```
    lock (threadLock)
    {
      ...
    }
  }
}
```

Ha bármikor megvizsgáljuk a `PrintNumbers()` metódust, akkor láthatjuk, hogy az a megosztott erőforrás, amelynek eléréséért a szálak versengenek, a konzolablak. Ezért, ha az összes a `Console` típussal kapcsolatos műveletet zároljuk a következőképpen:

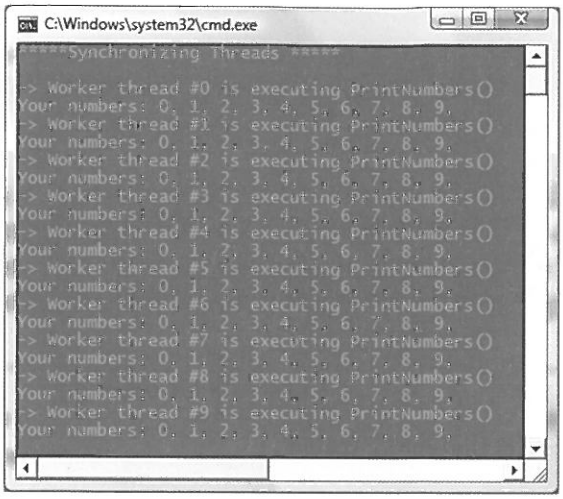
```
public void PrintNumbers()
{
  // Használjuk a privát objektumzároló tokent.
  lock (threadLock)
  {
    // A szálinformáció megjelenítése.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);

    // Számok kiíratása.
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
    Console.WriteLine();
  }
}
```

akkor gyakorlatilag megalkottunk egy olyan metódust, amely lehetővé teszi, hogy az aktuális szál befejezze a feladatát. Ha egy szál belép a zárolás hatókörébe, a zár tokenjéhez (ebben az esetben az aktuális objektum referenciája) egyéb szálak nem férhetnek mindaddig hozzá, amíg a zárolást fel nem oldjuk. Így, ha az A szál hozzájutott a zár tokenjéhez, más szálak addig nem léphetnek be a hatókörbe, amíg az A szál el nem engedi a zár tokenjét.

Megjegyzés Ha egy statikus metódus kódját szeretnénk zárolni, egyszerűen deklaráljunk egy privát statikus objektum-tagváltozót a zárolás tokenjeként.

Ha lefuttatjuk az alkalmazást, látható, hogy minden szálnak elég lehetősége van a feladata elvégzésére (lásd 18.10. ábra).



18.10. ábra: *Most már az összes szálat szinkronizáltuk*

Forráskód A `MultiThreadedPrinting` projekt a 18. fejezet alkönyvtárában található.

Szinkronizáció a System.Threading.Monitor típussal

A C# `lock` utasítása valójában csak egy olyan rövidített jelölés, amely a `System.Threading.Monitor` osztálytípust használja a háttérben. Miután a C#-fordító feldolgozta, a zár hatóköre valójában a következő lesz (ezt az `ildasm.exe` vagy a `reflector.exe` segítségével leellenőrizhetjük):

```
public void PrintNumbers()
{
  Monitor.Enter(threadLock);
  try
  {
    // A szálinformáció megjelenítése.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);

    // Számok kiíratása.
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
```

```
      Console.WriteLine();
    }
    finally
    {
      Monitor.Exit(threadLock);
    }
  }
```

Először is a `Monitor.Enter()` metódus a `lock` kulcsszó argumentumaként megadott száltoken végső címzettje. A zárhatókörön belüli összes kódot beburkolja egy `try` blokk. A megfelelő `finally` blokk biztosítja a száltoken feloldását (a `monitor.Exit()` metódus révén), bármilyen lehetséges futásidejű kivétel ellenére. Ha úgy módosítanánk a `MultiThreadSharedData` programot, hogy közvetlenül használja a `Monitor` típust (ahogy az imént láttuk), a kimenet ugyanez lenne.

Minthogy a `lock` kulcsszó, úgy tűnik, kevesebb kódolást igényel, mint a `System.Threading.Monitor` típus explicit használata, kérdésként merülhet fel: mik a `Monitor` típus közvetlen alkalmazásának előnyei. A válasz röviden: a közvetlen vezérlés lehetősége. Ha a `Monitor` típust használjuk, utasíthatjuk az aktív szálat, hogy várakozzon egy ideig (a `Wait()` metódus révén), majd értesítse a várakozó szálakat az aktuális szál befejeződéséről (a `Pulse()` és `PulseAll()` metódusok révén), és így tovább.

Az esetek nagy részében a C# `lock` kulcsszava elegendő. Ha meg szeretnénk vizsgálni a `Monitor` osztály további tagjait is, tanulmányozzuk a .NET keretrendszer 3.5 SDK dokumentációját.

Szinkronizáció a System.Threading.Interlocked típussal

Amíg nem látjuk a mögöttes köztes nyelvi kódot, nehéz elhinni, hogy a hozzárendelések és az aritmetikai műveletek *nem atomi* műveletek. Ezért a `System.Threading` névtér tartalmaz egy olyan típust, amely lehetővé teszi, hogy atomi módon, a `Monitor` típusénál kevesebb többletterheléssel manipulálhassunk valamilyen adatot. Az `Interlocked` osztálytípus a 18.4. táblázatban látható statikus tagokat definiálja.

Tag	Szerepe
`CompareExchange()`	Biztonságos módon teszteli két érték egyenértékűségét, és ha egyenlőek, az egyiket egy harmadikra módosítja.

Tag	Szerepe
`Decrement ()`	Biztonságosan 1-gyel csökkent egy adott értéket.
`Exchange()`	Biztonságosan felcserél két értéket.
`Increment()`	Biztonságosan 1-gyel növel egy adott értéket.

18.4. táblázat:. *A System.Threading.Interlocked típus tagjai*

Jóllehet elsőre talán nem feltűnő, de egyetlen érték atomi átalakításának folyamata elég gyakori a többszálú környezetekben. Tételezzük fel, hogy van egy `AddOne()` nevű metódus, amely megnövel egy `intVal` nevű tagváltozót. Ahelyett, hogy az alábbihoz hasonló szinkronizációs kódot írunk:

```
public void AddOne()
{
  lock(myLockToken)
  {
    intVal++;
  }
}
```

egyszerűsíthetjük a kódot a statikus `Interlocked.Increment()` metódus segítségével. Egyszerűen adjuk meg a változót, amelyet a hivatkozás megnövel. Az `Increment()` metódus nemcsak a bemeneti paraméter értékét állítja be, de az új értéket is visszaadja:

```
public void AddOne()
{
  int newVal = Interlocked.Increment(ref intVal);
}
```

Az `Increment()` és a `Decrement()` metódusokon kívül az `Interlocked` típus is lehetővé teszi numerikus és objektumadatok atomi hozzárendelését. Ha például a 83-as értéket egy tagváltozó értékéhez szeretnénk hozzárendelni, elkerülhetjük az explicit `lock` utasítást (vagy az explicit `Monitor` logikát), és használhatjuk az `Interlocked.Exchange()` metódust:

```
public void SafeAssignment()
{
  Interlocked.Exchange(ref myInt, 83);
}
```

Ha csak akkor szeretnénk értékadást végrehajtani, ha az értékadás célja pillanatnyilag egyenlő egy konkrét értékkel, akkor az `Interlocked.CompareExchange()` metódust használhatjuk az alábbi módon:

```
public void CompareAndExchange()
{
  // Ha az i értéke jelenleg 83, módosítsuk 99-re.
  Interlocked.CompareExchange(ref i, 99, 83);
}
```

Szinkronizáció a [Synchronization] attribútummal

Az utolsó szinkronizációs primitív, amelyet megvizsgálunk, a `[Synchronization]` attribútum, amely a `System.Runtime.Remoting.Contexts` névtér tagja. Ez az osztályszintű attribútum lényegében az objektum *összes* példánytagkódját zárolja a szálbiztonság érdekében. Amikor a CLR allokálja a `[Synchronization]` attribútummal ellátott objektumokat, az objektumot szinkronizált környezetbe helyezi. Azokat az objektumokat, amelyeket nem lehet eltávolítani a környezetükből, a `ContextBoundObject` osztályból kell származtatni (lásd a 17. fejezetet). Ezért, ha szálbiztossá akarjuk tenni a `Printer` osztálytípust (anélkül, hogy szálbiztos kódot írnánk az osztálytagokba), módosítsuk a definícióját a következőképpen:

```
using System.Runtime.Remoting.Contexts;
...

// Most már szálbiztos a Printer típus minden metódusa!
[Synchronization]
public class Printer : ContextBoundObject
{
  public void PrintNumbers()
  {
    ...
  }
}
```

Ez a szemlélet sok tekintetben kényelmesnek tűnhet, hiszen nem kell belemennünk olyan részletekbe, hogy valójában a típus mely részei manipulálják a szálérzékeny adatokat. Ennek a szemléletnek a legfőbb hátránya azonban az, hogy még ha egy adott metódus nem is használ szálérzékeny adatokat, a CLR *akkor is* zárolja a metódus hívásait. Ez nyilvánvalóan csökkenti a típus általános teljesítményét, ezért csak nagy körültekintéssel használjuk ezt a módszert.

A fentiekben a megosztott adatblokkok szinkronizált hozzáférési módjait ismerhettük meg. További szinkronizációs típusokat a `System.Threading` névtérben találhatunk. A szálprogramozási vizsgálataink zárásaként, bemutatunk négy további típust: a `TimerCallbacket`, a `Timert`, a `ThreadPoolt` és a `BackgroundWorkert`.

Programozás időzített visszahívással

Sok alkalmazásnak szabályos időközönként meg kell hívnia egy megadott metódust. Ilyen például az az alkalmazás, amelyik egy megadott segédfüggvény segítségével jeleníti meg az aktuális időt az állapotsávban, vagy ha azt szeretnénk, hogy az alkalmazásunk gyakran hívjon egy segédfüggvényt olyan nem kritikus háttérfeladatok elvégzésére, mint új e-mail üzenetek ellenőrzése. Ilyenkor használhatjuk a `System.Threading.Timer` típust a kapcsolódó, `TimerCallback` nevű metódusreferenciával együtt.

Megjegyzés A Windows Forms API tartalmaz egy GUI-alapú `Timer` vezérlőelemet, amely ugyanarra képes, mint a `TimerCallback` típus. Valójában a GUI-alapú `Timer` típus használata általában egyszerűbb, mert fejlesztési időben konfigurálható.

Ennek bemutatásához tételezzük fel, hogy van egy konzolalkalmazásunk (`TimerApp`), amely mindaddig kiírja másodpercenként az aktuális időt, amíg a felhasználó meg nem nyom egy billentyűt az alkalmazás leállításához. Az első lépés annak a metódusnak a megírása, amelyet a `Timer` típus meg fog hívni:

```
class Program
{
  static void PrintTime(object state)
  {
    Console.WriteLine("Time is: {0}",
    DateTime.Now.ToLongTimeString());
  }
  static void Main(string[] args)
  {
  }
}
```

A metódusnak egy egyedüli `System.Object` típusú paramétere van, és `void` értéket ad vissza. Ez nem opcionális, mivel a `TimerCallback` metódusreferencia csak az ilyen leírású metódusokat tudja meghívni. A `TimerCallback` metódusreferencia által mutatott metódusnak átadott érték bármilyen adat lehet (az e-mail példában ez a paraméter jelentheti a folyamat során használandó Microsoft Exchange Server nevét). Minthogy ez a paraméter `System.Object` típusú, több argumentumot is átadhatunk a metódusrefencia által mutatott metódusnak egy `System.Array`, illetve egy egyedi osztály vagy struktúra segítségével.

Ezután konfiguráljuk a `TimerCallback` metódusreferencia egy példányát, és adjuk át a `Timer` objektumnak. A `Timer` konstruktornak át kell adnunk a `TimerCallback` metódusreferenciát, a meghívott metódus bemenő paraméterét, az első hívás előtti várakozási időt és az egyes hívások közötti időintervallumot:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Working with Timer type *****\n");

  // Metódusreferencia létrehozása a Timer típushoz.
  TimerCallback timeCB = new TimerCallback(PrintTime);

  // Az időzítő beállításainak megalapozása.
  Timer t = new Timer(
    timeCB,  // A TimerCallback metódusreferencia-típus.
    null,    // Bármilyen információ, amely beadandó a meghívott
             // metódusba
             // (null akkor, ha nincs ilyen).
    0,       // Az indítás előtti várakozási idő.
    1000);   // A hívások közti időintervallum (ezredmásodpercben).

  Console.WriteLine("Hit key to terminate...");
  Console.ReadLine();
}
```

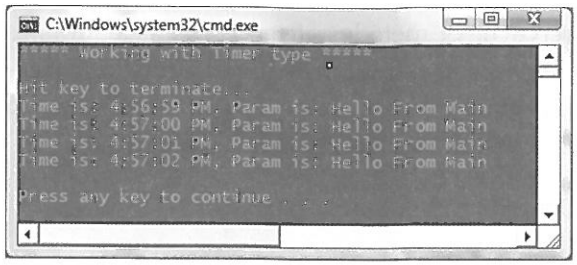
Ekkor a `PrintTime()` metódust nagyjából egy másodpercenként hívjuk, és nem továbbítunk egyéb adatot az említett metódusba. Ha adatokat szeretnénk átadni a metódusreferencia által mutatott metódusnak, a második konstruktorparaméter `null` értékét helyettesítsük be a megfelelő adattal:

```
// Az időzítő beállításainak megalapozása.
Timer t = new Timer(timeCB, "Hello From Main", 0, 1000);
```

Azután az alábbi módon férhetünk hozzá a bejövő adathoz:

```
static void PrintTime(object state)
{
  Console.WriteLine("Time is: {0}, Param is: {1}",
    DateTime.Now.ToLongTimeString(), state.ToString());
}
```

A 18.11. ábrán láthatjuk a kimenetet.



18.11. ábra: *Timer munka közben*

Forráskód A `TimerApp` projekt a 18. fejezet alkönyvtárában található.

A CLR ThreadPool

A következőkben a CLR-szálkészlet szerepét vizsgáljuk meg. Amikor aszinkron módon hívunk egy metódust a metódusreferencia-típusok segítségével (a `BeginInvoke()` metódus révén), a CLR nem a szó szoros értelmében hoz létre új szálat. A hatékonyság érdekében egy metódusreferencia `BeginInvoke()` metódusa felhasználja a futtatókörnyezet által fenntartott feldolgozószálkészletet. Azért, hogy kommunikálni tudjunk ezzel a várakozó szálkészlettel, a `System.Threading` névtér biztosítja a `ThreadPool` osztálytípust.

Ha szeretnénk egy metódushívást beállítani a sorba azért, hogy a készlet egyik dolgozószála fel tudja dolgozni, a `ThreadPool.QueueUserWorkItem()` metódust használhatjuk. Ez a metódus túlterhelt, így átadhatunk egy opcionális `System.Object` típust is az egyedi állapotadatok számára, a `WaitCallback` metódusreferencia egy példánya mellett:

```
public sealed class ThreadPool
{
  ...
  public static bool QueueUserWorkItem(WaitCallback callBack);
  public static bool QueueUserWorkItem(WaitCallback callBack,
    object state);
}
```

A `WaitCallback` metódusreferencia bármely olyan metódusra rá tud mutatni, amelynek egyedüli paramétere egy `System.Object` (amely az opcionális állapotadatot jelképezi), és eredményül nem ad vissza semmit. Ha nem biztosítunk egy `System.Object` típust a `QueueUserWorkItem()` meghívásakor, a CLR automatikusan `null` értéket ad át. A CLR-szálkészlet által használt metódu-

sok sorba állításának bemutatásához nézzük meg a következő programot, ez ismét a `Printer` típust használja. Ebben az esetben azonban nem manuálisan hozzuk létre a `Thread` típus tömbjét, hanem a készlet tagjai közül rendelünk hozzá a `PrintNumbers()` metódushoz:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with the CLR Thread Pool *****\n");

    Console.WriteLine("Main thread started. ThreadID = {0}",
      Thread.CurrentThread.ManagedThreadId);

    Printer p = new Printer();

    WaitCallback workItem = new WaitCallback(PrintTheNumbers);

    // A metódus sorba állítása tíz alkalommal.
    for (int i = 0; i < 10; i++)
    {
      ThreadPool.QueueUserWorkItem(workItem, p);
    }
    Console.WriteLine("All tasks queued");
    Console.ReadLine();
  }

  static void PrintTheNumbers(object state)
  {
    Printer task = (Printer)state;
    task.PrintNumbers();
  }
}
```

Vajon hasznos lenne-e a CLR által fenntartott szálkészletet használni inkább, mint explicit módon `Thread` objektumokat létrehozni? A szálkészlet felhasználásának előnyei a következők:

- A szálkészlet hatékonyan kezeli a szálakat, minimalizálva a létrehozandó, elindítandó és leállítandó szálak számát.
- A szálkészlet használatakor egyéb problémákra koncentrálhatunk, nem az alkalmazás szálinfrastruktúrájára.

Néhány esetben azonban jobb manuálisan kezelni a szálakat, például:

- Ha előtérbeli szálakra van szükségünk, vagy be kell állítanunk a szálprioritást. A készletben lévő szálak *mindig* háttérszálak, alapértelmezett prioritással (`ThreadPriority.Normal`).
- Ha fix azonosítóval rendelkező szálra van szükségünk azért, hogy név alapján szakítsuk meg, függesszük fel vagy vizsgáljuk meg.

Forráskód A `ThreadPoolApp` projekt a 18. fejezet alkönyvtárában található.

A BackgroundWorker komponens

Végül megvizsgáljuk a `BackgroundWorker` száltípust, amely az `mscorlib.dll` `System.ComponentModel` névterében van definiálva. A `BackgroundWorker` grafikus Windows Forms asztali alkalmazások készítésekor hasznos, vagy amikor hosszú futásidejű feladatokat kell végrehajtani (távoli webszolgáltatás meghívása, adatbázis-tranzakció végrehajtása, nagyméretű fájl letöltése stb.) az alkalmazás fő felhasználói felületét futtató száltól különböző szálon.

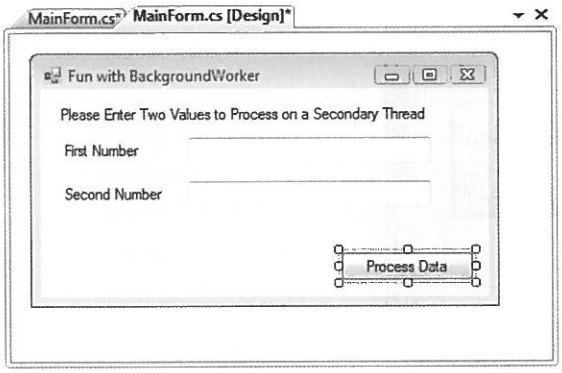
A `BackgroundWorkerrel` komolyabb nehézségek nélkül is tudunk többszálú grafikus alkalmazásokat készíteni. Ennek a típusnak a programozási modellje szinte ugyanazt a szálszintaxist használja fel, mint az aszinkron metódusreferenciáké, azaz a típus használatát könnyen meg lehet tanulni.

A `BackgroundWorker` használatához egyszerűen adjuk meg, hogy melyik metódust hajtsa végre a háttérben, és hívjuk meg a `RunWorkerAsync()` metódust. A hívószál (általában az elsődleges szál) továbbra is normál módon fut, amíg a feldolgozómetódus aszinkron módon. Amikor az időigényes metódus befejeződött, a `BackgroundWorker` típus értesíti a hívószálat a `RunWorkerCompleted` eseménnyel. A hozzárendelt eseménykezelő egy bejövő paramétert biztosít, ennek révén hozzáférhetünk a művelet eredményeihez (ha léteznek).

Megjegyzés A következő példa azt feltételezi, hogy bizonyos mértékben jártasak vagyunk a Windows Forms használatával végzett grafikus felület fejlesztésében. Ha ez nem így van, akkor nézzünk utána ennek a témának, mielőtt tovább haladnánk.

A BackgroundWorker típus használata

Ahhoz, hogy ennek a felhasználói felületet futtató szálösszetevőnek a használatát megértsük, hozzunk létre egy új, `WinFormsBackgroundWorkerThread` nevű Windows Forms-alkalmazást. Hogy az eddigi numerikus műveleti példáknál maradjunk, készítsünk egy egyszerű felhasználói felületet, amely lehetővé teszi, hogy a felhasználó két értéket adhasson be feldolgozásra (`TextBox` típusok révén), illetve egy `Button` típust a háttérművelet elindításához. Mindenképpen adjunk a felhasználói felület minden elemének egy megfelelő nevet a Properties ablak `Name` tulajdonságával. A 18.12. ábra egy lehetséges elrendezést mutat be.



18.12. ábra: *A Windows Forms felhasználóifelület- alkalmazás elrendezése*

Ezután készítsünk eseménykezelőt a `Button` típushoz: kattintsunk duplán a gombra az űrlaptervezőben:

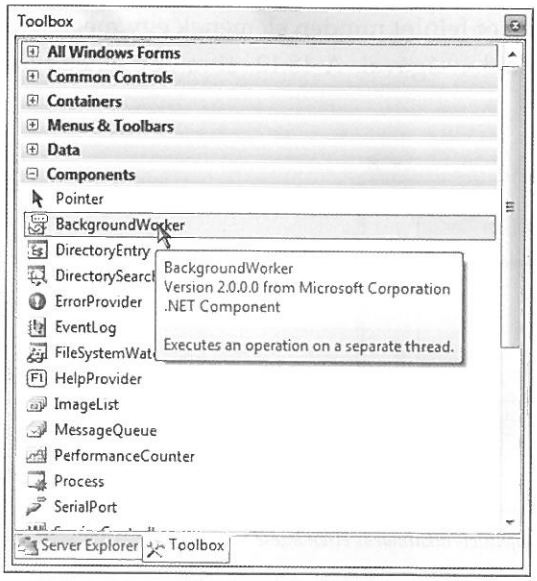
```
private void btnProcessData_Click(object sender, EventArgs e)
{
}
```

Nyissuk meg a Toolbox Components régióját, keressük meg a `BackgroundWorker` komponenst (lásd 18.13. ábra), és a típus egyik példányát húzzuk az űrlaptervezőbe.

Ekkor a típus egy változóját látjuk a tervező komponenstálcáján. Nevezzük át `ProcessNumbersBackgroundWorkerre` a komponenst a Properties ablak segítségével. Ezután váltsunk át a Properties ablak Event ablaktáblájára (a „villám" ikonra kattintva), és kattintsunk duplán a `DoWork` és a `RunWorkerCompleted` események nevére. Az eredmény az alábbi két új eseménykezelő lesz, amelyeket hozzáadtunk a kezdeti űrlapból származó típushoz:

```
private void ProcessNumbersBackgroundWorker_DoWork(object sender,
  DoWorkEventArgs e)
{
}

private void
  ProcessNumbersBackgroundWorker_RunWorkerCompleted(object sender,
  RunWorkerCompletedEventArgs e)
{
}
```



18.13. ábra: *A BackgroundWorker típus*

A `DoWork` eseménykezelő azt a metódust reprezentálja, amelyet a `BackgroundWorker` hív meg a másodlagos végrehajtási szálon. A kezelő második paramétere egy `DoWorkEventArgs` típus, amely minden olyan argumentumot tartalmaz, amelyre a másodlagos szálnak szüksége van a feladata elvégzéséhez. Amikor meghívjuk a `RunWorkerAsync()` metódust, hogy elindítsa ezt a szálat, lehetőségünk nyílik a kapcsolódó adatok átadására (hasonlóan, mint a fejezet korábbi részében alkalmazott `ParameterizedThreadStart` metódusreferencia-típus használatakor).

A `RunWorkerCompleted` esemény azt a metódust reprezentálja, amelyet a `BackgroundWorker` hív meg, mikor a háttérművelet befejeződött. A `RunWorkerCompletedEventArgs` típus használatával lehetőségünk nyílik az aszinkron művelet visszatérési értékeihez való hozzáférésre.

Adatfeldolgozás a BackgroundWorker típussal

Az alábbiakban megnézzük a felhasználói inputok feldolgozásának részleteit. Amikor értesíteni akartuk a `BackgroundWorker` típust, hogy indítsa el a másodlagos végrehajtási szálat, a `RunWorkerAsync()` metódust kellett meghívnunk. Ilyenkor lehetőségünk van egy `System.Object` típusú (azaz gyakorlatilag bármilyen) adat átadására, amely továbbadódik a `DoWork` által meghívott metódusnak. Használjuk fel újra az `AddParam` osztályt, amelyet a `ParameterizedThreadStart` példában hoztunk létre:

```
class AddParams
{
  public int a, b;

  public AddParams(int numb1, int numb2)
  {
    a = numb1;
    b = numb2;
  }
}
```

Ha ez a segédosztály a helyére került, a következőképpen valósíthatjuk meg a `Button` típus `Click` eseménykezelőjét:

```
private void btnProcessData_Click(object sender, EventArgs e)
{
  try
  {
    // Először kapjuk meg a felhasználói adatokat (számként).
    int numbOne = int.Parse(txtFirstNumber.Text);
    int numbTwo = int.Parse(this.txtSecondNumber.Text);
    AddParams args = new AddParams(numbOne, numbTwo);

    // Most indítsuk el az új szálat, és adjunk át
    // argumentumváltozókat.
    ProcessNumbersBackgroundWorker.RunWorkerAsync(args);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Amint meghívjuk a `RunWorkerAsync()` metódust, bekövetkezik a `DoWork` esemény, amelyet az eseménykezelőnk elkap. Valósítsuk meg ezt a típust, hogy hozzáférjünk az `AddParam` objektumhoz, a bejövő `DoWorkEventArgs` Argument

tulajdonságának a segítségével. Egy hosszadalmas művelet szimulálásához az aktuális szálat mintegy öt másodpercre felfüggesztjük. Ezután visszaadjuk az értéket a `DoWorkEventArgs` típus `Result` tulajdonságának a segítségével:

```
private void ProcessNumbersBackgroundWorker_DoWork(object sender,
  DoWorkEventArgs e)
{
  // Szerezzük meg a bejövő AddParam objektumot.
  AddParams args = (AddParams)e.Argument;

  // Mesterséges késleltetés.
  System.Threading.Thread.Sleep(5000);

  // Adjuk vissza az értéket.
  e.Result = args.a + args.b;
}
```

Végül, ha a `BackgroundWorker` típus kilépett a `DoWork` kezelő hatóköréből, a `RunWorkerCompleted` esemény bekövetkezik. A regisztrált kezelőnk egyszerűen csak megjeleníti a művelet eredményét a `RunWorkerCompletedEventArgs.Result` tulajdonság használatával:

```
private void ProcessNumbersBackgroundWorker_RunWorkerCompleted(
  object sender, RunWorkerCompletedEventArgs e)
{
  MessageBox.Show(e.Result.ToString(), "Your result is");
}
```

Az alkalmazás futtatásakor azt tapasztaljuk, hogy az adatok feldolgozása közben a felhasználói felületet hosztoló szál még mindig teljesen válaszképes (pl. az ablak átméretezhető, áthelyezhető, kis méretűvé tehető stb.). Ennek további érzékeltetéséhez adjunk egy új `TextBox` típust az űrlaphoz, és ellenőrizzük, hogy tudunk-e adatokat bevinni a felhasználói felületen, mialatt az öt másodperces összeadási művelet aszinkron módon végbemegy a háttérben.

Forráskód A `WinFormsBackgroundWorkerThread` pojekt a 18. fejezet alkönyvtárában található.

A .NET-alapú többszálú programozás vizsgálatának a végére érve, azt még meg kell említenünk, hogy a `System.Threading` névtér természetesen jóval több típust definiál azon kívül, mint amennyiről a fejezetben szó volt.

Összefoglalás

Először azt vizsgáltuk meg, hogy hogyan konfiguráljuk a .NET-metódusreferencia típusait ahhoz, hogy aszinkron módon hajtsanak végre metódusokat. A `BeginInvoke()` és az `EndInvoke()` metódusok lehetővé teszik, hogy közvetett módon, komolyabb nehézségek nélkül kezeljük a háttérszálakat. Megismertük az `IAsyncResult` interfészt és az `AsyncResult` osztálytípust. Ezek a típusok a hívószál szinkronizálásának és a lehetséges függvényvisszatérési-értékek hozzáférésének számos módját biztosítják.

A továbbiakban a `System.Threading` névtér szerepét tanulmányoztuk. Amikor egy alkalmazás kiegészítő végrehajtási szálakat hoz létre, az eredmény az, hogy az adott program több feladat (látszólag) egyidejű elvégzésére is képes lesz. Megnéztük a szálérzékeny kódblokkok megvédésének számos módját is. Ezek révén biztosíthatjuk, hogy a megosztott erőforrások ne váljanak használhatatlan, hamis adatokká.

A fejezet arra is rámutatott, hogy a CLR fenntart egy belső szálkészletet a teljesítmény és a kényelem érdekében. Végül, de nem utolsósorban, a `BackgroundWorker` típus használatával ismerkedtünk meg, amelynek segítségével könnyedén indíthatunk be új végrehajtási szálakat egy GUI-alapú alkalmazásban.

TIZENKILENCEDIK FEJEZET

A CIL és a dinamikus szerelvények szerepe

A fejezet célja kettős. Először a korábbiaknál részletesebben megvizsgáljuk a köztes nyelv (CIL - Common Intermediate Language) szintaxisát és szemantikáját. .NET-programozóként anélkül is boldogulhatunk, hogy ismernénk a CIL-kód részleteit. Ám ha megtanuljuk a CIL alapjait, sokkal jobban meg fogjuk érteni a .NET néhány „bűvös" vonatkozását (pl. a nyelvközi származtatást).

A továbbiakban megvizsgáljuk a `System.Reflection.Emit` névtér szerepét. Ezekkel a típusokkal olyan programokat írhatunk, amelyek a .NET-szerelvényeket futásidőben hozzák létre a memóriában. A memóriában definiált és futtatott szerelvényeket *dinamikus szerelvényeknek* nevezzük. A .NET-fejlesztésnek ez a szemlélete teszi szükségessé a CIL nyelv használatát, hiszen pontosan meg kell határoznunk a szerelvények szerkesztéséhez alkalmazott CIL-utasításkészletet.

A CIL-programozás természete

A .NET platform igazi „anyanyelve" a CIL. Ha egy általunk választott nyelven (C#, VB, COBOL.NET stb.) készítünk .NET-szerelvényt, az alkalmazott fordító a forráskódunkat CIL-kifejezéssé alakítja. Mint minden más porgramozói nyelv, a CIL is számos strukturális és megvalósításcentrikus tokennel rendelkezik. Minthogy a CIL .NET programozási nyelv, így a .NET-szerelvényeinket létrehozhatjuk közvetlenül a CIL és a CIL-fordító (`ilasm.exe`) használatával, ezek a .NET keretrendszer 3.5 SDK-val együtt járnak.

Bár kevés programozó készíti el a teljes .NET-alkalmazását közvetlenül a CIL-lel, ám a CIL mégis rendkívül érdekes, intellektuális tevékenységet kínál. Minél jobban megértjük a CIL nyelvtanát, annál inkább tudunk érvényesülni a haladó .NET-fejlesztés világában. Mire is lehet képes egy CIL nyelvet elsajátított programozó? Nézzünk néhány példát:

- Ismeri azt, hogy hogyan képződnek le a különböző .NET-programnyelvek megfelelő kulcsszavai CIL-tokenekké.
- Tisztában van a meglévő .NET-szerelvények lebontásával, a CIL-kód szerkesztésével, valamint a módosított kód újrafordításával .NET-binárissá.
- A `Sysetm.Reflection.Emit` névtér segítségével dinamikus szerelvényeket tud létrehozni.
- A CTS (egységes típusrendszer) előnyeit ki tudja használni. A CIL szintjén a CTS olyan aspektusai is elérhetővé válnak, amelyeket egy-egy magasabb szintű nyelv nem támogat. A CIL az egyetlen olyan .NET-nyelv, amellyel a CTS valamennyi aspektusa kezelhető. A nyers CIL használatával például globális szintű tagokat és mezőket tudunk definiálni (ez a C#-ban nem lehetséges).

Ha *nem* foglalkozunk a CIL-kód részleteivel, akkor is teljesen elsajátíthatjuk a C# és a .NET alaposztálykönyvtáraival kapcsolatos ismereteket. Voltaképpen a CIL ismerete azzal hozható párhuzamba, ahogyan a C(++)-programozó ismeri az assembly nyelvet. Azok, akik értik ennek az alacsony szintű „ingoványnak" minden részletét, képesek a meglévő feladathoz magasabb szintű megoldást létrehozni, valamint a mögöttes programozó (és futásidejű) környezetet mélyebben is megérteni. Így tehát vizsgáljuk meg részletesebben is a CIL nyelvet.

Megjegyzés A fejezet nem ismerteti a CIL szintaxisának és szemantikájának átfogó kezelését. Ha ennél még részletesebb tudásanyagra lenne szükségünk, vegyük kézbe Jason Bock *CIL Programming: Under the Hood of .NET* (Apress, 2002) című könyvét.

CIL-direktívák, -attribútumok és -vezérlőkódok

Ha egy alacsony szintű nyelvet kezdünk vizsgálni, akkor nyilván találunk új elnevezéseket a teljesen megszokott fogalmakra. Ha például a következő szókészletet mutatnák nekünk:

```
{new, public, this, base, get, set, explicit, unsafe, enum,
 operator, partial}
```

akkor bizonyára arra gondolnánk, hogy a C# nyelv kulcsszavaival van dolgunk. Ha azonban jobban megnézzük a készlet elemeit, észrevesszük, hogy mindegyik C#-kulcsszó ugyan, de a szemantikájuk egészen eltérő. Az `enum` kulcsszó például egy `System.Enum` származtatott típust definiál, míg a `this` és a `base` kulcsszavak az aktuális objektumra vagy az objektum szülőosztályára történő hivatkozást teszik lehetővé. Az `unsafe` kulcsszó segítségével olyan kódblokkot hozhatunk létre, amelyet nem lehet közvetlenül ellenőrizni a CLR-en keresztül, az `operator` kulcsszó pedig lehetővé teszi rejtett (speciálisan elnevezett) metódus létrehozását, amelyet akkor hívunk meg, amikor egy sajátságos C#-operátort használunk (pl. a plusz jel).

Ellentétben a magasabb szintű nyelvekkel (pl. C#) a CIL nem csupán meghatározza a generikus szókészletet *önmagában*, hanem a CIL-fordítóprogram három tágabb kategóriába sorolja szemantikájuk alapján a tokenkészletet:

- CIL–direktívák,
- CIL–attribútumok,
- CIL-vezérlőkódok (opcodes).

Minden CIL-tokennek egyéni szintaxisa van, és ezekből a tokenekből épülnek fel a .NET-szerelvények.

A CIL-direktívák szerepe

A tokenek első nagy csoportja a .NET-szerelvények felépítésének leírására szolgálnak. A CIL-direktívák segítségével tájékoztatjuk a CIL-fordítót arról, hogy hogyan definiálja azokat a névtere(ke)t, típus(oka)t és tago(ka)t, amelyek egy szerelvényt feltöltenek.

A direktívákat szintaktikailag egy pont (.) prefixummal (pl. `.namespace`, `.class`, `.publickeytoken`, `.method`, `.assembly` stb.) jelképezzük. Így, ha a `*.il` fájl (a köztes nyelvi kódot tartalmazó állományok hagyományos kiterjesztése) egyetlen `.namespace` direktívából és három `.class` direktívából áll, akkor a CIL-fordító olyan szerelvényt generál, amely három .NET-osztálytípust tartalmazó .NET-névteret definiál.

A CIL-attribútumok szerepe

Számos esetben a CIL-direktívák egy adott .NET-típus vagy -típustag definíciójának teljes kifejezéséhez önmagukban kevésnek bizonyulnak. Ezért sok CIL-direktívát tovább kell specifikálni különféle CIL-*attribútumokkal*, meghatározva ezzel a direktíva feldolgozását. A `.class` direktívát elláthatjuk `public` attribútummal (a láthatóság típusát határozva meg), `extends` attribútummal (explicit módon definiálva a típus ősosztályát) és `implements` attribútummal (a típus által támogatott interfészkészlet felsorolásához).

A CIL-vezérlőkódok szerepe

Miután különböző direktívák és kapcsolt attribútumok segítségével a CIL nyelvben meghatároztuk a .NET-szerelvényt, -névteret és -típuskészletet, utolsó feladatunk a típus megvalósítási logikájának megadása. Ez pedig az *vezérlőkódok* (röviden *opcode*) feladata. Más alacsony szintű nyelvek után sok CIL-opcode esetleg rejtélyesnek és kimondhatatlannak tűnhet. Ha például egy sztringváltozót szeretnénk meghatározni, nem egy barátságos `LoadString` vezérlőkódot használunk, hanem az `ldstr` elnevezésűt.

Néhány CIL-opcode-ot egészen természetes módon leképezhetünk C#-megfelelőjükből (pl. `box`, `unbox`, `throw` és `sizeof`). A CIL vezérlőkódjait mindig egy tag implementációjának hatókörén belül használjuk, és a CIL-direktívákkal ellentétben sohasem pont prefixummal.

A CIL-vezérlőkódok/CIL-mnemonikok megkülönböztetése

Az operátorkódok, például az `ldstr`, egy bizonyos típus tagjainak implementálására szolgálnak. A valóságban azonban az `ldstr` és az ehhez hasonló tokenek az aktuális *bináris CIL-vezérlőkódok, CIL-mnemonik*. A különbség tisztázásához tegyük fel, hogy a következő metódust hoztuk létre a C# nyelvben:

```
static int Add(int x, int y)
{
  return x + y;
}
```

```
      lock (threadLock)
      {
        ...
      }
    }
}
```

Ha bármikor megvizsgáljuk a `PrintNumbers()` metódust, akkor láthatjuk, hogy az a megosztott erőforrás, amelynek eléréséért a szálak versengenek, a konzolablak. Ezért, ha az összes a `Console` típussal kapcsolatos műveletet zároljuk a következőképpen:

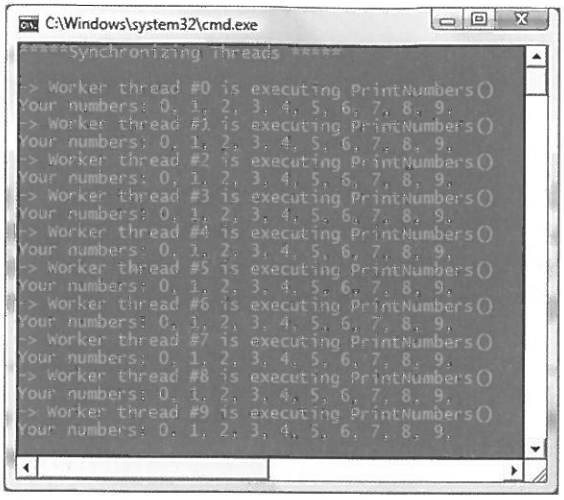
```
public void PrintNumbers()
{
  // Használjuk a privát objektumzároló tokent.
  lock (threadLock)
  {
    // A szálinformáció megjelenítése.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);

    // Számok kiíratása.
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
    Console.WriteLine();
  }
}
```

akkor gyakorlatilag megalkottunk egy olyan metódust, amely lehetővé teszi, hogy az aktuális szál befejezze a feladatát. Ha egy szál belép a zárolás hatókörébe, a zár tokenjéhez (ebben az esetben az aktuális objektum referenciája) egyéb szálak nem férhetnek mindaddig hozzá, amíg a zárolást fel nem oldjuk. Így, ha az A szál hozzájutott a zár tokenjéhez, más szálak addig nem léphetnek be a hatókörbe, amíg az A szál el nem engedi a zár tokenjét.

Megjegyzés Ha egy statikus metódus kódját szeretnénk zárolni, egyszerűen deklaráljunk egy privát statikus objektum-tagváltozót a zárolás tokenjeként.

Ha lefuttatjuk az alkalmazást, látható, hogy minden szálnak elég lehetősége van a feladata elvégzésére (lásd 18.10. ábra).



18.10. ábra: *Most már az összes szálat szinkronizáltuk*

Forráskód A `MultiThreadedPrinting` projekt a 18. fejezet alkönyvtárában található.

Szinkronizáció a System.Threading.Monitor típussal

A C# `lock` utasítása valójában csak egy olyan rövidített jelölés, amely a `System.Threading.Monitor` osztálytípust használja a háttérben. Miután a C#-fordító feldolgozta, a zár hatóköre valójában a következő lesz (ezt az `ildasm.exe` vagy a `reflector.exe` segítségével leellenőrizhetjük):

```
public void PrintNumbers()
{
  Monitor.Enter(threadLock);
  try
  {
    // A szálinformáció megjelenítése.
    Console.WriteLine("-> {0} is executing PrintNumbers()",
      Thread.CurrentThread.Name);

    // Számok kiíratása.
    Console.Write("Your numbers: ");
    for (int i = 0; i < 10; i++)
    {
      Random r = new Random();
      Thread.Sleep(1000 * r.Next(5));
      Console.Write("{0}, ", i);
    }
```

```
      Console.WriteLine();
    }
    finally
    {
      Monitor.Exit(threadLock);
    }
  }
```

Először is a `Monitor.Enter()` metódus a `lock` kulcsszó argumentumaként megadott száltoken végső címzettje. A zárhatókörön belüli összes kódot beburkolja egy `try` blokk. A megfelelő `finally` blokk biztosítja a száltoken feloldását (a `monitor.Exit()` metódus révén), bármilyen lehetséges futásidejű kivétel ellenére. Ha úgy módosítanánk a `MultiThreadSharedData` programot, hogy közvetlenül használja a `Monitor` típust (ahogy az imént láttuk), a kimenet ugyanez lenne.

Minthogy a `lock` kulcsszó, úgy tűnik, kevesebb kódolást igényel, mint a `System.Threading.Monitor` típus explicit használata, kérdésként merülhet fel: mik a `Monitor` típus közvetlen alkalmazásának előnyei. A válasz röviden: a közvetlen vezérlés lehetősége. Ha a `Monitor` típust használjuk, utasíthatjuk az aktív szálat, hogy várakozzon egy ideig (a `Wait()` metódus révén), majd értesítse a várakozó szálakat az aktuális szál befejeződéséről (a `Pulse()` és `PulseAll()` metódusok révén), és így tovább.

Az esetek nagy részében a C# `lock` kulcsszava elegendő. Ha meg szeretnénk vizsgálni a `Monitor` osztály további tagjait is, tanulmányozzuk a .NET keretrendszer 3.5 SDK dokumentációját.

Szinkronizáció a System.Threading.Interlocked típussal

Amíg nem látjuk a mögöttes köztes nyelvi kódot, nehéz elhinni, hogy a hozzárendelések és az aritmetikai műveletek *nem atomi* műveletek. Ezért a `System.Threading` névtér tartalmaz egy olyan típust, amely lehetővé teszi, hogy atomi módon, a `Monitor` típusénál kevesebb többletterheléssel manipulálhassunk valamilyen adatot. Az `Interlocked` osztálytípus a 18.4. táblázatban látható statikus tagokat definiálja.

Tag	Szerepe
`CompareExchange()`	Biztonságos módon teszteli két érték egyenértékűségét, és ha egyenlőek, az egyiket egy harmadikra módosítja.

Tag	Szerepe
`Decrement ()`	Biztonságosan 1-gyel csökkent egy adott értéket.
`Exchange()`	Biztonságosan felcserél két értéket.
`Increment()`	Biztonságosan 1-gyel növel egy adott értéket.

18.4. táblázat:. *A System.Threading.Interlocked típus tagjai*

Jóllehet elsőre talán nem feltűnő, de egyetlen érték atomi átalakításának folyamata elég gyakori a többszálú környezetekben. Tételezzük fel, hogy van egy `AddOne()` nevű metódus, amely megnövel egy `intVal` nevű tagváltozót. Ahelyett, hogy az alábbihoz hasonló szinkronizációs kódot írunk:

```
public void AddOne()
{
  lock(myLockToken)
  {
    intVal++;
  }
}
```

egyszerűsíthetjük a kódot a statikus `Interlocked.Increment()` metódus segítségével. Egyszerűen adjuk meg a változót, amelyet a hivatkozás megnövel. Az `Increment()` metódus nemcsak a bemeneti paraméter értékét állítja be, de az új értéket is visszaadja:

```
public void AddOne()
{
  int newVal = Interlocked.Increment(ref intVal);
}
```

Az `Increment()` és a `Decrement()` metódusokon kívül az `Interlocked` típus is lehetővé teszi numerikus és objektumadatok atomi hozzárendelését. Ha például a 83-as értéket egy tagváltozó értékéhez szeretnénk hozzárendelni, elkerülhetjük az explicit `lock` utasítást (vagy az explicit `Monitor` logikát), és használhatjuk az `Interlocked.Exchange()` metódust:

```
public void SafeAssignment()
{
  Interlocked.Exchange(ref myInt, 83);
}
```

Ha csak akkor szeretnénk értékadást végrehajtani, ha az értékadás célja pillanatnyilag egyenlő egy konkrét értékkel, akkor az `Interlocked.CompareExchange()` metódust használhatjuk az alábbi módon:

```
public void CompareAndExchange()
{
  // Ha az i értéke jelenleg 83, módosítsuk 99-re.
  Interlocked.CompareExchange(ref i, 99, 83);
}
```

Szinkronizáció a [Synchronization] attribútummal

Az utolsó szinkronizációs primitív, amelyet megvizsgálunk, a `[Synchronization]` attribútum, amely a `System.Runtime.Remoting.Contexts` névtér tagja. Ez az osztályszintű attribútum lényegében az objektum *összes* példánytagkódját zárolja a szálbiztonság érdekében. Amikor a CLR allokálja a `[Synchronization]` attribútummal ellátott objektumokat, az objektumot szinkronizált környezetbe helyezi. Azokat az objektumokat, amelyeket nem lehet eltávolítani a környezetükből, a `ContextBoundObject` osztályból kell származtatni (lásd a 17. fejezetet). Ezért, ha szálbiztossá akarjuk tenni a `Printer` osztálytípust (anélkül, hogy szálbiztos kódot írnánk az osztálytagokba), módosítsuk a definícióját a következőképpen:

```
using System.Runtime.Remoting.Contexts;
...

// Most már szálbiztos a Printer típus minden metódusa!
[Synchronization]
public class Printer : ContextBoundObject
{
  public void PrintNumbers()
  {
    ...
  }
}
```

Ez a szemlélet sok tekintetben kényelmesnek tűnhet, hiszen nem kell belemennünk olyan részletekbe, hogy valójában a típus mely részei manipulálják a szálérzékeny adatokat. Ennek a szemléletnek a legfőbb hátránya azonban az, hogy még ha egy adott metódus nem is használ szálérzékeny adatokat, a CLR *akkor is* zárolja a metódus hívásait. Ez nyilvánvalóan csökkenti a típus általános teljesítményét, ezért csak nagy körültekintéssel használjuk ezt a módszert.

A fentiekben a megosztott adatblokkok szinkronizált hozzáférési módjait ismerhettük meg. További szinkronizációs típusokat a `System.Threading` névtérben találhatunk. A szálprogramozási vizsgálataink zárásaként, bemutatunk négy további típust: a `TimerCallbacket`, a `Timert`, a `ThreadPoolt` és a `BackgroundWorkert`.

Programozás időzített visszahívással

Sok alkalmazásnak szabályos időközönként meg kell hívnia egy megadott metódust. Ilyen például az az alkalmazás, amelyik egy megadott segédfüggvény segítségével jeleníti meg az aktuális időt az állapotsávban, vagy ha azt szeretnénk, hogy az alkalmazásunk gyakran hívjon egy segédfüggvényt olyan nem kritikus háttérfeladatok elvégzésére, mint új e-mail üzenetek ellenőrzése. Ilyenkor használhatjuk a `System.Threading.Timer` típust a kapcsolódó, `TimerCallback` nevű metódusreferenciával együtt.

Megjegyzés A Windows Forms API tartalmaz egy GUI-alapú `Timer` vezérlőelemet, amely ugyanarra képes, mint a `TimerCallback` típus. Valójában a GUI-alapú `Timer` típus használata általában egyszerűbb, mert fejlesztési időben konfigurálható.

Ennek bemutatásához tételezzük fel, hogy van egy konzolalkalmazásunk (`TimerApp`), amely mindaddig kiírja másodpercenként az aktuális időt, amíg a felhasználó meg nem nyom egy billentyűt az alkalmazás leállításához. Az első lépés annak a metódusnak a megírása, amelyet a `Timer` típus meg fog hívni:

```
class Program
{
  static void PrintTime(object state)
  {
    Console.WriteLine("Time is: {0}",
    DateTime.Now.ToLongTimeString());
  }
  static void Main(string[] args)
  {
  }
}
```

A metódusnak egy egyedüli `System.Object` típusú paramétere van, és `void` értéket ad vissza. Ez nem opcionális, mivel a `TimerCallback` metódusreferencia csak az ilyen leírású metódusokat tudja meghívni. A `TimerCallback` metódusreferencia által mutatott metódusnak átadott értek bármilyen adat lehet (az e-mail példában ez a paraméter jelentheti a folyamat során használandó Microsoft Exchange Server nevét). Minthogy ez a paraméter `System.Object` típusú, több argumentumot is átadhatunk a metódusrefencia által mutatott metódusnak egy `System.Array`, illetve egy egyedi osztály vagy struktúra segítségével.

Ezután konfiguráljuk a `TimerCallback` metódusreferencia egy példányát, és adjuk át a `Timer` objektumnak. A `Timer` konstruktornak át kell adnunk a `TimerCallback` metódusreferenciát, a meghívott metódus bemenő paraméterét, az első hívás előtti várakozási időt és az egyes hívások közötti időintervallumot:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Working with Timer type *****\n");

  // Metódusreferencia létrehozása a Timer típushoz.
  TimerCallback timeCB = new TimerCallback(PrintTime);

  // Az időzítő beállításainak megalapozása.
  Timer t = new Timer(
    timeCB,  // A TimerCallback metódusreferencia-típus.
    null,    // Bármilyen információ, amely beadandó a meghívott
             // metódusba
             // (null akkor, ha nincs ilyen).
    0,       // Az indítás előtti várakozási idő.
    1000);   // A hívások közti időintervallum (ezredmásodpercben).

  Console.WriteLine("Hit key to terminate...");
  Console.ReadLine();
}
```

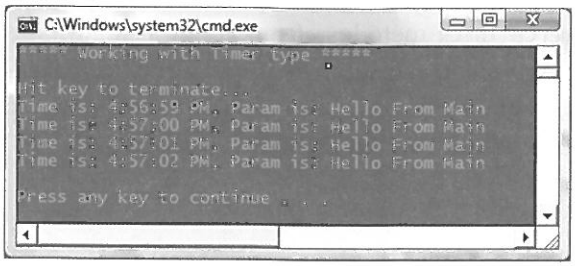
Ekkor a `PrintTime()` metódust nagyjából egy másodpercenként hívjuk, és nem továbbítunk egyéb adatot az említett metódusba. Ha adatokat szeretnénk átadni a metódusreferencia által mutatott metódusnak, a második konstruktorparaméter `null` értékét helyettesítsük be a megfelelő adattal:

```
// Az időzítő beállításainak megalapozása.
Timer t = new Timer(timeCB, "Hello From Main", 0, 1000);
```

Azután az alábbi módon férhetünk hozzá a bejövő adathoz:

```
static void PrintTime(object state)
{
  Console.WriteLine("Time is: {0}, Param is: {1}",
    DateTime.Now.ToLongTimeString(), state.ToString());
}
```

A 18.11. ábrán láthatjuk a kimenetet.



18.11. ábra: *Timer munka közben*

Forráskód A `TimerApp` projekt a 18. fejezet alkönyvtárában található.

A CLR ThreadPool

A következőkben a CLR-szálkészlet szerepét vizsgáljuk meg. Amikor aszinkron módon hívunk egy metódust a metódusreferencia-típusok segítségével (a `BeginInvoke()` metódus révén), a CLR nem a szó szoros értelmében hoz létre új szálat. A hatékonyság érdekében egy metódusreferencia `BeginInvoke()` metódusa felhasználja a futtatókörnyezet által fenntartott feldolgozószálkészletet. Azért, hogy kommunikálni tudjunk ezzel a várakozó szálkészlettel, a `System.Threading` névtér biztosítja a `ThreadPool` osztálytípust.

Ha szeretnénk egy metódushívást beállítani a sorba azért, hogy a készlet egyik dolgozószála fel tudja dolgozni, a `ThreadPool.QueueUserWorkItem()` metódust használhatjuk. Ez a metódus túlterhelt, így átadhatunk egy opcionális `System.Object` típust is az egyedi állapotadatok számára, a `WaitCallback` metódusreferencia egy példánya mellett:

```
public sealed class ThreadPool
{
  ...
  public static bool QueueUserWorkItem(WaitCallback callBack);
  public static bool QueueUserWorkItem(WaitCallback callBack,
    object state);
}
```

A `WaitCallback` metódusreferencia bármely olyan metódusra rá tud mutatni, amelynek egyedüli paramétere egy `System.Object` (amely az opcionális állapotadatot jelképezi), és eredményül nem ad vissza semmit. Ha nem biztosítunk egy `System.Object` típust a `QueueUserWorkItem()` meghívásakor, a CLR automatikusan `null` értéket ad át. A CLR-szálkészlet által használt metódu-

sok sorba állításának bemutatásához nézzük meg a következő programot, ez ismét a `Printer` típust használja. Ebben az esetben azonban nem manuálisan hozzuk létre a `Thread` típus tömbjét, hanem a készlet tagjai közül rendelünk hozzá a `PrintNumbers()` metódushoz:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with the CLR Thread Pool *****\n");

    Console.WriteLine("Main thread started. ThreadID = {0}",
      Thread.CurrentThread.ManagedThreadId);

    Printer p = new Printer();

    WaitCallback workItem = new WaitCallback(PrintTheNumbers);

    // A metódus sorba állítása tíz alkalommal.
    for (int i = 0; i < 10; i++)
    {
      ThreadPool.QueueUserWorkItem(workItem, p);
    }
    Console.WriteLine("All tasks queued");
    Console.ReadLine();
  }

  static void PrintTheNumbers(object state)
  {
    Printer task = (Printer)state;
    task.PrintNumbers();
  }
}
```

Vajon hasznos lenne-e a CLR által fenntartott szálkészletet használni inkább, mint explicit módon `Thread` objektumokat létrehozni? A szálkészlet felhasználásának előnyei a következők:

- A szálkészlet hatékonyan kezeli a szálakat, minimalizálva a létrehozandó, elindítandó és leállítandó szálak számát.
- A szálkészlet használatakor egyéb problémákra koncentrálhatunk, nem az alkalmazás szálinfrastruktúrájára.

Néhány esetben azonban jobb manuálisan kezelni a szálakat, például:

- Ha előtérbeli szálakra van szükségünk, vagy be kell állítanunk a szálprioritást. A készletben lévő szálak *mindig* háttérszálak, alapértelmezett prioritással (`ThreadPriority.Normal`).
- Ha fix azonosítóval rendelkező szálra van szükségünk azért, hogy név alapján szakítsuk meg, függesszük fel vagy vizsgáljuk meg.

Forráskód A `ThreadPoolApp` projekt a 18. fejezet alkönyvtárában található.

A BackgroundWorker komponens

Végül megvizsgáljuk a `BackgroundWorker` száltípust, amely az `mscorlib.dll` `System.ComponentModel` névterében van definiálva. A `BackgroundWorker` grafikus Windows Forms asztali alkalmazások készítésekor hasznos, vagy amikor hosszú futásidejű feladatokat kell végrehajtani (távoli webszolgáltatás meghívása, adatbázis-tranzakció végrehajtása, nagyméretű fájl letöltése stb.) az alkalmazás fő felhasználói felületét futtató száltól különböző szálon.

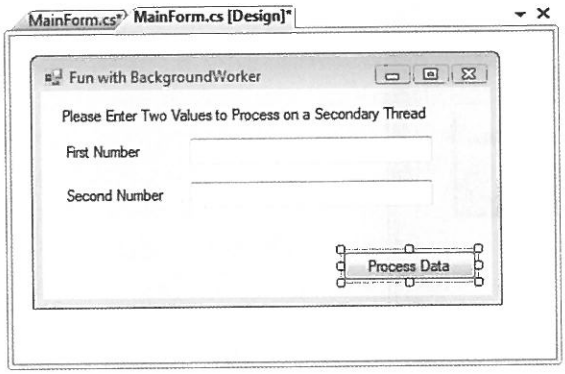
A `BackgroundWorkerrel` komolyabb nehézségek nélkül is tudunk többszálú grafikus alkalmazásokat készíteni. Ennek a típusnak a programozási modellje szinte ugyanazt a szálszintaxist használja fel, mint az aszinkron metódusreferenciáké, azaz a típus használatát könnyen meg lehet tanulni.

A `BackgroundWorker` használatához egyszerűen adjuk meg, hogy melyik metódust hajtsa végre a háttérben, és hívjuk meg a `RunWorkerAsync()` metódust. A hívószál (általában az elsődleges szál) továbbra is normál módon fut, amíg a feldolgozómetódus aszinkron módon. Amikor az időigényes metódus befejeződött, a `BackgroundWorker` típus értesíti a hívószálat a `RunWorkerCompleted` eseménnyel. A hozzárendelt eseménykezelő egy bejövő paramétert biztosít, ennek révén hozzáférhetünk a művelet eredményeihez (ha léteznek).

Megjegyzés A következő példa azt feltételezi, hogy bizonyos mértékben jártasak vagyunk a Windows Forms használatával végzett grafikus felület fejlesztésében. Ha ez nem így van, akkor nézzünk utána ennek a témának, mielőtt tovább haladnánk.

A BackgroundWorker típus használata

Ahhoz, hogy ennek a felhasználói felületet futtató szálösszetevőnek a használatát megértsük, hozzunk létre egy új, `WinFormsBackgroundWorkerThread` nevű Windows Forms-alkalmazást. Hogy az eddigi numerikus műveleti példáknál maradjunk, készítsünk egy egyszerű felhasználói felületet, amely lehetővé teszi, hogy a felhasználó két értéket adhasson be feldolgozásra (`TextBox` típusok révén), illetve egy `Button` típust a háttérművelet elindításához. Mindenképpen adjunk a felhasználói felület minden elemének egy megfelelő nevet a Properties ablak `Name` tulajdonságával. A 18.12. ábra egy lehetséges elrendezést mutat be.



18.12. ábra: *A Windows Forms felhasználóifelület- alkalmazás elrendezése*

Ezután készítsünk eseménykezelőt a `Button` típushoz: kattintsunk duplán a gombra az űrlaptervezőben:

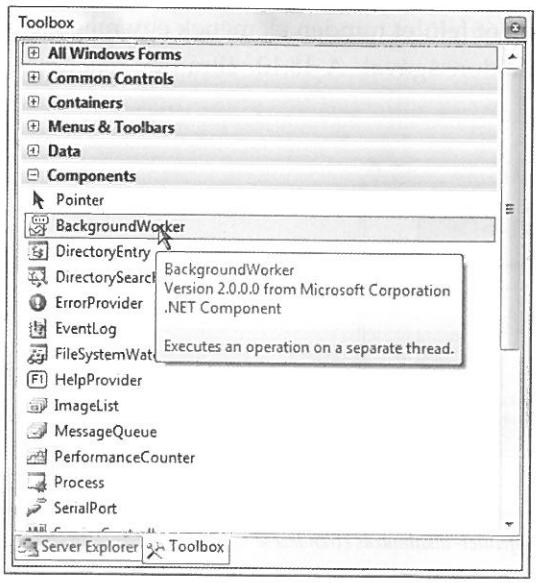
```
private void btnProcessData_Click(object sender, EventArgs e)
{
}
```

Nyissuk meg a Toolbox Components régióját, keressük meg a `BackgroundWorker` komponenst (lásd 18.13. ábra), és a típus egyik példányát húzzuk az űrlaptervezőbe.

Ekkor a típus egy változóját látjuk a tervező komponenstálcáján. Nevezzük át `ProcessNumbersBackgroundWorkerre` a komponenst a Properties ablak segítségével. Ezután váltsunk át a Properties ablak Event ablaktáblájára (a „villám" ikonra kattintva), és kattintsunk duplán a `DoWork` és a `RunWorkerCompleted` események nevére. Az eredmény az alábbi két új eseménykezelő lesz, amelyeket hozzáadtunk a kezdeti űrlapból származó típushoz:

```
private void ProcessNumbersBackgroundWorker_DoWork(object sender,
  DoWorkEventArgs e)
{
}

private void
  ProcessNumbersBackgroundWorker_RunWorkerCompleted(object sender,
  RunWorkerCompletedEventArgs e)
{
}
```



18.13. ábra: *A BackgroundWorker típus*

A `DoWork` eseménykezelő azt a metódust reprezentálja, amelyet a `BackgroundWorker` hív meg a másodlagos végrehajtási szálon. A kezelő második paramétere egy `DoWorkEventArgs` típus, amely minden olyan argumentumot tartalmaz, amelyre a másodlagos szálnak szüksége van a feladata elvégzéséhez. Amikor meghívjuk a `RunWorkerAsync()` metódust, hogy elindítsa ezt a szálat, lehetőségünk nyílik a kapcsolódó adatok átadására (hasonlóan, mint a fejezet korábbi részében alkalmazott `ParameterizedThreadStart` metódusreferencia-típus használatakor).

A `RunWorkerCompleted` esemény azt a metódust reprezentálja, amelyet a `BackgroundWorker` hív meg, mikor a háttérművelet befejeződött. A `RunWorkerCompletedEventArgs` típus használatával lehetőségünk nyílik az aszinkron művelet visszatérési értékeihez való hozzáférésre.

Adatfeldolgozás a BackgroundWorker típussal

Az alábbiakban megnézzük a felhasználói inputok feldolgozásának részleteit. Amikor értesíteni akartuk a `BackgroundWorker` típust, hogy indítsa el a másodlagos végrehajtási szálat, a `RunWorkerAsync()` metódust kellett meghívnunk. Ilyenkor lehetőségünk van egy `System.Object` típusú (azaz gyakorlatilag bármilyen) adat átadására, amely továbbadódik a `DoWork` által meghívott metódusnak. Használjuk fel újra az `AddParam` osztályt, amelyet a `ParameterizedThreadStart` példában hoztunk létre:

```
class AddParams
{
  public int a, b;

  public AddParams(int numb1, int numb2)
  {
    a = numb1;
    b = numb2;
  }
}
```

Ha ez a segédosztály a helyére került, a következőképpen valósíthatjuk meg a `Button` típus `Click` eseménykezelőjét:

```
private void btnProcessData_Click(object sender, EventArgs e)
{
  try
  {
    // Először kapjuk meg a felhasználói adatokat (számként).
    int numbOne = int.Parse(txtFirstNumber.Text);
    int numbTwo = int.Parse(this.txtSecondNumber.Text);
    AddParams args = new AddParams(numbOne, numbTwo);

    // Most indítsuk el az új szálat, és adjunk át
    // argumentumváltozókat.
    ProcessNumbersBackgroundWorker.RunWorkerAsync(args);
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Amint meghívjuk a `RunWorkerAsync()` metódust, bekövetkezik a `DoWork` esemény, amelyet az eseménykezelőnk elkap. Valósítsuk meg ezt a típust, hogy hozzáférjünk az `AddParam` objektumhoz, a bejövő `DoWorkEventArgs` Argument

tulajdonságának a segítségével. Egy hosszadalmas művelet szimulálásához az aktuális szálat mintegy öt másodpercre felfüggesztjük. Ezután visszaadjuk az értéket a `DoWorkEventArgs` típus `Result` tulajdonságának a segítségével:

```
private void ProcessNumbersBackgroundWorker_DoWork(object sender,
  DoWorkEventArgs e)
{
  // Szerezzük meg a bejövő AddParam objektumot.
  AddParams args = (AddParams)e.Argument;

  // Mesterséges késleltetés.
  System.Threading.Thread.Sleep(5000);

  // Adjuk vissza az értéket.
  e.Result = args.a + args.b;
}
```

Végül, ha a `BackgroundWorker` típus kilépett a `DoWork` kezelő hatóköréből, a `RunWorkerCompleted` esemény bekövetkezik. A regisztrált kezelőnk egyszerűen csak megjeleníti a művelet eredményét a `RunWorkerCompletedEventArgs.Result` tulajdonság használatával:

```
private void ProcessNumbersBackgroundWorker_RunWorkerCompleted(
  object sender, RunWorkerCompletedEventArgs e)
{
  MessageBox.Show(e.Result.ToString(), "Your result is");
}
```

Az alkalmazás futtatásakor azt tapasztaljuk, hogy az adatok feldolgozása közben a felhasználói felületet hosztoló szál még mindig teljesen válaszképes (pl. az ablak átméretezhető, áthelyezhető, kis méretűvé tehető stb.). Ennek további érzékeltetéséhez adjunk egy új `TextBox` típust az űrlaphoz, és ellenőrizzük, hogy tudunk-e adatokat bevinni a felhasználói felületen, mialatt az öt másodperces összeadási művelet aszinkron módon végbemegy a háttérben.

Forráskód A `WinFormsBackgroundWorkerThread` pojekt a 18. fejezet alkönyvtárában található.

A .NET-alapú többszálú programozás vizsgálatának a végére érve, azt még meg kell említenünk, hogy a `System.Threading` névtér természetesen jóval több típust definiál azon kívül, mint amennyiről a fejezetben szó volt.

Összefoglalás

Először azt vizsgáltuk meg, hogy hogyan konfiguráljuk a .NET-metódusreferencia típusait ahhoz, hogy aszinkron módon hajtsanak végre metódusokat. A `BeginInvoke()` és az `EndInvoke()` metódusok lehetővé teszik, hogy közvetett módon, komolyabb nehézségek nélkül kezeljük a háttérszálakat. Megismertük az `IAsyncResult` interfészt és az `AsyncResult` osztálytípust. Ezek a típusok a hívószál szinkronizálásának és a lehetséges függvényvisszatérési-értékek hozzáférésének számos módját biztosítják.

A továbbiakban a `System.Threading` névtér szerepét tanulmányoztuk. Amikor egy alkalmazás kiegészítő végrehajtási szálakat hoz létre, az eredmény az, hogy az adott program több feladat (látszólag) egyidejű elvégzésére is képes lesz. Megnéztük a szálérzékeny kódblokkok megvédésének számos módját is. Ezek révén biztosíthatjuk, hogy a megosztott erőforrások ne váljanak használhatatlan, hamis adatokká.

A fejezet arra is rámutatott, hogy a CLR fenntart egy belső szálkészletet a teljesítmény és a kényelem érdekében. Végül, de nem utolsósorban, a `BackgroundWorker` típus használatával ismerkedtünk meg, amelynek segítségével könnyedén indíthatunk be új végrehajtási szálakat egy GUI-alapú alkalmazásban.

TIZENKILENCEDIK FEJEZET

A CIL és a dinamikus szerelvények szerepe

A fejezet célja kettős. Először a korábbiaknál részletesebben megvizsgáljuk a köztes nyelv (CIL - Common Intermediate Language) szintaxisát és szemantikáját. .NET-programozóként anélkül is boldogulhatunk, hogy ismernénk a CIL-kód részleteit. Ám ha megtanuljuk a CIL alapjait, sokkal jobban meg fogjuk érteni a .NET néhány „bűvös" vonatkozását (pl. a nyelvközi származtatást).

A továbbiakban megvizsgáljuk a `System.Reflection.Emit` névtér szerepét. Ezekkel a típusokkal olyan programokat írhatunk, amelyek a .NET-szerelvényeket futásidőben hozzák létre a memóriában. A memóriában definiált és futtatott szerelvényeket *dinamikus szerelvényeknek* nevezzük. A .NET-fejlesztésnek ez a szemlélete teszi szükségessé a CIL nyelv használatát, hiszen pontosan meg kell határoznunk a szerelvények szerkesztéséhez alkalmazott CIL-utasításkészletet.

A CIL-programozás természete

A .NET platform igazi „anyanyelve" a CIL. Ha egy általunk választott nyelven (C#, VB, COBOL.NET stb.) készítünk .NET-szerelvényt, az alkalmazott fordító a forráskódunkat CIL-kifejezéssé alakítja. Mint minden más porgramozói nyelv, a CIL is számos strukturális és megvalósításcentrikus tokennel rendelkezik. Minthogy a CIL .NET programozási nyelv, így a .NET-szerelvényeinket létrehozhatjuk közvetlenül a CIL és a CIL-fordító (`ilasm.exe`) használatával, ezek a .NET keretrendszer 3.5 SDK-val együtt járnak.

Bár kevés programozó készíti el a teljes .NET-alkalmazását közvetlenül a CIL-lel, ám a CIL mégis rendkívül érdekes, intellektuális tevékenységet kínál. Minél jobban megértjük a CIL nyelvtanát, annál inkább tudunk érvényesülni a haladó .NET-fejlesztés világában. Mire is lehet képes egy CIL nyelvet elsajátított programozó? Nézzünk néhány példát:

- Ismeri azt, hogy hogyan képződnek le a különböző .NET-programnyelvek megfelelő kulcsszavai CIL-tokenekké.
- Tisztában van a meglévő .NET-szerelvények lebontásával, a CIL-kód szerkesztésével, valamint a módosított kód újrafordításával .NET-binárissá.
- A `Sysetm.Reflection.Emit` névtér segítségével dinamikus szerelvényeket tud létrehozni.
- A CTS (egységes típusrendszer) előnyeit ki tudja használni. A CIL szintjén a CTS olyan aspektusai is elérhetővé válnak, amelyeket egy-egy magasabb szintű nyelv nem támogat. A CIL az egyetlen olyan .NET-nyelv, amellyel a CTS valamennyi aspektusa kezelhető. A nyers CIL használatával például globális szintű tagokat és mezőket tudunk definiálni (ez a C#-ban nem lehetséges).

Ha *nem* foglalkozunk a CIL-kód részleteivel, akkor is teljesen elsajátíthatjuk a C# és a .NET alaposztálykönyvtáraival kapcsolatos ismereteket. Voltaképpen a CIL ismerete azzal hozható párhuzamba, ahogyan a C(++)-programozó ismeri az assembly nyelvet. Azok, akik értik ennek az alacsony szintű „ingoványnak" minden részletét, képesek a meglévő feladathoz magasabb szintű megoldást létrehozni, valamint a mögöttes programozó (és futásidejű) környezetet mélyebben is megérteni. Így tehát vizsgáljuk meg részletesebben is a CIL nyelvet.

Megjegyzés A fejezet nem ismerteti a CIL szintaxisának és szemantikájának átfogó kezelését. Ha ennél még részletesebb tudásanyagra lenne szükségünk, vegyük kézbe Jason Bock *CIL Programming: Under the Hood of .NET* (Apress, 2002) című könyvét.

CIL-direktívák, -attribútumok és -vezérlőkódok

Ha egy alacsony szintű nyelvet kezdünk vizsgálni, akkor nyilván találunk új elnevezéseket a teljesen megszokott fogalmakra. Ha például a következő szókészletet mutatnák nekünk:

```
{new, public, this, base, get, set, explicit, unsafe, enum,
 operator, partial}
```

akkor bizonyára arra gondolnánk, hogy a C# nyelv kulcsszavaival van dolgunk. Ha azonban jobban megnézzük a készlet elemeit, észrevesszük, hogy mindegyik C#-kulcsszó ugyan, de a szemantikájuk egészen eltérő. Az `enum` kulcsszó például egy `System.Enum` származtatott típust definiál, míg a `this` és a `base` kulcsszavak az aktuális objektumra vagy az objektum szülőosztályára történő hivatkozást teszik lehetővé. Az `unsafe` kulcsszó segítségével olyan kódblokkot hozhatunk létre, amelyet nem lehet közvetlenül ellenőrizni a CLR-en keresztül, az `operator` kulcsszó pedig lehetővé teszi rejtett (speciálisan elnevezett) metódus létrehozását, amelyet akkor hívunk meg, amikor egy sajátságos C#-operátort használunk (pl. a plusz jel).

Ellentétben a magasabb szintű nyelvekkel (pl. C#) a CIL nem csupán meghatározza a generikus szókészletet *önmagában*, hanem a CIL-fordítóprogram három tágabb kategóriába sorolja szemantikájuk alapján a tokenkészletet:

- CIL-direktívák,
- CIL-attribútumok,
- CIL-vezérlőkódok (opcodes).

Minden CIL-tokennek egyéni szintaxisa van, és ezekből a tokenekből épülnek fel a .NET-szerelvények.

A CIL-direktívák szerepe

A tokenek első nagy csoportja a .NET-szerelvények felépítésének leírására szolgálnak. A CIL-direktívák segítségével tájékoztatjuk a CIL-fordítót arról, hogy hogyan definiálja azokat a névtere(ke)t, típus(oka)t és tago(ka)t, amelyek egy szerelvényt feltöltenek.

A direktívákat szintaktikailag egy pont (.) prefixummal (pl. `.namespace`, `.class`, `.publickeytoken`, `.method`, `.assembly` stb.) jelképezzük. Így, ha a `*.il` fájl (a köztes nyelvi kódot tartalmazó állományok hagyományos kiterjesztése) egyetlen `.namespace` direktívából és három `.class` direktívából áll, akkor a CIL-fordító olyan szerelvényt generál, amely három .NET-osztálytípust tartalmazó .NET-névteret definiál.

A CIL-attribútumok szerepe

Számos esetben a CIL-direktívák egy adott .NET-típus vagy -típustag definíciójának teljes kifejezéséhez önmagukban kevésnek bizonyulnak. Ezért sok CIL-direktívát tovább kell specifikálni különféle CIL-*attribútumokkal*, meghatározva ezzel a direktíva feldolgozását. A `.class` direktívát elláthatjuk `public` attribútummal (a láthatóság típusát határozva meg), `extends` attribútummal (explicit módon definiálva a típus ősosztályát) és `implements` attribútummal (a típus által támogatott interfészkészlet felsorolásához).

A CIL-vezérlőkódok szerepe

Miután különböző direktívák és kapcsolt attribútumok segítségével a CIL nyelvben meghatároztuk a .NET-szerelvényt, -névteret és -típuskészletet, utolsó feladatunk a típus megvalósítási logikájának megadása. Ez pedig az *vezérlőkódok* (röviden *opcode*) feladata. Más alacsony szintű nyelvek után sok CIL-opcode esetleg rejtélyesnek és kimondhatatlannak tűnhet. Ha például egy sztringváltozót szeretnénk meghatározni, nem egy barátságos `LoadString` vezérlőkódot használunk, hanem az `ldstr` elnevezésűt.

Néhány CIL-opcode-ot egészen természetes módon leképezhetünk C#-megfelelőjükből (pl. `box`, `unbox`, `throw` és `sizeof`). A CIL vezérlőkódjait mindig egy tag implementációjának hatókörén belül használjuk, és a CIL-direktívákkal ellentétben sohasem pont prefixummal.

A CIL-vezérlőkódok/CIL-mnemonikok megkülönböztetése

Az operátorkódok, például az `ldstr`, egy bizonyos típus tagjainak implementálására szolgálnak. A valóságban azonban az `ldstr` és az ehhez hasonló tokenek az aktuális *bináris CIL-vezérlőkódok, CIL-mnemonik*. A különbség tisztázásához tegyük fel, hogy a következő metódust hoztuk létre a C# nyelvben:

```
static int Add(int x, int y)
{
  return x + y;
}
```

Két szám összeadása a 0X58 CIL-opcode segítségével történik. Ehhez hasonlóan, két szám különbségét a 0X59 opcode határozza meg, míg új objektum létrehozásához a felügyelt heapben a 0X73 vezérlőkódot használjuk. A JIT fordító által feldolgozott „köztes nyelvi kód" („CIL code") valójában nem más, mint bináris adatcsoport.

Minden CIL kétoperandusú vezérlőkódhoz tartozik egy megfelelő mnemoni. A `0x58` helyett például az `add`, a `0x59` helyett a `sub`, a `0x73` helyett pedig a `newobj` emlékeztető használható. Az opcode/emlékeztető különbségeiről szólva: a CIL-decompilerek, például az `ildasm.exe`, a szerelvény kétoperandusú vezérlőkódjait a megfelelő CIL-mnemoni formájába teszik. Az alábbiakban például a C# `Add()` metódust az `ildasm.exe` CIL nyelven jeleníti meg:

```
.method private hidebysig static int32 Add(int3
int32 y) cil managed
{
  // Kódméret 9 (0x9)
  .maxstack 2
  .locals init ([0] int32 CS$1$0000)
  IL_0000: nop
  IL_0001: ldarg.0
  IL_0002: ldarg.1
  IL_0003: add
  IL_0004: stloc.0
  IL_0005: br.s IL_0007
  IL_0007: ldloc.0
  IL_0008: ret
} // a MathStuff:Add metódus vége
```

Ha csak nem rendkívül alacsony szintű .NET-szoftvert készítünk (pl. egyedi vezérlésű fordító), akkor egyáltalán nem kell törődnünk a CIL numerikus kétoperandusú vezérlőkódokkal. A .NET-programozók sem a „CIL-vezérlőkódok" mögöttes numerikus értékéről beszélnek, hanem a barátságos sztringtoken-mnemonik készletre hivatkoznak (ahogy tesszük ebben a könyvben is).

Verem írása/olvasása: A CIL veremalapú természete

A magasabb szintű .NET-nyelvek (pl. a C#), amennyire csak lehet, megpróbálják háttérben tartani az alacsony szintű CIL nyelvet. A .NET-fejlesztésben elhallgatják, hogy a CIL veremalapú programozási nyelv. A névtérgyűjteményekről (lásd 10. fejezet) már tudhatjuk, hogy a `System.Collections.Stack` tí-

pust használhatjuk arra, hogy egy értéket a verembe írjunk, vagy a legfelső értéket a veremből kiemeljük. A CIL-fejlesztők persze nem szó szerint az értékek fel- és letöltésére használják a `System.Collections.Stack` típus egy objektumát; ám az ugyanilyen írási/olvasási berögződés létezik.

Szaknyelven az az egység, amely a kiértékelendő értékkészletet tárolja, a *virtuális végrehajtási verem*. Számos olyan CIL-vezérlőkód van, amely értéket tesz a verembe; ez a folyamat a *betöltés (loading)*. A CIL továbbá számos vezérlőkóddal rendelkezik, amelyek a verem legfelső elemét a memóriába szállítják (pl. egy lokális változót); a végrehajtott folyamat a *tárolás (storing)*.

A CIL világában nem lehet közvetlenül adatot elérni, beleértve a lokális változót, a bementi metódusargumentumokat vagy egy típus adatmezőjét. Explicit módon kell betöltenünk az elemet a verembe, csak ezután van mód kiolvasni a későbbi használathoz (ebből érthetővé válik, miért tűnik egy adott CIL-kódblokk kissé redundánsnak).

Megjegyzés A CIL végrehajtása nem közvetlenül történik, hanem szükség esetén lefordul. A CIL-kód fordításakor számos implementációredundancia optimalizálódik. Ha az aktuális projektünk számára engedélyezzük a kódoptimalizálási opciót (a Visual Studio Project Properties ablak Build lapján), a fordító is eltávolít néhány CIL-redundanciát.

Ahhoz, hogy megértsük, a CIL miképp használja ki a veremalapú feldolgozásmodellt, nézzünk meg egy egyszerű C#-metódust (`PrintMessage()`), amely nem kap argumentumokat, és nincs visszatérési értéke. A metódus a kimeneti folyamat egy egyszerű helyi sztringváltozó-értékét írja ki a konzolra:

```
public void PrintMessage()
{
  string myMessage = "Hello.";
  Console.WriteLine(myMessage);
}
```

A metódus fordítását részeletesebben megvizsgálva azt látjuk, hogy a `PrintMessage()` a `.locals` direktíva segítségével egy tárolót definiál a lokális változó számára. A lokális sztring betöltődik, és eltárolódik a helyi változóban – felhasználva az `ldstr` (sztring betöltése) és a `stloc.0` vezérlőkódokat (ezt úgy is olvashatnánk, hogy „tároljuk el az aktuális értéket egy helyi változóban nulla indexszel").

Az érték (ismét 0 indexen) az `ldloc.0` („helyi argumentum betöltése 0 indexszel") vezérlőkód segítségével és a `System.Console.WriteLine()` metódushívással (`call` vezérlőkóddal megadva) betöltődik a memóriába. A függvény végül visszatér a `ret` vezérlőkóddal. Az alábbiakban a `PringMessage()` metódus (magyarázattal ellátott) CIL-kódját látjuk:

```
.method public hidebysig instance void PrintMessage() cil managed
{
  .maxstack 1
  // Egy lokális sztring változót definiál (0 indexszel).
  .locals init ([0] string myMessage)
  //Beolvasunk egy stringet a verembe "Hello" értékkel.
  ldstr " Hello."
  // A string értékét a veremben egy lokális változóban tároljuk.
  stloc.0
  // Beolvassuk az értéket 0 indexszel.
  ldloc.0
  // Meghívjuk a metódust az aktuális értékkel.
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Megjegyzés A CIL támogatja a kódmegjegyzéseket dupla perjeles szintaxissal (egyébként a `/*...*/` szintaxis használatával is). A C#-hoz hasonlóan a kódmegjegyzéseket a CIL-fordító teljesen figyelmen kívül hagyja.

A következőkben nézzük meg a CIL-programozás gyakorlatát, a „round-trip engineering" témájával kezdve.

A round-trip engineering

Azt már láttuk, hogyan nézzük meg a C#-fordító által generált CIL-kódot az `ildasm.exe` segítségével (lásd 1. fejezet). Arról azonban még nem volt szó, hogy az `ildasm.exe` lehetővé teszi, hogy egy betöltött szerelvényben található CIL-t kiírjunk egy külső fájlba. Ha rendelkezésünkre áll a CIL-kód, szabadon szerkeszthetjük és újrafordíthatjuk a CIL-fordító, az `ilasm.exe` használatával.

Megjegyzés A `reflector.exe` segítségével is megnézhetjük egy adott szerelvény CIL-kódját, valamint a CIL-kódot lefordíthatjuk egy megközelítő C#-kódra. Ha azonban egy szerelvény olyan CIL-konstruktort tartalmaz, amelynek nincs megfelelője a C#-ban, le kell mondanunk az `ildasm.exe` használatáról.

Szaknyelven ezt a technikát *round-trip engineeringnek* hívják, és sok helyzetben hasznos lehet:

- Szükségünk lehet olyan szerelvény módosítására, amelynek forráskódjával már nem rendelkezünk.

- Olyan, kevésbé tökéletes .NET nyelvi fordítóval dolgozunk, amely nem hatékony (vagy nyilvánvalóan helytelen) CIL-kódot hoz létre, és szeretnénk módosítani a kódalapot.
- COM együttműködő szerelvényeket hozunk létre, és szeretnénk megtalálni néhány IDL-attribútumot, amelyek a konvertálási folyamat közben eltűntek (mint pl. a COM `[helpstring]` attribútum).

A round-tripping folyamatának bemutatását kezdjük egy új C#-kódfájl létrehozásával (`HelloProgram.cs`) egy egyszerű szövegszerkesztő segítségével, és definiáljuk a következő osztálytípusokat (akár a Visual Studio 2008 parancssori alkalmazás projektjét is használhatjuk; de mindenképpen töröljük az `AssemblyInfo.cs` fájlt, hogy csökkentsük a generált CIL-kódok mennyiségét):

```
// Egy egyszerű C# parancssori alkalmazás
using System;

class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("Hello CIL code!")
    Console.ReadLine();
  }
}
```

Megfelelő helyre mentsük a fájlt (például C:\HelloCilCode), és fordítsuk a programot a `csc.exe` használatával:

```
csc HelloProgram.cs
```

Nyissuk meg a `HelloProgram.exe` fájlt az `ildasm.exe` segítségével, és a File > Dump menüpontot használva mentsük a nyers CIL-kódot egy új könyvtárba, amelyik a lefordított szerelvényünket tartalmazza (a megjelenő párbeszédablak minden más alapértelmezett értéke megfelelő lesz).

Megjegyzés Az `ildasm.exe` egy `.res` fájlt is létrehoz, amikor egy szerelvény tartalmát fájlba kiírjuk. Ezeket az erőforrásfájlokat ebben a fejezetben mellőzhetjük (és törölhetjük), ugyanis nem fogjuk használni őket.

Megnézhetjük a fájlt egy tetszőleges szövegszerkesztővel. Az eredmény (némileg átformázva és megjegyzésekkel ellátva) a következő:

```
// Hivatkozott szerelvények.
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}

// A mi szerelvényünk.
.assembly HelloProgram
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module HelloProgram.exe
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003
.corflags 0x00000001

// A Program osztály definiálása.
.class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
{
  .method private hidebysig static void Main(string[] args)
    cil managed
  {

  // A végrahajtható fájl belépési pontjának megjelölése.
    .entrypoint
    .maxstack 8
    IL_0000: nop
    IL_0001: ldstr "Hello CIL code!"
    IL_0006: call void [mscorlib]System.Console::WriteLine(string)
    IL_000b: nop
    IL_000c: call string [mscorlib]System.Console::ReadLine()
    IL_0011: pop
    IL_0012: ret
  }

  //Az alapértelmezett konstruktor.
  .method public hidebysig specialname rtspecialname
    instance void .ctor() cil managed
  {
    .maxstack 8
    IL_0000: ldarg.0
    IL_0001: call instance void [mscorlib]System.Object::.ctor()
    IL_0006: ret
  }
}
```

A `*.il` file először minden olyan külső szerelvényt deklarál, amelyekre referenciát adtunk az aktuális szerelvényünk fordításakor. Ebben a példában egyetlen ilyen külső szerelvényre történő deklarálás látható: az `.assembly extern` token segítségével hivatkozunk a mindig jelen lévő `mscorlib.ddl`-re. Természetesen ha az osztálykönyvtárunk más, hivatkozott szerelvényeken belüli típusokat használ, további `.assembly extern` külső direktívákat találunk.

Ezután a `HelloProgram.exe` szerelvényünk formális definícióját látjuk, amely az alapértelmezett 0.0.0.0 verziószámot kapta (ugyanis nem adtunk meg értéket az `[AssemblyVersion]` attribútummal). A szerelvényt továbbá különböző CIL-direktívák írják le (pl. a `.module`, `.imagebase` stb.).

A külsőleg hivatkozott szerelvények dokumentálása és az aktuális szerelvény definiálása után találjuk a `Program` típus definícióját. A `.class` direktíva különböző attribútumokkal rendelkezik (amelyek tulajdonképpen opcionálisak), mint például az `extends`, amely a típus ősosztályát jelöli:

```
.class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
{ ... }
```

A fenti CIL-kód nagyobb része az osztályok alapértelmezett konstruktorának és a `Main()` metódusnak az implementációját mutatja be, mindkettőt (részben) a `.method` direktíva definiálja. A tagokat a megfelelő direktívák és attribútumok segítségével definiáltuk, majd különböző vezérlőkódok használatával implementáltuk.

Amikor a köztes nyelvben .NET-típusokkal (pl. a `System.Console`) kommunikálunk, *mindig* szükség van a típus teljesen meghatározott nevének a használatára. Továbbá a típus teljesen meghatározott nevének *mindig* a definiáló szerelvény barátságos nevével kell kezdődnie (szögletes zárójelben). Nézzük meg a `Main()` köztes nyelvi megvalósítását:

```
.method private hidebysig static void Main(string[] args)
  cil managed
{
  .entrypoint
  .maxstack 8
  IL_0000: nop
  IL_0001: ldstr "Hello CIL code!"
  IL_0006: call void [mscorlib]System.Console::WriteLine(string)
  IL_000b: nop
  IL_000c: call string [mscorlib]System.Console::ReadLine()
  IL_0011: pop
  IL_0012: ret
}
```

Az alapértelmezett konstruktor implementációja a CIL-kódban még más „betöltéscentrikus" instrukciót (`ldarg.0`) is használ. Ebben az esetben a verembe beolvasott érték nem egy általunk meghatározott egyedi változó, hanem az aktuális objektumreferencia (erről később részletesebben lesz szó). Az alapértelmezett konstruktor explicit módon hívja az ősosztály konstruktorát:

```
.method public hidebysig specialname rtspecialname
  instance void .ctor() cil managed
{
  .maxstack 8
  IL_0000: ldarg.0
  IL_0001: call instance void [mscorlib]System.Object::.ctor()
  IL_0006: ret
}
```

A CIL-kódcímkék szerepe

Az implementáció kódjának minden sora az `IL_xxx:` formájú tokennel kezdődik (pl. `IL_0000:`, `IL_0001:` és így tovább). Ezek a tokenek a *kódcímkék*, és bármilyen módon elnevezhetjük őket (de ugyanabban a hatókörben egy adott névből csak egy darab lehet). Amikor egy szerelvényt kiírunk fájlba az `ildasm.exe` használatával, automatikusan `IL_xxx:` elnevezési szabályt követő kódcímkéket állítunk elő. De kicserélhetjük őket sokkal többet mondó jelzőkre:

```
.method private hidebysig static void Main(string[] args)
  cil managed
{
  .entrypoint
  .maxstack 8
  Nothing_1: nop
  Load_String: ldstr "Hello CIL code!"
  PrintToConsole: call void
    [mscorlib]System.Console::WriteLine(string)
  Nothing_2: nop
  WaitFor_KeyPress: call string [mscorlib]System.Console::ReadLine()
  RemoveValueFromStack: pop
  Leave_Function: ret
}
```

A kódcímkék használata legtöbb esetben opcionális. Gyakorlatilag csak akkor van szükség rájuk, ha elágazásokat és ciklusokat tartalmazó CIL-kódot írunk, hiszen ekkor csak a kódcímkék segítségével tudjuk előírni, hogy hol folytatódjon a végrehajtási logika. Aktuális példánkban ezeket az automatikusan generált címkéket probléma és következmény nélkül mind eltávolíthatjuk:

```
.method private hidebysig static void Main(string[] args)
  cil managed
{
  .entrypoint
  .maxstack 8
  nop
  ldstr "Hello CIL code!"
  call void [mscorlib]System.Console::WriteLine(string)
  nop
  call string [mscorlib]System.Console::ReadLine()
  pop
  ret
}
```

Fejlesztés a CIL-lel: egy *.il fájl módosítása

A következőkben tegyük teljessé round-tripping ismereteinket. A cél az, hogy az alábbiak szerint módosítsuk a köztes nyelvi kódot egy létező `*.il` fájlban:

- Adjunk referenciát a `System.Windows.Forms.dll` szerelvényre.
- Töltsünk be egy lokális sztringet a `Main()` metóduson belül.
- Hívjuk meg a `System.Windows.Forms.MessageBox.Show()` metódust úgy, hogy a lokális sztringváltozót argumentumként használjuk.

Az első lépés egy új `.assembly` direktíva hozzáadása (`extern` attribútummal minősítve). Első lépésként hivatkozzunk a `System.Windows.Forms.dll` szerelvényre: ehhez egy `.assembly (extern)` direktívával bővítjük a `*.il` fájl tartalmát:

```
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89)
  .ver 2:0:0:0
}
```

A `.ver` direktívához rendelt érték eltérő lehet attól függően, hogy a .NET platform melyik verzióját használjuk. Itt a `System.Windows.Forms.dll` 2.0.0.0 verziója látható, amelynek nyilvánoskulcs-tokenje B77A5C561934E089. Ha megnyitjuk a GAC-ot (lásd a 15. fejezetben), és megkeressük a `System.Windows.Forms.dll` szerelvény verzióját, egyszerűen átmásolhatjuk a megfelelő verzió és a nyilvánoskulcs-token értékét a szerelvény Properties lapjáról (ezt az egér jobb gombjával érhetjük el).

Ezután meg kell változtatnunk a `Main()` metódus aktuális implementációját. Keressük meg ezt a metódust a `*.il` fájlban, és távolítsuk el az aktuális implementáció kódját (a `.maxstack` és a `.entrypoint` direktívák érintetlenek maradhatnak; ezekre később visszatérünk):

```
.method private hidebysig static void Main(string[] args)
  cil managed
{
  .entrypoint
  .maxstack 8
  // TODO: Új CIL-kódot írni!
}
```

A cél az, hogy egy új sztringet írjunk a verembe, és a `MessageBox.Show()` metódust hívjuk (a `Consol.WriteLine()` metódus helyett). Amikor egy külső típus nevét megadjuk, a típus teljesen meghatározott nevét kell használnunk (a szerelvény barátságos nevével együtt). Ennek figyelembevételével módosítsuk a `Main()` metódust a következők szerint:

```
.method private hidebysig static void Main(string[] args)
  cil managed
{
  .entrypoint
  .maxstack 8
  ldstr "CIL is way cool"
  call valuetype [System.Windows.Forms]
    System.Windows.Forms.DialogResult
    [System.Windows.Forms]
    System.Windows.Forms.MessageBox::Show(string)
  pop
  ret
}
```

Valójában csak módosítottuk a CIL-kódot úgy, hogy megfeleljen a következő C#-osztály definíciójának:

```
class Program
{
  static void Main(string[] args)
  {
    System.Windows.Forms.MessageBox.Show("CIL is way cool");
  }
}
```

A CIL-kód fordítása ilasm.exe használatával

Ha elmentettük a módosított `*.il` fájlt, fordíthatjuk az új .NET-szerelvényt az `ilasm.exe` segédprogrammal. A CIL-fordító számos parancssori paraméterrel rendelkezik (az összeset a `-?` paraméter megadásával nézhetjük meg), a 19.1 táblázat a leglényegesebb kapcsolókat mutatja:

Kapcsoló	Valós jelentése
`/debug`	Hibakeresési információkat tartalmaz (pl. lokális változó és argumentumnév, valamint az eredeti forráskód sorszámát).
`/dll`	Kimenetként egy `*.dll` fájlt hoz létre.
`/exe`	Kimenetként egy `*.exe` fájlt hoz létre. Ez az alapértelmezett beállítás, és mellőzhető.
`/key`	Erős névvel fordítja a szerelvényt egy adott `*.snk` fájl használatával.
`/noautoinherit`	Megvédi az osztálytípusokat attól, hogy automatikusan a `System.Object` könyvtártól örököljenek, amikor nem definiálunk speciális ősosztályt.
`/output`	Meghatározza a kimeneti fájl nevét és kiterjesztését. Ha nem használjuk az /output kapcsolót, az eredményül kapott fájlnév (leszámítva a fájl kiterjesztését) azonos lesz az első forrás fájlnevével.

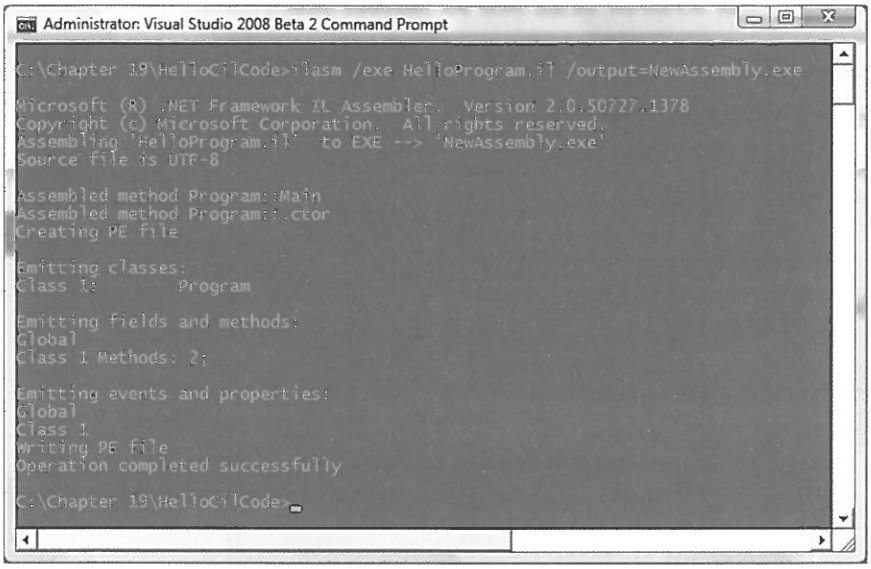
19.1. táblázat: *Gyakori ilasm.exe parancsori kapcsolók*

Ahhoz, hogy a módosított `HelloProgram.il` fájlunkat egy új .NET `*.exe` fájlba fordítsuk, kiadhatjuk a következő parancsot a Visual Studio 2008 parancssorában:

```
ilasm /exe HelloProgram.il /output=NewAssembly.exe
```

Ha minden sikeresen működött, a 19.1 ábrának megfelelő kimenetet látunk.

Ezután már futtathatjuk az új alkalmazásunkat. Nagy valószínűséggel a parancssori ablak helyett egy üzenetdobozban fogjuk látni az üzenetünket (lásd a 19.2 ábrát).



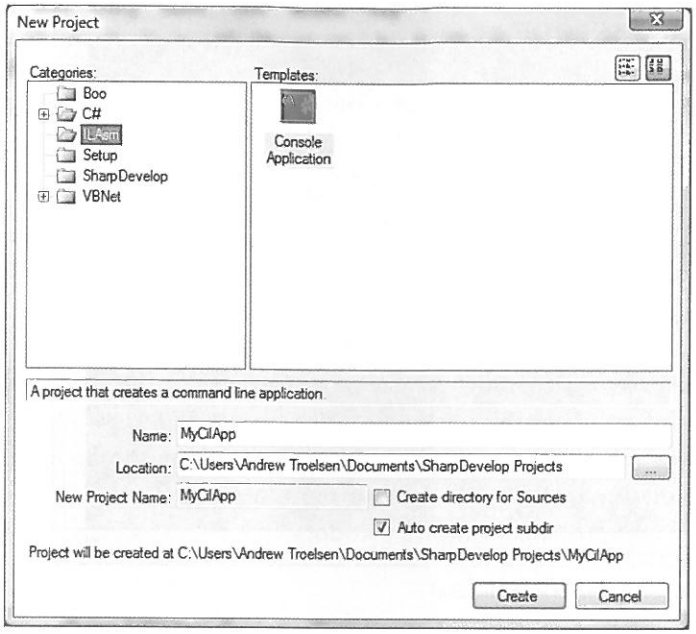
19.1. ábra: *A *.il fájl fordítása ilasm.exe használatával*

19.2. ábra: *A round-trip eredménye*

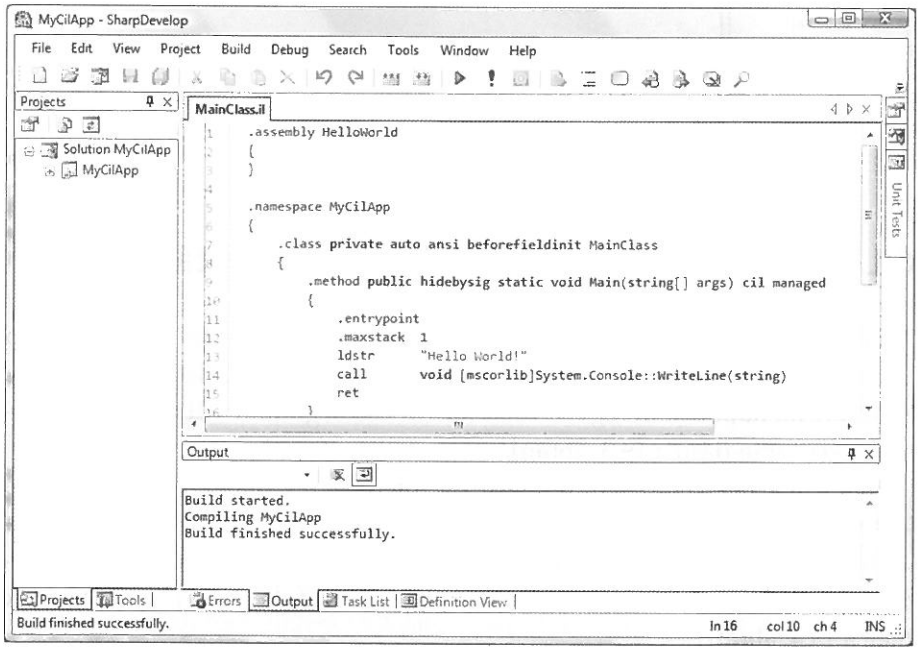
CIL-kód fordítása SharpDevelop használatával

Amikor `*.il` fájlokkal dolgozunk, használhatjuk az ingyenes SharpDevelop IDE-t (lásd a 2. fejezetet). Amikor egy új `Solutiont` hozunk létre (File > New Solution menüpont), az egyik választási lehetőségünk a CIL projekt workspace készítése (lásd a 19.3. ábrán).

Bár a SharpDevelop nem rendelkezik IntelliSense-támogatással a CIL-projekthez, a CIL-tokenek színezve jelennek meg a fejlesztőeszközben, és közvetlenül az integrált fejlesztői környezetben van lehetőségünk fordítani és futtatni az alkalmazásunkat (az `ilasm.exe` parancssori futtatása helyett). Nézzük meg a 19.4. ábrát.



19.3. ábra: *A SharpDevelop CIL-projekt sablonja*



19.4. ábra: *A CIL-kód készítése SharpDevelopban*

A peverify.exe szerepe

Ha CIL-kód írásával hozunk létre vagy módosítunk szerelvényeket, tanácsos a `peverify.exe` segítségével ellenőrizni, hogy a lefordított bináris megfelelő-e:

```
peverify MyNewAssembly.exe
```

Ez az eszköz minden vezérlőkódot átnéz a megadott szerelvényen belül a CIL-kód érvényességi vizsgálatához. A CIL-kódban például a kiértékelési veremnek mindig üresnek kell lennie egy metódusból való kilépés előtt. Ha elfelejtjük eltávolítani az összes maradék értéket, az `ilasm.exe` fordító még létrehozza az érvényes szerelvényt (hiszen ez a fordító csak a *szintaxissal* foglalkozik), a `peverify.exe` azonban a *szemantikával* is. Ha elfelejtjük törölni a verem elemeit egy adott funkcióból való kilépés előtt, a `peverify.exe` figyelmeztetni fog, amikor megpróbáljuk futtatni a kódot.

Forráskód A `HelloCilCode` példa megtalálható a 19. fejezet alkönyvtárában.

CIL-direktívák és -attribútumok

A következőkben vizsgáljuk meg a CIL szintaxisát és szemantikáját magát. Végigkísérjük egy típusokat tartalmazó egyedi névtér készítésének a folyamatát. Az egyszerűség kedvéért típusaink tagjai nem fognak implementációt tartalmazni. Ezután a „valós" tagok CIL-vezérlőkódokkal történő megvalósítási folyamatára összpontosítunk.

Külsőleg hivatkozott szerelvények meghatározása CIL-ben

Hozzunk létre egy új fájlt `CilTypes.il` névvel egy szabadon választott szövegszerkesztő használatával. Egy CIL-projektnek elsőként az aktuális szerelvény külső szerelvényeinek felsorolására van szüksége. Ebben a példában csak az `mscorlib.dll` könyvtárban található típusokat fogjuk használni. Ehhez az `.assembly` direktívát `extern` attribútum használatával minősítjük. Amikor egy erős névvel ellátott szerelvényre hivatkozunk, mint az `mscorlib.dll`, szeretnénk a `.publickeytoken` és a `.ver` direktívákat is meghatározni:

```
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
```

Megjegyzés Nincs szükség arra, hogy külső referenciaként explicit módon hivatkozzunk az `mscorlib.dll` könyvtárra, mert az ilasm.exe ezt automatikusan megteszi.

Az aktuális szerelvény definiálása CIL-ben

A következő teendő a `.assembly` direktíva használatával létrehozni kívánt szerelvény definiálása. Egy szerelvényt legegyszerűbben a bináris barátságos nevének meghatározásával definiálhatunk.

```
// A mi szerelvényünk
.assembly CILTypes { }
```

Míg ez egy új .NET-szerelvényt definiál, általában további direktívákat helyezünk el a szerelvény deklarációjának hatókörében. Ebben a példában a `.ver` direktívával módosítjuk a szerelvényünk definícióját, hogy tartalmazza az 1.0.0.0 verziószámot (minden számot *kettősponttal* választunk el, nem a C#-féle pont jelölésével).

```
// A mi szerelvényünk
.assembly CILTypes
{
  .ver 1:0:0:0
}
```

Mivel a `CILTypes` szerelvény egyfájlos, a szerelvény definícióját egy egyszerű `.module` direktíva használatával fogjuk befejezni, amely a .NET-binárisunk hivatalos nevét jelöli meg.

```
.assembly CILTypes
{
  .ver 1:0:0:0
}
// Egyfájlos szerelvényünk modulja
.module CILTypes.dll
```

A `.assembly` és `.module` olyan CIL-direktívák, amelyek tovább módosítják a létrehozott .NET-binárisok teljes struktúráját. A 19.2. táblázat a leggyakoribb szerelvényszintű direktívákat sorolja fel.

Direktíva	Valós jelentése
`.mresources`	Ha a szerelvényünk beágyazott erőforrásokat használ (mint a képek vagy a sztringtáblázatok), akkor ennek a direktívának a használatával azonosítani tudjuk a beágyazandó erőforrásokat tartalmazó fájl nevét.
`.subsystem`	Ezt a CIL-direktívát a preferált felhasználói felület létrehozására használjuk, amelyben a szerelvényt szeretnénk működtetni. A 2-es érték például azt jelzi, hogy a szerelvény űrlapalapú grafikus felület alatt fut, míg egy 3-as érték parancssori végrehajtást jelez.

19.2. táblázat: *További szerelvénycentrikus direktívák*

Névtér definiálása CIL-ben

A szerelvényünk megjelenésének és nagy vonalakban történő definiálásának (valamint a szükséges külső referenciáknak) a definiálása után létrehozhatunk egy .NET-névteret (`MyNamespace`) a `.namespace` direktíva segítségével.

```
// Szerelvényünknek egy egyszerű névtere van
.namespace MyNamespace {}
```

A C#-hoz hasonlóan a CIL névtérdefiníciói további névterekbe ágyazhatók. Nincs szükség gyökérnévtér definiálására; a példa kedvéért mégis hozzunk létre egyet `Intertech` néven:

```
.namespace Intertech
{
  .namespace MyNamespace {}
}
```

A C#-hoz hasonlóan, a CIL lehetővé teszi beágyazott névtér definiálását a következők szerint:

```
// Beágyazott névtér definiálása
.namespace Intertech.MyNamespace{}
```

Osztálytípusok definiálása CIL-ben

Egy új osztálytípus definiálására a `.class` direktívát használjuk. Ez az egyszerű direktíva számos további attribútummal bővíthető a típus természetének további minősítéséhez. Ennek illusztrálására adjunk egy `MyBaseClass` nevű nyilvános osztályt a névterünkhöz. Ahogyan a C#-ban is, ha nem határozunk meg explicit ősosztályt, a típusunk automatikusan a `System.Object` könyvtárból keletkezik, ha csak nem fordítjuk a `*.il` kódot az `ilasm.exe /noautoinherit` opciójának alkalmazásával.

```
.namespace MyNamespace
{
  // System.Object alaposztály használatát feltételezi
  .class public MyBaseClass {}
}
```

Ha egy olyan osztálytípust hozunk létre, amely nem a `System.Object`ből, hanem valamely más osztályból származik, az `extends` attribútumokat használjuk. Ha ugyanazon a szerelvényen belül definiált típusra kell hivatkoznunk, a CIL megköveteli, hogy használjuk a teljesen meghatározott nevet (bár ha az alaptípus ugyanabban a szerelvényben van, mellőzhetjük a szerelvény barátságos nevének prefixumát). Így a következő kísérlet a `MyBaseClass` bővítésére fordítási hibát fog eredményezni:

```
// Ez a kód nem fog lefordulni!
.namespace MyNamespace
{
  .class public MyBaseClass {}

  .class public MyDerivedClass
    extends MyBaseClass {}
}
```

A `MyDerivedClass` szülőosztályának helyes definiálásához a `MyBaseClass` teljes nevét kell megadni az alábbiak szerint:

```
// Már jobb!
.namespace MyNamespace
{
  .class public MyBaseClass {}

  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass {}
}
```

A `public` és az `extends` attribútumokon felül a CIL-osztálydefiníció számos további minősítőt fog használni, amelyek kontrollálják a típusok láthatóságát, a mezők elrendezését és így tovább. A 19.3. táblázat bemutat néhány (de nem minden) attribútumot, amely a `.class` direktívával kapcsolatosan használható.

Attribútumok	Valós jelentés
`public, private, nested assembly, nested famandassem, nested family, nested famorassem, nested public, nested private`	A CIL különböző attribútumokat definiál, amelyeket egy adott típus láthatóságának meghatározására használhatunk. A nyers CIL számos olyan lehetőséget kínál, amelyek eltérnek a C# által kínáltaktól.
`abstract, sealed`	Ez a két attribútum egy `.class` direktívába beépítve egy absztrakt vagy egy lezárt osztályt definiálhat.
`auto, sequential, explicit`	Ezek az attribútumok a CLR-t tájékoztathatják arról, hogyan rendezze el a mezőket a memóriában. Az osztálytípus számára az alapértelmezett elrendezési jelző (`auto`) a megfelelő.
`extends, implements`	Ezek az attribútumok lehetővé teszik egy típus ősosztályának definiálását (`extends`) vagy egy interfész implementálását a típuson (`implements`).

19.3. táblázat: *A .class direktívával kapcsolatban használt különböző attribútumok*

Interfészek definiálása és implementálása CIL-ben

Interfésztípusokat CIL-ben a `.class` direktívával definiálunk. Amikor a `.class` direktívát `interface` attribútummal egészítjük ki, a típus egy CTS-interfész típusaként valósul meg. Ha már létrehoztunk egy interfészt, hozzáköthetjük egy osztály- vagy struktúratípushoz a CIL `implements` attribútumának a használatával:

```
.namespace MyNamespace
{
  // Egy interfészdefiníció.
  .class public interface IMyInterface {}
  .class public MyBaseClass {}
```

```
  // MyDerivedClass most implementálja az IAmAn Interface-t.
  .class public MyDerivedClass
    extends MyNamespace.MyBaseClass
    implements MyNamespace.IMyInterface {}
}
```

Interfészhierarchiák felépítésénél az interfészek alapinterfészekként működhetnek más interfésztípusok számára (lásd 9. fejezet). Ám az `extends` attribútum nem használható az A interfész származtatására a B interfészből. Az `extends` attribútum csak a típus ősosztályának a meghatározására használható. Ha egy interfészt szeretnénk kiterjeszteni, ahhoz az `implements` attribútumot kell használni:

```
// Interfészek kiterjesztése CIL-ben.
.class public interface IMyInterface {}
.class public interface IMyOtherInterface
  implements MyNamespace.IMyInterface {}
```

Struktúrák definiálása CIL-ben

A `.class` direktíva segítségével definiálhatunk egy CTS-struktúrát, ha a típus a `System.ValueType`-ból származik. A `.class` direktívát a `sealed` attribútum minősíti (hiszen a struktúrákból nem lehet származtatni). Ha másképp próbáljuk meg, az `ilasm.exe` fordítási hibát fog eredményezni.

```
//A struktúradefiníció mindig zárt
  .class public sealed MyStruct
    extends [mscorlib]System.ValueType{}
```

A CIL a struktúratípusok definiálásához egyszerűbb megoldást kínál. Ha a `value` attribútumot használjuk, az új típus automatikusan a `[mscorlib] System.ValueType` típusból fog származni. Ennek következtében a `MyStruct` definiálása a következők szerint történhet:

```
// Gyorsazonosítók egy struktúra deklarálására
.class public sealed value MyStruct{}
```

Felsorolt típusok definiálása CIL-ben

A .NET-es felsorolt típusok a `System.Enumból` származnak, ez viszont egy `System.ValueType` (és így lezárt). Ha egy felsorolt típust szeretnénk CIL-ben definiálni, egyszerűen ki kell terjesztenünk az `[mscorlib]System.Enum` típust.

```
// Egy felsorolás
  .class public sealed MyEnum
    extends [mscorlib]System.Enum{}
```

A struktúradefiníciókhoz hasonlóan a felsorolások az `enum` attribútum használatával, egyszerűbben is definiálhatók:

```
// Enum gyorsazonosítók
.class public sealed enum MyEnum{}
```

A következőkben megnézzük, hogyan adhatjuk meg egy felsorolt típus név/érték párját.

Megjegyzés A másik alap .NET-típus, a metódusreferencia is rendelkezik speciális CIL-ábrázolással. Részletek a 11. fejezetben találhatók.

Generikus típusok definiálása a CIL-ben

A generikus típusok is rendelkeznek speciális ábrázolással a CIL szintaxisában. Egy adott generikus típus vagy generikus tag egy vagy több típusparaméterrel rendelkezhet (lásd 10. fejezet). A `List<T>` típus például egyetlen, míg a `Dictionary<TKey,TValue>` két típusparaméterrel rendelkezik. A CIL-kódban számos típusparamétert egy hátrafelé dőlő egyszerű vessző (`) használatával adunk meg, amelyet a típusparaméterek száma követ. A C#-hoz hasonlóan a típusparaméter aktuális értéke szögletes zárójelben van.

Megjegyzés A ` karakter az angol billentyűzeten a Tab feletti billentyűn (az 1 billentyűtől balra) található, a magyar billentyűzeten a jobb oldali Alt (AltGr) + 7 billentyűkombinációval hozható létre.

Tegyük fel például, hogy egy `List<T>` típust szeretnénk létrehozni, ahol a `T` `System.Int32` típusú. CIL-ben a következőket tehetjük:

```
// C#-ban: List<int> myInts = new List<int>();
newobj instance void class [mscorlib]
  System.Collections.Generic.List`1<int32>::.ctor()
```

Ez a `List`<int32>`-ként definiált generikus osztály, ugyanis a `List<T>` egyetlen típusparaméterrel rendelkezik. Ha egy `Dictionary<string, int>` típus definiálására van szükségünk, a következők szerint járhatunk el:

```
// C#-ban: Dictionary<string, int> d =
//         new Dictionary<string, int>();
newobj instance void class [mscorlib]
System.Collections.Generic.Dictionary`2<string,int32>::.ctor()
```

Másik példaként vegyünk egy olyan generikus típust, amelyet más generikus típus használ típusparaméterként, a CIL-kódot a következők szerint hozhatjuk létre:

```
// C#-ban: List<List<int>> myInts = new List<List<int>>();
newobj instance void class [mscorlib]
  System.Collections.Generic.List`1<class
    [mscorlib]System.Collections.Generic.List`1<int32>>::.ctor()
```

Amikor létrehozunk egy generikus osztályt, struktúrát vagy interfészt, a típusdeklarációnál ugyanezt a szintaxist használhatjuk. Például:

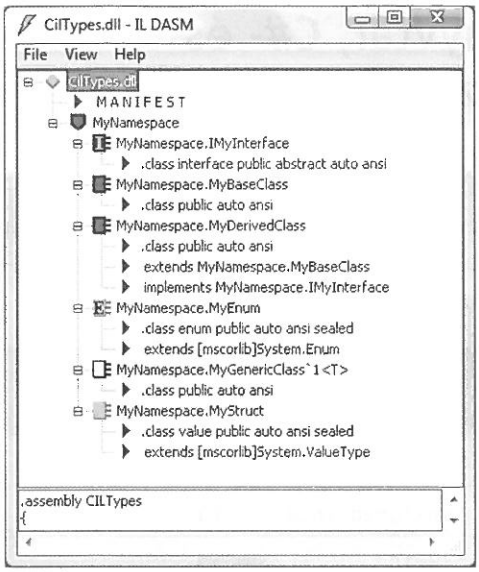
```
// Egyedi generikus osztály 1 paramétertípussal
.class public MyGenericClass`1<T>{}
```

A CILTypes.il fájl fordítása

Annak ellenére, hogy a típusunk még sem tagokkal, sem logikával nem rendelkezik, a `*.il` fájlunkat mégis lefordíthatjuk .NET DLL-re (valójában másra nem is tudnánk, hiszen a kód nem tartalmaz `Main()` metódust és így belépési pontot sem). Nyissunk meg egy parancssort, és gépeljük be a következő parancsot az `ilasm.exe`-nek.

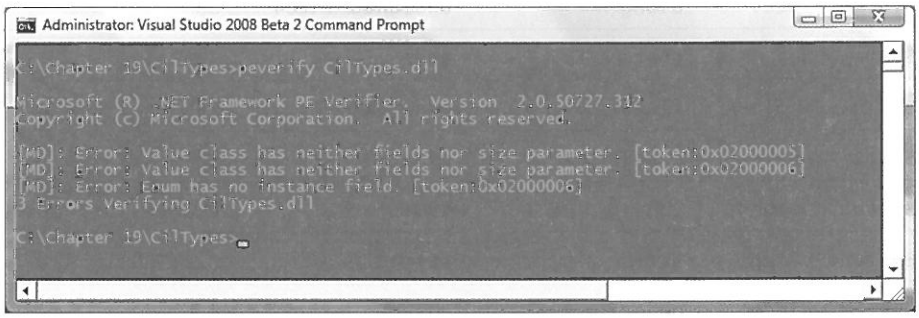
```
ilasm /dll CilTypes.il
```

Ekkor megnyithatjuk a binárisunkat az `ildasm.exe`-ben (lásd a 19.5. ábrát).



19.5. ábra: *A CIL Types.dll szerelvény*

Ha ellenőriztük a szerelvényünk tartalmát, futassuk újra a `peverify.exe` fájlt. Számos hibaüzenetet kapunk, ugyanis az összes típusunk teljesen üres (lásd a 19.6. ábrát).



19.6. ábra: *Üres típusok ellenőrzési hibákat eredményeznek*

Annak megértésére, hogy hogyan töltjük fel a típust tartalommal, először is meg kell vizsgálnunk a CIL alapvető adattípusait.

.NET-alaposztálykönyvtár, C#- és CIL-adattípus-leképzések

A 19.4. táblázat mutatja, hogy hogyan jön létre a .NET-ősosztálytípus leképzése a megfelelő C#-kulcsszavakba, és az egyes C#-kulcsszavak leképzése a nyers CIL-be. A táblázat a konstans álneveket is dokumentálja minden CIL-típushoz. Ezekre a konstansokra gyakran számos CIL-opcode hivatkozik.

.NET-ősosztály-típus	C#-kulcsszó	CIL-beli ábrázolás	CIL-konstans-azonosító
System.SByte	sbyte	int8	I1
System.Byte	byte	unsigned int8	U1
System.Int16	short	int16	I2
System.UInt16	ushort	unsigned int16	U2
System.Int32	int	int32	I4
System.UInt32	uint	unsigned int32	U4
System.Int64	long	int64	I8
System.UInt64	ulong	unsigned int64	U8
System.Char	char	char	CHAR
System.Single	float	float32	R4
System.Double	double	float64	R8
System.Boolean	bool	bool	BOOLEAN
System.String	string	string	n.a.
System.Object	object	object	n.a.
System.Void	void	void	VOID

19.4. táblázat: *.NET-ősosztálytípus leképzése C#-kulcsszavakba és C#-kulcsszavak leképzése CIL-be*

Típustagok definiálása CIL-ben

A .NET-típusok különböző tagokat támogatnak. A felsorolt típusok rendelkeznek név/érték párokkal. A struktúrák és osztályok tartalmazhatnak konstruktorokat, mezőket, metódusokat, tulajdonságokat, statikus tagokat és így tovább. A könyv eddigi fejezeteiben már láttunk részleges CIL-definíciókat ezekhez az elemekhez, ám érdemes átismételni, hogy a különböző tagokat hogyan képezzük le a CIL-primitívekbe.

Adatmezők definiálása CIL-ben

A felsorolások, struktúrák és osztályok mind támogatják az adatmezőket. Mindegyik esetben a `.field` direktívát használjuk. Vegyük például a `MyEnum` felsorolt típus vázlatát, és definiáljunk három név/érték párt (az értékeket zárójelben adjuk meg).

```
.class public sealed enum MyEnum
{
  .field public static literal valuetype
  MyNamespace.MyEnum A = int32(0)
  .field public static literal valuetype
  MyNamespace.MyEnum B = int32(1)
  .field public static literal valuetype
  MyNamespace.MyEnum C = int32(2)
}
```

Egy `System.Enumból` származó típus hatókörében található mezőket a `static` és `literal` attribútumokkal adjuk meg. Ezek az attribútumok az adatmezőt egy, magából a típusból elérhető, fix értéknek állítják be (pl. `MyEnum.NameOne`).

Megjegyzés Egy `enum` értékhez hozzárendelt értékek `0x` előtaggal hexadecimálisak is lehetnek.

Amikor egy adatmezőt szeretnénk definiálni az osztályon vagy a struktúrán belül, akkor nem korlátozódunk a nyilvános statikus adatokra. Módosíthatjuk például a `MyBaseClass` osztályt úgy, hogy támogasson két privát, példányszintű adatmezőpontot:

```
.class public MyBaseClass
{
  .field private string stringField
  .field private int32 intField
}
```

A C#-hoz hasonlóan, az osztályok adatmezői automatikusan megfelelő alapértelmezett értékkel jönnek létre. Ha lehetővé szeretnénk tenni az objektum felhasználójának, hogy a létrehozás idejében a saját adatmező minden egyes pontjához egyedi értékeket adjon meg, (természetesen) szükség lesz egyedi konstruktorok megalkotására.

Típuskonstruktorok definiálása CIL-ben

A CTS támogatja mind a példányszintű, mind az osztályszintű (statikus) konstruktorokat. A CIL-kódban a példányszintű konstruktorokat a `.ctor` token jelzi, míg a statikus szintű konstruktorok `.cctr` tokenen keresztül fejezhetők ki (osztálykonstruktor). Mindkét CIL-tokent a `rtspecialname` (a típus speciális nevét adja vissza) és a `specialname` attribútumok használatával adhatjuk meg. Ezek az attribútumok egyszerűen arra használhatók, hogy olyan adott CIL-tokeneket azonosítsunk, amelyek egy adott .NET-nyelven keresztül egyedileg kezelhetők. A C#-ban például a konstruktorok nem definiálnak visszatérési típust, de a CIL-kódban egy konstruktor visszatérési értéke valóban üres (`void`):

```
.class public MyBaseClass
{
  .field private string stringField
  .field private int32 intField

  .method public hidebysig specialname rtspecialname
    instance void .ctor(string s, int32 i) cil managed
  {
    // Teendő: A megvalósítás kódjának megírása...
  }
}
```

A `.ctor` direktíva az `instance` attribútummal adható meg (ugyanis ez nem statikus konstruktor). A *cil managed* attribútum, azt jelöli, hogy a metódus CIL-utasításokat tartalmaz, nem pedig nem felügyelt kódot, amelyet például az operációs rendszer hívására használhatnánk.

Tulajdonságok definiálása CIL-ben

A tulajdonságoknak és a metódusoknak is van saját CIL-jelölésük. Ha például a `MyBaseClass` osztályt módosítjuk egy `TheString` nevű nyilvános tulajdonság támogatásához, a következő CIL-kódot alkalmazhatjuk (ismét figyeljük meg a `specialname` attribútumot):

```
.class public MyBaseClass
{
  ...
  .method public hidebysig specialname
  instance string get_TheString() cil managed
  {
    // TODO: A megvalósítás kódjának megírása...
  }

  .method public hidebysig specialname
    instance void set_TheString(string 'value') cil managed
  {
    // TODO: A megvalósítás kódjának megírása...
  }

  .property instance string TheString()
  {
    .get instance string
      MyNamespace.MyBaseClass::get_TheString()
    .set instance void
      MyNamespace. MyBaseClass::set_TheString(string)
  }
}
```

A CIL-kódban egy tulajdonságot egy olyan metóduspárba képezünk le, amely `get_` és `set_` előtagokkal rendelkezik. A `.property` direktíva a kapcsolódó `.get` és `.set` direktívákat használja a tulajdonságszintaxis leképzéséhez a megfelelő „speciális név" metódusba.

Megjegyzés A fenti példában aposztrófok között helyezkedik el a tulajdonság `set` metódusának bemenő paramétere: ez lesz az értékadás-operátor jobb oldalán álló token neve.

Tagparaméterek definiálása

Röviden: az argumentumokat megadni a CIL-ben (többé-kevésbé) ugyanúgy lehet, mint a C#-ban. Például minden argumentumot adattípusának megadásával definiálunk, amelyet a paraméter neve követ. Továbbá a C#-hoz hason-

lóan a CIL is lehetőséget ad bemenő, kimenő és referencia szerint továbbított paraméterek definiálására. A CIL lehetővé teszi egy paramétertömb-argumentum (más néven C# `params` kulcsszó) definiálását éppúgy, mint opcionális paraméterekét (amelyek a C#-ban nem támogatottak, de a VB.NET-ben használatosak).

A paraméterek definiálási folyamatának ábrázolásához nyers CIL-ben hozzunk létre egy olyan metódust, amely egy `int32`-t használ (értékként), egy másik int32-t (referenciaként), egy `[mscorlib] System.Collection.ArrayList` típust és egy egyszerű kimeneti paramétert (int32 típus). A C#-kódban ez a metódus az alábbihoz hasonlóan nézne ki:

```
public static void MyMethod(int inputInt,
  ref int refInt, ArrayList ar, out int outputInt)
{
  outputInt = 0; // Csak a C#-fordító igényeinek kielégítésére...
}
```

Ha ugyanezt CIL-kódba képezzük le, a C#-referenciaparamétert úgy kell jelölni, hogy az adattípushoz (`int32&`) utótagként egy et jellel (&) illeszkedjen. A kimeneti paraméterek is használják az & utótagot, de ezen kívül még a CIL `[out]` tokenjét is alkalmazzuk a megadásukhoz. Ha a paraméter referenciatípus (a mi esetünkben a `[mscorlib] System.Collection.ArrayList` típus), akkor a `class` token az adattípus elé kerül (ne keverjük össze a `.class` direktívával):

```
.method public hidebysig static void MyMethod(int32 inputInt,
  int32& refInt
  class [mscorlib]System.Collections.ArrayList ar,
  [out] int32& outputInt) cil managed
{
  ...
}
```

A CIL-vezérlőkódok

Végül a CIL-kódot a különböző vezérlőkódok (opcode) szerepével kapcsolatban vizsgáljuk meg. Egy opcode egyszerűen egy olyan CIL-token, amelyet egy adott tag végrehajtási kódjának létrehozásához használunk. A CIL-vezérlőkódokat (elég sok van) a következő nagyobb kategóriákba csoportosíthatjuk:

- Vezérlőkódok, amelyek a program menetét vezérlik.
- Vezérlőkódok, amelyek kiértékelik a kifejezéseket.
- Vezérlőkódok, amelyek elérik az értékeket a memóriában (paramétereken, lokális változókon stb. keresztül).

Hogy lássuk a tagok megvalósítását a CIL-en keresztül, a 19.5 táblázat működésük szerint csoportosítva definiál néhányat azok közül a leggyakrabban használatos vezérlőkódok közül, amelyek közvetlenül a tag végrehajtási kódjához kapcsolódnak.

Vezérlőkódok	Valós jelentés
`add, sub, mul, div, rem`	Ezek a CIL-opcode-ok lehetővé teszik az összeadás, kivonás, szorzás és osztás műveletét (a `rem` adja vissza az osztási művelet maradékát).
`and, or, xor`	Ezek a CIL-opcode-ok lehetővé teszik bináris műveletek elvégzését két értéken.
`ceq, cgt, clt`	Ezek a CIL-opcode-ok lehetővé teszik két érték összehasonlítását a veremben különböző módokon, például: `ceq`: egyenlőségi összehasonlítás `cgt`: nagyobb, mint összehasonlítás `clt`: kisebb, mint összehasonlítás
`box, unbox`	Ezek az opcode-ok a referenciatípusok és az értéktípusok közötti konvertáláshoz használhatók.
`ret`	Ez az opcode arra használható, hogy kilépjünk egy metódusból, és a hívónak visszaadjunk egy értéket (ha szükséges).
`beq, bgt, ble, blt, switch`	Ezek az opcode-ok (továbbá még sok más kapcsolódó vezérlőkód) a metóduson belüli elágazó programkód felügyeletére használhatók, például: `beq`: Egyenlőség esetén megáll a kódcímkénél. `bgt`: Nagyobb mint reláció esetén megáll a kódcímkénél. `ble`: Nagyobb vagy egyenlő reláció esetén megáll a kódcímkénél. `blt`: Kisebb mint reláció esetén megáll a kódcímkénél. Minden elágazáscentrikus műveleti kód igényli, hogy CIL-kódcímkét definiáljunk az ugráshoz, amennyiben a teszt eredménye igaz.

Vezérlőkódok	Valós jelentés
`call`	Ez a CIL-opcode egy adott típus tagjának hívásához használható.
`nearer, newobj`	Ezek az opcode-ok lehetővé teszik egy új tömb-, illetve egy új objektumtípus elhelyezését a memóriában.

19.5. táblázat: *Különböző megvalósításspecifikus CIL-vezérlőkódok*

A CIL-vezérlőkódok következő nagyobb kategóriáját (ennek egy részét a 19.6. táblázat mutatja) argumentumok virtuális végrehajtó verembe betöltésére (írására) használjuk. Ezek a betöltésspecifikus opcode-ok egy `ld` (load) előtaggal rendelkeznek.

Műveleti kód	Valós jelentése
`ldarg` (számos variációval)	Betölti a metódus argumentumát a verembe. Az általános `ldarg` (amely az argumentumot azonosító adott indexszel együtt dolgozik) opcode-on kívül még számos más variáció létezik. Például azok az `ldarg` opcode-ok, amelyek numerikus utótaggal rendelkeznek (`ldarg_0`), huzalozottan kódolják, hogy melyik argumentum töltődjön be. Az `ldarg` opcode variációi lehetővé teszik az adattípus huzalozott kódolását a 19.4. táblázatban látható CIL-konstansazonosítók (`ldarg_I4`, `int32` számára), valamint -adattípus és érték (`ldarg_I4_5`, egy 5 értékű `int32` betöltéséhez) használatával is.
`ldc` (számos variációval)	Betölt egy konstans értéket a verembe.
`ldfld` (számos variációval)	Betölti egy példányszintű mező értékét a verembe.
`ldloc` (számos variációval)	Betölti egy lokális változó értékét a verembe.
`ldobj`	Megkapja az összes heapalapú objektum által összegyűjtött értéket, és elhelyezi őket a veremben.
`ldstr`	Egy sztringértéket tölt be a verembe.

19.6. táblázat: *A CIL elsődleges veremcentrikus műveleti kódjai*

Az olvasásspecifikus vezérlőkódokon kívül a CIL még számos opcode-ot nyújt, amelyek *explicit módon* távolítják el a legfelső értéket a veremből. Ahogyan a fejezet első néhány példájában bemutattuk, egy érték eltávolítása a

veremből tipikusan magában foglalja az érték tárolását is egy ideiglenes helyi tárolóban, további felhasználás céljából (pl. paraméterként a következő metódushíváshoz). Az aktuális értéket a virtuális végrehajtási veremből kiolvasó opcode-ok egy `st` (store) előtagot használnak. A 19.7. táblázat a legfontosabbakat érinti.

Müveleti kód	Valós jelentése
`pop`	Eltávolítja az aktuális értéket a végrehajtási verem tetejéről, de nem törődik az érték tárolásával.
`starg`	Eltárolja a verem tetején lévő értéket a metódus argumentumába, adott indexszel.
`stloc` (számos variációval)	Olvassa az aktuális értéket a végrehajtási verem tetejéről, és eltárolja azt egy lokális változólistában, adott indexszel.
`stobj`	Átmásolja egy adott típus értékét a végrehajtási veremből egy megadott memóriacímre.
`stsf ld`	Lecseréli a statikus mező értékét egy végrehajtási veremből származó értékre.

19.7. táblázat: *Különböző eltávolításcentrikus vezérlőkódok*

A különböző CIL-opcode-ok *implicit módon* olvassák a veremből az értéket az aktuális feladat elvégzéséhez. Ha például megpróbáljuk elvégezni két szám kivonását a `sub` opcode használatával, nyilvánvaló, hogy a `sub` opcode-nak olvasnia kell a következő két elérhető értéket, mielőtt végrehajtja a számítást. A számítás végeztével az eredmény ismét bekerül a verembe.

A .maxstack direktíva

Amikor nyers CIL felhasználásával egy metódus implementációját írjuk, figyelnünk kell a `.maxstack` nevű speciális direktívára. A `.maxstack` meghatározza, hogy maximálisan mennyi elem írható a verembe egyszerre a metódus futása alatt. A `.maxstack` direktívának van egy alapértelmezett értéke (8), amely az általunk használt metódusok többségének biztonságos lesz. Ha nagyon egyértelműek szeretnénk lenni, lehetőségünk van manuálisan meghatározni a lokális változók számát a veremben, valamint ezt az értéket explicit módon definiálni.

```
.method public hidebysig instance void
  Speak() cil managed
{
  // A metódus hatókörében, tulajdonképpen
  // az 1 érték (a literálstring) van a veremben.
  .maxstack 1
  ldstr "Hello there..."
  call void [mscorlib]System.Console::WriteLine(string)
  ret
}
```

Lokális változók deklarálása CIL-ben

Először is vizsgáljuk meg, hogyan kell deklarálni egy lokális változót. Tegyük fel, hogy szeretnénk létrehozni CIL-ben egy `MyLocalVariables()` nevű metódust, amely nem használ argumentumokat, és `void` a visszatérési értéke. A metóduson belül szeretnénk definiálni három lokális változót a következő típusokkal: `System.String`, `System.Int32` és `System.Object`. C#-ban ez a tag a következőképpen jelenne meg (a helyi hatókörű változók nem kapnak alapértelmezett értéket, és be kell állítani egy kiinduló állapotot a további használat előtt):

```
public static void MyLocalVariables()
{
  string myStr = "CIL code is fun!";
  int myInt = 33;
  object myObj = new object();
}
```

Ha a `MyLocalVariables()` metódust közvetlenül a CIL-ben hozzuk létre, a következőket kell tennünk:

```
.method public hidebysig static void
  MyLocalVariables() cil managed
{
  .maxstack 8
  //Három helyi változót definiál.
  .locals init ([0] string myStr, [1] int32 myInt, [2] object myObj)

  // Betölt egy stringet a virtuális végrehajtási verembe.
  ldstr "CIL code is fun!"
  // Eltávolítja az aktuális értéket és lokális változóban
  // tárolja [0].
  stloc.0

  //Betölt egy "i4" típusú konstanst
  // (int32 gyorsazonosítója) 33-ra állítja az értékét.
  ldc.i4 33
```

```
    // Eltávolítja az aktuális értéket, és lokális változóban
    // tárolja [1].
    stloc.1

    // Létrehoz egy új objektumot, és elhelyezi a veremben.
    newobj instance void [mscorlib]System.Object::.ctor()
    // Eltávolítja az aktuális értéket, és lokális változóban
    // tárolja [2].
    stloc.2
  ret
  }
```

Az első lépés egy lokális változó meghatározásához a nyers CIL-ben a `.locals` direktíva használata, amely az `init` attribútummal áll párban. Az ehhez tartozó zárójel hatókörében az a célunk, hogy egy adott numerikus indexet kapcsoljunk minden egyes változóhoz (ezek itt `[0]`, `[1]` és `[2]`). Minden index azonosítása az adattípusával és egy változónévvel történik. Ha a lokális változókat már definiáltuk, betöltünk egy értéket a verembe (a különböző betöltéscentrikus opcode-ok segítségével), és tároljuk az értéket a lokális változóban (a különböző tároláscentrikus opcode-ok segítségével).

Paraméterek leképzése lokális változókba CIL-ben

A következő kérdés: Hogyan képezzük le a bejövő paramétereket a helyi metódusokba? Nézzük meg a következő statikus C#-metódust:

```
public static int Add(int a, int b)
{
  return a + b;
}
```

Ez a metódus igen sokatmondó a CIL-kódban. Először is, a bejövő argumentumokat (`a` és `b`) be kell írni a virtuális végrehajtási verembe a `ldarg` (load argument) opcode használatával. Ezután az `add` opcode-dal kiolvassuk a veremből a következő két értéket, és elvégezzük az összeadást, majd az értéket ismét eltároljuk a veremben. Végül ezt az összeget kiolvassuk a veremből, és visszaadjuk a hívónak a `ret` opcode-on keresztül. Ha visszafejtenénk ezt a C#-metódust az `ildasm.exe` segítségével, számos olyan további tokent találnánk, amelyet a `csc.exe` helyezett el benne, de CIL-kód lényegi része nagyon egyszerű:

```
.method public hidebysig static int32 Add(int32 a,
  int32 b) cil managed
{
  .maxstack 2
  ldarg.0    // "a" betöltése a verembe.
  ldarg.1    // "b" betöltése a verembe.
  add        // Két érték hozzáadása.
  ret
}
```

A rejtett „this" referencia

A két bejövő argumentumra (a és b) a CIL-kódon belül indexelt pozíciójukkal (index 0 és index 1) hivatkozunk, hiszen a virtuális végrehajtási verem az indexelést a 0-tól kezdi.

CIL-kód vizsgálatakor vagy létrehozásakor figyelnünk kell arra, hogy minden, bejövő argumentumokat használó, nem statikus metódus automatikusan kap egy implicit paramétert, amely referenciát jelent az aktuális objektumhoz (gondoljunk a C# `this` kulcsszavára). Ebből adódóan ha az `Add()` metódust nem statikusként definiálnánk, a következőt kapnánk:

```
// Többé már nem statikus!
public int Add(int a, int b)
{
  return a + b;
}
```

A bejövő `a` és `b` argumentumokat a `ldarg.1` és a `ldarg.2` opcode-ok használatával olvassuk be (a várt `ldarg.0` és `ldarg.1` opcode-ok helyett). Ennek az az oka, hogy a 0 valójában a referenciát tartalmazza. Nézzük meg a következő pszeudokódot:

```
// Ez CSAK pszeudokód!
.method public hidebysig static int32 AddTwoIntParams(
  MyClass_HiddenThisPointer this, int32 a, int32 b) cil managed
{
  ldarg.0 // MyClass_HiddenThisPointer betöltése a verembe.
  ldarg.1 // "a" betöltése a verembe.
  ldarg.2 //"b" betöltése a verembe.
  ...
}
```

Ciklusok reprezentációja a CIL-ben

A C# programnyelvben a `for`, `foreach`, `while` és `do` kulcsszavakkal megvalósított iterációs szerkezetek mindegyikének van megfelelő reprezentációja a CIL-ben. Nézzük meg a klasszikus `for` ciklust:

```
public static void CountToTen()
{
  for(int i = 0; i < 10; i++)
    ;
}
```

A `br` opcode-ok (`br`, `blt` és így tovább) segítségével feltételes ugrásokat helyezhetünk el a programban. Ebben a példában olyan feltételt állítottunk fel, amelyben a `for` ciklusból akkor lép ki, ha az `i` lokális változó egyenlő vagy nagyobb, mint 10. Minden átadáskor az i értéke eggyel nő, ezután a tesztfeltételeket újra ki kell értékelni.

Amikor valamelyik CIL elágazási opcode-ot használjuk, szükségünk lesz egy (vagy két) saját kódcímke definiálására, amely az ugrás helyét jelöli, abban az esetben, ha a feltétel fennáll. Vizsgáljuk meg az itt található `ildasm.exe` által létrehozott (megjegyzésekkel ellátott) CIL-kódot (az automatikusan generált kódcímkékkel együtt):

```
.method public hidebysig static void CountToTen() cil managed
{
  .maxstack 2
  .locals init ([0] int32 i)  // Inicializálja a lokális "i"
                              // integert.
  IL_0000: ldc.i4.0           // Betölti ezt az értéket a verembe.
  IL_0001: stloc.0            // Eltárolja az értéket "0" indexszel.
  IL_0002: br.s IL_0008       // Ugrik a IL_0008-hoz.
  IL_0004: ldloc.0            // Betölti a 0 indexű változó értékét.
  IL_0005: ldc.i4.1           // Betölti az "1" értéket a verembe.
  IL_0006: add                // Az aktuális értéket hozzáadja
                              // a veremhez 0 indexszel.
  IL_0007: stloc.0
  IL_0008: ldloc.0            // Betölti a "0" indexű értéket.
  IL_0009: ldc.i4.s 10        // Betölti a "10" értéket a verembe.
  IL_000b: blt.s IL_0004      // Kevesebb? Ha igen, visszaugrik
                              // az IL_0004-re
  IL_000d: ret
}
```

Röviden: ez a CIL-kód a helyi `int32` definiálásával és verembe olvasásával kezdődik. Hátra és előre ugrunk az `IL_0008` és az `IL_0004` kódcímkék között, az i értékét minden alkalommal 1-gyel növeljük, és teszteljük, hogy az i még mindig kevesebb-e, mint 10. Ha igen, kilépünk a metódusból.

Forráskód A `CilTypes` példa megtalálható a 19. fejezet alkönyvtárában.

.NET-szerelvény készítése köztes nyelven

A következőkben létrehozunk egy .NET-alkalmazást, amelyet csak az `ilasm.exe` és egy tetszőleges szövegszerkesztő segítségével végzünk. Az alkalmazásunk egy saját fejlesztésű egyfájlos `*.dll` fájlból, amely két osztálytípus-definíciót tartalmaz, és egy konzolalapú `*.exe` fájlból fog állni, amely ezekkel a típusokkal kommunikál.

A CILCars.dll létrehozása

Először hozzuk létre a `*.dll`-t a kliens általi felhasználáshoz. Nyissunk meg egy szövegszerkesztőt, és készítsünk egy `CILCars.il` nevű új `*.il` fájlt. Az egyfájlos szerelvény két külső .NET-binárist fog használni. Kezdjük a kódfájl módosításával az alábbiak szerint:

```
// mscorlib.dll és
// System.Windows.Forms.dll referencia
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
.assembly extern System.Windows.Forms
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
```

```
// Egyfájlos szerelvény definiálása
.assembly CILCars
{
  .hash algorithm 0x00008004
 .ver 1:0:0:0
}
.module CILCars.dll
```

A szerelvény két osztálytípust fog tárolni. Az első típus, a `CILCar`, két adatmezőt és egy egyedi konstruktort definiál. A második típus, a `CarInfoHelper`, egy `DisplayCarInfo()` nevű egyszerű statikus metódust definiál, amely a `CILCar`-t használja paraméterként, és `voiddal` tér vissza. Mindkét típus a `CILCars` névtérben található. CIL-kódban a `CILCar` a következőképpen valósítható meg:

```
//A CILCars.CILCar típus megvalósítása.
.namespace CILCars
{
  .class public auto ansi beforefieldinit CILCar
    extends [mscorlib]System.Object
  {
    // A CILCar adatmezője.
    .field public string petName
    .field public int32 currSpeed

    //Az egyedi konstruktor egyszerűen megengedi a hívónak,
    //hogy értéket adjon az adatmezőnek.
    .method public hidebysig specialname rtspecialname
      instance void .ctor(int32 c, string p) cil managed
    {
      .maxstack 8

      // Betölti az első argumentumot a verembe, majd meghívja
      // az ősosztálykonstruktort.
      ldarg.0 // "this" object, not the int32!
      call instance void [mscorlib]System.Object::.ctor()

      // Most betölti az első és a második argumentumot a verembe.
      ldarg.0    // "this" objektum
      ldarg.1    // int32 arg

      // Tárolja a legfelső veremtagot (int 32) a currSpeed mezőben.
      stfld int32 CILCars.CILCar::currSpeed

      // Betölti a stringargumentumot, és tárolja a petName mezőben.
      ldarg.0    // "this" objektum
      ldarg.2    // string arg
      stfld string CILCars.CILCar::petName
      ret
    }
  }
}
```

A valódi első argumentum bármely nem statikus taghoz az aktuális objektumreferencia lesz. A CIL első blokkja egyszerűen betölti az objektumreferenciát, és meghívja az ősosztálykonstruktort. Ezután írjuk a bemeneti konstruktorargumentumokat a verembe, és tároljuk a típus adatmezőjében a `stfld` (store in field) opcode használatával.

Ezután implementáljuk a második típust a névtérben: `CILCarInfo`. A típus lényege a statikus `Display()` metóduson belül található. Röviden: ennek a metódusnak a szerepe a bemeneti `CILCar` paraméter használata, adatmezője értékének kifejtése és megjelenítése egy Windows Forms-üzenetdobozban. A `CILCarInfo` teljes implementációja magyarázatokkal együtt a következő:

```
.class public auto ansi beforefieldinit CILCarInfo
  extends [mscorlib]System.Object
{
  .method public hidebysig static void
    Display(class CILCars.CILCar c) cil managed
  {
    .maxstack 8

    // Szükségünk van egy lokális sztringváltozóra.
    .locals init ([0] string caption)

    // Betölti a sztringet és a bemeneti CILCar-t a verembe.
    ldstr "{0}'s speed is:"
    ldarg.0

    // Most elhelyezi a CILCar petName-jének értékét
    // a veremben, és hívja a statikus String.Format() metódust.
    ldfld string CILCars.CILCar::petName
    call string [mscorlib]System.String::Format(string, object)
    stloc.0

    // Most betölti a currSpeed mező értékét, és megkapja
    // a sztringjének a reprezentációját (figyeljük meg
    // a ToString() hívást).
    ldarg.0
    ldflda int32 CILCars.CILCar::currSpeed
    call instance string [mscorlib]System.Int32::ToString()
    ldloc.0

    // Most hívja a MessageBox.Show () metódust betöltött
    // értékekkel.
    call valuetype [System.Windows.Forms]
      System.Windows.Forms.DialogResult
       [System.Windows.Forms]
      System.Windows.Forms.MessageBox::Show(string, string)
    pop
    ret
  }
}
```

Bár a CIL-kód mennyisége egy kicsit több, mint amit a `CILCar` implementációjában látunk, mégis sokkal egyszerűbb megoldással van dolgunk. Először is, mivel egy statikus metódust definiálunk, nem kell foglalkoznunk a rejtett objektumreferenciával (a `ldarg.0` opcode valóban a bemeneti `CILCar` argumentumot tölti be).

A metódus a sztring („{0}' s speed is") verembe olvasásával kezdődik, és a `CILCar` argumentum követi. Ha ez a két érték a helyén van, betöltjük a `petName` mező értékét, és meghívjuk a statikus `System.String.Format()` metódust, hogy helyettesítsük a kapcsos zárójellel megadott helyőrzőt a `CILCar petname` mezőjével.

Ugyanez az általános eljárás megy végbe a `currSpeed` mező feldolgozása során, ám az `ldflda` opcode-ot használjuk, amely betölti az argumentumcímet a verembe. Ekkor meghívjuk a `System.Int32.ToString()`-et, hogy az említett címen található értéket egy sztringtípusba transzformáljuk. Végül, ha már mindkét sztring szükséges formázása megtörtént, meghívjuk a `MessageBox.Show()` metódust.

Ekkor lefordíthatjuk az új `*.dll` fájlt az `ilasm.exe` segítségével a következő paranccsal:

```
ilasm /dll CILCars.il
```

és ellenőrizzük a benne lévő CIL-kódot a `peverify.exe` használatával:

```
peverify CILCars.dll
```

A CILCarClient.exe létrehozása

Már létre tudunk hozni egy egyszerű `*.exe` szerelvényt, amely

- létrehoz egy `CILCar` típust,
- átadja a típust a statikus `CILCarInfo.Display()` metódusnak.

Hozzunk létre egy új fájlt `CarClient.il` néven, és adjunk referenciát az `mscorlib.dll` és `CILCars.dll` szerelvényekre (ne feledjük majd el az utóbbit a készülő `CarClient` alkalmazás könyvtárába helyezni). Ezután egy egyszerű típust (`Program`) definiálunk, amely kezeli a `CILCars.dll` szerelvényt. A teljes kód a következő:

```
// Külsőszerelvény-referencia.
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )
  .ver 2:0:0:0
}
.assembly extern CILCars
{
  .ver 1:0:0:0
}

// A mi végrehajtható szerelvényünk.
.assembly CarClient
{
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.module CarClient.exe

// A Program típus implementációja.
.namespace CarClient
{
  .class private auto ansi beforefieldinit Program
  extends [mscorlib]System.Object
  {
    .method private hidebysig static void
    Main(string[] args) cil managed
    {
      // Megjelöli a *.exe belépési pontját.
      .entrypoint
      .maxstack 8

      // Egy lokális CILCar típust deklarál, és elhelyezi
      // az értéket a veremben a konstruktor hívásához.
      .locals init ([0] class
       [CILCars]CILCars.CILCar myCilCar)
      ldc.i4 55
      ldstr "Junior"

      // Új CILCar-t hoz létre; tárolja és betölti a referenciát.
      newobj instance void
        [CILCars]CILCars.CILCar::.ctor(int32, string)
      stloc.0
      ldloc.0

      // Meghívja a Display()-t, és a legfelső értéket átadja
      // a verembe.
      call void [CILCars]
        CILCars.CILCarInfo::Display(
          class [CILCars]CILCars.CILCar)
      ret
    }
  }
}
```

Az egyik opcode az `.entrypoint`. Ez az opcode azt jelöli, hogy egy `*.exe` melyik metódusa lesz a modul belépési pontja. Mivel a CLR elsőként az `.entrypointtal` ellátott metódust indítja, ezt a metódust tetszőlegesen lehet hívni, jóllehet itt a szabványos `Main()` szabványmetódus-nevet használjuk. A `Main()` metódusban található további CIL-kódot csak a veremalapú értékek írására és olvasására használjuk.

A `CILCar` létrehozása magába foglalja a `.newobj` opcode használatát. Amikor egy típus tagját nyers CIL használatával szeretnénk hívni, akkor a kettőspont szintaxist és, mint mindig, a típus teljesen meghatározott nevét alkalmazzuk. Így lefordíthatjuk az új fájlunkat az `ilasm.exe` segítségével, ellenőrizhetjük szerelvényünket a `peverify.exe` segítségével, és végrehajthatjuk a programunkat. Kiadhatjuk a következő parancsot a parancssorunkban:

```
ilasm CarClient.il
peverify CarClient.exe
CarClient.exe
```

A 19.7. ábra mutatja a végeredményt.

19.7. ábra: *A CILCarunk működés közben*

Mindezzel lezárhatjuk a CIL-ről szóló tudnivalók bevezetését. Ez volt a fejezet elsődleges célja. Reméljük, az `ildasm.exe` (vagy egy egyszerű eszköz) használatával immár meg tudunk nyitni egy .NET szerelvényt, és jobban megértjük, hogy tulajdonképpen mi is történik a színfalak mögött.

Forráskód A `CILCars` példa megtalálható a 19. fejezet alkönyvtárában.

Dinamikus szerelvények

Egy összetett .NET-alkalmazás létrehozása CIL-ben viszonylag egyszerű lehet. Egyrészt a CIL rendkívül kifejező programozási nyelv, amely lehetővé teszi a CTS által engedélyezett összes programozási szerkezettel a kommunikációt. Másrészt a nyers CIL létrehozása meglehetősen unalmas, és hibákat rejthet magában. Vajon fontos-e a CIL-szintaxis szabályait kívülről tudnunk? A válasz: attól függ. A legtöbb .NET programozási törekvésünk nem fogja igényelni a CIL-kód megtekintését, szerkesztését vagy létrehozását. A továbbiakban a dinamikus szerelvények (a statikus szerelvények ellentettje) világának és a `System.Reflection.Emit` névtér szerepének a vizsgálatával folytatjuk.

Az első kérdésünk az lehet: Mi a különbség a statikus és dinamikus szerelvények között? A *statikus szerelvények* közvetlenül a háttértárból beolvasott .NET-binárisok, ez pedig azt jelenti, hogy valahol a merevlemezen találhatók egy fizikai fájlban (vagy esetleg fájlokban - többfájlos szerelvények esetén). Minden alkalommal, amikor a C#-forráskódunkat lefordítjuk, egy statikus szerelvényt készítünk.

Ezzel szemben a *dinamikus szerelvényt* valamely alkalmazás futás közben hozza létre a `System.Reflection.Emit` névtér típusainak a segítségével. A `System.Reflection.Emit` névtér egy szerelvény és moduljainak, típusdefinícióinak és CIL implementációs kódjának létrehozását *futásidőben* teszi lehetővé. Ezután már szabadon elmenthetjük memóriabeli binárisunkat a lemezre. Ez természetesen új statikus szerelvényt eredményez. Dinamikus szerelvényt létrehozni a `System.Reflection.Emit` névtér használatával a CIL-opcode-ok bizonyos fokú ismeretét igényli.

Bár a dinamikus szerelvények készítése haladószintű (és szokatlan) programozási feladat, nagyon hasznos lehet adott körülmények között:

- Ha egy olyan .NET programozási eszközt hozunk létre, amelyhez felhasználói input alapján, igény szerint van szükség szerelvények létrehozására.
- Ha olyan programot hozunk létre, amelyhez menet közben van szükség proxyk előállítására távoli típusokhoz, a megszerzett metaadatok alapján.
- Ha szeretnénk betölteni egy statikus szerelvényt, majd ahhoz dinamikusan új típust hozzáadni.

A .NET futásidejű motor több szempontból is magában foglalja a dinamikus szerelvények létrehozását a háttérben. Az ASP.NET is ezzel a technikával fésüli össze a markupot és a szerveroldali kódot `dll`-lé futásidőben. A LINQ is tud menet közben kódot létrehozni különböző lekérdezéskifejezések alapján. Vizsgáljuk meg a `System.Reflection.Emit` névtérben található típusokat.

A System.Reflection.Emit névtér

Dinamikus szerelvények létrehozásához szükség van némi jártasságra a CIL-opcode-okban, de a `System.Reflection.Emit` névtér típusai, amennyire csak lehet, elrejtik a CIL összetettségét. Egy osztálytípus definiálásához például a CIL-direktívák és -attribútumok közvetlen meghatározása helyett egyszerűen használhatjuk a `TypeBuilder` osztályt. Hasonlóképpen, ha egy új, példányszintű konstruktort szeretnénk definiálni, nincs szükség a `specialname`, `rtspecialname` vagy `.ctor` token beszúrására; ehelyett a `ConstructorBuildert` használhatjuk. A 19.8. táblázat dokumentálja a `System.Reflection.Emit` kulcsfontosságú tagjait.

Tagok	Valós jelentés
`AssemblyBuilder`	Egy szerelvény (`*.dll` vagy `*.exe`) futásidejű létrehozásához használható. A `*.exe` meghívja a `ModuleBuilder.SetEntryPoint()` metódust a modul belépési pontjának a beállításához. Ha nincs meghatározva belépési pont, egy `*.dll` jön létre.
`ModuleBuilder`	Az aktuális szerelvényen belüli modulok definiálásához használjuk.
`EnumBuilder`	.NET felsorolt típus létrehozásához használjuk.
`TypeBuilder`	Egy modulon belül az osztályok, interfészek, struktúrák és metódusreferenciák futásidejű létrehozásához használhatjuk.

`MethodBuilder`
`LocalBuilder`
`PropertyBuilder`
`FieldBuilder`
`ConstructorBuilder`
`CustomAttributeBuilder`
`ParameterBuilder`
`EventBuilder`
	Típustagok (mint a metódusok, lokális változók, tulajdonságok, konstruktorok és attribútumok) futásidejű létrehozásához használjuk.

Tagok	Valós jelentés
`ILGenerator`	CIL-opcode-okat generál egy adott típustagba.
`OpCodes`	Számos mezőt biztosít, amelyek CIL-opcode-okra képeződnek le. Ezt a típust a `System.Reflection.Emit.ILGenerator` különböző tagjaival együtt használjuk.

19.8. táblázat: *A System.Reflection.Emit névtér tagjai*

Általában a `System.Reflection.Emit` névtér típusai teszik lehetővé nyers CIL-tokenek programozott megjelenítését a dinamikus szerelvényünk létrehozása közben.

A System.Reflection.Emit.ILGenerator szerepe

Az `ILGenerator` típus szerepe a CIL-opcode-ok injektálása egy adott típustagba. Közvetlenül nem hozhatunk létre `ILGenerator` objektumokat, hiszen ez a típus nem rendelkezik nyilvános konstruktorral, olyan létrehozó típusok metódusai, mint a `MethodBuilder` és a `ConstructorBuilder` képesek azonban erre:

```
// Kap egy ILGenerator-t a "myCtorBuilder" nevű
// ConstructorBuilder objektumból.
ConstructorBuilder myCtorBuilder =
  new ConstructorBuilder(/* ...various args... */);
ILGenerator myCILGen = myCtorBuilder.GetILGenerator();
```

Ha már van egy `ILGenerator` a kezünkben, akkor beilleszthetjük a nyers CIL-opcode-ot valamelyik metódus használatával. A 19.9 táblázat dokumentálja az `ILGenerator` néhány (de nem minden) metódusát.

Metódus	Valós jelentése
`BeginCatchBlock()`	Elkezd egy `catch` blokkot.
`BeginExceptionBlock()`	Elkezd egy kivételblokkot a nem szűrt kivételeknek.
`BeginFinallyBlock()`	Elkezd egy `finally` blokkot.
`BeginScope()`	Elkezd egy lexikális hatókört.
`DeclareLocal()`	Lokális változót deklarál.
`DefineLabel()`	Új címkét deklarál.

Metódus	Valós jelentése
`Emit()`	Sokszorosan túlterhelt, hogy engedélyezze CIL-opcode-ok generálását.
`EmitCall()`	Beilleszt egy `call` vagy `callvirt` opcode-ot a CIL folyamatába.
`EmitWriteLine()`	`Console.WriteLine()` metódushívást generál, amely különböző típusú értékekkel paraméterezhető.
`EndExceptionBlock()`	Befejez egy kivételblokkot.
`EndScope()`	Befejez egy lexikális hatókört.
`ThrowException()`	Kivétel dobására utasít.
`UsingNamespace()`	Meghatározza a névteret, amelyet a lokálisok kiértékelésére használunk, és felügyeli az aktuális aktív lexikális hatókört.

19.9. táblázat: *Az ILGenerator különböző metódusai*

Az `ILGenerator` kulcsfontosságú metódusa az `Emit()`, amely a `System.Reflection.Emit.OpCodes` osztálytípussal dolgozik együtt. Ez a típus nagyszámú írásvédett mezőt tár fel, amelyek a nyers CIL-vezérlőkódokba képződnek le. A tagoknak teljes készlete dokumentálva van az online súgóban.

Dinamikus szerelvény generálása

A .NET-szerelvény futásidejű definiálásának bemutatásához nézzük meg egy `MyAssembly.dll` nevű egyfájlos dinamikus szerelvény készítésének a folyamatát. Ebben a modulban van egy `HelloWorld` nevű osztály. A `HelloWorld` típus egy alapértelmezett konstruktort és egy egyedi konstruktort támogat, amelyet sztringtípusú privát tagváltozó (`theMessage`) értékének beállítására használunk. Továbbá a `HelloWorld` támogat egy `SayHello()` nevű nyilvános példányszintű metódust, amely egy üdvözlőszöveget ír ki a szabványos konzolra, és egy másik, `GetMsg()` nevű példánymetódust, amely visszaadja a belső privát sztringet. Valójában programozottan létrehozzuk a következő osztálytípust:

```
// Ez az osztály futásidőben fog létrejönni a
// System.Reflection.Emit használatával.
public class HelloWorld
{
  private string theMessage;
  HelloWorld() {}
  HelloWorld(string s) {theMessage = s;}

  public string GetMsg() {return theMessage;}
  public void SayHello()
  {
    System.Console.WriteLine("Hello from the HelloWorld class!");
  }
}
```

Tegyük fel, hogy létrehoztunk egy új `DynamicAsmBuilder` nevű Visual Studio 2008 parancssorialkalmazás-projektet, és importáljuk a `System.Reflection`, `System.Reflection.Emit` és `System.Threading` névtereket. Létrehozunk egy statikus metódust `CrateMyAsm()` névvel. Ez a metódus a következőkért felelős:

- a dinamikus szerelvény tulajdonságainak definiálásáért (név, verzió stb.),
- a `HelloClass` típus implementálásáért,
- a memóriabeli szerelvény mentéséért egy fizikai fájlba.

A `CreateMyAsm()` metódus egyedüli paraméterként egy `System.AppDomain` típust kap, amelyet az aktuális alkalmazástartományhoz társított `AssemlyBuilder` típus elérésére használunk (lásd a 17. fejezetet a .NET-alkalmazástartományokhoz). A teljes kód, a magyarázatokkal együtt a következő:

```
// A hívó beküld egy AppDomain típust.
public static void CreateMyAsm(AppDomain curAppDomain)
{
  // Általános szerelvénykarakterisztikákat hoz létre.
  AssemblyName assemblyName = new AssemblyName();
  assemblyName.Name = "MyAssembly";
  assemblyName.Version = new Version("1.0.0.0");

  // Új szerelvény létrehozása az aktuális alkalmazástartományban.
  AssemblyBuilder assembly =
    curAppDomain.DefineDynamicAssembly(assemblyName,
    AssemblyBuilderAccess.Save);

  // Mivel létrehoztunk egy egyfájlos szerelvényt,
  // a modul neve ugyanaz, mint a szerelvényé.
  ModuleBuilder module =
    assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");
```

```
// Nyilvános osztályt definiál "HelloWorld" névvel.
TypeBuilder helloWorldClass =
  module.DefineType("MyAssembly.HelloWorld",
  TypeAttributes.Public);

// A "theMessage" nevű, String típusú privát tagváltozó
// definiálása.
FieldBuilder msgField =
  helloWorldClass.DefineField("theMessage",
  Type.GetType("System.String"),
  FieldAttributes.Private);

// Létrehozza az egyedi konstruktort.
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
  helloWorldClass.DefineConstructor(MethodAttributes.Public,
  CallingConventions.Standard,
  constructorArgs);
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
ConstructorInfo superConstructor =
  objectClass.GetConstructor(new Type[0]);
constructorIL.Emit(OpCodes.Call, superConstructor);
constructorIL.Emit(OpCodes.Ldarg_0);
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField);
constructorIL.Emit(OpCodes.Ret);

// Létrehozza az alapértelmezett konstruktort.
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);

// Most létrehozza a GetMsg() metódust.
MethodBuilder getMsgMethod =
  helloWorldClass.DefineMethod("GetMsg", MethodAttributes.Public,
  typeof(string), null);
ILGenerator methodIL = getMsgMethod.GetILGenerator();
methodIL.Emit(OpCodes.Ldarg_0);
methodIL.Emit(OpCodes.Ldfld, msgField);
methodIL.Emit(OpCodes.Ret);

// Létrehozza a SayHello metódust.
MethodBuilder sayHiMethod =
  helloWorldClass.DefineMethod("SayHello",
  MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();
methodIL.EmitWriteLine("Hello from the HelloWorld class!");
methodIL.Emit(OpCodes.Ret);
```

```
    // "Megsüti" a HelloWorld osztályt.
    // (A sütés a hivatalos elnevezés a típus beszúrásra.)
    helloWorldClass.CreateType();

    // (Opcionálisan) kimenti a szerelvényt egy fájlba.
    assembly.Save("MyAssembly.dll");
}
```

A szerelvény- és a modulkészlet generálása

A metódustörzs a szerelvény legfontosabb jellemzőinek definiálásával kezdődik, az `AssemblyName` és `Version` típus felhasználásával (ezek a `System.Reflection` névtérben vannak definiálva). Ezután megkapunk egy `AssemblyBuilder` típust a példányszintű `AppDomain.DefineDynamicAssembly()` metóduson keresztül (a hívó egy `AppDomain` referenciát ad át a `CreateMyAsm()` metódusnak):

```
// Általános szerelvénykarakterisztikákat létesít,
// és kialakítja a kapcsolatot az AssemblyBuilder típushoz.
public static void CreateMyAsm(AppDomain curAppDomain)
{
  AssemblyName assemblyName = new AssemblyName();
  assemblyName.Name = "MyAssembly";
  assemblyName.Version = new Version("1.0.0.0");

  // Új szerelvény létrehozása az aktuális alkalmazástartományban.
  AssemblyBuilder assembly =
    curAppDomain.DefineDynamicAssembly(assemblyName,
  AssemblyBuilderAccess.Save);
  ...
}
```

Amikor meghívjuk az `AppDomain.DefineDynamicAssembly()`-t, meg kell határoznunk a definiálandó szerelvény elérésének a módját, amely a 19.10. táblázatban bemutatott értékek bármelyike lehet.

Érték	Valós jelentése
`ReflectionOnly`	Azt reprezentálja, hogy egy dinamikus szerelvény csak reflexióval használható.
`Run`	Azt reprezentálja, hogy egy dinamikus szerelvény végrehajtható a memóriában, de nem menthető lemezre.
`RunAndSave`	Azt reprezentálja, hogy egy dinamikus szerelvény végrehajtható a memóriában, és menthető lemezre.

Érték	Valós jelentése
Save	Azt reprezentálja, hogy egy dinamikus szerelvény menthető lemezre, de nem hajtható végre a memóriában.

19.10. táblázat: *Az AssemblyBuilderAccess felsorolt típus lehetséges értékei*

A következő feladat a modulok definiálása lesz az új szerelvényhez. Mivel a szerelvény egyetlen fájl, csak egyetlen modult kell definiálnunk. Ha többfájlos szerelvényt hoznánk létre a `DefineDynamicModule()` metódussal, meghatároznánk egy opcionális második paramétert, amely egy adott modul nevét (pl. `myMod.dotnetmodule`) jelképezi. Amikor egyfájlos szerelvényt hozunk létre, a modul neve meg fog egyezni a szerelvény nevével. Mindenesetre, ha a `DefineDynamicModule()` metódus már visszatért, akkor rendelkezünk referenciával egy érvényes `ModuleBuilder` típushoz:

```
// Az egyfájlos szerelvény.
ModuleBuilder module =
  assembly.DefineDynamicModule("MyAssembly", "MyAssembly.dll");
```

A ModuleBuilder típus szerepe

A `ModuleBuilder` kulcsfontosságú típus a dinamikus szerelvények fejlesztésében. A `ModuleBuilder` sok tagot támogat, amelyek lehetővé teszik egy adott modulban (osztály, interfész, struktúra stb.) tárolt típusok, valamint az abban tárolt beágyazott erőforrások (táblázat, képek stb.) definiálását. A 19.11 táblázat néhány létrehozásáért felelős metódust mutat be. (Minden metódus visszaad egy hozzátartozó típust, amely a készíteni kívánt típust jelképezi.)

Metódus	Valós jelentése
`DefineEnum()`	Egy .NET-felsorolás definícióját generálja.
`DefineResource()`	Ebben a modulban irányítottan beágyazott erőforrást definiál.
`DefineType()`	Létrehoz egy `TypeBuildert`, amely lehetővé teszi értéktípusok, interfészek és osztálytípusok definiálását (beleértve a metódusreferenciákat).

19.11. táblázat: *A ModuleBuilder típus tagjai*

A `ModuleBuilder` osztály kulcsfontosságú tagja a `DefineType()`. A típus nevének (egy egyszerű sztringgel történő) meghatározásán túl a `System.Reflection.TypeAttributes` felsorolt típust is használjuk a típus formátumának leírásához. A 19.12. táblázat mutatja a `TypeAttributes` néhány (de nem az összes) kulcstagját.

Tag	Valós jelentése
`Abstract`	Azt határozza meg, hogy a típus absztrakt.
`Class`	Azt határozza meg, hogy a típus egy osztály.
`Interface`	Azt határozza meg, hogy a típus egy interfész.
`NestedAssembly`	Azt határozza meg, hogy az osztály úgy van beágyazva, hogy csak a szerelvényben látható, és így a szerelvényen belül csak metódusokon keresztül érhető el.
`NestedFamAndAssem`	Azt határozza meg, hogy az osztály úgy van beágyazva, hogy a szerelvényben és a családban is látható, és így a család és a szerelvény metszetében található metódusokkal érhető el.
`NestedFamily`	Azt határozza meg, hogy az osztály úgy van beágyazva, hogy csak a családban látható, és így csak a saját típusában és valamely altípusában található metódusokon keresztül érhető el.
`NestedFamORAssem`	Azt határozza meg, hogy az osztály úgy van beágyazva, hogy vagy a családban, vagy a szerelvényben látható, és így csak a család és a szerelvény uniójában található metódusokon keresztül érthető el.
`NestedPrivate`	Azt határozza meg, hogy az osztály privát láthatósággal van beágyazva.
`NestedPublic`	Azt határozza meg, hogy az osztály nyilvános láthatósággal van beágyazva.
`NotPublic`	Azt határozza meg, hogy az osztály nem nyilvános.
`Public`	Azt határozza meg, hogy az osztály nyilvános.
`Sealed`	Azt határozza meg, hogy az osztály nem kiterjeszthető.
`Serializable`	Azt határozza meg, hogy az osztály sorosítható.

19.12. táblázat: *A TypeAttributes tagjai*

A HelloClass típus és a sztringtípusú tagváltozó generálása

Az alábbiakban vizsgáljuk meg, hogy hogyan illeszthetjük be a nyilvános `HelloWorld` osztálytípust és a privát sztringváltozót:

```
// Egy nyilvános osztályt definiál "MyAssembly.HelloWorld" névvel.
TypeBuilder helloWorldClass =
  module.DefineType("MyAssembly.HelloWorld",
 TypeAttributes.Public);

// A "theMessage" nevű, String típusú privát tagváltozó definiálása.
FieldBuilder msgField =
  helloWorldClass.DefineField("theMessage",
  typeof(string),
  FieldAttributes.Private);
```

Figyeljük meg, hogyan teszi elérhetővé a `TypeBuilder.DefineField()` metódus a `FieldBuilder` típust. A `TypeBuilder` osztály más metódusokat is definiál, amelyek elérhetőséget biztosítanak más „builder" típusokhoz. A `DefineConstructor()` visszaad egy `ConstructorBuildert`, a `DefineProperty()` egy `PropertyBuildert` és így tovább.

A konstruktorok generálása

A `TypeBuilder.DefineConstructor()` metódus segítségével definiálhatunk egy konstruktort az aktuális típushoz. Amikor a `HelloClass` konstruktorának implementálásához érkezünk, szükségünk lesz nyers CIL-kód beszúrására a konstruktor törzsébe, amelynek feladata a bejövő paraméterek hozzárendelése a belső privát sztringekhez. Ahhoz, hogy `ILGenerator` típust kapjunk, meghívjuk a `GetILGenerator()` metódust a megfelelő „builder" típusból, amelyhez van referenciánk (a mi esetünkben a `ConstructorBuilder` típus).

Az `ILGenerator` osztály `Emit()` metódusa felelős a CIL elhelyezéséért a tag implementációjában. Maga az `Emit()` gyakran használja az `OpCodes` osztálytípust, amely az írásvédett mezőket használó CIL-vezérlőkódokat tárja fel. Az `OpCodes.Ret` például egy metódushívás végét, azaz a visszatérést jelzi. Az `OpCodes.Stfld` értéket rendel egy tagváltozóhoz. Az `OpCodes.Call` egy adott metódus (a mi esetünkben az ősosztálykonstruktor) hívásához használható. Ezután vizsgáljuk meg a következő konstruktorkódot:

```
//Létrehozza az egyedi konstruktort
// egyetlen System.String argumentum használatával.
Type[] constructorArgs = new Type[1];
constructorArgs[0] = typeof(string);
ConstructorBuilder constructor =
  helloWorldClass.DefineConstructor(MethodAttributes.Public,
  CallingConventions.Standard, constructorArgs);

// Most a szükséges CIL-t beillesztjük a konstruktorba.
ILGenerator constructorIL = constructor.GetILGenerator();
constructorIL.Emit(OpCodes.Ldarg_0);
Type objectClass = typeof(object);
ConstructorInfo superConstructor =
  objectClass.GetConstructor(new Type[0]);
constructorIL.Emit(OpCodes.Call, superConstructor);
                                    // Ősosztály konstruktort hív.

// Betölti az objektum "this" mutatóját a verembe.
constructorIL.Emit(OpCodes.Ldarg_0);

// Bejövő paramétert tölt be a virtuális verembe, és eltárolja
// az msgField-ben.
constructorIL.Emit(OpCodes.Ldarg_1);
constructorIL.Emit(OpCodes.Stfld, msgField); // msgField értékadás.
constructorIL.Emit(OpCodes.Ret);             // Visszatér.
```

Amint definiáltuk a típushoz az egyedi konstruktort, az alapértelmezett konstruktor eltűnt. Az argumentum nélküli konstruktor újradefiniálásához egyszerűen hívjuk meg a `TypeBuilder` típus `DefineDefaultConstructor()` metódusát az alábbiak szerint:

```
// Újra beilleszti az alapértelmezett konstruktort.
helloWorldClass.DefineDefaultConstructor(MethodAttributes.Public);
```

Ez az egyszerű hívás egy szabványos CIL-kódot generál, amelyet az alapértelmezett konstruktor definiálásához használunk:

```
.method public hidebysig specialname rtspecialname
  instance void .ctor() cil managed
{
  .maxstack 1
  ldarg.0
  call instance void [mscorlib]System.Object::.ctor()
  ret
}
```

A SayHello() metódus generálása

Végül, de nem utolsósorban, vizsgáljuk meg a `SayHello` metódus kigenerálásának a folyamatát. Az első feladat a `MethodBuilder` típus létrehozása a `helloWorldClass` változóból. Amint ez megvan, definiáljuk a metódust, és megkapjuk a mögöttes `ILGenerator`-t a CIL-instrukciók generálásához:

```
// Létrehozza a SayHello metódust.
MethodBuilder sayHiMethod =
  helloWorldClass.DefineMethod("SayHello",
  MethodAttributes.Public, null, null);
methodIL = sayHiMethod.GetILGenerator();

// Kiír egy sort a konzolra.
methodIL.EmitWriteLine("Hello there!");
methodIL.Emit(OpCodes.Ret);
```

Ekkor egy nyilvános metódust (`MethodAttributes.Public`) hoztunk létre, amely nem vesz fel paramétereket, és nincs visszatérési értéke (a `DefineMethod()` hívásakor `null` értékeket adtunk át). Figyeljük meg az `EmitWriteLine()` hívást. Az `ILGenerator` osztály ezen segédtagja automatikusan kiír egy sort a szabványos kimenetre különösebb gond nélkül.

Dinamikusan létrehozott szerelvény használata

Az egyetlen, amire még szükségünk van, az egy olyan osztály, amely elindítja a kódot. Tegyük fel, hogy az aktuális projektünk egy második osztályt defniál `AsmReader` névvel. A programkód a `Main()`-ben az aktuális `AppDomaint` a `Thread.GetDoMain()` metóduson keresztül éri el, amelyet majd a dinamikusan létrehozott szerelvény hosztolásához fogunk használni. Ha van referenciánk, meghívhatjuk a `CreateMyAsm()` metódust.

Kicsit érdekesebbé tehetjük a dolgot, ha a `CreatMyAsm()` hívásának visszatérésekor késői kötéseket alkalmazunk (lásd a 16. fejezetben) az újonnan létrehozott szerelvény memóriába töltéséhez és a HelloWorld osztály tagjaival való kapcsolatba lépéshez. Módosítsuk `Main()` metódusunkat a következők szerint:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Amazing Dynamic Assembly Builder App
    *****");
  //Megkapja az alkalmazástartományt az aktuális szálból.
  AppDomain curAppDomain = Thread.GetDomain();
```

```
    // Létrehozza a dinamikus szerelvényt a helper f(x) segítségével.
    CreateMyAsm(curAppDomain);
    Console.WriteLine("-> Finished creating MyAssembly.dll.");

    // Most betölti az új szerelvényt a fájlból.
    Console.WriteLine("-> Loading MyAssembly.dll from file.");
    Assembly a = Assembly.Load("MyAssembly");

    // Megkapja a HelloWorld típust.
    Type hello = a.GetType("MyAssembly.HelloWorld");

    // HelloWorld objektumot hoz létre, és hívja a megfelelő
    // konstruktort.
    Console.Write("-> Enter message to pass HelloWorld class: ");
    string msg = Console.ReadLine();
    object[] ctorArgs = new object[1];
    ctorArgs[0] = msg;
    object obj = Activator.CreateInstance(hello, ctorArgs);

    // Hívja a SayHellót, és megjeleníti a kapott sztringet.
    Console.WriteLine("-> Calling SayHello() via late binding.");
    MethodInfo mi = hello.GetMethod("SayHello");
    mi.Invoke(obj, null);

    // Metódust hív.
    mi = hello.GetMethod("GetMsg");
    Console.WriteLine(mi.Invoke(obj, null));
}
```

Valójában csak egy olyan .NET-szerelvényt hoztunk létre, amely képes létrehozni és végrehajtani .NET-szerelvényeket futásidőben. Ezzel be is fejeztük a CIL és a dinamikus szerelvények szerepének vizsgálatát.

Megjegyzés Nyugodtan töltsük be dinamikusan létrehozott szerelvényeinket az `ildasm.exe`-be, hogy a nyers CIL-kód és a `System.Reflection.Emit` névtérbeli funkcionalitás közötti pontokat összekapcsoljuk.

Forráskód A `DynamicAsmBuilder` projekt megtalálható a 19. fejezet alkönyvtárában.

Összefoglalás

Ebben a fejezetben áttekintettük a CIL szintaxisát és szemantikáját. A magas szinten támogatott nyelvekkel ellentétben, mint például a C#, a CIL nem egyszerűen definiálja a kulcsszavakat, hanem direktívákkal (a szerelvény és típusstruktúrájának meghatározásához), attribútumokkal (amelyek az adott direktíva további leírásai) és vezérlőkódokkal (a típustagok megvalósításához) is ellátja azokat. Betekintést kaptunk a CIL-fordító (`ilasm.exe`) működésébe, és megnéztük, hogyan módosíthatjuk egy .NET-szerelvény összetételét új CIL-kóddal, valamint megismerkedtünk a nyers CIL segítségével létrehozott .NET-szerelvények készítési folyamatának alapjaival is.

A fejezet későbbi részében megismerkedtünk a `System.Reflection.Emit` névtérrel. Az ilyen jellegű típusok használatával lehetőségünk van menet közben egy .NET-szerelvényt a memóriába generálni. Lementhetjük ezt a memóriában tárolt binárist egy fizikai fájlba is. A `System.Reflection.Emit` sok típusa automatikusan hozza létre a helyes CIL-direktívákat és -attribútumokat, olyan barátságos típusokat használva, mint a `ConstructorBuilder`, `TypeBuilder` és így tovább. Az `ILGenerator` típust használhatjuk arra is, hogy a szükséges CIL-vezérlőkódokat beültessük egy adott tagba. Míg sok segédtípus áll a rendelkezésünkre, amelyek segítik a programozás folyamatát a CIL-vezérlőkódokkal, a dinamikus szerelvényekkel való programozáshoz ismernünk kell a köztes nyelvet.

Tárgymutató

B

C

CS

D

E,É

F

G

GY

H

I,Í

J

K

L

M

N

NY

O,Ó

Ö,Ő

P

Q

R

S

SZ

T

U,Ú

Ü

V

W

X

Y

Z

A szakmai lektorról

GAVIN SMYTH hivatásos szoftvermérnök, aki években mérve sokkal több fejlesztői tapasztalattal rendelkezik, mint amennyit hajlandó bevallani. Eszközillesztőktől kezdve az elosztott webes alkalmazások fejlesztésével foglalkozó projekteken át dolgozott sokféle platformon - például a 8 bites „nyers vas" rendszereken, beágyazott valós idejű operációs rendszereken, Unix és Windows rendszereken -, valamint az assembler, a C++, az Ada és a C# nyelvek mellett más nyelveken is fejlesztett. Olyan ügyfeleknek dolgozott, mint a BT és a Nortel. Jelenleg a Microsoft alkalmazottja. Gavin ugyan publikált néhány tanulmányt (*EXE*, where are you now?), de úgy véli, hogy mások munkájának kritizálása, értékelése sokkal nagyobb kihívást jelentő feladat. Amikor nem a gyomnövényekkel és a hangyákkal harcol a kertjében, Gavin Lego-robotokat próbál rávenni arra, hogy engedelmeskedjenek akaratának (kizárólag a gyerekek kedvéért - őszintén).

A szerzőről

ANDREW TROELSEN a Microsoft MVP (Visual C#) cím tulajdonosa, partner, oktató, valamint az Intertech Training (http://www.Intertech.com) .NET és J2EE fejlesztői oktatási központ tanácsadója. Számos könyvet írt, többek között a *Developer's Workshop to COM and ATL 3.0* (Wordware Publishing, 2000), a *COM and .NET Interoperability* (Apress, 2002), a *Visual Basic .NET and the .NET Platform: An Advanced Guide* (Apress, 2001) és a díjnyertes *C# and the .NET Platform* (Apress, 2003) című munkákat. Rengeteg cikke jelent meg a .NET for MSDN online, a DevX és a MacTech fórumain, és gyakori előadó a különböző .NET-konferenciákon és felhasználói csoportokban.

Andrew a Minnesota állambeli Minneapolisban él feleségével, Amandával. Szabadidejében türelmetlenül várja, hogy a Minnesota Wild megszerezze a Stanley-kupát, de feladta a reményt, hogy a Minnesota Vikings megnyeri a Super Bowlt, és komolyan hiszi, hogy a Minnesota Timberwolves nem kerül a rájátszásba, amíg a jelenlegi vezetés a helyén marad.

Felelős kiadó: a SZAK Kiadó Kft. ügyvezetője
Felelős szerkesztő: Kis Ádám
Korrektor: Laczkó Krisztina és Trepák Mónika
Tördelő: Mamira György
Borítóterv: Flórián Gábor (Typoézis Kft.)
Terjedelem 58 (B5) ív.
Készült az OOK Press Nyomdában (Veszprém)
Felelős vezető: Szathmáry Attila