

## Iterátor metódus készítése a yield kulcsszóval

Régebben, ha meg akartunk valósítani egy egyedi gyűjteményt (mint a Garage), amely támogatja a foreach felsorolást, az IEnumerator-t (és esetleg az IEnumerator-2.0 megjelölésű alternatív lehetőségünk is van olyan típusok létrehozására, amelyek a foreach ciklussal működnek, és ez az *iterátor*).

Egyszerűen: az iterátor olyan tag, amely meghatározza, hogy egy tároló belső elemeit hogyan lehet egy foreach feldolgozás során visszaadni. Noha az iterátor metódus mindig GetEnumerator() nevű kell, hogy legyen, és a visszatérési értéke is IEnumerator típusú kell, hogy legyen, a saját osztályunkban nem szükséges semmilyen interfész megvalósítása.

Ennek bemutatásához hozzuk létre a customenumeratorwithyield nevű parametrossoralkalmazás-projektet, és tegyük bele a car, a radio és a garage típusokat az előző példából (megtámasztva a névtérdefiniciókat az aktuális projektre). Ezután tervezzük át a garage típust a következők szerint:

```
public class Garage
{
    private Car[] carArray = new Car[4];
    ...
    // Iterátor metódus.
    public IEnumerator GetEnumerator()
    {
        foreach (Car c in carArray)
        {
            yield return c;
        }
    }
}
```

A GetEnumerator() ezen megvalósítása az a lelemeket egy belső foreach logikát használva járja be, és minden egyes car objektumot a yield return szintissal ad vissza. A yield kulcsszót használjuk arra, hogy megadjuk azt az értéket (vagy értékeket), amelyet a hívó foreach szerkezetének vissza kell adnunk. Amikor elérjük a yield return utasítást, a pillanatnyi helyzet tárolódik, és az iterátor következő hívásakor innen indul újra a végrehajtás. Az iterátormetódusok nem feltétlenül használják a foreach kulcsszót a tartalom visszaadására. Ezt az iterátormetódust a következő módon is megadhatjuk:

```

    public IEnumerator GetEnumerator()
    {
        yield return carArray[0];
        yield return carArray[1];
        yield return carArray[2];
        yield return carArray[3];
    }

```

Ebben a megvalósításban a `GetEnumerator()` metódus explicit módon adja vissza a hívónak az új értéket minden egyes futásakor. Ebben a példánkban ennek nem sok értelme lenne, mert ha a `carArray` tagváltozózhoz további objektumokat adnánk, akkor a `GetEnumerator()` metódusunk nem működne helyesen. Mindazonáltal ez a szintaxis is hasznos lehet, ha lokális adatot szeretnénk visszaadni egy metódusból a `foreach` szintaxisának feldolgozására.

## Nevesített iterátor létrehozása

A `yield` kulcsszó technikailag bármely metódusban használható, a nevetől függetlenül. Ezek a metódusok (amelyek technikailag *nevesített iterátorok*) olyan szempontból is különlegesek, hogy akármilyen számú argumentumot tudnak fogadni. Ha nevesített iterátort készítettünk, legyünk mindig nagyon körültekintőek, mert a metódus az `IEnumerator`-kompatibilis típust. Ennek bemutatására adjuk a következő metódust a `garage` típushoz:

```

public IEnumerator<Car> GetEnumerator()
{
    // Az elemeket fordított sorrendben adjuk vissza.
    if (ReturnReversed)
    {
        for (int i = carArray.Length; i > 0; i--)
        {
            yield return carArray[i-1];
        }
    }
    else
    {
        // Az elemeket a tömbben elhelyezés szerint adjuk vissza.
        foreach (Car c in carArray)
        {
            yield return c;
        }
    }
}

```



Az új módszerünk az aelemeket szekvenciális sorrendben adja vissza, vagy ha a bejövő paraméter igaz értékű, akkor fordított sorrendben. Ezek után a következő módon használhatjuk az új módszerünket:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with the yield keyword *****\n");
    Garage carlot = new Garage();

    // Elemek beolvasása a GetEnumerator() segítségével.
    foreach (Car c in carlot)
    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.Speed);
    }

    Console.WriteLine();
    // Elemek beolvasása (fordított sorrendi) megnevezett iterátorral.
    foreach (Car c in carlot.GetCars(true))
    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.Speed);
    }
}
```

A nevesített iterátorok akkor hasznosak, ha egyetlen egyed! tároló elemeit többfelé képpen szeretnénk elérni.

## Az iterátormetódusok belső reprezentációja

Amikor a C#-fordító talál egy iterátormetódust, dinamikusan létrehoz hozzá egy beágyazott osztálydefiniációt a megadott típus hatókörén belül (jelen esetben ez a garage). Az automatikusan generált beágyazott osztály megvalósítja a GetEnumerator(), a MoveNext() és a Current tagokat helyezettük (különös módon a Reset() metódust nem, és futásidőben kivételt kapunk, ha megpróbáljuk meghívni). Ha betöltjük az aktuális alkalmazást az i1dasm.exe programmal, akkor két beágyazott típust találunk, amelyek a specifikus iterátormetódusok által igényelt logikákért felelnek. A 9.10. ábrán a számtógép által generált készített típusok neve <GetEnumerator>d\_0 és <GetCars>d\_5.

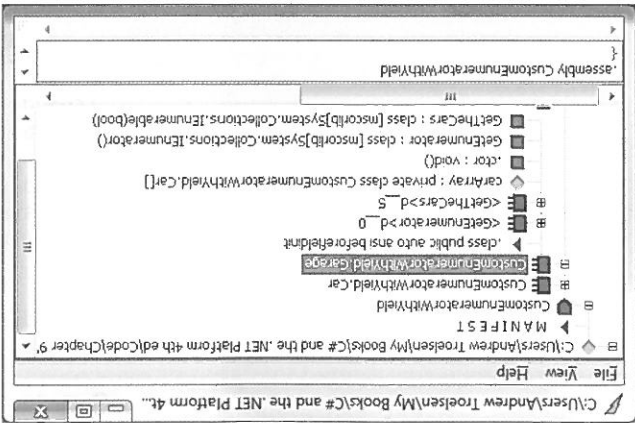
Ha az i1dasm.exe programot használjuk arra, hogy a garage típus GetEnumerator() metódusának megvalósítását megnezzük, akkor azt találjuk, hogy a rejtett <GetEnumerator>d\_0 típust használja a megvalósítás (hasonlóan a beágyazott <GetCars>d\_5 típust használja a GetCars() metódus).

```

        method public hidebysig instance class
        [mscorlib]System.Collections.IEnumerator
        GetEnumerator() cil managed
    }

    newobj instance void
        CustomEnumeratorWithYield.Garage/
        <GetEnumerator>d__0'::ctor(int32)
    } // A Garage::GetEnumerator módszer vége.

```



9.10. ábra: Az iterátormetódusok belső megvalósítása automatikusan generált beágyazott osztályokkal

Teljesse leve a felsorolható objektumok készítését, megismerhetjük: ahhoz, hogy az egyedi típusunk a C# foreach kulcsszavával együttműködjön a tárolónak definiálnia kell a GetEnumerator() nevű metódust, amelyet az IEnumerator interfész formalizál. Ennek a metódusnak a megvalósítását tipikusan a belső taghoz történő delegálással érjük el, ez tartalmazza az alobjektumokat; az is lehetséges azonban, hogy a yield return szintaxis használatával több „nevesített iterátormetódust” nyújtunk.

**Forráskód** A CustomEnumeratorWithYield projekt megtalálható a 9. fejezet alkönyvtárában.

## Klónozható objektumok készítése (Cloneable)

A `System.Object` osztály definiál egy `MemberwiseClone()` nevű tagot is (lásd 6. fejezet). Ez a módszer arra szolgál, hogy *sekély másolatot* kapjunk az aktuális objektumról. Az objektum használói nem hívják ezt a módszert közvetlenül (hiszen védett); az adott objektum azonban hívhatja a *klónozási* folyamat során. Ennek bemutatásához hozzunk létre egy új paramcsessorit alkalmazást `CloneablePoint` néven, amely egy `Point` nevű osztályt definiál:

```
// A Point nevű osztály.
public class Point
{
    // A könnyebb hozzáféréshez publicus.
    public int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public Point() {}
    // Az Object.ToString() felülírása.
    public override string ToString()
    {
        return string.Format("X = {0}; Y = {1}", x, y);
    }
}
```

Ha már ismerjük a referenciátípusokat és az értékípusokat (lásd 4. fejezet), akkor tudjuk, hogy ha egy referenciaváltozót értékül adunk egy másiknak, akkor két referenciánk mutat azonos objektumra a memóriában. Így módon a következő értékadás két referenciát eredményez ugyanarra a `Point` objektumra a heapmemóriában. Bármelyik referencia használataval módosítunk, ugyanaz az objektum módosul a heapmemóriában.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    // Ugyanazon objektum két referenciáját
    Point p1 = new Point(50, 50);
    Point p2 = p1;
    p2.x = 0;
    Console.WriteLine(p1);
    Console.WriteLine(p2);
    Console.ReadLine();
}
```

Ha szeretnénk megadni azt a képességet a saját típusunknak, hogy a hívónak azonos másolatot tudjon visszaadni, akkor a szabványos `ICloneable` interfész kell megvalósítanunk. Ez a típus – ahogy már láttuk – egyetlen `clone()` nevű metódust definiál:

```
public interface ICloneable
{
    object clone();
}
```

**Megjegyzés** Az `ICloneable` interfész hasznosságáról pillanatnyilag is vitatkoznak a .NET-kö-zösség tagjai. A probléma abból adódik, hogy a hivatalos specifikáció nem mondja ki kifejezet-ten az interfészt megvalósító objektumokról, hogy *mely másolatot* kell visszaadniuk az objekt-umról (azaz az objektumok belső referenciátípusai teljesen új objektumot eredményeznek, azonos állapottal). Így technikailag lehetséges az, hogy az `ICloneable` interfészt megvalósító objektumok csak *sekély másolatot* adják vissza az interfésznek (azaz a belső referenciáik a heapmemória azonos objektumára mutatnak), ez pedig tényleg zavaró eredményez. A pél-dáknban mi úgy valósítottuk meg a `clone()` metódust, hogy az az objektum teljes, mely máso-latát adja vissza az objektumnak.

Nyilvánvaló, hogy objektumonként változik a `clone()` metódus megvalósítá-sa. Az alapműködések azonban nagyjából azonosak: másoljuk le a tagválto-zók értékeit egy azonos típusú, új objektum példányba, és ezt adjuk vissza a felhasználónak. Ennek bemutatásához tegyük meg a következő módosításo-kat a `Point` osztályon:

```
// A Point támogatja a "klónozhatóságot."
public class Point : ICloneable
{
    public int x, y;
    public Point() { }
    public Point(int x, int y) { this.x = x; this.y = y; }

    // Az aktuális objektum másolatát adja vissza.
    public object clone()
    {
        return new Point(this.x, this.y);
    }

    public override string ToString()
    {
        return string.Format("X = {0}; Y = {1}", x, y);
    }
}
```

Igy a `Point` típus pontos, független másolatát hozhatjuk létre, ahogyan a kö-vetkező kód is mutatja:

```

static void main(String[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    // A clone() generikus objektumtípust ad vissza.
    // Explicit kasztozással olvasható be a leszármazott típus.
    Point p3 = new Point(100, 100);
    Point p4 = (Point)p3.Clone();
    // A p4.x módosítása (amely nem változtatja meg p3.x-et).
    p4.x = 0;
    // Az objektumok kitratása.
    Console.WriteLine(p3);
    Console.WriteLine(p4);
    Console.ReadLine();
}

```

Noha a `Point` pillanatnyi megvalósítása már megfelelő a számunkra, még modernizálhatjuk egy kicsit. Mivel a `Point` típus nem tartalmaz semmiféle belsőreferencia-típusú változót, egyszerűsíthetjük a `Clone()` metódus megvalósítását a következő módon:

```

public object Clone()
{
    // A Point mezőinek másolása tagról tagra.
    return this.MemberwiseClone();
}

```

Ha azonban a `Point` objektum mégis tartalmaz referenciátípusú tagváltozókat, akkor a `MemberwiseClone()` metódus lemásolja ezeknek az objektumoknak a referenciáit (úgy, mint a *sekély másolás*). Ha valóban mely másolást szeretnénk elérni, akkor a klonozás folyamán a referenciátípusú változóknak új példányait kell létrehozniuk. Nézzünk erre egy példát.

## Egy összetettebb klonozási példa

Tételezzük fel, hogy a `Point` osztály tartalmaz egy referenciátípusú tagváltozót, amely `PointDescription`-típusú. Ez az osztály egy pont barátságos nevét tartalmazza, valamint egy azonosítószámot `System.Guid` formájában (ha nincsen `COM-határunk`, akkor vegyük figyelembe, hogy a globális egyedi azonosító [GUID] statisztikailag egyedi 128 bites szám). A megvalósítás a következő:

```
// Az osztály egy pontot ír le.
public class PointDescription
{
    // Az egyszerűség kedvéért publikus módon.
    public String petName;
    public Guid pointID;

    public PointDescription()
    {
        this.petName = "No-name";
        pointID = Guid.NewGuid();
    }
}
```

Az alapvető frissítések a `Point` osztályon magukban foglalják a `ToString()` módszer módosítását ezeknek az állapotadatoknak a kezelésére, valamint a `PointDescription` referenciátípus definiálását és létrehozását. A túlterhelt konstruktor számára átadott paramétereket is módosítjuk, így tesszük lehetővé a felhasználó számára becenév hozzárendelését a `Point` osztályhoz.

```
public class Point : ICloneable
{
    public int x, y;
    public PointDescription desc = new PointDescription();

    public Point(){}
    public Point(int x, int y)
    {
        this.x = x; this.y = y;
    }
    public Point(int x, int y, string petName)
    {
        this.x = x;
        this.y = y;
        desc.petName = petName;
    }

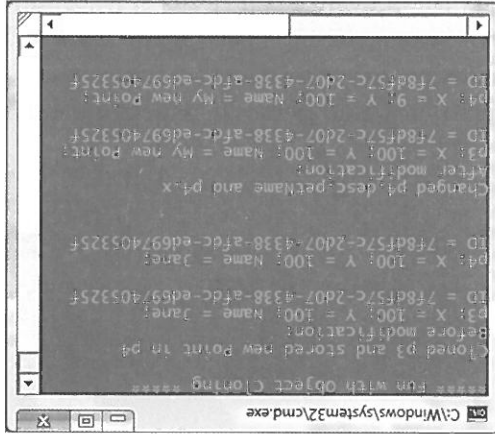
    public object Clone()
    { return this.MemberwiseClone(); }

    public override string ToString()
    {
        return string.Format("X = {0}; Y = {1};",
            Name = "{2}"; \nid = "{3}"; \n",
            x, y, desc.petName, desc.pointID);
    }
}
```

Egyelőre még nem módosítottuk a `clone()` módszerünket. Ennek megfelelően, ha az objektumhasználo a pillanatnyi implementációval próbál klónozni, akkor sekély (tagonként) másolat keletkezik. Ennek bemutatásához vegyük a következő `main()` módszert.

```
static void main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    Console.WriteLine("Cloned p3 and stored new Point in p4");
    Point p3 = new Point(100, 100, "Jane");
    Point p4 = (Point)p3.Clone();
    Console.WriteLine("Before modification:");
    Console.WriteLine("p3: {0}", p3);
    Console.WriteLine("p4: {0}", p4);
    p4.desc.petName = "My new Point";
    p4.x = 9;
    Console.WriteLine("\nChanged p4.desc.petName and p4.x");
    Console.WriteLine("After modification:");
    Console.WriteLine("p3: {0}", p3);
    Console.WriteLine("p4: {0}", p4);
    Console.ReadLine();
}
```

A 9.11. ábra mutatja a kimenetet. Míg az értéktípusok tényleg megváltoztak, a belső referenciátípusok továbbra is azonos értékek, hiszen azonos memóriaterületre mutatnak.



9.11. ábra: A `MemberwiseClone()` módszer az aktuális objektumnak a sekély másolatát adja vissza

Ahhoz, hogy a `clone()` módszer tényleg teljes mély másolatot készítsen a belső referenciátípusokról, a `memberwiseClone()` módszerrel visszaadott objektumokat kell úgy alakítanunk, hogy kezelje az aktuális pont nevét (a `system.guid` típus valójában egy struktúra, így a numerikus adatot tényleg másoljuk). Ennek egy lehetséges megvalósítása:

```
// A PointDescription taghoz kell igazodni.
public object Clone()
{
    // A sekély másolat beolvasása.
    Point newPoint = (Point)this.MemberwiseClone();

    // A hëzagok kitöltése.
    PointDescription currentDesc = new PointDescription();
    currentDesc.petName = this.desc.petName;
    newPoint.desc = currentDesc;
    return newPoint;
}
```

Ha ismét lefuttatjuk az alkalmazásunkat, akkor ahogyan a 9.12. ábrán is látható, a `clone()` módszerrel visszaadott `Point` objektumban a belső referenciátípusú tagváltozók is másolódtak (a becenév egyedi a p3 és a p4 esetében is).



9.12. ábra: Most már valós mély másolatunk van az objektumról.

A klónozási folyamat a következőképpen foglathatjuk össze: ha olyan osztályunk vagy struktúránk van, amely csak értéktípusokat tartalmaz, akkor a `clone()` módszert valósítsuk meg a `memberwiseClone()` használatával. Ha azonban a saját típusunk referenciátípusokat is kezel, akkor olyan típust kell létrehoznunk, amely figyelembe veszi a referenciátípusú tagváltozókat is.

**Forráskód** A `CloneablePoint` projekt megtalálható a 9. fejezet alkönyvtárban.



## Összehasonlítható objektumok készítése (Comparable)

A `System.Comparable` interfész olyan viselkedést definiál, amely lehetővé teszi egy kulcs alapján az objektumok sorba rendezését. A formális definíció a következő:

```
// Az interfész lehetővé teszi az objektum számára, hogy
// meghatározza más hasonló objektumokkal való kapcsolatait.
public interface Comparable
{
    int compareTo(Object o);
}
```

Hozzunk létre egy új parancssori alkalmazást `comparablecar` néven, amely a következő továbbfejlesztett `car` típust definiálja (alapvetően csak egy új tag-változót adtunk hozzá az egyedi azonosító tárolásához, és ennek beállítását és elérési lehetőségét valósítottuk meg):

```
public class Car
{
    ...
    private int carID;
    public int ID
    {
        get { return carID; }
        set { carID = value; }
    }
    public Car(string name, int currsp, int id)
    {
        currsp = currsp;
        petName = name;
        carID = id;
    }
    ...
}
```

Tételezzük fel, hogy van egy `car` típusokat tartalmazó tömbünk:

```
static void main(string[] args)
{
    Console.WriteLine("***** Fun with Object Sorting *****\n");
    // car típusok tömbjének elkészítése.
    Car[] myAutos = new Car[5];
    myAutos[0] = new Car("Rusty", 80, 1);
    myAutos[1] = new Car("Mary", 40, 234);
}
```

```

myAutos[2] = new Car("Viper", 40, 34);
myAutos[3] = new Car("Me1", 40, 4);
myAutos[4] = new Car("Chuck", 40, 5);
Console.ReadLine();
}

```

A `System.Array` osztály egy `sort()` nevű statikus módszert definiál. Ha ezt a módszert meghívjuk egy belső típust (`int`, `short`, `string` stb.) tartalmazó tömbre, akkor rendezni tudjuk a tömböt szám/ábécé sorrendben, minthogy a lényeges adattípusok megvalósítják az `IComparable` interfészt. Mi történik azonban akkor, ha egy `car` típusú elemeket tartalmazó tömböt adunk át a `sort()` módszernek?

```

// Rendezzük az autókát?
Array.Sort(myAutos);

```

Ha lefuttatjuk a tesztet, akkor azt tapasztaljuk, hogy futásidőben `ArgumentException` kivételt generálódik a következő üzenettel:

---

```

"at least one object must implement IComparable."

```

---

Ha saját típusokat hozunk létre, akkor megvalósíthatjuk az `IComparable` interfészt, lehetővé téve a tömbök rendezését. Ha megnezzük a `compareTo()` metódus részleteit, akkor kiderül, hogy a rendezés alapja csak rajtunk múlik. A `car` típusnál a belső `carId` logikus választásnak tűnik:

```

// A car objektumok iterációja rendezhető
// a carId alapján.
public class Car : IComparable
{
    ...
    // IComparable megvalósítás.
    int IComparable.CompareTo(object obj)
    {
        Car temp = (Car)obj;
        if (this.carId > temp.carId)
            return 1;
        if (this.carId < temp.carId)
            return -1;
        else
            return 0;
    }
}

```

A `compareTo()` mögötti logika mindössze annyi, hogy a bejövő típust adott adat alapján összehasonlítsa az aktuális példánnyal. A `compareTo()` visszatérési értékét használjuk annak megállapítására, hogy ez a típus kisebb, nagyobb vagy ugyanakkora, mint az aktuális objektum, amellyel hasonlítjuk (lásd a 9.1. táblázatot).

| compareTo()<br>visszatérési értéke |  | Valós jelentés |
|------------------------------------|--|----------------|
|------------------------------------|--|----------------|

|                               |                                                 |           |
|-------------------------------|-------------------------------------------------|-----------|
| Bármely nullánál kisebb       | A példány az aktuális objektum előtt következik | a sorban. |
| szám                          |                                                 |           |
| Nulla                         | A példány egyezik az aktuális objektummal.      |           |
| Bármely nullánál nagyobb szám | A példány az aktuális objektum után következik  | a sorban. |

9.1. táblázat: `compareTo()` visszatérési értékei

Modernizálhatjuk a `compareTo()` metódus megvalósítását annak a ténynek a tudatában, hogy a `C# int` adattípusa (amely egyszerű rövidítése a `CLR system.Int32` típusnak) megvalósítja az `IComparable` interfészt. Létre-

hozhatjuk a `car.compareTo()` metódusát a következő módon:

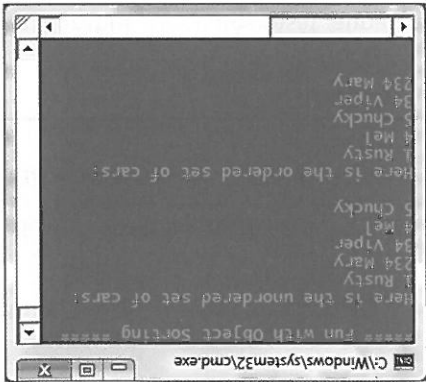
```
int IComparable.CompareTo(object obj)
{
    Car temp = (Car)obj;
    return this.CarID.CompareTo(temp.CarID);
}
```

A `car` típusunk mindkét esetben tudja, hogy hogyan kell összevetnie magát egy hasonló típusal, így a következő kódot használhatjuk:

```
// A IComparable interfész használata.
static void Main(string[] args)
{
    // Car típusok tömbjének elkészítése.
    ...
    // Az aktuális tömb megjelenítése.
    Console.WriteLine("Here is the unordered set of cars:");
    foreach(Car c in myAutos)
        Console.WriteLine("{0} {1}", c.ID, c.PetName);
    // Az elemek rendezése az IComparable segítségével.
    Array.Sort(myAutos);
    Console.WriteLine();
}
```

```
// A rendezett tömb megjelenítése.  
Console.WriteLine("Here is the ordered set of cars:");  
foreach(Car c in myAutos)  
    Console.WriteLine("{0} {1}", c.ID, c.PetName);  
Console.ReadLine();  
}
```

A 9.13. ábra mutatja a teszt futtatásának eredményét.



9.13. ábra: Autók összehasonlítása a carID alapján

## Többszörös rendezési sorrend megadása (IComparable)

A car típus fenti verziójában az azonosítókát használjuk a rendezés alapjául. Más esetben a becenevet is vehettük volna a rendezés alapjául (az autók alfabetikus listázásához). Mi a helyzet azonban akkor, ha olyan car típust szeretnénk, amely mind a becenev mind az ID alapján sorba rendezhető? Ekkor meg kell ismerkednünk egy másik szabványos interfésszel, a `System.Collections.IComparable`:

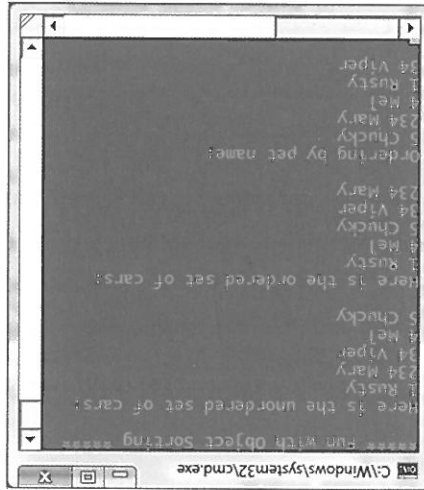
```
// két objektum összehasonlításának általános módja.  
interface IComparable  
{  
    int Compare(object o1, object o2);  
}
```

Ellentétben az `IComparable` interfésszel, az `IComparer` interfészt általában nem valósítjuk meg azokon a típusokon, amelyeket rendezni szeretnénk (azaz a caron). Inkább használjuk ezeket az interfészeket segédosztályokon, minden rendezéshöz külön (becenev, auto-ID stb). Pillanatnyilag a car típus tudja,

hogyan kell a belső ID alapján összehasonlítani magát más car példányokkal. Ennek megfelelően, ha szeretnénk megengedni a car típusok becenevé szerinti rendezését is, akkor egy további segédosztályra lesz szükségünk, amely megvalósítja az IComparer interfészt. A forráskód a következő:

```
// A segédosztályal becenev alapján rendezhető a car elemek tömbje.
using System.Collections;

public class PetNameComparer : IComparer
{
    // Az objektumok becenevének ellenőrzése.
    int IComparer.Compare(object o1, object o2)
    {
        car t1 = (car)o1;
        car t2 = (car)o2;
        return String.Compare(t1.PetName, t2.PetName);
    }
}
```



9.14. ábra: Autók rendezése becenev alapján

Az objektum forráskódjában használhatjuk ezt a segédosztályt. A System.Array osztályban vannak túlterhelt sort() metódusok, ezek közül az egyik olyan objektumot vár, amely megvalósítja az IComparer interfészt. A 9.14. ábrán az autók becenevé szerinti rendezése látható.

```
static void Main(string[] args)
{
    ...
    // Rendezés becenev szerint.
    Array.Sort(myAutos, new PetNameComparer());
}
```

```
// A rendezett tömb kitérása.
console.WriteLine("Ordering by pet name:");
foreach(Car c in myAutos)
    console.WriteLine("{0} {1}", c.ID, c.PetName);
...
}
```

## Egyedi tulajdonságok, egyedi rendezési típusok

Használhatunk egyedi statikus tulajdonságot is azért, hogy az objektumhasználatunk egyedi statikus tulajdonságot is ad vissza, amely megvalósítja az `ICompare` interfészt (petNameCompare ebben az esetben):

```
// Támogatjuk az egyedi tulajdonságot, amely
// a megfelelő IComparer interfészt adja vissza.
public class Car : IComparable
{
    ...
    // tulajdonság, amely a sortByPetName interfészt adja vissza.
    public static IComparer SortByPetName
    { get { return (Comparer)new PetNameComparer(); } }
}
```

Az objektumhasználat kódja most már rendezhet a `Compare` alapján egy közvetlen tulajdonsággal is, ahelyett, hogy „tudnia kellene” a különálló `PetNameCompare` osztályról:

```
// A becenév szerinti rendezés letisztultabb formája.
Array.Sort(myAutos, Car.SortByPetName);
```

---

**Forráskód** A `Comparable` projekttel megtalálható a 9. fejezet alkönyvtárában.

---

Eddig megismerkedtünk az interfésztípusok definiálásával és megvalósításával, valamint hasznosságukkal is. Természetesen minden .NET-névteremben találhatók interfészek. A továbbiakban nézzük meg azokat az interfészeket, amelyek visszahívás-mechanizmusokat tesznek lehetővé.

## A Callback interfész

Az interfészek a különböző osztályhierarchiák, névterek és szereplvények közötti polimorfizmus alkalmazásán kívül használhatóak *visszahívási mechanizmusok* megvalósítására is. Ez a technika lehetővé teszi az objektumok számára a kétirányú kommunikációt közös tagok használatával.

---

**Megjegyzés** A .NET-platform egy formális szerkezetet biztosít események létrehozásához (ez megtehetően elér az itt bemutatottól). A metódusreferenciákat, az eseményeket és a lambda-kifejezéseket tekinthetjük az objektumok közötti kommunikáció szabványos módjának (lásd 11. fejezet).

---

A visszahívási interfészek bemutatásához fejlesszük tovább a már jól ismert car típusunkat olyan módon, hogy képes legyen értesíteni a hívóját arról, hogy fel fog robbanni (a pillanatnyi sebesség 10 mérföld per órával van a maximális sebesség alatt), és arról, hogy felrobbant (a pillanatnyi sebesség elérte vagy meghaladta a maximális sebességet).

Kezdjük egy új callbackinterface nevű parancssori alkalmazás létrehozásával. Az események küldésének és fogadásának képességét egy új saját interfész fogja elősegíteni: az `IEngineNotification`:

```
// A visszahívási interfész.  
public interface IEngineNotification  
{  
    void AboutToBlow(string msg);  
    void Exploded(string msg);  
}
```

A callback interfészeket gyakran nem azok az objektumok valósítják meg, amelyek közvetlenül érdekeltek az események megkapásában, hanem segédobjektumok, amelyeket *nyelődobjektumnoknak* hívunk. Az események küldője (ebben az esetben a car típus) megfelelő körülmények között hívást intéz a nyelvökhöz. Tételjezzük fel, hogy a nyelvöszállynak `carEvents`-nek a neve. Ha ez az objektum értesül a különböző bejövő eseményekről, akkor egyszerűen kiírja a bejövő üzenetet a parancssorra. A nyelvö egy barátságos nevet tartalmazó sztringtagváltozót is kezel (később látni fogjuk, hogy ez miért hasznos):

```
// A car típus nyelvöszálja.  
public class CarEvents : IEngineNotification  
{  
    private string name;  
    public CarEvents(){}  
}
```

```

public CarEventsSink(string sinkName)
{
    name = sinkName;
}

public void AboutBelow(string msg)
{
    Console.WriteLine("{0} reporting: {1}", name, msg);
}

public void Exploded(string msg)
{
    Console.WriteLine("{0} reporting: {1}", name, msg);
}
}

```

Ha már van egy nyelőobjektumunk, amely megvalósítja a visszahívási inter-  
fészt, a következő feladatunk az lesz, hogy a car típusnak referenciát adjunk  
erre a nyelőobjektumra. A car tárolja a referenciát, és szükség esetén hívásokat  
indít a nyelő felé. Ahhoz, hogy a car objektum tárolhassa a referenciát a nye-  
lőobjektumra, szükségünk lesz egy segédtag hozzáadásra a car típushoz, ez lesz  
az `Advise()` módszer. Hasonlóképpen, ha a hívó szeretne lekapcsolódni az  
eseményforrásról, akkor másik segédmetódust kell hívnia a car típuson, még-  
hozza az `Unadvise()` nevűt. Végül, lehetővé téve a felhasználónak több ese-  
ménynyelő regisztrálását (többszörös esemény céljából), a car típus egy `Array-`  
`List` elemet is kezel, amely tartalmaz minden egyes meglévő kapcsolatot.

---

**Megjegyzés** Az `ArrayList`-t osztályt a `mscorlib.dll` szerezvény `System.Collections` név-  
teret tartalmazza. Ezt a névteret importálni kell a car osztály definícióját tartalmazó fájlban.  
A gyűjteményeket (és a generikus típusokat) lásd a 10. fejezetben.

---

```

// A car osztály és a hívó
// az IEngineNotification interfész segítségével kommunikál.
public class Car
{
    // Kapcsolt nyelőobjektumok készlete.
    ArrayList clients = new ArrayList();

    // Csatlakozás vagy leválás a forráseseményekről.
    public void Advise(IEngineNotification sink)
    {
        clients.Add(sink);
    }

    public void Unadvise(IEngineNotification sink)
    {
        clients.Remove(sink);
    }

    ...
}

```



Az események tényleges elküldetéséhez frissítsük a `car.accelerate()` metódust úgy, hogy járja be a kapcsolatok listáját az `ArrayList`-objektumban, és szükség esetén indítsa el a helyes értesítést. Először is hozzunk létre egy `carIsDead` logikai tagváltozót, hogy a motor állapotát megjelentsük:

```
class Car
{
    // Az autó működik vagy lerobant?
    bool carIsDead;
    ...
}
```

Ezután egészítsük ki az `accelerate()` metódusunkat úgy, hogy használja ezt az új tagváltozót:

```
public void accelerate(int delta)
{
    // Ha az autó lerobant, az exploded esemény kiáltása.
    if (carIsDead)
    {
        foreach (EngineNotification sink in clientsinks)
            sink.Exploded("Sorry, this car is dead...");
    }
    else
    {
        currspeed += delta;

        // Majdnem lerobant?
        if (10 == (maxspeed - currspeed))
        {
            foreach (EngineNotification sink in clientsinks)
                sink.AboutToBlow("Careful buddy! gonna blow!");
        }
        // Még mindig rendben!
        if (currspeed >= maxspeed)
            carIsDead = true;
        else
            Console.WriteLine("<currspeed = " + currspeed);
    }
}
```

Ezzel az infrastruktúrával már megvalósíthatjuk a `main()` metódusunkat úgy, hogy megkapja ezeket az eseményeket a `car` típustól:

```
// Autót készítünk, és figyeljük az eseményeket.
static void main(string[] args)
{
    Console.WriteLine("***** Interfaces as event enablers *****\n");
    Car c1 = new Car("StuBug", 100, 10);
}
```

```

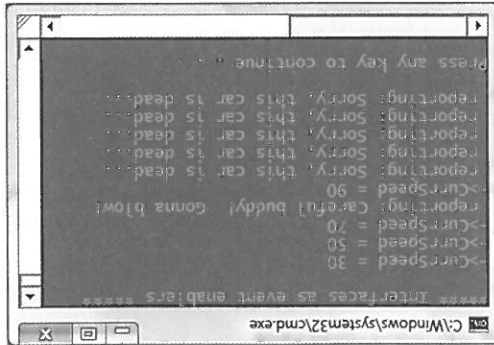
static void main(string[] args)
{
    Console.WriteLine("***** Interfaces as event enablers *****\n");
    Car c1 = new Car("StugBug", 100, 10);

    // két nyelődobjektum készítése.
    Console.WriteLine("***** Creating sinks *****");
    CarEventsSink sink = new CarEventsSink("First sink");
    CarEventsSink myOtherSink = new CarEventsSink("Other sink");
}

```

Az `unadvise()` metódus a hívó számára kifejezetten hasznos, mert így szelektíven le tud válni az eseményforrásokról. A `main()` metódusból kilépés előtt hívjuk az `unadvise()` metódust (bár ez technikailag nem feltétlenül szükséges). Tételizzzuk fel, hogy a program két nyelőt szeretne regisztrálni, dinamikusan elítárolítani egy adott nyelőt a végrehajtási folyamat során és folytatni a program futását:

9.15. ábra: Eseményértesítés vétele visszahívási interfész használatával



A 9.15. ábra mutatja az interfészalapú esemény protokollereeményeit.

```

// Nyelődobjektum készítése.
CarEventsSink sink = new CarEventsSink();

// Car referencia átadása a nyelőnek.
c1.Advise(sink);

// Gyorsítás (ezzel kíváltkunk néhány eseményt).
for(int i = 0; i < 10; i++)
    c1.Accelerate(20);

// Leválasztás az esemény forrásáról.
c1.Unadvise(sink);
Console.ReadLine();
}

```

```
// Nyelődobjektumok átadása a car típusnak.
Console.WriteLine("\n***** Sending 2 sinks into car *****");
cl.Advise(sink);
cl.Advise(myOthersink);

// Gyorsítás (ezzel kiváltunk néhány eseményt).
Console.WriteLine("\n***** Speeding up *****");
for(int i = 0; i < 10; i++)
    cl.Accelerate(20);

// Az első nyelődobjektum leválasztása az eseményekről.
Console.WriteLine("\n***** Removing first sink *****");
cl.Unadvise(sink);

// Ismét gyorsítás (csak a myOthersink nyelőt hívja).
Console.WriteLine("\n***** Speeding up again *****");
for(int i = 0; i < 10; i++)
    cl.Accelerate(20);

// Másik nyelődobjektum leválasztása az eseményekről.
Console.WriteLine("\n***** Removing second sink *****");
cl.Unadvise(myOthersink);
Console.ReadLine();
}
```

A visszahívási interfészek hasznosak lehetnek azért is, mert bármely nyelvben vagy platformon (.NET, J2EE vagy egyéb) használhatóak, ha azok támogatják az interfészalapú programozást. (A 11. fejezetben még részleteseen foglalkozunk néhány eseménycentrikus technikával, amelyek .NET-platform specifikusak.)

---

**Forráskód** A CallBackInterface projekt megtalálható a 9. fejezet alkönyvtárában.

---

## Összefoglalás

Egy interfészt *absztrakt lagok* nevesített gyűjteményeként definiálhatunk. Minél az interfészek nem nyújtanak semmilyen megvalósítási részletet, általában az interfészeket olyan viselkedésnek tekintjük, amelyek egy adott típus támogatását. Ha két vagy több osztály valósítja meg ugyanazt az interfészt, akkor mindegyik típust azonosan kezelhetjük (más néven ez az interfészalapú polimorfizmus), még akkor is, ha ezek a típusok egyedi osztályhierarchiákban vannak.

A C# nyelv rendelkezésünkre bocsátja az interface kulcsszót, amellyel új interfészt tudunk létrehozni. Egy típus annyi interfészt támogathat, amennyire csak szükség van, vesszővel elválasztott lista használataival. Továbbá lehetőség van olyan interfészek létrehozására is, amelyek több ösinterfészből származnak.

A saját interfészek létrehozásán kívül, a .NET-könyvtár definiál egy sor szabványos (azaz keretrendszer nyújtotta) interfészt is. Szabadon létrehozhatunk saját típusokat, amelyek megvalósítják ezeket az előre definiált interfészeket, hogy elérhessünk olyan igényelt jellegzetességeket, mint a klónozás, a rendezés és a felsorolás. Végül megneztük, hogy az interfésztípusok hogyan használhatók két-irányú kommunikációra a memóriában lévő két objektum között.

# Gyűjtemények és generikus típusok

A .NET-platfornon belüli leggyyszerűbb tároló a `system.array` típus. Ahogy láttuk, a `C#`-tömbök lehetővé teszik, hogy a `system.array` segítségével előre megadott számú, azonos típusú elemet tároljunk (beleértve a `system.object`ek tömbjét, amely lényegében tetszőleges típusú elemek tömbjét reprezentálja). Sokszor azonban ennél rugalmasabb adatstruktúrára van szükségünk, mint például a dinamikusan növekvő és zsugorodó tároló vagy egy olyan tároló, amely csak a meghatározott feltételeknek megfelelő elemeket tartalmazhatja (pl. csak a megadott össztájból származó osztályokat vagy bizonyos interfészeket megvalósító elemeket). A rugalmas és típusbiztos tárolók készítésének vizsgálatához ebben a fejezetben először a `system.collections` névtérrel tanulmányozzuk, amely a kezdeteinktől a .NET-alapozástáljkönyvtár része.

A .NET 2.0 kibocsátásakor azonban a `C#` programnyelvet kiegészítették a *generikus típusok* támogatásával. A generikus típusok közül több, naponta használatos a `system.collections.generic` névtérben található. A generikus tárolók felülműlják nem generikus párukat azzal, hogy nagyobb típusbiztonságot és jobb teljesítményt nyújtanak. Már láttunk példát az össztályon belüli generikus támogatásra, a továbbiakban megmutatjuk, hogyan építhetünk saját generikus tagokat, osztályokat, struktúrákat és interfészeket.

---

**Megjegyzés** Metódusreferencia-típusokat is létre lehet hozni. Ezekkel a következő fejezetben foglalkozunk.

---

## A System.Collections névtér interfészei

A leggyyszerűbb tárolószerkezet a már jól ismert `system.array`. Ez az osztály számos szolgáltatást nyújt (pl. Elemek sorrendjének megfordítása, rendezés, törlés és felsorolás; lásd 4. fejezet). A `system.array` azonban számos korláta van; különösen az, hogy nem méretezi újra magát automatikusan

elemek hozzáadásakor vagy elvételkor. Amikor rugalmasabb tárolóban kell a típusokat tárolni, az egyik lehetőség a `system.collections` névtérben definiált típusok használata.

A `system.collections` névtér számos interfészt definiál (közülük sokat már megvalósítottunk a 9. fejezetben). A `system.collections` névtérben található osztályok többsége megvalósítja ezeket az interfészeket, így hozzáférést biztosítanak a tartalmukhoz. A 10.1. táblázat összefoglalja a lényeges gyűjteménycentrikus interfészeket..

System.Collections

Interfész

Használata

|                                    |                                                                                                                                                                                                                           |
|------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>IComparer</code>             | Lehetővé teszi két objektum összehasonlítását.                                                                                                                                                                            |
| <code>IDictionary</code>           | Lehetővé teszi név/érték párok használatával a nem generikus gyűjtemény objektumtartalmának megjelölését.                                                                                                                 |
| <code>IDictionaryEnumerator</code> | <code>IDictionary</code> interfészt támogató típus tartalmának felsorolását biztosítja.                                                                                                                                   |
| <code>IEnumerable</code>           | Visszaadja egy adott objektum <code>IEnumerator</code> interfészét.                                                                                                                                                       |
| <code>IEnumerator</code>           | <code>foreach</code> típusú iterációt tesz lehetővé az alitípusokon.                                                                                                                                                      |
| <code>HashCodeProvider</code>      | Visszaadja a megvalósító típus <code>hash</code> kódját, testre szabott <code>hash</code> algoritmust használva.                                                                                                          |
| <code>IList</code>                 | Az elemek hozzáadását, elávolítását és indexelését biztosítja az objektumok listáján. Tagokat definiál a létrehozott gyűjtemény számára, ezek pedig meghatározzák a tároló típusát: csak olvasható, illetve fix méretű-e. |

10.1. táblázat System.Collections interfészei

Ezek közül az interfészek közül jó néhány összekapcsolódik egy adott interfészhierarchia révén, ezzel szemben a többi egyedülálló egyed. 10.1. ábra a közöttük lévő kapcsolatot illusztrálja (emlékezzünk rá, hogy egy szimpla interfésznek megengedhető, hogy több interfészből származzon).

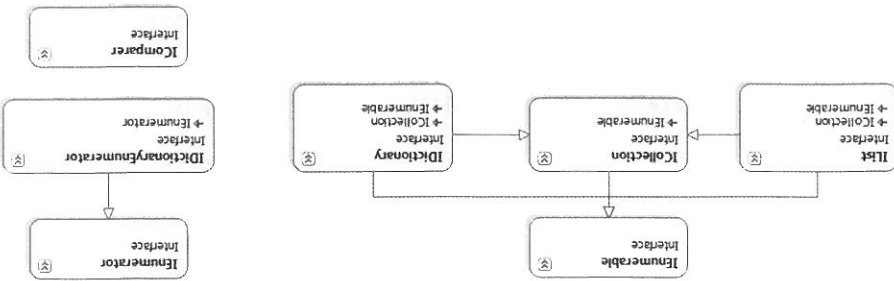
A *szoftár* egy olyan gyűjtemény, amely név/érték párokat tart fenn. Egyedi típusokat készíthetünk, amelyek az IDictionary interfészt valósítják meg például car típusok tárolása (értékek), amelyet aztán kinyerhetünk valamilyen azonosítóval (az értékhez tartozó nevek). Látható, hogy ezzel a működéssel az IDictionary interfész Keys (kulcsok) és Values (értékek) tulajdonságokat határoz meg, valamint Add(), Remove() és Contains() metódusokat. Hozzájárulhatunk az egyes elemekhez a típusindexelő segítségével. Ez a szerkezet a tömbtípusú szintaxissal teszi lehetővé az elemek használatát. A formális definíció a következő:

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo(Array array, int index);
}
```

Az ICollection a System.Collections névtér leggyakoribb olyan interfésze, amely a gyűjteménytípus által támogatott viselkedést definiálja. Röviden: ez az interfész olyan tagok kisebb gyűjteményét biztosítja, amely lehetővé teszi, hogy meghatározzuk: a) az elemek számát a tárolóban, b) a tároló szálbiztos-ságát, valamint c) a System.Array típusba való másolás képességét. Hivatalosan az ICollection interfészt a következőképpen definiáljuk (az ICollection kiterjeszti az IEnumerable interfészt):

## Az ICollection interfész

10.1. ábra A System.Collections interfészhierarchia



A System.Collections névtér interfészei

```

public interface IDictionary<TKey, TValue> : IEnumerable<KeyValuePair<TKey, TValue>>
{
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[object key] { get; set; }

    // Típus indexelő; részletekért lásd a 12. fejezetet.
    ICollection<TKey> Keys { get; }
    ICollection<TValue> Values { get; }
    void Add(object key, object value);
    void Clear();
    bool Contains(object key);
    IDictionaryEnumerator GetEnumerator();
    void Remove(object key);
}

```

## Az IDictionaryEnumerator interfész

Az előző fejezetben láttuk, hogy az `IDictionary.GetEnumerator()` metódus hívása `IDictionaryEnumerator` típusú példánnyal tér vissza. Az `IDictionaryEnumerator` interfész egy olyan erősen típusos felsoroló, amely az `IEnumerator` interfész kiterjesztése, a következő hozzáadott funkcionálisítással:

```

public interface IDictionaryEnumerator : IEnumerator
{
    DictionaryEntry Entry { get; }
    object Key { get; }
    object Value { get; }
}

```

Az `IDictionaryEnumerator` interfész a szótárban szereplő tételek felsorolását teszi lehetővé az általánosított `Entry` tulajdonsággal, amely `system.Collections.DictionaryEntry` típusú osztállyal tér vissza. Továbbá a név/érték párokat a `Key` (kulcs) és a `Value` (érték) tulajdonságok felhasználásával képesek vagyunk végigjárni.

## Az IList interfész

Az utolsó központi interfész a `System.Collections.IList`, amely tételek tárolókba illesztésének, eltávolításának és indexelésének képességét biztosítja:



```
public interface IList :  
    ICollection, IEnumerable  
{  
    bool IsFixedSize { get; }  
    bool IsReadOnly { get; }  
    object this[ int index ] { get; set; }  
    int Add(object value);  
    void Clear();  
    bool Contains(object value);  
    int IndexOf(object value);  
    void Insert(int index, object value);  
    void Remove(object value);  
    void RemoveAt(int index);  
}
```

## A System.Collections névtér osztálytípusai

Az interfészek önmagukban nem igazán hasznosak egészen addig, amíg nem implementáljuk őket egy adott osztállyal vagy struktúrával. A 10.2. táblázat összefoglalja a System.Collections névtérben található központi osztályokat és az általuk támogatott fontosabb interfészeket.

| System.Collections<br>osztály | Használata | A megvalósított<br>fontosabb<br>interfészek |
|-------------------------------|------------|---------------------------------------------|
|-------------------------------|------------|---------------------------------------------|

|           |                                                                                                                                                                                                    |                                                             |
|-----------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------|
| ArrayList | Objektumok dinamikusan mé-<br>retezett tömbjét reprezentálja.                                                                                                                                      | IList, ICollection, IEnumerable és<br>ICollectionable       |
| HashTable | Numerikus kulcs által definiált<br>objektumok gyűjteményét rep-<br>rezentálja. A hash táblában tárolt<br>egyedi típusoknak mindig felül<br>kell írniuk a System.Object.<br>GetHashCode() metódust. | IDictionary, ICollection, IEnumerable és<br>ICollectionable |
| Queue     | Szabványos FIFO-sort<br>reprezentál.                                                                                                                                                               | ICollection, ICollectionable és<br>IEnumerable              |

| System.Collections |  | Használata | A megvalósított fontosabb interfészek |
|--------------------|--|------------|---------------------------------------|
|--------------------|--|------------|---------------------------------------|

|            |                                |                                              |                            |
|------------|--------------------------------|----------------------------------------------|----------------------------|
| Sortedlist | A szótárhoz hasonló tároló, am | IDictionary, az elemekhez is hozzáférhetünk  | ICollection és IEnumerable |
|            | lással).                       | a sorszámokkal (például indexe-ICollection   |                            |
| Stack      | Push, pop és peek műveleteket  | ICollection, támogató LIFO-sort valósít meg. | ICollection és IEnumerable |

10.2. táblázat: System.Collections névtér osztályai

A fenti fontosabb típusok mellett a System.Collections névtér tartalmaz né-  
hány kevésbé jelentős osztályt is (legálábbis a mindennapi használhatóságukat  
tekintve): ezek a BitArray, a CaseInsensitiveComparer és a CaseInsensitiveVe-  
HashCodeProvider. Továbbá ez a névtér absztrakt ösztályok kis gyűjteményé-  
vel is rendelkezik (ICollectionBase, ReadOnlyCollectionBase és DictionaryBase),  
amelyek felhasználhatóak erősen típusos tárolók készítéséhez.  
A System.Collections névtér típusai közös funkciók megosztására töre-  
kednek (ez az interfészalapú programozás lényege). Így nem fogjuk minden  
gyűjteményosztály tagjait felsorolni, hanem három gyakori gyűjteménytípust  
mutatunk be: az ArrayList, a queue és a stack gyűjteményeket.  
Ezeknek a típusoknak a működésén keresztül könnyedén megértjük a  
többi gyűjteményosztályt is (pl. a hashtable-t). Az összes típus teljes doku-  
mentációja megtalálható a .NET Framework 3.5 leírásában.

## Az ArrayList típus használata

A gyűjteménytípusok szemléltetésére hoztunk létre egy új collectiontypes  
nevű parancssoralkalmazás-projektet. Az ArrayListben egyszerű car objek-  
tumokból álló csoportot tartunk fenn, amelyet következőképpen definiálunk:

```
class Car
{
    // Az egyszerűség kedvéért publikus.
    public string PetName;
    public int Speed;
```

```
// konstruktorok.
public Car(string name, int currentSpeed)
{ PetName = name; Speed = currentSpeed; }
}
```

Módosítsuk a projektet kezdett C#-fájlt a System.Collections névtér használatának megadásával:

```
using System.Collections;
```

Az Arraylist valószínűleg a leggyakrabban használt olyan típus a System.Collections névtérben, amely lehetővé teszi a tároló dinamikus átméretezését. Az alapjainak szemléltetésére vizsgáljuk meg a következő Program osztályba tartozó metódust, amelynek hatására az Arraylist megváltoztatja a Car objektumok gyűjteményét:

```
static void ArraylistTest()
{
    Console.WriteLine("\n=> Arraylist Test:\n");
    // Arraylist létrehozása és feltöltése kezdőértékekkel.
    Arraylist carArlist = new Arraylist();
    carArlist.AddRange(new Car[] { new Car("Fred", 90, 10),
    new Car("Mary", 100, 50), new Car("MB", 190, 11)});
    // Az Arraylist elemszámának kitírása.
    Console.WriteLine("Items in carArlist: {0}", carArlist.Count);
    // Aktuális értékek kitírása.
    foreach(Car c in carArlist)
        Console.WriteLine("Car pet name: {0}", c.PetName);
    // Új elem beszúrása.
    Console.WriteLine(">Inserting new Car.");
    carArlist.Insert(2, new Car("TheNewCar", 0, 12));
    Console.WriteLine("Items in carArlist: {0}", carArlist.Count);
    // Objektum tömb kiolvasása és kitírása az Arraylist-ből.
    object[] arrayOfCars = carArlist.ToArray();
    for(int i = 0; i < arrayOfCars.Length; i++)
    {
        Console.WriteLine("Car pet name: {0}",
            ((Car)arrayOfCars[i]).PetName);
    }
}
```

Az `AddRange()` módszer felhasználásával benépesítjük az `ArrayList`-et `car` típusú elemekkel (ez az `Add()` módszer többszöri meghívásával egyenértékű). Amint kitirajtjuk a gyűjteményben található tételek számát (és az összes tételt felsoroljuk, hogy megkapjuk az autók nevét), meghívhatjuk az `Insert()` metódust. Az `Insert()` segítségével tudunk egy új tételt megfelelő indexszel az `ArrayList`-be beszúrni.

Végül a `ToArray()` módszer hívása az eredeti `ArrayList`-t tartalmára építve `system.object` típusú tömbjével tér vissza. Ebből a tömbből az indexelő szintaxis felhasználásával még egyszer végiglépkedünk az elemeken.

## A Queue típus használata

A sorok olyan tárolók, amelyek gondoskodnak arról, hogy a tételek hozzá-  
férhetők legyenek az „elsőnek be, elsőnek ki”, azaz a FIFO módszerrel. Egész  
nap ki vagyunk téve a sorban állásnak: sorban állunk a banknál, a mozinál és  
reggel a kávéház előtt. Ha olyan modellt használunk, amelyben a tételeket az  
„először jött, először lesz kiszolgálva” elv alapján kezeljük, akkor a `system.  
collections.queue` osztály a megfelelő választás. A támogatott interfészek ál-  
tal biztosított funkcionálitásán túl a `queue` osztály kulcsfontosságú tagjait a  
10.3. táblázatban tüntettük fel.

| System.Collections.Queue<br>tagjai |  |
|------------------------------------|--|
| Használata                         |  |

|           |                                                                                        |
|-----------|----------------------------------------------------------------------------------------|
| Dequeue() | Eltávolítja és visszaadja a sor elején található objektumot.                           |
| Enqueue() | Hozzáad egy objektumot a sor végéhez.                                                  |
| Peek()    | A sor elején található objektummal tér vissza, anélkül, hogy azt eltávolítsa a sorból. |

10.3. táblázat: Queue típus tagjai

A metódusok szemléltetésére még egyszer módosítsuk az autós példánkat, és hozzunk létre objektumot, amely a mosásra váró autók sorát szimulálja. Először is vegyük a következő statikus segédfüggvényt:

```
static void WashCar(Car c)
{
    Console.WriteLine("Cleaning {0}", c.PetName);
}
```

Ezután vegyük a következő segédmetódust, amely meghívja a `washcar()` me-

tódust:

```
static void QueueTest()
{
    Console.WriteLine("\n=> Queue Test:\n");
    // Sor készítése három elemmel.
    Queue carwashq = new Queue();
    carwashq.Enqueue(new Car("Firstcar", 10));
    carwashq.Enqueue(new Car("Secondcar", 20));
    carwashq.Enqueue(new Car("Thirdcar", 30));
    // Az első autó megtekintése a sorban.
    Console.WriteLine("First in q is {0}",
        ((Car)carwashq.Peek()).PetName);
    // Elemek eltávolítása a sorból.
    carwashq.Enqueue();
    carwashq.Enqueue();
    carwashq.Enqueue();
    carwashq.Enqueue();
    carwashq.Enqueue();
    carwashq.Enqueue();
    // Ismét megpróbálunk elemeket eltávolítani a sorból?
    try
    {
        washcar((Car)carwashq.Dequeue());
    }
    catch (Exception e)
    {
        Console.WriteLine("Error! {0}", e.Message);
    }
}
```

Három tételt helyezünk a `queue` típusba az `enqueue()` metódussal. A `peek()` meghívásával megnézhetjük a `queue` első elemét (anélkül, hogy eltávolítsuk), amely ebben az esetben a `firstcar` nevű objektum. Végül a `dequeue()` meghívásával eltávolítjuk az elemet a sorból, és elküldjük a `washcar()` feldolgozó segédfüggvénynek. Ha egy üres sorból próbálunk elemet kitörölni, akkor futásidejű kivétel dobódik.

## A Stack típus használata

A `System.Collections.Stack` típus reprezentálja az „utolsónak be, elsőnek ki” (LIFO) szabállyal fenntartott gyűjteményt. A `stack` egy `push()` és egy `pop()` tagot definiál (belehelyez, illetve kivessz tételleket a veremből). A következő verem az alapvető használatot példázza:

```
static void stackTest()
```

```
{
    Console.WriteLine("\n=> stack Test:\n");
    Stack<string> stack = new Stack<>();
    stack.Push("one");
    stack.Push("two");
    stack.Push("three");

    // A legfelső elem megtekintése, levétele, a verem megtekintése.
    Console.WriteLine("Top item is: {0}", stringStack.Peek());
    Console.WriteLine("Popped off {0}", stringStack.Pop());
    Console.WriteLine("Top item is: {0}", stringStack.Peek());
    Console.WriteLine("Popped off {0}", stringStack.Pop());
    Console.WriteLine("Top item is: {0}", stringStack.Peek());
    Console.WriteLine("Popped off {0}", stringStack.Pop());
    Console.WriteLine("Top item is: {0}", stringStack.Peek());
    Console.WriteLine("Popped off {0}", stringStack.Pop());
    try
    {
        Console.WriteLine("Top item is: {0}", stringStack.Peek());
        Console.WriteLine("Popped off {0}", stringStack.Pop());
    }
    catch (Exception e)
    {
        Console.WriteLine("Error! {0}", e.Message);
    }
}
```

Három sztringtípust tartalmazó vermet készítettünk (a beszúrás sorrendje alapján nevezve el őket). Ha belenézünk a verembe, akkor mindig a legfelső tételt fogjuk látni, és így a `Peek()` metódus első hívása a harmadik sztringet fogja visszaadni. A `Pop()` és a `Peek()` hívások sorozata után a verem végül üres lesz, további `Peek()/Pop()` hívások hatására rendszerkivétel keletkezik.

---

**Forráskód** A `CollectionTypes` projekt megtalálható a 10. fejezet alkönyvtárában.

---

## A System.Collections.Specialized névtér

Erdemes figyelmeztelnünk a .NET-alapozott `System.dll` szerezvényében definiált `System.Collections.Specialized` névtérnek is, amely egy másik, jobban specializált csoportot definiál. A `StringDictionary` és a `StringDictionary` típusok például erősen típusos megvalósítást biztosítanak az `IDictionary` interfészből. A 10.4. táblázat dokumentálja a legtöbb osztálytípusokat.

|                                |           |
|--------------------------------|-----------|
| System.Collections.Specialized | Használat |
| tagjai                         |           |

BitVector32

Ez egy olyan egyszerű struktúra, amely logikai értékeket tárol, valamint kis egész számokat 32 biten.

CollectionUtil

Olyan gyűjteményt hoz létre, amely nem tesz különbséget a sztringekben a nagy- és a kisbetűk között.

HybridDictionary

A HybridDictionary osztály használatával az IDictionary interfészt valósítja meg kis gyűjteményeknél, majd a Dictionary felcseréli a Hashtable osztályra, amikor a gyűjtemény növekszik.

ListDictionary

Egyszeresen láncolt lista használatával megvalósítja az IDictionary interfészt. Legfeljebb tíz elemmel rendelkező tárológyűjteményekhez ajánlott.

NameValueCollection

Sztringkulcsok és sztringértékek párainak gyűjteményét reprezentálja, amelyhez a kulcs-csal vagy indexszel lehet hozzáférni.

StringCollection

Sztringek gyűjteményét reprezentálja.

StringDictionary

Hasháblát valósít meg objektum helyett erősen típusos sztringkulccsal.

StringEnumerator

Egyszerű iterációt támogat a StringCollection osztályon.

10.4. táblázat: A System.Collections.Specialized névtér típusai

A System.Collections névtér (és a System.Collections.Specialized névtér) belüli alapvető gyűjteménytípusokat, mint ahogy régi típusok, nem szoktuk felhasználni új projektek készítéséhez .NET 2.0 vagy későbbi változataiban. Nem azért, mert ezek a típusok valamilyen szempontból veszélyesek lennének, hanem a teljesítmény hatásfoka és a típusbiztonság hiánya az ok. Az új projekteknek figyelmen kívül kellene hagyniuk ezeket a ránk maradt tárolótípusokat, és a System.Collections névtér rokon típusait kellene támogatniuk. A generikus típusok tanulmányozása előtt célszerű pontosítaniunk, hogy ezekkel a típusokkal milyen problémákat fogunk tudni megoldani, ehhez pedig meg kell vizsgálnunk a be-, illetve a kidobozolás szerepét.

## A bedobozolás, kidobozolás és a System.Object kapcsolata

A .NET-platform két nagyobb adattípuscsoportot támogat, az *értéktípusokat* és a *referenciátípusokat* (lásd 4. fejezet). A .NET két fő típuskategoriat definiál, néha az egyik kategóriába tartozó változót a másikba tartozóként kell felhasználnunk. Hogy ezt megtehesünk, a C# egy nagyon egyszerű működési elvet biztosít: értéktípust referenciátípussá alakít, ezt a működést nevezzük *dobozolásnak*. Ehhez hozzunk létre egy short típusú változót:

```
// short típusú változó létrehozása.
short s = 25;
```

Ha az alkalmazás futása alatt ezt a változót referenciátípusként szeretnénk kifejezni, akkor a következőképpen dobozoljuk az értéket:

```
// Az érték dobozása objektum referenciába.
object objshort = s;
```

A dobozolás tehát olyan folyamat, amelynek során az értéktípust megfelelő referenciátípussá alakítjuk, miközben a változó a system.object típusban tárolódik. Amikor egy értéket bedobozolunk, a CLR egy új objektumot határoz meg a heapen, és átmásolja az értéktípus értékét (a fenti esetben ez 25) a példányba. Az újonnan létrehozott objektumhoz egy referencia tér vissza. Ezzel a technikával a .NET-fejlesztőknek nem kell burkolóosztályokat létrehoznunk a veremadatok mint heapen tárolt objektumok ideiglenes kezeléséhez. Az ellentétes művelet ugyancsak elérhető a *kidobozolásnak* köszönhetően. A kidobozolás az a folyamat, amely az értéket az objektumreferenciából megfelelő értéktípussá alakítja vissza a veremben. A kidobozolás az átvett adatit pus ellenőrzésével kezdődik, és ha ez azonos a bedobozolt típussal, akkor az érték a helyi veremalapú változóba másolódik. A következő kidobozolási műveletek sikeresen futnak, ha az objshort alapjául szolgáló típus valóban short:

```
// A referencia kidobozolása a megfelelő short értékre.
short anothershort = (short)objshort;
```

Fontos, hogy a megfelelő adattípusba dobozoljuk ki. Ennek megfelelően a következő kidobozolási logika InvalidCastException kivételt generál:



```
// Ervénytelen kidobozolás.
static void Main(string[] args)
{
    short s = 25;
    object objShort = s;

    try
    {
        // A dobozban található érték NEM int, hanem short!
        int i = (int)objShort;
    }
    catch (InvalidCastException e)
    {
        Console.WriteLine("Oops!\n{0} ", e.ToString());
    }
}
```

Első ránézésre a ki- és a bedobozolás hasonlíthat egy jelentéktelen nyelvi sajátosságához, amely inkább elméleti, mint gyakorlati jellegű. A valóságban a dobozolás azonban nagyon hasznos, segítségével minden típus `System.Object`-ként kezelhető, ugyanis a CLR gondoskodik helyettünk a memóriával kapcsolatos részletekről.

A módszer gyakorlati használatának bemutatásához vegyünk egy `System.Collections.ArrayList`-et, amely numerikus (veremfoglat) adatot tárol. Az `ArrayList`-eket tagjai típusuk `System.Object` típusok tárolását hivatottak kezelni:

```
public class System.Collections.ArrayList : Object,
    System.Collections.IList,
    System.Collections.ICollection,
    System.Collections.IEnumerable,
    ICollection
{
    ...
    public virtual int Add(object value);
    public virtual void Insert(int index, object value);
    public virtual void Remove(object obj);
    public virtual object this[int index] {get; set;}
}
```

A programozóknak nem kell manuálisan objektumba csomagolniuk a veremalapú eszközöket, mert a futtatórendszer ezt automatikusan végrehajtja a dobozolás segítségével:

```

static void Main(string[] args)
{
    // Az értéktípusokat a rendszer automatikusan dobozolja,
    // ha átadja az objektumot kérő tagnak.
    ArrayList myInts = new ArrayList();
    myInts.Add(10);
    Console.WriteLine();
}

```

Há ki szeretnénk nyerni a típusindexelő segítségével ezt az értéket az array-list objektumból, akkor a kasztozás segítségével ki kell dobozolnunk a heapen lefoglalt objektumot egy vermen lefoglalt egésszé:

```

static void Main(string[] args)
{

```

```

    ...
    // Az értéket kidobozoltuk.
    int i = (int)myInts[0];

```

```

    // Ujradobozoltuk, mivel a WriteLine() objektumtípusokat követeli
    Console.WriteLine("Value of your int: {0}", i)
    Console.ReadLine();
}

```

Amikor a C#-fordító átalakítja a bedobozolást köztes nyelvi kóddá, akkor belül a box műveletkódot használja. Hasonlóképpen a kidobozolás művelete CIL, un-box műveleté alakul át. Nézzük meg a kapcsolódó köztes nyelvi kódot az előző main() metódussal (amelyet megtekinthetünk az `ildasm.exe` segítségével):

```

.method private hidebysig static void Main(string[] args)
{
    cil managed
    ...
    box [mscorlib]System.Int32
        callvirt instance int32
            [mscorlib]System.Collections.ArrayList::Add(object)
        pop
        ldstr "Value of your int: {0}"
        ldc.i4.0
        ldc.i4.0
        callvirt instance object [mscorlib]
            System.Collections.ArrayList::getItem(int32)
        unbox [mscorlib]System.Int32
        ldind.i4
        box [mscorlib]System.Int32
        call void [mscorlib]System.Console::WriteLine(string, object)
    ...
}

```

A veremfoglalt `system.int32` dobozolója az `ArrayList.Add()` metódus meghívását megelőzően megtörténik, így előáll a megfelelő `system.object`. A `system.object`-et visszadobozoljuk a `system.int32`-be, amikor a típusindexelő használatalával az `ArrayList`-ből lekérdezzük az elemet (ez utóbbit leképezzük egy rejtett `get_item()` metódusba), és csak akkor *dobozoljuk be újra*, amikor átadjuk a `console.WriteLine()` metódusnak. Ebben a metódusban `system.object` típussal dolgozunk.

## A dobozással járó probléma

A programozó szemszögéből a dobozolás nagyon kényelmes megoldás, ez az egyszerűsített megközelítés azonban a `stack` és a `heap` közötti memóriamásolás miatt a teljesítmény csökkenéséhez (mind a futási sebesség, mind a kód-méret tekintetében) és a típusbiztonság hiányához vezet. A teljesítmény csökkenésére nézzük meg a dobozolásakor legfőbb ször alkalmazzott lépéseket egy egyszerű esetben:

1. Az új objektumot le kell foglalni a felügyelt `heap`-ben.
2. A veremalapú adat értéket át kell másolni az új objektum helyére.
3. Amikor kidobozolunk, a `heap`alapú objektumban tárolt értéket vissza kell másolni a verembe.
4. Az éppen nem használt objektum a `heap`-en (végül) a személgyűjtés je-lőltje lesz.

A jelenlegi `main()` metódus nem okoz szűk keresztmetszetet a teljesítményben, de természetesen érződik a kihatása akkor, ha egy `ArrayList`-t több ezer olyan egészet tartalmaz, amelyeket rendszeresen változtatunk a programunkban. Vizsgáljuk meg a típusbiztonság hiányát tekintve a kidobozolás műveletét. Egy `C#` által használt érték kidobozolásához a kasztolás szintaxisát használjuk. A kasztolás sikeresége azonban csak *futásidőben* derül ki. Ennek következtében, ha egy értéket rossz adattípussal próbálunk kidobozolni, akkor `InvalidCastException`ot kapunk:

```
static void Main(string[] args)
{
    ArrayList myInts = new ArrayList();
    myInts.Add(10);
}
```

```
// Futásidejű kivétel!
short i = (int)myints[0];

// újradobozoltuk, mivel a WriteLine() objektumtípusokat követeli
console.WriteLine("Value of your int: {0}", i);
console.ReadLine();
}
```

Ideális esetben a C#-fordító képes lenne megoldani ezeket az illegális kido-  
bozolás! műveleteket fordítási időben. *Valóban* ideális esetben el tudnánk tá-  
rolni az értékítételeket egy tárolóban, anélkül hogy dobozolásra lenne szük-  
ség. A generikus típusok jelentik a választ ezekre a kihívásokra.

## A típusbiztonság és az erősen típusos gyűjtemények

Egy generikus típusok nélküli világban a `System.Collections` névtér típusait  
használhatjuk tetszőleges elemek tárolására, ezeket ugyanis úgy tervezték,  
hogy `System.Object`-ekkel dolgozzanak:

```
static void Main(string[] args)
{
    // Az ArrayList bármít tárolhat.
    ArrayList a1MyObject = new ArrayList();
    a1MyObject.Add(true);
    a1MyObject.Add(new Car());
    a1MyObject.Add(66);
    a1MyObject.Add(3.14);
}
```

Esetenként rendkívül rugalmas tárolókra lesz szükségünk, amelyek szó sze-  
rint mindennek a tárolására alkalmasak. A legtöbbszor azonban *típusbiztos*  
*tárolót* szeretnénk, amely csak bizonyos típusú adatokkal működik. Szüksé-  
günk lehet például egy olyan tárolóra, amely csak adatbázis-kapcsolatokat,  
bittérképeket, `IntPtr`-kompatibilis objektumokat vagy hasonlóan specifikus  
dolgozat képes tárolni.

## Egyedi gyűjtemény készítése

A típusbiztonság és az erősen típusos gyűjtemények

A .NET 2.0-beli generikus típusok bevezetése előtt a programozók a típusbiztonság érdekében egyedi, erősen típusos gyűjteményeket készítettek. A problémákma érzeléket hozunk létre egy customNonGenericCollection nevű parancssoralkalmazás-projektet. Mindenképpen importáljuk a System.Collections.Névteret. Tegyük fel, hogy létre akarunk hozni egy egyedi gyűjteményt, amely csak Person típusú objektumot tartalmazhat:

```
public class Person
{
    // Az egyszerűség kedvéért publikus.
    public int Age;
    public string FirstName, LastName;

    public Person(){}
    public Person(string FirstName, string LastName, int age)
    {
        Age = age;
        FirstName = FirstName;
        LastName = LastName;
    }

    public override string ToString()
    {
        return string.Format("Name: {0} {1}, Age: {2}",
            FirstName, LastName, Age);
    }
}
```

Annak érdekében, hogy létrehozzunk egy Person gyűjteményt, definiálnunk kell egy System.Collections.ArrayList tagváltót a PeopleCollection osztályon belül, és minden tagjának át kell alakítanunk a működését úgy, hogy az erősen típusos Person objektumot használja a System.Object típus helyett:

```
public class PeopleCollection : IEnumerable
{
    private ArrayList arPeople = new ArrayList();
    public PeopleCollection(){}

    // Kasztolás a hívóhoz.
    public Person GetPerson(int pos)
    { return (Person)arPeople[pos]; }

    // Csak Person típusok beszűrése.
    public void AddPerson(Person p)
    { arPeople.Add(p); }
```

```
public void ClearArPeople()
{ arPeople.Clear(); }

public int CountArPeople
{ get { return arPeople.Count; } }

// Foreach felsorolás támogatása.
IEnumerator IEnumerableArPeople.GetEnumerator()
{ return arPeople.GetEnumerator(); }
}
```

A `ICollection` típus megvalósítja az `IEnumerable` interfészt, hogy lehetővé tegye a `foreach`-szerű iterációt a tartalmazott elemeken. A `GetPerson()` és az `AddPerson()` módszereinket úgy tervezzük, hogy csak a `Person` objektumon működjön (nem bittérképeken, sztringeken, adatbázis-kapcsolatokon vagy más tételéken). Ezekkel a megvalósított típusokkal biztosítjuk a típusbiztonságot, a `C#`-fordító képes lesz meghatározni bármelyik nem kompatibilis típus beillesztésére tett kísérletet:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Custom Person Collection *****\n");
    ICollection myPeople = new PersonCollection();
    myPeople.AddPerson(new Person("Homer", "Simpson", 40));
    myPeople.AddPerson("Marge", "Simpson", 38);
    myPeople.AddPerson("Lisa", "Simpson", 9);
    myPeople.AddPerson("Bart", "Simpson", 7);
    myPeople.AddPerson(new Person("Maggie", "Simpson", 2));
    // Ez fordítási hiba lenne!
    // myPeople.AddPerson(new Car());
    foreach (Person p in myPeople)
    {
        Console.WriteLine(p);
        Console.ReadLine();
    }
}
```

Bár az egyedi gyűjtemények gondoskodnak a típusbiztonságról, ám ezért létre kell hoznunk egy (majdnem teljesen egyező) egyedi gyűjteményt minden egyes olyan típusnak, amelyet tárolni szeretnénk. Így ha szükségünk van egy olyan egyedi gyűjteményre, amely csak a `Car` alaposztályból származtatott osztályokon képes működni, akkor egy nagyon hasonló típust kell készítenünk:

```
public class CarCollection : IEnumerable
{
    private ArrayList arCars = new ArrayList();
    public CarCollection() {}
}
```

## Az erősen típusos gyűjtemények és a dobozolás

A .NET-alapoztálykönyvtárban léteznek erősen típusos gyűjtemények, amelyek nagyon hasznos programozási szerkezetek. Az egyedi tárolók kevéssé jelennek meg a dobozolás hátrányaira. Ha létrehozunk egy `ICollection`-t, az az egyedi gyűjteményt, amely csak a `System.Int32` adat típusokon működik, az adatok tárolására akkor is valamilyen általános tárolót kellene használnunk (`System.Array`, `System.Collections.ArrayList` stb.).

```
static void Main(string[] args)
{
    // Generikus List típus csak autót tárol.
    List<Car> moreCars = new List<Car>();
    // Generikus List típus csak személyeket tárol.
    List<Person> morePeople = new List<Person>();
    morePeople.Add(new Person());
    moreCars.Add(new Person());
    // Fordítási hiba!
}
```

Több, erősen típusos gyűjtemény létrehozása a különböző típusok kezelése miatt nemcsak munkaidőigényes, de fenntartása sem túl egyszerű. A generikus gyűjtemények lehetővé teszik számunkra a tárolt típus meghatározásának késleltetését a létrehozás idejéig. A szintaktikai részletek helyett egyelőre nézzük meg a következő forráskódot, amely a `System.Collections.Generic.List<T>` nevű generikus osztály segítségével létrehoz két típusbiztos tárolóobjektumot:

```
// Kasztolás a hívóhoz.
public Car GetCar(int pos)
{
    return (Car) arcars[pos];
}

// csak Car típusok beszűrése.
public void AddCar(Car c)
{
    arcars.Add(c);
}

public void ClearCars()
{
    arcars.Clear();
}

public int Count
{
    get { return arcars.Count; }
}

//oreach felsorolás támogatása.
IEnumerator IEnumerable.Getenumerator()
{
    return arcars.GetEnumerator();
}
```

```

public class IntCollection : IEnumerable
{
    private ArrayList aints = new ArrayList();
    public IntCollection() { }

    // kidobozolás a hívónak.
    public int GetInt(int pos)
    { return (int)aints[pos]; }

    // bedobozolás művelet
    public void AddInt(int i)
    { aints.Add(i); }

    public void Clear()
    { aints.Clear(); }

    public int Count
    { get { return aints.Count; } }

    IEnumerator IEnumerable.GetEnumerator()
    { return aints.GetEnumerator(); }
}

```

Függelennül attól, hogy milyen típust választottunk az egészek tárolására, (System.Array, System.Collections.ArrayList stb.), nem szabadulunk meg a dobozolás dilemmájától, ha nem generikus tárolt használunk. Ismét a generikus típusok segítenek ki minket. A következő forráskód a System.Collections.Generic.List<T> típust használja egy integer típusokat tartalmazó tároló létrehozásához, amely *inics* kitéve a ki- és a bedobozolás hátrányainak, amikor beillesztjük vagy lekérdezzük az értéket:

```

static void Main(string[] args)
{
    // nincs bedobozolás!
    List<int> myints = new List<int>();
    myints.Add(5);
    // nincs kidobozolás!
    int i = myints[0];
}

```

Az előző main() metódus a következő köztes nyelvi kódban (figyeljünk meg a box és az unbox műveleti kódok hiányát) a következőképpen néz ki:

```

.method private hidebysig static void Main(string[] args)
{
    cil managed
    .entrypoint
    .maxstack 2
}

```



```
        .locals init ([0] class
            [mscorlib]System.Collections.Generic.List`1<int32>
            myInts, [1] int32 i
        newobj instance void class
            [mscorlib]System.Collections.Generic.List`1<int32>::ctor()
            stloc.0
            ldc.i4.5
            callvirt instance void class [mscorlib]
                System.Collections.Generic.List`1<int32>::Add(int)
            nop
            ldc.i4.0
            stloc.0
            ldc.i4.0
            callvirt instance int class [mscorlib]
                System.Collections.Generic.List`1<int32>::get_Item(int32)
            stloc.1
        ret
    }
```

Összefoglalva, a generikus tárolók a következők előnyöket nyújtják a nem generikusokkal szemben:

- A generikus típusok jobb teljesítményt nyújtanak azzal, hogy nem eredményeznek ki- vagy bedobozolási hátrányokat.
- A generikus típusok típusbiztonsága jobb azáltal, hogy csak olyan a típusú változókat tárolnak, amelyeket specifikáltunk.
- A generikus típusok nagyban csökkenti az egyedi gyűjteménytípus készítésének szükségességét, mivel az alaposztály számos előre gyártott tárolót biztosít.

---

**Forráskód** A CustomGenericCollection projekt megtalálható a 10. fejezet alkönyvtárában.

---

## A System.Collections.Generic névtér

Szétosztottan minden .NET-alaposztálykönyvtárban található generikus típusokat; a System.Collections.Generic névtér azonban tele van ezekkel (ahogy a nevében is látható). A nem generikus ellenpárjukhoz hasonlóan (System.Collections), a System.Collections.Generic számos osztályt és interfész típust tartalmaz, amely lehetővé teszi, hogy az elemeket különböző tárolókban tároljuk. A generikus interfész utánnozza a megfelelő nem generikus típusokat a System.Collections névtérben:

**Megjegyzés** A generikus típusok egyezményesen specifikálják a helyőrzőjüket a közös név használatával. Bármely betű (vagy szó) megfelelő, de tipikusan a T-t használjuk a típus jelölésére, a kulcsokhoz TKey-t, az értékekhez pedig a TValue-t.

|                   |                                          |           |
|-------------------|------------------------------------------|-----------|
| Generikus osztály | Nem generikus pár a System.Collectionben | Jelentése |
|-------------------|------------------------------------------|-----------|

A System.Collections.Generic névtér ugyancsak számos osztályt definiál, amelyek ezekből a kulcsinterfészekből sokat megvalósítanak. A 10.5. táblázat rövid összefoglalást ad a névtér főbb osztálytípusairól, az általuk megvalósított interfészekről és a System.Collections névtér megkegyezző típusairól.

|                                |                |                                                                    |
|--------------------------------|----------------|--------------------------------------------------------------------|
| ICollection<T>                 | CollectionBase | A generikus gyűjtemény alapjai                                     |
| Comparer<T>                    | Comparer       | Két generikus objektum egyenlőségét vizsgálja                      |
| Dictionary<TKey, TValue>       | HashTable      | Kulcs/érték párok generikus gyűjteménye                            |
| List<T>                        | ArrayList      | Elemek dinamikusán átmerendezhető listája                          |
| Queue<T>                       | Queue          | „Előnek be, előnek ki” FIFO-lista generikus megvalósítása          |
| SortedDictionary<TKey, TValue> | SortedList     | Rendezett kulcs/érték párok gyűjteményének generikus megvalósítása |

- ICollection<T>
- IComparer<T>
- IDictionary<TKey, TValue>
- IEnumerable<T>
- IEnumerator<T>
- IList<T>

A nem generikus osztályokhoz hasonlóan a generikus osztályokat is new kulcszóval és az esetlegesen szükséges konstruktorparaméterek megadásával hozzuk létre. Továbbá specifikálnunk kell, hogy mely típus(ok) helyettesítsék a generikus típus által definiált típusparamétereket. A system.Collections.Generic namespace-ben található osztály példái megköveteli, hogy megadjunk egy olyan értéket, amely a List<T> által felhasznált elem típusát írja le. Így, ha három List<Object>-t akarunk létrehozni annak érdekében, hogy egészeket, valaminthé sportscar és Person objektumokat tároljunk, akkor a következőket írhatjuk:

## A List<T> típus vizsgálata

A 10.5. táblázatban látható: sok generikus gyűjteményosztálynak van nem csú elemet próbálunk megragadni a tárolóból, és így tovább.

A 10.5. táblázatban látható: sok generikus gyűjteményosztálynak van nem generikus párja a system.Collections.Generic névtérben (közülük soknak a neve), Mivel a generikus osztályok elegendő hasonlítani a nem generikus típusukhoz, ha megértjük, hogy egy generikus elem hogyan működik egy adott tárolóval, a többi már magától értetődő lesz. A generikus típusokkal dolgozó folyamatok működését a List<T> osztály használatával illusztráljuk. Ha a system.Collections.Generic névtér többi tagjáról szerezniünk további részleteket meg tudni, akkor a .NET Framework 3.5 dokumentációjában nézhetünk utána.

10.5. táblázat. A System.Collections.Generic névtér osztályai

| Generikus osztály     |       | Nem generikus pár a System.Collections.Generic névtérben |                                                               | Jelentése |
|-----------------------|-------|----------------------------------------------------------|---------------------------------------------------------------|-----------|
| Stack<T>              |       | stack                                                    | „Útolsónak be, elsőnek ki” LIFO-lista generikus megvalósítása |           |
| LinkedList<T>         | n. a. | Kettirányú láncolt lista generikus megvalósítása         |                                                               |           |
| ReadOnlyCollection<T> |       | ReadOnlyCollection                                       | Írásvédelem elemek csoportjának generikus megvalósítása       |           |

```

static void Main(string[] args)
{
    // List készítése egész számok tárolására.
    List<int> myInts = new List<int>();

    // List készítése Sportscar objektumok tárolására.
    List<Sportscar> myCars = new List<Sportscar>();

    // List készítése Person objektumok tárolására.
    List<Person> myPeople = new List<Person>();
}

```

Elgondolkodtató, hogy vajon mi lett a megadott helyőrző érték pontosan. Ha a Code Definition ablakot nézzük, akkor a `List<T>` esetében mindig a `T` helykitöltőt fogjuk látni egy konkrét típus helyett. Nézzünk meg egy részleges listát (figyeljük meg a felkövetérel írt elemeket):

```

// A List<T> típus részleges listája.
namespace System.Collections.Generic
{
    List<T>, ICollection<T>, IEnumerable<T>,
    IList, ICollection, IEnumerable
    {
        ...
        public void Add(T item);
        public IList<T> AsReadOnly();
        public int BinarySearch(T item);
        public bool Contains(T item);
        public void CopyTo(T[] array);
        public int FindIndex(System.Predicate<T> match);
        public T FindLast(System.Predicate<T> match);
        public bool Remove(T item);
        public int RemoveAll(System.Predicate<T> match);
        public T[] ToArray();
        public bool TrueForAll(System.Predicate<T> match);
        public T this[int index] { get; set; }
    }
}

```

Egy `Sportscar` típusú `List<T>` osztály létrehozása a következő `List<T>` osztály definíciójának felel meg:

```

namespace System.Collections.Generic
{
    public class List<Sportscar> :
        IEnumerable<Sportscar>,
        ICollection<Sportscar>,
        IList, ICollection, IEnumerable
    {

```

```

    public void Add(Sportscar item);
    public IList<Sportscar> AsReadonly();
    public int BinarySearch(Sportscar item);
    public bool Contains(Sportscar item);
    public void CopyTo(Sportscar[] array);
    public int FindIndex(System.Predicate<Sportscar> match);
    public Sportscar FindLast(System.Predicate<Sportscar> match);
    public bool Remove(Sportscar item);
    public int RemoveAll(System.Predicate<Sportscar> match);
    public Sportscar[] ToArray();
    public bool TrueForAll(System.Predicate<Sportscar> match);
    public Sportscar this[int index] { get; set; }
}

```

Természetesen generikus `IList<T>` osztály létrehozásakor a fordító nem hoz létre egy vadonaltű `IList<T>` típusú implementációt. Helyette csak az általunk aktuálisan felhasznált generikus típus tagjait címezi meg. Ehhez hozzunk létre egy `Sportscar` objektumokat tartalmazó `IList<T>` osztályt a következőképpen:

```

static void Main(string[] args)
{
    // Sportscar objektumokat tartalmazó listák inicializálása
    List<Sportscar> myCars = new List<Sportscar>();
    myCars.Add(new Sportscar());
    Console.WriteLine("Your list contains {0} item(s).",
        myCars.Count);
}

```

Ha az `itdasm.exe` segítségével megvizsgáljuk a köztes nyelvi kódot, megta-

lálhatjuk a következő helyettesítést:

```

.method private hidebysig static void Main(string[] args)
{
    .entrypoint
    .maxstack 2
    .locals init ([0] class
        <mscorlib>System.Collections.Generic.List`1
        <mscorlib>Sportscar myCars)
    <mscorlib>Sportscar myCars
    newobj instance void class
        [mscorlib>System.Collections.Generic.List`1
    <mscorlib>Sportscar::ctor()
    stloc.0
    ldloc.0
    newobj instance void [mscorlib>System.Collections.Generic.List`1
    newobj instance void [mscorlib>Sportscar::ctor()
}

```

## Saját generikus metódusok létrehozása

A továbbiakban azt vizsgáljuk meg, hogy hogyan hozhatunk létre saját generikus metódust, típust vagy gyűjteményt.

```
callvirt instance void class
[mscorlib]System.Collections.Generic.List`1<
    class SportCar>::Add(i0)
nop
ldstr "Your list contains {0} item(s)."
```

A legtöbb programozó jellemzően az alaposztálykönnyvtárakban már meglévő generikus típusokat használja, ám természetesen elkészíthetjük saját generikus tagjainkat és egyedi generikus típusunkat is. Ahhoz, hogy megismerjük a generikus típusok felhasználását saját projekteinkben, először készítsünk egy generikus swap metódust. Hozzunk létre egy genericMethod nevű új parancs-sori alkalmazást.

A célja ennek a példának az, hogy egy olyan swap metódust alkossunk, amely bármely lehetséges adattípuson (érték- vagy referenciátípus) működik egyetlen paramétertípus használatával. A cserélő algoritmus természetéből fakadóan a bejövő paraméterek átadása referenciaként történik (a C# ref kulcsszavával). A kezdeti Program osztályban létrehozott generikus swap me-tódus teljes megvalósítása a következő:

```
// A metódus megcseréli két elemet.
// a <T> típus paraméternek megfelelően.
static void Swap<T>(ref T a, ref T b)
{
    Console.WriteLine("You sent the swap() method a {0}",
        typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

Figyeljük meg, hogyan definiálunk generikus metódusokat a típusparaméter specifikálásával, a metódus neve után, de a paraméterlista előtt. A `swap()` metódus bármely két `<T>` típusú paraméterrel működik. Írjuk ki a tárolt helyőrző típusának a nevét a konzolra a `C# typeOf()` operátorának használatával. Figyeljük meg a következő `main()` metódust, amely egész- és sztringtípusokat cserél ki:

```
static void main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Generic
        Methods *****\n");
    // két egész szám cseréje.
    int a = 10, b = 90;
    Console.WriteLine("Before swap: {0}, {1}", a, b);
    Swap<int>(ref a, ref b);
    Console.WriteLine("After swap: {0}, {1}", a, b);
    Console.WriteLine();
    // két sztring cseréje.
    string s1 = "Hello", s2 = "There";
    Console.WriteLine("Before swap: {0} {1}", s1, s2);
    Swap<string>(ref s1, ref s2);
    Console.WriteLine("After swap: {0} {1}", s1, s2);
    Console.ReadLine();
}
```

## A típusparaméterek kiköveteltetése

Amikor generikus metódusokat használunk fel, például a `swap<T>`-t, akkor elhagyhatjuk a típusparamétert, de csak akkor, ha a generikus metódusnak paraméterekre van szükség, hiszen a fordító képes a típusparamétert kiköveteltetni a tagparaméterekből. Fel tudunk cserélni például két darab `system.Boolean` típust a következő forráskódban:

```
// A fordító a System.Boolean osztályra következtet.
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

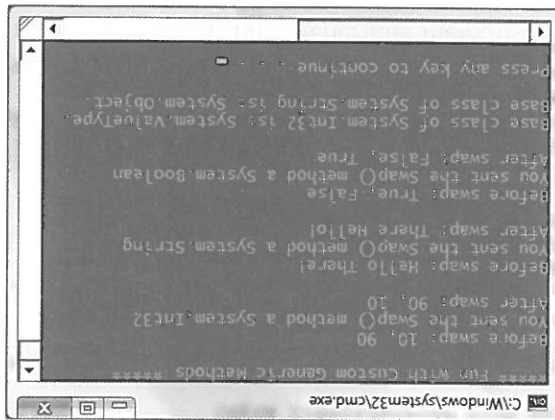
Ha azonban a következő `DisplayBaseClass<T>` nevű generikus metódussal rendelkezőnk, egyetlen bejövő paraméterre sincs:

```
static void DisplayBaseClass<T>()
{
    Console.WriteLine("Base class of {0} is: {1}.",
        typeOf(T), typeOf(T).BaseType);
}
```

akkor a meghívás során meg kell adnunk a típusparamétert:

```
static void Main(string[] args)
{
    // Meg kell adni a típus paramétert, ha
    // a módszernek nincs paramétere.
    DisplayBaseClass<int>();
    DisplayBaseClass<string>();
    // Fordítási hiba! Nincsenek paraméterek? Helyőrzőt kell megadni!
    // DisplayBaseClass();
    Console.ReadLine();
}
```

A 10.2. ábrán láthatjuk az alkalmazás kimenetét.



10.2. ábra. Generikus módszer működés közben

A generikus `Swap<T>` és `DisplayBaseClass<T>` metódusokat az alkalmazásobjektumon belül definiáljuk (azaz a `Main()` metódusban). Természetesen, mint minden metódussal, ezzel is megtehetjük, hogy ezeket a tagokat is különálló osztálytípusban definiáljuk:

```
public static class MyGenericMethods
{
    public static void Swap<T>(ref T a, ref T b)
    {
        Console.WriteLine("You sent the Swap() method a {0}",
            typeof(T));
        T temp;
        temp = a;
        a = b;
        b = temp;
    }
}
```



```
public static void DisplayBaseClass<T>()
{
    Console.WriteLine("Base class of {0} is: {1}.",
        typeof(T), typeof(T).BaseType);
}
```

Mivel a statikus swap<T> és displayBaseClass<T> metódusok hatóköre az új statikus osztálytípus, ezért, amikor felhasználjuk bármelyik tagjukat, meg kell adnunk a típus nevét. Például:

```
MyGenericMethods.Swap<int>(ref a, ref b);
```

A generikus metódusoknak nem kell statikusnak lenniük. Ha a swap<T> és a displayBaseClass<T> példányosított volt (és nem statikus osztályban definiáltak őket), akkor érdemes egyszerűen egy myGenericMethods példányt létrehozniuk és a metódusokat az objektumváltozón hívjuk meg:

```
MyGenericMethods c = new MyGenericMethods();
c.Swap<int>(ref a, ref b);
```

**Forráskód** A GenericMethod projekt megtalálható a 10. fejezet alkönyvtárában.

## Generikus struktúrák és osztályok létrehozása

A továbbiakban a generikus struktúra létrehozására koncentrálunk a GenericPoint nevű új parancssorialkalmazás-projektben belül (a generikus osztályok készítésének a folyamata hasonló). Tegyük fel, hogy elkészítettünk egy generikus Point struktúrát, amely egyetlen paramétertípust támogat, reprezentálva a koordináták (x,y) alapvető tárolását. A hívó képes Point<T> típust létrehozni a következőképpen:

```
// Egész számokkal működő Point.
Point<int> p = new Point<int>(10, 10);

// Double számokkal működő Point.
Point<double> p2 = new Point<double>(5.4, 3.3);
```

A `Point<T>` teljes definíciója a következő:

```
// Egy generikus Point struktúra
public struct Point<T>
{
    // Generikus állapot dátum.
    private T xPos;
    private T yPos;

    // Generikus konstruktor.
    public Point(T xVal, T yVal)
    {
        xPos = xVal;
        yPos = yVal;
    }

    // Generikus tulajdonságok.
    public T x
    {
        get { return xPos; }
        set { xPos = value; }
    }

    public T y
    {
        get { return yPos; }
        set { yPos = value; }
    }

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

    // Mezők visszaállítására a típusparaméter
    // alapértelmezett értékére.
    public void ResetPoint()
    {
        xPos = default(T);
        yPos = default(T);
    }
}
```

## A default kulcsszó a generikus kódban

A `Point<T>` osztály használja a típusparaméterét az adatmező definíálásánál, a konstruktornál és a tulajdonság meghatározásánál. Ezen kívül felüldefiníálja a `ToString()`-et, a `Point<T>` pedig egy `ResetPoint()` nevű metódust definiál, amely néhány új szintaktikai megoldást használ:

```
// A "default" kulcsszó #-ban túlterhelt.
// generikus kódban a típus paraméter
// alapértelmezett értéket képviseli.
public void ResetPoint()
{
    xpos = default(T);
    ypos = default(T);
}
```

A generikus típusok bevezetésével a C# default kulcsszavának kettős szerepe lett. A switch szerkezeten belüli használat mellett a típusparaméterek alapértelmezett értékeknek beállítására is felhasználhatjuk. Ez nagy segítség akkor, amikor egy generikus típus nem ismeri az aktuális helyzeteket rögtön a kezdetkor, és így nem tudja biztonsággal felitélelni az alapértelmezett értéket. A típusparaméter alapértelmezett értékei a következők lehetnek:

- a numerikus értékeknek az alapértelmezett értéke 0;
- a referenciátípusoknak az alapértelmezett értéke null érték;
- a struktúrák mezőinek értéke 0 (értéktípusnál) vagy null érték (referenciátípusnál).

A `Point<T>` osztállyal egyszerűen beállíthatunk `xpos`-t és `ypos`-t közvetlenül 0-ra, ugyanis biztosak lehetünk abban, hogy a hívó csak numerikus adatokkal dolgozik. A `default` (T) szintaxis használatával azonban megkönnyítjük a generikus típusunk rugalmasságát. Vizsgáljuk meg a következőképpen a `Point<T>` osztály metódusait:

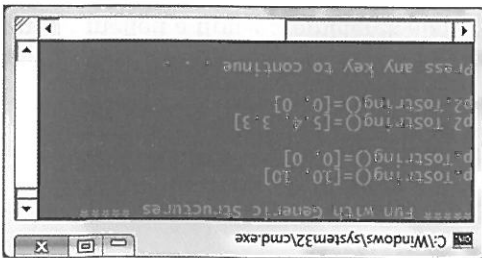
```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with generic structures *****\n");

    // Egész számokkal működő Point.
    Point<int> p = new Point<int>(10, 10);
    Console.WriteLine("p.ToString()={0}", p.ToString());
    p.ResetPoint();
    Console.WriteLine("p.ToString()={0}", p.ToString());

    // Double számokkal működő Point.
    Point<double> p2 = new Point<double>(5.4, 3.3);
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
    p2.ResetPoint();
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
}

Console.ReadLine();
```

A 10.3. ábrán láthatjuk a kimenetet.



10.3. ábra. Generikus Point típus használata

---

**Forráskód** The genericPoint projekt megtalálható a 10. fejezet alkönyvtárában.

---

## Saját generikus gyűjtemény készítése

A `System.Collections.Generic` névtér számos olyan típust biztosít, amelyek lehetővé teszik típusbiztos és hatékony tárolók létrehozását. A leghatékosabb lasztások köréből adódóan nagy eséllyel nem kell egyedi gyűjteménytípust létrehozunk, amikor .NET-platfornmal dolgozunk. Az erősen típusos generikus tárolót készítésének bemutatásához a következő feladat a `CarCollection<T>` névű generikus gyűjteményosztály létrehozása lesz (ki fog derülni közben, hogy milyen előnye van annak). Kezdjük azzal, hogy létrehozunk egy `CustomGenericCollection` névű új parancssori alkalmazást.

Mint a korábban már létrehozott `GenericCarCollection`, ez az iteráció is felhasználja a meglévő gyűjteménytípusokat az elemek tárolására (ebben az esetben egy `List<T>` típust). A `GenericCarCollection<T>` interfész megvalósításával a `foreach` iterációt támogatjuk. Az `IEnumerable<T>` a nem generikus `IEnumerable` interfészt bővíti, így azonban a fordító a `GetEnumerator()` metódus két változatát várja. A módosítás a következő:

```
public class CarCollection<T> : IEnumerable<T>
{
    private List<T> arcars = new List<T>();

    public T GetCar(int pos)
    { return arcars[pos]; }

    public void AddCar(T c)
    { arcars.Add(c); }
```

```

public void ClearCars()
{
    arcars.Clear();
}

public int Count
{
    get { return arcars.Count; }
}

// Az IEnumerabLe<T> bővíti az IEnumerabLe interfészt, ezért
// a GetEnumerator() mIndkét verzióját meg kell valósítani.
IEnumerator<T> IEnumerabLe<T>.GetEnumerator()
{
    return arcars.GetEnumerator();
}

IEnumerator IEnumerabLe.GetEnumerator()
{
    return arcars.GetEnumerator();
}

```

A módosított `carCollection`-t a következőképpen használhatjuk fel (egy egyszerű `car` definíciót feltevélezve az aktuális projektkben):

```

static void Main(string[] args)
{
    Console.WriteLine("***** Custom Generic Collection *****\n");

    // car objektumok készítésének létrehozása.
    carCollection<Car> myCars = new carCollection<Car>();
    myCars.AddCar(new Car("Rusty", 20));
    myCars.AddCar(new Car("Zippy", 90));

    foreach (Car c in myCars)
    {
        Console.WriteLine("PetName: {0}, Speed: {1}",
            c.PetName, c.Speed);
    }
    Console.ReadLine();
}

```

Létrehoztunk egy `car` típust tartalmazó `carCollection` típust. Hasonló végeredményt kell elérnünk, ha a `List` típust közvetlenül használjuk. Nagy előnye ennek, hogy szabadon adhatunk egyedi módszereket (`AddCar()`, `GetCar()` stb.) a `carCollection` típushoz, amely a hívásokat továbbítja a belső `List` típus felé.

## A saját generikus gyűjtemény korlátai

A `carCollection` osztály azonban nem sokat ad az egyedileg elnevezett publikus módszerek használatán kívül. Ráadásul az objektumfelhasználó létre tud hozni egy `carCollection` példányt egy teljesen nem ideillő paramétertípussal:

```
// Szintaktikailag helyes, de roppant bonyolult!
CarCollection<int> myInts = new CarCollection<int>();
myInts.AddCar(5);
myInts.AddCar(11);

foreach (int i in myInts)
{
    Console.WriteLine("int value: {0}", i);
}
```

Miért enged a fordító akkor ilyen kódolást? A generikususok nevének megfelelően *generikusak*, vagyis általában. Egy típusparaméter pedig minden lehet, még ha a generikus típuson belül teljesen értelmetlen is (pl. egy egészeket tároló autógyűjtemény).

Hogy szemléltessünk egy másik generikus felhasználást, tegyük fel, hogy létrehozunk két, a car típusból származtatott új osztályt (Sportscar és minivan):

```
public class Sportscar : Car
{
    public Sportscar(string p, int s)
    : base(p, s) {}
    // további Sportscar metódusok következnek.
}

public class minivan : Car
{
    public minivan(string p, int s)
    : base(p, s) {}
    // további minivan metódusok következnek.
}
```

A származtatás törvénye megengedi a minivan vagy a sportscar típus közötti hozzáférést a car típusú paraméterrel létrehozott CarCollection<T> osztályhoz:

```
// A CarCollection<Car> a car típusból lezármazott
// típusokat tárol.
CarCollection<Car> myAutos = new CarCollection<Car>();
myAutos.AddCar(new Minivan("Family Truckster", 55));
myAutos.AddCar(new Sportscar("Crusher", 40));
```

Bar ez szintaktikailag érvényes kód, de mi történik akkor, ha a CarCollection<T> osztályt egy új publikus PrintPetName() metódussal szeretnénk frissíteni? Ehhez csak hozzá kell fűznünk a megfelelő list<T> elemhez és kivenni belőle a név értékét:

```
// Hibai A System.Object nem rendelkezik
// PetName nevű tulajdonsággal.
public void PrintPetName(int pos)
{
    Console.WriteLine(arcars[pos].PetName);
}
```

Ez azonban nem fog lefordulni, ugyanis a `T` igazi azonossága nem ismert, és nem tudjuk biztosan, hogy a `List<T>` típusban tárolt elemnek van-e `PetName` tulajdonsága. Ha a típusparаметernek nincsenek megköötései (mint a fenti osztály esetében), akkor a generikus típust *kötetlennek* nevezzük. Feltételezhetjük, hogy a kötetlen típusparаметerek csak a `System.Object` tagjaival rendelkeznek (amelynek egyértelműen nincs `PetName` tulajdonsága). Megpróbálhatjuk azt is, hogy a `List<T>` indexelőből visszatérő elemet egy erősen típusos `car` elemként tüntetjük fel, és felhasználjuk a visszatérő objektumból a nevet:

```
// Hibai
// A "T" nem konvertálható "Car" típusra.
public void PrintPetName(int pos)
{
    Console.WriteLine(((Car)arcars[pos]).PetName);
}
```

Ez azonban megint csak nem fog lefordulni, hiszen a fordító nem ismeri még a `<T>` típusparаметert, és így nem tudja garantálni, hogy a kasztolás érvényes legyen. Ebből következően jogosan kérdezhetjük: mikor (ha egyáltalán vala-mikor) kellene ilyen egyedi generikus tárolót létrehoznunk?

## Tipusparаметer korlátozása a *where* kulcsszó használatával

A fő ok, amiért a fejlesztőknek egyedi generikus gyűjteménytípust kellene használniuk, az az, hogy kikényszerítsék a típusparаметerekben a *megszorítást* sokat azért, hogy rendkívül típusbiztos tárolót készítsenek. A `C#` nyelvben a megszorításokat a `where` kulcsszó használatával hozzuk létre, ez szabályozza a típusparаметerek tulajdonságait (10.6. táblázat).

Generikus kényszer

Használata

|                              |                                                                                                                                                                                                                                                                                                                                             |
|------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| where T : struct             | A <T> típusparaméter származási láncában System.<br>ValueType-nak kell szerepelnie.                                                                                                                                                                                                                                                         |
| where T : class              | A <T> típusparaméter származási láncában nem szabad<br>System.ValueType-nak lennie (pl. <T> referencia típus).                                                                                                                                                                                                                              |
| where T : new()              | A <T> típusparaméternek rendelkeznie kell egy alapér-<br>telmezett konstruktorral. Ez akkor segít, ha a generikus<br>típusnak létre kell hoznia egy példányt a típusparame-<br>terből, hiszen nem tudjuk az egyedi konstruktor forma-<br>tumát. Ezt a megszorítást a legutolsó helyen kell szere-<br>peltetni több megszorítás megadásakor. |
| where T :<br>NameOfBaseClass | A <T> típusparaméternek a NameOfBaseClass által spe-<br>cifikált osztályból kell származnia.                                                                                                                                                                                                                                                |
| where T :<br>NameOfInterface | A <T> típusparaméternek NameOfInterface által speci-<br>fikált interfész kell megvalósítania. Többszörös interfe-<br>szeket megadhatunk vesszővel elválasztott listában.                                                                                                                                                                    |

10.6. Táblázat. Generikus típusparaméter lehetséges kényszerei

Amikor megszorításokat használunk a where kulcsszó segítségével, a kény-  
szerlista a generikus típus összesítőja és az interfészek listája után helyezke-  
dik el. Konkrét példákkal nézzük meg a következő MyGenericClass nevű ge-  
nerikus osztály megköltését:

```
// A MyGenericClass az objekt tesztárazottja, az
// elemi alapértelmezett konstruktorral rendelkezik.
public class MyGenericClass<T> where T : new()
{...}

// A MyGenericClass az objekt tesztárazottja, az
// elemi támogatják az alapértelmezett konstruktor, és
// egyben osztályok, amelyek megvalósítják az IDrivable interfészt.
public class MyGenericClass<T> where T : class, IDrivable, new()
{...}

// A MyGenericClass a MyBase tesztárazottja, és megvalósítja
// a ISomeInterface, az elemi pedig struktúrák lehetnek.
public class MyGenericClass<T> : MyBase, ISomeInterface where T :
    struct
{...}
```

Ha többszörös típusparamétert specifikáló generikus típust készítünk, akkor  
a kényszerek egyedi halmazát mind egyikre specifikálhatjuk:



```
// A <K> alapértelmezett konstruktorral rendelkezik, <T> pedig
// megvalósítja a generikus IComparable interfészt.
public class MyGenericClass<K, T> where K : new()
where T : IComparable<T>
{...
```

Há a `CarCollection<T>` osztályt szeretnénk módosítani úgy, hogy csak a `Car` osztályból származó típusokat lehessen tárolni benne, akkor a következőt kell írunk:

```
public class CarCollection<T> : IEnumerable<T> where T : Car
{
    ...
    public void PrintPetName(int pos)
    {
        // mivel minden aJelen a Car családba tartozik,
        // a PetName tulajdonság közvetlenül hívható.
        Console.WriteLine(arcars[pos].PetName);
    }
}
```

Há egyszer korlátozzuk a `CarCollection<T>` osztályt úgy, hogy csak a `Car` osztályból származó típusokat tartalmazhat, akkor a `PrintPetName()` megvalósítása egyszerű lesz, ugyanis a fordító ekkor már felteszi, hogy `<T>` típus a `Car` típusból származik. Továbbá, ha a megadott típusparaméter nem `Car` típus kompatibilis, akkor fordítási hiba keletkezik:

```
// Fordítási hiba!
CarCollection<int> myInts = new CarCollection<int>();
```

A generikus módszer is használhatja a `where` kulcsszót. Ha például azt szeretnénk, hogy az előzőleg létrehozott `Swap()` módszerbe csak a `System.ValueType`-ből származó típusok kerüljenek, akkor módosítsuk a kódunkat a következőképpen:

```
// A módszer érték típusokat cserél, de osztályokat nem.
static void Swap<T>(ref T a, ref T b) where T : struct
{
    ...
}
```

Há így korlátozzuk a `Swap()` módszert, akkor nem tudunk sztring típusokat cserélni (ahogy a mintakódban csináltuk), hiszen azok referenciátípusok.

## Az operátorakorlátozások hiánya

Amikor generikus metódusokat hozunk létre, bármely C#-operátor (+, -, \*, == stb.) alkalmazása a típusparamétereken *fordítási hibát* eredményez. Példaként nézzünk egy olyan osztályt, amely Add(), Subtract(), Multiply() és Divide() metódusokat hajt végre generikus típusokon:

```
// Fordítási hiba! Operátorokat nem
// alkalmazhatunk típusparamétereken!
public class BasicMath<T>
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }
    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }
    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }
    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

Az előbbi BasicMath<T> osztály nem fog lefordulni. Ez komoly korlátozásnak tűnik, ám a generikus típusok *generikusnak*, vagyis általánosak. Természetesen a System.Int32 típus jól működik a C# kétooperandusú operátoraival. Ha azonban a <T> egy egyedi osztály- vagy struktúrátípus lenne, akkor a fordító nem feltehetően lehet, hogy a típus túlterhel a +, -, \* és / operátorokat. Elméletileg a C# nyelvnek engedni kellene, hogy támogatott operátorokkal korlátozzunk egy generikus típust. Például:

```
// Csak szemléltető kód!
// C# 2008 nyelvben ez a kód érvénytelen.
public class BasicMath<T> where T : operator +, operator -,
    operator *, operator /
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }
    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }
    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }
    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

Sajnos a C# aktuális változata nem támogatja az operátormegszorításokat. Ha generikus interfészeket használunk fel, akkor szimulálnunk kell az operátor használatának jellegét a típusparаметerekben.

**Forráskód** A CustomGenericCollection projekt megtalálható a 10. fejezet alkönyvtárában.

## Generikus osztály létrehozása

A generikus osztály lehet más osztályok osztálya, és ennek következtében bármennyi virtuális vagy absztrakt módszert definiálhat. A származtatott típusokra azonban vonatkozik néhány szabály. Először is, ha egy nem generikus osztályt származtatunk le egy generikus osztályból, akkor a származtatott osztálynak a konkrét típusparаметert meg kell határoznia:

```
// Tételjezzük fel, hogy létrehoztunk egy
// egyedi generikus listaosztályt.
public class MyList<T>
{
    private List<T> listofdata = new List<T>();
}

// A konkrét típusoknak meg kell határoznia a típus
// parаметert, ha generikus osztály
// leszármozttat.
public class MyStringList : MyList<string>
{ }
```

Továbbá, ha a generikus osztály generikus virtuális vagy absztrakt módszereket definiál, akkor a származtatott típusnak felül kell definiálnia a generikus módszereket a konkrét típusparаметerek felhasználásával:

```
// Generikus osztály virtuális módszerral.
public class MyList<T>
{
    private List<T> listofdata = new List<T>();
    public virtual void PrintList(T data) { }
}

public class MyStringList : MyList<string>
{
    // A szülőosztályban alkalmazott típusparаметert kell
    // helyettesíteni a leszármozott módszerben.
    public override void PrintList(string data) { }
}
```

Ha a származtatott típus szintén generikus, akkor a gyermekkiosztály újra fel tudja használni a típushelyőrzőt a definíciójában. Az összesítőba helyezett megszorításokat azonban a származtatott típusnál is meg kell adni. Például:

```
// két megszorítás vonatkozik az alapertelemzett konstruktorokra.
public class MyList<T> where T : new()
{
    private List<T> listofdata = new List<T>();
    public virtual void printlist(T data) { }
}

// A leszármazott típusok tiszteletben tartják a megszorításokat.
public class MyReadonlyList<T> : MyList<T> where T : new()
{
    public override void printlist(T data) { }
}
```

A napi programozási feladatok között nem gyakran fog szerepelni az egyedi generikusosztály-hierarchiák létrehozása.

## Generikus interfész létrehozása

A `System.Collections.Generic` névtér vizsgálata során már láttuk, hogy generikus interfészek (pl. `IEnumerator<T>`) létrehozása is lehetséges. Saját generikus interfésztünk teremtésén szabadon definiálhatjuk (megkövetésekkel vagy azok nélkül). Tétélezünk fel, hogy generikus típusarամերերen kétoperandusú művelet végrehajtására képes interfészt szeretnénk definiálni:

```
public interface IBinaryOperations<T> where T : struct
{
    T Add(T arg1, T arg2);
    T Subtract(T arg1, T arg2);
    T Multiply(T arg1, T arg2);
    T Divide(T arg1, T arg2);
}
```

Az interfészek többé-kevésbé használatlanok, amíg egy osztály vagy egy struktúra meg nem valósítja őket. Amikor egy generikus interfészt valósítunk meg, a támogató típus meghatározza a helyőrző típusát:

```
public class BasicMath : IBinaryOperations<int>
{
    public int Add(int arg1, int arg2)
    {
        return arg1 + arg2;
    }
}
```

```
public int Subtract(int arg1, int arg2)
{ return arg1 - arg2; }

public int Multiply(int arg1, int arg2)
{ return arg1 * arg2; }

public int Divide(int arg1, int arg2)
{ return arg1 / arg2; }
}
```

Ekkor az elvárt módon használhatjuk fel a BasicMath-t:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Generic Interfaces *****\n");
    BasicMath m = new BasicMath();
    Console.WriteLine("1 + 1 = {0}", m.Add(1, 1));
    Console.ReadLine();
}
```

Ha inkább egy olyan BasicMath osztályt hoznánk létre, amely működik a lebegő pontos számokon, akkor a típusparamétert a következőképpen specifikáljuk:

```
public class BasicMath : IBinaryOperations<float>
{
    public float Add(float arg1, float arg2)
    { return arg1 + arg2; }
    ...
}
```

Ebben az esetben a fordító biztosítja, hogy átvesszük a lebegő pontos számot a BasicMath osztály minden metódusában. A kódban szereplő lebegőpontos konstansok alapértelmezett típusa a double. Ha floatként szeretnénk kezelni őket, akkor ezt a literal végére írt f betűvel tehetjük:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Generic Interfaces *****\n");
    BasicMath m = new BasicMath();
    Console.WriteLine("1.98 + 1.3 = {0}", m.Add(1.98f, 1.3f));
    Console.ReadLine();
}
```

---

**Forráskód** A GenericInterface projekt megtalálható a 10. fejezet alkönyvtárában.

---

A következő fejezetben a .NET-metódusreferenciákat vizsgálva vissza fogunk még térni a generikus típusokra.

## Összefoglalás

10. fejezet: Gyűjtemények és generikus típusok

A fejezet a `system.lections` névtérben található „klasszikus” tárolók vizsgálatával kezdődött. Ezek a típusok továbbra is támogatottak maradnak a visszafelé kompatibilitás miatt, ám a `system.lections.genetic` névtér generikus pártjai az új .NET-alkalmazások számára is hasznosak. A generikus elemek lehetővé teszik „helyőrzők” definiálását (pl. típusparaméterek), amelyek a létrehozás (vagy a generikus metódusok esetén a meghívás) idején határozzunk meg konkrétan. A generikus típusok alapvetően megoldást nyújtanak a dobozoltásra és a típusbiztos használatra (ezek nehezeztették meg a .NET 1.1 alatti szoftver fejlesztését). Továbbá generikus típusok nagyjából el-tűntették az egyedi gyűjteménytípusok készítésének szükségességét.

Míg leggyakrabban .NET-alapozottalkönyvtárban megvalósított generikus típusokat fogjuk használni, emellett képesek vagyunk saját generikus típust is létrehozni. Ekkor figyelembe vehetünk többféle kényszert (a `where` kulcsszó használatával), hogy növeljük a típusbiztonság szintjét, és gondoskodjunk arról, hogy a típusok „ismert mennyiségűn” hajtsunk végre műveleteket.

# Metódusreferenciák, események és lambdák

Eddig a legtöbb általunk fejlesztett alkalmazás különböző olyan kódreszeket adott hozzá a `main()` metódushoz, amelyek valamilyen módon objektumoknak küldtek igényeket. A 9. fejezetben megvizsgáltuk, hogy az interfész típusok hogyan használhatók olyan objektumok létrehozására, amelyek „visszatudnak szólni” annak az elemnek, amely létrehozta őket. Bár visszahívható (callback) interfészek használhatók kettirányú kommunikáció megvalósítására, a .NET platformon a metódusreferencia a javasolt módszer a visszahívhatóság megvalósítására.

Alapvető meghatározása szerint a .NET-metódusreferencia olyan típusbiztos objektum, amely később meghívható metódusra vagy metóduslistára mutat. A tradicionális C++-függvénymutatótól eltérően a .NET-metódusreferenciák olyan osztályok, amelyek beépített támogatással rendelkeznek a többes küldéshez (multicasting) és az aszinkron függvényhíváshoz.

Ha már megismerkedtünk a metódusreferencia-típusok létrehozásával, valamint a használataikkal, megnézzük a C# event kulcsszót, amely a metódusreferenciák alkalmazását egyszerűsíti. Ekkorben a C# többféle metódusreferenciával és eseményközpontú nyelvi elemével is találkozunk, többek között a névtelen metódussal és a metóduscsoport (Method Group) átalakítással.

A fejezet végén a C# 2008 egyik új nyelvi elemével, a *lambda kifejezéssel* ismerkedünk meg. Az új lambda operátorral ( $\Rightarrow$ ) az utasítások egy olyan kódblokkját lehet már kialakítani (és ehhez a kódblokkhoz paramétereket is lehet megadni), ahol erősen típusos metódusreferenciára van szükség. A lambda kifejezés nem sokkal több, mint álcázott névtelen metódus.

## A.NET-metódusreferencia

Mielőtt formálisan definiálnánk a .NET-metódusreferenciákat, jöjjön egy kis történeti áttekintés. A Windows API gyakran használta a C típusú függvénymutatókat olyan entitások létrehozására, amelyeket *callback függvényeknek* vagy

egyszerűen csak *visszahívásnak* nevezték. A visszahívások használatával a programozóknak lehetősége nyílt úgy kialakítani egy függvényt, hogy az az alkalmazásban egy másik függvényt értesítsen (visszahívjon). Ezt a megközelítést használva a Win32-fejlesztők kezelni tudták a káthintást, az érgérmőzgatást, a menükiválasztást és az általános kétfirányú kommunikációt két entitás között. Az általános C típusú visszahívható függvényekkel az a probléma, hogy nem jelennek sokkal többet egy natív memóriacímnel. Ideális esetben a callback függvények olyan típusbiztos információkat is tartalmaznak, mint a paraméterek száma (és típusa), valamint a mutatott metódus visszatérési értéke (ha van). A tradicionális visszahívható függvényeknek sajnos nem ez a helyzet, emiatt gyakori hibaforrás, összeomlások és egyéb futásidejű katasztrofák okozója lehet. Ettől függetlenül a visszahívások hasznos megoldások.

A .NET keretrendszerben szintén van lehetőség visszahívások létrehozására, és a működésük a *metódusreferenciák* használatával sokkal biztonságosabb, objektumorientáltabbá vált. Lényegében a metódusreferencia olyan típusbiztos objektum, amely egy másik olyan metódusra (vagy akár metódusok listájára) mutat az alkalmazásban, amelyet később meghívhatunk. Pontosabban fogalmazva, a metódusreferencia-objektum három fontos információt kezel:

- annak a metódusnak a *címét*, amelyen hívásokat hajt végre;
- a metódus *argumentumait* (ha vannak);
- a metódus *visszatérési értékét* (ha van).

---

**Megjegyzés** A C(++)-függvénymutatóktól eltérően a .NET-metódusreferencia mutathat statikus vagy példánymetódusra is.

---

Ha a metódusreferencia elkészült, és megkapta a szükséges információkat, akkor az futásidőben dinamikusan meg tudja hívni azokat a metódusokat, amelyekre mutat. A .NET keretrendszer minden metódusreferenciája (beleértve az egyéni metódusreferenciákat is) automatikusan rendelkezik azzal a képességgel, hogy a metódusait *szinkron* vagy *aszinkron* módon hívja. Mindez nagyon megkönnyíti a programozási feladatokat, hiszen meghívhatunk egy metódust egy második szalon anélkül, hogy manuálisan létre kellene hoznunk és kezelnünk kellene egy *Thread* objektumot.

---

**Megjegyzés** A metódusreferencia-típusok aszinkron viselkedését lásd a 18. fejezetben a *System.Threading* névtér áttekintésénél.

---



## A metódusreferencia definiálása C#-ban

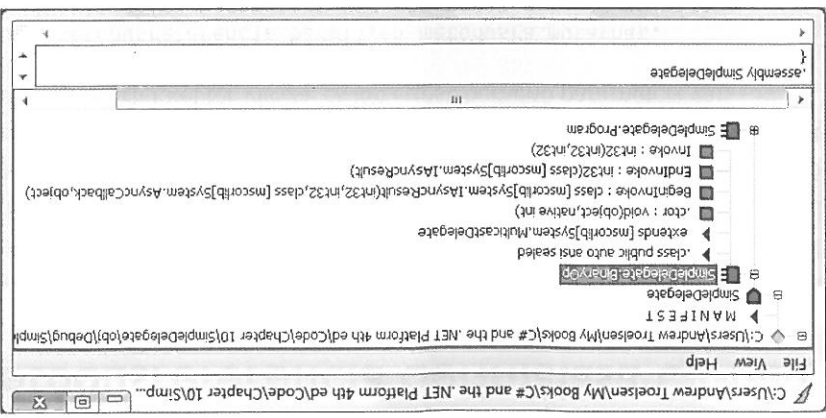
Ha a C# nyelvben létre akarunk hozni egy metódusreferenciát, akkor a *delegate* kulcsszót kell használnunk. A metódusreferencia neve tetszőlegesen lehet. Úgy kell azonban definiálnunk, hogy egyezzen annak a metódusnak a szignatúrájával, amelyre mutatni fog. Példaképpen tételezzük fel, hogy a *BinaryOp* nevű metódusreferenciát szeretnénk létrehozni, ez pedig bármely olyan metódusra mutat, amely egész értékekkel tér vissza, és két egész bemenő paramétert vár:

```
// Ez a metódusreferencia bármilyen metódusra mutathat,
// amely két egész paramétert vár, és egész értéket ad vissza.
public delegate int BinaryOp(int x, int y);
```

Mikor a C#-fordító metódusreferencia-típust dolgoz fel, automatikusan létrehoz egy lezárt osztályt a *System.MulticastDelegate* osztályból. Ez az osztály (az összestályához, a *System.Delegate*-hez kapcsolódva) a szükséges infrastruktúrát nyújtja a metódusreferenciának a később hívható metódusok listájának tárolásához. Ha például megnézzük a *BinaryOp* metódusreferenciát az *11.dasm.exe* programmal, akkor a 11.1. ábrán mutatott osztályt láthatjuk.

A fordító generálta *BinaryOp* osztály három nyilvános metódust definiál. Vegyük az *Invoke()* metódust alapul, hiszen ezt használjuk minden olyan metódus meghívására, amelyet a metódusreferencia *szinkron* módon kezel, ez pedig annyit tesz, hogy a hívónak meg kell várnia a hívás befejeződését, mielőtt tovább folytathatna. A *szinkron Invoke()* metódust nem kell explicit módon hívni a C#-kódból. A megfelelő C#-szintaxissal az *Invoke()* metódus hívása a háttérből történik.

A *BeginInvoke()* és az *EndInvoke()* metódusok szolgálnak az adott függvény *aszinkron* hívására egy külön végrehajtási szalon. Ismerve némileg a többszálás programozást, tudjuk, hogy a legaltalanosabb ok egy második szál létrehozására az, hogy olyan metódust szeretnénk meghívni, amelynek lefutása hosszú ideig tart. Habár a .NET-alapozástálkönyvtár egy egész névteret nyújt a többszálú programozás lehetővé tételéhez (*System.Threading*), a metódusreferencia ezt a funkcionálitást azonnal biztosítja.



11.1. ábra: A C# delegate kulcsszava egy lezárt osztályt jelöl, amely a System.MulticastDelegate osztályból származik

Pontosan hogyan tudja a fordító, hogy hogyan definiálja az Invoke(), a BeginInvoke() és az EndInvoke() metódusokat? A folyamat megértéséhez nézzük meg a fordító generálta BinarYop osztálytípus lenyegét (a felkövetően szedett elemeket a metódusreferencia-típus adta meg):

```
sealed class BinarYop : System.MulticastDelegate
{
    public BinarYop(object target, uint functionAddress);
    public int Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y, AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

Először is az Invoke() metódusban definiált paraméterek és a visszatérési érték pontosan egyezik a BinarYop metódusreferencia definíciójával. A BeginInvoke() tagok kezdeti paraméterei (esetünkben két egész) szintén a BinarYop metódusreferencián alapszanak; viszont a BeginInvoke() metódus mindig két végső paramétert szolgáltat (egy AsyncCallback és egy object típusút), s ezek megkönyítik az aszinkron függvényhívást. Végül az EndInvoke() visszatérési értéke azonos az eredeti metódusreferencia-deklarációjával, és mindig egyetlen paramétert kap, egy objektumot, amely megvalósítja az IAsyncResult interfészt. Nézzünk egy másik példát. Tételizzzuk fel, hogy definiáltunk egy metódusreferenciát, s ez bármely olyan metódusra mutathat, amely sztringet ad vissza, és három System.Boolean bemeneti paramétert vár:

```
public delegate string MyDelegate(bool a, bool b, bool c);
```

Ebben az esetben a fordító által generált osztály a következő lesz:

```
sealed class MyDelegate : System.MulticastDelegate
{
    public MyDelegate(object target, uint functionAddress);
    public string Invoke(bool a, bool b, bool c);
    public AsyncResult BeginInvoke(bool a, bool b, bool c,
        AsyncCallback cb, object state);
    public string EndInvoke(AsyncResult result);
}
```

A metódusreferencia „mutathat” olyan metódusokra is, amelyek out vagy ref paramétereket tartalmaznak (valamint a params kulcsszóval megjelölt tömbparamétereket is). Példaként nézzük a következő metódusreferencia-típust:

```
public delegate string MyOtherDelegate(out bool a, ref bool b,
    int c);
```

Az Invoke() és a BeginInvoke() metódusok szignatúrája úgy néz ki, ahogyan várjuk, nézzük meg azonban az EndInvoke() metódust, amely tartalmazza az összes, a metódusreferencia által definiált out/ref argumentum halmazát:

```
sealed class MyOtherDelegate : System.MulticastDelegate
{
    public MyOtherDelegate(object target, uint functionAddress);
    public string Invoke(out bool a, ref bool b, int c);
    public AsyncResult BeginInvoke(out bool a, ref bool b, int c,
        AsyncCallback cb, object state);
    public string EndInvoke(out bool a, ref bool b,
        AsyncResult result);
}
```

Összefoglalva: a C#-metódusreferencia definíciója lezárt osztályt eredményez határon, a fordító által generált olyan metódussal, amelyek paraméterei és visszatérési típusai a metódusreferencia deklarációján alapulnak. A következő pszeudokód bemutatja a hozzávetőlegesen egyeztetett mintát:

```
// Ez csak pszeudokód!
public sealed class DelegateName : System.MulticastDelegate
{
    public DelegateName(object target, uint functionAddress);
    public DelegateName(object target, uint functionAddress,
        InvokeDelegateInputRefAndOutputParams);
    public AsyncResult BeginInvoke(DelegateInputRefAndOutputParams,
        AsyncCallback cb, object state);
    public DelegateName EndInvoke(DelegateInputRefAndOutputParams,
        AsyncResult result);
}
```

# A System.MulticastDelegate és System.Delegate alapoztályok

11. fejezet: Metódusreferenciák, események és lambdák

Ha a C# delegate kulcsszavával készítenek típusokat, akkor indirekt módon olyan osztályt hozunk létre, amely a System.MulticastDelegate osztályból származik. Ennek az osztálynak a leszármazottai hozzáférnek ahhoz a listához, amely a metódusreferencia-típus által kezelt metódusok címait tartalmazza, valamint a hívási listával történő kommunikáláshoz különböző metódusokat (és néhány túlterhelt operátort). Nézzük meg a System.MulticastDelegate osztály néhány tagját:

```
public abstract class MulticastDelegate : Delegate
{
    // A "mutatott" függvények listáját adja vissza.
    public sealed override Delegate[] GetInvocationList();

    // Túlterhelt operátorok
    public static bool operator ==(MulticastDelegate d1,
    MulticastDelegate d2);
    public static bool operator !=(MulticastDelegate d1,
    MulticastDelegate d2);

    // Privát használatú, a metódusreferencia metóduslistájának
    // kezelésére.
    private IntPtr _invocationCount;
    private object _invocationList;
}
```

A System.MulticastDelegate osztály a System.Delegate nevű szülőosztályától is kap további funkcionálisításokat. A következőkben az osztálydefiníció egy részlete látható:

```
public abstract class Delegate : ICloneable, ISerializable
{
    // Metódusok a függvények listájának eléréséhez.
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
    public static Delegate RemoveAll(Delegate source, Delegate value);

    // Túlterhelt operátorok
    public static bool operator ==(Delegate d1, Delegate d2);
    public static bool operator !=(Delegate d1, Delegate d2);
}
```

```
// Tulajdonságok, amelyek megmutatják a metódusreferenciák célját.  
public MethodInfo Method { get; }  
public object Target { get; }  
}
```

Közvetlenül soha sem származtathatunk osztályokat ezekből az alaposztályokból (ha mégis megteszszük, az fordítási hibát generál). Mindeazonáltal, ha a delegate kulcsszót használjuk, akkor indirekt módon multicastdelegate osztályt hozunk létre. A 11.1. táblázat a metódusreferencia-típusok alapvető tagjait mutatja.

| Örökít tag | Valós jelentés |
|------------|----------------|
|------------|----------------|

|                     |                                                                                                                                                                                                                                                                                    |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Method              | Ez a tulajdonság System.Reflection.MethodInfo típusú ad vissza, amely a metódusreferencia által kezelt statikus metódusokról tartalmaz információt.                                                                                                                                |
| Target              | Ha a hívott függvényt objektumszinten definiáljuk (nem pedig statikus metódusként), akkor a Target egy olyan objektumot ad vissza, amely a metódusreferencia által kezelt függvényt reprezentálja. Ha a Target által visszaadott érték null, akkor a hívott függvény statikus tag. |
| Combine()           | Ez a statikus függvény a metódusreferencia listájához ad hozzá egy metódust. A C# nyelv ezt a metódust kíválthajuk a rövidítlésként használt += túlterhelt operátorral.                                                                                                            |
| GetInvocationList() | Ez a metódus System.Delegate típusok tömbjét adja vissza, amelyek mindegyike egy adott, meghívható metódust reprezentál.                                                                                                                                                           |
| Remove()            | Ezek a statikus metódusok eltávolítanak egy (vagy az összes) metódust a metódusreferencia hívási listájából. A C# nyelvben a Remove() metódust indirekt módon is meghívhatjuk a -= túlterhelt operátorral.                                                                         |

11.1. táblázat: System.MulticastDelegate/System.Delegate tagok

## A leggyorsabb metódusreferencia

A metódusreferencia eleinte bonyolultnak tűnik. Nézzünk meg egy nagyon egyszerű parametereket tartalmazó (a neve SimpleDelegate), amely a korábban láttatott bináris metódusreferencia-típust használja. A következő kódot fogjuk elemezni:

```

namespace SimpleDelegate
{
    // Ez a metódusreferencia bármilyen metódusra mutathat, amely
    // két egész szám paramétert vár, és egész szám értéket ad vissza.
    public delegate int BinaryOp(int x, int y);

    // Ez az osztály tartalmazza azokat a metódusokat,
    // amelyekre a BinaryOp mutatni fog.
    public class SimpleMath
    {
        public static int Add(int x, int y)
        { return x + y; }
        public static int Subtract(int x, int y)
        { return x - y; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Simple Delegate Example *****\n");

            // Hozzuk létre a BinaryOp objektumot, amely
            // a SimpleMath.Add() metódusra "mutat".
            BinaryOp b = new BinaryOp(SimpleMath.Add);

            // Hívjuk meg az Add() metódust indirekt módon,
            // a metódusreferencia objektumot használva.
            Console.WriteLine("10 + 10 is {0}", b(10, 10));
            Console.ReadLine();
        }
    }
}

```

A binaryop metódusreferencia formátuma bármilyen függvényre mutathat, amelyik két egész paramétert vár, és egész értéket ad vissza (a mutatót függvény tényleges neve nem számít). Létrehozunk egy simplemath nevű osztályt, amely két statikus metódust definiál, és ezek (meglepetésre) egyeznek a binaryop metódusreferencia mintájával.

Ha a célmetódust szeretnénk beletenni egy adott metódusreferenciába, akkor egyszerűen át kell adni a metódus nevét a metódusreferencia konstruktorának. Ekkor már meg lehet hívni a mutatót tagot, a közvetlen függvényhíváshoz hasonló szintaxist használva:

```

// Az Invoke() metódust itt hívjuk!
Console.WriteLine("10 + 10 is {0}", b(10, 10));

```

Futás közben, a háttérben valójában a fordító által generált `Invoke()` meghívása történik. Ezt leellenőrizhettük úgy, hogy a szereplőnyt megnyitjuk az `ildasm.exe` programmal, és megnézzük a `main()` függvényben a CIL-kódot:

```
.method private hidebysig static void main(string[] args)
{
    cil managed
    {
        ..callvirt instance int32 SimpleDelegate.BinaryOp::Invoke(int32,
            int32)
    }
}
```

Habár a C# nyelvben nincs szükség az `Invoke()` metódus explicit hívására, ám ezt mégis bármikor szabadon megtehetjük. Ennek megfelelően helyes a következő utasítás:

```
Console.WriteLine("10 + 10 is {0}", b.Invoke(10, 10));
```

A .NET-metódusreferenciák *tipusbiztosak*. Ennek következtében, ha egy metódusreferenciának olyan metódust adunk át, amelyik nem „illeszkedik a mintához”, akkor fordítás idejű hibát kapunk. Ennek bemutatására tegyük fel, hogy a `SimpleMath` osztályunk definícióját egy további `SquareNumber()` nevű metódust, amely egyetlen egész számot vár bemenetként:

```
public class SimpleMath
{
    ...
    public static int SquareNumber(int a)
    {
        return a * a;
    }
}
```

Mivel a `BinaryOp` metódusreferencia *csak* olyan metódusokra mutathat, amelyek két egész bemenő paramétert várnak, és egész a visszatérési értékük, így a következő kód helytelen, és nem fordul le:

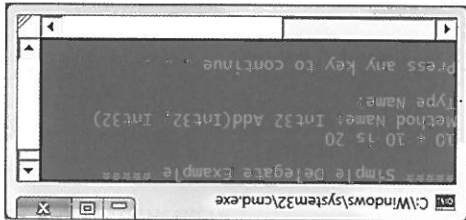
```
// Hiba! A metódus nem egyezik a metódusreferencia mintájával!
BinaryOp b2 = new BinaryOp(SimpleMath.SquareNumber);
```

## A metódusreferencia-objektum vizsgálata

Hozzunk létre a példánkban egy statikus metódust `DisplayDelegateInfo()` néven a `Program` típuson belül. Ez a metódus kiírja a bejövő metódusreferenciátípus által kezelt metódusok neveit, valamint a metódust definiáló osztály nevét is. Ehhez iterálni fogunk a `system.delegate` tömbön, amelyet a `get-InvocationList()` metódus ad vissza, és meghívjuk minden egyes objektum `Target` és `Method` tulajdonságát:

```
static void DisplayDelegateInfo(Delegate delObj)
{
    // Írjuk ki minden egyes tag nevét
    // a metódusreferencia hívási listájában.
    foreach (Delegate d in delObj.GetInvocationList())
    {
        Console.WriteLine("Method Name: {0}", d.Method);
        Console.WriteLine("Type Name: {0}", d.Target);
    }
}
```

Felteve, hogy módosítottuk a `Main()` metódusunkat ennek az új segédmetódusnak a meghívására, a 11.2. ábrának megfelelő kimenetet kapjuk.



11.2. ábra: A metódusreferencia hívási listájának vizsgálata

A típus nevét (`SimpleMath`) nem írja ki a `Target` tulajdonság. Ez azzal magyarázható, hogy a `BinaryOp` metódusreferencia *staticus* metódusra mutat, és így nincs hivatkozott objektum. Ha azonban módosítjuk az `Add()` és a `Subtract()` metódusokat úgy, hogy ne legyenek statikusak (egyszerűen töröljük a `static` kulcsszavakat), létrehozhatjuk a `SimpleMath` típus egy példányát, és objektumreferencia használatával megadhatjuk a meghívandó metódusokat:

```
static void Main(string[] args)
{
    Console.WriteLine("**** Simple Delegate Example ****\n");
}
```



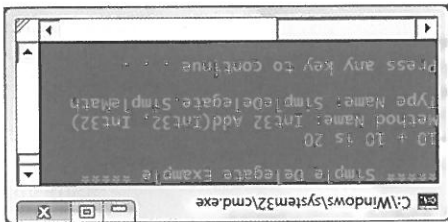
- Definíáljunk új metódusreferencia-típusokat, amelyek üzenetet küldenek a hívónak.
- Deklaráljunk tagváltozókat a car osztályban ehhez a metódusreferencia-típushoz.

Az előző simpleDelegate példa egyértelműen csak illusztráció, hiszen nem lenne értelme metódusreferenciát írni két szám összeadására. A metódusreferencia-típus elelszörbb felhasználásának bemutatásához tervezzük át a 9. fejezetben létrehozott car típust úgy, hogy az információkat .NET-metódusreferenciával küldjük az egyedi callback interfész helyett. Az EngineeringIntefication interfészt a car már nem valósítja meg, így a következő lépések szülkségeseek:

## A Car típus utólagos módosítása metódusreferenciákkal

Forráskód A SimpleDelegate projekt megtalálható a 11. fejezet alkönyvtárában.

11.3. ábra: A metódusreferencia hívási listájának vizsgálata



Ebben az esetben a 11.3. ábrán látható kimenetet kapjuk.

```
// A .NET-metódusreferencia mutathat példánymetódusokra is.  
SimpleMath m = new SimpleMath();  
BinaryOp b = new BinaryOp(m.Add);  
  
// Jelenítsünk meg információt az objektumról.  
DisplayDelegateInfo(b);  
  
Console.WriteLine("10 + 10 is {0}", b(10, 10));  
Console.ReadLine();  
}
```

A Car típus utólagos módosítása metódusreferenciákkal

- Hozzunk létre segédfüggvényeket a car osztályban, amelyek lehetővé teszik a metódusreferencia-tagváltozók által kezelt metódusok beállítását.
- Frissítsük az `Accelerate()` metódust úgy, hogy a megfelelő körülmények között hívja meg a metódusreferencia hívási listáját.

Ennek első lépéseként hozzuk létre a `CarDelegate` nevű parancssoralkalmazás-projektet, és a 9. fejezet `CarBackInterface` példájából illesszük bele a `Car` és a `Radio` definíciókat (megváltoztathatjuk az ezeket a típusokat tartalmazó névtér az aktuális projekt névre, vagy alternatívaként importálhatjuk a `CarBackInterface` névtérre is). Nézzük meg a `Car` osztály következő változtatásait, ezek az első három pontot valósítják meg:

```
public class Car
{
    // Defináljuk a metódusreferencia-típusokat.
    public delegate void AboutRadio(string msg);
    public delegate void Exploded (string msg);

    // Defináljuk az egyes metódusreferencia-típusok tagváltozót.
    private AboutRadio aboutRadio;
    private Exploded exploded;

    // Tagok hozzáadása a hívási listához segédmetódusokkal.
    public void OnAboutRadio(AboutRadio clientMethod)
    { aboutRadioList = clientMethod; }

    public void OnExploded(Exploded clientMethod)
    { explodedList = clientMethod; }
    ...
}
```

Ebben a példában a referenciátípusokat közvetlenül a `Car` típus hatókörén belül definiáltuk. Ha megnezzük az alaposztálykönyvtárakat, akkor kiderül: elégé általános, hogy a metódusreferenciákat annak a típusnak a hatókörén belül szokás definiálni, amellyel természetesen egyetemes osz- meg kell jegyeznünk, hogy ha a fordító a metódusreferenciát egy teljes osz- tálydefinicióvá alakítja, akkor valószínűleg két beágyazott osztályt (`AboutRadio` és `Exploded`) hozunk létre a `Car` osztályon belül.

Két privát tagváltozót (mindkét metódusreferencia-típusnak egyet) és két segédmetódust (`OnAboutRadio()` és `OnExploded()`) deklaráltunk, amelyek lehetővé teszik az ügyfélnek, hogy metódusokat adjon hozzá a metódusreferencia hívási listájához. Konceptiójukban ezek a metódusok nagyon hasonló- tanak a `CarBackInterface` példához (lásd `Advise()` és `Unadvise()` me-

tódusokhoz. Természetesen ebben az esetben a bejövő paraméter egy ügyfél által lefoglalt metódusreferencia–objektum lesz, nem pedig az egyedí interfész megvalósító osztály.

---

**Megjegyzés** Szigorúan véve publikusként is definiálható volna a metódusreferencia tagvaltozót, és ezzel elkerülhetők volna, hogy további regisztrációs metódusokat kelljen létrehozni. Mindamellett azzal, hogy a tagokat privátként hozzuk létre, az egyszerűsége zárást segítjük, és így sokkal típusbiztosabb megoldást nyújtunk. Meg fogjuk később vizsgálni a nyilvános metódusreferencia vesztélyeit a C# event kulcsszavának tanulmányozásakor.

---

Ezen a ponton módosítanunk kell az Accelerate() metódust a metódusreferenciák meghívására, ahelyett, hogy az Array-ist segítségével iterálnánk az ügyféloldali nyelön (ahogy a callbackinterface példában tettük). A módosítás a következő:

```
public void Accelerate(int delta)
{
    // Ha az autó tönkrement, akkor kiváltjuk az Exploded eseményt.
    if (carIsDead)
    {
        if (explodedList != null)
            explodedList("Sorry, this car is dead...");
    }
    else
    {
        currSpeed += delta;

        // Majdnem tönkrement?
        if (10 == maxSpeed - currSpeed
            && almostDeadList != null)
        {
            almostDeadList("careful buddy! gonna blow!");
        }

        // Még jó!
        if (currSpeed >= maxSpeed)
            carIsDead = true;
    }
    Console.WriteLine(">CurrSpeed = {0}", currSpeed);
}
```

Mielőtt meghívánk az almostDeadList és az explodedList tagváltozók által kezelt metódusokat, ellenőrizzük, hogy nem null-e az értékük. Erre azért van szükség, mert a hívó feladata ezeknek az objektumoknak a lefoglalása az onaboutTlow() és onExploded() segédmetódusok hívásával. Ha a hívó nem

hívja meg ezeket a metódusokat, mi viszont megpróbáljuk meghívni a metódusreferencia hívási listáját, akkor `NullPointerException` típusú hibát vettünk ki futási időben (és ez nyilvánvalóan nem jó). Ha megvan a metódusreferencia-infrastruktúránk, nézzük meg a Program osztály változtatásait is:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Delegates as event enablers *****\n");
        // Hozzuk létre a Car típust szokásos módon.
        Car c1 = new Car("SlugBug", 100, 10);
        // Az eseménykezelők bejegyzése.
        c1.OnAboutToBlow(new Car.AboutToBlow(Car.AboutToBlow));
        c1.OnExploded(new Car.Exploded(Car.Exploded));
        // Gyorsítás (ez kiváltja az eseményeket).
        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }
}

// A Car hívja a következő függvényeket.
public static void CarAboutToBlow(string msg)
{
    Console.WriteLine(msg);
}

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

A legfontosabb a fentiekben az, hogy a hívó rendeli hozzá az értékeket a metódusreferencia tagválasztáshoz a segédregisztrációs metódusokkal. Mivel az `AboutToBlow` és az `Exploded` metódusreferenciák a `Car` osztályon belül vannak, így ezekre teljes nevükkel (pl. `Car.AboutToBlow`) kell hivatkoznunk. Mint minden metódusreferencia-konstrukciónál, itt is a függvény nevét adjuk át bemenő paraméterként, amely így hozzáadódik a hívási listához, amely ebben az esetben két statikus tag a Program osztályban (ha ezeket a metódusokat új osztályba szeretnénk tenni, akkor hasonlóan kell eljárni, mint a `CarEvents`-in típusnál a `CallBackInterface` példában).

## Többes küldés engedélyezése

A .NET-metódusreferencia támogatja a *többes küldést* (multiplicast). Másként: a metódusreferencia-objektum a hívható metódusok listáját kezeli, nem csak egy metódust. Ha egy metódusreferencia-objektumhoz több metódust is hozzá szeretnénk adni, akkor egyszerűen a túlterhelt += operátort kell használnunk, nem pedig a közvetlen értékadást. A car típuson a többes küldés engedélyezéséhez a következőképpen kell átírnunk az onabouttrollow() és onexploded() függvényeket:

```
public class Car
{
    // Tagok hozzáadása a hívási listához.
    public void onabouttrollow(AboutTrollow clientMethod)
    { almostDeadlist += clientMethod; }

    public void onexploded(Exploded clientMethod)
    { explodedlist += clientMethod; }
}
```

Ezzel a hívó több célt is regisztrálhat ugyanahhoz a visszahíváshoz:

```
class Program
{
    static void main(string[] args)
    {
        Console.WriteLine("***** Delégates as event enablers *****\n");
        Car c1 = new Car("SlugBug", 100, 10);

        // Több eseménykezelő bejegyzése!
        c1.onabouttrollow(new Car.AboutTrollow(CarSlamMostDoomed));
        c1.onexploded(new Car.Exploded(CarExploded));
    }

    // A Car hívja a következőket.
    public static void CarAboutTrollow(string msg)
    { Console.WriteLine(msg); }
    public static void CarSlamMostDoomed(string msg)
    { Console.WriteLine("Critical Message from Car: {0}", msg); }
    public static void CarExploded(string msg)
    { Console.WriteLine(msg); }
}
```

A köztes nyelvi kódban (CIL) a `+` operátor fordítása a `Delegate.Combine` módszer hívására történik (valójában meghívhatunk a `Delegate.Combine` metódust közvetlenül is, de a `+` használata egyszerűbb). Nézzük meg az `onAboutBox() metódus CIL-megvalósítását`:

```
.method public hidebysig instance void OnAboutBox()
{
    class CarDelegate, Car/AboutBox
    {
        maxstack 8
        ldarg.0
        dup
        ldftd class CarDelegate, Car/AboutBox
        CarDelegate.CAR::almostDeadList
        ldarg.1
        call class [mscorlib]System.Delegate
        [mscorlib]System.Delegate::Combine
        class [mscorlib]System.Delegate,
        class [mscorlib]System.Delegate
        castclass CarDelegate, Car/AboutBox
        stfld class CarDelegate, Car/AboutBox
        CarDelegate.CAR::almostDeadList
        ret
    }
}
```

A `Delegate` osztály egy statikus `remove()` metódust is definiál, amely lehetővé teszi a hívónak, hogy egy tagot dinamikusan eltávolítson a listáról. A `#+` fejlesztők ehelyett a `--` túlterhelt operátort is használhatják.

**Megjegyzés** A `remove()` metódus számára átadott objektumnak a hozzárendelt szignatúrá-jával kell egyeznie, de a törlni kívánt aktuális metódusreferencia-példánnyal nem. Vagyis nem kell tárolnunk a gyűjteményhez hozzáadott pontos metódusreferencia-objektumot ahhoz, hogy pontosan ezt a példányt tudjuk eltávolítani. Létrehozhatunk egy új metódusreferencia-objektumot, amely ugyanarra a függvényre mutat az adott példányon, és ezt átadhatjuk a `remove()` metódusnak. A metódusreferencia egyezni fog az előző metódusreferenciával, hiszen ugyanarra az objektumra mutat.

Háa lehetővé szeretnénk tenni a hívónak, hogy leválassza magát az `AboutBox` és az `exploded` üzenetekről, akkor a `car` típushoz egy további segéd-metódust kell adnunk (figyeljük meg a `--` operátor működését):

```
public class Car
{
    // Tag eltávolítása a hívási listáról.
    public void RemoveAboutBox(AboutBox clientMethod)
    { almostDeadList -= clientMethod; }
}
```

```
public void RemoveExploded(Exploded clientMethod)
{ explodedList -= clientMethod; }
...
```

A -= szintaxis megint csak egy egyszerű alternatív jelölés a statikus Delegate.Remove() metódus kézi hívásához, ahogy azt a car típus RemoveAboutLow() tagjának alábbi CIL-kódja illusztrálja:

```
.method public hidebysig instance void RemoveAboutLow(class
CarDelegate.CAR/AboutLow clientMethod) cil managed
{
    .maxstack 8
    ldarg.0
    dup
    ldftd class CarDelegate.CAR/AboutLow
    CarDelegate.CAR::almostDeadList
    ldarg.1
    call class [mscorlib]System.Delegate
[miscorlib]System.Delegate::Remove(
class [mscorlib]System.Delegate,
class [mscorlib]System.Delegate)
castclass CarDelegate.CAR/AboutLow
stfld class CarDelegate.CAR/AboutLow
CarDelegate.CAR::almostDeadList
ret
}
```

## Cél eltávolítása a metódusreferencia hívási listájáról

A car osztály aktuális módosításaival leállíthatjuk az exploded jelentéseket a main() függvény következő módosításával:

```
static void main(string[] args)
{
    Console.WriteLine("***** Delegates as event enablers *****\n");
    Car c1 = new Car("StuBug", 100, 10);
    // Tartsuk meg a Car.Exploded metódusreferencia-objektumot
    // későbbi használatra.
    Car.Exploded d = new Car.Exploded(Car.Exploded);
    c1.OnExploded(d);
    // Távolítsuk el a CarExploded metódust
    // a hívási listából.
    c1.RemoveExploded(d);
    ...
}
```

```
// módosított Car osztály
public class Car
{
    ...
    // Le kell mosni? Forgatni kell a gumiabroncsokat?
    private bool isDirty;
    private bool shouldRotate;

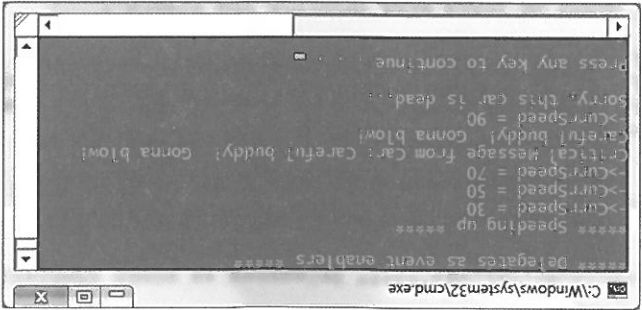
    // További paraméterek a logikai változók beállításához.
    public Car(string name, int max, int curr,
        bool washCar, bool rotateTires)
    {
        ...
        isDirty = washCar;
        shouldRotate = rotateTires;
    }
}
```

Egy összetettebb metódusreferencia-példához hozzunk létre egy parancssori alkalmazást CarGarage néven (a Car/Radio típusdefiniciókat is tartalmaznia kell az új projekteknek). Egészítsük ki a car típust két új logikai tagváltozóval. Az elsővel azt döntjük el, hogy le kell-e már mosni az autót (isDirty); a második azt mutatja, hogy a kérdéses autó gumiabroncsai forogjanak-e (shouldRotate). Az objektumhasználo számára lehetővé kell tennünk az új állapot-adatok alkalmazását, ehhez a car néhány további tulajdonságot és egy módosított konstruktort definiál. A módosított osztály a következő:

## Egy összetettebb metódusreferencia

**Forráskód** Az CarDelegate egate projekt megtalálható a 11. fejezet alkönyvtárában.

11.4. ábra: A CarDelegate alkalmazás működése



A CarDelegate alkalmazás kimenete a 11.4. ábrán látható.



```

    public boolean dirty
    {
        get { return isDirty; }
        set { isDirty = value; }
    }
    public boolean rotate
    {
        get { return shouldRotate; }
        set { shouldRotate = value; }
    }
}

```

Tételezzük fel azt is, hogy a `Car` típus tartalmaz egy további, `CarMaintenanceDelegate` nevű metódusreferencia-típust is:

```

// A Car definiál egy másik metódusreferenciát.
public class Car
{
    // Meg tud hívni bármilyen metódust, amelynek Car a
    // bemenő paramétere, és nincs visszatérési értéke.
    public delegate void CarMaintenanceDelegate (Car c);
    ...
}

```

A `CarMaintenanceDelegate` típus bármilyen függvényre mutathat, amely `Car` típust bemenő paramétert vár, és sincsen visszatérési értéke.

## Metódusreferenciák mint paraméterek

Az új metódusreferencia-típus mellett, amely `Car` típust paramétert váró metódusokra mutat, és sincsen visszatérési értéke, létrehozhatunk olyan más függvényeket is, amelyek ezt a metódusreferenciát várják paraméterként. Ehhez tegyük fel, hogy van egy új, `Garage` nevű osztályunk is, ez pedig olyan `Car` típusok gyűjteményét kezeli, amelyet a `List<T>` lista tartalmaz. Az ilyen `List<T>` listát felöltöttük néhány induló `Car` típussal (az új kódunkba importálni kell a `System.Collections.Generic` névtérét):

```

// A Garage osztály kezeli a Car típusok listáját.
public class Garage
{
    // A garázsban álló autók listája.
    private List<Car> theCars = new List<Car>();
    // Hozzuk létre a garázsban álló autókat.
}

```

```

    public Garage()
    {
        // Emlékezzünk, hogy módosítottuk a konstruktor az isDirty és
        // shouldrotate paraméterek beállítására.
        theCars.Add(new Car("Viper", 100, 0, true, false));
        theCars.Add(new Car("Fred", 100, 0, false, false));
        theCars.Add(new Car("BillyBob", 100, 0, false, true));
    }
}

```

A garage osztály még egy `ProcessCars()` metódust is definiál, amely az új metódusreferenciánk (`car.Carmaintenancedelegate`) egyetlen argumentumát veszi át. A `ProcessCars()` megvalósításában a gyűjteményünkben lévő `car` objektumokat a metódusreferencia paramétereként adja át a „mutatott függvénynek”. A `ProcessCars()` metódus a `system.MulticastDelegate Target` és `method` tagjait is használja annak meghatározására, hogy a metódusreferencia pontosan melyik függvényre mutat:

```

// A garage osztálynak van egy metódusa, amely használja
// a Carmaintenancedelegate metódusreferenciát.
public class Garage
{
    ...
    public void ProcessCars(Car.Carmaintenancedelegate proc)
    {
        // Hova irányítjuk át a hívást?
        console.WriteLine("**** Calling: {0} ****",
            proc.Method);
        // Példánymetódust vagy statikus metódust hívunk?
        if(proc.Target != null)
            console.WriteLine("-->Target: {0} ", proc.Target);
        else
            console.WriteLine("-->Target is a static method");
        // Hívjuk meg a „mutatott” metódust, átadva neki az autókat.
        foreach (Car c in theCars)
        {
            console.WriteLine("\n-> Processing a Car");
            proc(c);
        }
    }
}

```

Mint bármelyik metódusreferencia-műveletnél a `ProcessCars()` híváskor is elküldjük a metódus címét, amely a kérést fogja kezelni (a metódusreferencia-típuson keresztül). A metódusreferencia-típus mutathat statikus vagy cíatípuson keresztül). A metódusreferencia-típus mutathat statikus vagy példányszintű metódusokra is. Tetelezzük fel, hogy ezek példányszintű tag-

függvények, a nevük: washCar() és rotateTires(), és a ServiceDepartment nevű osztályunk definiálja őket. Ez a két metódus a Car típus új rotate és dirty tulajdonságait használja.

```
//Ez az osztály olyan metódusokat definiál, amelyeket
// a Car.carmaintenancedelegate típus hív meg.
public class ServiceDepartment
{
    public void washCar(Car c)
    {
        if(c.dirty)
            Console.WriteLine("Cleaning a car");
        else
            Console.WriteLine("This car is already clean...");
    }

    public void rotateTires(Car c)
    {
        if(c.Rotate)
            Console.WriteLine("Tires have been rotated");
        else
            Console.WriteLine("Don't need to be rotated...");
    }
}
```

Az új Car.carmaintenancedelegat, Garage és ServiceDepartment típusok közötti kapcsolat bemutatására tételjezzük fel a következő használatot:

```
// A Garage minden munkamegbízzást a ServiceDepartmenthez irányít
// (jó szerezőt találni mindig probléma...)
static void Main(string[] args)
{
    Console.WriteLine("*****Delegates as Parameters *****\n");

    // Hozzuk létre a garázst.
    Garage g = new Garage();

    // Hozzuk létre a szervízt.
    ServiceDepartment sd = new ServiceDepartment();

    // Mossuk le az összes piszkos autót.
    g.ProcessCars(new Car.carmaintenancedelegat(sd.WashCar));

    // Forasszuk a gumitábroncsokat.
    g.ProcessCars(new Car.carmaintenancedelegat(sd.RotateTires));

    Console.ReadLine();
}
```

A 11.5. ábrán láthatjuk az aktuális kimenetet.

```
// A CarDelegate a ServiceDepartment.washCar függvényre mutat.
public void ProcessCars(Car.CarMaintenanceDelegate proc)
{
    foreach(Car c in theCars)
    {
        proc(c); // proc(c) => ServiceDepartment.washCar(c)
    }
}
```

ProcessCars() a következőképpen értelmezhető:

megmagyarázza, hogy egy 30 perces olajcsere miért tart 2 óráig). Így a Pro-  
 több igazi garázsban is, a tényleges munkát kiosztják a szervizosztálynak (ez  
 juk át ezt az objektumot a garage.ProcessCars() metódusnak." Mint a leg-  
 washCar() metódus címét a Car.CarMaintenanceDelegate objektumba, és ad-  
 ez gyakorlatilag a következőt jelenti: "Illesszük be a ServiceDepartment.

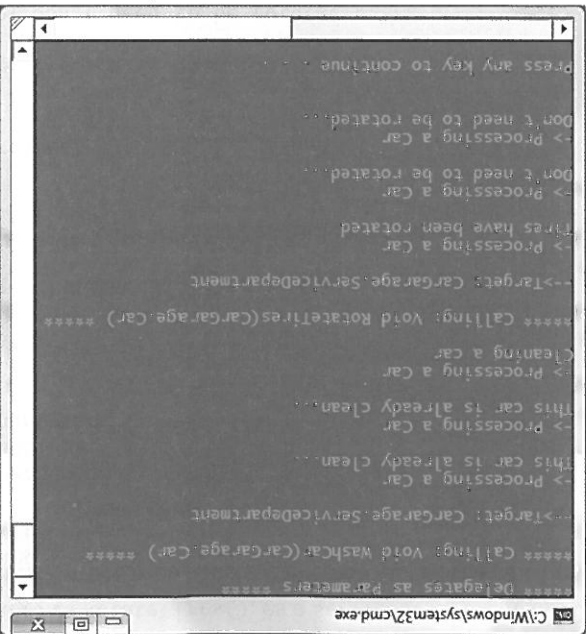
```
// Mossuk le az összes piszkos autót.
g.ProcessCars(new Car.CarMaintenanceDelegate(sd.washCar));
```

létrehozásával kezdődik. Ha a következőket írjuk:

A main() metódus a garage és a ServiceDepartment típusok egy példányának

## Az átírányító kód elemzése

11.5. ábra: Továbbpassezolás



Ehhez hasonlóan, ha a következőket mondjuk:

```
// Forgassuk a gumtábroncsokat.  
g.ProcessCars(new Car.CarMaintenancedelegat(sd.Rotaterires));  
  
akkor a ProcessCars() az alábbiak szerint értelmezhető:
```

```
// A Cardelegat a Servicedepartment.Rotaterires függvényre mutat:  
public void ProcessCars(Car.CarMaintenancedelegat proc)  
{  
    foreach(Car c in theCars)  
        proc(c); // proc(c) => Servicedepartment.Rotaterires(c)  
    ...  
}
```

---

**Forráskód** Az Cararage projekt megtalálható a 11. fejezet alkönyvtárában.

---

Eddig tehát három példát láttunk arra, hogyan működik a metódusreferenciát. Az első példa (simpledelegat) bemutatta a metódusreferencia-típus de-finiálásának és kezelésének alapjait. A metódusreferencia egy rendszeren metódusok listáját kezeli, amelyeket később az Invoke() metódus (implicit vagy explicit) hívásával érhetünk el. A második példánkban (cardelegat) bemutattuk, hogyan használható a metódusreferencia egy eseményarchitektúra létrehozására (a .NET-interfesztsípusok helyett). A harmadik példában pedig kiderült, hogy a metódusreferencia használható metódusparaméterként, épp úgy, mint minden más .NET-típus.

---

**Megjegyzés** A .NET-gyűjteménytípusokban észrevehetjük, hogy általában metódusreferenciát adnak át paraméterként, amikor az elemeket rendezik vagy szűrik.

---

Ezek a példák együttesen bemutatták a metódusreferencia-típus leglényegesebb aspektusait. A következőkben a kovariancia szerepével ismerkedünk meg.

## Metódusreferenciák kovarianciája

Az eddig létrehozott metódusreferenciák mindegyike olyan metódusokra mutat, amelyek visszatérési értéke numerikus adattípus (vagy void). Tegyük fel, hogy van egy parancssori alkalmazásunk DelegateCovariance névvel, és ez olyan metódusreferenciát definiál, amely valamilyen egyedi osztálytípust visszaadó metódusokra is mutathat (az új projekteknél a Car/Radio típusdefini-ciókat is tartalmaznia kell):

```
// Metódusreferenciát definiálunk, amely car típust visszaadó
// metódusokra mutat.
public delegate car obtainCarDelegate();
```

A metódusreferenciát – elvárható módon – az eddigiekhez hasonlóan adhatjuk meg:

```
class Program
{
    public delegate car obtainCarDelegate();

    public static car getBasicCar()
    { return new car("zippy", 150, 50, false, false); }

    static void main(string[] args)
    {
        obtainCarDelegate targetA = new obtainCarDelegate(getBasicCar);
        car c = targetA();
        Console.WriteLine("obtained a {0}", c);
        Console.ReadLine();
    }
}
```

De mi a helyzet akkor, ha a car típusból származtatunk egy sportscar nevű új osztályt, és szeretnénk egy olyan metódusreferencia-típust létrehozni, amely az új típust visszaadó metódusokra mutathat? A .NET 2.0 verzióját megelőződen ilyenkor kénytelenek lettünk volna létrehozni egy teljesen új metódusreferenciát, ugyanis a metódusreferenciák *annymint* típusbiztosak voltak, hogy a származtatás alapjait sem fogadták el:

```
// Sportscar típust visszaadó célokra mutató új metódusreferencia.
public delegate Sportscar obtainSportscarDelegate();
```

Mint ahogy már két metódusreferencia-típusunak van, két példányt kell létrehozunk a car és sportscar típusok kezeléséhez:

```
class Program
{
    public delegate car obtainCarDelegate();
    public delegate Sportscar obtainSportscarDelegate();

    public static car getBasicCar()
    { return new car(); }

    public static Sportscar getSportscar()
    { return new Sportscar(); }
}
```

```

static void Main(string[] args)
{
    Car c = targetA();
    Console.WriteLine("obtained a {0}", c);

    ObtainSportscarDelegate targetB =
        new ObtainSportscarDelegate(GetSportscar);
    Sportscar sc = targetB();
    Console.WriteLine("obtained a {0}", sc);
    Console.ReadLine();
}
}

```

A klasszikus öröklődés szabályait figyelembe véve, ideális lenne egyetlen olyan metódusreferencia-típust létrehozni, amely car és sportscar visszatérési értékekkel is mutathat metódusokra (végül is a sportautó "az egy" autó). A kovariancia (amely *tömör metódusreferencia* [relaxed delegates] néven is ismert) ezt teszi lehetővé. Vagyis a kovariancia olyan metódusreferencia létrehozását biztosítja, amely klasszikus örökléssel kapcsolódó osztálytípusokat visszaadó metódusokra mutathat:

```

class Program
{
    // Definálunk egyetlen metódusreferenciát, amely car vagy
    // sportscar típust adhat vissza.
    public static Car GetBasicCar()
    { return new Car(); }

    public static Sportscar GetSportscar()
    { return new Sportscar(); }

    static void Main(string[] args)
    {
        Console.WriteLine("**** Delegate Covariance ****\n");
        ObtainVehicalDelegate targetA =
            new ObtainVehicalDelegate(GetBasicCar);
        Car c = targetA();
        Console.WriteLine("obtained a {0}", c);

        // A kovariancia lehetővé teszi ezt a célhozzárendelést.
        ObtainVehicalDelegate targetB =
            new ObtainVehicalDelegate(GetSportscar);
        Sportscar sc = (Sportscar)targetB();
        Console.WriteLine("obtained a {0}", sc);
        Console.ReadLine();
    }
}

```

```
namespace GenericDelegate
{
    // Ez a generikus metódusreferencia bármely metódust meghívhatja,
    // amely void értékkel tér vissza, és egyetlen bemenő paraméterre van.
    public delegate void MyGenericDelegate<T>(T arg);

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Generic Delegates *****\n");

            // Céllok bejegyzése
            MyGenericDelegate<string> strTarget =
                new MyGenericDelegate<string>(StringTarget);
            strTarget("some string data");

            MyGenericDelegate<int> intTarget =
                new MyGenericDelegate<int>(IntTarget);
            intTarget(9);
            Console.ReadLine();
        }
    }
}
```

A C# lehetőséget ad generikus metódusreferencia-típus létrehozására (lásd az előző fejezetet). Tegyük fel például, hogy definiálunk egy metódusreferenciát, amely bármely, void visszatérési értékű és egy bemenő paramétert felvevő metódust meg tud hívni. Ha a kérdéses argumentum különbözhet, akkor ezt típusparaméter használataival modellezhetjük. Ehhez vegyük a következő kód-részletet egy új, GenericDelegate nevű paramcsoport alkalmazásában:

## Generikus metódusreferencia készítése

---

**Forráskód** Az DelegateCovariance projekt megtalálható a 11. fejezet alkönyvtárában.

---

---

**Megjegyzés** Hasonlóképpen a kovariancia lehetőséget nyújt egyetlen metódusreferencia létrehozására is, amely olyan metódusokra mutat, amelyek klasszikus származtatással kapcsolódó objektumokat kapnak meg. További információkért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

---

Az objektumok invariánsa metódusreferencia-típust úgy definiáljuk, hogy erősen típusított car típust visszaadó metódusokra mutathasson. A kovariancia azonban mutathat származtatott típusokat visszaadó metódusokra is. A származtatott osztályok tagjainak eléréséhez explicit kasztoást kell használni.



```
static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}

static void IntTarget(int arg)
{
    Console.WriteLine("++arg is: {0}", ++arg);
}
}
```

A `MyGenericDelegate<T>` definiálja azt a `T` típusparamétert, amellyel egy konkrét metódusreferencia felparaméterezhető. Ha ebből a típusból hozunk létre egy példányt, akkor meg kell adnunk a típusparaméter értékét, valamint annak a metódusnak a nevét, amelyet a metódusreferencia hívni fog. Így, ha megadunk egy sztringtípust, akkor sztringértéket küldünk a célmetódusnak:

```
// MyGenericDelegate<T> példány létrehozása
// sztringtípus-paraméterrel.
MyGenericDelegate<string> stringTarget =
    new MyGenericDelegate<string>(StringTarget);
stringTarget("Some string data");
```

Adott az `stringTarget` objektum, így a `stringTarget()` metódus egyetlen sztring-típust paramétert vár:

```
static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}
```

## A generikus metódusreferencia szimulációja generikus típusok nélkül

A generikus metódusreferencia rugalmasan biztosítja, hogy a meghívandó metódust típusbiztosan hívhassuk. A generikus típusok bevezetését megelőzően (a .NET 2.0 megjelenése előtt) hasonló eredményt lehetett elérni a generikus `system.Object` használatával:

```
public delegate void MyDelegate(object arg);
```

Bár ez bármilyen típusú adat elkülönböztetését lehetővé teszi a metódusreferencia-célnak, ám típusbiztonság nélkül és lehetséges dobозolási hátrányokkal. Tegyük fel például, hogy létrehoztunk két példányt a `MyDelegate` típusból, és mindkettő ugyanarra a `MyTarget` metódusra mutat. Figyeljük meg a `dobozolási/kidobozolási` hátrányokat és a típusbiztonság hiányát:

```
class Program
{
    static void Main(string[] args)
    {
        ...
        // Jegyezzük be a célt a „trácionális” metódusreferencia-
        // szintaxissal.
        MyDelegate d = new MyDelegate(MyTarget);
        d("more string data");
        // Metóduscsoport-átalakítás szintaxisa (a fejezetben később
        // részletesen elmagyarázzuk).
        MyDelegate d2 = MyTarget;
        d2(9); // Dobozolási hátrány.
        Console.ReadLine();
    }
    // A típusbiztonság hiánya miatt meg kell határoznunk
    // az alapul szolgáló típust kasztolás előtt.
    static void MyTarget(object arg)
    {
        if(arg is int)
        {
            int i = (int)arg; // Kidobozolási hátrány.
            Console.WriteLine(++arg is: {0}", ++i);
        }
        if(arg is string)
        {
            string s = (string)arg;
            Console.WriteLine(arg in uppercase is: {0}", s.ToUpper());
        }
    }
}
```

Ha értéktípust küldünk a célnak, az érték bedobozolódik, majd kidobozolódik, amikor a célmetódus megkapja. Mivel a bejövő paraméter bármi lehet, ezért kasztolás előtt dinamikusan ellenőrizni kell az alaptípust. Generikus metódusreferencia használatakor a kívánt flexibilitást a „problémák” nélkül is elérhetjük.

---

**Forráskód** Az `GenericDelegate` projekt megtalálható a 11. fejezet alkönyvtárában.

---

Ezzel alapjaiban áttekintettük a .NET metódusreferencia-típusát. (A további részleteket lásd a 18. fejezetben a többszálúság vizsgálatakor.) A következőkben a C# event kulcsszavával folytatjuk.

## C#-események

A metódusreferenciák lehetővé teszik a memóriában lévő objektumok közötti kétirányú kommunikációt. A metódusreferencia alkalmazása azonban formális kódok használatával is jár (metódusreferencia definiálása, szükségessé tagváltozók deklarálása, egyedi beregisztráló/kiregisztráló metódusok a be-

ágyazás megértéséhez stb.).

A gépelési időtől eltekintve, egy másik probléma is adódik akkor, ha a nyers kódban metódusreferenciát használunk az alkalmazásunk callback mechanizmusaként. *Ha* ugyanis nem privátként definiáljuk az osztály metódusreferencia-tagváltozóit, akkor a hívó közvetlenül elérheti a metódusreferencia-objektumot. Ekkor a hívó képes lehet egy másik metódusreferencia-objektumhoz rendelni a változót (gyakorlatilag kitörölve a pillanatnyi hívási listát), és ami még rosszabb, képes lehet közvetlenül aktivizálni a metódusreferencia hívási listáját. A probléma bemutatásához vegyük az előző cardele-

gate példa újrafeldolgozását (és egyszerűsítését):

```
public class Car
{
    // Egyetlen metódusreferencia
    public delegate void Exploded(string msg);

    // Most nyilvános! Nincs többé segédmetódus!
    public Exploded exploded;

    // Vátsuk ki az Exploded üzenetet.
    public void Accelerate(int delta)
    {
        if (exploded != null)
            exploded("Sorry, this car is dead...");
    }
}
```

Ekkor a privát metódusreferencia-tagváltozót nem ágyazzák be saját regisztrációs metódusai. Mivel ezek a tagváltozók valóban nyilvánosak, a hívó közvetlenül el tudja érni az exploded tagot, és újra hozzá tudja rendelni az új exploded objektumokhoz, továbbá akkor hívja meg a metódusreferenciát, amikor akarja:

## Az event kulcsszó

A C# event kulcsszavát használhatjuk arra, hogy olyan saját egyedi metódusokat hozzunk létre, amelyek a metódusreferencia hívási listájához adnak hozzá, illetve távolítanak el metódusokat. Amikor a fordító feldolgozza az event kulcsszót, automatikusan megkapjuk a beregisztráló és kiregisztráló metódusokat, valamint a metódusreferencia-típusokhoz szükséges tagváltozókat. Ezeket a metódusreferencia-tagváltozókat *mindig* privátként deklarál-

---

**Forráskód** Az `Publ1.cs` legutóbbi verziójának a 11. fejezet alkönyvtárában.

---

A nyilvános metódusreferencia-tagok megtörtik az egyébe zárást, és így nemcsak nehezen karbantartható (és nehezen hibakereshető) kódhoz vezetnek, de még potenciális biztonsági részeket is nyitnak. Nyilvánvalóan nem szeretnénk lehetőséget adni más alkalmazásoknak, hogy módosíthassák azt, hogy egy metódusreferencia mire mutasson, illetve hogy engedély nélkül meghívják a tagokat.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("**** Aghi No Encapsulation! ****\n");
        // Hozzunk létre egy autót.
        Car myCar = new Car();

        // közvetlen hozzáférésünk van a metódusreferenciához!
        myCar.explodedList = new Car.Exploded(WhenExploded);
        myCar.Accelerate(10);

        // Hozzárendelhetjük egy teljesen új objektumhoz...
        myCar.explodedList = new Car.Exploded(CallHereToo);
        myCar.Accelerate(10);

        // A hívó közvetlenül meghívhatja a metódusreferenciát!
        myCar.explodedList.Invoke("hee, hee, hee...");
    }

    static void CallWhenExploded(string msg)
    {
        Console.WriteLine(msg);
    }

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

juk, és ennek megfelelően az eseményt kiváltó objektum közvetlenül nem fér hozzájuk. Az event kulcsszó nem pusztán egy kis szintaktikai egyszerűsítés, amely némi gépelestől kímél meg minket.

Egy esemény definiálása kétlépcsős folyamat. Először egy módszerre-renciát kell definiálnunk, amely tárolni fogja az esemény bekövetkeztekor meghívandó módszerek listáját. Azután deklaráljuk az eseményt (a C# event kulcsszavával) a kapcsolódó módszerrenciaiban.

Az event kulcsszó bemutatására nézzük meg az alábbi példát. A car típus iterációja két eseményt definiál (azonos néven, mint az előző AboutTow és exploded módszerrencia). Ezeket az eseményeket egyetlen, CarEvent-handler nevű módszerrenciahoz köthetjük. A car típus első változtatásai a következők:

```
public class Car
{
    // Ez a módszerrencia a car osztály eseményeivel
    // működik együtt.
    public delegate void CarEventHandler(string msg);
    // Ez az autó küldi ezeket az eseményeket.
    public event CarEventHandler Exploded;
    public event CarEventHandler AboutTow;
    ...
}
```

Egy esemény küldése a hívóhoz nem más, mint nevének és a szükségessé para-métereknek a megadása, ahogyan ezt a hozzátartozó módszerrencia defi-niálja. Ahhoz, hogy megbizonyosodjunk róla, hogy a hívó regisztrálása az eseményhez tényleg megtörtént, az eseményt ellenőrizni kell null értékre, mie-lőtt a módszerrencia metódusait meghívjuk. A car osztály Accelerate() metódusának új iterációja a következő:

```
public void Accelerate(int delta)
{
    // Ha az autó tönkrement, akkor kiváltjuk az Exploded eseményt.
    if (CarIsDead)
    {
        if (Exploded != null)
            Exploded("Sorry, this car is dead...");
    }
    else
    {
        currspeed += delta;
    }
}
```

```
// Majdnem tönkrement?
if (10 == maxSpeed - currSpeed
    && AboutToBlow != null)
{
    AboutToBlow("careful buddy! gonna blow!");
}

// Még jó!
if (currSpeed >= maxSpeed)
    carIsDead = true;
else
    Console.WriteLine(">CurrSpeed = {0}", currSpeed);
}
```

Ezzel úgy állítottuk be a car osztályt, hogy két egyedi eseményt küldjön anélkül, hogy egyedi beregisztráló függvényeket kellett volna definiálnunk, vagy metoduserferencia-tagválasztásokat kellett volna deklarálnunk. Az új autó használatának ellenőrzése előtt nézzük meg az esemény szerkezetét kicsit részletesebben.

## Eseménykezelés a felszín alatt

Egy C#-esemény valójában két rejtett nyilvános metódusra bontható, az egyik az `add_`, a másik a `remove_` prefixummal rendelkezik. A prefixumot a C#-esemény neve követi. Az `exploded` esemény eredménye például két CIL-metódus: az `add_exploded()` és a `remove_exploded()`. Az `add_` és a `remove_` metódusok mellett a CIL-szintű eseménydefiníció a megfelelő metoduserferenciát rendel hozzá az adott eseményhez.

Ha megnéznénk az `add_AboutToExplode()` metódus mögötti CIL-utasítást, akkor olyan kódot találunk, amely nagyon hasonlít ahhoz az `onAboutToBlow()` segédmetódushoz, amelyet előzőleg a `carDelegate` példában írtunk (figyeljünk a `Delegate.Combine()` hívásra):

```
.method public hidebysig specialname instance void
    add_AboutToBlow(class CarEvents car/CareventHandler 'value')
{
    cil managed synchronized
        .maxstack 8
        ldarg.0
        ldarg.0
        ldftd class CarEvents car/CareventHandler
        Carevents.car::AboutToBlow
        ldarg.1
        call class [mscorlib]System.Delegate
```

```
[mscorlib]System.Delegate::Combine(
    class [mscorlib]System.Delegate,
    class [mscorlib]System.Delegate,
    castclass CarEvents.Car/CarEventHandler
    stfd class CarEvents.Car/CarEventHandler
    CarEvents.Car::AboutToBlow
    ret
}
```

A `remove_AboutToBlow` indirekt módon hívja a `Delegate.Remove()` metódust, és többé-kevésbé azonos az előző `remove_AboutToBlow()` segédmétódussal:

```
.method public hidebysig specialname instance void
    remove_AboutToBlow(class CarEvents.Car/CarEventHandler 'value')
{
    cil managed synchronized
```

```
,maxstack 8
    ldarg.0
    ldarg.0
    ldftd class CarEvents.Car/CarEventHandler
    CarEvents.Car::AboutToBlow
    ldarg.1
    call class [mscorlib]System.Delegate
    [mscorlib]System.Delegate::Remove(
    class [mscorlib]System.Delegate,
    class [mscorlib]System.Delegate)
    castclass CarEvents.Car/CarEventHandler
    stfd class CarEvents.Car/CarEventHandler
    CarEvents.Car::AboutToBlow
    ret
}
```

Végül pedig magát az eseményt leíró CIL-kód a `add` és a `remove` direktívákat használja a megfelelő meghívandó `add_xxx()` és `remove_xxx()` metódusok nevének leképezéséhez:

```
.event CarEvents.Car/EngineHandler AboutToBlow
{
    .addon void CarEvents.Car::AddAboutToBlow
    (class CarEvents.Car/CarEngineHandler)
    .removeon void CarEvents.Car::RemoveAboutToBlow
    (class CarEvents.Car/CarEngineHandler)
}
```

Mind Ezek után a következő kérdés az, hogy hogyan „figyeljük” a hívóoldalon a bejövő eseményeket.

## A bejövő események figyelése

A C#-események egy szerzősített hívóoldali eseménykezelők beregisztrálását is. Ahelyett, hogy saját segédmetódusokat kellene definiálnunk, a hívó közvetlenül a += és a -= operátorokat használja (amelyek a megfelelő ad-xxx() vagy remove-xxx() metódusokat váltják ki a háttérben). Ha egy eseményhez egy eseménykezelőt szeretnénk regisztrálni, akkor a következő lépésmintát kell követnünk:

```
// ObjectVariable.EventName +=
// new AssociatedDelegate(functionToCall);
Car.EngineHandler d = new
Car.CarEventHandler(Car.ExploDEDEventHandler)
myCar.ExploDED += d;
```

Ha le szeretnénk választani az eseményforrást, akkor a -= operátort használhatjuk:

```
// ObjectVariable.EventName -= delegateObject;
myCar.ExploDED -= d;
```

Adottak ezek az előre jelezhető minták, így a main() metódus átírása a C#-esemény regisztrációs szintaxisának használatára a következő:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Events *****\n");
        Car c1 = new Car("StingBug", 100, 10);

        // Eseménykezelők regisztrálása.
        c1.AboutToBlow += new Car.CarEventHandler(Car.IsAlmostDoomed);
        c1.AboutToBlow += new Car.CarEventHandler(Car.AboutToBlow);
        Car.CarEventHandler d = new Car.CarEventHandler(Car.ExploDED);
        c1.ExploDED += d;

        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);

        // Távolsítjuk el a Car.ExploDED metódust
        // a hívást listából.
        c1.ExploDED -= d;
    }
}
```



```

Console.WriteLine("\n***** Speeding up *****");
for (int i = 0; i < 6; i++)
    cl.Accelerate(20);
Console.ReadLine();
}

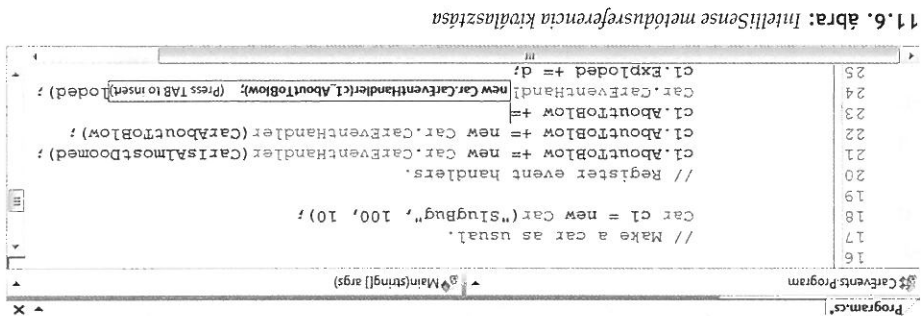
public static void CarAboutToBlow(string msg)
{ Console.WriteLine(msg); }
public static void CarIsAlmostDoomed(string msg)
{ Console.WriteLine("Critical Message from car: {0}", msg); }
public static void CarExploded(string msg)
{ Console.WriteLine(msg); }
}

```

## Esemény beregisztrálásának egyszerűsítése a Visual Studio 2008 használatával

A Visual Studio 2008 segítségével nyújt az eseménykezelők beregisztrálási folyamatában. Ha egy esemény beregisztrálásakor a += szintaxist használjuk, akkor egy IntelliSense ablak jelenik meg, lehetőséget adva arra, hogy a Tab billentyű megnyomásával automatikusan beillesztjük a hozzárendelt módszer

duszerencia-példányt (lásd a 11.6. ábrát).



11.6. ábra: IntelliSense módszerreferencia kiválasztása

Ha lenyomjuk a Tab billentyűt, akkor beírhatjuk az eseménykezelő nevét (vagy egyszerűen elfogadhatjuk az alapértelmezett), ahogyan a 11.7 ábrán látható. Ha megint megnyomjuk a Tab billentyűt, akkor a módszerreferencia céljához megfelelő formátumú kódárazat kapunk (ez a függvény statikusként deklarált, hiszen az eseményt a statikus main() módszerben regisztráltuk):

```

static void cl_AboutToBlow(string msg)
{
    // Irjuk be a forráskódunkat!
}

```



## Iterátor metódus készítése a yield kulcsszóval

Régebben, ha meg akartunk valósítani egy egyedi gyűjtőmenyt (mint a garage), amely támogatja a foreach felsorolást, az IEnumerator-t (és esetleg az IEnumerator-2.0 megjele-  
tenése óta azonban alternatív lehetőségünk is van olyan típusok létrehozásá-  
ra, amelyek a foreach ciklussal működnek, és ez az *iterátor*.

Egyszerűen: az iterátor olyan tag, amely meghatározza, hogy egy tároló  
első elemét hogyan lehet egy foreach feldolgozás során visszaadni. Noha  
az iterátor metódus mindig GetEnumerator() nevű kell, hogy legyen, és a  
visszatérési értéke is IEnumerator típusú kell, hogy legyen, a saját osztá-  
lyunkban nem szükséges semelyik interfész megvalósítása.

Ennek bemutatásához hozzuk létre a customEnumeratorWithYield nevű  
parancssorialkalmazás-projektet, és tegyük bele a car, a radio és a garage típu-  
sokat az előző példából (megtartva a névtérdefiniciókat az aktuális pro-  
jektüre). Ezután tervezzük át a garage típust a következők szerint:

```
public class Garage
{
    private Car[] carArray = new Car[4];

    ...
    // Iterátor metódus.
    public IEnumerator GetEnumerator()
    {
        foreach (Car c in carArray)
        {
            yield return c;
        }
    }
}
```

A GetEnumerator() ezen megvalósítása az elemeket egy belső foreach logi-  
kát használva járja be, és minden egyes car objektumot a yield return szinta-  
xisal ad vissza. A yield kulcsszót használjuk arra, hogy megadjuk azt az ér-  
teket (vagy értékeket), amelyet a hívó foreach szerkezetének vissza kell ad-  
nunk. Amikor elérjük a yield return utasítást, a pillanatnyi helyzet tárolódik,  
és az iterátor következő hívásakor innen indul újra a végrehajtás.

Az iterátormetódusok nem feltétlenül használják a foreach kulcsszót a  
tartalom visszaadására. Ezt az iterátormetódust a következő módon is meg-  
adhatjuk:

```
public IEnumerator()
{
    yield return carArray[0];
    yield return carArray[1];
    yield return carArray[2];
    yield return carArray[3];
}
```

Ebben a megvalósításban a `getenumerator()` módszer explicit módon adja vissza a hívónak az új értéket minden egyes futásakor. Ebben a példánkban ennek nem sok értelme lenne, mert ha a `cararray` tagváltozéhoz további objektumokat adnánk, akkor a `getenumerator()` módszerünk nem működne helyesen. Mindazonáltal ez a szintaxis is hasznos lehet, ha lokális adatot szeretnénk visszaadni egy módszerből a `foreach` szintaxisának feldolgozásra.

## Nevesített iterátor létrehozása

A yelid kulcsszó technikailag bármely módszerben használható, a nevetől függetlenül. Ezek a módszerek (amelyek technikailag *nevesített iterátorok*) olyan szempontból is különlegesek, hogy akármiilyen számú argumentumot tudnak fogadni. Ha nevesített iterátort készítettünk, legyünk mindig nagyon körültekintők, mert a módszer az *enumerator*-interfészt adja vissza, nem pedig a vártnumberator-kompatibilis típust. Ennek bemutatására adjuk a következő módszert a garage típushoz:

```

public IEnumerable<Car> GetTheCars(bool ReturnReversed)
{
    // Az elemeket fordított sorrendben adjuk vissza.
    if (ReturnReversed)
    {
        for (int i = carArray.Length; i != 0; i--)
        {
            yield return carArray[i-1];
        }
    }
    else
    {
        // Az elemeket a tömbben elhelyezés szerint adjuk vissza.
        foreach (Car c in carArray)
        {
            yield return c;
        }
    }
}

```

Az új módszerünk az a lelemeket szekvenciális sorrendben adja vissza, vagy ha a bejövő paraméter igaz értékű, akkor fordított sorrendben. Ezek után a következő módon használhatjuk az új módszerünket:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with the yield keyword *****\n");
    Garage carlot = new Garage();

    // Elemek beolvasása a GetEnumerator() segítségével.
    foreach (car c in carlot)
    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.Speed);
    }

    Console.WriteLine();
    // Elemek beolvasása (fordított sorrendi) megnevezett iterátorral.
    foreach (car c in carlot.GetTheCars(true))
    {
        Console.WriteLine("{0} is going {1} MPH",
            c.PetName, c.Speed);
    }
}
```

A nevesített iterátorok akkor hasznosak, ha egyetlen egyedi tároló elemeit többféleképpen szeretnénk elérni.

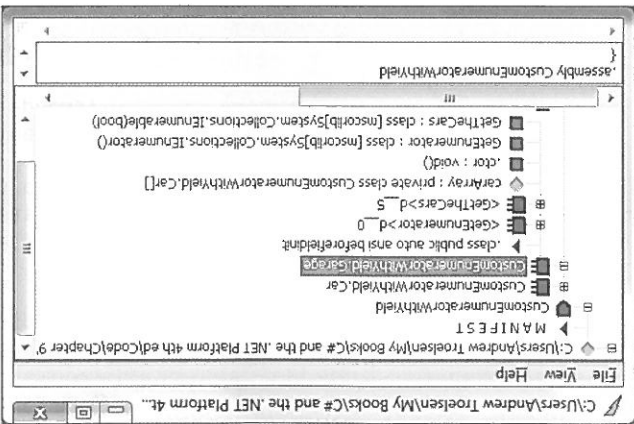
## Az iterátormetódusok belső reprezentációja

Amikor a C#-fordító talál egy iterátormetódust, dinamikusan létrehoz hozzá egy beágyazott osztálydefiniációt a megadott típus hatókörén belül (jelen esetben ez a Garage). Az automatikusan generált beágyazott osztály megvalósítja a GetEnumerator(), a MoveNext() és a Current tagokat helyettünk (különös módon a Reset() metódust nem, és futásidejű kivételt kapunk, ha megpróbáljuk meghívni). Ha betölthetjük az aktuális alkalmazást az i1dasm.exe programmal, akkor két beágyazott típust találunk, amelyek a specifikus iterátormetódusok által igényelt logikákért felelnek. A 9.10. ábrán a számitógép által generált készített típusok neve <GetEnumerator>d\_0 és <GetTheCars>d\_5.

Ha az i1dasm.exe programot használjuk arra, hogy a Garage típus GetEnumerator() metódusának megvalósítását megnézzük, akkor azt találjuk, hogy a rejtett <GetEnumerator>d\_0 típust használja a megvalósítás (hasznolva a beágyazott <GetTheCars>d\_5 típust használja a GetTheCars() metódus).

```

        method public hidebysig instance class
        [mscorlib]System.Collections.Enumerator
        GetEnumerator() cil managed
        {
            ...
            newobj instance void
            CustomEnumeratorWithYield.Garage/
            <GetEnumerator>d__0'::ctor(int32)
            ...
        } // A Garage::GetEnumerator metódus vége.
    
```



9.10. ábra: Az iterátormetódusok belső megvalósítása automatikusan generált beágyazott osztályokkal

Teljesse téve a felsorolható objektumok készítését, megismételjük: ahhoz, hogy az egyedi típusunk a C# foreach kulcsszavával együttműködjön a tárolónak definiálnia kell a GetEnumerator() nevű metódust, amelyet az IEnumerator interfacez formálizál. Ennek a metódusnak a megvalósítását tipikusan a belső taghoz történő delegálással érjük el, ez tartalmazza az alobjektumokat; az is lehetséges azonban, hogy a yield return szintaxis használatával több „nevesített iterátormetódust” nyújtunk.

**Forráskód** A CustomEnumeratorWithYield projekt megtalálható a 9. fejezet alkönyvtárában.

## Klónozható objektumok készítése (Cloneable)

A `System.Object` osztály definíciál egy `MemberwiseClone()` nevű tagot is (lásd 6. fejezet). Ez a módszer arra szolgál, hogy *sekély másolatot* kapjunk az aktuális objektumról. Az objektum használói nem hívják ezt a metódust közvetlenül (hiszen védett); az adott objektum azonban hívhatja a *klónozási* folyamat során. Ennek bemutatásához hozzunk létre egy új parancssori alkalmazást `CloneablePoint` néven, amely egy `Point` nevű osztályt definíál:

```
// A Point nevű osztály.
public class Point
{
    // A könnyebb hozzáféréshez publikus.
    public int x, y;
    public Point(int x, int y) { this.x = x; this.y = y; }
    public Point() {}
    // Az Object.ToString() felülírása.
    public override string ToString()
    { return string.Format("X = {0}; Y = {1}", x, y); }
}
```

Ha már ismerjük a referenciátípusokat és az értékítípusokat (lásd 4. fejezet), akkor tudjuk, hogy ha egy referenciaváltozót értékül adunk egy másiknak, akkor két referenciánk mutat azonos objektumra a memóriában. Így módon a következő értékadás két referenciát eredményez ugyanarra a `Point` objektumra a heapmemóriában. Bármelyik referencia használatával módosítunk, ugyanaz az objektum módosul a heapmemóriában.

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    // Ugyanazon objektum két referenciáját
    Point p1 = new Point(50, 50);
    Point p2 = p1;
    p2.x = 0;
    Console.WriteLine(p1);
    Console.WriteLine(p2);
    Console.ReadLine();
}
```

Ha szeretnénk megadni azt a képességet a saját típusunknak, hogy a hívónak azonos másolatot tudjon visszaadni, akkor a szabványos `ICloneable` interfész kell megvalósítanunk. Ez a típus – ahogy már láttuk – egyetlen `Clone()` nevű metódust definiál:

```
public interface ICloneable
{
    object Clone();
}
```

**Megjegyzés** Az `ICloneable` interfész hasznosságáról pillanatnyilag is vitatkoznak a .NET-közösség tagjai. A probléma abból adódik, hogy a hivatalos specifikáció nem mondja ki kifejezetten az interfész megvalósító objektumokról, hogy *mely másolatot* kell visszaadniuk az objektumról (azaz az objektumok belső referenciátípusai teljesen új objektumot eredményeznek, azaz állapottal). Így technikailag lehetséges az, hogy az `ICloneable` interfészt megvalósító objektumok csak *sekély másolatot* adják vissza az interfésznek (azaz a belső referenciáik a heapmemória azonos objektumára mutatnak), ez pedig tényleg zavaró eredményez. A példánkban mi úgy valósítjuk meg a `Clone()` metódust, hogy az az objektum teljes, mely másolatát adja vissza az objektumnak.

Nyilvánvaló, hogy objektumonként változik a `Clone()` metódus megvalósítása. Az alapműködések azonban nagyjából azonosak: másoljuk le a tagváltozókat értékeit egy azonos típusú, új objektumpéldányba, és ezt adjuk vissza a felhasználónak. Ennek bemutatásához tegyük meg a következő módosításokat a `Point` osztályon:

```
// A Point támogatja a "klonozhatóságot."
public class Point : ICloneable
{
    public int x, y;
    public Point() { }
    public Point(int x, int y) { this.x = x; this.y = y; }

    // Az aktuális objektum másolatát adja vissza.
    public object Clone()
    {
        return new Point(this.x, this.y);
    }

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

Így a `Point` típus pontos, független másolatát hozhatjuk létre, ahogyan a következő kód is mutatja:



Tételezzük fel, hogy a `Point` osztály tartalmaz egy referenciátípusú tagváltozót, amely `PointDescription`-típusú. Ez az osztály egy `Point` barátaságos nevét tartalmazza, valamint egy azonosítószámot `System.Guid` formájában (ha nincsen COM-háttérünk, akkor vegyük figyelembe, hogy a globális egyedi azonosító [GUID] statisztikailag egyedi 128 bites szám). A megvalósítás a következő:

## Egy összetettebb klónozási példa

Ha azonban a `Point` objektum mégis tartalmaz referenciátípusú tagváltozót, akkor a `MemberwiseClone()` metódus lemásolja ezeknek az objektumoknak a referenciáit (úgy, mint a *sekély másolás*). Ha valóban mely másolást szeretnénk elérni, akkor a klónozás folyamán a referenciátípusú változókhoz új példányait kell létrehozni. Nézzünk erre egy példát.

```
public object Clone()
{
    // A Point mezőinek másolása tagról tagra.
    return this.MemberwiseClone();
}
```

Noha a `Point` pillanatnyi megvalósítása már megfelelő a számunkra, még modernizálhatjuk egy kicsit. Mivel a `Point` típus nem tartalmaz semmiféle belsőreferenciátípusú változót, egyszerűsíthetjük a `Clone()` metódus megvalósítását a következő módon:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    // A Clone() generikus objektumtípust ad vissza.
    // Explicit kásztolással olvasható be a lezármasozott típus.
    Point p3 = new Point(100, 100);
    Point p4 = (Point)p3.Clone();
    // A p4.x módosítása (amely nem változtatja meg p3.x-et).
    p4.x = 0;
    // Az objektumok kiírása.
    Console.WriteLine(p3);
    Console.WriteLine(p4);
    Console.ReadLine();
}
```

```
// Az osztály egy pontot ír le.
public class PointDescription
{
    // Az egyszerűség kedvéért publikus módon.
    public String petName;
    public Guid pointID;

    public PointDescription()
    {
        this.petName = "No-name";
        pointID = Guid.NewGuid();
    }
}
```

Az alapvető frissítések a `Point` osztályon magukban foglalják a `ToString()` metódus módosítását ezeknek az állapotadatoknak a kezelésére, valamint a `PointDescription` referenciátípus definiálását és létrehozását. A túlterhelt konstruktor számára átadott paramétereket is módosítjuk, így tesszük lehetővé a felhasználó számára becenév hozzárendelését a `Point` osztályhoz.

```
public class Point : ICloneable
{
    public int x, y;
    public PointDescription desc = new PointDescription();

    public Point(){}
    public Point(int x, int y)
    {
        this.x = x; this.y = y;
    }
    public Point(int x, int y, string petName)
    {
        this.x = x;
        this.y = y;
        desc.petName = petName;
    }

    public object Clone()
    { return this.MemberwiseClone(); }

    public override string ToString()
    {
        return string.Format("x = {0}; y = {1}; Name = {2};\nid = {3};\n",
            x, y, desc.petName, desc.pointID);
    }
}
```

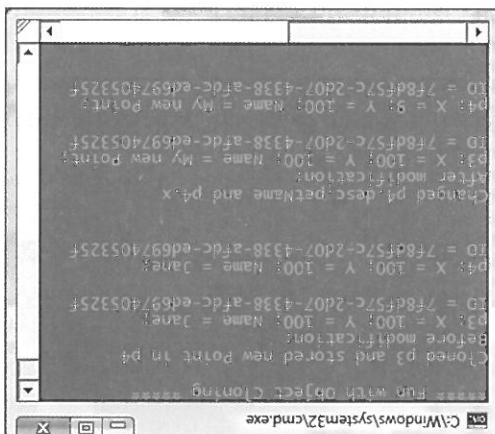
Egyelőre még nem módosítottuk a `clone()` módszerünket. Ennek megfelelően, ha az objektumhasználó a pillanatnyi implementációval próbál klónozni, akkor sekély (tagonként) másolat keletkezik. Ennek bemutatásához vegyük a következő `main()` módszert.

```
static void main(String[] args)
{
    Console.WriteLine("***** Fun with Object Cloning *****\n");
    Console.WriteLine("Cloned p3 and stored new Point in p4");
    Point p3 = new Point(100, 100, "Jane");
    Point p4 = (Point)p3.Clone();

    Console.WriteLine("Before modification:");
    Console.WriteLine("{0}", p3);
    Console.WriteLine("{0}", p4);
    p4.desc.petName = "My new Point";
    p4.x = 9;

    Console.WriteLine("\nchanged p4.desc.petName and p4.x");
    Console.WriteLine("After modification:");
    Console.WriteLine("{0}", p3);
    Console.WriteLine("{0}", p4);
    Console.ReadLine();
}
```

A 9.11. ábra mutatja a kimenetet. Míg az értéktípusok tényleg megváltoztak, a belső referenciátípusok továbbra is azonos értékek, hiszen azonos memóriaterületre mutatnak.



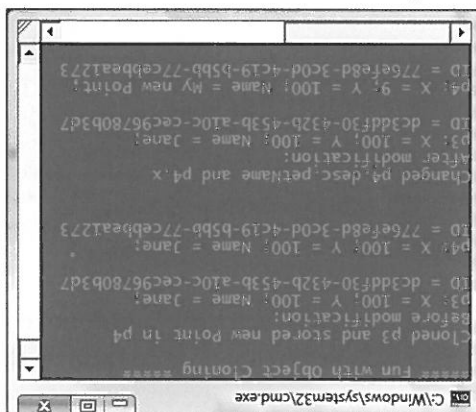
9.11. ábra: A `MemberwiseClone()` módszer az aktuális objektumnak a sekély másolatát adja vissza

Ahhoz, hogy a `clone()` módszer teljesen helyes másolatot készítsen a belső referenciátípusokról, a `memberwiseclone()` módszerrel visszaadott objektumokat kell úgy alakítanunk, hogy kezelje az aktuális pont nevét (a `system.guid` típus valójában egy struktúra, így a numerikus adatot tényleg másoljuk). Ennek egy lehetséges megvalósítása:

```
// A PointDescription taghoz kell igazodni.
public object clone()
{
    // A sekély másolat beolvasása.
    Point newPoint = (Point)this.MemberwiseClone();

    // A hëzagok kitöltése.
    PointDescription currentDesc = new PointDescription();
    currentDesc.petName = this.desc.petName;
    newPoint.desc = currentDesc;
    return newPoint;
}
```

Ha ismét lefuttatjuk az alkalmazásunkat, akkor ahogyan a 9.12. ábrán is látható, a `clone()` módszerrel visszaadott `Point` objektumban a belső referenciátípusú tagváltípusok is másolódtak (a becenév egyedi a `p3` és a `p4` esetében is).



9.12. ábra: Most már valós mély másolatunk van az objektumról.

A klónozási folyamat a következőképpen foglathatók össze: ha olyan osztályunk vagy struktúránk van, amely csak értéktípusokat tartalmaz, akkor a `clone()` módszert valószínűleg meg a `memberwiseclone()` használatával. Ha azonban a saját típusunk referenciátípusokat is kezel, akkor olyan típust kell létrehoznunk, amely figyelembe veszi a referenciátípusú tagváltípusokat is.

**Forráskód** A `CloneablePoint` projekt megtalálható a 9. fejezet alkönyvtárában.

## Összehasonlítható objektumok készítése (IComparable)

A `System.IComparable` interfész olyan viselkedést definiál, amely lehetővé teszi egy kulcs alapján az objektumok sorba rendezését. A formális definíció a következő:

```
// Az interfész lehetővé teszi az objektum számára, hogy
// meghatározza más hasonló objektumokkal való kapcsolatát.
public interface IComparable
{
    int CompareTo(object o);
}
```

Hozzunk létre egy új parancssori alkalmazást `comparable`-nak néven, amely a következő továbbfejlesztett `car` típust definiálja (alapvetően csak egy új tag-elérési lehetőséget valósítottuk meg):

```
public class car
{
    ...
    private int carID;
    public int ID
    {
        get { return carID; }
        set { carID = value; }
    }
    public car(string name, int currsp, int id)
    {
        currsp = currsp;
        petName = name;
        carID = id;
    }
    ...
}
```

Tételezzük fel, hogy van egy `car` típusokat tartalmazó tömbünk:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Object Sorting *****\n");
    // Car típusok tömbjének elkészítése.
    car[] myAutos = new car[5];
    myAutos[0] = new car("Rusty", 80, 1);
    myAutos[1] = new car("Mary", 40, 234);
}
```

```

myAutos[2] = new Car("Viper", 40, 34);
myAutos[3] = new Car("M1", 40, 4);
myAutos[4] = new Car("Chuck", 40, 5);
console.ReadLine();
}

```

A `System.Array` osztály egy `sort()` nevű statikus metódust definiál. Ha ezt a metódust meghívjuk egy belső típust (`int`, `short`, `string` stb.) tartalmazó tömbre, akkor rendezni tudjuk a tömböt szám/abcé sorrendben, minthogy a lenyeges adattípusok megvalósítják az `IComparable` interfészt. Mi történik azonban akkor, ha egy `car` típusú elemeket tartalmazó tömböt adunk át a `sort()` metódusnak?

```

// Rendezzük az autokat?
Array.Sort(myAutos);

```

Ha lefuttatjuk a tesztet, akkor azt tapasztaljuk, hogy futásidőben `ArgumentException` kivételt generálódik a következő üzenettel:

---

```

"At least one object must implement IComparable."

```

---

Ha saját típusokat hozunk létre, akkor megvalósíthatjuk az `IComparable` interfészt, lehetővé téve a tömbök rendezését. Ha megnezzük a `compareTo()` metódus részleteit, akkor kiderül, hogy a rendezés alapja csak rajtuk múlik. A `car` típusnál a belső `carID` logikus választásnak tűnik:

```

// A car objektumok iterációja rendezhető
// a carID alapján.
public class Car : IComparable
{
    ...
    // IComparable megvalósítás.
    int IComparable.CompareTo(object obj)
    {
        Car temp = (Car)obj;
        if (this.carID > temp.carID)
            return 1;
        if (this.carID < temp.carID)
            return -1;
        else
            return 0;
    }
}

```

A `compareTo()` mögötti logika mindössze annyi, hogy a bejövő típust adott adat alapján összehasonlítsa az aktuális példánnyal. A `compareTo()` visszatérési értéket használjuk annak megállapítására, hogy ez a típus kisebb, nagyobb vagy ugyanakkora, mint az aktuális objektum, amellyel hasonlítjuk (lásd a 9.1. táblázatot).

| compareTo() | Visszatérési értéke | Valós jelentés |
|-------------|---------------------|----------------|
|-------------|---------------------|----------------|

|                               |                                                           |
|-------------------------------|-----------------------------------------------------------|
| Bármely nullánál kisebb szám  | A példány az aktuális objektum előtt következik a sorban. |
| Bármely nullánál nagyobb szám | A példány az aktuális objektum után következik a sorban.  |

9.1. táblázat: `compareTo()` visszatérési értékei

Modernizálhatjuk a `compareTo()` metódus korábbi megvalósítását annak a ténynek a tudatában, hogy a `C# int` adattípusa (amely egyszerű rövidítése a `CLR system.int32` típusnak) megvalósítja az `IComparable` interfészt. Létre-

hozhatjuk a `car.compareTo()` metódusát a következő módon:

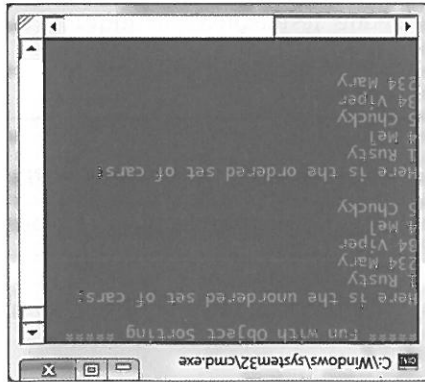
```
int IComparable.CompareTo(object obj)
{
    Car temp = (Car)obj;
    return this.carID.CompareTo(temp.carID);
}
```

A `car` típusunk mindkét esetben tudja, hogy hogyan kell összevetnie magát egy hasonló típusal, így a következő kódot használhatjuk:

```
// A IComparable interfész használata.
static void Main(string[] args)
{
    // Car típusok tömbjének elkészítése.
    ...
    // Az aktuális tömb megjelenítése.
    Console.WriteLine("Here is the unordered set of cars:");
    foreach(Car c in myAutos)
        Console.WriteLine("{0} {1}", c.ID, c.PetName);
    // Az elemek rendezése az IComparable segítségével.
    Array.Sort(myAutos);
    Console.WriteLine();
}
```

```
// A rendezett tömb megjelenítése.
Console.WriteLine("Here is the ordered set of cars:");
foreach(Car c in myAutos)
    Console.WriteLine("{0} {1}", c.ID, c.Petname);
Console.ReadLine();
}
```

A 9.13. ábra mutatja a teszt futtatásának eredményét.



9.13. ábra: Autók összehasonlítása a carID alapján

## Többszörös rendezési sorrend megadása (IComparable)

A car típus fenti verziójában az azonosítótak használjuk a rendezés alapjául. Más esetben a becenevet is vehettük volna a rendezés alapjául (az autók alfabetikus listázásához). Mi a helyzet azonban akkor, ha olyan car típust szeretnénk, amely mind a becenev mind az ID alapján sorba rendezhető? Ekkor meg kell ismerkednünk egy másik szabványos interfésszel, a `System.Collections.IComparable` névtérben definiált `IComparable` interfésszel.

```
// két objektum összehasonlításának általános módja.
interface IComparable
{
    int Compare(object o1, object o2);
}
```

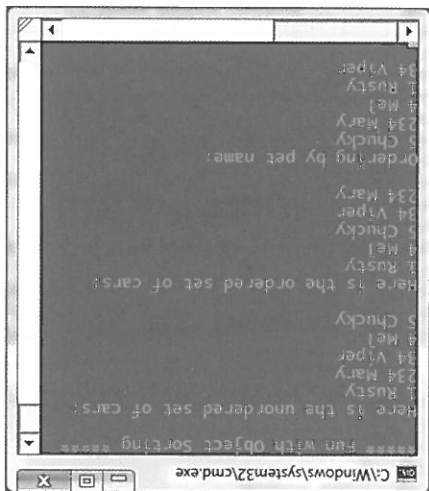
Ellentétben az `IComparable` interfésszel, az `IComparer` interfészt általában nem valósítjuk meg azokon a típusokon, amelyeket rendezni szeretnénk (azaz a caron). Inkább használjuk ezeket az interfészeket segédosztályokon, minden rendezéshöz külön (becenev, auto-ID stb). Például általában a car típus tudja,



hogyan kell a belső ID alapján összehasonlítani magát más car példányokkal. Ennek megfelelően, ha szeretnénk megengedni a car típusok becenev szerinti rendezését is, akkor egy további segédosztályra lesz szükségünk, amely megvalósítja az IComparer interfészt. A forráskód a következő:

```
// A segédosztályt becenev alapján rendezhető a car elemek tömbje.
using System.Collections;

public class PetNameComparer : IComparer
{
    // Az objektumok becenevének elrendezése.
    int IComparer.Compare(object o1, object o2)
    {
        car t1 = (Car)o1;
        car t2 = (Car)o2;
        return String.Compare(t1.PetName, t2.PetName);
    }
}
```



9.14. ábra: Autók rendezése becenev alapján

Az objektum forráskódjában használhatjuk ezt a segédosztályt. A System.Array osztályban vannak túlterhelt sort() metódusok, ezek közül az egyik olyan objektumot vár, amely megvalósítja az IComparer interfészt. A 9.14. ábrán az autók becenev szerinti rendezése látható.

```
static void Main(string[] args)
{
    ...
    // Rendezés becenev szerint.
    Array.Sort(myAutos, new PetNameComparer());
}
```

```
// A rendezett tömb kitérása.
console.writeln("Ordering by pet name:");
foreach(Car c in myAutos)
    console.writeln("{0} {1}", c.ID, c.PetName);
...
}
```

## Egyedi tulajdonságok, egyedi rendezési típusok

Használhatunk egyedi statikus tulajdonságot is azért, hogy az objektum-használo a car típusokat adott adatok szerint rendezni tudja. Tetelezzük fel, hogy a car osztály rendelkezik egy `sortByPetName` nevű statikus írásvédett tulajdonsággal, és ez olyan objektumot ad vissza, amely megvalósítja az `IComparer` interfészt (petNameCompare ebben az esetben):

```
// Támogatjuk az egyedi tulajdonságot, amely
// megfelelő IComparer interfészt adja vissza.
public class Car : IComparable
{
    ...
    // tulajdonság, amely a SortByPetName interfészt adja vissza.
    public static IComparer SortByPetName
    { get { return (Comparer)new PetNameComparer(); } }
}
```

Az objektumhasználo kódja most már rendezhet a becenév alapján egy köz- vetlen tulajdonsággal is, ahelyett, hogy „tudnia kellene” a különálló `PetName-Compare` osztályról:

```
// A becenév szerinti rendezés letisztultabb formája.
Array.Sort(myAutos, Car.SortByPetName);
```

---

**Forráskód** A `ComparableCar` projekt megtalálható a 9. fejezet alkönyvtárában.

---

Elddíg megismerkedtünk az interfésztípusok definiálásával és megvalósítá- sával, valamint hasznosságukkal is. Természetesen minden .NET-névterben találhatók interfészek. A továbbiakban nézzük meg azokat az interfészeket, amelyek visszahívás-mechanizmusokat tesznek lehetővé.

## A Callback interfész

Az interfészek a különböző osztályhierarchiak, névterek és szerelvények közötti polymorfizmus alkalmazásán kívül használhatóak *visszahívási mechanizmusok* megvalósítására is. Ez a technika lehetővé teszi az objektumok számára a kétirányú kommunikációt közös tagok használatával.

---

**Megjegyzés** A .NET-platform egy formális szerkezetet biztosít események létrehozásához (ez meg lehetőséget ad itt bemutatottól). A metódusreferenciákat, az eseményeket és a lambda-kifejezéseket tekinthetjük az objektumok közötti kommunikáció szabványos módjának (lásd 11. fejezet).

---

A visszahívási interfészek bemutatásához fejlesszük tovább a már jól ismert car típusunkat olyan módon, hogy képes legyen értesíteni a hívóját arról, hogy fel fog robbanni (a pillanatnyi sebesség 10 mérföld per órával van a maximális sebesség alatt), és arról, hogy felrobbant (a pillanatnyi sebesség elérte vagy meghaladta a maximális sebességet).

Keddjük egy új callbackinterface nevű paramcsort alkalmazás létrehozásával. Az események küldésének és fogadásának képességet egy új saját interfész fogja elősegíteni: az `IEngineNotification`:

```
// A visszahívási interfész.  
public interface IEngineNotification  
{  
    void AboutToBlow(string msg);  
    void Exploded(string msg);  
}
```

A callback interfészeket gyakran nem azok az objektumok valósítják meg, amelyek közvetlenül érdekeltek az események megkapásában, hanem segédobjektumok, amelyeket *nyelődobjektumnoknak* hívunk. Az események küldője (ebben az esetben a car típus) megfelelő körülmények között hívást intéz a nyelvőhöz. Tételizzzuk fel, hogy a nyelvőszállynak `careventsink` a neve. Ha ez az objektum értesül a különböző bejövő eseményekről, akkor egyszerűen kiírja a bejövő üzenetet a paramcsorra. A nyelvő egy barátságos nevet tartalmazó sztringtagváltozót is kezel (később látni fogjuk, hogy ez miért hasznos):

```
// A car típus nyelvőszálja.  
public class Careventsink : IEngineNotification  
{  
    private string name;  
    public Careventsink(){}  
}
```

```
public CarEventSink(string sinkName)
{
    name = sinkName;
}

public void AboutToBlow(string msg)
{
    Console.WriteLine("{0} reporting: {1}", name, msg);
}

public void Exploded(string msg)
{
    Console.WriteLine("{0} reporting: {1}", name, msg);
}
}
```

Ha már van egy nyelőobjektumunk, amely megvalósítja a visszahívási interfészt, a következő feladatunk az lesz, hogy a car típusnak referenciát adjunk erre a nyelőobjektumra. A car tárolja a referenciát, és szükség esetén hívásokat indít a nyelő felé. Ahhoz, hogy a car objektum tárolhassa a referenciát a nyelőobjektumra, szükségünk lesz egy segédtag hozzáadása a car típushoz, ez lesz az `Advise()` módszer. Hasonlóképpen, ha a hívó szeretne lekapcsolódni az eseményforrásról, akkor másik segédmetódust kell hívnia a car típuson, még hozzá az `Unadvise()` nevűt. Végül, lehetővé téve a felhasználónak több eseményelő regisztrálását (többszörös esemény céljából), a car típus egy `Array<IEventSink>` regisztrálását (többszörös esemény céljából), a car típus egy `Array<IEventSink>` regisztrálását (többszörös esemény céljából), a car típus egy `Array<IEventSink>` regisztrálását (többszörös esemény céljából).

**Megjegyzés** Az `Array<IEventSink>` osztályt a `mscorlib.dll` szerezvény `System.Collections` névtér tartalmazza. Ezt a névtérrel importálni kell a car osztály definícióját tartalmazó fájlban. A gyűjteményeket (és a generikus típusokat) lásd a 10. fejezetben.

```
// A car osztály és a hívó
// az IEventSink interfész segítségével kommunikál.
public class Car
{
    // Kapcsolt nyelőobjektumok készlete.
    Array<IEventSink> sinks = new Array<IEventSink>();

    // Csatlakozás vagy leválás a forrás eseményekről.
    public void Advise(IEventSink sink)
    {
        sinks.Add(sink);
    }

    public void Unadvise(IEventSink sink)
    {
        sinks.Remove(sink);
    }
}
```

Az események tényleges elküldéséhez frissítsük a `car.accelerate()` metódust úgy, hogy járja be a kapcsolatok listáját az `ArrayList` objektumban, és szükség esetén indítsa el a helyes értesítést. Először is hozzunk létre egy `carIsDead` logikai tagváltozót, hogy a motor állapotát megjelentsük:

```
class Car
{
    // Az autó működik vagy lerobbant?
    bool carIsDead;
    ...
}
```

Ezután egészítsük ki az `accelerate()` metódusunkat úgy, hogy használja ezt az új tagváltozót:

```
public void Accelerate(int delta)
{
    // Ha az autó lerobbant, az exploded esemény kiváltása.
    if (carIsDead)
    {
        foreach (IEngineNotification sink in clientsinks)
            sink.Exploded("Sorry, this car is dead...");
    }
    else
    {
        currspeed += delta;

        // Majdnem lerobbant?
        if (10 == (maxspeed - currspeed))
        {
            foreach (IEngineNotification sink in clientsinks)
                sink.AboutToBlow("Careful buddy! Gonna blow!");
        }
        // Még mindig rendben!
        if (currspeed >= maxspeed)
            carIsDead = true;
        else
            Console.WriteLine(">Currspeed = " + currspeed);
    }
}
```

Ezzel az infrastruktúrával már megvalósíthatjuk a `main()` metódusunkat úgy, hogy megkapja ezeket az eseményeket a `car` típustól:

```
// Autót készítettünk, és figyeljük az eseményeket.
static void Main(string[] args)
{
    Console.WriteLine("**** Interfaces as event enablers ****\n");
    Car c1 = new Car("StigBug", 100, 10);
}
```

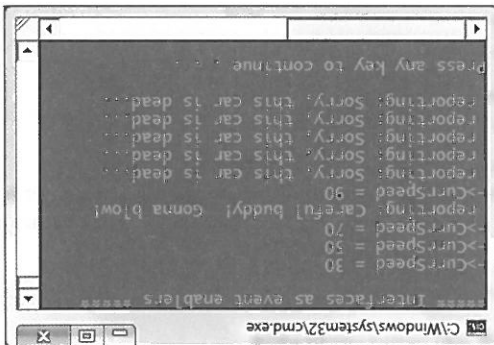
```
// Nyelődobjektum készítése.
careventsink sink = new careventsink();

// car referencia átadása a nyelvnek.
cl.Advise(sink);

// Gyorsítás (ezzel kívájtunk néhány eseményt).
for(int i = 0; i < 10; i++)
    cl.Accelerate(20);

// Leválasztás az esemény forrásáról.
cl.Unadvise(sink);
console.ReadLine();
}
```

A 9.15. ábra mutatja az interfészalapú esemény protokolleredményeit.



9.15. ábra: Eseményértesítés vétele visszahívási interfész használatával

Az `unadvise()` metódus a hívó számára kifejezetten hasznos, mert így szelektíven le tud válni az eseményforrásokról. A `main()` metódusból kilépés előtt hívjuk az `unadvise()` metódust (bár ez technikailag nem feltétlenül szükséges). Tételizzzuk fel, hogy a program két nyelöt szeretne regisztrálni, dinamikusan el távolítani egy adott nyelöt a végrehajtási folyamat során és folytatni a program futását:

```
static void main(string[] args)
{
    console.WriteLine("***** Interfaces as event enablers *****\n");
    car cl = new car("StingBug", 100, 10);

    // két nyelődobjektum készítése.
    console.WriteLine("***** Creating sinks *****");
    careventsink sink = new careventsink("First sink");
    careventsink myOthersink = new careventsink("Other sink");
}
```

## Összefoglalás

Egy interfészt *absztrakt tagok* nevesített gyűjteményeként definiálhatunk. Mi-vel az interfészek nem nyújtanak semmilyen megvalósítási részletet, általa-nosan az interfészeket olyan viselkedésnek tekintjük, amelyek egy adott tí-pus támogathat. Ha két vagy több osztály valószínűleg ugyanazt az inter-fészt, akkor mindegyik típust azonosan kezelhetjük (más néven ez az inter-fészalapú polimorfizmus), még akkor is, ha ezek a típusok egyedi osztályhie-rarchiákban vannak.

---

**Forráskód** A `CarInterface` projekt megtalálható a 9. fejezet alkönyvtárban.

---

A visszahívási interfészek hasznosak lehetnek azért is, mert bármely nyelvben vagy platformon (.NET, J2EE vagy egyéb) használhatóak, ha azok támogatják az interfészalapú programozást. (A 11. fejezetben még részletesebben foglalkozunk né-hány eseménycentrikus technikával, amelyek .NET-platform specifikusak.)

```
// Nyelődobjektmok átadása a Car típusnak.
Console.WriteLine("\n***** Sending 2 sinks into Car *****");
cl.Advise(sink);
cl.Advise(myOthersink);

// Gyorsítás (ezzel kiváltunk néhány eseményt).
Console.WriteLine("\n***** Speeding up *****");
for(int i = 0; i < 10; i++)
    cl.Accelerate(20);

// Az első nyelődobjektm leválasztása az eseményekről.
Console.WriteLine("\n***** Removing first sink *****");
cl.Unadvise(sink);

// Ismét gyorsítás (csak a myOthersink nyelőd hívja).
Console.WriteLine("\n***** Speeding up again *****");
for(int i = 0; i < 10; i++)
    cl.Accelerate(20);

// Másik nyelődobjektm leválasztása az eseményekről.
Console.WriteLine("\n***** Removing second sink *****");
cl.Unadvise(myOthersink);
Console.ReadLine();
}
```

A C# nyelv rendelkezésünkre bocsátja az interface kulcsszót, amellyel új interfészt tudunk létrehozni. Egy típus anyai interfészt támogathat, amelyre csak szükség van, vsszövel elválasztott lista használataival. Továbbá lehetőség van olyan interfészek létrehozására is, amelyek több ösinterfészből származnak.

A saját interfészek létrehozásán kívül, a .NET-könyvtár defniál egy sor szabványos (azaz keretrendszer nyújtotta) interfészt is. Szabadon létrehozhatunk saját típusokat, amelyek megvalósítják ezeket az előre defniált interfészeket, hogy elérhessünk olyan igényelt jellegzetességeket, mint a klónozás, a rendezés és a felsorolás. Végül megnéztük, hogy az interfésztípusok hogyan használhatók két-irányú kommunikációra a memóriában lévő két objektum között.



# Gyűjtemények és generikus típusok

A .NET-platformon belüli leggyyszerűbb tároló a `system.array` típus. Ahogy láttuk, a `C#`-tömbök lehetővé teszik, hogy a `system.array` segítségével előre megadott számú, azonos típusú elemet tároljunk (beleértve a `system.object`ek tömbjét, amely lényegében tetszőleges típusú elemek tömbjét reprezentálja). Sokszor azonban ennél rugalmasabb adatstruktúrára van szükségünk, mint például a dinamikusan növekvő és zsugorodó tároló vagy egy olyan tároló, amely csak a meghatározott felteteleknek megfelelő elemeket tartalmazhatja (pl. csak a megadott össztályból származó osztályokat vagy bizonyos interfészeket megvalósító elemeket). A rugalmas és típusbiztos tárolók készítésének vizsgálatahoz ebben a fejezetben először a `system.collections` névtérét tanulmányozzuk, amely a kezdetektől a .NET-alapozástól könnyvitar része.

A .NET 2.0 kibocsátásakor azonban a `C#` programnyelvet kiegészítették a *generikus típusok* támogatásával. A generikus típusok közül több, naponta használatos a `system.collections.generic` névtérben található. A generikus tárolók felülműlják nem generikus párjukat azzal, hogy nagyobb típusbiztonságot és jobb teljesítményt nyújtanak. Már láttunk példát az össztályon belüli generikus támogatásra, a továbbiakban megmutatjuk, hogyan építhetünk saját generikus tagokat, osztályokat, struktúrákat és interfészeket.

---

**Megjegyzés** Metódusreferencia-típusokat is létre lehet hozni. Ezekkel a következő fejezetben foglalkozunk.

---

## A System.Collections névtér interfészei

A leggyyszerűbb tárolószerkezet a már jól ismert `system.array`. Ez az osztály számos szolgáltatást nyújt (pl. Elemek sorrendjének megfordítása, rendezés, törlés és felsorolás; lásd 4. fejezet). A `system.array` osztálynak azonban számos korláta van; különösen az, hogy nem mériretezi újra magát automatikusan

elemek hozzáadásakor vagy elvételekor. Amikor rugalmasabb tárolóban kell a típusokat tárolni, az egyik lehetőség a `system.collections` névtérben definiált típusok használata.

A `system.collections` névtér számos interfészt definiál (közülük sokat már megvalósítottunk a 9. fejezetben). A `system.collections` névtérben található osztályok többsége megvalósítja ezeket az interfészeket, így hozzáférést biztosítanak a tartalmukhoz. A 10.1. táblázat összefoglalja a lényeges gyűjteménycentrikus interfészeket..

System.Collections

interfész

Használata

|             |                                                                                                                        |
|-------------|------------------------------------------------------------------------------------------------------------------------|
| ICollection | Általános tulajdonságokat definiál minden nem generikus gyűjteménytípushoz (például méret, felsorolás, szálbiztonság). |
|-------------|------------------------------------------------------------------------------------------------------------------------|

|           |                                                |
|-----------|------------------------------------------------|
| IComparer | Lehetővé teszi két objektum összehasonlítását. |
|-----------|------------------------------------------------|

|             |                                                                                                           |
|-------------|-----------------------------------------------------------------------------------------------------------|
| IDictionary | Lehetővé teszi név/érték párok használatával a nem generikus gyűjtemény objektumtartalmának megjelölését. |
|-------------|-----------------------------------------------------------------------------------------------------------|

|                       |                                                                            |
|-----------------------|----------------------------------------------------------------------------|
| IDictionaryEnumerator | IDictionary interfészt támogató típus tartalmának felsorolását biztosítja. |
|-----------------------|----------------------------------------------------------------------------|

|             |                                                         |
|-------------|---------------------------------------------------------|
| IEnumerable | Visszaadja egy adott objektum Enumerator interfészt.    |
| IEnumerator | foreach típusú iterációt tesz lehetővé az alitípusokon. |

|                   |                                                                                        |
|-------------------|----------------------------------------------------------------------------------------|
| IHashCodeProvider | Visszaadja a megvalósító típus hash kódját, testre szabott hash algoritmust használva. |
|-------------------|----------------------------------------------------------------------------------------|

|       |                                                                                                                                                                                                                           |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| IList | Az elemek hozzáadását, eltávolítását és indexelést biztosítja az objektumok listáján. Tagokat definiál a létrehozott gyűjtemény számára, ezek pedig meghatározzák a tároló típusát: csak olvasható, illetve fix méretű-e. |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

10.1. táblázat System.Collections interfészei

Ezek közül az interfészek közül jó néhány összekapcsolódik egy adott interfészhierarchia révén, ezzel szemben a többi egyedülálló egyed. 10.1. ábra a közöttük lévő kapcsolatot illusztrálja (emlékezzünk rá, hogy egy szimpla interfésznek megengedhető, hogy több interfészből származzon).

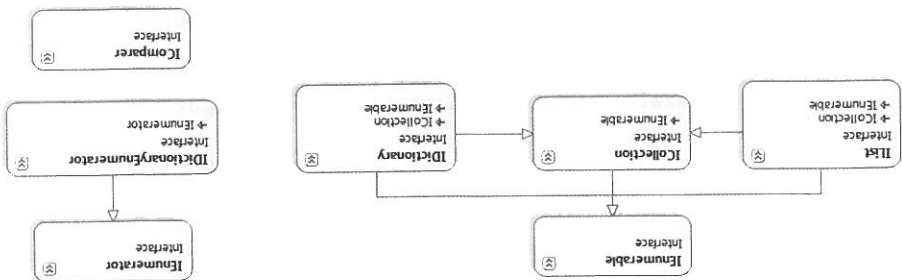
A *szoftár* egy olyan gyűjtemény, amely név/érték párokat tart fenn. Egyedi típusokat készíthetünk, amelyek az `IDictionary` interfészt valósítják meg például `car` típusok tárolása (értékek), amelyet aztán kinyerhetünk valamilyen azonosítóval (az értékhez tartozó nevek). Látható, hogy ezzel a működéssel az `IDictionary` interfész `Keys` (kulcsok) és `Values` (értékek) tulajdonságokat határoz meg, valamint `Add()`, `Remove()` és `Contains()` metódusokat. Hozzájuthatunk az egyes elemekhez a típusindexelő segítségével. Ez a szerkezet a tömbtípusú szintaxissal teszi lehetővé az elemek használatát. A formális definíció a következő:

```
public interface ICollection : IEnumerable
{
    int Count { get; }
    bool IsSynchronized { get; }
    object SyncRoot { get; }
    void CopyTo(Array array, int index);
}
```

Az `ICollection` a `System.Collections` névtér leggyakoribb olyan interfésze, amely a gyűjteménytípus által támogatott viselkedést definiálja. Röviden: ez az interfész olyan tagok kisebb gyűjteményét biztosítja, amely lehetővé teszi, hogy meghatározzuk: a) az elemek számát a tárolóban, b) a tároló szálbiztos-ságát, valamint c) a `System.Array` típusba való másolás képességét. Hivatalsan az `ICollection` interfészt a következőképpen definiáljuk (az `ICollection` kiterjeszti az `IEnumerable` interfészt):

## Az `ICollection` interfész

10.1. ábra A `System.Collections` interfészhierarchia



A `System.Collections` névtér interfészei

## Az IDictionaryEnumerator interfész

```
public interface IDictionary : ICollection, IEnumerable
{
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    // Típus indexelő; részletekért lásd a 12. fejezetet.
    object this[object key] { get; set; }
    ICollection Keys { get; }
    ICollection Values { get; }
    void Add(object key, object value);
    void Clear();
    bool Contains(object key);
    IDictionaryEnumerator GetEnumerator();
    void Remove(object key);
}
```

Az előző fejezetben láttuk, hogy az `IDictionary.GetEnumerator()` metódus hívása `IDictionaryEnumerator` típusú példánnyal tér vissza. Az `IDictionaryEnumerator` interfész egy olyan erősen típusos felsoroló, amely az `IEnumerator` interfész kiterjesztése, a következő hozzáadott funkcionálisással:

```
public interface IDictionaryEnumerator : IEnumerator
{
    DictionaryEntry Entry { get; }
    object Key { get; }
    object Value { get; }
}
```

Az `IDictionaryEnumerator` interfész a szótárban szereplő tételek felsorolását teszi lehetővé az általánosított `Entry` tulajdonsággal, amely `system.Collections.DictionaryEntry` típusú osztállyal tér vissza. Továbbá a név/érték párokat a `Key` (kulcs) és a `Value` (érték) tulajdonságok felhasználásával képesek vagyunk végigjárni.

## Az IList interfész

Az utolsó központi interfész a `system.Collections` névtérben az `IList`, amely tételek tárolókba illesztésének, eltávolításának és indexelésének képességét biztosítja:

# A System.Collections nevter osztálytípusai

Az interfészek önmagukban nem igazán hasznosak egészen addig, amíg nem implementáljuk őket egy adott osztállyal vagy struktúrával. A 10.2. táblázat összefoglalja a System.Collections nevterben található központi osztályokat és az általuk támogatott fontosabb interfészeket.

| System.Collections | Használat | A megvalósított fontosabb interfészek |
|--------------------|-----------|---------------------------------------|
|--------------------|-----------|---------------------------------------|

|           |                                                                                                                                                                                 |                                                                       |
|-----------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------|
| ArrayList | Objektumok dinamikusan méretezett tömbjét reprezentálja.                                                                                                                        | ICollection, IList, IEnumerable és IEnumerable<T>                     |
| HashTable | Numerikus kulcs által definiált objektumok gyűjteményét reprezentálja. A hashtáblában tárolt egyedi típusoknak mindig felül kell írniuk a System.Object.GetHashCode() metódust. | ICollection, IDictionary, IDictionary<TKey, TValue> és IEnumerable<T> |
| Queue     | Szabványos FIFO-sort reprezentál.                                                                                                                                               | ICollection, ICollection<T> és IEnumerable<T>                         |

A System.Collections nevter osztálytípusai

```
public interface IList :
    ICollection, IEnumerable
{
    bool IsFixedSize { get; }
    bool IsReadOnly { get; }
    object this[ int index ] { get; set; }
    int Add(object value);
    void Clear();
    bool Contains(object value);
    int IndexOf(object value);
    void Insert(int index, object value);
    void Remove(object value);
    void RemoveAt(int index);
}
```

| System.Collections osztály | Használata | A megvalósított fontosabb interfészek |
|----------------------------|------------|---------------------------------------|
|                            |            |                                       |

|            |                                                                                                                        |
|------------|------------------------------------------------------------------------------------------------------------------------|
| SortedList | A szótárhoz hasonló tároló, am az elemekhez is hozzáférhetünk IDictionary, ICollection, EnumerableView és ICollection. |
| Stack      | Push, pop és peek műveleteket támogat LIFO-sort valóst meg. ICollection, EnumerableView                                |

10.2. táblázat: System.Collections névtér osztályai

A fenti fontosabb típusok mellett a System.Collections névtér tartalmaz néhány kevésbé jelentős osztályt is (legálábbis a mindennapi használhatóságukat tekintve): ezek a BitArray, a CaseInsensitiveComparer és a CaseInsensitiveHashCodeProvider. Továbbá ez a névtér absztrakt ösztályok kis gyűjteményével is rendelkezik (ICollectionBase, ReadOnlyCollectionBase és DictionaryBase), amelyek felhasználhatóak erősen típusos tárolók készítéséhez.

A System.Collections névtér típusai közös funkciók megosztására törekednek (ez az interfészalapú programozás lényege). Így nem fogjuk minden gyűjteményosztály tagjait felsorolni, hanem három gyakori gyűjteménytípust mutatunk be: az ArrayList, a Queue és a Stack gyűjteményeket.

Ezeknek a típusoknak a működésén keresztül könnyedén megértjük a többi gyűjteményosztályt is (pl. a Hashtable-t). Az összes típus teljes dokumentációja megtalálható a .NET Framework 3.5 leírásában.

## Az ArrayList típus használata

A gyűjteménytípusok szemléltetésére hozzunk létre egy új CollectionTypes nevű parancssorialkalmazás-projektet. Az ArrayListben egyszerű car objektumokból álló csoportot tartunk fenn, amelyet következőképpen definiálunk:

```
class Car
{
    // Az egyszerűség kedvéért publikus.
    public string PetName;
    public int Speed;
}
```

```
// konstruktorok.
public Car(string name, int currentSpeed)
{ PetName = name; Speed = currentSpeed; }
}
```

Módosítsuk a projekt kezdeti C#-fájlját a System.Collections névtér használatának megadásával:

```
using System.Collections;
```

Az Arraylist valójában használt olyan típus a System.Collections névtérben, amely lehetővé teszi a tároló dinamikus átméretezését. Az alapjainak szemléltetésére vizsgáljuk meg a következő Program osztályba tartozó metódust, amelynek hatására az arraylist megváltoztatja a car objektumok gyűjteményét:

```
static void ArraylistTest()
{
    Console.WriteLine("\n=> Arraylist Test:\n");
    // Arraylist létrehozása és feltöltése kezdőértékekkel.
    Arraylist carArraylist = new Arraylist();
    carArraylist.AddRange(new Car[] { new Car("Fred", 90, 10),
    new Car("Mary", 100, 50), new Car("MB", 190, 11)});
    // Az Arraylist elemszámának kitírása.
    Console.WriteLine("Items in carArraylist: {0}", carArraylist.Count);
    // Aktuális értékek kitírása.
    foreach(Car c in carArraylist)
        Console.WriteLine("Car pet name: {0}", c.PetName);
    // új elem beszúrása.
    Console.WriteLine(">Inserting new Car.");
    carArraylist.Insert(2, new Car("TheNewCar", 0, 12));
    Console.WriteLine("Items in carArraylist: {0}", carArraylist.Count);
    // objektum tömb kiolvasása és kitírása az Arraylist-ből.
    object[] arrayOfCars = carArraylist.ToArray();
    for(int i = 0; i < arrayOfCars.Length; i++)
    {
        Console.WriteLine("Car pet name: {0}",
            ((Car)arrayOfCars[i]).PetName);
    }
}
```

Az `addRange()` módszer felhasználásával benépesítjük az `ArrayList`-et `car` típusú elemekkel (ez az `add()` módszer többszöri meghívásával egyenértékű). Amint kiterítjük a gyűjteményben található tételek számát (és az összes tételt felsoroljuk, hogy megkapjuk az autók nevét), meghívhatjuk az `insert()` metódust. Az `insert()` segítségével tudunk egy új tételt megfelelő indexszel az `ArrayList`-be beszúrni.

Végül a `toArray()` módszer hívása az eredeti `ArrayList`-t tartalmazó `ArrayList` `system.Object` típusú tömbjével tér vissza. Ebből a tömbből az indexelő szintaxis felhasználásával még egyszer végiglépkedünk az elemeken.

## A Queue típus használata

A sorok olyan tárolók, amelyek gondoskodnak arról, hogy a tételek hozzá-  
férhetők legyenek az „elsőnek be, elsőnek ki”, azaz a FIFO módszerrel. Egész  
nap ki vagyunk téve a sorban állásnak: sorban állunk a banknál, a mozinál és  
reggel a kávéház előtt. Ha olyan modellt használunk, amelyben a tételleket az  
„elsőször jött, elsőször lesz kiszolgálva” elv alapján kezeljük, akkor a `System.  
Collections.Queue` osztály a megfelelő választás. A támogatott interfészek al-  
tal biztosított funkcionálitásán túl a `Queue` osztály kulcsfontosságú tagjait a

| System.Collections.Queue | Használata | tagjai |
|--------------------------|------------|--------|
|--------------------------|------------|--------|

|           |                                                                                         |
|-----------|-----------------------------------------------------------------------------------------|
| Dequeue() | El távolítja és visszaadja a sor elején található objektumot.                           |
| Enqueue() | Hozzáad egy objektumot a sor végéhez.                                                   |
| Peek()    | A sor elején található objektummal tér vissza, anélkül, hogy azt eltávolítaná a sorból. |

10.3. táblázat: Queue típus tagjai

A metódusok szemléltetésére még egyszer módosítsuk az autós példánkat, és hozzunk létre objektumot, amely a mosásra váró autók sorát szimulálja. Először is vegyük a következő statikus segédfüggvényt:

```
static void WashCar(Car c)
{
    Console.WriteLine("Cleaning {0}", c.PetName);
}
```



Ezután vegyük a következő segédmetódust, amely meghívja a `washcar()` metódust:

```
static void QueueTest()
{
    Console.WriteLine("\n=> Queue Test:\n");
    // Sor készítése három elemmel.
    Queue washq = new Queue();
    washq.Enqueue(new Car("Firstcar", 10));
    washq.Enqueue(new Car("Secondcar", 20));
    washq.Enqueue(new Car("Thirdcar", 30));
    // Az első autó megtekintése a sorban.
    Console.WriteLine("First in q is {0}",
        ((Car)washq.Peek()).PetName);
    // Elemek eltávolítása a sorból.
    washcar((Car)washq.Dequeue());
    washcar((Car)washq.Dequeue());
    washcar((Car)washq.Dequeue());
    // Ismét megpróbálunk elemeket eltávolítani a sorból?
    try
    {
        washcar((Car)washq.Dequeue());
    }
    catch (Exception e)
    {
        Console.WriteLine("Error! {0}", e.Message);
    }
}
```

Három tételt helyezzünk a `queue` típusba az `enqueue()` metódussal. A `peek()` meghívásával megnézhetjük a `queue` első elemét (anélkül, hogy eltávolítsuk), amely ebben az esetben a `firstcar` nevű objektum. Végül a `dequeue()` meghívásával eltávolítjuk az elemet a sorból, és elküldjük a `washcar()` feloldozó segédfüggvénynek. Ha egy üres sorból próbálunk elemet kitörölni, akkor futásidejű kivételt dobódik.

## A Stack típus használata

A `System.Collections.Stack` típus reprezentálja az „utolsónak be, elsőnek ki” (LIFO) szabállyal fenntartott gyűjteményt. A `stack` egy `push()` és egy `pop()` tagot definiál (belehelyez, illetve kivessz tetteket a veremből). A következő verem az alapvető használatot példázza:



|                                |           |
|--------------------------------|-----------|
| System.Collections.Specialized | Használat |
|--------------------------------|-----------|

BitVector32 Ez egy olyan egyszerű struktúra, amely logikai értékeket tárol, valamint kis egész számokat 32 biten.

CollectionUtilities Olyan gyűjteményt hoz létre, amely nem tesz különbséget a sztringekben a nagy- és a kisbetűk között.

HybridDictionary A ListDictionary osztály használatával az IDictionary interfészt valósítja meg kis gyűjteményeknél, majd a ListDictionary interfészt felcseréli a Hashtable osztályra, amikor a gyűjtemény növekszik.

ListDictionary Egyszerezsen láncolt lista használatával megvalósítja az IDictionary interfészt. Legfeljebb tíz elemmel rendelkező tárológyűjteményekhez ajánlott.

NameValueCollection Sztringkulcsok és sztringértékek párijának gyűjteményét reprezentálja, amelyhez a kulcs-csal vagy indexszel lehet hozzáférni.

StringCollection Sztringek gyűjteményét reprezentálja. StringDictionary Hasháblát valósít meg objektum helyett erősen típusos sztringkulccsal.

StringEnumerator Egyszerű iterációt támogat a StringCollection osztályon.

10.4. táblázat: A System.Collections.Specialized névtér típusai

A System.Collections névtér (és a System.Collections.Specialized névtér) belüli alapvető gyűjteménytípusokat, mint ahogy régi típusok, nem szoktuk felhasználni új projekteknél készítéséhez. .NET 2.0 vagy későbbi változataiban. Nem azért, mert ezek a típusok valamilyen szempontból veszélyesek lennének, hanem a teljesítmény hatásfoka és a típusbiztonság hiánya az ok. Az új projekteknél figyelmen kívül kellene hagyniuk ezeket a ránk maradt tárolótípusokat, és a System.Collections névtér rokon típusait kellene támogatniuk. A generikus típusok tanulmányozása előtt célszerű pontosítaniunk, hogy ezekkel a típusokkal milyen problémákat fogunk tudni megoldani, ehhez pedig meg kell vizsgálnunk a be-, illetve a kidobozolás szerepét.

## A bedobozolás, kidobozolás és a System.Object kapcsolata

A .NET-platform két nagyobb adattípuscsoportot támogat, az *értéktípusokat* és a *referenciátípusokat* (lásd 4. fejezet). A .NET két fő típuskategóriát definiál, néha az egyik kategóriába tartozó változót a másikba tartozóként kell felhasználnunk. Hogy ezt megteheszünk, a C# egy nagyon egyszerű működési elvet biztosít: értéktípust referenciátípussá alakít, ezt a működést nevezzük *dobozolásnak*. Ehhez hozzunk létre egy `short` típusú változót:

```
// short típusú változó létrehozása.
short s = 25;
```

Ha az alkalmazás futása alatt ezt a változót referenciátípusként szeretnénk kifejezni, akkor a következőképpen dobozoljuk az értéket:

```
// Az érték dobozolása objektum referenciába.
object objShort = s;
```

A dobozolás tehát olyan folyamat, amelynek során az értéktípust megfelelő referenciátípussá alakítjuk, miközben a változó a `system.object` típusban tárolódik. Amikor egy értéket bedobozolunk, a CLR egy új objektumot határoz meg a heapen, és átmásolja az értéktípus értékét (a fenti esetben ez 25) a példányba. Az újonnan létrehozott objektumhoz egy referencia tér vissza. Ezzel a technikával a .NET-fejlesztőknek nem kell burkolódosszályokat létrehozniuk a veremadatok mint heapen tárolt objektumok ideiglenes kezeléséhez. Az ellentétes művelet ugyancsak elérhető a *kidobozolásnak* köszönhetően. A kidobozolás az a folyamat, amely az értéket az objektumreferenciából megfelelő értéktípussá alakítja vissza a veremben. A kidobozolás az átvett adattpus ellenőrzésével kezdődik, és ha ez azonos a bedobozolt típussal, akkor az érték a helyi veremalapú változóba másolódik. A következő kidobozolási műveletek sikeresen futnak, ha az `objShort` alapjául szolgáló típus valóban `short`:

```
// A referencia kidobozolása a megfelelő short értékre.
short anotherShort = (short)objShort;
```

Fontos, hogy a megfelelő adattípusbba dobozoljuk ki. Ennek megfelelően a következő kidobozolási logika `InvalidCastException` kivételt generál:

```
// Erőnytelten kidobozolás.
static void Main(string[] args)
{
    short s = 25;
    object objShort = s;

    try
    {
        // A dobozban található érték NEM int, hanem short!
        int i = (int)objShort;
    }
    catch (InvalidCastException e)
    {
        Console.WriteLine("Oops!\n{0} ", e.ToString());
    }
}
```

Elso ranezésre a ki- és a bedobozolás hasonlíthat egy jelentéktelen nyelvi sajátossághoz, amely inkább elméleti, mint gyakorlati jellegű. A valóságban a dobozolás azonban nagyon hasznos, segítségével minden típus system.objectként kezelhető, ugyanis a CLR gondoskodik helyettünk a memóriával kapcsolatos részletekről.

A módszer gyakorlati használatának bemutatásához vegyünk egy system.Collections.ArrayList-et, amely numerikus (veremfoglati) adatot tárol. Az ArrayList-eket tagjai típusok system.object típusok tárolását hivatottak kezelni:

```
public class System.Collections.ArrayList : object,
    System.Collections.IList,
    System.Collections.ICollection,
    System.Collections.IEnumerable,
    ICollection
{
    ...
    public virtual int Add(object value);
    public virtual void Insert(int index, object value);
    public virtual void Remove(object obj);
    public virtual object this[int index] {get; set; }
}
```

A programozóknak nem kell manuálisan objektumba csomagolniuk a veremalapú egészeket, mert a futtatórendszer ezt automatikusan végrehajtja a dobozolás segítségével:

```
static void Main(string[] args)
{
    // Az értéktípusokat a rendszer automatikusan dobozolja,
    // ha átadja az objektumot kérő tagnak.
    ArrayList myInts = new ArrayList();
    myInts.Add(10);
    Console.WriteLine();
}
```

Ha ki szeretnénk nyerni a típusindexelő segítségével ezt az értéket az array-list objektumból, akkor a kasztolás segítségével ki kell dobozolnunk a heapen lefoglalt objektumot egy vermen lefoglalt egészre:

```
static void Main(string[] args)
{
```

```
    ...
    // Az értéket kidobozoltuk.
    int i = (int)myInts[0];
```

```
    // Ujradobozoltuk, mivel a WriteLine() objektumtípusokat követeli!
    Console.WriteLine("Value of your int: {0}", i)
    Console.ReadLine();
}
```

Amikor a C#-fordító átalakítja a bedobozolást köztes nyelvi kóddá, akkor belül a box műveletekódot használja. Hasonlóképpen a kidobozolás művelete CIL-un-box műveletre alakul át. Nézzük meg a kapcsolódó köztes nyelvi kódot az előző Main() metódussal (amelyet megtekinthetünk az `ildasm.exe` segítségével):

```
.method private hidebysig static void Main(string[] args)
{
    ...
    box [mscorlib]System.Int32
        callvirt instance int32
            [mscorlib]System.Collections.ArrayList::Add(object)
    pop
    ldstr "Value of your int: {0}"
    ldc.i4.0
    ldc.i4.0
    callvirt instance object [mscorlib]
        System.Collections.ArrayList::getItem(int32)
    unbox [mscorlib]System.Int32
    ldind.i4
    box [mscorlib]System.Int32
    call void [mscorlib]System.Console::WriteLine(string, object)
    ...
}
```

A veremfoglalt `System.Int32` dobozolóása az `ArrayList.Add()` metódus meghívását megelőzően megtörténik, így előáll a megfelelő `System.Object`. A `System.Object`-et visszadobozoljuk a `System.Int32`-be, amikor a típusindexelő használatalával az `ArrayList`-ből lekérdezzük az elemet (ez utóbbit leképezzük egy rejtett `get_Item()` metódusba), és csak akkor *dobozoljuk be újra*, amikor átadjuk a `console.WriteLine()` metódusnak. Ebben a metódusban `System.Object` típussal dolgozunk.

## A dobozólással járó probléma

A programozó szemszögéből a dobozolás nagyon kényelmes megoldás, ez az egyszerűsített megközelítés azonban a `stack` és a `heap` közötti memóriamegosztás miatt a teljesítmény csökkensenéhez (mind a futási sebesség, mind a kód-méret tekintetében) és a típusbiztonság hiányához vezet. A teljesítmény csökkenésére nézzük meg a dobozolásakor leggyakrabban alkalmazott lépéseket egy egyszerű esetben:

1. Az új objektumot le kell foglalni a futógyélt `heap`-ben.
2. A veremalapú adat értéket át kell másolni az új objektum helyére.
3. Amikor kidobozolunk, a `heap`alapú objektumban tárolt értéket vissza kell másolni a verembe.
4. Az éppem nem használt objektum a `heap`-en (végül) a személggyűjtés je-

lölthe lesz.

A jelenlegi `Main()` metódus nem okoz szűk keresztmetszetet a teljesítményben, de természetesen érződik a kihatása akkor, ha egy `ArrayList`-t több ezer olyan egészet tartalmaz, amelyeket rendszeresen változtatunk a programunkban. Vizsgáljuk meg a típusbiztonság hiányát tekintve a kidobozolás műveletét. Egy `C#` által használt értékek kidobozolásához a *kasztozás szintaxisát* használjuk. A *kasztozás* síkkessege azonban csak *futásidőben* derül ki. Ennek következtében, ha egy értéket rossz adattípussá próbálunk kidobozolni, akkor `InvalidCastException`ot kapunk:

```
static void Main(string[] args)
{
    ArrayList myInts = new ArrayList();
    myInts.Add(10);
}
```

```
// Futásidő kiértékelés
short i = (int)myInts[0];

// Újradobozoltuk, mivel a WriteLine() objektumtípusokat követeli
Console.WriteLine("Value of your int: {0}", i);
Console.ReadLine();
}
```

Ideális esetben a C#-fordító képes lenne megoldani ezeket az illegális kido-bozólási műveleteket fordítási időben. *Valóban* ideális esetben el tudnánk tá-rolni az értékítételeket egy tárolóban, anélkül hogy dobozolásra lenne szük-ség. A generikus típusok jelentik a választ ezekre a kihívásokra.

## A típusbiztonság és az erősen típusos gyűjtemények

Egy generikus típusok nélküli világban a `System.Collections` névtér típusait használhatjuk tetszőleges elemek tárolására, ezeket ugyanis úgy tervezték, hogy `System.Object`-ekkel dolgozzanak:

```
static void Main(string[] args)
{
    // Az ArrayList bármít tárolhat.
    ArrayList a1 = new ArrayList();
    a1.Add(true);
    a1.Add(new Car());
    a1.Add(66);
    a1.Add(3.14);
}
```

Esetenként rendkívül rugalmas tárolókra lesz szükségünk, amelyek szó sze-rint mindennek a tárolására alkalmasak. A legtöbbszór azonban *típusbiztos tárolót* szeretnénk, amely csak bizonyos típusú adatokkal működik. Szüksé-günk lehet például egy olyan tárolóra, amely csak adatbázis-kapcsolatokat, bittérképeket, `IntPtr`-kompatibilis objektumokat vagy hasonlóan specifikus dolgokat képes tárolni.



## Egyedi gyűjtemény készítése

A .NET 2.0-beli generikus típusok bevezetése előtt a programozók a típusbiztonság érdekében egyedi, erősen típusos gyűjteményeket készítettek. A problema érzelkeitésére hoztunk létre egy customNonGenericCollection nevű parancssorialkalmazás-projektet. Mindenképpen importálnuk a System.Collections nevűteret. Tegyük fel, hogy létre akarunk hozni egy egyedi gyűjteményt, amely csak Person típusú objektumot tartalmazhat:

```

public class Person
{
    // Az egyszerűség kedvéért publicus.
    public int Age;
    public string FirstName, LastName;

    public Person()
    {
        public Person(string firstName, string lastName, int age)
        {
            Age = age;
            FirstName = firstName;
            LastName = lastName;
        }

        public override string ToString()
        {
            return string.Format("Name: {0} {1}, Age: {2}",
                FirstName, LastName, Age);
        }
    }
}

```

Amnak érdekében, hogy létrehozzunk egy `Person` gyűjteményt, definiálnunk kell egy `System.Collections.Generic.Array<T>` tagvalitozót a `PeopleCollection` osztálynak belül, és minden tagjának át kell alakítanunk a működését úgy, hogy az erősen típusos `Person` objektumot használja a `System.Object` típus helyett:

```
public class PeopleCollection : IEnumerable
{
    private ArrayList arpeople = new ArrayList();

    public PeopleCollection(){}

    // kátszólás a hívóhoz.
    public Person GetPerson(int pos)
    { return (Person)arpeople[pos]; }

    // csak Person típusok beszúrása.
    public void AddPerson(Person p)
    { arpeople.Add(p); }
}
```

```
public void ClearPeople()
{
    arPeople.Clear();
}

public int Count
{
    get { return arPeople.Count; }
}

// Foreach felsorolás támogatása.
IEnumerator IEnumerable.Getenumerator()
{
    return arPeople.GetEnumerator();
}
```

A `ICollection` típus megvalósítja az `IEnumerable` interfészt, hogy lehetővé tegye a `foreach`-szerű iterációt a tartalmazott elemeken. A `get Person()` és az `add Person()` metódusunkat úgy tervezzük, hogy csak a `Person` objektumon működjön (nem bitérképeken, sztringeken, adatbázis-kapcsolatokon vagy más tételéken). Ezekkel a megvalósított típusokkal biztosítjuk a típusbiztonságot, a C#-fordító képes lesz meghatározni bármelyik nem kompatibilis típus beillesztésére tett kísérletet:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Custom Person Collection *****\n");
    ICollection myPeople = new PeopleCollection();
    myPeople.AddPerson(new Person("Homer", "Simpson", 40));
    myPeople.AddPerson(new Person("Marge", "Simpson", 38));
    myPeople.AddPerson(new Person("Lisa", "Simpson", 9));
    myPeople.AddPerson(new Person("Bart", "Simpson", 7));
    myPeople.AddPerson(new Person("Maggie", "Simpson", 2));
    // Ez fordítási hiba lenne!
    myPeople.AddPerson(new Car());
    foreach (Person p in myPeople)
        Console.WriteLine(p);
    Console.ReadLine();
}
```

Bár az egyedi gyűjtemények gondoskodnak a típusbiztonságról, ám ezért létre kell hoznunk egy (majdnem teljesen egyező) egyedi gyűjteményt minden egyes olyan típusnak, amelyet tárolni szeretnénk. Így ha szükségünk van egy olyan egyedi gyűjteményre, amely csak a `Car` alaposztályból származtatott osztályokon képes működni, akkor egy nagyon hasonló típust kell készítenünk:

```
public class CarCollection : IEnumerable
{
    private ArrayList arCars = new ArrayList();
    public CarCollection() {}
}
```

```
// Kasztolás a hívóhoz.
public Car getCar(int pos)
{ return (Car) arcars[pos]; }

// Csak car típusok beszűrése.
public void AddCar(Car c)
{ arcars.Add(c); }

public void ClearArcars()
{ arcars.Clear(); }

public int Count
{ get { return arcars.Count; } }

// Foreach felsorolás támozgatása.
IEnumerator IEnumerable.Getenumerator()
{ return arcars.GetEnumerator(); }
}
```

Több, erősen típusos gyűjtemény létrehozása a különböző típusok kezelése miatt nemcsak munkaiágényes, de fenntartása sem túl egyszerű. A generikus gyűjtemények lehetővé teszik számunkra a tárolt típus meghatározásának késleltetését a létrehozás idejéig. A szintaktikai részletek helyett egyelőre nézzük meg a következő forráskódot, amely a `System.Collections.Generic.List<T>` nevű generikus osztály segítségével létrehoz két típusbiztos tárolóobjektumot:

```
static void Main(string[] args)
{
    // Generikus List típus csak személyeket tárol.
    List<Person> morePeople = new List<Person>();
    morePeople.Add(new Person());

    // Generikus List típus csak autót tárol.
    List<Car> moreCars = new List<Car>();
    moreCars.Add(new Person());
}
```

## Az erősen típusos gyűjtemények és a dobozolás

A .NET-alapoztatálykönyvtárban léteznek erősen típusos gyűjtemények, amelyek nagyon hasznos programozási szerkezetek. Az egyedi tárolók kevésbé jelentenek megoldást a dobozolás hátrányaira. Ha létrehozunk egy `IntCollection` nevű egyedi gyűjteményt, amely csak a `System.Int32` adat típusokon működik, az adatok tárolására akkor is valamilyen általános tárolót kellene használnunk (`System.Array`, `System.Collections.ArrayList` stb.):

```

public class IntCollection : IEnumerable
{
    private ArrayList aints = new ArrayList();
    public IntCollection() { }

    // Kidobozolás a hívónak.
    public int GetInt(int pos)
    { return (int)aints[pos]; }

    // Bedobozolási művelet!
    public void AddInt(int i)
    { aints.Add(i); }

    public void Clear()
    { aints.Clear(); }

    public int Count
    { get { return aints.Count; } }

    IEnumerator IEnumerable.GetEnumerator()
    { return aints.GetEnumerator(); }
}

```

Függeljenül attól, hogy milyen típust választottunk az egészek tárolására, (System.Array, System.Collections.ArrayList stb.), nem szabadulunk meg a dobbozolás dilemmájától, ha nem generikus tárolt használunk. Ismét a generikus típusok segítenek ki minket. A következő forráskód a System.Collections.Generic.List<T> típust használja egy integer típusokat tartalmazó tároló létrehozásához, amely *nincs* kitéve a ki- és a bedobozolás hátrányainak, amikor beillesztjük vagy lekérdezzük az értéket:

```

static void Main(string[] args)
{
    // Nincs bedobozolás!
    List<int> myints = new List<int>();
    myints.Add(5);
    // Nincs kidobozolás!
    int i = myints[0];
}

```

Az előző `Main()` metódus a következő köztes nyelvi kódban (figyeljük meg a `box` és az `unbox` műveleti kódok hiányát) a következőképpen néz ki:

```

.method private hidebysig static void Main(string[] args)
{
    .entrypoint
    .maxstack 2
}

```

```
.locals init ([0] class
[mscorlib]System.Collections.Generic, [list`1<int32>
myints, [1] int32 i)
newobj instance void class
[mscorlib]System.Collections.Generic, [list`1<int32>::.ctor()
stloc.0
ldloc.0
ldc.i4.5
callvirt instance void class [mscorlib]
System.Collections.Generic, [list`1<int32>::Add(i0)
nop
ldloc.0
ldc.i4.0
callvirt instance i0 class [mscorlib]
System.Collections.Generic, [list`1<int32>::get_Item(int32)
stloc.1
ret
}
```

Összefoglalva, a generikus tárolók a következők előnyöket nyújtják a nem generikusokkal szemben:

- A generikus típusok jobb teljesítményt nyújtanak azzal, hogy nem eredményeznek ki- vagy bedobozolási hátrányokat.
- A generikus típusok típusbiztonsága jobb azáltal, hogy csak olyan a típusú változókat tárolnak, amelyeket specifikáltunk.
- A generikus típusok nagyban csökkentik az egyedi gyűjteménytípus készítésének szükségességét, mivel az alaposztály számos előre gyártott tárolót biztosít.

---

**Forráskód** A CustomGenericCollection projekt megtalálható a 10. fejezet alkönyvtárában.

---

## A System.Collections.Generic névtér

Szetszórtan minden .NET-alaposztálykönyvtárban található generikus típusokat; a System.Collections.Generic névtér azonban tele van ezekkel (ahogy a nevében is látható). A nem generikus ellenpárjukhoz hasonlóan (System.Collections.Generic, a System.Collections.Generic névtér számos osztályt és interfésztípusot tartalmaz, amely lehetővé teszi, hogy az elemeket különböző tárolókban tároljuk. A generikus interfész utánozza a megfelelő nem generikus típusokat a System.Collections.Generic névtérben:

**Megjegyzés** A generikus típusok egyezményesen specifikálják a helyőrzőjüket a közös név használatával. Bármely betű (vagy szó) megfelelően, de tipikusan a T-t használjuk a típus jelölésére, a kulcsokhoz TKey-t, az értékekhez pedig a TValue-t.

Generikus osztály

Nem generikus pár a System.Collectionben

Jelentése

A System.Collections.Generic névtér ugyancsak számos osztályt definiál, amelyek ezekből a kulcsinterfészekből sokat megvalósítanak. A 10.5. táblázat rövid összefoglalást ad a névtér főbb osztálytípusairól, az általuk megvalósított interfészekről és a System.Collections névtér megegyező típusairól.

|                                |                |                                                                    |
|--------------------------------|----------------|--------------------------------------------------------------------|
| ICollection<T>                 | CollectionBase | A generikus gyűjtemény alapjai                                     |
| Comparer<T>                    | Comparer       | Két generikus objektum egyenlőségét vizsgálja                      |
| Dictionary<TKey, TValue>       | HashTable      | Kulcs/érték párok generikus gyűjteménye                            |
| List<T>                        | ArrayList      | Elemek dinamikususan átméretezhető listája                         |
| Queue<T>                       | Queue          | “Előnek be, előnek ki” FIFO-lista generikus megvalósítása          |
| SortedDictionary<TKey, TValue> | SortedList     | Rendezett kulcs/érték párok gyűjteményének generikus megvalósítása |

A nem generikus osztályokhoz hasonlóan a generikus osztályokat is new kulcs-  
szóval és az esetlegesen szükséges konstruktorparaméterek megadásával hoz-  
zuk létre. Továbbá specifikálnunk kell, hogy mely típus(ok) helyettesítsék a  
generikus típus által definiált típusparamétereket. A system.collections.gene-  
ric.list<T> osztály például megköveteli, hogy megadjunk egy olyan értéket,  
amely a list<T> által felhasznált elem típusát írja le. Így, ha három list<T> ob-  
jektumot akarunk létrehozni annak érdekében, hogy egésszekeet, valamint  
sportscar és Person objektumokat tároljunk, akkor a következőt írhatjuk:

### A List<T> típus vizsgálata

A 10.5. táblázatban látható: sok generikus gyűjteményosztálynak van nem  
generikus párja a system.collections.névterben (közülük soknak azonos a ne-  
ve). Mivel a generikus osztályok eléggé hasonlítanak a nem generikus típus-  
sukhoz, ha megértjük, hogy egy generikus elem hogyan működik egy adott tá-  
rolóval, a többi már magától értetődő lesz. A generikus típusokkal dolgozó fo-  
lyamatok működését a list<T> osztály használatával illusztráljuk. Ha a sys-  
tem.collections.generic.névter többi tagjáról szeretnénk további részleteket  
megtudni, akkor a .NET Framework 3.5 dokumentációjában nézhetünk utána.

A csú elemet próbálunk megagadni a tárolóból; és így tovább.  
tálja; a keyNotFoundException kivétel dobódik akkor, amikor egy nem létező kul-  
típus például a generikus linkedlist<T> osztályon belüli csomópontot reprezen-  
tál, aminek a specifikált tárolóval együtt működne. A linkedlistnode<T>  
A system.collections.generic.névter számos kiegészítő osztályt és struktúrát

10.5. táblázat. A System.Collections.Generic.névter osztályai

| Generikus osztály     |       | Nem generikus pár a System.Collections.névterben |  | Jelentése                                                        |  |
|-----------------------|-------|--------------------------------------------------|--|------------------------------------------------------------------|--|
| Stack<T>              |       | stack                                            |  | „Útolsónak be, elsőnek ki” LIFO-lista generikus megvalósítása    |  |
| LinkedList<T>         | n. a. |                                                  |  | Kéthányú láncolt lista ge-<br>nerikus megvalósítása              |  |
| ReadOnlyCollection<T> |       | ReadOnlyCollection                               |  | Írásvédett elemek csoport-<br>jának generikus megvalo-<br>sítása |  |

```

static void Main(string[] args)
{
    // List készítése egész számok tárolására.
    List<int> myInts = new List<int>();

    // List készítése Sportscar objektumok tárolására.
    List<Sportscar> myCars = new List<Sportscar>();

    // List készítése Person objektumok tárolására.
    List<Person> myPeople = new List<Person>();
}

```

Elgondolkodtató, hogy vajon mi lett a megadott helyőrző érték pontosan. Ha a Code Definition ablakot nézzük, akkor a `List<T>` esetében mindig a `T` helykitöltőt fogjuk látni egy konkrét típus helyett. Nézzünk meg egy részleges listát (figyeljük meg a felkövetérel írt elemeket):

```

// A List<T> típus részleges listája.
namespace System.Collections.Generic
{
    public class List<T> :
        IList<T>, ICollection<T>, IEnumerable<T>,
        IList, ICollection, IEnumerable
    {
        ...
        public void Add(T item);
        public IList<T> AsReadOnly();
        public int BinarySearch(T item);
        public bool Contains(T item);
        public void CopyTo(T[] array);
        public int FindIndex(System.Predicate<T> match);
        public T FindLast(System.Predicate<T> match);
        public bool Remove(T item);
        public int RemoveAll(System.Predicate<T> match);
        public T[] ToArray();
        public bool TrueForAll(System.Predicate<T> match);
        public T this[int index] { get; set; }
    }
}

```

Egy `Sportscar` típusú `List<T>` osztály létrehozása a következő `List<T>` osztály definíciójának felel meg:

```

namespace System.Collections.Generic
{
    public class List<Sportscar> :
        IEnumerable<Sportscar>,
        ICollection<Sportscar>,
        IList, ICollection, IEnumerable
    {

```



```

    public void Add(Sportscar item);
    public IList<Sportscar> AsReadOnly();
    public int BinarySearch(Sportscar item);
    public bool Contains(Sportscar item);
    public void CopyTo(Sportscar[] array);
    public int FindIndex(System.Predicate<Sportscar> match);
    public Sportscar FindLast(System.Predicate<Sportscar> match);
    public bool Remove(Sportscar item);
    public int RemoveAll(System.Predicate<Sportscar> match);
    public Sportscar[] ToArray();
    public bool TrueForAll(System.Predicate<Sportscar> match);
    public Sportscar this[int index] { get; set; }
}

```

Természetesen generikus `IList<T>` osztály létrehozásakor a fordító nem hoz létre egy vadonatú `IList<T>` típusú implementációt. Helyette csak az általunk aktuálisan felhasznált generikus típus tagjait címkezi meg. Ehhez hozzunk létre egy `sportscar` objektumokat tartalmazó `IList<T>` osztályt a következőképpen:

```

static void Main(string[] args)
{
    // Sportscar objektumokat tartalmazó listát alkalmazása
    List<Sportscar> myCars = new List<Sportscar>();
    myCars.Add(new Sportscar());
    Console.WriteLine("Your list contains {0} item(s).",
        myCars.Count);
}

```

Ha az `ildasm.exe` segítségével megvizsgáljuk a köztes nyelvi kódot, megte-  
lálhatjuk a következő helyettesítést:

```

.method private hidebysig static void Main(string[] args)
{
    .entrypoint
    .maxstack 2
    .locals init ([0] class
        [mscorlib]System.Collections.Generic.List`1<
        <class Sportscar> myCars)
        newobj instance void class
        [mscorlib]System.Collections.Generic.List`1<
        <class Sportscar>::.ctor()
    stloc.0
    newobj instance void Collections.Generic.Sportscar::ctor()
}

```

## Saját generikus metódusok létrehozása

A továbbiakban azt vizsgáljuk meg, hogy hogyan hozhatunk létre saját generikus metódust, típust vagy gyűjteményt.

```
callvirt instance void class
[microsoft]System.Collections.Generic.List`1
<class SportCar>::Add(i0)
nop
ldstr "Your list contains {0} item(s). "
ldloc.0
callvirt instance int32 class
[microsoft]System.Collections.Generic.List`1
<class SportCar>::get_Count()
box [mscorlib]System.Int32
call void [mscorlib]System.Console::WriteLine(string, object)
nop
ret
}
```

A legtöbb programozó jellemzően az alaposztálykönyvtárban már meglévő generikus típusokat használja, ám természetesen elkészíthetjük saját generikus tagjainkat és egyedi generikus típusunkat is. Ahhoz, hogy megismerjük a generikus típusok felhasználását saját projekteinkben, először készítsünk egy generikus swap metódust. Hozzunk létre egy generikus method nevű új parancs-sort alkatmazást.

A célja ennek a példának az, hogy egy olyan swap metódust alkossunk, amely bármely lehetséges adattípuson (érték- vagy referenciaalapú) működik egyetlen paramétertípus használatával. A cserélő algoritmus természetéből fakadón a bejövő paraméterek átadása referenciaként történik (a C# ref kulcsszavával). A kezdeti Program osztályban létrehozott generikus swap metódus teljes megvalósítása a következő:

```
// A metódus megcseréli két elemet.
// a <T> típus paraméternek megfelelően.
static void Swap<T>(ref T a, ref T b)
{
    Console.WriteLine("You sent the Swap() method a {0}",
        typeof(T));
    T temp;
    temp = a;
    a = b;
    b = temp;
}
```

Figyeljük meg, hogyan definiálunk generikus metódusokat a típusparaméter specifikálásával, a metódus neve után, de a paraméterlista előtt. A swap() metódus bármely két <T> típusú paraméterrel működik. Írjuk ki a tárolt helyőrző típusának a nevét a konzolra a C# typeof() operátorának használatával. Figyeljük meg a következő main() metódust, amely egész- és sztringtípusokat cserél ki:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Fun with Custom Generic  
Methods *****\n");
    // két egész szám cseréje.
    int a = 10, b = 90;
    Console.WriteLine("Before swap: {0}, {1}", a, b);
    Swap<int>(ref a, ref b);
    Console.WriteLine("After swap: {0}, {1}", a, b);
    Console.WriteLine();
    // két sztring cseréje.
    string s1 = "Hello", s2 = "There";
    Console.WriteLine("Before swap: {0} {1}", s1, s2);
    Swap<string>(ref s1, ref s2);
    Console.WriteLine("After swap: {0} {1}", s1, s2);
    Console.ReadLine();
}
```

## A típusparaméterek kikövetkeztetése

Amikor generikus metódusokat használunk fel, például a swap<T>-t, akkor elhagyhatjuk a típusparamétert, de csak akkor, ha a generikus metódusnak paraméterekre van szüksége, hiszen a fordító képes a típusparamétert kikövetkeztetni a tagparaméterekből. Fel tudunk cserélni például két darab system.Boolean típust a következő forráskódban:

```
// A fordító a System.Boolean osztályra következtet.
bool b1 = true, b2 = false;
Console.WriteLine("Before swap: {0}, {1}", b1, b2);
Swap(ref b1, ref b2);
Console.WriteLine("After swap: {0}, {1}", b1, b2);
```

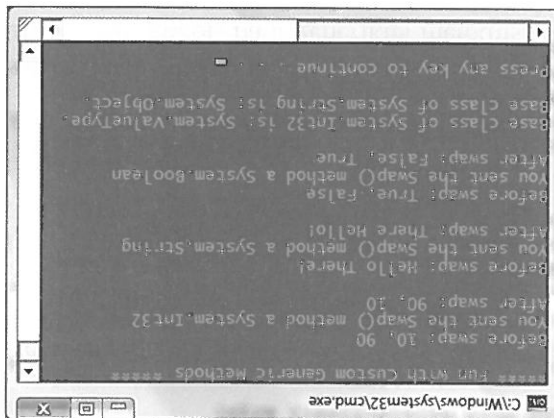
Ha azonban a következő DisplayBase<T> nevű generikus metódussal rendelkezőnk, amelynek egyetlen bejövő paramétere sincs:

```
static void DisplayBase<T>()
{
    Console.WriteLine("Base class of {0} is: {1}.",
        typeof(T), typeof(T).BaseType);
}
```

akkor a meghívás során meg kell adnunk a típusparamétert:

```
static void Main(string[] args)
{
    ...
    // meg kell adni a típus paramétert, ha
    // a módszernek nincsenek paraméterei.
    DisplayBaseClass<int>();
    DisplayBaseClass<string>();
    // Fordítási hiba! Nincsenek paraméterek? Helyőrzőt kell megadni!
    // DisplayBaseClass();
    Console.ReadLine();
}
```

A 10.2. ábrán láthatjuk az alkalmazás kimenetét.



10.2. ábra. Generikus módszerünk működés közben

A generikus `swap<T>` és `DisplayBaseClass<T>` metódusokat az alkalmazásból-jektumon belül definiáljuk (azaz a `Main()` metódusban). Természetesen, mint minden metódussal, ezzel is megtehetjük, hogy ezeket a tagokat is különálló osztálytípusban definiáljuk:

```
public static class MyGenericMethods
{
    public static void swap<T>(ref T a, ref T b)
    {
        Console.WriteLine("You sent the swap() method a {0}",
            typeof(T));
        T temp;
        temp = a;
        a = b;
        b = temp;
    }
}
```

```
public static void DisplayBaseClass<T>()
{
    Console.WriteLine("Base class of {0} is: {1}.",
        typeof(T), typeof(T).BaseType);
}
```

Mivel a statikus `Swap<T>` és `DisplayBaseClass<T>` metódusok hatóköre az új statikus osztálytípus, ezért, amikor felhasználjuk bármelyik tagjukat, meg kell adnunk a típus nevét. Például:

```
MyGenericMethods.Swap<int>(ref a, ref b);
```

A generikus metódusoknak nem kell statikusnak lenniük. Ha a `Swap<T>` és a `DisplayBaseClass<T>` példányozintű volt (és nem statikus osztályban definiáltak őket), akkor érdemes egyszerűen egy `MyGenericMethods` példányt létrehozunk és a metódusokat az objektumválonzon hívjuk meg:

```
MyGenericMethods c = new MyGenericMethods();
c.Swap<int>(ref a, ref b);
```

---

**Forráskód** A `GenericMethod` projekt megtalálható a 10. fejezet alkönyvtárában.

---

## Generikus struktúrák és osztályok létrehozása

A továbbiakban a generikus struktúra létrehozására koncentrálunk a `GenericPoint` nevű új parancssoralkalmazás-projektben belül (a generikus osztályok készítésének a folyamata hasonló). Tegyük fel, hogy elkészítettünk egy generikus `Point` struktúrát, amely egyetlen paramétertípust támogat, reprezentálva a koordináták (x,y) alapvető tárolását. A hívó képes `Point<T>` típust létrehozni a következőképpen:

```
// Egész számokkal működő Point.
Point<int> p = new Point<int>(10, 10);

// Double számokkal működő Point.
Point<double> p2 = new Point<double>(5.4, 3.3);
```

A `Point<T>` teljes definíciója a következő:

```
// Egy generikus Point struktúra
public struct Point<T>
{
    // generikus állapot dátum.
    private T xPos;
    private T yPos;

    // generikus konstruktor.
    public Point(T xVal, T yVal)
    {
        xPos = xVal;
        yPos = yVal;
    }

    // generikus tulajdonságok.
    public T X
    {
        get { return xPos; }
        set { xPos = value; }
    }

    public T Y
    {
        get { return yPos; }
        set { yPos = value; }
    }

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

    // Mezők visszaállítását a típusparaméter
    // alapértelmezett értékére.
    public void ResetPoint()
    {
        xPos = default(T);
        yPos = default(T);
    }
}
```

## A default kulcsszó a generikus kódban

A `Point<T>` osztály használja a típusparaméterét az adatmező definíálásánál, a konstruktornál és a tulajdonság meghatározásánál. Ezen kívül felüldefiniálja a `ToString()`-et, a `Point<T>` pedig egy `ResetPoint()` nevű metódust definiál, amely néhány új szintaktikai megoldást használ:

```
// A "default" kulcsszó C#-ban túlterhelt.
// Generikus kódban a típus paraméter
// alapértelmezett értéket képviseli.
public void ResetPoint()
{
    xpos = default(T);
    ypos = default(T);
}
```

A generikus típusok bevezetésével a C# default kulcsszavának kettős szerepe lett. A switch szerkezeten belüli használat mellett a típusparaméterek alapértelmezett értékének beállítására is felhasználhatjuk. Ez nagy segítség akkor, amikor egy generikus típus nem ismeri az aktuális helyőrzőket rögtön a kezdetkor, és így nem tudja biztonsággal felitélezni az alapértelmezett értéket. A típusparaméter alapértelmezett értékei a következők lehetnek:

- a numerikus értékeknek az alapértelmezett értéke 0;
- a referenciátípusoknak az alapértelmezett értéke null érték;
- a struktúrák mezőinek értéke 0 (értéktípusnál) vagy null érték (referenciátípusnál).

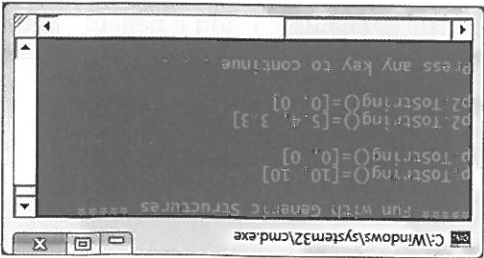
A `Point<T>` osztállyal egyszerűen beállíthatunk `x`-`pos`-t és `y`-`pos`-t közvetlenül 0-ra, ugyanis biztosak lehetünk abban, hogy a hívó csak numerikus adatokkal dolgozik. A default (`T`) szintaxis használatával azonban megnöveljük a generikus típusunk rugalmasságát. Vizsgáljuk meg a következőképpen a `Point<T>` osztály metódusait:

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

    // Egész számokkal működő Point.
    Point<int> p = new Point<int>(10, 10);
    Console.WriteLine("p.ToString()={0}", p.ToString());
    p.ResetPoint();
    Console.WriteLine("p.ToString()={0}", p.ToString());

    // Double számokkal működő Point.
    Point<double> p2 = new Point<double>(5.4, 3.3);
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
    p2.ResetPoint();
    Console.WriteLine("p2.ToString()={0}", p2.ToString());
}
```

A 10.3. ábrán láthatjuk a kimenetet.



10.3. ábra. *Generic Point* típus használata

**Forráskód** The `GenericPoint` projekt megtalálható a 10. fejezet alkönyvtárában.

## Saját generikus gyűjtemény készítése

A `System.Collections.Generic` névtér számos olyan típust biztosít, amelyek lehetővé teszik típusbiztos és hatékony tárolók létrehozását. A leghatékonyabb lasztások köréből adódóan nagy eséllyel nem kell egyedi gyűjteménytípust létrehozunk, amikor .NET-plattformmal dolgozunk. Az erősen típusos generikus tárolót készítésének bemutatásához a következő feladat a `CarCollection` névtérhez egy `CustomGenericCollection` nevű új parancssori alkalmazást.

Mint a korábban már létrehozott `GenericCarCollection`, ez az iteráció is felhasználja a meglévő gyűjteménytípusokat az elemek tárolására (ebben az esetben egy `List<T>` típust). A `GenericCarCollection` interfész megvalósításával a `foreach` iterációt támogatjuk. Az `IEnumerable<T>` a nem generikus `IEnumerable` interfészt bővíti, így azonban a fordító a `GetEnumerator()` metódus két változatát várja. A módosítás a következő:

```
public class CarCollection<T> : IEnumerable<T>
{
    private List<T> arcars = new List<T>();

    public T GetCar(int pos)
    { return arcars[pos]; }

    public void AddCar(T c)
    { arcars.Add(c); }
```



```

public void ClearCars()
{
    arcars.Clear();
}

public int Count
{
    get { return arcars.Count; }
}

// Az Enumerable<T> bővíti az IEnumerable interfészt, ezért
// a GetEnumerator() mindkét verzióját meg kell valósítani.
IEnumerator<T> IEnumerable<T>.GetEnumerator()
{
    return arcars.GetEnumerator();
}

System.Collections.IEnumerator
GetEnumerator()
{
    return arcars.GetEnumerator();
}

```

A módosított `CarCollection`-t a következőképpen használhatjuk fel (egy egyszerű `Car` definíciót feltételezve az aktuális projektünkben):

```

static void Main(string[] args)
{
    Console.WriteLine("***** Custom Generic Collection *****\n");

    // Car objektumok készítésének létrehozása.
    CarCollection<Car> myCars = new CarCollection<Car>();
    myCars.AddCar(new Car("Rusty", 20));
    myCars.AddCar(new Car("Zippy", 90));

    foreach (Car c in myCars)
    {
        Console.WriteLine("PetName: {0}, Speed: {1}",
            c.PetName, c.Speed);
    }
    Console.ReadLine();
}

```

Létrehoztunk egy `Car` típust tartalmazó `CarCollection` típust. Hasonló végrehajtást kell elvégeznünk, ha a `List` típust közvetlenül használjuk. Nagy előnye ennek, hogy szabadon adhatunk egyedi módszereket (`AddCar()`, `GetCar()` stb.) a `CarCollection` típushoz, amely a hívásokat továbbítja a belső `List` típus felé.

## A saját generikus gyűjtemény korlátai

A `CarCollection` osztály azonban nem sokat ad az egyedileg elnevezett publikus módszerek használatán kívül. Ráadásul az objektumfelfhasználó létre tud hozni egy `CarCollection` példányt egy teljesen nem ideillő paramétertípussal:

```
// Szintaktikailag helyes, de roppant bonyolult
CarCollection<int> myInts = new CarCollection<int>();
myInts.AddCar(5);
myInts.AddCar(11);

foreach (int i in myInts)
{
    Console.WriteLine("{0}", i);
}
```

Miért enged a fordító akkor ilyen kódolást? A generikususok nevének megfelelően *generikusak*, vagyis általánosak. Egy típusparaméter pedig minden lehet, még ha a generikus típuson belül teljesen értelemetlen is (pl. egy egészeket tároló autógyűjtemény).

Hogy szemléltessünk egy másik generikus felhasználást, tegyük fel, hogy létrehozunk két, a car típusból származtatott új osztályt (Sportscar és minivan):

```
public class Sportscar : Car
{
    : base(p, s) {}
    public Sportscar(string p, int s)
    {
        // További Sportscar metódusok következnek.
    }
}

public class Minivan : Car
{
    : base(p, s) {}
    public Minivan(string p, int s)
    {
        // További minivan metódusok következnek.
    }
}
```

A származtatás törvénye megengedi a minivan vagy a sportscar típus közötti hozzáférést a car típusú paraméterrel létrehozott CarCollection<T> osztályhoz:

```
// A CarCollection<Car> a car típusból lezármazott
// típusokat tárol.
CarCollection<Car> myAutos = new CarCollection<Car>();
myAutos.AddCar(new Minivan("Family Truckster", 55));
myAutos.AddCar(new Sportscar("Crusher", 40));
```

Bar ez szintaktikailag érvényes kód, de mi történik akkor, ha a CarCollection<T> osztályt egy új publikus PrintPetName() metódussal szeretnénk frissíteni? Ehhez csak hozzá kell férnünk a megfelelő List<T> elemhez és kiennünk belőle a név értékét:

```
// Hibai A System.Object nem rendelkezik  
// PetName nevű tulajdonsággal.  
public void PrintPetName(int pos)  
{  
    Console.WriteLine(arcars[pos].PetName);  
}
```

Ez azonban nem fog lefordulni, ugyanis a `T` igazi azonossága nem ismert, és nem tudjuk biztosan, hogy a `List<T>` típusban tárolt elemnek van-e `PetName` tulajdonsága. Ha a típusparaméternek nincsenek megköötései (mint a fenti osztály esetében), akkor a generikus típust *kötetlennek* nevezzük. Feltehetőenjük, hogy a kötetlen típusparaméterek csak a `System.Object` tagjaival rendelkeznek (amelynek egyértelműen nincs `PetName` tulajdonsága). Megpróbálhatjuk azt is, hogy a `List<T>` indexelőből visszatérő elemet egy erősen típusos `car` elemként tüntetjük fel, és felhasználjuk a visszatérő objektumból a nevet:

```
// Hibai  
// A "T" nem konvertálható "Car" típusra.  
public void PrintPetName(int pos)  
{  
    Console.WriteLine(((Car)arcars[pos]).PetName);  
}
```

Ez azonban megint csak nem fog lefordulni, hiszen a fordító nem ismeri még a `<T>` típusparamétert, és így nem tudja garantálni, hogy a kasztozás érvényes legyen. Ebből következően jogosan kérdezhetjük: mikor (ha egyáltalán vala-mikor) kellene ilyen egyedi generikus tárolót létrehoznunk?

## Típusparaméter korlátozása a *where* kulcsszó használatával

A fő ok, amiért a fejlesztőknek egyedi generikus gyűjteménytípust kellene használniuk, az az, hogy kikényszerítsék a típusparamétereken a *megszorítást* azért, hogy rendkívül típusbiztos tárolót készítsenek. A `C#` nyelvben a megszorításokat a *where* kulcsszó használatával hozzuk létre, ez szabályozza a típusparaméterek tulajdonságait (10.6. táblázat).

Generikus kényszer

Használata

|                              |                                                                                                                                                                                                                                                                                                                                           |
|------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| where T : struct             | A <T> típusparaméter származási láncában System.<br>ValueType-nak kell szerepelnie.                                                                                                                                                                                                                                                       |
| where T : class              | A <T> típusparaméter származási láncában nem szabad<br>System.ValueType-nak lennie (pl. <T> referenciátípus).                                                                                                                                                                                                                             |
| where T : new()              | A <T> típusparaméternek rendelkeznie kell egy alapér-<br>telmezett konstruktorral. Ez akkor segít, ha generikus<br>típusnak létre kell hoznia egy példányt a típusparamé-<br>terből, hiszen nem tudjuk az egyedi konstruktor formá-<br>tumát. Ezt a megszorítást a legutolsó helyen kell szere-<br>peltetni több megszorítás megadásakor. |
| where T :<br>NameOfBaseClass | A <T> típusparaméternek a NameOfBaseClass által spe-<br>cifikált osztályból kell származnia.                                                                                                                                                                                                                                              |
| where T :<br>NameOfInterface | A <T> típusparaméternek NameOfInterface által speci-<br>fikált interfész kell megvalósítania. Többesbörös interfe-<br>szeket megadhatunk vesszővel elválasztott listában.                                                                                                                                                                 |

10.6. Táblázat. Generikus típusparaméter lehetséges kényszerei

Amikor megszorításokat használunk a where kulcsszó segítségével, a kény-  
szerlista a generikus típus összesítőja és az interfészek listája után helyezke-  
dik el. Konkrét példákkal nézzük meg a következő myGenericClass nevű ge-  
nerikus osztály megkötésait:

```
// A MyGenericClass az object leszámazottja, az
// elemei támogatják az alapértelmezett konstruktort, és
// egyben osztályok, amelyek megvalósítják az IDrawable interfészt.
public class MyGenericClass<T> where T : class, IDrawable, new()
{...}

// A MyGenericClass a MyBase leszámazottja, és megvalósítja
// a ISomeInterface, az elemei pedig struktúrák lehetnek.
public class MyGenericClass<T> : MyBase, ISomeInterface where T :
    struct
{...}
```

Ha többszörös típusparamétert specifikáló generikus típust készítünk, akkor  
a kényszerek egyedi halmazát mindenképp specifikálhatjuk:

```
// A <K> alapértelmezett konstruktorral rendelkezik, <T> pedig
// megvalósítja a generikus Icomparable interfészt.
public class MyGenericCClass<K, T> where K : new()
where T : Icomparable<T>
{...
```

Ha a `CarCollection<T>` osztályt szeretnénk módosítani úgy, hogy csak a `Car` osztályból származó típusokat lehessen tárolni benne, akkor a következőt kell írunk:

```
public class CarCollection<T> : IEnumerable<T> where T : Car
{
    ...
    public void PrintPetName(int pos)
    {
        // mivel minden aJelem a Car csatládba tartozik,
        // a PetName tulajdonság közvetlenül hívható.
        Console.WriteLine(arcars[pos].PetName);
    }
}
```

Ha egyszerűen korlátozzuk a `CarCollection<T>` osztályt úgy, hogy csak a `Car` osztályból származó típusokat tartalmazhat, akkor a `PrintPetName()` megvalósítása egyszerű lesz, ugyanis a fordító ekkor már felteszi, hogy `<T>` típus a `Car` típusból származik. Továbbá, ha a megadott típusparaméter nem `Car` típus kompatibilis, akkor fordítási hiba keletkezik:

```
// Fordítási hiba!
CarCollection<int> myInts = new CarCollection<int>();
```

A generikus módszer is használhatja a `where` kulcsszót. Ha például azt szeretnénk, hogy az előzőleg létrehozott `Swap()` módszerbe csak a `System.ValueType` ból származó típusok kerülhessenek, akkor módosítsuk a kódunkat a következőképpen:

```
// A módszer érték típusokat cserél, de osztályokat nem.
static void Swap<T>(ref T a, ref T b) where T : struct
{
    ...
}
```

Ha így korlátozzuk a `Swap()` módszert, akkor nem tudunk sztring típusokat cserélni (ahogy a mintakódban csináltuk), hiszen azok referenciátípusok.

## Az operátorkorlátozások hiánya

Amikor generikus metódusokat hozunk létre, bármely C#-operátor (+, -, \*, == stb.) alkalmazása a típusparamétereken *fordítási hibát* eredményez. Példaként nézzünk egy olyan osztályt, amely Add(), Subtract(), Multiply() és Divide() metódusokat hajt végre generikus típusokon:

```
// Fordítási hibai operátorokat nem
// alkalmazhatunk típusparamétereken!
public class BasicMath<T>
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }
    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }
    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }
    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

Az előbbi BasicMath<T> osztály nem fog lefordulni. Ez komoly korlátozásnak tűnik, ám a generikus típusok *generikusak*, vagyis általánosak. Természetesen a System.Int32 típus jól működik a C# kétoperandusú operátoraival. Ha azonban a <T> egy egyedi osztály- vagy struktúrátípus lenne, akkor a fordító nem feltehetően lehet, hogy a típus túlterhelt a +, -, \* és / operátorokat. Elméletileg a C# nyelvnek engedni kellene, hogy támogatott operátorokkal korlátozzunk egy generikus típust. Például:

```
// Csak szemléltető kód!
// C# 2008 nyelvében ez a kód érvénytelen.
public class BasicMath<T> where T : operator +, operator -,
    operator *, operator /
{
    public T Add(T arg1, T arg2)
    { return arg1 + arg2; }
    public T Subtract(T arg1, T arg2)
    { return arg1 - arg2; }
    public T Multiply(T arg1, T arg2)
    { return arg1 * arg2; }
    public T Divide(T arg1, T arg2)
    { return arg1 / arg2; }
}
```

Sajnos a C# aktuális változata nem támogatja az operátormegszorításokat. Ha generikus interfezszekeket használunk fel, akkor szimulálnunk kell az operátor használatának jellegét a típusparamétereken.

---

**Forráskód** A CustomGenericCollection projekt megtalálható a 10. fejezet alkönyvtárában.

---

## Generikus osztály létrehozása

A generikus osztály lehet más osztályok osztálya, és ennek következtében bármennyi virtuális vagy absztrakt módszert definiálhat. A származtatott típusokra azonban vonatkozik néhány szabály. Először is, ha egy nem generikus osztályt származtatunk le egy generikus osztályból, akkor a származtatott osztálynak a konkrét típusparamétert meg kell határoznia:

```
// Tételjezzük fel, hogy létrehoztunk egy
// egyedi generikus listaosztályt.
public class MyList<T>
{
    private List<T> listofData = new List<T>();
}
// A konkrét típusoknak meg kell határoznia a típus
// paramétert, ha generikus osztály
// leszármaoztatni.
public class MyStringList : MyList<string>
{
}
```

Továbbá, ha a generikus osztály generikus virtuális vagy absztrakt metódusokat definiál, akkor a származtatott típusnak felül kell definiálnia a generikus metódusokat a konkrét típusparaméterek felhasználásával:

```
// Generikus osztály virtuális metódussal.
public class MyList<T>
{
    private List<T> listofData = new List<T>();
    public virtual void PrintList(T data) { }
}
public class MyStringList : MyList<string>
{
    // A származtatásban alkalmazott típusparamétert kell
    // helyettesíteni a leszármaozott metódusban.
    public override void PrintList(string data) { }
}
```

Ha a származtatott típus szintén generikus, akkor a gyermekosztály újra fel tudja használni a típushelyőrzőt a definíciójában. Az őssztályba helyezett megszorításokat azonban a származtatott típusnál is meg kell adni. Például:

```
// két megszorítás vonatkozik az alapételelmezett konstruktorokra.
public class MyList<T> where T : new()
{
    private List<T> listData = new List<T>();
    public virtual void PrintList(T data) { }
}

// A lezármazott típusok tiszteletben tartják a megszorításokat.
public class MyReadonlyList<T> : MyList<T> where T : new()
{
    public override void PrintList(T data) { }
}
```

A napi programozási feladatok között tehát nem gyakran fog szerepelni az egyedi generikusosztály-hierarchiák létrehozása.

## Generikus interfész létrehozása

A `System.Collections.Generic` névtér vizsgálata során már láttuk, hogy generikus interfészek (pl. `IEnumerable<T>`) létrehozása is lehetséges. Saját generikus interfészünket természetesen szabadon definiálhatjuk (megkötésekkel vagy azok nélkül). Tetelezzük fel, hogy generikus típusparaméteren kétoperandusú művelet végrehajtására képes interfészt szeretnénk definiálni:

```
public interface IBinaryOperations<T> where T : struct
{
    T Add(T arg1, T arg2);
    T Subtract(T arg1, T arg2);
    T Multiply(T arg1, T arg2);
    T Divide(T arg1, T arg2);
}
```

Az interfészek többé-kevésbé hasznontalanok, amíg egy osztály vagy egy struktúra meg nem valósítja őket. Amikor egy generikus interfészt valósítunk meg, a támogató típus meghatározza a helyőrző típusát:

```
public class BasicMath : IBinaryOperations<int>
{
    public int Add(int arg1, int arg2)
    {
        return arg1 + arg2;
    }
}
```



```
public int subtract(int arg1, int arg2)
{ return arg1 - arg2; }

public int multiply(int arg1, int arg2)
{ return arg1 * arg2; }

public int divide(int arg1, int arg2)
{ return arg1 / arg2; }
}
```

Ekkor az elvárt módon használhatjuk fel a BasicMath-t:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Generic Interfaces *****\n");
    BasicMath m = new BasicMath();
    Console.WriteLine("1 + 1 = {0}", m.Add(1, 1));
    Console.ReadLine();
}
```

Ha inkább egy olyan BasicMath osztályt hoznánk létre, amely működik a lebegő pontos számokon, akkor a típusparameétert a következőképpen specifikáljuk:

```
public class BasicMath : IBinaryOperations<float>
{
    public float Add(float arg1, float arg2)
    { return arg1 + arg2; }
    ...
}
```

Ebben az esetben a fordító biztosítja, hogy átvesszük a lebegő pontos számot a BasicMath osztály minden metódusában. A kódban szereplő lebegőpontos konstansok alapértelmezett típusa a double. Ha floatként szeretnénk kezelni őket, akkor ezt a literal végére írt f betűvel tehetjük:

```
static void Main(string[] args)
{
    Console.WriteLine("***** Generic Interfaces *****\n");
    BasicMath m = new BasicMath();
    Console.WriteLine("1.98 + 1.3 = {0}", m.Add(1.98f, 1.3f));
    Console.ReadLine();
}
```

---

**Forráskód** A GenericInterface projekt megtalálható a 10. fejezet alkönyvtárában.

---

A következő fejezetben a .NET-metódusreferenciákat vizsgálva vissza fogunk még térni a generikus típusokra.

## Összefoglalás

10. fejezet: Gyűjtemények és generikus típusok

A fejezet a `System.Collections` névtérben található „klasszikus” tárolók vizsgálatával kezdődött. Ezek a típusok továbbra is támogatottak maradnak a visszafelé kompatibilitás miatt, ám a `System.Collections.Generic` névtér generikus pártjai az új .NET-alkalmazások számára is hasznosak. A generikus elemek lehetővé teszik „helyőrzők” definiálását (pl. típusparaméterek), amelyek a létrehozás (vagy a generikus metódusok esetén a meghívás) idején határozzunk meg konkrétan. A generikus típusok alapvetően megoldást nyújtanak a dobozoltasra és a típusbiztos használatra (ezek nehezítették meg a .NET 1.1 alatti szoftver fejlesztését). Továbbá generikus típusok nagyjából el-tüntették az egyedi gyűjteménytípusok készítésének szűkségességét.

Míg leggyakrabban .NET-alapoztálykönyvtárban megvalósított generikus típusokat fogjuk használni, emellett képesek vagyunk saját generikus típust is létrehozni. Ekkor figyelembe vehetünk többféle kényszert (a `where` kulcsszó használatával), hogy növeljük a típusbiztoság szintjét, és gondoskodjunk arról, hogy a típusok „ismert mennyiségű” hajtásvégére műveleteket.

# Metódusreferenciák, események és lambdák

Eddig a legtöbb általunk fejlesztett alkalmazás különböző olyan kódreszeket adott hozzá a `main()` metódushoz, amelyek valamilyen módon objektumoknak küldtek igényeket. A 9. fejezetben megvizsgáltuk, hogy az interfész típusok hogyan használhatók olyan objektumok létrehozására, amelyek „visszatudnak szólni” annak az elemnek, amely létrehozta őket. Bár visszahívható (callback) interfészek használhatók kétirányú kommunikáció megvalósítására, a .NET platformon a metódusreferencia a javasolt módszer a visszahívhatóság megvalósítására.

Alapvető meghatározása szerint a .NET-metódusreferencia olyan típusbiztos objektum, amely később meghívható metódusra vagy metóduslistára mutat. A tradicionális C++-függvénymutatótól eltérően a .NET-metódusreferenciák olyan osztályok, amelyek beépített támogatással rendelkeznek a többes küldéshez (multicasting) és az aszinkron függvényhíváshoz.

Ha már megismerkedtünk a metódusreferencia-típusok létrehozásával, valamint a használataikkal, megnezzük a C# event kulcsszót, amely a metódusreferenciák alkalmazását egyszerűsíti. Ekközen a C# többféle metódusreferenciával és eseményközpontú nyelvi elemével is találkozunk, többek között a névtelen metódussal és a metóduscsoport- (Method Group) átalakítással.

A fejezet végén a C# 2008 egyik új nyelvi elemével, a *lambda kifejezéssel* ismerkedünk meg. Az új lambda operátorral ( $\Rightarrow$ ) az utasítások egy olyan kód-blokkját lehet már kialakítani (és ehhez a kódblokkhoz paramétereket is lehet megadni), ahol erősen típusos metódusreferenciára van szükség. A lambda kifejezés nem sokkal több, mint álcázott névtelen metódus.

## A.NET-metódusreferencia

Mielőtt formálisan definiálnánk a .NET-metódusreferenciákat, jöjjön egy kis történeti áttekintés. A Windows API gyakran használta a C típusú függvénymutatókat olyan entitások létrehozására, amelyeket *callback függvényeknek* vagy

egyszerűen csak *visszahívások* nevezték. A visszahívások használatával a programozóknak lehetősége nyílt úgy kialakítani egy függvényt, hogy az az alkalmazásban egy másik függvényt értesítsen (visszahívjon). Ezt a megközelítést használva a Win32-fejlesztők kezelni tudták a kattintást, az egérmozgást, a menüikiválasztást és az általános kétféle kommunikációt két entitás között. Az általános C típusú visszahívható függvényekkel az a probléma, hogy nem jelennek meg sokkal többet egy natív memóriacímnel. Ideális esetben a callback függvények olyan típusbiztos információkat is tartalmaznak, mint a paraméterek száma (és típusa), valamint a mutatott metódus visszatérési értéke (ha van). A tradicionális visszahívható függvényeknek sajnos nem ez a helyzet, emiatt gyakori hibaforrás, összeomlások és egyéb futásidejű katasztrofák okozója lehet. Ettől függetlenül a visszahívások hasznos megoldások.

A .NET keretrendszerben szintén van lehetőség visszahívások létrehozására, és a működésük a *metódusreferenciák* használatával sokkal biztonságosabb, objektumorientáltabbá vált. Lényegében a metódusreferencia olyan típusbiztos objektum, amely egy másik olyan metódusra (vagy akár metódusok listájára) mutat az alkalmazásban, amelyet később meghívhatunk. Pontosabban fogalmazva, a metódusreferencia-objektum három fontos információt kezel:

- annak a metódusnak a *címét*, amelyen hívásokat hajt végre;
- a metódus *argumentumait* (ha vannak);
- a metódus *visszatérési értékét* (ha van).

---

**Megjegyzés** A C(++)-függvénymutatóktól eltérően a .NET-metódusreferencia mutathat statikus vagy példánymetódusra is.

---

Ha a metódusreferencia elkészült, és megkapta a szükséges információkat, akkor az futásidőben dinamikusan meg tudja hívni azokat a metódusokat, amelyekre mutat. A .NET keretrendszer minden metódusreferenciája (beleértve az egyéni metódusreferenciákat is) automatikusan rendelkezik azzal a képességgel, hogy a metódusait *szinkron* vagy *aszinkron* módon hívja. Mindez nagyon megkönnyíti a programozási feladatokat, hiszen meghívhatunk egy metódust egy második szálon anélkül, hogy manuálisan létre kellene hoznunk és kezelnünk kellene egy Thread objektumot.

---

**Megjegyzés** A metódusreferencia-típusok aszinkron viselkedését lásd a 18. fejezetben a System.Threading névtér áttekintésénél.

---

## A módszerreferencia definiálása C#-ban

Ha a C# nyelvben létre akarunk hozni egy metódusreferenciát, akkor a delegáte kulcsszót kell használnunk. A metódusreferencia neve tetszőleges lehet. Úgy kell azonban definiálnunk, hogy egyezzen annak a metódusnak a szignatúrájával, amelyre mutatni fog. Példaképpen tételizzük fel, hogy a BináryOp nevű metódusreferenciát szerethünk létrehozni, ez pedig bármely olyan metódusra mutat, amely egész értékekkel tér vissza, és két egész bemenő paramétert vár:

```
// Ez a metódusreferencia bármilyen metódusra mutathat,
// amely két egész paramétert vár, és egész értéket ad vissza.
public delegate int BinaryOp(int x, int y);
```

Mikor a C#-fordító metódusreferencia-típust dolgoz fel, automatikusan létrehoz egy lezárt osztályt a `system.MulticastDelegate` osztályból. Ez az osztály (az ösosztályához, a `system.Delegate`-hez kapcsolódva) a szükséges információkat tartalmaz. Ha például megérezzük a bináryop metódusreferenciát tájának tárolásához. Ha például megérezzük a bináryop metódusreferenciát az `ildasm.exe` programmal, akkor a 11.1. ábrán mutatott osztályt láthatjuk.

A fordító generálta bináryop osztály három nyilvános metódust definiál. Vegyük az `Invoke()` metódust alapul, hiszen ezt használjuk minden olyan metódus meghívására, amelyet a metódusreferencia *szinkron* módon kezel, ez pedig annyit tesz, hogy a hívónak meg kell várnia a hívás befejeződését, mielőtt tovább folytathatna. A `szinkron Invoke()` metódust nem kell explicit módon hívni a C#-kódból. A megfelelő C#-szintaxisal az `Invoke()` metódus hívása a háttérben történik.

A `beginInvoke()` és az `endInvoke()` metódusok szolgáltatnak az adott függvény *aszkron* hívására egy külön végrehajtási szalon. Ismervé nemileg a többszörös programozást, tudjuk, hogy a legáltalánosabb ok egy másodikkal létrehozására az, hogy olyan metódust szeretnénk meghívni, amelynek lefutása hosszú ideig tart. Habár a .NET-alapozatlányvitar egy egész névteret nyújt a többszörös programozás lehetőségéhez (system.Threading), a metódusreferencia ezt a funkcionálitást azonnal biztosítja.

```
public delegate string MyDelegate(bool a, bool b, bool c);
```

Először is az invoke() metódusban definiált paraméterek és a visszatérési érték pontosan egyezik a Binaryop metódusreferencia definíciójával. A begininvoke() tagok kezdeti paraméterei (esetünkben két egész) szintén a Binaryop metódusreferencián alapszanak; viszont a begininvoke() metódus mindig két végso paramétert szolgáltat (egy AsyncCallback és egy object típust), s ezek megkönyítik az aszinkron függvényhívást. Végül az endinvoke() visszatérési értéke azonos az eredeti metódusreferencia-deklarációval, és mindig egyetlen paramétert kap, egy objektumot, amely megvalósítja az IAsyncResult interfészt.

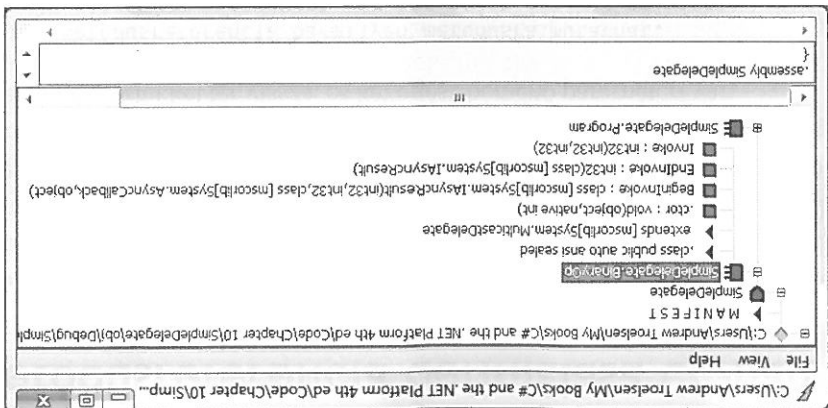
Nézünk egy másik példát. Tetelezzük fel, hogy definiáltunk egy metódusreferenciát, s ez bármely olyan metódusra mutathat, amely sztringet ad vissza, és három system.Boolean bemeneti paramétert vár:

```
sealed class Binaryop : System.MulticastDelegate
{
    public Binaryop(object target, uint functionaddress);
    public int Invoke(int x, int y);
    public IAsyncResult BeginInvoke(int x, int y,
        AsyncCallback cb, object state);
    public int EndInvoke(IAsyncResult result);
}
```

elemeket a metódusreferencia-típus adta meg):

Pontosan hogyan tudja a fordító, hogy hogyan definiálja az invoke(), a begininvoke() és az endinvoke() metódusokat? A folyamat megértéséhez nézzük meg a fordító generálta Binaryop osztálytípus lenyegét (a felkövően szedett

11.1.1. ábra: A C# delegate kulcsszava egy lezárt osztályt jelöl, amely a System.MulticastDelegate osztályból származik



Ebben az esetben a fordító által generált osztály a következő lesz:

```
sealed class MyDelegate : System.MulticastDelegate
{
    public MyDelegate(object target, uint functionAddress);
    public string Invoke(bool a, bool b, bool c);
    public AsyncResult BeginInvoke(bool a, bool b, bool c,
        AsyncCallback cb, object state);
    public string EndInvoke(AsyncResult result);
}
```

A metódusreferencia „mutathat” olyan metódusokra is, amelyek out vagy ref paramétereket tartalmaznak (valamint a params kulcsszóval megjelölt tömbparamétereket is). Példaként nézzük a következő metódusreferencia-típust:

```
public delegate string MyOtherDelegate(out bool a, ref bool b,
    int c);
```

Az Invoke() és a BeginInvoke() metódusok szignatúrája úgy néz ki, ahogyan várjuk, nézzük meg azonban az EndInvoke() metódust, amely tartalmazza az összes, a metódusreferencia által definiált out/ref argumentum halmazát:

```
sealed class MyOtherDelegate : System.MulticastDelegate
{
    public MyOtherDelegate(object target, uint functionAddress);
    public string Invoke(out bool a, ref bool b, int c);
    public AsyncResult BeginInvoke(out bool a, ref bool b, int c,
        AsyncCallback cb, object state);
    public string EndInvoke(out bool a, ref bool b,
        AsyncResult result);
}
```

Összefoglalva: a C#-metódusreferencia definíciója lezárt osztályt eredményez hátrom, a fordító által generált olyan metódussal, amelyek paraméterei és visszatérési típusai a metódusreferencia deklarációján alapulnak. A követ-

```
// Ez csak pszeudokód!
public sealed class DelegateName : System.MulticastDelegate
{
    public DelegateName(object target, uint functionAddress);
    public delegate ReturnValue Invoke(DelegateInputRefAndOutParams);
    public AsyncResult BeginInvoke(DelegateInputRefAndOutParams,
        AsyncCallback cb, object state);
    public delegate ReturnValue EndInvoke(DelegateRefAndOutParams,
        AsyncResult result);
}
```

# A System.MulticastDelegate és System.Delegate alapoztalyok

11. fejezet: Metodusreferenciák, események és lambdák

Ha a C# delegate kulcsszavával készítünk típusokat, akkor indirekt módon olyan osztályt hozunk létre, amely a System.MulticastDelegate osztályból származik. Ennek az osztálynak a leszármazottai hozzáférnek ahhoz a listához, amely a metodusreferencia-típus által kezelt metodusok címeit tartalmazza, valamint a hívási listával történő kommunikáláshoz különböző metodusokat (és néhány túlterhelt operátort). Nézzük meg a System.MulticastDelegate osztály néhány tagját:

```
public abstract class MulticastDelegate : Delegate
{
    // A „mutatott” függvények listáját adja vissza.
    public sealed override Delegate[] GetInvocationList();

    // Túlterhelt operátorok
    public static bool operator ==(MulticastDelegate d1,
    MulticastDelegate d2);
    public static bool operator !=(MulticastDelegate d1,
    MulticastDelegate d2);

    // Privát használatú, a metodusreferencia metoduslistájának
    // kezelése.
    private IntPtr _invocationCount;
    private object _invocationList;
}
```

A System.MulticastDelegate osztály a System.Delegate nevű szülőosztályától is kap további funkcionálitásokat. A következőkben az osztálydefiníció egy részlete látható:

```
public abstract class Delegate : ICloneable, ISerializable
{
    // Metodusok a függvények listájának eléréséhez.
    public static Delegate Combine(params Delegate[] delegates);
    public static Delegate Combine(Delegate a, Delegate b);
    public static Delegate Remove(Delegate source, Delegate value);
    public static Delegate RemoveAll(Delegate source, Delegate value);

    // Túlterhelt operátorok
    public static bool operator ==(Delegate d1, Delegate d2);
    public static bool operator !=(Delegate d1, Delegate d2);
}
```



```
// tulajdonságok, amelyek megmutatják a módszerreferenciák célját.  
public MethodInfo Method { get; }  
public object Target { get; }  
}
```

Követlenül soha sem származtathatunk osztályokat ezekből az alaposztályokból (ha mégis megesszük, az fordítási hibát generál). Mindazonáltal, ha a delegate kulcsszót használjuk, akkor indirekt módon multicasdelegate osztályt hozunk létre. A 11.1. táblázat a módszerreferencia-típusok alapvető tagjait mutatja.

Örökítő tag

Valós jelentés

|                     |                                                                                                                                                                                                                                                                                    |
|---------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Method              | Ez a tulajdonság System.Reflection.MethodInfo-t tartalmaz, amely a módszerreferencia által kezelt statikus módszerekről tartalmaz információt.                                                                                                                                     |
| Target              | Ha a hívott függvényt objektumszinten definiáljuk (nem pedig statikus módszerként), akkor a Target egy olyan objektumot ad vissza, amely a módszerreferencia által kezelt függvényt reprezentálja. Ha a Target által visszaadott érték null, akkor a hívott függvény statikus tag. |
| Combine()           | Ez a statikus függvény a módszerreferencia listájához ad hozzá egy módszert. A C# nyelven ezt a módszert kiváltó hívások a róliditásként használhatók a tülterhelt operátorral.                                                                                                    |
| GetInvocationList() | Ez a módszer System.Delegate típusok tömbjét adja vissza, amelyek mindegyike egy adott, meghívható módszer reprezentál.                                                                                                                                                            |
| Remove()            | Ezek a statikus módszerek eltávolítanak egy (vagy az összes) módszert a módszerreferencia hívási listájából. A C# nyelven a Remove() módszert indirekt módon is meghívhatjuk a -= tülterhelt operátorral.                                                                          |

11.1. táblázat: System.MulticastDelegate/System.Delegate tagok

## A leggyorsabb módszerreferencia

A módszerreferencia eleinte bonyolultnak tűnik. Nézzünk meg egy nagyon egyszerű példát (a neve simpleDelegate), amely a korábban látott BinaryOp módszerreferencia-típust használja. A következő kódot fogjuk elemezni:

```

namespace SimpleDelegate
{
    // Ez a metódusreferencia bármilyen metódusra mutathat, amely
    // két egész szám paramétert vár, és egész szám értéket ad vissza.
    public delegate int BinaryOp(int x, int y);

    public class SimpleMath
    {
        public static int Add(int x, int y)
        { return x + y; }
        public static int Subtract(int x, int y)
        { return x - y; }
    }

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("***** Simple Delegate Example *****\n");

            // Hozzuk létre a BinaryOp objektumot, amely
            // a SimpleMath.Add() metódusra "mutat".
            BinaryOp b = new BinaryOp(SimpleMath.Add);

            // Hívjuk meg az Add() metódust indirekt módon,
            // a metódusreferencia objektumot használva.
            Console.WriteLine("10 + 10 is {0}", b(10, 10));
            Console.ReadLine();
        }
    }
}

```

A `BinaryOp` metódusreferencia formátuma bármely függvényre mutathat, amelyik két egész paramétert vár, és egész értéket ad vissza (a mutatót függvény tényileges neve nem számít). Létrehozunk egy `SimpleMath` nevű osztályt, amely két statikus metódust definiál, és ezek (meglepetésre) egyeznek a `BinaryOp` metódusreferencia minijával.

Ha a célmetódust szeretnénk beletenni egy adott metódusreferenciába, akkor egyszerűen át kell adni a metódus nevét a metódusreferencia konst-ruktorának. Ekkor már meg lehet hívni a mutatót tagot, a közvetlen függ-vényhíváshoz hasonló szintaxist használva:

```

// Az Invoke() metódust itt hívjuk!
Console.WriteLine("10 + 10 is {0}", b(10, 10));

```

Futas közben, a háttérben valójában a fordító által generált `Invoke()` meghívása történik. Ezt leellenőrizhettük úgy, hogy a szerelvényt megnyitjuk az `ildasm.exe` programmal, és megnézzük a `main()` függvényben a CIL-kódot:

```
.method private hidebysig static void main(string[] args)
{
    cil managed
    ...
    callvirt instance int32 SimpleDelegate.BinaryOp::Invoke(int32,
        int32)
}
```

Habár a C# nyelvben nincs szükség az `Invoke()` metódus explicit hívására, ám ezt mégis bármikor szabadon megtehetjük. Ennek megfelelően helyes a következő utasítás:

```
Console.WriteLine("10 + 10 is {0}", b.Invoke(10, 10));
```

A .NET-metódusreferenciák *tipusbiztosak*. Ennek következtében, ha egy metódusreferenciának olyan metódust adunk át, amelyik nem „illeszkedik a mintához”, akkor fordítás idejű hibát kapunk. Ennek bemutatására tegyük fel, hogy a `SimpleMath` osztályunk definiál egy további `SquareNumber()` nevű metódust, amely egyetlen egész számot vár bemenetként:

```
public class SimpleMath
{
    ...
    public static int SquareNumber(int a)
    {
        return a * a;
    }
}
```

Mivel a `BinaryOp` metódusreferencia *csak* olyan metódusokra mutathat, amelyek két egész bemenő paramétert várnak, és egész a visszatérési értékük, így a következő kód helytelen, és nem fordul le:

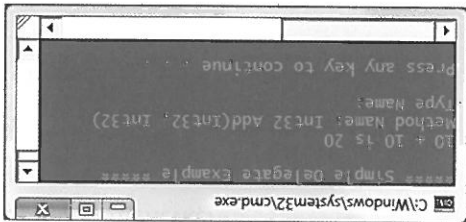
```
// Hiba! A metódus nem egyezik a metódusreferencia mintájával!
BinaryOp b2 = new BinaryOp(SimpleMath.SquareNumber);
```

## A metódusreferencia-objektum vizsgálata

Hozzunk létre a példánkban egy statikus metódust `displayDelegateInfo()` néven a `Program` típuson belül. Ez a metódus kiírja a bejövő metódusreferenciátípus által kezelt metódusok neveit, valamint a metódust definiáló osztály nevét is. Ehhez iterálni fogunk a `system.delegate` tömbön, amelyet a `InvocationList()` metódus ad vissza, és meghívjuk minden egyes objektum Target és Method tulajdonságát:

```
static void DisplayDelegateInfo(Delegate delObj)
{
    // Irjuk ki minden egyes tag nevét
    // a metódusreferencia hívási listájában.
    foreach (Delegate d in delObj.GetInvocationList())
    {
        Console.WriteLine("Method Name: {0}", d.Method);
        Console.WriteLine("Type Name: {0}", d.Target);
    }
}
```

Felteve, hogy módosítottuk a `Main()` metódusunkat ennek az új segédmetódusnak a meghívására, a 11.2. ábrának megfelelő kimenetet kapjuk.



11.2. ábra: A metódusreferencia hívási listájának vizsgálata

A típus nevét (`SimpleMath`) nem írja ki a Target tulajdonság. Ez azzal magyarázható, hogy a `BinaryOp` metódusreferencia *static* metódusra mutat, és így nincsen hivatkozott objektum. Ha azonban módosítjuk az `Add()` és a `Subtract()` metódusokat úgy, hogy ne legyenek statikusak (egyszerűen töröljük a `static` kulcsszavakat), létrehozhatjuk a `SimpleMath` típus egy példányát, és objektumreferencia használatával megadhatjuk a meghívandó metódusokat:

```
static void Main(string[] args)
{
    Console.WriteLine("**** Simple Delegate Example ****\n");
}
```

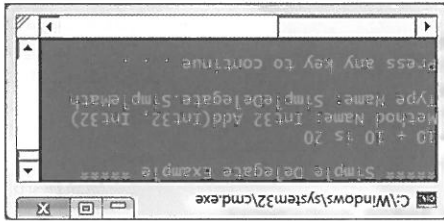
## A Car típus utólagos módosítása metódusreferenciákkal

Az előző simpledelegat példae gyértelműen csak illusztráció, hiszen nem lenne értelme metódusreferenciát írni két szám összeadására. A metódusreferencia-típus életszerűbb felhasználásának bemutatásához tervezzük át a 9. fejezetben leírt metódusreferenciát úgy, hogy az információkat .NET-metódusreferenciával küldjük az egyedi callback interfész helyett. Az engine notation interfészt a car már nem valósítja meg, így a következő lépések szükségesek:

- Defináljunk új metódusreferencia-típusokat, amelyek üzenetet küldenek a hívónak.
- Deklaráljunk tagváltozókat a car osztályban ehhez a metódusreferenciához.

**Forráskód** A SimpleDelegate projekt megtalálható a 11. fejezet alkönyvtárában.

11.3. ábra: A metódusreferencia hívási listájának vizsgálata



Ebben az esetben a 11.3. ábrán látható kimenetet kapjuk.

```
// A .NET-metódusreferencia mutathat példánymetódusokra is.
SimpleMath m = new SimpleMath();
BinaryOp b = new BinaryOp(m.Add);

// Jelentsünk meg információt az objektumról.
DisplayDelegateInfo(b);

Console.WriteLine("10 + 10 is {0}", b(10, 10));
Console.ReadLine();
}
```

A Car típus utólagos módosítása metódusreferenciákkal

- Hozzunk létre segédfüggvényeket a car osztályban, amelyek lehetővé teszik a metódusreferencia-tagváltozók által kezelt metódusok beállítását.
- Frissítjük az Accelerate() metódust úgy, hogy a megfelelő körülmények között hívja meg a metódusreferencia hívási listáját.

Ennek első lépéseként hozzuk létre a CarDelegate nevű parancssoralkalmazás-projektet, és a 9. fejezet callbackInterface példájából illesszük bele a Car és a Radio definíciókat (megváltoztathatjuk az ezeket a típusokat tartalmazó névteret az aktuális projektnévre, vagy alternatívaként importálhatjuk a callbackInterface névteret is). Nézzük meg a Car osztály következő változtatásait, ezek az első három pontot valósítják meg:

```
public class Car
{
    // Defináljuk a metódusreferencia-típusokat.
    public delegate void AboutToBlow(string msg);
    public delegate void Exploded(string msg);

    // Defináljuk az egyes metódusreferencia-típusok tagváltozóit.
    private AboutToBlow almostDeadList;
    private Exploded explodedList;

    // Tagok hozzáadása a hívási listához segédmetódusokkal.
    public void OnAboutToBlow(AboutToBlow clientMethod)
    { almostDeadList = clientMethod; }

    public void OnExploded(Exploded clientMethod)
    { explodedList = clientMethod; }
    ...
}
```

Ebben a példában a referencia-típusokat közvetlenül a car típus hatókörén belül definiáltuk. Ha megnézzük az alaposztálykönyvtárakat, akkor kiderül: elégé általános, hogy a metódusreferenciákat annak a típusnak a hatókörén belül szokás definiálni, amellyel természetesen elég együtműködik. Továbbá meg kell jegyeznünk, hogy ha a fordító a metódusreferenciát egy teljes osztálydefinicióvá alakítja, akkor valójában két beágyazott osztályt (AboutToBlow és Exploded) hozunk létre a car osztályon belül.

Két privát tagváltozót (mindkét metódusreferencia-típusnak egyet) és két segédmetódust (OnAboutToBlow() és OnExploded()) deklarálunk, amelyek lehetővé teszik az ügyfelek, hogy metódusokat adjon hozzá a metódusreferencia hívási listájához. Konceptjükhöz ezekben ezek a metódusok nagyon hasonlótanak a callbackInterface példához (Advise() és Unadvise()) me-

tódusokhoz. Természetesen ebben az esetben a bejövő paraméter egy ügyfél által lefoglalt metódusreferencia–objektum lesz, nem pedig az egyedi interfész megvalósító osztály.

**Megjegyzés** Szigorúan véve publikusként is definiálhatunk volna a metódusreferencia tagvaltozóit, és ezzel elkerülhetők volna, hogy további regisztrációs metódusokat kelljen létrehozni. Mindamellelt azzal, hogy a tagokat privátként hozzuk létre, az egyszerűbbé zárast segítjük, és így sokkal típusbiztosabb megoldást nyújtunk. Meg fogjuk később vizsgálni a nyilvános metódusreferencia vészélyeit a C# event kulcsszavának tanulmányozásakor.

Ezen a ponton módosítanunk kell az Accelerate() metódust a metódusreferenciák meghívására, ahelyett, hogy az arraylist segítségével iteralnánk az ügyféloldali nyelön (ahogy a callbackinterface példában tettük). A módosítás a következő:

```
public void Accelerate(int delta)
{
    // Ha az autó tönkrement, akkor kiváltjuk az Exploded eseményt.
    if (CarIsDead)
    {
        if (ExplodedList != null)
        {
            ExplodedList("Sorry, this car is dead...");
        }
        else
        {
            currspeed += delta;
        }
    }
    // Majdnem tönkrement?
    if (10 == maxspeed - currspeed
        && almosDeadList != null)
    {
        almosDeadList("careful buddy! gonna blow!");
    }
    // Még jó!
    if (currspeed >= maxspeed)
        CarIsDead = true;
    else
        Console.WriteLine(">Currspeed = {0}", currspeed);
}
```

Mielőtt meghívnánk az almosDeadList és az ExplodedList tagváltozók által kezelt metódusokat, ellenőrizzük, hogy nem null-e az értékek. Erre azért van szükség, mert a hívó feladata ezeknek az objektumoknak a lefoglalása az onaboutTlow() és onExploded() segédmetódusok hívásával. Ha a hívó nem

hívja meg ezeket a metódusokat, mi viszont megpróbáljuk meghívni a metódusreferencia hívási listáját, akkor nullreferenciacegypion típusú hibát váltunk ki futási időben (és ez nyilvánvalóan nem jó). Ha megvan a metódusreferencia-infrastukturánk, nézzük meg a Program osztály változtatásait is:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Delegates as event enablers *****\n");
        // Hozzuk létre a Car típust szokásos módon.
        Car c1 = new Car("StugBug", 100, 10);

        // Az eseménykezelők bejegyzése.
        c1.OnAboutToBlow(new Car.AboutToBlow(Car.AboutToBlow));
        c1.OnExploded(new Car.Exploded(Car.Exploded));

        // Gyorsítás (ez kiváltja az eseményeket).
        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        Console.ReadLine();
    }
}

// A Car hívja a következő függvényeket.
public static void CarAboutToBlow(string msg)
{
    Console.WriteLine(msg);
}

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

A legfontosabb a fentiekben az, hogy a hívó rendeli hozzá az értékeket a metódusreferencia tagváltóhoz a segítségével a metódusokkal. Mivel az AboutToBlow és az Exploded metódusreferenciák a Car osztályon belül vannak, így ezekre teljes nevükkel (pl. Car.AboutToBlow) kell hivatkoznunk. Mint minden metódusreferencia-konstrukciónál, itt is a függvény nevét adjuk át bemondó paraméterként, amely így hozzáadódik a hívási listához, amely ebben az esetben két statikus tag a Program osztályban (ha ezeket a metódusokat új osztályba szeretnénk tenni, akkor hasonlóan kell eljárni, mint a Carevntsink típusnál a callbackinterface példában).



## Többes küldés engedélyezése

A .NET-metódusreferencia támogatja a *többes küldést* (multicastot). Másként: a metódusreferencia-objektum a hívható metódusok listáját kezeli, nem csak egy metódust. Ha egy metódusreferencia-objektumhoz több metódust is hozzá szeretnénk adni, akkor egyszerűen a tültérhelt += operátort kell használnunk, nem pedig a közvetlen értékadását. A car típuson a többes küldés engedélyezéséhez a következőképpen kell átírnunk az onabouttoblow() és onexploded() függvényeket:

```
public class Car
{
    // Tagok hozzáadása a hívási listához.
    public void onabouttoblow(AboutToblow clientMethod)
    { almostDeadlist += clientMethod; }

    public void onexploded(Exploded clientMethod)
    { explodedlist += clientMethod; }
}
```

Ezzel a hívó több célt is regisztrálhat ugyanahhoz a visszatívháshoz:

```
class Program
{
    static void main(string[] args)
    {
        Console.WriteLine("***** Delegates as event enablers *****\n");
        Car c1 = new Car("SugBug", 100, 10);

        // Több eseménykezelő bejegyzése!
        c1.onabouttoblow(new Car.AboutToblow(Car.almostDoomed));
        c1.onexploded(new Car.Exploded(Car.exploded));
    }

    // A car hívja a következőket.
    public static void CarAboutToblow(string msg)
    { Console.WriteLine(msg); }
    public static void CarAlmostDoomed(string msg)
    { Console.WriteLine("Critical Message from Car: {0}", msg); }
    public static void CarExploded(string msg)
    { Console.WriteLine(msg); }
}
```

A köztes nyelvi kódban (CIL) a `+=` operátor fordítása a statikus `Delegate.Combine()` metódus hívására történik (valójában meghívhatnánk a `Delegate.Combine()` metódust közvetlenül is, de a `+=` használata egyszerűbb). Nézzük meg az `onAboutToShow()` metódus CIL-megvalósítását:

```
.method public hidebysig instance void onAboutToShow
{
    class CarDelegate, Car/AboutToShow
    {
        maxstack 8
        ldarg.0
        dup
        ldftd class CarDelegate, Car/AboutToShow
        CarDelegate.Carr:::almostDeadlist
        ldarg.1
        call class [mscorlib]System.Delegate
        [mscorlib]System.Delegate:::Combine
        class [mscorlib]System.Delegate,
        class [mscorlib]System.Delegate
        castclass CarDelegate, Car/AboutToShow
        stftd class CarDelegate, Car/AboutToShow
        CarDelegate.Carr:::almostDeadlist
        ret
    }
}
```

A `Delegate` osztály egy statikus `Remove()` metódust is definiál, amely lehetővé teszi a hívónak, hogy egy tagot dinamikusan eltávolítson a listáról. A `C#`-fejlesztők ehelyett a `--` túlterhelt operátort is használhatják.

**Megjegyzés** A `Remove()` metódus számára átadott objektumnak a hozzárendelt szignatúrájával kell egyeznie, de a törlni kívánt aktuális metódusreferencia-példánnyal nem. Vagyis nem kell tárolnunk a gyűjteményhez hozzáadott pontos metódusreferencia-objektumot ahhoz, hogy pontosan ezt a példányt tudjuk eltávolítani. Létrehozhatunk egy új metódusreferencia-objektumot, amely ugyanarra a függvényre mutat az adott példányon, és ezt átadhatjuk a `Remove()` metódusnak. A metódusreferencia egyezni fog az előző metódusreferenciával, hiszen ugyanarra az objektumra mutat.

Ha lehetővé szeretnénk tenni a hívónak, hogy leválassza magát az `AboutToShow` és az `explored` üzenetekről, akkor a `Car` típushoz egy további segéd-metódust kell adnunk (figyeljük meg a `--` operátor működését):

```
public class Car
{
    // Tag eltávolítása a hívási listáról.
    public void RemoveAboutToShow(AboutToShow clientMethod)
    { almostDeadlist -= clientMethod; }
}
```

```
public void RemoveExploded(Exploded ClientMethod)
{ explodedList -= ClientMethod; }
...
}
```

A -= szintaxis megint csak egy egyszerű alternatív jelölés a statikus Delegate.Remove() metódus kézi hívásához, ahogy azt a Car típus RemoveAboutToBlow() tagjának alábbi CIL-kódja illusztrálja:

```
.method public hidebysig instance void RemoveAboutToBlow(class
  CarDelegate Car/AboutToBlow ClientMethod) cil managed
{
  .maxstack 8
  ldarg.0
  dup
  ldftd class CarDelegate Car/AboutToBlow
  CarDelegate Car:::almostDeadList
  ldarg.1
  call class [mscorlib]System.Delegate
  [mscorlib]System.Delegate::Remove(
  class [mscorlib]System.Delegate,
  class [mscorlib]System.Delegate)
  castclass CarDelegate Car/AboutToBlow
  stfd class CarDelegate Car/AboutToBlow
  CarDelegate Car:::almostDeadList
  ret
}
```

## Cél eltávolítása a metódusreferencia hívási listájáról

A Car osztály aktuális módosításaival leállíthatjuk az exploded jelenséket a main() függvény következő módosításával:

```
static void main(string[] args)
{
  Console.WriteLine("**** Delegates as event enablers ****\n");
  Car cl = new Car("StugBug", 100, 10);

  // Tartsuk meg a Car.Exploded metódusreferencia-objektumot
  // későbbi használatra.
  Car.Exploded d = new Car.Exploded(Car.Exploded);
  cl.OnExploded(d);

  ...
  // Távolítsuk el a Car.Exploded metódust
  // a hívási listából.
  cl.RemoveExploded(d);
  ...
}
```

```
// Módosított Car osztály
public class Car
{
    ...
    // Le kell mosni? Forгатni kell a gumiabroncsokat?
    private bool isDirty;
    private bool shouldRotate;

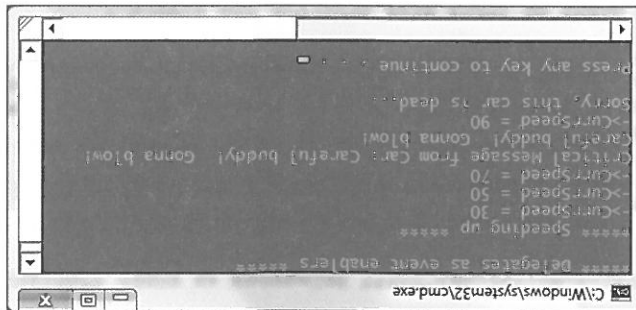
    // Többi paraméterek a logikai változók beállításához.
    public Car(string name, int max, int curr,
        bool washCar, bool rotateTires)
    {
        ...
        isDirty = washCar;
        shouldRotate = rotateTires;
    }
}
```

Egy összetettebb módszerreferencia-példához hozzunk létre egy paramssori alkalmazást Car/radio típusdefiníciókat is tartalmaznia kell az új projektnek). Egészítsük ki a car típust két új logikai tagváltozóval. Az elsővel azt döntjük el, hogy le kell-e már mosni az autót (isDirty); a más-sik azt mutatja, hogy a kérdéses autó gumiabroncsai forogjanak-e (should-rotate). Az objektumhasználó számára lehetővé kell tennünk az új állapot-adatok alkalmazását, ehhez a car néhány további tulajdonságot és egy módo-sított konstruktor definíal. A módosított osztály a következő:

## Egy összetettebb módszerreferencia

**Forráskód** Az CarDelegate projekt megtalálható a 11. fejezet alkönyvtárában.

11.4. ábra: A CarDelegate alkalmazás működése



A CarDelegate alkalmazás kimenete a 11.4. ábrán látható.

11. fejezet: Metódusreferenciák, események és lambdák

```

    public bool dirty
    {
        get { return isDirty; }
        set { isDirty = value; }
    }
    public bool rotate
    {
        get { return shouldRotate; }
        set { shouldRotate = value; }
    }
}

```

Tételezzük fel azt is, hogy a `car` típus tartalmaz egy további, `carMaintenanceDelegate` nevű metódusreferencia-típust is:

```

// A Car definiál egy másik metódusreferenciát.
public class Car
{
    // Meg tud hívni bármilyen metódust, amelynek Car a
    // bemenő paramétere, és nincs visszatérési értéke.
    public delegate void CarMaintenanceDelegate (Car c);
    ...
}

```

A `CarMaintenanceDelegate` típus bármilyen függvényre mutathat, amely `car` típusú bemenő paramétert vár, és sincsen visszatérési értéke.

## Metódusreferenciák mint paraméterek

Az új metódusreferencia-típus mellett, amely `car` típusú paramétert váró metódusokra mutat, és sincsen visszatérési értéke, létrehozhatunk olyan más függvényeket is, amelyek ezt a metódusreferenciát várják paraméterként. Ehhez tegyük fel, hogy van egy új, `garage` nevű osztályunk is, ez pedig olyan `car` típusok gyűjteményét kezeli, amelyet a `List<T>` lista tartalmaz. Az elején ezt a `List<T>` listát felöltöljük néhány `car` típussal (az új kódunkba importálni kell a `System.Collections.Generic` névteret):

```

// A Garage osztály kezeli a Car típusok listáját.
public class Garage
{
    // A garázsban álló autók listája.
    private List<Car> theCars = new List<Car>();
    // Hozzuk létre a garázsban álló autókat.
}

```

```

    public Garage()
    {
        // Emlekezzünk, hogy módosítottuk a konstruktort az isDirty és
        // shouldrotate paraméterek beállítására.
        theCars.Add(new Car("Viper", 100, 0, true, false));
        theCars.Add(new Car("Fred", 100, 0, false, false));
        theCars.Add(new Car("B117yBob", 100, 0, false, true));
    }
}

```

A garage osztály még egy ProcessCars() metódust is definiál, amely az új metódusreferenciánk (car.Carmaintenancedelegat) egyetlen argumentumát veszi át. A ProcessCars() megvalósításában a gyűjteményünkben lévő car objektumokat a metódusreferencia paramétereként adja át a „mutatott függvénynek”. A ProcessCars() metódus a System.MulticastDelegate Target és method tagjait is használja annak meghatározására, hogy a metódusreferencia pontosan melyik függvényre mutat:

```

// A garage osztálynak van egy metódusa, amely használja
// a Carmaintenancedelegat metódusreferenciát.
public class Garage
{
    ...
    public void ProcessCars(Car.Carmaintenancedelegat proc)
    {
        // Hova irányítjuk át a hívást?
        Console.WriteLine("***** Calling: {0} *****",
            proc.Method);
        // Példáymetódust vagy statikus metódust hívunk?
        if(proc.Target != null)
            Console.WriteLine("-->Target: {0} ", proc.Target);
        else
            Console.WriteLine("-->Target is a static method");
        // Hívjuk meg a "mutatott" metódust, átadva neki az autókat.
        foreach (Car c in theCars)
            Console.WriteLine("\n-> Processing a Car");
        proc(c);
    }
}
}

```

Mint bármelyik metódusreferencia-műveletnél a ProcessCars() hívásakor is elküldjük a metódus címét, amely a kérést fogja kezelni (a metódusreferencia-típuson keresztül). A metódusreferencia-típus mutathat statikus vagy cíla-típusú metódusokra is. Tétellezzük fel, hogy ezek példányszintű tag-

függvények, a nevük: washCar() és rotateTires(), és a ServiceDepartment nevű új osztályunk definiálja őket. Ez a két módszer a car típus új rotate és dirty tulajdonságait használja.

```
//Ez az osztály olyan metódusokat definiál, amelyeket
// a CarMaintenanceDelegate típus hív meg.
public class ServiceDepartment
{
    public void washCar(Car c)
    {
        if(c.dirty)
            Console.WriteLine("Cleaning a car");
        else
            Console.WriteLine("This car is already clean...");
    }

    public void rotateTires(Car c)
    {
        if(c.rotate)
            Console.WriteLine("Tires have been rotated");
        else
            Console.WriteLine("Don't need to be rotated...");
    }
}
```

Az új Car.MaintenanceDelegate, Garage és ServiceDepartment típusok közötti kapcsolat bemutatására tételezzük fel a következő használatot:

```
// A Garage minden munkamegbízást a ServiceDepartmenthez irányít
// (jó szerelőt találni mindig probléma...)
static void Main(string[] args)
{
    Console.WriteLine("****Delegates as Parameters ****\n");
    // Hozzuk létre a garázst.
    Garage g = new Garage();
    // Hozzuk létre a szervízt.
    ServiceDepartment sd = new ServiceDepartment();
    // Mossuk le az összes piszkos autót.
    g.ProcessCars(new Car.CarmaintenanceDelegate(sd.WashCar));
    // Forgassuk a gumitábroncsokat.
    g.ProcessCars(new Car.CarmaintenanceDelegate(sd.RotateTires));
    Console.ReadLine();
}
```

A 11.5. ábrán láthatjuk az aktuális kimenetet.

```

    ...
    foreach(Car c in theCars)
    {
        proc(c); // proc(c) => ServicedDepartment.WashCar(c)
    }
}

public void ProcessCars(Car.CarMaintenancedelegat proc)
{
    // A CarDelegat a ServicedDepartment.WashCar függvényre mutat.
}

```

cessCars()) a következőképpen értelmezhető:

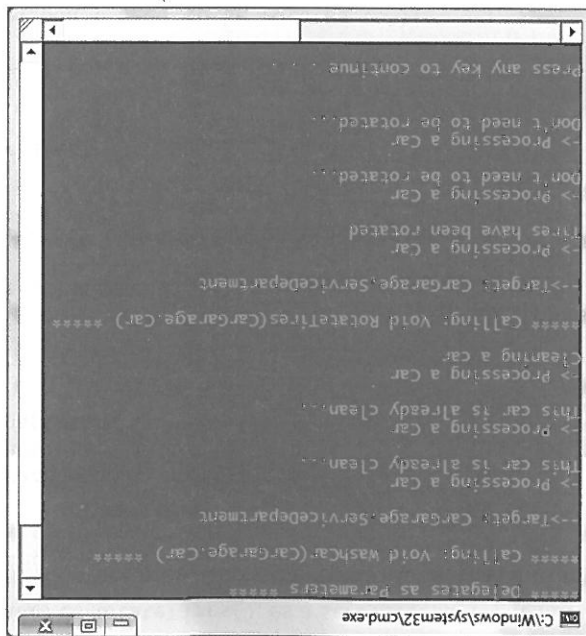
megmagyarázza, hogy egy 30 perces olajcsere miért tart 2 óráig). Így a Pro-  
több igazi garázsban is, a tényleges munkát kiosztják a szervizosztálynak (ez  
jűk át ezt az objektumot a garage.ProcessCars() metódusnak." Mint a leg-  
washCar() metódus címét a car.CarMaintenancedelegat objektumba, és ad-  
ez gyakorlatilag a következőt jelenti: "Illesszük be a servicedDepartment.  
// Mossuk le az összes piszkos autót.  
g.ProcessCars(new Car.CarMaintenancedelegat(sd.WashCar));

létrehozásával kezdődik. Ha a következőket írjuk:

A main() metódus a garage és a servicedDepartment típusok egy példányának

## Az átírányító kód elemzése

11.5. ábra: Továbbpárosszolás





Ehhez hasonlóan, ha a következőket mondjuk:

```
// Forgassuk a gumtábroncsokat.  
g.ProcessCars(new Car.CarMaintenancedelegatere(sd.Rotaterires));  
  
akkor a ProcessCars() az alábbiak szerint értelmezhető:  
  
// A Cardelegatere a Servicedepartment.Rotaterires függvényre mutat:  
public void ProcessCars(Car.CarMaintenancedelegatere proc)  
{  
    foreach(Car c in theCars)  
        proc(c); // proc(c) => Servicedepartment.Rotaterires(c)  
    ...  
}
```

**Forráskód** Az Carcarage projekt megtalálható a 11. fejezet alkönyvtárában.

Eddig tehát három példát láttunk arra, hogyan működik a metódusreferencia. Az első példa (simplifiedelegatere) bemutatja a metódusreferencia-típus definíciójának és kezelésének alapjait. A metódusreferencia egy rendszeren metódusok listáját kezeli, amelyeket később az Invoke() metódus (implicit vagy explicit) hívásával érhetünk el. A második példánkban (Cardelegatere) bemutatjuk, hogyan használható a metódusreferencia egy eseményarchitektúra létrehozására (a .NET-interfész típusok helyett). A harmadik példában pedig kiderült, hogy a metódusreferencia használható metódusparaméterként, épp úgy, mint minden más .NET-típus.

**Megjegyzés** A .NET-gyűjteménytípusokban észrevehetjük, hogy általában metódusreferenciát adnak át paraméterként, amikor az elemeket rendezik vagy szűrik.

Ezek a példák együttesen bemutatják a metódusreferencia-típus leggyakoribb aspektusait. A következőkben a kovariancia szerepével ismerkedünk meg.

## Metódusreferenciák kovarianciája

Az eddig létrehozott metódusreferenciák mindegyike olyan metódusokra mutat, amelyek visszatérési értéke numerikus adattípus (vagy void). Tegyük fel, hogy van egy parancssori alkalmazásunk Delegetecovariance névvel, és ez olyan metódusreferenciát definiál, amely valamilyen egyedi osztálytípust visszaadó metódusokra is mutathat (az új projekteknek a Car/Radio típusdefini-ciókat is tartalmaznia kell):

```
// Metódusreferenciát definiálunk, amely Car típust visszaadó
// metódusokra mutat.
public delegate Car ObtainCarDelegate();
```

A metódusreferenciát – elvárható módon – az eddigiekhez hasonlóan adhatjuk meg:

```
class Program
{
    public delegate Car ObtainCarDelegate();

    public static Car GetBasicCar()
    { return new Car("Zippy", 150, 50, false, false); }

    static void Main(string[] args)
    {
        ObtainCarDelegate targetA = new ObtainCarDelegate(GetBasicCar);
        Car c = targetA();
        Console.WriteLine("obtained a {0}", c);
        Console.ReadLine();
    }
}
```

De mi a helyzet akkor, ha a car típusból származtatunk egy sportscar nevű új osztályt, és szeretnénk egy olyan metódusreferencia-típust létrehozni, amely az új típust visszaadó metódusokra mutathat? A .NET 2.0 verzióját megelőzőben ilyenkor kénytelenek lettünk volna létrehozni egy teljesen új metódusreferenciát, ugyanis a metódusreferenciák *amennyire* típusbiztosak voltak, hogy a származtatás alapjait sem fogadták el:

```
// Sportscar típust visszaadó célokra mutató új metódusreferencia.
public delegate Sportscar ObtainSportscarDelegate();
```

Mint ahogy már két metódusreferencia-típusunak van, két példányt kell létrehozniunk a car és sportscar típusok kezeléséhez:

```
class Program
{
    public delegate Car ObtainCarDelegate();
    public delegate Sportscar ObtainSportscarDelegate();

    public static Car GetBasicCar()
    { return new Car(); }

    public static Sportscar GetSportscar()
    { return new Sportscar(); }
}
```

```

static void main(string[] args)
{
    Car c = targetA();
    obtainCarDelegate targetA = new ObtainCarDelegate(GetBasicCar);
    Console.WriteLine("obtained a {0}", c);

    ObtainSportsCarDelegate targetB =
        new ObtainSportsCarDelegate(GetSportsCar);
    SportsCar sc = targetB();
    Console.WriteLine("obtained a {0}", sc);
    Console.ReadLine();
}
}

```

A klasszikus öröklődés szabályait figyelembe véve, ideális lenne egyetlen olyan metódusreferencia-típust létrehozni, amely car és sportscar visszatérési értékekkel is mutathat metódusokra (végül is a sportautó „az egy” autó). A ko-variancia (amely *tömör metódusreferencia* [relaxed delegates] néven is ismert) ezt teszi lehetővé. Vagyis a kovariancia olyan metódusreferencia létrehozását biztosítja, amely klasszikus öröklessel kapcsolódó osztálytípusokat visszaadó metódusokra mutathat:

```

class Program
{
    // definiálunk egyetlen metódusreferenciát, amely Car vagy
    // SportCar típust adhat vissza.
    public delegate Car ObtainVehicleDelegate();

    public static Car GetBasicCar()
    { return new Car(); }

    public static SportsCar GetSportsCar()
    { return new SportsCar(); }

    static void main(string[] args)
    {
        Console.WriteLine("***** Delegate Covariance *****\n");
        ObtainVehicleDelegate targetA =
            new ObtainVehicleDelegate(GetBasicCar);
        Car c = targetA();
        Console.WriteLine("obtained a {0}", c);

        // A kovariancia lehetővé teszi ezt a célhozzárendelést.
        ObtainVehicleDelegate targetB =
            new ObtainVehicleDelegate(GetSportsCar);
        SportsCar sc = (SportsCar)targetB();
        Console.WriteLine("obtained a {0}", sc);
        Console.ReadLine();
    }
}

```

Az `obtainDelegatedMethodReference`-típust úgy definiáltuk, hogy erősen típusított `car`-típust visszaadó metódusokra mutathasson. A kovariancia azonban mutathat származtatott típusokat visszaadó metódusokra is. A származtatott osztályok tagjainak eléréséhez explicit kasztolást kell használni.

**Megjegyzés** Hasonlóképpen a kovariancia lehetőségét nyújt egyetlen metódusreferencia létrehozására is, amely olyan metódusokra mutat, amelyek klasszikus származtatással kapcsolódó objektumokat kapnak meg. További információkért lapozzuk fel a .NET Framework 3.5 SDK dokumentációját.

**Forráskód** Az `DelegateCovariance` projekt megtalálható a 11. fejezet alkönyvtárában.

## Generikus metódusreferencia készítése

A C# lehetőséget ad generikus metódusreferencia-típus létrehozására (lásd az előző fejezetet). Tegyük fel például, hogy definiálunk egy metódusreferenciát, amely bármely, `void` visszatérési értékű és egy bemenő paramétert felvevő metódust meg tud hívni. Ha a kérdéses argumentum különbözhet, akkor ezt a pusztaméret használataval modellezhetjük. Ehhez vegyük a következő kódrészletet egy új, `genericDelegate` nevű parametrsort alkalmazásban:

```
namespace GenericDelegate
{
    // Ez a generikus metódusreferencia bármely metódust meghívhatja,
    // amely void értékkkel tér vissza, és egyetlen bemenő paraméterre van.
    public delegate void MyGenericDelegate<T>(T arg);

    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("**** Generic Delegates ****\n");

            // Céllok bejegyzése
            MyGenericDelegate<string> strTarget =
                new MyGenericDelegate<string>(StringTarget);
            strTarget("Some string data");

            MyGenericDelegate<int> intTarget =
                new MyGenericDelegate<int>(IntTarget);
            intTarget(9);
            Console.ReadLine();
        }
    }
}
```

```

static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}

static void IntTarget(int arg)
{
    Console.WriteLine("++arg is: {0}", ++arg);
}
}
}

```

A `MyGenericDelegate<T>` definiálja azt a `T` típusparamétert, amellyel egy konkrét módszerreferencia felparaméterezhető. Ha ebből a típusból hozunk létre egy példányt, akkor meg kell adnunk a típusparaméter értéket, valamint annak a módszernek a nevét, amelyet a módszerreferencia hívni fog. Így, ha megadunk egy sztringtípust, akkor sztringértéket küldünk a célmóduśnak:

```

// MyGenericDelegate<T> példány létrehozása
// sztringtípus-paraméterrel.
MyGenericDelegate<string> stringTarget =
    new MyGenericDelegate<string>(StringTarget);
stringTarget("some string data");

```

Adott az `stringTarget` objektum, így a `StringTarget()` módszer egyetlen sztring-típust vár:

```

static void StringTarget(string arg)
{
    Console.WriteLine("arg in uppercase is: {0}", arg.ToUpper());
}

```

## A generikus módszerreferencia szimulációja generikus típusok nélkül

A generikus módszerreferencia rugalmasan biztosítja, hogy a meghívandó módszert típusbiztosan hívhassuk. A generikus típusok bevezetését megelőzően (a .NET 2.0 megjelenése előtt) hasonló eredményt lehetett elérni a generikus `System.Object` használatával:

```
public delegate void MyDelegate(object arg);
```

Bár ez bármilyen típusú adat elküldését lehetővé teszi a metódusreferencia-célnak, ám típusbiztonság nélkül és lehetséges dobozolósi hátrányokkal. Tegyük fel például, hogy létrehoztunk két példányt a `MyDelegate` típusból, és mindkettő ugyanarra a `MyTarget` metódusra mutat. Figyeljük meg a dobozolósi/kidobozolósi hátrányokat és a típusbiztonság hiányát:

```
class Program
{
    static void Main(string[] args)
    {
        ...
        // Jegyezzük be a célt a „tradicionális” metódusreferencia-
        // szintaxissal.
        MyDelegate d = new MyDelegate(MyTarget);
        d("More string data");

        // Metóduscsoport-átalakítás szintaxisa (a fejezetben később
        // részletesen elmagyarázzuk).
        MyDelegate d2 = MyTarget;
        d2(9); // Dobozolósi hátrány.
        Console.ReadLine();
    }

    // A típusbiztonság hiánya miatt meg kell határoznunk
    // az alapul szolgáló típust kasztolás előtt.
    static void MyTarget(object arg)
    {
        if(arg is int)
        {
            int i = (int)arg; // Kidobozolósi hátrány.
            Console.WriteLine(++arg is: {0}", ++i);
        }
        if(arg is string)
        {
            string s = (string)arg;
            Console.WriteLine(arg in uppercase is: {0}", s.ToUpper());
        }
    }
}
```

Ha érték típust küldünk a célnak, az érték bedobozolódik, majd kidobozolódik, amikor a célmetódus megkapja. Mivel a bejövő paraméter bármi lehet, ezért kasztolás előtt dinamikusan ellenőrizni kell az alaptípust. Generikus metódusreferencia használatakor a kívánt flexibilitást a „problémák” nélkül is elérhetjük.

Ezzel alapjaiban áttekintettük a .NET metódusreferencia-típusát. (A további részleteket lásd a 18. fejezetben a többszálúság vizsgálatakor.) A következőkben a C# event kulcsszavával folytatjuk.

## C#-események

A metódusreferenciák lehetővé teszik a memóriában lévő objektumok közötti kétirányú kommunikációt. A metódusreferencia alkalmazása azonban formális kódok használatával is jár (metódusreferencia definíciója, szükségesség tag-változók deklarálása, egyedi beregisztráló/kiregisztráló metódusok a be-

ágyazás megtartásához stb.).

A gépelési időtől eltekintve, egy másik probléma is adódik akkor, ha a nyers kódban metódusreferenciát használunk az alkalmazásunk callback mechanizmusaként. Ha ugyanis nem privátként definiáljuk az osztály metódusreferencia-tagválasztóit, akkor a hívó közvetlenül elérheti a metódusreferencia-objektumot. Ekkor a hívó képes lehet egy másik metódusreferencia-objektumhoz rendelni a választót (gyakorlatilag kitörölve a pillanatnyi hívási listát), és ami még rosszabb, képes lehet közvetlenül aktivizálni a metódusreferencia hívási listáját. A probléma bemutatásához vegyük az előző CarDelegate példát újrafeldolgozását (és egyszerűsítését):

```
public class Car
{
    // Egyetlen metódusreferencia
    public delegate void Explored(string msg);

    // Most nyilvános! Nincs többé segédmetódus!
    public Explored exploredList;

    // Vátsuk ki az Explored üzenetet.
    public void Accelerate(int delta)
    {
        if (exploredList != null)
        {
            exploredList("Sorry, this car is dead...");
        }
    }
}
```

Ekkor a privát metódusreferencia-tagválasztót nem ágyazzuk be saját regisztrációs metódusai. Mivel ezek a tagválasztók valóban nyilvánosak, a hívó közvetlenül el tudja érni az exploredList tagot, és újra hozza tudja rendelni az új explored objektumokhoz, továbbá akkor hívja meg a metódusreferenciát, amikor akarja:

## Az event kulcsszó

A C# event kulcsszavát használhatjuk arra, hogy olyan saját egyedi metódusokat, valamint a metódusreferencia-típusokhoz szükséges tagválto-event kulcsszót, automatikusan megkapjuk a bejegisztráló és kiregisztráló hozzá, illetve távolítanak el metódusokat. Amikor a fordító feldolgozza az sokat hozzunk létre, amelyek a metódusreferencia hívási listájához adnak sokat hozzunk létre, amelyek a metódusreferencia hívási listájához adnak

A nyilvános metódusreferencia-tagok megtörlik az egyébségbe zárást, és így nemcsak nehezen karbantartható (és nehezen hibakereshető) kódhoz vezet- nek, de még potenciális biztonsági réseket is nyitnak. Nyilvánvalóan nem szeretnénk lehetőséget adni más alkalmazásoknak, hogy módosíthassák azt, hogy egy metódusreferencia mire mutasson, illetve hogy engedély nélküli meghívják a tagokat.

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Aghi No Encapsulation! *****\n");
        // Hozzunk létre egy autót.
        Car myCar = new Car();
        // Közvetlen hozzáférésünk van a metódusreferenciához!
        myCar.explodedList = new Car.Exploded(CalWhenExploded);
        myCar.Accelerate(10);
        // Hozzárendelhetjük egy teljesen új objektumhoz...
        myCar.explodedList = new Car.Exploded(CalWhenExploded);
        // elég zavarba ejtő.
        myCar.explodedList = new Car.Exploded(CalWhenExploded);
        myCar.Accelerate(10);
        // A hívó közvetlenül meghívhatja a metódusreferenciát!
        myCar.explodedList.Invoke("hee, hee, hee...");
    }
}
```



jük, és ennek megfelelően az eseményt kiváltó objektum közveletlenül nem fér hozzájuk. Az event kulcsszó nem pusztán egy kis szintaktikai egyszerűsítés, amely némi gépéleléstől kímél meg minket.

Egy esemény definiálása kétlépcsős folyamat. Először egy módszerre-renciát kell definiálnunk, amely tárolni fogja az esemény bekövetkeztékor meghívandó metódusok listáját. Azután deklaráljuk az eseményt (a C# event kulcsszavával) a kapcsolódó metódusreferenciában.

Az event kulcsszó bemutatására nézzük meg az alábbi példát. A car típus iterációja két eseményt definiál (azonos néven, mint az előző AboutOblow és exploded metódusreferenciák). Ezeket az eseményeket egyetlen, carevent-handler nevű metódusreferenciához kötik. A car típus első változtatásai a következők:

```
public class Car
{
    // Ez a metódusreferencia a car osztály eseményeivel
    // működik együtt.
    public delegate void CareventHandler(string msg);
    // Ez az autó küldi ezeket az eseményeket.
    public event CareventHandler Exploded;
    public event CareventHandler AboutOblow;
    ...
}
```

Egy esemény küldése a hívóhoz nem más, mint nevének és a szükségessé para-métereknek a megadása, ahogyan ezt a hozzátartozó metódusreferencia defi-niálja. Ahhoz, hogy megbizonyosodjunk róla, hogy a hívó regisztrálása az eseményhez tényleg megtörtént, az eseményt ellenőrizni kell null értékre, mie-lőtt a metódusreferencia metódusait meghívjuk. A car osztály Accelerate() metódusának új iterációja a következő:

```
public void Accelerate(int delta)
{
    // Ha az autó tönkrement, akkor kiváltjuk az Exploded eseményt.
    if (CarIsDead)
    {
        if (Exploded != null)
            Exploded("Sorry, this car is dead...");
    }
    else
    {
        currspeed += delta;
    }
}
```

```
// Majdnem tönkrement?
if (10 == maxSpeed - currSpeed
    && AboutToBlow != null)
{
    AboutToBlow("careful buddy! gonna blow!");
}
else
{
    console.WriteLine(">CurrSpeed = {0}", currSpeed);
}
}
```

Ezzel úgy állítottuk be a car osztályt, hogy két egyedi eseményt küldjön anélkül, hogy egyedi beregisztráló függvényeket kellett volna definiálnunk, vagy metoduserferencia-tagváltozókat kellett volna deklarálnunk. Az új autó használatainak ellenőrzése előtt nézzük meg az esemény szerkezetét kicsit részletesebben.

## Eseménykezelés a felszín alatt

Egy C#-esemény valójában két rejtett nyilvános módszerre bővíthető, az egyik az `add_`, a másik a `remove_` prefixummal rendelkezik. A prefixumot a C#-esemény neve követi. Az `exploded` esemény eredménye például két CIL-metódus: az `add_exploded()` és a `remove_exploded()`. Az `add_xxx()` és a `remove_xxx()` metódusok mellett a CIL-szintű eseménydefiniáció a megfelelő metoduserferenciát rendelí hozzá az adott eseményhez.

Ha megnéznénk az `add_AboutToExplode()` metódus mögötti CIL-utasítást, akkor olyan kódot találunk, amely nagyon hasonlít ahhoz az `onAboutToBlow()` segédmetódushoz, amelyet előzőleg a `carDelegate` példában írtunk (figyeljünk a `delegate.Combine()` hívásra):

```
.method public hidebysig specialname instance void
    add_AboutToBlow(class CarEvents car/Careventhandler 'value')
{
    cil managed synchronized
        .maxstack 8
        ldarg.0
        ldarg.0
        ldarg.1
        ldarg.1
        CarEvents.Careventhandler::AboutToBlow
        call class [mscorlib]System.Delegate
```

```
[mscorlib]System.Delegate::Combine(
    class [mscorlib]System.Delegate,
    class [mscorlib]System.Delegate,
    class [mscorlib]System.Delegate)
    castclass CarEvents.Car/CarEventHandler
    stfld class CarEvents.Car/CarEventHandler
    CarEvents.Car::AboutToBlow
    ret
}
```

A `remove_AboutToBlow` indirekt módon hívja a `Delegate.Remove()` metódust, és többé-kevésbé azonos az előző `remove_AboutToBlow()` segédmetódussal:

```
.method public hidebysig specialname instance void
    remove_AboutToBlow(class CarEvents.Car/CarEventHandler 'value')
{
    cil managed synchronized
```

```
    ,maxstack 8
    ldarg.0
    ldarg.0
    ldftld class CarEvents.Car/CarEventHandler
    CarEvents.Car::AboutToBlow
    ldarg.1
    call class [mscorlib]System.Delegate
    [mscorlib]System.Delegate::Remove(
    class [mscorlib]System.Delegate,
    class [mscorlib]System.Delegate,
    class [mscorlib]System.Delegate)
    castclass CarEvents.Car/CarEventHandler
    stfld class CarEvents.Car/CarEventHandler
    CarEvents.Car::AboutToBlow
    ret
}
```

Végül pedig magát az eseményt leíró CIL-kód a `.addon` és a `.removeon` direktívákat használja a megfelelő meghívandó `add_xxx()` és `remove_xxx()` metódusok nevének leképezéséhez:

```
.event CarEvents.Car/EngineHandler AboutToBlow
{
    .addon void CarEvents.Car::AddAboutToBlow
    (class CarEvents.Car/CarEngineHandler)
    .removeon void CarEvents.Car::RemoveAboutToBlow
    (class CarEvents.Car/CarEngineHandler)
}
```

Mind Ezek után a következő kérdés az, hogy hogyan „figyeljük” a hívóoldalon a bejövő eseményeket.

## A bejövő események figyelése

A C#-események egy szervertől a hívóoldali eseménykezelők beregisztrálásáig is. Ahelyett, hogy saját segédmetódusokat kellene definiálnunk, a hívó közvetlenül a += és a -= operátorokat használja (amelyek a megfelelő ad-xxx() vagy remove-xxx() metódusokat váltják ki a háttérben). Ha egy eseményhez egy eseménykezelőt szeretnénk regisztrálni, akkor a következő lépésminit kell követnünk:

```
// ObjectVariable.EventName +=
// new AssociatedDelegate(functionToCall);
Car.EngineHandler d = new
Car.EventHandler(Car.ExploredEventHandler)
myCar.Explored += d;
```

Ha le szeretnénk választani az eseményforrást, akkor a -= operátort használhatjuk:

```
// ObjectVariable.EventName -= delegateObject;
myCar.Explored -= d;
```

Adottak ezek az előre jelezhető minták, így a main() metódus átirása a C#-esemény regisztrációs szintaxisának használatára a következő:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("***** Fun with Events *****\n");
        Car c1 = new Car("Stingbug", 100, 10);
        // Eseménykezelők regisztrálása.
        c1.AboutToBlow += new Car.EventHandler(Car.IsAlmostDoomed);
        c1.AboutToBlow += new Car.EventHandler(Car.AboutToBlow);
        Car.EventHandler d = new Car.EventHandler(Car.Explored);
        c1.Explored += d;
        Console.WriteLine("***** Speeding up *****");
        for (int i = 0; i < 6; i++)
            c1.Accelerate(20);
        // Távolság el a Car.Explored metódust
        // a hívást listából.
        c1.Explored -= d;
    }
}
```

```

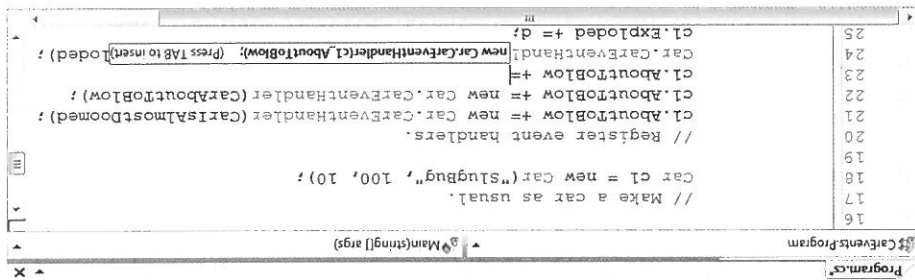
Console.WriteLine("\n***** Speeding up *****");
for (int i = 0; i < 6; i++)
    cl.Accelerate(20);
Console.ReadLine();
}

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

```

## Esemény bevezetése a Visual Studio 2008 használatával

A Visual Studio 2008 segítségével nyújt az eseménykezelők bevezetését a következők szerint. Ha egy esemény bevezetését akarjuk a szintaxis használatával, akkor egy `IntelliSense` ablak jelenik meg, lehetőséget adva arra, hogy a `Tab` billentyű megnyomásával automatikusan beillesztjük a hozzárendelt metódusreferencia-példányt (lásd a 11.6. ábrát).



11.6. ábra: `IntelliSense` metódusreferencia kiválasztása

Ha lenyomjuk a `Tab` billentyűt, akkor beírhatjuk az eseménykezelő nevét (vagy egyszerűen elfogadhatjuk az alapértelmezett), ahogyan a 11.7 ábrán látható. Ha megint megnyomjuk a `Tab` billentyűt, akkor a metódusreferencia céljához megfelelő formátumú kódvázat kapunk (ez a függvény statikusként deklarált, hiszen az eseményt a statikus `Main()` metódusban regisztráltuk):

```

{
    static void cl_AboutToBlow(string msg)
    {
        // Irjuk be a forráskódunkat!
    }
}

```

sztingus.

Egyszerű eseményeknél közvetlenül az Eventarags egy példányát küldhetjük. De ha saját adatokat szeretnénk küldeni, akkor az Eventaragsból származtatott oszályt kell készítenünk hozzá. Példánkban feltételezünk egy CarEventarags nevű oszályt, amely tartalmazza a fogadónak küldött üzenetet jelképező

```
public class EventArgs
{
    public static readonly System.EventArgs Empty;
    public EventArgs();
}
```

Még egy végső finomítása van a car típus aktuális iterációjának. Ez a Microsoft ajánlott eseménykezelő mintáját tükrözi. Ha megnézzük az alaposítály-könyvtár adott típusa által küldött eseményeket, azt látjuk, hogy a hozzátartozó módszerreferencia első paraméterre system.object típusú, míg a másodikk paraméter a system.eventargs osztályból származik.

A system.object típusú paraméterreferencia az eseményt küldő objektumra (például a carra), a második paraméter az eseményre vonatkozó információkat tartalmaz. A system.eventargs alaposítály olyan eseményt ír le, amely nem küld semmilyen egyedi információt:

## Egy „korrekt” esemény

**Forráskód** Az Carevents projekt megtalálható a 11. fejezet alkönyvtárában.

Ez az IntelliSense funkció a .NET-alapoztatálykönyvtár minden eseményéhez elérhető. Az IDE-funkció igen hatékony az időmegtakarítás szempontjából, ugyanis nincs szükség arra, hogy a .NET-szigorendszerben keressünk ahhoz, hogy a megfelelő metódusreferenciát tudjuk használni az adott eseményhez. Továbbá a metódusreferencia céljának formátumát sem kell keresni.

### 11.7. ábra: Intelligense módszerrel történő cellaformátum

```

public class CarEventArgs : EventArgs
{
    public readonly string msg;
    public CarEventArgs(string message)
    {
        msg = message;
    }
}

```

Ezzel a `CarEventHandler` metódusreferenciát a következő módon módosíthatjuk (az események közben változatlanok maradnak):

```

public class Car
{
    public delegate void CarEventHandler(object sender,
        CarEventArgs e);
    ...
}

```

Ha az eseményünket az `Accelerate()` metóduson belül váltjuk ki, akkor szükségünk van az aktuális autóra vonatkozó referenciára (a `this` kulcsszó-val), valamint a `CarEventArgs` típusunk egy példányára. Példaként nézzük a következő módosítást:

```

public void Accelerate(int delta)
{
    // Ha az autó tönkrement, akkor kiváltjuk az Exploded eseményt.
    if (CarIsDead)
    {
        if (Exploded != null)
            Exploded(this, new CarEventArgs("Sorry, this car is
            dead..."));
    }
    ...
}

```

A hívó oldalon csak annyit kell tennünk, hogy az eseménykezelőnket frissítsük úgy, hogy a bejövő paramétereket fogadja, és az írásvédelettől megszabadul hozzájusson az üzenethez. Például:

```

public static void CarAboutToBlow(object sender, CarEventArgs e)
{
    Console.WriteLine("{0} says: {1}", sender, e.msg);
}

```

A main() metódus használhatja az EventHandler<CarEventArg> metódust, ahol előtte definiáltuk a CarEventHandler-t:

```
public class Car
{
    public event EventHandler<CarEventArg> Exploded;
    public event EventHandler<CarEventArg> AboutToBlow;
    ...
}
```

Mint ahogy sok egyedi metódusreferencia vár objektumot első paraméterként és EventArg leszátmazottat másodikként, így tovább tökéletesíthetjük az előző példákat a generikus EventHandler<T> típus használataival, ahol T a saját EventArg típusunk. Vegyük a következő módosítást a Car típuson (saját metódusreferenciát nem kell készítenünk):

## A generikus EventHandler<T> metódusreferencia

**Forráskód** A PrimandProperCarEvents projekt megtalálható a 11. fejezet alkönyvtárában.

```
public static void CarIsAlmostDoomed(object sender, CarEventArg e)
{
    // Hogy biztosra menjünk, ellenőrizzük
    // futástdőben, kasztolás előtt.
    if (sender is Car)
    {
        Car c = (Car)sender;
        c.CrankTurnes(false);
    }
    Console.WriteLine("Critical Message from {0}: {1}", sender,
        e.Msg);
}
```

Ha a fogadó kommunikálni akar az eseményt küldő objektummal, akkor explicit módon kasztolni kell a System.Object osztályt. Ennek megfelelően, ha ki szeretnénk kapcsolni a rádiót, amikor a Car objektum tönkremegy, akkor a következő eseménykezelőhöz hasonlóan kell készítenünk:

11. fejezet: Metódusreferenciák, események és lambdák



```

static void Main(string[] args)
{
    Console.WriteLine("***** Prim and Proper Events *****\n");

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

    // Eseménykezelők regisztrálása.
    c1.AboutToBlow +=
        new EventHandler<CarEventArgs>(CarIsAlmostDoomed);
    c1.AboutToBlow += new EventHandler<CarEventArgs>(CarAboutToBlow);

    EventHandler<CarEventArgs> d =
        new EventHandler<CarEventArgs>(CarExploded);
    c1.Exploded += d;

    ...
}

```

---

**Forráskód** Az `PrimAndProperCarEvents` (Generic) projekt megtalálható a 11. fejezet al-könyvtárában.

---

## Névtelen metódusok a C#-ban

Tradicionálisan, ha egy hívónak figyelnie kell a bejövő eseményeket, akkor definiálnia kell egy egyedi metódust, amelynek szignatúrája megegyezik a hozzárendelt metódusreferenciájával. Például:

```

class Program
{
    static void Main(string[] args)
    {
        SomeType t = new SomeType();

        // Tegyük fel, hogy a "SomeDelegate" olyan metódusokra mutathat,
        // amelyek nem várnak paramétert, és void a visszatérési
        // értékük.
        t.SomeEvent += new SomeDelegate(MyEventHandler);
    }

    // Típikusán csak a SomeDelegate objektum hívja.
    public static void MyEventHandler()
    {
        // Csinaljunk valamit az esemény bekövetkeztekor.
    }
}

```

A `MyEventHandler()` metódushoz hasonló metódusokat ritkán szeretnénk a program más részéből hívni, mint ahonnan a hívó metódusreferencia teszi. Amíg a produktivitást is figyelembe vesszük, addig kicsit kényelmelen (bár nem gállo tényező), hogy manuálisan definiálni kell egy olyan külön metódust, amelyet a metódusreferencia meghív.

Lehetőség van arra is, hogy egy kódblokkot közvetlenül a metódusreferenciához rendeljünk az esemény regisztrálásában. Formálisan az ilyen kódot *névtelen metódusnak* nevezzük. A szintaxis bemutatásához nézzük meg a következő `main()` metódust, amely a `car` típus által küldött eseményeket névtelen metódussal kezel, nem pedig specifikusan elnevezett eseménykezelővel:

```
class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("**** Anonymous Methods ****\n");
        Car c1 = new Car("Stugbug", 100, 10);

        // Regisztráljuk az eseménykezelőket névtelen metódusként.
        c1.AboutToBlow += delegate {
            Console.WriteLine("Eeki going too fast!");
        };
        c1.AboutToBlow += delegate(object sender, CarEventArgs e) {
            Console.WriteLine("Message from car: {0}", e.msg);
        };
        c1.Explored += delegate(object sender, CarEventArgs e) {
            Console.WriteLine("Fatal Message from car: {0}", e.msg);
        };
        ...
    }
}
```

---

**Megjegyzés** A névtelen metódus végén a kapcsos zárójel után pontosvesszőt kell tenni. E nélkül fordítási hibát kapunk.

---

A Program típus nem definiál más statikus eseménykezelőket, mint a `car.AboutToBlow()` vagy a `car.Explored()`. Ehelyett a névtelen függvények akkor definiálódnak, amikor a hívó `+=` szintaxissal megadott eseményt kezel. A névtelen metódusok alapszintaxisa a következőknek felel meg: