```
class Program
{
  static void Main(string[] args)
  {
    SomeType t = new SomeType();
    t.SomeEvent += delegate (optionallySpecifiedDelegateArgs)
    { /* utasítások */ };
  }
}
```

Az előző `Main()` metódusban, amikor az első `AboutToBlow` eseményt kezeltük, nem adtuk meg a metódusreferenciából átadott argumentumokat:

```
c1.AboutToBlow += delegate {
  Console.WriteLine("Eek! Going too fast!");
};
```

Szigorúan véve nem szükséges átvenni az adott esemény által küldött argumentumokat. Ha azonban használni szeretnénk a bejövő argumentumokat, meg kell adnunk a paraméterek prototípusát a metódusreferenciában definiáltaknak megfelelően (ahogyan az `AboutToBlow` és `Exploded` események második kezelésében látható). Például:

```
c1.AboutToBlow += delegate(object sender, CarEventArgs e) {
  Console.WriteLine("Critical Message from Car: {0}", e.msg);
};
```

„Külső" változók elérése

A névtelen metódusok képesek elérni az őket definiáló metódus lokális változóit is. Formálisan ezeket a névtelen metódus *külső változóinak* hívjuk.

Megjegyzés A névtelen metódus nem éri el a definiáló metódus `ref` vagy `out` paramétereit.

Tegyük fel, hogy a `Main()` metódusban definiáltuk az `aboutToBlowCounter` nevű egész típusú lokális változót. Azon a névtelen metóduson belül, amelyik az `AboutToBlow` eseményt kezeli, eggyel növelni fogjuk a számláló értékét, és ellenőrzésként ki is íratjuk ezt a `Main()` befejezése előtt:

```
static void Main(string[] args)
{
  ...
  int aboutToBlowCounter = 0;

  // Hozzuk létre a Car típust szokásos módon.
  Car c1 = new Car(„SlugBug", 100, 10);

  // Regisztráljuk az eseménykezelőket névtelen metódusként.
  c1.AboutToBlow += delegate
  {
    aboutToBlowCounter++;
    Console.WriteLine(„Eek! Going too fast!");
  };

  c1.AboutToBlow += delegate(string msg)
  {
    aboutToBlowCounter++;
    Console.WriteLine(„Critical Message from Car: {0}", msg);
  };
  ...
  Console.WriteLine(„AboutToBlow event was fired {0} times.",
    aboutToBlowCounter);
  Console.ReadLine();
}
```

Ha lefuttatjuk ezt a módosított `Main()` metódust, akkor a végén a `Console.WriteLine()` azt írja ki, hogy az `AboutToBlow` esemény kétszer sült el.

Forráskód Az `AnonymousMethods` projekt a 11. fejezet alkönyvtárában.

A metóduscsoport átalakítása

A C# egy másik metódusreferencia- és eseménycentrikus sajátossága a *metóduscsoport-átalakítás*. Ez lehetővé teszi, hogy az eseménykezelőnek „egyszerűen" csak a nevét regisztráljuk (ezt a szintaxist már használtuk ebben a fejezetben a `GenericDelegate` példában). Ennek bemutatására nézzük meg ismét a `SimpleMath` típust, amelyet a `ComputationFinished` új eseménnyel egészítünk ki:

```
public class SimpleMath
{
  // Nem foglalkozunk a System.EventArgs osztályból
  // származó típus létrehozásával.
  public delegate void MathMessage(string msg);
  public event MathMessage ComputationFinished;
```

```
  public int Add(int x, int y)
  {
    ComputationFinished(„Adding complete.");
    return x + y;
  }

  public int Subtract(int x, int y)
  {
    ComputationFinished(„Subtracting complete.");
    return x - y;
  }
}
```

Ha nem használunk névtelenmetódus-szintaxist, akkor a következő módon alkalmazzuk a `ComputationComplete` eseményt:

```
class Program
{
  static void Main(string[] args)
  {
    SimpleMath m = new SimpleMath();
    m.ComputationFinished +=
      new SimpleMath.MathMessage(ComputationFinishedHandler);
    Console.WriteLine(„10 + 10 is {0}", m.Add(10, 10));
    Console.ReadLine();
  }
  static void ComputationFinishedHandler(string msg)
  { Console.WriteLine(msg); }
}
```

Az eseménykezelőt speciális eseménnyel azonban a következő módon is regisztrálhatjuk (a kód többi része változatlan):

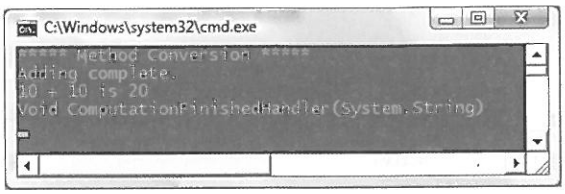
```
m.ComputationFinished += ComputationFinishedHandler;
```

Nem közvetlenül foglaljuk le a hozzárendelt metódusreferencia-objektumot, hanem egyszerűen egy olyan metódust adunk meg, amelyik megfelel a metódusreferencia várt szignatúrájának (ebben az esetben `void` visszatérési értékű és egyetlen `System.String` típusú paraméterrel). Tudjuk, hogy a C#-fordító mindig figyel a típusbiztonságra. Ennek megfelelően, ha a `ComputationFinishedHandler()` metódus nem void visszatérési értékű lenne `System.String` bemenő paraméterrel, akkor fordítási hibát kapnánk.

Explicit módon is át lehet alakítani egy eseménykezelőt a kapcsolódó metódusreferencia egy példányává. Ez akkor lehet hasznos, ha szükségünk van az alap-metódusreferencia megszerzésére, hogy a `System.MulticastDelegate` örökölt tagjaival kommunikáljon. Például:

```
// Eseménykezelők, amelyeket át kell alakítani
// az alapul szolgáló metódusreferenciára.
SimpleMath.MathMessage mmDelegate =
  (SimpleMath.MathMessage)ComputationFinishedHandler;
Console.WriteLine(mmDelegate.Method);
```

Ha lefuttatjuk ezt a kódot, a `Consol.WriteLine()` a `ComputationFinishedHandler` szignatúráját adja ki, ahogyan az a 11.8. ábrán látható.



11.8. ábra: *Kivonatolhatjuk a metódusreferenciát a hozzátartozó eseménykezelőből*

Forráskód Az `MethodGroupConversion` projekt megtalálható a 11. fejezet alkönyvtárában.

A C# 2008 lambda operátora

A .NET-eseménykezelő architektúrájához tartozik a C# 2008 *lambda kifejezése*. A C# támogatja az események „inline" feldolgozását azzal, hogy egy eseményhez direkt módon hozzárendel egy utasításokból álló kódblokkot (névtelen metódust használva), nem pedig egyedülálló metódust hív a hozzátartozó metódusreferencián keresztül. A lambda kifejezés valójában a névtelen metódusok létrehozásának tömörebb módja, amely nagymértékben leegyszerűsíti a munkát a .NET-metódusreferenciával.

Megjegyzés A C# 2008 megengedi a lambda *kifejezések* memóriaobjektumként való ábrázolását ún. kifejezésfák (expression tree) használatával. Ezt azok a külső fejlesztők értékelik, akiknek olyan szoftvert kell készíteni, amely a mások által készített lambda kifejezések funkcionalitását egészíti ki, valamint ha nyelvbe ágyazott lekérdezést (LINQ) programoznak. További információkért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

A lambda kifejezések vizsgálatához hozzunk létre egy `SimpleLambdaExpressions` nevű parancssorialkalmazás-projektet. Nézzük meg a generikus `List<T>` típus `FindAll()` metódusát. Ez a függvény `System.Predicate<T>` típusú generikus metódusreferenciát vár, ezzel becsomagol bármilyen olyan metódust, amelyik `Boolean` típust ad vissza, és a megadott `T` típust várja egyetlen bemenő paramé-

terként. Adjunk a Program osztályhoz egy olyan metódust (TraditionalDelegateSyntax() néven), amely a System.Predicate<T> típussal kommunikál azért, hogy kikeresse a List<T>-ben található egész számok közül a párosakat:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Lambdas *****\n");
    TraditionalDelegateSyntax();
    Console.ReadLine();
  }

  static void TraditionalDelegateSyntax()
  {
    // Hozzuk létre az egész számokat tartalmazó listát.
    List<int> list = new List<int>();
    list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

    // Hivjuk a FindAll() függvényt a hagyományos
    // metódusreferencia-szintaxissal.
    Predicate<int> callback = new Predicate<int>(IsEvenNumber);
    List<int> evenNumbers = list.FindAll(callback);

    Console.WriteLine("Here are your even numbers:");
    foreach (int evenNumber in evenNumbers)
    {
      Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
  }

  // A Predicate<> metódusreferencia célja.
  static bool IsEvenNumber(int i)
  {
    // Páros szám?
    return (i % 2) == 0;
  }
}
```

Az IsEvenNumber() metódus ellenőrzi, hogy a bemeneti egészszám-paraméter páros vagy páratlan, és ehhez a C# modulo operátorát (%) használja. Ha lefuttatjuk az alkalmazást, akkor a 20, 4, 8 és 44 számokat kapjuk a parancssorra kiírva.

Bár a metódusreferencia hagyományos megközelítése az elvárt módon működik, az IsEvenNumber() függvényt csak nagyon korlátozott körülmények között hívhatjuk; pontosan csak akkor, ha a FindAll() metódust hívjuk, ehhez pedig teljes metódusdefiníció szükséges. Ha ehelyett névtelen metódust használnánk, a kódunk jelentősen leegyszerűsödne. Nézzük a következő új metódust:

```
static void AnonymousMethodSyntax()
{
  // Hozzuk létre az egész számokat tartalmazó listát.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Most használjunk névtelen metódust.
  List<int> evenNumbers = list.FindAll(delegate(int i)
    { return (i % 2) == 0; } );

  Console.WriteLine("Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write("{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

Ebben az esetben, ahelyett, hogy a `Predicate<T>` metódusreferencia-típust direkt módon hoznánk létre, majd egyedülálló metódust készítenénk, beépíthetünk egy névtelen metódust. Ekkor még mindig szükség van a `delegate` kulcsszóra (vagy egy erősen tipizált `Predicate<T>` metódusreferenciára), és meg kell bizonyosodnunk arról, hogy a paraméterlista pontosan egyezik-e. A névtelen metódus definiálására használt szintaxis kissé nehezen átlátható; ez még nyilvánvalóbb az alábbiakban:

```
List<int> evenNumbers = list.FindAll(
  delegate(int i)
  {
    return (i % 2) == 0;
  }
);
```

A *lambda kifejezéseket* használhatjuk a `FindAll()` még egyszerűbb hívására. Ha az új szintaxist alkalmazzuk, akkor nyoma sincs az alapul szolgáló metódusreferenciának. Végezzük el a következő módosítást a kódunkon:

```
static void LambdaExpressionSyntax()
{
  // Hozzuk létre az egész számokat tartalmazó listát.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Most használjuk a C# 2008 lambda kifejezést.
  List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
  Console.WriteLine("Here are your even numbers:");
```

```
    foreach (int evenNumber in evenNumbers)
    {
      Console.Write("{0}\t", evenNumber);
    }
    Console.WriteLine();
}
```

A `FindAll()` metódusnak átadott, meglehetősen furcsa kódkifejezés valójában lambda kifejezés. A példának ebben az iterációjában semmi nyoma nincs a `Predicate<T>` metódusreferenciának (vagy a `delegate` kulcsszónak). Csak a lambda kifejezést definiáltuk: `i => (i % 2) == 0`.

Lambda kifejezést bárhol használhatunk, ahol névtelen metódust vagy erősen tipizált metódusreferenciát használnánk (általában sokkal kevesebb gépeléssel). A háttérben a C#-fordító átalakítja a kifejezést szabványos névtelen metódussá a `Predicate<T>` metódusreferencia-típus felhasználásával (ezt leellenőrizhetjük az `ildasm.exe` vagy a `reflector.exe` használatával). Az alábbi kódutasítást:

```
// Ez a lambda kifejezés...
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

a fordító közelítőleg a következő C#-kóddá alakítja:

```
// ... lesz a névtelen metódus.
List<int> evenNumbers = list.FindAll(delegate (int i)
{
  return (i % 2) == 0;
});
```

A lambda kifejezések részletezése

A lambda kifejezés létrehozásakor először egy paraméterlistát definiálunk, amelyet a => token követ (ez a C# nyelv tokenje, amely azt a szerepet látja el, mint a lambda operátor a *lambda-kalkulusban*), ezt utasítások olyan csoportja (vagy egyetlen olyan utasítás) követi, amely feldolgozza ezeket az argumentumokat. A lambda kifejezés nagyon magas szintről a következő módon értelmezhető:

```
bemenő paraméterek => feldolgozó utasítások
```

A `LambdaExpressionSyntax()` metódusunkban ez a következőképpen részleteződik:

```
// "i" a paraméterlista.
// "((i % 2) == 0" a parancsunk "i" feldolgozására.
List<int> evenNumbers = list.FindAll(i => (i % 2) == 0);
```

A lambda kifejezés paramétereit expliciten vagy impliciten lehet tipizálni. Az `i` paramétert kifejező alapadattípus (egész) a fenti példában implicit módon határozódott meg. A fordító az általános lambda kifejezés és a hozzátartozó metódusreferencia környezetének alapján képes megállapítani, hogy az `i` egész szám. Emellett lehetőség van a kifejezésben minden egyes paramétertípus explicit megadására is, a következőképpen, együttesen zárójelezve a változó nevét és típusát:

```
// Explicit módon megadjuk a paraméter típusát.
List<int> evenNumbers = list.FindAll((int i) => (i % 2) == 0);
```

Ha a lambda kifejezésnek egyetlen, implicit típusos paramétere van, akkor a paraméterlistából elhagyhatók a zárójelek. Ha következetesen szeretnénk használni a lambda-paramétereket, akkor *mindig* zárójelezhetjük a paraméterlistát, ez pedig a következő kifejezéshez vezet:

```
List<int> evenNumbers = list.FindAll((i) => (i % 2) == 0);
```

Maga a kifejezés nincsen zárójelben (a modulo kifejezést természetesen zárójeleztük, hogy biztosan az egyenlőségvizsgálat előtt történjen meg a végrehajtása). A lambda kifejezés az alábbiak szerint megengedi a parancsok zárójelezését:

```
// Most zárójelezzük a kifejezést is.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Kérdés, hogy hogyan olvashatjuk a lambda-utasításokat a lehető legegyszerűbben? Mellőzve a tiszta matematikát, használjuk a következő magyarázatot:

```
// A paraméterlistánkat (ebben az esetben egyetlen,
// i nevű egész szám) a (i % 2) == 0 kifejezés dolgozza fel.
List<int> evenNumbers = list.FindAll((i) => ((i % 2) == 0));
```

Argumentumok feldolgozása több utasításon keresztül

Az első lambda kifejezésünk egyetlen utasításból állt, amely végül egy `boolean` típust eredményezett. Persze sok metódusreferencia-cél több utasításból épül fel. Ehhez a C# 2008-ban többszörös utasításblokkok használatával lehet

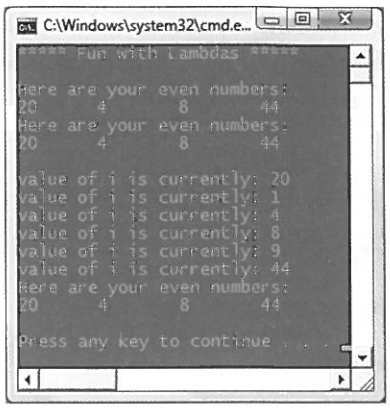
lambda kifejezéseket létrehozni. Ha a kifejezésünknek a paramétereket több sor kóddal kellene feldolgoznia, akkor ezt a hatókör megadásával, a szokásos kapcsos zárójelekkel tehetjük meg. Vegyük a `LambdaExpressionSyntax()` metódusunk következő módosítását:

```
static void LambdaExpressionSyntax()
{
  // Hozzuk létre az egész számokat tartalmazó listát.
  List<int> list = new List<int>();
  list.AddRange(new int[] { 20, 1, 4, 8, 9, 44 });

  // Most dolgozzuk fel az összes argumentumot
  // utasítások csoportján belül.
  List<int> evenNumbers = list.FindAll((i) =>
  {
    Console.WriteLine(„value of i is currently: {0}", i);
    bool isEven = ((i % 2) == 0);
    return isEven;
  });

  Console.WriteLine(„Here are your even numbers:");
  foreach (int evenNumber in evenNumbers)
  {
    Console.Write(„{0}\t", evenNumber);
  }
  Console.WriteLine();
}
```

Ebben az esetben a paraméterlistánkat (megint egyetlen i nevű egész szám) több utasítás együttese dolgozza fel. A `Console.WriteLine()` hívásán kívül a modulo utasítást is két utasítássá alakítottuk a jobb olvashatóság kedvéért. Tegyük fel, hogy mindhárom metódust a `Main()` metódusból hívjuk:

```
static void Main(string[] args)
{
  Console.WriteLine(„***** Fun with Lambdas *****\n");
  TraditionalDelegateSyntax();
  AnonymousMethodSyntax();
  Console.WriteLine();
  LambdaExpressionSyntax();
  Console.ReadLine();
}
```

A 11.9. ábrán látható a kimenet.



11.9. ábra: *Az első lambda kifejezésünk kimenete*

Forráskód A `SimpleLambdaExpressions` projekt megtalálható a 11. fejezet alkönyvtárában.

A CarDelegate példa áttervezése lambda kifejezések használatával

A lambda kifejezések célja, hogy tisztán és tömören adhassuk meg a névtelen metódusokat (és ennek megfelelően indirekt módon egyszerűsítsük a metódusreferenciák használatát). Tervezzük át a már korábban létrehozott `CarDelegate` projektet. Az alábbiakban a projekt `Program` osztályának egyszerűsített verziója látható, amely a „normális" metódusreferenciát használja a visszahívások megválaszolására:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** More Fun with Lambdas *****\n");

    // Hozzunk létre autót a szokásos módon.
    Car c1 = new Car("SlugBug", 100, 10);

    // A normális metódusreferencia-szintaxis.
    c1.OnAboutToBlow(new Car.AboutToBlow(CarAboutToBlow));
    c1.OnExploded(new Car.Exploded(CarExploded));

    // Gyorsítás (ez fogja létrehozni az eseményeket).
    Console.WriteLine("\n***** Speeding up *****");
```

```
    for (int i = 0; i < 6; i++)
      c1.SpeedUp(20);
    Console.ReadLine();
  }

  // A metódusreferencia-célok.
  public static void CarAboutToBlow(string msg)
  { Console.WriteLine(msg); }

  public static void CarExploded(string msg)
  { Console.WriteLine(msg); }
}
```

A `Main()` metódusban újra használjuk a névtelen metódusokat:

```
static void Main(string[] args)
{
  Console.WriteLine("***** More Fun with Lambdas *****\n");

  // Hozzunk létre autót a szokásos módon.
  Car c1 = new Car("SlugBug", 100, 10);

  // Használjuk a névtelen metódusokat.
  c1.OnAboutToBlow(delegate(string msg)
    { Console.WriteLine(msg); });
  c1.OnExploded(delegate(string msg) { Console.WriteLine(msg); });

  // Gyorsítás (ez fogja létrehozni az eseményeket).
  Console.WriteLine("\n***** Speeding up *****");
  for (int i = 0; i < 6; i++)
    c1.SpeedUp(20);

  Console.ReadLine();
}
```

És végül a `Main()` metódus lambda kifejezések használatával a következő:

```
static void Main(string[] args)
{
  Console.WriteLine("***** More Fun with Lambdas *****\n");

  // Hozzuk létre a Car típust szokásos módon.
  Car c1 = new Car("SlugBug", 100, 10);

  // Most Lambda kifejezésekkel!
  c1.OnAboutToBlow(msg => { Console.WriteLine(msg); });
  c1.OnExploded(msg => { Console.WriteLine(msg); });
```

```
    // Gyorsítás (ez fogja létrehozni az eseményeket).
    Console.WriteLine("\n***** Speeding up *****");
    for (int i = 0; i < 6; i++)
      c1.SpeedUp(20);

    Console.ReadLine();
  }
```

Forráskód A `CarDelegateWithLambdas` projekt megtalálható a 11. fejezet alkönyvtárában.

Lambda kifejezések több paraméterrel (vagy paraméterek nélkül)

Az eddig látott lambda kifejezések mindegyike egyetlen paramétert dolgozott fel. Ez azonban nem követelmény, ugyanis egy lambda kifejezés több argumentumot is feldolgozhat, vagy akár paraméter nélküli is lehet. Az első lehetőség bemutatására hozzunk létre egy `LambdaExpressionsMultipleParams` nevű parancssorialkalmazás-projektet. Vegyük a `SimpleMath` típus következő megvalósítását:

```
public class SimpleMath
{
  public delegate void MathMessage(string msg, int result);
  private MathMessage mmDelegate;

  public void SetMathHandler(MathMessage target)
  {mmDelegate = target; }

  public void Add(int x, int y)
  {
    if (mmDelegate != null)
      mmDelegate.Invoke(„Adding has completed!", x + y);
  }
}
```

A `MathMessage` metódusreferencia két paramétert vár. Ennek lambda kifejezéssel történő hívása esetén a `Main()` metódus a következő lesz:

```
static void Main(string[] args)
{
  // A w/ metódusreferencia regisztrálása lambda kifejezésként.
  SimpleMath m = new SimpleMath();
  m.SetMathHandler((msg, result) =>
    {Console.WriteLine(„Message: {0}, Result: {1}", msg, result);});
```

```
    // Ez hajtja végre a lambda kifejezést.
    m.Add(10, 10);
    Console.ReadLine();
}
```

Itt használjuk ki a típuskövetkeztetést, hiszen a két paraméterünk az egyszerűség kedvéért nem erősen tipizált. A `SetMathHandler()` metódust azonban a következő módon is meghívhatnánk:

```
m.SetMathHandler((string msg, int result) =>
  {Console.WriteLine("Message: {0}, Result: {1}", msg, result);});
```

Végül, ha olyan lambda kifejezést használunk a metódusreferenciákkal való kommunikációra, amely nem vár paramétert, akkor ezt a kifejezést paraméterként üres zárójel megadásával tehetjük meg. Feltételezve, hogy az alábbi metódusreferencia-típust definiáltuk:

```
public delegate string VerySimpleDelegate();
```

a hívás eredményét a következő módon lehet kezelni:

```
// Kiírja az „Enjoy your string!" szöveget a parancssorra.
VerySimpleDelegate d = new VerySimpleDelegate( () =>
  {return "Enjoy your string!";} );
Console.WriteLine(d.Invoke());
```

Ezen a ponton remélhetőleg világossá vált a lambda kifejezések általános szerepe, és az, hogy hogyan segítik „funkcionális értelemben" a névtelen metódusok és metódusreferencia-típusok használatát. Bár az új lambda operátor (=>) nyilván egy kicsit szokatlan még, azt ne felejtsük el, hogy a lambda kifejezések erre a következő egyszerű egyenletre bonthatók:

```
bemenő paraméterek => feldolgozó utasítások
```

A LINQ programozási modell is sokat használja a lambda kifejezéseket a kódolás egyszerűbbé tételéhez. (A LINQ programozást lásd a 14. fejezet elején.)

Forráskód A `LambdaExpressionsMultipleParams` projekt megtalálható a 11. fejezet alkönyvtárában.

Összefoglalás

Ebben a fejezetben több lehetőséget is láttunk arra, hogy hogyan tud több objektum kétirányú párbeszédet folytatni. Először a C# `delegate` kulcsszavát vizsgáltuk meg, amely a `System.MulticastDelegate` osztályból indirekt módon hoz létre leszármazott osztályt. A metódusreferencia egyszerű objektumként olyan metódusok listáját kezeli, amelyeket meg tud hívni, ha arra kap utasítást. Ezek a hívások lehetnek szinkron hívások (az `Invoke()` metódussal) vagy aszinkron hívások (a `BeginInvoke()` és `EndInvoke()` metódusokkal). (A .NET metódusreferencia-típusainak aszinkron működését lásd a 18. fejezetben.)

Megvizsgáltuk a C# `event` kulcsszavának működését is, amely – a metódusreferencia-típusokkal együtt – az esemény bejelentését egyszerűsíti a várakozó hívó számára. Ahogyan azt az eredményről kapott CIL-kód révén megmutattuk, a .NET eseménykezelő rendszere a `System.Delegate/System.MulticastDelegate` típusok rejtett hívására épül. Így a C# `event` kulcsszava tisztán opcionális, amely csak gépelési időt takarít meg számunkra.

Megemlítettük a C# nyelv *névtelen metódus* lehetőségét is. Ennek a szintaktikai elemnek a használatával képesek vagyunk közvetlenül adott eseményekhez utasításblokkokat kötni. A névtelen metódusok figyelmen kívül hagyhatják az esemény által küldött paramétereket, és elérhetik a definiáló függvény „külső változóit" is. Megnéztük a metóduscsoport átalakításának használatával az események egyszerűbb regisztrálási módját is.

Végül a C# 2008 *lambda operátorának* (`=>`) alkalmazását néztük át. Ez a szintaxis nagyszerű alternatív jelölés azoknak a névtelen metódusoknak a létrehozásához, ahol több argumentumot adhatunk át feldolgozó utasítások egy csoportjának.

TIZENKETTEDIK FEJEZET

Indexelők, operátorok és mutatók

Ebben a fejezetben néhány fejlett szintaktikai szerkezet vizsgálatával bővítjük a C# programozási nyelvről szóló ismereteinket. Először megismerkedünk az *indexelő metódus* létrehozásával és használatával. Ez a C#-mechanizmus teszi lehetővé olyan egyedi típusok létrehozását, amelyek tömbszerű szintaxissal biztosítanak hozzáférést a belső típusokhoz. Ezután megvizsgáljuk a különböző operátorok (+, -, <, > stb.) túlterhelését és a típusok explicit vagy implicit konverziós rutinjainak létrehozását (valamint azt, hogy erre miért lehet szükség).

A fejezet többi részében a kevésbé gyakran használt (ám annál érdekesebb) C#-kulcsszavak egy kisebb csoportját tanulmányozzuk. Megtudjuk például, hogy hogyan hozhatunk létre „nem felügyelt" kódkörnyezetet azért, hogy közvetlenül tudjuk a mutatótípusokat kezelni és a különböző előfeldolgozó direktívákat használni.

Az indexelő metódus

Programozóként jól ismerhetjük a normál tömbökben tárolt egyes elemek elérésének módját az index operátor ([]) segítségével. Például:

```
static void Main(string[] args)
{
  // Végigiterálunk az induló bemeneti paramétereken.
  for(int i = 0; i < args.Length; i++)
    Console.WriteLine("Args: {0}", args[i]);

  // Lokális egészek tömbjének deklarációja.
  int[] myInts = { 10, 9, 100, 432, 9874};

  // Az egyes elemek elérése az [] operátor segítségével.
  for(int j = 0; j < myInts.Length; j++)
    Console.WriteLine("Index {0} = {1} ", j, myInts[j]);
  Console.ReadLine();
}
```

Az előző kód feltehetően semmiféle újdonságot nem jelent. Viszont a C# nyelvben lehetőségünk van olyan egyedi osztályok és struktúrák definiálására, amelyeket egy indexelő metódus definiálásával a normál tömbökhöz hasonlóan indexelhetünk. Ez a bizonyos nyelvi sajátosság az egyedi (generikus vagy nem generikus) gyűjteménytípusok létrehozásánál a leghasznosabb.

Mielőtt megvizsgálnánk az efféle szerkezetek megvalósítását, nézzünk meg egyet működés közben. Tegyük fel, hogy a 10. fejezetben létrehozott egyedi `PeopleCollection` típust (konkrétabban a `CustomNonGenericCollection` projektet) az indexelő metódus támogatásával bővítettük. Használatát a következő, `SimpleIndexer` nevű új parancssori alkalmazás mutatja:

```
// Az indexelők lehetővé teszik az elemek tömbszerű elérését.
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Indexers *****\n");

    PeopleCollection myPeople = new PeopleCollection();

    // Objektumok létrehozása indexelő szintaxissal.
    myPeople[0] = new Person("Homer", "Simpson", 40);
    myPeople[1] = new Person("Marge", "Simpson", 38);
    myPeople[2] = new Person("Lisa", "Simpson", 9);
    myPeople[3] = new Person("Bart", "Simpson", 7);
    myPeople[4] = new Person("Maggie", "Simpson", 2);

    // Most az indexelő segítségével előhívjuk és megjelenítjük
    // mindegyik elemet.
    for (int i = 0; i < myPeople.Count; i++)
    {
      Console.WriteLine("Person number: {0}", i);
      Console.WriteLine("Name: {0} {1}",
        myPeople[i].FirstName, myPeople[i].LastName);
      Console.WriteLine("Age: {0}", myPeople[i].Age);
      Console.WriteLine();
    }
  }
}
```

Az indexelők viselkedése nagyon hasonlít az `IEnumerator` és `IEnumerable` interfészeket támogató egyedi gyűjteményekéhez annyiban, hogy hozzáférést biztosítanak a konténerben lévő elemekhez. A legjelentősebb különbség természetesen az, hogy a tartalmak `foreach` szerkezettel való elérése helyett az objektumok belső gyűjteményét normál tömbhöz hasonlóan tudjuk kezelni.

A kérdés: hogyan alakítsuk ki a `PeopleCollection` osztályt (vagy bármely osztályt/struktúrát) úgy, hogy támogassa ezt a funkcionalitást? Az indexelő úgy jelenik meg, mint egy kissé eltorzult C#-tulajdonság. A legegyszerűbb formájában az indexelőt a `this[]` szintaxissal állítjuk elő. Így kell módosítanunk a `PeopleCollection` osztályt:

```
// Az indexelő hozzáadása a meglévő osztályhoz.
public class PeopleCollection : IEnumerable
{
  private ArrayList arPeople = new ArrayList();

  // Egyedi indexelő ehhez az osztályhoz.
  public Person this[int index]
  {
    get { return (Person)arPeople[index]; }
    set { arPeople.Insert(index, value); }
  }
  ...
}
```

A `this` kulcsszótól eltekintve az indexelő pontosan úgy néz ki, mint minden más C#-tulajdonság deklarációja. A `get` blokk szerepe például visszaadni a megfelelő objektumot a meghívónak. Itt ezt tulajdonképpen az `ArrayList` objektum indexelőjével tesszük meg. A `set` blokk feladata, a bejövő objektumot elhelyezni a tárolóban a megadott index helyére; a példánkban ezt az `ArrayList Insert()` metódusának meghívásával tesszük.

Az indexelők is a szintaktikai „ínyencségek" közé tartoznak, hiszen ezt a működést „normál" publikus metódusok, mint az `AddPerson()` vagy a `GetPerson()`, használatával is el tudnánk érni. Viszont ha az egyedi gyűjteménytípusainkon támogatjuk az indexelő metódusokat, akkor jól integrálhatók lesznek a .NET-alaposztálykönyvtárak szerkezetébe.

Bár az indexelő metódusok létrehozása elég gyakori egyedi gyűjtemények kialakításakor, a generikus típusoknál ez a funkcionalitás már eleve megtalálható. Nézzük meg a következő metódust, amely a `Person` objektumok `List<T>` általános listáját használja. Egyszerűen a `List<T>` indexelőjét is használhatjuk közvetlenül, például:

```
static void UseGenericListOfPeople()
{
  List<Person> myPeople = new List<Person>();
  myPeople.Add(new Person("Lisa", "Simpson", 9));
  myPeople.Add(new Person("Bart", "Simpson", 7));
```

```
    // Az első személy módosítása indexelő segítségével.
    myPeople[0] = new Person("Maggie", "Simpson", 2);

    // Most az indexelő segítségével előhívjuk és megjelenítjük
    // mindegyik elemet.
    for (int i = 0; i < myPeople.Count; i++)
    {
      Console.WriteLine("Person number: {0}", i);
      Console.WriteLine("Name: {0} {1}", myPeople[i].FirstName,
        myPeople[i].LastName);
      Console.WriteLine("Age: {0}", myPeople[i].Age);
      Console.WriteLine();
    }
  }
```

Forráskód A `SimpleIndexer` projekt a 12. fejezet alkönyvtárában található.

Objektumok indexelése sztringértékekkel

A jelenlegi `PeopleCollection` típusunk egy olyan indexelőt definiál, amely numerikus értékek segítségével teszi lehetővé a meghívónak az alelemek azonosítását. Tudnunk kell azonban, hogy ez nem követelménye az indexelő metódusnak. Tegyük fel, hogy a `Person` objektumokat inkább a `System.Collections.Generic.Dictionary<TKey, TValue>`, nem pedig az `ArrayList` használatával tárolnánk. Mivel a `ListDictionary` típusok sztringkulcsok (pl. a keresztnevek) alkalmazásával teszik lehetővé a tartalmazott típusok elérését, az indexelőt a következőképpen definiálhatnánk:

```
public class PeopleCollection : IEnumerable
{
  private Dictionary<string, Person> listPeople =
    new Dictionary<string, Person>();

  // Ez az indexelő sztring index alapján adja vissza a személyeket.
  public Person this[string name]
  {
    get { return (Person)listPeople[name]; }
    set { listPeople[name] = value; }
  }

  public void ClearPeople()
  { listPeople.Clear(); }

  public int Count
  { get { return listPeople.Count; } }
```

```
    IEnumerator IEnumerable.GetEnumerator()
    { return listPeople.GetEnumerator(); }
}
```

A meghívó ekkor már az itt bemutatott módon léphet interakcióba a `Person` objektummal:

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

  PeopleCollection myPeople = new PeopleCollection();

  myPeople["Homer"] = new Person("Homer", "Simpson", 40);
  myPeople["Marge"] = new Person("Marge", "Simpson", 38);

  // Visszakapjuk a "Homer" nevet, és kiíratjuk az adatot.
  Person homer = myPeople["Homer"];
  Console.WriteLine(homer.ToString());
  Console.ReadLine();
}
```

Ha tehát közvetlenül a generikus `Dictionary<TKey, TValue>` típust használnánk, az indexelő funkcionalitást készen kapnánk.

Forráskód A `StringIndexer` projekt megtalálható a 12. fejezet alkönyvtárában.

Az indexelő metódusok túlterhelése

Az indexelő metódusok túlterhelhetők. Így, ha lenne értelme engedélyezni, hogy a meghívó numerikus index *vagy* sztringérték segítségével is elérhesse az elemeket, többszörös indexelőt is definiálhatnánk az egyes típusokhoz. Ha például valaha már programoztunk az ADO.NET (a .NET saját adatbázis-hozzáférés API-ja) használatával, tudjuk, hogy a `DataSet` típus támogat egy `Tables` nevű tulajdonságot, amely egy erősen tipizált `DataTableCollection` típust ad vissza. A `DataTableCollection` három indexelőt definiál a `DataTable` objektumok lekérésére és beállítására; egyet sorszám alapján, a másik kettőt pedig egy barátságos sztringkulcs és egy opcionális tartalmazó táblanévtér alapján:

```
public sealed class DataTableCollection : InternalDataCollectionBase
{
  ...
  // Túlterhelt indexelők!
```

```
    public DataTable this[string name] { get; }
    public DataTable this[string name, string tableNamespace] { get; }
    public DataTable this[int index] { get; }
}
```

Igazság szerint az alaposztálykönyvtárakban elég sok típus támogat indexelő metódusokat. Még ha az aktuális projektünk nem is igényli egyedi indexelők létrehozását az osztályokhoz és a struktúrákhoz, vegyük figyelembe, hogy sok típus már támogatja ezt a szintaxist.

Az indexelő metódusok belső ábrázolása

A C# indexelő metódusok néhány változatának megismerése után nézzük meg az indexelők köztes nyelvi ábrázolását. Ha megnyitnánk az aktuális `PeopleCollection` típus indexelőjét, azt látnánk, hogy a fordító létrehozott egy `Item` nevű tulajdonságot, amely a helyes lekérdező/módosító metódusra képez le:

```
.property instance class StringIndexer.Person Item(string)
{
  .get instance class StringIndexer.Person
    StringIndexer.PeopleCollection::get_Item(string)
 .set instance void StringIndexer.PeopleCollection::set_Item(string,
    class StringIndexer.Person)
} // vége a PeopleCollection::Item elemnek
```

A `get_Item()` és `set_Item()` metódusok megvalósítása hasonlít bármely .NET-tulajdonságéhoz. Vizsgáljuk meg például a következő módosító logikát:

```
.method public hidebysig specialname instance void
  set_Item(string name,
    class StringIndexer.Person 'value') cil managed
{
  // Kódméret 16 (0x10)
  .maxstack 8
  IL_0000: nop
  IL_0001: ldarg.0
  IL_0002: ldfld class
    [System]System.Collections.Specialized.ListDictionary
    StringIndexer.PeopleCollection::listPeople
  IL_0007: ldarg.1
  IL_0008: ldarg.2
  IL_0009: callvirt instance void
  [System]System.Collections.Specialized.ListDictionary::Add(object,
    object)
```

```
  IL_000e: nop
  IL_000f: ret
} // a PeopleCollection::set_Item metódus vége
```

Megjegyzés A .NET Framework SDK 3.5 dokumentációjában `Item` tulajdonság néven találhatók meg az egyes osztályokhoz vagy struktúrákhoz tartozó indexelő metódusok. A Visual Studio Object Browser azonban az indexelő metódusokat az ismert `this[]` szintaxissal definiált tulajdonságként jelzi.

Többdimenziós indexelők

Arra is van lehetőség, hogy több paramétert fogadó indexelő metódust hozzunk létre. Tegyük fel, hogy egy olyan egyedi gyűjteményünk van, amely egy kétdimenziós tömbben tárolja az elemeket. Ekkor a következőképpen konfigurálhatjuk az indexelő metódust:

```
public class SomeContainer
{
  private int[,] my2DintArray = new int[10, 10];

  public int this[int row, int column]
  { /* érték módosítása vagy lekérése 2 dimenziós tömbben  */ }
}
```

Indexelő definíciók interfésztípusokon

Indexelők definiálhatók egy adott .NET-interfésztípusra is, így a támogató típusokhoz egyedi megvalósításokat lehet létrehozni. Ilyen a következő interfész is:

```
public interface IStringContainer
{
  // Ez az interfész indexelőt definiál, amely
  // numerikus index alapján sztringeket ad vissza.
  string this[int index] { get; set; }
}
```

Ezzel az interfészdefinícióval bármely osztály vagy struktúra, amely megvalósítja ezt az interfészt, feltétlenül támogat egy író/olvasó indexelőt, amely numerikus értékkel manipulálja az elemeket. Hasonlóképpen tervezhetnénk generikus interfészt is, ahol a típusindexelő lehetővé teszi a megvalósító számára, hogy eldöntse, mit használ az objektumok lekérésére vagy módosítására:

```
public interface IStringContainer<Key>
{
  string this[int Key] { get; set; }
}
```

Nézzünk egy megvalósítást numerikus index használatával:

```
class MyStrings : IStringContainer<int>
{
  string[] strings = { "First", "Second" };

  public string this[int Key]
  {
    get
    {
      return strings[Key];
    }
    set
    {
      strings[Key] = value;
    }
  }
}
```

Az operátor-túlterhelés

A C#, mint minden programozási nyelv, rendelkezik egy előre elkészített tokenkészlettel, amelyet a lényeges típusokon alapvető műveletek végrehajtására használunk. A + operátor például két egész számon alkalmazható, hogy egy egész számot kapjunk:

```
// A + operátor int típusokkal.
int a = 100;
int b = 240;
int c = a + b; // a c most 340
```

Ez megint csak nem nagy újdonság, de ugyanez a + operátor használható a legtöbb belső C#-adattípusra is. Nézzük például a következő kódot:

```
// A + operátor string típusokkal.
string s1 = "Hello";
string s2 = " world!";
string s3 = s1 + s2; // az s3 most "Hello world!"
```

Lényegében a + operátor a megadott adatok típusának (ebben az esetben egész számok és sztringek) megfelelően egyedi módon működik. Amikor a + operátort numerikus típusokra alkalmazzunk, az eredmény a két operandus összege lesz. Ha azonban a + operátort sztringtípusokra alkalmazzuk, eredményként az összefűzött sztringet kapjuk.

A C# nyelv lehetőséget biztosít egyedi osztályok és struktúrák építésére, amelyek szintén egyedi módon reagálnak ugyanarra az alapvető jelkészletre (mint pl. a + operátor). Nem minden belső C#-operátor terhelhető túl. A 12.1. táblázat összefoglalja a legfontosabb operátorok „túlterhelhetőségét".

C#-operátor	Túlterhelhetőség
`+, -, !, ~, ++,--, true, false`	Ezek az egyoperandusú operátorok túlterhelhetők.
`+,-,*, /, %, &, |, A, «, >>`	Ezek a kétoperandusú operátorok túlterhelhetők.
`==, ! =, <, >, <=, >=`	Az összehasonlító operátorok túlterhelhetők. A C# megköveteli, hogy a „hasonló" operátorok (vagyis < és >, <= és >=, == és !=) együtt legyenek túlterhelve.
`[]`	A [] operátor nem terhelhető túl. Az indexelő konstrukció azonban ugyanezt a funkcionalitást biztosítja.
`()`	A () operátor nem terhelhető túl. Az egyedi konverziós metódusok azonban ugyanezt a funkcionalitást biztosítják (lásd később).
`+=, -=, *=, /=, %=, &=, | =, ^=, <<=, >>=`	A rövidített értékadó operátorok nem terhelhetők túl; viszont a kapcsolódó bináris operátorok túlterhelésekor ezeket „automatikusan" megkapjuk.

12.1. táblázat: *A C#- operátorok túlterhelhetősége*

Megjegyzés A C# nyelvben a `true` és `false` használható operátorként is, nem csak literálként. Ez a szerep nagyon jól jöhet olyan egyedi típusok létrehozásakor, amelyek igaz, hamis és null (sem nem igaz, sem nem hamis) értékeket jelölnek.

Kétoperandusú operátorok túlterhelése

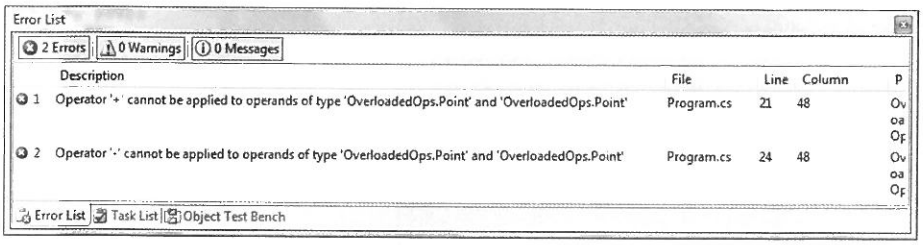
A kétoperandusú operátorok túlterhelésének bemutatására tegyük fel, hogy a következő egyszerű `Point` struktúrát definiáltuk egy `OverloadedOps` nevű új parancssori alkalmazásban:

```
// Egy egyszerű hétköznapi C#-struktúra
public struct Point
{
  private int x, y;
  public Point(int xPos, int yPos)
  {
    x = xPos;
    y = yPos;
  }

  public override string ToString()
  {
    return string.Format("[{0}, {1}]", this.x, this.y);
  }
}
```

A logikát tekintve értelmesnek tűnik a két `Point` összeadása. Ha például összeadnánk két `Point` változót, az eredmény egy új `Point` lenne, amely az `x` és az `y` értékek összegéből állna elő. Ugyancsak hasznos lehet egy `Point` kivonása is egy másikból. Ideális esetben a következő kódot szeretnénk megírni:

```
// Két pont összeadása és kivonása?
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Overloaded Operators *****\n");

  // Két pont létrehozása
  Point ptOne = new Point(100, 100);
  Point ptTwo = new Point(40, 40);
  Console.WriteLine("ptOne = {0}", ptOne);
  Console.WriteLine("ptTwo = {0}", ptTwo);

  // A pontok összeadása egy nagyobb pontot eredményez?
  Console.WriteLine("ptOne + ptTwo: {0} ", ptOne + ptTwo);

  // A pontok kivonása egy kisebb pontot eredményez?
  Console.WriteLine("ptOne - ptTwo: {0} ", ptOne - ptTwo);
  Console.ReadLine();
}
```

Viszont a `Point` jelenlegi állapotában fordítási idejű hibákat kapunk, ugyanis a `Point` típus nem tudja, hogyan reagáljon a `+` és `-` operátorokra (lásd 12.1. ábra).



12.1. ábra: *Az alapértelmezés szerint az egyedi osztályok/struktúrák nem támogatnak egyedi operátorokat*

Ahhoz, hogy egy egyedi típus egyedi módon reagálhasson a lényeges operátorokra, a C# rendelkezik az `operator` kulcsszóval, amelyet csak *statikus* metódusokkal lehet használni. Amikor egy kétoperandusú operátort terhelünk túl (mint pl. a + és a -), leggyakrabban a definiáló osztály típusával (jelen példában a `Point`) megegyező két argumentumot adunk át neki, ahogy a következő kódmódosításban látható:

```
// Egy intelligensebb Point típus
public struct Point
{
  ...
  // Túlterhelt + operátor
  public static Point operator + (Point p1, Point p2)
  { return new Point(p1.x + p2.x, p1.y + p2.y); }

  // Túlterhelt - operátor
  public static Point operator - (Point p1, Point p2)
  { return new Point(p1.x - p2.x, p1.y - p2.y); }
}
```

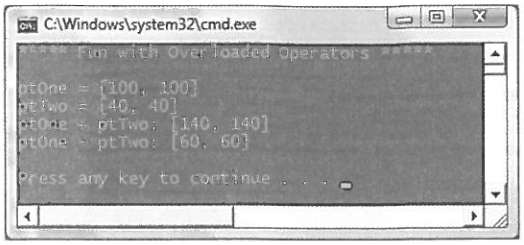
A + operátor mögött meghúzódó logika az, hogy a bemeneti `Point` paraméterek mezőinek összegzése alapján egy teljesen új `Point` értéket ad vissza. Így amikor a `pt1 + pt2` kifejezést leírjuk, a háttérben a statikus + operátormetódus következő rejtett meghívását képzelhetjük el:

```
// Point p3 = Point.operator+ (p1, p2)
Point p3 = p1 + p2;
```

Hasonlóan a p1 - p2 a következőre képeződik le:

```
// Point p4 = Point.operator- (p1, p2)
Point p4 = p1 - p2;
```

Ezzel a módosítással a programunk most már lefordítható, és képesek vagyunk `Point` objektumokat összeadni és kivonni (lásd 12.2. ábra).



12.2. ábra: *A + és a - újradefiniálása a Point típushoz*

Szigorúan véve egy kétoperandusú operátor túlterhelésekor nem követelmény az, hogy két azonos típusú paramétert adjunk át. Ha úgy praktikus, az egyik argumentum lehet más típusú. Az alábbi például egy túlterhelt + operátor, amely lehetővé teszi a hívó számára, hogy összeadjon egy `Point` objektumot egy egésszel (geometriai értelemben eltolja a pontot).

```
public struct Point
{
  ...
  public static Point operator +(Point p1, int change)
  {
    return new Point(p1.x + change, p1.y + change);
  }
  public static Point operator +(int change, Point p1)
  {
    return new Point(p1.x + change, p1.y + change);
  }
}
```

Így a + operátor új verzióit a következőképpen használhatjuk:

```
// [110,110] kiíratása
Point biggerPoint = ptOne + 10;
Console.WriteLine("ptOne + 10 = {0}", biggerPoint);

// [120,120] kiíratása
Console.WriteLine("10 + biggerPoint = {0}", 10 + biggerPoint);
Console.WriteLine();
```

Mi a helyzet a += és -= operátorokkal?

Ha C++-háttérrel kezdünk a C# megismeréséhez, valószínűleg hiányoljuk a rövidített értékadó operátorok túlterhelésének lehetőségét. A C# nyelvben a rövidített értékadó operátorok automatikusan szimulálódnak, ha egy típus túlterheli a hozzájuk kapcsolódó kétoperandusú operátort. Így mivel a `Point` struktúra már túlterhelte a + és a - operátorokat, a következőt írhatjuk:

```
// A kétoperandusú operátorok túlterhelése ingyenes rövidített
// operátort eredményez.
static void Main(string[] args)
{
  ...
  // Automatikus +=
  Point ptThree = new Point(90, 5);
  Console.WriteLine("ptThree = {0}", ptThree);
  Console.WriteLine("ptThree += ptTwo: {0}", ptThree += ptTwo);

  // Automatikus -=
  Point ptFour = new Point(0, 500);
  Console.WriteLine("ptFour = {0}", ptFour);
  Console.WriteLine("ptFour -= ptThree: {0}", ptFour -= ptThree);
  Console.ReadLine();
}
```

Egyoperandusú operátorok túlterhelése

A C# különböző egyoperandusú operátorok, mint például a ++ és a --, túlterhelését is lehetővé teszi. Egyoperandusú operátor túlterhelésekor is statikus metódust definiálunk az `operator` kulcsszó segítségével; ebben az esetben viszont csupán egyetlen, a definiáló osztály/struktúra típusával megegyező típusú paramétert adunk át. Ha például a `Point` definícióját a következő túlterhelt operátorokkal módosítanánk:

```
public struct Point
{
  ...
  // 1 hozzáadása a bemeneti Point-hoz.
  public static Point operator ++(Point p1)
  { return new Point(p1.x+1, p1.y+1); }

  // 1 kivonása a bemeneti Point-ból,
  public static Point operator --(Point p1)
  { return new Point(p1.x-1, p1.y-1); }
}
```

a `Point` x és y értékeit a következőképpen növelhetjük és csökkenthetjük:

```
static void Main(string[] args)
{
  ...
  // A ++ és -- egyoperandusú operátorok alkalmazása egy Point-ra.
  Point ptFive = new Point(1, 1);
  Console.WriteLine("++ptFive = {0}", ++ptFive); // [2, 2]
  Console.WriteLine("--ptFive = {0}", --ptFive); // [1, 1]
```

```
    // Ugyanazon operátorok alkalmazása utólagos
    // növelésre/csökkentésre.
    Point ptSix = new Point(20, 20);
    Console.WriteLine("ptSix++ = {0}", ptSix++); // [20, 20]
    Console.WriteLine("ptSix-- = {0}", ptSix--); // [21, 21]
    Console.ReadLine();
  }
```

Az előző példakódban az egyedi ++ és -- operátorainkat két teljesen különböző módon használtuk. A C++-ban az előzetes és utólagos növelő/csökkentő operátorokat külön-külön terhelhetjük túl. Ez a C# nyelvben nem lehetséges, viszont a növelés/csökkentés automatikusan helyes lesz (vagyis a túlterhelt ++ operátor esetén a pt++ értéke a kifejezésekben a módosítás előtti objektum értéke, míg a ++pt esetén már az új értéket használja a kifejezésben).

Az egyenlőségvizsgáló operátorok túlterhelése

A System.Object.Equals() túlterhelhető úgy, hogy értékalapú (és ne hivatkozásalapú) összehasonlítást végezzen a típusokon (lásd 6. fejezet). Ha úgy döntünk, hogy felülbíráljuk az Equals() (és gyakran a hozzá kapcsolódó System.Object.GetHashCode()) metódust, akkor egyértelmű, hogy az egyenlőségvizsgáló operátorokat (== és !=) túl kell terhelni. Ennek illusztrálására lássuk a módosított Point típust:

```
// A Point jelen verziója a == és != operátorokat is túlterheli.
public struct Point
{
  ...
  public override bool Equals(object o)
  {
    return o.ToString() == this.ToString();
  }

  public override int GetHashCode()
  { return this.ToString().GetHashCode(); }

  // Most terheljük túl a == és a != operátorokat!
  public static bool operator ==(Point p1, Point p2)
  { return p1.Equals(p2); }

  public static bool operator !=(Point p1, Point p2)
  { return !p1.Equals(p2); }
}
```

A == és a != operátorok megvalósítása egyszerűen csak meghívja a felülbírált Equals() metódust a munka nehezének elvégzésére. Ennek ismeretében most a következőképpen gyakorolhatunk a Point osztályunkon:

```
// A túlterhelt egyenlőségvizsgáló operátorok használata.
static void Main(string[] args)
{
...
Console.WriteLine("ptOne == ptTwo : {0}", ptOne == ptTwo);
Console.WriteLine("ptOne != ptTwo : {0}", ptOne != ptTwo);
Console.ReadLine();
}
```

Mint látható, elég intuitív módon hajtható végre két objektum összehasonlítása a jól ismert == és != operátorok segítségével az Object.Equals() meghívása helyett. Egy C#-osztályban nincs lehetőség csak az egyik egyenlőségvizsgáló operátor túlterhelésére, az == és a != csak együtt terhelhető túl (erre a fordító is emlékeztet).

Összehasonlító operátorok túlterhelése

A 9. fejezetben már láttuk, hogyan kell megvalósítani az IComparable interfészt két hasonló objektum közötti relatív kapcsolat megvizsgálásához. Ezen túlmenően az összehasonlító operátorokat (<, >, <= és >=) is túlterhelhetjük ugyanabban az osztályban. Az egyenlőségvizsgáló operátorokhoz hasonlóan a C# itt is megköveteli, hogy ha a < operátort túlterheljük, akkor a > operátort is túl kell terhelnünk. Ugyanez érvényes a <= és a >= operátorokra is. Ha a Point típus túlterheli ezeket az összehasonlító operátorokat, az objektum használója a következőképpen hasonlíthat össze két Point objektumot:

```
// A túlterhelt < és > operátorok használata
static void Main(string[] args)
{
...
  Console.WriteLine("ptOne < ptTwo : {0}", ptOne < ptTwo);
  Console.WriteLine("ptOne > ptTwo : {0}", ptOne > ptTwo);
  Console.ReadLine();
}
```

Ha megvalósítottuk az IComparable interfészt, az összehasonlító operátorok túlterhelése már egyszerű. A módosított osztálydefiníció a következő:

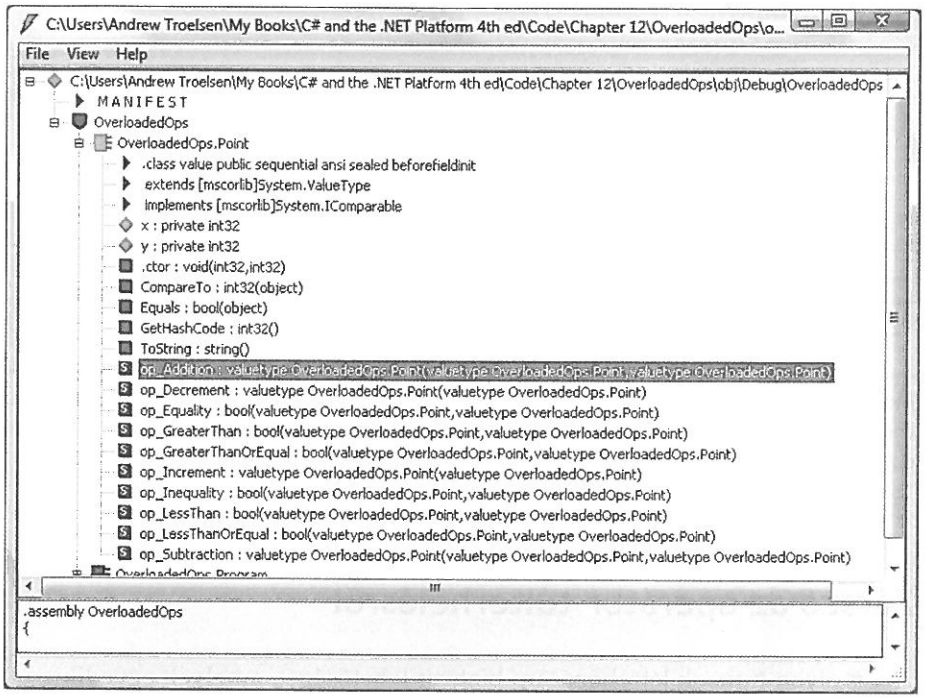
```
// A Point az összehasonlító operátorokkal is összehasonlítható.
public struct Point : IComparable
{
  ...
  public int CompareTo(object obj)
  {
    if (obj is Point)
    {
      Point p = (Point)obj;
      if (this.x > p.x && this.y > p.y)
        return 1;
      if (this.x < p.x && this.y < p.y)
        return -1;
      else
        return 0;
    }
    else
      throw new ArgumentException();
  }

  public static bool operator <(Point p1, Point p2)
  { return (p1.CompareTo(p2) < 0); }

  public static bool operator >(Point p1, Point p2)
  { return (p1.CompareTo(p2) > 0); }

  public static bool operator <=(Point p1, Point p2)
  { return (p1.CompareTo(p2) <= 0); }

  public static bool operator >=(Point p1, Point p2)
  { return (p1.CompareTo(p2) >= 0); }
}
```

A túlterhelt operátorok belső ábrázolása

A C# minden programozási eleméhez hasonlóan a túlterhelt operátorokat is speciális CIL-szintaxis jelöli. A háttéresemények vizsgálatához az `ildasm.exe` segítségével nyissuk meg az `OverloadedOps.exe` szerelvényt. A 12.3. ábrán látható: a túlterhelt operátorokat belsőleg rejtett metódusokkal (pl. `op_Addition()`, `op_Subtraction()`, `op_Equality()`, stb.) ábrázoljuk.

Megvizsgálva az `op_Addition` metódus pontos CIL-utasításait, azt találjuk, hogy a `specialname` metódus deklarációját is beillesztette a `csc.exe`:

```
.method public hidebysig specialname static
  valuetype OverloadedOps.Point
  op_Addition(valuetype OverloadedsOps.Point p1,
    valuetype OverloadedOps.Point p2) cil managed
{
  ...
}
```



12.3. ábra: *A köztes nyelv vonatkozásában a túlterhelt operátorok rejtett metódusokra képeződnek le*

Minden túlterhelhető operátorhoz tartozik egy speciálisan elnevezett köztes nyelvi metódus. A 12.2. táblázatban a leggyakrabban használt C#-operátorok CIL-leképezése látható.

Belső C#-operátor	CIL-beli ábrázolás
`--`	`op_Decrement()`
`++`	`op_Increment()`
`+`	`op_Addition()`
`-`	`op_Subtraction()`
`*`	`op_Multiply()`

Belső C#-operátor	CIL-beli ábrázolás
/	op_Division()
==	op_Equality()
>	op_GreaterThan()
<	op_LessThan()
!=	op_Inequality()
>=	op_GreaterThanOrEqual()
<=	op_LessThanOrEqual()
-=	op_SubtractionAssignment()
+=	op_AdditionAssignment()

12.2. táblázat: *C#-operátor – CIL speciális név leképezések*

Megjegyzés A túlterhelt operátorok „speciális nevét" gyakorlati okból érdemes ismerni. Mivel sok nyelvben nem használhatók a túlterhelt operátorral rendelkező osztályok, az ezeken dolgozó programozók a belső neveket tudják statikusan meghívni a definiáló típusból (pl. `Point.op_Addition(myPoint, yourPoint)`).

Zárszó az operátor-túlterhelésről

A C# olyan típusok létrehozására biztosít lehetőséget, amelyek egyedileg reagálnak a különböző jól ismert belső operátorokra. Mielőtt visszamenőleg átalakítanánk az összes osztályunkat az ilyen viselkedés támogatására, meg kell győződnünk arról, hogy az operátor(ok) túlterhelése logikus lépés-e.

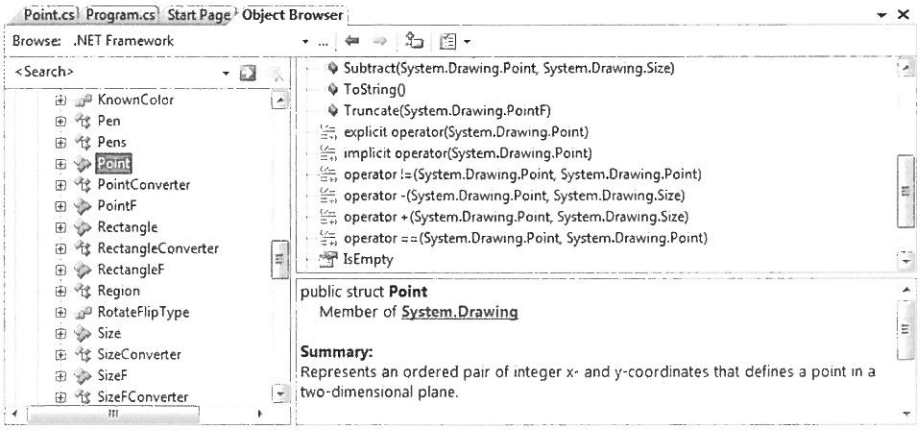
Túlterheljük például a `MiniVan` osztály szorzás operátorát. Mit is jelentene pontosan két `MiniVan` objektum összeszorzása? Valójában nem sokat. Sőt, a csoport többi tagja számára kifejezetten zavaró lehet, ha a következő `MiniVan` objektumokkal találkoznának:

```
// Huh?! Ez nem nevezhető éppen intuitívnak...
MiniVan newVan = myVan * yourVan;
```

Az operátorok túlterhelése általában segédtípusok létrehozásakor hasznos. A sztringek, pontok, négyszögek, törtek vagy hatszögek megfelelőek az operátor-túlterhelésre. Az emberek, vezetők, autók, adatbázis-kapcsolatok és weboldalak viszont nem.

A fő szabály az, hogy ha a túlterhelt operátor *nehezebbé* teszi a felhasználó számára a típus működését, akkor ne alkalmazzuk. Használjuk megfelelően ezt a szolgáltatást.

Tudnunk kell, hogy ha az egyedi osztályokban nem is szoktuk túlterhelni az operátorokat, az alaposztálykönyvtárakban sok típus megteszi ezt. A `System.Drawing.dll` szerelvényben található „hivatalos" `Point` definíció például számos operátort túlterhel. Figyeljük meg a 12.4. ábrán a Visual Studio 2008 Object Browser operátor ikonját.



12.4. ábra: *Az alaposztálykönyvtárak sok osztálya rendelkezik már eleve túlterhelt operátorokkal*

Forráskód Az `OverloadedOps` projekt megtalálható a 12. fejezet alkönyvtárában.

Egyedi típuskonverziók

A következőkben ismerkedjünk meg az egyedi típuskonverzióval, amely szorosan kapcsolódik az operátor-túlterhelés témájához. Ehhez idézzük fel a numerikus adatok és a hozzájuk kapcsolódó osztálytípusok közötti explicit és implicit konverziót.

A numerikus konverzió

A lényeges numerikus típusokhoz (`sbyte`, `int`, `float` stb.) *explicit konverzió* szükséges, ha nagyobb értéket szeretnénk kisebb tárolóban tárolni, ugyanis ez adatvesztéssel járhat. Az explicit konverziót mi magunk hajtjuk végre. Ez-

zel ellentétben az *implicit konverzió* automatikusan végbemegy, amikor kisebb típust akarunk elhelyezni egy nagyobb típusban, ez pedig nyilván nem eredményez adatvesztést.

```
static void Main()
{
  int a = 123;
  long b = a;        // Implicit konverzió int-ről long-ra
  int c = (int) b;   // Explicit konverzió long-ról int-re
}
```

Konverzió az egymásból származó osztálytípusok között

Az osztálytípusok klasszikus származtatás („az egy" kapcsolat) alapján kapcsolódhatnak egymáshoz (lásd 6. fejezet). Ekkor a C# konverziós eljárás lehetővé teszi az osztályhierarchiában a lefelé vagy felfelé történő kasztolást. Egy származtatott típus például implicit módon mindig konvertálható az alaptípusra. Ha viszont egy ősosztálytípust szeretnénk származtatott változóban tárolni, akkor explicit kasztolást kell végrehajtanunk:

```
// Két kapcsolódó osztálytípus
class Base{}
class Derived : Base{}
class Program

{
  static void Main()
  {
    // Implicit kasztolás a származtatottak között ősosztályba.
    Base myBaseType;
    myBaseType = new Derived();

    // Explicit kasztolást igényel az ősreferencia tárolása
    // származtatott típusban.
    Derived myDerivedType = (Derived)myBaseType;
  }
}
```

Ez az explicit kaszt azért működik, mert az ős- és a származtatott osztályok között klasszikus származtatási kapcsolat van. De mi történik akkor, ha *különböző hierarchiák*ban (a `System.Objecten` kívül más) közös ős nélkül van két különböző típusunk, és ezek igényelnek konverziót? Minthogy ezek a típusok a klasszikus származtatás alapján nem állnak kapcsolatban, az explicit kaszt nem segít.

Nézzünk meg az értéktípusokat (pl. struktúrákat). Tegyük fel, hogy van két .NET-struktúránk `Square` és `Rectangle` néven. Mivel a struktúrák nem rendelkeznek a származtatás képességével (ugyanis mindig zártak), nincs természetes módszer a két látszólag kapcsolódó típus közötti kasztolásra.

Bár létrehozhatunk segédmetódusokat a struktúrákban (pl. `Rectangle.ToSquare()`), a C# lehetőséget biztosít olyan egyedi konverziók készítésére is, amelyek hatására a típusaink reagálhatnak a `()` kasztoperátorra. Így, ha a struktúrákat megfelelően konfiguráltuk, a következő szintaxissal explicit módon tudunk konvertálni a két típus között:

```
// Rectangle konvertálása Square-ré.
Rectangle rect;
rect.Width = 3;
rect.Height = 10;
Square sq = (Square)rect;
```

Egyedi konverziós rutinok készítése

Hozzunk létre egy új, `CustomConversions` nevű parancssori alkalmazást. A C# rendelkezik két kulcsszóval, egy implicittel és egy explicittel, ezek azt szabályozzák, hogy a típusok hogyan reagáljanak a konverziós próbálkozásokra. Tegyük fel, hogy a következő struktúradefiníciókkal rendelkezünk:

```
public struct Rectangle
{
  // Public, a könnyebb használhatóságért;
  // de azért szabadon beágyazhatunk tulajdonságokat.
  public int Width, Height;

  public Rectangle(int w, int h)
  {
    Width = w; Height = h;
  }

  public void Draw()
  {
    for (int i = 0; i < Height; i++)
    {
      for (int j = 0; j < Width; j++)
      {
        Console.Write("*");
      }
      Console.WriteLine();
    }
  }
```

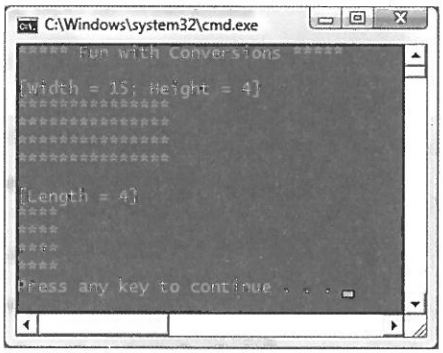
```
    public override string ToString()
    {
      return string.Format("[Width = {0}; Height = {1}]",
        Width, Height);
    }
}

public struct Square
{
    public int Length;
    public Square(int l)
    {
      Length = l;
    }

    public void Draw()
    {
      for (int i = 0; i < Length; i++)
      {
        for (int j = 0; j < Length; j++)
        {
          Console.Write("*");
        }
        Console.WriteLine();
      }
    }

    public override string ToString()
    { return string.Format("[Length = {0}]", Length); }

    // A Rectangle explicit módon konvertálható
    // Square-ré.
    public static explicit operator Square(Rectangle r)
    {
      Square s;
      s.Length = r.Height;
      return s;
    }
}
```

A `Square` típusnak ez a verziója explicit típuskonverziós operátort definiál. Az operátorok túlterheléséhez hasonlóan a konverziós rutinok is a C# `operator` kulcsszót használják (az `implicit` vagy `explicit` kulcsszóval együtt), és statikusként definiálandók. A bemenő paraméter az az elem, amelyet konvertálunk, míg az operátor típusa az az elem amely*re* konvertáljuk.

Ekkor azt feltételezzük, hogy a négyzet (azaz az olyan geometriai mintázat, amelynek minden oldala egyenlő hosszúságú) megkapható a téglalap magasságából. Így a `Rectangle` (téglalap) szabadon konvertálható `Square` (négyzet) típussá a következőképpen:

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

  // Téglalap létrehozása.
  Rectangle r = new Rectangle(15, 4);
  Console.WriteLine(r.ToString());
  r.Draw();

  Console.WriteLine();

  // r konvertálása négyzetté,
  // a téglalap magassága alapján.
  Square s = (Square)r;
  Console.WriteLine(s.ToString());
  s.Draw();
  Console.ReadLine();
}
```

A kimenet a 12.5. ábrán látható.



12.5. ábra: *A Rectangle struktúra konvertálása Square struktúrává*

Bár a `Rectangle` `Square`-ré konvertálása ugyanazon a hatókörön belül nem feltétlenül hasznos, tegyük fel mégis, hogy van egy olyan függvényünk, amely `Square` paramétereket fogad:

```
// Ez a metódus Square típusokat vár.
static void DrawSquare(Square sq)
{
  Console.WriteLine(sq.ToString());
  sq.Draw();
}
```

Ha az explicit konverzióoperátort a `Square` típusra alkalmazzuk explicit kaszt segítségével, akkor már `Rectangle` típusokat is átadhatunk feldolgozásra:

```
static void Main(string[] args)
{
  ...
  //  A Rectangle konvertálása Square-ré a metódus meghívásához.
  Rectangle rect = new Rectangle(10, 5);
  DrawSquare((Square)rect);
  Console.ReadLine();
}
```

A Square típus további explicit konverziói

Mindezek után vizsgáljunk meg néhány további explicit konverziót is. Mivel a négyzet minden oldalról szimmetrikus, hasznosíthatunk egy olyan explicit konverziós rutint, amely lehetővé teszi a `System.Int32` típus kasztolását `Square`-ré (amelynek természetesen az oldalhosszúsága lesz a bejövő egész számmal egyenlő). Mi történik akkor, ha a `Square` osztályunkat módosítjuk úgy, hogy a meghívó `Square`-ből `System.Int32`-vé tudjunk kasztolni? A meghívólogika a következő:

```
static void Main(string[] args)
{
  ...
  // System.Int32 konvertálása Square-ré.
  Square sq2 = (Square)90;
  Console.WriteLine("sq2 = {0}", sq2);

  // Square konvertálása System.Int32-vé.
  int side = (int)sq2;
  Console.WriteLine("Side length of sq2 = {0}", side);
  Console.ReadLine();
}
```

A `Square` típus módosítása pedig így néz ki:

```
public struct Square
{
  ...
  public static explicit operator Square(int sideLength)
  {
    Square newSq;
    newSq.Length = sideLength;
    return newSq;
  }

  public static explicit operator int (Square s)
  {return s.Length;}
}
```

A `Square` konvertálása `System.Int32`-vé nem feltétlenül a leghasznosabb művelet. Viszont rávilágít az egyedi konverziós rutinokkal kapcsolatban egy nagyon fontos tényre: a fordítót nem érdekli, miről mire konvertálunk, ha szintaktikailag helyes kódot írtunk. Így az operátorok túlterheléséhez hasonlóan, csak mert egy adott típushoz explicit kasztoperátort tudunk létrehozni, ez nem azt jelenti, hogy meg is kell tenni. Általában ez a technika .NET-struktúratípusok készítésekor a leghasznosabb, azok ugyanis nem vehetnek részt klasszikus származtatásban (ahol a kasztolás automatikusan adott).

Implicit konverziós rutinok definiálása

Eddig különböző *explicit* egyedi konverziós műveleteket hoztunk létre. De mi a helyzet a következő *implicit* konverzióval?

```
static void Main(string[] args)
{
  ...
  // Implicit kaszttal próbálkozunk?
  Square s3;
  s3.Length = 83;
  Rectangle rect2 = s3;
  Console.ReadLine();
}
```

Ez a kód nem fordul le, ugyanis nem adtunk meg implicit konverziós rutint a `Rectangle` típushoz. Itt a probléma: nem legális ugyanarra a típusra explicit és implicit konverziós függvényeket definiálni, ha nem eltérő a visszatérési típusuk vagy paraméterkészletük. Ez korlátozásnak tűnhet; ám a másik probléma az, hogy ha egy típus *implicit* konverziós rutint definiál, a meghívó alkalmazhatja az *explicit* kasztszintaxist.

A tisztánlátás kedvéért az `implicit` C#-kulcsszót alkalmazva bővítsük egy implicit konverziós rutinnal a `Rectangle` struktúrát (a következő kód feltételezi, hogy az előálló téglalap szélességét a négyzet oldalhosszának kétszeres szorzatával nyerjük).

```
public struct Rectangle
{
  ...
  public static implicit operator Rectangle(Square s)
  {
    Rectangle r;
    r.Height = s.Length;
```

```
    // Tegyük fel, hogy az új téglalap hossza
    // (Length x 2)
    r.Width = s.Length * 2;
    return r;
  }
}
```

Ezzel a módosítással már tudunk a típusok között konvertálni a következőképpen:

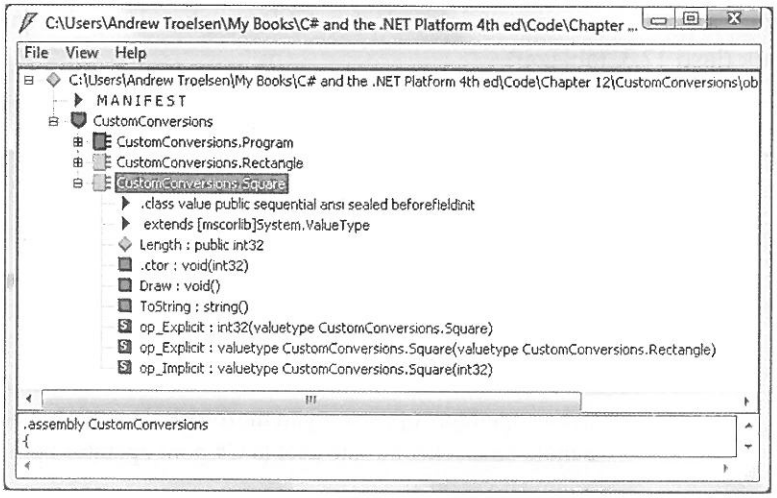
```
static void Main(string[] args)
{
  ...
  // Implicit kaszt OK!
  Square s3;
  s3.Length= 7;
  Rectangle rect2 = s3;
  Console.WriteLine("rect2 = {0}", rect2);
  DrawSquare(s3);

  // Explicit kasztolási szintaxis még mindig OK!
  Square s4;
  s4.Length = 3;
  Rectangle rect3 = (Rectangle)s4;
  Console.WriteLine("rect3 = {0}", rect3);
  Console.ReadLine();
}
```

Ne felejtsük el, hogy csak akkor lehet ugyanarra a típusra explicit és implicit konverziós rutinokat definiálni, ha a szignatúrájuk eltérő. Így a `Square` a következőképpen módosítható:

```
public struct Square
{
  ...
  // Meghívható így:
  // Square sq2 = (Square)90;
  // vagy így:
  // Square sq2 = 90;
  public static implicit operator Square(int sideLength)
  {
    Square newSq;
    newSq.Length = sideLength;
    return newSq;
  }

  // Így kell meghívni:
  // int side = (int)mySquare;
  public static explicit operator int (Square s)
  { return s.Length; }
}
```

Az egyedi konverziós rutinok belső ábrázolása

A túlterhelt operátorokhoz hasonlóan az implicit vagy az explicit kulcsszóval jelölt metódusoknak is „speciális" nevük van a köztes nyelv használatában: `op_Implicit` illetve `op_Explicit` (lásd 12.6. ábra).



12.6. ábra: *A felhasználó által definiált konverziós rutinok köztes nyelvi ábrázolása*

Megjegyzés A Visual Studio 2008 Objektum Browser az „explicit operátor" és „implicit operátor" ikonokkal jelzi az egyedi konverziós operátorokat.

A túlterhelt operátorokhoz hasonlóan, ez esetben se felejtsük el, hogy ez a szintaxis egyszerűen a „normál" tagfüggvények rövidített jelzése, és így mindig opcionális. Megfelelően alkalmazva azonban az egyedi struktúráinkat használhatóbbakká teszi, mivel ekkor úgy tudjuk kezelni őket, mintha származtatással kapcsolódó, valódi osztályok lennének.

Forráskód A `CustomConversions` projekt megtalálható a 12. fejezet alkönyvtárában.

Mutatótípusok használata

A .NET platform két fő adatkategóriát definiál: értéktípusokat és referenciatípusokat (lásd 4. fejezet). Létezik egy harmadik kategória is: a *mutatótípusoké*. A mutatótípusokhoz speciális operátorok és kulcsszavak is tartoznak, amelyek lehetővé teszik, hogy a memóriakezelést a CLR megkerülésével saját kézben tartsuk (lásd 12.3. táblázat).

Operátor/kulcsszó	Valós jelentés
*	Ezt az operátort mutatóváltozó (vagyis olyan változó, amely a memória egy közvetlen helyét jelöli) létrehozására használjuk. Ahogy a C(++) nyelvben, ugyanezt az operátort használjuk mutatóindirekcióra is.
&	Ezt az operátort egy változó memóriabeli címének lekérdezésére használjuk.
->	Ezt az operátort típusok olyan mezőinek elérésére használjuk, amelyeket mutató jelöl (a C# pont operátor nem felügyelt verziója).
[]	A [] operátor (nem felügyelt környezetben) lehetővé teszi a mutatóváltozó által hivatkozott memóriaterület címzését (lásd a C(++)-beli mutató és [] operátor közötti összefüggést).
++, --	Nem felügyelt környezetben a növelő és a csökkentő operátorok alkalmazhatók mutatótípusokra.
+, -	Nem felügyelt környezetben az összeadó és a kivonó operátorok alkalmazhatók mutatótípusokra.
==, ! =, <, >, <=, =>	Nem felügyelt környezetben az összehasonlító és az egyenlőségvizsgáló operátorok alkalmazhatók mutatótípusokra.
stackalloc	Nem felügyelt környezetben a `stackalloc` kulcsszó használható C#-tömbök foglalására, közvetlenül a veremben.
fixed	Nem felügyelt környezetben a `fixed` kulcsszó használható egy változó időleges rögzítésére címének a megtalálásához.

12.7. ábra: *Mutatócentrikus C#-operátorok és -kulcsszavak*

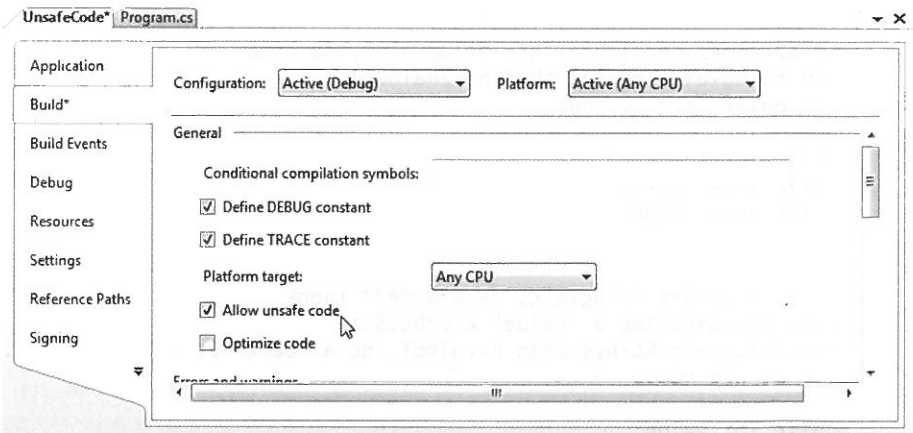
Fontos hangsúlyoznunk, hogy *nagyon ritkán*, ám az is lehet, hogy *sohasem* lesz szükségünk a mutatótípusok alkalmazására. Bár a C# lehetőséget ad a mutatóműveletekre, a .NET-futtatókörnyezet nincs tisztában a szándékunkkal. Így, ha egy mutatót rosszul kezelünk, ennek következményei a mi felelősségünk. Mikor kell tehát mutatótípusokat használnunk? Két gyakori lehetőség van:

- Optimalizálni szeretnénk alkalmazásunk egyes részeit azzal, hogy közvetlenül manipuláljuk a CLR kezelésén kívüli memóriát.
- C-alapú `*.dll` vagy COM szervermetódusait hívjuk meg, amelyek paraméterként mutatótípusokat várnak. Még ekkor is gyakran kikerülhetjük azonban a mutatók használatát a `System.IntPtr` és a `System.Runtime.InteropServices.Marshal` típus tagjainak az alkalmazásával.

Ha mégis úgy döntünk, hogy használjuk ezt a C# nyelvi szolgáltatást, értesítenünk kell a C#-fordítót (`csc.exe`) erről a szándékunkról oly módon, hogy engedélyezzük a projektben a „nem felügyelt kód" támogatását. Ezt parancssorban úgy tehetjük meg, hogy egyszerűen argumentumként megadjuk az `unsafe` kapcsolót:

```
csc /unsafe *.cs
```

A Visual Studio 2008-ban a projekt Properties oldalán a Build fülön be kell jelölnünk az Allow Unsafe Code jelölőnégyzetet (12.7. ábra). A mutatótípusokkal való kísérletezgetéshez pedig hozzunk létre egy UnsafeCode nevű új parancssorialkalmazás-projektet, és engedélyezzük a nem felügyelt kódot.



12.7. ábra *Nem felügyelt kód engedélyezése a Visual Studio 2008-ban*

Megjegyzés A következő példák feltételeznek némi C(++) mutatókezelési tapasztalatot. Ennek hiányában nyugodtan átugorhatjuk a teljes témakört. A nem felügyelt kód írása nem lesz általános feladat a C#-alkalmazások túlnyomó többségénél.

Az unsafe kulcsszó

Ha a C#-ban mutatókat akarunk használni, külön deklarálnunk kell egy „nem felügyelt kód" blokkot az `unsafe` kulcsszó segítségével (ekkor azok a kódok, amelyeket az `unsafe` nem jelöl, automatikusan „felügyelt" értelmezést kapnak). A következő `Program` osztály a felügyelt `Main()` metódusban deklarálja egy nem felügyelt kód hatókörét:

```
class Program
{
  static void Main(string[] args)
  {
    unsafe
    {
      // Itt dolgozhatunk mutatótípusokkal!
    }
    // Itt nem dolgozhatunk mutatótípusokkal!
  }
}
```

A metóduson belüli nem felügyelt kód hatóköreinek definiálásán túl létrehozhatunk „nem felügyelt" struktúrákat, osztályokat, típustagokat és paramétereket is. Nézzük meg az alábbi néhány példát (ezeket a típusokat nem kell definiálni az aktuális projektünkben):

```
// Ez az egész strultúra "nem felügyelt", és csak
// nem felügyelt környezetben használható.
public unsafe struct Node
{
  public int Value;
  public Node* Left;
  public Node* Right;
}

// Ez a struktúra felügyelt, de a Node2* tagok
// nem. Technikailag a "Value" elérhető a
// nem felügyelt környezeten kívülről, de a "Left" és a "Right" nem.
public struct Node2
{
  public int Value;
```

```
  // Ezek csak nem felügyelt környezetben érhetők el!
  public unsafe Node2* Left;
  public unsafe Node2* Right;
}
```

A metódusok (statikusak vagy példányszintűek) is megjelölhetők nem felügyeltként. Tegyük fel például, hogy ismerünk egy konkrét statikus metódust, amely mutatókat alkalmaz. Annak biztosítására, hogy a metódus csak nem felügyelt környezetben legyen meghívható, a következő definíciót adhatjuk:

```
unsafe static void SquareIntPointer(int* myIntPointer)
{
  // Emeljük négyzetre az értékét az ellenőrzés kedvéért.
  *myIntPointer *= *myIntPointer;
}
```

A metódusunk konfigurációjához arra van szükség, hogy a meghívó a következőképpen hívja meg a `SquareIntPointer ()` metódust:

```
static void Main(string[] args)
{
  unsafe
  {
    int myInt = 10;
    // OK, mert nem felügyelt környezetben vagyunk.
    SquareIntPointer(&myInt);
    Console.WriteLine("myInt: {0}", myInt);
  }

  int myInt2 = 5;
  // Fordítási hiba! Nem felügyelt környezetben kell lennie!
  SquareIntPointer(&myInt2);
  Console.WriteLine("myInt: {0}", myInt2);
}
```

Ha nem akarjuk a meghívót arra kényszeríteni, hogy a meghívást nem felügyelt környezetbe burkolja, módosítanunk kell a `Main()` metódusunkat az `unsafe` kulcsszóval. Ekkor lefordítható a következő kód:

```
unsafe static void Main(string[] args)
{
  int myInt2 = 5;
  SquareIntPointer(&myInt2);
  Console.WriteLine("myInt: {0}", myInt2);
}
```

A * és az & operátor

A nem felügyelt környezet megteremtésekor a * operátor segítségével szabadon előállíthatunk mutatókat a különböző adattípusokhoz, míg az & operátor segítségével lekérhetjük a mutatók címét. A C vagy C++ nyelvektől eltérően a C# * operátora csak az alapadattípusra alkalmazható, nem pedig prefixumként az egyes mutatóváltozó-nevekre. Vizsgáljuk meg például a következő kódot, amely mutatók deklarálásának helyes és helytelen módját illusztrálja az egész változókhoz:

```
// Nem! Ez a C#-ban nem helyes!
int *pi, *pj;

// Igen! Ez a C#-ban helyes!
int* pi, pj;
```

Nézzük meg a következő nem felügyelt metódust:

```
unsafe static void PrintValueAndAddress()
{
  int myInt;

  // Definiálunk egy int mutatót, és
  // a myInt értékét rendeljük hozzá.
  int* ptrToMyInt = &myInt;

  // A myInt értékének beállítása mutató indirekció segítségével.
  *ptrToMyInt = 123;

  // Kiíratunk néhány adatot.
  Console.WriteLine("Value of myInt {0}", myInt);
  Console.WriteLine("Address of myInt {0:X}", (int)&ptrToMyInt);
}
```

A nem felügyelt (és felügyelt) cserefüggvény

Mutatók deklarálása a helyi változókra pusztán az értékadás kedvéért soha nem szükségszerű. Vegyünk egy gyakorlati(asabb) példát a nem felügyelt kód használatára: egy cserefüggvényt szeretnénk mutatóaritmetika segítségével előállítani:

```
unsafe public static void UnsafeSwap(int* i, int* j)
{
  int temp = *i;
  *i = *j;
  *j = temp;
}
```

C-szerűnek tűnik, ám a cserealgoritmus következő felügyelt változatát is megírhatnánk a C# `ref` kulcsszó segítségével (lásd a 4. fejezetet):

```
public static void SafeSwap(ref int i, ref int j)
{
  int temp = i;
  i = j;
  j = temp;
}
```

A két metódus szerepe megegyezik, így megerősítik azt, hogy a közvetlen mutatómanipulálás nem kötelező feladat a C#-ban. A meghívólogika felügyelt `Main()` használatával, nem felügyelt környezetben a következő:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Calling method with unsafe code *****");

  // A felcserélendő értékek.
  int i = 10, j = 20;

  // Az értékek "felügyelt" cseréje.
  Console.WriteLine("\n***** Safe swap *****");
  Console.WriteLine("Values before safe swap: i = {0}, j = {1}",
    i, j);
  SafeSwap(ref i, ref j);
  Console.WriteLine("Values after safe swap: i = {0}, j = {1}",
    i, j);

  // Az értékek "nem felügyelt" cseréje.
  Console.WriteLine("\n***** Unsafe swap *****");
  Console.WriteLine("Values before unsafe swap: i = {0}, j = {1}",
    i, j);
  unsafe { UnsafeSwap(&i, &j); }
  Console.WriteLine("Values after unsafe swap: i = {0}, j = {1}",
    i, j);
  Console.ReadLine();
}
```

Mezőelérés mutatókkal (a -> operátor)

Tételezzük fel, hogy a következőképpen definiáltunk egy egyszerű felügyelt `Point` struktúrát:

```
struct Point
{
  public int x;
  public int y;
  public override string ToString()
  { return string.Format("({0}, {1})", x, y);}
}
```

Ha a `Point` típushoz mutatót deklarálunk, akkor nyilvános tagjainak az eléréséhez a mutató-mezőelérési operátort (a -> szimbólum jelöli) kell használnunk. Ahogy a 12.3. táblázatban láttuk, ez a normál (felügyelt) pont opetátor (.) nem felügyelt verziója. Igazság szerint a mutatóindirekció-operátor (*) használatával lehetőségünk van a mutató hivatkozásának eltávolítására, hogy (ismét) használhassuk a pont operátor jelölést. Nézzük meg a nem felügyelt metódust:

```
unsafe static void UsePointerToPoint()
{
  // Tagok elérése mutatókkal.
  Point point;
  Point* p = &point;
  p->x = 100;
  p->y = 200;
  Console.WriteLine(p->ToString());

  // Tagok elérése mutatóindirekcióval.
  Point point2;
  Point* p2 = &point2;
  (*p2).x = 100;
  (*p2).y = 200;
  Console.WriteLine((*p2).ToString());
}
```

A stackalloc kulcsszó

Előfordulhat, hogy nem felügyelt környezetben olyan lokális változót kell deklarálnunk, amely közvetlenül a hívó veremből foglalja le a memóriát (és így nem terjed ki rá a .NET-szemétgyűjtés). A C# ehhez a `stackalloc` kulcsszót biztosítja, amely a szabványos C programkönyvtártárában lévő `_alloca` függvény C#-beli megfelelője. Nézzünk egy egyszerű példa:

```
unsafe static void UnsafeStackAlloc()
{
  char* p = stackalloc char[256];
  for (int k = 0; k < 256; k++)
    p[k] = (char)k;
}
```

Típuspéldány rögzítése a fixed kulcsszóval

Egy memóriadarab lefoglalása nem felügyelt környezetben megoldható a `stackalloc` kulcsszó segítségével. A művelet természeténél fogva a lefoglalt memória felszabadul, amikor a foglaló metódus visszatér (mivel a memóriát a veremtől szerzi). Nézzünk egy bonyolultabb példát. A -> operátor vizsgálatakor létrehoztunk egy `Point` nevű típust. Mint minden értéktípusnál, a lefoglalt memória itt is kidobódik a veremből a végrehajtó hatókör lezárulásakor. Tegyük fel, hogy a `Point` inkább *referenciatípusként* lett definiálva:

```
class PointRef // <= Átnevezve és áttípusolva.
{
  public int x;
  public int y;
  public override string ToString()
  { return string.Format("({0}, {1})", x, y);}
}
```

Ha a hívó `Point` típusú változót deklarál, a memória a szemétgyűjtött heapen foglalódik le. Ekkor felmerül a kérdés: Mi történik, ha egy nem felügyelt környezet akar ezzel az objektummal (vagy bármely objektummal a heapen) interakcióba lépni? Mivel a szemétgyűjtés bármelyik pillanatban bekövetkezhet, képzeljük csak el, milyen problémákat eredményezhet, ha éppen a heap takarításának pillanatában próbáljuk meg a `Point` tagjait elérni. Elméletileg előfordulhat, hogy a nem felügyelt környezet már nem elérhető, vagy a heap generációs takarítását túlélt áthelyezett taggal próbál meg interakcióba lépni (ez pedig nyilvánvaló problémát okoz).

A referenciatípusú változó nem felügyelt környezetből történő zárolásához a C# a `fixed` kulcsszót biztosítja. A `fixed` utasítás beállítja a mutatót a felügyelt típusra, és „rögzíti" a változót az utasítás végrehajtásának az idejére. A `fixed` nélkül a felügyelt típusokra mutató mutatók nem járnának túl sok előnnyel, hiszen a szemétgyűjtés kiszámíthatatlan módon áthelyezheti őket. (A C#-fordító nem engedélyezi mutató beállítását a felügyelt változóra, csak `fixed` utasításban.)

Így, ha létrehozzuk a `Point` típust (immár osztályként újratervezve), és a tagjaival akarunk dolgozni, a következő kódot kell írnunk (máskülönben fordítási hibát kapunk):

```
unsafe public static void UseAndPinPoint()
{
  PointRef pt = new PointRef ();
  pt.x = 5;
  pt.y = 6;

  // A pt rögzítése, hogy ne legyen
  // elmozdítva vagy szemétgyűjtve.
  fixed (int* p = &pt.x)
  {
    // int* változót használunk!
  }

  // A pt már nincs rögzítve, szabadon begyűjthető.
  Console.WriteLine ("Point is: {0}", pt);
}
```

Röviden, a `fixed` kulcsszó lehetővé teszi olyan utasítás létrehozását, amely lezár egy referenciaváltozót a memóriában, így ennek a címe az utasítás időtartamára állandó marad. A biztonság kedvéért, ha nem felügyelt kódkörnyezetből dolgozunk a referenciatípusokkal, a referencia rögzítése kötelező.

A sizeof kulcsszó

Vizsgáljunk meg még egy nem felügyelt környezeti kulcsszót: ez a `sizeof`. A C(++) nyelvhez hasonlóan a C# `sizeof` kulcsszó is egy értéktípus (soha nem referenciatípus) byte-ban mért méretének lekérdezésére használatos, és kizárólag nem felügyelt környezeten belül használható. Ez a lehetőség nagy segítség lehet akkor, amikor nem felügyelt C-alapú API-kkal kell interakcióba lépnünk. A használata egyértelmű:

```
unsafe static void UseSizeOfOperator()
{
  Console.WriteLine("The size of short is {0}.", sizeof(short));
  Console.WriteLine("The size of int is {0}.", sizeof(int));
  Console.WriteLine("The size of long is {0}.", sizeof(long));
}
```

Mivel a `sizeof` minden `System.ValueType`-ból származtatott egyedben kiszámítja a byte-ok számát, az egyedi struktúrák méretét is megkaphatjuk. A `Point` struktúrát például a következőképpen adhatjuk át a `sizeof` kulcsszónak:

```
unsafe static void UseSizeOfOperator()
{
  ...
  Console.WriteLine("The size of Point is {0}.", sizeof(Point));
}
```

Forráskód Az `UnsafeCode` projekt megtalálható a 12. fejezet alkönyvtárában.

A C# előfordítói direktívái

A C-család sok más tagjához hasonlóan a C# is támogatja olyan különböző szimbólumok használatát, amelyekkel a fordítási folyamatot befolyásolhatjuk. A C# különböző előfeldolgozó direktívái előtt módosítsunk a terminológián. A „C# előfeldolgozó direktíva" kifejezés nem teljesen pontos. Valójában ezt csak a C és C++ programozási nyelvekkel való átjárhatóság miatt használjuk. A C#-ban nincs előfeldolgozási lépés. Az előfeldolgozó direktívák feldolgozása a fordító lexikaianalízis-fázisának részeként történik.

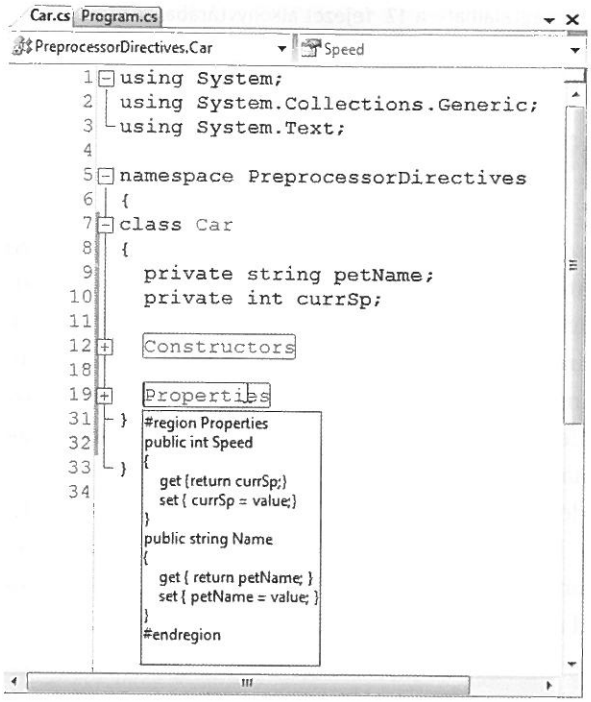
Mindenesetre a C# előfeldolgozó direktívák szintaxisa nagyon hasonlít a C-család más tagjaiéhoz abban, hogy a direktívák prefixuma mindig a kettőskereszt (#) jel. A 12.4. táblázatban látható a leggyakrabban használt direktívák közül néhány (a részletekért lásd a .NET Framework 3.5 SDK dokumentációját).

Direktíva	Valós jelentés
`#region, #endregion`	Az összecsukható forráskód szakaszainak jelölésére használatos.
`#define, #undef`	Feltételes fordítási szimbólumok definiálására és a definíció visszavonására használatos.
`#if, #elif, #else, #endif`	A forráskód egyes szakaszainak feltételes (speciális fordítási jelek alapján történő) átugrására használatos.

12.4. táblázat: *Gyakori C# előfeldolgozó direktívák*

Kódrégiók megadása

Talán az egyik leghasznosabb előfeldolgozó direktíva a `#region` és `#endregion`. Ezeknek a címkéknek az alkalmazásával meghatározhatunk egy olyan kódblokkot, amelyet elrejthetünk, és egy barátságos szöveges jelölővel azonosíthatunk. A régiók használatával a hosszú `*.cs` fájlok sokkal könnyebben kezelhetők. Létrehozhatunk például egy régiót a típus konstruktorai számára, egyet a tulajdonságainak és így tovább.



12.8. ábra: *Régiók működés közben*

```
class Car
{
  private string petName;
  private int currSp;

  #region Constructors
  public Car()
  { ... }
  public Car (int currSp, string petName)
  { ... }
  #endregion
```

```
  #region Properties
  public int Speed
  { ... }
  public string Name
  { ... }
  #endregion
}
```

Amikor az egérkurzort egy összecsukott régió fölé helyezzük, kicsiben megpillanthatjuk a háttérben rejtőzködő kódot (lásd 12.8. ábra).

Feltételes kódfordítás

Az előfeldolgozó direktívák következő csoportja (`#if`, `#elif`, `#else`, `#endif`) lehetővé teszi, hogy egy kódblokkot az előzetesen definiált szimbólumok szerint feltételesen fordítsunk. A direktívák klasszikus alkalmazása nem más, mint egy kódblokk megjelölése azért, hogy csak hibakeresési (és nem kibocsátási) konfiguráció alatt legyen lefordítva:

```
class Program
{
  static void Main(string[] args)
  {
    #region Print machine info under DEBUG build
    // Ez a kód csak akkor hajtódik végre, ha a projekt
    // hibakeresési konfigurációval lett lefordítva.
    #if DEBUG
    Console.WriteLine("App directory: {0}",
      Environment.CurrentDirectory);
    Console.WriteLine("Box: {0}",
      Environment.MachineName);
    Console.WriteLine("OS: {0}",
      Environment.OSVersion);
    Console.WriteLine(".NET Version: {0}",
      Environment.Version);
    #endif
    #endregion
  }
}
```

Keressük a DEBUG nevű szimbólumot. Ha megvan, számos érdekes statisztikát tudunk kidobni a `System.Environment` osztály statikus tagjainak a segítségével. Ha a DEBUG szimbólumot nem definiáljuk, az „és az" közé helyezett kód fordítása nem kerül bele az előálló szerelvénybe, és így gyakorlatilag figyelmen kívül marad.

Megjegyzés A `System.Diagnostics` névtér rendelkezik egy `[Conditional]` attribútummal, amely osztályra vagy metódusra alkalmazható. A `[Conditional]`-t alkalmazva nem kell használnunk a kapcsolódó előfeldolgozó szimbólumokat (az attribútumok szerepét lásd a 16. fejezetben).

Az alapértelmezés szerint a Visual Studio 2008 mindig definiálja a DEBUG szimbólumot; ez elkerülhető, ha a projekt Properties oldalának Build lapján eltávolítjuk a Define DEBUG constant jelölőnégyzetet. Ha kikapcsoltuk az automatikusan generált DEBUG szimbólumot, ettől kezdve ezt fájlonként definiálhatjuk a `#define` előfeldolgozó direktíva segítségével:

```
#define DEBUG
using System;
namespace PreprocessorDirectives
{
  class Program
  {
    static void Main(string[] args)
    {
      // Ugyanaz a kód, mint az előbb...
    }
  }
}
```

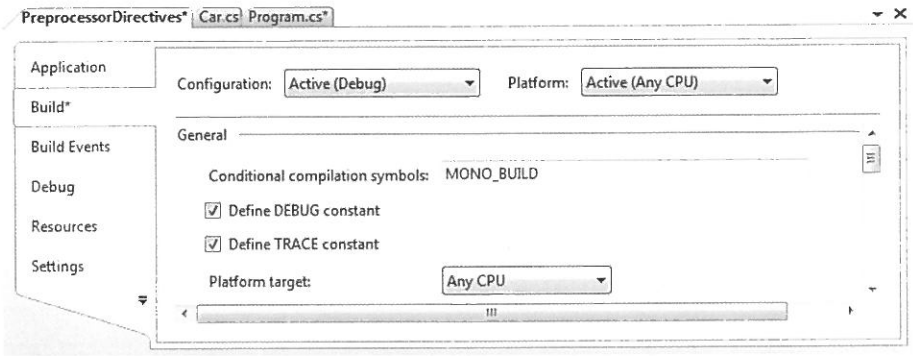
Megjegyzés A `#define` direktívákat a C#-kódfájl legelején, minden más előtt kell felsorolni.

Definiálhatunk saját egyedi előfeldolgozó szimbólumokat is. Tegyük fel például, hogy egy olyan C#-osztályt írtunk, amelyet a .NET Mono disztribúciója (lásd B függelék) alatt kicsit másként kell fordítani. A `#define` segítségével fájlonként definiálhatunk egy `MONO_BUILD` nevű szimbólumot:

```
#define DEBUG
#define MONO_BUILD

using System;

namespace PreprocessorDirectives
{
  class Program
  {
    static void Main(string[] args)
    {
      #if MONO_BUILD
        Console.WriteLine("Compiling under Mono!");
      #else
```

```
            Console.WriteLine("Compiling under Microsoft .NET");
        #endif
    }
  }
}
```

A teljes projektben alkalmazott szimbólumok létrehozásához használjuk a projekt Properties oldalának Build fülén található Conditional compilation symbols (feltételes fordítási szimbólum) szövegmezőt (lásd 12.9. ábra).



12.9. ábra: *A teljes projektben használatos előfeldolgozó szimbólum definiálása*

Forráskód A `PreprocessorDirectives` projekt megtalálható a 12. fejezet alkönyvtárában.

Összefoglalás

A fejezet célja az volt, hogy elmélyítse a C# programozási nyelv ismeretét. Különféle fejlett típuslétrehozási technikákkal kezdtünk (indexelő metódusok, túlterhelt operátorok és egyedi konverziós rutinok). Majd a kevésbé ismert kulcsszavak egy kis csoportját vizsgáltuk (pl. `sizeof`, `checked`, `unsafe` stb.), és közben megismerkedtünk a natív mutatótípusok használatával is. A mutatótípusok tanulmányozásakor végig azt állítottuk, hogy C#-alkalmazásainkban nagy valószínűséggel *soha* nem lesz szükség a használatukra.

Végezetül megvizsgáltuk az alapvető C# előfeldolgozó direktívákat, amelyek a kódfájl fordításával kapcsolatban a fordítóval (vagy a `#region`/`#endregion` esetében a Visual Studio 2008-cal) való interakciót teszik lehetővé.

TIZENHARMADIK FEJEZET

A C# 2008 nyelv szolgáltatásai

A C# 2008, a Microsoft vezető .NET programozási nyelvének aktuális verziója számos új szintaxiskonstrukciót tartalmaz. (Ezek egyikével, a lambda operátorral már megismerkedtünk a 11. fejezetben.) Ebben a fejezetben a C# 2008 új nyelvi szolgáltatásai közül megvizsgáljuk az implicit adattípusok meghatározását, az automatikus tulajdonságokat, a kiterjesztő metódusokat, a részleges metódusokat, az objektuminicializálókat, valamint a névtelen (`anonymous`) típusok szerepét.

Az új nyelvi szolgáltatások többsége közvetlenül használható jól működő és hatékony .NET-szoftverek készítéséhez, sok ilyen új szerkezet azonban a LINQ-technológiakészlettel együtt alkalmazva nyújtja a legnagyobb segítséget (a LINQ-t lásd a 14. fejezetben). Lehet, hogy néhány új szolgáltatás hasznossága nem fog első látásra egyértelműnek tűnni, a LINQ szerepének tisztázásával azonban az új szolgáltatások funkciója is kristálytisztává válik.

Implicit típusú lokális változók

A C# 2008 nyelvi szolgáltatásai közül elsőként a lokális változók *implicit típusmeghatározását* nézzük meg a stílusosan elnevezett `ImplicitlyTypedLocalVars` új konzolalkalmazás segítségével. A lokális változók (mint az egyes metódusok hatókörében definiált változók) nagyon egyértelműen (és *explicit* módon) vannak deklarálva:

```
static void DeclareExplicitVars()
{
  // Az explicit típusú lokális változók
  // a következőképpen vannak deklarálva:
  // adatTípus változóNév = kezdőÉrték;
  int myInt = 0;
  bool myBool = true;
  string myString = "Time, marches on...";
}
```

A C# 2008 rendelkezik a `var` kulcsszóval, amelyet egy formális adattípus (mint pl. az `int`, a `bool` vagy a `string`) megadása helyett használhatunk. Ha ezt tesszük, a fordító a lokális változó inicializálására használt kezdőérték alapján automatikusan kitalálja az adat típusát. Így például az előző változókat a következőképpen is deklarálhatjuk:

```
static void DeclareImplicitVars()
{
  // Az implicit típusú lokális változók
  // a következőképpen vannak deklarálva:
  // var változóNév = kezdőÉrték;
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";
}
```

Megjegyzés Szigorúan véve a var nem is C#-kulcsszó. A „var" nevű változókat, paramétereket és mezőket fordítási idejű hiba nélkül lehet deklarálni. Viszont ha a var-t adattípus helyett használjuk, a környezet alapján a fordító kulcsszóként kezeli. Az egyszerűség kedvéért a „var kulcsszó" elnevezést fogjuk használni a nehézkesebb „környezetfüggő var" jelölés helyett.

Ekkor a fordító a kezdetben megadott érték alapján ki tudja következtetni, hogy a `myInt` `System.Int32`, a `myBool` `System.Boolean` és a `myString` tulajdonképpen `System.String` típusú. Erről meg is győződhetünk, ha reflexió használatával kiíratjuk a típusneveket:

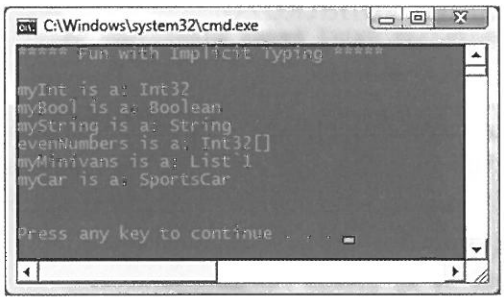
```
static void DeclareImplicitVars()
{
  // Implicit típusú lokális változók.
  var myInt = 0;
  var myBool = true;
  var myString = "Time, marches on...";

  // Kiíratjuk az alaptípust.
  Console.WriteLine("myInt is a: {0}", myInt.GetType().Name);
  Console.WriteLine("myBool is a: {0}", myBool.GetType().Name);
  Console.WriteLine("myString is a: {0}", myString.GetType().Name);
}
```

Az ilyen implicit tipizálást bármilyen típusra alkalmazhatjuk, beleértve a tömböket, a generikus típusokat, valamint a saját egyedi típusainkat is:

```
static void DeclareImplicitVars()
{
  ...
  // Még több implicit típusú lokális változó.
  var evenNumbers = new int[] { 2, 4, 6, 8 };
```

```
    var myMinivans = new List<MiniVan>();
    var myCar = new SportsCar();

    Console.WriteLine("evenNumbers is a: {0}",
      evenNumbers.GetType().Name);
    Console.WriteLine("myMinivans is a: {0}",
      myMinivans.GetType().Name);
    Console.WriteLine("myCar is a: {0}", myCar.GetType().Name);
  }
```

Ha a `Main()` metóduson belül meghívjuk a `DeclareImplicitVars()` metódust, a 13.1. ábrán látható kimenetet kapjuk.



13.1. ábra: *Az implicit módon definiált lokális változók áttekintése*

A var használata foreach szerkezetekben

Használhatunk implicit típusokat a `foreach` ciklusokban is. A fordító ki tudja következtetni a megfelelő típusú változót. Vizsgáljuk meg a következő metódust, amely egy implicit típusú egész tömb felett iterál:

```
static void VarInForeachLoop()
{
  var evenNumbers = new int[] { 2, 4, 6, 8 };

  // A "var" használata egy szokványos foreach ciklusban.
  foreach (var item in evenNumbers)
  {
    Console.WriteLine("Item value: {0}", item);
  }
}
```

Tudnunk kell ugyanakkor, hogy a `foreach` ciklus alkalmazhat erősen tipizált iterátort az implicit módon megadott lokális tömb feldolgozásakor. Így a következő kód is helyes szintaktikailag:

```
static void VarInForeachLoop()
{
  var evenNumbers = new int[] { 2, 4, 6, 8 };

  // Erősen tipizált System.Int32 használata a tartalom feletti
  // iterációhoz.
  foreach (int item in evenNumbers)
  {
    Console.WriteLine("Item value: {0}", item);
  }
}
```

Az implicit típusú változókra vonatkozó korlátozások

Természetesen több korlátozás is érvényes a `var` kulcsszó használatára. Először is, az implicit tipizálás *kizárólag* lokális változókra érvényes egy metódus vagy tulajdonság hatókörén belül. A `var` kulcsszó nem használható visszatérési értékek, paraméterek vagy adott típusú adatmezők definiálására:

```
class ThisWillNeverCompile
{
  // Hiba! A var nem használható mezőadatként!
  private var myInt = 10;

  // Hiba! A var nem használható visszatérési értékként
  // vagy paramétertípusként!
  public var MyMethod(var x, var y){}
}
```

Hasonlóan, a `var` kulcsszóval definiált lokális változóknak *kötelező* a deklarálás idején kezdőértéket adni, és *nem* adható nekik kezdőértékként `null`. Az első megkötés az implicit változók definiálását némileg a konstans adatok `const` kulcsszóval történő definiálásához (lásd 5. fejezet) teszi hasonlatossá. A második megkötésnek logikusnak kell tűnnie, hiszen a fordító pusztán a `null` értékből képtelen kikövetkeztetni, hogy a változó milyen típusra mutathat a memóriában:

```
// Hiba! Értéket kell adni!
var myData;

// Hiba! Az értéket pontosan a deklaráció idejében kell megadni!
var myInt;
myInt = 0;
```

```
// Hiba! Nem adható null kezdőértéknek!
var myObj = null;
```

Ugyanakkor megengedett a kezdeti értékadás után a már kikövetkeztetett típusú lokális változónak `null` értéket adni (feltéve, hogy referenciatípusú):

```
// OK, a SportsCar referenciatípusú!
var myCar = new SportsCar();
myCar = null;
```

Sőt, az is megengedett, hogy egy implicit típusú lokális változó értékét adjuk értékül egy másik változónak, akár implicit típusú, akár nem:

```
// Ez is OK!
var myInt = 0;
var anotherInt = myInt;

string myString = "Wake up!";
var myData = myString;
```

Megengedett továbbá, hogy implicit típusú lokális változót adjunk vissza a meghívónak, feltéve, hogy a függvény visszatérési típusa ugyanaz az alapadattípus, mint a `var` kulcsszóval definiált adaté:

```
static int GetAnInt()
{
  var retVal = 9;
  return retVal;
}
```

És végül, de nem utolsósorban, nem legális nullázható lokális változót létrehozni a C# `?` jelölésével (a nullázható adattípusok részleteit lásd a 4. fejezetben):

```
// Nem, nem definiálhatunk nullázható implicit változókat,
// mivel az implicit változóknak soha nem lehet
// null-t adni kezdőértékként!
var? nope = new SportsCar();
var? stillNo = 12;
var? noWay = null;
```

Implicit típusú lokális tömbök

Az implicit típusú lokális változókhoz szorosan kapcsolódnak az *implicit típusú lokális tömbök*. Ezzel a technikával lefoglalhatunk egy új tömbtípust anélkül, hogy meghatároznánk a tömbben tárolt adatok típusát:

```
static void DeclareImplicitArrays()
{
  // Az a valóban int[].
  var a = new[] { 1, 10, 100, 1000 };
  Console.WriteLine("a is a: {0}", a.ToString());

  // A b valóban double[].
  var b = new[] { 1, 1.5, 2, 2.5 };
  Console.WriteLine("b is a: {0}", b.ToString());

  // A c valóban string[].
  var c = new[] { "hello", null, "world" };
  Console.WriteLine("c is a: {0}", c.ToString());

  // A myCar valóban SportsCar[].
  var myCars = new[] { new SportsCar(), new SportsCar() };
  Console.WriteLine("myCars is a: {0}", myCars.ToString());
  Console.WriteLine();
}
```

Természetesen éppúgy, mint amikor az explicit C#-szintaxissal foglaljuk le a tömböt, a tömb inicializálási listájában szereplő elemeknek ugyanolyan típusúaknak kell lenniük (csupa `int`, csupa `string`, csupa `SportsCar` stb.). Várakozásunktól eltérően az implicit típusú lokális tömbök az alapértelmezés szerint nem a `System.Object` elemei, így a következő fordítási idejű hibát okoz:

```
// Hiba! Kevert típusok!
var d = new[] { 1, "one", 2, "two", false };
```

Az implicit típusú adatok erősen tipizáltak

Nagyon ügyeljünk arra, hogy a lokális változók implicit tipizálása *erősen tipizált adatok*at eredményez. Ezért a `var` kulcsszó használata *nem* ugyanazt a technikát igényli, mint a szkriptnyelvekben (pl. VBScript vagy Perl) vagy a COM `Variant` adattípusánál, ahol a változó a programon belüli életciklusa alatt különböző típusú értékeket tárolhat (ezt gyakran „dinamikus tipizálás" néven emlegetjük).

Ehelyett a típuskövetkeztetés megőrzi a C# nyelv erős tipizálási szemléletét, és csupán a változók fordítási idejű deklarálását érinti. Ezután az adat úgy működik, mintha azzal a típussal lett volna deklarálva; ha ettől eltérő típusú értéket adunk a változónak, fordítási idejű hibát kapunk:

```
static void ImplicitTypingIsStrongTyping()
{
  // A fordító tudja, hogy az "s" egy System.String.
  var s = "This variable can only hold string data!";
  s = "This is fine...";

  // Az alaptípus bármely tagja meghívható.
  string upper = s.ToUpper();

  // Hiba! Numerikus adat nem adható sztring értékéül!
  s = 44;
}
```

Az implicit típusú lokális változók előnye

Az implicit típusú lokális változók deklarálására szolgáló szintaxis ismeretében a következő kérdés az, hogy mikor használjuk ezt a szerkezetet? Először is, a `var` alkalmazása lokális változók deklarálásához pusztán a használata miatt nem jár semmi előnnyel. Csak összezavarhatja azokat, akik a kódunkat olvassák, ugyanis nehezebbé válik az adattípus meghatározása (és így még nehezebb a változó általános szerepének megértése). Éppen ezért, ha tudjuk, hogy `int`-re lesz szükségünk, deklaráljuk is `int`-ként.

A LINQ technológiakészlet *lekérdezési kifejezéseket* is használ, amelyek a lekérdezés formátumán alapuló, dinamikusan előállított eredményhalmazokat adhatnak (lásd 14. fejezet). Ilyen esetekben az implicit tipizálás nagy segítséget jelent, mivel nem kell explicit módon definiálnunk azokat a típusokat, amelyekkel a lekérdezés visszatérhet - ez sok esetben szó szerint lehetetlen is lenne. A következő LINQ-példakódon nézzük meg, hogy meg tudjuk-e állapítani a `subset` típusát:

```
static void QueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };
  var subset = from i in numbers where i < 10 select i;

  Console.Write("Values in subset: ");
  foreach (var i in subset)
  {
    Console.Write("{0} ", i);
  }
  Console.WriteLine();
```

```
    // Hmm...milyen típus is a subset?
    Console.WriteLine("subset is a: {0}", subset.GetType().Name);
    Console.WriteLine("subset is defined in: {0}",
      subset.GetType().Namespace);
}
```

A részhalmaz típusának ellenőrzéséhez futtassuk le az előző kódot (ez nem `int` tömb lesz). Az implicit adattípusoknak a LINQ-technológiakészletben van a helyük. Tulajdonképpen azt is mondhatnánk, hogy a `var` kulcsszót *csak* a LINQ-lekérdezés által visszaadott adatok definiálásakor használjuk.

Forráskód Az `ImplicitlyTypedLocalVars` projekt a 13. fejezet alkönyvtárában található.

Automatikus tulajdonságok

A .NET programozási nyelvek sokkal inkább tulajdonságokat alkalmaznak a különböző típusú privát adatmezők biztonságos bekéréséhez és értékének megadásához, mint a hagyományos `GetXXX()` és `SetXXX()` metódusokat (lásd az 5. fejezetben a beágyazási szolgáltatásoknál). Vizsgáljuk meg a következő beágyazott sztringtípust:

```
// A Car típus, szabványos tulajdonság-
// szintaxis alkalmazásával.
class Car
{
  private string carName = string.Empty;
  public int PetName
  {
    get { return carName; }
    set { carName = value; }
  }
}
```

Bár a C#-tulajdonságok definiálása nem túl bonyolult, ám amikor a tulajdonságok egyszerűen csak kijelölik és azonnal visszaadják az értéket, mint ahogy a fentiekben, akkor elég hosszadalmas többször definiálni a háttérmezőket és az egyszerű tulajdonságokat. Ha például egy olyan típust modellezünk, amely 15 privátadatmező-értékkel rendelkezik, akkor 15 kapcsolódó tulajdonságot kell létrehoznunk, amelyek alig többek az egységbe zárás érdekében alkalmazott vékony burkolóknál.

Hogy leegyszerűsítse az adatmezők egyszerű egységbe zárásának folyamatát, a C# 2008-ban megjelent az *automatikustulajdonság-szintaxis*. Ahogy a nevéből sejthető, ez a szolgáltatás egy új szintaxisrész alkalmazásával áthelyezi a privát segédmezők és a hozzájuk kapcsolódó C#-tulajdonságok definiálásának feladatát a fordítóra. A C# 2008-ban például az előző `Car` típust a következőképpen definiálnánk:

```
class Car
{
  // Automatikus tulajdonság szintaxisa
  public string PetName { get; set; }
}
```

Megjegyzés A Visual Studio 2008 „prop" kódrészletét átírták, hogy az automatikustulajdonság-szintaxist használja a hagyományos C#-tulajdonságlogika helyett (a kódrészletek magyarázatát lásd a 2. fejezetben).

Első pillantásra az automatikustulajdonság-szintaxisról a meg nem valósított `get` és `set` blokkok miatt azt hihetnénk, hogy egy absztrakt tulajdonságot definiáltunk, amelyet felülírnak a származtatott tulajdonságok. De nem ez a helyzet. Ha absztrakt tulajdonságot akarnánk a `Car` típusban létrehozni, az `abstract` C#-kulcsszót kellene használnunk a következőképpen:

```
abstract class Car
{
  // Absztrakt tulajdonság absztrakt ősosztályban
  public abstract string PetName { get; set; }
}
```

Az automatikus tulajdonságok definiálásakor egyszerűen megadjuk a hozzáférési módosítót, az adattípust, a tulajdonság nevét és üres `get/set` blokkokat. A fordítás idején a típusunkhoz létrejön egy automatikusan generált privát segédmező és a `get/set` logika megfelelő megvalósulása.

Megjegyzés Az automatikusan generált privát segédmező nem látható a C#-kódunkban. Megnézni például a ildasm.exe segítségével lehet.

A hagyományos C#-tulajdonságoktól eltérően viszont nem lehet írásvédett vagy csak írható automatikus tulajdonságokat létrehozni. Bár azt gondolhatnánk, hogy egyszerűen csak kihagyható a `get;` vagy a `set;` a tulajdonságdeklarációinkban, ahogy a következőkben:

```
// Írásvédett tulajdonság? Hiba!
public int MyReadOnlyProp { get; }

// Csak írható tulajdonság? Hiba!
public int MyWriteOnlyProp { set; }
```

ez azonban fordítási hibát eredményez. Az automatikus tulajdonságokat úgy kell definiálni, hogy mind az írási, mind az olvasási funkciókat támogassák.

Interakció automatikus tulajdonságokkal

Mivel a fordító a privát segédmezőket fordítási időben definiálja, az automatikus tulajdonságokat meghatározó osztályoknál mindig tulajdonságszintaxist kell alkalmazni az alapérték lekéréséhez és beállításához. Ez azért fontos, mert sok programozó az *osztálydefinícióban* közvetlenül használja a privát mezőket, ám ez ebben az esetben lehetetlen. Ha például a `Car` típus felülírná a `ToString()` metódust, akkor ezt a tulajdonság nevével kellene megvalósítanunk:

```
class Car
{
  public string PetName { get; set; }
  public override string ToString()
  {
    // Nem érhető el a privát tag a definiáló
    // osztályban. Tulajdonságokat kell használni!
    return string.Format("PetName = {0}", PetName);
  }
}
```

Ha automatikus tulajdonságokkal definiált objektumot használunk, az elvárt tulajdonságszintaxis segítségével adhatunk és kérhetünk be értékeket:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Automatic Properties *****");
  Car c = new Car();
  c.PetName = "Frank";
  Console.WriteLine("Your car is named {0}? That's odd...",
    c.PetName);
  Console.ReadLine();
}
```

Az automatikus tulajdonságok elérésének korlátozása

A „normális" .NET-tulajdonságokat úgy is létre lehet hozni, hogy a `get` és a `set` logika egyedi hozzáférés-módosítót kap. Nyilvános `get` hatókört és sokkal korlátozottabb, védett hatókört tudunk például definiálni a következőképpen:

```
// Bárki hozzáférhet a PetName értékéhez, de
// csak a definiáló típus és gyermekei állíthatják be.
public int PetName
{
  get { return carName; }
  protected set { carName = value; }
}
```

Ugyanez a lehetőség megvan az automatikustulajdonság-szintaxis használatakor is, a következőképpen:

```
public string PetName { get; protected set; }
```

Természetesen ezzel a módosítással az előbbi `Main()` metódus most fordítási hibát eredményezne, ha megpróbálnánk a `PetName` tulajdonág értékét megadni:

```
static void Main(string[] args)
{
  ...
  // Hiba! A PetName beállítása csak a
  // Car típus vagy egy gyermek típus számára lehetséges!
  c.PetName = "Frank";

  // Az érték kérése még mindig OK.
  Console.WriteLine("Your car is named {0}? That's odd...",
    c.PetName);
  Console.ReadLine();
}
```

Az automatikus tulajdonságok és az alapértelmezett értékek

Amikor numerikus vagy logikai adatok beágyazására használunk automatikus tulajdonságokat, az automatikusan generált típustulajdonságok azonnal használhatók lesznek a kódbázisban, ugyanis a rejtett segédmezők közvetle-

nül alkalmazható, biztonságos alapértelmezett értéket kapnak. Ám ha automatikustulajdonság-szintaxist használunk referenciatípusok beburkolására, a rejtett privát referenciatípust is alapértelmezett `null` értékre kell beállítani:

```
class Garage
{
  // A rejtett int segédmező nullára van állítva!
  public int NumberOfCars { get; set; }

  // A rejtett Car segédmező nullára van állítva!
  public Car MyAuto { get; set; }
}
```

A C# alapértelmezett adatmezőértékei miatt a `NumberOfCars` azonnal kiíratható úgy, ahogy van (minthogy automatikusan nulla értéket kap), de ha közvetlenül meghívjuk a `MyAuto` változót, nullreferencia-kivételt kapunk:

```
static void Main(string[] args)
{
  ...
  Garage g = new Garage();

  // OK, kiírja az alapértelmezett 0 értéket
  Console.WriteLine("Number of Cars: {0}", g.NumberOfCars);

  // Futásidejű hiba! A segédmező jelenleg null!
  Console.WriteLine(g.MyAuto.PetName);
  Console.ReadLine();
}
```

Mivel a privát segédmezők fordítási időben jönnek létre, ezért nem használhatjuk bennük a `new` kulcsszót sem a referenciatípusok közvetlen példányosítására. Ezért ezt a munkát a tartalmazó osztály konstruktorában kell elvégezni, hogy biztosítsuk az objektum biztonságos létrehozását. Például:

```
class Garage
{
  // A rejtett segédmező nullára van állítva!
  public int NumberOfCars { get; set; }
  // A rejtett segédmező null értékre van állítva!
  public Car MyAuto { get; set; }
  // Konstruktorok használata szükséges
  // a segédmezőknek adott alapértelmezett értékek felülírásához.
  public Garage()
  {
    MyAuto = new Car();
    NumberOfCars = 1;
  }
```

```
    public Garage(Car car, int number)
    {
      MyAuto = car;
      NumberOfCars = number;
    }
}
```

A továbbfejlesztett szintaxis segítségével rengeteg tulajdonságot definiálhatunk az osztályokhoz. Ám ügyelnünk kell arra is, hogy ha olyan tulajdonságot hozunk létre, amely az alap-privátmező értékének beállításán és lekérésén túl további kódolást igényel (pl. ellenőrzési logika, eseménynaplóba írás, kommunikáció egy adatbázissal stb.), akkor a „normális" .NET-tulajdonságot manuálisan kell definiálnunk. A C# 2008 automatikus tulajdonságai soha nem nyújtanak többet, mint az alapadattípus egyszerű beágyazását.

Forráskód Az `AutomaticProperties` projekt a 13. fejezet alkönyvtárában található.

Bővítő metódusok

A következő C# 2008 nyelvi szolgáltatás, amelyet megvizsgálunk, a *bővítő metódusok* használata lesz. Ha egy típust definiáltunk, majd lefordítottunk egy .NET-szerelvénybe, akkor ez a definíció többé-kevésbé végleges marad. Csak úgy vehetünk fel, módosíthatunk vagy törölhetünk tagokat, ha újrakódoljuk és módosított szerelvényként újrafordítjuk az alapkódot (vagy a `System.Reflection.Emit` névteret alkalmazzuk a memóriába fordított típus dinamikus átalakítására).

A C# 2008 lehetőséget biztosít bővítő metódusok definiálására. Röviden: a bővítő metódusok lehetővé teszik, hogy a meglévő lefordított típusok (konkrétan osztályok, struktúrák vagy interfészek implementációi) és a fordítás alatt álló típusok (pl. a bővítő metódusokat tartalmazó projektekben) új funkcionalitást kapjanak anélkül, hogy a bővítendő típust közvetlenül módosítani kellene.

Ez a technika akkor lehet nagyon hasznos, amikor új funkcionalitásokat kell olyan típusokba beilleszteni, amelyeknek nem rendelkezünk a kódbázisával. Továbbá, amikor egy típust arra kell kényszerítenünk, hogy (a polimorfizmus érdekében) támogasson bizonyos tagkészletet, de az eredeti típusdeklarációt nem tudjuk módosítani. A bővítő metódusok segítségével új funkcionalitással bővíthetjük az előfordított típusokat, miközben fenntartjuk azt az illúziót, hogy ezek a metódusok végig megvoltak.

Megjegyzés A bővítő metódusok nem módosítják ténylegesen a lefordított kész kódbázist. Ez a technika csupán tagokkal bővít egy típust az aktuális alkalmazás kontextusában.

A bővítő metódusok definiálásakor az első megszorítás az, hogy egy *statikus osztályon* (lásd 5. fejezet) belül kell őket megadni, így tehát minden bővítő metódust a `static` kulcsszóval kell deklarálni. A második az, hogy a `this` kulcsszóval kell megjelölnünk a bővítő metódusok első (és csak az első) paraméterét. A harmadik, hogy minden bővítő metódus vagy a memóriában szereplő megfelelő példányból, vagy *statikusan* az őt definiáló statikus osztályból hívható meg. Nézzünk erre egy teljes példát.

A bővítő metódusok definiálása

Hozzunk létre egy `ExtensionMethods` nevű parancssorialkalmazás-projektet. Tegyük fel, hogy egy `MyExtensions` nevű segédosztályt hozunk létre, amely két bővítő metódust definiál. Az első metódus lehetővé teszi a .NET-alaposztálykönyvtár bármelyik objektuma számára, hogy egy `DisplayDefiningAssembly()` nevű teljesen új metódussal bővüljön, amely a `System.Reflection` névtérben szereplő típusok segítségével megjeleníti a megadott típus szerelvényét.

A `ReverseDigits()` nevű második bővítő metódus lehetővé teszi bármely `System.Int32` számára, hogy önmaga olyan új verzióját vegye fel értékül, amelyben a számjegyek fordított sorrendben szerepelnek. Ha például az 1234 értékű egész szám meghívja a `ReverseDigits()` metódust, a visszaadott egész szám értéke 4321 lesz. Vizsgáljuk meg a következő osztálymegvalósítást:

```
static class MyExtensions
{
  // Ez a metódus lehetővé teszi bármely objektum számára,
  // hogy megjelenítse a szerelvényt, amelyben definiálva van.
  public static void DisplayDefiningAssembly(this object obj)
  {
    Console.WriteLine("{0} lives here:\n\t->{1}\n",
      obj.GetType().Name,
    Assembly.GetAssembly(obj.GetType()));
  }

  // Ez a metódus lehetővé teszi az egész számoknak,
  // hogy megfordítsák számjegyeiket. Például az 56 65-öt ad vissza.
  public static int ReverseDigits(this int i)
  {
    // Az int-et átfordítjuk sztringgé, majd
    // kérjük az összes karaktert.
    char[] digits = i.ToString().ToCharArray();
```

```
        // Most megfordítjuk az elemeket a tömbben.
        Array.Reverse(digits);

        // Visszahelyezzük sztringbe.
        string newDigits = new string(digits);

        // Végül visszaadjuk a módosított sztringet int-ként.
        return int.Parse(newDigits);
    }
}
```

Figyeljük meg, hogy a két bővítő metódus első paraméterére hogyan használtuk a `this` kulcsszót, mielőtt definiáltuk volna a paraméter típusát. A bővítő metódus első paramétere mindig jelöli azt a típust, amelyet bővítünk. Mivel a `DisplayDefiningAssembly()`, prototípusa szerint a `System.Objectet` bővíti ki, minden szerelvény minden osztálya rendelkezik ettől kezdve ezzel az új taggal. Ugyanakkor a `ReverseDigits()`, prototípusa szerint csak az egész típusokat bővíti ki, így ha az egész számokon kívül bármilyen más típus megpróbálja meghívni ezt a metódust, fordítási idejű hibát kapunk.

Egy adott bővítő metódusnak több paramétere is lehet, de *kizárólag* az első paraméter kap `this` módosítót. Egy túlterhelt bővítő metódus egy másik, `TesterUtilClass` nevű segédosztályban például a következő:

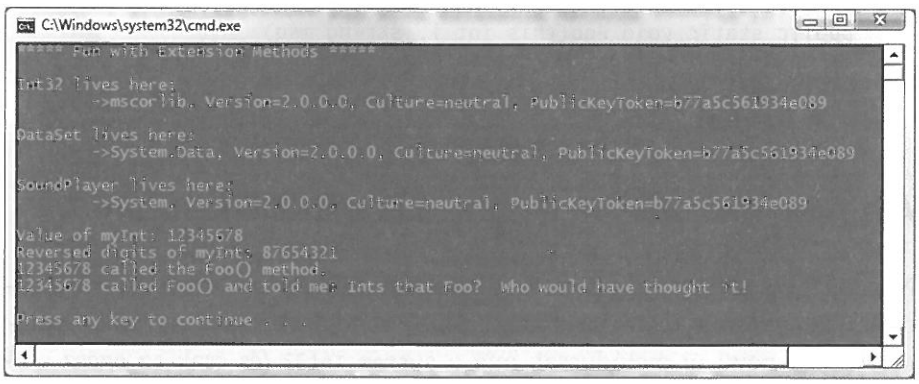
```
static class TesterUtilClass
{
  // Már minden Int32-nek van Foo() metódusa...
  public static void Foo(this int i)
  { Console.WriteLine("{0} called the Foo() method.", i); }

  // ...amelyet túlterheltünk, hogy sztringet fogadjon!
  public static void Foo(this int i, string msg)
  { Console.WriteLine("{0} called Foo() and told me: {1}",
    i, msg); }
}
```

A bővítő metódusok meghívása példányszinten

Nézzük meg ezután, hogy az összes objektum (ez természetesen a .NET-alaposztálykönyvtárak teljes tartalmát jelenti) rendelkezik a `DisplayDefiningAssembly()` nevű új metódussal, míg a `System.Int32` (és csak az egész számok) rendelkeznek `ReverseDigits()` és `Foo()` nevű metódusokkal:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Extension Methods *****\n");

  // Az int új személyazonosságot vett fel!
  int myInt = 12345678;
  myInt.DisplayDefiningAssembly();

  // A DataSet is!
  System.Data.DataSet d = new System.Data.DataSet();
  d.DisplayDefiningAssembly();

  // És a SoundPlayer is!
  System.Media.SoundPlayer sp = new System.Media.SoundPlayer();
  sp.DisplayDefiningAssembly();

  // Használjuk az új egész szerepet.
  Console.WriteLine("Value of myInt: {0}", myInt);
  Console.WriteLine("Reversed digits of myInt: {0}",
    myInt.ReverseDigits());
  myInt.Foo();
  myInt.Foo("Ints that Foo? Who would have thought it!");

  bool b2 = true;

  // Hiba! A logikai típusoknak nincs Foo() metódusuk!
  // b2.Foo();

  Console.ReadLine();
}
```

A 13.2. ábrán látható a kimenet.



13.2. ábra: *A bővítő metódusok működése*

A bővítő metódusok statikus meghívása

A bővítő metódus első paramétere tehát a `this` kulcsszóval van megjelölve, amelyet annak az elemnek a típusa követ, amelyre a metódus vonatkozik. Ha megnézzük, mi történik a hátérben (az `ildasm.exe`-vel vagy hasonló eszközzel), akkor azt találjuk, hogy a fordító egyszerűen meghívja a normális statikus metódust, paraméterként átadva neki a metódust meghívó változót (azaz a `this` értékét). Vizsgáljuk meg a következő C#-kódot, amely a végrehajtott helyettesítést imitálja:

```
private static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Extension Methods *****\n");
  int myInt = 12345678;
  MyExtensions.DisplayDefiningAssembly(myInt);

  DataSet d = new DataSet();
  MyExtensions.DisplayDefiningAssembly(d);

  SoundPlayer sp = new SoundPlayer();
  MyExtensions.DisplayDefiningAssembly(sp);

  Console.WriteLine("Value of myInt: {0}", myInt);
  Console.WriteLine("Reversed digits of myInt: {0}",
    MyExtensions.ReverseDigits(myInt));
  TesterUtilClass.Foo(myInt);
  TesterUtilClass.Foo(myInt, "A Foo metódus egész számokkal működik?
    Ki gondolta volna!?");
  Console.ReadLine();
}
```

Mivel a fordító azt az illúziót kelti, hogy a bővítő metódust az objektumból hívtuk meg (ettől úgy tűnik, mintha a metódus tulajdonképpen példányszintű lenne), így a bővítő metódusokat bármikor szabadon meghívhatjuk a szokásos C#-szintaxis használatával (lásd a példában).

A bővítő metódus hatóköre

A bővítő metódusok lényegében olyan statikus metódusok, amelyek a kibővített típus egy példányából hívhatók meg. Az ilyen szintaxisegyszerűsítés miatt kell hangsúlyoznunk, hogy a „normális" metódusoktól eltérően a bővítő metódusoknak nincs közvetlen hozzáférésük annak a típusnak a tagjaihoz, amelyet bővítenek; másképpen: a *bővítés* nem *öröklés*. Vizsgáljuk meg a következő egyszerű `Car` típust:

```
public class Car
{
  public int Speed;
  public int SpeedUp()
  {
    return ++Speed;
  }
}
```

Ha ehhez akarunk egy `SlowDown()` nevű bővítő metódust írni, nincs közvetlen hozzáférésünk a `Car` tagjaihoz a bővítő metódus hatókörén belül, ugyanis nem klasszikus származtatást hajtunk végre. Éppen ezért a következő metódus fordítási hibát eredményez:

```
public static class CarExtensions
{
  public static int SlowDown(this Car c)
  {
    // Hiba! Ez a metódus nem a Car-ból származik!
    return --Speed;
  }
}
```

Az a gond, hogy a statikus `SlowDown()` bővítő metódus a `Car` típus `Speed` mezőjéhez próbál hozzáférni; viszont mivel a `SlowDown()` a `CarExtensions` osztály statikus tagja, a `Speed` ebben a kontextusban nem létezik. Ellenben a `this` jelzésű paraméter használata lehetséges a kibővített típus összes nyilvános tagjának (és *csak* a nyilvános tagjainak) az elérésére. Így a következő kód a várakozásunknak megfelelően lefordul:

```
public static class CarExtensions
{
  public static int SlowDown(this Car c)
  {
    // OK!
    return --c.Speed;
  }
}
```

Ekkor létrehozhatunk egy `Car` objektumot, és meghívhatjuk a `SpeedUp()` és a `SlowDown()` metódusokat a következőképpen:

```
static void UseCar()
{
  Car c = new Car();
  Console.WriteLine("Speed: {0}", c.SpeedUp());
  Console.WriteLine("Speed: {0}", c.SlowDown());
}
```

A bővítő metódusokat definiáló típusok importálása

Amikor bővítő metódusokat tartalmazó statikus osztályok egy halmazát valamilyen egyedi névtérbe helyezzük, a szerelvényben szereplő más névterek ezeket a statikus osztályokat és bővítő metódusokat a C# `using` kulcsszóval importálhatják.

Ezt nagyon fontos megjegyeznünk, ugyanis ha explicit módon nem importáljuk a megfelelő névteret, a bővítő metódusok nem lesznek elérhetők az adott C#-kódfájlban.

Úgy tűnhet elsőre, hogy a bővítő metódusok globális természetűek, de valójában az őket definiáló vagy importáló névterekre korlátozódnak. Így ha a statikus osztályaink (`MyExtensions`, `TesterUtilClass` és `CarExtensions`) definícióit egy `MyExtensionMethods` nevű névtérbe burkoljuk:

```
namespace MyExtensionMethods
{
  static class MyExtensions
  {
    ...
  }

  static class TesterUtilClass
  {
    ...
  }

  static class CarExtensions
  {
    ...
  }
}
```

akkor a projektben szereplő más névtereknek explicit módon importálniuk kell a `MyExtensionMethods` névteret ahhoz, hogy a fenti típusok által definiált bővítő függvényeket használhassák. Ha ez nem történik meg, fordítási hibát kapunk:

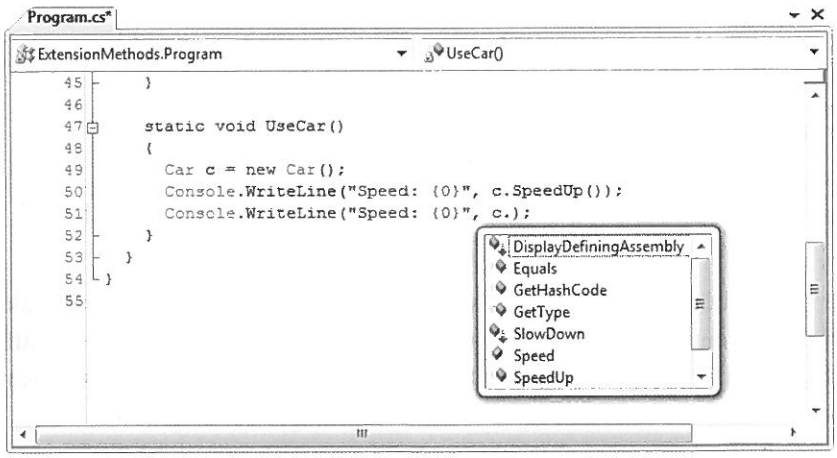
```
// Itt az egyetlen using utasításunk.
using System;

namespace MyNewApp
{
  class JustATest
  {
    void SomeMethod()
    {
```

```
        // Hiba! Importálni kell a MyExtensionMethods
        // névteret az int Foo() metódussal való kiterjesztéséhez.
        int i = 0;
        i.Foo();
      }
    }
  }
```

IntelliSense a bővítő metódusokhoz

Mivel a bővítő metódusok nem pontosan azon a típuson vannak definiálva, amelyet kibővítenek, zavaróak lehetnek a meglévő kódbázis vizsgálatakor. Tegyük fel például, hogy egy olyan névteret importálunk, amely néhány, nem általunk írt bővítő metódust definiál. A kódunk írása közben létrehozhatunk egy kibővített típusú változót, alkalmazzuk rá a pont operátort, és tucatnyi olyan új metódussal találhatjuk szemben magunkat, amelyek nem tagjai az eredeti osztálydefiníciónak.

A Visual Studio IntelliSense mechanizmusa az összes bővítő metódust megjelöli egy egyedi „lefelé nyíl" ikonnal (lásd 13.3. ábra), amely a képernyőn kék színnel jelenik meg.



13.3. ábra: *Az Extension metódusok IntelliSense-e*

Minden ilyen vizuális jelzéssel ellátott metódus arra emlékeztet, hogy a metódus az eredeti osztálydefiníción kívül bővítő metódusként lett definiálva.

Forráskód: Az `ExtensionMethods` projekt a 13. fejezet alkönyvtárában található.

Bővítő könyvtárak létrehozása és használata

Az előző példa a különböző típusok (pl. a `System.Int32`) funkcionalitását bővítette ki az aktuális konzolalkalmazásban való használathoz. Ugyanakkor nyilvánvaló, hogy milyen hasznos lehet egy olyan, számos bővítést tartalmazó .NET-kódkönyvtár létrehozása, amelyre több alkalmazás is hivatkozhat. Szerencsére ez nagyon könnyen megoldható.

Hozzunk létre egy (`MyExtensionsLibrary` nevű) osztálykönyvtár-projektet. Ezután nevezzük át az eredeti C#-kódfájlunkat `MyExtensions.cs`-re, és másoljuk át a `MyExtensions` osztálydefiníciót az új névterünkbe:

```
namespace MyExtensionsLibrary
{
  // Ne felejtsük el importálni a System.Reflection-t.
  public static class MyExtensions
  {
    // Ugyanúgy valósítjuk meg, mint korábban.
    public static void DisplayDefiningAssembly(this object obj)
    {...}
    // Ugyanúgy valósítjuk meg, mint korábban.
    public static int ReverseDigits(this int i)
    {...}
  }
}
```

Megjegyzés Ha egy .NET-kódkönyvtárból bővítő metódusokat akarunk exportálni, a definiáló típust publikusan kell deklarálnunk (a típusoknak belső alapértelmezett hozzáférés-módosítója van).

Ekkor lefordíthatjuk a könyvtárunkat, és hivatkozhatunk a `MyExtensionsLibrary.dll` szerelvényre az új .NET-projektben. A `System.Object` és a `System.Int32` új funkcionalitását bármelyik alkalmazásban használhatjuk, amely hivatkozik a könyvtárra.

Ennek bemutatására hozzunk létre egy (`MyExtensionsLibraryClient` nevű) parancssorialkalmazás-projektet. Ezután hivatkozzunk a `MyExtensionsLibrary.dll` szerelvényre. A kiindulási kódfájlunkban adjuk meg, hogy a `MyExtensionsLibrary` névteret használjuk, és írjunk valami egyszerű kódot, amely egy lokális egész számra meghívja az új metódusokat:

```
using System;
// Importáljuk az egyedi névterünket.
using MyExtensionsLibrary;
```

```
namespace MyExtnesionsLibraryClient
{
  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Using Library with Extensions
        *****\n");
      // Ezúttal ezek a bővítő metódusok
      // egy külső .NET-osztálykönyvtárban
      // lettek definiálva.
      int myInt = 987;
      myInt.DisplayDefiningAssembly();
      Console.WriteLine("{0} is reversed to {1}",
        myInt, myInt.ReverseDigits());
      Console.ReadLine();
    }
  }
}
```

Ajánlatos a bővítő metódusokkal rendelkező típusokat dedikált szerelvénybe helyezni (egy dedikált névtérben). Ennek oka egyszerűen a programozási környezet túlzsúfolásának csökkentése. Ha például egy olyan alapkönyvtárat kellene létrehoznunk a cégünk számára, amelyet minden alkalmazás használ, és az adott könyvtár alapértelmezett névtere 30 bővítő metódust definiál, a végeredmény az lenne, hogy ezek a metódusok minden alkalmazáskor felbukkannának az IntelliSense-ben (még akkor is, ha nincs rájuk szükség).

Forráskód A `MyExtensionsLibrary` és a `MyExtensionsLibraryClient` projekt megtalálható a 13. fejezet alkönyvtárában.

Interfésztípusok kiterjesztése bővítő metódusokkal

A fentiekben azt vizsgáltuk meg, hogy hogyan lehet az osztályokat (és indirekt módon az ugyanilyen szintaxist követő struktúrákat) a bővítő metódusok segítségével új funkcionalitásokkal ellátni. Lehetőség van azonban arra is, hogy az interfésztípusokat új metódusokkal bővítsük; ám az ilyen műveletek szemantikája a várttól némileg eltérő lesz.

Hozzunk létre egy `InterfaceExtensions` nevű új parancssori alkalmazást, és definiáljunk egy egyszerű interfésztípust (`IBasicMath`), amely egyetlen `Add()` nevű metódust tartalmaz. Ezután valósítsuk meg ezt az interfészt egy osztálytípuson (`MyCalc`). Például:

```
// Normál CLR-interfész definiálása C#-ban
interface IBasicMath
{
  int Add(int x, int y);
}

// A IBasicMath megvalósítása
class MyCalc : IBasicMath
{
  public int Add(int x, int y)
  {
    return x + y;
  }
}
```

Tegyük fel, hogy nincs hozzáférésünk az `IBasicMath` definíciós kódjához, de egy új tagot szeretnénk felvenni (pl. kivonásmetódust) viselkedésének bővítésére. Megpróbálhatjuk a következő bővítő osztállyal:

```
static class MathExtensions
{
  // Az IBasicMath kibővítése megvalósítással
  public static int Subtract(this IBasicMath itf,
    int x, int y);
}
```

Ez azonban fordítási hibákat eredményez. Amikor egy interfészt új tagokkal bővítünk, akkor *meg kell adni* az ilyen tagok egy *implementációját* is. Ez látszólag ellentmondásban áll az interfésztípusok természetével, mivel az interfészek nem megvalósításokat, csak definíciókat adnak. Mégis, a `MathExtensions` osztályunkat a következőképpen kell definiálnunk:

```
static class MathExtensions
{
  // Az IBasicMath this metódusának és this
  // megvalósításának
  public static int Subtract(this IBasicMath itf,
    int x, int y)
  {
    return x - y;
  }
}
```

Ekkor feltételezhetjük, hogy létre tudunk hozni egy `IBasicMath` típusú változót, és közvetlenül meg tudjuk hívni a `Subtract()` metódust. Ez azonban nem lehetséges. A valóságban voltaképpen annyit állítottunk, hogy „a projekten belül bármelyik osztály, amelyik megvalósítja az `IBasicMath` interfészt, rendelkezik egy így megvalósított `Subtract()` metódussal". Az összes alapvető

szabály érvényben marad ekkor is, ezért a `MyCalcot` definiáló névtérnek hozzáféréssel kell rendelkeznie a `MathExtensionst` definiáló névtérhez. Vizsgáljuk meg a következő Main() metódust:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Extending an interface *****\n");

  // Az IBasicMath tagok meghívása a MyCalc objektumból.
  MyCalc c = new MyCalc();
  Console.WriteLine("1 + 2 = {0}", c.Add(1, 2));
  Console.WriteLine("1 - 2 = {0}", c.Subtract(1, 2));

  // A bővítés meghívásához kasztolhatjuk IBasicMath típussá is.
  Console.WriteLine("30 - 9 = {0}",
    ((IBasicMath)c).Subtract(30, 9));

  // Ez nem működik!
  // IBasicMath itfBM = new IBasicMath();
  // itfBM.Subtract(10, 10);
  Console.ReadLine();
}
```

A bővítő metódusok tehát nagyon hasznosak lehetnek bármikor, ha egy típus funkcionalitását akarjuk kibővíteni, még ha nem is rendelkezünk hozzáféréssel az eredeti forráskódhoz a polimorfizmus érdekében. Az implicit típusú lokális változókhoz nagyon hasonlóan, a bővítő metódusok kulcsszerepet töltenek be a LINQ API használatakor. A következőkben látni fogjuk, hogy az alaposztálykönyvtárak számos meglévő típusa új funkcionalitással egészül ki (bővítő metódusok segítségével) ahhoz, hogy integrálható legyen a LINQ programozási modellbe.

Forráskód Az `InterfaceExtension` projekt megtalálható a 13. fejezet alkönyvtárában.

Részleges metódusok

A .NET 2.0 megjelenése óta tudunk részleges osztályokat definiálni a `partial` kulcsszó segítségével (lásd 5. fejezet). Ez a szintaxis lehetővé teszi egy típus teljes megvalósításának szétosztását több kódfájlban (vagy eltérő helyeken, pl. a memóriában). Amíg a részleges típus minden része ugyanazzal a teljesen meghatározott névvel rendelkezik, a végeredmény egyetlen „normális" lefordított osztálytípus lesz a létrehozott szerelvényben.

A C# 2008 kibővíti a `partial` kulcsszó hatókörét abban az értelemben, hogy már metódusszinten is alkalmazható. Röviden: lehetővé teszi egy metódus prototípusának létrehozását az egyik fájlban, valamint a megvalósítását egy másikban. Ha C++-tapasztalatokkal rendelkezünk, ez a C++ header/implementáció fájl elvére emlékeztethet bennünket; viszont a C# részleges metódusaira több fontos korlátozás érvényes:

- Részleges metódusokat csak részleges osztályokban lehet definiálni.
- A részleges metódusok visszatérési értéke mindig `void`.
- A részleges metódusok statikusak vagy példányszintűek lehetnek.
- A részleges metódusoknak lehetnek argumentumaik (többek között `this`, `ref` vagy `params` kulcsszóval, de soha nem az `out` módosítóval jelölve).
- A részleges metódusok mindig impliciten privát metódusok.

Még furcsább, hogy a részleges metódusok nem feltétlenül kerülnek be a lefordított szerelvénybe.

A részleges metódusok első pillantásra

Egy részleges metódus definíciójának megvalósításához hozzunk létre egy `PartialMethods` nevű parancssorialkalmazás-projektet. A `CarLocator.cs` nevű C#-fájlban definiáljunk egy `CarLocator` nevű osztályt:

```
// CarLocator.cs
partial class CarLocator
{
  // Ez a tag mindig része lesz a
  // CarLocator osztálynak.
  public bool CarAvailableInZipCode(string zipCode)
  {
    // Ez a meghívás része "lehet" a metódus
    // megvalósításának.
    VerifyDuplicates(zipCode);

    // Tegyük fel, hogy itt van valami érdekes adatbázislogika...
    return true;
  }

  // Ez a tag része "lehet" a CarLocator osztálynak!
  partial void VerifyDuplicates(string make);
}
```

A `VerifyDuplicates()` metódust a `partial` módosítóval definiáltuk, de ezen a fájlon belül nem készítettük el a metódus törzsét. A `CarAvailableInZipCode()` metódus ebben a megvalósításban meghívja a `VerifyDuplicates()` metódust.

Ha ezt az alkalmazást jelenlegi formájában lefordítanánk, és a lefordított szerelvényt megnyitnánk egy `ildasm.exe` vagy `reflector.exe` jellegű eszközben, *nyomát sem* látnánk a `VerifyDuplicates()` metódusnak a `CarLocator` osztályban, és *nyoma sem lenne* a `VerifyDuplicates()` meghívásának a `CarAvailableInZipCode()` metódusban. A projekt jelenlegi állásában a fordító szemszögéből nézve *valójában* a `CarLocator` osztály következő definícióját írtuk meg:

```
internal class CarLocator
{
  public bool CarAvailableInZipCode(string zipCode)
  {
    return true;
  }
}
```

A kód ilyen furcsa „megcsonkításának" oka abban keresendő, hogy a részleges `VerifyDuplicates()` metódusunk soha nem kapott valódi megvalósítást. Ha egy új (mondjuk `CarLocatorImpl.cs` nevű) fájllal kiegészítenénk a projektünket, amely definiálná a részleges metódusunk további részét:

```
// CarLocatorImpl.cs
partial class CarLocator
{
  partial void VerifyDuplicates(string make)
  {
    // Tegyük fel, hogy itt valami költséges adatellenőrzés
    // történik...
  }
}
```

azt találnánk, hogy a `CarLocator` osztályban definiált minden kódrészlet figyelembe lenne véve fordítási időben (mint a következő közel teljes C#-kódban látható):

```
internal class CarLocator
{
  public bool CarAvailableInZipCode(string zipCode)
  {
    this.VerifyDuplicates(zipCode);
    return true;
  }
```

```
    private void VerifyDuplicates(string make)
    {
    }
}
```

Amikor a `partial` kulcsszóval definiálunk egy metódust, a fordító az alapján dönti el, hogy ez a metódus része legyen-e a szerelvénynek, hogy rendelkezik-e metódustörzzsel, vagy csupán egy üres szignatúra. Ha nincs metódustörzse, a fordító a metódus minden nyomát (hívások, metaadat-leírások, prototípusok) eltakarítja a fordítás során.

Bizonyos tekintetben a C# részleges metódusai (az `#if`, az `#elif`, az `#else` és az `#endif` előfeldolgozó utasításokat használó metódusok; lásd 12. fejezet) a *feltételes kódfordítás* erősen tipizált változatai. A legfontosabb eltérés az, hogy a részleges metódus a fordítási ciklusban (a felépítés beállításaitól függetlenül) teljesen figyelmen kívül marad akkor, ha nincs hozzátartozó megvalósítás.

A részleges metódusok használata

A részleges metódusokkal járó korlátozások miatt, legfőképpen mert implicit módon privátnak és `void` visszatérési értékűnek kell lenniük, nehéz hasznos felhasználási területet találni ehhez az új nyelvi szolgáltatáshoz. Az igazat megvallva a C# 2008 új nyelvi szolgáltatásai között minden valószínűséggel a részleges metódusokat használjuk a legkevésbé.

Az előző példánkban a `VerifyDuplicates()` metódust részlegesként jelöltük meg; képzeljük el azonban azt, hogy ennek a metódusnak, ha megvalósul, nagyon intenzív számításokat kell elvégeznie.

Azzal, hogy ezt a metódust `partial` módosítóval jelöltük meg, a többi osztályépítőnek *lehetősége* van az implementációra. Ez tisztább megoldás, mintha előfeldolgozó direktívák alapján „üres" implementációkat vagy `NotImplementedException` példányokat dobó kódot készítenénk.

Ezt a szintaxist legáltalánosabban az ún. *könnyű események* definiálására használjuk. Lehetővé teszi az osztálytervezők számára, hogy az eseménykezelőkhöz hasonló metóduskapaszkodókat definiáljanak, amelyet a fejlesztők vagy megvalósítanak, vagy nem. Az elnevezési konvenció szerint az ilyen könnyű eseménykezelők egy `On` prefixet kapnak. Például:

```
// CarLocator.EventHandler.cs
partial class CarLocator
{
  public bool CarAvailableInZipCode(string zipCode)
  {
    ...
    OnZipCodeLookup(zipCode);
    return true;
  }
  ...
  // Egy "könnyű" eseménykezelő.
  partial void OnZipCodeLookup(string make);
}
```

Ha az osztály készítője információt szeretne kapni arról, hogy mikor történik a `CarAvailableInZipCode()` metódus meghívása, akkor megvalósíthatja a `OnZipCodeLookup()` metódust. Ellenkező esetben nem történik semmi.

Forráskód A `PartialMethods` projekt a 13. fejezet alkönyvtárában található.

Az objektuminicializáló szintaxis

A C# 2008 új módszert vezet be az új osztály vagy struktúraváltozók létrehozásához, ezt *objektuminicializáló szintaxis*nak nevezzük. Ezzel a technikával lehetőségünk nyílik néhány sornyi kódban új típusváltozót készíteni és több tulajdonságot és/vagy nyilvános mezőt hozzáadni. Szintaktikailag az objektuminicializáló adott értékek vesszővel elválasztott listájából áll, amelyet a { és a } jelek fognak közre. Az inicializációs lista minden tagja az inicializált objektum egy nyilvános mezőjére vagy nyilvános tulajdonságára képződik le. A szintaxis működésének megvizsgálásához hozzunk létre egy `ObjectInitializers` nevű parancssorialkalmazás-projektet. Gondoljuk végig az eddig létrehozott különböző geometriai típusainkat (`Point`, `Rectangle`, `Hexagon` stb.), vegyük például az egyszerű `Point` típust (amely nem alkalmazta a C# 2008 automatikus tulajdonságait):

```
public class Point
{
  private int xPos, yPos;

  public Point(int x, int y)
  { xPos = x; yPos = y; }
  public Point(){}
```

```
  public int X
  {
    get { return xPos; }
    set { xPos = value; }
  }
  public int Y
  {
    get { return yPos; }
    set { yPos = value; }
  }

  public override string ToString()
  { return string.Format("[{0}, {1}]", xPos, yPos); }
}
```

A C# 2008-ban a következő megközelítések bármelyikével létrehozhatunk `Point` típusokat:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Object Init Syntax *****\n");
  // Point létrehozása az összes tulajdonság kézi beállításával...
  Point firstPoint = new Point();
  firstPoint.X = 10;
  firstPoint.Y = 10;

  // ...vagy Point létrehozása egyedi konstruktor segítségével...
  Point anotherPoint = new Point(20, 20);

  // ...vagy néhány Point típus létrehozása a new
  // objektuminicializáló szintaxis segítségével.
  var yetAnotherPoint = new Point { X = 30, Y = 30 };
  Point finalPoint = new Point { X = 30, Y = 30 };
  Console.ReadLine();
}
```

Az utolsó két `Point` típus (amelyek közül az egyik a példa kedvéért implicit típusú) nem alkalmaz egyedi típusú konstruktort (mint hagyományosan tenné), helyette a nyilvános `X` és `Y` tulajdonság értékét állítja be. A háttérben a típus alapértelmezett konstruktora hívódik meg, amelyet a megadott tulajdonságok értékének beállítása követ. Ezért a `yetAnotherPoint` és a `finalPoint` csupán a (tulajdonságról tulajdonságra haladó) `firstPoint` változó létrehozására használt szintaxis rövidítése.

Ugyanez a szintaxis alkalmazható a típusok nyilvános mezőinek beállítására, amelyet a `Point` jelenleg nem támogat. A példa kedvéért tegyük fel, hogy az `xPos` és az `yPos` tagváltozókat nyilvánosként definiáltuk. Így ezeknek a mezőknek az értékeit a következőképpen állíthatnánk be:

```
var p = new Point {xPos = 2, yPos = 3};
```

Mivel a `Point` most négy nyilvános taggal rendelkezik, a következő szintaxis is lehetséges. Próbáljuk meg kitalálni az `xPos` és az `yPos` tényleges értékét:

```
var p = new Point {xPos = 2, yPos = 3, X = 900};
```

Az `xPos` értéke 900, míg az `yPos` értéke 3. Ebből azt a következtetést vonhatjuk le, hogy az objektuminicializálás balról jobbra halad. Az egyértelműség kedvéért, a `p` előző inicializációja a hagyományos objektumkonstruktor-szintaxissal a következő lenne:

```
Point p = new Point();
p.xPos = 2;
p.yPos = 3;
p.X = 900;
```

Egyedi konstruktor meghívása az inicializációs szintaxissal

Az előző példák a `Point` típust úgy inicializálták, hogy implicit módon meghívták a típus alapértelmezett konstruktorát:

```
// Itt az alapértelmezett konstruktort implicit módon hívjuk meg.
Point finalPoint = new Point { X = 30, Y = 30 };
```

Ha ezt nagyon egyértelművé szeretnénk tenni, az alapértelmezett konstruktort explicit módon is meghívhatjuk a következőképpen:

```
// Itt az alapértelmezett konstruktort explicit módon hívjuk meg.
Point finalPoint = new Point() { X = 30, Y = 30 };
```

Amikor az új inicializáló szintaxissal hozunk létre egy típust, az osztály vagy a struktúra által definiált *bármely* konstruktort meg tudjuk hívni. A `Point` típusunk jelenleg egy kétargumentumos konstruktort definiál az (x, y) pozíció megadására. Ezért a következő `Point` deklaráció eredményeként az `X` értéke 100 és az `Y` értéke is 100 lesz, függetlenül attól, hogy a konstruktor argumentumai a 10 és a 16 értékeket határozták meg:

```
// Egyedi konstruktor meghívása.
Point pt = new Point(10, 16) { X = 100, Y = 100 };
```

A Point típus jelenlegi definíciójával az egyedi konstruktor meghívása nem igazán hasznos az inicializáló szintaxis használata mellett (és kissé hosszadalmas). Ha viszont a Point típus egy új konstruktora a meghívó számára lehetővé teszi egy szín beállítását (a PointColor nevű egyedi felsorolást alkalmazva), az egyedi konstruktor és az objektuminicializáló szintaxis kombinációja egyértelművé válik. Tegyük fel, hogy a következőképpen módosítottuk a Point kódját:

```
public enum PointColor
{ LightBlue, BloodRed, Gold }

public class Point
{
  public int xPos, yPos;
  private PointColor c;

  public Point(PointColor color)
  {
    xPos = 0; yPos = 0;
    c = color;
  }
  public Point(){}
  public Point(int x, int y)
  {
    xPos = x; yPos = y;
    c = PointColor.Gold;
  }
  ...
  public override string ToString()
  { return string.Format("[{0}, {1}, Color = {2}]", xPos,
    yPos, c); }
}
```

Ezzel az új konstruktorral most a következőképpen hozhatunk létre egy arany színű pontot (helye 90, 20):

```
// Egy sokkal érdekesebb egyedi konstruktor meghívása
// az init szintaxisában.
Point goldPoint = new Point(PointColor.Gold){ X = 90, Y = 20 };
Console.WriteLine("Value of Point is: {0}", goldPoint);
```

A belső típusok inicializálása

A „van egy" kapcsolat lehetővé teszi új típusok létrehozását oly módon, hogy létező típusok tagváltozóit definiáljuk (lásd 6. fejezet). Tegyük fel például, hogy van egy Rectangle osztályunk, amely a Point típust használja a bal felső/jobb alsó koordinátáinak a jelölésére.

```
public class Rectangle
{
  private Point topLeft = new Point();
  private Point bottomRight = new Point();

  public Point TopLeft
  {
    get { return topLeft; }
    set { topLeft = value; }
  }
  public Point BottomRight
  {
    get { return bottomRight; }
    set { bottomRight = value; }
  }

  public override string ToString()
  {
    return string.Format("[TopLeft: {0}, {1}, BottomRight: {2},
      {3}]", topLeft.X,
    topLeft.Y, bottomRight.X, bottomRight.Y);
  }
}
```

Az objektuminicializáló szintaxis segítségével a következőképpen hozhatunk létre egy új `Rectangle` típust, és állíthatjuk be a belső pontokat:

```
// Téglalap létrehozása és inicializálása.
Rectangle myRect = new Rectangle
{
  TopLeft = new Point { X = 10, Y = 10 },
  BottomRight = new Point { X = 200, Y = 200}
};
```

Az új szintaxis előnye abban rejlik, hogy alapvetően csökkenti a gépelési időt (feltéve, hogy van egy megfelelő konstruktorunk). Egy hasonló `Rectangle` létrehozására a hagyományos verzió a következő:

```
// A régi megközelítés.
Rectangle r = new Rectangle();
Point p1 = new Point();
p1.X = 10;
p1.Y = 10;
r.TopLeft = p1;
Point p2 = new Point();
p2.X = 200;
p2.Y = 200;
r.BottomRight = p2;
```

Gyűjteményinicializálók

Az objektuminicializáló szintaxis elvéhez szorosan kapcsolódik a *gyűjtemények inicializálása*. Ez az egyszerű tömbökéhez hasonló szintaxis lehetővé teszi egy tároló (például `ArrayList` vagy `List<T>`) feltöltését elemekkel. Vizsgáljuk meg a következő példákat:

```
// A normál tömb inicializálása.
int[] myArrayOfInts = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };

// Az egész számok általános listájának inicializálása.
List<int> myGenericList = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7,
  8, 9 };

// Az ArrayList inicializálása numerikus adatokkal.
ArrayList myList = new ArrayList { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
```

Ha a tárolónk objektumtípusok gyűjteményét kezeli, az objektuminicializáló szintaxis és a gyűjteményinicializáló szintaxis keverésével a következőt kapjuk:

```
List<Point> myListOfPoints = new List<Point>
{
  new Point { X = 2, Y = 2 },
  new Point { X = 3, Y = 3 },
  new Point(PointColor.BloodRed){ X = 4, Y = 4 }
};

foreach (var pt in myListOfPoints)
{
  Console.WriteLine(pt);
}
```

Ennek az előnye megint csak az, hogy gépelést spórolunk meg vele. Bár a beágyazott kapcsos zárójelek nehezen olvashatók, ha nem ügyelünk a formázásra, az azonban egyértelmű, hogy nagyon sok gépeléssel járna a következő `List<T>` `Rectangle` lista feltöltése, ha nem lenne a gyűjteményinicializáló szintaxis.

```
List<Rectangle> myListOfRects = new List<Rectangle>
{
  new Rectangle {TopLeft = new Point { X = 10, Y = 10 },
    BottomRight = new Point { X = 200, Y = 200}},
  new Rectangle {TopLeft = new Point { X = 2, Y = 2 },
    BottomRight = new Point { X = 100, Y = 100}},
  new Rectangle {TopLeft = new Point { X = 5, Y = 5 },
    BottomRight = new Point { X = 90, Y = 75}}
};
```

```
foreach (var r in myListOfRects)
{
  Console.WriteLine(r);
}
```

Forráskód Az `ObjectInitializers` projekt a 13. fejezet alkönyvtárában található.

Névtelen típusok

Objektumorientált programozóként tisztában vagyunk az olyan osztályok definiálásának az előnyeivel, amelyek egy adott programozási entitás állapotát és funkcionalitását jelölik. Egyértelmű, hogy amikor több projektben újrahasználható osztályt kell definiálnunk, amely számos funkcionalitást kínál különböző metódusai, eseményei, tulajdonságai és egyedi konstruktorai révén, akkor egy új C#-osztály létrehozása a szokványos és gyakran kötelező gyakorlat.

Vannak viszont olyan alkalmak is a programozásban, amikor egy osztályt csupán azért szeretnénk definiálni, hogy egy sor beágyazott (és valamelyest kapcsolódó) adatot modellezzünk bármiféle kapcsolódó metódus, esemény vagy egyéb egyedi működés nélkül. Ráadásul mi van akkor, ha ezt a típust csak az aktuális alkalmazáson belül akarjuk használni az újrahasznosítás szándéka nélkül? Ha ilyen „ideiglenes" típusra volt szükségünk, a C# korábbi verzióinál manuálisan kellett az új osztályt megépítenünk:

```
internal class SomeClass
{
// Definiáljuk a privát tagváltozókat...

// Minden egyes tagváltozóhoz létrehozzuk a tulajdonságokat...

// Minden tagváltozóhoz felüldefiniáljuk a ToString() metódust...

// Felüldefiniáljuk a GetHashCode() és Equals() metódusokat
// az értékalapú egyenlőség kezeléséhez...
}
```

Bár az ilyen osztályok létrehozásához nincs szükség túl nagy tudásra, igencsak munkaigényes lehet, ha egynél több tagot szeretnénk beágyazni (még akkor is, ha az automatikus tulajdonságok ebben a tekintetben nagy segítséggel járnak). A C# 2008-ban már rendelkezésünkre áll éppen az ilyen esetekre egy *névtelen típus* elnevezésű rövidítési mód, amely sok tekintetben a C# (11. fejezetben vizsgált) névtelenmetódus-szintaxisának a kiterjesztése.

Névtelen típusok definiálásakor az új `var` kulcsszót az előbb vizsgált objektuminicializáló szintaxissal együttesen alkalmazzuk. Ennek bemutatására hozzunk létre egy `AnonymousTypes` nevű parancssorialkalmazás-projektet. Egészítsük ki a `Main()` metódust a következő névtelen osztállyal, amely egy egyszerű autót modellez:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Anonymous Types *****\n");

  // Névtelen típus létrehozása autó jelölésére.
  var myCar = new { Color = "Bright Pink", Make = "Saab",
    CurrentSpeed = 55 };

  // Mutassuk meg a színét és a típusát.
  Console.WriteLine("My car is a {0} {1}.", myCar.Color,
    myCar.Make);
  Console.ReadLine();
}
```

A `myCar` változó ekkor implicit típusú, s ez logikus is, hiszen nem egy erősen tipizált osztálydefinícióval modellezzük az autó fogalmát. Fordítási időben a C#-fordító automatikusan létrehoz egy egyedi elnevezésű osztályt. Mivel ez az osztálynév nem látható a C#-ban, a `var` kulcsszóval jelölt implicit típus alkalmazása kötelező lesz.

Az objektuminicializáló szintaxissal meg kell határoznunk azt a tulajdonságkészletet, amellyel a beágyazni próbált adatokat modellezzük. Ha már definiálva vannak, ezek az értékek a szabványos C# tulajdonsághívási szintaxis alkalmazásával érhetők el.

A névtelen típusok belső ábrázolása

Minden névtelen típus automatikusan a `System.Object` leszármazottja, és így az ősosztály által biztosított összes tagot támogatja. Ennek köszönhetően a `ToString()`, a `GetHashCode()`, az `Equals()` vagy a `GetType()` meghívható az `implicit` típusú `myCar` objektumra. Tegyük fel, hogy a `Program` osztályunk a következő statikus segédfüggvényt definiálja:

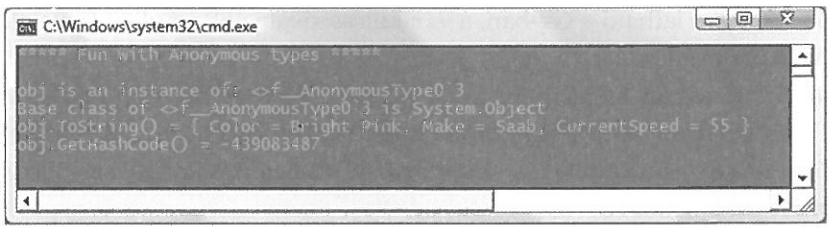
```
static void ReflectOverAnonymousType(object obj)
{
  Console.WriteLine("obj is an instance of: {0}",
    obj.GetType().Name);
```

```
    Console.WriteLine("Base class of {0} is {1}",
      obj.GetType().Name, obj.GetType().BaseType);
    Console.WriteLine("obj.ToString() = {0}", obj.ToString());
    Console.WriteLine("obj.GetHashCode() = {0}", obj.GetHashCode());
    Console.WriteLine();
}
```

Ekkor tételezzük fel, hogy meghívjuk ezt a metódust a `Main()`-ből, a `myCar` objektumot adva át paraméterként:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Anonymous types *****\n");

  // Névtelen típus létrehozása autó jelölésére.
  var myCar = new {Color = "Bright Pink", Make = "Saab",
    CurrentSpeed = 55};

  // Nézzük meg, mit állított elő a fordító.
  ReflectOverAnonymousType(myCar);
  Console.ReadLine();
}
```

Ellenőrizzük a kimenetet a 13.4. ábrán.



13.4. ábra: *A névtelen típusokat egy, a fordító által generált osztálytípus képviseli*

A `myCar` objektum típusa `<>f_AnonymousTypeCT'` (a név ettől eltérő is lehet). Ez a típusnév kizárólag a fordítótól függ, és nem érhető el közvetlenül a C#-kódban.

Még ennél is fontosabb, hogy az objektuminicializáló szintaxissal definiált minden név/érték pár egy ugyanolyan nevű írásvédett tulajdonságra és az ennek megfelelő privát írásvédett segédmezőre képződik le. A következő C#-kód a fordító által generált osztályt közelíti a `myCar` objektum jelölésére (ezt megint csak egy `reflector.exe` vagy `ildasm.exe` jellegű eszközzel ellenőrizhetjük).

```
internal sealed class <>f__AnonymousType0<<Color>j__TPar,
  <Make>j__TPar, <CurrentSpeed>j__TPar>
{
  // Írásvédett mezők
  private readonly <Color>j__TPar <Color>i__Field;
  private readonly <CurrentSpeed>j__TPar <CurrentSpeed>i__Field;
  private readonly <Make>j__TPar <Make>i__Field;

  // Alapértelmezett konstruktor
  public <>f__AnonymousType0(<Color>j__TPar Color,
    <Make>j__TPar Make, <CurrentSpeed>j__TPar CurrentSpeed);

  // Felülírt metódusok
  public override bool Equals(object value);
  public override int GetHashCode();
  public override string ToString();

  // Írásvédett tulajdonságok
  public <Color>j__TPar Color { get; }
  public <CurrentSpeed>j__TPar CurrentSpeed { get; }
  public <Make>j__TPar Make { get; }
}
```

A ToString() és a GetHashCode() megvalósítása

Minden névtelen típus automatikusan a `System.Object` leszármazottja, és rendelkezik az `Equals()`, a `GetHashCode()` és a `ToString()` felüldefiniált verziójával. A `ToString()` megvalósítása egyszerűen egy sztringet hoz létre minden név/érték párból. Például:

```
public override string ToString()
{
  StringBuilder builder = new StringBuilder();
  builder.Append("{ Color = ");
  builder.Append(this.<Color>i__Field);
  builder.Append(", Make = ");
  builder.Append(this.<Make>i__Field);
  builder.Append(", CurrentSpeed = ");
  builder.Append(this.<CurrentSpeed>i__Field);
  builder.Append(" }");
  return builder.ToString();
}
```

A `GetHashCode()` megvalósítása az egyes névtelen típusok tagváltozóiból számít ki egy hashértéket. A `System.Collections.Generic.EqualityComparer<T>` típus ezt a hashértéket használja bemenetként. A `GetHashCode()`-nak ezt a megvalósítását használva két névtelen típus akkor (és csak akkor) adja ugyanazt a

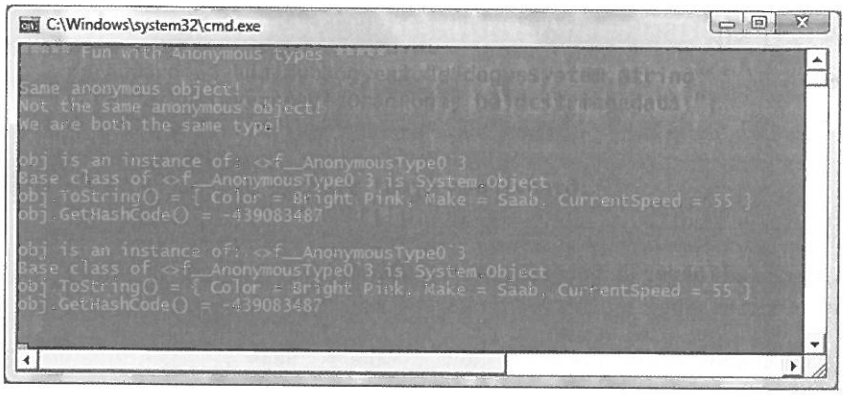
hashértéket, ha azonos tulajdonságkészlettel rendelkeznek, és ugyanazokat az értékeket rendeltük hozzájuk. Az ilyen megvalósítás miatt a névtelen típusok tökéletesen megfelelnek a `Hashtable` típusú tárolóban történő tárolásra.

Az egyenlőség szemantikája névtelen típusoknál

Míg a `ToString()` és a `GetHashCode()` metódusok felüldefiniált megvalósítása meglehetősen egyértelmű, az `Equals()` metódus megvalósítása érdekes lehet. Vajon, ha például két „névtelen autó" változót definiálnánk, amelyek ugyanazt a név/érték párt határozzák meg, akkor ezt a két változót egyenlőnek tekintjük, vagy sem? Ehhez egészítsük ki a `Program` típusunkat a következő új metódussal:

```
static void EqualityTest()
{
  // 2 névtelen osztály létrehozása azonos név/érték párral.
  var firstCar = new { Color = "Bright Pink", Make = "Saab",
    CurrentSpeed = 55 };
  var secondCar = new { Color = "Bright Pink", Make = "Saab",
    CurrentSpeed = 55 };

  // Egyenlőnek tekinthetők az Equals() használatakor?
  if (firstCar.Equals(secondCar))
    Console.WriteLine("Same anonymous object!");
  else
    Console.WriteLine("Not the same anonymous object!");

  // Egyenlőnek tekinthetők az == használatakor?
  if (firstCar == secondCar)
    Console.WriteLine("Same anonymous object!");
  else
    Console.WriteLine("Not the same anonymous object!");

  // Megegyezik az objektumok alaptípusa?
  if (firstCar.GetType().Name == secondCar.GetType().Name)
    Console.WriteLine("We are both the same type!");
  else
    Console.WriteLine("We are different types!");

  // Az összes részlet megjelenítése.
  Console.WriteLine();
  ReflectOverAnonymousType(firstCar);
  ReflectOverAnonymousType(secondCar);
}
```

Ha ezt a metódust meghívjuk a `Main()`-ben, a 13.5. ábra mutatja (a kissé meglepő) kimenetet.



13.5. ábra: *Az névtelen típusok egyenlősége*

Ha lefuttatjuk ezt a tesztkódot, látható, hogy az első feltételes ellenőrzés, ahol az `Equals()` metódust hívjuk meg, igaz értéket ad vissza, tehát a „Same anonymous object!" (Azonos névtelen objektumok!) üzenet jelenik meg a képernyőn. Ennek az a magyarázata, hogy a fordító által előállított `Equals()` metódus *értékalapú szemantikát* alkalmaz az egyenlőség ellenőrzésekor (vagyis az összehasonlítandó objektumok egyes mezőinek értékét ellenőrzi).

A második feltételes teszt esetén (amely a == C# egyenlőségvizsgáló operátort használja) viszont a „Not the same anonymous object!" (Nem azonos névtelen objektumok!) üzenet jelenik meg, és ez első pillantásra egy kissé érthetetlennek tűnhet. Az ok, hogy a névtelen típus *nem rendelkezik* a C# egyenlőségvizsgáló operátorok (== és `!=`) túlterhelt verziójával. Így a névtelen típusok egyenlőségének C# egyenlőségvizsgáló operátorokkal (és nem az `Equals()` metódussal) történő ellenőrzésekor a *referenciáknak* és nem az objektum által fenntartott értékeknek az egyenlőségét vizsgáljuk. Minden típus esetén ez az alapértelmezett viselkedés (lásd 12. fejezet) addig, amíg közvetlenül nem terheljük túl az operátorokat a kódunkban (ez névtelen típusok esetén nem lehetséges, mivel nem definiáljuk a típust).

Végül, az utolsó összehasonlítást végző tesztünkben (amelyben az alaptípus nevét vizsgáljuk) azt tapasztaljuk, hogy a névtelen típusok ugyanannak a fordító által generált osztálytípusnak (ebben az esetben az `AnonymousTypeCT`') a példányai, hiszen a `firstCar` és a `secondCar` ugyanolyan tulajdonságokkal rendelkezik (`Color`, `Make` és `CurrentSpeed`).

Ez egy lényeges, de apró részletet illusztrál: a fordító csak akkor generál új osztálydefiníciót, ha a névtelen típus *egyedi* neveket tartalmaz. Így ha azonos `anonymous` típusokat (ez megint csak azt jelenti, hogy azonos nevűeket) deklarálunk ugyanabban a szerelvényben, a fordító csak egyetlen névtelen típusdefiníciót generál.

Névtelen típusokat tartalmazó névtelen típusok

Névtelen típust másik névtelen típusból is létre tudunk hozni. Tegyük fel például, hogy egy megrendelést szeretnénk modellezni, amely időbélyeget, árat és a megvásárolt autót tartalmazza. Egy új (sokkal kifinomultabb) névtelen típus az ilyen elemek jelölésére a következő:

```
// Névtelen típus létrehozása egy másikból.
  var purchaseItem = new {
    TimeBought = DateTime.Now,
    ItemBought = new {Color = "Red", Make = "Saab",
    CurrentSpeed = 55}, Price = 34.000};

ReflectOverAnonymousType(purchaseItem);
```

Kérdés, hogy hol (és mikor) használhatjuk ezt az új nyelvi szolgáltatást. A névtelen típusok deklarálásával csínján kell bánni, tipikusan csak a LINQ technológiakészlet alkalmazásakor használatosak (lásd a 14. fejezetet). Soha ne akarjunk lemondani az erősen tipizált osztályok/struktúrák használatáról csupán kedvtelésből, hiszen a névtelen típusok rengeteg korlátozással járnak, többek között a következőkkel:

- Nem szabályozhatjuk a névtelen típusok nevét.
- A névtelen típusok mindig a `System.Object` kiterjesztései.
- A névtelen típus mezői és tulajdonságai mindig írásvédettek.
- A névtelen típusok nem támogatják az eseményeket, az egyedi metódusokat, az egyedi operátorokat vagy az egyedi felüldefiniálásokat.
- A névtelen típusok mindig implicit módon zártak.
- A névtelen típusok mindig az alapértelmezett konstruktor használatával jönnek létre.

Viszont a LINQ technológiakészlettel való programozás során gyakran tapasztaljuk majd, hogy ez a szintaxis nagyon hasznos lehet olyan esetekben, amikor egy elem *entitásjellegét* és nem a funkcionalitását szeretnénk gyorsan modellezni.

Forráskód Az `AnonymousTypes` projekt megtalálható a 13. fejezet alkönyvtárában.

Összefoglalás

A C# 2008 több olyan érdekes szolgáltatással rendelkezik, amelynek hatására a C# a *funkcionális nyelvek* csoportjába sorolható. Ebben a fejezetben végignéztük az összes alapvető módosítást, az implicit típusú lokális változóktól kiindulva. Bár a lokális változók túlnyomó többségét nem kell a `var` kulcsszóval deklarálni, ha mégis így teszünk, nagymértékben egyszerűsödik a LINQ technológiacsaláddal az együttműködés (lásd következő fejezet).

Megvizsgáltuk továbbá az automatikus tulajdonságok szerepét, a részleges metódusokat, a bővítő metódusokat (amelyekkel új funkcionalitást adhatunk a lefordított típusoknak), valamint az objektuminicializálás szintaxisát (amely arra használható, hogy a létrehozás idején adjuk meg a tulajdonságok értékeit).

A fejezet a névtelen típusok használatának tanulmányozásával zárult. Ez a nyelvi szolgáltatás a funkcionalitás helyett a típusok „formájának" definiálását teszi lehetővé. Nagyon jó szolgálatot tehet olyankor, amikor limitált használhatóságú típust kell modelleznünk a programban, mert a munka jelentős részét áthárítja a fordítóra.

TIZENNEGYEDIK FEJEZET

A LINQ. Bevezetés

Az előző fejezet számos C# 2008 programozási konstrukciót mutatott be. A különböző eszközök, mint az implicit típusú lokális változók, a névtelen típusok, az objektuminicializáló szintaxis és a lambda kifejezések (ezeket lásd a 11. fejezetben) használatával funkcionális C#-kódot írhatunk. Noha az eszközök közül sok közvetlenül úgy is használható, ahogy van, előnyük leginkább akkor látványos, ha a LINQ (Language Integrated Query - nyelvbe ágyazott lekérdezés) technológián belül használjuk őket.

Az alábbiakban a LINQ-modellt és annak szerepét vizsgáljuk meg a .NET platformon belül. Megtanuljuk a *lekérdezésoperátorok* és a *lekérdező kifejezések* szerepét. Ezek segítségével lehetőségünk nyílik olyan utasítások definiálására, amelyek lekérdezik az adatforrást, és az erdeményhalmazzal térnek visszsza. Elkészítünk számos LINQ-példát, amelyek tömbben tárolt adatokkal, illetve különböző gyűjteménytípusokkal (akár generikus akár nem generikus) dolgoznak, továbbá megismerjük a LINQ technológiát támogató szerelvényeket és típusokat is.

Megjegyzés A második kötetben helyet kapó 24. fejezet további LINQ-centrikus API-kat ismertet, amelyek révén relációs adatbázisokkal és XML-dokumentumokkal dolgozhatunk.

A LINQ szerepe

Szoftverfejlesztőként programozási időnk legnagyobb részét adatok kinyerésével és manipulálásával töltjük. Ha azt mondjuk, adat, leginkább valamilyen relációs adatbázisban tárolt információra gondolunk. Egy másik közkedvelt adattárolási helyet az XML-dokumentumok jelentenek (`*.config` fájlok, lokálisan tárolt `DataSetek`, memóriában tárolt, XML-webszolgáltatások által visszaadott adatok stb.).

A fentieken kívül még számos más helyen tárolható adat. Van például egy 300 integert tároló generikus `List<T>` típusunk, amelynek egy részhalmazát szeretnénk megkapni valamilyen feltétel mentén (pl. csak a páros vagy párat-

lan tagokat, csak a prímszámokat, csak az olyan nem ismétlődő számokat, amelyek nagyobbak 50-nél stb.). Vagy mondjuk a reflexió szolgáltatásait használjuk, és azoknak az osztályoknak a metadatára van szükségünk, amelyek, egy `Types` tömbben szereplő bizonyos szülőosztályoktól származnak. Valójában az adat mindenütt jelen van.

A .NET 3.5 előtt bizonyos adatok kezelésére a programozóknak különféle API-kat kellett használniuk. A 14.1. táblázat néhány általánosan használt API-t mutat be különböző adattípusok eléréséhez.

A kívánt adat	Hogyan érjük el?
Relációs adat	`System.Data.dll`, `System.Data.SqlClient.dll`, stb.
XML-dokumentum	`System.Xml.dll`
Metaadat-táblázatok	`System.Reflection` névtér
Objektumok gyűjteménye	`System.Array` és a `System.Collections`/`System.Collections.Generic` névterek

14.1. táblázat: *Lehetőségek különböző típusú adatok kezelésére*

Természetesen ezek az adatkezelési megközelítések is megfelelőek. A .NET 3.5/C# 2008 programozása közben tudjuk (és fogjuk is) használni az ADO.NET-et, az XML-névtereket, a reflexiós szolgáltatásokat és a különböző gyűjteménytípusokat. A probléma csak az, hogy ezen API-k mindegyike önmagában nagyon kevéssé támogatja az integrációt. Igaz, hogy lehetőség van rá például, hogy egy ADO.NET `DataSetet` XML-be mentsünk, és utána a `System.Xml` névtér segítségével kezeljük az adatokat, ám az adatkezelés aszimmetrikus marad.

A LINQ API megpróbál konzisztens, *szimmetrikus* lehetőséget biztosítani a programozóknak, hogy az „adatokat" – a szó legtágabb értelmében – elérjék és manipulálják. A LINQ használatával lehetőségünk nyílik közvetlenül a C# programozási nyelven belül *lekérdező kifejezéseknek* nevezett entitások létrehozására. Ezek a lekérdező kifejezések számos *lekérdezésoperátorra* támaszkodnak, amelyeket szándékosan úgy terveztek, hogy nagyon hasonlítsanak az SQL-kifejezésekre (de nem teljesen azonosak velük).

A probléma azonban az, hogy a lekérdező kifejezés számos adattípus kezelésére használható – olyan adatok kezelésére is, amelyeknek semmi köze a relációs adatbázishoz.

A LINQ lehetővé teszi olyan lekérdező kifejezések létrehozását, amelyek egységesen manipulálják a `DataSeteket`, a relációs adatbázisokat, az XML-kifejezéseket, illetve minden olyan típust, amely implementálja az `IEnumerable<T>` interfészt (akár közvetlenül, akár bővítő metódusok segítségével).

Megjegyzés A „LINQ" tulajdonképpen egy olyan kifejezés, amellyel az adatkezelés általános megközelítését írjuk le. A LINQ to Objects nem más, mint a LINQ olyan objektumok felett, amelyek implementálják az `IEnumerable<T>` típust, a LINQ to SQL ugyanígy a LINQ a relációs adatok felett, a LINQ to DataSet a LINQ to SQL-nél egy bővebb halmazt jelöl, a LINQ to XML pedig a LINQ az XML-dokumentumok felett. Valószínűleg lesznek majd további API-k is, amelyeket elláttak LINQ-funkcionalitásokkal (a Microsoftnál már fejlesztenek egyéb LINQ-központú projekteket).

A LINQ-kifejezések jellemzői (tipizáltság és kiterjeszthetőség)

A LINQ lekérdező kifejezések (nem úgy, mint a tradicionális SQL-utasítások) *erősen tipizáltak*. Ezért a C#-fordító ellenőrzi, hogy ezek a kifejezések szintaktikailag jól formáltak-e. A lekérdező kifejezéseknek létezik egy metaadat-reprezentációja az őket használó szerelvényen belül. A különböző eszközök, mint például a Visual Studio 2008, hatékonyan fel tudják használni ezt a metaadatot például az IntelliSense, az automatikus kiegészítés vagy egyéb hasonló támogatások számára.

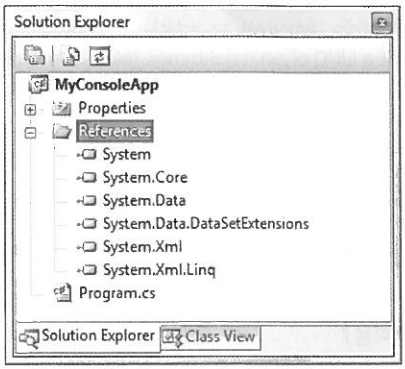
A LINQ-t bővíthető technológiaként alakították ki. Az első kiadásának az a célja, hogy kezelni tudja a relációs adatbázisokat/adathalmazokat, az XML-dokumentumokat és az `IEnumerable<T>` interfészt implementáló objektumokat, de a bővítő metódusok segítségével bárkinek lehetősége van új lekérdezésoperátorok létrehozására (vagy a meglévők felüldefiniálására) és így más adatforrások kezelésére.

Megjegyzés A fejezet tanulmányozása előtt ismételjük át a 13. fejezet anyagát (amely a C# 2008 specifikus szerkezeteit mutatja be). A LINQ ugyanis nagyon sok új C#-eszközt használ a kódolás leegyszerűsítésére.

Alapvető LINQ-szerelvények

A Visual Studio 2008 New Project dialógusablakában a jobb felső sarokban található legördülő listamező segítségével lehetőség van kiválasztani, hogy a .NET platform melyik verziójával szeretnénk a projektet fordítani (lásd 2. fe-

jezet). Ha a .NET keretrendszer 3.5-öt választjuk, akkor minden projektsablon automatikusan hivatkozni fog a kulcsfontosságú LINQ-szerelvényekre. Ha például éppen egy új .NET 3.5 konzolalkalmazást készítünk, a 14.1. ábrán mutatott szerelvényeket találjuk majd a Solution Explorer ablakunkban.



14.1. ábra: *A .NET 3.5 projekttípusok automatikusan hivatkoznak a kulcsfontosságú LINQ-szerelvényekre*

A 14.2. táblázat az alapvető LINQ-specifikus szerelvények szerepét mutatja be.

Szerelvény	Valós jelentés
`System.Core.dll`	Olyan típusokat definiál, amelyek az alapvető LINQ API-t reprezentálják. Ez az egyetlen szerelvény, amelyhez mindenképpen hozzá kell férnünk.
`System.Data.Linq.dll`	A LINQ használatát teszi lehetővé relációs adatbázissal (LINQ to SQL).
`System.Data.DataSetExtensions.dll`	Néhány olyan típust definiál, amelyek integrálják az ADO.NET típusait a LINQ programozási paradigmába (LINQ to DataSet).
`System.Xml.Linq.dll`	A LINQ XML-dokumentumokban tárolt adatokkal történő használatát teszi lehetővé (LINQ to XML).

14.2. táblázat: *Alapvető LINQ-szerelvények*

Ha bármilyen LINQ-programozást szeretnénk használni, akkor importálnunk kell a projektünkbe legalább a `System.Linq` névteret (amely a `System.Core.dll` fájlban van definiálva), ezt a Visual Studio 2008 meg is teszi akkor, ha új projektet nyitunk. Egy új .NET 3.5 parancssorialkalmazás-projekt kezdő forráskódja a következő:

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

namespace MyConsoleApp
{
  class Program
  {
    static void Main(string[] args)
    {
    }
  }
}
```

Bevezetés a LINQ lekérdező kifejezésekbe

A LINQ programozási modell vizsgálatához készítsünk pár egyszerű lekérdező kifejezést különböző tömbökben tárolt adatok kezelésére. Hozzunk létre egy új .NET 3.5 parancssori alkalmazást `LinqOverArray` névvel, és definiáljunk egy `QueryOverStrings()` nevű statikus segédmetódust a `Program` osztályon belül. Ebben a metódusban állítsunk elő egy olyan sztringtömböt, amely mintegy hat, tetszés szerinti elemet tartalmaz (én most befejezésre váró videojátékokat soroltam fel).

```
static void QueryOverStrings()
{
  // Tegyük fel, hogy van egy sztringeket tartalmazó tömbünk.
  string[] currentVideoGames = {"Morrowind", "BioShock",
    "Half Life 2: Episode 1", "The Darkness",
    "Daxter", "System Shock 2"};
  Console.ReadLine();
}
```

Módosítsuk a `Main()` metódust, hogy meghívja a `QueryOverStrings()` metódusunkat.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with LINQ *****\n");
  QueryOverStrings();
  Console.ReadLine();
}
```

Gyakori feladat, hogy egy tömbből valamilyen feltétel alapján elemeket válogatunk ki. Előfordul, hogy olyan elemekre van szükségünk, amelyek számokat tartalmaznak (pl. a „System Shock 2" és „Half Life 2: Episode 1"), a megadottnál több karakterük van, vagy nincsen bennük szóköz (pl. Morrowind). Noha biztosan meg lehet oldani a problémát a `System.Array` névtér és némi erőfeszítés segítségével, a LINQ lekérdező kifejezések lényegesen megkönnyítik a folyamatot.

Induljunk ki abból, hogy a halmaz azon részhalmazára vagyunk kíváncsiak, amelyek hat karakternél hosszabb nevet tartalmaznak. Elkészíthetjük a következő lekérdező kifejezést:

```
static void QueryOverStrings()
{
  // Tegyük fel, hogy van egy sztringeket tartalmazó tömbünk.
  string[] currentVideoGames = {"Morrowind", "BioShock",
    "Half Life 2: Episode 1", "The Darkness",
    "Daxter", "System Shock 2"};

  // Készítsünk egy lekérdező kifejezést, amely azokat az
  // elemeket reprezentálja a tömbben, amelyek több,
  // mint hat betűt tartalmaznak.
  IEnumerable<string> subset = from g in currentVideoGames
    where g.Length > 6 orderby g select g;

  // Írjuk ki az eredményt.
  foreach (string s in subset)
    Console.WriteLine("Item: {0}", s);
}
```

A létrehozott lekérdező kifejezés a `from`, `in`, `where`, `orderby` és `select` LINQ-lekérdezésoperátorokat használta. Az utasítást a következőképpen kell értelmezni: „Add azokat az elemeket a `currentVideoGamesből`, amelyeknek több, mint hat karakter a hossza, és rendezd ábécésorrendbe őket". Minden elemnek, amely eleget tesz a kritériumnak, a „g" nevet adtuk (mint game), egyébként bármilyen érvényes C#-változó neve megtette volna.

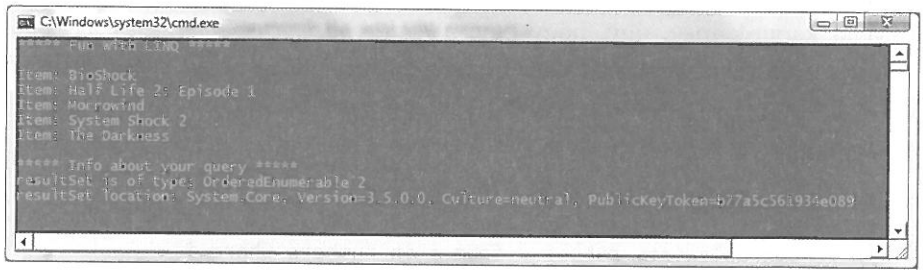
```
IEnumerable<string> subset = from game in currentVideoGames
  where game.Length > 6 orderby game select game;
```

Az eredményhalmaz tárolására szolgáló változó, a részhalmaz, egy olyan objektummal van reprezentálva, amely az `IEnumerable<T>` generikus verzióját implementálja, ahol `TSystem.String` típusú (hiszen sztringek tömbjét kérdezzük le). Miután megkaptuk az eredményhalmazunkat, egyszerűen kiíratjuk az elemeit a `foreach` szerkezet segítségével.

Mielőtt megnéznénk a lekérdezés eredményét, tegyük fel, hogy van a `Program` osztályunknak egy `ReflectOverQueryResults()` nevű segédfüggvénye, amely kiírja a LINQ-eredményhalmaz különböző tulajdonságait (a paraméter `System.Object` azért, hogy többfajta típusú eredményhalmazt támogasson).

```
static void ReflectOverQueryResults(object resultSet)
{
  Console.WriteLine("***** Info about your query *****");
  Console.WriteLine("resultSet is of type: {0}",
    resultSet.GetType().Name);
  Console.WriteLine("resultSet location: {0}",
    resultSet.GetType().Assembly);
}
```

Tételezzük fel, hogy meghívjuk ezt a metódust a `QueryOverString()` metódusunkban közvetlen azután, hogy kiírattuk a visszakapott részhalmazt. Ha futtatjuk az alkalmazást, látható, hogy a részhalmaz valóban az `OrderedEnumerable<TElement, TKey>` generikus típus egy példánya (a köztes nyelvi kódban [CIL] `OrderedEnumerable`'2-ként szerepel), amely pedig a `System.Core.dll` szerelvény belső absztrakt típusa (ld. 14.2. ábra).



14.2. ábra: *A LINQ-lekérdezésünk eredménye*

Megjegyzés Sok, LINQ-eredményt reprezentáló típus rejtve van a Visual Studio 2008 objektumböngészője elől. Használjuk az ildasm.exe vagy a `reflector.exe` programokat ezeknek a belső rejtett típusoknak a megtekintésére.

A LINQ és az implicit típusú lokális változók

A példaprogramból relatíve könnyen meghatározható, hogy az eredményhalmaz felsorolható sztringgyűjteményként, az azonban nyilván *nem* világos, hogy a részhalmaz valóban az `OrderedEnumerable<TElement,TKey>` típusú-e. Mivel a LINQ-eredményhalmazok jó néhány típussal reprezentálhatók a különböző LINQ-központú névterekben, nincs szükség a megfelelő típus definiálására az eredményhalmaz számára, hiszen az alapjául szolgáló típus gyakran vagy nem egyértelmű, vagy direkt elérhető a kódunkból (és ahogy látni fogjuk, bizonyos esetekben a típus generálása csak fordítási időben történik).

A probléma bemutatására nézzük meg a következő segédmetódust, amelyet a `Program` osztályon belül definiálunk (ezt pedig valószínűleg a `Main()` metódusból fogjuk meghívni).

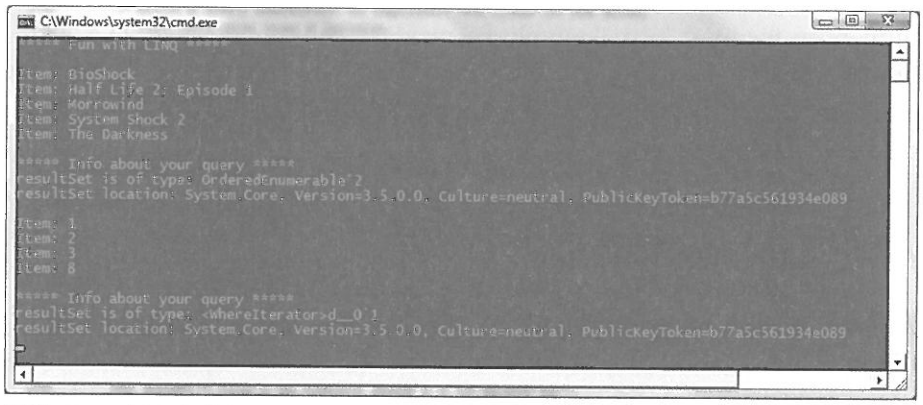
```
static void QueryOverInts()
{
  int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};

  // Csak a tíznél kisebb elemeket írjuk ki.
  IEnumerable<int> subset = from i in numbers where i < 10 select i;

  foreach (int i in subset)
    Console.WriteLine("Item: {0}", i);
  ReflectOverQueryResults(subset);
}
```

Ekkor egy `subset` nevű változót a `System.Linq.Enumerable.Where<T>` metódus hívásával kapunk meg, úgy, hogy egy, a *fordító által generált névtelen metódust* adunk át második paraméterként. És ez a fordító által generált `subset` változó belső definíciójának problémája (tegyük fel, hogy a névtelen metódust `9__CachedAnonymousMethodDelegate8`-nak nevezzük el):

```
// A következő LINQ lekérdező kifejezés:
//
// IEnumerable<int> subset = from i in numbers where i <
// 10 select i;
//
// átalakul egy Enumerable.Where<int>() metódushívássá:
//
IEnumerable<int> subset = Enumerable.Where<int>(numbers,
  Program.<>9__CachedAnonymousMethodDelegate8);
```

Megjegyzés LINQ-alapú alkalmazásainkat ajánlatos betölteni egy decompilerbe (mint az `ildasm.exe` vagy a `reflector.exe`). Az ilyen jellegű alkalmazások nagyban segítik a LINQ belső működésének megértését.

A 14.3. ábrán látható a LINQ lekérdező kifejezésünk formátuma alapján, hogy minden lekérdező kifejezés alaptípusa egyedi.



14.3. ábra: *A LINQ lekérdező kifejezések számos eredményhalmazzal térhetnek vissza*

Mivel a LINQ-lekérdezés alaptípusa nyilvánvalóan nem magától értetődő, az aktuális példa a lekérdezés eredményét mint lokális `IEnumerable<T>` változót reprezentálta. Ha az `IEnumerable<T>` kiterjeszti a nem generikus `IEnumerable` interfészt, akkor egy LINQ-lekérdezés eredményét a következőképpen is tárolhatjuk:

```
System.Collections.IEnumerable subset =
  from i in numbers where i < 10 select i;
```

Ez szintaktikailag helyes ugyan, ám az implicit tipizálás lényegesen átláthatóbbá teszi a dolgokat, ha a LINQ-lekérdezésekkel dolgozunk:

```
static void QueryOverInts()
{
  int[] numbers = {10, 20, 30, 40, 1, 2, 3, 8};

  // Használjuk az implicit tipizálást itt...
  var subset = from i in numbers where i < 10 select i;

  // ... és itt.
  foreach (var i in subset)
    Console.WriteLine("Item: {0} ", i);

  ReflectOverQueryResults(subset);
}
```

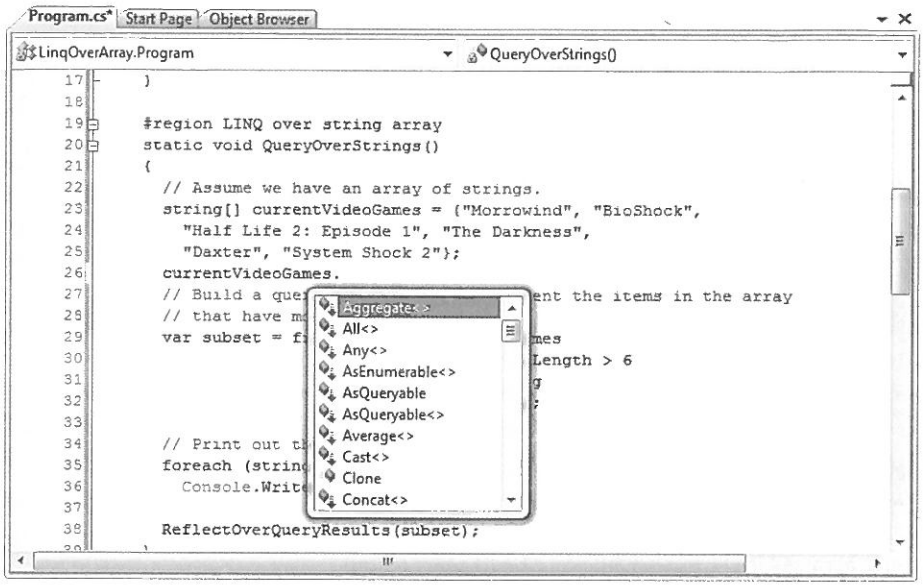
A var kulcsszót ne keverjük össze a COM Variant hagyatékával vagy a számos szkriptnyelvben megtalálható, gyengén tipizált változók deklarációjával. Az alaptípust a fordító határozza meg a kezdeti értékadástól függően. Ezen túlmenően fordítási hibát kapunk, ha megkíséreljük módosítani a „típus típusát". Továbbá: az alaptípus sok esetben dinamikusan generált névtelen típus eredménye, így megszokott dolog típusokat használni bárhol, ahol LINQ-eredményhalmazt szeretnénk megkapni.

A LINQ és a bővítő metódusok

A *bővítő metódusok* lehetővé teszik új tagok hozzáadását egy korábban lefordított típushoz az adott projekt hatókörén belül (lásd előző fejezet). A lenti példában nem készítünk bővítő metódusokat, de a háttérben használni fogjuk őket. A LINQ lekérdező kifejezések használhatóak arra, hogy végigiteráljunk azokon az adattárolókon, amelyek a generikus IEnumerable<T> interfészt implementálják. A .NET System.Array osztálya (amelyet itt sztringek és egészek tárolására használunk) azonban nem implementálja ezt a viselkedést:

```
// A System.Array osztály nem implementálja a megfelelő
// infrastruktúrát a lekérdező kifejezésekhez!
public abstract class Array : ICloneable, IList, ICollection,
  IEnumerable
{
  ...
}
```

Mivel a System.Array osztály közvetlenül nem implementálja az IEnumerable<T> interfészt, ezért indirekt módon tesz szert ennek a típusnak (valamint sok más LINQ-központú tagnak) a szükséges funkcionalitására a System.Linq.Enumerable statikus osztálytípus révén. Ez a típus számos generikus bővítő metódust definiált (mint az Aggregate<T>(), First<T>(), Max<T> stb.), amelyeket a System.Array (és más típusok) a háttérben átvesznek. Ezért, ha használjuk a pont operátort a currentVideoGames lokális változón, számos olyan tagra lelünk, amelyeket *nem* találunk a System.Array formális definíciójában (lásd 14.4. ábra).



14.4. ábra: *A System.Array típus a System.Linq.Enumerable tagjaival bővül*

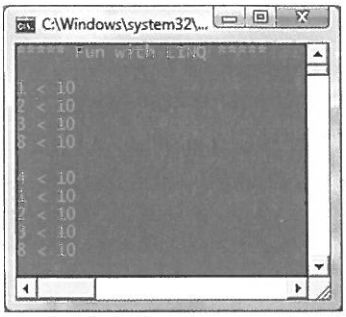
A késleltetett futtatás szerepe

A LINQ lekérdező kifejezések kiértékelése nem történik meg, amíg a tartalmukon végig nem iterálunk. Formálisan ezt *késleltetett futtatásnak* hívjuk. Ennek a megközelítésének az az előnye, hogy lehetőségünk van ugyanazt a LINQ-lekérdezést többször is alkalmazni ugyanarra a tárolóra, és bízhatunk benne, hogy a legutolsó és a legnagyobb eredményt kapjuk. Nézzük meg a következő módosítást a `QueryOverInts()` metóduson

```
static void QueryOverInts()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  //A tíznél kisebb számokat szeretnénk megkapni.
  var subset = from i in numbers where i < 10 select i;

  //A LINQ-utasítás itt kerül kiértékelésre.
  foreach (var i in subset)
    Console.WriteLine("{0} < 10", i)
  Console.WriteLine();

  //Néhány adatot módosítunk a tömbben.
  numbers[0] = 4;
```

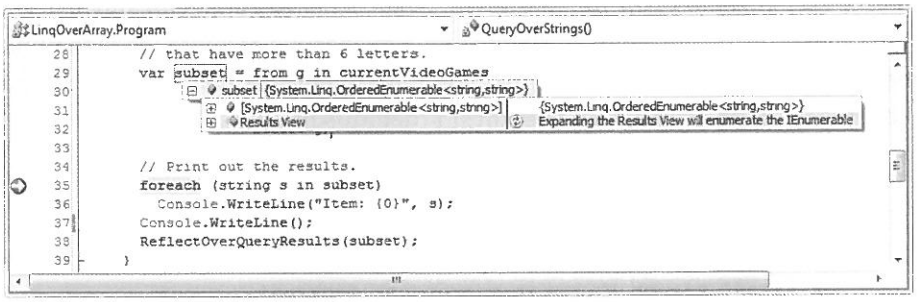
```
    //Újra kiértékeljük.
    foreach (var j in subset
      Console.WriteLine("{0} < 10", j);
    ReflectOverQueryResults(subset);
}
```

Ha ismét lefuttatjuk a programot, a 14.5. ábrán látható eredményt kapjuk.



14.5. ábra: *A LINQ-kifejezések futtatása a kiértékeléskor történik*

A Visual Studio 2008 egyik hasznos lehetősége, hogy ha közvetlen a LINQ-lekérdezés kiértékelése elé teszünk egy töréspontot, akkor lehetőségünk van megnézni a tartalmát a hibakeresési folyamat közben. Ehhez egyszerűen mozgassuk az egérkurzort a LINQ-eredményhalmazt definiáló változóra (a `subsetre` a 14.6. ábrán). Ekkor lehetőségünk nyílik kiértékelni a lekérdezést a Results View opció kinyitásával.



14.6. ábra: *LINQ-kifejezések hibakeresése*

Az azonnali futtatás szerepe

Ha egy LINQ-kifejezést a `foreach` hatókörén kívül szeretnénk kiértékelni, lehetőségünk van meghívni az `Enumerable` típus által definiált bármely bővítő metódust, hogy ezt megtegye. Az `Enumerable` típus több bővítő metódust definiál,

például a `ToArray<T>()`, a `ToDictionary<TSource,TKey>()` és a `ToList<T>()` metódusokat, amelyek lehetővé teszik, hogy a LINQ-lekérdezés eredményhalmazát erősen tipizált tárolóban kapjuk vissza. Ekkor a tároló nem marad tovább „kapcsolatban" a LINQ-kifejezéssel, attól függetlenül kezelhető:

```
static void ImmediateExecution()
{
  int[] numbers = { 10, 20, 30, 40, 1, 2, 3, 8 };

  //Az adatokat pont MOST kapjuk meg egy int[] tömbbe.
  int[] subsetAsIntArray =
    (from i in numbers where i < 10 select i).ToArray<int>();

  //Az adatokat pont MOST kapjuk meg egy List<int> típusba.
  List<int> subsetAsListOfInts =
    (from i in numbers where i < 10 select i).ToList<int>();
}
```

A teljes LINQ-kifejezést zárójelek közé tettük, hogy a megfelelő, ténylegesen jelen lévő típussá kasztoljuk (bármi is legyen az) azért, hogy meghívhassuk az `Enumerable` bővítő metódusait.

Ha a C#-fordító egyértelműen meg tudja határozni egy generikus elem típusparaméterét, akkor nem kell nekünk meghatároznunk (lásd 10. fejezet). Ezért a következőképpen is meghívhatjuk a `ToArray<T>()` (vagy a `ToList<T>()`) metódust:

```
int[] subsetAsIntArray =
  (from i in numbers where i < 10 select i).ToArray();
```

Forráskód A `LinqOverArray` projekt a 14. fejezet alkönyvtárában található.

A LINQ és a generikus gyűjtemények

Azon túl, hogy egyszerű tömbökből elemeket tud kiválogatni, a LINQ arra is képes, hogy a `System.Collections.Generic` névtér típusaiban tárolt adatokat manipulálja. Hozzunk létre egy új .NET 3.5 parancssorialkalmazás-projektet `LinqOverCustomObjects` néven. Definiáljunk egy alap `Car` típust, amely karbantartja az aktuális sebességet, a színt, a márkát és a becenevet. (Használjunk publikus mezőket, hogy egyszerűen üres szöveggel töltsük fel a sztringmezőket. Nyugodtan alkalmazzunk automatikus tulajdonságokat és osztálykonstruktorokat kedvünk szerint):

```
class Car
{
  public string PetName = string.Empty;
  public string Color = string.Empty;
  public int Speed;
  public string Make = string.Empty;
}
```

Ezután a `Main()` metóduson belül definiáljunk egy lokális, `Car` típusú `List<T>` változót, és használjuk a 13. fejezetben megismert új objektuminicializáló szintaxist, hogy feltöltsük a listánkat új `Car` objektumokkal:

```
static void Main(string[] args)
{
  Console.WriteLine("***** More fun with LINQ Expressions *****\n");

  // Hozzunk létre a Car objektumok egy listáját (List<>)
  // az objektuminicializáló szintaxis használatával.
  List<Car> myCars = new List<Car>() {
    new Car{ PetName = "Henry", Color = "Silver", Speed = 100,
      Make = "BMW"},
    new Car{ PetName = "Daisy", Color = "Tan", Speed = 90,
      Make = "BMW"},
    new Car{ PetName = "Mary", Color = "Black", Speed = 55,
      Make = "VW"},
    new Car{ PetName = "Clunker", Color = "Rust", Speed = 5,
      Make = "Yugo"},
    new Car{ PetName = "Melvin", Color = "White", Speed = 43,
      Make = "Ford"}
  };
}
```

A LINQ-kifejezés alkalmazása

A feladatunk olyan lekérdező kifejezés létrehozása, amelynek segítségével csak azokat az elemeket kapjuk vissza a `myCar` listánkból, amelyek sebessége nagyobb, mint 55. Ha megvan a kívánt részhalmaz, írassuk ki a megfelelő `Car` objektumok nevét. Tegyük fel, hogy rendelkezésünkre áll a következő segédmetódus (bemeneti paramétere: `List<Car>`), amelyet a `Main()` metódusból hívunk meg:

```
static void GetFastCars(List<Car> myCars)
{
  //Hozzunk létre egy lekérdező kifejezést.
  var fastCars = from c in myCars where c.Speed > 55 select c;
```

```
    foreach (var car in fastCars)
    {
      Console.WriteLine("{0} is going too fast!", car.PetName);
    }
}
```

A lekérdező kifejezésünk csak azokat az elemeket választja ki a `List<T>` listából, amelyeknek a `Speed` tulajdonsága nagyobb, mint 55. Ha futtatjuk az alkalmazást, látni fogjuk, hogy csak a „Henry" és a „Daisy" azok az elemek, amelyek megfelelnek a keresési feltételnek.

Ha bonyolultabb lekérdezést szeretnénk készíteni, akkor keressük meg például azokat a BMW-ket, amelyeknek a sebességértéke 90 fölött van. Ehhez egyszerűen hozzunk létre egy összetett `Boolean` utasítást a C# && operátorának segítségével:

```
//Hozzunk létre egy lekérdező kifejezést.
var fastCars = from c in myCars where
  c.Speed > 90 && c.Make == "BMW" select c;
```

Ebben az esetben az egyetlen kiírt becenév a „Henry".

Forráskód A `LinqOverCustomObject` projekt a 14. fejezet alkönyvtárában található meg.

A LINQ és a nem generikus gyűjtemények

A LINQ lekérdezésoperátorai olyan típusokkal dolgoznak együtt, amelyek implementálják az `IEnumerable<T>` interfészt (vagy közvetlenül, vagy bővítő metódusokon keresztül). Mivel a `System.Array`-t elláttuk az ehhez szükséges infrastruktúrával, meg fog lepni minket, hogy a hagyaték- (nem generikus) tárolók a `System.Collections` névtéren belül *nincsenek* biztosítva. Szerencsére lehetőségünk van arra, hogy a nem generikus gyűjtemények adatait is végigiteráljuk a generikus `Enumerable.OfType<T>()` metódus segítségével.

Az `OfType<T>()` metódus egyike azon kevés `Enumerable` tagnak, amelynek nincs megfelelője a generikus típusok esetén. Amikor meghívjuk egy nem generikus tároló (pl. `ArrayList`) `OfType<T>()` metódusát, a `T`-ben specifikálhatjuk azt a belső tárolt típust, amellyel kompatibilis `IEnumerable<T>` objektumot szeretnénk visszakapni.

Tegyük fel, hogy van egy új parancssori alkalmazásunk `LinqOverArratList` névvel, amely a következő `Main()` metódust definiálja (a korábban definiált `Car` típust használjuk, és mindenképpen importáljuk a `System.Collections` névteret).

```
static void Main(string[] args)
{
  Console.WriteLine("***** LINQ over ArrayList *****\n");

  // Itt van az autók egy nem generikus gyűjteménye.
  ArrayList myCars = new ArrayList() {
    new Car{ PetName = "Henry", Color = "Silver", Speed = 100,
      Make = "BMW"},
    new Car{ PetName = "Daisy", Color = "Tan", Speed = 90,
      Make = "BMW"},
    new Car{ PetName = "Mary", Color = "Black", Speed = 55,
      Make = "VW"},
    new Car{ PetName = "Clunker", Color = "Rust", Speed = 5,
      Make = "Yugo"},
    new Car{ PetName = "Melvin", Color = "White", Speed = 43,
      Make = "Ford"}
  };

  //Az ArrayList típust alakítsuk át egy
  // IEnumerable<T>-kompatibilis típussá.
  IEnumerable<Car> myCarsEnum = myCars.OfType<Car>();

  //Hozzunk létre egy lekérdező kifejezést.
  var fastCars = from c in myCarsEnum where c.Speed > 55 select c;

  foreach (var car in fastCars)
  {
    Console.WriteLine("{0} is going too fast!", car.PetName);
  }
}
```

Adatok szűrése az OfType<T>() használatával

A nem generikus típusok az elemek bármilyen kombinációját képesek tárolni, hiszen az ilyen tároló tagjait (pl. az `ArrayList`) úgy alakították ki, hogy a `System.Object` elemeit kapják meg. Tegyük fel például, hogy az `ArrayList` különféle elemeket tartalmaz, amelyeknek csak egy részhalmaza numerikus. Ha olyan részhalmazt szeretnénk megkapni, amely csak numerikus adatokat tartalmaz, ezt az `OfType<T>()` segítségével meg tudjuk tenni, ez a metódus ugyanis kiszűri az iteráció során azokat az elemeket, amelyek típusa különbözik a megadott típustól:

```
//Nyerjük ki az integer értékeket az ArrayListből.
ArrayList myStuff = new ArrayList();
myStuff.AddRange(new object[] { 10, 400, 8, false, new Car(),
  "string data" });
IEnumerable<int> myInts = myStuff.OfType<int>();

//Írjuk ki a 10, a 400 és a 8 értékeket.
foreach (int i in myInts)
{
  Console.WriteLine("Int value: {0}", i);
}
```

Forráskód A `LinqOverArrayList` projekt a 14. fejezet alkönyvtárában található.

A továbbiakban nézzük meg részletesebben a LINQ használatát.

A LINQ-lekérdezésoperátorok belső ábrázolása

Azt már láttuk, hogy hogyan kell lekérdező kifejezéseket írni a különböző C#-lekérdezőoperátorok segítségével (pl. az `in`, a `where`, az `orderby` és a `select`). A C#-fordító ezeket a tokeneket a `System.Linq.Enumerable` típus (és valószínűleg a LINQ-lekérdezésünk alapján egyéb típusok) különböző metódusainak hívására fordítja le.

Számos `Enumerable` metódus képes metódusreferenciák fogadására argumentumként. Sok metódus `Func<>` típusú generikus metódusreferenciát igényel, amelyet a `System.Core.dll` `System` névterében definiálunk. Nézzük például az `Enumerable` következő tagjait, amelyek kibővítik az `IEnumerable<T>` interfészt:

```
// Az Enumerable.Where<T>() metódus túlterhelt verziói.
// Figyeljük meg, hogy a második paraméter típusa System.Func<>.
public static IEnumerable<TSource> Where<TSource>(
  this IEnumerable<TSource> source,
  System.Func<TSource,int,bool> predicate)

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

Ez a metódusreferencia (ahogy azt a neve is sugallja) az adott függvény mintáját reprezentálja argumentumok halmazával és egy visszatérési értékkel. Ha a Visual Studio 2008 objektumböngészőjével vizsgáljuk meg ezt a típust, láthatjuk, hogy a `Func<>` metódusreferencia nullától négyig tetszőleges számú bemenő paramétert (a típusuk itt `T0`, `T1`, `T2`, és `T3`, a nevük pedig `arg0`, `arg1`, `arg2` és `arg3`) és `TResult` névvel jelzett visszatérési típust tud felvenni.

```
// A Func<> metódusreferencia különböző formái.
public delegate TResult Func<T0,T1,T2,T3,TResult>(
  T0 arg0, T1 arg1, T2 arg2, T3 arg3)
public delegate TResult Func<T0,T1,T2,TResult>(T0 arg0, T1 arg1,
  T2 arg2)
public delegate TResult Func<T0,T1,TResult>(T0 arg0, T1 arg1)
public delegate TResult Func<T0,TResult>(T0 arg0)
public delegate TResult Func<TResult>()
```

Mivel a `System.Linq.Enumerable` jó néhány tagja bemenetként metódusreferenciát igényel, így, amikor meghívjuk őket, két dolog lehetséges: vagy manuálisan készítünk egy új metódusreferencia-típust, és megszerkesztjük a szükséges célmetódusokat a C# névtelen metódusainak felhasználásával, vagy definiálunk egy megfelelő lambda kifejezést. Függetlenül attól, hogy melyik megoldást választjuk, az eredmény ugyanaz lesz.

A C# LINQ-lekérdezésoperátorainak segítségével messze a legegyszerűbb LINQ lekérdező kifejezéseket készíteni. A következőkben nézzük meg mindegyik lehetőséget, hogy lássuk, mi a kapcsolat a C#-lekérdezőoperátorok és az alap `Enumerable` típus között.

Lekérdező kifejezések létrehozása lekérdezésoperátorok segítségével. Ismétlés

Kiindulásként hozzunk létre egy konzolalkalmazást `LinqUsingEnumerable` névvel. A `Program` osztály számos statikus segédmetódust definiál (amelyeket mind a `Main()` metódusból hívunk meg) a különböző módszerek illusztrálására, amelyekkel LINQ-lekérdezéseket írhatunk. Az első metódus a `QueryStringsWithOperators()`, amely a legnyilvánvalóbb lehetőséget mutatja be lekérdező kifejezés készítésére, és megegyezik az előző `LinqOverArray` mintapélda kódjával: