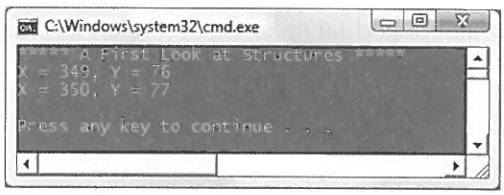
Megjegyzés Általában nem tekintik elegánsnak nyilvános adatok definiálását egy osztályon vagy struktúrán belül. Ehelyett érdemes *privát* adatokat definiálni, amelyek a *nyilvános* tulajdonságok használatával férhetők hozzá, valamint módosíthatók. Ezekről részletesebben az 5. fejezetben lesz szó.

A `Main()` metódus teszteli a `Point` típusunkat. A 4.9. ábra mutatja a program kimenetét.

```
static void Main(string[] args)
{
  Console.WriteLine("***** A First Look at Structures *****");
  // A kiindulási Point változó létrehozása.
  Point myPoint;
  myPoint.X = 349;
  myPoint.Y = 76;
  myPoint.Display();

  // Az X és Y értékek beállítása.
  myPoint.Increment();
  myPoint.Display();
  Console.ReadLine();
}
```



4.9. ábra: *A Point struktúra működés közben*

Struktúraváltozók létrehozása

Ha struktúraváltozót szeretnénk megalkotni, több lehetőség közül választhatunk. Az alábbiakban egyszerűen egy `Point` változót hozunk létre, majd minden nyilvános adatmezőhöz hozzárendelünk egy értéket, mielőtt meghívnánk a tagjait. Ha *nem* rendelünk az egyes részekhez értéket (jelen esetben X-et és Y-t) a struktúra használata előtt, akkor fordítási hibát kapunk:

```
// Hiba! Az Y nem kapott értéket.
Point p1;
p1.X = 10;
p1.Display();
```

```
// OK! Mindkét mezőt hozzárendeltük a használat előtt.
Point p2;
p2.X = 10;
p2.Y = 10;
p2.Display();
```

Alternatívaként a C# new kulcsszavának használatával is létrehozhatunk struktúraváltozókat, és ez meg fogja hívni a struktúra *alapértelmezett konstruktorát*. Az alapértelmezett konstruktor, természetéből adódóan, nem kaphat bemeneti paramétert. Az alapértelmezett konstruktor aktivizálásának előnye az, hogy minden mező automatikusan az alapértelmezett értékre lesz állítva:

```
// Az alapértelmezett konstruktor segítségével a mezőket
// alapértelmezett értékre állítjuk.
Point p1 = new Point();

// A következőket írja ki: X=0,Y=0.
p1.Display();
```

Struktúrát *egyedi konstruktor* használatával is lehet tervezni. Ez lehetővé teszi, hogy a változó létrehozásakor megadjuk az adatmezők értékeit ahelyett, hogy mezőről mezőre kéne beállítanunk az egyes adattagokat. A példa kedvéért frissítsük a Point struktúrát az alábbi kóddal (a konstruktorokat lásd az 5. fejezetben):

```
struct Point
{
  // A struktúra mezői.
  public int X;
  public int Y;

  // Egyedi konstruktor.
  public Point(int XPos, int YPos)
  {
    X = XPos;
    Y = YPos;
  }
  ...
}
```

Ezzel most már a következőképpen hozhatunk létre Point típusokat:

```
// Meghívjuk az egyedi konstruktort.
Point p2 = new Point(50, 60);

// A következőket írja ki: X=50,Y=60.
p2.Display();
```

A felszínen tehát viszonylag könnyű a struktúrákkal dolgozni. A típus mélyebb megértéséhez meg kell azonban vizsgálnunk a különbséget a .NET-értéktípus és .NET-referenciatípus között.

Forráskód A `FunWithStructures` projekt megtalálható a 4. fejezet alkönyvtárában.

Érték- és referenciatípusok

Megjegyzés Az érték- és referenciatípusok alábbi vizsgálata azt feltételezi, hogy rendelkezünk objektumorientált programozási háttérrel. Ha ez nem így van, akkor ehhez a részhez az 5. és a 6. fejezet elolvasása után térjünk vissza!

A tömbökkel, a sztringekkel és a felsorolásokkal ellentétben a C#-struktúrák nem rendelkeznek ugyanúgy elnevezett reprezentációval a .NET-könyvtárban (vagyis nincsen `System.Structure` osztály), hanem a `System.ValueType` típusból származnak. A `System.ValueType` szerepe annak biztosítása, hogy a származott típust (pl. bármilyen struktúra) a rendszer a *veremben* foglalja le, nem pedig a szemétgyűjtött *heapben*.

A `System.ValueType` egyetlen célja az, hogy „felüldefiniálja" a `System.Object` által meghatározott virtuális metódusokat, és így ne referencia-, hanem értékalapú szemantikát használjon. A felüldefiniálás az ősosztályon belül meghatározott virtuális (vagy absztrakt) metódus implementációjának a módosítása. A `ValueType` ősosztálya a `System.Object`. A `System.ValueType` által meghatározott példánymetódusok valójában hasonlóak a `System.Object` által meghatározottakhoz:

```
// A struktúrák és felsorolások bővítik a System.ValueType típust.
public abstract class ValueType : object
{
  public virtual bool Equals(object obj);
  public virtual int GetHashCode();
  public Type GetType();
  public virtual string ToString();
}
```

Mivel az értéktípusok értékalapú szemantikát használnak, a struktúra (amely magában foglalja az összes numerikus adattípust [`int`, `float` stb.], valamint felsorolásokat vagy egyedi struktúrákat) élettartama előre kiszámítható. Amikor egy struktúraváltozó kikerül a meghatározó hatókörből, a rendszer azonnal eltávolítja a memóriából:

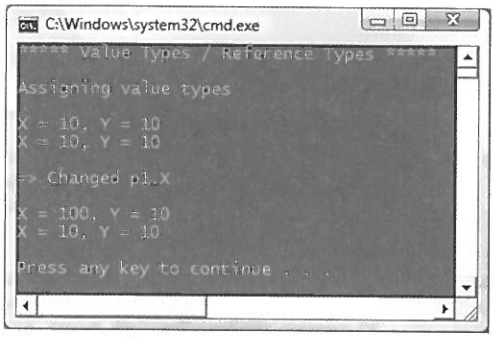
```
// A lokális struktúrákat a rendszer eltávolítja a veremből
// amikor a metódus visszatér.
static void LocalValueTypes()
{
// Figyelem! Az "int" valójában System.Int32 struktúra.
int i = 0;

// Figyelem! A Point valójában struktúratípus.
Point p = new Point();
} // "i" és "p" kikerül a veremből!
```

Érték- és referenciatípusok, valamint az értékadó operátor

Amikor egy értéktípust egy másikhoz rendelünk hozzá, akkor az adatmezők tagról tagra történő másolása következik be. Az olyan egyszerű adattípusok esetében, mint a `System.Int32`, az egyetlen másolandó tag a numerikus érték. A `Point` esetében azonban az X és az Y értékek átmásolással az új struktúraváltozóba kerülnek. Példaként hozzunk létre egy új parancssorialkalmazás-projektet `ValueAndReferenceTypes` néven, majd másoljuk az előző `Point` definíciót az új névtérbe. Azután adjuk hozzá a következő metódust a `Program` típushoz:

```
// Két belső értéktípus hozzárendelésének eredménye
// két független változó a veremben.
static void ValueTypeAssignment()
{
  Console.WriteLine("Assigning value types\n");

  Point p1 = new Point(10, 10);
  Point p2 = p1;

  // Mindkét pont kiírása.
  p1.Display();
  p2.Display();

  // p1.X módosítás és ismételt kiírása. p2.X nem változik.
  p1.X = 100;
  Console.WriteLine("\n=> Changed p1.X\n");
  p1.Display();
  p2.Display();
}
```

Létrehoztuk a `Point` típus egy változóját (p1 néven), majd hozzárendeltük egy másik `Point` típushoz (p2). Mivel a `Point` egy értéktípus, ezért a `MyPoint`-nak két másolata van a veremben, és ezeket egymástól függetlenül tudjuk kezelni. Ezért, amikor megváltoztatjuk a p1.X értékét, a p2.X értéke nem módosul. A 4.10. ábra mutatja azt a kimenetet, amelyet akkor kapunk, ha meghívjuk ezt a metódust a `Main()` metódusból.



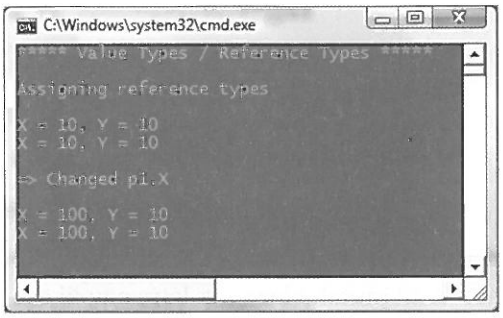
4.10. ábra: *Értéktípusok hozzárendelése minden mező szó szerinti másolatát eredményezi*

Éles ellentétben az értéktípusokkal, ha referenciatípusokra (ideértve az összes osztálypéldányt) alkalmazzuk az értékadó operátort, akkor ezt oda irányítjuk át, ahova a referenciaváltozó a memóriában mutat. Hozzunk létre egy új osztálytípust `PointRef` néven, amely pontosan ugyanazokkal a tagokkal rendelkezik, mint a `Point` struktúra, és nevezzük át a konstruktort, hogy az egyezzen az osztály nevével:

```
// Az osztályok mindig referenciatípusok.
class PointRef
{
  // Ugyanazon tagokkal rendelkezik, mint a Point struktúra.

  // A konstruktor nevét módosítsuk PointRef névre!
  public PointRef(int XPos, int YPos)
  {
    X = XPos;
    Y = YPos;
  }
}
```

Ezután használjuk a `PointRef` típust a következő új metóduson belül (figyeljük meg, hogy a kód hasonló a `ValueTypeAssignment()` metódushoz). Feltételezve, hogy meghívtuk ezt az új metódust a `Main()` metóduson belül, a kimenetnek úgy kell kinéznie, ahogy a 4.11. ábrán látható.

```
static void ReferenceTypeAssignment()
{
  Console.WriteLine("Assigning reference types\n");
  PointRef p1 = new PointRef(10, 10);
  PointRef p2 = p1;

  // Mindkét referencia kiírása.
  p1.Display();
  p2.Display();

  // p1.X módosítása és ismételt kiírása.
  p1.X = 100;
  Console.WriteLine("\n=> Changed p1.X\n");
  p1.Display();
  p2.Display();
}
```



4.11. ábra: *A referenciatípusok értékadása átmásolja a referenciát*

Ebben az esetben két referenciánk mutat *ugyanarra* az objektumra a felügyelt heapben. Ezért, amikor módosítjuk az X értékét a p2 referencia használatával, akkor a p1.X is változik, hiszen ugyanazt az értéket jelenti.

Referenciatípusokat tartalmazó értéktípusok

Az érték- és referenciatípusok közötti alapvető különbségek tisztázása után vizsgáljunk meg egy összetettebb példát. Tegyük fel, hogy van egy olyan referencia- (osztály-) típusunk, amely rendelkezik egy olyan információs sztringgel, amelyet egyedi konstruktor használatával állíthatunk be:

```
class ShapeInfo
{
  public string infoString;
  public ShapeInfo(string info)
  { infoString = info; }
}
```

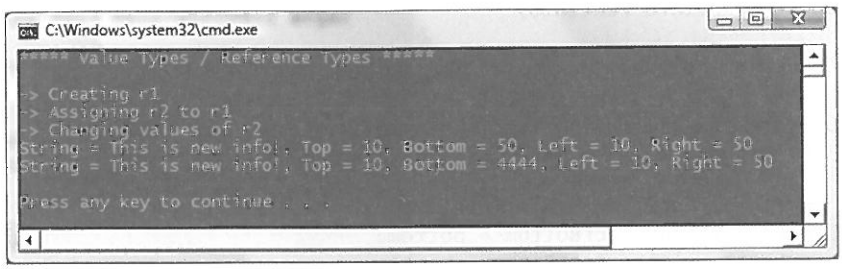
Ezután tételezzük fel, hogy ennek az osztálytípusnak egy változóját szeretnénk belefoglalni egy `Rectangle` nevű érték típusba. Hogy a hívó beállíthassa a belső `ShapeInfo` tagváltozó értékét, megadhatunk egy egyedi konstruktort is. A `Rectangle` típus teljes definíciója a következő:

```
struct Rectangle
{
  // A Rectangle struktúra referenciatípus tagot foglal magában.
  public ShapeInfo rectInfo;

  public int rectTop, rectLeft, rectBottom, rectRight;

  public Rectangle(string info, int top, int left, int bottom,
    int right)
  {
    rectInfo = new ShapeInfo(info);
    rectTop = top; rectBottom = bottom;
    rectLeft = left; rectRight = right;
  }

  public void Display()
  {
    Console.WriteLine("String = {0}, Top = {1}, Bottom = {2}," +
      "Left = {3}, Right = {4}",
      rectInfo.infoString, rectTop, rectBottom, rectLeft,
      rectRight);
  }
}
```

Az értéktípus ekkor már tartalmaz egy referenciatípust. A kérdés: mi történik, ha egy `Rectangle` változót hozzárendelünk egy másikhoz? Az a helyes feltételezés, hogy a számadatnak (amely valójában egy struktúra) független entitásnak kell lennie minden `Rectangle` változó számára. De mi a helyzet a belső referenciatípussal? Az objektum állapotát vagy az objektum hivatkozását is átmásolja a rendszer? A kérdés megválaszolásához definiáljuk az alábbi metódust, és hívjuk meg a `Main()` metódusból. A 4.12. ábra tartalmazza a választ.

```
static void ValueTypeContainingRefType()
{
  // Az első Rectangle létrehozása.
  Console.WriteLine("-> Creating r1");
  Rectangle r1 = new Rectangle("First Rect", 10, 10, 50, 50);

  // Új Rectangle változó hozzárendelése r1-hez.
  Console.WriteLine("-> Assigning r2 to r1");
  Rectangle r2 = r1;
```

```
    // r2 néhány értékének módosítása.
    Console.WriteLine("-> Changing values of r2");
    r2.rectInfo.infoString = "This is new info!";
    r2.rectBottom = 4444;

    // A két téglalap értékeinek kiírása.
    r1.Display();
    r2.Display();
}
```



4.12. ábra: *A belső referenciák ugyanarra az objektumra mutatnak*

Ha az r2 referencia használatával módosítjuk az információs sztring értékét, akkor az r1 referencia ugyanazt az értéket jeleníti meg. Az alapértelmezés szerint, amikor egy értéktípus más referenciatípusokat tartalmaz, akkor az értékadás a referenciák másolatát eredményezi. Ily módon két független struktúrával rendelkezünk, amelyek mindegyike tartalmaz egy olyan hivatkozást, amely ugyanarra az objektumra mutat a memóriában (azaz egy „sekély másolatra"). Amikor „mélymásolást" szeretnénk végrehajtani, ahol a belső referenciák állapota teljes mértékben átmásolódik az új objektumba, az egyik megoldás az `ICloneable` interfész megvalósítása (lásd a 9. fejezetet).

Forráskód A `ValueAndReferenceTypes` projekt megtalálható a 4. fejezet alkönyvtárában.

Referenciatípusok átadása értékként

A referencia- vagy értéktípusok természetesen paraméterként is átadhatók a típustagoknak. Egy referenciatípus (pl. egy osztály) átadása referenciaként azonban különbözik attól, hogy ha értékként adjuk át. A különbség megértéséhez tételezzük fel, hogy a `RefTypeValTypeParams` új parancssorialkalmazás-projektben létezik egy egyszerű `Person` osztály, amelynek definíciója a következő:

```
class Person
{
  public string personName;
  public int personAge;

  // Konstruktorok.
  public Person(string name, int age)
  {
    personName = name;
    personAge = age;
  }
  public Person(){}

  public void Display()
  {
    Console.WriteLine("Name: {0}, Age: {1}", personName, personAge);
  }
}
```

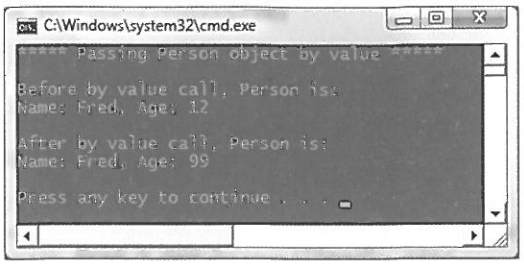
Hozzunk létre egy olyan metódust, amely lehetővé teszi a hívónak, hogy értékként küldje be a `Person` típust (figyeljük meg a paramétermódosítók hiányát):

```
static void SendAPersonByValue(Person p)
{
  // Módosítsuk "p" életkorát?
  p.personAge = 99;

  // Látja a hívó ezt a hozzárendelést?
  p = new Person("Nikki", 99);
}
```

Látható, hogy a `SendAPersonByValue()` metódus hogyan próbálja újra hozzárendelni a bejövő `Person` referenciát az új objektumhoz, valamint módosítani néhány állapotadatot. Teszteljük ezt a metódust az alábbi `Main()` metódus felhasználásával:

```
static void Main(string[] args)
{
  // Referenciatípus átadása értékként.
  Console.WriteLine("***** Passing Person object by value *****");
  Person fred = new Person("Fred", 12);
  Console.WriteLine("\nBefore by value call, Person is:");
  fred.Display();

  SendAPersonByValue(fred);
  Console.WriteLine("\nAfter by value call, Person is:");
  fred. Display();
  Console.ReadLine();
}
```

A 4.13. ábra mutatja a hívás kimenetét.



4.13. ábra: *A referenciatípusok átadása értékként zárolja a referenciát*

Látható, hogy a `personAge` értéke megváltozott. Úgy tűnik, hogy ez a viselkedés ellentmond annak, amit a paraméter értékkénti átadása jelent. Mivel meg tudjuk változtatni a bejövő `Person` állapotát, akkor mi történt az átmásolással? A válasz: a referencia másolata került a hívó objektumába. Ezért, mivel a `SendAPersonByValue()` metódus ugyanarra az objektumra mutat, mint a hívó, módosítani lehet az objektum állapotadatait. Azt viszont nem lehet újra hozzárendelni, amire a referencia mutat.

Referenciatípusok átadása referenciaként

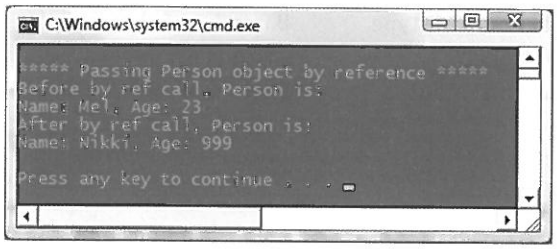
Tételezzük fel, hogy van egy `SendAPersonByReference()` metódusunk, amely a referenciatípust referenciaként adja át (figyeljük meg a `ref` paramétermódosítót):

```
static void SendAPersonByReference(ref Person p)
{
  // "p" néhány adatának módosítása.
  p.personAge = 555;

  // "p" a heap területen új objektumra mutat!
  p = new Person("Nikki", 999);
}
```

Ez teljesen flexibilis arra nézve, hogy a hívott fél hogyan képes kezelni a bejövő paramétert. Nemcsak az objektum állapotát tudja megváltoztatni, hanem újra hozzárendelheti a referenciát egy új `Person` típushoz is. Tekintsük át a következő frissített `Main()` metódust, majd nézzük meg a kimenetet a 4.14. ábrán:

```
static void Main(string[] args)
{
  // Referenciatípus átadása referenciaként.
  Console.WriteLine("\n***** Passing Person object by
    reference *****");
```

```
    Person mel = new Person("Mel", 23);
    Console.WriteLine("Before by ref call, Person is:");
    mel.Display();

    SendAPersonByReference(ref mel);
    Console.WriteLine("After by ref call, Person is:");
    mel.Display();
    Console.ReadLine();
}
```



4.14. ábra: *Referenciatípusok átadása referenciaként lehetővé teszi a referencia átirányítását*

Ahogy láthatjuk, a `Mel` nevű objektum `Nikki` nevű típusként tér vissza a hívás után, mivel a metódus képes volt megváltoztatni azt, hogy a bejövő referencia mire mutasson a memóriában. A referenciatípusok átadásának aranyszabálya:

- Ha a referenciatípust referenciaként adjuk át, akkor a hívott fél módosíthatja az objektum állapotadatainak értékét, valamint azt, hogy melyik objektumra hivatkozik.
- Ha a referenciatípust értékként adjuk át, akkor a hívott fél módosíthatja az objektum állapotadatainak értékét, azt azonban nem, hogy melyik objektumra hivatkozik.

Forráskód A `RefTypeValTypeParams` projekt megtalálható a 4. fejezet alkönyvtárában.

Érték- és referenciatípusok: további részletek

Végül tekintsük át a 4.3. táblázatot, amely összefoglalja az alapvető különbségeket az érték- és referenciatípusok között.

Kérdés	Értéktípus	Referenciatípus
Hol történik a típus lefoglalása?	A veremben.	A felügyelt heapben.
Hogyan van jelképezve egy változó?	Az értéktípus-változók lokális másolatok.	A referenciatípusú-változók a lefoglalt példány által elfoglalt memóriára mutatnak.
Mi az alaptípus?	A `System.ValueType` osztályból kell származnia.	Bármely másik típusból származhat (kivéve a `System.ValueType` típust) egészen addig, amíg az a típus nincs „lezárva" (lásd a 6. fejezetben).
Szolgálhat-e ez a típus alapként más típusok számára?	Nem. Az értéktípusok mindig zártak, és nem lehet őket bővíteni.	Igen. Ha a típus nincs lezárva, akkor szolgálhat alapként más típusok számára.
Mi az alapértelmezett paraméterátadási viselkedés?	A változók értékként adódnak át (azaz a változó egy másolata átadódik a meghívott függvénynek).	A változók referenciaként adódnak át (azaz a változó címe átadódik a meghívott függvénynek).
Felüldefiniálhatja ez a függvény a `System.Object.Finalize()` metódust?	Nem. Az értéktípusok nem kerülhetnek a heapbe, ezért nem kell őket véglegesíteni.	Igen, közvetetten (lásd a 8. fejezetet).
Lehet-e definiálni konstruktorokat a típus számára?	Igen, de az alapértelmezett konstruktor le van foglalva (vagyis az egyedi konstruktoroknak mind rendelkezniük kell paraméterekkel).	Természetesen.
Mikor halnak meg a típus változói?	Amikor kiesnek a meghatározó hatókörből.	Amikor az objektum szemétgyűjtött lesz.

4.3. táblázat: *Érték- és referenciatípusok egymás mellett*

A különbségek ellenére az érték- és a referenciatípusok is rendelkeznek azzal a képességgel, hogy megvalósítsanak interfészeket, valamint támogassanak bármennyi mezőt, metódust, túlterhelt operátort, konstanst, tulajdonságot és eseményt.

Nullázható típusok a C#-ban

Végül vizsgáljuk meg a *nullázható adattípusok* szerepét egy új, `NullableTypes` nevű parancssori alkalmazás használatával. A CLR-adattípusok rögzített tartománnyal rendelkeznek, és típusként jelennek meg a `System` névtérben. A `System.Boolean` adattípushoz például a {`true`, `false`} készletből lehet értéket hozzárendelni. Nyilvánvalóan az összes numerikus adattípus (ugyanígy a `Boolean` adattípusok is) *értéktípus*. Az értéktípusokhoz sosem lehet null értéket hozzárendelni, az ugyanis üres objektumreferencia készítésére használatos:

```
static void Main(string[] args)
{
  // Fordítási hibák!
  // Az értéktípusokhoz nem lehet null értéket rendelni!
  bool myBool = null;
  int myInt = null;

  // OK! A sztringek referenciatípusok.
  string myString = null;
}
```

A .NET 2.0 kiadása óta lehet nullázható adattípusokat létrehozni. A nullázható típus ábrázolhatja az alapjául szolgáló típus összes értékét, valamint a null értéket. Ezért, ha nullázható `System.Booleant` deklarálunk, akkor a {`true`, `false`, `null`} készletből rendelhetünk hozzá értéket. Ez nagyon jól jöhet akkor, amikor relációs adatbázisokkal dolgozunk, ugyanis definiálatlan oszlopokkal találkozni az adatbázis-táblázatokban teljesen szabályos dolog. A nullázható adattípusok koncepciója nélkül nem lenne megfelelő módszer a C#-ban arra, hogy érték nélküli numerikus adatelemet ábrázoljunk.

Nullázható változótípus definiálása esetén kérdőjel (?) társul utótagként az alapadattípushoz. Ez a szintaxis csak akkor érvényes, ha értéktípusokra alkalmazzuk. Ha nullázható referenciatípust (beleértve a sztringeket) próbálunk meg létrehozni, akkor fordítási idejű hibát kapunk. A nem nullázható változóhoz hasonlóan, a lokális nullázható változókhoz is hozzá kell rendelnünk egy kezdőértéket:

```
static void LocalNullableVariables()
{
  // Néhány lokális nullázható típus definiálása.
  int? nullableInt = 10;
  double? nullableDouble = 3.14;
  bool? nullableBool = null;
  char? nullableChar = 'a';
  int?[] arrayOfNullableInts = new int?[10];
```

```
    // Hiba! A sztringek referenciatípusok!
    // sztring? s = "oops";
}
```

A C#-ban a ? utótagjelölés a generikus System.Nullable<T> struktúratípus egy példányának létrehozását segítő rövidített forma. A System.Nullable<T> típus biztosítja az olyan tagok gyűjteményét, amelyeket az összes nullázható típus használhat (a generikus típusokat lásd a 10. fejezetben).

Programozottan például a HasValue tulajdonság vagy a != operátor használatával kideríthetjük, hogy a nullázható változóhoz valóban null érték van-e hozzárendelve. A nullázható típus hozzárendelt értékét közvetlenül vagy a Value tulajdonság révén is megszerezhetjük. Mivel a ? utótag csak egyszerűsítés a Nullable<T> használatához, a LocalNullableVariables() metódust a következőképpen implementálhatjuk:

```
static void LocalNullableVariables()
{
  // Lokális nullázható típus definiálása a Nullable<T> típussal.
  Nullable<int> nullableInt = 10;
  Nullable<double> nullableDouble = 3.14;
  Nullable<bool> nullableBool = null;
  Nullable<char> nullableChar = 'a';
  Nullable<int>[] arrayOfNullableInts = new int?[10];
}
```

Munka a nullázható típusokkal

A nullázható adattípusok akkor lehetnek nagyon hasznosak, ha adatbázisokkal dolgozunk, ugyanis a táblák oszlopai szándékosan üresek (pl. meghatározatlanok) is lehetnek. Tételezzük fel például, hogy a következő osztály szimulálja a hozzáférést egy olyan adatbázishoz, amelynek táblája két olyan oszlopot is tartalmazhat, amelyek null értékkel rendelkezhetnek. Figyeljük meg, hogy a GetIntFromDatabase() metódus nem rendel értéket a nullázható egész típusú tagváltozóhoz, amíg a GetBoolFromDatabase() metódus érvényes értéket rendel a bool? taghoz:

```
class DatabaseReader
{
  // Nullázható adatmező.
  public int? numericValue = null;
  public bool? boolValue = true;
```

```
    // Vegyük észre a nullázható visszatérési típust.
    public int? GetIntFromDatabase()
    { return numericValue; }

    // Vegyük észre a nullázható visszatérési típust.
    public bool? GetBoolFromDatabase()
    { return boolValue; }
}
```

Nézzük meg a következő `Main()` metódust, amely a `DatabaseReader` osztály minden tagját meghívja, majd a `HasValue` és a `Value` tagok, valamint a C# egyenlőségvizsgáló operátor (nem egyenlő, hogy pontosak legyünk) használatával megtalálja a hozzárendelt értékeket:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Nullable Data *****\n");
  DatabaseReader dr = new DatabaseReader();

  // int beolvasása az "adatbázisból".
  int? i = dr.GetIntFromDatabase();
  if (i.HasValue)
    Console.WriteLine("Value of 'i' is: {0}", i.Value);
  else
    Console.WriteLine("Value of 'i' is undefined.");

  // bool beolvasása az "adatbázisból".
  bool? b = dr.GetBoolFromDatabase();
  if (b != null)
    Console.WriteLine("Value of 'b' is: {0}", b.Value);
  else
    Console.WriteLine("Value of 'b' is undefined.");
  Console.ReadLine();
}
```

A ?? operátor

Amit még érdemes tudni a nullázható típusokkal kapcsolatban az az, hogy használhatják a C# `??` operátorát. Ez az operátor lehetővé teszi érték hozzárendelését egy nullázható típushoz, ha a visszakapott érték tényleg nulla. Ehhez a példához tételezzük fel, hogy szeretnénk a 100-at hozzárendelni egy lokális, null értéket is felvehető egész számhoz, ha a `GetIntFromDatabase()` metódustól visszakapott érték null (természetesen ez a metódus úgy van programozva, hogy *mindig* null értéket adjon vissza, de az általános koncepció érthető):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Nullable Data *****\n");
  DatabaseReader dr = new DatabaseReader();
  ...
  // Ha a GetIntFromDatabase() metódustól kapott érték null,
  // a lokális változó értéke legyen 100.
  int? myData = dr.GetIntFromDatabase() ?? 100;
  Console.WriteLine("Value of myData: {0}", myData.Value);
  Console.ReadLine();
}
```

Forráskód A NullableTypes alkalmazás megtalálható a 4. fejezet alkönyvtárában.

Összefoglalás

Ez a fejezet olyan C#-kulcsszavak vizsgálatával kezdett el foglalkozni, amelyek lehetővé teszik egyedi metódusok készítését. Az alapértelmezés szerint a paramétereket értékként adjuk át; ám referenciaként is átadhatjuk őket, ha meg vannak jelölve a `ref` vagy az `out` módosítóval. Láttuk az opcionális paraméterek szerepét, és azt is, hogyan definiáljunk és hívjunk tömbparamétert kapó metódusokat.

Miután megvizsgáltuk a metódus-túlterhelés témáját, a fejezet további része azzal foglalkozott, hogy a tömbök, a felsorolások és a struktúrák hogyan definiálhatók a C#-ban, és hogyan képviselik őket a .NET-alaposztálykönyvtárakban. Emellett képet kaptunk az érték- és referenciatípusokról is, arról, hogyan reagálnak, amikor paraméterként adjuk át őket a metódusoknak, továbbá hogy hogyan kezelhetjük a nullázható adattípusokat a ? és ?? operátorok használatával. Ezzel a C# programozási nyelv vizsgálatának a végére értünk. A következő fejezetben az objektumorientált fejlesztés részleteivel ismerkedünk meg.

ÖTÖDIK FEJEZET

Beágyazott osztálytípusok definiálása

Az előző két fejezet olyan alapvető szintaktikai konstrukciókkal foglalkozott, amelyek mindennaposak a .NET-alkalmazások fejlesztésében, a következőkben pedig a C# objektumorientált lehetőségeinek a bemutatására térünk rá. Az első feladat: megvizsgálni az olyan, jól definiált osztálytípusok készítésének folyamatát, amelyek bármennyi *konstruktort* támogatnak. Az osztályok definiálásának és az objektumok létrehozásának alapjait követően az *egységbe zárás* szerepét tanulmányozzuk, majd az osztálytulajdonságok definiálását, valamint a statikus mezők és tagok, továbbá az írásvédett mezők és a konstans adatok szerepét. Ezen túlmenően az XML-kóddokumentációs szintaxis szerepének vizsgálatába is belemélyedünk.

Bevezetés a C# osztálytípusaiba

A .NET-platformról esetében a legalapvetőbb programozási szerkezet az *osztálytípus*. Az osztály, hivatalosan: egy felhasználó által meghatározott típus, adatmezőkből (ezt gyakran *tagváltozónak* is nevezik) és olyan tagokból áll, amelyek műveleteket végeznek ezekkel az adatokkal (pl. konstruktorok, tulajdonságok, metódusok, események stb.). Az adatmezők készlete együttesen képviseli az osztálypéldány (másnéven *objektum*) „állapotát". Az objektumalapú nyelvek (mint amilyen a C# is) ereje abban rejlik, hogy az adatok és a kapcsolódó működés osztálydefinícióba csoportosításával képesek vagyunk az életszerű entitásokat követve lemodellezni a szoftverünket.

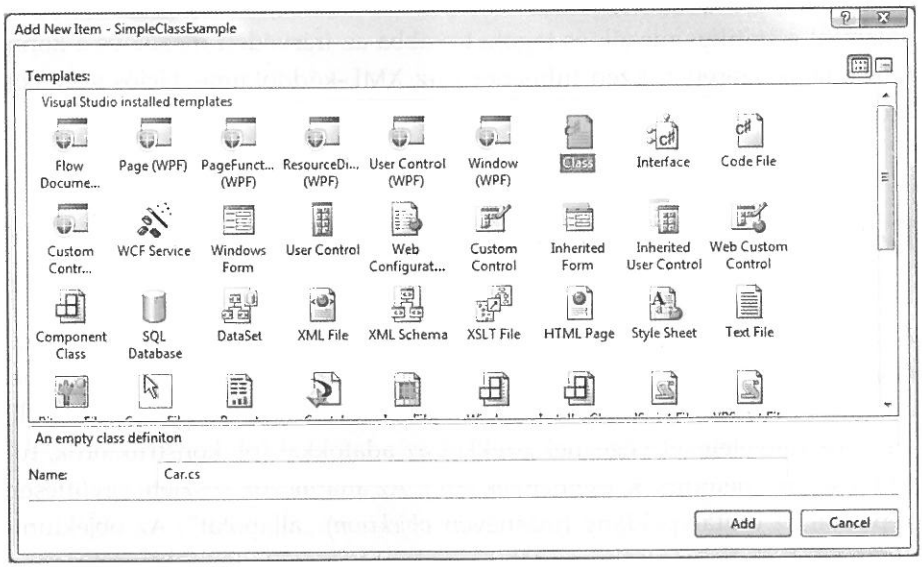
Kiindulásként hozzunk létre egy új C# parancssori alkalmazást `SimpleClassExample` névvel. Ezután szúrjunk be egy új osztályfájlt (`Car.cs` néven) a projektbe a Project > Add Class menü használatával: a megjelenő ablakban válasszuk a Class ikont (lásd 5.1. ábra), majd kattintsunk az Add gombra.

A C#-ban a `class` kulcsszó használatával definiálhatunk egy osztályt. A legegyszerűbb lehetséges deklaráció a következő:

```
class Car
{
}
```

Az osztálytípus definiálása után át kell gondolnunk a tagváltozók készletét: ezeket az osztálytípus állapotának a leképezésére fogjuk használni. Dönthetünk úgy például, hogy az autóknak legyen egy egész adattípusa az aktuális sebesség jelképezésére, továbbá egy sztringadattípusa az autó becenevének a jelölésére. Ha adottak ezek a kezdeti megjegyzések, a következők szerint módosítsuk a `Car` osztályunkat:

```
class Car
{
  // A Car osztály "állapota".
  public string petName;
  public int currSpeed;
}
```



5.1. ábra: *Új C#-osztálytípus beszúrása*

Ezeket a tagváltozókat a `public` hozzáférés-módosítóval deklaráltuk. Egy osztály nyilvános tagjai közvetlenül hozzáférhetővé válnak, amint létrehozunk egy ilyen típusú *objektumot*. Az „objektum" kifejezést pedig arra használjuk, hogy képviselje egy adott osztálytípus `new` kulcsszóval létrehozott példányát.

Megjegyzés Egy osztály adatmezőjét ritkán (vagy sohasem) érdemes nyilvánosként meghatározni. Az állapotadatok integritásának megőrzéséhez sokkal jobb úgy tervezni, hogy privátként (vagy lehetőleg védettként) határozzuk meg az adatokat, és típustulajdonságokkal engedélyezzük az adatokhoz való hozzáférést (erről a fejezet későbbi részében még lesz szó). Az első példánkhoz azonban az egyszerűség kedvéért a nyilvános adat is megfelelő.

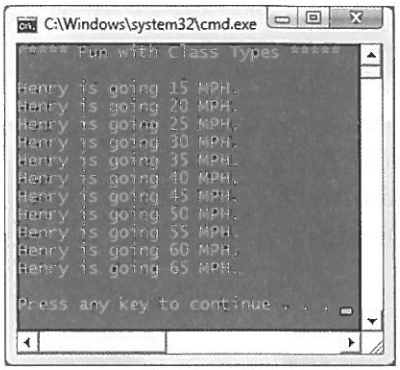
Miután meghatároztuk azoknak a tagváltozóknak a készletét, amelyek a típus állapotát jelképezik, a tervezés következő lépése azoknak a tagoknak a bevezetése, amelyek a viselkedését modellezik. Ebben a példában a `Car` osztály egy `SpeedUp()` nevű és egy `PrintState()` nevű metódust fog meghatározni:

```
class Car
{
  // A Car osztály "állapota".
  public string petName;
  public int currSpeed;

  // A Car osztály funkcionalitása.
  public void PrintState()
  {
    Console.WriteLine("{0} is going {1} MPH.", petName, currSpeed);
  }
  public void SpeedUp(int delta)
  {
    currSpeed += delta;
  }
}
```

A `PrintState()` többé-kevésbé diagnosztikai függvény, amely egyszerűen kiíratja az adott `Car` objektum aktuális állapotát a parancssorba. A `SpeedUp()` megnöveli a `Car` sebességét a bemeneti int paraméterben megadott mennyiséggel. Ezután frissítsük a `Main()` metódust a következő forráskóddal:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Class Types *****\n");
  // A Car objektum lefoglalása és beállítása.
  Car myCar = new Car();
  myCar.petName = "Henry";
  myCar.currSpeed = 10;

  // Az autó gyorsítása és az új állapot kiírása.
  for (int i = 0; i <= 10; i++)
  {
    myCar.SpeedUp(5);
    myCar.PrintState();
  }
  Console.ReadLine();
}
```

Ha lefuttatjuk a programot, látható, hogy a Car objektum (myCar) megtartja aktuális állapotát végig az alkalmazás élettartamán át (lásd az 5.2. ábrán is).



5.2. ábra: *Az autó tesztelése*

Objektumok memóriaterületének lefoglalása a new kulcsszóval

Ahogy az előző példakódban is látható volt, az objektumoknak a new kulcsszóval foglalunk memóriát. Ha nem használjuk ezt a kulcsszót, és megpróbáljuk az osztályváltozót egy későbbi utasításban alkalmazni, akkor fordítási hibát kapunk:

```
static void Main(string[] args)
{
  // Hiba! Alkalmazni kell a 'new' kulcsszót!
  Car myCar;
  myCar.petName = "Fred";
}
```

Osztálytípus-változó helyes létrehozásához egy kódsorban is meghatározhatunk és lefoglalhatunk Car objektumot:

```
static void Main(string[] args)
{
  Car myCar = new Car();
  myCar.petName = "Fred";
}
```

Másik lehetőségként, ha külön kódsorokban szeretnénk meghatározni és lefoglalni egy adott objektumot, a következőt tehetjük:

```
static void Main(string[] args)
{
  Car myCar;
  myCar = new Car();
  myCar.petName = "Fred";
}
```

Ekkor az első forráskód-utasítás egyszerűen egy *referenciát* deklarál egy még meghatározandó `Car` objektumhoz. Amíg hozzá nem rendeljük a referenciát az objektumhoz a `new` kulcsszóval, addig ez a referencia nem mutat érvényes osztálypéldányra a memóriában.

Bármelyik megoldást is választjuk, ekkorra már van egy triviális osztálytípusunk, amely definiál néhány adatot és egyszerű metódust. Ahhoz, hogy kibővítsük az aktuális `Car` típust, meg kell ismerkednünk az *osztálykonstruktorok* funkciójával is.

A konstruktor

Minthogy az objektumok rendelkeznek valamilyen adott állapottal (amelyet az objektum tagváltozóinak értékei jelképeznek), az objektumfelhasználó az alkalmazás előtt általában releváns értékeket akar hozzárendelni az objektum adatmezőjéhez. Példánkban a `Car` típus megköveteli, hogy a `petName` és a `currSpeed` mezők hozzárendelése mezőről mezőre történjen. Ebben az esetben ez nem is túlságosan problémás, hiszen mindössze két nyilvános adatunk van. Az osztályok esetében azonban nem szokatlan, hogy mezők tucatjaival rendelkezzenek. Természetesen 20 adatnak már nem szeretnénk 20 külön utasítással értéket adni.

Szerencsére a C# támogatja az *osztálykonstruktorok* használatát, ezek pedig lehetővé teszik az objektumállapot meghatározását a létrehozáskor. A konstruktor az osztály egy olyan speciális metódusa, amelyet közvetlenül hívunk meg akkor, ha a `new` kulcsszóval hozunk létre egy objektumot. A „normális" metódussal ellentétben azonban a konstruktorok soha nem rendelkeznek visszatérési értékkel (még voiddal sem), és mindig azonos nevet kapnak, mint az általuk felépített osztály.

Megjegyzés A 13. fejezetben szó lesz róla, hogy a C# 2008 egy olyan új objektuminicializáló szintaxist biztosít, amely lehetővé teszi a nyilvános mező értékének beállítását, és lehívja a nyilvános tulajdonságokat a felépítéskor.

Az alapértelmezett konstruktor szerepe

Minden C#-osztály rendelkezik *alapértelmezett konstruktorral*, amelyet igény szerint át lehet definiálni. Az alapértelmezett konstruktor, természetéből adódóan, soha nem rendelkezik argumentumokkal. Amellett, hogy az új objektumot elhelyezi a memóriában, az alapértelmezett konstruktor biztosítja azt is, hogy minden adatmező a megfelelő alapértékre legyen állítva (a C#-adattípusok alapértékeinek leírása a 3. fejezetben található).

Ha nem vagyunk elégedettek ezekkel az alapértelmezett hozzárendelésekkel, akkor igényeinknek megfelelően újradefiniálhatjuk az alapértelmezett konstruktort. Ehhez például a következők szerint módosítsuk a C# `Car` osztályunkat:

```
class Car
{
  // A Car osztály "állapota".
  public string petName;
  public int currSpeed;

  // Egyedi alapértelmezett konstruktor.
  public Car()
  {
    petName = "Chuck";
    currSpeed = 10;
  }
  ...
}
```

Ekkor arra kényszerítünk minden `Car` objektumot, hogy működését `Chuck` néven, óránként 10 mérföldes sebességgel kezdje meg. Ezzel létrehozhatunk olyan `Car` objektumot, amelynek alapértékei a következők:

```
static void Main(string[] args)
{
  // Az alapértelmezett konstruktor hívása.
  Car chuck = new Car();

  // Kiírja a "Chuck is going 10 MPH." szöveget.
  chuck.PrintState();
}
```

Egyedi konstruktor definiálása

Az osztályok általában az alapértelmezetten túl további konstruktorokat is meghatároznak. Így az objektum felhasználójának egyszerűen és konzisztensen tesszük lehetővé, hogy közvetlenül a létrehozáskor beállítsa az objektum kezdeti állapotát. Végezzük el a következő módosítást a `Car` osztálynál, amely így már összesen három osztálykonstruktort támogat:

```
class Car
{
  // A Car osztály "állapota".
  public string petName;
  public int currSpeed;

  // Egyedi alapértelmezett konstruktor.
  public Car()
  {
    petName = "Chuck";
    currSpeed = 10;
  }

  // A currSpeed az alapértelmezett
  // int (zero) értéket kapja.
  public Car(string pn)
  {
    petName = pn;
  }

  // A hívó a Car objektum tele „állapotát" állíthatja be.
  public Car(string pn, int cs)
  {
    petName = pn;
    currSpeed = cs;
  }
  ...
}
```

A konstuktorargumentumok típusa és száma különbözteti meg az egyik konstruktort a másiktól (a C#-fordító szemében). Amikor meghatározunk egy olyan metódust, amely nevében egyezik, ám a másiktól az argumentumok számában vagy típusában különbözik, akkor *túlterheltük* a metódust (lásd a 4. fejezetet). A `Car` típus tehát *túlterhelte* a konstruktort, hogy a deklarációkor több módszert biztosítson az objektum létrehozásához. Mindenesetre most már a nyilvános konstruktorok bármelyikével létre tudunk hozni `Car` objektumokat, például:

```
static void Main(string[] args)
{
  // Chuck nevű Car objektum, amely sebessége 10 mérföld/h.
  Car chuck = new Car();
  chuck.PrintState();

  // Mary nevű Car objektum, amely sebessége 0 mérföld/h.
  Car mary = new Car("Mary");
  mary.PrintState();

  // Daisy nevű Car objektum, amely sebessége 75 mérföld/h.
  Car daisy = new Car("Daisy", 75);
  daisy.PrintState();
}
```

Az újradefiniált alapértelmezett konstruktor

Minden osztály rendelkezik tehát egy alapértelmezett konstruktorral. Így, ha új osztályt szúrunk be az aktuális, `Motorcycle` nevű projektünkbe, a következőképpen határozhatjuk meg:

```
class Motorcycle
{
  public void PopAWheely()
  {
    Console.WriteLine("Yeeeeeee Haaaaaeewww!");
  }
}
```

Az alapértelmezett konstruktor révén azonnal létre lehet hozni a `Motorcycle` típus egy példányát:

```
static void Main(string[] args)
{
  Motorcycle mc = new Motorcycle();
  mc.PopAWheely();
}
```

Ha azonban egy egyedi konstruktort adunk meg, akkor az alapértelmezett konstruktor *észrevétlenül eltűnik* az osztályból, és többé nem lesz elérhető. Ha tehát nem határozunk meg egyedi konstruktort, akkor a C#-fordító ad egy alapértelmezettet, hogy lehetővé tegye az objektumfelhasználó számára a típus egy példányának a létrehozását a helyes alapértelmezett értékre állított adatmezőkkel. Ám, ha megadunk egy egyedi konstruktort, akkor a fordító azt feltételezi, hogy átvettük tőle az irányítást.

Így, ha lehetővé szeretnénk tenni az objektumfelhasználó számára, hogy létrehozza a típus egy példányát az alapértelmezett konstruktorral, valamint az egyedi konstruktorunkkal, akkor *explicit módon* újra kell definiálnunk az alapértelmezettet. Ezért az esetek többségében egy osztály alapértelmezett konstruktorának a megvalósítása szándékosan üres, hiszen csupán arra a lehetőségre van szükségünk, hogy alapértelmezett értékekkel rendelkező objektumot hozhassunk létre. Nézzük meg a `Motorcycle` osztály alábbi módosítását:

```
class Motorcycle
{
  public int driverIntensity;

  public void PopAWheely()
  {
    for (int i = 0; i <= driverIntensity; i++)
    {
      Console.WriteLine("Yeeeeeee Haaaaaeewww!");
    }
  }

  // Az alapértelmezett konstruktor visszahelyezése.
  public Motorcycle() {}

  // Az egyedi konstruktorunk.
  public Motorcycle(int intensity)
  { driverIntensity = intensity; }
}
```

A this kulcsszó szerepe

A többi C-alapú nyelvhez hasonlóan a C#-ban is létezik `this` kulcsszó, amely az aktuális osztálypéldányhoz biztosít hozzáférést. A `this` kulcsszó egyik lehetséges felhasználása a félreérthető hatókör feloldása, ez a hatókör akkor jelentkezik, ha egy bejövő paraméter neve ugyanaz, mint a típus egyik adatmezőjéé. Ideális esetben, egyszerűen olyan elnevezési megállapodáshoz folyamodhatnánk, amely nem eredményezne ilyen félreérthetőséget, de nem csak ez a megoldás létezik. A `this` kulcsszó ebbéli funkciójának bemutatásához módosítsuk a `Motorcycle` osztályt egy új sztringmezővel (name névvel), hogy jelképezzük a sofőr nevét. Ezután adjunk hozzá egy `SetDriverName()` nevű metódust a következőképpen:

```
class Motorcycle
{
  public int driverIntensity;
  public string name;

  public void SetDriverName(string name)
  { name = name; }
  ...
}
```

Noha ennek a kódnak a fordítása rendben lezajlik, ha úgy módosítjuk a `Main()` metódust, hogy meghívja a `SetDriverName()` metódust, majd kiírja a `name` mező értékét, akkor meglepetésre a `name` mező értéke üres sztring lesz:

```
// Motorcycle objektum készítése, amelyet Tiny vezet?
Motorcycle c = new Motorcycle(5);
c.SetDriverName("Tiny");
c.PopAWheely();
Console.WriteLine("Rider name is {0}", c.name);
    // Üres név értéket ír ki!
```

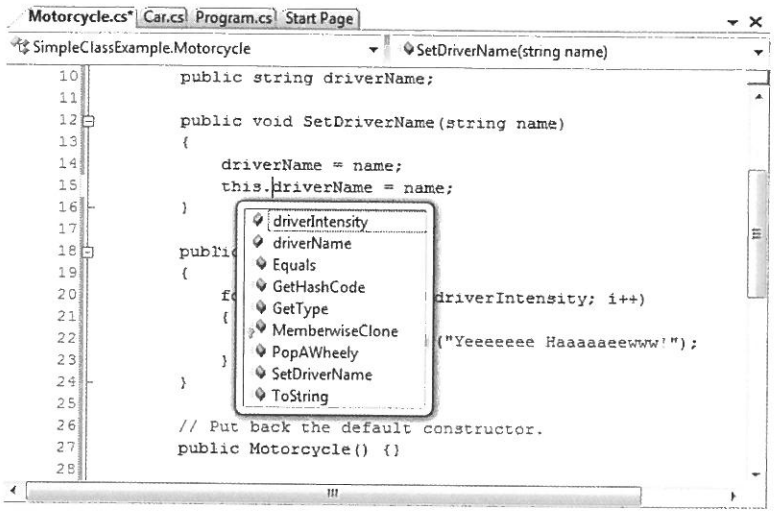
A probléma az, hogy a `SetDriverName()` metódus implementációja *saját magához rendeli vissza* a bejövő paramétert, hiszen a fordító azt feltételezi, hogy a `name` a jelenlegi metódushatókörben lévő változóra hivatkozik az osztályhatókör `name` mezője helyett. A félreérthetőség feloldásához használjuk a `this` kulcsszót, ezzel tájékoztatjuk a fordítót arról, hogy az aktuális objektum `name` adatmezőjét szeretnénk beállítani a bejövő `name` paraméterhez:

```
public void SetDriverName(string name)
{ this.name = name; }
```

Ha nincs félreérthetőség, nem kell a `this` kulcsszót használnunk akkor, amikor egy osztály hozzá akar férni saját adataihoz vagy tagjaihoz. Ha például átnevezzük a sztringadattagot `driverName`-re, akkor a `this` kulcsszó alkalmazása tetszőleges lesz, ugyanis nincs többé félreérthető hatókör:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  public void SetDriverName(string name)
  {
    // Funkcionális szempontból a két utasítás megegyezik.
    driverName = name;
    this.driverName = name;
  }
  ...
}
```

Noha a `this` kulcsszónak egyértelmű szituációkban túl sok haszna nincsen, mégis szükség lehet tagok megvalósításakor, ugyanis, ha a `this` meg van adva, az integrált fejlesztői környezetek, mint a SharpDevelop és a Visual Studio 2008, engedélyezni fogják az IntelliSense-t. A `this` nagy segítségünkre lehet akkor, ha elfelejtettük egy osztálytag nevét, és szeretnénk gyorsan előhívni a definíciót. Nézzük meg ehhez az 5.3. ábrát.



5.3. ábra: *A this IntelliSense-e*

Megjegyzés A `this` kulcsszó használata egy statikus tag megvalósításán belül fordítói hibát eredményez (ezt a későbbiekben megmagyarázzuk). A statikus tagok osztályszinten (nem objektumszinten) működnek, ezért osztályszinten nincsen aktuális objektum (azaz nincs `this`).

Konstruktorhívások láncolása a this kulcsszóval

A `this` kulcsszót akkor használjuk még, ha a *konstruktorláncolás* technikájával osztályt tervezünk meg. Ez a tervezési minta abban az esetben hasznos, ha több konstruktort meghatározó osztályról van szó. Minthogy a konstruktorok gyakran ellenőriznek bejövő paramétereket különböző üzleti szabályok kikényszerítésére, meglehetősen általános dolog redundáns ellenőrzési logikát találni egy osztály konstruktorkészletében. Nézzük meg az alábbi frissített `Motorcycle` osztályt:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  public Motorcycle() { }

  // Redundáns konstruktor logika!
  public Motorcycle(int intensity)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
  }

  public Motorcycle(int intensity, string name)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

Itt (talán azért, hogy a sofőr biztonsága szavatolva legyen) minden konstruktor azt biztosítja, hogy az intenzitás szintje soha ne legyen nagyobb 10-nél. Mindeközben két konstruktorban is vannak redundáns kódutasításaink. Ez nem túl ideális, ugyanis, ha a szabályaink megváltoznak (pl. úgy, hogy az intenzitás ne legyen nagyobb, mint 5), akkor több helyen is módosítanunk kell a forráskódunkat.

Az egyik megoldás az, ha meghatározunk egy olyan metódust a `Motorcycle` osztályban, amely érvényesíteni fogja a bejövő paraméter(eke)t. Ekkor az egyes konstruktorok meghívhatják ezt a metódust, mielőtt elvégzik a mezőhozzárendelés(eke)t. Így elszigetelhetjük azt a kódot, amelyet módosítanunk kell, ha megváltoznak az üzleti szabályok. Erre nézzük meg a következő redundanciát:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;
```

```
  // Konstruktorok.
  public Motorcycle() { }
  public Motorcycle(int intensity)
  {
    SetIntensity(intensity);
  }

  public Motorcycle(int intensity, string name)
  {
    SetIntensity(intensity);
    driverName = name;
  }

  public void SetIntensity(int intensity)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
  }
  ...
}
```

A másik megoldás az, ha kijelöljük azt a konstruktort, amely az *argumentumok legnagyobb számát* veszi „főkonstruktorként", és megvalósítása végrehajtja a kívánt érvényesítési logikát. A többi konstruktor a `this` kulcsszót használhatja arra, hogy továbbítsa a bejövő paramétereket a főkonstruktorhoz, valamint szükség esetén biztosítsa a további paramétereket. Ily módon csak egy konstruktort kell fenntartanunk az egész osztály számára, míg a többi konstruktor alapvetően üres marad.

Nézzük meg a `Motorcycle` osztály utolsó iterációját (egy további konstruktorral a példa kedvéért). A konstruktorok láncolásánál figyeljük meg, hogy a `this` kulcsszó hogyan „lógatja ki" a konstruktor deklarációját (kettőspont operátor lévén) a konstruktor hatókörén kívülre:

```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  // Konstruktorok láncolása.
  public Motorcycle() {}
  public Motorcycle(int intensity)
    : this(intensity, "") {}
  public Motorcycle(string name)
    : this(0, name) {}
```

```
  // Ez a "fő" konstruktor, ez végzi a tényleges munkát.
  public Motorcycle(int intensity, string name)
  {
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

A `this` kulcsszó használata nem kötelező a konstruktorhívási láncoknál. Ha azonban ki akarjuk használni ezt a technikát, akkor rendszerint karbantartható és tömör osztálydefiníciót kapunk eredményként. Ennek a technikának a használatával tehát egyszerűsíthetjük a programozási feladatokat, hiszen az igazi munka egyetlen konstruktorra hárul (általában a legtöbb paraméterrel rendelkezőre), míg a többi konstruktor egyszerűen „átpasszolja" neki a feladatokat.

A konstruktorok lefutásának sorrendje

Végezetül: amint a konstruktor paramétert ad át a kijelölt főkonstruktornak (és a konstruktor feldolgozta az adatot), a hívó által eredetileg aktivizált konstruktor fogja végrehajtani a többi kódutasítást. Hogy egyértelmű legyen, módosítsuk a `Motorcycle` osztály minden konstruktorát a `Console.WriteLine()` metódus meghívásával:

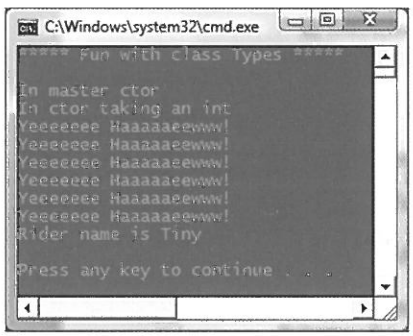
```
class Motorcycle
{
  public int driverIntensity;
  public string driverName;

  // Konstruktorok láncolása.
  public Motorcycle()
  {
    Console.WriteLine("In default ctor");
  }
  public Motorcycle(int intensity)
    : this(intensity, "")
  {
    Console.WriteLine("In ctor taking an int");
  }
```

```
  public Motorcycle(string name)
    : this(0, name)
  {
    Console.WriteLine("In ctor taking a string");
  }

  // Ez a "fő" konstruktor, ez végzi a tényleges munkát.
  public Motorcycle(int intensity, string name)
  {
    Console.WriteLine("In master ctor ");
    if (intensity > 10)
    {
      intensity = 10;
    }
    driverIntensity = intensity;
    driverName = name;
  }
  ...
}
```

Ekkor győződjünk meg róla, hogy a `Main()` metódus az alábbiak szerint használja a `Motorcycle` konstruktort:

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

  // Motorcycle konstruktor készítése.
  Motorcycle c = new Motorcycle(5);
  c.SetDriverName("Tiny");
  c.PopAWheely();
  Console.WriteLine("Rider name is {0}", c.driverName);
  Console.ReadLine();
}
```

Ha ez megvan, nézzük meg az 5.4. ábrán is látható kimenetet.



5.4. ábra: *Konstruktorláncolás munka közben*

Láthatjuk, hogy a konstruktor logikájának a menete a következő:

- Létrehozzuk az objektumot egy egész számot igénylő konstruktor aktivizálásával.
- Ez a konstruktor továbbítja az adatokat a főkonstruktornak, és olyan további indítási paramétereket ad meg, amelyeket a hívó nem határozott meg.
- A főkonstruktor hozzárendeli a bejövő adatokat az objektum adatmezőjéhez.
- A vezérlés visszatér az eredetileg meghívott konstruktorhoz, és végrehajtja a többi kódutasítást.

Minthogy már meg tudunk határozni egy osztályt adatmezővel (vagyis tagváltozókkal) és olyan különböző tagokkal, amelyeket bármennyi konstruktorral létre lehet hozni, a következőkben formalizáljuk a `static` kulcsszó szerepét.

Forráskód A `SimpleClassExample` projekt megtalálható az 5. fejezet alkönyvtárában.

A static kulcsszó

Egy C#-osztály (vagy -struktúra) a `static` kulcsszóval bármennyi *statikus tagot* meghatározhat. Ilyenkor a kérdéses tagot közvetlenül az osztályszintről kell aktivizálni, nem pedig egy típuspéldányból. A különbség bemutatásához nézzük meg a `System.Console`-t. A `WriteLine()` metódust nem objektumszinten aktivizáljuk:

```
// Hiba! A WriteLine() nem példányszintű metódus!
Console c = new Console();
c.WriteLine("I can't be printed...");
```

hanem előtaggal jelöljük a típusnevet a statikus `WriteLine()` tagnál:

```
// Helyes! A WriteLine() statikus metódus.
Console.WriteLine("Thanks...");
```

Röviden: a statikus tagok olyan elemek, amelyeket a típusszerkesztő annyira mindennaposnak ítél, hogy a tag aktivizálásakor nem kell létrehozni a típus egy példányát. Noha bármelyik osztály (vagy struktúra) meghatározhat statikus tagokat, azok általában a „segédosztályokon" belül vannak. Ha például a Visual Studio 2008 objektumböngészőjét használva (Nézet > Objektumböngésző menüelem) vizsgáljuk meg a `System.Console`, a `System.Math`, a `System.Environment` vagy a `System.GC` (hogy csak néhányat említsünk) osztályok tagjait, akkor láthatjuk, hogy azok funkcióit a statikus tagok tárják fel.

Statikus metódusok (és mezők) definiálása

Tegyük fel, hogy van egy új, `StaticMethods` nevű parancssorialkalmazás-projektünk, és ebbe beszúrtunk egy `Teenager` elnevezésű osztályt, amely a `Complain()` statikus metódust definiálja. Ez a metódus egy véletlenszerű sztringet ad vissza, részben a `GetRandomNumber()` statikus segédfüggvény meghívásával:

```
class Teenager
{
  public static Random r = new Random();

  public static int GetRandomNumber(short upperLimit)
  {
    return r.Next(upperLimit);
  }

  public static string Complain()
  {
    string[] messages = {"Do I have to?", "He started it!",
      "I'm too tired...", "I hate school!",
      "You are soooooooo wrong!"};
    return messages[GetRandomNumber(5)];
  }
}
```

A `System.Random` tagváltozó és a `GetRandomNumber()` segédfüggvény-metódus egyaránt statikus tagként vannak deklarálva a `Teenager` osztályban, ugyanis a szabályok értelmében a statikus tagok csak más statikus tagokon működhetnek.

Megjegyzés A statikus tagok csak statikus adatokon működhetnek tehát, és csak a meghatározó osztály statikus metódusait hívhatják. Ha megpróbálunk nem statikus osztályadatokat használni, illetve meghívni az osztály egyik nem statikus metódusát egy statikus tag megvalósításán belül, akkor fordítási idejű hibákat kapunk.

A statikus tagokhoz hasonlóan a `Complain()` meghívásához prefixumként kell jelölni a definiáló osztály nevét:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Methods *****\n");
  for(int i =0; i < 5; i++)
    Console.WriteLine(Teenager.Complain());
  Console.ReadLine();
}
```

Forráskód A `StaticMethods` alkalmazás megtalálható az 5. fejezet alkönyvtárában.

Statikus adatok definiálása

A típus a statikus tagok mellett statikus adatmezőt is definiálhat (mint pl. az előző `Teenager` osztályban látott Random tagváltozó). Amikor egy osztály nem statikus adatokat (pontosabban *példányadatokat*) definiál, akkor a típus objektumai fenntartják a mező független másolatát. Tegyük fel például, hogy egy betétszámlát modellező osztályt a `StaticData` új parancssorialkalmazás-projektben definiáltuk:

```
// Egyszerű betétszámla osztály.
class SavingsAccount
{
  public double currBalance;
  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
}
```

A `SavingsAccount` objektumok létrehozásakor a memória minden egyes osztálypéldánynál le van foglalva a `currBalance` mező számára. A statikus adat azonban csak egyszer foglalódik le, és oszlik meg az ugyanolyan típusú összes objektum között. A statikus adatok hasznának bemutatására `currInterestRate` néven adjunk hozzá egy olyan statikus adatot a `SavingsAccount` osztályhoz, amelynek az alapértelmezett értéke 0.04:

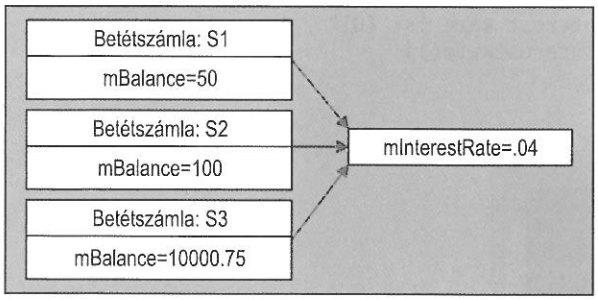
```
// Egyszerű betétszámla osztály.
class SavingsAccount
{
  public double currBalance;
```

```
  // Statikus adatelem.
  public static double currInterestRate = 0.04;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
}
```

Ha az alábbiak szerint a SavingsAccount három példányát hoznánk létre:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Data *****\n");
  Dim s1 As new SavingsAccount(50);
  Dim s2 As new SavingsAccount(100);
  Dim s3 As new SavingsAccount(10000.75);
  Console.ReadLine();
}
```

akkor a memóriában lévő adatfoglalás az 5.5. ábrához hasonló lenne.



5.5. ábra: *A statikus adatok lefoglalása csak egyszer történik meg, és az adat megoszli az osztály összes példánya között*

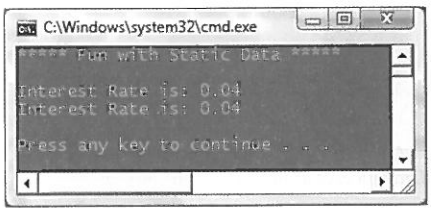
Módosítsuk a SavingsAccount osztályt, hogy két statikus metódust definiáljon, amellyel lekérdezhetjük és beállíthatjuk a kamatláb értékét:

```
// Egyszerű betétszámla osztály.
class SavingsAccount
{
  public double currBalance;

  // Statikus adatelem.
  public static double currInterestRate = 0.04;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
```

```
    // Statikus tagok a kamatláb beolvasásához és beállításához.
    public static void SetInterestRate(double newRate )
    { currInterestRate = newRate; }
    public static double GetInterestRate()
    { return currInterestRate; }
}
```

Figyeljük meg a következő használatot és a kimenetet az 5.6. ábrán.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Data *****\n");
  SavingsAccount s1 = new SavingsAccount(50);
  SavingsAccount s2 = new SavingsAccount(100);

  // Az aktuális kamatláb kiírása.
  Console.WriteLine("Interest Rate is: {0}",
    SavingsAccount.GetInterestRate());

  // Új objektum készítése; ezzel a kamatláb értékét NEM állítjuk
  // alapállapotba.
  SavingsAccount s3 = new SavingsAccount(10000.75);
  Console.WriteLine("Interest Rate is: {0}",
    SavingsAccount.GetInterestRate());
  Console.ReadLine();
}
```



5.6. ábra: *A statikus adatok lefoglalása csak egyszer történik meg*

Ahogy láthatjuk, a `SavingsAccount` osztály új példányainak létrehozásakor a statikus adatok értéke nem áll alaphelyzetbe, mivel a CLR pontosan egyszer foglalja le a memóriaterületet az adatok számára. Ezután a `SavingsAccount` típus összes objektuma már ugyanazzal az értékkel működik.

Ahogy láttuk, a statikus metódusok csak statikus adatokon működhetnek. Egy nem statikus metódus azonban használhat mind statikus, mind nem statikus adatokat, hiszen a statikus adat elérhető a típus minden példánya számára. Ennek bemutatásához módosítsuk a `SavingsAccount` osztályt a következő példányszintű tagokkal:

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate = 0.04;

  // Példányszintű tagok a kamatláb beolvasásához és beállításához.
  public void SetInterestRateObj(double newRate)
  { currInterestRate = newRate; }
  public double GetInterestRateObj()
  { return currInterestRate; }
  ...
}
```

Ekkor a `SetInterestRateObj()` és a `GetInterestRateObj()` egyaránt ugyanazon a statikus mezőn működnek, mint a statikus `SetInterestRate()`/`GetInterestRate()` metódusok. Így, ha egy objektum módosítja a kamatlábat, akkor minden más objektum ugyanazt az értéket jelenti:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Static Data *****\n");
  SavingsAccount.SetInterestRate(0.09);
  SavingsAccount s1 = new SavingsAccount(50);
  SavingsAccount s2 = new SavingsAccount(100);

  // Mindhárom sor az "Interest Rate is: 0.09" szöveget írja ki.
  Console.WriteLine("Interest Rate is: {0}",
    s1.GetInterestRateObj());
  Console.WriteLine("Interest Rate is: {0}",
    s2.GetInterestRateObj());
  Console.WriteLine("Interest Rate is: {0}",
    SavingsAccount.GetInterestRate());
  Console.ReadLine();
}
```

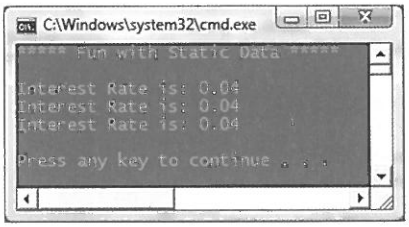
Ebben az esetben a 0.09 érték tér vissza, függetlenül attól, hogy melyik `SavingsAccount` objektumot kérdezzük (beleértve a statikus `GetInterestRate()` metódus általi lekérdezést is).

Statikus konstruktorok definiálása

A konstruktorok arra használatosak tehát, hogy a létrehozáskor beállítsák egy típus adatainak az értékét. Ezért, ha egy értéket hozzárendelünk a statikus adatokhoz egy példányszintű konstruktoron belül, azt tapasztalhatjuk, hogy az érték minden esetben alaphelyzetbe áll egy új objektum létrehozásakor. Tételezzük fel például, hogy a következők szerint módosítottuk a `SavingsAccount` osztályt:

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate;

  public SavingsAccount(double balance)
  {
    currInterestRate = 0.04;
    currBalance = balance;
  }
  ...
}
```

Ha végrehajtjuk az előző `Main()` metódust, megfigyelhetjük, hogy a `currInterestRate` változó hogyan áll alaphelyzetbe minden egyes olyan alkalommal, amikor létrehozunk egy új `SavingsAccount` objektumot (lásd 5.7. ábra).



5.7. ábra: *Statikus adat hozzárendelése egy példányszintű konstruktorban „alaphelyzetbe állítja" az értéket*

Az egyértelmű, hogy bármikor nyugodtan beállíthatjuk a statikus adatok kezdőértékét a taginicializálási szintaxis használatával, ám mi van akkor, ha a statikus adat értékét egy adatbázisból vagy egy külső fájlból kell beolvasni? Az ilyen feladatok metódushatókört igényelnek (pl. egy konstruktort), hogy a kódutasítások végrehajthatók legyenek. Ezért a C# lehetővé teszi *statikus konstruktor* definiálását:

```
class SavingsAccount
{
  public double currBalance;
  public static double currInterestRate;

  public SavingsAccount(double balance)
  {
    currBalance = balance;
  }
```

```
    // Statikus konstruktor.
    static SavingsAccount()
    {
      Console.WriteLine("In static ctor!");
      currInterestRate = 0.04;
    }
    ...
}
```

Röviden meghatározva: a statikus konstruktor egy olyan speciális konstruktor, amely ideális helyet biztosít a statikus adatok kezdőértékeinek beállításához akkor, ha az érték nem ismeretes a fordítás idején (pl. egy külső fájlból kell beolvasni, vagy véletlenszerű számot kell létrehozni, stb.). Nézzünk néhány érdekességet a statikus konstruktorokkal kapcsolatban:

- Egy adott osztály (vagy struktúra) csak egy statikus konstruktort határozhat meg.
- Egy statikus konstruktor nem vesz fel hozzáférési módosítót, és nem vehet fel paramétereket.
- Egy statikus konstruktor végrehajtása pontosan egyszer történik meg, függetlenül attól, hogy a típushoz hány objektumot hozunk létre.
- A futtatórendszer meghívja a statikus konstruktort, amikor az létrehozza az osztály egy példányát, vagy mielőtt hozzáfér a hívó által aktivizált első statikus taghoz.
- A statikus konstruktor végrehajtása megelőzi a példányszintű konstruktorokét.

Így, ha létrehozunk új `SavingsAccount` objektumokat, akkor a statikus adatok értéke megőrződik, mivel a statikus tag beállítására csak egyszer kerül sor a statikus konstruktoron belül, függetlenül a létrehozott objektumok számától.

Statikus osztályok definiálása

A .NET 2.0 megjelenése óta a C# nyelv a *statikus osztályok* bemutatásával kibővítette a `static` kulcsszó hatókörét. Ha egy osztályt statikusként definiálunk, akkor nem hozható létre a `new` kulcsszóval, és csak a `static` kulcsszóval jelölt tagokat vagy mezőket tartalmazhatja (ha nem ez a helyzet, akkor fordítási hibákat kapunk).

Első pillantásra ez nem tűnik túlságosan hasznosnak, ugyanis az, hogy egy osztályt nem lehet létrehozni, nem jelent túl nagy segítséget. Ha azonban olyan osztályt alkotunk, amely nem tartalmaz mást, csak statikus tagokat és/vagy konstans adatokat, akkor az osztály számára tulajdonképpen nem kell lefoglalni a memóriaterületet. Nézzük meg az alábbi új statikus osztálytípust:

```
// A statikus osztályok kizárólag
// statikus tagokat foglalhatnak magukban!
static class TimeUtilClass
{
  public static void PrintTime()
  { Console.WriteLine(DateTime.Now.ToShortTimeString()); }
  public static void PrintDate()
  { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}
```

Minthogy ezt az osztályt a `static` kulcsszóval definiáltuk, nem hozhatjuk létre a `TimeUtilClass` új példányát a `new` kulcsszóval:

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

  // Fordítási hiba! Nem lehet statikus osztályokat létrehozni.
  TimeUtilClass u = new TimeUtilClass ();
  ...
}
```

A .NET 2.0 alkalmazása előtt két lehetőség volt egy osztálytípus létrehozásának a megakadályozására: vagy privátként újradefiniáltuk az alapértelmezett konstruktort, vagy absztrakt típusként jelöltük meg az osztályt a C# `abstract` kulcsszavával (az absztrakt típusokat lásd részletesen a 6. fejezetben):

```
class TimeUtilClass
{
  // Az alapértelmezett konstruktort privátként kell definiálni
  // a statikus osztályok létrehozásának elkerülésére.
  private TimeUtilClass (){}

  public static void PrintTime()
  { Console.WriteLine(DateTime.Now.ToShortTimeString()); }
  public static void PrintDate()
  { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}
```

```
// A típust absztrakt típusként kell definiálni
// a statikus osztályok létrehozásának elkerülésére.
abstract class TimeUtilClass
{
  public static void PrintTime()
  { Console.WriteLine(DateTime.Now.ToShortTimeString()); }
  public static void PrintDate()
  { Console.WriteLine(DateTime.Today.ToShortDateString()); }
}
```

Bár ezek a konstrukciók továbbra is használhatók, a statikus osztályok alkalmazása tisztább és típusbiztonságosabb megoldás, ugyanis az előző két technika lehetővé tette azt, hogy a nem statikus tagok hiba nélkül megjelenjenek az osztálydefiníción belül.

Meg kell említenünk, hogy egy projekt alkalmazásobjektumát (pl. a `Main()` metódust meghatározó osztály) gyakran statikus osztályként definiáljuk annak biztosítása érdekében, hogy csak statikus tagokat tartalmazzon, és közvetlenül ne lehessen létrehozni. Például:

```
// Az alkalmazásobjektumot statikus objektumként kell definiálni.
static class Program
{
  static void Main(string[] args)
  {
    ...
  }
}
```

Miután megtanultuk, hogy kell konstruktorokat, mezőket, valamint különböző statikus (és nem statikus) tagokat tartalmazó egyszerű osztálytípusokat definiálni, vizsgáljuk meg az objektumorientált programozás három alappillérét.

Forráskód A `StaticData` projekt megtalálható a 5. fejezet alkönyvtárában.

Az objektumorientált programozás alappillérei

Minden objektumalapú nyelv meg kell, hogy feleljen az objektumorientált programozás három alapelvének. Ezeket gyakran az „objektumorientált programozás (OOP) alappilléreinek" is nevezik:

- *Egységbe zárás:* Hogyan rejti el ez a nyelv egy objektum belső megvalósítási részleteit és védi meg az adatintegritást?
- *Származtatás:* Hogyan segíti ez a nyelv a kód újrafelhasználását?
- *Polimorfizmus:* Hogyan teszi lehetővé ez a nyelv a kapcsolódó objektumok hasonló kezelését?

Az egyes alappillérek szintaktikai részleteinek vizsgálata előtt fontos megérteni a szerepüket is.

Az egységbe zárás

Az objektumorientált programozás első alappillére az *egységbe zárás*. Ez a jellemző a nyelv azon képessége, hogy képes elrejteni a felesleges megvalósítási részleteket az objektum használója elől. Tételezzük fel például, hogy a két metódussal rendelkező, `DatabaseReader` nevű osztályt használjuk: `Open()` és `Close()`:

```
// Ez a típus egységbe zárja az adatbázis megnyitásával és
// lezárásával kapcsolatos részleteket.
DatabaseReader dbReader = new DatabaseReader();
dbReader.Open(@"C:\MyCars.mdf");

// Műveletek az adatfájllal, majd a fájl zárása.
dbReader.Close();
```

Az elképzelt `DatabaseReader` osztály egységbe zárja az adatfájl helyének meghatározását, a fájl betöltését, szerkesztését és bezárását. Az objektumfelhasználók szeretik az egységbe zárást, hiszen az objektumorientált programozásnak ez az alappillére egyszerűsíti a programozási feladatokat. Nem kell aggódni a sok kódsor miatt, amelyek a háttérben dolgozva hajtják végre a `DatabaseReader` osztály munkáját. Minden, amit tennünk kell, az az, hogy létrehozunk egy példányt, és elküldjük a megfelelő üzeneteket (pl. „Nyisd meg a C meghajtón található `MyCars.mdf` nevű fájlt").

Szorosan kapcsolódik a programozási logika egységbe zárásának elképzeléséhez az adatok elrejtésének az elve. Ideális esetben egy objektum állapotadatát a `private` (vagy a `protected`) módosítóval kell ellátni. Így a mögöttes értékek módosítását vagy megszerzését kérni kell. Ez azért jó, mert a nyilvánosan deklarált adatelemek könnyen hibássá válhatnak (remélhetőleg véletlenül és nem szándékosan). A későbbiekben formálisan is megvizsgáljuk az egységbe zárásnak ezt az aspektusát.

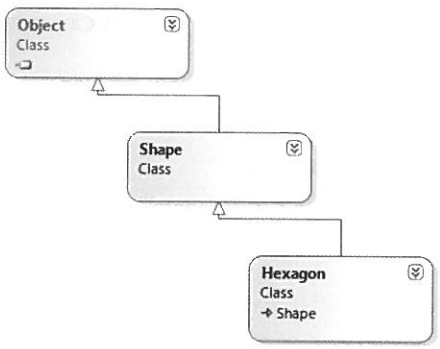
A származtatás

Az objektumorientált programozás következő alappillére a *származtatás*. Ez a nyelv azon képessége, amely biztosítja, hogy új osztálydefiníciókat lehessen építeni már létező osztálydefiníciók alapján. A származtatás lényegében lehetővé teszi számunkra, hogy kibővítsük egy alap- (vagy *szülő*-) osztály viselkedését azzal, hogy az alapfunkcionalitás a származott alosztályba is (más néven: *gyermekosztály*) öröklődik. Az 5.8. ábra egy egyszerű példát mutat.

Az ábrán lévő diagramot így olvashatjuk le: „A Hatszög »az egy« Forma és »az egy« Objektum." Amikor az osztályok ilyen formában kapcsolódnak egymáshoz, akkor *„az egy" kapcsolatot* hoztunk létre a típusok között. Az „az egy" kapcsolat neve: *klasszikus származtatás.*

Ekkor feltételezhetjük, hogy a `Forma` meghatároz néhány olyan tagot, amelyek az összes leszármazottnál közösek. Mivel a `Hatszög` osztály kibővíti a `Formát`, ezért örökli a `Forma` és `Objektum` osztályok alapfunkcionalitását, emellett meghatároz további, hatszögekre jellemző részleteket (bármelyek is legyenek azok).

Megjegyzés A .NET platform minden osztályhierarchiájában mindig a `System.Object` a legfelső ős, és meghatároz néhány alapvető fontosságú funkcionalitást (bővebben lásd a 6. fejezetben).



5.8. ábra: *Az „az egy" kapcsolat*

A kód újrafelhasználásának egy másik formája az OOP világában: a tartalmazás/delegálás modell (más néven: *„van egy" kapcsolat* vagy *aggregáció*). Az újrafelhasználás ezen formája *nem* használatos szülő/gyermek kapcsolatok létrehozására. Ehelyett a „van egy" kapcsolat azt teszi lehetővé, hogy egy osztály meghatározzon egy tagváltozót a másik osztályból, és közvetve megmutassa a funkcionalitást (ha szükséges) az objektum használójának.

Tételezzük fel ismét, hogy egy autót modellezünk. Ki akarjuk fejezni azt, hogy az autónak „van-egy" rádiója. Nem lenne logikus, ha az Car osztályt a Radio osztályból – vagy fordítva – származtatnánk. (Az autó „az egy" rádió? Nyilván nem.) Inkább két független, egymással dolgozó osztályra van szükség, ahol az Car osztály létrehozza és hozzáférhetővé teszi a Radio osztály funkcionalitását:

```
class Radio
{
  public void Power(bool turnOn)
  {
    Console.WriteLine("Radio on: {0}", turnOn);
  }
}

class Car
{
  // Az autónak "van-egy" rádiója.
  private Radio myRadio = new Radio();

  public void TurnOnRadio(bool onOff)
  {
    // Hívás továbbítása a belső objektumnak.
    myRadio.Power(onOff);
  }
}
```

Ekkor az objektum használójának elvben fogalma sincs róla, hogy a Car osztály használ egy belső Radio objektumot.

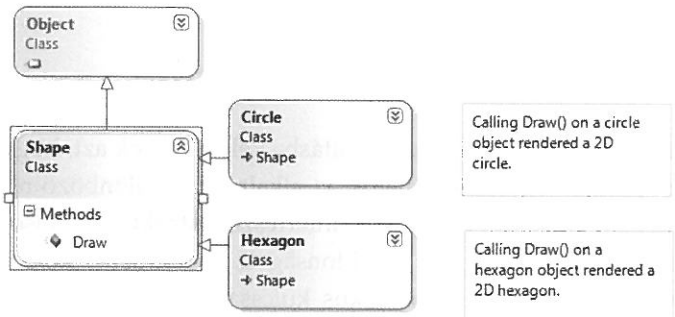
```
static void Main(string[] args)
{
  // A hívás belső továbbítása a Radio osztálynak.
  Car viper = new Car();
  viper.TurnOnRadio(false);
}
```

A polimorfizmus

Az objektumorientált programozás harmadik alappillére a *polimorfizmus*. Ez a nyelv azon képessége, hogy hasonló módon tudja kezelni a rokon objektumokat. Pontosabban, az objektumorientált nyelveknél ez teszi lehetővé azt, hogy egy ősosztály meghatározzon egy olyan tagkészletet (hivatalos nevén: *polimorf interfész*), amely felüldefiniálható az összes leszármazott osztályban. Egy osztály polimorf interfésze tetszőleges számú *virtuális* vagy *absztrakt* tagból állhat (részletekért lásd a 6. fejezetet).

Röviden: a *virtuális tag* az ősosztály egy olyan tagja, amely meghatározza az alapértelmezett megvalósítást, amelyet viszont módosíthat (hivatalosabban: *felüldefiniálhat*) egy leszármazott osztály. Ezzel szemben az *absztrakt metódus* az ősosztály egy olyan tagja, amely *nem* biztosít alapértelmezett megvalósítást, ám gondoskodik a szignatúráról. Amikor egy osztály egy absztrakt metódust meghatározó ősosztályból származik, akkor ezt a leszármazott típusnak át *kell* írnia. Amikor a származtatott típusok felüldefiniálják az ősosztály által meghatározott tagokat, akkor azok mindkét esetben alapvetően újradefiniálják ugyanarra a kérésre a választ.

Hogy előzetes képet kapjunk a polimorfizmus jelenségéről, tekintsünk át néhány részletet, amelyek az 5.8. ábrán látható hierarchia mögött állnak. Tételezzük fel, hogy a `Shape` osztály meghatározott egy `Draw()` nevű virtuális metódust, amely nem fogad el paramétereket. Mivel minden formának egyedi módon kell renderelnie önmagát, az alosztályok (mint a `Hexagon` és a `Circle`) szabadon felüldefinálhatják ezt a metódust azért, hogy saját magukhoz igazítsák (lásd 5.9. ábra).



5.9. ábra: *Klasszikus polimorfizmus*

Amint megterveztünk egy polimorf interfészt, jogos feltételezésekkel élhetünk a kód készítése során. Ha adva van például, hogy a `Hexagon` és a `Circle` osztályok egy közös szülőtől (`Shape`) származnak, akkor a `Shape` egy tömbje tartalmazhat bármit, ami ebből az ősosztályból származik. Továbbá, ha adva van, hogy a `Shape` osztály az összes származtatott típushoz (ebben a példában a `Draw()` metódus) meghatároz egy polimorf interfészt, akkor feltételezhetjük, hogy a tömbben levő minden egyes tag rendelkezik ezzel a funkcionalitással.

Figyeljük meg a következő `Main()` metódust, amely utasítja a `Shape`-ből származó típusok tömbjét arra, hogy rendereljék magukat a `Draw()` metódus felhasználásával:

```
class Program
{
  static void Main(string[] args)
  {
    Shape[] myShapes = new Shape[3];
    myShapes[0] = new Hexagon();
    myShapes[1] = new Circle();
    myShapes[2] = new Hexagon();

    foreach (Shape s in myShapes)
    {
      s.Draw();
    }
    Console.ReadLine();
  }
}
```

Ezzel valójában áttekintettük az OOP alappilléreit. A továbbiakban megvizsgáljuk az egységbe zárás működését a C#-ban. A következő fejezetben pedig a származtatás és a polimorfizmus részleteivel foglalkozunk.

Hozzáférés-módosítók a C#-ban

Ha az egységbe zárással dolgozunk, mindig számításba kell vennünk azt, hogy egy típus milyen szempontjai legyenek láthatók az alkalmazás különböző részei számára. Pontosabban, a típusok (osztályok, interfészek, struktúrák, felsorolások, metódusreferenciák) és a tagjaik (tulajdonságok, metódusok, konstruktorok, mezők stb.) esetében mindig specifikus kulcsszó vezérli azt, hogy mennyire „látható" az adott elem az alkalmazás egyéb részei számára. A C# számos kulcsszót meghatároz a hozzáférés-szabályozáshoz, s ezek abban különböznek, hogy hol lehet őket sikeresen alkalmazni (típusnál vagy tagnál). Az 5.1. táblázat leírja a hozzáférés-módosítók szerepét és alkalmazhatóságukat.

C#-hozzáférés-módosító	Alkalmazási terület	Jelentés
`public`	Típusok vagy típustagok	A nyilvános elemeknél nincsenek hozzáférési korlátozások. A nyilvános tag hozzáférhető objektumból és bármilyen leszármazott osztályból is. A nyilvános típus hozzáférhető egyéb külső szerelvényekből.

C#-hozzáférés-módosító	Alkalmazási terület	Jelentés
`private`	Típustagok vagy beágyazott típusok	A privát elemekhez csak az az osztály (vagy struktúra) férhet hozzá, amelyik meghatározza az elemet.
`protected`	Típustagok vagy beágyazott típusok	A védett elemek közvetlenül nem férhetők hozzá egy objektumváltozóból; a meghatározó típus és a leszármazott osztályok azonban hozzájuk férhetnek.
`internal`	Típusok vagy típustagok	A belső elemekhez csak az aktuális szerelvényen belül lehet hozzáférni. Ezért, ha meghatározunk egy belső típuskészletet egy .NET-osztálykönyvtáron belül, akkor más szerelvények azt nem tudják használni.
`protected internal`	Típustagok vagy beágyazott típusok	Amikor kombináljuk a `protected` és az `internal` kulcsszavakat egy elemen, akkor az elemhez a meghatározó osztály, a meghatározó szerelvényen belül pedig a leszármazott osztályok férhetnek hozzá.

5.1. táblázat: *C#-hozzáférésmódosítók*

Ebben a fejezetben csak a `public` és a `private` kulcsszavakkal foglalkozunk. A későbbiekben megvizsgáljuk az `internal` és a `protected internal` módosítók (osztálykönyvtárak készítésekor hasznos), valamint a `protected` módosító (osztályhierarchiák létrehozásakor hasznos) szerepét is.

Az alapértelmezett hozzáférés-módosítók

Az alapértelmezés szerint a típustagok *privát elemek*, míg a típusok *implicit módon belsők*. Ezért a következő osztálydefiníció automatikusan belsőre, míg a típus alapértelmezett konstruktora automatikusan privátra áll be:

```
// Belső osztály privát alapértelmezett konstruktorral.
class Radio
{
  Radio(){}
}
```

Ahhoz, hogy lehetővé tegyük más típusok számára is egy objektum tagjainak elérését, nyilvánosan hozzáférhetőnek kell megjelölnünk őket. Emellett, ha szeretnénk elérhetővé tenni a `Radio` osztályt külső szerelvényeknek (megint csak: osztálykönyvtárak készítésénél hasznos; lásd a 15. fejezetet), ugyancsak hozzá kell adnunk a publikus módosítót.

```
// Nyilvános osztály nyilvános alapértelmezett konstruktorral.
public class Radio
{
  public Radio(){}
}
```

Hozzáférés-módosítók és beágyazott típusok

Az 5.1. táblázatból láthattuk, a privát, a védett és a védett belső hozzáférés-módosítók *beágyazott típusra* alkalmazhatók. (A beágyazással részletesen a 6. fejezet foglalkozik.) Előzetesen: a beágyazott típus olyan típus, amely közvetlenül az osztály vagy struktúra hatókörén belül deklarálódik. Nézzünk meg például egy privát felsorolást (`Color`) egy nyilvános osztályba (`SportsCar`) beágyazva:

```
public class SportsCar
{
  // OK! A beágyazott típusok lehetnek privát típusok.
  private enum CarColor
  {
    Red, Green, Blue
  }
}
```

Ebben az esetben a privát hozzáférés-módosítót alkalmazni lehet a beágyazott típusra. A nem beágyazott típusokat (mint pl. a `SportsCar`) azonban csak a `public`, illetve az `internal` módosítókkal lehet meghatározni. Az alábbi osztálydefiníció tehát szabálytalan:

```
// Hiba! Nem beágyazott típusok nem lehet privát típusok!
private class SportsCar
{}
```

Az első pillér: az egységbe zárás a C#-ban

Az egységbe zárás koncepciója a köré az elképzelés köré épül, hogy egy objektum belső adata ne legyen közvetlenül hozzáférhető egy objektumpéldányból. Ehelyett, ha a hívó meg akarja változtatni egy objektum állapotát, akkor ezt közvetve a lekérdező (pl. „getter") és a módosító (pl. „setter") metódusok révén teheti meg. A C#-ban az egységbe zárást szintaktikai szinten lehet kikényszeríteni a `public`, a `private`, az `internal` és a `protected` kulcsszavak használatával. Ahhoz, hogy az egységbe zárás szolgáltatásának szükségességét bemutassuk, tegyük fel, hogy létrehoztuk a következő osztálydefiníciót:

```
// Osztály egyetlen nyilvános mezővel.
class Book
{
  public int numberOfPages;
}
```

A probléma a nyilvános adatmezővel az, hogy az elemek nem tudják valójában „megérteni", hogy az aktuális érték, amelyhez hozzárendelték őket, érvényes-e a rendszer aktuális üzleti szabályaihoz viszonyítva. A C# egész számok felső határa meglehetősen nagy (2147483647). Ezért a fordító megengedi a következő hozzárendelést:

```
// Hmmm. Ez aztán a kisregény!
static void Main(string[] args)
{
  Book miniNovel = new Book();
  miniNovel.numberOfPages = 30000000;
}
```

Noha nem léptük túl egy int adattípus határait, nyilvánvaló, hogy egy 30000000 oldalas „kisregény" meglehetősen ésszerűtlen. Belátható, hogy a nyilvános mezők nem nyújtanak védelmet arra, hogy átlépjük a logikai felső (vagy alsó) határokat. Ha az aktuális rendszerünk olyan üzleti szabállyal rendelkezik, hogy egy könyvnek 1 és 1000 oldal között kell lennie, akkor problémás lehet, hogy ezt hogyan érjük el programozásilag. Emiatt a nyilvános mezőknek általában nincs helyük termékszintű osztálydefiníciókban.

Az egységbe zárás lehetőséget biztosít arra, hogy megőrizzük egy objektum állapotadatának az integritását. Nyilvános mezők meghatározása helyett (amelyek gyakran táptalajai az adatvesztésnek), érdemes a *privát adatok* meghatározását alkalmazni, amelyeket közvetve manipulálhatunk a két fő technika egyikével:

- lekérdező (`get`) és módosító (`set`) metóduspár megadásával;
- típustulajdonság megadásával.

Emellett a C#-ban megtalálható a `readonly` (írásvédett) kulcsszó, amely ugyancsak bizonyos szintű adatvédelmet biztosít. Bármelyik technikát is választjuk, a lényeg az, hogy egy jól beágyazott osztálynak el kell rejtenie a működésének részleteit a külvilág elől. Ezt gyakran *feketedoboz-programozásnak* nevezik. Ennek a megközelítésnek az az előnye, hogy egy objektum szabadon módosíthatja egy adott metódus implementálását a felszín alatt. Teheti ezt anélkül, hogy megtörné az őt használó, már létező kódot, és biztosítja, hogy a metódus aláírása konstans marad.

Egységbe zárás tradicionális lekérdezők és módosítók használatával

A következőkben egy olyan nagyjából teljes osztályt építünk, amely egy általános átlagos dolgozót modellez. Kiindulásként hozzunk létre egy új parancssori alkalmazást `EmployeeApp` néven, és szúrjunk be egy új osztályfájlt (`Employee.cs`) a Project > Add class menüelemmel. Módosítsuk az `Employee` osztályt a következő mezőkkel, metódusokkal és konstruktorokkal:

```
class Employee
{
  // Mezőadatok.
  private string empName;
  private int empID;
  private float currPay;

  // Konstruktorok.
  public Employee() {}
  public Employee(string name, int id, float pay)
  {
    empName = name;
    empID = id;
    currPay = pay;
  }
```

```
  // Tagok.
  public void GiveBonus(float amount)
  {
    currPay += amount;
  }
  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", empName);
    Console.WriteLine("ID: {0}", empID);
    Console.WriteLine("Pay: {0}", currPay);
  }
}
```

Az `Employee` osztály mezőit jelenleg a `private` hozzáférési kulcsszó használatával határoztuk meg. Emiatt az `empName`, az `empID` és a `currPay` mezők közvetlenül nem férhetők hozzá egy objektumváltozóból:

```
static void Main(string[] args)
{
  // Hiba! Privát tagokhoz nem lehet közvetlenül az
  // objektumból hozzáférni!
  Employee emp = new Employee();
  emp.empName = "Marv";
}
```

Ha azt szeretnénk, hogy a külvilág is elérje a dolgozó teljes nevét jelképező sztringet, a hagyományosan meg kell adni egy lekérdezőt (`get` metódus) és egy módosítót (`set` metódus). Az `empName` mező egységbe zárásához például hozzáadhatjuk az alábbi nyilvános tagokat a már meglévő `Employee` osztálytípushoz:

```
class Employee
{
  // Mezőadatok.
  private string empName;
  ...
  // Lekérdező (get metódus).
  public string GetName()
  {
    return empName;
  }

  // Módosító (set metódus).
  public void SetName(string name)
  {
    // Érvénytelen karakterek (!,@,#,$,%) eltávolítása, a maximális
    // hossz és a kis- és nagybetűk ellenőrzése hozzárendelés előtt.
    empName = name;
  }
}
```

Ez a technika két egyedileg elnevezett metódust igényel ahhoz, hogy egyetlen adatelemen működhessen. Ennek bemutatásához a következők szerint módosítsuk a `Main()` metódusunkat:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
  Employee emp = new Employee("Marvin", 456, 30000);
  emp.GiveBonus(1000);
  emp.DisplayStats();

  // Munka az objektum nevével a get/set metódusok segítségével.
  emp.SetName("Marv");
  Console.WriteLine("Employee is named: {0}", emp.GetName());
  Console.ReadLine();
}
```

Egységbe zárás típustulajdonságok használatával

Noha egységbe zárhatunk egy adatmezőt a tradicionális `get` és `set` metódusok felhasználásával, a .NET-nyelvek előnyben részesítik a *tulajdonságok* használatával kivitelezett adatvédelmet. A köztes nyelvi kód (CIL) szempontjából a tulajdonságok mindig leképezik a „valódi" lekérdező és módosító metódusokat. Ezért osztálytervezőként még mindig képesek vagyunk végrehajtani a szükséges belső logikát, mielőtt hozzárendelnénk az értékeket (pl. nagybetűs érték megadása, az érték megtisztítása az érvénytelen karakterektől, numerikus érték határainak ellenőrzése stb.).

Nézzük meg az alábbi módosított `Employee` osztályt, amelyben a tulajdonság szintaxisának használatával (a tradicionális `get` és `set` metódusok helyett) már megtörtént az egyes mezők egységbe zárásának kikényszerítése:

```
class Employee
{
  // Mezőadatok.
  private string empName;
  private int empID;
  private float currPay;

  // Tulajdonságok.
  public string Name
  {
    get { return empName; }
    set { empName = value; }
  }
```

```
  public int ID
  {
    get { return empID; }
    set { empID = value; }
  }

  public float Pay
  {
    get { return currPay; }
    set { currPay = value; }
  }
  ...
}
```

Egy C#-tulajdonság egy `get` blokk (lekérdező) és egy `set` blokk (módosító) megadásából áll, közvetlenül a tulajdonság deklarációján belül. Ha tulajdonságokat használunk, a hívó számára úgy tűnik, mintha a nyilvános adatelemeket olvasná vagy írná, a háttérben azonban a `get` és `set` blokkok futnak le, biztosítva az egysébe zárást.

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
  Employee emp = new Employee("Marvin", 456, 30000);
  emp.GiveBonus(1000);
  emp.DisplayStats();

  // A Name tulajdonság beolvasása és beállítása.
  emp.Name = "Marv";
  Console.WriteLine("Employee is named: {0}", emp.Name);
  Console.ReadLine();
}
```

A tulajdonságok (ellentétben a lekérdezőkkel és a módosítókkal) megkönnyítik a típusok kezelését is annyiban, hogy képesek válaszolni a C# belső operátorainak. Tételezzük fel, hogy az `Employee` osztálytípusnak van egy belső privát tagváltozója, amely az alkalmazott korát jelöli. A releváns módosítás az alábbiakban látható:

```
class Employee
{
  ...
  private int empAge;
  public int Age
  {
    get { return empAge; }
    set { empAge = value; }
  }
```

```
    // Konstruktorok
    public Employee() {}
    public Employee(string name, int age, int id, float pay)
    {
      empName = name;
      empID = id;
      empAge = age;
      currPay = pay;
    }

    public void DisplayStats()
    {
      Console.WriteLine("Name: {0}", empName);
      Console.WriteLine("ID: {0}", empID);
      Console.WriteLine("Age: {0}", empAge);
      Console.WriteLine("Pay: {0}", currPay);
    }
}
```

Ezután tegyük fel, hogy létrehoztunk egy `joe` elnevezésű `Employee` objektumot. A születésnapján eggyel meg akarjuk növelni az egyén életkorát. A tradicionális lekérdező és módosító metódusok használatával ilyen kódot kellene írnunk:

```
Employee joe = new Employee();
joe.SetAge(joe.GetAge() + 1);
```

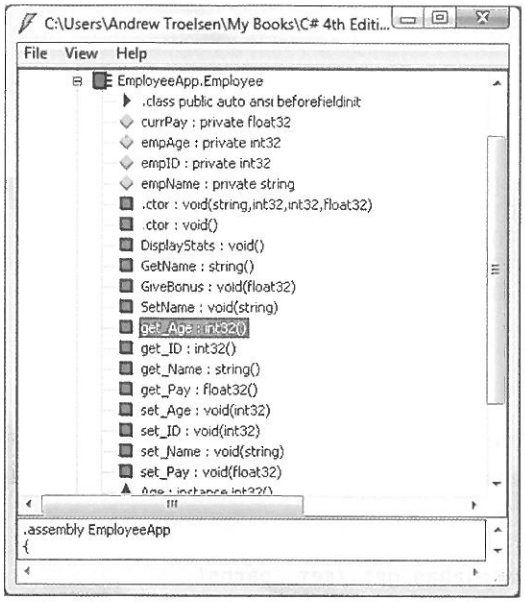
Ha azonban beágyazzuk az `empAge` tagváltozót egy `Age` nevű tulajdonság használatával, akkor egyszerűen ezt írjuk:

```
Employee joe = new Employee();
joe.Age++;
```

A tulajdonságok belső reprezentációja

Sok programozó (különösen azok, akik C-alapú - például C++ - nyelvvel programoznak) hajlamos a `get_` és `set_` előtagokkal megnevezni a tradicionális lekérdező és módosító metódusokat (pl. `get_Name()` és `set_Name()`). Ez az elnevezési hagyomány önmagában nem problémás, amíg a C#-ról van szó. Ugyanezeket az előtagokat használva azonban a köztes nyelvi kód (CIL) egy felszín alatti tulajdonságot jelképez.

Ha például az ildasm.exe használatával megnyitjuk az `EmployeeApp.exe` szerelvényt, akkor azt fogjuk látni, hogy minden egyes tulajdonság azokra a rejtett `get_XXX()/set_XXX()` metódusokra lett leképezve, amelyeket a CLR hív meg belsőleg (lásd 5.10. ábra).



5.10. ábra: *A tulajdonságot belsőleg a get/set metódusok jelképezik*

Tételezzük fel, hogy az `Employee` típus rendelkezik egy privát tagváltozóval (`empSSN` néven), amely az egyének taj számát jelképezi, és amelyet a `SocialSecurityNumber` nevű tulajdonság kezel; valamint azt is tegyük fel, hogy módosítottuk a típus egyedi konstruktorát és a `DisplayStats()` metódust emiatt az új adatmező miatt.

```
// A munkavállaló TAJ számát jelképező új mező támogatása.
class Employee
{
  ...
  private string empSSN;
  public string SocialSecurityNumber
  {
    get { return empSSN; }
    set { empSSN = value; }
  }

  // Konstruktorok
  public Employee() {}
  public Employee(string name, int age, int id, float pay,
    string ssn)
  {
    empName = name;
    empID = id;
    empAge = age;
```

```
    currPay = pay;
    empSSN = ssn;
  }

  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", empName);
    Console.WriteLine("ID: {0}", empID);
    Console.WriteLine("Age: {0}", empAge);
    Console.WriteLine("SSN: {0}", empSSN);
    Console.WriteLine("Pay: {0}", currPay);
  }
  ...
}
```

Ha ugyanebben az osztályban meghatároznánk a `get_SocialSecurityNumber()` és a `set_ SocialSecurityNumber()` metódusokat is, akkor fordítási idejű hibákkal szembesülnénk:

```
// Figyelem, a tulajdonság valójában get_/set_ páros!
class Employee
{
  ...
  public string get_SocialSecurityNumber()
  {
    return empSSN;
  }
  public void set_SocialSecurityNumber(string ssn)
  {
    empSSN = ssn;
  }
}
```

Megjegyzés A .NET-alaposztálykönyvtárak mindig előnyben részesítik a típustulajdonságokat a tradicionális lekérdező és módosító metódusokkal szemben, ha adatmező egységbe zárásáról van szó. Ezért, ha olyan egyedi típusokat szeretnénk építeni, amelyek megfelelően illeszkednek a .NET-platformba, akkor kerüljük el a tradicionális `get` és `set` metódusok alkalmazását.

A get/set utasítástulajdonságok láthatósági szintjeinek vezérlése

A .NET 2.0 elterjedése előtt a `get` és `set` logika láthatóságát egyedül a tulajdonság deklarálásának hozzáférés-módosítója vezérelte:

```
// Mind a get, mind a set logika nyilvános
// a tulajdonság deklarációjának köszönhetően.
public string SocialSecurityNumber
{
  get { return empSSN; }
  set { empSSN = value; }
}
```

Néhány esetben hasznos lehet egyedi elérhetőségi szinteket megadni a `get` és a `set` logikához. Ehhez egyszerűen tegyünk hozzá előtagként egy elérhetőségi kulcsszót a megfelelő (`get` vagy `set`) kulcsszóhoz (a nem minősített hatókör megkapja a tulajdonság deklarációjának láthatóságát):

```
// Az objektum felhasználók csak hozzáférhetnek, az Employee
// osztály és a származtatott típusok beállíthatják az értéket.
public string SocialSecurityNumber
{
  get { return empSSN; }
  protected set { empSSN = value; }
}
```

Ebben az esetben a `SocialSecurityNumber set` logikáját csak az aktuális és a belőle származó osztályok hívhatják meg, ezért nem hívható meg egy objektumpéldányból. (A `protected` kulcsszót lásd a következő fejezetben a származtatás és a polimorfizmus vizsgálatánál.)

Csak olvasható és csak írható tulajdonságok

Az adatok beágyazásakor szükségünk lehet *írásvédett tulajdonság* beállítására is. Ehhez egyszerűen hagyjuk ki a `set` blokkot. Ugyanígy, ha *csak írható tulajdonságot* szeretnénk, hagyjuk ki a `get` blokkot. Példaként nézzük meg, hogy a `SocialSecurityNumber` tulajdonságot hogyan lehet csak olvashatóra módosítani:

```
public string SocialSecurityNumber
{
  get { return empSSN; }
}
```

Ez után a kiigazítás után csak konstruktorparaméterrel lehet beállítani a dolgozó taj számát. Ebből következően fordítási hibát kapnánk, ha az alábbiak szerint próbálnánk meg beállítani egy alkalmazott taj számát:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");
  Employee emp = new Employee("Marvin", 24, 456, 30000,
    "111-11-1111");
  emp.GiveBonus(1000);
  emp.DisplayStats();

  // Hiba! A TAJ szám írásvédett!
  emp.SocialSecurityNumber = "222-22-2222";
  Console.ReadLine();
}
```

Statikus tulajdonságok

A C# statikus tulajdonságokat is támogat. A statikus tagokhoz, ahogy láttuk, csak osztályszinten lehet hozzáférni, az osztály egy példányából (objektumából) nem. Tételezzük fel például, hogy az `Employee` típus meghatároz egy statikus adatelemet, amely a dolgozókat foglalkoztató cég nevét jelöli. Az alábbiak szerint ágyazhatunk be statikus tulajdonságot:

```
// A statikus tulajdonságok statikus adatokkal dolgoznak!
class Employee
{
  ...
  private static string companyName;
  public static string Company
  {
    get { return companyName; }
    set { companyName = value; }
  }
  ...
}
```

A statikus tulajdonságok kezelése ugyanúgy történik, mint a statikus metódusoké:

```
// Munka a statikus tulajdonsággal.
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Encapsulation *****\n");

  // Cég beállítása.
  Employee.Company = "Intertech Training";
  Console.WriteLine("These folks work at {0}.", Employee.Company);
```

```
  Employee emp = new Employee("Marvin", 24, 456, 30000,
    "111-11-1111");
  emp.GiveBonus(1000);
  emp.DisplayStats();

  Console.ReadLine();
}
```

Végül: az osztályok támogathatnak statikus konstruktorokat. Ezért, ha biztosítani szeretnénk, hogy a statikus `companyName` mező neve mindig „Intertech Training" legyen, írjuk a következőt:

```
// Statikus adatokat statikus konstruktorokkal lehet inicializálni.
public class Employee
{
  private Static companyName As string
  ...
  static Employee()
  {
    companyName = "Intertech Training";
  }
}
```

Ennek a megközelítésnek a használatával nem kell kifejezetten meghívni a `Company` tulajdonságot a kezdőérték beállításához:

```
// "Intertech Training" kezdőérték automatikus beállítása statikus
konstruktorral.
static void Main(string[] args)
{
  Console.WriteLine("These folks work at {0}", Employee.Company);
}
```

Ezek a szintaktikai elemek tehát ugyanazt a célt szolgálják, mint a tradicionális lekérdező (`get`) és módosító (`set`) metódusok. A tulajdonságok előnye az, hogy az objektumfelhasználók egyetlen megnevezett elem használatával képesek kezelni a belső adatelemeket.

Megjegyzés A 13. fejezetben megvizsgálunk egy új C# 2008 konstrukciót, amelynek *automatikus tulajdonságok* a neve. Ez a funkció lehetővé teszi tulajdonságdefiníció és releváns privát tagváltozó megadását egy tömör szintaxis felhasználásával.

Konstans adatok

Minthogy már létre tudunk hozni típustulajdonságok révén módosítható mezőket, a következő kérdés: hogyan adhatunk meg olyan adatokat, amelyek soha nem változnak meg a kezdeti értékadás után. A C# biztosítja számunkra a `const` kulcsszót a konstans adatok megadásához. Ez akkor lehet hasznos, amikor olyan ismert értékek készletét adjuk meg az alkalmazásoknak, amelyek logikailag kapcsolódnak egy adott osztályhoz vagy struktúrához.

Az `Employee` példát félretéve, tételezzük fel, hogy egy olyan segédosztályt építünk (`MyMathClass` néven), amelynek meg kell határoznia egy értéket a PI érték számára (amelyet mi a példában 3,14-nek definiálunk). Hozzunk létre egy új parancssorialkalmazás-projektet `ConstData` néven. Ha azt akarjuk, hogy más fejlesztők ne változtathassák meg ezt az értéket a kódban, akkor a következő konstanssal adhatjuk meg a PI-t:

```
namespace ConstData
{
  class MyMathClass
  {
    public const double PI = 3.14;
  }

  class Program
  {
    static void Main(string[] args)
    {
      Console.WriteLine("***** Fun with Const *****\n");
      Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);

      // Hiba! Konstans értéket nem lehet módosítani!
      MyMathClass.PI = 3.1444;
      Console.ReadLine();
    }
  }
}
```

A `MyMathClass` által meghatározott konstans adatra osztálynévprefixummal (pl. `MyMathClass.PI`) hivatkozunk. Ez azért lehetséges, mert egy osztály vagy struktúra konstans mezői implicit módon *statikusak*. Ám egy típustagon belül meg lehet határozni és hozzá lehet férni egy lokális konstans változóhoz. Például:

```
static void LocalConstStringVariable()
{
  // A helyi konstans adatelem közvetlenül hozzáférhető.
  const string fixedStr = "Fixed string Data";
  Console.WriteLine(fixedStr);

  // Hiba!
  fixedStr = "This will not work!";
}
```

Függetlenül attól, hogy hol adtuk meg a konstans adatrészletet, az egyetlen dolog, amire figyelnünk kell, az az, hogy a konstanshoz rendelt kezdőértéket akkor kell megadni, amikor meghatározzuk a konstanst. Ezért, ha úgy szeretnénk módosítani a `MyMathClass` osztályt, hogy a PI értéke osztálykonstruktorban legyen hozzárendelve, a következők szerint:

```
class MyMathClass
{
  // A PI értékének beállítása konstruktorban?
  public const double PI;

  public MyMathClass()
  {
    // Hiba!
    PI = 3.14;
  }
}
```

akkor fordítási idejű hibát kapnánk. Ennek a korlátozásnak az az oka, hogy a konstans adat értékét ismerni kell *a fordítás időpontjában*. Hiszen a konstruktorok hívása *futás közben* történik meg.

Írásvédett mezők

Szorosan kapcsolódik a konstans adatokhoz az *írásvédett adatmező* (amely nem tévesztendő össze az írásvédett tulajdonsággal). A konstanshoz hasonlóan az írásvédett mezőt sem lehet módosítani a kezdeti értékadás után. A konstansokkal ellentétben azonban az írásvédett mezőhöz hozzárendelt érték meghatározható futás közben is, ezért legálisan hozzárendelhető egy konstruktor hatókörén belül (de máshol nem).

Ez akkor lehet hasznos, ha a futásidőig nem tudjuk egy mező értékét (pl. mert egy külső fájlból kell beolvasni), de biztosítani akarjuk, hogy az érték később nem változik. Ennek bemutatására, tegyük fel, hogy az alábbiakkal frissítettük a `MyMathClass` osztályt:

```
class MyMathClass
{
  // írásvédett mezőket csak konstruktorokban lehet hozzárendelni,
  // máshol nem.
  public readonly double PI;
  public MyMathClass ()
  {
    PI = 3.14;
  }
}
```

Ha tehát egy írásvédettként megjelölt mezőhöz a konstruktor hatókörén kívül próbálunk hozzáadni értékeket, akkor fordítási hibát kapunk:

```
class MyMathClass
{
  public readonly double PI;
  public MyMathClass ()
  {
    PI = 3.14;
  }

  // Hiba!
  public void ChangePI()
  { PI = 3.14444;}
}
```

Statikus írásvédett mezők

A konstans mezőkkel ellentétben az írásvédett mezők nem implicit módon statikusak. Ezért, ha osztályszinten szeretnénk megmutatni a PI-t, akkor kizárólag a `static` kulcsszót használhatjuk. Ha a fordítás idején már ismerjük a statikus írásvédett mező értékét, akkor a kezdeti értékadás nagyon hasonlít egy konstanshoz:

```
class MyMathClass
{
  public static readonly double PI = 3.14;
}
```

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with Const *****");
    Console.WriteLine("The value of PI is: {0}", MyMathClass.PI);
    Console.ReadLine();
  }
}
```

Ha azonban a statikus írásvédett mező értéke a futás idejéig ismeretlen, akkor egy statikus konstruktort kell használnunk a korábban már leírtak szerint:

```
class MyMathClass
{
  public static readonly double PI;

  static MyMathClass()
  { PI = 3.14; }
}
```

Mindezek után térjünk vissza az `Employee` példához.

Forráskód A `ConstData` projekt megtalálható az 5. fejezet alkönyvtárában.

Részleges típusok

Az osztályok és a struktúrák meghatározhatók a partial nevű típusmódosítóval, amely lehetővé teszi, hogy több `*.cs` fájlban is meghatározzuk a típust. A nyelv korábbi verzióiban egy adott típus teljes forráskódját egyetlen `*.cs` fájlon belül kellett meghatározni. Mivel egy termékszintű C#-osztály több száz sornyi (vagy még több) kódból állhat, ez valóban hosszú fájlt eredményezhet.

Ezekben az esetekben hasznos lehet részekre, több `*.cs` fájlba bontani a típus implementációját annak érdekében, hogy elkülönítsük a típus definíciójának szempontjából valamiért fontosabb kódokat. A részleges osztálymódosító használatával például az összes Employee konstruktort és tulajdonságot egy helyre, az `Employee.Internals.cs` nevű fájlba helyezhetjük át:

```
partial class Employee
{
  // Konstruktorok.
  ...
  // Tulajdonságok.
  ...
}
```

A privát adatmezők és típusmetódusok a kezdeti `Employee.cs` fájlban lesznek meghatározva:

```
partial class Employee
{
  // Mezőadatok.
  private string empName;
  private int empID;
  private float currPay;
  private int empAge;
  private string empSSN;
  private static string companyName;

  public void GiveBonus(float amount)
  {
    currPay += amount;
  }

  public void DisplayStats()
  {
    Console.WriteLine("Name: {0}", empName);
    Console.WriteLine("ID: {0}", empID);
    Console.WriteLine("Age: {0}", empAge);
    Console.WriteLine("SSN: {0}", empSSN);
    Console.WriteLine("Pay: {0}", currPay);
  }
}
```

Ez kifejezetten hasznosnak bizonyulhat az új csapattagok számára, akiknek minél gyorsabban meg kell ismerkedniük a típus nyilvános interfészével. Ahelyett, hogy végig kéne olvasniuk egyetlen (hosszú) C#-fájlt, hogy megtalálják a lényeges tagokat, a nyilvános tagokra figyelhetnek. Ha ezeket a fájlokat a C#-fordító lefordítja, akkor az eredmény egyetlen, egyesített típus lesz. Így a partial módosító csupán egy tervezési idővel kapcsolatos konstrukció.

Azt is fontos tudni, hogy a részleges típusdefiníciókat tartalmazó fájlok neve teljes mértékben rajtunk múlik. Itt az `Employee.Internal.cs` nevet pusztán annak jelölésére választottuk, hogy a fájl csak csúnya infrastruktúrakódot tartalmaz, amelyet a legtöbb fejlesztő figyelmen kívül hagyhat.

Az egyetlen követelmény a részleges típusok meghatározásakor az, hogy a típus neve (jelen esetben az `Employee`) ugyanaz legyen, és ugyanazon a .NET-névtéren belül határozódjon meg.

Megjegyzés A Visual Studio 2008 arra használja a partial kulcsszót, hogy részekre ossza az IDE tervezői eszközei (pl. különböző GUI-tervezők) által generált forráskódot. Ezzel a megközelítéssel az aktuális megoldásra összpontosíthatunk, és figyelmen kívül hagyhatjuk a tervező által generált forráskódot.

A C#-forráskód dokumentálása XML segítségével

Végezetül megvizsgálunk egy olyan módszert a forráskód kommentálásához, amely XML-alapú kóddokumentációt eredményez. Ha már dolgoztunk a Java nyelvvel, akkor ismerős lehet a javadoc segédprogram. A javadoc használatával megfelelően ábrázolhatjuk HTML-ben a Java-forráskódot (biztosítva, hogy a *.java fájl a helyes kódmegjegyzés-szintaxist tartalmazza). A C#-dokumentáció modellje ettől egy kicsit annyiban különbözik, hogy a megjegyzések XML-be történő kódolása a C#-fordító feladata (a `/doc` opció révén), nem pedig egy különálló segédprogramé.

Miért használjunk XML-t a típusdefiníciók kódolásához HTML helyett? A fő ok az, hogy az XML nagyon „engedélyező technológia". Mivel az XML elkülöníti az adatok definícióját ugyanazon adatok megjelenítésétől, ezért bármennyi XML-transzformációt végrehajthatunk az alapjául szolgáló XML-fájlon, hogy a dokumentációt különböző formátumokban (MSDN-formátum, HTML stb.) jelenítsük meg.

Amikor XML-ben szeretnénk dokumentálni a C#-típusokat, az első lépés az új hármas perjel (///) használata a kódmegjegyzés jelölésére. Amint egy megjegyzést deklaráltunk, szabadon használhatunk bármilyen jól formázott XML-elemet, beleértve az 5.2. táblázatban használt ajánlott készletet.

Előre meghatározott XML-dokumentumelem	Jelentés
<c>	Azt jelzi, hogy a következő szöveget egy bizonyos „kódbetűtípussal" kell megjeleníteni.

Előre meghatározott XML-dokumentumelem	Jelentés
`<code>`	Azt jelzi, hogy több sort kódként kell megjelölni.
`<example>`	Egy kódpéldát mintáz a leírt elem számára.
`<exception>`	Dokumentálja, hogy egy adott osztály milyen kivételeket dobhat.
`<list>`	Beszúr egy listát vagy egy táblázatot a dokumentációs fájlba.
`<param>`	Leír egy adott paramétert.
`<paramref>`	XML-taget társít egy adott paraméterhez.
`<permission>`	Dokumentálja egy adott tag biztonsági korlátozásait.
`<remarks>`	Felépít egy leírást egy adott tag számára.
`<returns>`	Dokumentálja a tag visszatérési értékét.
`<see>`	Kereszthivatkozás a dokumentum kapcsolódó elemei között.
`<seealso>`	Létrehoz egy „lásd még" részt egy leíráson belül.
`<summary>`	Dokumentálja a „végrehajtási összegzést" egy adott tag esetében.
`<value>`	Dokumentál egy adott tulajdonságot.

5.2. táblázat: *A forráskód-megjegyzések ajánlott XML-elemei*

Ha használjuk az új C# XML-kódmegjegyzés jelöléseit, legyünk tisztában azzal, hogy a Visual Studio 2008 IDE a mi kérésünkre fogja a dokumentáció vázát létrehozni. Ha például három perjelet teszünk az `Employee` osztály definíciója fölé, az a következő vázat eredményezi:

```
/// <summary>
///
/// </summary>
partial class Employee
{
  ...
}
```

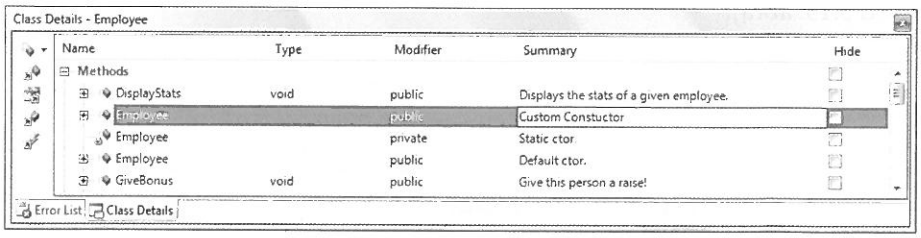
Egyszerűen töltsük ki az üres helyeket egyedi megjegyzéseinkkel:

```
/// <summary>
/// Ez az osztály egy alkalmazottat képvisel.
/// </summary>
partial class Employee
{
  ...
}
```

Másik példaként szúrjunk be három perjeles kommentet az egyedi ötparaméteres konstruktorunkba. Ez alkalommal a kommentező segédprogram hozzáadja a <param> elemeket:

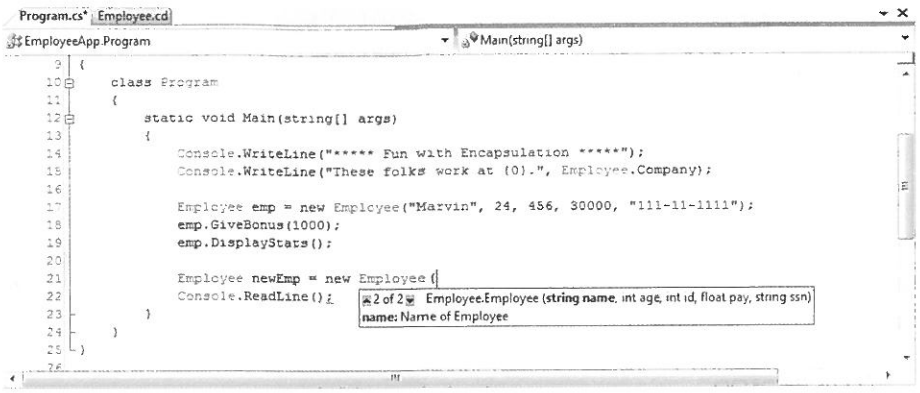
```
/// <summary>
///
/// </summary>
/// <param name="name"></param>
/// <param name="age"></param>
/// <param name="id"></param>
/// <param name="pay"></param>
/// <param name="ssn"></param>
public Employee(string name, int age, int id, float pay, string ssn)
{
  empName = name;
  empID = id;
  empAge = age;
  currPay = pay;
  empSSN = ssn;
}
```

Ezeket az XML-kódmegjegyzéseket a Visual Studio 2008 Class Details ablakának (lásd 2. fejezet) használatával is beírhatjuk, ahogy az 5.11. ábrán is látható.



5.11. ábra: *XML-megjegyzések hozzáadása a Class Details ablak használatával*

Egyik előnye annak, ha ellátjuk a forráskódot XML-megjegyzésekkel, az, hogy később megnézhetjük ezt az információt a Visual Studio IntelliSense-én belül is (lásd 5.12. ábra). Ez olyanok számára lehet hasznos, akik esetleg nem ismerik egy adott típustag szerepét.



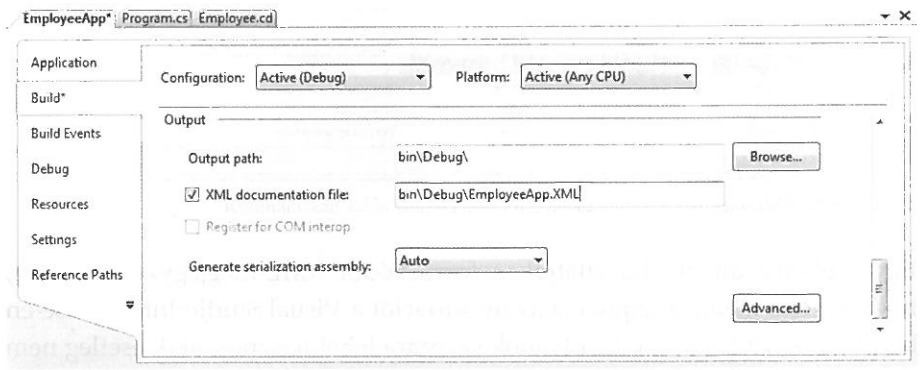
5.12. ábra: *Az XML-megjegyzések az IntelliSense révén is megtekinthetők*

Az XML-fájl generálása

Ha elláttuk XML-megjegyzésekkel a forráskódunkat, a következő lépés egy megfelelő `*.xml` fájl generálása lesz az XML-adatok alapján. Ha a parancssoros fordítóval (`csc.exe`) építjük fel a C#-programunkat, akkor a `/doc` kapcsolót használjuk az XML-kódmegjegyzéseken alapuló `*.xml` fájl létrehozására:

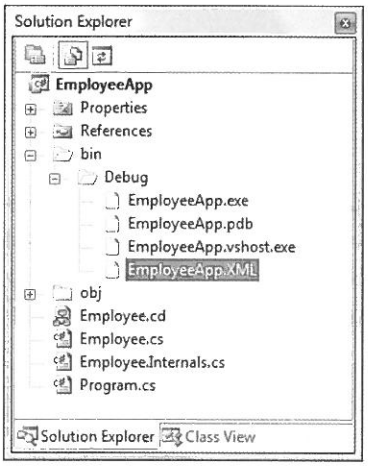
```
csc /doc:XmlCarDoc.xml *.cs
```

A Visual Studio 2008 projektek lehetővé teszik az XML dokumentációs fájl nevének megadását a „Generate XML documentation file" jelölőnégyzet lehetőségének felhasználásával, amely a Properties ablak Build lapján található (lásd 5.13. ábra).



5.13. ábra: *XML-kódmegjegyzésfájl létrehozása a Visual Studio 2008-ban*

Ha ezt engedélyeztük, akkor a fordító a projekt \bin\Debug mappájába helyezi a létrehozott *.xml fájlt. Ezt leellenőrizhetjük, ha rákattintunk a Show All Files gombra a Solution Exploreren – az eredmény az 5.14. ábrán látható.



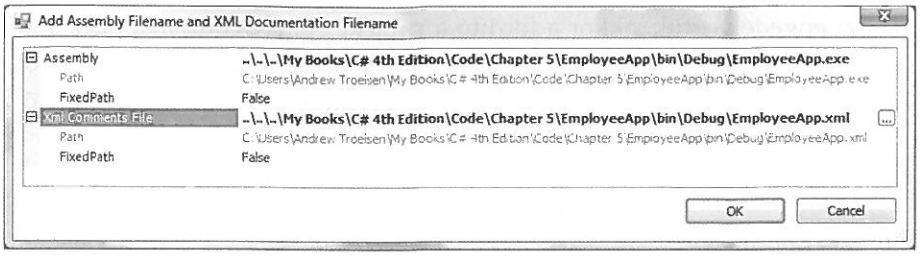
5.14. ábra: *A létrehozott XML-dokumentációfájl helyének meghatározása*

Megjegyzés Létezhet sok más elem és jelölés, amelyek megjelenhetnek a C# XML-kommentjeiben. További részletekért lapozzuk fel az „XML Documentation Comments (C# Programming Guide)" témakört a .NET Framework SDK 3.5 dokumentációjában.

XML-megjegyzések átalakítása az NDoc-on keresztül

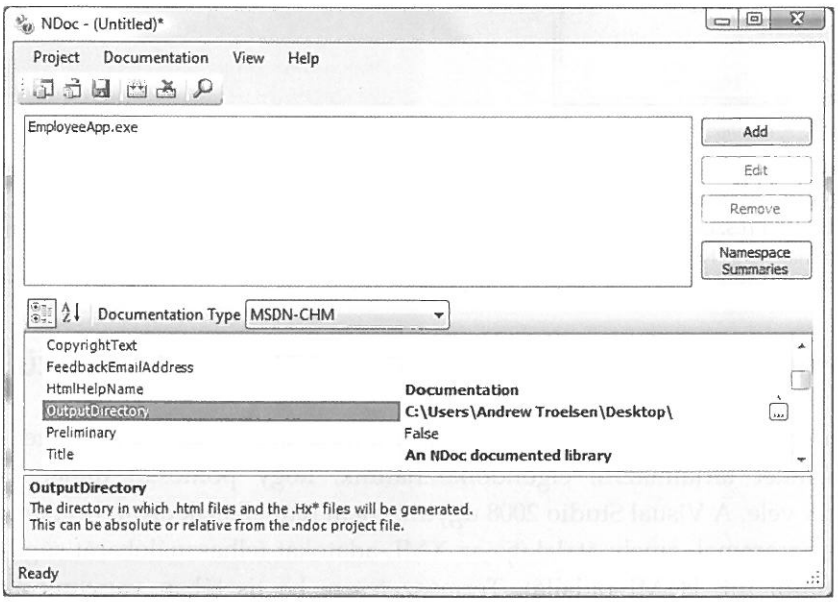
Most, hogy létrehoztuk az *.xml fájlt, amely a forráskóddal kapcsolatos megjegyzéseinket tartalmazza, elgondolkozhatunk, hogy pontosan mihez is kezdjünk vele. A Visual Studio 2008 ugyanis nem rendelkezik olyan beépített segédprogrammal, amely átalakítja az XML-adatokat felhasználóbarát súgóformátumra (pl. HTML-oldallá). Természetesen, ha tisztában vagyunk az XML-transzformációk be- és kimeneteivel, nyugodtan hozzuk létre manuálisan saját stíluslapunkat.

Egyszerűbb alternatívaként azonban létezik számos, külső gyártótól származó eszköz, amelyek lefordítják az XML-kódfájlokat különböző hasznos formátumokra. Az NDoc alkalmazás például számos különböző formátumban létre tudja hozni a dokumentációt. Tegyük fel, hogy már telepítettük ezt az eszközt, így az első lépés az, hogy megadjuk az *.xml fájlunk és a megfelelő szerelvény helyét. Ehhez kattintsunk az Add gombra az NDoc grafikus felületén. Ekkor megnyílik az 5.15. ábrán is látható párbeszédpanel.



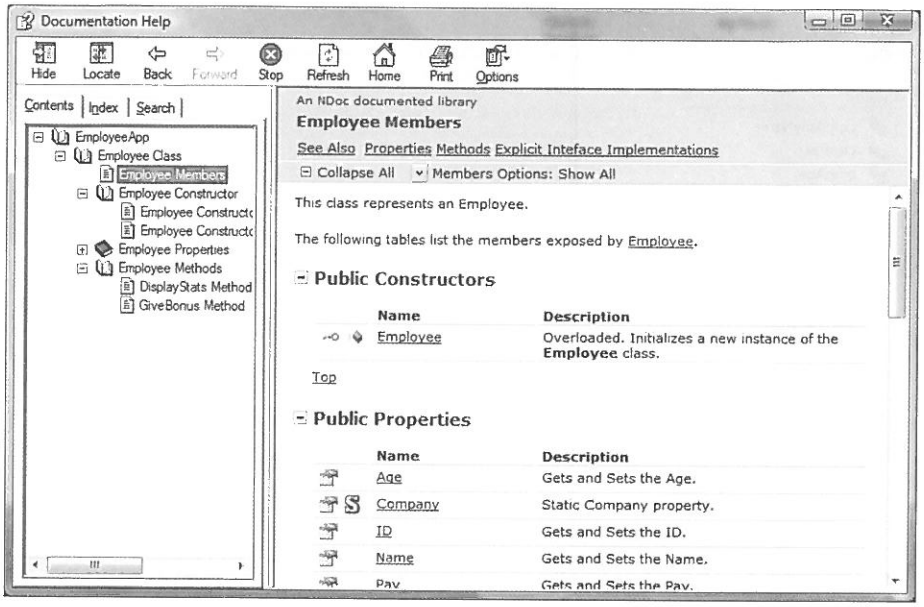
5.15. ábra: *Az XML-fájl és a megfelelő szerelvény megadása*

Ekkor válasszuk ki a kimeneti helyet (az `OutputDirectory` tulajdonságon keresztül) és a dokumentumtípust (a Document Type legördülő listából). Ehhez a példához válasszuk az MSDN-CHM formátumot, ez olyan dokumentációt hoz létre, amely hasonlít a .NET Framework 3.5 dokumentációjára (lásd 5.16. ábra).



5.16. ábra: *A kimeneti könyvtár és a dokumentáció formátumának megadása*

Természetesen más beállításokat is használhatunk az NDoc grafikus felületén, ha azonban kiválasztjuk a Documentation > Build menüt, akkor az NDoc a teljes súgórendszert generálja az alkalmazásunkhoz. Az 5.17. ábra mutatja a végeredményt.



5.17. ábra: *Az MSDN-hez hasonló súgórendszer az EmployeeApp projekt számára*

Megjegyzés Jelen írás idején a Microsoft Community Technology Preview-ként (CTP) kiadott egy Sandcastle nevű eszközt, amely működésében hasonló a nyílt forrású NDoc-segédprogramhoz. További információkért keressük fel a http://www. sandcastledocs.com oldalt (az URL változhat).

Az elvégzett munka eredményének láthatóvá tétele

Létrehoztunk tehát egy viszonylag érdekes osztályt `Employee` néven. Ha használjuk a Visual Studio 2008-at, akkor beszúrhatunk egy új osztálydiagramfájlt (lásd 2. fejezet), hogy megnézhessük (és karbantarthassuk) az osztályunkat a tervezési időben. Az 5.18. ábra mutatja a kész `Employee` osztálytípust.

A következőkben látni fogjuk, hogy ez az `Employee` osztály ősosztályként fog működni a származtatott osztálytípusok (`WageEmployee`, `SalesEmployee` és `Manager`) családjának a számára.

Forráskód Az EmployeeApp projekt megtalálható az 5. fejezet alkönyvtárában.

5.18. ábra: *A befejezett Employee osztály*

Összefoglalás

Ennek a fejezetnek az volt a célja, hogy bemutassa a C# osztálytípusszerepét. Az osztályok felvehetnek bármennyi *konstruktort*, amelyek lehetővé teszik az objektum használója számára, hogy a létrehozás idején megadja az objektum állapotát. Emellett több osztálytervezési technikát bemutattunk (a kapcsolódó kulcsszavakkal együtt). A `this` kulcsszóval például hozzáférhetünk az aktuális objektumhoz, a static kulcsszó lehetővé teszi az osztály- (és nem objektum-) szinten kötött mezők és tagok meghatározását, míg a `const` kulcsszó (és a `readonly` módosító) segítségével megadhatunk olyan adatelemet, amelyet a kezdeti értékadás után többet nem lehet módosítani.

Foglalkoztunk az objektumorientált programozás első alappillérével, az egységbe zárással. Képet kaptunk a C# hozzáférés-módosítóiról, valamint a típustulajdonságok, a részleges osztályok és az XML-forráskód-dokumentáció szerepéről. A következő fejezetben megvizsgáljuk, hogyan lehet felépíteni a kapcsolódó osztályok családját a származtatás és a polimorfizmus használatával.

HATODIK FEJEZET

A származtatás és a polimorfizmus

Az előző fejezetben az objektumorientált programozás első alappillérét: az *egységbe zárást* vizsgáltuk. Megtanultuk, hogyan építsünk egyetlen, jól meghatározott osztálytípust konstruktorokkal és különböző tagokkal (mezők, tulajdonságok, konstansok, írásvédett mezők stb.) Ez a fejezet az OOP két másik alappillérére összpontosít: a származtatásra és a polimorfizmusra.

Először megvizsgáljuk, hogyan építsük fel a kapcsolódó osztályok családját a *származtatás* használatával. Ez a fajta kód-újrafelhasználás lehetővé teszi, hogy közös funkciót határozzunk meg egy olyan szülőosztályban, amelyet a gyermekosztályok kihasználhatnak (és esetleg megváltoztathatnak). Kiderül az is, hogy hogyan helyezzünk el egy *polimorf interfészt* az osztályhierarchiákba virtuális és absztrakt tagok felhasználásával. Vizsgálódásunkat végül a .NET alaposztálykönyvtáraiban megtalálható legfőbb szülőosztály szerepének tanulmányozásával zárjuk: a `System.Objecttel`.

A származtatás alapvető szerkezete

A *származtatás* elősegíti a kód újrafelhasználását (lásd az előző fejezetet). Pontosabban a kód újrafelhasználása két lehetőséget biztosít: a klasszikus származtatást („az egy" kapcsolat) és a tartalmazás/delegálás modellt („van egy" kapcsolat). Kezdjük a fejezetet a klasszikus, „az egy" származtatási modellel.

Amikor „az egy" kapcsolatot hozunk létre az osztályok között, akkor függőséget építünk ki két vagy több osztálytípus között. A klasszikus származtatás alapötlete az, hogy az új osztályok használhatják (és kibővíthetik) a már meglévő osztályok funkcionalitását. Kezdjük egy nagyon egyszerű példával: hozzunk létre egy új parancssorialkalmazás-projektet `BasicInheritance` néven. Tegyük fel, hogy megterveztünk egy olyan egyszerű osztályt `Car` néven, amely egy autó alaprészleteit modellezi:

```
// Egyszerű alaposztály.
class Car
{
  public readonly int maxSpeed;
  private int currSpeed;

  public Car(int max)
  {
    maxSpeed = max;
  }
  public Car()
  {
    maxSpeed = 55;
  }

  public int Speed
  {
    get { return currSpeed; }
    set
    {
      currSpeed += value;
      if (currSpeed > maxSpeed)
      {
        currSpeed = maxSpeed;
      }
    }
  }
}
```

A `Car` osztály arra használja az egységbe zárást, hogy vezérelje a hozzáférést a privát `currSpeed` mezőhöz a `Speed` nevű nyilvános tulajdonsággal. Ezután a következők szerint érvényesíthetjük `Car` típusunkat:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");
  // Car típus létrehozása.
  Car myCar = new Car(80);
  myCar.Speed = 50;
  Console.WriteLine("My car is going {0} MPH", myCar.Speed);
  Console.ReadLine();
}
```

Az osztálytípus szülőosztálya

Tételezzük fel, hogy szeretnénk építeni egy `Minivan` nevű új osztályt. Az alap `Car` osztályhoz hasonlóan a `Minivan` osztályt is úgy szeretnénk definiálni, hogy támogassa a maximális és az aktuális sebességet, valamint egy `Speed` nevű tu-

lajdonságot, amely révén az objektumhasználó módosíthatja az objektum állapotát. Világos, hogy a `Car` és a `MiniVan` osztályok kapcsolódnak egymáshoz; vagyis azt mondhatjuk, hogy a `MiniVan` *„az egy"* `Car`. Az „az egy" kapcsolat (hivatalos nevén *klasszikus származtatás*) lehetővé teszi olyan új osztálydefiníciók építését, amelyek kibővítik egy már létező osztály funkcionalitását.

A már meglévő osztály - amely az új osztály alapjául fog szolgálni - neve *alap-* vagy *szülőosztály*. Az alaposztály szerepe az, hogy definiálja az őt kibővítő osztályok számára az összes közös adatot és tagot. A kibővítő osztályok hivatalos neve *származott* vagy *gyermekosztály*. A C#-ban a kettőspont operátort használjuk az osztálydefiníción, hogy létrehozzunk egy „az egy" kapcsolatot az osztályok között:

```
// A MiniVan "az egy" Car.
class MiniVan : Car
{
}
```

Mit nyertünk azzal, hogy a `MiniVan` osztályt a `Car` alaposztályból származtattuk? A `MiniVan` objektumok most már hozzáférnek a szülőosztályon belül megadott nyilvános tagokhoz. A két osztálytípus közti kapcsolat révén például a következőképpen használhatjuk a `MiniVan` típust:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");
  ...
  // MiniVan típus készítése.
  MiniVan myVan = new MiniVan();
  myVan.Speed = 10;
  Console.WriteLine("My van is going {0} MPH",
  myVan.Speed);
  Console.ReadLine();
}
```

Noha nem adtunk tagokat a `MiniVan` osztályhoz, mégis közvetlen hozzáférésünk van a szülőosztály nyilvános `Speed` tulajdonságához, így újra felhasználtuk a kódot. Az egységbe zárás megőrződik; ezért a következő kód fordítási hibát eredményez:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Basic Inheritance *****\n");
  ...
```

```
  // MiniVan típus készítése.
  MiniVan myVan = new MiniVan();
  myVan.Speed = 10;
  Console.WriteLine("My van is going {0} MPH",
  myVan.Speed);

  // Hiba! Privát tagok nem hozzáférhetők objektum referenciákkal!
  myVan.currSpeed = 55;
  Console.ReadLine();
}
```

Meg kell jegyeznünk, ha a `MiniVan` saját tagokat definiál, akkor azokban sem férhet hozzá a `Car` ősosztály privát tagjaihoz:

```
// A MiniVan a Car osztályból származik.
class MiniVan : Car
{
  public void TestMethod()
  {
    // OK! Származtatott típuson belül a szülő
    // nyilvános tagjai hozzáférhetőek.
    Speed = 10;

    // Hiba! Származtatott típuson belül a szülő
    // privát tagjai nem hozzáférhetőek.
    currSpeed = 10;
  }
}
```

Több ősosztály

Az ősosztályokról szólva fontos tudnivaló, hogy a .NET-platform megköveteli, hogy egy adott osztálynak pontosan *egy* közvetlen ősosztálya legyen. Nem lehet létrehozni olyan osztálytípust, amely közvetlenül két vagy több ősosztályból származik (ezt a technikát [amelyet más C-alapú nyelvek, pl. a nemfelügyelt C++, támogatnak] úgy ismerik, mint *többszörös öröklést* [Multiple Inheritance]):

```
// Szabálytalan! A .NET-platform nem teszi lehetővé
// az osztályok többszörös öröklését!
class WontWork
  : BaseClassOne, BaseClassTwo
{}
```

A .NET-platform lehetővé teszi, hogy egy adott osztály- (vagy struktúra-) típus megvalósítson bármennyi különálló interfészt (lásd 9. fejezet). Ily módon egy C#-osztály számos viselkedést valósíthat meg, miközben kikerüli a többszörös öröklés összetettségét. Érdemes megemlíteni, hogy míg egy osztálynak csak egy közvetlen ősosztálya lehet, addig egy interfész közvetlenül több interfészből is származhat. Ennek a technikának a használatával kifinomult interfészhierarchiákat építhetünk, amelyek összetett viselkedést modelleznek (lásd 9. fejezet).

A sealed kulcsszó

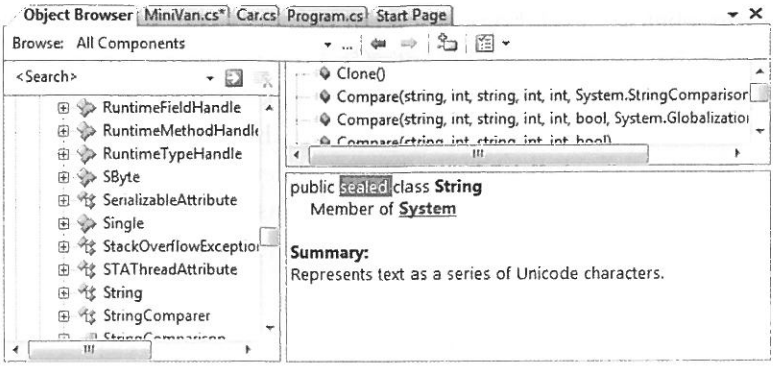
A C# rendelkezik egy másik kulcsszóval (`sealed`) is, amely *megakadályozza* a származtatást. Ha a `sealed` kulcsszóval jelölünk meg egy osztályt, akkor a fordító nem engedi meg, hogy ebből a típusból származtassuk. Tegyük fel például, hogy úgy döntöttünk, nincs értelme tovább bővíteni a `MiniVan` osztályt:

```
// Ez az osztály nem bővíthető!
sealed class MiniVan : Car
{
}
```

Ha megpróbálnánk származtatni ebből az osztályból, akkor fordítási idejű hibát kapnánk:

```
// Hiba! A sealed kulcsszóval jelölt
// osztályokat nem lehet bővíteni!
class DeluxeMiniVan
  : MiniVan
{}
```

Az osztály lezárásának leginkább akkor van értelme, ha segédosztályt tervezünk. A System névtér például számos lezárt osztályt határoz meg. Ezt mi magunk is leellenőrizhetjük, ha megnyitjuk a Visual Studio 2008 Object Browserét (a View menün keresztül), és kiválasztjuk az mscorlib.dll szerelvényben definiált `System.String` típust. A 6.1. ábrán láthatjuk a `Summary` ablakban kiemelt `sealed` kulcsszót.



6.1. ábra: *Az alaposztálykönyvtárak számos lezárt típust definiálnak*

Ezért, ha a `MiniVan` osztályhoz hasonlóan megpróbálnánk építeni egy olyan új osztályt, amely kibővíti a `System.String` típust, akkor fordítási idejű hibát kapnánk:

```
// Másik hiba! A sealed kulcsszóval jelölt
// osztályokat nem lehet bővíteni!
class MyString
  : String
{}
```

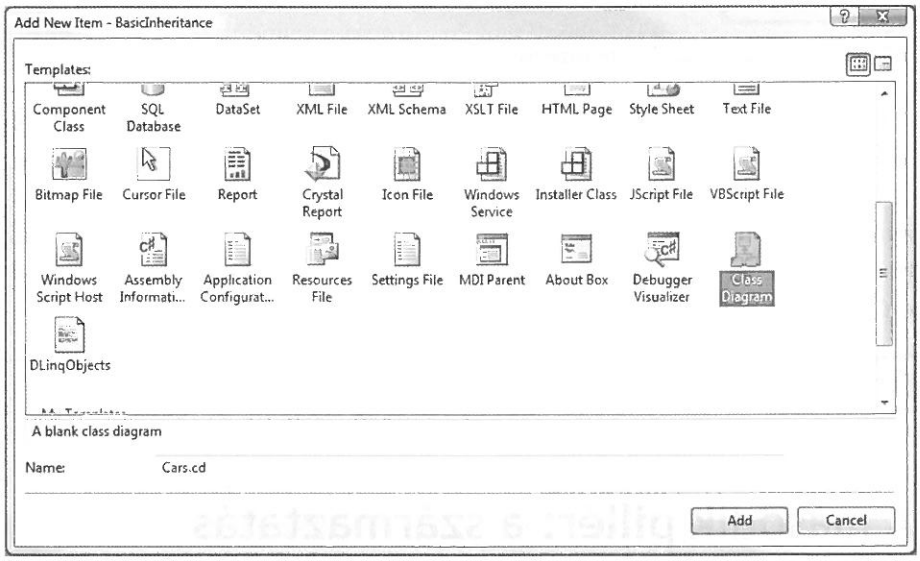
Megjegyzés Ahogy a 4. fejezetben láttuk, hogy a C#-struktúrák mindig implicit módon vannak lezárva (lásd a 4.3. táblázatot). Ezért soha nem származtathatunk egy struktúrát egy másikból, egy osztályt egy struktúrából, illetve egy struktúrát egy osztályból.

A származtatással kapcsolatos más részletekkel a későbbiekben fogunk megismerkedni. Egyelőre szögezzük el azt, hogy a kettőspont operátor teszi lehetővé alap/leszármazott osztályok kapcsolatát, míg a `sealed` kulcsszó megakadályozza a származtatást.

Megjegyzés A C# 2008 bevezette a bővítménymetódusok koncepcióját. A 13. fejezetben látni fogjuk, ez a technika teszi lehetővé új szerep hozzáadását előfordított típusokhoz (beleértve a lezárt típusokat is) az aktuális projekten belül.

Osztálydiagramok készítése a Visual Studio 2008-ban

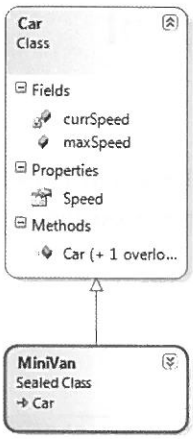
A 2. fejezetben röviden volt szó arról, hogy a Visual Studio 2008 lehetővé teszi ős/származtatott osztálykapcsolatok vizuális létrehozását a tervezési időben. Ehhez az első lépés egy új osztálydiagramfájl hozzáadása az aktuális projekthez. Ehhez nyissuk meg a Project > Add New Item menüt, majd válasszuk a Class Diagram ikont (a 6.2. ábrán átnevezzük a fájlt `ClassDiagram1.cd`-ről `Cars.cd`-re).



6.2. ábra: *Új osztálydiagram beszúrása*

Ha rákattintunk az Add gombra, megjelenik egy üres tervezőfelület. Ahhoz, hogy fájlokat adjunk hozzá az osztálytervezőhöz, egyszerűen húzzuk be őket a Solution Explorer ablakból a tervezőfelületre. Ekkor az IDE automatikusan hozzáadja az összes típust a tervezőfelülethez. Ha kitörlünk egy elemet a vizuális tervezőből, akkor a hozzátartozó forráskód nem vész el, hanem egyszerűen csak kikerül a felületről. Az aktuális osztályhierarchiánkat a 6.3. ábra mutatja.

Megjegyzés Ha automatikusan hozzá szeretnénk adni projektünk összes típusát a tervezőfelülethez, akkor a Solution Exploreren belül válasszuk a Project csomópontot, majd kattintsunk a View Class Diagram gombra a Solution Explorer ablak jobb felső sarkában.



6.3. ábra: *A Visual Studio vizuális tervezője*

Az aktuális alkalmazásban lévő típusok közti kapcsolatok megjelenítése mellett új típusokat is létrehozhatunk (és feltölthetjük tagjaikat) a Class Designer eszköztár és a Class Details ablak használatával (lásd 2. fejezet). Ezeket a vizuális eszközöket a továbbiakban is fel lehet használni, ha kényelmesnek tűnnek. Viszont mindig elemezzük a létrehozott forráskódot, hogy egyértelművé váljon, mit csináltak ezek az eszközök.

Forráskód A `BasicInheritance` projekt megtalálható a 6. fejezet alkönyvtárában.

A második pillér: a származtatás

A származtatási szintaxis alapjai után hozzunk létre egy összetettebb példát, hogy megismerkedjünk az osztályhierarchia építésének jó néhány részletével. Ehhez újra fel fogjuk használni az előző fejezetben tervezett `Employee` osztályt. Kiindulásként hozzunk létre egy új C# parancssori alkalmazást `Employees` névvel. Ezután a Project > Add Existing Item menüben keressük meg az `Employee.cs` és `Employee.Internals.cs` fájlokat. Válasszuk ki őket (Ctrl + klikk), majd kattintsunk az OK gombra. A Visual Studio 2008 bemásolja a fájlokat az aktuális projektbe.

Mielőtt elkezdenénk építeni a származtatott osztályokat, egy apróságra még ügyelnünk kell. Mivel az `Employee` osztályt az `EmployeeApp` nevű projektben hoztuk létre, a típus egy ugyanígy elnevezett .NET-névtér hatókörben van. A 15. fejezet fogja részletesen megvizsgálni ezt a névteret; most az egyszerűség kedvéért nevezzük át az aktuális névteret (mindkét fájlban) `Employee`-ra, hogy egyezzen az új projektnévvel:

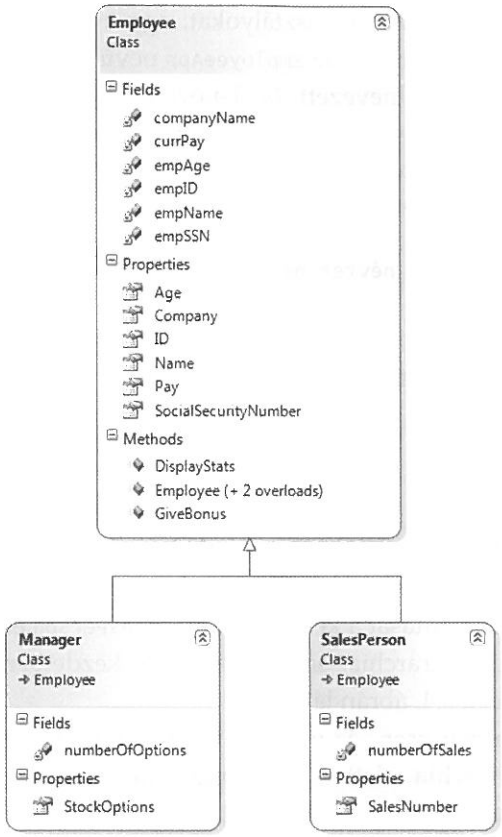
```
// Mindkét fájlban módosítani kell a névtér nevét!
namespace Employees
{
  /// <summary>
  /// Ez az osztály egy alkalmazottat ábrázol.
  /// </summary>
  partial class Employee
  {...}
}
```

A célunk az, hogy létrehozzuk az osztályok egy olyan családját, amely egy cég különböző alkalmazottait modellezi. Tegyük fel, hogy ki szeretnénk használni az `Employee` osztály funkcionalitását a két új osztály (`SalesPerson` és `Manager`) megalkotásához. Az osztályhierarchia, amelyet építünk, kezdetben hasonlóan fog kinézni, mint amelyik a 6.4. ábrán látható.

A 6.4. ábrán láthatjuk, hogy a `SalesPerson` „az egy" `Employee` (mint a `Manager`). Az osztályszármaztatási hierarchia alatt az alaposztályok (mint az `Employee`) határozzák meg azokat az általános jellemzőket, amelyek közösek minden leszármazottnál. Az alosztályok (mint a `SalesPerson` és a `Manager`) kibővítik ezt az általános működést, még sajátságosabb viselkedést adnak hozzá.

A példánkhoz tegyük fel, hogy a `Manager` osztály úgy bővíti ki az `Employee` osztályt, hogy rögzíti a részvényopciók számát, míg a `SalesPerson` osztály karbantartja az eladások számát. Szúrjunk be egy új osztályfájlt (`Manager.cs`), amely a következők szerint határozza meg a `Manager` típust:

```
// A vezetőknek tudniuk kell a részvényopcióik számát.
class Manager : Employee
{
  private int numberOfOptions;
  public int StockOptions
  {
    get { return numberOfOptions; }
    set { numberOfOptions = value; }
  }
}
```



6.4. ábra: *A kezdeti Employees hierarchiája*

Ezután adjunk hozzá egy új osztályfájlt (`SalesPerson.cs`), amely a `SalesPerson` típust határozza meg:

```
// A kereskedőknek tudniuk kell hány üzletet kötöttek.
class SalesPerson : Employee
{
  private int numberOfSales;
  public int SalesNumber
  {
    get { return numberOfSales; }
    set { numberOfSales = value; }
  }
}
```

Ezzel létrehoztunk egy „az egy" kapcsolatot, így a `SalesPerson` és a `Manager` osztályok automatikusan öröklik az `Employee` alaposztály összes nyilvános tagját. Ennek bemutatásához a következők szerint módosítsuk `Main()` metódusunkat:

```
// Alosztály létrehozása, és az alaposztály funkcionalitásának
// használata.
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");
  SalesPerson danny = new SalesPerson();
  danny.Age = 31;
  danny.Name = "Danny";
  danny.SalesNumber = 50;
  Console.ReadLine();
}
```

Ősosztály konstruktorának meghívása a base kulcsszóval

A `SalesPerson` és `Manager` osztályok jelenleg csak az alapértelmezett konstruktor használatával hozhatók létre (lásd 5. fejezet). Tételezzük fel, hogy hozzáadtunk egy új, hatparaméteres konstruktort a `Manager` típushoz, amely az alábbiak szerint aktivizálódik:

```
static void Main(string[] args)
{
  ...
  // A Manager osztály konstruktora a következő aláírással
  // rendelkezik: (string fullName, int age, int empID,
  // float currPay, string ssn, int numbOfOpts).
  Manager chucky = new Manager("Chucky", 50, 92, 100000,
    "333-23-2322", 9000);
  Console.ReadLine();
}
```

Ha megnézzük a paraméterlistát, látható, hogy a paraméterek legtöbbjét az `Employee` ősosztály által definiált tagváltozókban kell tárolni. Ehhez a következők szerint valósíthatjuk meg ezt az egyedi konstruktort a `Manager` osztályon:

```
public Manager(string fullName, int age, int empID,
  float currPay, string ssn, int numbOfOpts)
{
  // A mezőt a Manager osztály definiálja.
  numberOfOptions = numbOfOpts;

  // Bejövő paraméterek hozzárendelése a
  // szülőosztály örökölt tulajdonságainak segítségével.
  ID = empID;
  Age = age;
  Name = fullName;
  Pay = currPay;
```

```
    // HOPPÁ! Ez fordítási hiba lenne,
    // mivel az SSN tulajdonság írásvédett!
    SocialSecurityNumber = ssn;
}
```

Az első megállapítás ennél a megközelítésnél az, hogy a szülőosztályban írásvédettként határoztuk meg a `SocialSecurityNumber` tulajdonságot; ezért ehhez a mezőhöz nem rendelhetünk hozzá bejövő sztringparamétert ennek az egyedi konstruktornak az utolsó sorában.

A második megállapítás az, hogy közvetve létrehoztunk egy nem túl hatékony konstruktort, ugyanis C# alatt, ha csak máshogy nem rendelkezünk, az ősosztály alapértelmezett konstruktorának meghívása automatikusan végbemegy, mielőtt a származott konstruktor logikájának a végrehajtása megtörténik. Ezután az aktuális implementáció hozzáfér az `Employee` ősosztály nyilvános tulajdonságaihoz, hogy megállapítsa az állapotát. Így igazából hét legyet ütöttünk egy csapásra (öt származtatott tulajdonság és két konstruktorhívás) a `Manager` objektum létrehozásakor.

A származott osztály optimalizált megalkotásában segít, ha úgy implementáljuk az leszármazott osztály konstruktorait, hogy kizárólagosan a megfelelő egyedi ősosztály konstruktorát hívják meg az alapértelmezett helyett. Ily módon csökkenthetjük a származtatott inicializálási tagok meghívását (ez pedig csökkenti a feldolgozási időt). Módosítsuk a `Manager` típus egyedi konstruktorát a `base` kulcsszó használatával:

```
public Manager(string fullName, int age, int empID,
  float currPay, string ssn, int numbOfOpts)
    : base(fullName, age, empID, currPay, ssn)
{
  // A mezőt a Manager osztály definiálja.
  numberOfOptions = numbOfOpts;
}
```

A `base` kulcsszó itt felfüggeszti a konstruktorszignatúrát (igen hasonlóan ahhoz a szintaxishoz, amelyet a konstruktorok láncolására használtunk egyetlen osztályon a `this` kulcsszóval; lásd az 5. fejezetet), amely mindig azt jelöli, hogy egy leszármazott konstruktor adatokat ad át a közvetlen szülőkonstruktornak. Ebben a helyzetben nyíltan meghívjuk az `Employee` osztály által definiált ötparaméteres konstruktort, így felesleges hívásokat spórolunk magunknak a gyermekosztály létrehozásakor. Az egyedi `SalesPerson` konstruktor majdnem ugyanígy néz ki:

```
// Általános szabály, hogy az alosztályok explicit módon hívják
// a megfelelő alaposztály konstruktort.
public SalesPerson(string fullName, int age, int empID,
float currPay, string ssn, int numbOfSales)
: base(fullName, age, empID, currPay, ssn)
{
  // Ez ide tartozik!
  numberOfSales = numbOfSales;
}
```

A `base` kulcsszót bármikor használhatjuk, ha egy származtatott osztály szeretne hozzáférni a szülőosztály által definiált nyilvános vagy védett taghoz. A `this` kulcsszó nem korlátozódik a konstruktorlogikára. (A polimorfizmus vizsgálatakor látni fogjuk a `this` kulcsszó ilyen jellegű használatát.)

Végül: ha egyszer hozzáadunk egy egyedi konstruktort az osztálydefinícióhoz, akkor az alapértelmezett konstruktort észrevétlenül eltávolítjuk. Ezért mindenképpen definiáljuk újra az alapértelmezett konstruktort a `SalesPerson` és `Manager` típusok számára. Például:

```
// Az alapértelmezett konstruktor ismételt definiálása
// a Manager osztályban.
public SalesPerson() {}
```

Családi titok megtartása: a protected kulcsszó

A nyilvános elemekhez bárhonnan közvetlenül hozzá lehet férni, míg a privát elemekhez az azokat definiáló osztályon kívüli objektumokon kívül máshonnan nem. A C# megelőzve sok más modern objektumnyelvet, egy további kulcsszót biztosít a tagok elérhetőségének definiálásához: ez a `protected` (lásd az 5. fejezetet).

Amikor egy alaposztály védett adatokat vagy védett tagokat határoz meg, akkor létrehoz egy olyan elemkészletet, amelyet a leszármazottak bárhonnan közvetlenül elérhetnek. Ha szeretnénk megengedni a `SalesPerson` és a `Manager` gyermekosztályok számára, hogy közvetlenül hozzáférjenek az `Employee` által definiált adatszektorhoz, akkor a következők szerint módosíthatjuk az eredeti `Employee` osztálydefiníciót:

```
// Védett adatok.
partial class Employee
{
  // A származtatott osztályok közvetlenül hozzáférnek az adatokhoz.
  protected string empName;
  protected int empID;
```

```
    protected float currPay;
    protected int empAge;
    protected string empSSN;
    protected static string companyName;
    ...
}
```

Az alaposztályban definiált védett tagoknak az az előnye, hogy a leszármazott típusoknak többé nem közvetve, nyilvános metódusok vagy tulajdonságok használatával kell hozzáférniük az adatokhoz. A lehetséges hátulütője viszont az, hogy amikor egy leszármazott típusnak közvetlen hozzáférése van a szülőosztály belső adataihoz, akkor könnyen előfordulhat, hogy véletlenül kihagy néhány olyan létező üzleti szabályt, amelyek a nyilvános adatokban találhatók. Védett tagok definiálásakor bizalmi szintet hozunk létre a szülő- és a gyermekosztály között, mivel a fordító nem kapja el a típus üzleti szabályainak a megsértését.

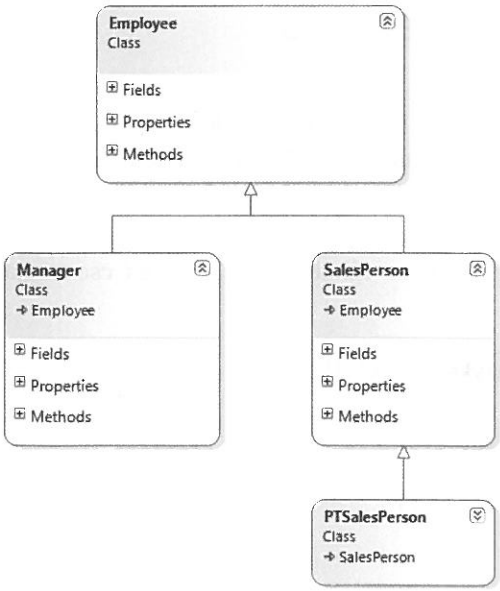
Végül, amíg az objektumhasználóról van szó, addig a védett adatokat privátként kezeljük (mivel a használó „kívül áll" a családon). Ezért a következő példában szereplő hivatkozás szabálytalan:

```
static void Main(string[] args)
{
  // Hiba! A védett adatok nem hozzáférhetők az objektumpéldányból.
  Employee emp = new Employee();
  emp.empName = "Fred";
}
```

Megjegyzés Noha a védett adatmezők megtörhetik az egységbe zárást, a védett metódusok definiálása mégis elég biztonságos (és hasznos). Osztályhierarchia építésekor megszokott dolog olyan metóduskészletet készíteni, amelyet csak a származott típusok használnak.

Lezárt osztály hozzáadása

A *lezárt* osztályt nem bővíthetik ki más osztályok. Ahogy már szó volt róla, ezt a technikát leggyakrabban segédosztály tervezésekor használjuk. Osztályhierarchiák építésekor azonban szükség lehet arra, hogy a származtatási lánc egy bizonyos ágát „lezárjuk", mert nincs értelme tovább növelni a sorok számát. Tegyük fel, hogy hozzáadtunk még egy osztályt (`PTSalesPerson`) a programhoz, amely kibővíti a már létező `SalesPerson` típust. A 6.5. ábra mutatja az aktuális módosítást.



6.5. ábra: *A PTSalesPerson osztály*

A `PTSalesPerson` osztály (természetesen) a részmunkaidős üzletkötőket jelképezi. A paraméterezés kedvéért biztosítani szeretnénk, hogy más fejlesztő ne tudjon származtatni a `PTSalesPersonból`. (Végül is, hogy lehet valaki „részmunkaidősebb" dolgozó, mint egy részmunkaidős?) Hogy megakadályozzuk az osztály kibővítését, használjuk a `sealed` kulcsszót:

```
sealed class PTSalesPerson : SalesPerson
{
  public PTSalesPerson(string fullName, int age, int empID,
    float currPay, string ssn, int numbOfSales)
    :base (fullName, age, empID, currPay, ssn, numbOfSales)
  {
  }
  // Tételezzük fel, hogy más tagok is vannak itt...
}
```

Minthogy a lezárt osztályok nem bővíthetők, vajon fel lehet-e újra használni egy ilyen a kódot? Ha egy olyan új osztályt szeretnénk építeni, amely kihasználja egy lezárt osztály funkcionalitását, az egyetlen lehetőség az, ha a klasszikus származtatás helyett a tartalmazás/delegálás modellt alkalmazzuk (vagyis a „van egy" kapcsolatot).

A tartalmazás és a delegálás

A kód újrafelhasználására tehát két lehetőség van. A klasszikus „az egy" kapcsolatot már feltérképeztük. Mielőtt megvizsgálnánk az OOP harmadik alappillérét, a polimorfizmust, nézzük még meg a „van egy" kapcsolatot (más néven: *tartalmazás/delegálás modell* vagy *aggregáció*). Tegyük fel, hogy létrehoztunk egy olyan új osztályt, amely az alkalmazott juttatási csomagját modellezi:

```
// A típus tartalmazott osztályként működik.
class BenefitPackage
{
  // Egyéb tagok nyugdíjpénztári terveket, egészségpénztári
  // juttatásokat stb. képviselnek.
  public double ComputePayDeduction()
  {
    return 125.0;
  }
}
```

Egyértelműen furcsa lenne, ha „az egy" kapcsolatot hoznánk létre a `BenefitPackage` osztály és az alkalmazott típusok között. (Az `Employee` „az egy" `BenefitPackage`? Nyilván nem.) Nyilvánvaló azonban, hogy valamifajta kapcsolat létrehozható a két osztály között. Röviden: azt szeretnénk kifejezni, hogy minden alkalmazottnak „van egy" juttatási csomagja. Ehhez a következőképpen módosítsuk az `Employee` osztály definícióját:

```
// Az alkalmazottak juttatási csomaggal rendelkeznek.
partial class Employee
{
  // Az osztály BenefitPackage objektumot tartalmaz.
  protected BenefitPackage empBenefits = new BenefitPackage();
  ...
}
```

Ezen a ponton sikeresen tartalmaztattunk egy másik objektumot. Megbízásra van azonban szükség ahhoz, hogy a tartalmazott objektum funkcionalitását feltárjuk a külvilág felé. A *delegálás* egyszerűen olyan tagok hozzáadása a tartalmazó osztályhoz, amelyek kihasználják a tartalmazott objektum funkcionalitását. Módosíthatjuk például úgy az `Employee` osztályt, hogy az feltárja a tartalmazott `empBenefits` objektumot egy egyedi tulajdonsággal, valamint belül használja annak funkcionalitását az új `GetBenefitCost()` metódusban:

```
public partial class Employee
{
  // Az osztály BenefitPackage objektumot tartalmaz.
  protected BenefitPackage empBenefits = new BenefitPackage();

  // Az objektum bizonyos juttatási viselkedésének feltárása.
  public double GetBenefitCost()
  { return empBenefits.ComputePayDeduction(); }

  // Az objektum feltárása egyedi tulajdonság segítségével.
  public BenefitPackage Benefits
  {
    get { return empBenefits; }
    set { empBenefits = value; }
  }
  ...
}
```

Az alábbi módosított `Main()` metódusban látható, hogyan férhetünk hozzá az `Employee` típusban definiált belső `BenefitsPackage` típusú változóhoz:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");
  Manager chucky = new Manager("Chucky", 50, 92, 100000,
    "333-23-2322", 9000);
  double cost = chucky.GetBenefitCost();
  Console.ReadLine();
}
```

A beágyazott típus definiálása

A beágyazott típusok a korábban megvizsgált „van egy" kapcsolat fordítottjai. A C#-ban (valamint más .NET-nyelvekben) meg lehet határozni egy típust (`enum`, `class`, `interface`, `struct` vagy `delegate`) közvetlenül egy osztály vagy struktúra hatókörén belül. Ilyenkor a beágyazott (vagy „belső") típust a beágyazó (vagy „külső") osztály tagjának tekintik, és a futtatórendszer számára a többi más taghoz (mezők, tulajdonságok, metódusok, események stb.) hasonlóan kezelhető. A típus beágyazásához használt szintaxis meglehetősen egyszerű:

```
public class OuterClass
{
  // A nyilvános beágyazott típust bárki alkalmazhatja.
  public class PublicInnerClass {}
```

```
    // A privát beágyazott típust csak a
    // tartalmazó osztály tagjai alkalmazhatják.
    private class PrivateInnerClass {}
}
```

Noha a szintaxis világos, a használata korántsem nyilvánvaló. A technika megértéséhez vegyük sorra a beágyazott típus alábbi jellegzetességeit:

- A beágyazott típusok lehetővé teszik, hogy teljes vezérlést nyerjünk a belső típus hozzáférési szintje fölött, mivel ezek privátként is deklarálhatók (a nem beágyazott osztályok nem deklarálhatók a `private` kulcsszóval).
- Mivel a beágyazott típus az őt tartalmazó osztály tagja, hozzáférhet az osztály privát tagjaihoz.
- A beágyazott típus gyakran csak a külső osztály segítőjeként funkcionál, és nem a külső használatra szánjuk.

Amikor egy típus beágyaz egy másik osztálytípust, akkor ugyanúgy hozhatunk létre ugyanolyan típusú tagváltozókat, mint ahogy bármilyen más típusból. Ha egy tartalmazott típust az őt tartalmazó típuson kívülről szeretnénk használni, akkor ugyanazzal a láthatósággal kell ellátni, mint a tartalmazó típust. Nézzük meg a következő kódot:

```
static void Main(string[] args)
{
  // A nyilvános belső osztály létrehozása és használata. OK!
  OuterClass.PublicInnerClass inner;
  inner = new OuterClass.PublicInnerClass();

  // Fordítási hiba! A privát osztály nem hozzáférhető.
  OuterClass.PrivateInnerClass inner2;
  inner2 = new OuterClass.PrivateInnerClass();
}
```

Hogy felhasználhassuk ezt a koncepciót az alkalmazottakkal kapcsolatos példánkon, tegyük fel, hogy közvetlenül az `Employee` osztálytípusba ágyaztuk be a `BenefitPackage` típust:

```
partial class Employee
{
  public class BenefitPackage
  {
  // Egyéb tagok nyugdíjpénztári terveket, egészségpénztári
  // juttatásokat stb. képviselnek.
```

```
    public double ComputePayDeduction()
    {
      return 125.0;
    }
  }
  ...
}
```

A beágyazási folyamat olyan „mély" lehet, amennyire csak szeretnénk. Tételezzük fel például, hogy szeretnénk létrehozni egy felsorolást (`BenefitPackageLevel` néven), amely dokumentálja az alkalmazott által választható különböző juttatási szinteket. Hogy programozottan kényszerítsünk ki szoros kapcsolatot az `Employees`, a `BenefitPackage` és a `BenefitPackageLevel` között, az alábbiak szerint ágyazhatjuk be a felsorolást:

```
// Az Employee beágyazza a BenefitPackage osztályt.
public partial class Employee
{
  // A BenefitPackage beágyazza a BenefitPackageLevel osztályt.
  public class BenefitPackage
  {
    public enum BenefitPackageLevel
    {
      Standard, Gold, Platinum
    }
    public double ComputePayDeduction()
    {
      return 125.0;
    }
  }
  ...
}
```

A beágyazási kapcsolatokra figyelve nézzük meg, hogyan kell használni ezt a felsorolást:

```
static void Main(string[] args)
{
  ...
  // Saját juttatási szint meghatározása.
  Employee.BenefitPackage.BenefitPackageLevel myBenefitLevel =
    Employee.BenefitPackage.BenefitPackageLevel.Platinum;
  Console.ReadLine()
}
```

Feltártunk tehát az eddigiekben néhány olyan kulcsszót (és koncepciót), amelyek lehetővé teszik kapcsolódó típusok hierarchiájának felépítését a klasszikus származtatás, a tartalmazás és a beágyazott típusok révén. A részletek

még nem szükségszerűen világosak, ám még számos további hierarchiát fogunk építeni. Most azonban vizsgáljuk meg az objektumorientált programozás harmadik alappillérét: a polimorfizmust.

A harmadik pillér: a polimorfizmus támogatása

Az `Employee` alaposztály egy `GiveBonus()` nevű metódust határozott meg, amelyet eredetileg így valósítottunk meg:

```
public partial class Employee
{
  public void GiveBonus(float amount)
  {
    currPay += amount;
  }
  ...
}
```

Minthogy ezt a metódust a `public` kulcsszóval határoztuk meg, adhatunk jutalmakat az üzletkötőknek és a menedzsereknek (valamint a részmunkaidős üzletkötőknek):

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");

  // Minden alkalmazott kapjon bónuszt?
  Manager chucky = new Manager("Chucky", 50, 92, 100000,
    "333-23-2322", 9000);
  chucky.GiveBonus(300);
  chucky.DisplayStats();
  Console.WriteLine();

  SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000,
    "932-32-3232", 31);
  fran.GiveBonus(200);
  fran.DisplayStats();
  Console.ReadLine();
}
```

A jelenlegi tervezéssel az a probléma, hogy a nyilvánosan származtatott `GiveBonus()` metódus azonosan működik minden alosztálynál. Ideális esetben az üzletkötők, illetve a részmunkaidős üzletkötők bónuszának figyelembe kell vennie az eladások számát. A menedzserek esetleg kaphatnak további részvényopciókat a pénzbeli jutalom mellett. Ezen a ponton szembesülnünk kell a következő kérdéssel: Hogyan reagálhatnak a kapcsolódó típusok különbözőképpen ugyanarra a kérésre?

A virtual és override kulcsszavak

A polimorfizmus módot ad egy alosztálynak, hogy elkészítse az ősosztálya által definiált metódus saját verzióját, méghozzá a *metódus-felüldefiniálás* elnevezésű folyamattal. A jelenlegi tervezés módosításához meg kell ismerkednünk a `virtual` és az `override` kulcsszavak jelentésével. Ha az alaposztály meg akar határozni egy olyan metódust, amelyet egy alosztály felüldefiniál*hat* (ám nem szükségszerűen), akkor a metódust a `virtual` kulcsszóval kell megjelölni:

```
partial class Employee
{
  // A metódust egy származtatott osztály "felülírhatja".
  public virtual void GiveBonus(float amount)
  {
    currPay += amount;
  }
  ...
}
```

Megjegyzés A `virtual` kulcsszóval megjelölt metódusok neve (értelemszerűen) *virtuális metódus.*

Egy alosztály az `override` kulcsszóval módosítja egy virtuális metódus megvalósításának részleteit. A `SalesPerson` és Manager például a következők szerint bírálhatják felül a `GiveBonus()` metódust (tegyük fel, hogy a `PTSalesPerson` nem fogja felüldefiniálni a `GiveBonus()`-t, hanem egyszerűen örökli a `SalesPerson` által definiált verziót):

```
class SalesPerson : Employee
{
  ...
  // A kereskedő bónuszát befolyásolja a megkötött üzletek száma.
  public override void GiveBonus(float amount)
  {
    int salesBonus = 0;
```

```
    if (numberOfSales >= 0 && numberOfSales <= 100)
    { salesBonus = 10; }
    else
    {
      if (numberOfSales >= 101 && numberOfSales <= 200)
      {
        salesBonus = 15;
      }
      else
      { salesBonus = 20; }
    }
    base.GiveBonus(amount * salesBonus);
  }
}

class Manager : Employee
{
  ...
  public override void GiveBonus(float amount)
  {
    base.GiveBonus(amount);
    Random r = new Random();
    numberOfOptions += r.Next(500);
  }
}
```

Az egyes felüldefiniált metódusok szabadon használhatják az alapértelmezett viselkedést a `base` kulcsszó segítségével. Ily módon nem kell ismét teljes egészében megvalósítani a `GiveBonus()` mögött álló logikát, hanem felhasználhatjuk (és kibővíthetjük) a szülőosztály alapértelmezett viselkedését.

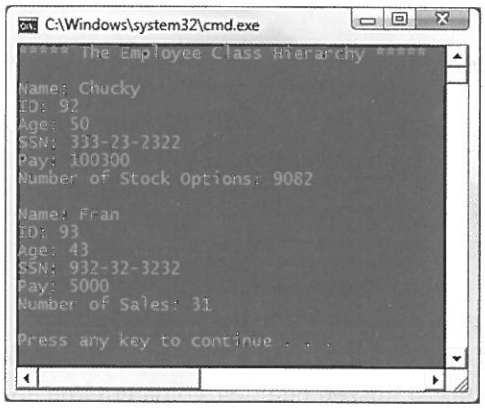
Tételezzük fel azt is, hogy az `Employee` osztály jelenlegi `DisplayStats()` metódusát virtuálisan deklaráltuk. Ezért minden alosztály felülbírálhatja ezt a metódust az eladások számának megjelenítéséért (az üzletkötők számára) és az aktuális részvényopciókat (a menedzserek számára). Figyeljük meg például a `DisplayStats()` metódus `Manager` verzióját (a `SalesPerson` osztály hasonló módon valósítja meg a `DisplayStats()` metódust):

```
public override void DisplayStats()
{
  base.DisplayStats();
  Console.WriteLine("Number of Stock Options: {0}",
    numberOfOptions);
}
```

Ha minden alosztály értelmezheti, hogy ezek a virtuális metódusok mit jelentenek a számára, az objektumpéldányok függetlenebb entitásként viselkednek:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The Employee Class Hierarchy *****\n");

  // Egy jobb bónuszrendszer!
  Manager chucky = new Manager("Chucky", 50, 92, 100000,
    "333-23-  2322", 9000);
  chucky.GiveBonus(300);
  chucky.DisplayStats();
  Console.WriteLine();

  SalesPerson fran = new SalesPerson("Fran", 43, 93, 3000,
    "932-32-3232", 31);
  fran.GiveBonus(200);
  fran.DisplayStats();
  Console.ReadLine();
}
```

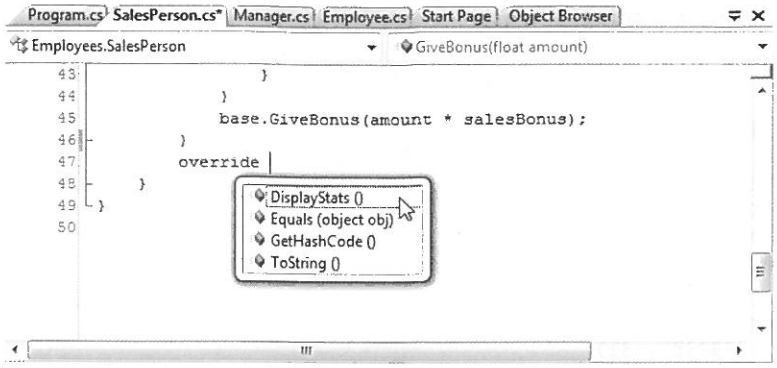
A 6.6. ábra az eddigi alkalmazásunk lehetséges tesztfutását mutatja.



6.6. ábra: *A jelenlegi Employee alkalmazás kimenete*

Virtuális tagok felülbírálása a Visual Studio 2008 segítségével

Tagok felülbírálásakor újra meg kell hívnunk minden egyes paraméter típusát – nem is beszélve a metódusnévről és a paraméterátadási konvenciókról (`ref`, `params` stb.). A Visual Studio 2008 rendelkezik egy olyan hasznos szolgáltatással, amelyet a virtuális tag felülbírálására használhatunk. Ha egy osztálytípus hatókörén belül beírjuk az `override` szót, akkor az IntelliSense automatikusan megjeleníti a szülőosztályban megadott felülbírálható tagok listáját (lásd 6.7. ábra).



6.7. ábra: *A felülbírálható metódusok gyors megtekintése a Visual Studio 2008-ban*

Ha kiválasztunk egy tagot, és megnyomjuk az Enter billentyűt, akkor az IDE automatikusan elkészíti a metódustörzset. Emellett kapunk egy kódutasítást is, amely meghívja a virtuális tag szülőosztályának verzióját (ha nincs rá szükségünk, nyugodtan töröljük ezt a sort):

```
public override void DisplayStats()
{
  base.DisplayStats();
}
```

A virtuális tagok lezárása

A `sealed` kulcsszót azért lehet egy osztályra alkalmazni, hogy megakadályozzuk, hogy a származtatás révén más típusok kibővítsék a viselkedését. Egyik példánkban lezártuk a `PTSalesPerson` típust, hiszen nem lett volna értelme annak, ha más fejlesztők tovább bővítették volna ezt a származtatási ágat.

Előfordulhat, hogy nem egy teljes osztályt akarunk lezárni, csak meg akarjuk akadályozni, hogy származott típusok felüldefiniáljanak bizonyos virtuális metódusokat. Nem akarjuk például, hogy a részmunkaidős üzletkötők egyéni jutalmakat kapjanak. Annak megakadályozására, hogy a `PTSalesPerson` osztály felülbírálja a virtuális `GiveBonus()` metódust, ez utóbbit a következő módon zárhatjuk le hatékonyan a `SalesPerson` osztályban:

```
// A SalesPerson osztály sealed GiveBonus() metódust alkalmaz!
class SalesPerson : Employee
{
  ...
```

```
  public override sealed void GiveBonus(float amount)
  {
    ...
  }
}
```

Ekkor a `SalesPerson` osztály felüldefiniálta az `Employee` osztályban definiált virtuális `GiveBonus()` metódust, és lezártként jelölte meg. Ezért, ha megpróbáljuk felüldefiniálni ezt a metódust a `PTSalesPerson` osztályban:

```
sealed class PTSalesPerson : SalesPerson
{
  public PTSalesPerson(string fullName, int age, int empID,
  float currPay, string ssn, int numbOfSales)
  :base (fullName, age, empID, currPay, ssn, numbOfSales)
  {
  }

  // Nincs bónusz! Hiba!
  public override void GiveBonus(float amount)
  {
    // Ez a metódus nem módosítható tovább.
  }
}
```

akkor fordítási idejű hibákat kapunk.

Az absztrakt osztályok

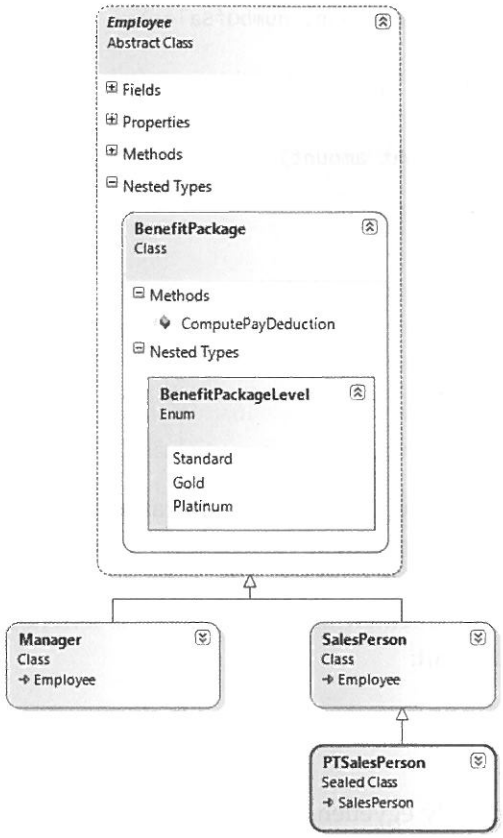
Az `Employee` alaposztály jelenleg védett tagváltozókat szolgáltat a leszármazottainak, valamint két olyan virtuális metódust (`GiveBonus()` és `DisplayStats()`), amelyeket egy adott leszármazott felüldefiniálhat. Ennek a tervezésnek van egy meglehetősen furcsa melléktermék: közvetlenül is létrehozhatjuk az `Employee` alaposztály példányait:

```
// Ez pontosan mit jelent?
Employee X = new Employee();
```

Ebben a példában az `Employee` alaposztály egyetlen igazi célja az, hogy gyakori tagokat határozzon meg az alosztályok számára. Azt nyilván nem szeretnénk, hogy bárki létrehozhassa ennek az osztálynak a közvetlen példányait; az `Employee` típus már önmagában is túlságosan általános koncepció. Az „Alkalmazott vagyok" kijelentésre magától értetődően következik a kérdés: „Milyen*fajta* alkalmazott?" (Tanácsadó, edző, titkár, lektor, fehér házi gyakornok stb.?)

Minthogy sok alaposztály meglehetősen tágan értelmezhető, sokkal célszerűbb úgy tervezni a példánkat, hogy megakadályozzuk egy új `Employee` objektum közvetlen létrehozását a kódban. A C#-ban ezt programozottan kényszeríthetjük ki az `abstract` kulcsszó használatával, amellyel egy *absztrakt ősosztályt* hozunk létre:

```
// Az Employee osztály legyen absztrakt
// így közvetlenül nem lehet létrehozni az újabb példányait.
abstract partial class Employee
{
...
}
```



6.8. ábra. *Az Employee-hierarchia*

Ha most megpróbáljuk létrehozni az `Employee` osztály egy példányát, akkor fordítási idejű hibát kapunk:

```
// Hiba! Nem lehet létrehozni absztrakt osztályt!
Employee X = new Employee();
```

Létrehoztunk tehát egy viszonylag érdekes dolgozói hierarchiát. Később, a C# típusmódosítási szabályainak vizsgálatakor még egy funkcionalitást hozzáadunk ehhez az alkalmazáshoz. A 6.8. ábra a jelenlegi típusunk tervezésének a magját mutatja be.

Forráskód Az `Employees` projekt megtalálható a 6. fejezet alkönyvtárában.

Polimorfikus interfész építése

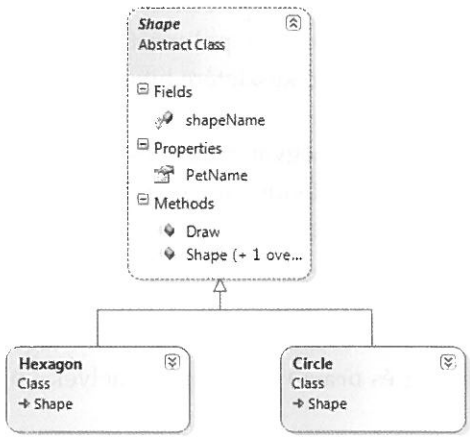
Amikor egy osztályt absztrakt ősosztályként határozunk meg (az `abstract` kulcsszóval), akkor ez az ősosztály bármennyi *absztrakt tagot* definiálhat. Az absztrakt tagokat bármikor használhatjuk, ha olyan tagokat kívánunk készíteni, amelyek *nem* biztosítanak alapértelmezett megvalósítást. Így minden leszármazottra egy *polimorf interfészt* kényszerítünk, rájuk hagyva azt a feladatot, hogy megoldják az absztrakt metódus mögött álló részletek biztosítását.

Egyszerűbben fogalmazva: egy absztrakt ősosztály polimorf interfésze csak a saját virtuális és absztrakt metódusokból álló készletére hivatkozik. Az OOP-nak ez a szolgáltatása teszi lehetővé a kiterjeszthető és rugalmas szoftveralkalmazások építését. Ennek illusztrálására megvalósítjuk (és kicsit módosítjuk) az 5. fejezetben az alappillérek kapcsán röviden megvizsgált alakhierarchiát. Kiindulásként hozzunk létre a C#-ban egy új parancssorialkalmazás-projektet `Shapes` névvel.

A 6.9. ábrán látható, hogy a `Hexagon` és a `Circle` típusok kibővítették a `Shape` ősosztályt. A többi ősosztályhoz hasonlóan a `Shapes` is számos olyan tagot definiál (jelen esetben `PetName` tulajdonság és `Draw()` metódus), amelyek közösek minden leszármazott számára.

Mint az `Employee` hierarchiájánál, itt is meg kell tiltanunk az objektumhasználónak, hogy közvetlenül létrehozhassa a `Shape` egy példányát a túl absztrakt koncepció miatt. A `Shape` típus közvetlen létrehozásának megakadályozására absztrakt osztályként kell definiálnunk a példányt. Továbbá mivel azt szeretnénk, hogy a leszármazott típusok egyénileg válaszoljanak a `Draw()` metódusra, jelöljük meg ezt virtuálisként, és határozzunk meg egy alapértelmezett megvalósítást:

```
// A hierarchia absztrakt alaposztálya.
abstract class Shape
{
  protected string shapeName;

  public Shape()
  { shapeName = "NoName"; }

  public Shape(string s)
  { shapeName = s; }

  // Egyetlen virtuális metódus.
  public virtual void Draw()
  {
    Console.WriteLine("Inside Shape.Draw()");
  }

  public string PetName
  {
    get { return shapeName; }
    set { shapeName = value; }
  }
}
```



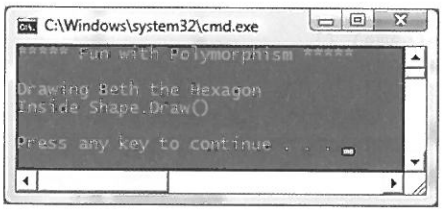
6.9. ábra. *A Shape-hierarchia*

A virtuális `Draw()` metódus biztosít egy alapértelmezett implementációt, ez az implementáció pedig egyszerűen kiír egy olyan üzenetet, amellyel meghívjuk a `Draw()` metódust a `Shape` ősosztályon belül. A `virtual` kulcsszóval megjelölt metódus alapértelmezett implementációja olyan, hogy minden leszármazott típus automatikusan örökli. Ha egy gyermekosztály ilyet választ, akkor *lehet*, hogy felüldefiniálja a metódust, de ez nem szükségszerű. Ennek tudatában nézzük meg a `Circle` és a `Hexagon` típusok alábbi megvalósítását:

```
// A Circle NEM bírálja felül a Draw() metódust.
class Circle : Shape
{
  public Circle() {}
  public Circle(string name) : base(name){}
}

// A Hexagon FELÜLBÍRÁLJA a Draw() metódust.
class Hexagon : Shape
{
  public Hexagon() {}
  public Hexagon(string name) : base(name){}
  public override void Draw()
  {
    Console.WriteLine("Drawing {0} the Hexagon", shapeName);
  }
}
```

Az absztrakt metódusok hasznát egyértelműen mutatja, hogy a származtatott osztályok soha nem kötelesek felülbírálni a virtuális metódusokat (mint most a `Circle` esetében). Így, ha létrehozzuk a `Hexagon` és a `Circle` típusok egy-egy példányát, akkor látható, hogy a `Hexagon` megérti, hogyan „rajzolja ki" önmagát helyesen (vagy legalábbis kiír egy megfelelő üzenetet a konzolra), a `Circle` azonban nem kicsit zavaros (lásd a 6.10. ábra kimenetét):

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

  Hexagon hex = new Hexagon("Beth");
  hex.Draw();

  Circle cir = new Circle("Cindy");
  // Az alaposztály-implementáció hívása!
  cir.Draw();
  Console.ReadLine();
}
```



6.10. ábra: *Valami nincs rendben*

Egyértelműen kiderül, hogy ez a hierarchia nem túl intelligens tervezésű. Hogy rákényszerítsük az összes gyerekosztályra a `Draw()` metódus felüldefiniálását, ez utóbbit a `Shape` osztály absztrakt metódusaként is definiálhatjuk, ez pedig lényegében azt jelenti, hogy nincs semmilyen alapértelmezett megvalósításunk. A C#-ban az `abstract` kulcsszóval jelölhetünk meg az egyes metódusokat absztraktként. Az absztrakt tagok *semmilyen* implementációt nem biztosítanak:

```
// A gyermekosztályok kényszerítése a renderelés meghatározására.
public abstract class Shape
{
  public abstract void Draw();
  ...
}
```

Megjegyzés Absztrakt metódusokat csak absztrakt osztályokban lehet definiálni. Ha máshol próbáljuk meg, akkor fordítási hibát kapunk.

Az absztraktként megjelölt metódusok egyfajta megállapodást rögzítenek. Egyszerűen csak definiálják a nevet, a visszatérési értéket (ha van) és a paraméterkészletet (ha szükséges). A példánkban az absztrakt `Shape` osztály informálja a származtatott típusokat: „Van egy `Draw()` nevű metódus, amely nem vesz fel paramétereket. Ha származtatjátok tőlem, ti találjátok ki a részleteket."

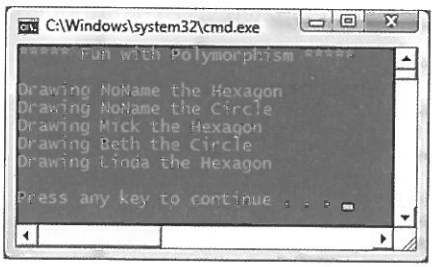
Ezért felül kell bírálnunk a `Draw()` metódust a `Circle` osztályban. Ha nem tesszük meg, akkor a `Circle` is nem létrehozható absztrakt típusnak minősül, amelyet az `abstract` kulcsszó kísér (és ez nyilvánvalóan nem túlságosan hasznos ebben a példában). A kódmódosítás a következő:

```
// Ha nem valósítottuk meg az absztrakt Draw() metódust,
// a Circle is absztrakt lesz,
// és absztrakt osztályként kell megjelölni!
class Circle : Shape
{
  public Circle() {}
  public Circle(string name) : base(name) {}
  public override void Draw()
  {
    Console.WriteLine("Drawing {0} the Circle", shapeName);
  }
}
```

Innen kezdve feltételezhetjük, hogy minden, ami a `Shape` osztályból származik, rendelkezik a saját, egyedi `Draw()` metódus verziójával. A polimorfizmus bemutatásához - a teljesség igényével - nézzük meg az alábbi forráskódot:

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

  // Shape-kompatibilis objektumok tömbjének készítése.
  Shape[] myShapes = {new Hexagon(), new Circle(),
    new Hexagon("Mick"), new Circle("Beth"),
    new Hexagon("Linda")};

  // Az elemek ciklikus feldolgozása,
  // együttműködés a polimorfikus interfésszel.
  foreach (Shape s in myShapes)
  {
    s.Draw();
  }
  Console.ReadLine();
}
```

A 6.11. ábra mutatja a kimenetet.



6.11. ábra: *A polimorfizmus működés közben*

A fenti `Main()` metódus illusztrálja a polimorfizmus jelenségét. Noha *közvetlenül* nem lehet példányosítani egy absztrakt ősosztályt (a `Shape`-et), de egy absztrakt ős- (`Shape`) típusú változóban bármely leszármaztatott osztálypéldányra tárolhatunk referenciát. A `Shapes` egy tömbjét létrehozva, ez a tömb tartalmazhat bármilyen objektumot, amely a `Shape` ősosztályból származik (ha megpróbálunk `Shape`-inkompatibilis objektumot helyezni a tömbbe, akkor fordítási hibát kapunk).

Minthogy a `myShapes` tömb összes eleme a `Shape` osztályból származik, mindegyik támogatja ugyanazt a polimorf interfészt (vagyis mindegyik rendelkezik egy `Draw()` metódussal). Ahogy iteráljuk a `Shape` referenciák tömbjét, a futásidő alatt meghatározódik a referencia mögött lévő tényleges típus. A `Draw()` metódus megfelelő verziója ezen a ponton aktivizálódik.

Ez a technika az aktuális hierarchia biztonságos kibővítését is leegyszerűsíti. Tegyük fel, hogy például további öt osztályt származtattunk le az absztrakt `Shape` ősosztályból (`Triangle`, `Square` stb.). A polimorf interfésznek köszönhetően a `for` ciklusban lévő forráskódot nem kell módosítanunk, hiszen a fordító kikényszeríti, hogy csak a `Shape`-kompatibilis típusok kerüljenek a `myShapes` tömbbe.

Tagok eltakarása

A C# rendelkezik a felüldefiniálási metódus logikai ellentétével: az *eltakarással*. Ha egy leszármaztatott osztály olyan tagot határoz meg, amely megegyezik az alaposztályban definiált tagok egyikével, akkor a származtatott osztály *eltakarja* a szülő verzióját. A gyakorlatban ennek előfordulási lehetősége akkor a legnagyobb, ha egy olyan osztályból származtatunk le, amelyet nem mi hoztunk létre (hanem például egy mások által fejlesztett .NET-komponens valamelyik osztályából származtatunk).

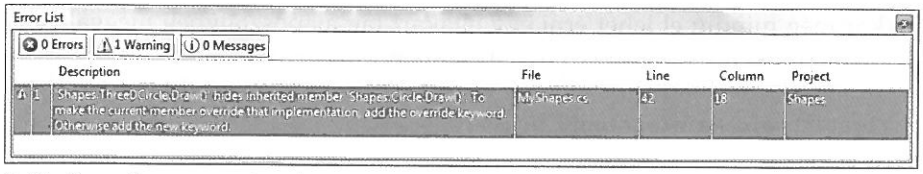
A példa kedvéért tételezzük fel, hogy az egyik munkatársunktól kaptunk egy `ThreeDCircle` nevű osztályt, amely egy olyan `Draw()` nevű metódust definiál, amely nem vesz át paramétereket:

```
class ThreeDCircle
{
  public void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

A `ThreeDCircle` „az egy" `Circle`, ezért a már meglévő `Circle` típusból származtatjuk:

```
class ThreeDCircle : Circle
{
  public void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

Ha újrafordítottuk, egy figyelmeztetést látunk a Visual Studio 2008 hibaablakában (lásd 6.12. ábra).



6.12. ábra: *Éppen most takartuk el a szülőosztály egy tagját*

A problémát kétféleképpen oldhatjuk meg. Egyszerűen módosíthatjuk a `Draw()` metódus szülőosztálybeli verzióját az `override` kulcsszó használatával (ahogy a fordító javasolja). Ekkor a `ThreeDCircle` típus a kívánalmaknak megfelelően képes kibővíteni a szülő alapértelmezett viselkedését. Ha azonban nincs hozzáférésünk az alaposztályt definiáló forráskódhoz (ez harmadik féltől származó könyvtáraknál fordulhat elő), akkor nem tudjuk a `Draw()` metódust virtuális tagra módosítani, ugyanis nem érhetjük el a kódfájlt.

Alternatívaként használhatjuk a `new` kulcsszót a származtatott típus (jelen példában a `ThreeDCircle`) `Draw()` tagján. Ezzel deklaráljuk, hogy a származtatott típus implementációja szándékosan úgy lett megtervezve, hogy elrejtse a szülő verzióját (ez tehát akkor hasznos, ha a külső .NET-komponens ütközik a mi szoftverünkkel).

```
// Az osztály kibővíti a Circle osztályt, és elrejti az örökölt
// Draw() metódust.
class ThreeDCircle : Circle
{
  // A szülő Draw() implementációk elrejtése.
  public new void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

A `new` kulcsszót alkalmazhatjuk az alaposztálytól öröklődő bármilyen tagtípusra (mező, konstans, statikus tag, tulajdonság stb.). Tegyük fel, hogy a `ThreeDCircle` el akarja takarni az örökölt `shapeName` mezőt:

```
// Az osztály kibővíti a Circle osztályt, és elrejti az örökölt
// Draw() metódust.
class ThreeDCircle : Circle
{
  // A szülő shapeName mezők elrejtése.
  protected new string shapeName;
  // A szülő Draw() implementációk elrejtése.
  public new void Draw()
  {
    Console.WriteLine("Drawing a 3D Circle");
  }
}
```

Ekkor még mindig el lehet érni egy eltakart tag alaposztálybeli megvalósítását explicit kaszt használatával (lásd a következő részt). Például:

```
static void Main(string[] args)
{
  ...
  // A ThreeDCircle Draw() metódusának hívása.
  ThreeDCircle o = new ThreeDCircle();
  o.Draw();

  // A szülő Draw() metódusának hívása!
  ((Circle)o).Draw();
  Console.ReadLine();
}
```

Forráskód A Shapes projekt megtalálható a 6. fejezet alkönyvtárában.

Az ősosztály, a leszármazott osztály és a típusmódosítási szabályok

A következőkben az osztálytípus kasztolásának műveleteit vizsgáljuk meg. Ehhez ismét térjünk vissza a korábban létrehozott `Employees` hierarchiájához. A .NET-platform alatt a rendszer legfelső ősosztálya a `System.Object`. Ezért minden „az egy" objektumként jelenik meg, és eszerint is kezelendő. Ezért bármelyik típus egy példányát legálisan eltárolhatjuk egy objektumváltozón belül:

```
// A Manager "az egy" System.Object objektum.
object frank = new Manager("Frank Zappa", 9, 3000, 40000,
  "111-11-1111", 5);
```

Az `Employees` rendszerben a `Managers`, a `SalesPerson` és a `PTSalesPerson` típusok mind kibővítik az `Employee` osztályt, így ezeknek az objektumoknak bármelyikét eltárolhatjuk az érvényes ősosztály-referenciában. Ezért a következő utasításokat is alkalmazhatjuk:

```
// A Manager "az egy" Employee objektum is.
Employee moonUnit = new Manager("MoonUnit Zappa", 2, 3001, 20000,
  "101-11-1321", 1);

// A PTSalesPerson "az egy" SalesPerson.
SalesPerson jill = new PTSalesPerson("Jill", 834, 3002, 100000,
  "111-12-1119", 90);
```

Az osztálytípusok közti kasztolás első törvénye az, hogy amikor két osztály „az egy" formában kapcsolódik egymáshoz, akkor mindig biztonságos lesz egy származtatott típust egy ősosztály-referencián belül eltárolni. Ezt formálisan *implicit kasztnak* hívják, mivel „egyszerűen csak működik" a származtatás törvényei miatt. Az implicit kaszt hatékony programozási konstrukciókat eredményez. Tegyük fel például, hogy készítettünk egy új metódust az aktuális Program osztályon belül:

```
static void FireThisPerson(Employee emp)
{
  // Eltávolítás az adatbázisból...
  // Az elbocsátott alkalmazott holmijának összegyűjtése...
}
```

Mivel ez a metódus egyetlen, `Employee` típusú paraméterrel rendelkezik, gyakorlatilag átadhatjuk neki az `Employee` osztály bármelyik leszármazottját, közvetlenül megadva az „az egy" kapcsolatot:

```
// A személyzet karcsúsítása.
FireThisPerson(moonUnit); // "moonUnit" egy Employee.
FireThisPerson(jill);     // "jill" egy SalesPerson.
```

Az előző kód lefordul, hiszen a származtatott típust átadhatjuk ősosztálytípusként az implicit kasztolásnak köszönhetően. De mi a helyzet akkor, ha például Frank Zappát (aki jelenleg egy általános `System.Object` referenciában van eltárolva) is ki akarjuk rúgni? Ha a frank objektumot közvetlenül átadjuk a `FireThisPerson()` metódusnak a következők szerint:

```
// Hiba!
object frank = new Manager("Frank Zappa", 9, 3000, 40000,
  "111-11-1111", 5);
FireThisPerson(frank);
```

akkor fordítási hibát kapunk. Látható, hogy az objektumreferencia egy `Employee`-kompatibilis objektumra mutat. Eleget tehetünk tehát a fordító igényeinek, ha egy explicit kasztolást hajtunk végre. A kasztolás második törvénye szerint explicit módon kell a konvertálást végrehajtani a leszármazottra a C#-kasztoperátor használatával. Az előző probléma tehát így kerülhető el:

```
// OK!
FireThisPerson((Manager)frank);
```

Az *as* kulcsszó

Az explicit kasztolás futásidőben történik, nem pedig a fordításkor. Ezért ha mi lennénk a következő C#-forráskód szerzői:

```
// frank-et nem kasztolhatjuk a Hexagon osztályhoz!
Hexagon hex = (Hexagon)frank;
```

akkor futásidejű hibát vagy szakszerűebben *futásidejű kivételt* kapnánk. Amikor explicit kasztolást hajtunk végre, akkor elkaphatjuk az érvénytelen kasztolást a `try` és `catch` kulcsszavak használatával (a strukturált kivitelezés részleteit lásd a 7. fejezetben):

```
// Lehetséges érvénytelen kasztolás ellenőrzése.
try
{
  Hexagon hex = (Hexagon)frank;
}
catch (InvalidCastException ex)
{
  Console.WriteLine(ex.Message);
}
```

Mindez a védekező programozásra nyújt szép példát, a C# azonban biztosítja az `as` kulcsszót, amely a futáskor gyorsan el tudja dönteni, hogy egy adott típus kompatibilis-e a másikkal. Az `as` kulcsszó inkompatibilitás esetén null értékkel tér vissza. Így a kompatibilitás egy nullérték-vizsgálattal eldönthető. Nézzük meg erre a következőt:

```
// A kompatibilitás tesztelése az 'as' kulcsszóval.
Hexagon hex2 = frank as Hexagon;
if (hex2 == null)
  Console.WriteLine("Sorry, frank is not a Hexagon...");
```

Az *is* kulcsszó

Mivel a `FireThisPerson()` metódus arra szolgál, hogy felvegyen bármilyen lehetséges tagot, amely az `Employee` osztályból származik, fel lehet tenni a kérdést, hogy ez a metódus hogyan dönti el, hogy melyik származott típus küldése történt meg. Mivel adva van, hogy a bejövő paraméter `Employee` típusú, hogyan lehet hozzáférni a `SalesPerson` és a `Manager` típusok speciális tagjaihoz?

Az as mellett a C# nyelv ismeri az is kulcsszót is, amely két elem kompatibilitását definiálja. Az as kulcsszóval ellentétben az is kulcsszó false értéket ad vissza nullreferencia helyett, ha a típusok nem kompatibilisek. Nézzük meg a FireThisPerson() metódus következő megvalósítását:

```
static void FireThisPerson(Employee emp)
{
  if (emp is SalesPerson)
  {
    Console.WriteLine("Lost a sales person named {0}", emp.Name);
    Console.WriteLine("{0} made {1} sale(s)...", emp.Name,
      ((SalesPerson)emp).SalesNumber);
    Console.WriteLine();
  }
  if (emp is Manager)
  {
    Console.WriteLine("Lost a suit named {0}", emp.Name);
    Console.WriteLine("{0} had {1} stock options...", emp.Name,
      ((Manager)emp).StockOptions);
    Console.WriteLine();
  }
}
```

Hajtsunk végre egy futásidejű ellenőrzést annak eldöntésére, hogy a bejövő ősosztály-referencia valójában mit mutat a memóriában. Amint meghatároztuk, hogy SalesPerson vagy Manager típust kaptunk-e, végre tudunk hajtani egy explicit kasztolást, hogy hozzáférhessünk az osztály speciális tagjaihoz. Nem szükséges try/catch formába önteni a kasztolási műveleteket, hiszen tudjuk, hogy a kaszt biztonságos. Ha belépünk bármelyik if hatókörbe, a feltétel-ellenőrzésünk adott lesz.

A fő szülőosztály: a System.Object

A .NET-platform fő ősosztálya az Object. Látható, hogy a hierarchiáink (Car, Shape, Employee) alaposztályai soha nem határozzák meg explicit módon a saját szülőosztályukat:

```
// Ki a Car szülőosztálya?
class Car
{...}
```

A .NET-univerzumban minden típus végső soron a `System.Object` alaposztályból származik. Az `Object` osztály határozza meg a keretrendszerben levő összes típus közös tagjainak a készletét. Amikor olyan osztályt építünk, amely nem határozza meg kifejezetten a szülőjét, akkor azt a fordító automatikusan az `Objectből` származtatja. Ha nagyon világosan ki akarjuk fejezni a szándékunkat, akkor határozzuk meg, hogy mely osztályok származnak az `Objectből`:

```
// Explicit származtatás a System.Object osztályból.
class Car : object
{...}
```

Más osztályokhoz hasonlóan a `System.Object` is megadja a tagok készletét. Az alábbi formális C#-definícióban néhány elemet virtuálisan deklaráltunk, s ez eldönti, hogy egy adott tagot felülbírálhat-e egy alosztály, míg mások statikusként lesznek megjelölve (és így osztályszinten lesznek meghívva):

```
public class Object
{
  // Virtuális tagok.
  public virtual bool Equals(object obj);
  protected virtual void Finalize();
  public virtual int GetHashCode();
  public virtual string ToString();

  // Példányszintű, nem virtuális tagok.
  public Type GetType();
  protected object MemberwiseClone();

  // Statikus tagok.
  public static bool Equals(object objA, object objB);
  public static bool ReferenceEquals(object objA, object objB);
}
```

A 6.1. táblázat az egyes metódusok által kínált funkcionalitást összegzi.

Az objektumosztály példánymetódusa	Valós jelentése
`Equals()`	Az alapértelmezés szerint ez a metódus csak akkor ad vissza `true` értéket, ha az összehasonlított elemek pontosan ugyanarra az elemre hivatkoznak a memóriában. Ezért az `Equals()` metódust objektumreferenciák és nem állapotok összehasonlítására használják. Ezt a metódust általában felüldefinálják, hogy akkor adjon vissza `true` értéket, ha az összehasonlított objektumok ugyanazokkal

Az objektumosztály példánymetódusa	Valós jelentése
	a belső állapotértékekkel rendelkeznek (értékalapú szemantika). Ha felüldefiniáljuk az `Equals()` metódust, akkor a `GetHashCode()` metódust is felül kell definiálnunk, ezeket ugyanis belsőleg használják a `Hashtable` típusok, hogy visszakeressék az eltárolt objektumokat.
`GetHashCode()`	Ez a metódus egy egész számot ad vissza, amely egy adott objektumpéldányt azonosít.
`GetType()`	Ez a metódus egy olyan `Type` objektumot ad vissza, amely teljesen leírja a jelenleg hivatkozott objektumot. Röviden: ez egy Runtime Type Identification (RTTI) metódus, amely minden objektum számára elérhető (részletesebben lásd a 16. fejezetben).
`ToString()`	Ez a metódus visszaadja az objektum sztringreprezentációját <névtér>.<típusnév> formátumban (teljesen definiált név). Ezt a metódust egy alosztály felüldefiniálhatja, hogy visszaadja a név/érték párok tokenizált sztringjét a teljesen definiált név helyett az objektum belső állapotának ábrázolására.
`Finalize()`	Ezt a metódust azért kell meghívni, amikor felüldefiniálják, hogy felszabadítsa a lefoglalt memóriát, mielőtt az objektum megsemmisül. (A CLR szemétgyűjtő szolgáltatásáról bővebben a 8. fejezetben lesz szó.)
`MemberwiseClone()`	Ez a metódus visszaadja az aktuális objektum tagról tagra történő másolatát. Ezt gyakran használjuk egy objektum klónozásakor (lásd 9. fejezet).

6.1. táblázat: *A System.Object alaptagjai*

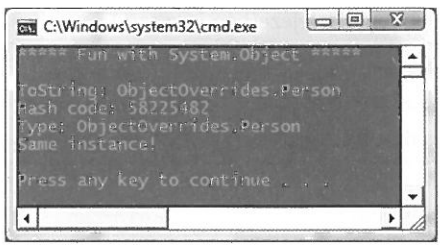
Az `Object` ősosztály néhány alapértelmezett viselkedésének illusztrálásához hozzunk létre egy új C# parancssori alkalmazást `ObjectOverrides` néven. Szúrjunk be egy új C#-osztálytípust, amely a következő üresosztály-definíciót tartalmazza a `Person` nevű típus számára:

```
// Figyelem! A Person kibővíti az Object osztályt.
class Person {}
```

Módosítsuk a `Main()` metódust, hogy meghívja a `System.Object` származtatott tagjait:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with System.Object *****\n");
    Person p1 = new Person();

    // A System.Object örökölt tagjainak alkalmazása.
    Console.WriteLine("ToString: {0}", p1.ToString());
    Console.WriteLine("Hash code: {0}", p1.GetHashCode());
    Console.WriteLine("Type: {0}", p1.GetType());

    // p1 további referenciái.
    Person p2 = p1;
    object o = p2;

    // A referenciák a memóriában ugyanarra az objektumra mutatnak?
    if (o.Equals(p1) && p2.Equals(o))
    {
      Console.WriteLine("Same instance!");
    }
    Console.ReadLine();
  }
}
```

A 6.13. ábra mutatja a kimenetet.



6.13. ábra: *A System.Object származtatott tagjainak az aktivizálása*

A `ToString()` alapértelmezett implementációja visszaadja az aktuális típus (`ObjectOverrides.Person`) teljesen definiált nevét. Az egyedi névterek készítésének vizsgálatakor (15. fejezet) látni fogjuk, hogy minden C#-projekt létrehoz egy „gyökérnévteret", amelynek ugyanaz a neve, mint a projektnek. Itt egy `ObjectOverrides` néven létrehozott projekttel dolgoztunk; így a `Person` típus (valamint a Program osztály) az `ObjectOverrides` névtéren belülre került.