaju ciljni proces po imenu ili identifikatoru (ID) procesa, ili sve procese u toku na tekućem ili zadatom računaru. To obuhvata i upravljane i neupravljane procese. Svaka instanca klase `Process` ima više svojstava koja pružaju podatke kao što su ime procesa, ID, prioritet,zauzeće memorije i procesora, ručice prozora itd. Sledeći primer nabraja sve procese u toku izvršavanja na tekućem računaru:

```
foreach (Process p in Process.GetProcesses())
using (p)
{
  Console.WriteLine (p.ProcessName);
  Console.WriteLine ("   PID:      " + p.Id);
  Console.WriteLine ("   Memory:   " + p.WorkingSet64);
  Console.WriteLine ("   Threads:  " + p.Threads.Count);
}
```

Metoda `Process.GetCurrentProcess` vraća tekući proces. Ako ste otvorili nove aplikacione domene, oni će svi deliti jedan isti proces.

Proces možete prekinuti pomoću njegove metode `Kill`.

Ispitivanje niti jednog procesa

Svojstvo `Process.Threads` omogućava da nabrajate niti drugih procesa. Međutim, objekti koje dobijate nisu tipa `System.Threading.Thread`, nego tipa `ProcessThread`, koji su namenjeni za administrativne svrhe, a ne za poslove sinhronizovanja procesa. Objekat `ProcessThread` sadrži dijagnostičke podatke o niti koju predstavlja i omogućava da upravljate određenim svojstvima te niti, kao što su njen prioritet i iskorišćavanje procesora:

```
public void EnumerateThreads (Process p)
{
  foreach (ProcessThread pt in p.Threads)
  {
    Console.WriteLine (pt.Id);
    Console.WriteLine ("   State:    " + pt.ThreadState);
    Console.WriteLine ("   Priority: " + pt.PriorityLevel);
    Console.WriteLine ("   Started:  " + pt.StartTime);
    Console.WriteLine ("   CPU time: " + pt.TotalProcessorTime);
  }
}
```


Klase StackTrace i StackFrame

Klase `StackTrace` i `StackFrame` daju prikaz stabla izvršavanja metoda (samo za čitanje) i sastavni su deo standardnog .NET Frameworka. Prikaze tog stabla možete dobiti za tekuću nit, za drugu nit unutar istog procesa ili za dati objekat `Exception`. Ta vrsta informacija je korisna uglavnom za dijagnostičke namene, mada se može iskoristiti i u programiranju (hakovanje). Objekat `StackTrace` predstavlja kompletno stablo pozivanja metoda, dok klasa `StackFrame` predstavlja poziv jedne metode na tom stablu.

Ako instancirate objekat tipa `StackTrace` bez argumenata ili s jednim argumentom tipa `bool`, dobijate snimak stabla pozivanja u tekućoj niti. Argument tipa `bool`, ako mu je vrednost `true`, nalaže objektu `StackTrace` da čita *.pdb* (project debug) datoteke sklopa, ako su na raspolaganju, što vam pruža podatke kao što su ime datoteke, broj reda i broj kolone. Te .pdb datoteke se generišu kada projekat prevedete s opcijom `/debug`. (Visual Studio standardno prevodi kôd s tom opcijom, osim kada zahtevate drugačije u odeljku *Advanced Build Settings*.)

Pošto formirate objekat `StackTrace`, možete ispitati određenu metodu u njemu pomoću metode `GetFrame` ili, pomoću metode `GetFrames`, formirati kolekciju svih metoda na stablu:

```
static void Main() { A (); }
static void A()    { B (); }
static void B()    { C (); }
static void C()
{
  StackTrace s = new StackTrace (true);
```

```
    Console.WriteLine ("Total frames:   " + s.FrameCount);
    Console.WriteLine ("Current method: " + s.GetFrame(0).GetMethod().Name);
    Console.WriteLine ("Calling method: " + s.GetFrame(1).GetMethod().Name);
    Console.WriteLine ("Entry method:   " + s.GetFrame
                                            (s.FrameCount-1).GetMethod().Name);
    Console.WriteLine ("Call Stack:");
    foreach (StackFrame f in s.GetFrames())
      Console.WriteLine (
        "  File: "   + f.GetFileName() +
        "  Line: "   + f.GetFileLineNumber() +
        "  Col: "    + f.GetFileColumnNumber() +
        "  Offset: " + f.GetILOffset() +
        "  Method: " + f.GetMethod().Name);
}
```

Rezultat izgleda ovako:

```
Total frames:   4
Current method: C
Calling method: B
Entry method: Main
Call stack:
  File: C:\Test\Program.cs  Line: 15  Col: 4  Offset: 7  Method: C
  File: C:\Test\Program.cs  Line: 12  Col: 22  Offset: 6  Method: B
  File: C:\Test\Program.cs  Line: 11  Col: 22  Offset: 6  Method: A
  File: C:\Test\Program.cs  Line: 10  Col: 25  Offset: 6  Method: Main
```

IL pomak (vrednost `Offset`) pokazuje položaj naredbe koja će se *sledeća* izvršiti, a ne naredbe koja se trenutno izvršava. Međutim, čudno je to da se broj reda i kolone (ako je na raspolaganju *.pdb* datoteka) najčešće odnose na tekuću tačku izvršavanja.

Ovo se događa zato što se CLR trudi najbolje što ume da tekuću tačku izvršavanja *izvede* izračunavanjem reda i kolone na osnovu IL pomaka. Kompajler generiše IL kôd na način koji to omogućava, pa zato u tok generisanog IL koda umeće naredbe `nop` (no-operation).

Međutim, kada se izvorni kôd prevodi sa uključenim optimizacijama, to uklanja naredbe nop usled čega stablo pozivanja može da prikaže broj reda i kolone naredbe koja će se sledeća izvršiti. Pribavljanje upotrebljivog stabla pozivanja može biti dalje otežano zbog činjenice da se pri optimizovanju mogu primeniti i drugi trikovi, kao što je skrivanje celih metoda.

Prečica za pribavljanje zaista korisnih podataka o celom stablu pozivanja (objekat `StackTrace`) jeste da pozovete metodu `ToString` tog objekta. Rezultat izgleda ovako:

```
   at DebugTest.Program.C() in C:\Test\Program.cs:line 16
   at DebugTest.Program.B() in C:\Test\Program.cs:line 12
   at DebugTest.Program.A() in C:\Test\Program.cs:line 11
   at DebugTest.Program.Main() in C:\Test\Program.cs:line 10
```

Da biste dobili stablo pozivanja u drugoj niti, prosledite objekat `Thread` koji je predstavlja konstruktoru klase `StackTrace`. To može biti dobra strategija za profilisanje programa, mada dok formirate stablo pozivanja morate zamrznuti nit. To je zapravo prilično teško izvesti bez rizika od uzajamne blokade, a pouzdano rešenje problema opisujemo u odeljku „Metode Suspend i Resume“, na strani, 789 poglavlja 22.

Stablo pozivanja metoda možete dobiti i uz objekat `Exception` (gde se vidi šta je sve dovelo do generisanja izuzetka), tako što objekat `Exception` prosledite konstruktoru objekta `StackTrace`.