A `System.Data` névtéren belül akkor használjuk a legtöbb osztályt, amikor az ADO.NET kapcsolat nélküli modelljével dolgozunk. A következő fejezetben megismerkedünk a `DataSetnek` és társainak (`DataTable`, `DataRelation`, `DataRow` stb.) részleteivel, valamint azzal, hogy segítségükkel (és a kapcsolódó adatillesztőkkel) hogyan lehet megjeleníteni és kezelni a távoli adatok kliensoldali másolatát.

Most azonban magasabb szintről megvizsgáljuk a `System.Data` névtér alapvető interfészeit, és azt, hogy milyen közös funkcionalitást nyújtanak általában az adatszolgáltatók.

Az IDbConnection interfész szerepe

Az `IDbConnection` típust az adatszolgáltató *kapcsolatobjektuma* valósítja meg. Az interfész egy megadott adattároló kapcsolatának konfigurálására szolgáló tagok halmazát definiálja, és lehetővé teszi, hogy hozzáférjünk az adatszolgáltató tranzakcióobjektumához. Az `IDbConnetcion` formális definíciója a következő:

```
public interface IDbConnection : IDisposable
{
  string ConnectionString { get; set; }
  int ConnectionTimeout { get; }
  string Database { get; }
  ConnectionState State { get; }
  IDbTransaction BeginTransaction();
  IDbTransaction BeginTransaction(IsolationLevel il);
  void ChangeDatabase(string databaseName);
  void Close();
  IDbCommand CreateCommand();
  void Open();
}
```

Megjegyzés Ahogy a .NET alaposztálykönyvtáraiban szereplő számos más típus, a `Close()` metódus meghívása megegyezik funkcionálisan a `Dispose()` metódus direkt vagy indirekt hívásával a C# `using` blokkjában. (lásd az előző kötet 8. fejezetét).

Az IDbTransaction interfész szerepe

Az `IDbConnection` által definiált, túlterhelt `BeginTransaction()` metódus hozzáférést biztosít a szolgáltató *tranzakcióobjektumához*. Az `IDbTransaction` tagjainak használatával lehetőségünk van programozottan kezelni a tranzakciós munkamenetet és a mögöttes adattárolót:

```
public interface IDbTransaction : IDisposable
{
  IDbConnection Connection { get; }
  IsolationLevel IsolationLevel { get; }
  void Commit();
  void Rollback();
}
```

Az IDbCommand interfész szerepe

Az `IDbCommand` interfészt az adatszolgáltató *parancsobjektuma* implementálja. Más adatelérési objektummodellhez hasonlóan a parancsobjektumok SQL-utasítások, tárolt eljárások és paraméterezett lekérdezések programozott kezelését teszik lehetővé. Továbbá a parancsobjektumok a túlterhelt `ExecuteReader()` metódus segítségével az adatszolgáltató adatolvasó típusához biztosítanak hozzáférést.

```
public interface IDbCommand : IDisposable
{
  string CommandText { get; set; }
  int CommandTimeout { get; set; }
  CommandType CommandType { get; set; }
  IDbConnection Connection { get; set; }
  IDataParameterCollection Parameters { get; }
  IDbTransaction Transaction { get; set; }
  UpdateRowSource UpdatedRowSource { get; set; }
  void Cancel();
  IDbDataParameter CreateParameter();
  int ExecuteNonQuery();
  IDataReader ExecuteReader();
  IDataReader ExecuteReader(CommandBehavior behavior);
  object ExecuteScalar();
  void Prepare();
}
```

Az IDbDataParameter és az IDataParameter interfészek szerepe

Az `IDbCommand` `Parameter` tulajdonsága egy olyan erősen típusos gyűjteményt ad vissza, amely az `IDataParameterCollection` interfészt valósítja meg. Ez az interfész hozzáférést biztosít `IDbDataParameter`-kompatibilis osztálytípusok halmazához (pl. paraméterobjektumokhoz):

```
public interface IDbDataParameter : IDataParameter
{
  byte Precision { get; set; }
  byte Scale { get; set; }
  int Size { get; set; }
}
```

Az `IDbDataParameter` interfész az `IDataParameter` interfészt bővíti ki, és a további viselkedéseket biztosítja:

```
public interface IDataParameter
{
  DbType DbType { get; set; }
  ParameterDirection Direction { get; set; }
  bool IsNullable { get; }
  string ParameterName { get; set; }
  string SourceColumn { get; set; }
  DataRowVersion SourceVersion { get; set; }
  object Value { get; set; }
}
```

Az `IDbDataParameter` és az `IDataParameter` interfészek lehetővé teszik, hogy az SQL-parancsokon belül (ide értve a tárolt eljárásokat is) paramétereket állítsunk be a specifikus ADO.NET-paraméterobjektumokon keresztül, ahelyett, hogy kódolt sztringliterálokat használnánk.

Az IDbDataAdapter és az IDataAdapter interfészek szerepe

Az *adatillesztők* segítségével `DataSeteket` vehetünk ki egy adattárolóból, vagy oda vissza is írhatjuk őket. Az `IDbDataAdapter` interfész olyan tulajdonságok halmazát definiálja, amelyek SQL-utasítások kezelésére használatosak a hozzájuk kapcsolódó lekérdezési, beszúrási, módosítási és törlési műveletekhez:

```
public interface IDbDataAdapter : IDataAdapter
{
  IDbCommand DeleteCommand { get; set; }
  IDbCommand InsertCommand { get; set; }
  IDbCommand SelectCommand { get; set; }
  IDbCommand UpdateCommand { get; set; }
}
```

Ugyanez igaz a tagok visszatérési értékeire is. Nézzük meg például a következő egyszerű C#-konzolalkalmazást (MyConnectionFactory névvel), amely egyedi felsorolt típus alapján lehetővé teszi egy adott kapcsolatobjektum megszerzését. Diagnosztikai célokból egyszerűen csak kiírjuk az alatta található kapcsolatobjektumot a reflexiós szolgáltatás segítségével:

```
// Erre szükségünk van, hogy a közös interfészek definícióit
// és néhány kapcsolatobjektumot megkapjunk a tesztünkhöz.
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.Odbc;
using System.Data.OleDb;

// Szükség van a System.Data.OracleClient.dll-re történő
// referenciára ennek a névtérnek a használatához.
using System.Data.OracleClient;

namespace MyConnectionFactory
{
  // A lehetséges szolgáltatók listája.
  enum DataProvider
  { SqlServer, OleDb, Odbc, Oracle, None }

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

      // Egy adott kapcsolat beolvasása.
      IDbConnection myCn = GetConnection(DataProvider.SqlServer);
      Console.WriteLine("Your connection is a {0}",
                        myCn.GetType().Name);

      // A kapcsolat megnyitása, használata és bezárása...
      Console.ReadLine();
    }

    // Ez a metódus tér vissza egy adott kapcsolatobjektummal
    // a DataProvider értéke alapján.
    static IDbConnection GetConnection(DataProvider dp)
    {
      IDbConnection conn = null;
      switch (dp)
      {
        case DataProvider.SqlServer:
          conn = new SqlConnection();
          break;
```

```
            case DataProvider.OleDb:
              conn = new OleDbConnection();
              break;
            case DataProvider.Odbc:
              conn = new OdbcConnection();
              break;
            case DataProvider.Oracle:
              conn = new OracleConnection();
              break;
          }
          return conn;
        }
      }
    }
```

A `System.Data` általános interfészeivel (vagy akár ugyanígy a `System.Data.Common` absztrakt ősosztályaival) nagyobb esélyünk van arra, hogy olyan rugalmas kódot írjunk, amely idővel továbbfejleszthető. Lehet, hogy például most egy olyan alkalmazást írunk, amely a Microsoft SQL Serverrel dolgozik, de mi van akkor, ha pár hónap múlva át kell térni Oracle-re? Ha olyan alkalmazást írtunk, amely bedrótozta az MS SQL Server specifikus `System.Data.SqlClient` típusokat, nyilvánvalóan a szerelvény módosítására, újrafordítására és újratelepítésére van szükség, ha a háttérben futó adatbáziskezelő rendszer változik.

Rugalmasság növelése az alkalmazáskonfigurációs fájlokkal

Hogy tovább növelhessük az ADO.NET-alkalmazásaink rugalmasságát, felhasználhatunk egy kliensoldali `*.config` fájlt, amely az `<appSettings>` elem kulcs/érték párjait használja fel. A `*.config` fájlban tárolt egyedi adatok programozottan kinyerhetőek a `System.Configuration` névtér típusainak a segítségével (lásd az előző kötet 15. fejezetét). Tételezzük fel például, hogy egy konfigurációs fájlban a következőképpen adtuk meg az adatszolgáltató értékét:

```
<configuration>
  <appSettings>
    <!-- Ez a kulcsérték a felsorolt típusunk valamely értékével
         egyezik meg -->
    <add key="provider" value="SqlServer"/>
  </appSettings>
</configuration>
```

Ezek alapján úgy módosíthatjuk a `Main()` függvényt, hogy programozottan szerezze meg a mögöttes adatszolgáltatót. Ekkor lényegében egy *kapcsolatobjektum-factoryt* készítettünk, amely lehetővé teszi, hogy úgy változtassuk meg a szolgáltatót, hogy ne kelljen hozzá újrafordítani a kódot (egyszerűen a `*.config` fájlt kell módosítani). Az alábbiakban a `Main()` releváns módosításait látjuk:

```
static void Main(string[] args)
{
  Console.WriteLine("**** Very Simple Connection Factory *****\n");

  // Olvassuk be az adatszolgáltató kulcsot.
  string dataProvString =
         ConfigurationManager.AppSettings["provider"];

  // Alakítsuk át a sztringértéket felsorolássá.
  DataProvider dp = DataProvider.None;
  if(Enum.IsDefined(typeof(DataProvider), dataProvString))
    dp = (DataProvider)Enum.Parse(typeof(DataProvider),
         dataProvString);
  else
    Console.WriteLine("Sorry, no provider exists!");

  // Egy adott kapcsolat beolvasása.
  IDbConnection myCn = GetConnection(dp);
  if(myCn != null)
    Console.WriteLine("Your connection is a {0}",
           myCn.GetType().Name);

  // Nyissuk meg, használjuk és zárjuk a kapcsolatot...

  Console.ReadLine();
}
```

Megjegyzés Ahhoz, hogy a `ConfigurationManager` típust használjuk, állítsunk referenciát a `System.Configuration.dll` szerelvényre, és importáljuk a `System.Configuration` névteret.

Az eddig létrehozott ADO.NET-kód lehetővé teszi, hogy dinamikusan határozzuk meg a mögöttes kapcsolatot. Az egyetlen nyilvánvaló probléma, hogy ezt az absztrakciót csak a `MyConnectionFactory.exe` alkalmazásban használhatjuk. Ha ezt a példaprogramot egy .NET-kódkönyvtárban készítettük volna el (pl. a `MyConnectionFactory.dll`-ben), akkor lehetőségünk lenne tetszőleges számú klienst létrehozni, amely különböző kapcsolatobjektumokat használhatna az absztrakciós rétegek segítségével.

A kapcsolatobjektum megszerzése az ADO.NET használatának csak egyik lényeges szempontja. Ahhoz, hogy valamirevaló data provider factory kódkönyvtárat hozzunk létre, a parancsobjektumokat, az adatolvasókat, az adatillesztőket, a tranzakcióobjektumokat és az egyéb adatközpontú típusokat hasonlóan kell kezelni. Noha egy ilyen kódkönyvárat nem feltétlenül nehéz elkészíteni, de nagy mennyiségű kódot és sok időt igényelne.

A .NET 2.0 verziójától kezdve ez a funkcionalitás közvetlenül beépült a .NET-alaposztálykönyvtáraiba. Mielőtt megvizsgálnánk ezt a formális API-t, létre kell hoznunk egy olyan egyedi adatbázist, amelyet a későbbiekben is használni fogunk.

Forráskód A MyConnectionFactory kódfájlokat a forráskódkönyvtárban a 22. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv.oldalát.

Az AutoLot adatbázis létrehozása

Ebben a fejezetben mindig egy egyszerű, AutoLot nevű SQL Server tesztadatbázison hajtunk végre lekérdezéseket. Kapcsolódva az eddigi autós témához, ez az adatbázis három egymással kapcsolatban álló táblát tartalmaz (Inventory - raktár, Orders - megrendelések és Customers - vásárlók). Ezek a táblák különböző jellegű adatokat tartalmaznak, amelyekben egy kitalált autókereskedés megrendelési információi találhatók.

Tételezzük fel, hogy van egy Microsoft SQL Server (7.0 vagy magasabb verziójú) vagy egy Microsoft SQL Server 2005 Express Edition adatbáziskezelő szoftverünk (http://msdn.microsoft.com/vstudio/express/sql). Ez az adatbázisszerver tökéletesen megfelel a céljainkra, hiszen egyrészt ingyenes, másrészt olyan grafikus felületet biztosít (az SQL Server Management Tool eszközt), amelynek segítségével létrehozhatjuk és felügyelhetjük az adatbázisainkat, és harmadrészt a Visual Studio 2008/Visual C# Express Edition változattal integrálható.

Ez utóbbi előnyének illusztrálására megnézzük, hogy hogyan kell létrehozni az AutoLot adatbázist a Visual Studio 2008 segítségével. Ha Visual C# Express verziónk van, hasonló műveleteket hajthatunk rajta végre, mint amelyeket itt részletezünk a Database Explorer ablak segítségével (amelyet a View > Other Windows menüpont használatával lehet betöltetni).

Megjegyzés Az AutoLot adatbázist a könyv további részeiben is használni fogjuk.

Az Inventory tábla létrehozása

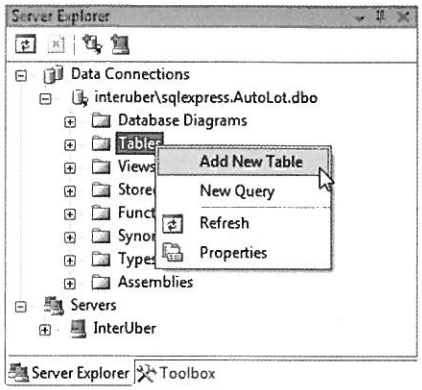
Hogy létrehozhassuk a tesztadatbázisunkat, indítsuk el a Visual Studio 2008-at, és nyissuk meg a Server Explorer nézetet az integrált fejlesztői környezet View menüpontjából. Kattintsunk a jobb egérgombbal a Data Connections csomópontra, és válasszuk ki a Create New SQL Server Database menüpontot. Az eredményként megjelenő párbeszédablakban kapcsolódjunk a helyi gépünkön található SQL Server példányhoz, és adjuk meg az AutoLot kifejezést az adatbázis neveként (a Windows Authentication megfelelő lesz - lásd a 22.3 ábrát).

22.3. ábra: *Egy új SQL Server Express adatbázis létrehozása Visual Studio 2008 használatával*

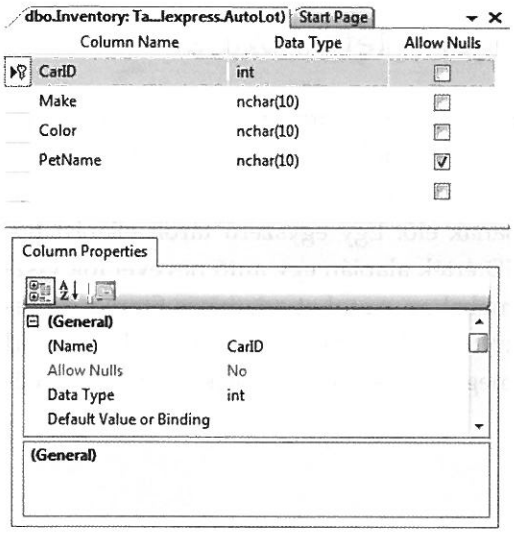
Megjegyzés Ahelyett, hogy konkrétan megadnánk a számítógépünk nevét (mint az INTERUBER a 22.3. ábrán), a Server name szövegdobozba egyszerűen beírhatjuk, hogy **(local)\SQLEXPRESS.**

Ekkor az AutoLot adatbázis egyetlen adatbázisobjektumot (táblát, tárolt eljárásokat stb.) sem tartalmaz. Az Inventory tábla beszúrásához egyszerűen kattintsunk jobb egérgombbal a Tables csomópontra, és válasszuk ki az Add New Table parancsot (lásd a 22.4. ábrát).

A táblázatszerkesztővel hozzunk létre négy oszlopot (CarID, Make, Color, PetName). Bizonyosodjunk meg róla, hogy a CarID oszlopot elsődleges kulcsként állítottuk be. Ehhez jobb gombbal kattintsunk a CarID sorra, és válasszuk a Set Primary Key opciót. A 22.5. ábra mutatja a végső táblabeállításokat.



22.4. ábra: *Az Inventory tábla hozzáadása*



22.5. ábra: *Az Inventory tábla tervezése*

Mentsük el (majd zárjuk be) az új táblánkat, és bizonyosodjunk meg róla, hogy az új adatbázis-objektumunknak az Inventory nevet adtuk. Ekkor látnunk kell az Inventory táblát a Tables csomópont alatt a Server Explorerben. Kattintsunk jobb gombbal az Inventory tábla ikonjára, és válasszuk ki a Show Table Data opciót. Vigyünk be vagy egytucatnyi tetszőleges autó adatait (hogy érdekesebb legyen a dolog, legyenek azonos színű és gyártmányú autóink). A 22.6. ábra egy lehetséges raktárkészletlistát mutat.

Inventory: Query(...lexpress.AutoLot) | Start Page

	CarID	Make	Color	PetName
▶	1	BMW	Green	Sidd
	2	VW	Red	Zippy
	3	Ford	Black	Mel
	4	BMW	Silver	Henry
	5	Yugo	Pink	Sally
	6	Saab	Blue	Sven
	7	BMW	Black	Bimmer
	8	VW	Tan	Sal
*	NULL	NULL	NULL	NULL

1 of 8

22.6. ábra: *Az Inventory tábla feltöltése*

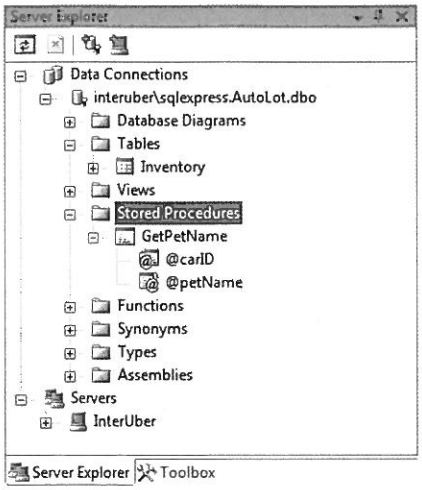
A GetPetName() tárolt eljárás létrehozása

A későbbiekben megvizsgáljuk, hogyan használható az ADO.NET tárolt eljárások hívására. A tárolt eljárások olyan rutinok, amelyeket egy adott adatbázis tárol, és legtöbbször táblákon hajtanak végre műveleteket azért, hogy valamilyen visszatérési értéket állítsanak elő. Egy egyszerű tárolt eljárást hozunk létre, amely a megadott CarID érték alapján egy autó nevével fog visszatérni. Kattintsunk jobb egérgombbal az AutoLot adatbázis Stored Procedures csomópontjára a Server Explorer ablakban, és válasszuk ki az Add New Stored Procedure opciót. A megjelenő szerkesztőablakba írjuk be a következő kódot:

```
CREATE PROCEDURE GetPetName
@carID int,
@petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarID = @carID
```

Amikor mentjük az eljárásunkat, automatikusan a GetPetName nevet kapja a `CREATE PROCEDURE` utasítás alapján. Ha készen vagyunk, az új tárolt eljárást a Server Explorer ablakban látjuk (lásd a 22.7. ábrát).

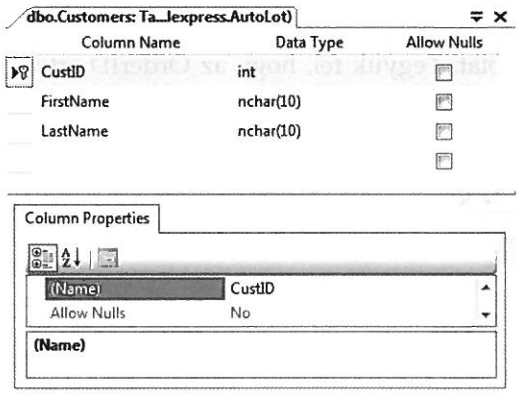
Megjegyzés A tárolt eljárásoknak nem kell kimeneti paraméterrel visszatérniük (mint ahogy itt látható). A fejezet „Tárolt eljárások futtatása" című részében lesz szó az `SqlParameter Direction` tulajdonságának vizsgálatáról.



22.7. ábra: *A GetPetName tárolt eljárás*

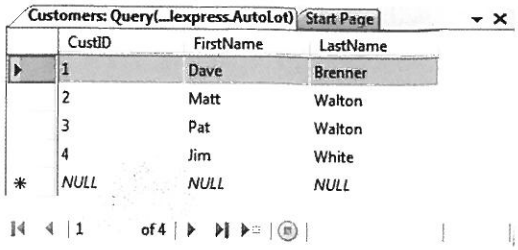
A Customers és az Orders táblák létrehozása

A tesztadatbázisunk két további táblával gyarapodik. A Customers tábla (ahogy azt a neve is mutatja) a vásárlók listáját fogja tartalmazni, amelyet három oszlop képvisel (CustID [amelyet elsődleges kulcsként kell majd beállítani], FirstName és LastName). Az Inventory tábla létrehozásakor végrehajtott lépésekkel a következő séma alapján hozzuk létre a Customers táblát (lásd a 22.8. ábrát).



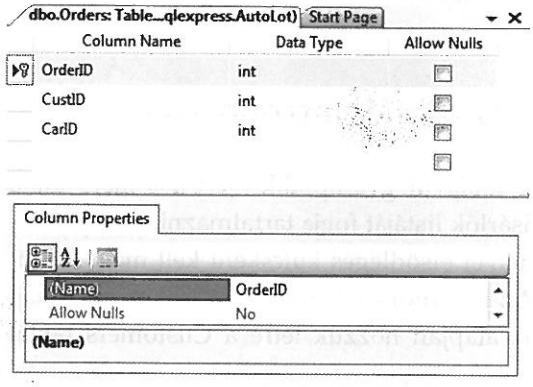
22.8. ábra: *A Customers tábla tervezése*

Miután elmentettünk, töltsük fel a táblát vásárlói rekordokkal (lásd a 22.9. ábrát).



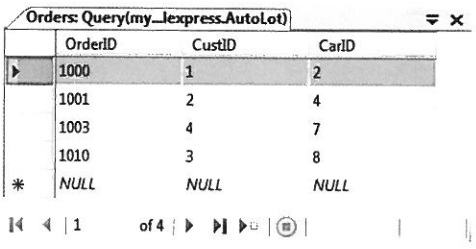
22.9. ábra: *A Customers tábla feltöltése*

Utolsó táblánk, az Orders azokat az autókat tartalmazza, amelyeket az adott vásárlók szeretnének megvenni. Ehhez az OrderID értékeket CarID/CustID értékekre kell leképezni. A 22.10. ábra mutatja be a tábla végső struktúráját (az OrderID lesz tehát az elsődleges kulcs).



22.10. ábra: *Az Orders tábla tervezése*

Töltsük fel adatokkal az Orders táblát. Tegyük fel, hogy az OrderID értéke 1000-rel kezdődik. Válasszunk egy egyedi CarID értéket minden egyes CustID értékhez (lásd a 22.11. ábrát).



22.11. ábra: *Az Orders tábla feltöltése*

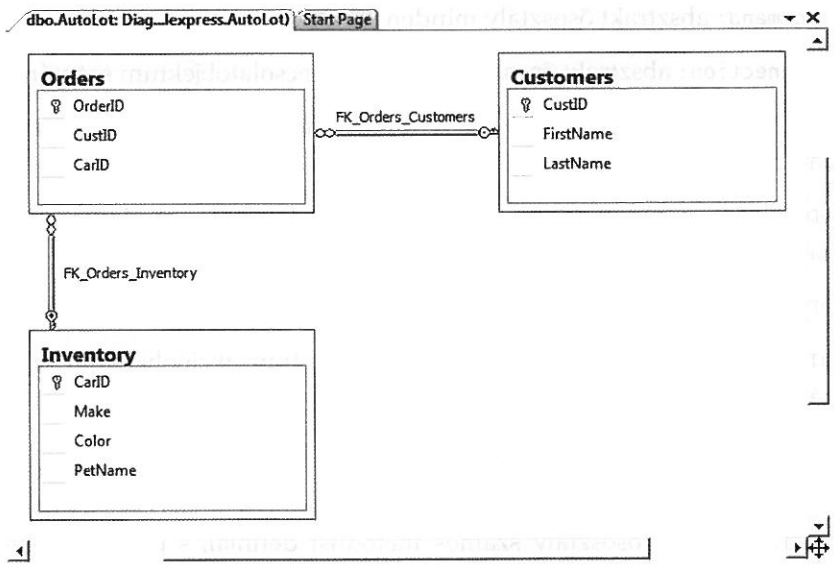
Ha a könyvbeli adatokat vittük be, akkor láthatjuk, hogy Dave Banner (CustID = 1) a piros Volkswagent szeretné megvenni (CarID = 2), míg Pat Walton (CustID = 3) a rozsdabarna Volkswagenre (CarID = 8) vetett szemet.

Táblakapcsolatok vizuális bemutatása

Az utolsó feladatunk az, hogy a szülő/gyermek kapcsolatokat beállítsuk a Customers, Orders és az Inventory táblák között. Ez a Visual Studio 2008 segítségével egyszerű, hiszen ha a Server Explorerben jobb egérgombbal kattintunk az AutoLot adatbázis Database Diagrams csomópontjára, egy új adatbázis-diagram létrehozását választhatjuk ki. Mielőtt az Add gombot megnyomnánk, nézzük meg, hogy minden táblát kiválasztottunk-e a megjelenő párbeszédablakban.

Ahhoz, hogy a kapcsolatokat felállítsuk a táblák között, először kattintsunk az Inventory tábla CarID kulcsára (mialatt lenyomva tartjuk az egér gombját), majd húzzuk oda az Orders tábla CarID mezőjéhez. Amikor elengedjük az egér gombját, fogadjunk el minden alapbeállítást az eredményként megjelenő párbeszédablakokban.

Ugyanennek a műveletsornak a végrehajtásával a Customers tábla CustID kulcsát rendeljük az Orders tábla CustID mezőjéhez.



22.12. ábra: *Az összekapcsolt Orders, Inventory és Customers táblák*

Ha végeztünk, akkor a 22.12. ábrán látható osztály-párbeszédablakot kapjuk (a táblák közti kapcsolatok megjelenítésének engedélyezéséhez a tervezőben kattintsunk jobb egérgombbal, és válasszuk a Show Relationship Labels opciót).

Ezzel elkészítettük az AutoLot adatbázist. Noha ez elég messze áll egy valós életben is használatos adatbázistól, számunkra a továbbiakban megfelelő lesz.

Az ADO.NET data provider factory modell

A .NET data provider factory minta egyszerű kódbázis létrehozását teszi lehetővé általánosított adatelérési típusok alkalmazásával. Továbbá alkalmazáskonfigurációs fájlok (és a `<connectionStrings>` elem) használatával deklaratív módon kaphatunk szolgáltatókat és kapcsolatsztringeket anélkül, hogy újra kellene fordítani és telepíteni a szerelvényt.

A data provider factory implementációjának megértéséhez emlékezzünk vissza a 22.1. táblában foglaltakra: egy adatszolgáltatóban az objektumok ugyanabból az ősosztályból származnak, amelyet a `System.Data.Common` névtér definiál:

- `DbCommand`: absztrakt ősosztály minden parancsobjektum számára.
- `DbConnection`: absztrakt ősosztály minden kapcsolatobjektum számára.
- `DbDataAdapter`: absztrakt ősosztály minden adatillesztő objektum számára.
- `DbDataReader`: absztrakt ősosztály minden adatolvasó objektum számára.
- `DbParameter`: absztrakt ősosztály minden paraméterobjektum számára.
- `DbTransact ion`: absztrakt ősosztály minden tranzakcióobjektum számára.

Ezenkívül minden egyes Microsoft által szállított adatszolgáltató tartalmaz egy olyan osztálytípust, amely a `System.Data.Common.DbProviderFactory` osztályból származik. Az ősosztály számos metódust definiál, s ezek szolgáltatóspecifikus objektumokkal térnek vissza. Nézzük meg a `DbProviderFactory` fontosabb tagjait:

```
public abstract class DbProviderFactory
{
  ...
  public virtual DbCommand CreateCommand();
  public virtual DbCommandBuilder CreateCommandBuilder();
  public virtual DbConnection CreateConnection();
  public virtual DbConnectionStringBuilder
                 CreateConnectionStringBuilder();
  public virtual DbDataAdapter CreateDataAdapter();
  public virtual DbDataSourceEnumerator
                 CreateDataSourceEnumerator();
  public virtual DbParameter CreateParameter();
}
```

Megszerezhetjük a `DbProviderFactory` leszármazott típusát az adatszolgáltatónk számára: ehhez a `System.Data.Common` névtér a `DbProviderFactories` osztálytípust biztosítja (figyeljük meg a többes számot a típus nevében). A statikus `GetFactory()` metódus használatával megkaphatjuk a megadott adatszolgáltató specifikus `DbProviderFactory` objektumát, például:

```
static void Main(string[] args)
{
  // Kérjük az SQL adatszolgáltatóhoz tartozó factoryt.
  DbProviderFactory sqlFactory =
    DbProviderFactories.GetFactory("System.Data.SqlClient");
  ...
  // Kérjük az Oracle adatszolgáltatóhoz tartozó factoryt.
  DbProviderFactory oracleFactory =
    DbProviderFactories.GetFactory("System.Data.OracleClient");
  ...
}
```

Ahelyett, hogy bedrótozott sztringliterálokkal kérnénk le a factoryt, ezt az információt a kliensoldali `*.config` fájlból is beolvashatjuk (az előző `MyConnectionsFactory` példához hasonlóan). Miután a factory rendelkezésre áll az adatszolgáltatónk számára, megszerezhetjük a hozzátartozó szolgáltatóspecifikus adatobjektumokat (kapcsolatokat, parancsokat, adatolvasókat stb.).

Regisztrált data provider factoryk

A `DbProviderFactories` típus csak a lehetséges adatszolgáltatók egy részhalmazára képes factorykat lekérdezni. Az érvényes provider factoryk listáját a .NET 3.5 telepítés machine.config fájljának `<DbProviderFactories>` eleme tárolja (az *invariant* attribútum, értéke megegyezik a `DbProviderFactories.GetFactory()` metódusnak átadott értékkel):

```
<system.data>
  <DbProviderFactories>
    <add name="Odbc Data Provider" invariant="System.Data.Odbc"
      description=".Net Framework Data Provider for Odbc"
      type="System.Data.Odbc.OdbcFactory,
      System.Data, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089" />
    <add name="OleDb Data Provider" invariant="System.Data.OleDb"
      description=".Net Framework Data Provider for OleDb"
      type="System.Data.OleDb.OleDbFactory,
      System.Data, Version=2.0.0.0, Culture=neutral,
      PublicKeyToken=b77a5c561934e089" />
    <add name="OracleClient Data Provider"
              invariant="System.Data.OracleClient"
      description=".Net Framework Data Provider for Oracle"
      type="System.Data.OracleClient.OracleClientFactory,
            System.Data.OracleClient,
      Version=2.0.0.0, Culture=neutral,
              PublicKeyToken=b77a5c561934e089" />
    <add name="SqlClient Data Provider"
              invariant="System.Data.SqlClient"
      description=".Net Framework Data Provider for SqlServer"
      type="System.Data.SqlClient.SqlClientFactory, System.Data,
      Version=2.0.0.0, Culture=neutral,
              PublicKeyToken=b77a5c561934e089" />
  </DbProviderFactories>
</system.data>
```

Megjegyzés Ha olyan data provider factory mintát szeretnénk használni, amelyhez a kapcsolódó adatbáziskezelőt a `machine.config` fájl nem tartalmazza, akkor technikailag lehetőség van arra, hogy olyan új invariáns értékeket adjunk hozzá, amelyek a globális szerelvénytár megosztott szerelvényeire mutatnak. Meg kell bizonyosodnunk arról, hogy az adatszolgáltató ADO.NET 2.0 kompatibilis, és együtt tud működni a data provider factory modellel.

Egy teljes data provider factory példa

A teljes példához hozzunk létre egy C#-konzolalkalmazást (az alkalmazás neve legyen `DataProviderFactory`), amely listázza az AutoLot adatbázis autóraktárkészletét. Az első példában az adathozzáférési logikát közvetlenül a `DataProviderFactory.exe` szerelvénybe drótozzuk be (hogy az elején minél egyszerűbb dolgunk legyen). Ahogy egyre jobban megismerjük az ADO.NET programozási modellt, elkülönítjük az adatlogikánkat egy külön .NET-kódkönyvtárba, és ezt a továbbiakban is használni fogjuk.

Először is adjunk referenciát a `System.Configuration.dll` szerelvényre, és importáljuk a `System.Configuration` névteret. Ezután szúrjunk be egy `App.config` fájlt az aktuális projektünkbe, és definiáljunk egy üres `<appSettings>` elemet. Adjunk hozzá egy új, `provider` nevű kulcsot, amely ahhoz az adatszolgáltató névtér nevéhez tartozik, amelyet meg szeretnénk kapni (`System.Data.SqlClient`). Ezen kívül definiáljunk egy kapcsolatsztringet, amely az AutoLot adatbázishoz állítja be a kapcsolatot:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <appSettings>
    <!-- Melyik szolgáltató? -->
    <add key="provider" value="System.Data.SqlClient" />
    <!-- Melyik kapcsolatsztring? -->
    <add key="cnStr" value=
                    "Data Source=(local)\SQLEXPRESS;Initial
                    Catalog=AutoLot;Integrated Security=True"
    />
  </appSettings>
</configuration>
```

Megjegyzés Később részletesebben megvizsgáljuk a kapcsolatsztringeket. Ha az Server Exploreren belül kiválasztjuk az AutoLot adatbázis ikonját, a Visual Studio 2008 Properties ablakának Connection String tulajdonságából egyszerűen kimásolható és beilleszthető a megfelelő kapcsolatsztring.

Miután van egy megfelelő `*.config` fájlunk, beolvashatjuk a szolgáltatót és a `cnStr` értékeket a `ConfigurationManager.AppSettings()` metódussal. A szolgáltatót a `DbProviderFactories.GetFactory()` metódusnak adjuk tovább, hogy megkapjuk az adatszolgáltató specifikus factorytípusát. A `cnStr` érték segítségével beállítjuk a `DbConnection` leszármazott típusának `ConnectionString` tulajdonságát. Tételezzük fel, hogy importáltuk a `System.Data` és a `System.Data.Common` névtereket. Ezek után módosítsuk a `Main()` metódust a következők szerint:

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

  // Beolvassuk a kapcsolatsztringet és a szolgáltatót
  // a *.config fájlból.
  string dp =
         ConfigurationManager.AppSettings["provider"];
  string cnStr =
         ConfigurationManager.AppSettings["cnStr"];
```

```
    // Beolvassuk a factoryszolgáltatót.
    DbProviderFactory df = DbProviderFactories.GetFactory(dp);

    // Hozzunk létre egy kapcsolatobjektumot.
    DbConnection cn = df.CreateConnection();
    Console.WriteLine("Your connection object is a: {0}",
                      cn.GetType().FullName);
    cn.ConnectionString = cnStr;
    cn.Open();

    // Hozzunk létre egy parancsobjektumot.
    DbCommand cmd = df.CreateCommand();
    Console.WriteLine("Your command object is a: {0}",
                      cmd.GetType().FullName);
    cmd.Connection = cn;
    cmd.CommandText = "Select * From Inventory";

    // Írassuk ki az adatokat az adatolvasó segítségével.
    // Mivel a CommandBehavior.CloseConnectiont választottuk, nem kell
    // explicit módon meghívni a Close() metódust a kapcsolatra.
    DbDataReader dr =
                cmd.ExecuteReader(CommandBehavior.CloseConnection);
    Console.WriteLine("Your data reader object is a: {0}",
                      dr.GetType().FullName);
    Console.WriteLine("\n***** Current Inventory *****");
    while (dr.Read())
      Console.WriteLine("-> Car #{0} is a {1}.",
              dr["CarID"], dr["Make"].ToString().Trim());
    dr.Close();
    Console.ReadLine();
  }
```

Megjegyzés Ha a `CommandBehavior.CloseConnection` értéket választottuk az `ExecuteReader()` paramétereként, akkor a kapcsolat automatikusan lezárul, ahogy az adatolvasó objektumot bezárjuk.

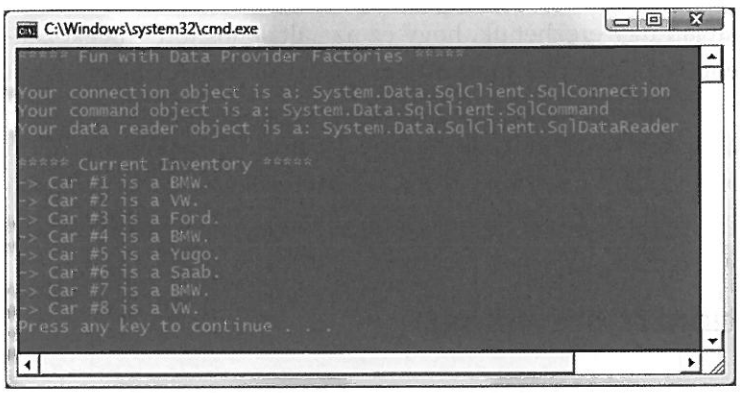
Diagnosztikai célokból a reflexiós szolgáltatás segítségével a mögöttes kapcsolat-, parancs- és adatolvasó teljesen meghatározott nevét íratjuk ki. Ha futtatjuk az alkalmazást, láthatjuk, hogy a Microsoft SQL Server szolgáltató segítségével olvasta ki a program az adatokat az AutoLot adatbázis Inventory táblájából (lásd a 22.13. ábrát).

Ha a következők szerint módosítjuk a `*.config` fájlt, hogy a `System.Data.OleDb` legyen az adatszolgáltató (és bővítjük a kapcsolatsztringet egy `Provider` szegmenssel):

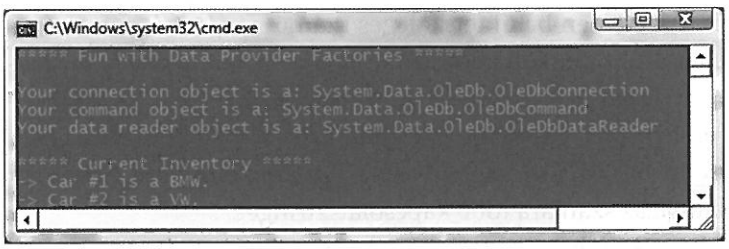
```
<configuration>
  <appSettings>
    <!--   Melyik szolgáltató? -->
    <add key="provider" value="System.Data.OleDb" />
```

```
    <!--   Melyik kapcsolatsztring?  -->
    <add key="cnStr" value=
            "Provider=SQLOLEDB;Data Source=(local)\SQLEXPRESS;
    Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </appSettings>
</configuration>
```

akkor láthatjuk, hogy a `System.Data.OleDb` típusokat használja a program a háttérben (lásd a 22.14. ábrát).



22.13. ábra: *Az SQL Server data provider factory megszerzése*



22.14. ábra: *Az OLE DB data provider factory megszerzése*

Az eddigi ADO.NET-es tapasztalataink alapján nem biztos, hogy tudjuk, pontosan mire is szolgálnak a kapcsolat-, a parancs- és az adatolvasó objektumok. Egyelőre elég annyit megértenünk, hogy az ADO.NET data provider factory modellel lehetőségünk van olyan egyszerű kódbázist írni, amely deklaratív módon használhatja a különböző adatszolgáltatókat.

A data provider factory modell lehetséges hátránya

Noha a data provider factory modell meglehetősen hatékony, biztosnak kell lennünk abban, hogy a kódbázis valóban csak azokat a típusokat és metódusokat használja, amelyek az absztrakt ősosztályok révén minden szolgáltatóban megtalálhatók. Ezért a program létrehozásakor csak olyan tagokat használhatunk, amelyek a `DbConnetion`, a `DbCommand` és a `System.Data.Common` névtér más típusaiban szerepelnek.

Mindezek alapján úgy érezhetjük, hogy ez az „általánosított" megközelítés nem teszi lehetővé, hogy az adott adatbáziskezelők egyedi lehetőségeit közvetlenül elérjük. Ha a mögöttes szolgáltató speciális tagjait kell meghívnunk (pl. az `SqlConnection` tagot), akkor erre az explicit kasztolás lehet a megoldás. Ekkor azonban a kódunk sokkal nehezebben lesz karbantartható (és kevésbé rugalmas), mivel számos futásidejű ellenőrzést kell beleépíteni.

A <connectionStrings> elem

A kapcsolatsztring-adatunk jelenleg a `*.config` fájlunk `<appSettings>` elemében szerepel. Az alkalmazáskonfigurációs fájl definiálhat egy `<connectionStrings>` elemet is. Az elemen belül tetszőleges számú név/érték párt definiálhatunk, amelyeket programozottan a memóriába olvashatunk a `ConfigurationManager.ConnectionStrings` indexelő segítségével. Ennek a megközelítésnek az egyik előnye (az `<appSettings>` elem és a `ConfigurationManager.AppSettings` indexelő használata helyett) az, hogy konzisztensen adhatunk meg egyetlen alkalmazás számára több kapcsolatsztringet.

Módosítsuk a következők szerint az `App.config` fájlunkat (figyeljük meg, hogy minden egyes kapcsolatsztringet a `name` és a `connectionString` attribútumokkal adunk meg, nem pedig a `key` és a `value` attribútumokkal, mint az `<appSettings>` esetén):

```
<configuration>
  <appSettings>
    <!--   Melyik szolgáltató? -->
    <add key="provider" value="System.Data.SqlClient" />
  </appSettings>

  <!--  Itt következnek a kapcsolatsztringek  -->
  <connectionStrings>
    <add name ="AutoLotSqlProvider" connectionString =
               "Data Source=(local)\SQLEXPRESS;
```

```
    Integrated Security=SSPI;Initial Catalog=AutoLot"/>
    <add name ="AutoLotOleDbProvider" connectionString =
              "Provider=SQLOLEDB;Data Source=(local)\SQLEXPRESS;
    Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
</configuration>
```

Ezek alapján a következőképpen módosíthatjuk a `Main()` metódusunkat:

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Data Provider Factories
                    *****\n");
  string dp =
         ConfigurationManager.AppSettings["provider"];
  string cnStr =
      ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].
                           ConnectionString;
...
}
```

A jelenlegi alkalmazásunk semleges kóddal jeleníti meg az AutoLot adatbázis Inventory táblájának tartalmát. Ha a szolgáltató nevét és a kapcsolatsztringet kiírjuk egy külső `*.config` fájlba, a data provider factory modell a háttérben dinamikusan betölti a megfelelő szolgáltatót.

Megjegyzés A könyv további példái expliciten használják a `System.Data.SqlClient` névtér típusait, hogy a továbbiakban a lényegre tudjunk összpontosítani. Ha más adatbáziskezelő rendszert szeretnénk használni (pl. Oracle-t), akkor ennek megfelelően módosítani kell a kódot.

Forráskód A DataProviderFactory kódfájlokat a forráskódkönyvtárban a 22. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv.oldalát.

Az ADO.NET kapcsolatalapú modellje

Az ADO.NET *kapcsolatalapú modellje* lehetőséget nyújt arra, hogy az adatbázissal az adatszolgáltató kapcsolat-, parancs-, és adatolvasó objektumain keresztül kommunikáljunk. Bár az előző `DataProviderFactory` alkalmazásunkban már használtuk ezeket az objektumokat, nézzük végig a folyamatot még egyszer egy kibővített példával. Ha egy adatbázishoz szeretnénk csatlakozni és egy adatolvasó objektum segítségével kiolvasni a rekordokat, a következő lépéseket kell végrehajtanunk:

1. A kapcsolatobjektum allokálása, konfigurálása és megnyitása.
2. Egy parancsobjektum allokálása és konfigurálása, a kapcsolatobjektum meghatározása a konstruktor argumentumaként vagy a `Connection` tulajdonságon keresztül.
3. `ExecuteReader()` metódus meghívása a beállított parancsobjektumon.
4. Az adatolvasó `Read()` metódusával minden egyes rekord feldolgozása.

Kiindulásként hozzunk létre az új `AutoLotDataReader` konzolalkalmazást, és importáljuk a `System.Data` és a `System.Data.SqlClient` névtereket. A cél az, hogy megnyissunk egy kapcsolatot (az `SqlConnection` objektumon keresztül), és elküldjünk egy SQL-lekérdezést (az `SqlCommand` objektumon keresztül), hogy megkapjuk az Inventory tábla összes rekordját. Ekkor az `SqlDataReader` objektumot fogjuk használni, hogy kiírassuk az eredményt a típusindexelővel. A teljes kód a `Main()` metóduson belül a következő:

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

    // Egy nyitott kapcsolat létrehozása.
    SqlConnection cn = new SqlConnection();
    cn.ConnectionString =
      @"Data Source=(local)\SQLEXPRESS;Integrated Security=SSPI;" +
      "Initial Catalog=AutoLot";
    cn.Open();

    // Egy SQL-parancsobjektum létrehozása.
    string strSQL = "Select * From Inventory";
    SqlCommand myCommand = new SqlCommand(strSQL, cn);

    // Adatolvasó beolvasása az ExecuteReader() metódussal.
    SqlDataReader myDataReader;
    myDataReader =
          myCommand.ExecuteReader(CommandBehavior.CloseConnection);

    // Lépkedjünk végig az eredményen egy ciklus segítségével.
    while (myDataReader.Read())
    {
      Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
      myDataReader["Make"].ToString().Trim(),
      myDataReader["PetName"].ToString().Trim(),
      myDataReader["Color"].ToString().Trim());
    }
```

```
      // Mivel a CommandBehavior.CloseConnection típust specifikáltuk,
      // nem szükséges explicit módon meghívni a Close() metódust
      // a kapcsolatra.
      myDataReader.Close();
      Console.ReadLine();
    }
  }
```

A kapcsolatobjektumok használata

Az első lépés, ha egy adatszolgáltatóval dolgozunk, az, hogy létrehozunk egy munkamenetet az adatforrással a kapcsolatobjektumot felhasználva (amely a `DbConnection` típusból származik). A .NET-kapcsolatobjektumok formázott *kapcsolatsztringgel* rendelkeznek, amely pontosvesszővel elválasztott név/érték párokat tartalmaz. Ezt az információt használjuk a gép nevének meghatározására, amelyikhez csatlakozni szeretnénk, a kívánt biztonsági beállítások meghatározására, a gépen található adatbázis nevének definiálására és más adatszolgáltató-specifikus információ meghatározására.

Az Initial Catalog név utal arra az adatbázisra, amelyhez a munkamenetet próbáljuk meg létrehozni. A Data Source név határozza meg a gép nevét, ahol az adatbázis található. Itt a `(local)` kulcsszó segítségével definiálhatunk egy tokent, amely meghatározza az aktuális helyi gépet (függetlenül attól, hogy az adott gépnek mi a valódi neve), míg az `\SQLEXPRESS` token tájékoztatja az SQL Server szolgáltatót, hogy az alapértelmezett SQL Server Express telepítéshez csatlakozunk (ha az AutoLot adatbázist egy SQL Server 2005 teljes vagy korábbi verzión hoztuk létre, egyszerűen adjuk meg a `Data Source = (local)` adatforrást).

Ezek kívül megadhatunk további olyan tokeneket, amelyek meghatározzák a biztonsági azonosítóadatokat. Ebben a példában az Integrated Security értékét SSPI-re állítjuk (ez az igaz értékkel egyenlő), amely az aktuális Windows felhasználói bizonyítványokat használja a felhasználói hitelesítés során.

Megjegyzés Keressük meg a .NET Framework 3.5 SDK dokumentációjában az adatszolgáltatónk kapcsolatobjektumának `ConnectionString` tulajdonságát, hogy megismerjük az adott adatbáziskezelő rendszerhez tartozó összes név/érték párt. Egy szegmens (mint pl. az Integrated Security) több, redundáns értékre is állítható (pl. az SSPI, a `true` és a `yes` ugyanúgy működik az Integrated Security értékre). Továbbá több kifejezést is találunk majd ugyanahhoz a feladathoz (pl. az Initial Catalog és a Database kifejezések felcserélhetők).

Ha létrehoztuk a felépített sztringet, az `Open()` metódus hívásával felépítjük a relációs adatbáziskezelő rendszerünkkel a kapcsolatot. A `ConnectionString`, az `Open()` és a `Close()` tagokon kívül a kapcsolatobjektum számos olyan tagot tartalmaz még, amelyekkel további beállításokat végezhetünk a kapcsolatunkon, ilyen például az időtúllépés és a tranzakciós információk. A 22.5. táblázat néhány (de nem az összes) `DbConnection` alaposztálytagot tartalmaz.

Tag	Jelentés
`BeginTransaction()`	Ez a metódus egy adatbázis-tranzakció indítására szolgál.
`ChangeDatabase()`	Ez a metódus adatbázist vált egy megnyitott kapcsolat alatt.
`ConnectionTimeout`	Ez az írásvédett tulajdonság azzal az értékkel tér vissza, hogy mennyi időt várhatunk a kapcsolat felépítésére, mielőtt félbeszakadna a próbálkozás, és hiba generálódna (az alapértelmezett érték 15 másodperc). Ha szeretnénk megváltoztatni az alapértelmezett értékét, akkor meg kell adni a kapcsolatsztringben egy „Connect Timeout" szegmenst (pl. `Connect Timeout = 30`).
`Database`	Ez a tulajdonság adja meg, hogy a kapcsolatobjektum melyik adatbázissal dolgozik.
`DataSource`	Ez a tulajdonság adja meg, hogy a kapcsolatobjektum által használt adatbázis hol található.
`GetSchema()`	Ez a metódus olyan `DataSettel` tér vissza, amely az adatforrásra vonatkozó sémainformációkat tartalmazza.
`State`	Ez a tulajdonság állítja be a kapcsolat aktuális állapotát, amelyet a `ConnectionState` felsorolás képvisel.

22.5. táblázat: *A DbConnection típus tagjai*

A `DbConnection` típus tulajdonságai tipikusan írásvédettek, és arra szolgálnak, hogy futásidőben megkapjuk a kapcsolat tulajdonságait. Ha szeretnénk felülírni az alapértelmezett beállításokat, akkor magát a felépített sztringet kell módosítani. Például a kapcsolatsztring a kapcsolat időtúllépését 15 másodpercről 30 másodpercre állítja:

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

```
    SqlConnection cn = new SqlConnection();
    cn.ConnectionString =
       @"Data Source=(local)\SQLEXPRESS;" +
       "Integrated Security=SSPI;Initial Catalog=AutoLot;
       Connect Timeout=30";
    cn.Open();

    // Új segédfüggvény (lásd lent).
    ShowConnectionStatus(cn);
    ...
}
```

Az előző kódban látható, hogy a kapcsolatobjektumunkat a `Program` osztályban paraméterként adtuk át egy új statikus segédfüggvénynek, amelynek `ShowConnetionStatus()` a neve, és a következőképpen implementáljuk (bizonyosodjunk meg róla, hogy importáltuk a `System.Data.Common` névteret, hogy megkapjuk a `DbConnection` definícióját):

```
static void ShowConnectionStatus(DbConnection cn)
{
  // Az aktuális kapcsolatobjektum különböző információinak
  // megjelenítése.
  Console.WriteLine("***** Info about your connection *****");
  Console.WriteLine("Database location: {0}", cn.DataSource);
  Console.WriteLine("Database name: {0}", cn.Database);
  Console.WriteLine("Timeout: {0}", cn.ConnectionTimeout);
  Console.WriteLine("Connection state: {0}\n", cn.State.ToString());
}
```

Míg a legtöbb tulajdonság a nevében hordozza a jelentését is, a `State` tulajdonságot érdemes külön megemlíteni. Noha ez a tulajdonság a `ConnectionState` felsorolt típus bármely értékét megkaphatja:

```
public enum ConnectionState
{
  Broken, Closed,
  Connecting, Executing,
  Fetching, Open
}
```

az érvényes értékek a `ConnectionState` érték, a `ConnectionState.Open` és a `ConnectionState.Closed` (a felsorolt típus további tagjait későbbi használatra tartjuk fenn). Mindig biztonságos egy olyan kapcsolatot bezárni, amelynek aktuális állapota `ConnectionState.Closed`.

A ConnectionStringBuilder objektumok

A kapcsolatsztringekkel programozottan dolgozni eléggé nehézkes, hiszen gyakran sztringliterálként jelennek meg, ezeket pedig nehéz kezelni, és sok a hibázási lehetőség. A Microsoft által kínált ADO.NET-adatszolgáltatók támogatják azokat a *kapcsolatsztring-építő objektumokat*, amelyek erősen típusos tulajdonságokkal lehetővé teszik a név/érték párok használatát. Nézzük meg a `Main()` metódus következő módosítását:

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

  // Hozzunk létre egy kapcsolatsztringet az építő objektum
  // segítségével.
  SqlConnectionStringBuilder cnStrBuilder =
                             new SqlConnectionStringBuilder();
  cnStrBuilder.InitialCatalog = "AutoLot";
  cnStrBuilder.DataSource = @"(local)\SQLEXPRESS";
  cnStrBuilder.ConnectTimeout = 30;
  cnStrBuilder.IntegratedSecurity = true;
  SqlConnection cn = new SqlConnection();
  cn.ConnectionString = cnStrBuilder.ConnectionString;
  cn.Open();
  ShowConnectionStatus(cn);
  ...
}
```

Ebben a lépésben létrehozunk egy `SqlConnectionStringBuilder` példányt, megfelelően beállítjuk a tulajdonságokat, és lekérjük a belső sztringet a `ConnectionString` tulajdonság segítségével. Figyeljük meg, hogy a típus alapértelmezett konstruktorát használtuk. Ha ezt a megoldást választjuk, akkor létrehozhatjuk az adatszolgáltatónk kapcsolatsztring-építő objektumának egy példányát, ha kiindulási pontként átadunk egy létező kapcsolatsztringet (ez hasznos lehet, ha az értékeket dinamikusan egy `App.config` fájlból olvassuk be). Ha feltöltöttük az objektumunkat a kezdeti sztringadatokkal, akkor a különböző név/érték párokat a kapcsolódó tulajdonságok segítségével módosíthatjuk, például:

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

```
  // Tételezzük fel, hogy a conStr valóban egy *.config fájlból
  // érkezett.
  string cnStr = @"Data Source=(local)\SQLEXPRESS;" +
                 "Integrated Security=SSPI;Initial Catalog=AutoLot";
  SqlConnectionStringBuilder cnStrBuilder =
                             new SqlConnectionStringBuilder(cnStr);

  // Módosítsuk az időtúllépés értékét.
  cnStrBuilder.ConnectTimeout = 5;
  ...
}
```

A parancsobjektumok

A kapcsolatobjektum szerepének megismerése után, a következő feladatunk azt megvizsgálni, hogyan tudunk SQL-lekérdezéseket küldeni az adatbázisnak. Az `SqlCommand` típus (amely a `DbCommand` típusból származik) egy SQL-lekérdezés, táblanév vagy tárolt eljárás objektumorientált megjelenítése. A parancs típusát a `CommandType` tulajdonsággal állíthatjuk be, amely a `CommandType` felsorolás bármely értékét felveheti:

```
public enum CommandType
{
  StoredProcedure,
  TableDirect,
  Text  // Alapértelmezett érték.
}
```

Amikor létrehozunk egy parancsobjektumot, az SQL-lekérdezést vagy konstruktorparaméterként adjuk át, vagy közvetlenül a `CommandText` tulajdonságon keresztül. A parancsobjektum létrehozásakor meg kell adni, hogy melyik kapcsolatot akarjuk használni. Ezt vagy szintén konstruktorparaméterként adhatjuk meg, vagy a `Connection` tulajdonságon keresztül:

```
static void Main(string[] args)
{
  ...
  SqlConnection cn = new SqlConnection();
  ...
  // Parancsobjektum létrehozása a konstruktorargumentumok
  // segítségével.
  string strSQL = "Select * From Inventory";
  SqlCommand myCommand = new SqlCommand(strSQL, cn);
```

```
    // Egy másik parancsobjektum létrehozása tulajdonságok
    // segítségével.
    SqlCommand testCommand = new SqlCommand();
    testCommand.Connection = cn;
    testCommand.CommandText = strSQL;
    ...
}
```

Itt nem küldtük el az SQL-lekérdezést az AutoLot adatbázisnak, hanem előkészítettük a parancsobjektum állapotát a későbbi felhasználásra. A 22.6. táblázat a `DbCommand` típus további tagjait emeli ki.

Tag	Jelentés
`CommandTimeout`	Lekérdezi vagy beállítja azt az időértéket, amennyit a rendszer a parancs futtatása közben vár, mielőtt megszakítaná a próbálkozást, és hibát generálna. Az alapértelmezett érték 30 másodperc.
`Connection`	Lekérdezi vagy beállítja a `DbCommand` példánya által használt `DbConnection` típust.
`Parameters`	Beolvassa a `DbParameter` típusok gyűjteményét, amelyeket a paraméterezett lekérdezéshez használunk.
`Cancel()`	Felfüggeszti egy parancs futását.
`ExecuteReader()`	Az adatszolgáltató `DbDataReader` objektumával tér vissza, amely egyirányú, írásvédett hozzáférést biztosít a mögöttes adatokhoz.
`ExecuteNonQuery()`	Lefuttatja a parancsot az adattárolón, és nem vár vissza semmilyen eredményt.
`ExecuteScalar()`	Az `ExecuteNonQuery()` metódus könnyített verziója, kifejezetten az egyelemű lekérdezések számára (például a rekordok számának meghatározására).
`ExecuteXmlReader()`	A Microsoft SQL Server (2000 vagy magasabb verziói) képesek az eredményhalmazt XML-formátumban visszaadni. Ez a metódus visszaad egy `System.Xml.XmlReader` típust, amely lehetővé teszi számunkra a bejövő XML-folyam feldolgozását.

Tag	Jelentés
`Prepare()`	A parancs egy előkészített (vagy lefordított) verzióját hozza létre az adatforráson. Egy előkészített lekérdezés valamelyest gyorsabban fut le, és olyankor hasznos, ha ugyanazt a lekérdezést többször szeretnénk futtatni.

22.6. táblázat: *A DbCommand típus tagjai*

Megjegyzés Ahogy azt majd a fejezet későbbi részében láthatjuk, az `SqlCommand` objektum további tagokat tartalmaz, amelyek az adatbázis-kezelés aszinkron módját teszik lehetővé.

Az adatolvasók

Miután létrehoztunk egy aktív kapcsolatot és egy SQL-parancsot, a következő lépés az, hogy elküldjük a lekérdezést az adatforráshoz. Többféleképpen megtehetjük ezt. A `DbDataReader` típus (amely az `IDataReadert` implementálja) a legegyszerűbb és a leggyorsabb módja annak, hogy információt nyerjünk ki az adattárolóból. Az adatolvasó írásvédett, egyirányú adatfolyamot képvisel, és egyszerre egy rekorddal tér vissza. Ezek alapján nyilvánvaló, hogy az adatolvasók akkor hasznosak, ha egy lekérdező SQL-utasítást küldünk el a mögöttes adattárolóba.

Az adatolvasók akkor hasznosak tehát, ha nagy adathalmazon szeretnénk végiglépkedni, és közben nincs szükség arra, hogy az adatok memóriában tárolt megjelenítése a rendelkezésünkre álljon. Ha például 20 000 rekordot szeretnénk lekérdezni egy táblából, hogy szöveges fájlban tároljuk, nagyon sok memóriát használna fel, ha ezt az információt egy `DataSet` típusban tárolnánk. Sokkal jobb megoldás, ha adatolvasót használunk, ez ugyanis nagyon gyorsan lépked végig a rekordokon. Figyeljünk azonban arra, hogy az adatolvasó objektumok (ellentétben az adatillesztővel, ezeket lásd később) az adatforrással élő kapcsolatot tartanak fenn mindaddig, amíg expliciten be nem zárjuk a munkamenetet.

Az adatolvasó objektum a parancsobjektumból nyerhető az `ExecuteReader()` metódus hívásával. Amikor meghívjuk ezt a metódust, az olvasóhoz tartozó kapcsolatobjektum `CommandBehavior.CloseConnection` tulajdonságának beállításával megadhatjuk az olvasónak, hogy automatikusan zárja le a kapcsolatot.

A következő példában az olvasó a `Read()` metódust használja annak meghatározására, hogy mikor értük el az utolsó rekordot (a `false` visszatérési érték segítségével). Minden egyes bemeneti rekordnál használjuk a típusindexelőt, hogy listázzuk minden egyes autó gyártmányát, nevét és színét. Figyeljük meg, hogy amint befejeztük a rekordok feldolgozását, meghívjuk a `Close()` metódust, hogy felszabadítsuk a kapcsolatobjektumot:

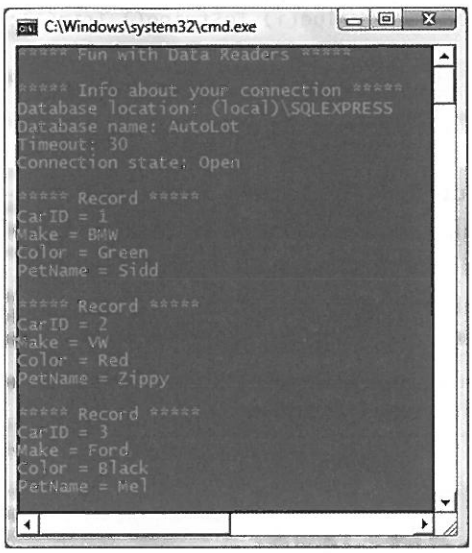
```
static void Main(string[] args)
{
  // Adatolvasó beolvasása az ExecuteReader() segítségével.
  SqlDataReader myDataReader;
  myDataReader =
        myCommand.ExecuteReader(CommandBehavior.CloseConnection);
  // Lépkedjünk végig az eredményen egy ciklus segítségével.
  while (myDataReader.Read())
  {
    Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
                      myDataReader["Make"].ToString().Trim(),
                      myDataReader["PetName"].ToString().Trim(),
                      myDataReader["Color"].ToString().Trim());
  }
  myDataReader.Close();
  Console.ReadLine();
}
```

Megjegyzés A sztringadatok csonkítását csak azért hajtjuk végre, hogy leszedjük a szóközöket az adatbázis-bejegyzésekből; ezek nem kapcsolódnak közvetlenül az ADO.NET-hez. Az, hogy szükség van-e erre a lépésre, attól függ, hogy hogyan definiáltuk az oszlopot, és milyen adatot szúrtunk be a táblába. Így erre a műveletre természetesen nem minden egyes esetben van szükség.

Az olvasó objektumindexelője túlterhelt: vagy egy sztringet (amely az oszlop nevét képviseli), vagy egy egész számot (amely az oszlop sorszámát jelzi) kaphat paraméterként. Így elegánsabbá tehetjük az aktuális olvasólogikánkat (és elkerülhetjük a bedrótozott sztringneveket), ha a következőképpen módosítjuk az eddigieket (figyeljük meg a `FieldCount` tulajdonság használatát):

```
while (myDataReader.Read())
{
  Console.WriteLine("***** Record *****");
  for (int i = 0; i < myDataReader.FieldCount; i++)
  {
    Console.WriteLine("{0} = {1} ",
                      myDataReader.GetName(i),
                      myDataReader.GetValue(i).ToString().Trim());
  }
  Console.WriteLine();
}
```

Ha lefordítjuk és futtatjuk a projektünket, az AutoLot adatbázis Inventory táblájának tartalomlistáját láthatjuk (lásd a 22.15. ábrát).



22.15. ábra: *Az adatolvasó objektumok*

Több eredményhalmaz kinyerése adatolvasóval

Az adatolvasó objektumok egyetlen parancsobjektummal több eredményhalmazt kezelnek. Ha például egyszerre szeretnénk megkapni az Inventory és a Customers tábla sorait, megadhatjuk mindkét `SELECT` utasítást pontosvesszővel elválasztva:

```
string strSQL = "Select * From Inventory;Select * from Customers";
```

Amikor megkapjuk az adatolvasó objektumot, a különböző eredményhalmazok között a `NextResult()` metódussal iterálhatunk. Figyeljünk arra, hogy automatikusan mindig az első eredményhalmazt kapjuk vissza. Ezért, ha a táblák minden során szeretnénk végiglépkedni, a következő iterációkonstruktort kell elkészítenünk:

```
do
{
  while (myDataReader.Read())
  {
    Console.WriteLine("***** Record *****");
```

```
      for (int i = 0; i < myDataReader.FieldCount; i++)
      {
        Console.WriteLine("{0} = {1}",
                          myDataReader.GetName(i),
                          myDataReader.GetValue(i).ToString().Trim());
      }
      Console.WriteLine();
    }
  } while (myDataReader.NextResult());
```

Egy adatolvasó csak egy SQL `Select` utasítást tud feldolgozni, és nem használható létező adatbázistábla adatainak a módosítására beszúrással, törléssel vagy frissítéssel. Egy létező adatbázis módosításának megértéséhez a parancsobjektum további vizsgálatára van szükség.

Forráskód Az AutoLotDataReader kódfájlokat a forráskódkönyvtárban a 22. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv.oldalát.

Újrafelhasználható adatelérési könyvtár készítése

Az `ExecuteReader()` metódus egy olyan adatolvasó objektumot biztosít, amely lehetővé teszi, hogy egy SQL-utasítás eredményét végignézzük egy egyirányú, írásvédett információfolyamban. Ha olyan SQL-utasítást szeretnénk végrehajtani, amely az adott táblát módosítja, akkor a parancsobjektum `ExecuteNonQuery()` metódusát kell meghívni. Ez a metódus beszúrásokat, módosításokat és törléseket hajt végre a parancsszöveg alapján.

Megjegyzés Technikai szempontból a *nonquery* egy olyan SQL-utasítás, amely nem tér vissza eredményhalmazzal. Tehát a `Select` utasítások lekérdezések, míg az `Insert`, az `Update` és a `Delete` utasítások nem. Ezért az `ExecuteNonQuery()` metódus egy integertípussal tér vissza, amely az érintett sorok számát adja vissza, nem pedig a rekordok egy új halmazát.

A következőkben megvizsgáljuk, hogyan kell módosítani egy létező adatbázist úgy, hogy kizárólag az `ExecuteNonQuery()` utasítást alkalmazzuk. A következő feladatunk pedig az, hogy olyan egyedi adatelérési könyvtárat hozzunk létre, amely magába foglalja az AutoLot adatbázison végrehajtott műveletek folyamatát. Egy termékszintű környezetben az ADO.NET-logikánk szinte mindig elkülönített .NET `*.dll` szerelvény, nagyon egyszerű okból: a

kód-újrafelhasználás miatt. A fejezet első példái nem így működtek, ennek oka az volt, hogy szem előtt tarthassuk az éppen tárgyalt problémaköröket. Ám időpocsékolás lenne minden egyes alkalmazáshoz, amely az AutoLot adatbázissal dolgozik, *ugyanazt* a kapcsolódási logikát, *ugyanazt* az adatolvasási logikát és *ugyanazt* a parancslogikát létrehozni.

Ha az adatelérési logikát elkülönítjük egy .NET-kódkönyvtárban, több alkalmazás, amely tetszőleges front endet használ (parancssoros, asztali, webes stb.) nyelvfüggetlenül hivatkozhat a könyvtárra. Ha tehát az adatelérési könyvtárunkat C#-ban hozzuk létre, más .NET-fejlesztők is elkészíthetik a felhasználói felületet a nekik tetsző nyelven (VB, C++/CLI stb.).

Ebben a fejezetben az adatkönyvtárunk (az `AutoLotDAL.dll`) egyetlen névteret tartalmaz (az `AutoLotConnectedLayert`), amely az AutoLot adatbázissal működik együtt az ADO.NET kapcsolatalapú modellje alapján . A következő fejezet ugyanehhez a `*.dll`-hez új névteret ad (`AutoLotDisconnectedLayer`), amelynek típusai biztosítják az AutoLot adatbázissal folytatott kommunikációt a kapcsolat nélküli modell segítségével. A könyv további részében ezt a könyvárat számos alkalmazás fogja használja.

Először is hozzunk létre egy új C#-osztálykönyvtárprojektet `AutoLotDal` névvel (ez az AutoLot adathozzáférési réteg rövidítése - AutoLot Data Access Layer), és a kezdeti C#-kódfájlt nevezzük át AutoLotConnDAL.cs-re. Következő lépésként a névteret átnevezzük AutoLotConnectedLayerre, és megváltoztatjuk a kezdeti osztályunk nevét `InventoryDAL`-ra, ugyanis ez az osztály definiálja majd az AutoLot adatbázis Inventory tábláját kezelő különböző tagokat. Végül importáljuk a következő .NET-névtereket:

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

// Az SQL-kiszolgáló adatszolgáltatóját használjuk;
// noha megengedett az ADO.NET factoryminták alkalmazása
// a nagyobb rugalmasság érdekében.
using System.Data;
using System.Data.SqlClient;

namespace AutoLotConnectedLayer
{
  public class InventoryDAL
  {
  }
}
```

Megjegyzés Ha olyan típusokat használunk, amelyek natív erőforrásokat kezelnek (pl. az adatbázis-kapcsolatot), érdemes implementálni az `IDisposable` interfészt és elkészíteni a megfelelő véglegesítőt (vö. az előző kötet 8. fejezetét). Termékszintű környezetben az olyan osztályokkal, mint az `InventoryDAL`, ezt a gyakorlatot követnénk; ezt azonban most elhagyjuk, hogy az ADO.NET lényegére koncentráljunk.

A kapcsolatlogika létrehozása

Először definiálnunk kell néhány metódust, amely lehetővé teszi a hívó számára, hogy egy érvényes kapcsolatsztring segítségével kapcsolódjon, és bontsa a kapcsolatot az adatforrással. Mivel az `AutoLotDAL.dll` szerelvényünk bedrótozottan használja a `System.Data.SqlClient` típusait, definiáljunk egy `SqlConnection` privát tagváltozót, amelyet a rendszer akkor foglal le, amikor az `InventoryDAL` objektumot létrehozzuk. Készítsünk egy `OpenConnection()` és egy `CloseConnection()` metódust, amelyek ezzel a tagváltozóval működnek együtt a következőképpen:

```
public class InventoryDAL
{
  // Ezt a tagot használja minden metódus.
  private SqlConnection sqlCn = new SqlConnection();

  public void OpenConnection(string connectionString)
  {
    sqlCn.ConnectionString = connectionString;
    sqlCn.Open();
  }

  public void CloseConnection()
  {
    sqlCn.Close();
  }
}
```

A tömörség kedvéért az `InventoryDAL` típus nem vizsgálja a lehetséges kivételeket, és nem vált ki egyedi kivételeket sem különböző körülmények között (pl. hibás kapcsolatsztring megadásakor). Ha egy ipari környezetben futó adatelérési könyvtárat kellene készítenünk, nagy valószínűséggel használnánk a strukturált kivételkezelési módszereket, hogy minden futásidejű anomáliát kezeljünk.

A beszúrást végző logika létrehozása

Az Inventory táblába olyan egyszerű új rekordot beszúrni, mint amilyen egyszerű megírni egy SQL `Insert` utasítást (a felhasználói bemenetek alapján) és meghívni az `ExecuteNonQuery()` utasítást a parancsobjektum segítségével. Ennek bemutatására adjuk az `InsertAuto()` nyilvános metódust az `InventoryDAL` típusunkhoz. A metódus négy paraméterrel rendelkezik, amelyek az Inventory tábla négy oszlopának felelnek meg (CarID, Color, Make és PetName). A bemeneti paraméterekkel hozzunk létre egy sztringet, amely beszúrja az új rekordot. Végül az `SqlConnection` objektumunk használatával futtassuk le az SQL-utasítást:

```
public void InsertAuto(int id, string color, string make,
                       string petName)
{
  // Az SQL-utasítás létrehozása és futtatása.
  string sql = string.Format("Insert Into Inventory" +
        "(CarID, Make, Color, PetName) Values" +
        "('{0}', '{1}', '{2}', '{3}')", id, make, color, petName);

  // Futtatás a kapcsolatunk segítségével.
  using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    cmd.ExecuteNonQuery();
  }
}
```

Megjegyzés SQL-utasításokat sztringösszefűzéssel létrehozni biztonsági okból elég veszélyes (gondoljunk az SQL injection jellegű támadásokra). Előnyben részesítjük ezért az olyan megoldásokat, ahol paraméterezett lekérdezéseket használunk. Rövidesen megismerkedünk a módszerrel.

A törlést végrehajtó logika létrehozása

Törölni egy létező rekordot éppen ugyanolyan egyszerű, mint beszúrni. Az `InsertAuto()` kódjától eltérően megnézünk egy fontos `try/catch` blokkot, amely azt az esetet kezeli, amikor megrendelés alatt álló autót próbálunk törölni a Customers táblából. Adjuk a következő metódust az `InventoryDAL` osztálytípushoz:

```
public void DeleteCar(int id)
{
  // Olvassuk be a törölni kívánt kocsi azonosítóját, majd töröljük.
  string sql = string.Format("Delete from Inventory where CarID =
                             '{0}'", id);
```

```
    using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
      try
      {
        cmd.ExecuteNonQuery();
      }
      catch(SqlException ex)
      {
        Exception error = new Exception("Sorry! That car is on
                                         order!", ex);
        throw error;
      }
    }
  }
```

A módosítást végző logika létrehozása

Amikor az Inventory táblának egy rekordját kell módosítani, az első nyilvánvaló kérdésünk az lesz, hogy pontosan melyik adatok módosítását tesszük lehetővé a hívó számára. Az autó színét? A nevét vagy a márkáját? Vagy az összes fent említett tulajdonságot? A hívó fél számára úgy tudnánk a teljes rugalmasságot biztosítani, ha olyan metódust hoznánk létre, amely egy sztringtípust kap bemenetként, és ez a sztring egy tetszőleges SQL-utasítást tartalmaz. Ám ez a legkockázatosabb megoldás. Ideális esetben metódusok olyan halmazát kellene segítségül hívni, amelyek különböző lehetőségeket biztosítanak a hívó számára a rekordok módosításához. A mi egyszerű adatelérési programkönyvtárunkban azonban egyetlen metódust fogunk létrehozni, ez pedig a hívó számára meghatározott autó nevét engedi módosítani:

```
public void UpdateCarPetName(int id, string newPetName)
{
  // Beolvassuk a módosítani kívánt autó azonosítóját és
  // az új nevét.
  string sql =
         string.Format("Update Inventory Set PetName = '{0}'
                        Where CarID = '{1}'", newPetName, id);
  using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    cmd.ExecuteNonQuery();
  }
}
```

A lekérdezést végrehajtó logika létrehozása

A következő metódus, amelyet a könyvtárunkhoz hozzá kell adni, a lekérdező metódus. Ahogyan azt a fejezet korábbi részében már láthattuk, az adatszolgáltató adatolvasó objektuma egy írásvédett, egyirányú szerveroldali kurzor segítségével engedi a rekordok lekérdezését. Amikor meghívjuk a `Read()` metódust, minden egyes rekordot megfelelően tudunk feldolgozni. Ám nekünk azt kell megoldanunk, hogy valamilyen módon visszaadjuk a kívánt rekordokat az alkalmazás hívórétegének.

Az egyik lehetőség az lenne, ha feltöltenénk egy többdimenziós tömböt (vagy bármilyen hasonló objektumot, pl. egy generikus `List<T>` objektumot) a `Read()` metódus által visszaadott adatokkal, és azzal térnénk vissza. A másik lehetőség (amelyet a jelenlegi példánkban választunk), hogy egy `System.Data.DataTable` objektumot adunk vissza, amely valójában része az ADO.NET kapcsolat nélküli rétegének.

A `DataTable` típus teljes leírása a következő fejezetben található, egyelőre elég annyit tudni róla, hogy a `DataTable` olyan osztálytípus, amely az adatok táblázatos formáját képviseli (mint a rács egy táblázatkezelő programban). Ennek érdekében a `DataTable` típus sorok és oszlopok gyűjteményét kezeli. Míg ezek a gyűjtemények programozottan feltölthetők, a `DataTable` típus felkínálja a `Load()` metódust, amely automatikusan feltölti ezeket a gyűjteményeket egy adatolvasó objektummal. Nézzük meg a következőt:

```
public DataTable GetAllInventory()
{
  // Ez tárolja a rekordokat.
  DataTable inv = new DataTable();

  // Készítsük elő a parancsobjektumot.
  string sql = "Select * From Inventory";
  using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    SqlDataReader dr = cmd.ExecuteReader();
    // Töltsük fel a DataTable objektumot adatokkal,
    // amelyeket az olvasó szolgáltat, és takarítsunk.
    inv.Load(dr);
    dr.Close();
  }
  return inv;
}
```

A paraméterezett parancsobjektumok

Az `InventoryDAL` típus beszúrás-, módosítás- és törléslogikái jelenleg bedrótozott sztringliterálokat tartalmaznak minden egyes SQL-lekérdezéshez. A *paraméterezett lekérdezés* SQL-paraméterek objektumként történő alkalmazását teszi lehetővé ahelyett, hogy egyszerűen szövegként használnánk őket. Az SQL-lekérdezések objektumorientált használata nemcsak csökkenti a gépelési hibák lehetőségét (az erősen típusos tulajdonságok révén), hanem a paraméterezett lekérdezések végrehajtása tipikusan gyorsabb, mint egy literál-SQL-sztring, hiszen a rendszer pontosan egyszer elemzi őket (ahelyett, hogy minden alkalommal a `CommandText` tulajdonsághoz rendelné az SQL-sztringet). Továbbá a paraméterezett lekérdezések segítenek elkerülni az SQL injection jellegű támadásokat (ez egy jól ismert adatelérési biztonsági probléma).

A paraméterezett lekérdezések támogatása érdekében az ADO.NET parancsobjektumai egyedi paraméterobjektumok gyűjteményét kezelik. Alapértelmezésben ez a gyűjtemény üres, de lehetőségünk van szabadon feltölteni tetszőleges számú paraméterobjektummal, amelyeket sorra hozzárendelhetünk az SQL-lekérdezésben a „helyőrző paraméterekhez". Ha egy paramétert szeretnénk az SQL-lekérdezésben a parancsobjektum paramétergyűjteményének egy tagjához kapcsolni, akkor lássuk el az SQL-szövegben szereplő paramétert a `@` prefixummal (legalábbis Microsoft SQL Server esetén; nem minden relációs adatbáziskezelő támogatja ezt a jelölésrendszert).

Paraméterek megadása a DbParameter típus segítségével

Mielőtt elkészítenénk egy paraméterezett lekérdezést, ismerkedjünk meg a `DbParameter` típussal (amely a szolgáltatóspecifikus paraméterobjektum ősosztálya). Az osztály számos tulajdonságot tart karban, ezek lehetővé teszik, hogy beállítsuk a paraméter nevét, méretét és adattípusát. Más jellemzőket is kezel, például a paraméter irányát (kimenő, bemenő stb.). A 22.7. táblázat bemutatja a `DbParameter` típus néhány kulcsfontosságú tulajdonságát.

Tulajdonság	Jelentés
`DbType`	Lekérdezi vagy beállítja az adatforrásból azt a natív adattípust, amelyet a CLR-adattípus képvisel.
`Direction`	Lekérdezi vagy beállítja, hogy a paraméter csak bemenő, csak kimenő, kétirányú vagy egy visszatérési értékkel rendelkező paraméter lesz-e.

Tulajdonság	Jelentés
`IsNullable`	Lekérdezi vagy beállítja, hogy a paraméter elfogad-e `null` értékeket.
`ParameterName`	Lekérdezi vagy beállítja a `DbParameter` nevét.
`Size`	Lekérdezi vagy beállítja az adatok maximális paraméterméretét (valójában csak szöveges adatoknál hasznos).
`Value`	Lekérdezi vagy beállítja a paraméter értékét.

22.7. táblázat: *A DbParameter típus kulcsfontosságú tagjai*

Hogy bemutassuk, hogyan kell egy parancsobjektum `DbParameter`-kompatibilis objektumgyűjteményét feltölteni, módosítsuk a korábbi `InsertAuto()` metódusunkat úgy, hogy használjunk benne paraméterobjektumokat (hasonló módosítást lehetne végezni a többi SQL-központú metódusainkon, de erre most nincs szükség).

```
public void InsertAuto(int id, string color, string make,
                       string petName)
{
  // Figyeljük meg a "helyőrzőket" az SQL-lekérdezésben.
  string sql = string.Format("Insert Into Inventory" +
                     "(CarID, Make, Color, PetName) Values" +
                     "(@CarID, @Make, @Color, @PetName)");

  // Ez a parancs belső paraméterekkel rendelkezik.
  using(SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
  {
    // Töltsük fel a paraméterek gyűjteményét.
    SqlParameter param = new SqlParameter();
    param.ParameterName = "@CarID";
    param.Value = id;
    param.SqlDbType = SqlDbType.Int;
    cmd.Parameters.Add(param);

    param = new SqlParameter();
    param.ParameterName = "@Make";
    param.Value = make;
    param.SqlDbType = SqlDbType.Char;
    param.Size = 10;
    cmd.Parameters.Add(param);

    param = new SqlParameter();
    param.ParameterName = "@Color";
    param.Value = color;
    param.SqlDbType = SqlDbType.Char;
    param.Size = 10;
    cmd.Parameters.Add(param);
```

```
        param = new SqlParameter();
        param.ParameterName = "@PetName";
        param.Value = petName;
        param.SqlDbType = SqlDbType.Char;
        param.Size = 10;
        cmd.Parameters.Add(param);

        cmd.ExecuteNonQuery();
    }
}
```

Az SQL-lekérdezésünk négy beágyazott helyőrző szimbólumot tartalmaz, amelyek mindegyike a `@` tokennel kezdődik. Az `SqlParameter` típus használatával lehetőségünk van minden helyőrzőt leképezni a `ParameterName` tulajdonsággal és különböző részleteket erősen típusosan meghatározni (az értékét, típusát, méretét stb.). Miután minden egyes paraméterobjektumot beállítottunk, az `Add()` metódus segítségével hozzáadjuk őket a parancsobjektum gyűjteményéhez.

Megjegyzés Különféle tulajdonságokat használtunk a paraméterobjektum létrehozásához. Tudni kell azonban, hogy a paraméterobjektum több túlterhelt konstruktort támogat, amelyek lehetővé teszik, hogy különböző tulajdonságok értékeit beállítsuk (ez egy sokkal tömörebb kódot eredményez). A Visual Studio 2008 számos grafikus tervezőt biztosít, amelyek a paraméter-központú forráskód nagy részét elkészítik számunkra (lásd a 23. fejezetet).

A paraméterezett lekérdezések létrehozásakor gyakran nagyobb mennyiségű kódot kell megírni, s ennek az eredménye, hogy sokkal egyszerűbben lehet az SQL-utasításokat programozottan beállítani, valamint jobb teljesítményt érhetünk el. Bár ezt a módszert bármikor alkalmazhatjuk, ha valamilyen SQL-utasítással dolgozunk, a paraméterezett lekérdezések leginkább tárolt eljárások hívásakor a leghasznosabbak.

Tárolt eljárások végrehajtása

A *tárolt eljárások* tulajdonképpen adatbázisban tárolt, névvel ellátott SQL-utasításokblokkok. Sorok halmazát adják vissza, vagy visszatérhetnek skaláris adattípusokkal is, és tetszőleges számú opcionális paraméterrel rendelkezhetnek. A végeredmény egy olyan feldolgozási egység, amely függvényhez hasonlóan működik, azzal a nyilvánvaló különbséggel, hogy az eljárás az adattárolón helyezkedik el, nem pedig a bináris üzleti objektumon. Pillanatnyilag az AutoLot adatbázisunk egyetlen tárolt eljárást tartalmaz, `GetPetName` névvel, ezt pedig a következőképpen hoztuk létre:

```
CREATE PROCEDURE GetPetName
@carID int,
@petName char(10) output
AS
SELECT @petName = PetName from Inventory where CarID = @carID
```

A továbbiakban nézzük meg az `InventoryDAL` típusunk utolsó metódusát, amely meghívja a tárolt eljárásunkat:

```
public string LookUpPetName(int carID)
{
  string carPetName = string.Empty;

  // A tárolt eljárás nevének beállítása.
  using (SqlCommand cmd = new SqlCommand("GetPetName", this.sqlCn))
  {
    cmd.CommandType = CommandType.StoredProcedure;

    // Bemeneti paraméter.
    SqlParameter param = new SqlParameter();
    param.ParameterName = "@carID";
    param.SqlDbType = SqlDbType.Int;
    param.Value = carID;
    param.Direction = ParameterDirection.Input;
    cmd.Parameters.Add(param);

    // Kimeneti paraméter.
    param = new SqlParameter();
    param.ParameterName = "@petName";
    param.SqlDbType = SqlDbType.Char;
    param.Size = 10;
    param.Direction = ParameterDirection.Output;
    cmd.Parameters.Add(param);

    // A tárolt eljárás futtatása.
    cmd.ExecuteNonQuery();

    // Visszatérés a kimeneti paraméterrel.
    carPetName = ((string)cmd.Parameters["@petName"].Value).Trim();
  }
  return carPetName;
}
```

Az első fontos tudnivaló a tárolt eljárások meghívásáról az, hogy a parancsobjektum SQL-utasítást (ez az alapértelmezett) *vagy* a tárolt eljárás nevét képviselheti.

Ha szeretnénk tájékoztatni a parancsobjektumot arról, hogy tárolt eljárást kell meghívnia, meg kell adnunk az eljárás nevét (a konstruktor argumentumában vagy a `CommandText` tulajdonságon keresztül), és be kell állítanunk a

`CommandType` tulajdonságot a `CommandType.StoredProcedure` értékre (ha ezt elfelejtjük beállítani, futásidejű kivételt kapunk, hiszen a parancsobjektum alapértelmezés szerint SQL-utasítást vár):

```
SqlCommand cmd = new SqlCommand("GetPetName", this.sqlCn);
cmd.CommandType = CommandType.StoredProcedure;
```

A paraméterobjektum `Direction` tulajdonságával lehet megadni, hogy a tárolt eljárásnak megadott különböző paraméterek milyen irányban közlekednek (pl. bemeneti paraméter és kimeneti paraméter). A korábbiakhoz hasonlóan minden egyes paramétert hozzáadunk a parancsobjektum paramétergyűjteményéhez:

```
// Bemeneti paraméter.
SqlParameter param = new SqlParameter();
param.ParameterName = "@carID";
param.SqlDbType = SqlDbType.Int;
param.Value = carID;
param.Direction = ParameterDirection.Input;
cmd.Parameters.Add(param);
```

Végül, ha a tárolt eljárást az `ExecuteNonQuery()` metódus hívásával hajtjuk végre, akkor a kimeneti paraméter értékét a parancsobjektum paramétergyűjteményének vizsgálatával és a megfelelő kasztolással kapjuk meg:

```
// Visszaadjuk a kimeneti paramétert.
carPetName = (string)cmd.Parameters["@petName"].Value;
```

Forráskód Az `AutoLotDAL` kódfájokat a forráskódkönyvtárban a 22. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv.oldalát.

Parancssoros front end létrehozása

Az `AutoLotDAL.dll` adatelérési könyvtár első iterációjával elkészültünk. A szerelvény használatával bármilyen front endet létrehozhatunk adataink megjelenítésére és módosítására (konzolalapú, Windows Forms-alapú, WCF-alkalmazásokat vagy egy HML-alapú webalkalmazást). Mivel még nem vizsgáltuk meg, hogyan kell grafikus felhasználói felületet készíteni, az adatkönyvtárunkat az új `AutoLotCUIClient` nevű konzolalkalmazásból teszteljük. Amikor létrehozzuk a projektet, mindenképpen hivatkozzunk az `AutoLotDAL.dll` szerelvényre, valamint a `System.Configuration.dll` szerelvényre, és módosítsuk a `using` utasításainkat a következők szerint:

```
using AutoLotConnectedLayer;
using System.Configuration;
using System.Data;
```

Szúrjunk be új `App.config` fájlt a projektbe, amely `<connectionString>` elemet tartalmaz. Az elem segítségével csatlakozzunk az AutoLot adatbázispéldányukhoz, például:

```
<configuration>
  <connectionStrings>
    <add name ="AutoLotSqlProvider" connectionString =
               "Data Source=(local)\SQLEXPRESS;" +
               "Integrated Security=SSPI;Initial Catalog=AutoLot"/>
  </connectionStrings>
</configuration>
```

A Main() metódus implementálása

A `Main()` metódus feladata, hogy a felhasználótól bekérje a kívánt műveletet, és a kérést a `switch` utasítás segítségével végrehajtsa. A program a következő parancsok meghatározását teszi lehetővé a felhasználó számára:

- `I`: Új rekord beszúrása az Inventory táblába.
- `U`: Létező rekord módosítása az Inventory táblában.
- `D`: Létező rekord törlése az Inventory táblából.
- `L`: Az aktuális raktárkészlet megjelenítése adatolvasó segítségével.
- `S`: Megjeleníti ezeket a lehetőségeket a felhasználó számára.
- `P`: Azonosító alapján kikeresi az autó nevét.
- `Q`: Kilép a programból.

Az összes lehetőséget a `Program` osztály egyik statikus metódusa kezeli. Itt található a `Main()` teljes implementációja. Figyeljük meg, hogy a `do/while` ciklusból meghívott metódusok (a `ShowInstruction()` metódus kivételével) egy `InventoryDAL` objektumot kapnak egyetlen paraméterként:

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

```
// A kapcsolatsztring lekérdezése az App.config fájlból.
string cnStr =
  ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].
                       ConnectionString;
bool userDone = false;
string userCommand = "";

// Az InventoryDAL objektumunk létrehozása.
InventoryDAL invDAL = new InventoryDAL();
invDAL.OpenConnection(cnStr);

// Bemenetet kérünk a felhasználótól,
// amíg nem nyomja meg a Q betűt.
try
{
  ShowInstructions();
  do
  {
    Console.Write("Please enter your command: ");
    userCommand = Console.ReadLine();
    Console.WriteLine();
    switch (userCommand.ToUpper())
    {
      case "I":
        InsertNewCar(invDAL);
        break;
      case "U":
        UpdateCarPetName(invDAL);
        break;
      case "D":
        DeleteCar(invDAL);
        break;
      case "L":
        ListInventory(invDAL);
        break;
      case "S":
        ShowInstructions();
        break;
      case "P":
        LookUpPetName(invDAL);
        break;
      case "Q":
        userDone = true;
        break;
      default:
        Console.WriteLine("Bad data! Try again");
        break;
    }
  } while (!userDone);
}
```

```
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
    }
    finally
    {
      invDAL.CloseConnection();
    }
  }
```

A ShowInstructions() metódus implementálása

A `ShowInstructions()` metódus pontosan azt teszi, amire a nevéből következtethetünk:

```
private static void ShowInstructions()
{
  Console.WriteLine("I: Inserts a new car.");
  Console.WriteLine("U: Updated an existing car.");
  Console.WriteLine("D: Deletes an existing car.");
  Console.WriteLine("L: List current inventory.");
  Console.WriteLine("S: Show these instructions.");
  Console.WriteLine("P: Look up pet name.");
  Console.WriteLine("Q: Quits program.");
}
```

A ListInventory() metódus implementálása

A `ListInventory()` metódus `DataTable` objektumot olvas be az `InventoryDAL` objektum `GetAllInventory()` metódusától. Ezt követően meghívunk egy (még megvalósításra váró) függvényt, a `DisplayTable()`-t:

```
private static void ListInventory(InventoryDAL invDAL)
{
  // A raktárlista lekérdezése.
  DataTable dt = invDAL.GetAllInventory();
  DisplayTable(dt);
}
```

A `DisplayTable()` segédmetódus megjeleníti a táblaadatokat a bemenetként megkapott `DataTable` Rows és `Columns` tulajdonságainak a felhasználásával (a `DataTable` objektum részleteit lásd a következő fejezetben).

```
private static void DisplayTable(DataTable dt)
{
  // Az oszlopnevek kiírása.
  for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
  {
    Console.Write(dt.Columns[curCol].ColumnName.Trim() + "\t");
  }
  Console.WriteLine("\n---------------------------------");

  // A DataTable kiírása.
  for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
  {
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Console.Write(dt.Rows[curRow][curCol].ToString().Trim() +
                                                        "\t");
    }
    Console.WriteLine();
  }
}
```

A DeleteCar() metódus implementálása

Létező autó törlésekor meg kell kérdezni a felhasználótól a törölni kívánt autó azonosítóját és az értéket átadni az `InventoryDAL` típus `DeleteCar()` metódusának:

```
private static void DeleteCar(InventoryDAL invDAL)
{
  // A törölni kívánt autó azonosítójának a beolvasása.
  Console.Write("Enter ID of Car to delete: ");
  int id = int.Parse(Console.ReadLine());

  // Arra az esetre, ha megsértenénk
  // az elsődleges kulcsot.
  try
  {
    invDAL.DeleteCar(id);
  }
  catch(Exception ex)
  {
    Console.WriteLine(ex.Message);
  }
}
```

Az InsertNewCar() metódus implementálása

Új rekord beszúrásához be kell kérni a felhasználótól a szükséges adatokat (`Console.ReadLine()` hívásokkal), és átadni az `InventoryDAL` `InsertAuto()` metódusának:

```
private static void InsertNewCar(InventoryDAL invDAL)
{
  // Először kérjük be a felhasználói adatokat.
  int newCarID;
  string newCarColor, newCarMake, newCarPetName;

  Console.Write("Enter Car ID: ");
  newCarID = int.Parse(Console.ReadLine());
  Console.Write("Enter Car Color: ");
  newCarColor = Console.ReadLine();
  Console.Write("Enter Car Make: ");
  newCarMake = Console.ReadLine();
  Console.Write("Enter Pet Name: ");
  newCarPetName = Console.ReadLine();

  // Adjuk át az adatelérési könyvtárnak.
  invDAL.InsertAuto(newCarID, newCarColor, newCarMake,
                    newCarPetName);
}
```

Az UpdateCarPetName() metódus implementálása

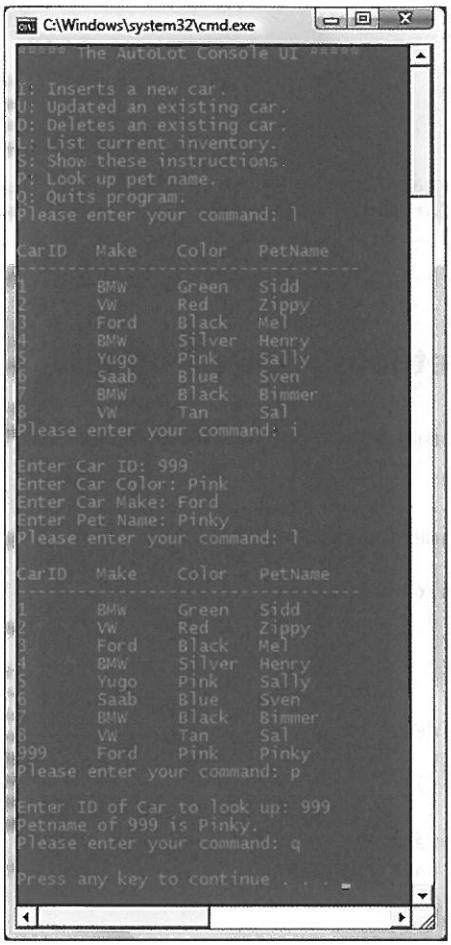
Az `UpdateCarPetName()` metódus implementálása nagyon hasonlít az előző metódushoz:

```
private static void UpdateCarPetName(InventoryDAL invDAL)
{
  // Először kérjük be a felhasználói adatokat.
  int carID;
  string newCarPetName;

  Console.Write("Enter Car ID: ");
  carID = int.Parse(Console.ReadLine());
  Console.Write("Enter New Pet Name: ");
  newCarPetName = Console.ReadLine();

  // Adjuk át az adatelérési könyvtárnak.
  invDAL.UpdateCarPetName(carID, newCarPetName);
}
```

A tárolt eljárásunk meghívása

Az autó nevének lekérdezése ugyancsak nagyon hasonlít az eddigi metódusok működéséhez, hiszen az adatelérési programkönyvtár minden alsóbb szintű ADO.NET-hívást beágyazott:

```
private static void LookUpPetName(InventoryDAL invDAL)
{
  // A keresett autó azonosítójának beolvasása.
  Console.Write("Enter ID of Car to look up: ");
  int id = int.Parse(Console.ReadLine());
  Console.WriteLine("Petname of {0} is {1}.",
                    id, invDAL.LookUpPetName(id));
}
```



22.16. ábra: *Rekordok beszúrása, módosítása és törlése parancsobjektumok segítségével*

Ezzel a parancssori front end elkészült. A 22.16. ábra egy tesztfuttatást mutat be.

Forráskód Az AutoLotCUIClient kódfájlokat a forráskódkönyvtárban a 22. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv.oldalát.

Aszinkron adatelérés az SqlCommand használatával

Minden jelenlegi adatelérési logikánk egyetlen szálon fut. A .NET 2.0 megjelenése óta az SQL-adatszolgáltató fejlődött, és támogatja az aszinkron adatbáziskezelést az `SqlCommand` újabb tagjainak a segítségével:

- `BeginExecuteReader()/EndExecuteReader()`
- `BeginExecuteNonQuery()/EndExecuteNonQuery()`
- `BeginExecuteXmlReader()/EndExecuteXmlReader()`

Az előző kötet 18. fejezetében megvizsgáltuk a metóduspárok névadási konvencióját. A .NET aszinkron metódusreferencia mintája a „begin" metódussal futtat másodlagos szálon egy feladatot, az „end" metódus pedig arra szolgál, hogy az aszinkron módon meghívott művelet eredményét beolvassa az `IAsyncResult` interfész tagjai és az opcionális `AsyncCallback` metódusreferencia segítségével. Mivel az aszinkron parancsokkal történő munka folyamata a standard metódusreferencia-minták alapján működik, ezt egy egyszerű példával mutatjuk be (a 18. fejezet részletesen tárgyalja az aszinkron metódusreferenciákat).

Tegyük fel, hogy adatolvasó objektum segítségével szeretnénk lekérdezni az Inventory tábla rekordjait egy másodlagos végrehajtási szálon. A teljesen új `AsyncCmdObjectApp` parancssori alkalmazás (amely nem használja az `InventoryDAL.dll` szerelvényünket) a következő:

Megjegyzés Ha aszinkron módon szeretnénk engedélyezni az adatelérést, módosítani kell a kapcsolatsztringet és beállítani az `Asynchronous Prosessing = true` szegmenst (az alapértelmezett érték a false).

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

  // Hozzunk létre és nyissunk meg egy aszinkron működésre
  // képes kapcsolatot.
  SqlConnection cn = new SqlConnection();
  cn.ConnectionString =
      @"Data Source=(local)\SQLEXPRESS;Integrated Security=SSPI;" +
      "Initial Catalog=AutoLot;Asynchronous Processing=true";
  cn.Open();

  // Hozzunk létre egy SQL-parancsobjektumot, amely körülbelül
  // 2 másodpercig várakozik.
  string strSQL = "WaitFor Delay '00:00:02';
                  Select * From Inventory";
  SqlCommand myCommand = new SqlCommand(strSQL, cn);

  // Futtassuk az olvasót a második szálon.
  IAsyncResult itfAsynch;
  itfAsynch =
    myCommand.BeginExecuteReader(CommandBehavior.CloseConnection);

  // Csináljunk valamit, amíg a másik szál dolgozik.
  while (!itfAsynch.IsCompleted)
  {
    Console.WriteLine("Working on main thread...");
    Thread.Sleep(1000);
  }
  Console.WriteLine();

  // Kész. Kérjük le az olvasót, és menjünk végig az eredményeken.
  SqlDataReader myDataReader =
                myCommand.EndExecuteReader(itfAsynch);
  while (myDataReader.Read())
  {
    Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
    myDataReader["Make"].ToString().Trim(),
    myDataReader["PetName"].ToString().Trim(),
    myDataReader["Color"].ToString().Trim());
  }
  myDataReader.Close();
}
```

Az első érdekes pont az, hogy a kapcsolatsztring Asynchronous Processing szegmensén keresztül engedélyezni kell az aszinkron működést. Az `SqlCommand` objektum parancsszövegébe beágyaztunk egy `WaitFor Delay` szegmenst, hogy egy hosszan futó adatbázis-műveletet szimuláljunk.

A `BeginExecuteDataReader()` hívása a kívánt `IAsyncResult`-kompatibilis típussal tért vissza, amelyet arra használunk, hogy a hívószállal szinkronizáljunk (az `IsCompleted` tulajdonságon keresztül), illetve megkapjuk az `SqlDataReadert`, ha a lekérdezés befejeződött.

Forráskód Az AsyncCmdObjectApp kódfájokat a forráskódkönyvtárban a 22. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv.oldalát.

Az adatbázis-tranzakciók

A *tranzakció* adatbázis-műveletek halmaza, amelyek *mindegyike* vagy sikeresen végrehajtódik, vagy *mindegyike* sikertelen. A tranzakciók meglehetősen fontosak, mert biztosítják a táblaadatok biztonságát, érvényességét és konzisztenciáját.

A tranzakciók nagyon lényegesek, ha az adatbázis-művelet több táblát érint, vagy több tárolt eljárás kezelését foglalja magában (vagy adatbázisatomok kombinációját). A klasszikus tranzakció mintapéldája két bankszámla közötti átutalások folyamatát mutatja be. Tételezzük fel, hogy 500 dollárt utalunk át a takarékszámlánkról a folyószámlánkra. A rendszer a következő tranzakciós lépéseket hajtja végre:

- A bank 500 dollárt levon a takarékszámlánkról.
- A bank 500 dollárt könyvel a folyószámlánkra.

Kellemetlen lenne, ha a takarékszámláról már levették volna a pénzt, de a folyószámlára még nem utalták volna át (valamilyen belső banki hiba miatt), és így elveszítenénk 500 dollárt. Ha ezeket a lépéseket adatbázis-tranzakcióban fogjuk össze, akkor az adatbáziskezelő rendszer biztosítja, hogy minden kapcsolódó lépést egyetlen egységként kezeljen. Ha a tranzakció bármely része meghiúsul, a rendszer az egész műveletet „visszagörgeti" az eredeti állapotba. Másrészről, ha minden lépés sikeres volt, a rendszer „jóváhagyja" a tranzakciót.

Megjegyzés Az ACID betűszó egy tranzakció négy kulcsfontosságú tulajdonságára utal: az atomicitásra (mindent vagy semmit), a konzisztenciára (az adat állandó marad a tranzakció során), az izolációra (a tranzakciók nem lépnek egymás „lábára") és a tartósságra (a tranzakciókat a rendszer menti és naplózza).

A .NET platform különbözőképpen támogatja a tranzakciókezelést. Számunkra most ezek közül a legfontosabb az ADO.NET-adatszolgáltató tranzakcióobjektuma (a `System.Data.SqlClient` esetén az `SqlTransaction`). A .NET-alaposztálykönyvtárak több API segítségével biztosítanak tranzakciós támogatást, ezek például a következők:

- `System.EnterpriseServices`: A névtér olyan típusokat szolgáltat, amelyekkel az alkalmazást a COM+ futtató réteggel integrálhatjuk, és ez magában foglalja a COM+ elosztott tranzakciók támogatását is.
- `System.Transactions`: A névtér osztályai lehetővé teszik, hogy saját tranzakciós alkalmazásokat és erőforrás-kezelő programokat írjunk különböző szolgáltatások számára (MSMQ, ADO.NET, COM+ stb.).
- *Windows Communication Foundation:* A WCF API tranzakciókezelést biztosító szolgáltatásokat nyújt.
- *Windows Workflow Foundations:* A WF API workflow-tevékenységek számára nyújt tranzakciós támogatást.

A .NET-alaposztálykönyvtárba beépített tranzakciós támogatás mellett az adatbáziskezelő rendszer SQL-nyelvét is alkalmazhatjuk ilyen célokra. Írhatunk például egy olyan tárolt eljárást, amely használja a `BEGIN TRANSACTION`, a `ROLLBACK` és a `COMMIT` utasításokat.

Az ADO.NET-tranzakcióobjektum kulcsfontosságú tagjai

Noha tranzakciókezelő típusok léteznek az alaposztálykönyvtárban, az ADO.NET-adatszolgáltatóban található tranzakcióobjektumok mindegyike a `DBTransaction` osztályból származik, és az `IDbTransaction` interfészt implementálja. Az `IDbTransaction` több tagot definiál:

```
public interface IDbTransaction : IDisposable
{
  IDbConnection Connection { get; }
  IsolationLevel IsolationLevel { get; }
  void Commit();
  void Rollback();
}
```

Mindenekelőtt vizsgáljuk meg a `Connection` tulajdonságot. A tulajdonság a kapcsolatobjektum referenciáját adja vissza, amely az aktuális tranzakciót indította (ahogyan majd látni fogjuk, az adott kapcsolatobjektumtól kapjuk a tranzakcióobjektumot). A `Commit()` metódust akkor hívjuk, ha minden adatbázisműveletünk sikeresen lefutott. Így minden, eddig függőben levő módosítást a rendszer rögzít az adattárolóban. A `Rollback()` metódust ellenben akkor hívjuk, ha futásidejű kivétel lép fel, amely tájékoztatja az adatbáziskezelő rendszert, hogy minden függőben levő módosítást hagyjon figyelmen kívül, és az eredeti adatokat hagyja érintetlenül.

Megjegyzés A tranzakcióobjektum IsolationLevel tulajdonságával beállíthatjuk, hogy milyen agresszíven kell védeni a tranzakciónkat a párhuzamosan futó tranzakciók műveleteivel szemben. Az alapértelmezés szerint a tranzakciók a véglegesítésig teljesen izoláltak. Az `IsolationLevel` felsorolt típus értékeivel kapcsolatban bővebb információkat a .NET Framework 3.5 SDK dokumentációjában találhatunk.

Az `IDbTransaction` interfész által definiált tagokon kívül az `SqlTransaction` típus még a `Save()` tagot definiálja, amely *mentési pontok* elhelyezését teszi lehetővé a tranzakción belül. Ez a megközelítés megoldja, hogy hiba esetén csak a megnevezett pontig kelljen visszagörgetni a tranzakciót ahelyett, hogy a rendszer az egész tranzakciót visszagörgetné. Amikor az `SqlTransaction` objektum segítségével meghívjuk a `Save()` metódust, egy barátságos sztringmonikert definiálhatunk. A `Rollback()` hívásakor argumentumként megadhatjuk ugyanezt a monikert, amelynek segítségével „részleges visszagörgetést" hajtunk végre. Ha argumentum nélkül hívjuk meg a `Rollback()` metódust, akkor a rendszer visszagörget minden függőben levő módosítást.

Tranzakciómetódus hozzáadása az InventoryDAL osztályhoz

Az ADO.NET-tranzakciók bemutatására indítsunk el a Server Explorert a Visual Studio 2008-ban, és adjuk a CreditRisks táblát az AutoLot adatbázisunkhoz. A tábla ugyanazokkal az oszlopokkal rendelkezik, mint a fejezet korábbi részében elkészített Customer tábla (CustID [ez az elsődleges kulcs], FirstName és LastNamer). Ahogyan a neve sugallja, a táblába száműzzük azokat a nemkívánatos vásárlókat, akik valamilyen okból nem feleltek meg a hitelképességi vizsgálatnak.

Megjegyzés Ezt az új tranzakciós funkcionalitást használjuk majd a 26. fejezetben, ahol a Windows Workflow Foundation API-t vizsgáljuk, ezért a CreditRisks táblát az itt leírak szerint adjuk az AutoLot adatbázishoz.

A korábban látott átutalási példához hasonlóan a nem megbízható ügyfelek átmozgatását a Customer táblából a CreditRisks táblába szintén tranzakciós hatókör felügyelete alatt kell végrehajtani (hiszen azoknak az azonosítóját és a nevét szeretnénk megjegyezni, akik nem hitelképesek). Biztosítanunk kell, hogy *vagy* sikeresen töröljük a megbízhatatlannak minősített vásárlókat a Customers táblából, és beszúrjuk őket a CreditRisks táblába, *vagy* egyik adatbázis-műveletet sem hajtuk végre.

Megmutatjuk, hogy hogyan lehet programozottan használni az ADO.NET-tranzakciókat. Nyissuk meg a fejezet korábbi részében létrehozott `AutoLotDAL` kódkönyvtárprojektünket. Adjuk hozzá az `InventoryDAL` osztályhoz az új `ProcessCreditRisk()` metódust, amely a következőképpen kezeli a nem hitelképes ügyfeleket:

```
// Az InventoryDAL osztály új tagja.
public void ProcessCreditRisk(bool throwEx, int custID)
{
  // Az ügyfél azonosítója alapján megkeressük a nevet.
  string fName = string.Empty;
  string lName = string.Empty;
  SqlCommand cmdSelect = new SqlCommand(
            string.Format("Select * from Customers
            where CustID = {0}", custID), sqlCn);
  using (SqlDataReader dr = cmdSelect.ExecuteReader())
  {
    while (dr.Read())
    {
      fName = (string)dr["FirstName"];
      lName = (string)dr["LastName"];
    }
  }

  // Hozzunk létre parancsobjektumokat, amelyek az egyes műveleti
  // lépéseket képviselik.
  SqlCommand cmdRemove = new SqlCommand(
  string.Format("Delete from Customers where CustID = {0}", custID),
                sqlCn);
  SqlCommand cmdInsert = new SqlCommand(string.Format("Insert Into
                         CreditRisks" +
                         "(CustID, FirstName, LastName) Values" +
                         "({0}, '{1}', '{2}')", custID, fName,
                          lName), sqlCn);
```

```
    // Ezt a kapcsolatobjektumtól kapjuk meg.
    SqlTransaction tx = null;
    try
    {
      tx = sqlCn.BeginTransaction();

      // Igénybe vesszük a parancsokat a tranzakcióban.
      cmdInsert.Transaction = tx;
      cmdRemove.Transaction = tx;

      // A parancsok futtatása.
      cmdInsert.ExecuteNonQuery();
      cmdRemove.ExecuteNonQuery();

      // Hiba szimulálása.
      if (throwEx)
      {
        throw new ApplicationException("Sorry! Database error!
                                        Tx failed...");
      }

      // Véglegesítsünk.
      tx.Commit();
    }
    catch (Exception ex)
    {
      Console.WriteLine(ex.Message);
      // Bármilyen hiba visszagörgeti a tranzakciót.
      tx.Rollback();
    }
  }
```

Ebben az esetben `bool` típusú bemeneti paraméterben jelöljük meg azt, hogy szeretnénk tetszőleges kivételt kiváltani, ha megpróbáljuk a megbízhatatlan vásárlót kezelni. Így könnyedén szimulálhatunk egy olyan előre nem látható körülményt, amely miatt az adatbázis-tranzakció végrehajtása sikertelen lesz. Természetesen ezt most csak a bemutató kedvéért tesszük, egy valós adatbázis-tranzakciós metódus nem engedi a hívónak, hogy tetszés szerint meghiúsítson egy logikát.

Miután megkerestük a vásárló vezeték- és keresztnevét a bemeneti custID paraméter alapján, két `SqlCommand` objektumot használunk, amelyek a tranzakció lépéseit képviselik, és a kapcsolatobjektumtól a `BeginTransaction()` metódus segítségével érvényes `SqlTransaction` objektumot kapunk. Következő fontos lépés, hogy *minden parancsobjektumnak állítsuk be* a `Transaction` tulajdonságát: mégpedig a `BeginTransaction()` eredményeként kapott tranzakcióobjektumra. Ha ezt nem tesszük meg, a beszúrás/törlés logika nem lesz benne a tranzakciós kontextusban.

Miután minden parancson meghívtuk az `ExecuteNonQuery()` metódust, kivételt jelzünk akkor (és csak akkor), ha a `bool` paraméter értéke igaz. Ebben az esetben a rendszer minden függő adatbázis-műveletet visszagörget. Ha nem jelzünk kivételt, a rendszer mindkét lépést jóváhagyja az adatbázistáblákban a `Commit()` metódus meghívásakor. Fordítsuk le a módosított AutoLotDAL projektünket, hogy elkerüljük a gépelési hibákat.

Az adatbázis-tranzakciónk tesztelése

Noha olyan új opcióval módosíthatnánk a korábban megírt `AutoLotCUICLient` alkalmazásunkat, amely meghívja a `ProcessCreditRisk()` metódust, hozzuk létre inkább az új `AdoNetTransaction` parancssori alkalmazást. Állítsunk be referenciát az `AutoLotFAL.dll` szerelvényünkre, és importáljuk az `AutoLotConnectedLayer` névteret.

Nyissuk meg a Customers táblánkat, hogy új adatokat tudjunk bevinni. Ehhez a Server Explorerben jobb egérgombbal kattintsunk a táblaikonra, és válasszuk a Show Table Data menüpontot. Adjunk hozzá egy új vásárlót, aki alacsony hitelképességi mutatóval rendelkezik, például:

- *CustID:* 333
- *FirstName:* Homer
- *LastName:* Simpson

Most módosítsuk a következőképpen a `Main()` metódusunkat.

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

  // Egyszerű módja annak, hogy megengedjük a tranzakciónak,
  // hogy sikeres legyen vagy sem.
  bool throwEx = true;
  string userAnswer = string.Empty;
  Console.Write("Do you want to throw an exception (Y or N): ");
  userAnswer = Console.ReadLine();
  if (userAnswer.ToLower() == "n")
  {
    throwEx = false;
  }
  InventoryDAL dal = new InventoryDAL();
  dal.OpenConnection(@"Data Source=(local)\SQLEXPRESS;Integrated
      Security=SSPI;" + "Initial Catalog=AutoLot");
```

```
    // Dolgozzuk fel a 333-as vásárló adatait.
    dal.ProcessCreditRisk(throwEx, 333);
    Console.ReadLine();
}
```

Ha futtatjuk a programunkat, és a kivételjezés opcióját választjuk, akkor láthatjuk, hogy Homert a rendszer *nem* törölte a Customers táblából, hiszen az egész tranzakciót visszagörgette. Ha azonban nem jelzünk kivételt, akkor a 333-as azonosítójú vásárló nincs a Customers táblában, hanem átkerült a `CreditRisks()` táblába (lásd a 22.17. ábrát).

CreditRisks: Quer...lexpress.AutoLot)

	CustID	FirstName	LastName
▶	333	Homer	Simpson
*	NULL	NULL	NULL

1 of 1

22.17. ábra: *Az adatbázis-tranzakciónk eredménye*

Forráskód Az AdoNetTransaction kódfájlokat a forráskódkönyvtárban a 22. fejezet alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv.oldalát.

Összefoglalás

Az ADO.NET a .NET platform egy olyan natív adatelérési technológiája, amelyet két különböző megközelítésben használhatunk: élő kapcsolatalapú és kapcsolat nélküli módon. Ebben a fejezetben a kapcsolatalapú modellt vizsgáltuk meg, és tanulmányoztuk az adatszolgáltatók szerepét, amelyek valójában absztrakt ősosztályok (a `System.Data.Common` névtérben) és interfésztípusok (a `System.Data` névtérben) tényleges implementációi. Ahogyan láthattuk, az ADO.NET data provider factory modell segítségével lehetőség van szolgáltatófüggetlen kód megírására.

A kapcsolatalapú modell kapcsolat-, tranzakció-, parancs- és adatolvasó objektumainak segítségével lehetőségünk nyílik rekordokat lekérdezni, módosítani, beszúrni és törölni. A parancsobjektumok belső paramétergyűjteményt támogatnak, amelynek a segítségével gondoskodhatunk SQL-lekérdezéseink típusbiztonságáról, és meglehetősen hasznosnak bizonyul a tárolt eljárások hívásakor.

HUSZONHARMADIK FEJEZET

ADO.NET, 2. rész: A bontott kapcsolat

Ebben a fejezetben az előző folytatásaként az ADO.NET *kapcsolat nélküli modelljét* ismerhetjük meg. Ennek az aspektusnak a használatával a memóriában a hívórétegen belül modellezhetjük az adatbázisadatokat a `System.Data` névtér tagjai (főként a `DataSet`, a `DataTable`, a `DataRow`, a `DataColumn`, a `DataView` és a `DataRelation`) segítségével. Így azt az illúziót kelthetjük, hogy a hívóréteg folyamatosan kapcsolatot tart fenn az adatforrással, ám a valóságban egyszerűen a relációs adatok helyi másolatát használja.

Noha az ADO.NET-nek ezt a „kapcsolat nélküli" aspektusát úgy is lehet használni, hogy egyáltalán nem hozunk létre valódi kapcsolatot egy relációs adatbázissal, ám leggyakrabban az adatszolgáltatónk adatillesztő objektumával fogunk feltöltött `DataSet` objektumokat megszerezni. Az adatillesztő objektumok hídként működnek a kliensréteg és egy relációs adatbázis között. Ezekkel az objektumokkal megszerezhetünk `DataSet` objektumokat, kezelhetjük a tartalmukat, és a módosított sorokat feldolgozásra visszaküldhetjük. A végeredmény egy pontosan skálázható, adatcentrikus .NET-alkalmazás lesz.

A kapcsolat nélküli modell előnyeinek bemutatásához új névterekkel egészítjük ki a 22. fejezetben létrehozott `AutoLotDAL.dll` adatkönyvtárat, amely a kapcsolat nélküli modell nyújtotta típusokat használja. Megvizsgáljuk az adatkötés és a Windows Forms API-n belül lévő felhasználóifelület-elemek szerepét: ezek teszik lehetővé a kliensoldali lokális adatok megjelenítését és módosítását. Végül megvizsgáljuk az erősen típusos `DataSet` objektumok funkcióját, és megnézzük, hogyan lehet egy objektumorientáltabb modell segítségével felhasználni őket adatok kinyeréséhez.

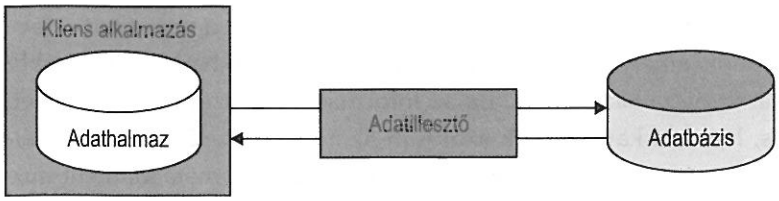
Az ADO.NET kapcsolat nélküli modellje

A kapcsolatalapú modell egy adatbázissal a munkát az elsődleges kapcsolat-, a parancs- és az adatolvasó objektumok révén teszi lehetővé. Ezekkel a típusokkal tetszés szerint kiválaszthatunk, beszúrhatunk, módosíthatunk és törölhetünk rekordokat (valamint hívhatunk tárolt eljárásokat). Ezzel az ADO.NET-nek csak az egyik oldalát ismertük meg. Az ADO.NET-objektummodellt ugyanis kapcsolat nélkül is használhatjuk.

A kapcsolatot nem igénylő típusokkal a memóriában elhelyezett objektummodell révén is modellezhetjük a relációs adatokat. A sorok és az oszlopok táblázatos blokkjának egyszerű modellezése mellett a `System.Data` típusai lehetővé teszik, hogy táblázatkapcsolatokat, oszlopmegszorításokat, elsődleges kulcsokat, nézeteket és egyéb adatbázis-primitíveket is ábrázoljunk. Az adatok modellezése után alkalmazhatunk szűrőket, futtathatunk memóriabeli lekérdezéseket, és menthetünk (vagy betölthetünk) adatokat XML- és bináris formátumban is. Mindezekhez nem kell valódi kapcsolatot létrehoznunk egy DBMS-hez sem (innen jön a *kapcsolat nélküli modell* kifejezés).

Jóllehet valóban használhatjuk a kapcsolat nélküli típusokat anélkül, hogy csatlakoznánk egy adatbázishoz, többnyire mégis használjuk a kapcsolat- és a parancsobjektumokat. Emellett egy bizonyos objektum, az *adatillesztő* (amely kibővíti az absztrakt `DbDataAdapter` osztályt) segítségével kérhetünk le és módosíthatunk adatokat. A kapcsolatalapú modellel ellentétben az adatillesztőn keresztül érkezett adatokat a rendszer nem adatolvasó objektumokkal dolgozza fel, hanem ehelyett az adatillesztő objektumok a `DataSet` objektumokat használják adatok mozgatására a hívó és az adatforrás között. A `DataSet` típus olyan tetszőleges számú `DataTable` objektum számára szolgál tárolóként, amelyek mindegyike `DataRow` és `DataColumn` objektumok gyűjteménye.

Az adatszolgáltatónk adatillesztő objektuma automatikusan kezeli az adatbázis-kapcsolatot. A skálázhatóság tökéletesítése érdekében az adatillesztők a lehető legkevesebb ideig tartják nyitva ezt a kapcsolatot. Amint a hívó megkapja a `DataSet` objektumot, a hívóréteg bontja a kapcsolatot az adatbázissal, és mindössze a távoli adatok helyi másolatát őrzi meg. A hívó egy adott `DataTable` objektumban nyugodtan beszúrhat, törölhet vagy módosíthat sorokat, de maga az adatbázis nem frissül addig, amíg a hívó explicit módon át nem adja frissítésre a `DataSet` típust az adatillesztőnek. Röviden, a `DataSet` típusok azt a látszatot keltik a kliens számára, hogy azok folyamatosan kapcsolatban állnak az adatbázissal, ám valójában egy memóriában tárolt adatbázissal dolgoznak (lásd a 23.1. ábrát).

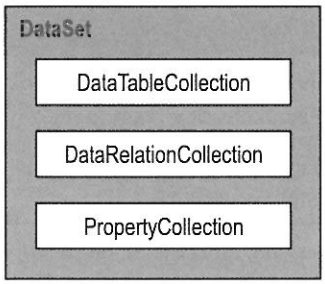


23.1. ábra: *Az adatillesztő objektumok mozgatják a DataSet típusokat az kliensrétegben*

Mivel a kapcsolat nélküli modell legfontosabb eleme a `DataSet` típus, ezért ebben a fejezetben először azt vizsgáljuk meg, hogy hogyan kezelhetjük manuálisan a típust. Ezután nem okoz majd problémát az adatillesztő objektumtól lekért `DataSet` tartalmának a kezelése.

A DataSet szerepe

A `DataSet` a relációs adatok memóriabeli megjelenítése. Pontosabban, a `DataSet` olyan osztálytípus, amely három belső, erősen típusos gyűjteményt (lásd a 23.2. ábrát) tart karban.



23.2. ábra: *A DataSet anatómiája*

A `DataSet` típus `Tables` tulajdonsága teszi lehetővé, hogy hozzáférjünk az egyes `DataTable` objektumokat tartalmazó `DataTableCollection` gyűjteményhez. A `DataSet` által használt másik fontos gyűjtemény a `DataRelationCollection`. Mivel a `DataSet` egy adatbázisséma leválasztott változata, segítségével programozottan megjeleníthetjük a szülő/gyermek kapcsolatot az adatbázis táblái között. A `DataRelation` típus használatával például létre lehet hozni olyan kapcsolatot két táblázat között, amely idegenkulcs-korlátozást modellez. Ezt az objektumot aztán a `Relations` tulajdonság segítségével hozzáadhatjuk a `DataRelationCollection` gyűjteményhez. Ezután már tudunk navigálni a két összekapcsolt tábla között, adatkereséskor a későbbiekben megnézzük ennek mikéntjét is.

Az `ExtendedProperties` tulajdonság hozzáférést biztosít a `PropertyCollection` objektumhoz, amely név/érték párként lehetővé teszi bármilyen további információ társítását a `DataSethez`. Ez az információ szó szerint bármi lehet, még akkor is, ha nem kapcsolódik szorosan az adatokhoz. Társíthatjuk például a cégünk nevét a `DataSethez`, amely így memóriabeli metaadatként működhet. A bővítő tulajdonságok további példái az időbélyegek, egy olyan titkosított jelszó, amelyet meg kell adni, ha szeretnénk hozzáférni a `DataSet` tartalmához, egy szám, amely az adatfrissítési sebességet jelképezi stb.

Megjegyzés A `DataTable` osztály is támogatja a bővítő tulajdonságokat az `ExtendedProperties` tulajdonság révén.

A DataSet alapvető tulajdonságai

Először nézzük meg a `DataSet` néhány fő tagját. A `Tables`, a `Relations` és az `ExtendedProperties` tulajdonságok mellett a 23.1. táblázat ismertet néhány további érdekes tulajdonságot.

Tulajdonság	Jelentés
`CaseSensitive`	Azt jelzi, hogy a sztring-összehasonlítások a `DataTable` objektumokban kis- és nagybetű érzékenyek-e (vagy sem).
`DataSetName`	A `DataSet` barátságos nevét képviseli. Ezt az értéket általában konstruktorparaméterként vezetjük be.
`EnforceConstraints`	Olyan értéket ad meg vagy vesz fel, amely jelzi, hogy valamilyen frissítési művelet végrehajtása során követni kell a korlátozási szabályokat.
`HasErrors`	Felvesz egy értéket, amely jelzi, hogy vannak-e hibák a `DataSet` bármelyik `DataTables` objektumának bármelyik sorában.
`Remoting Format`	Segítségével megadhatjuk, hogy a `DataSet` hogyan sorosítsa tartalmát (bináris vagy XML-formában).

23.1. táblázat: *A DataSet tulajdonságai*

A DataSet kulcsfontosságú metódusai

A `DataSet` metódusai a fent említett tulajdonságokkal működnek együtt. Amellett, hogy képes XML-folyamokkal dolgozni, a `DataSet` olyan metódusokat biztosít, amelyekkel átmásolhatjuk a `DataSet` tartalmát, navigálhatunk a belső táblák között, és megállapíthatjuk egy frissítési köteg kezdő- és végpontjait. A 23.2. táblázat bemutat néhányat az alapvető metódusok közül.

Metódusok	Jelentés
`AcceptChanges()`	Jóváhagyja a `DataSet` betöltése vagy az `AcceptChanges()` legutóbbi meghívása óta végrehajtott összes módosítást.
`Clear()`	Teljesen törli a `DataSet` adatait, eltávolít minden sort minden egyes `DataTable` objektumból.
`Clone()`	Klónozza a `DataSet` struktúráját, beleértve minden `DataTable` objektumot, valamint az összes kapcsolatot és korlátozást.
`Copy()`	Lemásolja a `DataSet` struktúráját és adatait.
`GetChanges()`	Visszaadja a `DataSet` egy olyan másolatát, amely tartalmaz minden módosítást annak legutóbbi betöltése vagy az `AcceptChanges()` meghívása óta.
`GetChildRelations()`	Visszaadja azoknak a gyermekkapcsolatoknak a gyűjteményét, amelyek az adott táblához tartoznak.
`GetParentRelations()`	Visszaadja azoknak a szülőkapcsolatoknak a gyűjteményét, amelyek az adott táblához tartoznak.
`HasChanges()`	Visszaadja, hogy a `DataSet` változott-e, beleértve az új, a törölt és a módosított sorokat is.
`Merge()`	Egyesíti ezt a `DataSet` típust egy megadott `DataSet` típussal.
`ReadXml()` `ReadXmlSchema()`	Lehetővé teszi XML-adatok beolvasását egy folyamból (fájlalapú, memóriaalapú vagy hálózatalapú) a `DataSet`be.
`RejectChanges()`	Visszagörgeti a `DataSet` típuson a létrehozása vagy az `AcceptChanges()` legutóbbi meghívása óta végrehajtott összes módosítást.
`WriteXml()` `WriteXmlSchema()`	Lehetővé teszi, hogy kiírjuk az `DataSet` tartalmát egy XML-folyamba.

23.2. táblázat: *A DataSet metódusai*

DataSet létrehozása

Készítsünk egy új parancssori alkalmazást `SimpleDataSet` névvel. A `Main()` metóduson belül hozzunk létre egy új `DataSet` objektumot, amely három bővítő tulajdonságot tartalmaz: a cégünk nevét, egy egyedi azonosítót (`System.Guid` típus formájában) és egy időbélyeget (ne felejtsük el importálni a `System.Data` névteret):

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

  // Hozzuk létre a DataSet objektumot, és adjunk hozzá néhány
  // tulajdonságot.
  DataSet carsInventoryDS = new DataSet("Car Inventory");

  carsInventoryDS.ExtendedProperties["TimeStamp"] = DateTime.Now;
  carsInventoryDS.ExtendedProperties["DataSetID"] = Guid.NewGuid();
  carsInventoryDS.ExtendedProperties["Company"] =
                                    "Intertech Training";
  Console.ReadLine();
}
```

Ha nem vagyunk járatosak a globálisan egyedi azonosítók (GUID) világában, akkor elég most annyi, hogy a GUID egy egyedi 128 bites szám. Jóllehet ezeket széles körben használják a COM keretrendszerben számos COM-atom (osztályok, interfészek, alkalmazások stb.) azonosítására, a System.GUID típus továbbra is nagyon hasznos lehet a .NET alatt akkor, ha gyorsan kell egyedi azonosítót generálnunk.

A következő feladatunk az lesz, hogy megvizsgáljuk a `DataTable` belső felépítését, és megismerkedjünk a `DataColumn` típussal.

DataColumn típusok használata

A `DataColumn` típus egyetlen oszlopot képvisel egy `DataTable` objektumon belül. Az adott `DataTable` objektumhoz tartozó összes `DataColumn` jelképezi egy tábla *séma*információinak az alapját. Ha például modelleznénk az AutoLot adatbázis Inventory tábláját (lásd a 22. fejezetet), akkor négy `DataColumn` típust hoznánk létre, egyet-egyet minden oszlophoz (CarID, Make, Color és PetName). Amint létrehoztuk a `DataColumn` objektumainkat, tipikusan hozzá szoktuk őket adni a `DataTable` típus oszlopgyűjteményéhez (a `Columns` tulajdonság révén).

Egy adatbázis-tábla adott oszlopához kényszereket rendelhetünk (pl. beállítás elsődleges kulcsként, alapértelmezett érték hozzárendelése, írásvédelem beállítása stb.). Emellett egy tábla minden oszlopa leképezhető egy alapadattípusra. Az Inventory táblázat sémája például megköveteli, hogy a CarID oszlop egy egész szám legyen, míg a Make, a Color és a PetName karakterek sorozata. A `DataColumn` osztály számos tulajdonsággal rendelkezik, amelyekkel beállíthatjuk ezeket a jellemzőket. A 23.3. táblázat néhány alapvető tulajdonság leírását tartalmazza.

Tulajdonságok	Jelentés
`AllowDBNull`	Ez a tulajdonság azt jelzi, hogy egy sorban az oszlop tartalmazhat-e `null` értéket, vagy sem. Az alapértelmezett érték: true.

`AutoIncrement`
`AutoIncrementSeed`
`AutoIncrementStep`
	Ezekkel a tulajdonságokkal lehet beállítani, hogy egy adott oszlop értéke automatikusan növekedjen. Ez akkor lehet hasznos, ha biztosítani szeretnénk az egyedi értékeket egy adott `DataColumn` típusban (pl. az elsődleges kulcsnál). Az alapértelmezés szerint a `DataColumn` nem támogatja az automatikus növelést.

`Caption`	Ez a tulajdonság az oszlop címét kérdezi le vagy állítja be. Lehetővé teszi, hogy meghatározzuk egy adatbázisoszlop felhasználóbarát nevét.
`ColumnMapping`	Ez a tulajdonság határozza meg, hogy a rendszer hogyan ábrázoljon egy `DataColumn` objektumot, ha a `DataSet` típust XML-dokumentumként a `DataSet.WriteXml()` metódus segítségével mentettük.
`ColumnName`	Ez a tulajdonság kérdezi le vagy állítja be a `Columns` gyűjteményben lévő oszlop nevét (azt mutatja meg, hogyan jeleníti meg belsőleg a `DataTable`). Ha expliciten nem állítjuk be a `ColumnName` tulajdonságot, akkor az alapértelmezett érték a Column *(n + 1)* numerikus utótaggal (vagyis Column1, Column2, Column3 stb.).
`DataType`	Ez a tulajdonság határozza meg az oszlopban tárolt adattípust (`boolean`, `sztring`, `float` stb.).
`DefaultValue`	Ez a tulajdonság új sorok beszúrásakor az oszlophoz rendelt alapértelmezett értéket kérdezi le vagy állítja be. A rendszer ezt az értéket használja, ha csak nem rendelkezünk másként.

Tulajdonságok	Jelentés
`Expression`	Ez a tulajdonság kérdezi le vagy állítja be a sorok szűrésére, egy oszlop értékének kiszámítására vagy egy összegző oszlop létrehozására használt kifejezést.
`Ordinal`	Ez a tulajdonság veszi fel az oszlop numerikus pozícióját a `DataTable` által fenntartott `Columns` gyűjteményben.
`ReadOnly`	Ez a tulajdonság dönti el, hogy az oszlop módosítható-e, ha egy új sort adtunk a táblához. Az alapértelmezett érték: `false`.
`Table`	Ez a tulajdonság kérdezi le a `DataColumn` típust tartalmazó `DataTable` objektumot.
`Unique`	Ez a tulajdonság kérdezi le vagy állítja be, hogy az oszlop egyes soraiban lévő értékeknek egyedinek kell lenniük, vagy az értékek ismétlődése megengedett. Ha egy oszlop rendelkezik egy elsődlegeskulcs-korlátozással, akkor a Unique tulajdonságot `true` értékre ajánlott állítani.

23.3. táblázat: *A DataColumn tulajdonságai*

A DataColumn létrehozása

A SimpleDataSet projekt folytatásához (és a `DataColumn` használatának bemutatásához) tételezzük fel, hogy szeretnénk modellezni az Inventory tábla oszlopait. Mivel a CarID oszlop lesz a tábla elsődleges kulcsa, a `DataColumn` objektumon írásvédett, egyedi és nem nullázható jellemzőket kell beállítanunk (a `ReadOnly`, a `Unique` és az `AllowDBNull` tulajdonságokkal). Módosítsuk a `Main()` metódust úgy, hogy létrehozzunk benne négy `DataColumn` objektumot:

```
static void Main(string[] args)
{
  ...
  // Hozzunk létre adatoszlopokat, amelyek leképezik
  // az "igazi" oszlopokat az AutoLot adatbázis
  // Inventory táblájában.
  DataColumn carIDColumn = new DataColumn("CarID", typeof(int));
  carIDColumn.Caption = "Car ID";
  carIDColumn.ReadOnly = true;
  carIDColumn.AllowDBNull = false;
  carIDColumn.Unique = true;
```

```
  DataColumn carMakeColumn = new DataColumn("Make", typeof(string));
  DataColumn carColorColumn =
              new DataColumn("Color", typeof(string));
  DataColumn carPetNameColumn =
              new DataColumn("PetName", typeof(string));
  carPetNameColumn.Caption = "Pet Name";
  Console.ReadLine();
}
```

A `carIDColumn` objektum konfigurálásakor hozzárendeltünk egy értéket a `Caption` tulajdonsághoz. Ez a tulajdonság azért hasznos, mert lehetővé teszi, hogy a megjelenítéshez más sztringértéket használjunk, mint az oszlop neve. (Az adatbázistáblákban szereplő oszlopnevek általában jobban megfelelnek a programozási céloknak [pl. au_fname], mint a megjelenítési céloknak [pl. Author First Name]).

A mezők automatikus növelésének engedélyezése

A `DataColumn` egyik olyan jellemzője, amelyet konfigurálhatunk, az *automatikus növelés* képessége. Az automatikusan növekvő oszlopot annak biztosítására használjuk, hogy amikor egy új sort hozzáadunk egy adott táblához, az oszlop értéke automatikusan beállítódjon a sorban következő értékre. Ez hasznos lehet akkor, ha azt akarjuk, hogy az oszlopban ne legyenek ismétlődő értékek (pl. egy elsődleges kulcs esetén).

Ezt a viselkedést az `AutoIncrement`, az `AutoIncrementSeed` és az `AutoIncrementStep` tulajdonságokkal irányíthatjuk. A `seed` értékkel jelöljük az oszlop kezdőértékét, míg a `step` érték határozza meg azt a számot, amelyet hozzá kell adni a növeléskor. Nézzük meg a következő módosítást a `carIDColumn` `DataColumn` szerkezetében:

```
static void Main(string[] args)
{
  ...
  DataColumn carIDColumn = new DataColumn("CarID", typeof(int));
  carIDColumn.ReadOnly = true;
  carIDColumn.Caption = "Car ID";
  carIDColumn.AllowDBNull = false;
  carIDColumn.Unique = true;
  carIDColumn.AutoIncrement = true;
  carIDColumn.AutoIncrementSeed = 0;
  carIDColumn.AutoIncrementStep = 1;
  ...
}
```

A `carIDColumn` objektum beállításai biztosítják, hogy ha sorokat adunk a megfelelő táblához, akkor a CarID oszlopban az érték 1-gyel növekedjen. Mivel a kezdőérték 0, ezért ennek az oszlopnak a számozása 0, 1, 2, 3 stb. lesz.

DataColumn objektumok hozzáadása egy DataTable típushoz

A `DataColumn` típus általában nem önálló entitás, hanem egy kapcsolódó `DataTable` objektumba illeszkedik. Ennek bemutatásához hozzunk létre új `DataTable` típust, majd szúrjuk be az oszlopgyűjteményben található összes `DataColumn` objektumot a `Columns` tulajdonság alkalmazásával:

```
static void Main(string[] args)
{
  ...
  // DataColumn objektumok hozzáadása egy DataTable típushoz.
  DataTable inventoryTable = new DataTable("Inventory");
  inventoryTable.Columns.AddRange(new DataColumn[]
                { carIDColumn, carMakeColumn, carColorColumn,
                  carPetNameColumn });
  Console.ReadLine();
}
```

A `DataTable` objektum jelenleg négy `DataColumn` objektumot tartalmaz, amelyek a memóriabeli Inventory tábla sémáját jelképezik. A tábla azonban most nem tartalmaz adatokat, és jelenleg a `DataSet` által fenntartott táblázatgyűjteményen kívül található. A továbbiakat a táblázat feltöltésével kezdjük a `DataRow` objektumok révén.

DataRow típusok használata

A `DataColumn` objektumok gyűjteménye egy `DataTable` sémáját jelképezi. Ezzel szemben a `DataRow` típusok gyűjteménye a táblázat tényleges adatait képviseli. Ezért, ha 20 tétel szerepel az AutoLot adatbázis Inventory táblájában, akkor ezeket a rekordokat 20 `DataRow` típussal reprezentálhatjuk. A `DataRow` osztály tagjainak használatával beilleszthetjük, eltávolíthatjuk, kiértékelhetjük és manipulálhatjuk a tábla értékeit. A 23.4. táblázat összefoglalja a `DataRow` típus néhány (de nem az összes) tagját.

Tagok	Jelentés

`HasErrors`
`GetColumnsInError()`
`GetColumnError()`
`ClearErrors()`
`RowError`
	A `HasErrors` tulajdonság egy olyan logikai értéket ad vissza, amely azt jelzi, hogy vannak-e hibák. Ha igen, akkor a `GetColumnsInError()` metódussal megkaphatjuk a hibás tagokat, a `GetColumnError()` metódussal a hibaleírást, míg a `ClearErrors()` metódus törli a sor hibalistáját. A `RowError` tulajdonsággal beállíthatunk szöveges leírást egy adott sor hibájáról.

`ItemArray`	Ez a tulajdonság lekérdezi vagy beállítja a sorhoz tartozó összes értéket objektumtömbök használatával.
`RowState`	Ezzel a tulajdonsággal állapíthatjuk meg a `DataRow` aktuális „állapotát" a `RowState` felsorolt típus értékeinek az alkalmazásával.
`Table`	Ezzel a tulajdonsággal kaphatjuk meg az a `DataRow` típust tartalmazó `DataTable` objektum referenciáját.
`AcceptChanges()` `RejectChanges()`	Ezekkel a metódusokkal jóváhagyhatunk vagy elvethetünk az `AcceptChanges()` legutóbbi meghívása óta a soron végrehajtott minden módosítást.

`BeginEdit()`
`EndEdit()`
`CancelEdit()`
	Ezekkel a metódusokkal megkezdhetünk, befejezhetünk vagy megszakíthatunk szerkesztési műveleteket egy `DataRow` objektumon.

`Delete()`	Ez a metódus megjelöli a sort, hogy a következő `AcceptChanges()` metódushíváskor törölni kell.
`IsNull()`	Ez a metódus lekérdezi, hogy az adott oszlop tartalmaz-e `null` értéket.

23.4. táblázat: *A DataRow típus kulcsfontosságú tagjai*

A `DataRow` típusokkal egy kicsit másképp kell dolgozni, mint a `DataColumn` típusokkal, ugyanis nem hozhatjuk létre a típus közvetlen példányát, mivel nincsen nyilvános konstruktor:

```
// Hiba! Nincs nyilvános konstruktor.
DataRow r = new DataRow();
```

Ehelyett be kell szereznünk egy `DataRow` referenciát egy adott `DataTable` objektumból. Tegyük fel, hogy szeretnénk beszúrni két sort az Inventory táblába. A `DataTable.NewRow()` metódus lehetővé teszi a táblázat következő „rekeszé-

nek" megszerzését, és ekkor a típusindexelővel feltölthetjük az oszlopokat az új adatokkal. Ilyenkor megadhatjuk akár a DataColumnhoz rendelt sztringnevet, akár annak a sorszámot:

```
static void Main(string[] args)
{
  ...
  // Most adjunk hozzá néhány sort az Inventory táblához.
  DataRow carRow = inventoryTable.NewRow();
  carRow["Make"] = "BMW";
  carRow["Color"] = "Black";
  carRow["PetName"] = "Hamlet";
  inventoryTable.Rows.Add(carRow);
  carRow = inventoryTable.NewRow();

  // A 0. oszlop az automatikusan megnövelt azonosítómező,
  // úgyhogy kezdjünk 1-gyel.
  carRow[1] = "Saab";
  carRow[2] = "Red";
  carRow[3] = "Sea Breeze";
  inventoryTable.Rows.Add(carRow);
  Console.ReadLine();
}
```

Megjegyzés Ha érvénytelen oszlopnevet vagy sorszámot adunk át a DataRow indexelőjének, akkor futásidejű kivételt kapunk.

Jelenleg egy két sort tartalmazó DataTable objektummal rendelkezünk. Ezt az általános eljárást (több DataTable objektum létrehozásához) természetesen megismételhetjük azért, hogy meghatározzuk a sémát és az adattartalmat. Mielőtt beillesztenénk az inventoryTable objektumunkat a DataSet objektumba, vizsgáljuk meg a RowState tulajdonságot.

A RowState tulajdonság

A RowState tulajdonság akkor hasznos, ha programozottan kell azonosítanunk egy táblában az összes olyan sort, amelynek az eredeti értéke megváltozott, amely újonnan lett beszúrva, és így tovább. Ehhez a tulajdonsághoz hozzárendelhetünk bármilyen értéket a DataRowState felsorolásból, amely a 23.5. táblázatban látható.

Érték	Jelentés
Added	A sort hozzáadtuk egy DataRowCollection gyűjteményhez, az AcceptChanges() metódust pedig még nem hívtuk meg.
Deleted	A sor törlésre ki lett jelölve a DataRow Delete() metódusa révén.
Detached	A sort ugyan létrehoztuk, de ez nem része egy DataRowCollection gyűjteménynek sem. Egy DataRow objektum közvetlenül azután található ebben az állapotban, hogy létrehoztuk, de még nem adtuk hozzá egyik gyűjteményhez sem, illetve ha eltávolítottuk egy gyűjteményből.
Modified	A sor módosult, de az AcceptChanges() metódust még nem hívtuk meg.
Unchanged	A sor nem változott az AcceptChanges() legutóbbi meghívása óta.

23.5. táblázat: *A DataRowState felsorolt típus értékei*

Míg egy adott DataTable sorait programozottan kezelhetjük, a RowState tulajdonság automatikusan beállítódik. Ennek példájaként adjunk hozzá egy olyan új metódust a Program osztályunkhoz, amely egy helyi DataRow objektummal dolgozik, és menetközben kiírja a sor állapotát:

```
private static void ManipulateDataRowState()
{
  // Hozzunk létre egy DataTable objektumot tesztelési célból.
  DataTable temp = new DataTable("Temp");
  temp.Columns.Add(new DataColumn("TempColumn", typeof(int)));

  // RowState = Detached.
  DataRow row = temp.NewRow();
  Console.WriteLine("After calling NewRow(): {0}", row.RowState);

  // RowState = Added.
  temp.Rows.Add(row);
  Console.WriteLine("After calling Rows.Add(): {0}", row.RowState);

  // RowState = Added.
  row["TempColumn"] = 10;
  Console.WriteLine("After first assignment: {0}", row.RowState);

  // RowState = Unchanged.
  temp.AcceptChanges();
  Console.WriteLine("After calling AcceptChanges: {0}",
                    row.RowState);

  // RowState = Modified.
  row["TempColumn"] = 11;
  Console.WriteLine("After first assignment: {0}", row.RowState);
```

```
    // RowState = Deleted.
    temp.Rows[0].Delete();
    Console.WriteLine("After calling Delete: {0}", row.RowState);
}
```

Az ADO.NET `DataRow` típusa elég intelligens ahhoz, hogy megjegyezze az aktuális állapotát. Ennek köszönhetően a tulajdonos `DataTable` képes beazonosítani, hogy mely sorok módosultak. Ez a `DataSet` egyik kulcsfontosságú tulajdonsága, ugyanis amikor eljön az ideje, hogy a frissített információkat elküldjük az adattárba, akkor csak a módosított adatokkal kell foglalkoznunk.

A DataRowVersion tulajdonság

Amellett, hogy a `RowState` tulajdonsággal számon tartja egy sor aktuális állapotát, a `DataRow` objektum a `DataRowVersion` tulajdonság révén megőrzi az általa tartalmazott adatok három lehetséges verzióját. Egy `DataRow` objektum létrehozásakor csak az adatok egyetlen másolatát tartalmazza, amely „aktuális változatként" van jelen. Ahogy azonban programozottan dolgozunk a `DataRow` objektummal (a különböző metódushívások segítségével), további adatváltozatok jönnek létre. A `DataRowVersion` tulajdonságot beállíthatjuk a `DataRowVersion` felsorolt típus bármelyik értékére (lásd a 23.6. táblázatot).

Érték	Jelentés
`Current`	Egy sor aktuális értékét jelzi még a módosítások után is.
`Default`	A `DataRowState` alapértelmezett verziója. Az `Added`, a `Modified` vagy a `Deleted` értékű `DataRowState` esetén az alapértelmezett változat a `Current`. A `Detached` értékű `DataRowState` esetében ez a verzió a `Proposed`.
`Original`	A `DataRow` tulajdonságba először beírt vagy az `AcceptChanges()` legutóbbi meghívásakor fennálló értéket képviseli.
`Proposed`	A sor értéke jelenleg szerkesztés alatt áll a `BeginEdit()` meghívásának köszönhetően.

23.6. táblázat: *A DataRowVersion felsorolás értékei*

Ahogy a 23.6. táblázat mutatja, a `DataRowVersion` tulajdonság értéke számos esetben függ a `DataRowState` tulajdonság értékétől. A `DataRowVersion` tulajdonság a háttérben változik, amikor különböző metódusokat hívunk meg a `DataRow` (vagy bizonyos esetekben a `DataTable`) objektumon. A következőkben megnézzük azoknak a metódusoknak a kifejtését, amelyek befolyásolhatják egy sor `DataRowVersion` tulajdonságának az értékét:

- Ha meghívjuk a `DataRow.BeginEdit()` metódust, és megváltoztatjuk a sor értékét, akkor elérhetővé válnak a `Current` és a `Proposed` értékek.
- Ha meghívjuk a `DataRow.CancelEdit()` metódust, akkor a `Proposed` érték törlődik.
- A `DataRow.EndEdit()` meghívása után a `Proposed` érték lesz a `Current` érték.
- A `DataRow.AcceptChanges()` metódus meghívása után az `Original` érték ugyanaz lesz, mint a `Current` érték. Ez történik akkor is, ha meghívjuk a `DataTable.AcceptChanges()` metódust.
- A `DataRow.RejectChanges()` meghívása után a `Proposed` érték törlődik, és a változat `Current` lesz.

Ez így egy kissé bonyolult – különösen amiatt, hogy egy `DataRow` rendelkezhet is meg nem is az összes változattal egy adott pillanatban (futásidejű kivételeket kapunk, ha olyan sorváltozatot próbálunk megszerezni, amelyet jelenleg nem követünk nyomon). Az összetettségétől függetlenül, mivel a `DataRow` három adatmásolatot tart karban, nagyon egyszerű lesz olyan front endet létrehozni, amely lehetővé teszi a végfelhasználó számára az értékek módosítását, illetve azt, hogy meggondolhassa magát, és visszaállítsa az értékeket, vagy végérvényesen jóváhagyjon értékeket. A továbbiakban különböző példákat fogunk látni ezeknek a metódusoknak a kezelésére.

DataTable típusok használata

A `DataTable` típusnak számos tagja van, amelyek közül soknak a neve és a funkcionalitása azonos a `DataSet` típusban lévőkkel. A 23.7. táblázat bemutatja a `DataTable` típus néhány alapvető tagját a `Rows` és `Columns` tagokon túlmenően.

Tag	Jelentés
`CaseSensitive`	Azt jelzi, hogy a sztring-összehasonlítások a táblákban kis- vagy nagybetűérzékenyek-e (vagy sem). Az alapértelmezett érték: `false`.
`ChildRelations`	Visszaadja a `DataTable` gyermekkapcsolatainak gyűjteményét (ha van).
`Constraints`	Lekérdezi a táblázat által fenntartott korláltozások gyűjteményét.
`Copy()`	A metódus átmásolja egy adott `DataTable` sémáját és adatait egy új példányba.
`DataSet`	Lekérdezi a táblázatot tartalmazó `DataSetet` (ha van).
`DefaultView`	Lekérdezi a táblázat testre szabott nézetét, amely állhat szűrt nézetből vagy valamilyen kurzorpozícióból.
`MinimumCapacity`	Lekérdezi vagy beállítja a táblázat sorainak kezdeti számát (az alapértelmezett érték: 25).
`ParentRelations`	Lekérdezi a `DataTable` szülőkapcsolatainak gyűjteményét.
`PrimaryKey`	Lekérdezi vagy beállítja azon oszlopok tömbjét, amelyek elsődleges kulcsként szolgálnak a táblában.
`RemotingFormat`	Segítségével megadhatjuk, hogy a `DataSet` hogyan sorosítsa tartalmát (bináris vagy XML-formában) a .NET-remotingréteg számára.
`TableName`	Lekérdezi vagy beállítja a táblázat nevét. Ezt a tulajdonságot megadhatjuk konstruktorparaméterként is.

23.7. táblázat: *A DataTable típus kulcsfontosságú tagjai*

Állítsuk be a példánkban a `DataTable` `PrimaryKey` tulajdonságát a `carIDColumn` objektumra. A `PrimaryKey` tulajdonsághoz `DataColumn` objektumok gyűjteményét kell rendelnünk, hogy gondoskodjunk a többoszlopos kulcsról. Esetünkben azonban csak a CarID oszlopot kell megadni (amely az első oszlop a táblában):

```
static void Main(string[] args)
{
  ...
  // A táblázat elsődleges kulcsának megjelölése.
  inventoryTable.PrimaryKey =
                 new DataColumn[] { inventoryTable.Columns[0] };
  ...
}
```

DataTable típusok beszúrása DataSet objektumokba

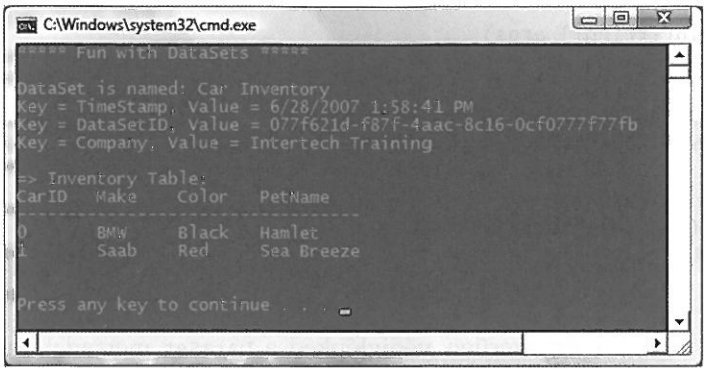
Utolsó lépésként a `DataTable` objektumot illesszük be a `carsInventoryDS` `DataSet` objektumba a `Tables` gyűjtemény használatával. Tételezzük fel, hogy ehhez módosítottuk a `Main()` metódust, és átadtuk a `DataSet` objektumot a `PrintDataSet()` új segédmetódusnak (amelyet még meg kell írni):

```
static void Main(string[] args)
{
  ...
  // Végül adjuk hozzá táblánkat a DataSethez.
  carsInventoryDS.Tables.Add(inventoryTable);

  // A DataSet kiíratása.
  PrintDataSet(carsInventoryDS);
  Console.ReadLine();
}
```

A `PrintDataSet()` metódus egyszerűen végiglépked a `DataSet` metaadatain (az `ExtendedProperties` gyűjtemény segítségével) és a `DataSet` minden `DataTable` objektumán, kiírja az oszlopneveket, valamint a sorok értékeit a típusindexelőkkel:

```
static void PrintDataSet(DataSet ds)
{
  // Kiír minden nevet és bővített tulajdonságot.
  Console.WriteLine("DataSet is named: {0}", ds.DataSetName);
  foreach (System.Collections.DictionaryEntry de in
           ds.ExtendedProperties)
  {
    Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
  }
  Console.WriteLine();

  foreach (DataTable dt in ds.Tables)
  {
    Console.WriteLine("=> {0} Table:", dt.TableName);

    // Kiírja az oszlopok neveit.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Console.Write(dt.Columns[curCol].ColumnName + "\t");
    }
    Console.WriteLine("\n---------------------------------");

    // Kiírja a DataTable-t.
    for (int curRow = 0; curRow < dt.Rows.Count; curRow++)
    {
```

```
            for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
            {
              Console.Write(dt.Rows[curRow][curCol].ToString() + "\t");
            }
            Console.WriteLine();
          }
        }
      }
```



23.3. ábra: *A példa DataSet objektumának tartalma*

A DataTable adatainak feldolgozása DataTableReader objektumokkal

Az előző fejezetben végzett munkánk azt bizonyította, hogy máshogy dolgozzuk fel az adatokat a kapcsolatalapú modell (pl. adatolvasó objektumok), illetve a kapcsolat nélküli modell (pl. `DataSet` objektumok) esetében. Az adatolvasó használata általában magában foglalja egy `while` ciklus létrehozását, a `Read()` metódus hívását és egy indexelő alkalmazását a név/érték párok kiszedéséhez. Másrészről a `DataSet` feldolgozása általában magában foglal egy iterációszerkezet-sorozatot, amely beleássa magát a táblákban, sorokban és oszlopokban lévő adatokba.

A .NET 2.0 megjelenése óta a `DataTable` típusok rendelkeznek a `CreateDataReader()` metódussal. Ez a metódus lehetővé teszi egy `DataTable` típusból az adatok kinyerését adatolvasó-szerű navigációs séma használatával (csak előre léptethető és írásvédett). Ennek a megközelítésnek a legfőbb előnye az, hogy egyetlen modellt használunk az adatok feldolgozásához, függetlenül attól, hogy az ADO.NET melyik „modelljét" használjuk azok megszerzéséhez. Tételezzük fel, hogy megírtuk a következő `PrintTable()` segédfüggvényt, és az alábbiak szerint implementáltuk:

```
static void PrintTable(DataTable dt)
{
  // Megkapjuk a DataTableReader típust.
  DataTableReader dtReader = dt.CreateDataReader();

  // A DataTableReader úgy működik, mint a DataReader.
  while (dtReader.Read())
  {
    for (int i = 0; i < dtReader.FieldCount; i++)
    {
      Console.Write("{0}\t",
              dtReader.GetValue(i).ToString().Trim());
    }
    Console.WriteLine();
  }
  dtReader.Close();
}
```

A `DataTableReader` ugyanúgy működik, mint az adatszolgáltatónk adatolvasó objektuma. Ideális választás lehet akkor, ha gyorsan szeretnénk kinyerni az adatokat egy `DataTable` típusból anélkül, hogy be kéne járnunk a belső sor- és oszlopgyűjteményeket. Tételezzük fel, hogy úgy módosítottuk az előző `PrintDataSet()` metódust, hogy az meghívja a `PrintTable()` metódust, ahelyett, hogy beleásná magát a sor- és oszlopgyűjteményekbe:

```
static void PrintDataSet(DataSet ds)
{
  // Kiír minden nevet és bővített tulajdonságot.
  Console.WriteLine("DataSet is named: {0}", ds.DataSetName);
  foreach (System.Collections.DictionaryEntry de in
           ds.ExtendedProperties)
  {
    Console.WriteLine("Key = {0}, Value = {1}", de.Key, de.Value);
  }
  Console.WriteLine();

  foreach (DataTable dt in ds.Tables)
  {
    Console.WriteLine("=> {0} Table:", dt.TableName);

    // Kiírja az oszlopok neveit.
    for (int curCol = 0; curCol < dt.Columns.Count; curCol++)
    {
      Console.Write(dt.Columns[curCol].ColumnName.Trim() + "\t");
    }
    Console.WriteLine("\n---------------------------------");

    // Hívjuk meg az új segédmetódusunkat.
    PrintTable(dt);
  }
}
```

Ha futtatjuk az alkalmazást, a kimenet hasonló lesz, mint amelyet a 23.3. ábrán láthatunk. Az egyetlen különbség az, hogy hogyan fértünk hozzá belsőleg a `DataTable` tartalmához.

A DataTable/DataSet objektumok sorosítása XML-ként

Mind a `DataSet`, mind a `DataTable` objektumok támogatják a `WriteXml()` és `ReadXml()` metódusokat. A `WriteXml()` lehetővé teszi, hogy egy helyi fájlba (valamint bármilyen `System.IO.Stream` leszármazott típusba) elmentsük az objektum tartalmát XML-dokumentumként. A `ReadXml()` lehetővé teszi, hogy feltöltsük egy `DataSet` (vagy `DataTable`) állapotát adott XML-dokumentumból. Emellett, mind a `DataSet`, mind a `DataTable` támogatja a `WriteXmlSchema()` és `ReadXmlSchema` metódusokat `*.xsd` fájlok mentéséhez vagy betöltéséhez.

Ehhez módosítsuk a `Main()` metódust, hogy meghívja a következő és egyben utolsó segédfüggvényt (a `DataSetet` adjuk át egyedüli paraméterként):

```
static void DataSetAsXml(DataSet carsInventoryDS)
{
  // A DataSet mentése XML-ként.
  carsInventoryDS.WriteXml("carsDataSet.xml");
  carsInventoryDS.WriteXmlSchema("carsDataSet.xsd");

  // A DataSet kiürítése.
  carsInventoryDS.Clear();

  // A DataSet betöltése XML-fájlból.
  carsInventoryDS.ReadXml("carsDataSet.xml");
}
```

Ha megnyitjuk a `carsDataSet.xml` fájlt (amely a projektünk \bin\Debug könyvtárában található), látható, hogy a táblázat minden egyes sora XML-elemként van kódolva:

```
<?xml version="1.0" standalone="yes"?>
<Car_x0020_Inventory>
  <Inventory>
    <CarID>0</CarID>
    <Make>BMW</Make>
    <Color>Black</Color>
    <PetName>Hamlet</PetName>
  </Inventory>
```

```
    <Inventory>
      <CarID>1</CarID>
      <Make>Saab</Make>
      <Color>Red</Color>
      <PetName>Sea Breeze</PetName>
    </Inventory>
  </Car_x0020_Inventory>
```

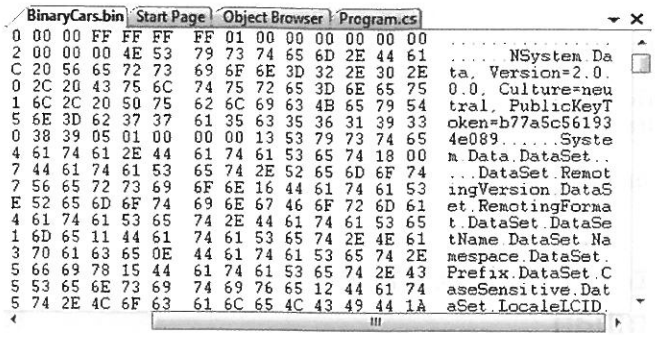
A DataTable/DataSet objektumok sorosítása bináris formátumban

A `DataSet` (vagy önálló `DataTable`) tartalmát tömör bináris formában is elmenthetjük. Ez akkor lehet különösen hasznos, ha egy `DataSet` objektumot számítógéphatáron keresztül kell átadni (megosztott alkalmazás esetén); az XML-adatok reprezentációjának egyik hátránya pedig, hogy leíró természetének köszönhetően jókora többletköltséget eredményezhet.

Ahhoz, hogy bináris formátumban rögzítsük a `DataTable` vagy a `DataSet` objektumokat, egyszerűen állítsuk át a `RemotingFormat` tulajdonságot `SerializationFormat.Binary` értékre. Mostantól úgy használhatjuk a `BinaryFormatter` típust (lásd a 21. fejezetet), ahogy azt elvárhatjuk. Vizsgáljuk meg a `SimpleDataSet` projekt alábbi, utolsó metódusát (amelyhez importálni kell a `System.IO` és a `System.Runtime.Serialization.Formatters.Binary` névtereket):

```
static void DataSetAsBinary(DataSet carsInventoryDS)
{
  // Állítsuk be a bináris sorosítás kapcsolóját.
  carsInventoryDS.RemotingFormat = SerializationFormat.Binary;

  // A DataSet mentése binárisként.
  FileStream fs = new FileStream("BinaryCars.bin", FileMode.Create);
  BinaryFormatter bFormat = new BinaryFormatter();
  bFormat.Serialize(fs, carsInventoryDS);
  fs.Close();

  // A DataSet kiürítése.
  carsInventoryDS.Clear();

  // A DataSet betöltése bináris fájlból.
  fs = new FileStream("BinaryCars.bin", FileMode.Open);
  DataSet data = (DataSet)bFormat.Deserialize(fs);
}
```

A 23.4. ábra mutatja a `BinaryCars.bin` fájl tartalmát.



23.4. ábra: *DataSet sorosítása bináris formátumba*

Forráskód A SimpleDataSet kódfájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

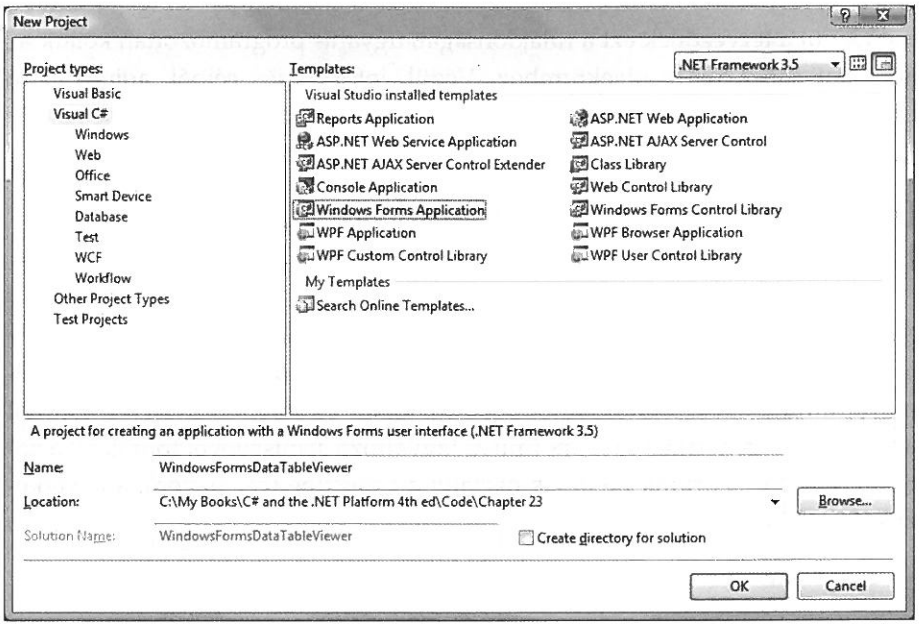
DataTable objektumok kötése felhasználói felületekhez

Megvizsgáltuk, hogyan hozhatjuk létre és tölthetjük fel manuálisan egy `DataSet` objektum tartalmát, illetve hogyan iterálhatunk rajta végig az ADO.NET objektummodelljének használatával. Noha ennek a mikéntje meglehetősen fontos, a .NET platform számos olyan API-t biztosít, amelyek automatikusan „hozzákötik" az adatokat a felhasználói felület elemeihez.

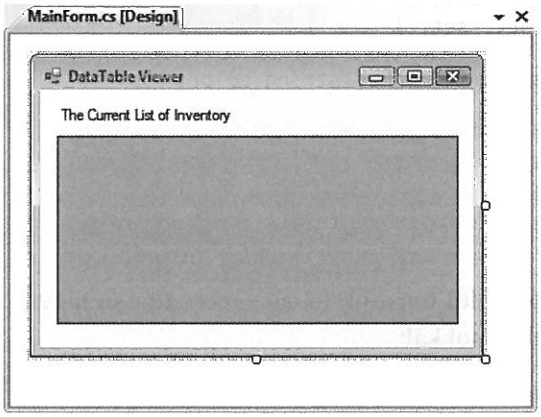
A .NET eredeti GUI-eszközrendszere, a Windows Forms például rendelkezik a `DataGridView` nevű vezérlőelemmel, amelynek megvan az a beépített képessége, hogy megjeleníti egy `DataSet` vagy egy `DataTable` objektum tartalmát mindössze néhány kódsor használatával. Az ASP.NET (a .NET webfejlesztő API-ja) és a Windows Presentation Foundation API-ja (a .NET 3.0-ban bemutatott új, hatékony grafikusfelület-API) szintén támogatja valamilyen formában az adatkötés fogalmát.

Az ADO.NET kapcsolat nélküli modelljének vizsgálatában a következő feladatunk olyan Windows Forms-alkalmazás készítése, amely a felhasználói felületén megjeleníti egy `DataTable` objektum tartalmát. Közben azt is megvizsgáljuk, hogyan lehet szűrni és módosítani a táblázat adatait, valamint megismerkedünk a `DataView` objektum szerepével. Először hozzunk létre egy teljesen új Windows Forms-projektet WindowsFormsDataTableViewer néven (lásd a 23.5. ábrát).

Megjegyzés Ez a példa azt feltételezi, hogy van némi tapasztalatunk a Windows Forms grafikus felhasználói felületek létrehozásában. Ellenkező esetben egyszerűen nyissuk meg a kész példát, és kövessük azt figyelemmel, vagy térjünk vissza ehhez a részhez a 27. fejezet elolvasása után, amelyben formálisan megvizsgáljuk a Windows Forms API-t.



23.5. ábra: *Windows Forms-alkalmazás létrehozása*



23.6. ábra: *A kezdeti felhasználói felület*

Nevezzük át a kezdeti `Form1.cs` fájlt a sokkal megfelelőbb `MainForm.cs` névre. Ezután a Visual Studio 2008 Toolbox használatával húzzunk át egy `DataGridView` vezérlőelemet (a Properties ablak `Name` tulajdonságának révén a `carInventoryGridView` nevet kapja) a tervezői felületre. Ekkor előugrik egy helyi menü, amely egy fizikai adatforráshoz teszi lehetővé a csatlakozást. Hagyjuk figyelmen kívül a tervezőnek ezt a tulajdonságát, ugyanis programozottan kötjük az adatokat a `DataTable` objektumhoz. Végül, információs célból, adjunk leíró `Label` típust a tervezőhöz. A 23.6. ábra egy lehetséges kivitelezést mutat be.

DataTable feltöltése egy generikus List<T> használatával

Az előző SimpleDataSet példához hasonlóan a `WindowsFormsDataTableViewer` alkalmazás olyan `DataTable` típust hoz létre, amely `DataColumn` típusok halmazát tartalmazza, adatok különböző oszlopait és sorait képviselve. Ezúttal azonban egy generikus `List<T>` típusú tagváltozó segítségével töltjük fel a sorokat. Először szúrjunk be új C#-osztályt a projektbe (`Car` néven), a következők definícióval:

```
class Car
{
  // A C# automatikus tulajdonságok használata.
  public string carPetName { get; set; }
  public string carMake { get; set; }
  public string carColor { get; set; }

  public Car(string petName, string make, string color)
  {
    carPetName = petName;
    carColor = color;
    carMake = make;
  }
}
```

Az alapértelmezett konstruktoron belül töltsünk fel a `List<T>` típusú tagváltozót (neve: `listCars`) új `Car` objektumokkal:

```
public partial class MainForm : Form
{
  // Car objektumok gyűjteménye.
  List<Car> listCars = new List<Car>();
  public MainForm()
  {
    InitializeComponent();
```

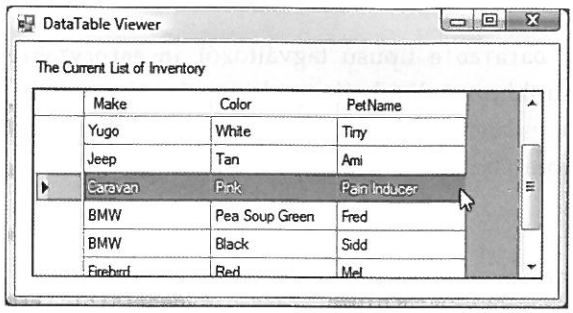
```
    // Töltsük fel a listát néhány autóval.
    listCars.Add(new Car("Chucky", "BMW", "Green"));
    listCars.Add(new Car("Tiny", "Yugo", "White"));
    listCars.Add(new Car("Ami", "Jeep", "Tan"));
    listCars.Add(new Car("Pain Inducer", "Caravan", "Pink"));
    listCars.Add(new Car("Fred", "BMW", "Pea Soup Green"));
    listCars.Add(new Car("Sidd", "BMW", "Black"));
    listCars.Add(new Car("Mel", "Firebird", "Red"));
    listCars.Add(new Car("Sarah", "Colt", "Black"));
  }
}
```

Ezután adjunk hozzá egy új, `DataTable` típusú tagváltozót `inventoryTable` néven a `MainForm` osztálytípusunkhoz. Adjunk új segédfüggvényt az osztályhoz `CreateDataTable()` névvel, és hívjuk meg ezt a metódust a `MainForm` osztály alapértelmezett konstruktorán belül:

```
void CreateDataTable()
{
// A táblázatséma létrehozása.
  DataColumn carMakeColumn = new DataColumn("Make", typeof(string));
  DataColumn carColorColumn =
                new DataColumn("Color", typeof(string));
  DataColumn carPetNameColumn =
                new DataColumn("PetName", typeof(string));
  carPetNameColumn.Caption = "Pet Name";
  inventoryTable.Columns.AddRange(new DataColumn[] { carMakeColumn,
                          carColorColumn, carPetNameColumn });

  // Iteráljunk végig a tömblistán sorok készítéséhez.
  foreach (Car c in listCars)
  {
    DataRow newRow = inventoryTable.NewRow();
    newRow["Make"] = c.carMake;
    newRow["Color"] = c.carColor;
    newRow["PetName"] = c.carPetName;
    inventoryTable.Rows.Add(newRow);
  }

  // Kössük hozzá a DataTable-t a carInventoryGridView
  // vezérlőelemhez.
  carInventoryGridView.DataSource = inventoryTable;
}
```

A metódusimplementáció elején létrehozzuk a `DataTable` sémáját három `DataColumn` objektum megvalósításával (az egyszerűség kedvéért most nem tettük hozzá az automatikusan növekvő CarID mezőt), és ezt követően az objektumokat hozzáadjuk a `DataTable` tagváltozóhoz. A soradatok leképzése a `List<T>`-ből a `DataTable`-be egy `foreach` ciklusban a natív ADO.NET-objektummodell segítségével történik.

A `CreateDataTable()` metódus utolsó utasítása hozzárendeli az `inventoryTable` táblát a `DataSource` tulajdonsághoz. Ez az egyetlen tulajdonság, amelyet be kell állítanunk, hogy a `DataTable` típust a `DataGridView` objektumhoz kössük. Ez a GUI-vezérlőelem a háttérben belsőleg olvassa be a sor- és az oszlopgyűjteményeket, nagyjából úgy, mint azt a `SimpleDataSet` példa `PrintDataSet()` metódusa esetében. Ezt követően futtathatjuk az alkalmazásunkat, és megnézhetjük a `DataTablet` a `DataGridView` vezérlőelemben (lásd a 23.7. ábrát).



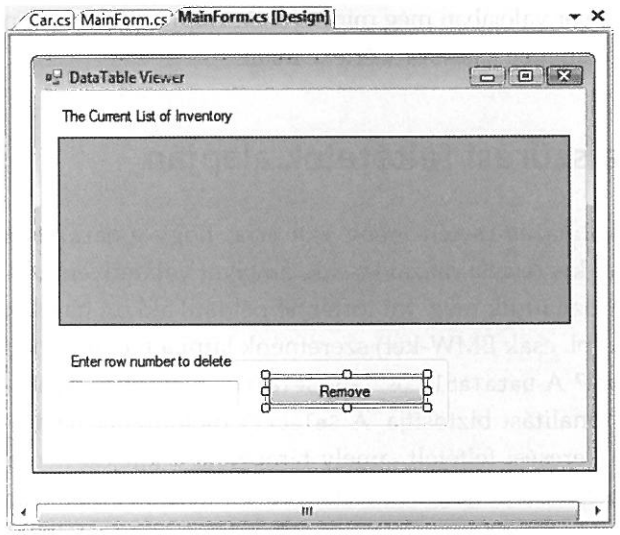
23.7. ábra: *Egy DataTable kötése egy Windows Forms DataGridView vezérlőelemhez*

Sorok programozott törlése

Tételezzük fel, hogy úgy szeretnénk módosítani a grafikus felületünket, hogy a felhasználó sorokat törölhessen a `DataGridView` vezérlőelemhez kötött `DataTable` objektumból. Ennek egyik módja az, ha meghívjuk annak a `DataRow` objektumnak a `Delete()` metódusát, amely az eltávolítani kívánt sort képviseli. Egyszerűen határozzuk meg a törölni kívánt sort jelképező indexet (vagy `DataRow` objektumot). Ahhoz, hogy lehetővé tegyük a felhasználó számára a törölni kívánt sor meghatározását, adjunk hozzá egy `TextBox` (`txtRowToRemove` névvel) és egy `Button` (`btnRemoveRow` névvel) vezérlőelemet a tervezőhöz a 23.8. ábrán látható módon.

Az új `Button` vezérlőelem `Click` eseménykezelőjében található logika törli a felhasználó által meghatározott sort a memóriában lévő `DataTable` objektumból. A törlési logika egy `try` blokkban van, hiszen számításba vesszük annak lehetőségét, hogy a felhasználó olyan sor számát írta be, amely jelenleg nincs a `DataGridView` vezérlőelemben:

```
// A sor eltávolítása a DataRowCollection gyűjteményből.
private void btnRemoveRow_Click (object sender, EventArgs e)
{
  try
  {
    inventoryTable.Rows[(int.Parse(txtRowToRemove.Text))].Delete();
    inventoryTable.AcceptChanges();
  }
  catch(Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```



23.8. ábra: *A felhasználói felület módosítása sorok törlésének az engedélyezéséhez az alapul szolgáló DataTable objektumból*

A `Delete()` metódust találóbb lenne a `MarkedAsDeletable()` névvel illetni, ugyanis a sort a rendszer valójában addig nem távolítja el, amíg nem hívjuk a `DataTable.AcceptChanges()` metódust. A `Delete()` metódus voltaképpen beállít egy kapcsolót, amely azt jelzi: „Készen állok a megsemmisülésre, amikor a tábla erre utasítást ad." Ha egy sort törlésre jelöltünk, a `DataTable` elvetheti a törlési műveletet a `RejectChanges()` metódus révén. Ezt most ebben a példában nem kell megtennünk, de módosíthatnánk a kódot a következők szerint:

```
// Sor megjelölése töröltként, de a módosítások visszautasítása.
private void btnRemoveRow_Click (object sender, EventArgs e)
{
  ...
  inventoryTable.Rows[(int.Parse(txtRemove.Text))].Delete();
```

```
    // Csináljunk még valamit?
    ...

    // Az előző RowState érték visszaállítása.
    inventoryTable.RejectChanges();
    ...
}
```

Most már futtathajtuk az alkalmazást, és meghatározott sort törölhetünk a DataTable objektumból a DataGridView adott sora alapján (a belső sorgyűjtemény indexere nulla alapú). Ha DataRow objektumokat távolítunk el a DataTable objektumból, a táblázat azonnal frissül, hiszen az objektum állapotához kötöttük. Azt azonban ne feledjük, hogy a sor valójában még mindig a DataTable objektumban van, csak a táblázat nem jeleníti meg a RowState értéke miatt.

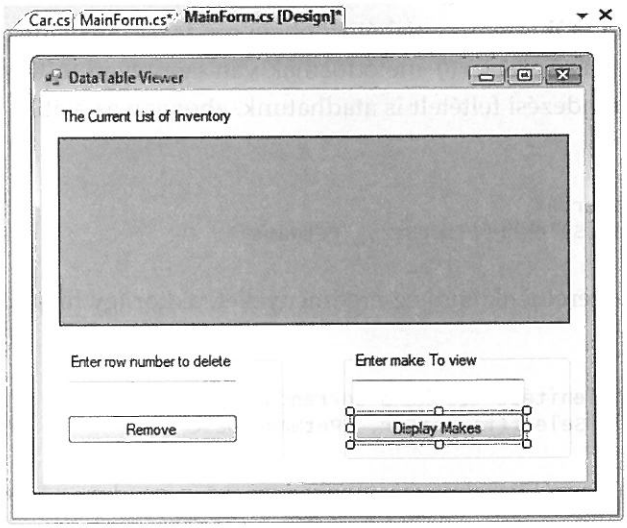
Sorok kiválasztása szűrési feltételek alapján

Számos adatközpontú alkalmazás esetén igény van arra, hogy a DataTable objektumok adatainak csak kis részhalmazát lássuk, amelyet valamilyen szűrési feltételek alapján határozhatunk meg. Mi történne például akkor, ha csak bizonyos márkájú autókat (pl. csak BMW-ket) szeretnénk látni a memóriában lévő DataTable objektumból? A DataTable osztály Select() metódusa pontosan ezt az alapvető funkcionalitást biztosítja. A Select() metódus használatával megadhatunk olyan keresési feltételt, amely támogatja a kifejezetten a szabványos SQL-lekérdezések modellezésére tervezett szintaxist. A metódus visszaadja a DataRow objektumok azon tömbjét, amelyben benne van a feltételeknek megfelelő összes bejegyzés. Ennek bemutatásához ismét módosítsuk a felhasználói felületet - ezúttal lehetővé téve a felhasználók számára, hogy megadják a látni kívánt autómárkák nevét (lásd a 23.9. ábrát) - egy új Textbox vezérlőelem (txtMakeToView) és egy új Button vezérlőelem (btnDisplayMakes) hozzáadásával.

A Select() metódus többszörös túlterhelése különböző szűrési szemantikákat biztosít. Alapjában véve a Select() metódusnak átadott paraméter egy olyan sztring, amely tartalmaz néhány feltételes műveletet. Figyeljük meg az új gomb Click eseménykezelőjének következő logikáját:

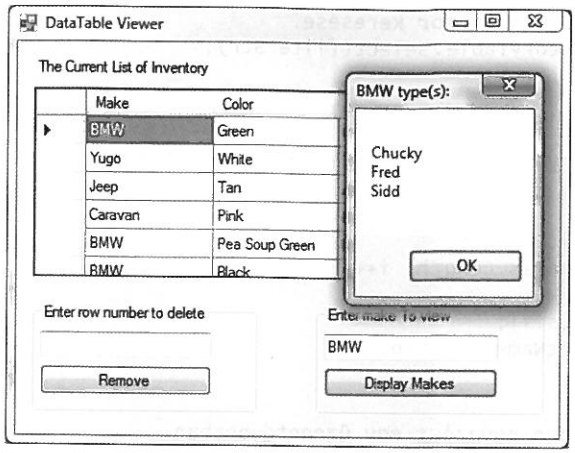
```
private void btnDisplayMakes_Click(object sender, EventArgs e)
{
  // Szűrő létrehozása a felhasználói bevitel alapján.
  string filterStr =
              string.Format("Make= '{0}'", txtMakeToView.Text);
```

```
    // A szűrőnek megfelelő összes sor keresése.
    DataRow[] makes = inventoryTable.Select(filterStr);

    // Lássuk, mink van!
    if (makes.Length == 0)
      MessageBox.Show("Sorry, no cars...", "Selection error!");
    else
    {
      string strMake = null;
      for (int i = 0; i < makes.Length; i++)
      {
        DataRow temp = makes[i];
        strMake += temp["PetName"] + "\n";
      }

      // Mutassunk meg minden egyezést egy üzenetdobozban.
      MessageBox.Show(strMake,
            string.Format("{0} type(s):", txtMakeToView.Text));
    }
  }
```



23.9. ábra: *A felhasználói felület módosítása a sorok szűrésének engedélyezéséhez*

Először egy egyszerű szűrőt hoztunk létre, amely a hozzákapcsolódó `TextBox` értékén alapul. Ha BMW-t írunk bele, akkor a szűrő `Make = 'BMW'` lesz. Ha elküldjük ezt a szűrőt a `Select()` metódusnak, akkor visszakapjuk a `DataRow` típusok egy tömbjét, amely minden olyan sort megjelenít, amely megfelel a szűrőnek (lásd a 23.10. ábrát).



23.10. ábra: *A szűrt adatok megjelenítése*

A szűrési logika a szabványos SQL-szintaxison alapul. Tételezzük fel, hogy szeretnénk megkapni az előző `Select()` hívásnevek szerint ábécésorrendbe rendezett eredményeit. Az SQL szempontjából ez olyan rendezés, amelynek a PetName oszlop az alapja. A `Select()` metódustnak van egy olyan túlterhelt változata, amelynek rendezési feltételt is átadhatunk, ahogyan az alábbiakban látható:

```
// Rendezés PetName szerint.
makes = inventoryTable.Select(filterStr, "PetName");
```

Ha csökkenő sorrendben szeretnénk látni az eredményeket, akkor így hívjuk meg a `Select()` metódust:

```
// Az eredmények megjelenítése csökkenő sorrendben.
makes = inventoryTable.Select(filterStr, "PetName DESC");
```

A rendezősztring az oszlop nevét tartalmazza, amelyet az `ASC` (ascending = növekvő, ez az alapértelmezett) vagy a `DESC` (descending = csökkenő) kulcsszó követ. Ha szükséges, akkor vesszővel elválasztva több oszlopot is megadhatunk. A szűrősztring bármennyi kapcsolódó operátorból állhat. Mi történik például akkor, ha minden olyan autóra kíváncsiak vagyunk, amelynek az azonosítója 5-nél nagyobb? Az alábbi segédfüggvény pontosan ezt biztosítja:

```
private void ShowCarsWithIdGreaterThanFive()
{
  // Most megmutatjuk az összes olyan autó nevét,
  // amelynek az azonosítója nagyobb, mint 5.
  DataRow[] properIDs;
```

```
    string newFilterStr = "ID > 5";
    properIDs = inventoryTable.Select(newFilterStr);
    string strIDs = null;
    for(int i = 0; i < properIDs.Length; i++)
    {
      DataRow temp = properIDs[i];
      strIDs += temp["PetName"]
             + " is ID " + temp["ID"] + "\n";
    }
    MessageBox.Show(strIDs, "Pet names of cars where ID > 5");
  }
```

Sorok módosítása

Végül vizsgáljuk meg a már meglévő sorok értékeinek a módosítását. Ennek egyik módja az, ha először megszerezzük az összes olyan sort, amely megfelel egy adott szűrési feltételnek – természetesen a `Select()` metódus használatával. Amint megvan(nak) a kérdéses `DataRow` objektum(ok), módosíthatjuk őket. Tételezzük fel például, hogy van egy új gomb az űrlapon `btnChangeBeemersToYugos` névvel, amely (ha rákattintunk) a táblában megkeres minden olyan sort, amelyben a márka BMW. Amikor azonosítottuk ezeket az elemeket, változtassuk meg a márkát BMW-ről Yugóra:

```
// Szűrő segítségével keressük meg a szerkeszteni kívánt sorokat.
private void btnChangeBeemersToYugos_Click(object sender,
                                           EventArgs e)
{
  // Bizonyosodjunk meg arról, hogy a felhasználó nem őrült meg.
  if (DialogResult.Yes ==
      MessageBox.Show("Are you sure??
                       BMWs are much nicer than Yugos!",
                      "Please Confirm!", MessageBoxButtons.YesNo))
  {
    // Hozzunk létre egy szűrőt.
    string filterStr = "Make='BMW'";
    string strMake = string.Empty;

    // Keressünk meg minden sort, amely megfelel a szűrőnek.
    DataRow[] makes = inventoryTable.Select(filterStr);

    // Az összes BMW megváltoztatása Yugóra!
    for (int i = 0; i < makes.Length; i++)
    {
      makes[i]["Make"] = "Yugo";
    }
  }
}
```

A DataRow osztály biztosítja a BeginEdit(), az EndEdit() és a CancelEdit() metódusokat, amelyek lehetővé teszik, hogy megszerkesszük egy sor tartalmát, miközben ideiglenesen felfüggesztünk minden érvényesítési szabályt. Az előző kódban nem függesztettük fel ezeket a szabályokat, így minden értékadás ellenőrizve lett. (Emellett, ha a DataRow bármilyen eseményét kezeljük, azok minden módosításkor elsünek.) Ha meghívjuk a BeginEdit() metódust egy adott DataRow objektumon, akkor a sor szerkesztésmódba kerül. Ilyenkor elvégezhetjük a kívánt módosításokat, és meghívhatjuk az EndEdit() metódust ezek véglegesítéséhez, vagy a CancelEdit() metódust, hogy visszatérjünk az eredeti változathoz, például:

```
private void UpdateSomeRow()
{
  // Tegyük fel, hogy megkaptunk egy sort szerkesztésre.
  // Most tegyük ezt a sort szerkesztésmódba.
  rowToUpdate.BeginEdit();

  // Küldjük el a sort egy segédfüggvénynek,
  // amely egy logikai értéket ad vissza.
  if( ChangeValuesForThisRow( rowToUpdate) )
    rowToUpdate.EndEdit();      // OK!
  else
    rowToUpdate.CancelEdit();  // Felejtsük el!
}
```

Noha manuálisan is meghívhatjuk ezeket a metódusokat egy adott DataRow objektumra, a tagok automatikusan meghívódnak, amikor egy DataTable típushoz kötött DataGridView céleszközt szerkesztünk. Amikor például a DataGridView vezérlőelemből választunk ki egy sort szerkesztésre, a sor automatikusan szerkesztésmódba kerül. Amikor áttérünk egy új sorra, az EndEdit() metódus automatikusan meghívódik.

A DataView típus használata

Az adatbázisok szaknyelvében a *nézet* egy tábla (vagy táblák halmazának) alternatív megjelenítését jelenti. A Microsoft SQL Server segítségével például létrehozhatunk olyan nézetet az Inventory táblázathoz, amely csak adott színű autókat tartalmazó táblát ad vissza. Az ADO.NET-ben a DataView típus lehetővé teszi adatok részhalmazának programozott kinyerését a DataTable típusból egy önálló objektumba.

Egyik nagy előnye, ha ugyanannak a táblázatnak többféle nézete van, hogy a nézeteket hozzákötheti jük különböző GUI-célelemekhez (pl. a DataGridView vezérlőelemhez). Az egyik DataGridView nézetet hozzá lehet kötni olyan DataView objektumhoz, amely az Inventory összes autóját mutatja, míg egy másikat be lehet állítani úgy, hogy csak a zöld autókat jelenítse meg.

Ennek bemutatásához egészítsük ki a jelenlegi felhasználói felületet egy további DataGridView-val, amelynek a neve dataGridColtsView, és egy leíró Labellel. Ezután definiáljunk egy DataView típusú, coltsOnlyView nevű tagváltozót:

```
public partial class MainForm : Form
{
  // A DataTable nézete.
  DataView coltsOnlyView;
  ...
}
```

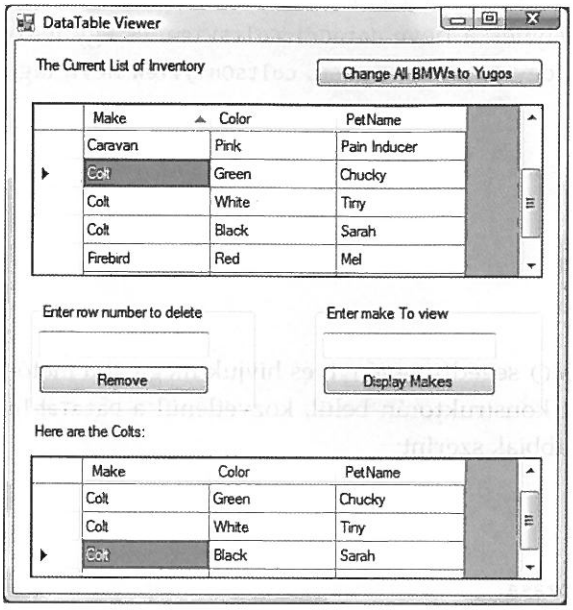
Hozzuk létre a CreateDataView() segédfüggvényt, és hívjuk meg ezt a metódust az űrlap alapértelmezett konstruktorán belül, közvetlenül a DataTable sikeres létrehozása után, az alábbiak szerint:

```
public MainForm()
{
  ...
  // Adattáblázat létrehozása.
  CreateDataTable();

  // Nézet létrehozása.
  CreateDataView();
}
```

Alább látható az új segédfüggvény implementációja. A DataView konstruktorának átadtuk a DataTablet, amelyet az adatsorok egyéni készletének létrehozására fogunk használni.

```
private void CreateDataView()
{
  // Állítsuk be a táblát, amelyet a nézet létrehozására használunk.
  coltsOnlyView = new DataView(inventoryTable);

  // Most állítsuk be a nézeteket egy szűrő segítségével.
  coltsOnlyView.RowFilter = "Make = 'Colt'";

  // Kössük hozzá az új táblázathoz.
  dataGridColtsView.DataSource = coltsOnlyView;
}
```

A `DataView` osztály támogatja a `RowFilter` nevű tulajdonságot, amely a megfelelő sorok kinyerésére használt szűrési feltételeket jelképező sztringet tartalmazza. Miután létrehoztuk ezt a nézetet, ennek megfelelően állítsuk be a táblázat `DataSource` tulajdonságát. A 23.11. ábra mutatja a kész alkalmazást működés közben.



23.11. ábra: *Adataink egyéni nézete*

Egy utolsó felhasználói felületbővítmény: sorok számának megjelenítése

Adjunk hozzá még egy apró kiegészítést az aktuális alkalmazásunkhoz. Az ablakban lévő táblázatok jelenleg nem adnak támpontot a végfelhasználónak ahhoz, hogy hányas számú sort szerkeszti (vagy törli) éppen. Ha szeretnénk megjeleníteni a sorok számait a `DataGridView` vezérlőelemen, először a `RowPostPaint` eseményt kell kezelnünk magán a táblázaton. Ezt az eseményt akkor váltja ki a rendszer, ha a táblázat celláiban lévő összes adatot megjelenítettük a felhasználói felületen. Az esemény lehetőséget ad arra, hogy véglegesítsük a sorok grafikai megjelenését, mielőtt megmutatnánk őket a felhasználónak.

Miután kezeltük ezt az eseményt a bejövő `DataGridViewRowPostPaintEventArgs` paraméterben, megkapunk egy `Graphics` objektumot. A `Graphics` típus a GDI+ API része, amely a Windows Forms 2D saját renderelő eszközrendszere (lásd 27. fejezet). A `Graphics` objektum segítségével meghívhatunk különböző metódusokat (pl.az ehhez a példához megfelelő `DrawString()`) a tartalom megjelenítéséhez. (A GDI+ eszközrendszert lásd a 27. fejezetben.) A `RowPostPaint` esemény implementációja minden sor számát megjeleníti a `carInventoryGridView` objektumon (ugyanezt az eseményt természetesen a `dataGridColtsView` objektumon is kezelhetjük, a hatás hasonló lesz):

```
void carInventoryGridView_RowPostPaint(object sender,
                                DataGridViewRowPostPaintEventArgs e)
{
  // "Fessük fel" a sorok számát egyszínű ecset használatával,
  // az aktuális sor stílusának megfelelő saját betűtípussal.
  using (SolidBrush b = new SolidBrush(Color.Black))
  {
    e.Graphics.DrawString((e.RowIndex).ToString(),
                          e.InheritedRowStyle.Font, b,
                          e.RowBounds.Location.X + 15,
                          e.RowBounds.Location.Y + 4);
  }
}
```

Forráskód A WindowsFormsDataTableViewer kódfájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

DataSet/DataTable objektumok feltöltése adatillesztőkkel

A következőkben vizsgáljuk meg az adatillesztő objektumokat. Ezeket arra használjuk, hogy feltöltsünk egy `DataSet` típust `DataTable` objektumokkal. Továbbá képesek módosított `DataTable` objektumokat visszaküldeni az adatbázisnak feldolgozásra. A 23.8. táblázat összefoglalja a `DbDataAdapter` ősosztálynak, minden adatillesztő objektum közös szülőjének alapvető tagjait.

Tagok	Jelentés
`Fill()`	Feltölti egy `DataSet` adott tábláját a parancsobjektum-specifikus `SelectCommand` alapján.

Tagok	Jelentés

`SelectCommand`
`InsertCommand`
`UpdateCommand`
`DeleteCommand`
	Olyan SQL-parancsokat hoz létre, amelyek az adattárolóban futnak a `Fill()` és az `Update()` metódusok meghívásakor.

`Update()`	Az `InsertCommand`, az `UpdateCommand` vagy a `DeleteCommand` tulajdonságban megadott parancsobjektumok használatával frissíti a `DataTable`-t. A végrehajtott pontos parancs (egy adott `DataSet`) adott `DataTable` típusának adott `DataRow` objektumához tartozó `RowState` értéken alapul.

23.8. táblázat: *A DbDataAdapter alapvető tagjai*

Egy adatillesztő négy tulajdonságot határoz meg (a `SelectCommand`, az `InsertCommand`, az `UpdateCommand` és a `DeleteCommand`), amelyek mindegyike külön parancsobjektumokon működik. Amikor létrehozzuk az adatillesztő objektumot az adott adatszolgáltató (pl. `SqlDataAdapter`) számára, átadhatunk egy sztringtípust, amely a `SelectCommand` parancsobjektuma által használt parancs szövegét képviseli. A fennmaradó három parancsobjektumot (amelyeket az `InsertCommand`, `UpdateCommand` és `DeleteCommand` tulajdonságok használnak) manuálisan kell beállítanunk.

Ha mind a négy parancsobjektumot megfelelően konfiguráltuk, meghívhatjuk a `Fill()` metódust, hogy beolvasson egy `DataSet` (vagy igény szerint egyetlen `DataTable`) típust. Ehhez az adatillesztő azt a parancsobjektumot fogja használni, amelyiket megtalálja a `SelectCommand` tulajdonság révén. Hasonló módon, amikor vissza akarunk adni egy módosított `DataSet` (vagy `DataTable`) objektumot az adatbázisnak feldolgozásra, meghívhatjuk az `Update()` metódust, amely a fennmaradó parancsobjektumok valamelyikét használja annak alapján, hogy mi az egyes sorok állapota a `DataTable` objektumban (ezt részletesebben lásd később).

Az adatillesztő objektumok egyik legkülönösebb aspektusa az a tény, hogy velük dolgozva soha nem kell csatlakoznunk egy adatbázishoz vagy lezárni az adatbázis-kapcsolatot. Ezek az objektumok helyettünk kezelik az alapul szolgáló adatbázis-kapcsolatot. Meg kell adnunk az adatillesztőnek egy érvényes kapcsolatobjektumot vagy kapcsolatsztringet (ezzel belsőleg épül fel egy kapcsolatobjektum), amely tájékoztatja az adatillesztőt, hogy pontosan melyik adatbázishoz szeretnénk csatlakozni.

Egy egyszerű adatillesztő

Mielőtt új funkcionalitással bővítenénk a 22. fejezetben létrehozott `AutoLotDAL.dll` szerelvényt, kezdjük egy nagyon egyszerű példával, amelyben egy `DataSet` objektumot töltünk fel egy táblával, az ADO.NET adatillesztő objektumának segítségével. Hozzunk létre egy új parancssori alkalmazást `FillDataSetWithSqlDataAdapter` néven, majd importáljuk a `System.Data` és a `System.Data.SqlClient` névtereket a kezdeti a C#-kódfájlunkba.

Ezután módosítsuk a `Main()` metódust a következők szerint (az egyszerűség kedvéért nyugodtan használjunk bedrótozott kapcsolatsztringet):

```
static void Main(string[] args)
{
  Console.WriteLine("***** Fun with Data Adapters *****\n");
  // Bedrótozott kapcsolatsztring.
  string cnStr = "Integrated Security =
                  SSPI;Initial Catalog=AutoLot;" +
                  @"Data Source=(local)\SQLEXPRESS";

  // A hívó létrehozza a DataSet objektumot.
  DataSet ds = new DataSet("AutoLot");

  // Az illesztő tájékoztatása a Select parancsszövegről és
  // a kapcsolatról.
  SqlDataAdapter dAdapt =
        new SqlDataAdapter("Select * From Inventory", cnStr);

  // A DataSet feltöltése az Inventory nevű új táblával.
  dAdapt.Fill(ds, "Inventory");

  // A DataSet tartalmának megjelenítése.
  PrintDataSet(ds);
  Console.ReadLine();
}
```

Az adatillesztőt egy olyan sztringliterál megadásával hoztuk létre, amely az SQL Select utasítására fog leképződni. Ennek az értéknek az alapján fog létrejönni a belső parancsobjektum, amelyet később a `SelectCommand` tulajdonság révén érhetünk el.

A hívó dolga a `DataSet` típus egy példányának létrehozása, amelyet aztán átadunk a `Fill()` metódusnak. A `Fill()` metódusnak opcionálisan átadható egy második sztringargumentum is, amellyel az új `DataTable` `TableName` tulajdonságát lehet beállítani (ha nem adunk meg táblázatnevet, akkor az adatillesztő egyszerűen a `Table` nevet fogja használni). Bár a legtöbb esetben a `DataTable` neve ugyanaz lesz, mint a fizikai tábla neve a relációs adatbázisban, de ez nem kötelező, hogy így legyen.

Megjegyzés A `Fill()` metódus olyan egész számot ad vissza, amely az SQL-lekérdezés által visszaadott sorok számát jelképezi.

A `Main()` metóduson belül explicit módon sehol nem nyitunk meg vagy zárunk be adatbázis-kapcsolatot. Egy adott adatillesztő `Fill()` metódusa automatikusan megnyitja, majd bezárja a kapcsolatot, mielőtt visszatérne a metódusból. Ezért, amikor átadjuk a `DataSet` objektumot a `PrintDataSet()` metódusnak (a fejezet korábbi részében megvalósítottuk), akkor az adatok helyi másolatán dolgozunk, és az adatokat nem kérjük le feleslegesen.

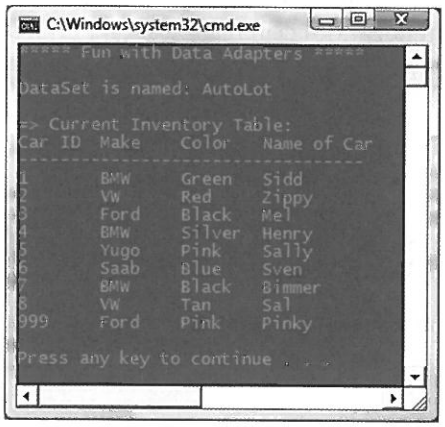
Adatbázisnevek leképezése barátságos nevekre

Az adatbázis-rendszergazdák hajlamosak olyan tábla- és oszlopneveket kitalálni, amelyek nem érthetők a végfelhasználók számára (pl. au_id, au_fname, au_lname stb.). Ám az adatillesztő objektumok karbantartanak egy erősen típusos `System.Data.Common.DataTableMapping` belső gyűjteményt (`DataTableMappingCollection` néven). A gyűjteményhez az adatillesztő objektum `TableMappings` tulajdonságának révén lehet hozzáférni.

Ezt a gyűjteményt módosíthatjuk is, ezzel tájékoztathatjuk a `DataTable` típust, hogy mely „megjelenítendő neveket" használja, amikor ki kell írnia a tartalmát. Tételezzük fel például, hogy szeretnénk leképezni az Inventory táblanevet Current Inventory névre a megjelenítéshez. Továbbá, a CarID oszlop nevét mondjuk Car ID-ként (egy plusz szóközzel), a PetName oszlop nevét pedig Name of Carként szeretnénk megjeleníteni. Ehhez adjuk hozzá a következő kódot, mielőtt meghívnánk az adatillesztő `Fill()` metódusát (és ne felejtsük el importálni a `System.Data.Common` névteret, hogy elérjük a `DataTableMapping` típust):

```
static void Main(string[] args)
{
  ...
  // Most képezzük le az adatbázis oszlopneveit
  // felhasználóbarát nevekre.
  DataTableMapping custMap =
      dAdapt.TableMappings.Add("Inventory", "Current Inventory");
  custMap.ColumnMappings.Add("CarID", "Car ID");
  custMap.ColumnMappings.Add("PetName", "Name of Car");
  dAdapt.Fill(myDS, "Inventory");
  ...
}
```

Ha megint futtatjuk a programot, látható, hogy a `PrintDataSet()` metódus most a `DataTable` és a `DataRow` objektumok „barátságos nevét" jeleníti meg, nem pedig az adatbázissémában megadott neveket. A 23.12. ábra mutatja a példa kimenetét.



23.12. ábra: *DataTable objektumok egyedi leképezésekkel*

Forráskód A FillDataSetWithSqlDataAdapter kódfájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Az AutoLotDAL.dll ismételt vizsgálata

Annak a folyamatnak a bemutatásához, amelyben az adatillesztő visszaküldi a `DataTable` módosításait az adatbázisnak feldolgozásra, módosítsuk az `AutoLotDAL.dll` szerelvényt – amelyet a 22. fejezetben hoztunk létre –, hogy egy új névteret tartalmazzon (`AutoLotDisconnectedLayer` névvel). A névtérben lévő új osztály, az `InventoryDALDisLayer` egy adatillesztőt használ arra, hogy a `DataTable`-lel kommunikáljon.

A kiinduló osztálytípus definiálása

Nyissuk meg az AutoLotDAL projektet a Visual Studio 2008-ban, szúrjunk be egy új osztálytípust `InventoryDALDisLayer` névvel a Project >Add New Item menüpont segítségével, és győződjünk meg róla, hogy az új kódfájlban van egy *nyilvános* osztálytípus. Az `InventoryDAL` típus kapcsolat-központú

modelljével ellentétben ennek az új osztálynak nem kell megadni egyedi megnyitás/bezárás metódusokat, mivel az adatillesztő automatikusan kezeli ezeket a részleteket.

Először adjunk hozzá egy egyedi konstruktort, amely beállítja a kapcsolatsztringet jelképező sztringváltozót. Ezután definiáljunk egy privát `SqlDataAdapter` tagváltozót, amelyet egy (még létrehozandó) `ConfigureAdapter()` nevű segédmetódus meghívásával konfigurálunk, amely egy `SqlDataAdapter` kimeneti paramétert kap:

```
public class InventoryDALDisLayer
{
  // Adatmező.
  private string cnString = string.Empty;
  private SqlDataAdapter dAdapt = null;

  public InventoryDALDisLayer(string connectionString)
  {
    cnString = connectionString;

    // Az SqlDataAdapter konfigurálása.
    ConfigureAdapter(out dAdapt);
  }
}
```

Az adatillesztő konfigurálása az SqlCommandBuilder használatával

Amikor az adatillesztővel módosítunk táblákat egy `DataSetben`, az első dolgunk hozzárendelni az `UpdateCommand`, a `DeleteCommand` és az `InsertCommand` tulajdonságokhoz érvényes parancsobjektumokat (amíg ezt nem tesszük meg, a tulajdonságok nullreferenciát adnak vissza). Az „érvényes" parancsobjektumok alatt azt a parancsobjektum-készletet kell érteni, amelyet a módosítani kívánt táblával (jelen esetben az Inventory táblával) együtt használunk.

Az illesztő feltöltése a szükséges adatokkal jókora mennyiségű kódot igényel, különösen akkor, ha paraméterezett lekérdezéseket használunk. A paraméterezett lekérdezés lehetővé teszi SQL-utasítások létrehozását paraméterobjektumok segítségével (lásd 22. fejezet). Ezért, ha az időigényesebb megoldást választjuk, implementálhatjuk úgy a `ConfigureAdapter()` metódust, hogy manuálisan létrehozunk három új `SqlCommand` objektumot, amelyek mindegyike tartalmazza a szükséges `SqlParametereket`. Ezután mindegyik objektumot beállíthatjuk az illesztő `UpdateCommand`, `DeleteCommand` `InsertCommand` tulajdonságához.

Szerencsére a Visual Studio 2008 több tervezőeszközt biztosít, amelyek kiváltják ezt a kódolási feladatot. Néhány ilyen egyszerűsítést megvizsgálunk a fejezet összefoglalójában. Ahelyett, hogy a rengeteg kódutasítással bajlódnánk az adatillesztő teljes konfigurálásához, egyszerűsítsük a feladatot a `ConfigureAdapter()` metódus következő megvalósításával:

```
private void ConfigureAdapter(out SqlDataAdapter dAdapt)
{
  // Az illesztő létrehozása és a SelectCommand beállítása.
  dAdapt = new SqlDataAdapter("Select * From Inventory", cnString);

  // A fennmaradó parancsobjektumok dinamikus megszerzése
  // futásidőben, az SqlCommandBuilder használatával.
  SqlCommandBuilder builder = new SqlCommandBuilder(dAdapt);
}
```

Az adatillesztő objektumok létrehozásának egyszerűsítéséhez, az ADO.NET-adatszolgáltatók mindegyike biztosít egy *parancsépítő (Command Builder)* típust. Az `SqlCommandBuilder` automatikusan generálja az `SqlDataAdapter`, az `InsertCommand`, az `UpdateCommand` és a `DeleteCommand` tulajdonságaiban lévő értékeket a kezdeti `SelectCommand` alapján. Ennek előnye nyilvánvalóan az, hogy nem kell manuálisan létrehoznunk az összes `SqlCommand` és `SqlParameter` típust.

A kérdés magától értetődően felmerül: a parancsépítő hogyan tudja létrehozni ezeket az SQL-parancsobjektumokat menet közben? Röviden a válasz a metaadat. Futásidőben, amikor meghívjuk az adatillesztő `Update()` metódusát, a parancsépítő kiolvassa az adatbázis sémájának adatait, hogy automatikusan generálja az alapul szolgáló beszúrási, törlési és módosítási parancsobjektumokat.

Ehhez nyilvánvalóan üzenetváltásra van szükség a távoli adatbázissal, ezért biztosan kisebb lesz a teljesítmény, ha egy alkalmazáson belül többször is használjuk az `SqlCommandBuildert`. Úgy csökkentjük ezt a negatív hatást, hogy akkor hívjuk meg a `ConfigureAdapter()` metódust, amikor létrehozzuk az `InventoryDALDisLayer` objektumot, és megtartjuk a konfigurált `SqlDataAdaptert` az objektum teljes életciklusának az idejére.

Az előző kódban megfigyelhettük, hogy nem használtuk a parancsépítő objektumot (jelen esetben az `SqlCommandBuilder` objektumot), csak átadtuk az adatillesztő objektumot konstruktorparaméterként. Ennél többet nem is kell tennünk (tehetünk, ha szeretnénk, de nem szükséges). A felszín alatt ez a típus beállítja az adatillesztő többi parancsobjektumát.

A parancsépítők azonban több komoly korlátozással bírnak. Egészen pontosan, a parancsépítő csak akkor képes automatikusan SQL-parancsokat generálni az adatillesztőhöz, ha az alábbi feltételek mindegyike megvalósul:

- Az SQL `Select` parancsa csak egyetlen táblával lép kapcsolatba (azaz nincsenek összekapcsolások).
- Ez a tábla rendelkezik elsődleges kulccsal.
- Az SQL `Select` utasításnak tartalmaznia kell az(oka)t az oszlopo(ka)t, amely(ek) a tábla elsődleges kulcsa(i).

Az AutoLot adatbázis létrehozása alapján ezek a korlátozások nem jelentenek problémát. Egy „iparibb" adatbázis esetén azonban meg kell fontolnunk, hogy hasznos-e ez a típus (ha nem, akkor a Visual Studio 2008 automatikusan generálja a szükséges kód jó részét).

A GetAllInventory() implementálása

Az új osztálytípusunk első metódusa az `SqlDataAdapter` objektum `Fill()` metódusát fogja használni arra, hogy beolvassa az AutoLot adatbázis Inventory táblájának összes rekordját jelképező `DataTable` objektumot:

```
public DataTable GetAllInventory()
{
  DataTable inv = new DataTable("Inventory");
  dAdapt.Fill(inv);
  return inv;
}
```

Az UpdateInventory() implementálása

Az `UpdateInventory()` metódus nagyon egyszerű:

```
public void UpdateInventory(DataTable modifiedTable)
{
  dAdapt.Update(modifiedTable);
}
```

Az adatillesztő objektum itt megvizsgálja a bejövő `DataTable` minden sorának `RowState` értékét. Ennek az értéknek (`RowState.Added`, `RowState.Deleted` vagy `RowState.Modified`) az alapján a megfelelő parancsobjektumot használja fel a háttérben.

Forráskód Az AutoLotDAL (Part 2) kódfájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

Windows Forms front end létrehozása

Most már létrehozhatunk egy front endet, hogy teszteljük az új `InventoryDALDisLayer` objektumunkat, amely egy új Windows Forms-alkalmazás lesz WindowsFormsInventoryUI néven. Ha létrehoztuk a projektet, állítsunk be egy referenciát a módosított `AutoLotDAL.dll` szerelvényre, és importáljuk a következő névteret:

```
using AutoLotDisconnectedLayer;
```

Az űrlap egy `Label`, egy `DataGridView` (neve: `inventoryGrid`) és egy `Button` (neve: `btnUpdateInventory`) típusból áll, ez utóbbi fogja kezelni a *Click* eseményt. Az űrlap definíciója (amely az egyszerűség kedvéért nem tartalmazza a hibakezelési logikát; ám nyugodtan hozzáadhatunk egy try/catch blokkot, ha szeretnénk) a következő:

```
public partial class MainForm : Form
{
  InventoryDALDisLayer dal = null;

  public MainForm()
  {
    InitializeComponent();
    // Tételezzük fel, hogy van egy App.config fájlunk, amely
    // a kapcsolatsztringet tárolja.
    string cnStr =
           ConfigurationManager.ConnectionStrings[
           "AutoLotSqlProvider"].ConnectionString;

    // Hozzuk létre az adathozzáférési objektumunkat.
    dal = new InventoryDALDisLayer(cnStr);

    // Töltsük fel a táblázatot.
    inventoryGrid.DataSource = dal.GetAllInventory();
  }

  private void btnUpdateInventory_Click(object sender, EventArgs e)
  {
    // Szerezzük meg a módosított adatokat a táblázatból.
    DataTable changedDT = (DataTable)inventoryGrid.DataSource;

    // Hagyjuk jóvá a módosításokat.
    dal.UpdateInventory(changedDT);
  }
}
```

Ez a példa azt feltételezi, hogy létezik egy `App.config` fájl, amely tárolja a kapcsolatsztringadatokat a `<connectionStrings>` részben. A `ConfigurationManager` típus `ConnectionStrings` indexelőjének használatához győződjünk meg, hogy referenciát adtunk a `System.Configuration.dll` szerelvényre. Miután létrehoztuk az `InventoryDALDisLayer` objektumot, kössük hozzá a `GetAllInventory()` metódusból visszaadott `DataTable` típust a `DataGridView` objektumhoz. Amikor a felhasználó az `Update` gombra kattint, kiszedjük a módosított `DataTable` típust a táblázatból (a `DataSource` tulajdonsággal), és átadjuk az `UpdateInventory()` metódusunknak.

Ha futtatjuk az alkalmazást, adjunk hozzá új sorokat a táblázathoz, és módosítsunk/töröljünk néhány másikat. Ha a gombra kattintunk, látni fogjuk, hogy a módosításokat az AutoLot adatbázis rögzítette.

Forráskód A módosított WindowsFormsInventoryUI kódfájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

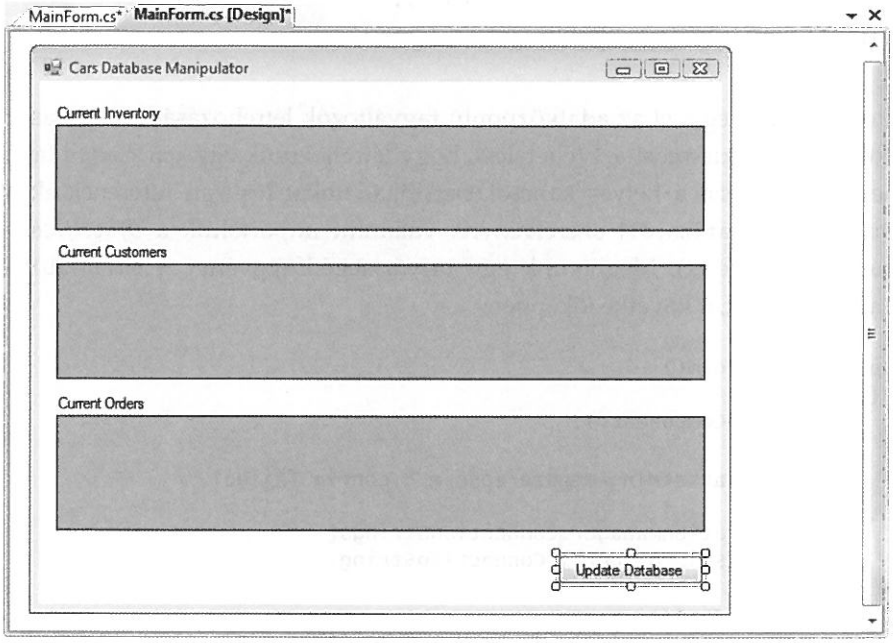
Navigálás a többtáblázatos DataSet objektumokban

Eddig a fejezet összes példájában egyetlen `DataTable` objektummal dolgoztunk. A kapcsolat nélküli modell valódi erejére akkor derül igazán fény, ha egy `DataSet` objektum több, egymással összefüggő `DataTable` objektumot tartalmaz. Ilyenkor tetszőleges számú `DataRelation` objektumot szúrhatunk be a `DataSet` `DataRelation` gyűjteményébe, hogy gondoskodjunk a táblázatok összefüggéseiről. Ezekkel az objektumokkal a kliensréteg navigálhat a táblázat adatai között anélkül, hogy hálózati forgalmat generálna.

Megjegyzés Ahelyett, hogy ismét módosítanánk az `AutoLotDAL.dll` szerelvényt, hogy használja a Customers és Orders táblákat, ez a példa új Windows Forms-projektben különít el minden adatelérési logikát. Mindazonáltal kifejezetten ellenjavallt összekeverni a felhasználóifelület- és az adatlogikát egy termékszintű alkalmazásban. A fejezet utolsó példái kihasználják a különböző adatbázistervező eszközöket, hogy elkülönítsék a felhasználóifelület- és adatlogikakódokat.

Kezdjük ezt a példát egy új Windows Forms-alkalmazás létrehozásával MultitabledDataSetApp néven. A grafikus felület meglehetősen egyszerű. A 23.13. ábrán láthatunk három DataGridViewt, amelyek az AutoLot adatbá-

zis Inventory, Orders és Customers táblázataiból lekért adatokat tartalmazzák. Emellett a kezdeti `Button` (`btnUpdateDatabase`) típus az adatillesztő objektumok révén az adatbázisnak visszaküldi feldolgozásra az összes módosítást, amelyet beírtunk a táblázatba.



23.13. ábra: *A kezdeti felhasználói felület megjeleníti az AutoLot adatbázis tábláinak adatait*

Az adatillesztők előkészítése

Ahhoz, hogy az adat-hozzáférési kód a lehető legegyszerűbb legyen, a `MainForm` parancsépítő objektumokat használ az SQL-parancsok automatikus generálásához mindhárom `SqlDataAdapter` esetében (egy darabot mindegyik táblázathoz). A Form-leszármazott típus első módosítása a következő:

```
public partial class MainForm : Form
{
  // Az egész űrlapra vonatkozó DataSet.
  private DataSet autoLotDS = new DataSet("AutoLot");

  // Parancsépítők használata az adatillesztő konfigurálásának
  // egyszerűsítéséhez.
  private SqlCommandBuilder sqlCBInventory;
  private SqlCommandBuilder sqlCBCustomers;
  private SqlCommandBuilder sqlCBOrders;
```

```
    // Az adatillesztők (minden táblázathoz).
    private SqlDataAdapter invTableAdapter;
    private SqlDataAdapter custTableAdapter;
    private SqlDataAdapter ordersTableAdapter;

    // Az egész űrlapra vonatkozó kapcsolatsztring.
    private string cnStr = string.Empty;
    ...
}
```

A konstruktor végzi el az adatközpontú tagváltozók létrehozását és a `DataSet` feltöltését. A példa most azt feltételezi, hogy létrehoztunk egy `App.config` fájlt, amely tartalmazza a helyes kapcsolatsztringadatokat (és van referenciánk a `System.Configuration.dll` szerelvényre, valamint importáltuk a `System.Configuration` névteret). Meghívunk egy privát segédfüggvényt, a `BuildTableRelationship()`-t, a következőképpen:

```
public MainForm()
{
  InitializeComponent();

  // Kapcsolatsztring megszerzése a *.config fájlból.
  cnStr =
    ConfigurationManager.ConnectionStrings[
    "AutoLotSqlProvider"].ConnectionString;

  // Illesztők létrehozása.
  invTableAdapter = new SqlDataAdapter("Select * from Inventory",
                                       cnStr);
  custTableAdapter = new SqlDataAdapter("Select * from Customers",
                                        cnStr);
  ordersTableAdapter = new SqlDataAdapter("Select * from Orders",
                                          cnStr);

  // Parancsok automatikus generálása.
  sqlCBInventory = new SqlCommandBuilder(invTableAdapter);
  sqlCBOrders = new SqlCommandBuilder(ordersTableAdapter);
  sqlCBCustomers = new SqlCommandBuilder(custTableAdapter);

  // Táblák hozzáadása a DataSethez.
  invTableAdapter.Fill(autoLotDS, "Inventory");
  custTableAdapter.Fill(autoLotDS, "Customers");
  ordersTableAdapter.Fill(autoLotDS, "Orders");

  // Kapcsolatok kiépítése a táblázatok között.
  BuildTableRelationship();
```

```
  // Hozzákötés a táblázatokhoz.
  dataGridViewInventory.DataSource = autoLotDS.Tables["Inventory"];
  dataGridViewCustomers.DataSource = autoLotDS.Tables["Customers"];
  dataGridViewOrders.DataSource = autoLotDS.Tables["Orders"];
}
```

A táblázatok közötti kapcsolatok kiépítése

A `BuildTableRelationship()` segédfüggvény végzi el a két `DataRelation` objektum hozzáadását az `autoLotDS` objektumhoz. Az AutoLot adatbázis több olyan szülő/gyermek kapcsolatot kifejez, amelyeket a következő kóddal veszünk figyelembe (lásd a 22. fejezetet):

```
private void BuildTableRelationship()
{
  // CustomerOrder adatkapcsolati objektum létrehozása.
  DataRelation dr = new DataRelation("CustomerOrder",
              autoLotDS.Tables["Customers"].Columns["CustID"],
              autoLotDS.Tables["Orders"].Columns["CustID"]);
  autoLotDS.Relations.Add(dr);

  // InventoryOrder adatkapcsolati objektum létrehozása.
  dr = new DataRelation("InventoryOrder",
       autoLotDS.Tables["Inventory"].Columns["CarID"],
       autoLotDS.Tables["Orders"].Columns["CarID"]);
  autoLotDS.Relations.Add(dr);
}
```

Amikor létrehozunk egy `DataRelation` objektumot, akkor egy barátságos sztringmonikert adunk meg az első paraméterrel (ennek hasznát lásd később), ezután magának a kapcsolatnak a kiépítéséhez használt kulcsok követnek. A szülőtáblázatot (a második konstruktorparaméter) a gyermektáblázat (harmadik konstruktorparaméter) előtt adjuk meg.

Az adatbázistáblák módosítása

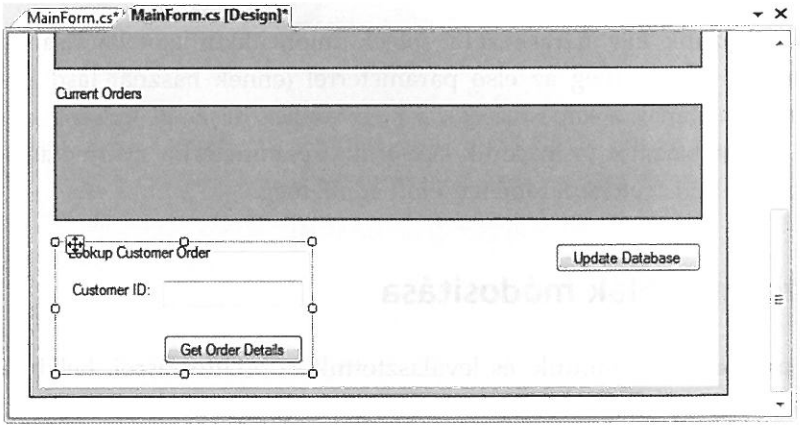
Miután a `DataSetet` feltöltöttük, és leválasztottuk az adatforrásról, helyben kezelhetünk minden `DataTable` objektumot. Ehhez egyszerűen illesszünk be, módosítsunk vagy töröljünk értékeket a három `DataGridView` bármelyikéből. Amikor vissza szeretnénk küldeni az adatokat feldolgozásra, kattintsunk az `Update` gombra. A *Click* esemény mögötti kód ekkor már egyértelmű:

```
private void btnUpdateDatabase_Click(object sender, EventArgs e)
{
  try
  {
    invTableAdapter.Update(carsDS, "Inventory");
    custTableAdapter.Update(carsDS, "Customers");
    ordersTableAdapter.Update(carsDS, "Orders");
  }
  catch (Exception ex)
  {
    MessageBox.Show(ex.Message);
  }
}
```

Futtassuk az alkalmazást, és hajtsunk végre különböző módosításokat. Amikor újra futtatjuk az alkalmazást, a táblázatokban megjelennek az korábbi változtatások.

Navigálás a kapcsolódó táblázatok között

Annak illusztrálására, hogy a `DataRelation` programozottan hogyan teszi lehetővé a mozgást a kapcsolódó táblák között, bővítsük a felhasználói felületet új gombbal (`btnGetOrderInfo`), a hozzá kapcsolódó szövegdobozzal (`txtCustID`) és leíró címkével (a tetszetősség kedvéért ezeket egy `GroupBox` típuson belül csoportosítjuk). A 23.14. ábra egy lehetséges elrendezést mutat.



23.14. ábra: *A módosított felhasználói felület lehetővé teszi a felhasználó számára, hogy utánanézzen az ügyfelek megrendeléseinek*

A módosított felület segítségével a felhasználó beírhatja egy ügyfél azonosítóját, és visszakeresheti az ügyfél rendelésével kapcsolatos információkat (név, megrendelés azonosítója, autórendelés stb.), amelyeket a rendszer sztringtípussá formáz, és végül egy üzenetdobozban megjelenít. Gondoljuk végig az új gomb `Click` eseménykezelője mögötti kódot:

```
private void btnGetOrderInfo_Click(object sender,
                                   System.EventArgs e)
{
  string strOrderInfo = string.Empty;
  DataRow[] drsCust = null;
  DataRow[] drsOrder = null;

  // Megkapjuk az ügyfél azonosítóját a szövegdobozban.
  int custID = int.Parse(this.txtCustID.Text);

  // Most a custID alapján lekérdezzük a Customers táblázat
  // megfelelő sorát.
  drsCust = autoLotDS.Tables["Customers"].Select(
            string.Format("CustID = {0}", custID));
  strOrderInfo += string.Format("Customer {0}: {1} {2}\n",
                  drsCust[0]["CustID"].ToString(),
                  drsCust[0]["FirstName"].ToString().Trim(),
                  drsCust[0]["LastName"].ToString().Trim());

  // A Customers táblázatból navigáljunk az Orders táblázatba.
  drsOrder =
     drsCust[0].GetChildRows(autoLotDS.Relations["CustomerOrder"]);

  // Lekérdezzük a rendelés számát.
  foreach (DataRow r in drsOrder)
    strOrderInfo += string.Format("Order Number: {0}\n",
                                  r["OrderID"]);

  // Most navigáljunk az Orders táblázatból az Inventory táblázatba.
  DataRow[] drsInv =
            drsOrder[0].GetParentRows(
            autoLotDS.Relations["InventoryOrder"]);

  // Lekérdezzük az autó adatait.
  foreach (DataRow r in drsInv)
  {
    strOrderInfo += string.Format("Make: {0}\n", r["Make"]);
    strOrderInfo += string.Format("Color: {0}\n", r["Color"]);
    strOrderInfo += string.Format("Pet Name: {0}\n", r["PetName"]);
  }
  MessageBox.Show(strOrderInfo, "Order Details");
}
```

Elemezzük a kódot lépésről lépésre. Először megszerezzük az ügyfél azonosítóját a szövegdobozból, és felhasználjuk arra, hogy kiválasszuk a megfelelő sort a Customers táblázatból a `Select()` metódus segítségével. Mivel a `Select()` metódus a `DataRow` objektumok *tömbjét* adja vissza, kétszeres indexelést kell használnunk annak biztosítására, hogy a tömb első (és egyetlen) tagjának adatait kapjuk meg:

```
// Megkapjuk az ügyfél azonosítóját a szövegdobozban.
int custID = int.Parse(this.txtCustID.Text);

// Most a custID alapján lekérdezzük a Customers táblázat
// megfelelő sorát.
drsCust = autoLotDS.Tables["Customers"].Select(
          string.Format("CustID = {0}", custID));
strOrderInfo += string.Format("Customer {0}: {1} {2}\n",
                        drsCust[0]["CustID"].ToString(),
                        drsCust[0]["FirstName"].ToString().Trim(),
                        drsCust[0]["LastName"].ToString().Trim());
```

Ezután a Customers táblázatból az Orders táblázatba navigálunk a `CustomerOrder` adatkapcsolat segítségével. A `DataRow.GetChildRows()` metódus lehetővé teszi sorok kiragadását a gyermektáblázatból. Amint ez kész, kiolvashatjuk belőle az adatokat:

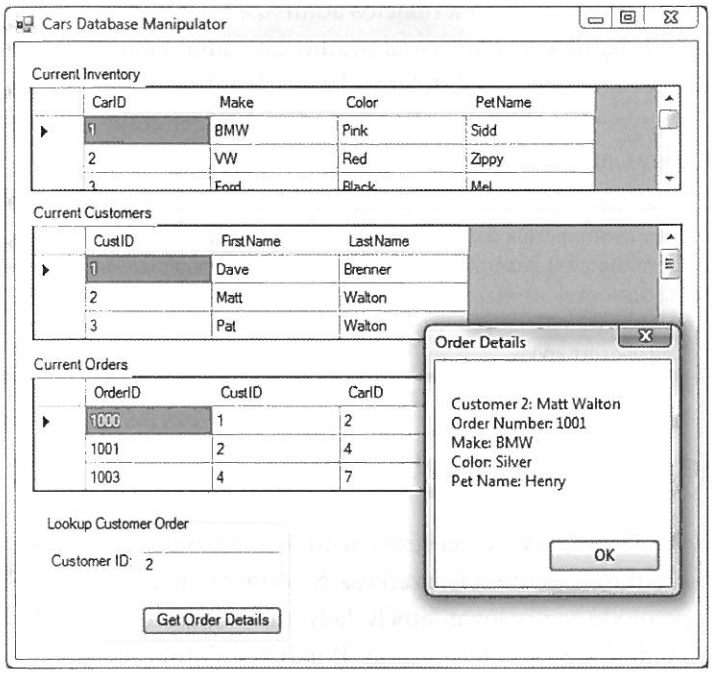
```
// A Customers táblázatból navigáljunk az Orders táblázatba.
drsOrder =
   drsCust[0].GetChildRows(autoLotDS.Relations["CustomerOrder"]);

// Lekérdezzük a rendelés számát.
foreach (DataRow r in drsOrder)
  strOrderInfo += string.Format("Order Number: {0}\n",
                                r["OrderID"]);
```

Az utolsó lépés az, hogy az Orders táblából annak szülőtáblázatába (Inventory) navigálunk a `GetParentRows()` metódus használatával. Ekkor kiolvashatjuk az adatokat az Inventory táblából a Make, PetName és Color oszlopok használatával, a következőképpen:

```
// Most navigáljunk az Orders táblázatból az Inventory táblázatba.
DataRow[] drsInv =
          drsOrder[0].GetParentRows(
          autoLotDS.Relations["InventoryOrder"]);
```

```
// Lekérdezzük az autó adatait.
foreach (DataRow r in drsInv)
{
  strOrderInfo += string.Format("Make: {0}\n", r["Make"]);
  strOrderInfo += string.Format("Color: {0}\n", r["Color"]);
  strOrderInfo += string.Format("Pet Name: {0}\n", r["PetName"]);
}
```

A 23.15. ábra egy lehetséges kimenetet mutat, ha a 2-es ügyfélazonosítót adjuk meg (a mi AutoLot adatbázisunkban Matt Walton).



23.15. ábra: *Navigálás az adatkapcsolatok között*

Mivel a `DataSet` egyáltalán nem kapcsolódik az alapjául szolgáló adatforráshoz, dolgozhatunk az adatok memóriában lévő másolatával, és körbemehetünk minden táblázaton, hogy elvégezzük a szükséges módosításokat, törléseket és beszúrásokat. Ha végeztünk, elküldhetjük feldolgozásra a módosításokat az adattárolónak. A végeredmény egy rendkívül skálázható és stabil alkalmazás lesz.

Forráskód A MultitabledDataSetApp kódfájlokat a forráskódkönyvtár 23. fejezetének alkönyvtára tartalmazza. A forráskódkönyvtárról lásd a Bevezetés xlv. oldalát.

A Visual Studio 2008 adatelérési eszközei

Az eddigi ADO.NET-példák mindegyike megterhelő annyiban, hogy minden adatelérési logikát manuálisan hoztunk létre. Bár az előbb említett kód jó részét áthelyeztük egy .NET-kódkönyvtárba (`AutoLotDAL.dll`) a későbbi újrafelhasználás érdekében, mégis manuálisan kell létrehoznunk az adatszolgáltatónk különböző objektumait, mielőtt a relációs adatbázissal dolgozhatnánk.

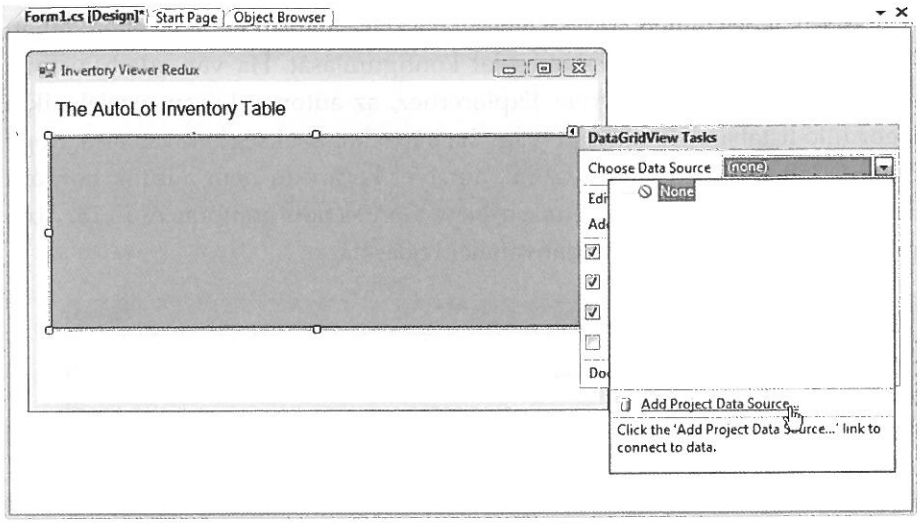
A következőkben vizsgáljuk meg a Visual Studio 2008 által biztosított néhány eszközt, amelyek segítenek az adatelérési logika létrehozásában. Ez az integrált fejlesztői környezet jó pár vizuális tervező és kódgeneráló eszközt (vagy varázslót) támogat, amelyek a kiinduló kód jó részét megalkotják.

Megjegyzés Nyilvánvalóan nem hihetjük azt, hogy soha nem kell manuálisan létrehoznunk ADO.NET-logikát, vagy a varázsló által generált kód mindig tökéletesen megfelelő lesz az aktuális projektünk számára. Noha ezek az eszközök rengeteg időt megtakaríthatnak, mégis érdemes minél többet tudunk az ADO.NET programozási modelljéről, ugyanis ezáltal egyedi igényeinkhez alakíthatjuk a létrehozott kódot.

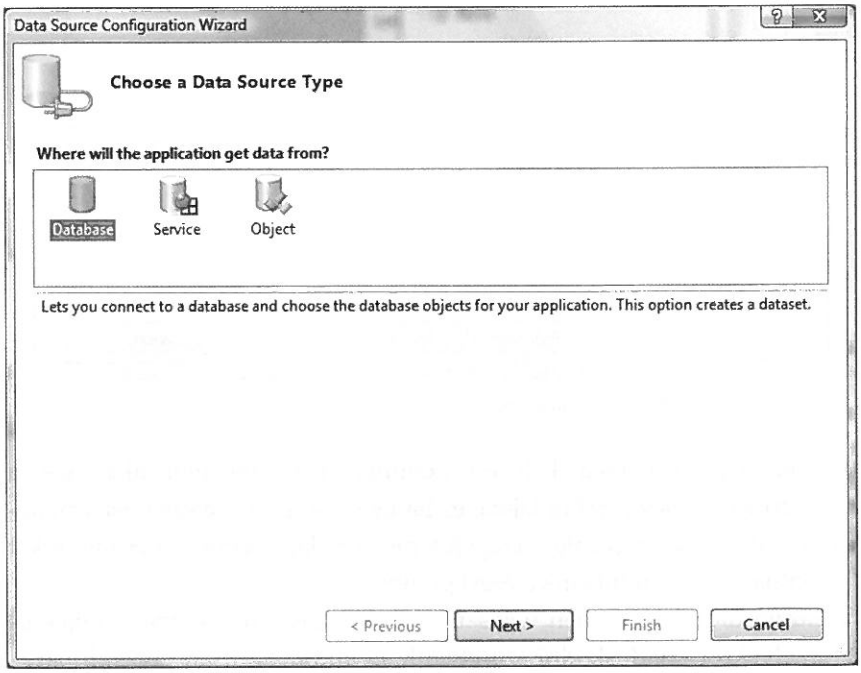
A DataGridView vizuális megtervezése

Az első egyszerűsítést a DataGridView tervező jelenti. Bár előző példákban már használtuk ezt az eszközt megjelenítési és szerkesztési célokra, még nem foglalkoztunk a hozzá kapcsolódó varázslóval, amely helyettünk generál adatelérési kódot. Először hozzunk létre egy teljesen új Windows Forms-alkalmazásprojektet VisualDataGridViewApp néven. Adjunk hozzá egy leíró `Label`t, valamint a `DataGridView` vezérlőelem egy példányát. Ekkor megnyílik egy inline szerkesztő a felhasználói felülettől jobbra. A Choose Data Soure legördülő listából válasszuk az Add Project Data Source lehetőséget (lásd a 23.16. ábrát).

Ezzel elindul a Data Source Configuration Wizard. Ez az eszköz néhány lépésben segít kiválasztani és beállítani egy adatforrást, amelyet aztán egyedi adatillesztő típussal hozzákötünk a `DataGridView` típushoz. Az első lépésben a varázsló megkér minket, hogy adjuk meg azt az adatforrástípust, amellyel dolgozni szeretnénk. Válasszuk a Database lehetőséget (lásd a 23.17. ábrát), majd kattintsunk a Next gombra.



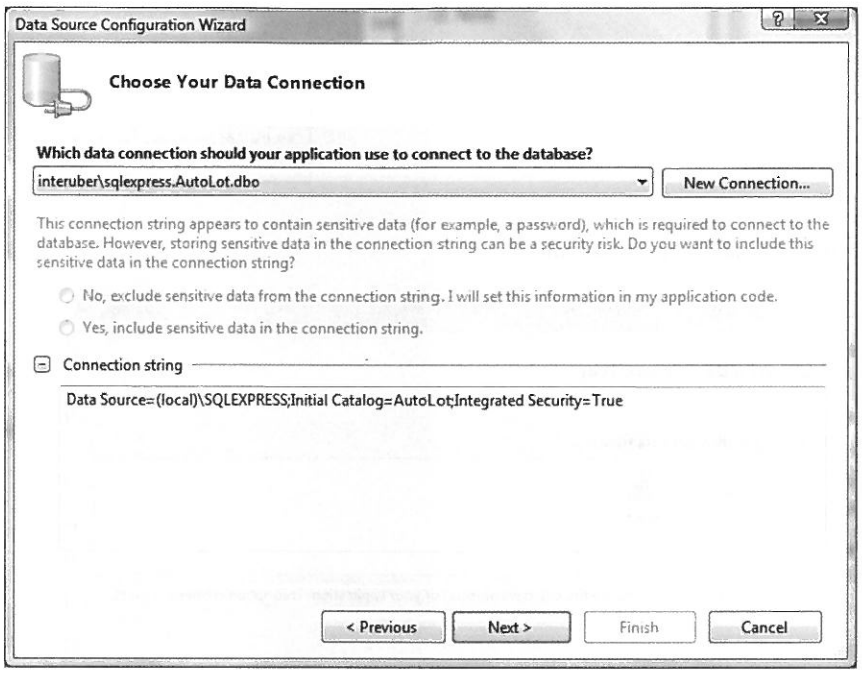
23.16. ábra: *A DataGridView szerkesztő*



23.17. ábra: *Az adatforrás típusának kiválasztása*

Megjegyzés Ebben a lépésben arra is lehetőségünk nyílik, hogy egy külső XML-webszolgáltatásból vagy külön .NET-szerelvényben lévő egyedi üzleti objektumból származó adatokat csatlakoztassunk.

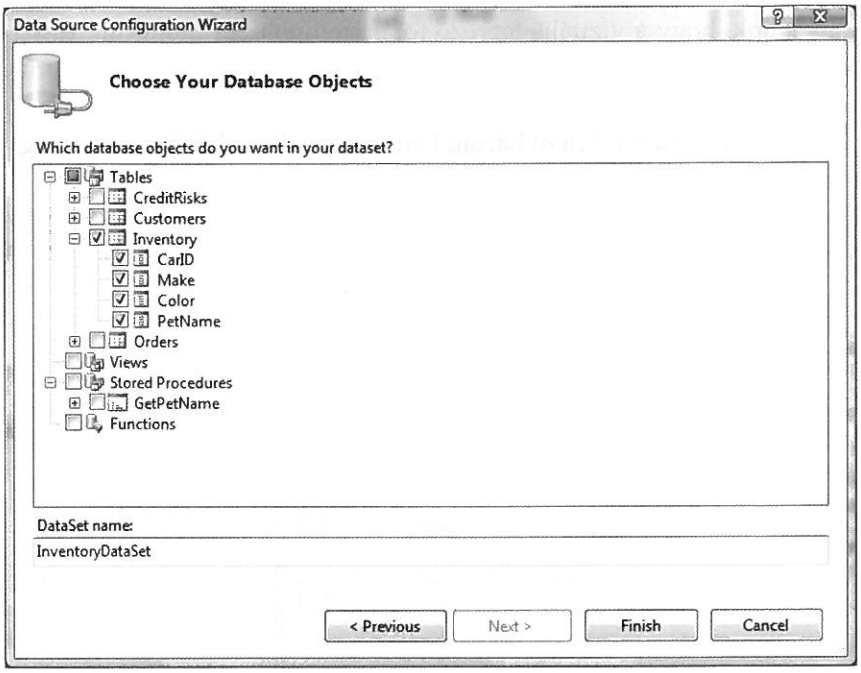
A második lépés (amely az első lépés választása alapján némileg eltérő lehet) lehetővé teszi az adatbázis-kapcsolat konfigurálását. Ha van adatbázisunk, amelyet hozzáadtunk a Server Explorerhez, az automatikusan megjelenik a legördülő listában. Ha nincsen (vagy ha bármikor szükségünk van arra, hogy olyan adatbázishoz csatlakozzunk, amelyet korábban nem adtunk hozzá a Server Explorerhez), kattintsunk a New Connection gombra. A 23.18. ábra mutatja az AutoLot helyi példányának kiválasztását.



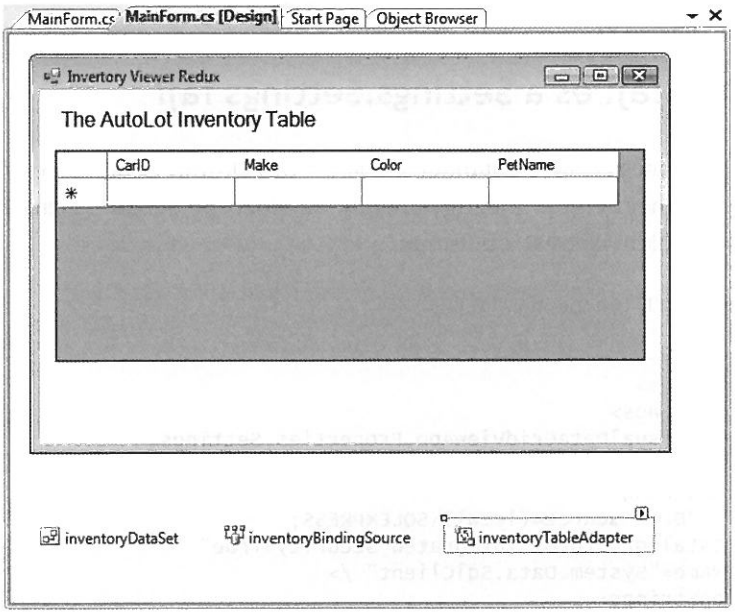
23.18. ábra: *Az AutoLot adatbázis kiválasztása*

A harmadik lépésben meg kell erősítenünk, hogy menteni akarjuk-e a kapcsolatsztringet az `App.config` fájlba, és ha igen, akkor a `<connectionStrings>` elemben használt nevet ugyanígy. Hagyjuk meg az alapértelmezett beállításokat ennél a pontnál, majd kattintsunk a Next gombra.

A varázsló utolsó lépésében választhatjuk ki azokat az adatbázis-objektumokat, amelyekről gondoskodni szeretnénk az automatikusan generált `DataSet` és a kapcsolódó adatillesztők segítségével. Noha az AutoLot adatbázis adatobjektumainak mindegyikét kiválaszthatjuk, most csak az Inventory táblázattal foglalkozunk. Ennek alapján változtassuk meg a javasolt DataSet nevet InventoryDataSetre (lásd a 23.19. ábrát), jelöljük ki az Inventory táblát, majd kattintsunk a Finish gombra.



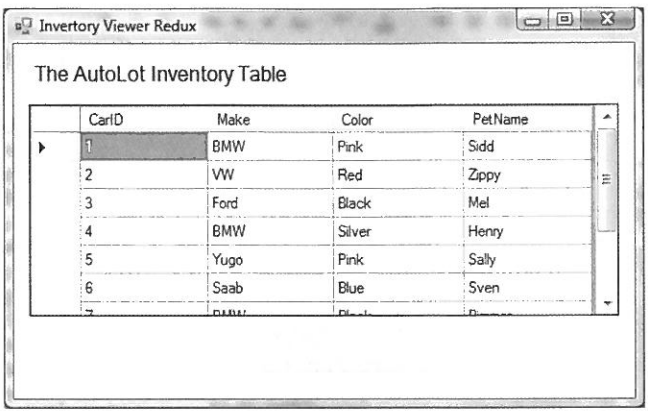
23.19. ábra: *Az Inventory tábla kiválasztása*



23.20. ábra: *A Windows Forms-projektünk a Data Source Configuration Wizard futtatása után*

Ekkor láthatjuk, hogy a vizuális tervező több szempontból módosult. A legfeltűnőbb az a tény, hogy a DataGridView az Inventory táblázat sémáját mutatja úgy, ahogy azt az oszlopok fejlécei illusztrálják. Emellett az űrlaptervező alján (a *komponenstálca* területén) három komponenst látunk: egy `DataSet`, egy `BindingSource` és egy `TableAdapter` komponenst (lásd a 23.20. ábrát).

Ekkor futtatható az alkalmazás, a táblázatot a rendszer feltöltötte az Inventory tábla rekordjaival (lásd a 23.21. ábrát).



23.21. ábra: *Feltöltött DataGridView – nincs szükség manuális kódolásra*

Az App.config fájl és a Settings.Settings fájl

A Solution Explorert megvizsgálva láthatjuk, hogy a projektünk tartalmaz egy `App.config` fájlt. Ha megnyitjuk, megfigyelhetjük az előző példában használt `<connectionStrings>` elem *name* attribútumát:

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
  <connectionStrings>
    <add name="VisualDataGridViewApp.Properties.Settings.
              AutoLotConnectionString"
    connectionString=
             "Data Source=(local)\SQLEXPRESS;
    Initial Catalog=AutoLot;Integrated Security=True"
    providerName="System.Data.SqlClient" />
  </connectionStrings>
</configuration>
```