Az `Equals()` metódus alapértelmezett viselkedése annak tesztelése, hogy két változó ugyanarra az objektumra mutat-e a memóriában. Létrehozunk egy új `Person` változót p1 néven. Ekkor egy új `Person` objektum kerül a felügyelt heapre. A p2 szintén `Person` típus. Nem hozunk azonban létre *új* példányt, hanem hozzárendeljük ezt a változót a p1 referenciához. Ennélfogva a p1 és p2 ugyanarra az objektumra mutatnak a memóriában, ugyanúgy, mint az o változó (amely a típusobjektumé, és amelyet ráadásként dobtunk be). Mivel a p1, a p2 és az o ugyanarra a memóriahelyre mutatnak, az egyenlőségi teszt sikeres lesz.

Noha a `System.Object` rögzített viselkedése legtöbbször megfelelő, előfordul, hogy az egyedi típusok felülbírálnak néhány örökölt metódust. Ennek bemutatásához módosítsuk a `Person` osztályt úgy, hogy támogasson néhány állapotadatot, amelyek egy egyén kereszt- és vezetéknevét, valamint a korát jelölik, és amelyeket egy egyedi konstruktor állít be:

```
// Figyelem! A Person kibővíti az Object osztályt.
class Person
{
  // Csak az egyszerűség kedvéért nyilvános. A tulajdonságok és
  // privát adatok alkalmazása tanácsos.
  public string fName;
  public string lName;
  public byte personAge;

  public Person(string firstName, string lastName, byte age)
  {
    fName = firstName;
    lName = lastName;
    personAge = age;
  }
  public Person(){}
}
```

A System.Object.ToString() felüldefiniálása

Sok általunk létrehozott osztálynak (és struktúrának) előnyös lehet, ha felüldefiniáljuk a `ToString()` metódust annak érdekében, hogy visszaadja a típus aktuális állapotának sztringszöveges reprezentációját. Ez (többek között) hibakeresésnél bizonyul nagyon hasznosnak. A sztring konstruálása egyéni ízlés kérdése; ajánlott azonban úgy elvégezni, hogy kettőspontokkal elkülönítjük az egyes név/érték párokat, és az egész sztringet szögletes zárójelbe foglaljuk (a .NET-alaposztálykönyvtárak számos típusa ezt a megközelítést használja). Tekintsük meg Person osztályunk alábbi, felülbírált `ToString()` metódusát:

```
public override string ToString()
{
  string myState;
  myState = string.Format("[First Name: {0}; Last Name: {1};
    Age: {2}]", fName, lName, personAge);
  return myState;
}
```

A `ToString()` metódusnak ez a megvalósítása meglehetősen lényegre törő, ugyanis a `Person` osztály csak három darab állapotadattal rendelkezik. Egy helyes `ToString()` felülbírálhatnak azonban minden, a származási láncban definiált adatot figyelembe kell vennie. Amikor felülbíráljuk egy egyedi alaposztályt kibővítő osztály `ToString()` metódusát, akkor az első feladat megszerezni a `ToString()` értékét a szülőosztálytól a `base` felhasználásával. Amint megszereztük a szülő sztringadatait, csatolhatjuk a származott osztály egyedi információit.

A System.Object.Equals() felüldefiniálása

Definiáljuk felül az `Object.Equals()` viselkedését is, hogy értékalapú szemantikával dolgozzunk. Az `Equals()` csak akkor ad vissza `true` értéket, ha a két összehasonlított objektum ugyanarra az objektumpéldányra hivatkozik a memóriában. A `Person` osztály esetében hasznos lehet úgy implementálni az `Equals()` metódust, hogy akkor adjon vissza `true` értéket, ha a két összehasonlítandó változó ugyanazokat az állapotadatokat tartalmazza (pl. keresztnév, vezetéknév, kor).

Az `Equals()` metódus bejövő paramétere egy generikus `System.Object`. Ezért először győződjünk meg róla, hogy a hívó valóban átadott-e egy `Person` típust, valamint hogy a bejövő paraméter lefoglalt objektum-e.

Ha a hívó átadott egy lefoglalt `Person` típust, az `Equals()` megvalósításának az egyik módja az, hogy mezőről mezőre összehasonlítjuk a bejövő objektum és az aktuális objektum adatait:

```
public override bool Equals(object obj)
{
  if (obj is Person && obj != null)
  {
    Person temp;
    temp = (Person)obj;
    if (temp.fName == this.fName && temp.lName == this.fName
      && temp.personAge == this.personAge)
    {
      return true;
    }
```

```
    else
    {
      return false;
    }
  }
  return false;
}
```

Ekkor összehasonlítjuk a bejövő objektum értékeit a belső értékekkel (figyeljük meg a `this` kulcsszó használatát). Ha a név és a kor is megegyezik, akkor két objektumunk van pontosan ugyanazokkal az állapotadatokkal, ezért a visszaadott érték `true`. Minden más lehetőség `false` értéket eredményez.

Noha ez a megközelítés működőképes, egy egyedi `Equals()` metódus megvalósítása nagyon sok munkát igényel a nemtriviális típusok esetében, amelyek többtucatnyi adatmezőt tartalmazhatnak. Gyakori egyszerűsítés az, hogy kihasználjuk a `ToString()` metódus saját megvalósítását. Ha egy osztály tartalmazza a `ToString()` metódus olyan megfelelő megvalósítását, amely figyelembe veszi a származási lánc összes adatmezőjét, akkor egyszerűen összehasonlíthatjuk az objektum sztringadatait:

```
public override bool Equals(object obj)
{
  // Felesleges az 'obj' kasztolása a Person osztályra,
  // mert minden osztály rendelkezik ToString() metódussal.
  return obj.ToString() == this.ToString();
}
```

A System.Object.GetHashCode() felüldefiniálása

Ha egy osztály felüldefiniálja az `Equals()` metódust, akkor a `GetHashCode()` metódus alapértelmezett megvalósítását is felül kell definiálni. A *hashkód* egy olyan numerikus érték, amely adott állapotúként ábrázol egy objektumot. Ha például létrehozunk két olyan sztringobjektumot, amelyeknek Hello az értéke, akkor ugyanazt a hashkódot kapjuk. Ha azonban az egyik sztringobjektum kisbetűs (hello), akkor különböző hashkódokat kapunk.

Az alapértelmezés szerint a `System.Object.GetHashCode()` az objektum memóriában elfoglalt aktuális helyét használja a hashérték megszerzéséhez. Ha azonban olyan egyedi típust építünk, amelyet `Hashtable` típusként szeretnénk tárolni (a `System.Collections` névtérben), akkor mindig felül kell írnunk ezt a tagot, mivel a `Hashtable` az `Equals()` és a `GetHashCode()` metódusokat belsőleg fogja aktivizálni, hogy vissza tudja keresni a helyes objektumot.

Jóllehet, nem akarjuk a `System.Collections` névtérben található hashtáblába helyezni a `Person` típusunkat, a teljesség kedvéért írjuk felül a `GetHashCode()` metódust. Számos algoritmus használható hashkódok létrehozására; közülük néhány különleges, néhány pedig nem. Legtöbbször úgy is tudunk hashkód-értékeket generálni, hogy kihasználjuk a `System.String GetHashCode()` implementációját.

Minthogy a `String` osztály már rendelkezik egy egyszerű hashelési algoritmussal, amely a `String` karakteradatait használja a hashérték kiszámításához, így, ha azonosítani tudunk az osztályban egy adatmezőrészletet, amely minden példány esetében egyedi (mint a taj szám), akkor egyszerűen hívjuk meg a `GetHashCode()` metódust erre az adatmeződarabra. Ha nem ez a helyzet, ám felüldefiniáltuk a `ToString()` metódust, akkor hívjuk meg a `GetHashCode()` metódust a saját sztringábrázolásunkhoz:

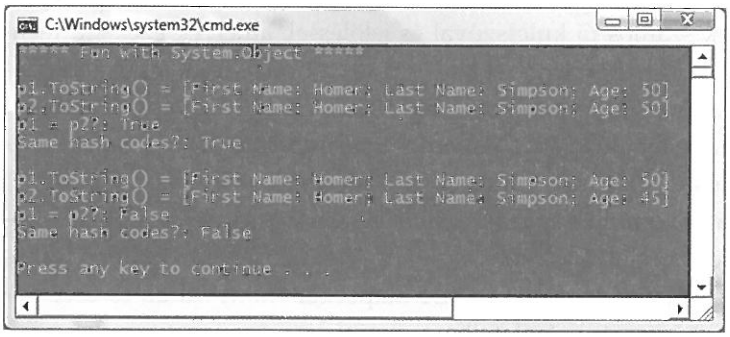
```
// Visszaadja a ToString() érték alapján kiszámított hashértéket.
public override int GetHashCode()
{
  return this.ToString().GetHashCode();
}
```

A módosított Person osztály tesztelése

Miután felülbíráltuk az `Object` virtuális tagjait, frissítsük a `Main()` metódusunkat, hogy leteszteljük a módosításokat (a kimenetet lásd a 6.14. ábrán).

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

  // MEGJEGYZÉS: Az Equals() és a GetHashCode() teszteléséhez
  // ezeknek egyezniük kell.
  Person p1 = new Person("Homer", "Simpson", 50);
  Person p2 = new Person("Homer", "Simpson", 50);

  // Az objektumok szöveges verziójának beolvasása.
  Console.WriteLine("p1.ToString() = {0}", p1.ToString());
  Console.WriteLine("p2.ToString() = {0}", p2.ToString());

  // A felülbírált Equals() tesztelése
  Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));

  // A hashértékek tesztelése.
  Console.WriteLine("Same hash codes?: {0}",
    p1.GetHashCode() == p2.GetHashCode());
  Console.WriteLine();
```

```
    // A p2 korának módosítása, és ismételt tesztelése.
    p2.personAge = 45;
    Console.WriteLine("p1.ToString() = {0}", p1.ToString());
    Console.WriteLine("p2.ToString() = {0}", p2.ToString());
    Console.WriteLine("p1 = p2?: {0}", p1.Equals(p2));
    Console.WriteLine("Same hash codes?: {0}",
      p1.GetHashCode() == p2.GetHashCode());
    Console.ReadLine();
  }
```



6.14. ábra: *Az egyedi Person típusunk*

A System.Object statikus tagjai

Az eddig megvizsgált példányszintű tagok mellett a `System.Object` két (nagyon hasznos) statikus tagot is definiál, amelyek ugyancsak tesztelik az értékalapú vagy referenciaalapú egyenlőséget. Nézzük meg a következő kódot:

```
static void SharedMembersOfObject()
{
  // A System.Object statikus tagjai.
  Person p3 = new Person("Sally", "Jones", 4);
  Person p4 = new Person("Sally", "Jones", 4);
  Console.WriteLine("P3 and P4 have same state: {0}",
    object.Equals(p3, p4));
  Console.WriteLine("P3 and P4 are pointing to same object: {0}",
    object.ReferenceEquals(p3, p4));
}
```

Itt egyszerűen átadunk két objektumot (bármilyen típusút), és lehetővé tehetjük a `System.Object` osztálynak, hogy automatikusan definiálja a részleteket. Ezek a metódusok sokat segíthetnek, ha felüldefiniáltuk az egyenlőséget egy egyedi típus esetében, és mégis szükségünk van még annak gyors meghatározására, hogy a két referencia ugyanarra a pontra mutat-e a memóriában (a statikus `ReferenceEquals()` metódus révén).

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

Összefoglalás

Ez a fejezet részletesen feltárta a származtatás és a polimorfizmus szerepét. Megismerkedtünk számos új kulcsszóval és jelöléssel, amelyek ezeket a technikákat támogatják. A kettőspont például egy adott típus szülőosztályának definiálására szolgál. A szülőtípusok bármennyi virtuális és/vagy absztrakt tagot definiálhatnak, hogy létrehozzanak egy polimorf interfészt. A származtatott típusok az override kulcsszó használatával felüldefinálják az ilyen tagokat.

Számos osztályhierarchia építése mellett azt is megvizsgáltuk, hogy hogyan lehet explicit módon kasztolni az alap- és a származtatott típusok között; végül pedig belemélyedtünk a .NET-alaposztálykönyvtárak fő ősosztályának, a `System.Objectnek` a részleteibe.

HETEDIK FEJEZET

A strukturált kivételkezelés

Ez a fejezet a *strukturált kivételkezelés* segítségével a futási időben keletkezett anomáliák kezelését mutatja be. Nemcsak az ilyen jellegű problémák megoldására szolgáló C#-kulcsszavakkal (`try`, `catch`, `throw`, `finally`) ismerkedhetünk meg, hanem az alkalmazásszintű és a rendszerszintű kivételek közötti különbséggel, valamint a `System.Exception` ősosztály szerepével is. Megtanuljuk továbbá az egyedi kivételek készítését, és megvizsgáljuk a Visual Studio 2008 kivételcentrikus hibakeresésre szolgáló eszközeinek a használatát.

Hibák, programhibák és kivételek

Önbizalom ide vagy oda, tökéletes programozó nincsen. A programozás összetett feladat, és éppen emiatt még a legjobb szoftverek esetében is előfordulnak bizonyos *problémák*. A problémát néha egy „hibás kód" okozza (pl. túlcsordulás egy tömb határain), máskor egy hamis felhasználói bemenet, amelyre nem készítették fel az alkalmazást a fejlesztés során (pl. egy telefonszám-beviteli mezőbe a „Chucky" értéket adjuk meg). Függetlenül attól, hogy mi okozta a problémát, a végeredmény az, hogy az alkalmazás nem a kívánt módon működik. A strukturált kivételkezelés tanulmányozása előtt nézzük meg a három általánosan használt rendellenesség-központú kifejezés definícióját:

- *Programhibák:* A programhibákat egyszerűen a programozó követi el. Tételezzük fel például, hogy nem menedzselt C++-ban programozunk. Ha nem szabadítjuk fel a dinamikusan lefoglalt memóriát (ami memóriaszivárgást eredményez), akkor programhiba lép fel.
- *Felhasználói hibák:* A programhibákkal ellentétben a felhasználói hibákat általában az alkalmazást futtató személy okozza, és nem az, aki létrehozta. Az a végfelhasználó például, aki rossz szöveget ír be egy szövegdobozba, könnyen hibát generálhat, *ha a programozó* nem kezelte ezt a hibás bemenetet a kódban.

- *Kivételek:* A kivételek többnyire futásidejű rendellenességek, amelyeket nehéz - ha nem is lehetetlen - számításba venni az alkalmazás programozása közben. A lehetséges kivételek közé tartozik a nem létező adatbázishoz való csatlakozás, egy hibás fájl megnyitása vagy kapcsolatba lépés egy olyan számítógéppel, amely jelenleg nem érhető el. Ezekben az esetekben a programozó (és a felhasználó) nem igazán tudja kezelni a „kivételes" körülményeket.

A fenti definíciók alapján egyértelmű, hogy a .NET strukturált *kivétel*kezelése olyan technika, amely megfelel a futásidejű *kivételek* kezelésére. Mivel azonban a program- és a felhasználói hibák észrevétlenek tudnak maradni, a CLR gyakran generál olyan kivételeket, amelyek a felmerült problémát azonosítják.

A .NET-alaposztálykönyvtárak számos kivételt definiálnak, ilyen például a `FormatException`, az `IndexOutOfRangeException`, a `FileNotFoundException`, az `ArgumentOutOfRangeException` stb. A .NET terminológiájában a „kivételek" közé tartoznak a programhibák, a hamis felhasználói bemenetek és a futásidejű hibák, még akkor is, ha a programozók az egyes lehetőségeket különálló kérdésként tekintik. Formalizáljuk a strukturált kivételkezelés szerepét, és nézzük meg, hogy miben különbözik a tradicionális hibakezelő technikáktól.

Megjegyzés Hogy a könyvben használt példakódok a lehető legvilágosabbak legyenek, nem fogunk minden lehetséges kivételt megnézni, amelyet egy adott metódus az alaposztálykönyvtáraiban dobhat. A termékszintű projektek esetén természetesen érdemes szabadon használni a fejezetben felsorolt technikákat.

A kivételkezelés szerepe a .NET-ben

A .NET előtt a Windows operációs rendszer hibakezelése a különböző technikák zavaros egyvelege volt. Sok programozó a saját hibakezelési logikáját használta egy adott alkalmazás kontextusában. Egy fejlesztői csapat definiálhat például egy numerikus konstanskészletet, amelynek elemei ismert hibakörülményeket képviselnek, és használhatják ezeket függvény-visszatérési értékekként. Ehhez példaként tekintsük meg az alábbi C-kód-részletet:

```
/* A very C-style error trapping mechanism. */
#define E_FILENOTFOUND 1000

int SomeFunction()
{
  // Valami történik az f(x) függvénnyel,
  // ami a következő visszatérési értéket eredményezi.
  return E_FILENOTFOUND;
}

void Main()
{
  int retVal = SomeFunction();
  if(retVal == E_FILENOTFOUND)
    printf("Cannot find file...");
}
```

A fenti megközelítés nem túl ideális, mivel a konstans `E_FILENOTFOUND` alig több, mint egy numerikus érték, és korántsem hasznos a probléma kezelését illetően. Ideális esetben szeretnénk látni a hiba nevét, továbbá egy leíró üzenetet és egyéb hasznos információkat ezzel a hibakörülménnyel kapcsolatban, egy jól definiált egységben (a strukturált kivételkezelés során pontosan ez történik).

A fejlesztők ad hoc technikái mellett a Windows API több száz hibakódot definiál, amelyek `#define`, `HRESULT` és az egyszerű logikai érték tárolására alkalmazott sok különböző típuson keresztül (`bool`, `BOOL`, `VARIANT_BOOL` stb.) jelennek meg. Emellett több C++ COM fejlesztő (és közvetve sok VB6 COM fejlesztő) használta a standard COM-interfészek (pl. `ISupportErrorInfo`, `IErrorInfo`, `ICreateErrorInfo`) kis készletét, hogy értelmes hibainformációkat tudjanak visszaadni a COM-kliensnek.

Az előbbi technikák nyilvánvaló problémája a szimmetria teljes hiánya. Az egyes megközelítések többé-kevésbé egy adott technológiához, egy adott nyelvhez, sőt esetleg egy adott projekthez tartoznak. Hogy ennek a káosznak véget vessen, a .NET-platform egységes technikát biztosít a futásidejű hibák kiváltása és elkapása érdekében: ez a strukturált kivételkezelés (Structured Exception Handling - SEH).

Így a fejlesztők most már rendelkeznek egy egységes hibakezelési megközelítéssel, amely a .NET-platformot célzó összes nyelv esetén közös lesz. Ezért a C#-programozó hibakezelése szintaktikailag hasonlít a VB.NET, illetve a felügyelt C++-programozóéhoz. További pluszként említhetjük, hogy szerelvényeken és számítógéphatárokon át a kivételek dobására és elkapására használt szintaxis megegyezik.

A .NET-kivételek másik előnye az, hogy a hiba azonosítására nem egy misztikus számértéket használ, hanem egy olyan kivételobjektumot, amely a probléma olvasható leírását tartalmazza ráadásul részletes pillanatfelvételt ad arról a hívási veremről, amely elsődlegesen kiváltotta a kivételt. Továbbá megadhatunk a felhasználónak egy olyan hivatkozást is, amely az aktuális problémát tárgyaló, valamint a programozó által definiált egyedi adatokkal rendelkező, részletes információkkal ellátott URL-re mutat.

A .NET-kivételkezelés

A strukturált kivételkezeléssel járó programozás tartalmaz négy, egymással kölcsönös kapcsolatban álló entitást:

- egy osztálytípust, amely ábrázolja a kivétel részleteit;
- egy tagot, amely *dobja* a kivételosztály egy példányát a hívónak;
- egy kódblokkot a hívó oldalán, amely meghívja a kivételre hajlamos tagot;
- és egy kódblokkot a hívó oldalán, amely feldolgozza (vagy *elkapja*) az előforduló kivételt.

A C# programozási nyelv négy kulcsszót biztosít (`try`, `catch`, `throw` és `finally`), amelyek lehetővé teszik a kivételek dobását és elkapását. Az a típus, amely képviseli az aktuális problémát, a `System.Exception` osztályból (vagy annak egy leszármazottjából) származó osztály. A következőkben megvizsgáljuk ennek a kivételközpontú ősosztálynak a szerepét.

A System.Exception ősosztály

Minden, a felhasználó, illetve a rendszer által definiált kivétel végső soron a `System.Exception` ősosztályból származik, amely viszont a `System.Object` ősosztályból ered. Ez a típus kritikus pontja (figyeljük meg, hogy néhány tag virtuális, ezért a származott osztályok felüldefiniálhatjuk):

```
public class Exception : ISerializable, _Exception
{
  // Nyilvános konstruktorok.
  public Exception(string message, Exception innerException);
  public Exception(string message);
  public Exception();
```

```
    // Metódusok.
    public virtual Exception GetBaseException();
    public virtual void GetObjectData(SerializationInfo info,
    StreamingContext context);

    // Tulajdonságok.
    public virtual IDictionary Data { get; }
    public virtual string HelpLink { get; set; }
    public System.Exception InnerException { get; }
    public virtual string Message { get; }
    public virtual string Source { get; set; }
    public virtual string StackTrace { get; }
    public MethodBase TargetSite { get; }
}
```

Látható, hogy a `System.Exception` által definiált számos tulajdonság írásvédett. Ez annak az egyszerű ténynek köszönhető, hogy a származtatott típusok általában alapértelmezett értéket biztosítanak az egyes tulajdonságok esetében (pl. az `IndexOutOfRangeException` típus alapértelmezett üzenete: „Index was outside the bounds of the array").

Megjegyzés Az `Exception` osztály két .NET-interfészt valósít meg. Az `_Exception` interfész lehetővé teszi, hogy egy nemfelügyelt kód (például egy COM-alkalmazás) feldolgozzon egy .NET-kivételt, míg az `ISerializable` interfész azt teszi lehetővé, hogy egy kivételobjektum megőrződjön a határokon (pl. egy géphatáron) túl (az interfészeket lásd a 9. fejezetben).

A 7.1. táblázat részletezi a `System.Exception` néhány (de nem az összes) tagját.

System.Exception-tulajdonság	Valós jelentése
`Data`	Ez a tulajdonság visszakeresi azoknak a kulcs/érték pároknak a gyűjteményét (ezeket az `IDictionaryt` megvalósító objektum képviseli), amelyek további, a programozó által megadott információkat tartalmaznak a kivételről. Az alapértelmezés szerint ez a gyűjtemény üres (pl. null).
`HelpLink`	Ez a tulajdonság egy URL-t ad vissza egy súgófájlhoz vagy egy webhelyhez, amely részletesen leírja a hibát.
`InnerException`	Ez az írásvédett tulajdonság arra használható, hogy információkat szerezzünk azokról a korábbi hibá(k)ról, amely(ek) a jelenlegi hibát okozták. A korábbi hiba (vagy hibák) rögzítése úgy történik, hogy átadjuk őket a legújabb hiba konstruktorának.

System.Exception-tulajdonság	Valós jelentése
`Message`	Ez az írásvédett tulajdonság visszaadja egy adott hiba szöveges leírását. Maga a hibaüzenet konstruktorparaméterként van beállítva.
`Source`	Ez a tulajdonság visszaadja annak a szerelvénynek a nevét, amely a kivételt dobta.
`StackTrace`	Ez az írásvédett tulajdonság egy olyan sztringet tartalmaz, amely azonosítja a kivételt kiváltó hívások listáját. Ez a tulajdonság rendkívül hasznos a hibakereséskor, vagy ha szeretnénk kiíratni a hibát egy külső hibanaplóba.
`TargetSite`	Ez az írásvédett tulajdonság visszaad egy `MethodBase` típust, amely számos részletet leír a hibát dobó metódusról (a `ToString()` meghívása név szerint azonosítja a metódust).

7.1. táblázat: *A System.Exception típus alaptagjai*

A lehető legegyszerűbb példa

A strukturált kivételkezelés hasznának bemutatásához létre kell hoznunk egy olyan típust, amely helyes körülmények között is dobhat kivételt. Tegyük fel, hogy létrehoztunk egy új parancssorialkalmazás-projektet (`SimpleException` néven), amely két osztálytípust határoz meg (`Car` és `Radio`), „van egy" kapcsolatban. A `Radio` típus olyan metódust definiál, amely ki-/bekapcsolja a rádiót:

```
class Radio
{
  public void TurnOn(bool on)
  {
    if(on)
      Console.WriteLine("Jamming...");
    else
      Console.WriteLine("Quiet time...");
  }
}
```

Amellett, hogy kihasználja a `Radio` típust a tartalmazás/delegálás révén, a `Car` típus úgy lett definiálva, hogy ha a felhasználó az előre megadott maximális sebességnél (ezt a `MaxSpeed` nevű konstans tagváltozó írja elő) jobban felgyorsít egy `Car` objektumot, akkor annak a motorja felrobban, és így használhatatlanná válik (ez egy `carIsDead` nevű `bool` tagváltozóval van rögzítve).

Továbbá a Car típus rendelkezik néhány olyan tagváltozóval, amelyek az aktuális sebességet és a felhasználó által megadott „becenevet" ábrázolják, valamint rendelkezik különböző konstruktorokkal is, amelyek egy új Car objektum állapotát állítják be. A teljes definíció (megjegyzésekkel ellátva) a következő:

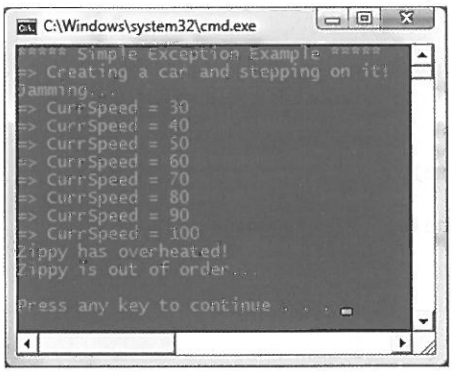
```
class Car
{
  // A maximális sebesség konstansa.
  public const int MaxSpeed = 100;

  // Belső állapot adatok.
  private int currSpeed;
  private string petName;

  // Az autó még mindig működőképes?
  private bool carIsDead;

  // Az autónak "van egy" rádiója.
  private Radio theMusicBox = new Radio();

  // Konstruktorok.
  public Car() {}
  public Car(string name, int currSp)
  {
    currSpeed = currSp;
    petName = name;
  }

  public void CrankTunes(bool state)
  {
    // Metódusreferencia kérés a belső objektumhoz.
    theMusicBox.TurnOn(state);
  }

  // Túlmelegedett az autó?
  public void Accelerate(int delta)
  {
    if (carIsDead)
      Console.WriteLine("{0} is out of order...", petName);
    else
    {
      currSpeed += delta;
      if (currSpeed > MaxSpeed)
      {
        Console.WriteLine("{0} has overheated!", petName);
        currSpeed = 0;
        carIsDead = true;
      }
```

```
        else
          Console.WriteLine("=> CurrSpeed = {0}", currSpeed);
      }
    }
  }
```

Ha most megvalósítanánk azt a `Main()` metódust, amely egy `Car` objektumot arra kényszerít, hogy meghaladja a megadott maximális sebességet:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Exception Example *****");
  Console.WriteLine("=> Creating a car and stepping on it!");
  Car myCar = new Car("Zippy", 20);
  myCar.CrankTunes(true);

  for (int i = 0; i < 10; i++)
    myCar.Accelerate(10);
  Console.ReadLine();
}
```

akkor a 7.1. ábra kimenetét látnánk.



7.1. ábra: *A Car típus működés közben*

Általános kivétel dobása

Miután már rendelkezünk egy működő `Car` típussal, nézzük meg, hogy hogyan lehet a legegyszerűbben kivételt dobni. Az `Accelerate()` metódus jelenlegi megvalósítása egyszerűen hibaüzenetet jelenít meg, ha a hívó megpróbálja a felső határon túlra gyorsítani az autót.

Ahhoz, hogy ez a metódus a túlzott felgyorsításkor kivételt dobjon, létre kell hoznunk, majd fel kell paramétereznünk a `System.Exception` osztály egy új példányát, beállítva az írásvédett `Message` tulajdonság értékét az osztálykonstruktorral. Ha vissza akarjuk küldeni a hibaobjektumot a hívónak, használjuk a C# `throw` kulcsszavát. A releváns kódmódosítás az `Accelerate()` metódushoz a következő:

```
// Kivételt jelez, ha a felhasználó a MaxSpeed értéke fölé
// gyorsítja az autót.
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", petName);
  else
  {
    currSpeed += delta;
    if (currSpeed >= MaxSpeed)
    {
      carIsDead = true;
      currSpeed = 0;

      // Kivétel kiváltása a "throw" kulcsszó segítségével.
      throw new Exception(string.Format("{0} has overheated!",
        petName));
    }
    else
      Console.WriteLine("=> CurrSpeed = {0}", currSpeed);
  }
}
```

Nézzük meg, milyen érdekességeket mutat a fenti metódus. Először is, egy kivétel dobásakor mindig rajtunk múlik, hogy pontosan miből is áll a szóban forgó hiba, és azt mikor is kell dobni. Tegyük fel, hogy ha a program megpróbál egy már nem létező autót gyorsítani, akkor egy `System.Exception` típust kell dobni annak jelzésére, hogy az `Accelerate()` metódus nem folytatható (ez lehet érvényes vagy érvénytelen feltételezés).

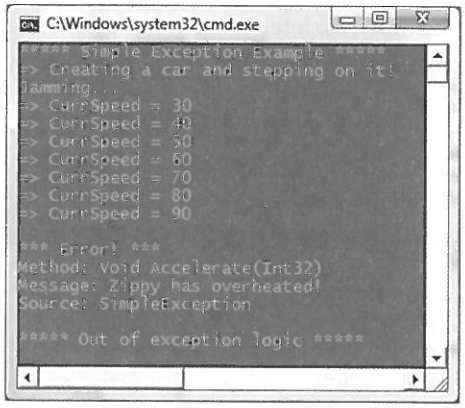
Vagy pedig megvalósíthatjuk úgy is az `Accelerate()` metódust, hogy automatikusan helyreálljon anélkül, hogy először kivételt dobna. Kivételeket nagyjából akkor érdemes dobni, ha valamilyen szélsőséges feltétel teljesül (pl. nincs meg egy szükséges fájl, nem lehet csatlakozni egy adatbázishoz, stb.). Az a döntés, hogy pontosan mi is dobjon kivételt, pusztán tervezési kérdés, amellyel mindig a programozónak kell foglalkoznia. Jelenlegi céljainkhoz tételezzük fel, hogy egy tönkrement autó sebességének növelése megfelelő ok a kivétel dobására.

Kivételek elkapása

Ha az `Accelerate()` metódus kivételt dob, a hívónak készen kell állnia arra, hogy ezt kezelni tudja. Amikor egy olyan metódust hívunk, amely dobhat kivételt, használjuk a `try/catch` blokkot. Amint elkaptuk a kivételobjektumot, máris meg tudjuk hívni a `System.Exception` típus tagjait, hogy részletezzék a problémát. Hogy mihez kezdünk az adatokkal, az már rajtunk múlik. Naplózhatjuk ezt az információt egy jelentésfájlba, beírhatjuk az adatokat a Windows eseménynaplójába, elküldhetjük e-mailben a rendszergazdának, vagy megjeleníthetjük a problémát a felhasználónak. Alább egyszerűen kiírtuk a tartalmat a konzolablakba:

```
// A kiváltott kivétel kezelése.
static void Main(string[] args)
{
  Console.WriteLine("***** Simple Exception Example *****");
  Console.WriteLine("=> Creating a car and stepping on it!");
  Car myCar = new Car("Zippy", 20);
  myCar.CrankTunes(true);

  // Az autó maximális sebessége fölé gyorsítással
  // a kivétel kiváltása.
  try
  {
    for(int i = 0; i < 10; i++)
      myCar. Accelerate(10);
  }
  catch(Exception e)
  {
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Method: {0}", e.TargetSite);
    Console.WriteLine("Message: {0}", e.Message);
    Console.WriteLine("Source: {0}", e.Source);
  }

  // A hibát kezeltük, a feldolgozás a következő utasítással
  // folytatódik.
  Console.WriteLine("\n***** Out of exception logic *****");
  Console.ReadLine();
}
```

A `try` blokk lényegében egy olyan utasításcsoport, amely dobhat kivételt a végrehajtás közben. Kivétel észlelésekor a vezérlést a `catch` blokk kapja meg. Ha viszont a `try` blokkon belüli kód nem idéz elő kivételt, akkor a `catch` blokk teljes mértékben kimarad, és minden rendben működik. A 7.2. ábra ennek a programnak a tesztfutását mutatja.



7.2. ábra: *A hiba kezelése a strukturált kivételkezeléssel*

Egy kivétel kezelése után az alkalmazás a `catch` blokk utáni résszel folytathatja a működést. Bizonyos körülmények között egy kivétel elég kritikus lehet ahhoz, hogy leállítsa a programot. Az esetek nagy részében azonban a kivételkezelő logikája biztosítja, hogy az alkalmazás tovább működjön (jóllehet, bizonyos funkciók nem szükségszerűen fognak ekkor működni, például ha nem lehet csatlakozni egy távoli adatforráshoz).

A kivétel tulajdonságainak beállítása

Az `Accelerate()` metóduson belül létrehozott `System.Exception` objektum a jelenlegi példánkban pusztán létrehoz egy értéket, amelyet aztán átad a `Message` tulajdonságnak (egy konstruktorparaméter révén). Ahogy azonban a 7.1. táblázatban láthattuk, az `Exception` osztály számos más tagot is támogat (`TargetSite`, `StackTrace`, `HelpLink` és `Data`). Példánk további finomításához vizsgáljuk meg a tagok részleteit esetről esetre.

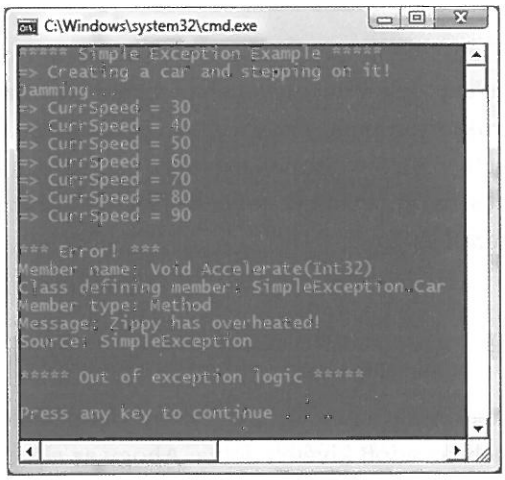
A TargetSite tulajdonság

A `System.Exception.TargetSite` tulajdonság lehetővé teszi, hogy definiáljuk annak a metódusnak a részleteit, amely az adott kivételt dobta. Ahogy az előző `Main()` metódusban látszott, a `TargetSite` értékének a kiírása megjeleníti a kivételt dobó metódus visszatérési értékét, nevét és paramétereit. A `TargetSite` azonban nemcsak egy sztringet ad vissza, hanem egy `System.Reflection.Met-`

hodBase objektumot is. Ezzel a típussal számos részletet össze lehet gyűjteni a hibát okozó metódusról, valamint az ezt definiáló osztályról. Tegyük fel, hogy az előző catch logikát a következőképpen módosítottuk:

```
static void Main(string[] args)
{
  ...
  // A TargetSite a MethodBase objektumot adja vissza.
  catch(Exception e)
  {
    Console.WriteLine("\n*** Error! ***");
    Console.WriteLine("Member name: {0}", e.TargetSite);
    Console.WriteLine("Class defining member: {0}",
      e.TargetSite.DeclaringType);
    Console.WriteLine("Member type: {0}", e.TargetSite.MemberType);
    Console.WriteLine("Message: {0}", e.Message);
    Console.WriteLine("Source: {0}", e.Source);
  }
  Console.WriteLine("\n***** Out of exception logic *****");
  Console.ReadLine();
}
```

Ebben az esetben a MethodBase.DeclaringType tulajdonságot használtuk arra, hogy megadjuk annak az osztálynak a teljesen definiált nevét, amely a hibát dobta (jelen esetben a SimpleException.Car), valamint, hogy a MethodBase objektum MemberType tulajdonságával azonosítsuk azt a tagot (pl. tulajdonság vagy metódus), ahol ez a kivétel történt. A 7.3. ábra mutatja a frissített kimenetet.



7.3. ábra: *A céloldal szempontjainak megszerzése*

A StackTrace tulajdonság

A `System.Exception.StackTrace` tulajdonság lehetővé teszi annak a hívássorozatnak az azonosítását, amely kivételt eredményezett. Soha nem mi állítjuk be a `StackTrace` értékét, ugyanis az automatikusan jön létre a kivétel keletkezésekor. Ennek bemutatására tegyük fel, hogy ismét módosítottuk a `catch` logikát:

```
catch(Exception e)
{
  ...
  Console.WriteLine("Stack: {0}", e.StackTrace);
}
```

Ha futtatnánk a programot, akkor a következő verem-nyomkövetést látnánk a konzolon (a sorok száma és a fájlok elérési útja természetesen különbözhet):

```
Stack: at SimpleException.Car.Accelerate(Int32 delta)
in c:\MyApps\SimpleException\car.cs:line 65 at SimpleException.
Program.Main()
in c:\MyApps\SimpleException\Program.cs:line 21
```

A `StackTrace`-től visszakapott sztring dokumentálja azt a híváslistát, amely a kivételt eredményezte. A sztring legalsó sorszáma azonosítja a lista első hívását, míg a legfelső a kivételt okozó tag pontos helyét. Ez az információ hasznos lehet hibakeresés vagy egy adott alkalmazás naplózása során, hiszen így nyomon követhetjük a hibát okozó folyamatot.

A HelpLink tulajdonság

Míg a `TargetSite` és a `StackTrace` tulajdonságok révén a programozó megért egy adott kivételt, ugyanez az információ semmit nem mond a végfelhasználónak. A `System.Exception.Message` tulajdonsággal azonban olvasható és megjeleníthető információkhoz jutunk. Emellett a `HelpLink` tulajdonságot be lehet úgy állítani, hogy a felhasználót egy olyan URL-re vagy szabványos Windows-súgófájlhoz irányítsa, amely részletesebb információkat tartalmaz.

Az alapértelmezés szerint a `HelpLink` tulajdonság által kezelt érték üres sztring. Ha valamilyen értéket szeretnénk adni ennek a tulajdonságnak, akkor ezt még azelőtt meg kell tennünk, mielőtt dobnánk a `System.Exception` típust. A releváns módosítás a `Car.Accelerate()` metódushoz a következő:

```
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", petName);
  else
  {
    currSpeed += delta;
    if (currSpeed >= MaxSpeed)
    {
      carIsDead = true;
      currSpeed = 0;

      // A HelpLink tulajdonságot kell hívni, ezért
      // az Exception objektum kiváltása előtt
      // lokális változót kell létrehozni.
      Exception ex =
        new Exception(string.Format("{0} has overheated!",
          petName));
      ex.HelpLink = "http://www.CarsRUs.com";
      throw ex;
    }
    else
      Console.WriteLine("=> CurrSpeed = {0}", currSpeed);
  }
}
```

A `catch` logika most már frissíthető úgy, hogy kiírja ezt a súgóhivatkozást:

```
catch(Exception e)
{
  ...
  Console.WriteLine("Help Link: {0}", e.HelpLink);
}
```

A Data tulajdonság

A `System.Exception` `Data` tulajdonsága lehetővé teszi, hogy kitöltsünk egy kivételobjektumot a releváns kiegészítő információkkal (például időbélyeg stb.). A `Data` tulajdonság egy olyan objektumot ad vissza, amely megvalósítja a `System.Collections` névtérben megadott `IDictionary` nevű interfészt. (Az interfészalapú programozást, valamint a `System.Collections` névteret lásd a 9. fejezetben.) A szótárgyűjteményekkel olyan értékkészletet lehet létrehozni, amelyet egy bizonyos kulcs használatával kereshetünk vissza. Nézzük meg a `Car.Accelerate()` metódus alábbi módosítását:

```
public void Accelerate(int delta)
{
  if (carIsDead)
    Console.WriteLine("{0} is out of order...", petName);
  else
  {
    currSpeed += delta;
    if (currSpeed >= MaxSpeed)
    {
      carIsDead = true;
      currSpeed = 0;

      // A HelpLink tulajdonságot kell hívni, ezért
      // az Exception objektum kiváltása előtt
      // lokális változót kell létrehozni.
      Exception ex =
        new Exception(string.Format("{0} has overheated!",
          petName));
      ex.HelpLink = "http://www.CarsRUs.com";

      // A hibára vonatkozó egyedi adatok betöltése.
      ex.Data.Add("TimeStamp",
        string.Format("The car exploded at {0}", DateTime.Now));
      ex.Data.Add("Cause", "You have a lead foot.");
      throw ex;
    }
    else
      Console.WriteLine("=> CurrSpeed = {0}", currSpeed);
  }
}
```

A kulcs/érték párok sikeres felsorolásához először meg kell adnunk egy használati irányelvet a `System.Collections` névtér számára, ugyanis a `DictionaryEntry` típust fogjuk használni abban a fájlban, amely a `Main()` metódus megvalósítását tartalmazza:

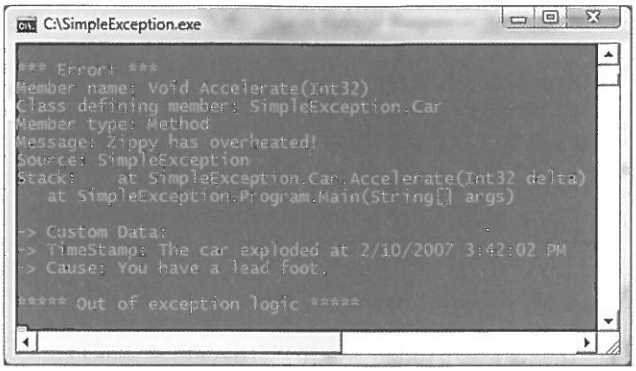
```
using System.Collections;
```

Ezután úgy kell módosítanunk a `catch` logikát, hogy ellenőrizze, hogy a `Data` tulajdonságtól visszakapott érték nem null (vagyis az alapértelmezett érték). Ezt követően a `DictionaryEntry` típus `Key` és `Value` tulajdonságait felhasználva írathatjuk ki az egyedi adatokat a konzolra:

```
catch (Exception e)
{
  ...
  // Alapértelmezés szerint az adatmező üres, ellenőrizni kell
  // a null értéket.
  Console.WriteLine("\n-> Custom Data:");
```

```
    if (e.Data != null)
    {
      foreach (DictionaryEntry de in e.Data)
      Console.WriteLine("-> {0}: {1}", de.Key, de.Value);
    }
  }
```

A 7.4. ábrán a bemutatott módosítást láthatjuk.



7.4. ábra. *A programozó által definiált adatok megszerzése*

A `Data` haszna abban rejlik, hogy lehetővé teszi, hogy a hibával kapcsolatos információkat belefoglalhassuk anélkül, hogy egy olyan teljesen új osztálytípust kellene építenünk, amely kibővíti az `Exception` ősosztályt (a .NET 2.0 előtt ez volt az egyetlen lehetőségünk). Bármilyen hasznos is a Data tulajdonság, a .NET-fejlesztők körében mégis szokványos, hogy erősen típusos kivételosztályokat építenek, amelyek aztán erősen típusos tulajdonságok felhasználásával felelnek az egyedi adatokért.

Ez a megközelítés teszi lehetővé a hívó számára, hogy egy specifikus `Exception`ből származtatott típust kapjon el anélkül, hogy bele kéne ásnia magát egy adatgyűjteménybe a további részletek megszerzéséért. Hogy megértsük ennek a mikéntjét, meg kell vizsgálnunk a különbséget a rendszerszintű és az alkalmazásszintű kivételek között.

Forráskód A `SimpleException` projekt megtalálható a 7. fejezet alkönyvtárában.

Rendszerszintű kivételek (System.SystemException)

A .NET-alaposztálykönyvtárak számos olyan osztályt definiálnak, amelyek végső soron a `System.Exception` osztályból származnak. A System névtér például olyan alapvető hibaobjektumokat definiál, mint az `ArgumentOutOfRangeException`, az `IndexOutOfRangeException` vagy a `StackOverflowException` stb. Más névterek olyan kivételeket határoznak meg, amelyek tükrözik az adott névtér viselkedését (pl. a `System.Drawing.Printing` a nyomtatási kivételeket, a `System.IO` az IO-alapú kivételeket, a `System.Data` az adatbázissal kapcsolatos kivételeket definiálja; stb.).

A .NET-platform által dobott kivételeket (ennek megfelelően) *rendszerkivételeknek* nevezzük. Ezek a kivételek nem helyreállítható, végzetes hibák. A rendszerkivételek közvetlenül a `System.SystemException` ősosztályból származnak, amely viszont a `System.Exception` osztályból (amely a `System.Object` osztályból):

```
public class SystemException : Exception
{
  // Különböző konstruktorok.
}
```

Mivel a `System.SystemException` típus nem rendelkezik további funkcionalitással egy egyedi konstruktorkészleten kívül, kérdés, hogy miért a `SystemException` áll az első helyen. Ha egy kivételtípus a `System.SystemException` osztályból származik, akkor egyértelmű, hogy a .NET-futtatórendszer dobta a kivételt, nem pedig a végrehajtó alkalmazás forráskódja. Ezt egészen egyszerűen az is kulcsszó használatával ellenőrizhetjük:

```
// Igaz! A NullReferenceException „az egy" SystemException.
NullReferenceException nullRefEx = new NullReferenceException();

Console.WriteLine("NullReferenceException is-a SystemException? :
  {0}", nullRefEx is SystemException);
```

Alkalmazásszintű kivételek (System.ApplicationException)

Mivel az összes .NET-kivétel osztálytípus, ezért nyugodtan létrehozhatjuk a saját, alkalmazásspecifikus kivételeinket. Mivel azonban a `System.SystemException` ősosztály CLR-ből dobott kivételeket képvisel, így feltételezhetjük, hogy az egyedi kivételeket a `System.Exception` típusból kell származtatnunk. Noha ezt is megtehetjük, a legjobb mégis az, ha inkább a `System.ApplicationException` típusból származtatjuk ezeket:

```
public class ApplicationException : Exception
{
  // Különböző konstruktorok.
}
```

A `SystemException` típushoz hasonlóan az `ApplicationException` sem definiál konstruktorkészleten kívüli további tagokat. A `System.ApplicationException` egyetlen célja a hibaforrás azonosítása. Ha a `System.ApplicationException` típusból származó kivételt kezelünk, akkor feltételezhetjük, hogy a kivétel a végrehajtó alkalmazás forráskódjából származik, nem pedig a .NET-alaposztálykönyvtárakból vagy a .NET-futtatómotorból.

Egyedi kivétel készítése, 1. rész

Bár a `System.Exception` osztályt mindig használhatjuk a futásidejű hibák jelzésére (ahogy az első példánkban is), néha mégis célszerűbb olyan *erősen típusos kivételt* készíteni, amely az aktuális probléma egyedi részleteit ábrázolja. Tegyük fel például, hogy szeretnénk egy egyedi kivételt készíteni (`CarIsDeadException` néven) a működésképtelen autó gyorsításából fakadó hiba jelzésére. Az első lépés az, hogy egy új osztályt származtatunk a `System.ApplicationException` típusból (a megállapodás szerint minden kivételosztály az „`Exception`" utótaggal végződik).

Megjegyzés A szabály az, hogy minden egyedi kivételosztályt nyilvános típusként kell definiálni (egy nem beágyazott típus alapértelmezett hozzáférés-módosítója belső lesz). Ennek az az oka, hogy a kivételek gyakran a szerelvényhatárokon kívülre kerülnek, és a meghívó kódnak ezért hozzá kell tudni férni.

Hozzunk létre egy új parancssorialkalmazás-projektet `CustomException` névvel, majd másoljuk át az előző `Car` és `Radio` definíciókat az új projektbe a Project > Add Existing Item menüpont felhasználásával (ne felejtsük el megváltoztatni a `Car` és `Radio` típusokat definiáló névteret `SimpleExceptionről CustomExceptionre`). Ezután adjuk hozzá a következő osztálydefiníciót:

```
// Az egyedi kivétel leírja a
// car-is-dead feltétel részleteit.
public class CarIsDeadException : ApplicationException
{}
```

A többi osztályhoz hasonlóan hozzáadhatunk bármennyi egyedi tagot, amelyeket a hívó logika `catch` blokkján belül meg lehet hívni. Emellett felülbírálhatjuk a szülőosztály által definiált bármelyik virtuális tagot is. Megvalósíthatjuk például a `CarIsDeadExceptiont` úgy, hogy felüldefiniáljuk a virtuális `Message` tulajdonságot.

Továbbá ahelyett, hogy kitöltenénk az adatgyűjteményt (a `Data` tulajdonság révén), amikor dobjuk a kivételünket, a konstruktorunk lehetővé teszi, hogy a küldő átadjon egy időbélyeget, valamint a hiba okát. Végül, az időbélyeg és a hiba oka megszerezhető erősen típusos tulajdonságok használatával is:

```
public class CarIsDeadException : ApplicationException
{
  private string messageDetails;
  private DateTime errorTimeStamp;
  private string causeOfError;

  public DateTime TimeStamp
  {
    get {return errorTimeStamp;}
    set {errorTimeStamp = value;}
  }

  public string Cause
  {
    get {return causeOfError;}
    set {causeOfError = value;}
  }

  public CarIsDeadException(){}
  public CarIsDeadException(string message,
  string cause, DateTime time)
  {
    messageDetails = message;
    causeOfError = cause;
    errorTimeStamp = time;
  }
```

```
    // Az Exception.Message tulajdonság felülbírálása.
    public override string Message
    {
      get
      {
        return string.Format("Car Error Message: {0}",
          messageDetails);
      }
    }
}
```

A `CarIsDeadException` típus fenntart egy privát adattagot (`messageDetails`), amely az aktuális kivétellel kapcsolatos adatokat jelképezi. Ezeket egyedi konstruktor használatával lehet beállítani. Az `Accelerate()` metódus ezen hibájának a dobása nagyon lényegre törő. Egyszerűen hozzunk létre, majd a paraméterek beállításával dobjunk egy `CarIsDeadException` típust egy `System.Exception` helyett (ebben az esetben többé nem kell saját kezűleg kitöltenünk az adatgyűjteményt):

```
// Az egyedi CarIsDeadException kiváltása.
public void Accelerate(int delta)
{
  ...
  CarIsDeadException ex =
    new CarIsDeadException (string.Format("{0} has overheated!",
      petName), "You have a lead foot", DateTime.Now);
  ex.HelpLink = "http://www.CarsRUs.com";
  throw ex;
  ...
}
```

Ennek az eldobott kivételnek (a specifikus `CarIsDeadException` típusnak) az elkapásához a `catch` hatókörét kell módosítani (mivel azonban a `CarIsDeadException` „az egy" `System.Exception`, ezért még mindig el lehet fogni egy `System.Exception` kivételt is):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Custom Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);

  try
  {
    // Utazási kivétel.
    myCar.Accelerate(50);
  }
```

```
    catch (CarIsDeadException e)
    {
      Console.WriteLine(e.Message);
      Console.WriteLine(e.TimeStamp);
      Console.WriteLine(e.Cause);
    }
    Console.ReadLine();
  }
```

Mikor van szükség egyedi kivétel készítésére? Általában csak akkor kell létrehoznunk ezeket, ha egy hiba szorosan kapcsolódik az őt okozó osztályhoz (egy egyedi fájlcentrikus osztályhoz például, amely dob számos, fájlhoz kapcsolódó hibát; egy `Car` osztályhoz, amely dob számos, autóhoz kapcsolódó hibát; stb.). Azáltal, hogy egyedi kivételtípusokat hozunk létre, lehetőséget adunk a hívónak arra, hogy a kivételeket típusuk szerint csoportosítva (hibáról hibára haladva) dolgozza fel.

Egyedi kivétel készítése, 2. rész

Az aktuális `CarIsDeadException` típus felülbírálta a `System.Exception.Message` tulajdonságot azért, hogy egyedi hibaüzenetet állítson be, valamint megadott két egyedi tulajdonságot a további adatrészletek kedvéért. Valójában azonban nem muszáj felülbírálnunk a virtuális `Message` tulajdonságot, ugyanis egyszerűen átadhatjuk a bejövő üzenetet a szülőosztály konstruktorának:

```
public class CarIsDeadException : ApplicationException
{
  private DateTime errorTimeStamp;
  private string causeOfError;

  public DateTime TimeStamp
  {
    get { return errorTimeStamp; }
    set { errorTimeStamp = value; }
  }

  public string Cause
  {
    get { return causeOfError; }
    set { causeOfError = value; }
  }
  public CarIsDeadException() { }
```

```
    // Üzenet átadása a szülőkonstruktornak.
    public CarIsDeadException(string message,
      string cause, DateTime time)
      :base(message)
    {
      causeOfError = cause;
      errorTimeStamp = time;
    }
}
```

Ebben az esetben *nem* definiáltunk sztringváltozót az üzenet ábrázolására, és *nem* definiáltuk felül a `Message` tulajdonságot. Ehelyett egyszerűen átadtuk a paramétert az ősosztály konstruktorának. Így egy egyedi kivétel alig több, mint a `System.ApplicationException` osztályból származó egyedileg elnevezett osztály - az ősosztály felüldefiniálása nélkül.

Célszerű, ha a legtöbb (ha nem az összes) egyedi kivételosztályunk ezt az egyszerű mintát követi. Sokszor egy egyedi kivétel szerepe nem szükségszerűen az, hogy további funkcionalitást biztosítson azon túl, amit az ősosztályoktól örökölt, hanem az, hogy egy *erősen megnevezett típust* biztosítson, amely egyértelműen azonosítja a hiba természetét.

Egyedi kivétel készítése, 3. rész

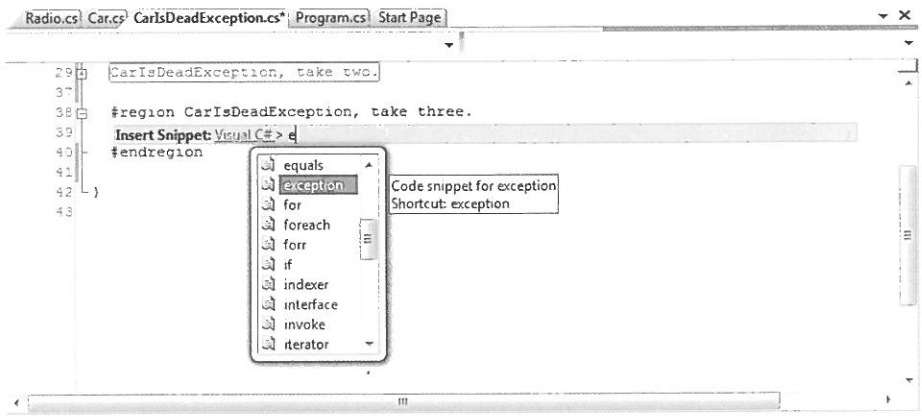
Ha egy valóban megfelelő egyedi kivételosztályt szeretnénk építeni, akkor meg kell bizonyosodnunk róla, hogy a típus megfelel a kivételközpontú .NET legjobb gyakorlatának. Pontosabban, ehhez az kell, hogy az egyedi kivételünk

- az `Exception/ApplicationException` típusból származzon;
- a `[System.Serializable]` attribútum jelölje meg;
- definiáljon egy alapértelmezett konstruktort;
- definiáljon egy olyan konstruktort, amely beállítja az örökölt `Message` tulajdonságot;
- definiáljon egy konstruktort a „belső kivételek" kezelésére;
- definiáljon egy konstruktort a típusunk sorosításának kezelésére.

A .NET-háttérinformációink alapján előfordulhat, hogy nincs elképzelésünk az objektumsorosítás szerepéről, erről később még lesz szó (a 16. fejezetben pl. az attribútumokról). Az egyedi kivételek készítéséhez nézzük még meg a `CarIsDeadException` végleges változtatását, amely valamennyi fent említett speciális konstruktorral rendelkezik:

```
[Serializable]
public class CarIsDeadException : ApplicationException
{
  public CarIsDeadException() { }
  public CarIsDeadException(string message) : base( message ) { }
  public CarIsDeadException(string message,
    System.Exception inner) : base( message, inner ) { }
  protected CarIsDeadException(
    System.Runtime.Serialization.SerializationInfo info,
    System.Runtime.Serialization.StreamingContext context)
      : base( info, context ) { }

  // További egyedi tulajdonságok, konstruktorok és
  // adattagok...
}
```

Mivel a .NET legjobb gyakorlatának megfelelően készített egyedi kivételek csak a nevükben különböznek, a Visual Studio 2008 biztosít egy kódrészlet-sablont „`Exception`" néven (lásd 7.5. ábra), amely automatikusan olyan új kivételosztályt generál, amely megfelel a .NET legjobb gyakorlatának (a kódrészletsablonok magyarázata a 2. fejezetben található).



7.5. ábra. *A kivételkódrészlet-sablon*

Forráskód A `CustomException` projekt megtalálható a 7. fejezet alkönyvtárában.

Többszörös kivételek feldolgozása

Egy try blokknak legegyszerűbb formájában egy catch blokkja van. A try blokk utasításai *számos* lehetséges kivételt okozhatnak. Hozzunk létre egy új parancssorialkalmazás-projektet ProcessMultipleExceptions néven, és adjuk hozzá a már meglévő Car, Radio és CarIsDeadException osztályokat (Project > Add Existing Item menüpont), majd ennek megfelelően módosítsuk a névteret.

Aztán módosítsuk úgy a Car Accelerate() metódusát, hogy dobjon egy előre definiált, alaposztály-könyvtárbeli ArgumentOutOfRangeException kivételt helytelen paraméter átadásakor (amely bármilyen nullánál kisebb érték lehet):

```
// A folytatás előtt esetleges érvénytelen argumentumok ellenőrzése.
public void Accelerate(int delta)
{
  if(delta < 0)
    throw new ArgumentOutOfRangeException("Speed must be greater
      than zero!");
  ...
}
```

A catch logika ekkor specifikusan reagálhat az egyes kivételtípusokra:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);

  try
  {
    // Utazási ArgumentOutOfRange kivétel.
    myCar.Accelerate(-10);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

Amikor többszörös catch blokkot készítünk, figyeljünk arra, hogy kivétel dobásakor a kivételt az „első rendelkezésre álló" catch fogja feldolgozni. Hogy mit is jelent az „első rendelkezésre álló" catch, ehhez példaként tegyük

fel, hogy frissítettük az előző logikát egy további `catch` blokkal, amely egy általános `System.Exception` elkapásával próbál meg kezelni minden kivételt a `CarIsDeadException` és az `ArgumentOutOfRangeException` kivételeken túl:

```
// Ezt a kódot a rendszer nem fordítja le!
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);

  try
  {
    // Utazási ArgumentOutOfRange kivétel.
    myCar.Accelerate(-10);
  }
  catch(Exception e)
  {
    // Minden más kivételt feldolgozzon a rendszer?
    Console.WriteLine(e.Message);
  }
  catch (CarIsDeadException e)
  {
    Console.WriteLine(e.Message);
  }
  catch (ArgumentOutOfRangeException e)
  {
    Console.WriteLine(e.Message);
  }
  Console.ReadLine();
}
```

Ez a kivételkezelési logika fordítási idejű hibát eredményez. A probléma amiatt áll fenn, hogy az első `catch` blokk képes kezelni *mindent*, ami a `System.Exception` osztályból származik (minthogy „az egy" kapcsolat), beleértve a `CarIsDeadException` és az `ArgumentOutOfRangeException` típusokat. Emiatt az utolsó két `catch` blokk elérhetetlenné válik.

Szabályosan a `catch` blokkok struktúráját úgy kell definiálni, hogy a legelső `catch` legyen a legspecifikusabb kivétel (pl. az utolsó leszármazott típus egy kivételtípus-származási láncban), míg a legutolsó a legáltalánosabb (pl. egy adott kivétel származási láncának ősosztálya – jelen esetben a `System.Exception`).

Ezért, ha olyan `catch` blokkot szeretnénk definiálni, amely kezel bármilyen hibát a `CarIsDeadException` és az `ArgumentOutOfRangeException` kivételeken kívül, akkor a következőt kell írnunk:

```
// Ezt a kódot lefordítja a rendszer.
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
```

```
    Car myCar = new Car("Rusty", 90);
    try
    {
      // Utazási ArgumentOutOfRange kivétel.
      myCar.Accelerate(-10);
    }
    catch (CarIsDeadException e)
    {
      Console.WriteLine(e.Message);
    }
    catch (ArgumentOutOfRangeException e)
    {
      Console.WriteLine(e.Message);
    }
    // A CarIsDeadException vagy
    // az ArgumentOutOfRangeException kivételeken túl
    // minden egyéb kivételt észlel.
    catch(Exception e)
    {
      Console.WriteLine(e.Message);
    }
    Console.ReadLine();
}
```

Általános catch utasítások

A C# biztosít egy „általános" `catch` hatókört, amely nem kapja meg explicit módon az adott tag által dobott kivételt:

```
// Általános catch hatókör.
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  try
  {
    myCar.Accelerate(90);
  }
  catch
  {
    Console.WriteLine("Something bad happened...");
  }
  Console.ReadLine();
}
```

Ez a kivételkezelésnek nyilvánvalóan nem a leginformatívabb módja, hiszen sehogy sem tudunk részletes adatokat szerezni a hibáról (a metódus nevét, a hívásvermet vagy egyedi üzenetet). Ennek ellenére a C# engedélyezi ezt a konstrukciót, amely az összes hiba általános kezelésekor lehet hasznos.

Kivételek továbbdobása

A try blokk logikája megengedi a kivétel *továbbdobását* az előző hívó hívásvermébe. Ehhez a throw kulcsszót használjuk a catch blokkban. A throw feladja a kivételt a hívási logika listáján, ez akkor lehet hasznos, ha a catch blokk csak részlegesen képes kezelni az aktuális hibát:

```
// A felelősség áthárítása.
static void Main(string[] args)
{
  ...
  try
  {
    // Az autó gyorsításának logikája...
  }
  catch(CarIsDeadException e)
  {
    // A hiba részleges feldolgozása, és a felelősség áthárítása.
    throw;
  }
}
```

Ebben a példakódban a CarIsDeadException kivételt a CLR kapja el, ugyanis a kivételt a Main() metódus dobja tovább. Ilyenkor a végfelhasználó egy hibapárbeszédpanellel szembesül, amelyet a rendszer ad ki. Általában csak olyan hívónak dobunk tovább részlegesen kezelt kivételt, amely ezt jobban tudja kezelni.

Nem explicit módon dobjuk tovább a CarIsDeadException objektumot, hanem paraméter nélkül használjuk a throw kulcsszót. Ez pedig megőrzi az eredeti célpont kontextusát.

Belső kivételek

Kivételt elő lehet idézni olyankor is, amikor egy másik kivételt kezelünk. Tegyük fel például, hogy egy adott catch hatókörön belül kezelünk egy CarIsDeadException kivételt, és a művelet során megpróbáljuk rögzíteni egy carErrors.txt nevű fájlba a vermet a C meghajtón (meg kell adnunk hozzá a System.IO névteret, hogy hozzáférhessünk ezekhez az I/O-központú típusokhoz):

```
catch(CarIsDeadException e)
{
  // A carErrors.txt fájl megnyitásának kísérlete a C meghajtón.
  FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
  ...
}
```

Ha a megadott fájl nem található a C meghajtón, akkor a `File.Open()` metódus meghívása `FileNotFoundException` kivételt eredményez. Később meg fogjuk vizsgálni részletesen a `System.IO` névteret, valamint azt, hogy hogyan lehet egy fájl létezését ellenőrizni a meghajtón még a megnyitása előtt (s ezzel egyidejűleg a kivételt is elkerülni). Most azonban tételezzük fel, hogy létrejött a kivétel.

Ha az egyik kivétel feldolgozása közben egy másikba botlunk, akkor érdemes „belső kivételként" rögzíteni az új kivételobjektumot, méghozzá ugyanolyan típus új objektumán belül, mint amilyen az első kivétel volt. Azért kell a kezelt kivétel új objektumát lefoglalni, mert egy belső kivétel dokumentálása csakis egy konstruktorparaméterrel lehetséges. Nézzük meg a következő kódot:

```
catch (CarIsDeadException e)
{
  try
  {
    FileStream fs = File.Open(@"C:\carErrors.txt", FileMode.Open);
    ...
  }
  catch (Exception e2)
  {
    // Kivétel kiváltása, amely az új kivételt,
    // valamint az első kivétel üzenetét rögzíti.
    throw new CarIsDeadException(e.Message, e2);
  }
}
```

Jelen esetben második paraméterként adtuk át a `FileNotFoundException` objektumot a `CarIsDeadException` konstruktornak. Amint konfiguráltuk ezt az új objektumot, feldobjuk a hívásverembe a következő hívónak, amely ebben az esetben a `Main()` metódus.

Mivel a `Main()` után nincs „következő hívó", amely elkaphatná a kivételt, ezért újabb hiba-párbeszédpanelt láthatunk. Hasonlóan a kivétel továbbdobásához, a belső kivételek rögzítése általában csak akkor hasznos, ha a hívó az első helyen tudja elfogni a kivételt. Ekkor a hívó `catch` logikája használhatja az `InnerException` tulajdonságot arra, hogy kivonja a belsőkivétel-objektum részleteit.

A finally blokk

Egy `try/catch` hatókör tetszőleges `finally` blokkot is definiálhat. A `finally` blokk azt biztosítja, hogy egy kódutasítás-készlet *mindig* végrehajtódjon, függetlenül attól, hogy van-e (bármilyen) kivétel vagy nincs.

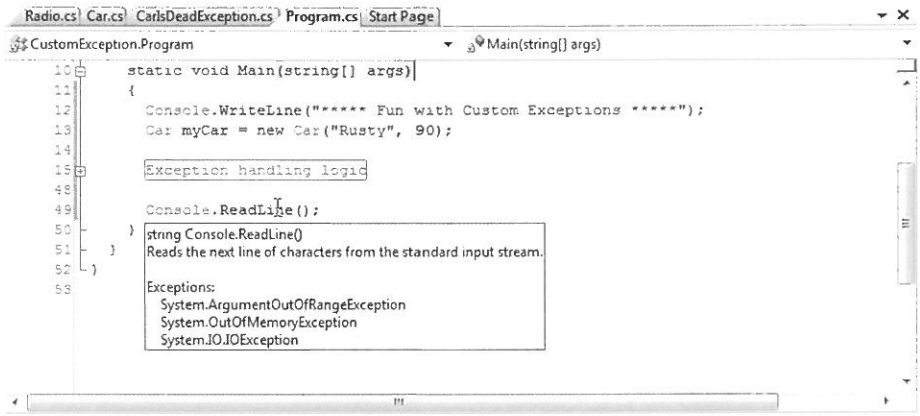
Ennek illusztrálásához tételezzük fel, hogy mindig szeretnénk kikapcsolni az autó rádióját, ha vége a `Main()` metódusnak, függetlenül a kezelt kivételektől:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Handling Multiple Exceptions *****\n");
  Car myCar = new Car("Rusty", 90);
  myCar.CrankTunes(true);

  try
  {
    // Az autó gyorsításának logikája.
  }
  catch(CarIsDeadException e)
  {
    // A CarIsDeadException feldolgozása.
  }
  catch(ArgumentOutOfRangeException e)
  {
    // Az ArgumentOutOfRangeException feldolgozása.
  }
  catch(Exception e)
  {
    // Bármely további kivétel feldolgozása.
  }
  finally
  {
    // Ez mindig bekövetkezik. Kivétel vagy sem.
    myCar.CrankTunes(false);
  }
  Console.ReadLine();
}
```

Ha nincsen `finally` blokk, akkor a rádió nem lesz kikapcsolva, ha felmerül egy kivétel (és ez akár problémát is okozhat). Ha tehát szükségünk van objektumok megszüntetésére, egy fájl bezárására, egy adatbázisról történő lecsatlakozásra (vagy bármire), a `finally` blokk a dolgok megfelelő lezárását biztosítja.

Ki mit dob?

Mivel a .NET-keretrendszerben egy metódus bármennyi kivételt dobhat (különböző körülmények között), felmerül a kérdés: „Honnan tudom, hogy melyik kivételt dobhatja egy adott alaposztálykönyvtár-metódus?" A válasz egyszerű: meg kell nézni a .NET Framework 3.5 SDK dokumentációját. A súgórendszerben minden metódus dokumentálja, hogy egy adott tag milyen kivételeket dobhat. Vagy a Visual Studio 2008-ban megnézzük egy alaposztálykönyvtár tagja által dobott kivételek listáját (ha van), úgy, hogy a kódablakban a tag neve fölé visszük a kurzort (lásd 7.6. ábra).



7.6. ábra: *Egy adott metódusból dobott kivételek azonosítása*

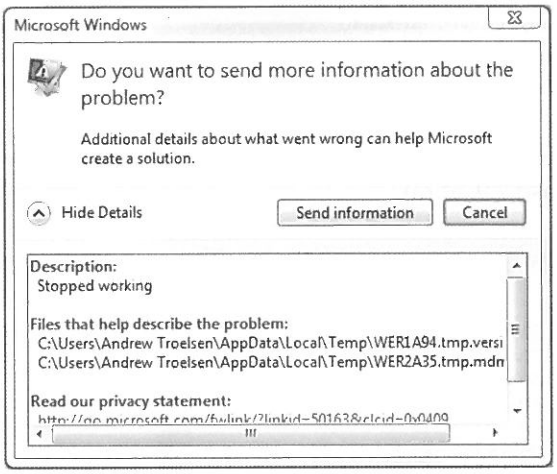
Akik Java-háttérrel vágnak bele a .NET-be, láthatják, hogy a típustagoknak *nincsenek* prototípusaik a dobható kivételek készletével (más szóval: a .NET nem támogatja az ellenőrzött kivételeket). Akárhogy is, *nem* kell az adott tagok által dobott minden egyes kivételt kezelnünk. Sokszor az összes lehetséges dobott hibát egy hatókörkészletből kezelhetjük, egyetlen `System.Exception` elfogásával:

```
static void Main(string[] args)
{
  try
  {
    File.Open("IDontExist.txt", FileMode.Open);
  }
  catch(Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Ha azonban egyenként akarunk kezelni bizonyos kivételeket, akkor többszörös `catch` blokkokat kell használnunk.

A kezeletlen kivételek eredménye

Kérdés: mi történik, ha nem kezelünk egy dobott kivételt. Tegyük fel, hogy a `Main()` metódus logikája a maximálisnál nagyobbra növeli a `Car` objektum sebességét a `try/catch` logika nélkül. Egy kivétel figyelmen kívül hagyása nagyban akadályozhatja az alkalmazás végfelhasználóját, hiszen megjelenik egy „kezeletlen kivétel" párbeszédpanel (lásd 7.7. ábra).



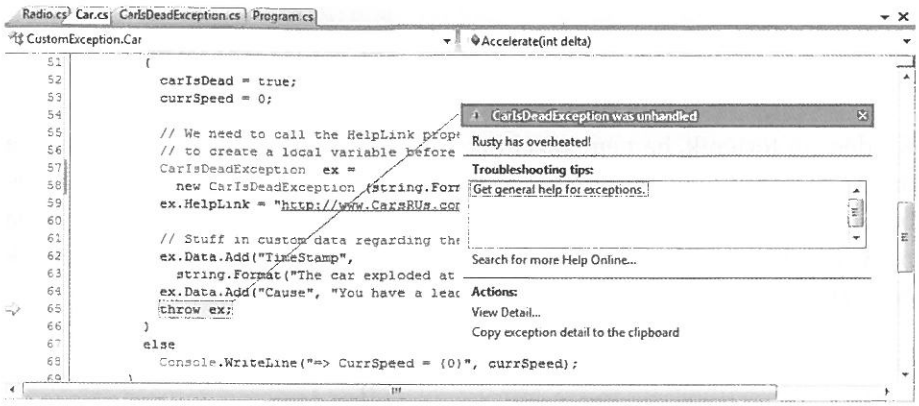
7.7. ábra: *A kivételek figyelmen kívül hagyásának eredménye*

Forráskód A `ProcessMultipleExceptions` projekt megtalálható a 7. fejezet alkönyvtárában.

Kezeletlen kivételek hibakeresése a Visual Studio használatával

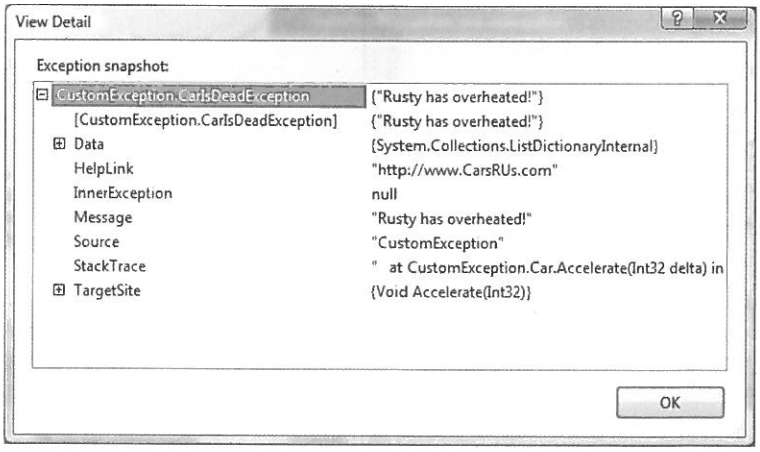
A Visual Studio 2008 számos eszközt biztosít, amelyek segítenek megkeresni a nem kezelt egyedi kivételeket. Ismét tegyük fel, hogy egy `Car` objektum sebességét a maximálisnál nagyobbra növeltük. Ha egy hibakeresési munkamenetet

kezdenénk (a Debug > Start menüpont használatával), a Visual Studio automatikusan leállna az el nem kapott kivétel dobásakor. Megjelenik egy olyan ablak is (lásd 7.8. ábra), amely megmutatja a `Message` tulajdonság értékét.



7.8. ábra: *Nem kezelt egyedi kivételek hibakeresése a Visual Studio 2008-ban*

Ha rákattintunk a View Detail hivatkozásra, látni fogjuk az objektum állapotával kapcsolatos részleteket (lásd 7.9. ábra).



7.9. ábra: *Kivétel részleteinek megtekintése*

Megjegyzés Ha nem kezelünk egy olyan kivételt, amelyet a .NET alaposztálykönyvtáraiban lévő metódus dob, akkor a Visual Studio 2008 hibakeresője leáll annál az utasításnál, amely meghívta az adott metódust.

Összefoglalás

Ebben a fejezetben megvizsgáltuk a strukturált kivételkezelés szerepét. Amikor egy metódusnak hibaobjektumot kell küldenie a hívó számára, akkor lefoglal, felparaméterez és dob egy adott `System.Exception` leszármazott típust a C# `throw` kulcsszavával. A hívó képes kezelni bármilyen lehetséges bejövő kivételt a C# `catch` kulcsszava és egy opcionális `finally` hatókör használatával.

Saját egyedi kivételeink létrehozásakor végső soron egy olyan, a `System.ApplicationException` osztályból származó osztálytípust hozunk létre, amely jelzi, hogy a kivételt a jelenleg használt végrehajtó alkalmazás dobta. Ezzel ellentétben a `System.SystemException` osztályból származó hibaobjektumok a CLR által dobott, kritikus (és végzetes) hibákat jeleznek. Végül, de nem utolsósorban ez a fejezet bemutatott számos eszközt a Visual Studio 2008-on belül, amelyekkel egyedi kivételeket lehet létrehozni (a .NET legjobb gyakorlatának megfelelően), valamint kivételeket lehet keresni.

NYOLCADIK FEJEZET

Az objektumok élettartama

A C# segítségével készített egyedi osztálytípusok tanulmányozása után ebben a fejezetben megvizsgáljuk, hogy a CLR (Common Language Runtime - közös nyelvi futtatórendszer) hogyan kezeli a szemétgyűjtő szolgáltatással a lefoglalt objektumokat. A C#-programozók soha nem közvetlenül a memóriában szabadítják fel a kezelt objektumot (emlékezzünk rá, hogy a C# nyelvben nincs `delete` kulcsszó), hanem a felügyelt heap memóriaterületén foglalják a .NET-objektumokat, ahol a szemétgyűjtő „valamikor a közeljövőben" automatikusan megsemmisíti őket.

A gyűjtési folyamat legfontosabb részletei után megnézzük, hogyan lehet programból együttműködni a szemétgyűjtővel a `System.GC` osztálytípus felhasználásával. Ezt követően a virtuális `System.Object.Finalize()` metódussal és az `IDisposable` interfésszel készítünk olyan típusokat, amelyek bizonyos időközönként felszabadítják a belső *nem felügyelt erőforrásokat*. Célunk tehát, hogy bemutassuk a CLR .NET-objektumkezelési lehetőségeit.

Osztályok, objektumok és referenciák

A fejezet későbbi részeihez elengedhetetlen, hogy pontosan megértsük az osztályok, az objektumok és a referenciák közötti különbségeket. Az osztály csupán egy részletes tervrajz, amely leírja, hogy a típus példánya milyen formában jelenik meg a memóriában. Az osztályokat természetesen a forráskód fájljában definiáljuk (amelynek kiterjesztése a C#-ban konvencionálisan `*.cs` lesz). Vizsgáljuk meg az egyszerű `Car` osztályt, amelyet egy új, `SimpleGC` névre keresztelt parancssorialkalmazás-projektben definiálunk:

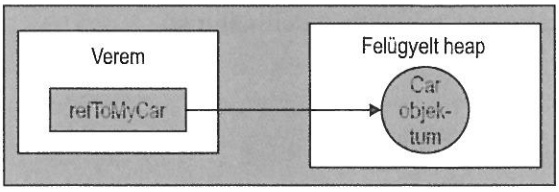
```
// Car.cs
public class Car
{
  private int currSp;
  private string petName;
```

```
  public Car(){}
  public Car(string name, int speed)
  {
    petName = name;
    currSp = speed;
  }
  public override string ToString()
  {
    return string.Format("{0} is going {1} MPH",
      petName, currSp);
  }
}
```

Az osztály definiálását követően a C# new kulcsszavának segítségével tetszőleges számú objektumot foglalhatunk le. A new kulcsszó azonban a heapen található objektum *referenciáját* adja vissza, nem pedig a tényleges objektumot. A referenciaváltozót az alkalmazásban a további felhasználáshoz a verem tárolja. Ha szeretnénk az objektumon tagokat hívni, alkalmazzuk a C# pont operátorát a tárolt referencián:

```
class Program
{
  static void Main(string[] args)
  {
    // Létrehoz egy új Car objektumot
    // a felügyelt heapben. Visszaadunk
    // egy hivatkozást erre az
    // objektumra ("refToMyCar").
    Car refToMyCar = new Car("Zippy", 50);

    // A C# pont operátor (.) használatával
    // hívtunk tagokat az objektumon,
    // a referencia változó segítségével.
    Console.WriteLine(refToMyCar.ToString());
    Console.ReadLine();
  }
}
```

A 8.1. ábrán az osztályok, az objektumok és a referenciák kapcsolata látható.



8.1. ábra: *A felügyelt heap objektumainak referenciái*

Az objektumok élettartama

C#-alkalmazások készítésekor joggal feltételezhetjük, hogy a felügyelt heap közvetlen beavatkozás nélkül is ellátja saját magát. Valójában a .NET-memóriakezelés aranyszabálya rendkívül egyszerű:

Szabály A new kulcsszó segítségével foglalhatunk helyet a felügyelt heap az objektum számára, majd el is feledkezhetünk róla.

A példányosítás után, ha már nincs rá szükség, a szemétgyűjtő megsemmisíti az objektumot. Természetesen a következő nyilvánvaló kérdés: „A szemétgyűjtő hogyan határozza meg, hogy az objektumra már nincs szükség?" A rövid (és nem teljes) válasz az, hogy a szemétgyűjtő eltávolítja a heapről az objektumot, ha az a forráskód egyetlen részéről *sem elérhető*. Tételezzük fel, hogy a Program osztályban az egyik metódus foglalja le a helyi `Car` objektumot:

```
static void MakeACar()
{
  // Ha a myCar az egyetlen referencia a Car objektumra,
  // akkor a metódus visszatértekor *megsemmisíthető*.
  Car myCar = new Car();
}
```

A `Car` referenciát (`myCar`) közvetlenül a `MakeACar()` metódusban hoztuk létre, és nem adtuk a definíció hatókörén kívülre (visszatérési érték vagy a `ref/out` paraméter segítségével). Így, ha a metódus hívása befejeződik, a `myCar` referencia nem érhető el, és a hozzátartozó `Car` objektum a szemétgyűjtés lehetséges jelöltje lesz. A `MakeACar()` metódus befejezése után a rendszer nem garantálja az objektum azonnali felszabadítását a memóriában. Ezen a ponton pusztán annyit feltételezhetünk, hogy amikor a CLR végrehajtja a következő szemétgyűjtést, a `myCar` objektum biztonsággal megsemmisíthető.

A szemétgyűjtött környezetben végzett programozás nagymértékben megkönnyíti az alkalmazásfejlesztést. Ezzel éles ellentétben a C++-programozók tisztában vannak azzal, hogy ha megfeledkeznek a heapen lefoglalt objektumok kézi törléséről, a memóriaszivárgások hamarosan jelentkeznek. A memóriaszivárgások lenyomozása az egyik legidőigényesebb (és legfárasztóbb) oldala a nemfelügyelt nyelvekkel végzett fejlesztésnek. Mivel a szemétgyűjtő feladata az objektumok megsemmisítése, a memóriakezelés terhe már nem a fejlesztőkre hárul, megvalósítása a CLR feladata.

Megjegyzés Ha rendelkezünk COM fejlesztői háttérrel, tudjuk, hogy a .NET-objektumoknak nincsen belső referenciaszámlálója, ezért a kezelt objektumok nem rendelkeznek `AddRef()` vagy `Release()` metódussal.

A new kulcsszó köztes nyelve

Amikor a C#-fordító a `new` kulcsszóval találkozik, a rendszer a CIL (Common Intermediate Language – köztes nyelv) `newobj` utasítást helyezi el a metódus megvalósításában. Ha a következő példa forráskódját szeretnénk lefordítani, és az ildasm.exe segítségével megvizsgálnánk az eredményként előállított szerelvényt, a `MakeACar()` metódusban a következő köztes nyelvi utasításokat találjuk:

```
.method public hidebysig static void MakeACar() cil managed
{
  // Code size 7 (0x7)
  .maxstack 1
  .locals init ([0] class SimpleGC.Car c)
  IL_0000: newobj instance void SimpleGC.Car::.ctor()
  IL_0005: stloc.0
  IL_0006: ret
} // a Program::MakeACar metódus vége
```

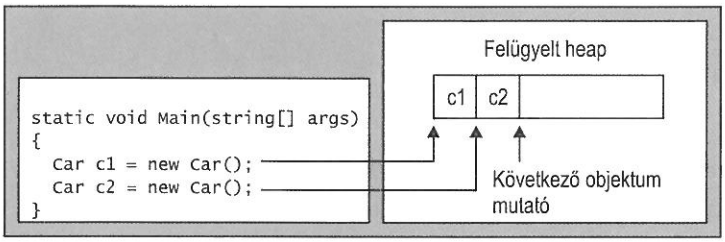
Mielőtt részletesen megnéznénk azokat a szabályokat, amelyek meghatározzák, hogy az objektumot mikor lehet eltávolítani a felügyelt heapről, ismerkedjünk meg közelebbről a CIL `newobj` utasításának szerepével. Először is a felügyelt heap nem pusztán a CLR által hozzáférhető véletlenszerűen kiválasztható memóriadarab. A .NET-szemétgyűjtő gondos házvezetőnő módjára karbantartja a heapet, ugyanis az optimalizálás érdekében (szükség esetén) tömöríti az üres memóriablokkokat. Ezért a felügyelt heap fenntart egy mutatót (*következőobjektum-mutató* vagy *újobjektum-mutató* néven ismert), amely pontosan azonosítja azt a helyet, ahol a rendszer a következő objektumot lefoglalja.

A `newobj` utasítás a következő alapvető feladatok végrehajtását kéri a CLR-től:

- Számolja ki az objektum lefoglalásához szükséges memória méretét (a típus adattagja és ősosztályai számára szükséges memóriát is beleértve).
- Ellenőrizze a felügyelt heapet, hogy vajon tényleg elegendő hely áll a rendelkezésre a lefoglalt objektum tárolásához. Ha elég memória áll a rendelkezésre, a rendszer meghívja a típus konstruktorát, és a hívó az új objektum referenciáját kapja vissza. Az objektum címe éppen megegyezik a következőobjektum-mutató utolsó pozíciójával.

- Mielőtt visszaadná a referenciát a hívónak, a felügyelt heap következő rendelkezésre álló helyére állítja át a következőobjektum-mutatót.

A folyamat a 8.2. ábrán látható.



8.2. ábra: *Az objektumok lefoglalása a felügyelt heap*

Minthogy az alkalmazás objektumok lefoglalását hajtja végre, a felügyelt heap megtelhet. A `newobj` utasítás feldolgozásakor, ha a CLR megállapítja, hogy a felügyelt heapen nincs a kért típus lefoglalásához elegendő memória, a szemétgyűjtés végrehajtásával szabadítja fel a memória egy részét. A szemétgyűjtés következő szabálya tehát szintén egyszerű:

Szabály Ha a felügyelt heapen nem áll rendelkezésre a kért objektum lefoglalásához elegendő memória, a rendszer szemétgyűjtést végez.

A szemétgyűjtés során a szemétgyűjtő *minden aktív szálat* felfüggeszt az *aktuális folyamban*, így biztosítja azt, hogy az alkalmazás a szemétgyűjtés közben ne férjen hozzá a heaphez. A szálakat a végrehajtható programon belüli futási útvonalaknak tekintjük (a szálakat lásd a 18. fejezetben). A szemétgyűjtési ciklus befejezését követően a felfüggesztett szálak folytathatják a munkájukat. Szerencsére a .NET-szemétgyűjtés erősen optimalizált; az alkalmazás működésekor ritkán (vagy szinte soha sem) vesszük észre ezt a rövid megszakítást.

Objektumreferenciák null értékre állítása

Ha Visual Basic 6.0-val már készítettünk COM-objektumokat, tudhatjuk, hogy használatuk befejeztével ajánlatos a referenciákat `Nothing` értékre állítani. A háttérben a rendszer eggyel csökkenti a COM-objektum referencia-számlálóját, és eltávolítja az objektumot a memóriából, ha a referencia-számláló értéke 0. Hasonlóképpen, a C/C++ programozók gyakran állítják a mutatóváltozókat `null` értékre, így biztosítják, hogy azok ne hivatkozzanak nem felügyelt memóriaterületre.

Vajon a C# nyelvben az objektumreferenciáknál a `null` érték hozzárendelésének mi lesz a végeredménye? Tételezzük fel, hogy a `MakeACar()` szubrutin a következőképpen módosult:

```
static void MakeACar()
{
  Car myCar = new Car();
  myCar = null;
}
```

Ha az objektumreferenciákhoz `null` értéket rendelünk, a fordító köztes nyelvi kódot generál, amely biztosítja, hogy a hivatkozás (ebben az esetben a `myCar`) nem mutat semmilyen objektumra. Ha az ildasm.exe segítségével ismét megnéznénk a módosított `MakeACar()` köztes nyelvi kódját, láthatnánk, hogy az `ldnull` műveleti kódot (`null` értéket helyez a virtuális végrehajtási veremre) az stloc.0 műveleti kód követi (beállítja a `null` referenciát a lefoglalt `Car` objektumon):

```
.method private hidebysig static void MakeACar() cil managed
{
  // Code size       10 (0xa).
  .maxstack 1
  .locals init ([0] class SimpleGC.Car myCar)
  IL_0000: nop
  IL_0001: newobj instance void SimpleGC.Car::.ctor()
  IL_0006: stloc.0
  IL_0007: ldnull
  IL_0008: stloc.0
  IL_0009: ret
} // a Program::MakeACar metódus vége.
```

A referencia `null` értékre állítása nem kényszeríti a rendszert a szemétgyűjtés azonnali elindítására, valamint az objektum eltávolítására a heapről. Az egyetlen, amit elérhetünk, hogy explicit módon megszüntetjük a referencia és a referencia által korábban mutatott objektum közötti kapcsolatot. Ennek értelmében a C# nyelvben a referenciák `null` értékre állítása nem jár olyan következményekkel, mint más C-alapú nyelvekben (vagy VB 6.0-nál), a művelettel nem okozhatunk kárt.

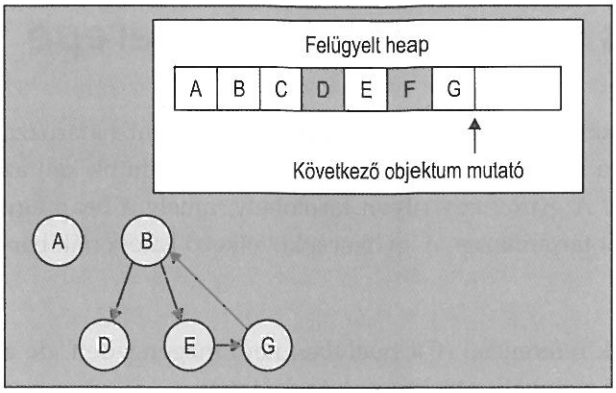
Az alkalmazás gyökérelemének szerepe

Térjünk vissza ahhoz a kérdéshez, hogy a szemétgyűjtő miként határozza meg azt, ha az objektumra már „nincsen szükség". Ehhez ismernünk kell az *alkalmazásgyökér* fogalmát. A *gyökér* egy olyan tárolóhely, amely a heap egy objektumának referenciáját tartalmazza. A gyökér a következő kategóriák bármelyikébe tartozhat:

- globális objektumok referenciái (C# nyelvben nem megengedett, de a CIL kód megengedi a globális objektumok foglalását);
- statikus objektumok/statikus mezők referenciái;
- az alkalmazás forráskódján belüli lokális objektumok referenciái;
- metódusnak átadott objektumparaméterek referenciái;
- *véglegesítésre* váró objektumok referenciái (lásd később);
- bármely CPU-regiszter, amely objektumra hivatkozik.

A szemétgyűjtési folyamat során a futtatórendszer megvizsgálja a felügyelt heap objektumait, és meghatározza, hogy azok továbbra is elérhetők-e (azaz rendelkeznek-e *gyökerrel*) az alkalmazás számára. Ennek érdekében a CLR *objektumgráfot* épít, amelyben megjelenik a heapen elérhető összes objektum. Az objektumgráfok dokumentálják az elérhető objektumokat. A szemétgyűjtő minden objektumot pontosan egyszer ábrázol a gráfban, így elkerülhetők a COM-programozásban ismert körkörösreferencia-számlálás kellemetlen következményei.

Tételezzük fel, hogy a felügyelt heap az A, B, C, D, E, F és G objektumkészletet tartalmazza. A szemétgyűjtés során a rendszer megvizsgálja az objektumokat (valamint az objektumok esetleges belső referenciáit), és aktív gyökereket keres. Ha elkészült a gráf, az elérhetetlen objektumokat (ebben az esetben az C és az F objektumot) a rendszer szemétként megjelöli. Az ismertetett példa lehetséges objektumgráfja a 8.3. ábrán látható (a nyilakat az „X objektum Y-tól függ" vagy a „szükséges" szavakkal értelmezhetjük, például: „E objektum G-től függ, indirekt módon pedig B-től", „A objektum semmitől sem függ" és így tovább).



8.3. ábra: *Az objektumgráfok készítésével meghatározható, hogy melyek az alkalmazásgyökér számára elérhető objektumok*

Ha a rendszer szemétgyűjtésre alkalmasnak jelöli az objektumot (ebben az esetben a C és az F objektumokat — hiszen a gráf nem tartalmazza őket), akkor ezeket a megjelölt objektumokat kisöpri a memóriából. Ekkor a heapen fennmaradó helyet a rendszer tömöríti, ennek hatására a CLR pedig frissíti az aktívalkalmazás-gyökereket (és a mögöttes mutatókat), hogy az aktuális memóriahelyekre mutassanak (ezt a művelet automatikusan és a felhasználó számára láthatatlanul megy végbe). Végül, de nem utolsósorban a következőobjektum-mutatót a rendszer a következő rendelkezésre álló helyre állítja. A 8.4. ábrán az újrabeállítás eredménye látható.

8.4. ábra: *Tiszta és tömörített heap*

Megjegyzés A szemétgyűjtő tulajdonképpen két különböző heapet használ, amelyek egyike a kimondottan nagy objektumok tárolására alkalmas. Ezt a területet a gyűjtési ciklusban ritkán használja a rendszer, hiszen a nagy objektumok áthelyezése negatívan befolyásolja a teljesítményt. Ettől függetlenül a „felügyelt heapet" egyetlen memóriaterületnek tekinthetjük.

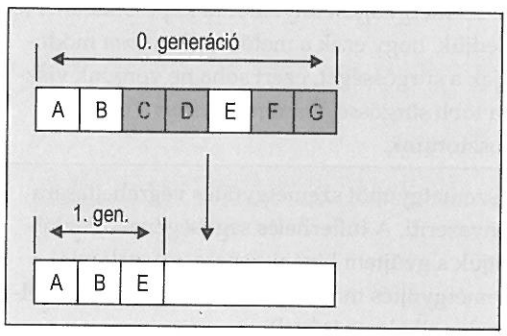
Az objektumgenerációk

Amikor a CLR megpróbálja megkeresni az elérhetetlen objektumokat, *nem* vizsgálja meg egyesével a felügyelt heapen elhelyezett összes objektumot. A művelet nyilvánvalóan hosszú ideig tartana, különösen nagyobb (például életszerű) alkalmazások esetén.

A folyamat optimalizálásának érdekében a heapen tárolt objektumok mindegyike meghatározott „generációhoz" tartozik. A generációk mögötti ötlet rendkívül egyszerű: minél hosszabb ideje létezik az objektum a heapen, annál valószínűbb, hogy ott is marad. A `Main()` metódust megvalósító objektum például a program befejezéséig a memóriában található. Ellenben a heapen rövid ideje elhelyezett objektumok (pl. a metódus hatókörén belül lefoglalt objektumok) valószínűleg hamar el nem érhetővé válnak. Ezen feltételek mellett a heapen található objektumok a következő generációk egyikéhez tartoznak:

- *0. generáció:* Újonnan foglalt objektumot azonosít, amelyet a rendszer még soha nem jelölt szemétgyűjtésre.
- *1. generáció:* Olyan objektumot azonosít, amely túlélte a 0. generációs szemétgyűjtést.
- *2. generáció:* Olyan objektumot azonosít, amely egynél több szemétgyűjtési ciklust túlélt.

A szemétgyűjtő először mindig a 0. generációs objektumokat vizsgálja meg. Nézzük meg az objektumgenerációk szemétgyűjtésre gyakorolt hatását a 8.5. ábra segítségével, amely bemutatja, hogy a túlélő 0. generációs objektumok (A, B és E) hogyan lépnek előre, miután a rendszer a szükséges memóriaterületet felszabadította.



8.5. ábra: *A szemétgyűjtést túlélő 0. generációs objektumok 1. generációs objektumokká lépnek elő*

Ha az összes 0. generációs objektumot kiértékelte a rendszer, és további memóriaterületre van szükség, megvizsgálja az 1. generációs objektumok „elérhetőségét", és ennek megfelelően gyűjti őket. Az 1. generációs túlélő objektumok 2. generációssá lépnek elő. Ha a szemétgyűjtőnek *még mindig* további memóriára van szüksége, a rendszer megvizsgálja a 2. generációs objektumok elérhetőségét. Ha a 2. generációs objektum túléli a szemétgyűjtést, az objektumgenerációk előre definiált felső határának köszönhetően 2. generációs objektum marad.

A lényeg az, hogy az objektumgenerációk hozzárendelésével a heapen az újabb objektumokat (pl. a lokális változókat) hamar kisöpri a rendszer, míg az idősebb objektumokat (pl. a programok alkalmazásobjektumait) nem „piszkálja" olyan sűrűn.

A System.GC típus

Az alaposztálykönyvtárak biztosítják a `System.GC` osztálytípust, amely lehetővé teszi, hogy statikus tagok segítségével programozottan együttműködjünk a szemétgyűjtővel. Ritkán (vagy akár soha nem) lesz szükségünk erre a forráskódunkban. Általában kizárólag akkor használjuk a `System.GC` típus tagjait, amikor *nem felügyelt erőforrásokat* alkalmazó típusokat hozunk létre. A 8.1. táblázatban a gyakrabban alkalmazott tagok rövid összefoglalását találjuk (a teljes referenciát a .NET-keretrendszer 3.5 SDK dokumentációja tartalmazza).

System.GC tag	Gyakorlati jelentőség
`AddMemoryPressure()` `RemoveMemoryPressure()`	Lehetővé teszi numerikus érték definiálását, amely a hívóobjektum „sürgősségi szintjét" jelöli a szemétgyűjtési folyamattal kapcsolatban. Ne feledjük, hogy ezek a metódusok *párban* módosítják a sürgősséget, ezért soha ne vonjunk vissza több sürgősséget, mint amennyit összesen kiosztottunk.
`Collect()`	A szemétgyűjtőt szemétgyűjtés végrehajtására kényszeríti. A túlterhelés segítségével definiálhatjuk a gyűjteni kívánt generációt, valamint a szemétgyűjtés módját (a `GCCollectionMode` felsorolás alkalmazásával).

System.GC tag	Gyakorlati jelentőség
`CollectionCount()`	Numerikus értéket ad vissza, amely jelzi, hogy a rendszer az adott generációt hányszor söpörte ki a memóriából.
`GetGeneration()`	Visszaadja azt, hogy az objektum pillanatnyilag melyik generációhoz tartozik.
`GetTotalMemory()`	A felügyelt heapen pillanatnyilag lefoglalt memória becsült méretét adja vissza (byte-ban). A logikai paraméter meghatározza, hogy a hívás a visszatérés előtt várjon a szemétgyűjtés végrehajtására vagy sem.
`MaxGeneration`	A célrendszerben támogatott maximum generációk számát adja vissza. A Microsoft .NET 3.5-ös rendszerekben három lehetséges generáció létezik (0, 1, és 2).
`SuppressFinalize()`	Beállít egy jelzőt, amely azt mutatja, hogy a definiált objektum `Finalize()` metódusát nem lehet meghívni.
`WaitForPendingFinalizers()`	Felfüggeszti az aktuális szálat, amíg a rendszer a véglegesíthető objektumokat véglegesíti. A metódust rendszerint közvetlenül a `GC.Collect()` tag indítása után hívjuk meg.

8.1. táblázat: *A System.GC típus tagjai*

A következő `Main()` metódus bemutatja, hogy a `System.GC` típust hogyan lehet különböző szemétgyűjtésre vonatkozó részletek kinyerésére használni. A metódus a `GC` típus több tagját is alkalmazza:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with System.GC *****");

  // Kiírja a bájtok becsült számát a heapben.
  Console.WriteLine("Estimated bytes on heap: {0}",
    GC.GetTotalMemory(false));

  // A MaxGeneration nulla alapú, úgyhogy adjunk hozzá egyet
  // a megjelenítéshez.
  Console.WriteLine("This OS has {0} object generations.\n",
    (GC.MaxGeneration + 1));
```

```
    Car refToMyCar = new Car("Zippy", 100);
    Console.WriteLine(refToMyCar.ToString());

    // Kiírja a refToMyCar objektum generációját.
    Console.WriteLine("Generation of refToMyCar is: {0}",
      GC.GetGeneration(refToMyCar));

    Console.ReadLine();
  }
```

A szemétgyűjtés kikényszerítése

A .NET-szemétgyűjtés célja, hogy megoldja helyettünk a memóriakezelés feladatát. Néhány különleges helyzetben rendkívül hasznos lehet, hogy a `GC.Collect()` segítségével programozottan kényszeríthetjük a rendszert a szemétgyűjtésre. Ha, például:

- az alkalmazás futtatása olyan kódblokkhoz érkezik, amelynek végrehajtását nem szeretnénk egy esetleges szemétgyűjtéssel megszakítani;
- az alkalmazás éppen rendkívül sok objektum lefoglalásával végzett, és a lehető legtöbb lefoglalt memóriát szeretnénk törölni.

Ha úgy határozunk, hogy érdemes lenne a szemétgyűjtővel ellenőriztetni az elérhetetlen objektumokat, explicit módon indíthatunk szemétgyűjtést:

```
static void Main(string[] args)
{
  ...
  // Elindítja a szemétgyűjtést, és vár arra,
  // hogy minden objektum véglegesítve legyen.
  GC.Collect();
  GC.WaitForPendingFinalizers();
  ...
}
```

Amikor manuálisan kényszerítjük a rendszert szemétgyűjtésre, mindig meg kell hívnunk a `GC.WaitForPendingFinalizers()`-t. Ezzel a megközelítéssel biztosak lehetünk abban, hogy a program végrehajtásának folytatása előtt *minden véglegesíthető objektumnak* lehetősége nyílott a szükséges takarítás végrehajtására. A háttérben a gyűjtési folyamat során a `GC.WaitForPendingFinalizers()` felfüggeszti a „hívószálat". Ez a lehetőség biztosítja azt, hogy a forráskód ne indítson el semmilyen metódust olyan objektumon, amelyet a rendszer éppen megsemmisít.

A `GC.Collect()` metódusnak numerikus értéket adhatunk, így azonosíthatjuk a legidősebb generációt, amelyen a rendszer szemétgyűjtést végezhet. Ha például azt szeretnénk, hogy a CLR csak 0. generációs objektumokat vizsgáljon, a következő kódot alkalmazhatjuk:

```
static void Main(string[] args)
{
  ...
  // Csak a 0. generációs objektumok vizsgálata.
  GC.Collect(0);
  GC.WaitForPendingFinalizers();
  ...
}
```

A `Collect()` metódus átadható a `GCCollectionMode` felsorolás egy értékének második paraméterként, ebből következően finomíthatjuk, hogy a futtatórendszer pontosan hogyan indítsa el a szemétgyűjtést. A felsorolás a következő értékeket határozza meg:

```
public enum GCCollectionMode
{
  Default,    // Forced a jelenlegi alapértelmezett.
  Forced,     // Kikényszeríti az azonnali szemétgyűjtést!

  Optimized   // Lehetővé teszi, hogy a futtatókörnyezet
              // határozza meg, hogy az aktuális időpont
              // alkalmas-e az objektumok újrakérésére.
}
```

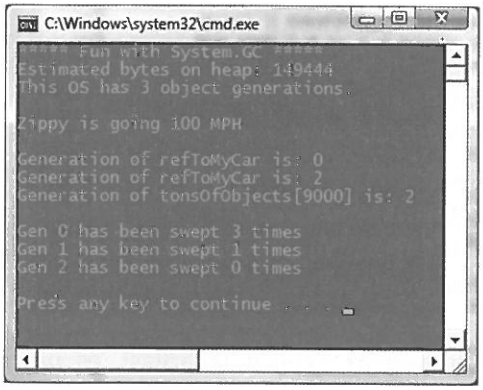
A többi szemétgyűjtéshez hasonlóan a `GC.Collect()` metódus hívása előlépteti a túlélő generációkat. Ennek bemutatására tételezzük fel, hogy a `Main()` metódus a következőképpen módosult:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with System.GC *****");

  // Kiírja a bájtok becsült számát a heapben.
  Console.WriteLine("Estimated bytes on heap: {0}",
    GC.GetTotalMemory(false));

  // A MaxGeneration nulla alapú.
  Console.WriteLine("This OS has {0} object generations.\n",
    (GC.MaxGeneration + 1));

  Car refToMyCar = new Car("Zippy", 100);
  Console.WriteLine(refToMyCar.ToString());
```

```
    // Kiírja a refToMyCar generációját.
    Console.WriteLine("\nGeneration of refToMyCar is: {0}",
      GC.GetGeneration(refToMyCar));

    // Létrehoz rengeteg objektumot tesztelési célokra.
    object[] tonsOfObjects = new object[50000];
    for (int i = 0; i < 50000; i++)
      tonsOfObjects[i] = new object();

    // Csak a 0. generációs objektumok összegyűjtése.
    GC.Collect(0, GCCollectionMode.Forced);
    GC.WaitForPendingFinalizers();

    // Kiírja a refToMyCar generációját.
    Console.WriteLine("Generation of refToMyCar is: {0}",
      GC.GetGeneration(refToMyCar));

    // Megnézi, hogy a tonsOfObjects[9000] még működik-e.
    if (tonsOfObjects[9000] != null)
    {
      Console.WriteLine("Generation of tonsOfObjects[9000] is: {0}",
        GC.GetGeneration(tonsOfObjects[9000]));
    }
    else
      Console.WriteLine("tonsOfObjects[9000] is no longer alive.");

    // Kiírja, hogy a generáció hányszor lett kitakarítva.
    Console.WriteLine("\nGen 0 has been swept {0} times",
      GC.CollectionCount(0));
    Console.WriteLine("Gen 1 has been swept {0} times",
      GC.CollectionCount(1));
    Console.WriteLine("Gen 2 has been swept {0} times",
      GC.CollectionCount(2));
    Console.ReadLine();
  }
```

Tesztelés céljából szándékosan nagy objektumtípustömböt hoztunk létre (egészen pontosan 50000 eleműt). Mint az a 8.6. ábrán bemutatott kimenetből látható, annak ellenére, hogy a `Main()` metódus csak egyetlen kifejezett szemétgyűjtési kérést indított el (a `GC.Collect()` metódussal), a háttérben a CLR mégis többször végzett szemétgyűjtést.

A továbbiakban részletesebben foglalkozunk szemétgyűjtési folyamattal, továbbá megvizsgáljuk, hogy hogyan *készíthetünk véglegesíthető objektumokat*, valamint *eldobható objektumokat*. A következő módszereket csak akkor tudjuk hasznosítani, ha olyan felügyelt osztályokat készítünk, amelyek belső nem felügyelt erőforrásokat tartanak karban.



8.6. ábra: *Együttműködés a CL- szemétgyűjtővel a System.GC segítségével*

Forráskód A `SimpleGC` projekt a 8. fejezet alkönyvtárában található.

Véglegesíthető objektumok készítése

A .NET `System.Object` mint az egyik legfontosabb ősosztály definiálja a `Finalize()` virtuális metódust (lásd 6. fejezet). A metódus alapértelmezett megvalósítása voltaképpen semmit nem csinál:

```
// System.Object
public class Object
{
  ...
  protected virtual void Finalize() {}
}
```

Amikor az egyedi osztályaink számára felüldefiniáljuk a `Finalize()` metódust, tulajdonképpen a típus szükségszerű takarításának hozunk létre meghatározott helyet. Mivel a tagot védettként definiáltuk, a pont operátor segítségével az osztálypéldányból nem lehet közvetlenül meghívni egy objektum `Finalize()` metódusát. Ehelyett, mielőtt a memóriából törölné az objektumot, a *szemétgyűjtő hívja* meg az objektum `Finalize()` metódusát (ha a művelet támogatott).

Megjegyzés Struktúratípusokon a `Finalize()` metódus felüldefiniálására illegális művelet. Ez azért fontos, mert a struktúrák értéktípusok, amelyeket nem a heapen foglalunk le, így a rendszer a struktúrákon soha nem hajt végre szemétgyűjtést.

A rendszer egy automatikus szemétgyűjtéskor vagy a `GC.Collect()` segítségével programozottan végzett gyűjtéskor (végül) meghívja a `Finalize()` metódust. A típus véglegesítő metódusát a rendszer automatikusan meghívja, amikor az alkalmazásunkat hosztoló *alkalmazástartományt* eltávolítja a memóriából. Az alkalmazástartományok hosztolják a végrehajtható szerelvényeket és a szükséges külső kódkönyvtárakat (ezt lásd a 17. fejezetben). Amikor a rendszer eltávolítja a memóriából az alkalmazástartományt, a CLR automatikusan elindítja a véglegesítőket az életciklusa során létrehozott a véglegesíthető objektumok számára.

A *legtöbb* C#-osztály nem követeli meg az explicit takarítást, és nincs szüksége egyedi véglegesítőre. Ennek egyszerű az oka: ha a típusok más felügyelt objektumokat alkalmaznak, a rendszer végül minden összetevőn szemétgyűjtést hajt végre. Csak akkor van szükség olyan osztály tervezésére, amely saját maga után takarítást végez, ha *nem felügyelt erőforrásokat* (pl. nyers OS-fájl-azonosítókat, nyers nem felügyelt adatbázis kapcsolatokat, nem felügyelt memóriaterületet vagy egyéb nem felügyelt erőforrást) használunk. A .NET-platformon a nem felügyelt erőforrásokhoz a Platform Invocation Services (`PInvoke`) révén közvetlenül az operációs rendszer API-hívásával vagy valamilyen körülményes COM együttműködési eljárás alkalmazásával férhetünk hozzá. Ennek tükrében a szemétgyűjtés szabálya a következő:

Szabály A `Finalize()` metódus felüldefiniálásának egyetlen oka, hogy használjuk vagy a C#-osztályunk nem felügyelt erőforrásait a `PInvoke` segítségével, vagy a bonyolult COM együttműködési feladatokat (rendszerint a `System.Runtime.InteropServices.Marshal` típus különböző tagjait).

A System.Object.Finalize() felüldefiniálása

Ritkán, amikor nem felügyelt erőforrásokat alkalmazó C#-osztályokat készítünk, nyilvánvalóan szeretnénk megbizonyosodni arról, hogy a rendszer a mögöttes memóriát prediktív módon szabadítja fel. Tételezzük fel, hogy létrehoztuk a `SimpleFinalize` nevű új C#-konzolalkalmazást, amelynek `MyResourceWrapper` osztálya nem felügyelt erőforrásokat használ (bármi legyen is az), és szeretnénk felülbírálni a `Finalize()` metódust. A C# nyelv érdekessége, hogy ezt a feladatot nem hajthatjuk végre az elvárt `override` kulcsszóval:

```
public class MyResourceWrapper
{
  // Fordítási idejű hiba!
  protected override void Finalize(){ }
}
```

Ehelyett, ha azt szeretnénk, hogy az egyedi C#-osztálytípusaink felülbírálják a `Finalize()` metódust, a (C++ nyelvhez hasonló módon) destruktorszintaxis alkalmazásával érhetjük el ugyanezt a hatást. A virtuális metódus felülbírálásának alternatív módja annak köszönhető, hogy amikor a C#-fordító a véglegesítő szintaxist dolgozza fel, automatikusan elkészíti a szükséges infrastruktúra nagy részét az implicit módon felülbírált `Finalize()` metódusban (lásd később).

A C#-véglegesítők nagyon hasonlítanak a konstruktorokhoz, mivel annak az osztálynak a nevét kapják, amelyben definiáljuk őket. Ezenkívül a véglegesítőket a rendszer hullámjellel (~) látja el. A konstruktoroktól eltérően a véglegesítők soha nem rendelkeznek sem hozzáférés-módosítóval (a rendszer implicit módon védi őket), sem paraméterekkel, illetve nem lehet őket túlterhelni (osztályonként egy véglegesítő engedélyezett).

A következő példában a `MyResourceWrapper` egyedi véglegesítőjét láthatjuk, amely indításakor sípoló hangjelzést ad, nyilvánvalóan csak bemutató jelleggel. Az életszerű véglegesítő nem tesz mást, mint felszabadítja a nem felügyelt erőforrásokat, és főleg *nem* működik együtt más felügyelt objektumokkal, még azokkal sem, amelyekre az aktuális objektum hivatkozik, minthogy nem lehetünk biztosak abban, hogy azok még mindig élnek, amikor a szemétgyűjtő elindítja a `Finalize()` metódusunkat:

```
// A System.Object.Finalize() felüldefiniálása a véglegesítő
// szintaxissal.
class MyResourceWrapper
{
  ~MyResourceWrapper()
  {
    // Itt kitakarítja a nem felügyelt erőforrásokat.

    // Sípol egyet, amikor kész (csak tesztelési célokra!)
    Console.Beep();
  }
}
```

Ha az ildasm.exe segítségével megvizsgálnánk ezt a C#-destruktort, láthatnánk, hogy a fordító beillesztette a szükséges hibaellenőrzési kódot. Először a `Finalize()` metódus hatókörében található forráskódutasításokat helyezi a rendszer a `try` blokkba (lásd a 7. fejezetet). A kapcsolódó `finally` blokk biztosítja, hogy az ősosztályaink `Finalize()` metódusait a rendszer mindig végrehajtja attól függetlenül, hogy a `try` hatókörében bekövetkezett-e kivétel, avagy sem:

```
.method family hidebysig virtual instance void
  Finalize() cil managed
{
  // Code size 13 (0xd)
  .maxstack 1
  .try
  {
    IL_0000: ldc.i4 0x4e20
    IL_0005: ldc.i4 0x3e8
    IL_000a: call
    void [mscorlib]System.Console::Beep(int32, int32)
    IL_000f: nop
    IL_0010: nop
    IL_0011: leave.s IL_001b
  } // .try vége

  finally
  {
    IL_0013: ldarg.0
    IL_0014:
    call instance void [mscorlib]System.Object::Finalize()
    IL_0019: nop
    IL_001a: endfinally
  } // végkezelő
  IL_001b: nop
  IL_001c: ret
} // a MyResourceWrapper::Finalize metódus vége
```

Ha tesztelnénk a `MyResourceWrapper` típust, azt tapasztalhatnánk, hogy az alkalmazás befejezése után a számítógép hangjelzést ad, ugyanis a CLR az alkalmazástartomány leállításakor automatikusan elindítja a véglegesítőket:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Finalizers *****\n");
  Console.WriteLine("Hit the return key to shut down this app");
  Console.WriteLine("and force the GC to invoke Finalize()");
  Console.WriteLine("for finalizable objects created in this
    AppDomain.");
  Console.ReadLine();
  MyResourceWrapper rw = new MyResourceWrapper();
}
```

Forráskód A `SimpleFinalize` projekt a 8. fejezet alkönyvtárában található.

A véglegesítő folyamat részletei

A `Finalize()` metódus biztosítja tehát, hogy a .NET-objektum a szemétgyűjtéskor felszabadíthatja a nem felügyelt erőforrásokat. Ha olyan típust készítünk, amely nem használ nem felügyelt entitásokat (ez a leggyakoribb eset), a véglegesítésnek nincsen sok haszna. Úgy kell voltaképp megterveznünk a típusainkat, hogy ne támogassuk a `Finalize()` metódust, hiszen a véglegesítés időbe telik.

Ha a felügyelt heapen foglaljuk le az objektumot, a futtatórendszer automatikusan meghatározza, hogy az objektumunk egyedi `Finalize()` metódust támogat-e. Ha igen, akkor az objektumot *véglegesíthetőnek* jelöli a rendszer, és egy belső sorban, a *véglegesítő sorban* tárolt mutatót hoz létre az objektumhoz. A véglegesítő sor a szemétgyűjtő által karbantartott olyan táblázat, amely a heapről az eltávolításuk előtt véglegesítésre váró objektumokra mutat.

Amikor a szemétgyűjtő úgy dönt, hogy elérkezett az idő az objektum felszabadítására a memóriában, megvizsgálja a véglegesítő sor bejegyzéseit, és az objektumot a heapről egy másik felügyelt struktúrába, a *véglegesítéselérési táblázatba* másolja. Ekkor egy külön szál indul el, és a véglegesítési elérhető tábla összes objektumára meghívja a `Finalize()` metódust *a következő szemétgyűjtés* során. Ennek köszönhetően legalább *két* szemétgyűjtés végrehajtása szükséges az objektumok véglegesítéséhez.

A lényeg az, hogy amíg az objektum véglegesítése biztosítja, hogy a nem felügyelt erőforrások felszabadulhassanak, a véglegesítés mégsem determinisztikus jellegű, és a háttérben zajló feldolgozások következtében jelentősen lassabb.

Eldobható objektumok készítése

A véglegesítők segítségével a szemétgyűjtő indítása előtt felszabadíthatjuk a nem felügyelt erőforrásokat. Mivel több nem felügyelt objektum „drága elem" (pl. az adatbázis és a fájlazonosítók), érdemes a lehető leghamarabb felszabadítani őket, és nem a szemétgyűjtés megkezdésére várni. A `Finalize()` metódus felülbírálása mellett a másik lehetőség az, ha az osztály az `IDisposable` interfészt valósítja meg, amely egyedül a `Dispose()` metódust definiálja:

```
public interface IDisposable
{
  void Dispose();
}
```

(Az interfészalapú programozáshoz lásd a 9. fejezetet.) Röviden: az interfész az osztály vagy a struktúra által támogatott absztrakt tagok gyűjteménye. Amikor az `IDisposable` interfészt támogatjuk, azt feltételezzük, hogy amikor az *objektumfelhasználó* többé már nem használja az objektumot, manuálisan meghívja a `Dispose()` metódust, mielőtt engedné, hogy az objektumreferencia kiessen a metódus hatóköréből. Így az objektum elvégezheti a nem felügyelt erőforrások szükséges takarítását anélkül, hogy a véglegesítő sorra kerülne, és várnia kellene, amíg a szemétgyűjtő elindítja az osztály véglegesítését.

Megjegyzés Mind a struktúrák, mind az osztálytípusok megvalósíthatják az `IDisposable` interfészt (nem úgy mint a `Finalaze()` metódus, amelyet csak az osztályok definiálhatnak felül), ugyanis az objektumfelhasználó (nem a szemétgyűjtő) indítja el a `Dispose()` metódust.

Az interfész használatának bemutatásához hozzuk létre a `SimpleDispose` nevű C#-konzolalkalmazást. A módosított `MyResourceWrapper` osztály, amely megvalósítja az `IDisposable` interfészt ahelyett, hogy felüldefiniálná a `System.Object.Finalize()` metódust, a következő:

```
// Az IDisposable megvalósítása.
public class MyResourceWrapper : IDisposable
{
  // Az objektum használójának meg kell hívnia ezt a metódust,
  // ha befejezte az objektum használatát.
  public void Dispose()
  {
    // A nem felügyelt erőforrások kitakarítása...

    // Más tartalmazott eldobható objektumok eldobása...

    // Csak tesztképpen.
    Console.WriteLine("***** In Dispose! *****");
  }
}
```

A `Dispose()` metódus nem csupán a típus nem felügyelt erőforrásait szabadítja fel, de az alkalmazásba foglalt többi eldobható metódusban is meghívja a `Dispose()` metódust. A `Finalize()` metódustól eltérően a `Dispose()` metódusban biztonságosan kommunikálhatunk más felügyelt objektumokkal. Ennek egyszerű az oka: a szemétgyűjtő nem ismeri az `IDisposable` interfészt, és soha nem hívja a `Dispose()` metódust. Ezért amikor az objektumfelhasználó meghívja ezt a metódust, az objektum még mindig tevékeny életet él a felügyelt heapen, és hozzáfér a heapen lefoglalt objektumokhoz. A hívás lényegre törő:

```
public class Program
{
  static void Main()
  {
    Console.WriteLine("***** Fun with Dispose *****\n");

    // Eldobható objektum létrehozása és a Dispose() metódus
    // meghívása beágyazott erőforrások felszabadításához.
    MyResourceWrapper rw = new MyResourceWrapper();
    rw.Dispose();
    Console.ReadLine();
  }
}
```

Mielőtt az objektumon meghívjuk a `Dispose()` metódust, biztosítanunk kell azt, hogy a típus támogassa az `IDisposable` interfészt. A .NET-keretrendszer 3.5 SDK dokumentációjának segítségével megkereshetjük, hogy melyik alaposztálykönyvtár-típusok valósítják meg az `IDisposable` interfészt; programozottan ugyanezt a 6. fejezetben megismert „`is`" vagy „`as`" kulcsszavakkal ellenőrizhetjük:

```
public class Program
{
  static void Main()
  {
    Console.WriteLine("***** Fun with Dispose *****\n");
    MyResourceWrapper rw = new MyResourceWrapper();
    if (rw is IDisposable)
      rw.Dispose();
    Console.ReadLine();
  }
}
```

A példa a szemétgyűjtést alkalmazó típusok újabb szabályát mutatja.

Szabály Ha az objektum támogatja az `IDisposable` interfészt, a közvetlenül létrehozott objektumokon mindig hívjuk meg a `Dispose()` metódust. Ha az osztály tervezője a `Dispose()` metódus támogatása mellett döntött, a típus valamilyen módon takarítást fog végezni.

A fenti szabálynak egyetlen buktatója van. Az alaposztálykönyvtárakban sok típus, amely megvalósítja az `IDisposable` interfészt, egy álnevet biztosít (kissé zavaróan) a `Dispose()` metódus számára, ugyanis így próbál az eldobással kapcsolatos/eldobás jellegű metódusnév helyett valamilyen – a definiáló típus szempontjából kifejezőbb – metódusnevet biztosítani.

Nézzünk egy példát: a `System.IO.FileStream` osztály megvalósítja az `IDisposable` interfészt (tehát támogatja a `Dispose()` metódust), és *egyben* definiálja a `Close()` metódust, amely ugyanezt a célt szolgálja:

```
// Tegyük fel, hogy importáltuk
// a System.IO névteret...
static void DisposeFileStream()
{
  FileStream fs = new FileStream("myFile.txt",
    FileMode.OpenOrCreate);

  // Mit ne mondjak, zavaró!
  // Ezek a metódusmeghívások ugyanazt csinálják!
  fs.Close();
  fs.Dispose();
}
```

Míg természetesebbnek tűnik „lezárni", nem pedig „eldobni" a fájlt, az eldobás jellegű metódusok megkettőzése zavaró. Abban a néhány típusban, amelyek álnevet biztosítanak, ha a típus megvalósítja az `IDisposable` interfészt, a `Dispose()` metódus hívása minden esetben számít helyes gyakorlatnak.

A C# using kulcsszó egy újabb alkalmazása

Amikor az `IDisposable` interfészt megvalósító felügyelt objektummal dolgozunk, gyakran alkalmazunk strukturált kivételkezelést annak biztosítására, hogy a rendszer futásidejű kivétel esetén a típus `Dispose()` metódusát hívja:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");
  MyResourceWrapper rw = new MyResourceWrapper ();
  try
  {
    // Használjuk az rw tagjait.
  }
  finally
  {
    // Akár van hiba, akár nincs, mindig hívjuk meg
    // a Dispose() metódust.
    rw.Dispose();
  }
}
```

Ez a defenzív programozás kiváló példája, de igazság szerint kevéssé érdekes az összes eldobható típus becsomagolásának lehetősége a `try/finally` blokkban annak érdekében, hogy a rendszer meghívja a `Dispose()` metódust. Ugyanezt az eredményt másképp is elérhetjük a C# speciális szintaxisával:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Dispose *****\n");
  // A Dispose() metódus automatikusan meghívásra kerül, amikor a
  // használó hatókör kilép.
  using(MyResourceWrapper rw = new MyResourceWrapper())
  {
    // rw objektum használata.
  }
}
```

Ha az `ildasm.exe` segítségével megtekintenénk a `Main()` metódus köztes nyelvi kódját, láthatnánk, hogy a `using` szintaxis valóban kibővíti a `try/final` logikát, a várt `Dispose()` hívással együtt:

```
.method private hidebysig static void
  Main(string[] args) cil managed
{
  ...
  .try
  {
    ...
  } // .try vége
  finally
  {
    ...
    IL_0012: callvirt instance void
      SimpleFinalize.MyResourceWrapper::Dispose()
  } // Végkezelő.
  ...
} // Program::Main metódus vége
```

Megjegyzés Ha olyan objektumot próbálunk „használni", amely nem valósítja meg az `IDisposable` interfészt, a fordítás során hibát kapunk.

Noha a szintaxis használatával szükségtelenné válik az eldobható objektumok manuális becsomagolása a `try/finally` logikába, a C# `using` kulcsszó sajnos kettős jelentéssel bír (névterek definiálására és a `Dispose()` metódus indítására használható). Mindamellett, amikor olyan .NET-típusokkal dolgozunk, amelyek támogatják az `IDisposable` interfészt, a szintaktikai konstrukció biztosítja, hogy az „alkalmazott" objektumban a `using` blokk végrehajtása után a rendszer automatikusan hívja a `Dispose()` metódust.

A `using` hatókörén belül több olyan objektumot deklarálhatunk, amelyeknek a *típusa megegyezik*. Elvárásainknak megfelelően a fordító a deklarált objektumokba beilleszti a `Dispose()` metódus hívását:

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

  // Vesszővel elválasztott lista használatával deklarálhatunk
  // több objektumot is eldobhatónak.
  using(MyResourceWrapper rw = new MyResourceWrapper(),
    rw2 = new MyResourceWrapper())
  {
    // rw és rw2 objektumok használata.
  }
}
```

Forráskód A `SimpleDispose` projekt a 8. fejezet alkönyvtárában található.

Véglegesíthető és eldobható típusok készítése

Az előzőekben megismertük a belső nem felügyelt erőforrások takarítását végrehajtó osztályok készítésének két különböző megközelítését. Egyrészt felülbírálhatjuk a `System.Object.Finalize()` metódust. Az objektum a szemétgyűjtés során (bármikor legyen is) felhasználói beavatkozás nélkül saját maga után takarít. Másrészt megvalósíthatjuk az `IDisposable` interfészt, amely lehetővé teszi az objektumfelhasználó számára, hogy eltakarítsa az objektumot, miután végzett a használatával. Ha azonban a hívó megfeledkezik a `Dispose()` metódus hívásáról, a nem felügyelt erőforrások a végtelenségig a memóriában maradhatnak.

A két technikát egyetlen osztálydefiníció segítségével ötvözhetjük. Így kiaknázhatjuk a két modell leghasznosabb lehetőségeit. Ha az objektumfelhasználó nem felejti el meghívni a `Dispose()` metódust, a `GC.SuppressFinalize()` metódus hívásával utasítjuk a szemétgyűjtőt, hogy kihagyhatja a véglegesítési folyamatot. Ha az objektumfelhasználó elfelejti meghívni a `Dispose()` metódust, a rendszer véglegesíti az objektumot, és lehetősége nyílik a belső erőforrások felszabadítására. Az objektum belső nem felügyelt erőforrásait a rendszer valamilyen módon felszabadítja.

A MyResourceWrapper következő iterációját, amely formalizált és eldobható, a FinalizableDisposableClass C#-konzolalkalmazás definiálja:

```
// Kifinomult erőforrás-burkoló.
public class MyResourceWrapper : IDisposable
{
  // A szemétgyűjtő meghívja ezt a metódust, ha az
  // objektum használó elfelejti meghívni a Dispose() metódust.
  ~ MyResourceWrapper()
  {
    // Kitakarít bármilyen belső nemfelügyelt erőforrást.
    // *Ne* hívjuk meg a Dispose() metódust felügyelt objektumokra.
  }

  // Az objektum használója azért hívja meg ezt a metódust, hogy
  // AZONNAL kitakarítsa az erőforrásokat.
  public void Dispose()
  {
    // Itt kitakarítja a nem felügyelt erőforrásokat. Meghívja a
    // Dispose() metódust más tartalmazott eldobható objektumokra.

    // Nem kell véglegesíteni, ha a használó meghívta a Dispose()
    // metódust, úgyhogy szüntessük meg a véglegesítést.
    GC.SuppressFinalize(this);
  }
}
```

A Dispose() metódus a GC.SuppressFinalize() hívással frissült, ez értesíti a CLR-t, hogy az objektum szemétgyűjtésekor nincs szükség a destruktor hívására, ugyanis a nem felügyelt erőforrásokat a rendszer a Dispose() segítségével már felszabadította.

A formalizált felszabadítási minta

A MyResourceWrapper aktuális megvalósítása ugyan jól működik, de a kód tartalmaz néhány kisebb hibát. Először is a Finalize() és a Dispose() metódusok ugyanazon nem felügyelt erőforrások takarítását végzik. Ennek eredménye a kétszerezett kód lehet, amelynek karbantartása nem könnyű feladat. Ideális esetben privát segítőfunkció definiálásával oldhatjuk meg a problémát, amelyet aztán bármelyik metódus meghívhat.

Aztán biztosítanunk kell azt, hogy a Finalize() metódus ne próbálja meg a felügyelt objektumokat eldobni, hiszen ez a feladat a Dispose() metódusra hárul. Végül pedig szeretnénk megbizonyosodni arról, hogy az objektumfelhasználó nyugodtan hívhatja a Dispose() metódust akár többször is anélkül,

hogy hibát kapna. Pillanatnyilag a `Dispose()` metódusunk nem foglal magában ehhez hasonló óvintézkedéseket. Ezekhez a tervezési feladatokhoz a Microsoft formális, prim and proper felszabadítási mintát definiált, amely megteremti a robusztusság, a fenntarthatóság és a teljesítmény közötti egyensúlyt. A `MyResourceWrapper` végső (és magyarázatokkal ellátott) verziója, amely ezt a hivatalos mintát alkalmazza, a következő:

```
public class MyResourceWrapper : IDisposable
{
  // Annak megállapítására használatos, hogy a Dispose()
  // már meghívásra került-e.
  private bool disposed = false;

  public void Dispose()
  {
    // A segédmetódusunk meghívása.
    // A "true" érték megadása azt jelzi, hogy
    // az objektumhasználó elindította a takarítást.
    CleanUp(true);
    // Most szüntessük meg a véglegesítést.
    GC.SuppressFinalize(this);
  }

  private void CleanUp(bool disposing)
  {
    // Bizonyosodjunk meg róla, hogy még nem dobtunk el semmit!
    if (!this.disposed)
    {
      // Az eldobás igaz értéket eredményez, akkor dobjuk el
      // az összes felügyelt erőforrást.
      if (disposing)
      {
        // A felügyelt erőforráso eldobása.
      }
      // Itt kitakarítja a nem felügyelt erőforrásokat.
    }
    disposed = true;
  }

  ~MyResourceWrapper()
  {
    // A segédmetódusunk meghívása.
    // A "false" érték megadása azt jelzi, hogy
    // a GC elindította a takarítást.
    CleanUp(false);
  }
}
```

A `MyResourceWrapper` a `CleanUp()` nevű privát segítőmetódust definiálja. Ha a „`true`" argumentumot adjuk meg, ezzel azt jelezzük, hogy az objektumfelhasználó megkezdte a takarítást, ezért az összes felügyelt *és* nem felügyelt erőforrást takarítanunk kell. Ha azonban a szemétgyűjtő indítja a takarítást, a `CleanUp()` hívása során a „`false`" értéket adjuk meg annak érdekében, hogy a belső eldobható objektumokat a rendszer *ne* dobja el (ugyanis nem feltételezhetjük, hogy azok még mindig a memóriában vannak). Végül, de nem utolsósorban a `bool` tagváltozó (az eldobott) `Cleanup()` befejezése előtt „`true`" értéket kap, ezzel biztosítjuk, hogy a `Dispose()` metódust hiba nélkül többször hívhatjuk.

A `MyResourceWrapper` utolsó iterációjának teszteléséhez adjuk a `Console.Beep()` metódust a véglegesítő hatóköréhez:

```
~MyResourceWrapper()
{
  Console.Beep();
  // A segédmetódusunk meghívása.
  // A "false" érték megadása azt jelzi, hogy
  // a GC elindította a takarítást.
  CleanUp(false);
}
```

A `Main()` metódust módosítsuk a következőképpen:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Dispose() /
    Destructor Combo Platter *****");

  // A Dispose() manuális meghívása, ez nem hívja meg
  // a véglegesítőt.
  MyResourceWrapper rw = new MyResourceWrapper();
  rw.Dispose();

  // Ez nem a Dispose() metódust hívja, hanem elindítja
  // a véglegesítőt és sípolást eredményez.
  MyResourceWrapper rw2 = new MyResourceWrapper();
}
```

A `Dispose()` metódust explicit módon hívjuk az `rw` objektumon, ezért a destruktorhívást a rendszer elhagyja. „Elfeledkeztünk" azonban a `Dispose()` metódus hívásáról az `rw2` objektumon, és az alkalmazás befejezésekor egyetlen hangjelzést hallunk. Ha a kódból eltávolítanánk a `Dispose()` metódus az `rw` objektumon történő hívását, két sípoló hangjelzést hallanánk.

Forráskód A `FinalizableDisposableClass` projekt a 8. fejezet alkönyvtárában található.

Ebben a fejezetben megvizsgáltuk tehát, hogy a CLR hogyan kezeli az objektumainkat a szemétgyűjtés segítségével. Noha elmerülhetnénk a gyűjtési folyamattal kapcsolatos további (kissé ezoterikus) részletekben, (pl. a gyenge referenciák és az objektum feltámasztásában), innen kezdve inkább már magunktól fedezzük fel ezeket az érdekességeket.

Összefoglalás

A fejezet célja az volt, hogy eloszlassa a szemétgyűjtési folyamattal kapcsolatos félreértéseket. A szemétgyűjtő csak akkor lép működésbe, ha nem tudja lefoglalni a szükséges memóriát a felügyelt heapen (vagy definiált alkalmazástartományt távolít el a memóriából). Ha a rendszer gyűjtést hajt végre, biztosak lehetünk abban, hogy a Microsoft gyűjtési algoritmusa az objektumgenerációk, az objektumok véglegesítését szolgáló másodlagos szálak és a nagy objektumok hosztolása révén optimalizálódik.

Láthattuk, hogyan lehet a `System.GC` osztálytípus segítségével programozottan együttműködni a szemétgyűjtővel. Erre kizárólag akkor lehet szükség, ha véglegesíthető vagy eldobható osztálytípusokat készítünk, amelyek nem felügyelt erőforrásokkal dolgoznak.

A véglegesíthető típusok olyan osztályok, amelyek felülbírálják a virtuális `System.Object.Finalize()` metódust, hogy a szemétgyűjtés idején eltakarítsák a nem felügyelt erőforrásokat. Az eldobható objektumok pedig olyan osztályok (vagy struktúrák), amelyek az `IDisposable` interfészt valósítják meg. Az interfészt az objektum felhasználója hívja, amikor végzett az említett objektummal. Végül megismertük a formalizált felszabadítási mintát, amelyben a két megközelítés keveredik.

3. rész

Haladó programozási szerkezetek a C#-ban

KILENCEDIK FEJEZET

Interfészek használata

Ez a fejezet az objektumorientált fejlesztés interfészalapú programozásának ismeretére épít. Megismerkedünk az interfészek megadásával és megvalósításával, és megmutatjuk az olyan típusok létrehozásának az előnyét, amelyek „többféle viselkedést" támogatnak. Eközben olyan témákat érintünk, mint az interfészreferencia elérése, az explicit interfészmegvalósítás és az interfészhierarchia készítése.

Továbbá foglalkozunk a .NET-alaposztálykönyvtár néhány alapvető interfészével. Az egyedi típusok szabadon valósíthatják meg ezeket az interfészeket olyan haladó tulajdonságok támogatásához, mint az objektumklónozás, az objektumfelsorolás vagy az objektumok rendezése. Végül megtudjuk, hogy az interfésztípusok hogyan használhatóak olyan visszahívási mechanizmusok létrehozására, amelyek lehetővé teszik a memóriában lévő két objektum közötti kétirányú kommunikációt.

Az Interface típus

Először is tisztázni kell az interfésztípus formális definícióját. Az *interfész* az *absztrakt tagok* nevesített halmaza. A 6. fejezetben már szó volt róla: az absztrakt eljárások tisztán csak megállapodások, amelyek nem biztosítanak alapértelmezett megvalósítást. Az interfész által megadott saját tagok attól a pontos viselkedéstől függenek, amelyet modelleznek. Az interfész azt a *viselkedést* adja meg, amelyet egy adott osztály vagy struktúra megvalósíthat. Továbbá egy osztály (vagy struktúra) annyi interfészt támogathat, amennyit szükséges, s így (lényegében) több viselkedést támogat.

A .NET-alaposztálykönyvtár több száz előre definiált interfésztípust nyújt, amelyeket különböző osztályok és struktúrák valósítanak meg. Az ADO.NET több adatszolgáltatót biztosít (lásd a 22. fejezetet), amelyek lehetővé teszik bizonyos adatbáziskezelő rendszerekkel a kommunikációt. Ezért az ADO.NET környezetben, a COM-alapú ADO-val ellentétben számos olyan kapcsolatobjektum létezik, amelyek közül választhatunk (`SqlConnection`, `OracleConnection`, `OdbcConnection` etc.).

Eltekintve attól, hogy minden kapcsolatobjektumnak saját neve van, jólehet külön névtérben van definiálva, és (bizonyos esetekben) külön szerelvényben található, minden kapcsolati objektum egyetlen közös interfészt valósít meg, az IDbConnectiont:

```
// Az IDbConnection interfész a kapcsolat objektumok által
// támogatott közös taggyűjteményt definiál.
public interface IDbConnection : IDisposable
{
  // Metódusok.
  IDbTransaction BeginTransaction();
  IDbTransaction BeginTransaction(IsolationLevel il);
  void ChangeDatabase(string databaseName);
  void Close();
  IDbCommand CreateCommand();
  void Open();

  // Tulajdonságok.
  string ConnectionString { get; set;}
  int ConnectionTimeout { get; }
  string Database { get; }
  ConnectionState State { get; }
}
```

Megjegyzés Megegyezés szerint a .NET interfésztípusai nagy „I" betűvel kezdődnek. Ha saját interfészt készítünk, a legjobb ha mi is így teszünk.

Az `IDbConnection` interfész megadja a tagok egy olyan csoportját, amelyek közösek az ADO.NET kapcsolati objektumokban. Ezzel biztosítja, hogy minden egyes kapcsolatobjektum támogatja az olyan tagokat, mint az `Open()`, `Close()`, `CreateCommand()` stb. Ezen kívül ezek az interfésztagok mindig absztraktak, így minden kapcsolatobjektum szabadon implementálhatja ezeket a metódusokat a saját egyedi módján.

Egy másik példa: a `System.Windows.Forms` névtér definiálja a `Control` nevű osztályt, ez az alaposztálya több Windows `Forms` grafikus felületi elemnek (`DataGridView`, `Label`, `StatusBar`, `TreeView` stb.). A `Control` osztály megvalósítja az `IDropTarget` nevű interfészt, amely alapvető „húzd és dobd" funkciókat definiál.

```
public interface IDropTarget
{
  // Metódusok.
  void OnDragDrop(DragEventArgs e);
  void OnDragEnter(DragEventArgs e);
  void OnDragLeave(EventArgs e);
  void OnDragOver(DragEventArgs e);
}
```

Ennek az interfésznek az alapján helyes az a feltételezésünk, hogy a `System.Windows.Forms.Control` osztályt kiegészítő osztály támogatja az `OnDragDrop()`, az `OnDragEnter()`, az `OnDragLeave()` és az `OnDragOver()` nevű metódusokat.

A következőkben tucatnyi olyan interfésszel fogunk találkozni, amelyek a .NET-alaposztálykönyvtárban vannak. Ezeket az interfészeket megvalósíthatjuk a saját egyedi osztályainkban és struktúráinkban, olyan típusokat létrehozva, amelyek szorosan integrálódnak a keretrendszerbe.

Az interfésztípusok és absztrakt ősosztályok összehasonlítása

Az interfésztípus nagyon hasonlít az absztrakt ősosztályhoz. Ha egy osztályt absztraktként adunk meg, akkor akárhány absztrakt tagot *definiálhat* egy polimorf interfészhez a leszármaztatott osztályok számára. Mindemellett, ha egy osztálytípus absztrakt tagok egy csoportját definiálja, akkor szabadon definiálhatóak hozzá konstruktorok, adatmezők, nem absztrakt tagok (megvalósításukkal együtt) stb. Ezzel szemben az interfészek *csak* absztrakt tagokat foglalhatnak magukban.

Az absztrakt ősosztály által létrehozott polimorf interfészek egy korláttal rendelkeznek: *csak a leszármazott típusok* támogatják az absztrakt szülő által definiált tagokat. Nagy szoftverrendszerekben általános az olyan többszörös osztályhierarchia létrehozása, amelyeknek a `System.Object` osztályon kívül nincsen közös ősük. Ha az absztrakt tagokat az absztrakt ősosztályokban csak a leszármazott típusokra vonatkoztatjuk, akkor nincsen lehetőség más hierarchiákban ugyanannak a polimorf interfésznek a támogatására. Vegyünk egy példát. Tételezzük fel, hogy a következő absztrakt osztályt definiáltuk:

```
abstract class CloneableType
{
  // Csak a leszármazott típusok támogatják ezt a
  // "polimorfikus interfészt." Más hierarchiák osztályai
  // nem férnek hozzá ehhez az absztrakt
  // taghoz.
  public abstract object Clone();
}
```

Ekkor csak olyan tagok tudják használni a `Clone()` metódust, amelyek a ClonableType osztályt terjesztik ki. Ha olyan osztályokat hozunk létre, amelyek nem ezt az ősosztály egészítik ki, akkor nem férünk hozzá ehhez a polimorf interfészhez. Az interfésztípusok ezen a problémán segítenek. Ha defi-

niáltunk egy interfészt, akkor azt bármilyen típus megvalósíthatja, bármely hierarchiában, bármely névtérben vagy gyűjteményben (bármely .NET-nyelven íródott is). Ezek alapján az interfészek *messzemenően* polimorfok. Vegyük az `ICloneable` standard .NET-interfészt, amelyet a `System` névtérben definiáltunk. Az interfész egyetlen metódust definiál, a `Clone()` metódust:

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

Ha megnézzük a .NET Framework 3.5 SDK dokumentációját, akkor kiderül, hogy a látszólag független típusok (`System.Array`, `System.Data.SqlClient.SqlConnection`, `System.OperatingSystem`, `System.String` stb.) mind megvalósítják ezt az interfészt. Jóllehet ezeknek a típusoknak nincsen közös őse (a `System.Object` osztályon kívül), polimorfikusan kezelhetjük őket az `ICloneable` interfésztípuson keresztül.

Ha például van egy `CloneMe()` nevű metódusunk, amely egy `ICloneable` interfészparamétert vár, akkor bármelyik olyan objektumot átadhatjuk neki, amelyik megvalósítja az adott interfészt. Vegyük a következő egyszerű Program nevű osztályt, amelyet az `ICloneableExample` nevű parancssori alkalmazásban definiáltunk:

```
class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** A First Look at Interfaces *****\n");

    // Mindezen típusok támogatják az ICloneable interfészt.
    string myStr = "Hello";
    OperatingSystem unixOS = new OperatingSystem(PlatformID.Unix,
      new Version());
    System.Data.SqlClient.SqlConnection sqlCnn =
      new System.Data.SqlClient.SqlConnection();

    // Ezért átadhatók olyan metódusnak, amely az
    // Icloneable interfésszel dolgozik.
    CloneMe(myStr);
    CloneMe(unixOS);
    CloneMe(sqlCnn);
    Console.ReadLine();
  }
```

```
  private static void CloneMe(ICloneable c)
  {
    // Klónozzuk, amit kapunk, és kiírjuk a nevét.
    object theClone = c.Clone();
    Console.WriteLine("Your clone is a: {0}",
      theClone.GetType().Name);
  }
}
```

Ha lefuttatjuk ezt az alkalmazást, akkor kiírja a parancssorra az osztályok teljes nevét a `GetType()` metódussal, amelyet a `System.Object` osztálytól örököl (ezt a metódust és a .NET reflexiós szolgáltatást lásd a 16. fejezetben).

Forráskód Az `ICloneableExample` projekt megtalálható a 9. fejezet alkönyvtárában.

A tradicionális absztrakt ősosztályoknak a másik korlátja az, hogy minden egyes származtatott típusnak tartalmaznia kell az absztrakt tagok csoportját és ezek megvalósítását. Ennek a problémának a megvizsgálásához vegyük a 6. fejezetben tárgyalt alakzatok hierarchiáját. Definiáljunk egy új `GetNumberOfPoints()` nevű absztrakt metódust a `Shape` ősosztályban, amely lehetővé teszi a származtatott típusoknak, hogy az alakzat kirajzolásához szükséges pontok számát visszaadják.

```
abstract class Shape
{
  ...
  // Minden leszármazott osztály támogatja a metódust!
  public abstract byte GetNumberOfPoints();
}
```

Világosan az egyetlen pontokkal rendelkező típus a `Hexagon`. Mindemellett, ezzel a módosítással, *minden* leszármazott típus (`Circle`, `Hexagon` és `ThreeDCircle`) megvalósítást kell, hogy adjon erre a funkcióra, még akkor is, ha igazán nincs értelme.

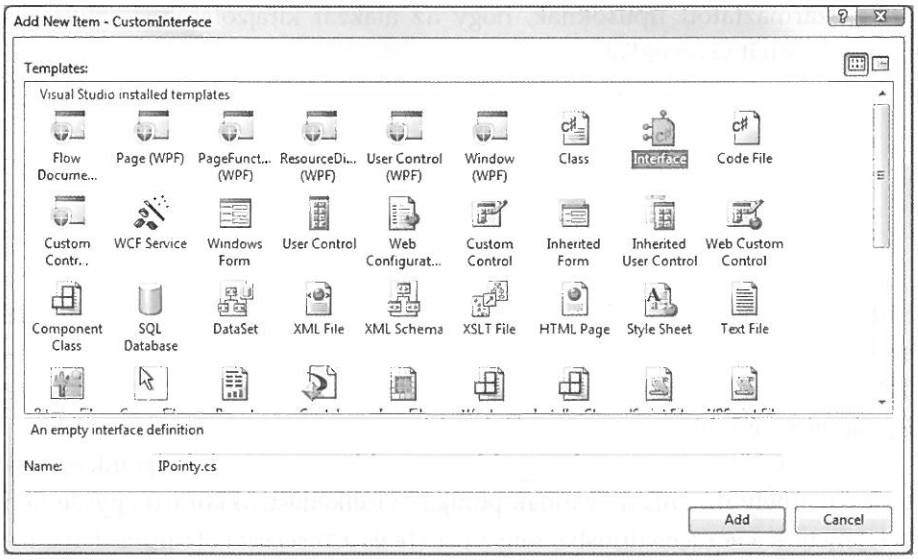
Ismét az interfésztípus ad megoldást a problémára. Ha definiálunk egy interfészt, amely mutatja a „vannak pontjai" viselkedést, akkor ezt egyszerűen beletehetjük a `Hexagon` típusba, míg a `Circle` és a `ThreeDCircle` típusokat érintetlenül hagyjuk.

Saját interfészek definiálása

Hogy jobban megértsük az interfésztípusok szerepét, nézzünk egy példát az egyedi interfész definíciójára. Ehhez hozzunk létre egy teljesen új parancssori alkalmazást `CustomInterface` néven. A Project > Add Existing Item menü használatával adjuk hozzá a projekthez az alakzatok definícióit tartalmazó állományokat (`MyShapes.cs` és `Shape.cs`), amelyeket a 6. fejezetben a `Shapes` példában hoztunk létre. Ezután nevezzük át a névteret, amely az alakzatok típusait tartalmazza `CustomInterface`-re (egyszerűen azért, hogy elkerüljük a névtér importálását az új projektünkbe):

```
namespace CustomInterface
{
  // Előzőleg itt definiáltuk az alakzat típusokat...
}
```

Adjunk hozzá a projekthez egy `IPointy` nevű új interfészt a Project > Add New Item menüponttal, ahogyan a 9.1. ábrán is látható.



9.1. ábra: *Az interfészeket, mint minden osztályt, bármilyen *.cs állományban definiálhatjuk*

Szintaktikailag az interfészt a C# nyelven az `interface` kulcsszóval definiáljuk. A többi .NET-típustól eltérően, az interfész sohasem ad meg alaposztályt (még a `System.Object` osztályt sem), és a tagjai sosem adnak meg hozzáférés-módosítót (ugyanis az interfészek tagjai mindig impliciten publikusak és absztraktak). Nézzünk egy interfészpéldát C# nyelven, hogy lássuk, hogyan működik:

```
// Ez az interfész definiálja a "vannak csúcsai" viselkedést.
public interface IPointy
{
  // Implicit módon publikus és absztrakt.
  byte GetNumberOfPoints();
}
```

Ha interfésztagot definiálunk, akkor nem adunk meg megvalósítási hatókört a kérdéses taghoz. Az interfészek tisztán csak megvalósítások, és ezért sohasem adnak megvalósítást (ez a támogató osztály vagy struktúra feladata). Ennek megfelelően az `IPointy` következő verziója különböző fordítási hibákhoz vezet:

```
// Hoppá! Rengeteg a hiba!
public interface IPointy
{

  // Hiba! Az interfészek nem rendelkezhetnek mezőkkel!
  public int numbOfPoints;

  // Hiba! Az interfészek nem rendelkezhetnek konstruktorokkal!
  public IPointy() { numbOfPoints = 0;};

  // Hiba! Az interfészek nem biztosítanak megvalósításokat!
  byte GetNumberOfPoints() { return numbOfPoints; }
}
```

Mindenesetre a kiindulási `IPointy` interfész egyetlen metódust definiál. A .NET-interfésztípusok akármennyi tulajdonságprototípust definiálhatnak. Létrehozhatjuk például az `IPointy` interfészt úgy, hogy írásvédett tulajdonságot használjon a tradicionális hozzáférő metódus helyett:

```
// A "vannan csúcsai" viselkedés mint írásvédett tulajdonság.
public interface IPointy
{
  // Az interfész írható-olvasható tulajdonsága a kövekező lenne:
  // retVal PropName { get; set; }
  // míg az interfész csak írható tulajdonsága így néz ki:
  // retVal PropName { set; }
  byte Points{ get; }
}
```

Megjegyzés Az interfésztípusok tartalmazhatnak definíciókat eseményekre (lásd 11. fejezet) és indexelőkre (lásd 12. fejezet).

Az interfésztípusok önmagukban eléggé haszontalanok, hiszen nem mások, mint absztrakt tagok nevesített gyűjteményei. Definiálhatunk például egy interfésztípusú változót úgy, ahogyan egy osztályt vagy egy struktúrát:

```
// Hoppá! Az interfész típusok foglalása érvénytelen művelet.
static void Main(string[] args)
{
  IPointy p = new IPointy(); // Fordítási hiba!
}
```

Az interfészeknek nincsen sok előnyük addig, amíg meg nem valósítja őket egy osztály vagy egy struktúra. A fenti példában az `IPointy` egy olyan interfész, amely a „vannak pontjai" viselkedést írja le. Az elképzelés egyszerű: az alakzatok hierarchiájában bizonyos osztályoknak vannak pontjaik (mint például a `Hexagon` osztálynak), míg másoknak nincsenek (mint például a `Circle` osztálynak).

Az interfész implementálása

Ha egy osztály (vagy egy struktúra) kiegészíti a funkcionalitását az interfésztípusok támogatásával, akkor ezt a típusdefinícióban egy vesszővel elválasztott listával teszi. Figyeljünk rá, hogy közvetlen ősosztály legyen a kettőspont operátor után az első listában szereplő elem. Amennyiben az osztály közvetlenül a `System.Object` osztályt tekinti ősnek, akkor egyszerűen fel lehet sorolni az osztály által támogatott interfészeket, a C#-fordító ugyanis a `System.Object` osztályból veszi a típusokat, ha nem adunk meg mást. Ha a struktúrák mindig a `System.ValueType` osztályból származnak (lásd a 4. fejezetet), egyszerűen fel lehet sorolni az interfészeket a struktúradefiníció után. Nézzük a következő példákat:

```
// Az osztály a System.Object leszármazottja, és
// egyetlen interfészt valósít meg.
public class Pencil : IPointy
{...}

// Az osztály szintén a System.Object leszármazottja, és
// egyetlen interfészt valósít meg.
public class SwitchBlade : object, IPointy
{...}
```

```
// Az osztály egyedi alaposztály leszármazottja, és
// egyetlen interfészt valósít meg.
public class Fork : Utensil, IPointy
{...}

// Az osztály implicit módon a System.ValueType osztályból
// származik, és két interfészt valósít meg.
public struct Arrow : IClonable, IPointy
{...}
```

Az interfészek implementálása mindent vagy semmit lehetőség. A támogató típus nem választhatja ki, hogy mely tagokat valósíthatja meg. Az IPointy interfész adottan csak egyetlen írásvédett tulajdonságot definiál, ez pedig nem jelent túl nagy terhet.

Viszont, ha olyan interfészt valósítunk meg, amely tíz tagot definiál (mint például a korában látott `IDbConnection`), akkor az osztálytípus felelős azért, hogy definiálja a metódusok működését.

Példaként hozzunk létre egy új `Triangle` nevű osztályt, amely a `Shape` osztályból származik („az egy" kapcsolattal), és támogatja az `IPointy` interfészt:

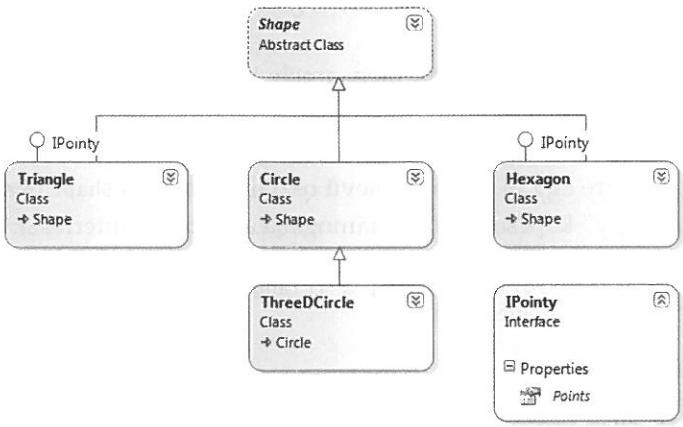
```
// A Shape újabb leszármazott osztálya, a Triangle.
public class Triangle : Shape, IPointy
{
  public Triangle() { }
  public Triangle(string name) : base(name) { }
  public override void Draw()
  { Console.WriteLine("Drawing {0} the Triangle", PetName); }

  // IPointy megvalósítás.
  public byte Points
  {
    get { return 3; }
  }
}
```

Frissítsük a már létező `Hexagon` típusunkat úgy, hogy szintén támogassa az `IPointy` interfésztípust:

```
// A Hexagon megvalósítja az IPointy interfésztípust.
public class Hexagon : Shape, IPointy
{
  public Hexagon(){ }
  public Hexagon(string name) : base(name){ }
  public override void Draw()
  { Console.WriteLine("Drawing {0} the Hexagon", PetName); }
```

```
    // IPointy megvalósítás.
    public byte Points
    {
      get { return 6; }
    }
}
```

Összegezve, a Visual Studio 2008 osztálydiagramja megmutatja az `IPointy`-kompatibilis osztályokat, ahogyan az a 9.2. ábrán látható. Figyeljük meg, hogy a `Circle` és `ThreeDCircle` osztályok nem valósítják meg az `IPointy` interfészt, minthogy ez a viselkedés nem értelmes ezekre a típusokra.



9.2. ábra: *Az alakzatok hierarchiája (már interfészekkel)*

Megjegyzés A Class Designerben az interfészek neveinek megjelenítéséhez vagy elrejtéséhez kattintsunk jobb gombbal az interfész ikonjára, és válasszuk a Collapse vagy Expand menüpontot.

Interfésztagok hívása az objektumon

A következő kérdésünk az, hogy hogyan lehet használni ezt az új funkcionalitást. Egy adott interfész által nyújtott funkcionalitás használatának legegyszerűbb módja a metódus hívása közvetlenül az objektum szintjéről (feltéve, hogy az interfésztagok nincsenek explicit módon megvalósítva; lásd részletesen a „Névütközés feloldása explicit interfészimplementációval" című részben). Példaként vegyük a következő `Main()` metódust:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  // Az IPointy által definiált Points tulajdonság hívása.
  Hexagon hex = new Hexagon();
  Console.WriteLine("Points: {0}", hex.Points);
  Console.ReadLine();
}
```

Ez a megközelítés jól működik ebben az esetben, ehhez persze tudnunk kell, hogy a `Hexagon` típus megvalósította a kérdéses interfészt, és így van Points tulajdonsága. Más esetekben azonban nem tudjuk megállapítani, hogy mely interfészeket támogatja az adott típus. Tételezzük fel például, hogy van egy tömbünk 50 `Shape`-kompatibilis típussal, amelyek közül csak némelyik támogatja az `IPointy` interfészt. Nyilvánvaló, hogy ha olyan típusra használjuk a `Points` tulajdonságot, amely nem implementálta az `IPointyt`, akkor hibát kapunk. A következő kérdés tehát: hogyan tudjuk dinamikusan eldönteni, hogy egy típus mely interfészeket támogatja?

Az egyik lehetőség arra, hogy megállapítsuk futásidőben, hogy egy típus támogat-e egy adott interfészt, az explicit kaszt használata. Ha a típus nem támogatja a kért interfészt, `InvalidCastException` kivételt kapunk. Ennek a lehetőségnek a megfelelő használatához szükség van a strukturált kivételkezelésre, például a következőképpen:

```
static void Main(string[] args)
{
  ...
  // Lehetséges InvalidCastException kivétel kezelése.
  Circle c = new Circle("Lisa");
  IPointy itfPt = null;
  try
  {
    itfPt = (IPointy)c;
    Console.WriteLine(itfPt.Points);
  }
  catch (InvalidCastException e)
  { Console.WriteLine(e.Message); }
  Console.ReadLine();
}
```

Jóllehet használhatnánk a `try/catch` szerkezetet is, de az ideális az lenne, ha meg lehetne határozni, hogy mely interfészek támogatottak, mielőtt meghívnánk az interfész tagjait. Lássunk két megoldást ennek megvalósítására.

Az interfészreferenciák megszerzése: az *as* kulcsszó

A második lehetőség annak eldöntésére, hogy egy típus támogatja-e az adott interfészt, az az, hogy használjuk az as kulcsszót (ezt a 6. fejezetben mutattuk be). Ha az objektum a megadott interfészként kezelhető, akkor a kérdéses interfészre referenciát kapunk vissza. Amennyiben nem, a visszatérési érték null referencia lesz. Ezért ne felejtsük el a folytatás előtt null értékre vizsgálni a visszakapott referenciát:

```
static void Main(string[] args)
{
  ...
  // Kezelhetjük a hex2 objektumot IPointy interfészként?
  Hexagon hex2 = new Hexagon("Peter");
  IPointy itfPt2 = hex2 as IPointy;

  if(itfPt2 != null)
    Console.WriteLine("Points: {0}", itfPt2.Points);
  else
    Console.WriteLine("OOPS! Not pointy...");
  Console.ReadLine();
}
```

Az as kulcsszó használatával nincsen szükség a try/catch szerkezetre, ha a referencia nem null értékű, akkor érvényes interfészreferenciát hívunk.

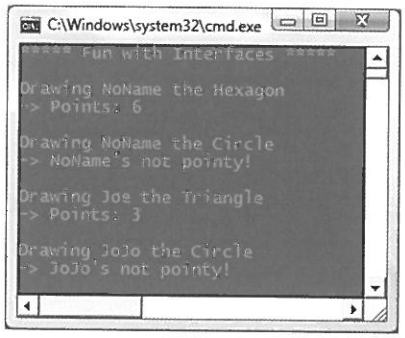
Az interfészreferenciák megszerzése: az *is* kulcsszó

A megvalósított interfész az is kulcsszóval is ellenőrizhető (ugyancsak a 6. fejezetben láthattuk). Ha a kérdéses objektum nem kompatibilis a megadott interfésszel, akkor a visszatérési érték hamis. Ha ellenben a típus kompatibilis a kérdéses interfésszel, akkor biztonságosan hívhatjuk a tagokat a try/catch szerkezet használata nélkül.

Tételezzük fel példaként, hogy van egy Shape típusokat tartalmazó tömbünk, amelynek tagjai közül néhány megvalósítja az IPointy interfészt. Figyeljük meg a módosított Main() metódusban, hogy hogyan lehet megállapítani az is kulcsszót használva, hogy a tömb elemei közül melyek támogatják ezt az interfészt:

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

  // Shapes objektumok tömbjének készítése.
  Shape[] s = { new Hexagon(), new Circle(), new Triangle("Joe"),
    new Circle("JoJo")} ;

  for(int i = 0; i < s.Length; i++)
  {
    // A Shape alaposztály definiálja az absztrakt Draw()
    // tagot, így minden alakzat tudja, hogyan lehet megrajzolni.
    s[i].Draw();

    // Melyik objektum rendelkezik csúcsokkal?
    if(s[i] is IPointy)
      Console.WriteLine("-> Points: {0}", ((IPointy)s[i]).Points);
    else
      Console.WriteLine("-> {0}\'s not pointy!", s[i].PetName);
    Console.WriteLine();
  }
  Console.ReadLine();
}
```

Az eredményt a 9.3 ábra mutatja.



9.3. ábra: *Az implementált interfészek dinamikus lekérdezése*

Interfészek mint paraméterek

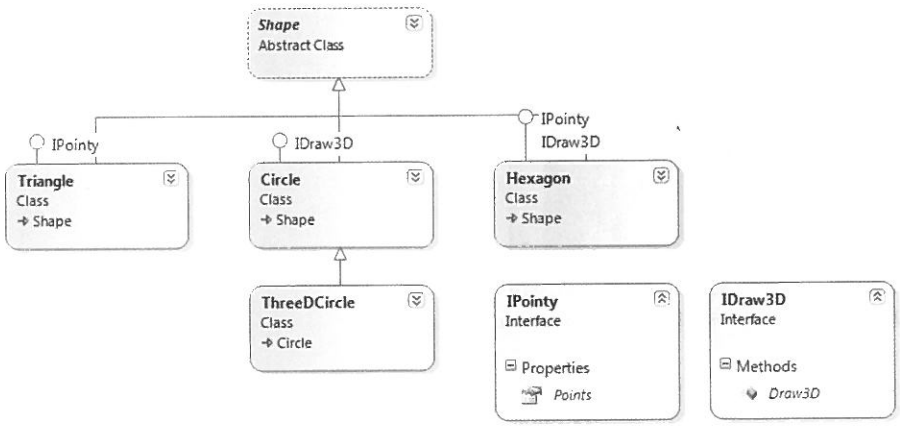
Feltéve, hogy az interfészek érvényes .NET-típusok, létre lehet hozni olyan metódusokat, amelyek interfészeket várnak paraméterként, ahogyan ezt a `CloneMe()` metódus illusztrálta korábban ebben a feladatban. Ehhez a példához tételezzük fel, hogy definiáltunk egy másik `IDraw3D` nevű interfészt:

```
// A típus 3D renderelésének lehetőségét modellezi.
public interface IDraw3D
{
  void Draw3D();
}
```

Ezután tételezzük fel, hogy a három alakzatból kettő (a `Circle` és a `Hexagon` típusok) támogatja ezt a viselkedést:

```
// A Circle támogatja az IDraw3D interfésztípust.
public class Circle : Shape, IDraw3D
{
  ...
  public void Draw3D()
  { Console.WriteLine("Drawing Circle in 3D!"); }
}

// A Hexagon támogatja az IPointy és az IDraw3D interfésztípusokat.
public class Hexagon : Shape, IPointy, IDraw3D
{
  ...
  public void Draw3D()
  { Console.WriteLine("Drawing Hexagon in 3D!"); }
}
```

A 9.4. ábra mutatja a módosított Visual Studio 2008 osztálydiagramot.



9.4. ábra: *A módosított alakzathierarchia*

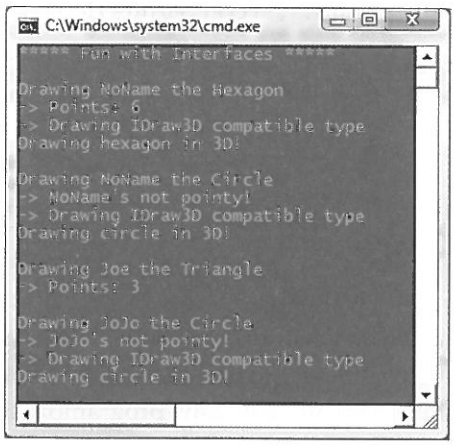
Ha definiálunk egy metódust, amely `IDraw3D` interfészt vár paraméterként, hatékonyan átadhatunk olyan objektumokat, amelyek implementálják az `IDraw3D` interfészt (ha olyan típust adunk át, amely nem támogatja a szükséges interfészt, akkor fordítási idejű hibát kapunk). Vizsgáljuk meg a következő, a programunkban definiált metódust:

```
// Az IDraw3D interfészt támogató alakzatok rajzolása.
static void DrawIn3D(IDraw3D itf3d)
{
  Console.WriteLine("-> Drawing IDraw3D compatible type");
  itf3d.Draw3D();
}
```

Tesztelhetjük, hogy egy elem a `Shape` tömbben támogatja-e az új interfészt, és ha igen, akkor átadhatjuk a `DrawIn3D()` metódusnak feldolgozásra:

```
static void Main()
{
  Console.WriteLine("***** Fun with Interfaces *****\n");
  Shape[] s = { new Hexagon(), new Circle(),
  new Triangle(), new Circle("JoJo") } ;

  for(int i = 0; i < s.Length; i++)
  {
    ...
    // Rajzolhatok három dimenzióban?
    if(s[i] is IDraw3D)
      DrawIn3D((IDraw3D)s[i]);
  }
}
```

A `Triangle` típust sosem rajzoljuk 3D-ben, mert nem `IDraw3D`-kompatibilis (lásd a 9.5. ábrát).



9.5. ábra: *Interfészek mint paraméterek*

Interfészek mint visszatérési értékek

Az interfészeket a metódusok visszatérési értékeként is használhatjuk. Írhatunk például olyan metódust, amely bármely `System.Object` objektumot vár, ellenőrzi az `IPointy`-kompatibilitást, és referenciát ad vissza az interfészére (ha az támogatott):

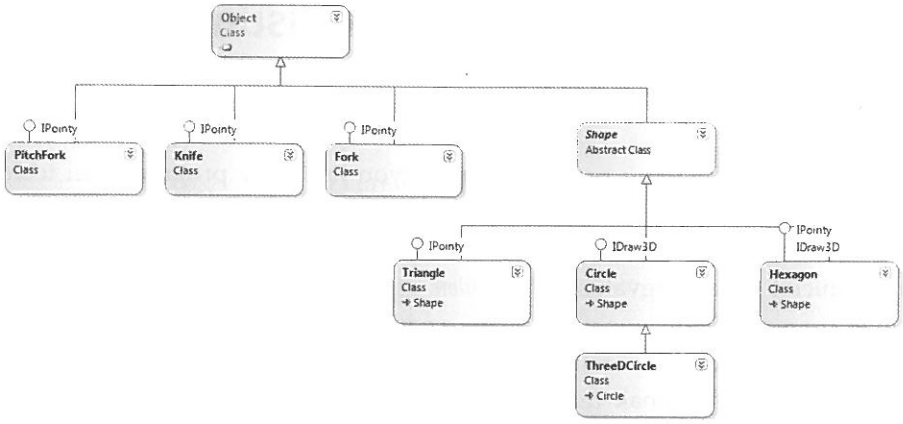
```
// A metódus teszteli az IPointy kompatibilitást, és, ha lehet,
// interfész referenciát ad vissza.
static IPointy ExtractPointyness(object o)
{
  if (o is IPointy)
    return (IPointy)o;
  else
    return null;
}
```

A következő módon kommunikálhatunk ezzel az eljárással:

```
static void Main(string[] args)
{
  ...
  // Az IPointy kiolvasása int típusú adatok tömbjéből.
  int[] myInts = {10, 20, 30};
  IPointy itfPt = ExtractPointyness(myInts);
  if(itfPt != null)
    Console.WriteLine("Object has {0} points.", itfPt.Points);
  else
    Console.WriteLine("This object does not implement IPointy");
  Console.ReadLine();
}
```

Interfésztípusok tömbje

Egy adott interfészt több típus is megvalósíthat tehát, még akkor is, hogy ha nem azonos osztályhierarchiában vannak, és nincsen közös szülőosztályuk a `System.Object` osztályon kívül. Ez néhány nagyon hatékony programozási konstrukcióhoz vezethet. Tételezzük fel például, hogy három új osztálytípust fejlesztettünk az aktuális projektben, amely konyhai eszközöket modellez (`Knife` és `Fork` osztályokkal), és egy másikat, amely kerti szerszámokat modellez (mint pl. a `PitchFork`). Nézzük meg a 9.6. ábrát.



9.6. ábra: *Emlékezzünk rá, hogy az interfészek „becsatolhatóak" bármilyen típusba bármely osztályhierarchiában*

Ha definiáltuk a `PitchFork`, a `Fork` és a `Knife` típusokat, akkor definiálhatunk egy `IPointy`-kompatibilis objektumtömböt. Feltéve, hogy az összes tag támogatja ugyanazt az interfészt, iterálhatunk a tömb elemei között, és minden elemet `IPointy`-kompatibilis objektumként tudunk kezelni, a teljes osztály hierarchia különbözőségétől függetlenül:

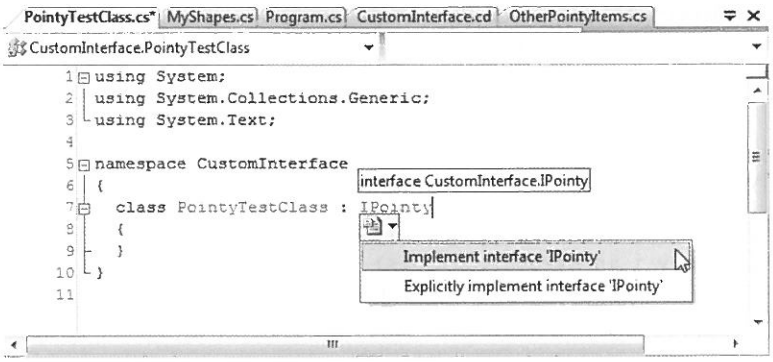
```
static void Main(string[] args)
{
  ...
  // A tömb csak olyan típusokat foglalhat magában,
  // amelyek megvalósítják az IPointy interfészt.
  IPointy[] myPointyObjects = {new Hexagon(), new Knife(),
    new Triangle(), new Fork(), new PitchFork()};
  foreach(IPointy i in myPointyObjects)
    Console.WriteLine("Object has {0} points.", i.Points);
  Console.ReadLine();
}
```

Forráskód A `CustomInterface` projekt megtalálható a 9. fejezet alkönyvtárában.

Interfészimplementálás a Visual Studio 2008 segítségével

Habár az interfészalapú programozás nagyon hatékony programozási technika, az interfészmegvalósítás komoly gépelési mennyiséggel jár. Amennyiben az interfész absztrakt tagok nevesített csoportja, akkor be kell gépelnünk a definíciót és a megvalósítást *minden* egyes interfészmetódushoz *minden* egyes típus esetében, amely támogatja az adott viselkedést.

A Visual Studio 2008 különböző eszközöket biztosít, amelyekkel az interfészek megvalósításának feladata kevésbé megterhelő. A legegyszerűbb ellenőrzésként illesszünk egy utolsó osztályt az aktuális projektbe `PointyTestClass` néven. Ha megvalósítjuk az `IPointy` interfészt (vagy bármilyen interfészt) az egyik típusban, amikor befejezzük az interfész nevének begépelését (vagy ha az egérmutatót a kódablakban egy interfész neve fölé mozgatjuk), az első betű alá lesz húzva (formálisan fogalmazva „intelligens címke" lesz). Ha az intelligens címkére kattintunk, akkor egy legördülő listát kapunk, amely lehetővé teszi az interfész megvalósítását (lásd a 9.7. ábrát).



9.7. ábra: *Interfészimplementálás a Visual Studio 2008 segítségével*

Két lehetőség közül választhatunk, ezek közül a másodikat (az explicit interfészmegvalósítást) később tárgyaljuk. Egyelőre nézzük az első lehetőséget: a Visual Studio 2008 létrehoz egy szerkeszthető kódot egy nevesített kódrészben (az alapértelmezett megvalósítás dob egy `System.Exception` kivételt, amely nyilvánvalóan törölhető).

```
namespace CustomInterface
{
  class PointyTestClass : IPointy
  {
    #region IPointy Members
    public byte Points
    {
      get { throw new Exception("The method or operation
        is not implemented."); }
    }
    #endregion
  }
}
```

Megjegyzés A Visual Studio 2008 Refractoring menüpontjában rendelkezésünkre áll az interfészátszervezés is. Ez lehetővé teszi egy interfészdefiníció kimásolását egy már létező osztálydefinícióból. Erről bővebben a „Refactoring C# Code Using Visual Studio 2005" című MSDN-cikkben lehet olvasni (természetesen az ott leírtak érvényesek a Visual Studio 2008 verzióra is).

Névütközés feloldása explicit interfészimplementációval

Egyetlen struktúra vagy osztály tehát akármennyi interfészt megvalósíthat. Így könnyen előfordulhat, hogy a megvalósított interfészekben azonos nevű tagok szerepelnek, vagyis a névütközés problémájával találjuk szembe magunkat. A problémamegoldás többféle lehetőségének bemutatásához nyissunk meg egy új parancssori alkalmazást, amelynek adjuk az `InterfaceNameClash` nevet. Készítsünk három saját interfészt, amelyek azokat a lehetőségeket képviselik, ahová egy megvalósított típus magát kirajzolhatja.

```
// Kép rajzolása egy Form objektumhoz.
public interface IDrawToForm
{
  void Draw();
}

// Rajzolás a memória pufferébe.
public interface IDrawToMemory
{
  void Draw();
}
```

```
// Renderelés a nyomtatóra.
public interface IDrawToPrinter
{
  void Draw();
}
```

Mindegyik interfész definiál egy `Draw()` metódust. Ha ezek után mind a három interfészt szeretnénk támogatni az `Octagon` nevű osztályban, akkor a fordító a következő definíciót fogja engedni:

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  public void Draw()
  {
    // Megosztott rajzolási logika.
    Console.WriteLine("Drawing the Octagon...");
  }
}
```

Jóllehet a kód rendesen lefordul, valószínűleg problémákat fog okozni. Egyszerűen megfogalmazva, a `Draw()` metódusból egyetlen megvalósítás biztosítása nem teszi lehetővé, hogy eltérő eljárásokat tudjunk az interfésztől függően az `Octagon` objektumból hívni. A következő kód például mindig ugyanazt a `Draw()` metódust hívja, függetlenül attól, hogy melyik interfészt használjuk:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interface Name
    Clashes  *****\n");
  // Ezen hívások mindegyike ugyanazt
  // a Draw() metódust hívja!
  Octagon oct = new Octagon();
  oct.Draw();
  IDrawToForm itfForm = (IDrawToForm)oct;
  itfForm.Draw();
  IDrawToPrinter itfPriner = (IDrawToPrinter)oct;
  itfPriner.Draw();
  IDrawToMemory itfMemory = (IDrawToMemory)oct;
  itfMemory.Draw();
  Console.ReadLine();
}
```

Világos, hogy egy kép ablakra történő rajzolásához teljesen eltérő kód szükséges, mintha ugyanezt a képet egy hálózati nyomtatóra vagy memóriaterületre rajzolnánk. Amennyiben olyan interfészek csoportját hozzuk létre, amelyekben azonos nevű tagok szerepelnek, akkor ezt a névütközést az *explicit interfészmegvalósítási* szintaxissal tudjuk feloldani. Vegyük példának a következő módosított `Octagon` típust:

```
class Octagon : IDrawToForm, IDrawToMemory, IDrawToPrinter
{
  // A Draw() megvalósítások explicit kötése
  // meghatározott interfészhez.
  void IDrawToForm.Draw()
  {
    Console.WriteLine("Drawing to form...");
  }
  void IDrawToMemory.Draw()
  {
    Console.WriteLine("Drawing to memory...");
  }
  void IDrawToPrinter.Draw()
  {
    Console.WriteLine("Drawing to a printer...");
  }
}
```

Az interfészek tagjainak explicit megvalósításakor az általános minta megváltozik:

```
visszatérésiÉrték InterfészNév.MetódusNév(argumentumok)
```

Ennek a szintaxisnak a használatakor nem adunk meg hozzáférés-módosítót; az explicit módon megvalósított tagok automatikusan privát tagok. A következő például helytelen szintaxis:

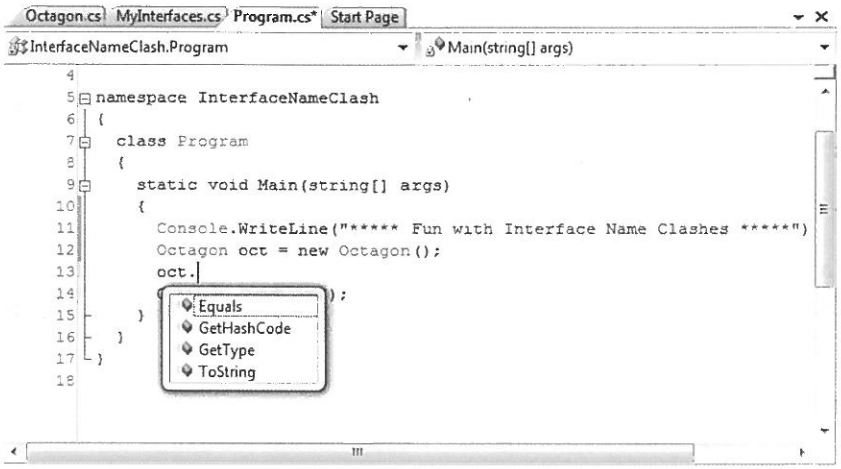
```
// Hiba! Nincs hozzáférés-módosító!
public void IDrawToForm.Draw()
{
  Console.WriteLine("Drawing to form...");
}
```

Mivel az explicit módon megvalósított tagok mindig privát tagok, így ezeket nem lehet elérni objektumszintről. Így, amikor a pont operátort alkalmazzuk egy `Octagon` típuson, az IntelliSense nem mutatja semelyik `Draw()` tagot (ahogyan a 9.8. ábrán látható).

Explicit kasztolást kell tehát alkalmaznunk a kívánt funkció eléréséhez, például:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Interface Name
    Clashes *****\n");
  Octagon oct = new Octagon();
```

```
    // Kasztolás segítségével lehet hozzáférni a Draw()
    // tagjaihoz.
    IDrawToForm itfForm = (IDrawToForm)oct;
    itfForm.Draw();

    // Rövidített jelölés, ha az interfész változóra később
    // már nem lesz szükség.
    ((IDrawToPrinter)oct).Draw();

    // Az "as" kulcsszó is alkalmazható lenne.
    if(oct is IDrawToMemory)
      ((IDrawToMemory)oct).Draw();
    Console.ReadLine();
  }
```



9.8. ábra: *Explicit módon megvalósított interfésztagok nem hívhatók meg az objektum szintjéről*

Noha ez a szintaxis segít feloldani a névütközést , az explicit interfészmegvalósítás lehetővé teszi a sokkal „fejlettebb" tagok elrejtését is az objektumszint elől. Ebben az esetben a pont operátor használatakor a felhasználó csak egy részhalmazát láthatja a típus teljes funkcionalitásának. Emellett azok, akik igénylik a fejlettebb funkcionalitást, elérhetik azt a szükséges interfészből az explicit kasztolással.

Forráskód Az `InterfaceNameClash` projekt megtalálható a 9. fejezet alkönyvtárában.

Interfészhierarchia tervezése

Az interfészeket interfészhierarchiákba szervezhetjük. Úgy, mint az osztályhierarchia esetében, ha egy interfész egy másik interfészt egészít ki, akkor örökli a szülőtípus absztrakt tagjait. Természetesen az osztályalapú származtatástól eltérően, a származtatott interfészek sohasem örökölnek tényleges megvalósítást. A származtatott interfészek inkább kiegészítik a saját definíciójukat további absztrakt tagokkal. Az interfészhierarchia akkor hasznos, ha egy már létező interfészt szeretnénk bővíteni további funkciókkal anélkül, hogy hozzányúlnánk a létező kódokhoz. Ennek bemutatásához hozzunk létre egy parancssori alkalmazást `InterfaceHierarchy` néven. Ezután tervezzük újra a rajzolásközpontú interfészeket (az `InterfaceNameClash` példából) úgy, hogy az `IDrawable` legyen a fa gyökere:

```
public interface IDrawable
{
  void Draw();
}
```

Ha az `IDrawable` alaprajzolási viselkedéseket definiál, létrehozhatunk egy származtatott interfészt, amely kiegészíti ennek a típusnak a képességeit a nyomtatóra való rajzolás képességével. Tételezzük fel, hogy ez a `Print()` metódus:

```
public interface IPrintable : IDrawable
{
  void Print();
}
```

Ráadásul definiálhatunk egy további `IRenderToMemory` nevű interfészt, amely kiegészíti az `IPrintable` interfészt egy új taggal, a `Render()` metódussal:

```
public interface IRenderToMemory : IPrintable
{
  void Render();
}
```

Ekkor, ha egy típus megvalósítja az `IRenderToMemory` interfészt, akkor kénytelenek lennénk megvalósítani minden egyes definiált tagot a származási hiererchiában (különösen a `Render()`, a `Print()`, a `Draw()` metódusokat). Másrészt, ha a típus csak az `IPrintable()` interfészt valósítja meg, akkor csak a `Print()` és `Draw()` metódusokkal kellene törődnünk. Például:

```
public class SuperShape : IRenderToMemory
{
  public void Draw()
  {
    Console.WriteLine("Drawing...");
  }

  public void Print()
  {
    Console.WriteLine("Printing...");
  }

  public void Render()
  {
    Console.WriteLine("Rendering...");
  }
}
```

Ha a `SuperShape` típust szeretnénk használni, akkor meghívhatjuk objektumszintről minden metódusát (ugyanis mind publikusak), és referenciát kaphatunk minden támogatott interfészre explicit kasztolással:

```
static void Main(string[] args)
{
  Console.WriteLine("***** The SuperShape *****");
  // Objektumszintről érkező hívás.
  SuperShape myShape = new SuperShape();
  myShape.Draw();

  // Az IPrintable explicit
  // (és az IDrawable implicit!) beolvasása.
  IPrintable iPrint;
  iPrint = (IPrintable)myShape;
  iPrint.Draw();
  iPrint.Print();
  Console.ReadLine();
}
```

Forráskód Az `InterfaceHierarchy` projekt megtalálható a 9. fejezet alkönyvtárában.

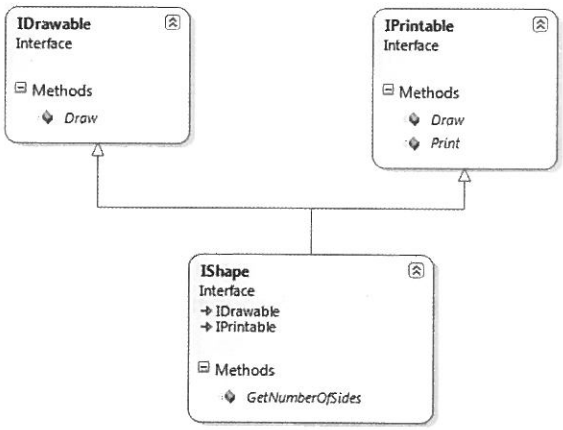
Többszörös öröklés interfésztípusokkal

Az osztálytípusokkal ellentétben egyetlen interfész kiegészíthet több alapinterfészt is. Ez nagyon hatékony és rugalmas absztrakciók tervezését teszi lehetővé. Hozzunk létre egy parancssorialkalmazás-projektet `MIInterfaceHierarchy` néven.

Egy teljesen új interfészgyűjtemény különböző rajzolási és alakzatcentrikus absztrakciókat modellez. Az `IShape` interfész az `IDrawable` és az `IPrintable` interfészeket terjeszti ki.

```
// Az interfésztípusok többszörös öröklése működik.
public interface IDrawable
{
  void Draw();
}

public interface IPrintable
{
  void Print();
  void Draw(); // <-- Lehetséges névütközés!
}

// Többszörös öröklés interfészekkel. OK!
public interface IShape : IDrawable, IPrintable
{
  int GetNumberOfSides();
}
```

A 9.9 ábrán látható a pillanatnyi interfészhierarchia.



9.9. ábra: *Az osztályoktól eltérően az interfészek több interfésztípust is kiegészíthetnek*

A kérdés az, hogy ha van egy olyan osztályunk, amely támogatja az `IShape` interfészt, hány metódusra van szükség ennek megvalósítására? A válasz: attól függ. Ha csak a `Draw()` metódus egyszerű megvalósítására van szükség, akkor csak három tag kell, ahogyan a következő `Rectangle` típusban látható:

```
class Rectangle : IShape
{
  public int GetNumberOfSides()
  { return 4; }

  public void Draw()
  { Console.WriteLine("Drawing..."); }

  public void Print()
  { Console.WriteLine("Prining..."); }
}
```

Ha minden `Draw()` metódushoz saját megvalósításra van szükségünk (ennek ebben az esetben lenne értelme), használhatjuk a névütközés feloldására az explicit interfészmegvalósítást, ahogyan a következő `Square` típus példáján láthatjuk:

```
class Square : IShape
{
  // A névütközés feloldása explicit interfészmegvalósítással.
  void IPrintable.Draw()
  { // Rajzolás nyomtatóra ...
  }
  void IDrawable.Draw()
  { // Rajzolás képernyőre ...
  }
  public void Print()
  { // Kiírás ...
  }
  public int GetNumberOfSides()
  { return 4; }
}
```

Remélhetőleg most már könnyen tudjuk definiálni és megvalósítani C# nyelven a saját interfészeket. Az interfészalapú programozáshoz hozzá kell szokni, és ez egy kis időbe telik.

Az interfészek a .NET-keretrendszer alapvető területei. Függetlenül a fejlesztett alkalmazás típusától (webalapú, grafikus felületű alkalmazások, adatbáziselérési könyvtárak stb.), az interfészek a folyamat részei lesznek. Az eddigieket összegezve, az interfészek nagyon hasznosak, ha

- egyetlen hierarchiánk van, amelyben csak a származtatott osztályok egy részhalmaza támogat közös viselkedést;
- közös viselkedést kell modellezni, amely több hierarchiában is megtalálható, és nincsen közös őse a `System.Object` osztályon kívül.

A továbbiakban a .NET-alaposztálykönyvtár előre definiált interfészei közül néhányat vizsgálunk meg.

Forráskód Az `MIInterfaceHierarchy` projekt megtalálható a 9. fejezet alkönyvtárában.

Felsorolható típusok készítése (IEnumerable és IEnumerator)

A létező .NET-interfészek megvalósítási folyamatának vizsgálatához nézzük meg az `IEnumerable` és az `IEnumerator` interfészek szerepét. A C# támogatja a `foreach` kulcsszót, amely lehetővé teszi bármilyen típust tartalmazó tömb bejárását:

```
// Az elemek tömbjének bejárása.
int[] myArrayOfInts = {10, 20, 30, 40};
foreach(int i in myArrayOfInts)
{
  Console.WriteLine(i);
}
```

Noha úgy tűnhet, hogy csak a tömbtípusok használhatják ezt a szerkezetet, az igazság az, hogy ha a típus támogatja a `GetEnumerator()` nevű metódust, bejárható a `foreach` szerkezettel. Ennek bemutatására hozzunk létre egy új parancssorialkalmazás-projektet `CustomEnumerator` néven. Ezután adjuk hozzá a `Car.cs` és a `Radio.cs` fájlokat a 7. fejezetben létrehozott `SimpleException` példából (a Project > Add Existing Item menüponttal), és egészítsük ki az aktuális osztálydefiníciót két új tulajdonsággal (a `PetName` és `Speed` nevűekkel), amelyek a `currSpeed` és a `petName` tagváltozókat tartalmazzák:

```
public class Car
{
  private int currSpeed;
  private string petName;
  public int Speed
  {
    get { return currSpeed; }
    set { currSpeed = value; }
  }
```

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

Megjegyzés Átnevezhetjük a `Car` és a `Radiotypes` osztályokat tartalmazó névteret `CustomEnumeratorra`, egyszerűen azért, hogy elkerüljük a `CustomException` névtér importálását ebbe a projektbe.

Ezután adjunk hozzá egy `Garage` nevű új osztályt, amely a `Car` típusokat egy `System.Array` tömbben tárolja:

```
// A Garage Car objektumok készletét foglalja magában.
public class Garage
{
  private Car[] carArray = new Car[4];

  // Indításkor megtöltjük néhány Car objektummal.
  public Garage()
  {
    carArray[0] = new Car("Rusty", 30);
    carArray[1] = new Car("Clunker", 55);
    carArray[2] = new Car("Zippy", 30);
    carArray[3] = new Car("Fred", 30);
  }
}
```

Ideális esetben kényelmesen lehetne iterálni a `Garage` objektum alelemeit a C# `foreach` szerkezettel, akár csak egy egyszerű adatokat tartalmazó tömböt:

```
// Ez ésszerűnek tűnik...
public class Program
{
  static void Main(string[] args)
  {
    Console.WriteLine("***** Fun with IEnumerable /
      IEnumerator *****\n");
    Garage carLot = new Garage();

    // A készletben az összes autót átadjuk?
    foreach (Car c in carLot)
    {
      Console.WriteLine("{0} is going {1} MPH",
        c.PetName, c.Speed);
    }
    Console.ReadLine();
  }
}
```

A fordító azonban informál minket arról, hogy a Garage osztály nem valósítja meg a GetEnumerator() nevű metódust. Ezt a metódust az IEnumerable interfész formalizálja, amely rejtetten a System.Collections névtérben található. Azok a típusok, amelyek támogatják ezt a viselkedést, azt hirdetik, hogy képesek az elemeiket a hívóknak ilyen módon megmutatni (ebben a példában magával a foreach szerkezettel).

```
// Az interfész tájékoztatja a hívót, hogy
// az objektum alelemeit áttekintheti.
public interface IEnumerable
{
  IEnumerator GetEnumerator();
}
```

A GetEnumerator() metódus referenciát ad vissza egy másik interfészre, amelynek a neve System.Collections.IEnumerator. Ez az interfész adja a hívónak az infrastruktúrát ahhoz, hogy az IEnumerable-kompatibilis tároló tartalmazta belső objektumokat áttekinthesse.

```
// Az interfész lehetővé teszi a hívó számára, hogy
// beolvassa a tároló alelemeit.
public interface IEnumerator
{
  bool MoveNext ();          // A kurzor belső pozíciójának
                             // léptetése.
  object Current { get;}     // Az aktuális elem beolvasása
                             // (csak olvasható tulajdonság).
  void Reset ();             // A kurzor visszaállítása az
                             // első elem elé.
}
```

Ha szeretnénk kiegészíteni a Garage típust ennek az interfésznek a támogatásával, akkor választhatjuk a hosszabb utat, és megvalósíthatjuk a metódusokat mi magunk. Szabadon létrehozhatunk saját verziókat a GetEnumerator(), a MoveNext(), a Current és a Reset() metódusokra, de létezik egyszerűbb módszer is. Mivel a System.Array típus (több más típussal együtt) már megvalósítja az IEnumerable és az IEnumerator interfészeket, egyszerűen metódusreferenciát használhatunk a System.Array felé a következőképpen:

```
using System.Collections;
...
public class Garage : IEnumerable
{
  // A System.Array már megvalósítja az IEnumerator interfészt!
  private Car[] carArray = new Car[4];
```

```
  public Garage()
  {
    carArray[0] = new Car("FeeFee", 200, 0);
    carArray[1] = new Car("Clunker", 90, 0);
    carArray[2] = new Car("Zippy", 30, 0);
    carArray[3] = new Car("Fred", 30, 0);
  }

  public IEnumerator GetEnumerator()
  {
    // Visszaadja a tömbobjektum IEnumerator értékét.
    return carArray.GetEnumerator();
  }
}
```

Ha frissítettük a `Garage` típusunkat, akkor már biztonságosan használhatjuk a C# `foreach` szerkezetében. Ráadásul, ha a `GetEnumerator()` metódus definíciója publikus, az objektum használója kommunikálhat az `IEnumerator` típussal:

```
// Dolgozzunk kézzel az IEnumerator interfésszel.
IEnumerator i = carLot.GetEnumerator();
i.MoveNext();
Car myCar = (Car)i.Current;
Console.WriteLine("{0} is going {1} MPH", myCar.PetName,
  myCar.Speed);
```

Ha azonban szeretnénk elrejteni az `IEnumerable` funkcióit az objektumszinten, akkor használjuk az explicit interfészmegvalósítást:

```
IEnumerator IEnumerable.GetEnumerator()
{
  // Visszaadja a tömbobjektum IEnumerator értékét.
  return carArray.GetEnumerator();
}
```

Ha így teszünk, akkor a hétköznapi felhasználó nem fogja megtalálni a `Garage` típus `GetEnumerator()` metódusát, mivel a `foreach` szerkezet szükség esetén a háttérben kapja meg az interfészt.

Forráskód A `CustomEnumerator` projekt megtalálható a 9. fejezet alkönyvtárában.