

ADO.NET osztályai

ADO.NET OSZTÁLYAI	1
0. ESZKÖZÖK	2
<i>Adatbázis szerver</i>	2
<i>ADO.NET osztályai</i>	2
<i>Server Explorer</i>	2
<i>SQL Server Management Studio Express</i>	3
1. SZÁMLÁLÁS EGY ADATTÁBLÁBAN	4
2. ADATTÁBLA MÓDOSÍTÁSA	7
3. ADATTÁBLA ADATAINAK MEGJELENÍTÉSE	8
4. TÖBB LÉPÉSES ADATBÁZIS MŰVELETEK	11
5. ADATTÁBLA-KARBANTARTÁS A MEMÓRIÁBAN	12
<i>Első megoldás (elemi eszközökkel)</i>	12
<i>Második megoldás (adapterrel)</i>	21
<i>Harmadik megoldás (varázslóval)</i>	24

A .NET Framework az adatbázisok kezelését az ADO.NET-ben definiált osztályokkal támogatja. Ennek a fejezetnek az a célja, hogy kisméretű alkalmazások bemutatásán keresztül ízelítőt adjon ezen osztályok használatáról.¹ Nem törekszünk ezen osztályok minden részletre kiterjedő ismertetésével – ezt mind nyomtatott, mind online szakirodalmakban megtalálhatjuk –, helyette az alkalmazásaikra helyezük a hangsúlyt. Az alábbiakban olyan konzol-alkalmazásokat mutatunk, amelyekben különféle SQL parancsokkal, tárolt eljárásokkal és tranzakciókkal kezelünk egy adatbázist.

A feladatokat egy képzelt utazási iroda apartman-foglalási tevékenységéhez használt „Apartments” adatbázisára fogalmazzuk meg. Ebben a fejezetben ennek az adatbázisnak kizárólag az épületek leírására szolgáló „building” nevű táblája szerepel.

```
CREATE TABLE building (
    building_id INTEGER PRIMARY KEY,
    name VARCHAR(30),
    city_id INTEGER NOT NULL,
    street VARCHAR(30) NOT NULL,
    sea_distance INTEGER,
    shore_id INTEGER,
    features INTEGER,
    comment VARCHAR(100),
    FOREIGN KEY (city_id) REFERENCES city,
    FOREIGN KEY (shore_id) REFERENCES shore
);
```

A feladatok kitűzése és megoldása előtt néhány szó az alkalmazásokban használt eszközökről.

¹ Az osztályok bemutatásánál Jason Price *C# adatbázisprogramozás* c. könyvére (2004, KisKapu Kft.) támaszkodtam

0. ESZKÖZÖK

Adatbázis szerver

Az adatbázisok kezeléséhez (létrehozásához, tárolásához, tulajdonságainak beállításához, adatainak felviteléhez, törléséhez, módosításához) valamilyen adatbázis szerverre van szükségünk. A .NET sokféle adatbázis szerver használatát támogatja, de ezek között prioritást élvez a Microsoft saját terméke: az MS SQL SERVER. A .NET feltelepítésével együtt hozzájuthatunk ennek egyszerűbb változatához az MS SQL SERVER EXPRESS-hez. A továbbiakban bemutatott példa-alkalmazásokban ezt használjuk.²

ADO.NET osztályai

Az ADO.NET azoknak az osztályoknak a gyűjteménye, amelyek az adatkezelést támogatják. Ezen belül beszélhetünk az úgynevezett adatszolgáltató osztályokról, amelyek az adatbázis és a kliens program közötti adatforgalom megvalósításához nyújtanak segítséget, valamint az úgynevezett adattárolási osztályokról, amelyek az adatbázisból letöltött adatok tárolására, manipulálására adnak lehetőséget. Az adatszolgáltató osztályokat aszerint csoportosítjuk, hogy milyen típusú kommunikációt támogatnak adatbázis szerverrel. A különböző csoportok osztályai más-más névtérben találhatóak, és az osztályok nevének előtagja (prefixe) is eltér. Ugyanakkor a különböző előtagú, de különben azonos nevű osztályok funkcionalitása megegyezik.

SQL Server

- SQL Server 7.0 -tól; ilyen például az MS SQL Express
- System.Data.SqlClient névtérben "Sql"-lel kezdődő osztálynevek

OLE DB (Object Linking and Embedding for Databases)

- OLE DB-t támogató adatszerverek; ilyenek SQL Server 7.0 alatti verziói
- System.Data.OleDb névtérben "OleDb"-vel kezdődő osztálynevek

ODBC (Open Database Connectivity)

- ODBC-t támogató adatbázisszerverek, ilyen például az Access, a MySQL
- System.Data.Odbc névtérben "Odbc"-vel kezdődő osztálynevek

Oracle

- Oracle (8.1.7.-től) támogató adatbázisszerverek
- System.Data.OracleClient névtérben "OracleClient"-vel kezdődő osztálynevek

Server Explorer

Ha elindítjuk a .NET Visual Studio-ját, akkor a Server Explorer nézetben tudunk új adatbázist létrehozni, táblákat, tárolt eljárásokat definiálni, táblákat feltölteni, meglévő adatbázisokat megtekinteni, módosítani, kiegészíteni.

Előfordul, hogy a Visual Studio elindítását megelőzően más eszközzel (lásd következő pont) hoztunk létre egy új adatbázist, akkor az a Server Explorerben még nem látszik. Ekkor a „DataConnections” felett jobb egérfül kattintásra felnyíló menüben az Add Connection ... pontot kell kiválasztani, majd az erre megnyíló panelen a kívánt adatbázist beazonosítani.

² Ebben és a későbbi fejezetekben a .NET 2005-ös változatával teszteltem a mintaprogramokat.

SQL Server Managment Studio Express

Az SQL Server Managment Studio Express egy ingyenes szoftver, mellyel Visual Studio nélkül tudjuk az MS SQL SERVER-t kezelni. Segítségével új adatbázist hozhatunk létre, beállíthatjuk annak tulajdonságait, feltölthetjük adatokkal az adattábláit, SQL parancsokat, tárolt eljárásokat, tranzakciókat próbálhatunk ki.

Gyakran előforduló feladat, hogy egy adatbázist két munkahely MS SQL szervere között kell hordozni. Ilyenkor például a `dump database <adatbázisnév> to disk='fájlnév.dat'` parancs segítségével menthetjük el az adatbázist. Ez a fájl a Microsoft SQL Server\Mssql.1\Mssql\Backup mappába kerül. A fájlt átszállítva a másik szerver ugyanilyen nevű mappájába a `load database <adatbázisnév> from disk='fájlnév.dat'` paranccsal tudjuk az adatbázist birtokba venni.

1. Számlálás egy adattáblában

Feladat: A „building” tábla hány épülete (sora) található a „via Fausta” utcában?

A megoldáshoz két speciális objektumra lesz szükségünk.

Az adatbázishoz egy Connection típusú objektum (legyen a neve: con) segítségével tudunk kapcsolódni. A kapcsolat kiépítéséhez meg kell adnunk az úgynevezett kapcsolati sztringet, amely tartalmazza az adatbázis nevét, az adatbázis szerveret, a kapcsolat biztonsági szintjét. A con.Open() metódus kapcsolatot épít az adatbázissal, amit a con.Close() metódus bont szét. Amíg a kapcsolat fenn áll, csak mi használhatjuk az adatbázist.

Connection

Tulajdonság	Típus	Magyarázat
ConnectionString	string	A kapcsolati sztringet tárolja
State	ConnectionState	A kapcsolat státusza lehet: Open, Connecting, Executing, Fetching, Broken, Closed
Metódus	Visszatérési típus	Magyarázat
Open()	void	Megnyitja a kapcsolatot
Close()	void	Lezárja a kapcsolatot
CreateCommand()	SqlCommand	Létrehozza a kapcsolat egy parancsobjektumát

A kapcsolati objektum létrehozásának egy módja:

```
SqlConnection con = new SqlConnection (
    @"database=Apartments; server=computer\sqlexpress" +
    "Persist Security Info=False; Integrated Security=SSPI;"
);
```

A kapcsolati sztring megadásának van ennél elegánsabb módja is. Ilyenkor a kapcsolati sztringet nem közvetlenül a programkódba írjuk be, hanem egy úgynevezett konfigurációs fájlba. Ebben a fájlban XML formában szerepelhetnek olyan adatok, amelyeket a programunk használ, és amelyek megváltoztatásakor nem kell a programot újrafordítani. Ez akkor előnyös, amikor az adatbázis-kezelő alkalmazásunkat különböző környezetekbe telepítjük, ahol más-más kapcsolati sztringgel kell dolgoznunk.

Adjunk hozzá a projektünkhöz egy konfigurációs fájlt! Ehhez:

1. Kattintsunk a Solution Explorer-ben a projekt neve felett a jobb egérgombbal.
2. Válasszuk ki a felnyíló menüből az Add / New Item pontokat, majd a megjelenő ablakban az Application Configuration File sémát, és fogadjuk el a felajánlott fájlnevet (App.config).
3. Írjuk bele az így létrehozott konfigurációs fájlba az alábbi XML kódot! Ebben egyetlen egységet, egy ConnectionString-et definiálunk, amelynek van neve (Apartments) és tartalma (ez a kapcsolati sztring).

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <connectionStrings>
    <add
      name = "Apartments"
      connectionString = "server=computer\squlexpress;
                          database=Apartments;
                          Integrated Security=SSPI;
                          Persist Security Info=False;" />
    </connectionStrings>
  </configuration>
```

A kapcsolati objektum létrehozásánál az App.config állományban Apartments néven azonosított kapcsolati sztringet használjuk. Ehhez a sztringhez a programban úgy férhetünk hozzá, hogy példányosítunk egy kapcsolatsztring-beállító objektumot a ConfigurationManager osztály ConnectionStrings (statikus) tulajdonságának segítségével. Ennek át adjuk a konfigurációs állományban a kapcsolati sztringet azonosító „Apartments” nevet. Ezután a kapcsolatsztring-beállító objektum ConnectionString tulajdonsága tartalmazza a kapcsolati sztringet.

```
ConnectionStringSettings settings =ConfigurationManager.
    ConnectionStrings["Apartments"];
if (settings == null) return;

SqlConnection con = new
    SqlConnection(settings.ConnectionString);
```

A kód lefordításához szükség van a program elején a using System.Configuration sorra. Ezt a könyvtárat hozzá kell venni a projektünk referenciáihoz:

1. Kattintsunk a Solution Explorer-ben a projekt neve felett a jobb egérgombbal.
2. Válasszuk ki az Add Reference ... pontot, majd a megjelenő ablak Recent fülén a System Configuration komponenst.

A feladat megoldásához szükséges SQL lekérdezést ("SELECT COUNT(*) FROM building WHERE street = 'via Fausta'") egy Command (parancs) objektum (legyen a neve: cmd) segítségével küldhetjük el az adatbázisnak. A parancs objektumnak ismernie kell azt a kapcsolati objektumot (con), amely azonosítja számára a lekérdezés adatbázisát. Ez a cmd.Connection = con értékadással állítható be, vagy úgy, ha a cmd objektumot a con.CreateCommand() metódussal hozzuk létre. A lekérdezés elküldése előtt meg kell nyitnunk az adatbázis irányában a kapcsolatot. A lekérdezés elküldésének módja a végrehajtandó SQL parancs fajtájától függ. Ha az SQL parancs – mint ebben ez esetben is – válaszként egyetlen értéket vár, akkor a cmd parancsot annak ExecuteScalar() metódusával kell elküldeni az adatbáziszervernek. Ez a metódus visszatérési értékében adja vissza a lekérdezés eredményét. Az eredmény object típusú, amit a kívánt típusra tudunk konvertálni.

Command

Tulajdonság	Típus	Magyarázat
CommandText	string	SQL parancs szövege vagy a kiolvasandó tábla neve vagy a végrehajtandó tárolt eljárás neve
CommandType	CommandType	A CommandText-ben tárolt szöveg fajtája (Text, TableDirect, StoredProcedure) alapértelmezett értéke a Text.
Connection	SqlConnection	A parancshoz tartozó kapcsolati objektum
Parameters	SqlParameterCollection	Itt adhatóak meg a parancs paramétereinek konkrét értékei a parancs végrehajtása előtt.
Metódus	Visszatérési típus	Magyarázat
ExecuteScalar()	object	Megnyitja a kapcsolatot
ExecuteNonQuery()	int	Lezárja a kapcsolatot
ExecuteReader()	SqlReader	Létrehozza a kapcsolat egy parancsobjektumát

```

using System;
using System.Configuration;
using System.Data;
using System.Data.SqlClient;
...
ConnectionStringSettings settings = ConfigurationManager.
    ConnectionStrings["Apartments"];
if (settings == null) return;

SqlConnection con = new
    SqlConnection(settings.ConnectionString);

SqlCommand cmd = con.CreateCommand();
cmd.CommandText = "SELECT COUNT(*) FROM building" +
    "WHERE street = 'via Fausta' ";

try
{
    con.Open();
    int count = (int)cmd.ExecuteScalar();
    Console.WriteLine("Ekordok száma: {0}", count);
}
catch (SqlException ex)
{
    Console.WriteLine("Hiba: {0}", ex.Message);
}
finally
{
    con.Close();
    Console.ReadLine();
}

```

A kapcsolat létrehozása, illetve az SQL parancs végrehajtása során fellépő hibák `SqlException` kivételt dobhatnak, amelyeket kezelni kell.

2. Adattábla módosítása

Feladat: Szúrjuk be egy új épület adatait, töröljük a 2-es azonosítójú épületet, és változtassuk a 12-es azonosítójú épület tengertől mért távolságát 50-re!

A megoldáshoz szükség van egy kapcsolat (`SqlCommand con`) és három parancs (`SqlCommand insertcmd`, `deletecmd`, `updatecmd`) objektumra. Amikor az SQL parancs megváltoztatja az adatbázist (INSERT, DELETE, UPDATE, DROP, ALTER, stb.), a parancsot az `ExecuteNonQuery()` metódussal kell elküldeni az adatbázis szervernek. Az `ExecuteNonQuery()` metódus visszaadja a parancs által ténylegesen megváltoztatott sorok számát. Ezt hiba-ellenőrzésre tudjuk felhasználni. Természetesen itt is figyelni kell az `SQLException`-t, hiszen sokféle hiba (SQL parancs szintaxisa, elsődleges kulcs nem egyedi volta, nem megengedett null érték, nem létező kulcs, stb.) felléphet az alábbi kódrészletben.

```
...
SqlCommand insertcmd = con.CreateCommand();
insertcmd.CommandText = "INSERT INTO building (building_id, " +
    "name,city_id,street,sea_distance,shore_id,features,comment)" +
    "VALUES ('23','Villa','1','via Europa','100','2','5','etc')";

SqlCommand deletecmd = con.CreateCommand();
deletecmd.CommandText = "DELETE FROM building" +
    "WHERE building_id = '2'";

SqlCommand updatecmd = con.CreateCommand();
updatecmd.CommandText = "UPDATE building " +
    "SET sea_distance = '50' WHERE building_id = '12'";

int ni, nd, nu;
ni = nd = nu = 0;
try
{
    con.Open();

    ni = insertcmd.ExecuteNonQuery();
    nd = deletecmd.ExecuteNonQuery();
    nu = updatecmd.ExecuteNonQuery();
}
catch (SQLException ex)
{
    Console.WriteLine("Hiba: {0}", ex.Message);
}
finally
{
    con.Close();
    Console.WriteLine("beszúrt sorok: {0}", ni);
    Console.WriteLine("törölt sorok: {0}", nd);
    Console.WriteLine("módosított sorok: {0}", nu);
    Console.ReadLine();
}
...
```

3. Adattábla adatainak megjelenítése

Feladat: Írjuk ki konzol ablakba a „building” tábla összes sora *name*, *street* és *sea_distance* mezőinek értékét!

A megoldáshoz – a már megismert kapcsolat (`SqlCommand con`) és parancs (`SqlCommand cmd`) objektumok mellett – egy harmadik fajta objektum is kell. A `cmd` objektum most a `"SELECT name, street, sea_distance FROM building"` SQL parancsot tartalmazza, mellyel a `building` tábla összes sorát le tudjuk kérni az adatbázistól. Azt az SQL parancsot, amely sorokat kér le, az `ExecuteReader()` metódussal hajtjuk végre. Ez a metódus egy `DataReader` típusú, úgynevezett adatolvasó objektumot (legyen a neve: `reader`) ad vissza, amellyel hozzáférhetünk a lekérdezés eredményéhez. A `reader.Read()` metódussal ráállhatunk a lekérdezett sorok közül a soron következőre (kezdetben az elsőre). A `reader.Read()` `false` értéke jelzi, hogy már nincs következő sor. Ha van, akkor a `reader` egy sorra (az aktuálisra) hivatkozik, amelynek oszlopait például a `reader["oszlopnév"]` formában tudjuk elérni.

SqlDataReader

Tulajdonság	Típus	Magyarázat
<code>FieldCount</code>	<code>int</code>	Az adott sor oszlopainak száma
Metódus	Visszatérési típus	Magyarázat
<code>Read()</code>	<code>bool</code>	A következő sorra lép, ha nincs, akkor <code>false</code>
<code>NextResult()</code>	<code>bool</code>	Visszaadja, van-e következő sor.
<code>GetName(sorszám)</code>	<code>string</code>	Aktuális sor adott sorszámú oszlopának nevét adja meg
<code>GetOrdinal("oszlopnév")</code>	<code>int</code>	Aktuális sor adott nevű oszlopának sorszámát adja meg
<code>[sorszám]</code> <code>["oszlopnév"]</code>	<code>object</code>	Aktuális sor adott sorszámú vagy nevű oszlopának értékét adja meg
<code>GetValues(tömb)</code>	<code>int</code>	Aktuális sor oszlopainak értékét az adott objektumtömbbe másolja és visszaadja a tömb elemeinek számát
<code>IsDBNull(sorszám)</code>	<code>bool</code>	Vizsgálja, hogy az aktuális sor adott sorszámú oszlopa null értéket tartalmaz-e
<code>Get*(sorszám)</code> <code>GetValue(sorszám)</code> <code>GetBoolean(sorszám)</code> <code>GetInt16(sorszám)</code> ...	<code>object</code> <code>bool</code> <code>short</code>	Aktuális sor adott sorszámú oszlopának értékét a megfelelő típusra konvertálva adja meg

Az SQL szerver adattípusai (az oszloptípusok) nem azonosak a programozási nyelvek típusaival, de a C# nyelven minden oszloptípushoz megtalálható az annak megfelelő típus, és az arra konvertáló függvény (lásd `Get*` metódusokat). Ezek mellett a .NET definiálja az SQL szerveren érvényes úgynevezett `Sql*` típusokat (`SqlBoolean`, `SqlByte`, `SqlDouble`,

SqlInt16, SqlString stb.) is, amelyekre GetSql* (GetSqlBoolean, GetSqlByte, GetSqlDouble, GetSqlInt16, GetSqlString stb.) metódusokkal lehet konvertálni.

A reader["oszlopnév"] értékét is általában konvertálni kell, de az alábbi kódban konverzióra nem lesz szükség, hiszen a Console.Write() az object típusú értékeket úgyis sztringgé alakítja. A megoldó kódnak most csak a lényeges, a korábbi kódtól eltérő részletét mutatjuk meg.

```
SqlCommand cmd = con.CreateCommand();
cmd.CommandText = "SELECT * FROM building";

SqlDataReader reader = null;
try
{
    con.Open();
    reader = cmd.ExecuteReader();
    while (reader.Read())
    {
        Console.Write("name = ");
        Console.WriteLine(reader["name"]);
        Console.Write("street = ");
        Console.WriteLine(reader["street"]);
        Console.Write("sea_distance = ");
        Console.WriteLine(reader["sea_distance"]);
        Console.WriteLine();
    }
}
catch (SqlException ex)
{
    Console.WriteLine("Hiba: {0}", ex.Message);
    Console.ReadLine();
    return;
}
finally
{
    if (reader != null) reader.Close();
    if (con != null) con.Close();
    Console.ReadLine();
}
```

Feladat: *Hány épület (sor) van a „building” táblában, és mennyi a legközelebbi tengertől mért távolság?*

A számított érték lekérdezését – a sorok lekérdezéséhez hasonlóan – e az `ExecuteReader()` metódussal végezzük.

```
...
SqlCommand cmd = con.CreateCommand();
cmd.CommandText = "SELECT COUNT(*), MIN(sea_distance) " +
                  "FROM building;";

con.Open();

object[] results = new object[2];
SqlDataReader reader = cmd.ExecuteReader();
reader.Read();
reader.GetValues(results);

Console.WriteLine("Összes={0}", (int)results[0]);
Console.WriteLine("Minimális={0}", (int)results[1]);

reader.Close();
con.Close();
...
```

4. Több lépéses adatbázis műveletek

Az összetett SQL parancsokat általában a szerver oldalon megírt tárolt eljárások tartalmazzák. A „DeleteBuilding” nevű tárolt eljárás egy épületet töröl, de előtte kitörli az összes olyan apartmant az „apartment” táblából, amelyek a törölni kívánt épületben találhatóak:

```
CREATE PROCEDURE dbo.DeleteBuilding
(
    @id int
)
AS
BEGIN
    DELETE apartment WHERE building_id = @id;
    DELETE building WHERE building_id = @id;
    RETURN
END
```

Az alábbi kódrészlet a „DeleteBuilding” tárolt eljárását hívja meg. A cmd objektum most nem CommandType.Text formájú, hanem CommandType.StoredProcedure. Ez jelzi, hogy a CommandText egy tárolt eljárás nevét tartalmazza. A parancs végrehajtása előtt értéket kell átadnunk a tárolt eljárás paraméterének.

```
SqlCommand cmd = con.CreateCommand();
cmd.CommandType = CommandType.StoredProcedure;
cmd.CommandText = "DeleteBuilding";
cmd.Parameters.Add("@id", SqlDbType.Int).Value = 12;

cmd.ExecuteNonQuery();
```

Ha a kliens oldalon készítjük el a több lépésből álló adatbázis kezelésünket, akkor azt érdemes tranzakcióba fűzni. Az alábbi kódrészlet középpontjában egy Transaction objektum (neve: trans) áll. Az egyazon tranzakcióban végrehajtandó parancsokat ehhez kell hozzárendelni. A parancsok végrehajtása után lehetőségünk van véglegesíteni (trans.Commit(), vagy visszavonni (trans.Rollback()) azokat.

```
SqlTransaction trans = con.BeginTransaction(
    Data.IsolationLevel.Serializable);
SqlCommand cmd1 = con.CreateCommand();
cmd1.Transaction = trans;
...
int n1 = cmd1.ExecuteNonQuery();

SqlCommand cmd2 = con.CreateCommand();
cmd2.Transaction = trans;
...
int n2 = cmd2.ExecuteNonQuery();

if(...)
    trans.Commit();
else
    trans.Rollback();
```

5. Adattábla-karbantartás a memóriában

Feladat: Olvassuk be az „Apartments” adatbázis „building” nevű tábláját, változtassuk meg néhány adatát, majd mentjük vissza az adatbázisba!

Háromféle megoldást fogunk mutatni, amelyek nem az eredményükben, de még csak nem is kódjukban, hanem a kódjukat előállító technikákban térnek el. Mindegyik megoldás futás közben egy olyan adattábla (DataTable) objektumot hoz majd létre, amelyben (a memóriában) az adatbázis „building” tábláját a letöltés után el tudjuk helyezni. Az adattábla objektumot beágyazzuk egy másik, úgynevezett adathalmaz (DataSet) objektumba, amely képes több táblát és az azok közötti kapcsolatokat is tárolni.

Először egy olyan konzol-alkalmazást látunk, amelyben ez eddig bemutatott elemi eszközök (Connection, Command) segítségével olvassuk be a teljes adattáblát az adatbázisból egy adathalmazba (pontosabban annak adattábla objektumába), majd az így letöltött adatok megváltoztatása (törlés, beszúrás, módosítás) után visszamentjük a változtatásokat az adatbázisba. Ezt követően úgy módosítjuk ezt a megoldást, hogy a letöltés és visszamentés fázisok a megírt kód szintjén egyszerűsödjenek. Ehhez egy úgynevezett adapter objektumot fogunk használni, mert ez rendelkezik a letöltést és a visszamentést végző metódusokkal, amelyek meghívásával az első megoldásban alkalmazott ciklusokat helyettesíthetjük. Harmadik lépésben megnézzük, hogyan lehet a Visual Studio varázslójával olyan osztályokat generálni, amelyek példányosításával előállíthatjuk az adathalmazt és az adaptert.

Első megoldás (elemi eszközökkel)

Az első megoldás az alábbi részekből áll:

1. Egy adathalmazbeli adattábla objektum létrehozása a „building” tábla adatai számára.
2. Kapcsolati objektum és a „building” tábla adatainak letöltéséhez szükséges parancs objektum létrehozása.
3. A „building” tábla letöltése az adatbázisból az adattábla objektumba.
4. Az adattábla objektum adatainak változtatása.
5. Parancs objektumok definiálása a mentéshez.
6. Mentés az adattábla objektumból az adatbázis „building” táblájába.

Adathalmaz és adattábla objektumok létrehozása

A jelen feladatunk megoldásához tulajdonképpen elegendő lenne egy önálló adattábla objektum. A későbbi feladatok előkészítése, az általánosítás lehetősége miatt már most beleágyazzuk ezt egy adathalmaz objektumba.

Egy adathalmaz (DataSet) szélsőséges esetben akár a teljes adatbázist tárolhatja a memóriában, általában azonban az adatbázisnak csak egy részét, egy-két tábláját (sokszor azokat is megszűrve) tároljuk benne. Az adathalmaz lényegében két gyűjteményből áll. A Tables adattábla objektumokat (DataTable), a Relations pedig az adattáblák közötti (pl. idegen-kulcs) kapcsolatokat leíró objektumokat (DataRelation) tárolja.

Az adathalmaz objektum számos metódussal rendelkezik. Az alábbi táblázat csak néhányat említ ezek közül. Érdekes felfigyelni az adathalmaz azon jellemzőjére, hogy adatváltozás esetén „emlékszik” az adatok eredeti értékére is, amelyek – ha szükséges – visszaállíthatók

(`RejectChanges()`). További érdekesség, hogy az adathalmazbeli adatoknak meg kell felelniük azon megszorításoknak, feltételeknek, amelyeket egy adatbázisban is meg szoktunk adni, feltéve, hogy ezek ellenőrzését nem kapcsoljuk ki (`EnforceConstraints`). A hibás adatokra a `HasErrors` tulajdonság utal.

DataSet

<i>Tulajdonság</i>	<i>Típus</i>	<i>Magyarázat</i>
<code>Tables</code>	<code>DataTableCollections</code>	Adattáblák gyűjteménye
<code>Relations</code>	<code>DataRelationCollections</code>	Adattáblák közötti kapcsolatok gyűjteménye
<code>DataSetName</code>	<code>string</code>	Objektum neve
<code>HasErrors</code>	<code>bool</code>	Jelzi, hogy vannak hibákat tartalmazó sorok a táblákban
<code>EnforceConstraints</code>	<code>bool</code>	Beállítható illetve lekérdezhető, hogy frissítéskor figyelembe kell-e venni a megszorításokat.
<i>Metódus</i>	<i>Visszatérési típus</i>	<i>Magyarázat</i>
<code>AcceptChanges()</code>	<code>void</code>	Véglegessé teszi az utolsó feltöltés vagy az <code>AcceptChanges()</code> legutóbbi meghívása után keletkezett változásokat.
<code>RejectChanges()</code>	<code>void</code>	Visszavonja az utolsó feltöltés vagy az <code>AcceptChanges()</code> legutóbbi meghívása után keletkezett változásokat.
<code>GetChanges()</code>	<code>DataSet</code>	Lemásolja és visszaadja az utolsó feltöltés vagy az <code>AcceptChanges()</code> legutóbbi meghívása után keletkezett változásokat.
<code>Clear()</code>	<code>void</code>	Törli az összes tábla összes sorát

Az adattábla (`DataTable`) objektum tartalmaz minden olyan információt, amely egy adattábla esetében lényeges: az oszlopokat, illetve azok tulajdonságait leíró (`DataColumn`) objektumok gyűjteményét; a sorokat megadó (`DataRow`) objektumok gyűjteményét; az elsődleges kulcshoz tartozó oszlopokat; a táblában levő idegenkulcs kapcsolatokat és a kulcsokra (elsődleges és idegen) vonatkozó megszorításokat. További tulajdonság az adattábla neve, valamint hivatkozás a táblát tartalmazó adathalmazra.

Az alábbi táblázatban felsorolt néhány metódus közül most még egyet sem fogunk használni, hiszen az elsődleges kulcs megadásán kívül csak az oszlopok gyűjteményével dolgozunk: tehát oszlopokat kell definiálnunk. Később azonban szükség lesz ezekre a metódusokra, például új sor létrehozásakor.

DataTable

<i>Tulajdonság</i>	<i>Típus</i>	<i>Magyarázat</i>
Columns	DataColumnCollection	Oszlopok gyűjteménye
Rows	DataRowCollection	Sorok gyűjteménye
PrimaryKey	DataColumn[]	Elsődleges kulcs oszlopai
Constraints	ConstraintCollection	Megszorítások gyűjteménye
ChildRelations	DataRelationCollection	Kapcsolatok gyűjteménye
TableName	string	Táblanév
DataSet	DataSet	Hivatkozás a beágyazó DataSet objektumra
<i>Metódus</i>	<i>Visszatérési típus</i>	<i>Magyarázat</i>
Clear()	void	Törli az összes sorát
GetChanges()	DataTable	Lemásolja az utolsó feltöltés vagy az AcceptChanges() legutóbbi meghívása után keletkezett változásokat.
GetErrors()	DataRow[]	Visszaadja az összes hibás sorát
NewRow()	DataRow	Új sort hoz létre a DataTable tulajdonságaival
LoadDataRow	DataRow	Keres vagy frissít vagy létrehoz egy megadott sort
AcceptChanges()	void	Véglegessé teszi az utolsó feltöltés vagy az AcceptChanges() legutóbbi meghívása után keletkezett változásokat.
RejectChanges()	void	Visszavonja az utolsó feltöltés vagy az AcceptChanges() legutóbbi meghívása után keletkezett változásokat.
Select()	DataRow[]	Megadott feltételeknek eleget tevő sorainak tömbjét adja vissza

A következő kódrészletben létrehozunk egy adattáblát egy adathalmazban, majd definiáljuk a táblának az oszlopait. Ez utóbbit többféleképpen is megtehetjük. Mi egy olyan Add() metódust használunk, amelyik létrehoz egy oszlopot, ehhez az oszlop nevét és típusát kell megadni, és azt hozzáveszi a táblához. Ezután az oszlop nevével tudunk hivatkozni az adattábla objektum Columns gyűjteményének adott oszlopára. Ezt használjuk ki, amikor a „name” oszlopnál beállítjuk azt a megszorítást, hogy adatbázis-null értéket is tartalmazhat. Végül megadjuk az adattábla elsődleges kulcsát a hozzátartozó oszlopok felsorolásával.

Az adattábla objektum oszlopaira minden olyan megszorítást beállíthatnánk, amely az objektumot az adatbázis „building” táblájához hasonlóvá tenné. Most azonban szándékosan nem definiálunk minden megszorítást azért, hogy illusztrálhassuk azt az esetet, amikor az adattábla objektum önmagában ugyan helyes adatokat tárol, de ezek visszamentése az adatbázisba hibát okoz.

```

DataSet ds = new DataSet("buildings");
DataTable dt = new DataTable("building");
ds.Tables.Add(dt);

dt.Columns.Add("building_id", typeof(int));
dt.Columns.Add("name", typeof(string));
dt.Columns.Add("city_id", typeof(int));
dt.Columns.Add("street", typeof(string));
dt.Columns.Add("sea_distance", typeof(int));
dt.Columns.Add("shore_id", typeof(int));
dt.Columns.Add("features", typeof(int));
dt.Columns.Add("comment", typeof(string));

dt.Columns["name"].AllowDBNull = true;

dt.PrimaryKey = new DataColumn[] { dt.Columns["building_id"] };

```

Az adattábla objektum oszlopainak legfontosabb jellemzői:

DataColumn

<i>Tulajdonság</i>	<i>Típus</i>	<i>Magyarázat</i>
ColumnName	string	Oszlop neve
Ordinal	int	Oszlop sorszáma
DataType	Type	Oszlop típusa
DefaultValue	object	Alapértelmezett érték
AutoIncrement	bool	Beállítható, hogy az oszlop értéke automatikusan számítható-e
AutoIncrementSeed	long	Automatikus érték generálás esetén az induló érték
AutoIncrementStep	long	Automatikus érték generálás növekménye
Caption	string	Oszlop címsora
Unique	bool	Beállítható, hogy az oszlop értéke egyedi-e
AllowDBNull	bool	Beállítható, hogy az oszlop elfogad-e null értéket
ReadOnly	bool	Beállítható, hogy az oszlop csak olvasható-e
Table	DataTable	Hivatkozás a beágyazó DataTable objektumra

Kapcsolati objektum és a letöltés parancs objektumának létrehozása

```
ConnectionStringSettings settings =
    ConfigurationManager.ConnectionStrings["Apartments"];
if (settings == null) return;

SqlConnection con = new
    SqlConnection(settings.ConnectionString);

SqlCommand cmd = con.CreateCommand();
cmd.CommandText = "SELECT * FROM building";
```

A „building” tábla letöltése az adattábla objektumba

```
SqlDataReader reader = null;
try{
    con.Open();
    reader = cmd.ExecuteReader();
    while (reader.Read()){
        DataRow row = dt.NewRow();
        row["building_id"] = reader["building_id"];
        row["name"] = reader["name"];
        row["city_id"] = reader["city_id"];
        row["street"] = reader["street"];
        row["sea_distance"] = reader["sea_distance"];
        row["shore_id"] = reader["shore_id"];
        row["features"] = reader["features"];
        row["comment"] = reader["comment"];
        dt.Rows.Add(row);
    }
    dt.AcceptChanges();
} catch (SqlException ex){
    Console.WriteLine("Adatbázis hiba: {0}", ex.Message);
    Console.ReadLine();
    return;
}
finally
{
    if(reader != null) reader.Close();
    if(con != null) con.Close();
    Console.ReadLine();
}
```

Ez a fázis nagyon hasonlít arra, amikor egy tábla tartalmát kilistáztuk a konzolablakba. A tábla sorait az `ExecuteReader()` által visszaadott adatolvasó (`DataReader`) objektummal olvassuk ki, de ezeket most nem konzolra írjuk ki, hanem egy új sort hozunk létre az adattábla objektumban. Az új sor oszlopértékeire az oszlopok azon nevének segítségével hivatkozunk (`row["oszlopnév"]`), amelyeket az adattábla objektum oszlopainak létrehozásánál adtunk

meg. Ezeket az oszlop neveket azért ismeri a `row` objektumunk, mert az adattábla objektum `NewRow()` metódusával hoztuk létre. Az új sort végül hozzá adjuk az adattábla objektum `Rows` gyűjteményéhez. A feltöltés végén elhelyezett `AcceptChanges()` beállítja a tábla sorainak státuszát `Unchanged`-re, ami a sor törlése, módosítása, beszúrása esetén változik meg.

DataRow

Tulajdonság	Típus	Magyarázat
<code>RowState</code>	<code>DataRowState</code>	státusz:(régi) <code>Unchanged</code> ,(új) <code>Added</code> , (törölt) <code>Deleted</code> , (módosított) <code>Modified</code> , (még/már gyűjteményen kívül) <code>Detached</code>
<code>Table</code>	<code>DataTable</code>	Hivatkozás a beágyazó <code>DataTable</code> objektumra
[sorszám] ["oszlopnév"]	<code>object</code>	A sor adott sorszámú vagy nevű oszlopának értékét adja meg
Metódus	Visszatérési típus	Magyarázat
<code>Delete()</code>	<code>void</code>	Törli a sort
<code>AcceptChanges()</code> <code>RejectChanges()</code>	<code>Void</code> <code>void</code>	Elfogadja illetve visszautasítja az utolsó betöltés vagy az <code>AcceptChanges()</code> legutóbbi meghívása után keletkezett változtatásokat.
<code>BeginEdit()</code> <code>EndEdit()</code> <code>CancelEdit()</code>	<code>Void</code> <code>void</code> <code>void</code>	Elindítja, lezárja illetve visszavonja a sor szerkesztését.

Adattábla objektum változtatása

```
try{
    // Egy sor törlése
    DataRow drd = dt.Rows[0];
    drd.Delete();
    // Egy sor módosítása
    DataRow dru = dt.Rows[1];
    dru["sea_distance"] = 666;
    // Egy új sor beszúrása
    DataRow dri = dt.NewRow();
    dri["building_id"] = 4;
    dri["name"] = "name";
    dri["city_id"] = 1;
    dri["street"] = "street";
    dri["sea_distance"] = 4000;
    dri["shore_id"] = 1;
    dri["features"] = 0;
    dri["comment"] = "comment";
    dt.Rows.Add(dri);
} catch (DataException ex) {
    Console.WriteLine("Adatállomány hiba: {0}", ex.Message);
}
```

A megoldásnak ez a része tetszés szerint kicserélhető annak megfelelően, hogy mit szeretnénk ezzel az alkalmazással megmutatni. Itt most törlésre, módosításra és beszúrára mutatunk példát. Természetesen egyszerre több törlést, módosítást és beszúrást is elhelyezhetünk tetszőleges sorrendben.

Ez a kódrész akkor dob `DataException` kivételt, ha a változtatások sértik az adattábla objektumnál definiált megszorításokat. Ha minden adatbázisbeli megszorítást rávezettünk az adattábla objektumra, akkor, amennyiben itt nem keletkezik hiba, akkor az adatbázisba történő visszamentésnek is sikerülnie kell (legalább is, ami a megszorítások betartását illeti).

Parancs objektumok definiálása a mentéshez

Parancs objektumokkal már eddig is dolgoztunk. Az itt közölt kódrészletben legfeljebb annyi újdonság van, hogy paraméterezett SQL parancsokat adunk meg benne. Az SQL parancs `@` jellel kezdődő paraméterváltozókat tartalmaz, amelyeket `Parameter` objektumként kell létrehozni (a létrehozásnál mi csak a paraméter azonosítóját és típusát adjuk meg). Ezeket a paraméter objektumokat hozzáadjuk a paraméterezett SQL parancsot hordozó parancs objektumhoz. (Ugyanezt tettük korábban tárolt eljárást tartalmazó parancs objektum esetében a tárolt eljárás paraméterével.)

A beszúrási parancs objektuma:

```
SqlCommand insertcmd = con.CreateCommand();
insertcmd.CommandText = "INSERT INTO building " +
    "(building_id,name,city_id,street,sea_distance,shore_id, " +
    "features,comment) VALUES (@building_id,@name,@city_id, " +
    "@street,@sea_distance,@shore_id,@features,@comment) ";
insertcmd.Parameters.Add(new SqlParameter(
    "@building_id", SqlDbType.Int));
insertcmd.Parameters.Add(new SqlParameter(
    "@name", SqlDbType.VarChar));
insertcmd.Parameters.Add(new SqlParameter(
    "@city_id", SqlDbType.Int));
insertcmd.Parameters.Add(new SqlParameter(
    "@street", SqlDbType.VarChar));
insertcmd.Parameters.Add(new SqlParameter(
    "@sea_distance", SqlDbType.Int));
insertcmd.Parameters.Add(new SqlParameter(
    "@shore_id", SqlDbType.Int));
insertcmd.Parameters.Add(new SqlParameter(
    "@features", SqlDbType.Int));
insertcmd.Parameters.Add(new SqlParameter(
    "@comment", SqlDbType.VarChar));
```

A törlési parancs objektuma:

```
SqlCommand deletecmd = con.CreateCommand();
deletecmd.CommandText = "DELETE FROM building WHERE " +
    "building_id = @Original_building_id";
deletecmd.Parameters.Add(
    new SqlParameter("@Original_building_id", SqlDbType.Int));
```

A módosítás parancs objektuma:

```
SqlCommand updatecmd = con.CreateCommand();
updatecmd.CommandText = "UPDATE building " +
    "SET building_id = @building_id, name = @name, " +
    "city_id = @city_id, street = @street, " +
    "sea_distance = @sea_distance, shore_id = @shore_id, " +
    "features = @features, comment = @comment " +
    "WHERE building_id = @Original_building_id";
updatecmd.Parameters.Add(new SqlParameter("@building_id",
    SqlDbType.Int));
updatecmd.Parameters.Add(new SqlParameter("@name",
    SqlDbType.VarChar));
updatecmd.Parameters.Add(new SqlParameter("@city_id",
    SqlDbType.Int));
updatecmd.Parameters.Add(new SqlParameter("@street",
    SqlDbType.VarChar));
updatecmd.Parameters.Add(new SqlParameter("@sea_distance",
    SqlDbType.Int));
updatecmd.Parameters.Add(new SqlParameter("@shore_id",
    SqlDbType.Int));
updatecmd.Parameters.Add(new SqlParameter("@features",
    SqlDbType.Int));
updatecmd.Parameters.Add(new SqlParameter("@comment",
    SqlDbType.VarChar));
updatecmd.Parameters.Add(new
    SqlParameter("@Original_building_id", SqlDbType.Int));
```

Az adattábla objektum mentése a „building” táblába

A mentés fázisában végig nézzük az adattábla objektum azon sorait, amelyek státusza Added, Deleted vagy Modified, és ezeket az adatbázisbeli „building” táblában is megváltoztatjuk.

```
con.Open();
foreach(DataRow row in dt.Rows){
    try {
        switch (row.RowState){
            case DataRowState.Added:
                ...
            case DataRowState.Deleted:
                ...
            case DataRowState.Modified:
                ...
        }
    }catch (SqlException ex) {
        Console.WriteLine("Adatbázis hiba: {0}", ex.Message);
        Console.ReadLine();
    }
}
con.Close();
```

A fenti ciklusmagban található elágazás ágaiban használjuk az előzőleg definiált három paraméterezhető parancs objektumot. Az elvégzendő művelettől függően paraméterezzük fel az insertcmd, a deletcmd, az updatecmd objektumokat az aktuális sor adataival a megkívánt módon, majd hajtjuk végre az ExecuteNonQuery() metódust.

```
case DataRowState.Added:
```

```
insertcmd.Parameters["@building_id"].Value = row["building_id"];
insertcmd.Parameters["@name"].Value = row["name"];
insertcmd.Parameters["@city_id"].Value = row["city_id"];
insertcmd.Parameters["@street"].Value = row["street"];
insertcmd.Parameters["@sea_distance"].Value=row["sea_distance"];
insertcmd.Parameters["@shore_id"].Value = row["shore_id"];
insertcmd.Parameters["@features"].Value = row["features"];
insertcmd.Parameters["@comment"].Value = row["comment"];
int i = insertcmd.ExecuteNonQuery();
if(i==0) Console.WriteLine("Sikertelen beszúrás");
break;
```

```
case DataRowState.Deleted:
```

```
deletcmd.Parameters["@Original_building_id"].Value =
    row["building_id", DataRowVersion.Original];
int i = deletcmd.ExecuteNonQuery();
if(i==0) Console.WriteLine("Sikertelen törlés");
break;
```

```
case DataRowState.Modified:
```

```
updatecmd.Parameters["@Original_building_id"].Value =
    row["building_id", DataRowVersion.Original];
updatecmd.Parameters["@building_id"].Value = row["building_id"];
updatecmd.Parameters["@name"].Value = row["name"];
updatecmd.Parameters["@city_id"].Value = row["city_id"];
updatecmd.Parameters["@street"].Value = row["street"];
updatecmd.Parameters["@sea_distance"].Value=row["sea_distance"];
updatecmd.Parameters["@shore_id"].Value = row["shore_id"];
updatecmd.Parameters["@features"].Value = row["features"];
updatecmd.Parameters["@comment"].Value = row["comment"];
int i = updatecmd.ExecuteNonQuery();
if(i==0) Console.WriteLine("Sikertelen javítás");
break;
```

Második megoldás (adapterrel)

Az előző megoldásában elemi eszközökkel (Command) és saját magunk által tervezett kódrésszel (ciklusokkal) végeztük el az adatbázis betöltését és visszamentését. A feladat egyszerűbben is megoldható egy megfelelő tulajdonságokkal rendelkező úgynevezett adapter objektum bevezetésével (DataAdapter).

A második megoldás az alábbi részekből áll:

1. Egy adathalmazbeli adattábla objektum létrehozása. (nem változott)
2. Kapcsolati objektum és egy adapter objektum definiálása.
3. A „building” tábla letöltése az adatbázisból az adattábla objektumba.
4. Az adattábla objektum változtatása (nem változott)
5. Mentés az adattábla objektumból az adatbázis „building” táblájába.

Connection és DataAdapter objektumok létrehozása

Természetesen először most is a kapcsolati objektumot (con) definiáljuk. Ezután létrehozzuk az adapter objektumot. Ez az objektum lényegében négy Command objektumot fog össze: a letöltésért felelős SelectCommand mellett a visszamentést meghatározó InsertCommand, DeleteCommand, UpdateCommand objektumokat. Az adapter Fill() metódusa olvassa be a sorokat az adatbázisból egy adattábla objektumba vagy adathalmazba (ez a Fill paraméterezésén múlik.) A visszamentést az Update() metódus végzi.

SqlDataAdapter

<i>Tulajdonság</i>	<i>Típus</i>	<i>Magyarázat</i>
SelectCommand	SqlCommand	Letöltést meghatározó objektum
InsertCommand	SqlCommand	Új sor(oka)t adatbázisba író objektum
DeleteCommand	SqlCommand	Sor(oka)t adatbázisból törölő objektum
UpdateCommand	SqlCommand	Sor(oka)t adatbázisban módosító objektum
ContinueUpdateError	bool	Beállítható, hogy folytatódhat-e kivétel dobás nélkül az adatbázis frissítése hiba esetén
<i>Metódus</i>	<i>Visszatérési típus</i>	<i>Magyarázat</i>
Fill()	int	Letöltés
Update()	int	Mentés
FillSchema()	DataTable DataTable[]	Az adatbázis szerkezetét, megszorításait tölti le

Az adapter objektum létrehozása még mindig meglehetősen nagy munka, de használatával a letöltés és a visszamentés már gyerekjáték.

```
SqlDataAdapter adapter = new SqlDataAdapter(
    "SELECT * FROM building", con);

adapter.InsertCommand = new SqlCommand("INSERT INTO building" +
    "(building_id,name,city_id,street,sea_distance, shore_id," +
    "features,comment) VALUES (@building_id,@name, @city_id," +
    "@street,@sea_distance,@shore_id,@features,@comment)", con);
adapter.InsertCommand.Parameters.Add(new SqlParameter(
    "@building_id", SqlDbType.Int, 0, "building_id"));
adapter.InsertCommand.Parameters.Add(new SqlParameter(
    "@name", SqlDbType.VarChar, 0, "name"));
adapter.InsertCommand.Parameters.Add(new SqlParameter(
    "@city_id", SqlDbType.Int, 0, "city_id"));
adapter.InsertCommand.Parameters.Add(new SqlParameter(
    "@street", SqlDbType.VarChar, 0, "street"));
adapter.InsertCommand.Parameters.Add(new SqlParameter(
    "@sea_distance", SqlDbType.Int, 0, "sea_distance"));
adapter.InsertCommand.Parameters.Add(new SqlParameter(
    "@shore_id", SqlDbType.Int, 0, "shore_id"));
adapter.InsertCommand.Parameters.Add(new SqlParameter(
    "@features", SqlDbType.Int, 0, "features"));
adapter.InsertCommand.Parameters.Add(new SqlParameter(
    "@comment", SqlDbType.VarChar, 0, "comment"));

adapter.DeleteCommand = new SqlCommand("DELETE FROM building" +
    "WHERE building_id = @Original_building_id", con);
adapter.DeleteCommand.Parameters.Add(new SqlParameter(
    "@Original_building_id", SqlDbType.Int, 0, "building_id"));

adapter.UpdateCommand = new SqlCommand("UPDATE building SET " +
    "building_id=@building_id, name=@name, city_id=@city_id, " +
    "street=@street, sea_distance=@sea_distance, shore_id = " +
    "@shore_id, features=@features, comment=@comment " +
    "WHERE building_id = @Original_building_id", con);
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@building_id", SqlDbType.Int, 0, "building_id"));
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@name", SqlDbType.VarChar, 0, "name"));
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@city_id", SqlDbType.Int, 0, "city_id"));
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@street", SqlDbType.VarChar, 0, "street"));
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@sea_distance", SqlDbType.Int, 0, "sea_distance"));
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@shore_id", SqlDbType.Int, 0, "shore_id"));
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@features", SqlDbType.Int, 0, "features"));
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@comment", SqlDbType.VarChar, 0, "comment"));
adapter.UpdateCommand.Parameters.Add(new SqlParameter(
    "@Original_building_id", SqlDbType.Int, 0, "building_id"));
```

A „building” tábla letöltése a DataTable objektumba

A `Fill()` metódus rendre végig megy a `SelectCommand`-ban megadott SQL parancs (vagy tárolt eljárás) által letöltött összes soron, és azokat elhelyezi a megfelelően létrehozott adathalmazban. (Nem kezeltük le a `Fill` metódus által visszaadott értéket, amely a betöltött sorok számát mutatja.)

```
try
{
    adapter.Fill(ds, "building");
}
catch (SqlException ex)
{
    Console.WriteLine("Adatbázis hiba: {0}", ex.Message);
    Console.ReadLine();
    return;
}
```

A DataTable objektum mentése a „building” táblába

Az `Update()` metódus a visszamentést végzi. A metódus végig nézi az adattábla vagy adathalmaz azon sorait, amelyek megváltoztak, és ezeket a változásokat (annak megfelelően, hogy újak, töröltettek vagy módosultak) az `InsertCommand`, `DeleteCommand`, `UpdateCommand` valamelyikével visszamenti az adatbázisba. (Nem kezeltük le az `Update` metódus által visszaadott értéket, amely a visszamentett sorok számát mutatja.)

```
try
{
    adapter.Update(ds, "building");
}
catch (SqlException ex)
{
    Console.WriteLine("Adatbázis hiba: {0}", ex.Message);
    Console.ReadLine();
    return;
}
```

Harmadik megoldás (varázslóval)

Most az adathalmaz (azon belül az adattábla) illetve az adapter objektum definiálását egyszerűsítjük oly módon, hogy segítségül hívjuk a Visual Studio varázslóját, amivel úgynevezett típusos adathalmaz és típusos adapter hozható létre.

Adatforrás (DataSource) készítése varázslóval:

1. Válasszuk ki a menübár Data menüjéből vagy a DataSource View ablakból az „Add New DataSource...” menüt! Ekkor megjelenik a Data Source Configuration Wizard ablak, és elindul az új adatforrás generálását végző varázsló.
2. Jelöljük meg az adat származási helyének a Database-t!
3. Válasszuk ki vagy adjuk meg (New Connection ...) az elérni kívánt adatbázisunk (ez most az Apartments) kapcsolati sztringjét!
3. Kívánságunkra a kapcsolati sztringből App.config néven generálódik egy már kitöltött „Application Configuration File” (lásd 1. fejezet).
4. Jelöljük be az adatbázis letölteni kívánt részét! (Ez a teljes building tábla.) Itt adhatjuk meg a létrehozandó úgynevezett típusos DataSet osztályának nevét is. (Legyen ez BuildingDataSet.)

A fenti lépések hatására létrejön egy BuildingDataSet.xsd állomány, amely többek között az alábbi származtatott osztályokat tartalmazza:

- BuildingDataSet : DataSet
- buildingDataTable : DataTable
- buildingDataRow : DataRow
- buildingDataAdapter (tartalmaz egy SqlDataAdapter objektumot)

Ezek az osztályok a projektünk részét alkotják. Megfelelően hivatkozva ezekre az osztályokra létrehozhatjuk azok példányaiként a megoldáshoz szükséges objektumokat. (Vegyük észre, hogy ezek az osztályok más névtérben jöttek létre.) Érdemes belenézni a generált kódba (BuildingDataSet.Designer.cs). Keressük meg, hol definiálja a BuildingDataSet a tablebuilding adattagját, amire building névvel hivatkozhatunk majd; milyen oszlopai vannak a buildingDataTable osztálynak, hogy lehet rájuk hivatkozni, milyen tulajdonságaik vannak; hol van a buildingDataAdapter osztályban a számunkra fontos SqlDataAdapter objektum SelectCommand, InsertCommand, DeleteCommand és UpdateCommand beállítása, hogyan hívható meg a Fill() valamint Update() metódusa, és hol van elrejtve a megfelelő SqlConnection objektum kezelése.

A BuildingDataSet példányaként egy úgynevezett típusos adathalmazt hozunk létre. Ez abban különbözik a közönséges adathalmaz objektumtól, hogy olyan tulajdonságai is vannak, amelyek csak az Apartments adatbázis esetében értelmezhető. Látni fogjuk, hogyan lehet a típusos adathalmaz building publikus adattagjával arra az adattáblára hivatkozni, amelybe a „building” tábla adatait tölthetjük be, vagy hogyan lehet ennek a táblának egy sorát tartalmazó objektumnál publikus adattagként használni az oszlopneveket.

A harmadik megoldás fő modulja az alábbi részekből áll:

1. A BuildingDataSet példányosítása és building táblájának átnevezése (dt)
2. A buildingDataAdapter példányosítása
3. A „building” tábla letöltése az adatbázisból az adattábla objektumba (*nem változott*)
4. Az adattábla objektum változtatása (*nem változott*)
5. Mentés az adattábla objektumból az adatbázis „building” táblájába. (*nem változott*)

```
BuildingDataSet ds = new BuildingDataSet();
BuildingDataSet.buildingDataTable dt = ds.building;
BuildingDataSetTableAdapters.buildingTableAdapter adapter =
    new BuildingDataSetTableAdapters.buildingTableAdapter();
try
{
    adapter.Fill(ds.building);
}
catch (SqlException ex)
{
    Console.WriteLine("Adatbázis hiba: {0}", ex.Message);
    Console.ReadLine();
    return;
}
```

Kihasználva a típusos adathalmaz előnyeit tovább egyszerűsíthetjük a building tábla karbantartását végző kódrészt is. Figyeljük meg, hogyan lehet hivatkozhatunk egy típusos adatsor (BuildingDataSet.buildingRow) objektum mezőire (oszlopaira). (A dt változó helyett ds.building-et írunk.).

```
try
{
    // Egy sor törlése
    BuildingDataSet.buildingRow drd =
        (BuildingDataSet.buildingRow)ds.building.Rows[0];
    drd.Delete();
    // Egy sor módosítása
    BuildingDataSet.buildingRow dru =
        (BuildingDataSet.buildingRow)ds.building.Rows[1];
    dru.sea_distance = 666;
    // Egy sor beszúrása
    BuildingDataSet.buildingRow dri =
        (BuildingDataSet.buildingRow)ds.building.NewRow();
    dri.building_id = 4;
    dri.name = "name";
    dri.city_id = 1;
    dri.street = "street";
    dri.sea_distance = 4000;
    dri.shore_id = 1;
    dri.features = 0;
    dri.comment = "comment";
    ds.building.Rows.Add(dri);
} catch (DataException ex) {
    Console.WriteLine("Adatállomány hiba: {0}", ex.Message);
}
```

Adattábla karbantartás

ADATTÁBLA KARBANTARTÁS	26
1. DURVA VÁLTOZAT	27
<i>Tervező nélkül</i>	27
<i>Objektumok Tervezővel, tulajdonságaik kóddal</i>	30
<i>Mindent a Tervezővel</i>	31
<i>Egyetlen húzással</i>	32
2. ADATTÁBLA OSZLOPAINAK TESTRE SZABOTT MEGJELENÍTÉSE.....	34
<i>Azonosító elrejtése és értékének automatikus generálása</i>	34
<i>Idegenkulcs értékének megjelenítése, kombináltdobozos kiválasztása</i>	35
<i>Egyedi cellaformák</i>	36
3. HIBAELLENŐRZÉS	45
<i>Mezőellenőrzés</i>	45
<i>Sorellenőrzés</i>	47
<i>Táblaellenőrzés</i>	47
<i>Adatbázis ellenőrzés</i>	47

Az alábbiakban olyan grafikus felületű alkalmazásokat mutatunk, amelyek egy adattábla karbantartásának problémáját járják körül. Megismerjük, hogyan lehet felhasználói felületet alkotó vezérlőkkel egy adathalmaz objektum adatait megjeleníteni, módosítani.¹ Nem törekszünk ezen osztályok minden részletre kiterjedő ismertetésével – ezt mind nyomtatott, mind online szakirodalmakban megtalálhatjuk –, helyette az alkalmazásaikra helyezzük a hangsúlyt. A feladatokat egy képzelt utazási iroda apartman-foglalási tevékenységéhez használt „Apartments” adatbázisára fogalmazzuk meg. Ebben a fejezetben ennek az adatbázisnak az épületek leírására szolgáló „building” nevű táblájának karbantartásáról lesz szó, de mivel ez idegen-kulcs kapcsolatban áll két másik táblával, így azokkal is számolnunk kell.

```

CREATE TABLE building (
    building_id INTEGER PRIMARY KEY,
    name VARCHAR(30),
    city_id INTEGER NOT NULL,
    street VARCHAR(30) NOT NULL,
    sea_distance INTEGER,
    shore_id INTEGER,
    features INTEGER,
    comment VARCHAR(100),
    FOREIGN KEY (city_id) REFERENCES city,
    FOREIGN KEY (shore_id) REFERENCES shore
);
CREATE TABLE city (
    city_id INTEGER PRIMARY KEY,
    name VARCHAR(30),
);
CREATE TABLE shore (
    shore_id INTEGER PRIMARY KEY,
    type VARCHAR(10),
);

```

¹ Az osztályok bemutatásánál John Sharp Visual C# 2005 Lépésről lépésre c. könyvére (2005, SZAK Kiadó.) támaszkodtam.

Feladat: *Készítsünk olyan grafikus felületű alkalmazást, amely lehetőséget ad a „building” tábla karbantartására!*

Fokozatosan módosítjuk megoldó programot egyfelől abból a célból, hogy annak újbóli előállítására minél kevesebb terhet rójon a programozóra, másfelől azért, hogy a felhasználói barátságosság szempontjainak minél jobban megfeleljen (ne kelljen a felhasználónak egyetlen azonosítókódot sem ismerni/megadni, új sor beszúrásánál töltsük ki a mezőket alapértelmezett értékekkel, csak értelmes, hibát nem okozó adatbevitelt engedélyezzünk).

Először az alkalmazás puritán, durva verzióját készítjük el. Ez még semmilyen hibakezelést nem végez majd, az adattábla sorai egységesen TextBox-szerűen módosíthatók, a felületen minden adat abban a formában jelenik meg, ahogy azt a tábla tartalmazza, azaz a kényelmes használatot zavaró belső azonosítókat és kódokat is fogunk látni. Ennél fogva ez az alkalmazás teljesen általános, bármilyen adattábla karbantartása esetében elkészíthető. A hangsúlyt arra helyezzük, hogy bemutassunk néhány különböző technikát az alkalmazás létrehozására. Második lépésben hozzáigazítjuk az alkalmazást a konkrét feladathoz: egyedi megjelenéssel, majd a harmadik lépésben a hiba ellenőrzéssel.

1. Durva változat

A következőkben azt mutatjuk meg, hogyan lehet egy úgynevezett DataGridView vezérlő segítségével egy DataTable adatait megtekinteni és módosítani. Ugyanannak a megoldásnak több előállítási módját is megmutatjuk: először egyáltalán nem használjuk a Visual Studio Designer-t, más néven Tervezőt, utána bizonyos lépéseket már annak segítségével végzünk el, végül az egész alkalmazást a Tervezővel készítjük.

Minden esetben azzal kezdjük a munkát, hogy egy Windows Application projektet indítunk, amihez hozzáadunk egy BuildingDataSet nevű adatforrást (Data Source, lásd előző fejezet) az Apartments adatbázis „building” táblájáról. Ennek következtében lesz egy BuildingDataSet típusos DataSet osztályunk, és a BuildingDataSetTableAdapters alnévtérben egy buildingTableAdapter típusos adapterosztályunk.

Tervező nélkül

A feladat megoldásához négy objektumra van szükségünk. Egy adathalmaz (DataSet) objektumra, amelyben a „building” tábla adatait tároljuk; egy adapter objektumra, amellyel – a program lelegején az adatbázisból letöltjük az adatokat és – egy nyomógomb (Button) hatására – visszamentjük azokat; egy adatrács (DataGridView) vezérlőre, amelyen keresztül a felhasználó az adathalmazbeli adatokat nézegetheti, módosíthatja. Az adathalmaz és adapter objektumokat a korábban generált típusos DataSet és típusos DataAdapter osztályokból példányosítjuk, a másik két objektumot pedig a megfelelő System.Windows.Forms-beli osztályokból.

```
public partial class Form1 : Form
{
    private BuildingDataSet buildingDataSet;
    private BuildingDataSetTableAdapters.buildingTableAdapter
        buildingTableAdapter;
    private DataGridView buildingDataGridView;
    private Button buttonSave;
```

Az űrlap-osztály konstruktorában az `InitializeComponent()` metódushívás után először beállítjuk az űrlap méretét, majd a `buildingDataGridView` vezérlőt hozzuk létre megadva néhány, az elhelyezkedésre vonatkozó tulajdonságát. Ugyanezt tesszük a `buttonSave` nyomógombbal is, és ennek `Click` eseményéhez delegálunk egy eseménykezelő függvényt is.

```
using System.Drawing;
...
public Form1()
{
    InitializeComponent();

    ClientSize = new Size(540, 350);

    buildingDataGridView = new DataGridView();
    buildingDataGridView.Location = new Point(20, 40);
    buildingDataGridView.Size = new Size(500, 250);
    Controls.Add(buildingDataGridView);

    buttonSave = new Button();
    buttonSave.Location = new Point(230, 300);
    buttonSave.Size = new Size(80, 25);
    buttonSave.Text = "Save";
    buttonSave.Click += new
        System.EventHandler(buttonSave_Click);
    Controls.Add(buttonSave);
}
```

Ezt követően létrehozuk a `buildingDataSet` és `buildingTableAdapter` objektumokat. Most következik talán a kód legérdekesebb, leglényegesebb része. A `buildingDataGridView` objektum `DataSource` tulajdonságát ráállítjuk a `buildingDataSet`-re, `DataMember` tulajdonságát a „building” táblára (annak nevére). Ennek hatására kapcsolódik össze az adatrács az adathalmazban tárolt `building` adattáblával. Ezután az adatrácsban bekövetkezett változás az adathalmazban is rögtön látszani fog. Az adatrácshoz definiálni kell azokat az oszlopokat is, amelyeken keresztül megmutatja az adattábla oszlopait. Ezt a fáradságos és unalmas kódolást azonban megúszhatjuk a `buildingDataSet` `SchemaSerializationMode` tulajdonságának beállításával. Ennek hatására a `building` tábla oszlopai automatikusan átmásolódnak az adatrács oszlopaivá.

```
buildingDataSet = new BuildingDataSet();
buildingTableAdapter = new
    BuildingDataSetTableAdapters.buildingTableAdapter();

buildingDataSet.SchemaSerializationMode =
    System.Data.SchemaSerializationMode.IncludeSchema;

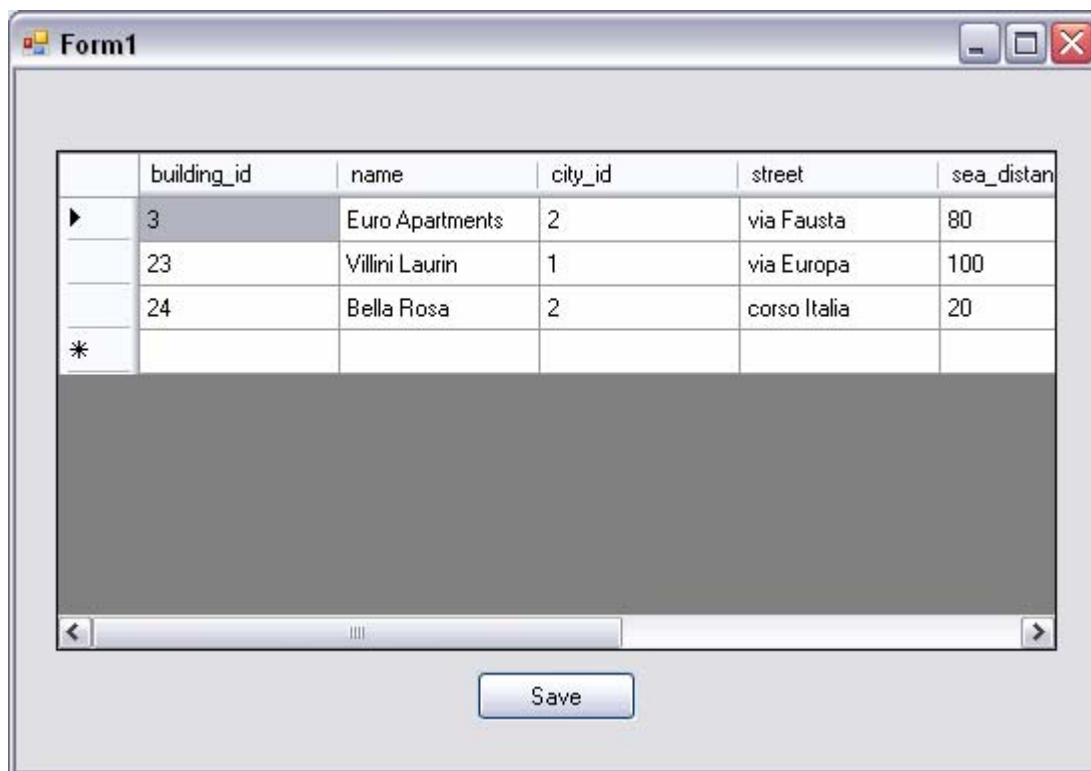
buildingDataGridView.DataSource = buildingDataSet;
buildingDataGridView.DataMember =
    buildingDataSet.building.TableName;

buildingTableAdapter.Fill(buildingDataSet.building);
}
```

Legvégül a konstruktorban betöltjük az adatbázis „building” tábláját a `buildingDataSet`-be a `buildingTableAdapater Fill()` metódussal. Ennek fordítottja a mentés, amely a `buttonSave` nyomógomb lenyomására következik be. Ehhez a `buildingTableAdapater Update()` metódusát a nyomógomb `Click` eseménykezelőjében helyezzük el.

```
private void buttonSave_Click(object sender, EventArgs e)
{
    buildingTableAdapter.Update(buildingDataSet);
}
```

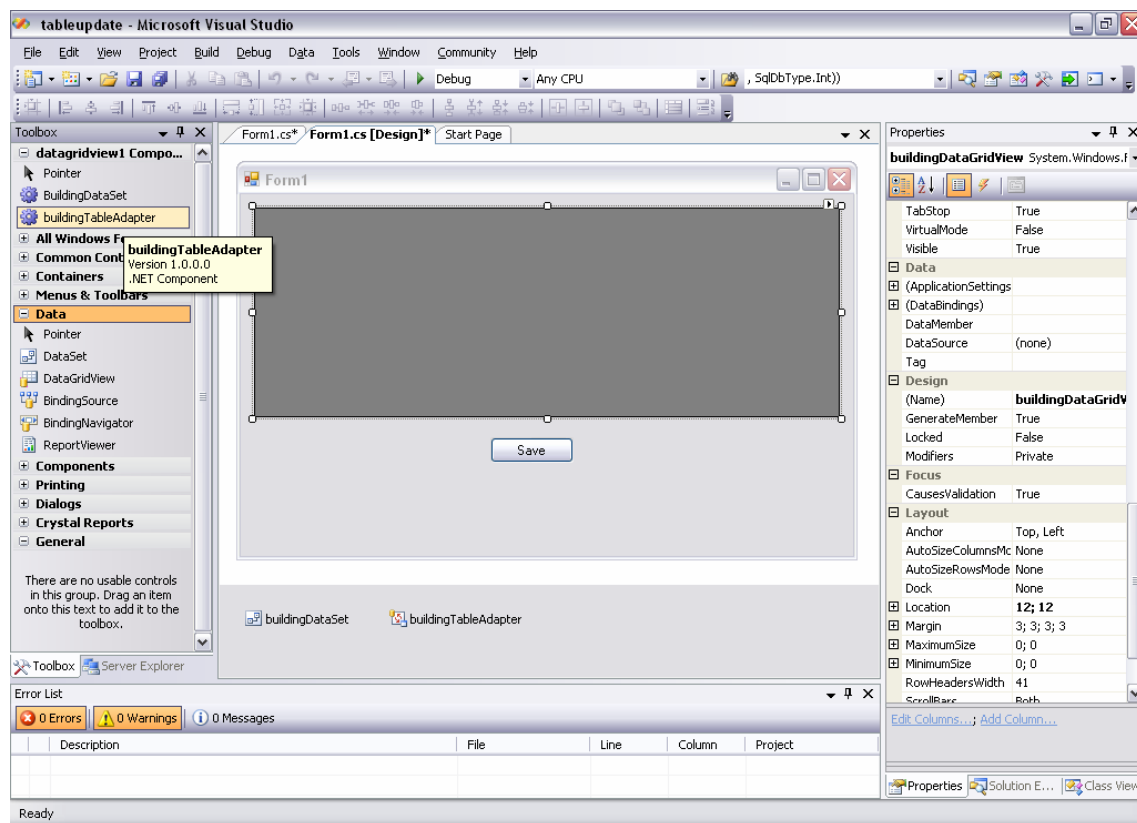
Az alkalmazást ezután fordítjuk és futtatjuk. Az alkalmazás működik, de még korántsem felel meg a feladatban megfogalmazott elvárásoknak (hibás adatbevitel esetén kezeletlen kivételt dob, belső azonosítókat mutat).



	building_id	name	city_id	street	sea_distan
▶	3	Euro Apartments	2	via Fausta	80
	23	Villini Laurin	1	via Europa	100
	24	Bella Rosa	2	corso Italia	20
*					

Save

Objektumok Tervezővel, tulajdonságaik kóddal



A VS Tervezőjének használatával lényegesen lerövidíthető a kódolás. Már azáltal, hogy az űrlap komponenseit a Tervezővel helyezzük el, sok időt spórolunk. (Az adatforrás létrehozása és egy fordítási lépés elvégzése után a BuildingDataSet és a buildingTableAdapter választható komponensként felkerül a ToolBox-ban, úgy mint a DataGridView vagy a Button.) Húzzuk rá a tervező felületre „drag&drop” technikával mind a négy komponenst. A kódban automatikusan létrejönnek a komponensek objektumai. Ekkor a saját kódrész konstruktorában mindössze az alábbiakat kell írunk. (A buttonSave_Click() eseménykezelő ugyanaz, mint korábban.)

```
public Form1 ()
{
    InitializeComponent ();

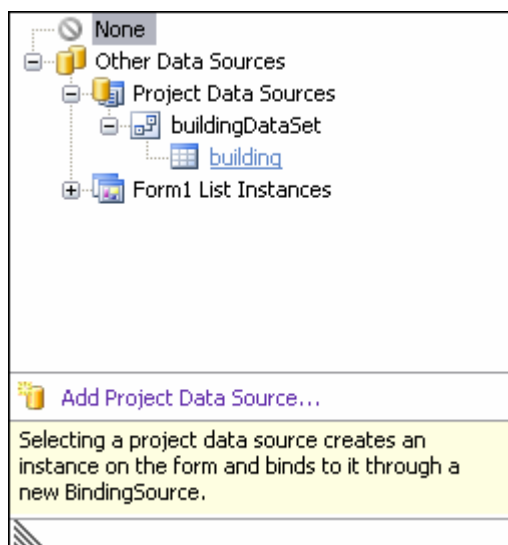
    buildingDataGridView.DataSource = buildingDataSet;
    buildingDataGridView.DataMember =
        buildingDataSet.building.TableName;
    buttonSave.Click += new
        System.EventHandler(buttonSave_Click);

    buildingTableAdapter.Fill (buildingDataSet.building);
}
```

Mindent a Tervezővel

Az Olvasó jól tudja, hogy egy nyomógomb Click eseményhez úgy is delegálhat kezelőfüggvényt, ha duplát kattint a nyomógombon vagy a Properties ablak eseménypaneljének Click sorában. Nyilván azon sem lepődik meg, hogy a `buildingDataGridView` `DataSource` tulajdonságát is be lehet állítani a Properties ablakban (ami a Tervező nézetben kijelölt adatrács jobb felső sarkán található úgynevezett SmartTag bekapcsolásával felnyíló ablakban is megtalálható).

A `DataSource` tulajdonság beállításakor egy újabb ablak nyílik fel.



Ebben kétféleképpen is beállíthatjuk az adatforrást: első esetben a `buildingDataSet`-et, második esetben annak `building` tábláját megjelölve.

Ennek a megoldásnak előnye, hogy már tervezési időben láthatóak az adatrácsban az adattábla oszlopai, így azok tulajdonságai is alakíthatóak a Tervezővel. Az adatkapcsoló objektum jelentősége azonban csak akkor érthető meg, ha az adatrácson kívül más vezérlőkben is látni akarjuk a „building” tábla adatait. Vegyünk fel például az űrlapra egy szövegdobozt (`TextBox`) és egy listadobozt (`ListBox`). Állítsuk be a szövegdoboz `DataBindings.Text` tulajdonságát „name”-re. (Kézzel írt kód esetén: `textBox.DataBindings.Add(new Binding("Text", buildingBindingSource, "name", true));`). Rendeljük a listadoboz `DataSource` tulajdonságához a `buildingBindingSource` -ot, `DisplayMember` tulajdonságához pedig a „street”-et. Futtassuk az alkalmazást. Azt látjuk, hogy egyszerre és szinkronban változnak az adatok az adatrácsban, a szövegdobozban és a listadobozban.

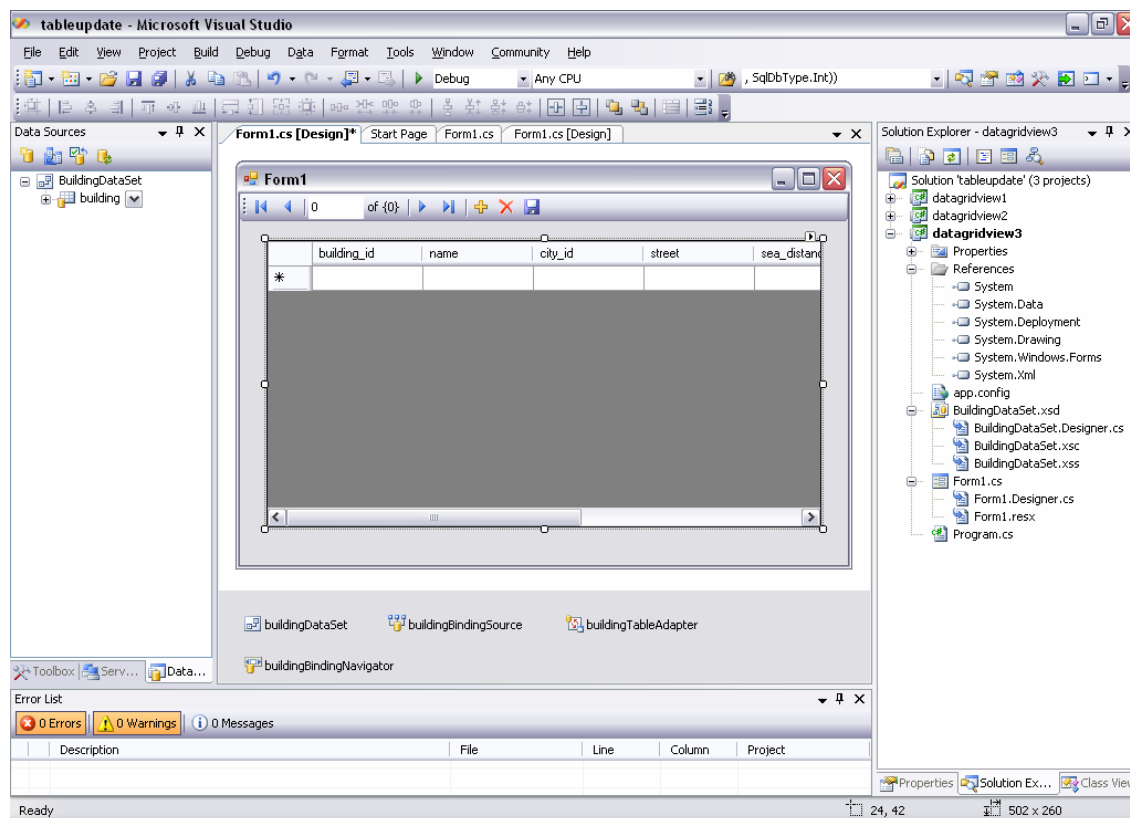
Visszatérve az eredeti alkalmazásunkhoz, – annak köszönhetően, hogy ott minden beállítást a Tervezővel csináltunk – mindössze két sort kell a programba „kézzel” beleírni. Az adatbázis „building” tábláját betöltő `Fill()` metódushívást (a konstruktorba az `InitializeComponent()` hívása után) és a visszamentést végző `Update()` metódushívást (a nyomógomb eseménykezelőjébe).

Mindkét esetben egy érdekes dolog történik: létrejön egy új objektum. Ez egy úgynevezett adatkapcsoló objektum, amely beékelődik a `buildingDataGridView` és első esetben a `buildingDataSet`, a második esetben annak `building` táblája közé. Ennek generált neve az első esetben a `buildingDataSetBindingSource`, a másodikban `buildingBindingSource` lesz. Ha megnézzük a tulajdonságait, akkor láthatjuk, hogy most az ő `DataSource` tulajdonsága mutat a `buildingDataSet`-re, az adatrács `DataSource` tulajdonsága pedig erre az új adatkapcsoló objektumra.

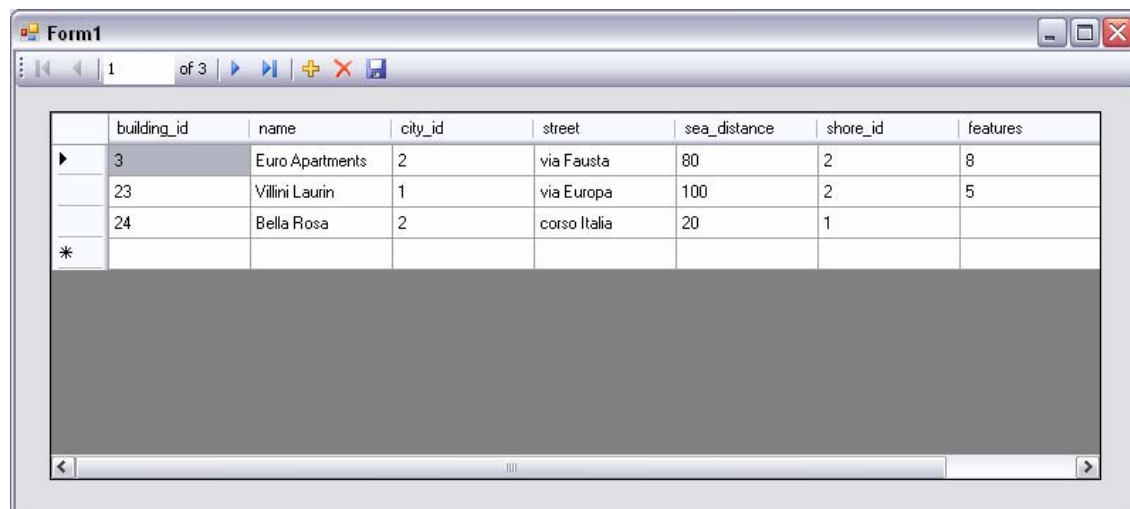
(Az első esetben még a `DataMember` tulajdonságot is be kell majd állítani a `building`-re vagy a rácsnál, vagy az adatkapcsoló objektumnál.)

Egyetlen húzással

Az alkalmazásunk egyetlen „drag&drop” mozdulattal is elkészíthető. Húzzuk rá a Data Source nézet building tábláját az űrlapra. Ennek hatására minden felkerül rá, sőt egy úgynevezett navigáló sáv is megjelenik az űrlapon a mentés nyomógombbal együtt.



A kódba semmit nem kell beleírunk, ugyanis az már a `Fill()` és `Update()` metódushívásokat is tartalmazza. Ebben az esetben a `Fill()` hívása nem a konstruktorban, hanem az űrlap mindenkor betöltésekor bekövetkező `Load` eseménykezelőjében található, az `Update()` hívását pedig megelőzi az aktuális szerkesztési folyamat validálása és lezárása.



Hangsúlyozzuk, hogy az alkalmazásunk működése még mindig ugyanaz, mint amit az első verziónál láttunk. Ez azt is jelenti, hogy továbbra sem kezeli a hibás adatbevitel. A felület még most is olyan belső azonosítókat mutat, amivel a felhasználó nem igen tud mit kezdeni, legfeljebb hibásan megadni. A hibaellenőrzés hiánya miatt az alkalmazás használata közben többféle kezeletlen kivétel keletkezhet.

Mielőtt az alkalmazásunkat továbbfejlesztenénk meg kell jegyeznünk, hogy a bemutatott „drag&drop” technikával a kiválasztott adattáblát nemcsak adatrácscsal jeleníthetjük meg. Az adattábla egy sorának mezőit önálló vezérlőkben is kezelhetjük (ezek formáját a Data Source nézetben lehet beállítani). Ezek a vezérlők az adatrácscsal egy időben is fenn lehetnek az űrlapon, mert szinkronizált működésüket a generált adatkapcsoló objektum felügyeli.

building id:	23	sea distance:	100
name:	Villini Laurin	shore id:	2
city id:	1	features:	5
street:	via Europa	comment:	etc

	building_id	name	city_id	street	sea
	3	Euro Apartments	2	via Fausta	80
▶	23	Villini Laurin	1	via Europa	100
	24	Bella Rosa	2	corso Italia	20
*					

Ez egy közkedvelt változata a táblakarbantartásnak, különösen, ha az adatrácscot csak nézegetésre, egy sor kiválasztására használjuk (azaz az adatrácsc ReadOnly tulajdonságú), az aktuális illetve az üres sor szerkesztése pedig a mezőnkénti vezérlőkben történik.

2. Adattábla oszlopainak testre szabott megjelenítése

Induljunk ki az előzőekben már elkészített alkalmazásból azzal a különbséggel, hogy egy új `ApartmentsDataSource` adatforrás hozzáadásakor nemcsak a `building` táblát, hanem a `city` és `shore` táblákat is megjelöljük az adatbázisban. Ekkor a típusos `DataSet` osztályba bekerül a két utóbbi adattábla, amelyekhez külön-külön típusos adapter osztályok jönnek létre. Az alkalmazást ugyanúgy készítjük el, mint korábban. Tervező nézetben az `ApartmentsDataSource` adatforrás `building` tábláját „drag&drop” technikával az űrlapra húzzuk.

Vegyük ezek után sorra, milyen módosításokat kell az alkalmazás durva változatán elvégezni.

1. A `building_id` oszlopot el kell rejtenuk, hogy itt ne módosíthassa a felhasználó (ezt a `ReadOnly` tulajdonságával is beállíthatnánk), és ne is lássa, hiszen ez úgysem fejez ki számára a feladat szempontjából hasznos információt. Gondoskodnunk kell ugyanakkor arról, hogy egy új sor beszúrásakor a `building_id` értéke automatikus generálódjon. (Ezt sokszor az adatbázisnál, tehát a szerver oldalon megírt tárolt eljárás végzi, mi azonban most a kliens oldalon oldjuk meg ezt.)
2. A `city_id` oszlopban a település-azonosító helyett mindig az annak megfelelő település neve jelenjen meg. Egy település megváltoztatásakor nem az azonosítót, hanem annak nevét szeretnénk megadni, de azt a `city` táblában tárolt településneveket tartalmazó listából, hogy véletlenül se írassunk be olyan nevet, ami nincsen. (Egy másik modulban, a `city` tábla karbantartása keretében lehetne új települést felvenni az adatbázisba.) Az oszlop stílusa ne szövegdoboz legyen, hanem a fenti funkcionalitást támogató kombináldoboz (`ComboBox`), amit a keretrendszer készen szolgáltat számunkra.
3. A `shore_id` oszlopnál ugyanaz a helyzet, mint a `city_id` esetében.
4. A `features` oszlopban nem számkódot akarunk látni, hanem az ennek keretében bejelölt (bejelölhető) épület-tulajdonságok listáját, mondjuk egy ellenőrződoboz-lista (`CheckedBoxList`) formájában. Ehhez nekünk kell megírni azt a programrészletet, amely a számkód alapján kitölti az ellenőrződoboz-listát, és fordítva, az ellenőrződoboz-lista alapján elkészíti az adattáblába írandó számkódot. Ez egy egyedi oszlop definiálását igényli.
5. A `street` oszlop az adatbázisban nem tartalmazhat nullértéket, ezért ennek kitöltését kötelezővé kell tennünk. Ezt az igényt azonban csak a hibakezelésnél fogjuk kielégíteni.

Nézzük meg most ezeket a lépéseket részletesen!

Azonosító elrejtése és értékének automatikus generálása

Egy oszlop elrejtése, azaz az adatrácsból való kivétele nagyon egyszerű. Kattintsunk a Tervező nézetben az adatrács jobb felső sarkában levő `SmartTag`-re, és a felnyíló ablakban válasszuk ki az „`Edit Columns ...`” opciót. A megjelenő „`Edit Columns`” ablakban törölhetjük (`Remove`) az adatrácsból a `building_id` oszlopot.

Az automatikus értékgeneráláshoz jelöljük ki a Tervező nézetben az `ApartmentsDataSet` komponenst, és annak „`SmartTag`”-jére kattintva válasszuk ki a felnyíló ablak „`Edit in DataSet Designer ...`” opcióját. (Ugyanezt a hatást érjük el a `Solution explorer`-beli `ApartmentsDataSet.xsd` állomány kinyitásával.) Ekkor megjelenik a típusos adathalmaz

mindhárom táblája. Jelöljük ki a building tábla building_id oszlopát és a Properties ablakban állítsuk be az alábbi tulajdonságot:

AutoIncrement : true

Idegenkulcs értékének megjelenítése, kombináldobozos kiválasztása

Ismét az adatrács „Edit Columns” ablakában dolgozunk. Jelöljük ki a city_id oszlopot. Állítsuk be az alábbi tulajdonságokat:

ColumnType: DataGridViewComboBoxColumn

DataSource: Other Data Sources/ Project Data Sources/ ApartmentsDataSet/ city

DisplayMember: name

ValueMember: city_id

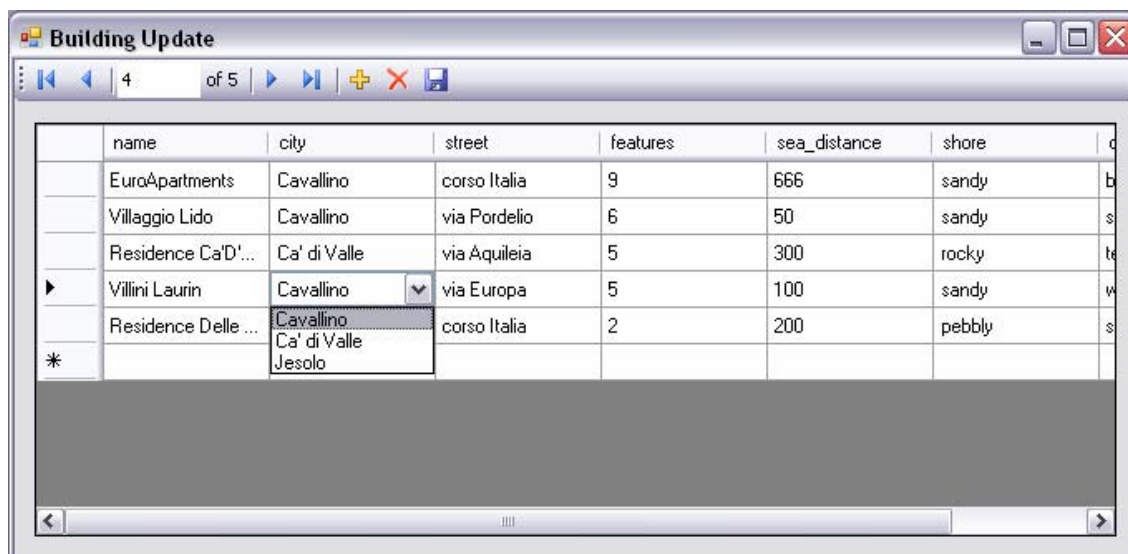
HeaderText: city

DisplayStyleForCurrentCellOnly: true

A legutolsó tulajdonság az inaktív cella megjelenését módosítja: csak akkor látszódjon erről az oszlopról, hogy ennek cellái kombináldobozok, amikor a cellára „rálépünk”. Ne lepődjünk meg azon, hogy közben újabb adatkapcsolat objektum jelent meg az alkalmazásunkban: ez köti az adathalmaz city tábláját a city_id oszlop kombináldobozának listájához.

Az oszlop beállításakor még eljátszhatunk az oszlopszélességének beállításával.

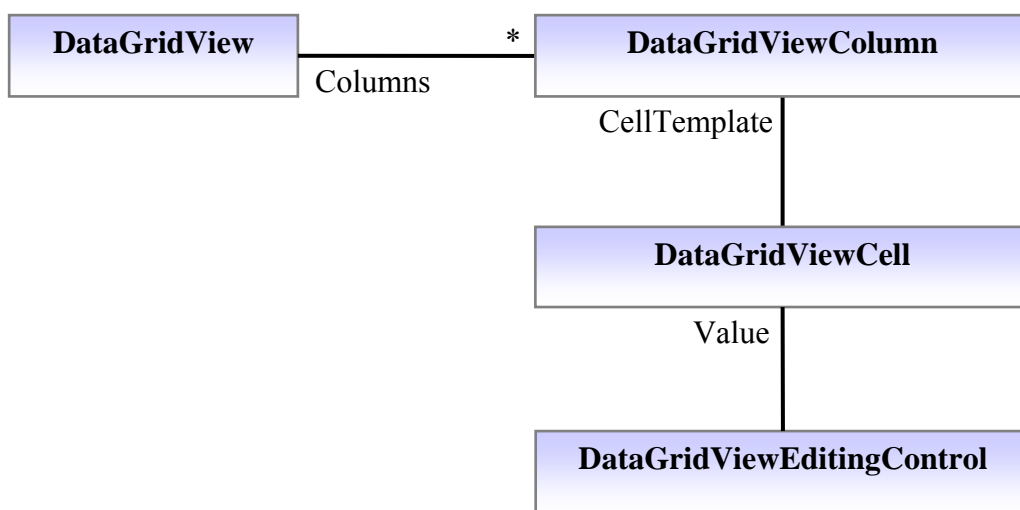
A fentieket értelemszerűen megismételjük a shore_id oszlopra is.



Egyedi cellaformák

Alaphelyzetben az adatrács csak néhány oszlop-típussal rendelkezik. Ezeket az adatrács oszlopai `ColumnType` tulajdonságának beállításánál megjelenő kombináltdobozban olvashatjuk. Ha ezektől eltérő cellaformájú oszlopot szeretnénk definiálni, akkor közelebbről meg kell ismerkednünk az adatrács belső világával.

Mivel az egy oszlopba tartozó cellák azonos típusúak, ezért az oszlopra jellemző cellatípust egy `DataGridViewColumn` osztályból származtatott osztályban írhatjuk le. Ennek az osztálynak egy objektumát kell az adott oszlophoz hozzárendelni, amelynek a `CellTemplate` tulajdonságából olvasható ki az oszlophoz tartozó cellák típusa. Az oszlophoz tartozó cellákat egy `DataGridViewCell` osztályból származtatott osztály írja le. Amikor az adatrács újabb sorral bővül, akkor az adott oszlophoz tartozó cella ennek a származtatott osztálynak új objektuma lesz. Ez az objektum felel a cella megjelenéséért passzív illetve aktív állapotban, és az utóbbi esetben természetesen ismernie kell azt a vezérlőt, amelynek működése során a cella értékét (`Value`) meg lehet változtatni. Ez egy úgynevezett szerkesztő-vezérlő osztálynak a példánya, amely osztálynak a `DataGridViewEditingControl` osztályból kell származnia, amely biztosítja a cella értékének szerkeszthetőségét.



Nekünk most egy olyan oszloptípusra van szükségünk, amely lehetővé teszi, hogy egy cellában tetszőleges számú szövegesen megadott összetevőt felsoroljunk (az épületek esetében ezek a parkoló hely, kert, úszómedence, strandszolgáltatás), és lehetővé teszi, hogy ezek közül bármelyiket – egyszerre akár többet is – megjelöljünk. Erre még találhatunk is „gyári” vezérlőt (ilyen a `CheckedListBox`), de nekünk a megjelöléseket egész szám formájában kódolva kell visszaadni, és fordítva, egy egész szám alapján be kell tudni jelölnünk a megfelelő összetevőket. Ehhez készítünk egy egyedi vezérlőt, amely osztályát a `CheckedListBox` osztályból származtatjuk, hogy felhasználhassuk annak már meglévő kódját. Kiegészítjük ezt a szükséges számításokkal. Ezután ebből elkészítjük a saját `DataGridViewEditingControl` osztályunkat, amelyhez legyártjuk a megfelelő saját `DataGridViewCell` és `DataGridViewColumn` osztályokat is. Végül a VS tervezőjével hozzárendeljük a megfelelő adatrács oszlophoz azt az új oszloptípust.

1. Egyedi szerkesztő-vezérlő készítése

A vezérlőnk alapját tehát egy ellenőrződoboz-lista alkotja (`CheckedListBox`), amit kiegészítünk egy tulajdonsággal (`Value`), ami a megjelöléseket kódoló egész szám lesz. Ennek a számnak 2-komplement kódjában a bitek mutatják meg azt, hogy egy összetevő meg van-e jelölve (1-es bit), vagy nincs (0-ás bit). Így a 0 egész szám azt kódolja, amikor egyik összetevő sincs megjelölve, az 1 azt, amikor csak a legutolsó, a kettő hatványai azt, amikor pontosan egy, és például a 6 azt, amikor az utolsó előtti és az azelőtti.

Az új vezérlő osztályát a `CheckedListBox` osztályra támaszkodva kétféleképpen is elkészíthetjük: egy `CheckedListBox` vezérlőt adattagként tartalmazó (kompozíció vagy erős tartalmazási kapcsolat) vezérlő osztállyal (azaz a `MyDataGridViewColumn`-ból származtatva), vagy közvetlenül a `CheckedListBox` osztályból származtatva.

Adjunk egy `Windows Control Library` projektet a `solution`-hoz mondjuk `MyDataGridViewColumn` néven. Ennek egy osztálya lesz a `FeaturesSettingControl`, amit a `CheckedListBox`-ból származtatunk. Ehhez fel kell venni a projekt referenciái közé a `.NET/System.Windows.Forms`-ot, és meg kell azt hivatkozni a `using` direktívával. A konstruktorban a `CheckedListBox` két tulajdonságát állítjuk be.

```
...
using System.Windows.Forms;

namespace MyDataGridViewColumn
{
    public class FeaturesSelectingControl : CheckedListBox
    {

        public FeaturesSelectingControl():base()
        {
            CheckOnClick = true;
            Click += new EventHandler(checkedListBox_Click);
        }
    }
}
```

A `FeaturesSelectingControl` osztályt egészítsük ki két tulajdonsággal. Az egyikkel le lehet kérdezni és meg lehet változtatni a vezérlő értékét, azt a számkódot, amely a listában megjelölt összetevőktől függ.

```
public int Value
{
    get { return Code(); }
    set { Decoding(value); }
}
```

A másik tulajdonsággal le lehet kérdezni és újra lehet definiálni a `checkedListBox`-ban megjelenő összetevők nevét.

```
public object[] List
{
    get
    {
        object[] list = new object[this.Items.Count];
        int i = 0;
        foreach (object item in Items)
        {
            list[i] = item;
            ++i;
        }
        return list;
    }
    set
    {
        this.Items.Clear();
        if (value != null) this.Items.AddRange(value);
    }
}
```

A kódolás és dekódolást végző privát metódusok egy egész szám értékének kettes számrendszerbeli alakjának, illetve kettes számrendszerbeli alak egész értékének meghatározására épülnek. A kódoláshoz és dekódoláshoz bitműveleteket (bitenként vagy, léptetés) használunk.

```
private void Decoding(int code)
{
    int n = this.Items.Count - 1;
    while (code != 0)
    {
        checkedListBox.SetItemChecked(n, (code & 1) == 1);
        code >>= 1;
        --n;
    }
    for (; n >= 0; --n)
        checkedListBox.SetItemChecked(n, false);
}
```

```
private int Code()
{
    int code = 0;
    int n = checkedListBox.Items.Count - 1;
    foreach (int k in checkedListBox.CheckedIndices)
        code |= (1 << n - k);
    return code;
}
```

Végül elkészítjük a Click eseménykezelőt. Ennek egyetlen feladata, hogy meghívja azt az OnValueChanged() virtuális metódust, amelyet majd a most készített osztályból származtatott osztályban lesz lehetőségünk felülírni.

```
private void checkedListBox_Click(object sender, EventArgs e)
{
    OnValueChanged(e);
}

protected virtual void OnValueChanged(EventArgs eventargs)
{
}
```

2. Új adatrács-oszlop típus definiálása

Ha megvan a kívánt vezérlő, akkor – függetlenül attól, hogy saját vagy „gyári” – el kell belőle készíteni egy új adatrács-oszlop típust. Ezt a munkát érdemes külön projektben végezni, így az újrafelhasználhatóság egyszerű lesz. A projekt neve legyen DataGridViewFeaturesColumn. Ez a projekt három osztályt fog tartalmazni: a DataGridViewColumn osztályból származó DataGridViewFeaturesColumn osztályt, a DataGridViewTextBoxCell osztályból származó a DataGridViewFeaturesCell osztályt és a FeaturesEditingControl osztályt, amelyet egyszerre két osztályból is származtatni kellene, egyrészt a FeaturesSelectingControl, másrészt a DataGridViewEditingControl osztályokból. Mivel azonban a C# nem támogatja a többszörös öröklődést², a FeaturesEditingControl osztály a DataGridViewEditingControl osztályból való származtatás helyett implementálni fogja az IDataGridViewEditingControl interfészt. Az itt beígért osztályok definiálásához érdemes megnézni a súgóban (DataGridView/ Customization), hogyan szokták az ilyen jellegű problémákat megoldani, és annak mintájára készíthetjük el a saját osztályainkat. Projektünk referenciái közé vegyük fel a MyDataGridViewColumn projektet, hiszen abban definiáltuk a FeaturesSelectingControl osztályt.

² A C# nem támogatja a többszörös öröklődést; több osztályból csak úgy lehet származtatni, ha azok között csak egy „rendes” osztály van, a többi pedig úgynevezett interfészt. Az interfész – nem a programnyelv felől szemlélve – tulajdonképpen egy olyan absztrakt osztály, aminek nincsenek adatai, és az összes metódusa is absztrakt. Az ilyen „nagyon absztrakt” osztályból való származtatás, az úgynevezett implementálás esetén az összes metódushoz implementációt kell írni.

A DataGridViewFeaturesColumn osztály

Ezt az osztályt DataGridViewColumn-ból származtatjuk, hiszen rendelkeznie kell az adatrács-oszlopok olyan szokásos tulajdonságaival, mint az oszlop fejléce, az oszlop szélességének beállítása, vagy annak az információnak tárolása (CellTemplate), amely megmondja az adatrácsnak, hogy egy adott sor adott oszlopánál milyen típusú cella jelenjen meg. Az alább bemutatott osztály két egyedi elemet mutat az őszosztályához képest. Van egy publikus – tehát kívülről beállítható – List adattagja, amely alapján a cellában megjelenítendő egyedi FeaturesSelectingControl vezérlőnk List tulajdonságát állítjuk be (ettől függ, hogy milyen szövegek jelennek meg a CheckedListBox-ban). A másik a CellTemplate tulajdonság felüldefiniálása úgy, hogy annak az általunk definiálandó DataGridViewFeaturesCell osztályt lehessen megadni.

```
public class DataGridViewFeaturesColumn: DataGridViewColumn
{
    public DataGridViewFeaturesColumn()
        :base(new DataGridViewFeaturesCell()){ }

    public override DataGridViewCell CellTemplate
    {
        get{ return base.CellTemplate; }
        set
        {
            if (value != null && !value.GetType().
                IsAssignableFrom(typeof(DataGridViewFeaturesCell)))
            {
                throw new InvalidCastException(
                    "Must be a FeatureCell");
            }
            base.CellTemplate = value;
        }
    }

    public virtual object[] List
    {
        get { return((DataGridViewFeaturesCell)this.CellTemplate)
                .List; }
        set {((DataGridViewFeaturesCell)this.CellTemplate)
                .List = value; }
    }
}
```


A DataGridViewFeaturesCell osztály

Ezt az osztályt a DataGridViewTextBoxCell osztályból származtatjuk azért, hogy a cellánk inaktív állapotban egy közönséges szövegdoboznak látszódjon. Ilyenkor az adatbázisban szereplő kódot látjuk a cellában, de ha rákattintunk, akkor megjelenik az ellenőrződobozos-listánk, amelyet szerkeszthetünk. Ez természetesen nem szép megoldás, amelyen a Paint metódus megfelelő felülírásával tudnánk változtatni, de ne felejtjük el, hogy most még csak az első változatot készítjük.

Az igazán lényeges része ennek az osztálynak az InitializeEditingControl() metódus felülírása. Ennek két funkciója van. Egyrészt értékül adja a DataGridViewFeaturesColumn-ban definiált List tulajdonságot az egyedi vezérlőnek képviselő FeaturesEditingControl (ennek definícióját mindjárt medmutatjuk) List tulajdonságának, másrészt a cella Value tulajdonságát a FeaturesEditingControl Value tulajdonságának. Ezen kívül még három tulajdonságát kell felülírni az osztálynak.

```
class DataGridViewFeaturesCell: DataGridViewTextBoxCell
{
    public DataGridViewFeaturesCell() : base() { }

    public override void InitializeEditingControl(
        int rowIndex, object initialFormattedValue,
        DataGridViewCellStyle dataGridViewCellStyle)
    {
        base.InitializeEditingControl(
            rowIndex, initialFormattedValue, dataGridViewCellStyle);
        FeaturesEditingControl ctl = DataGridView.EditingControl
            as FeaturesEditingControl;
        DataGridViewFeaturesColumn fc;
        DataGridViewColumn dgvc = OwningColumn;
        if (dgvc is DataGridViewFeaturesColumn)
        {
            fc = dgvc as DataGridViewFeaturesColumn;
            ctl.List = fc.List;
            ctl.Value = (Value == null || Value.ToString() == "")
                ? 0 : (int)Value;
        }
    }

    public override Type EditType
    {
        get{ return typeof(FeaturesEditingControl); }
    }

    public override Type ValueType
    {
        get{ return typeof(int); }
    }

    public override object DefaultNewRowValue
    {
        get{ return ""; }
    }
}
```

A FeaturesEditingControl osztály

Ennek az osztálynak egyszerre két osztályra is hasonlítani kell. Egyrészt az általunk kreált a FeaturesSelectingControl osztályra, hiszen azt akartuk a adatrács-cellában megjeleníteni. Másrészt olyannak kell lennie, mint a DataGridViewEditingControl osztály, hiszen ilyenre van szüksége az előbb definiált DataGridViewFeaturesCell osztálynak. A .NET rendelkezik a DataGridViewEditingControl osztálynak az interfész párjával, az IDataGridViewEditingControl interfésszel, amit a FeaturesEditingControl implementálni fog, de ennek az a következménye, hogy nagyon sok olyan metódust is tartalmaz, amelynek csak annyi a szerepe, hogy van.

```
class FeaturesEditingControl:
    MyDataGridViewColumn.FeaturesSelectingControl,
    IDataGridViewEditingControl
{
    protected DataGridView dataGridView;
    protected bool valueChanged = false;
    protected int rowIndex;

    public FeaturesEditingControl(): base() {}

    protected override void OnValueChanged(EventArgs eventargs)
    {
        base.OnValueChanged(eventargs);
        valueChanged = true;
        EditingControlDataGridView.NotifyCurrentCellDirty(true);
    }

    protected virtual void NotifyDataGridViewOfValueChange()
    {
        valueChanged = true;
        if (this.dataGridView != null)
        {
            dataGridView.NotifyCurrentCellDirty(true);
        }
    }
}
```

És végül az interfész implementálása miatt megadandó tulajdonságok.

```
public object EditingControlFormattedValue
{
    get{return this.Value.ToString();}
    set{this.Value = (int) value; }
}
public object GetEditingControlFormattedValue(
    DataGridViewDataErrorContexts context)
{
    return EditingControlFormattedValue;
}

public void ApplyCellStyleToEditingControl(
    DataGridViewCellStyle dataGridViewCellStyle) {}

public int EditingControlRowIndex
{
    get{return rowIndex; }
    set{rowIndex = value; }
}

public bool EditingControlWantsInputKey(
    Keys key, bool dataGridViewWantsInputKey)
{
    return false;
}

public void PrepareEditingControlForEdit
    (bool selectAll) {}

public bool RepositionEditingControlOnValueChange
{
    get{return false; }
}

public DataGridView EditingControlDataGridView
{
    get{return dataGridView; }
    set{dataGridView = value; }
}

public bool EditingControlValueChanged
{
    get{ return valueChanged; }
    set{ valueChanged = value; }
}

public Cursor EditingPanelCursor
{
    get{ return base.Cursor; }
}
}
```

3. Adatrács oszloptípusának megváltoztatása

Utolsó lépésként térjünk vissza az első projektünkhöz, és adjuk ennek a referenciáihoz a DataGridViewFeaturesColumn projektet. Ezután az űrlapra feltett adatrács oszlopainak szerkesztésénél (Tervező nézetben az adatrács kijelölése után annak jobb felső sarkán megjelenő SmartTag-gel) válasszuk ki a „features” oszlopot, majd annak ColumnType tulajdonságához jelöljük meg az ott felnyíló ablakban már látható DataGridViewFeaturesColumn oszlop típust. Adjuk a „features” adatrács-oszlop objektumának a dataGridViewFeaturesColumn nevet.

Az űrlap konstruktorában paraméterezzük fel a features adatrács-oszlopot. Sajnos e nélkül kezeletlen kivételt dob a programunk.

```
public BuildingUpdateForm()
{
    InitializeComponent();
    object[] list = { "parking site", "garden",
                    "swimming pool", "beach service" };
    dataGridViewFeaturesColumn.List = list;
}
```

Érdeemes az adatrács szerkesztési mód tulajdonságát átállítani: **EditMode** : EditOnEnter



Megjegyezzük, hogy az itt bemutatott változat kényelmes és tetszetős használatához még néhány módosítást végre kellene hajtani. Például nem szép, hogy a cella passzív állapotban az adatbázisban tárolt egész számot mutatja. Így „read only” üzemmódban nem lehet megtekinteni a ténylegesen bejelölt jellemzőket, csak a bejelölés kódját. Ezt a következő fejezetben orvosoljuk majd.

3. Hibaellenőrzés

A hibaellenőrzés alapvető elve az, hogy a keletkező hibát annak keletkezési helyéhez és idejéhez minél közelebb fedezzük fel. Egy adattábla karbantartást végző modulban, amikor az adattáblát az adatbázisból a memóriába másoljuk, a hiba ellenőrzést több szinten kell végezni.

1. Mező szint: egy sor egy mezőjének helyességét önmagában, a sor illetve a tábla többi adatától függetlenül vizsgáljuk.
2. Sor szint: egy sor mezőinek kapcsolatát ellenőrizzük.
3. Tábla szint: A szerkesztett sornak és a tábla többi sorának kapcsolatát ellenőrizzük.
4. Adatbázis szint: A szerkesztett sort összevetjük az adatbázis többi táblájával. Ezt többnyire nem soronként végezzük el, hanem akkor, amikor a táblát visszamentjük az adatbázisba. Értelemszerűen ilyenkor az ellenőrzés az adatbázis oldalon történik. (Kivételt képezhetnek az idegenkulcs-ellenőrzések, melyek érdemes a mező szintű ellenőrzés alatt úgy megoldani, ahogy ezt a `city_id` illetve a `shore_id` esetében tettük, azaz eleve kizárjuk a hibázás lehetőségét.)

Megjegyezzük, hogy ha az első három szinten az adatbázisbeli megszorításoktól eltérő módon végzünk hibaellenőrzést, akkor számos hiba csak a 4. szinten jelentkezik.

A felfedett hibáról valamilyen formában mindig tájékoztatni kell a felhasználót, és ki kell kényszeríteni a hiba kijavítását. Óvakodjunk azonban attól, hogy alkalmazásunk túl erőszakos, ennél fogva felhasználó ellenes legyen. Például nem kell ahhoz ragaszkodni, hogy a felhasználó azonnal helyes adatokat vigyen fel egy sor szerkesztése során, elég, ha a sor szerkesztésének befejezésekor ellenőrizzük a mezőket.

Mezőellenőrzés

A hibaellenőrzés legalsó szintje az egy egységként bevitt vagy szerkesztett adatok önálló vizsgálata. Itt ellenőrizzük, hogy egy mező (cella, oszlop) értéke megfelelő típusú-e (szám, karakterlánc, dátum, stb.), megfelelő-e a formája (pl. telefonszám forma, EHA-kód forma), megadott tartományba esik-e (pl. 1 és 10 közötti egész szám), és idegenkulcs esetén létező elemre hivatkozik-e. (Ez utóbbi kilóg a többi ellenőrzés közül, hiszen eldöntéséhez egy másik táblát kell vizsgálni.)

A helyes mező biztosításának legegyszerűbb módja az, amikor olyan felületet alakítunk ki, amelyen keresztül nem lehet hibás adatot bevinni. Ezt alkalmaztuk a `builing_id` elrejtésénél és automatikus generálásánál, vagy a `city_id`, `shore_id` és `features` megadásánál. Számos olyan vezérlőt (`NumericUpDown`, `DateTimePicker`, `MaskedTextBox`) találunk, amelyekkel a `features` mező kitöltésénél bemutatott technikához hasonlóan csak helyes adatokat lehet megadni. Ha ezt mégsem tudjuk biztosítani, akkor hibaellenőrző metódust kell írni. Adatrácsbeli adatszerkesztés esetén ezt a metódust célszerű az adatrács `CellValidating` eseményéhez delegálni. Ez az esemény akkor következik be, amikor a cellát éppen el akarjuk hagyni. (Ne tévesszük össze a `CellValidated` eseménnyel, amely már csak a cella elhagyása után következik be.) Mivel mindegyik cella esetében ugyanaz a `CellValidating` esemény váltódik ki, erre közös hibakezelő metódust kell írni.

```
private void buildingDataGridView_CellValidating(object sender,
        DataGridViewCellValidatingEventArgs e)
{
    DataGridViewRow row = buildingDataGridView.Rows[e.RowIndex];
    row.ErrorText = "";

    string colname = buildingDataGridView.Columns[e.ColumnIndex]
        .DataPropertyName;

    int n;

    if ((colname == "sea_distance" ) &&
        !(int.TryParse(e.FormattedValue.ToString(), out n) && n > 0))
    {
        row.ErrorText = "Sea_distance must be positive integer!";
        e.Cancel = true;
    }
}
```

A hibellenőrzés egy sokágú elágazás, ahol az egyes ágak, az aktuális sor egyes oszlopaihoz tartozó cellaértékeket ellenőrzik. Az `e.ColumnIndex` megmutatja, hogy éppen melyik oszlophoz tartozó cellaértéket kell ellenőrizni, azaz melyik ágra kerüljön a vezérlés. Az ellenőrizendő értéket az `e.FormattedValue`-ban találjuk. Ha az érték hibás, akkor az `e.Cancel = true` értékadással tudjuk a felhasználót visszakényszeríteni az éppen vizsgált cellába.

A hiba kiírásának egy egyszerű módja az aktuális sor hibaüzenetének használata. (Ugyanezt használjuk a sor szintű ellenőrzésnél is.) Az aktuális sort az `e.RowIndex` segítségével kapjuk meg. Ha ennek `Errors` tulajdonsága az üres sztringet tartalmazza, akkor nincs hibaüzenet. Ellenkező esetben egy úgynevezett hiba ikon (alap esetben piros korongban egy felkiáltójel) utal a hibára, a kurzort az ikonra mozgatva egy felbukkanó dobozban megjelenik az `Errors` tulajdonságban tárolt hibaüzenet.

Ha nem adatrácsban, hanem külön-külön vezérlőkben szerkesztjük az adattábla aktuális sorát (erre láttunk példát az 1. – Durva változat testre szabás nélkül c. – fejezet végén), akkor a vezérlők `Validating` eseményéhez kell delegálnunk eseménykezelőt. Itt is lehet közös eseménykezelőt írni, amelynek első paramétere az eseményt kiváltó vezérlő objektum lesz, ami alapján beazonosítható, hogy mely értéket kell ellenőriznünk. Az `e.Cancel = true` értékadás kényszeríti vissza itt is a felhasználót a hibát okozó vezérlőhöz.

A hibaüzenet kijelzésére elegáns eszköz az `ErrorProvider`. Ez egy speciális objektum, amit a Tervező nézetben a `ToolBox`-ból tudunk az alkalmazásunk űrlapjához hozzáadni. Ennek az objektumnak a `SetError()` metódusával jelezhetjük a hibát. A metódus első paraméterében megadhatjuk azt a vezérlőt, amelyben a hiba keletkezett, második paramétere pedig a hibaüzenet. Ha az üzenet nem üres sztring, akkor a jelzett vezérlő mellett megjelenik egy hiba ikon, és ha arra állítjuk a kurzort, egy felbukkanó dobozban a hibaüzenet olvasható. „Kikapcsolni” úgy lehet a hibajelzést, ha a `SetError()` metódus hibaüzenet paraméterének az üres sztringet adjuk.

Sorellenőrzés

A building tábla esetében nincs a mezők között olyan kapcsolat, amelyet ellenőrizni kellene. Ugyanakkor a sorellenőrzés keretében meg lehet vizsgálni azt, hogy azok a mezők, amelyek nem tartalmazhatnak null értéket, ki lettek-e töltve. Ez lényegét tekintve mezőellenőrzési eset, de itt célszerű vizsgálni, hogy amíg a felhasználó az aktuális sor mezői között lépked, ne bosszantsuk feleslegesen a kitöltetlenségre vonatkozó hibaüzenetekkel.

Adatrács esetén a RowValidating kivételkezelő a sorellenőrzés helye. Az adatkapcsolati objektum EndEdit() metódusa minden függőben levő módosítást elküld az adatforrásnak, esetünkben az adathalmazbeli building adattáblának. Ha egy olyan mező értéke null, amit az adattábla nem enged meg, akkor kiváltódik a NonNullAllowedException kivétel.

```
private void buildingDataGridView_RowValidating(object sender,
    DataGridViewCellCancelEventArgs e) {
    DataGridViewRow row = buildingDataGridView.Rows[e.RowIndex];
    row.ErrorText = "";

    try{
        this.buildingBindingSource.EndEdit();
    }catch (NonNullAllowedException ex) {
        row.ErrorText = ex.Message.Substring(0,15)
            + " must be filled in";
        e.Cancel = true;
    }
}
```

Amennyiben külön vezérlőkkel szerkesztjük a sort, akkor a szerkesztés befejezését jelző nyomógomb Click eseményéhez rendelt hibaellenőrzést használjuk.

Táblaellenőrzés

Megfelelő mező és sorellenőrzés mellett kiszűrhetőek az olyan, valójában tábla szintű hibák, mint például az egyedi oszlopérték megsértése vagy egy idegenkulcs tulajdonságot sértő mező érték. Ha ezt nem tettük volna meg, akkor a DataError esemény váltódik ki. (A kapcsolati objektum is, és az adatrács is rendelkezik DataError hibaeseménnyel.)

Adatbázis ellenőrzés

Ha elég figyelmesek voltunk, és az adatbázis minden megszorítását figyelembe vettük, a tábla visszamentéskor még mindig bekövetkezhetnek hibák, kezdve attól, hogy nem nyitható meg a korábban már felépített kapcsolat, egészen addig, hogy egy több felhasználós alkalmazás esetén az általunk módosított adatokat közben más felhasználó is módosította. Ilyenkor a legkevesebb, hogy ezekről a hibajelenségekről tájékoztassuk a felhasználót, amelyhez a visszamentésnél az SQLException kivételt kell lekezelni.

Összetett megjelenítések

ÖSSZETETT MEGJELENÍTÉSEK	48
1. MEZŐK EGYEDI LÁTVÁNNYAL.....	50
<i>Ellenőrző dobozos oszlop</i>	50
<i>Napok neveit megjelenítő oszlop</i>	50
<i>Egyedi vezérlővel ellátott oszlop</i>	52
2. SZÁMÍTOTT MEZŐK	54
<i>Számított mező saját sor alapján</i>	54
<i>Számított mező szülőtábla alapján</i>	55
<i>Számított mező gyermektábla alapján</i>	56
<i>Számított mező saját tábla alapján</i>	56
3. KAPCSOLT TÁBLA ÉS KARBANTARTÁSA	57
<i>Gyermektábla megjelenítése</i>	57
<i>Gyermektábla karbantartása</i>	57
<i>Dinamikusan változó listájú kombináldoboz</i>	58
4. SZŰRÉS.....	62
<i>Szűrés beállításának helye</i>	62
<i>Összetett szűrést beállító egyedi vezérlő</i>	62

Egy adatbázis egymással összefüggő adattáblákból áll. Ezért egy adattáblának akár nézegetés, akár módosítás céljából történő megjelenítésénél a vele kapcsolatban álló más táblákban tárolt adatok is szerepet játszanak.

Egy adattábla összetett megjelenítésén azt értjük, amikor nemcsak a táblában tárolt adatokat tesszük ki a felhasználói felületre, hanem az ezekkel az adatokkal kapcsolatos más adatokat is. Ezt tettük már az előző fejezetben is, amikor idegenkulcs megjelenítése helyett, az azzal azonosított másik táblában tárolt jelentést mutattuk meg a felhasználói felületen. De ide sorolható az épület jellemzőinek ellenőrződoboz-listás egyedi megjelenítése is.

Az összetett megjelenítés további lehetősége, hogy olyan, úgynevezett számított oszlopokat is kiírhatunk, amelyek közvetlenül nem szerepelnek az adatbázisban. Ezek értékét az aktuális tábla, illetve az azzal kapcsolatban álló más táblák alapján számíthatjuk ki. Előfordulhat az is, hogy szülő-gyermek kapcsolatban álló táblákat egyszerre kell megmutatnunk, de a gyermektáblának csak azon sorait, amelyek a szülőtábla aktuális sorához vannak rendelve.

A feladatokat egy képzelt utazási iroda apartman-foglalási tevékenységéhez használt „Apartments” adatbázisára fogalmazzuk meg. Ebben a fejezetben ennek az adatbázisnak számos táblájával dolgozni fogunk, amelyek között a központban az „apartment” nevű tábla fog állni.

```
CREATE TABLE apartment (
    apartment_id INTEGER PRIMARY KEY,
    building_id INTEGER NOT NULL,
    number VARCHAR(10),
    floor TINYINT,
    room VARCHAR(30) NOT NULL,
    bed TINYINT,
    spare_bed TINYINT,
    turnday TINYINT,
    features INTEGER,
    renovation INTEGER
    comment VARCHAR(100),
```



```

        FOREIGN KEY (building_id) REFERENCES building
    );

CREATE TABLE price (
    apartment_id INTEGER PRIMARY KEY,
    season_id INTEGER PRIMARY KEY,
    price INTEGER,
    FOREIGN KEY (apartment_id) REFERENCES apartment,
    FOREIGN KEY (season_id) REFERENCES season
);

CREATE TABLE season (
    season_id INTEGER PRIMARY KEY,
    name VARCHAR(30)
    start_date DATETIME,
    end_date DATETIME,
);

CREATE TABLE building (
    building_id INTEGER PRIMARY KEY,
    name VARCHAR(30),
    city_id INTEGER NOT NULL,
    street VARCHAR(30) NOT NULL,
    sea_distance INTEGER,
    shore_id INTEGER,
    features INTEGER,
    comment VARCHAR(100),
    FOREIGN KEY (city_id) REFERENCES city,
    FOREIGN KEY (shore_id) REFERENCES shore
);

CREATE TABLE city (
    city_id INTEGER PRIMARY KEY,
    name VARCHAR(30)
);

```

Tűzzük ki a megoldandó feladatot:

Feladat: *Készítsünk olyan grafikus felületű alkalmazást, amely lehetőséget ad az apartmanok megjelenítésére és egy apartmannak a különböző szezonokban megállapított bérleti díjainak megadására!*

Először az apartment adattábla megjelenítésével foglalkozunk. Elrejtjük a felhasználó elől a táblában tárolt belső azonosítókat (ehhez, ha kell, egyedi cella formát definiálunk), de megjelenítjük az olyan fontos kapcsolódó adatokat, mint az apartmannak a building táblából kiolvasható neve és címe, az apartment tábla aktuális sorából kiszámítható ágy/szoba arány, az apartmannal egy épületben levő apartmanok száma, és az apartman (price gyermek táblából kiszámítható) átlag ára.

Ezután az apartment táblához kapcsolódó price tábla megjelenítését és karbantartását biztosító felületet alakítjuk ki, de úgy, hogy a price táblának csak azokat a sorait mutatjuk meg, amelyek az éppen kijelölt apartmanhoz kapcsolódnak. Ahogy az apartmanokat mutató felületen megváltoztatjuk a kijelölt apartmant, úgy fog változni az árlistát mutató nézet.

A továbbiakban több olyan műveletsort végzünk, amelyet az előző fejezetben már részletesen megtárgyaltunk. Ezért ezekre itt kisebb figyelmet szentelünk.

1. Mezők egyedi látvánnyal

Hozzunk létre egy Windows Application alkalmazást! Rendeljük hozzá adatforrást, amelybe beemeljük az apartments adatbázis fent felsorolt adattábláit. Legyen az így létrejött adathalmaz neve: priceDataset. (Ne felejtjük el még ezelőtt az adatbázisban az összes táblakapcsolatot definiálni és elmenteni.) Húzzuk rá (drag&drop) az adatforrásból az űrlapra az apartment táblát, amely által létrejön egy adatrács, egy navigátor, egy adathalmaz, egy adapter és egy adatkapcsolati objektum. Vegyük le az adatrácsból az apartment_id-t és a building_id-t, gondoskodjunk a turnday, features és renovation oszlopok dekódolt megjelenítéséről. Állítsuk át az **EditMode** tulajdonságot **EditOnEnter**-re. Végül legyen az adatrács **ReadOnly** és töröljük a navigátorsáv „mentés” nyomógombja (apartmentBindingNavigatorSaveItem) Click eseménykezelőjének törzsét, hiszen a jelen alkalmazásnak nem feladata az apartmanok módosítása. Ezzel két legyet is ütöttünk egy csapásra: nem kell az apartment_id automatikus generálásáról gondoskodni, és nincs szükség az újonnan bevitt adatok hibaellenőrzésére.

Ellenőrző dobozos oszlop

Jelenítsük meg az igaz/hamis értékű renovation mezőt egy ellenőrző doboz segítségével!

A renovation mező az apartman azon tulajdonságát mutatja, hogy az használható-e éppen, nincs-e felújítás alatt. A renovation oszlop ColumnType tulajdonságát a gyári DataGridViewCheckBoxColumn típusra állítsuk be. Ezt a tervező nézetben az adatrács kijelölése után annak jobb felső sarkán megjelenő SmartTag-gel felnyitott ablak EditColumn menüpontjánál megjelenő oszlopszerkesztővel tehetjük meg legegyszerűbben.

Napok neveit megjelenítő oszlop

A turnday mező a hét egyik napjának sorszámát tárolja. Jelenítsük meg ezt a megfelelő nap nevével!

Az apartmanok csak teljes hetekre bérelhetők, és minden apartmannál rögzített, hogy a bérlet a hét melyik napján kezdődik. A turnday mező a hét ezen napjának sorszámát tartalmazza, amely helyett a megfelelő nap nevét szeretnénk megmutatni. Ez a helyzet nem nagyon különbözik attól az esettől, amikor a building tábla karbantartásánál a city_id helyett a település nevet kellett megjeleníteni.

Az egyetlen eltérés az, hogy a hét napjai nincsenek az adatbázis egy külön táblájában felsorolva. Egy adatrácsbeli oszlop adatforrása azonban nemcsak egy adatbázisbeli tábla lehet, hanem bármi: esetünkben a hét napjait felsoroló gyűjtemény, például egy tömb. Persze ebben a napokat a sorszámukkal együtt kell felsorolni, hiszen a napok sorszáma a különböző nyelvi kultúrákban eltérhet. (Magyarországon a hét első napja a hétfő, de Angliában a napok felsorolása a vasárnapkal kezdődik.) A .NET alatt az aktuális nyelvi kultúrát könnyen be tudjuk építeni a programunkba. A `Thread.CurrentThread.CurrentCulture = new CultureInfo("en-GB");` utasítással (a `Thread` a `Systems.Threading` névtérben található) a brit-angol nyelvi kultúrát állíthatjuk be. Ezután a napok angol neveit a `Thread.CurrentThread.CurrentCulture.DateTimeFormat.DayNames` gyűjteményből tudjuk kiolvasni.

Elsőként definiálunk a projektünkben egy Day osztályt, amely a hét egy napjának típusát írja meg. Ennek két tulajdonsága van: a nap neve illetve a nap sorszáma.

```
public class Day
{
    private byte dayNumber;
    public byte DayNumber
    {
        get { return dayNumber; }
        set { dayNumber = value; }
    }
    private string dayName;
    public string DayName
    {
        get { return dayName; }
        set { dayName = value; }
    }
    public Day(string name, byte n)
    {
        dayName = name;
        dayNumber = n;
    }
}
```

Másodszor a Data Sources nézetben egy új adatforrást készítünk a varázslóval. Az adatforrás alapja most egy Day típusú objektum (`object`) lesz, nem adatbázis. Kijelöljük a projektünk Day osztályát (ez csak azt követően látszik a varázslóban, ha már újrafordítottuk a projektet).

Harmadik lépésben az apartment táblát mutató adatrácshoz szerkesztjük hozzá (SmartTag / Edit column) a turnday oszlopot.

ColumnType: DataGridViewComboBoxColumn
DataSource: Other Data Sources/ Project Data Sources/ price/ Day
DisplayMember: DayName
ValueMember: DayNumber
DisplayStyle: Nothing

Megjegyezzük, hogy most többet is tettünk a kelleténél, hiszen ha ennek az oszlopnak nem lenne igaz a `ReadOnly` tulajdonsága, akkor az oszlop mezőben egy kombinált doboz segítségével lehetne kiválasztani a megfelelő napot.

Hátra van még, hogy a most létrejött `dayBindingSource` objektum `DataSource` tulajdonságához hozzárendeljük a hét napjait tartalmazó gyűjteményt. Legyen ez a `week` tömb, amelyet az űrlapunk osztályának adatagjaként definiálunk, és a konstruktorban töltjük fel a megfelelő sorszámozással együtt.

```

using System.Globalization;
using System.Threading;

public partial class PriceViewForm : Form
{
    public Day[] week = new Day[8];

    public PriceViewForm()
    {
        InitializeComponent();
        ...

        week[0] = new Day("none", 0);
        byte n = 0;
        Thread.CurrentThread.CurrentCulture =
            new CultureInfo("en-GB");
        foreach (string dayname in Thread.CurrentThread.
            CurrentCulture.DateTimeFormat.DayNames)
        {
            week[n] = new Day(dayname, n);
            ++n;
        }
        dayBindingSource.DataSource = week;
    }
    ...
}

```

Egyedi vezérlővel ellátott oszlop

A features mezőben tárolt kód helyett azon tulajdonságok ellenőrző dobozos listáját jelenítsük meg, amelyek egy apartman tulajdonságait alkotják: kilátás, terasz, légkondicionálás!

A features oszlop típusának beállítása pont ugyanúgy történik, ahogy ezt korábban, a building tábla features oszlopával tettük. Felhasználhatjuk az ott önálló projekt keretében elkészített DataGridViewFeaturesColumn oszloptípust. (Még az sem kell, hogy az a project a mostani solution része legyen, elég a megfelelő dll kiterjesztésű fájlt a mostani projektünk hivatkozásai közé felvennünk.) Újrafordítás után az adatrács oszlopszerkesztőjének segítségével beállítjuk a „features” oszlop ColumnType tulajdonságát az ott felnyíló ablak által most már felkínált DataGridViewFeaturesColumn típusra. Végül adjuk a „features” adatrács-oszlopnak a dataGridViewFeaturesColumn nevet.

Egyetlen probléma adódik csak, nevezetesen, hogy mi módon akadályozzuk meg, hogy a features mezőben felsorolt elemek kijelölését meg tudja változtatni a felhasználó. Ha az oszlop ReadOnly tulajdonságát állítjuk igazra, akkor a cellákban felsorolt elemeket sem tudjuk végiggörgetni. E helyett egy igen egyszerű megoldást kínál az, ha a features mező vezérlője egy CheckedListBox típusú vezérlőből származik, amelyik rendelkezik egy úgynevezett SelectionMode tulajdonsággal. Ha ezt None-ra állítjuk, akkor a vezérlő csak olvasható lesz. Ehhez egyrészt fel kell venni a DataGridViewFeaturesColumn osztályba egy publikus SelectionMode típusú tagot (`public SelectionMode selectionmode`), másrészt gondoskodni kell arról, hogy a DataGridViewFeaturesCell osztály InitializeEditingControl() metódusa (lásd korábban) átadja ezt az információt az egyedi vezérlőknek. Ehhez egyetlen értékadást kell beilleszteni a korábbi kódba.

```

public override void InitializeEditingControl(...)
{
    base.InitializeEditingControl(...);
    FeaturesEditingControl ctl =
        DataGridView.EditingControl as FeaturesEditingControl;
    DataGridViewFeaturesColumn fc;
    ...
    fc = this.OwningColumn as DataGridViewFeaturesColumn;
    ctl.SelectionMode = fc.SelectionMode;
    ctl.List = fc.List;
    ctl.Value = (this.Value == null
        || this.Value.ToString() == "") ? 0 : (int)this.Value;
}
}

```

Az újonnan bevezetett tulajdonságot beállíthatjuk az alkalmazásunk űrlapjának konstruktorában ott, ahol a vezérlőben megjelenített összetevők listáját is megadjuk. (A lista megadása nélkül kezeletlen kivételt dob a programunk.)

```

object[] list = { "looking sea", "looking green",
                 "terrace", "air condition" };
dataGridViewFeaturesColumn.List = list;
dataGridViewFeaturesColumn.SelectionMode = SelectionMode.None;

```

Hátra van még egy, az előző fejezetből maradt adósság törlesztése. Ez az egyedi oszloptípus egyelőre olyan, hogy amikor egy cella nem aktív, akkor nem az ellenőrződoboz-listás formában jelenik meg, hanem azt a számkódot mutatja, amely ténylegesen az adatbázisban tárolva van. Ez nem jó, általános cél, hogy a belső kódokat nem mutatjuk a felhasználó felé.

Az egyedi vezérlővel ellátott oszlopok esetén a megjelenést a `DataGridViewCell` osztály szintjén, esetünkben az abból származtatott `DataGridViewFeaturesCell` osztály `Paint()` metódusában tudjuk szabályozni. Ez egy felüldefiniált metódus.

```

protected override void Paint( Graphics graphics,
    Rectangle clipBounds, Rectangle cellBounds,
    int rowIndex, DataGridViewElementStates cellState,
    object value, object formattedValue,
    string errorText, DataGridViewCellStyle cellStyle,
    DataGridViewAdvancedBorderStyle advancedBorderStyle,
    DataGridViewPaintParts paintParts)
{
    base.Paint(graphics, clipBounds, cellBounds, rowIndex,
        cellState, value, formattedValue, errorText, cellStyle,
        advancedBorderStyle, paintParts);
    Rectangle newRect = new Rectangle(
        cellBounds.X + 1, cellBounds.Y + 1,
        cellBounds.Width - 4, cellBounds.Height-4);
    graphics.FillRectangle(Brushes.LightGray, newRect);
    graphics.DrawString("click here", new Font("Arial", 8),
        Brushes.Black, newRect);
}

```

A metódus meghívja az őszosztály `Paint()` metódusát, majd megtervezi a passzív cellák látványát. Ebben az esetben ez egy egyszerű fedőtéglaalap a „click here” felirattal. Nyilván ennél ügyesebb megjelenést is lehetne tervezni. Semmi akadálya például az aktuálisan kiválasztott jellemzőket (azok rövidítéseit) felsorolni. Hozzáférhetünk a jellemezők teljes listájához (az `InitializeEditingControl`-ban látott módon), a kiválasztást kódoló számhoz (`value` paraméter), melynek dekódolását a `FeaturesSelectingControl` osztály `Decoding()` metódusához hasonló módon elvégezhetjük.

2. Számított mezők

Számított mezőnek egy tábla azon oszlopait nevezzük, amelyek az eredeti táblában nincsenek jelen, értéküket valamilyen módon, többnyire más adatbázisbeli értékek alapján számolhatjuk, és ezért természetesen nem lehet őket közvetlenül módosítani.

Annak megfelelően, hogy egy számított mező értékének előállításához milyen más adatokra van szükség, különféle eseteket különböztethetünk meg.

Mielőtt ezen eseteket megvizsgálánk, módosítsuk az űrlap `Load` eseményekor lefutó eseménykezelőt. Ez jelenleg az `apartment` tábla betöltését végzi, de nekünk a többi táblát is be kell tölteni az adathalmazba. Hozzuk létre a `building`, `city`, `price` és `season` táblák kezeléséhez alkalmas adapter objektumokat (ezeket a `ToolBox` segítségével példányosíthatjuk), és írjuk bele a `Load` eseménykezelőjébe ezen táblák betöltését végző metódushívásokat.

```
private void Form1_Load(object sender, EventArgs e)
{
    apartmentTableAdapter.Fill(apartmentsDataSet.apartment);
    priceTableAdapter1.Fill(apartmentsDataSet.price);
    buildingTableAdapter1.Fill(apartmentsDataSet.building);
    cityTableAdapter1.Fill(apartmentsDataSet.city);
    seasonTableAdapter1.Fill(apartmentsDataSet.season);
}
```

Számított mező saját sor alapján

Egészítsük ki az `apartment` tábla megjelenítését egy olyan mezővel, amely az apartmanbeli ágy/szoba arányt (ráta) mutatja!

A számított mezőt a memóriában tárolt `apartment` adattáblához új oszlopként vegyük hozzá. Tekintsük az adatállomány tervezői nézetét (`priceDataSet.xsd`). Kattintsunk az `apartment` táblára a jobb egérrel, és a felnyíló ablakok segítségével adjunk egy új oszlopot a táblához. Ennek tulajdonságait az alábbiak szerint állítsuk be.

Name: rate
Caption: rate
Expression: bed/room
DataType: System.Double
ReadOnly: true

A számított értéket definiáló kifejezés az `apartment` tábla aktuális sora `bed` és `room` mezőinek értékeire hivatkozik.

Ezt követően az apartmanokat mutató adatrács oszlopaihoz is vegyük hozzá az új mezőt. Az oszlopszerkesztő ilyenkor már „látja” ezt, és fel is kínálja a hozzávehető oszlopok között.

Számított mező szülőtábla alapján

Egészítsük ki az apartment tábla megjelenítését olyan mezőkkel, amely az apartmant tartalmazó épület tulajdonságait, mondjuk a building táblából kiolvasható nevet, település és utca nevet mutatja!

Ebben az esetben lényeges szerep jut a building tábla és az apartment tábla közötti szülő-gyerek kapcsolatnak. Az apartment tábla building_id mezője idegenkulcs a building táblára nézve. Minden apartman pontosan egy épülethez tartozik. Ezt a kapcsolatot az adatbázisban FK_apartment_building néven jegyeztük be.

Adjunk új oszlopokat az apartment táblához az adatállomány tervezői nézetét (priceDataSet.xsd) használva.

Az épület nevének, mint számított mezőnek az értéke a building táblában (ő a szülő tábla) található, annak a name mezőjében. Ezt, amikor a building az egyedüli szülő tábla, a parent.name hivatkozás definiálja. Több szülő tábla esetén a parent(FK_apartment_building).name kifejezést kell használnunk. Teljesen hasonló módon járjunk el az épület utcanevének számított mezőként való felvételénél. (Mindkét mező automatikusan ReadOnly tulajdonságú lesz.)

Table: apartment
Name: name
Expression: parent.name

Table: apartment
Name: street
Expression: parent.street

Bonyolultabb a helyzet a település nevének megjelenítésénél. Erre egy két lépésből álló technikát mutatunk. Először számított mezőt hozunk létre a building táblában, majd erre az új számított mezőre hivatkozó másik számított mezőt az apartment táblában.

Table: building
Name: city
Expression:
parent(FK_building_city).name

Table: apartment
Name: city
Expression: parent.city

Végül gondoskodunk az apartment tábla új számított mezőinek az adatrácsban való megjelentetéséről.

(Megjegyezzük, hogy amikor a building tábla karbantartásánál a city_id-ek helyett a megfelelő település neveket mutattuk meg az adatrácsban, akkor hatásában ugyanazt értük el, mintha külön számított mezőt definiáltunk volna a településneveknek, és az adatrácsban csak ezt, a city_id-t pedig nem jelenítettük volna meg.

Csak a teljesség igénye miatt említjük meg azt is, hogy fogalmilag ebbe a pontba sorolható az olyan eset is, amikor egy mező számított értékét nem egy másik táblából, hanem egy tetszőleges gyűjtemény alapján számoljuk ki. Erre a legjobb példa az előző fejezetben szerkesztett turnday mező, amelynek háttérében a week objektum áll. Itt is igaz – akárcsak a building_id esetében –, hogy eredetileg egy belső azonosítót tartalmazó mezőnk volt, amely helyett annak jelentését jelenítettük meg, amelyet a week gyűjteményből „számolhattunk ki”.)

Számított mező gyermektábla alapján

Egészítsük ki az apartment tábla megjelenítését egy olyan mezővel, amely az apartman átlagos bérleti díját mutatja! Ez a price tábla segítségével számolható, amely az apartman egy hétre szóló bérleti díjait tartalmazza a különböző szezonokban.

A price tábla gyermektáblája az apartment táblának. Kapcsolatukat az FK_price_apartment név azonosítja. Most a price az egyetlen gyermektáblája az apartment-nek, ezért a child(FK_price_apartment) hivatkozás helyett használhatjuk az egyszerűbb child-ot. Minden price-beli sor valamelyik apartmanhoz tartozik, és egy apartmannak a különböző szezonokban kért bérleti díjait tartalmazza. Ezen díjak átlagát szeretnénk kiszámítani. Ehhez használható az *Avr()* aggregációs függvény.

Először az apartment tábla új számított mezőjét definiáljuk az eddig látott módon.

Table: apartment
Name: average fee
Expression: Avg(child.price)

Utána felvesszük ezt a mezőt az adatrács oszlopai közé.

Számított mező saját tábla alapján

Egészítsük ki az apartment tábla megjelenítését egy olyan mezővel, amely az apartmannal egy épületben levő apartmanok összes számát mutatja!

Úgy tűnhet, hogy ez egyszerűbb eset, mint az előző kettő, hiszen nem igényli a kapcsolt táblák vizsgálatát. Ha azonban nem akarunk az eddigieken kívül más eszközöket használni, akkor – a *Count()* függvény korlátai miatt – az alábbi körülményesebb utat kell választanunk.

Először a building táblánál hozzuk létre az itt igényelt számított mezőt,

Table: building
Name: count
Expression: Count(child.building_id)

majd ezt a mezőt az apartment táblába „másoljuk”

Table: apartment
Name: neighbour
Expression: parent.count

Ezt követően az adatrács oszlopaihoz is vegyük hozzá ezt a neighbour mezőt.

3. Kapcsolt tábla és karbantartása

Most azt mutatjuk meg, hogy kapcsolt táblák esetén hogyan lehet olyan nézetet létrehozni, amelyben egy tábla a gyermek táblájával együtt jelenik meg az úrlapon (a gyermek táblában mindig a fő tábla aktuális sorához kapcsolódó sorokat látszódnak), és hogyan módosíthatóak a gyermek tábla adatait.¹

Gyermektábla megjelenítése

Jelenítsük meg az apartment tábla mellett az ahhoz tartozó bérleti díjakat szezononként elkülönítve!

Váltsunk át az alkalmazás úrlapját mutató tervező nézetbe! Az ehhez rendelt adatforrásban keressük meg a price táblát. Vigyázat! Az adatforrásban több helyen is találkozhatunk a price táblával. Önállóan is, de az apartment táblának alárendelve is. Ez utóbbi nem a teljes price táblát, hanem annak csak azt nézetét szimbolizálja, amely az aktuális apartmanhoz tartozó price táblabeli sorokat mutatja. A price táblának ezt a szűrését a varázsló automatikusan beállítja.

Húzzuk be az úrlapra az apartment táblának alárendelt price táblát! (Feltételezzük, hogy az apartment tábla adatrácsa az előző fejezetek módosításain átesve már ott van.) Ekkor megjelenik egy újabb adatrács (megfelelő adapter objektum és adatkapcsolati objektum kíséretében), amely után az alkalmazás fordítható, futtatható.

Először végezzünk el néhány módosítást a price adatrácsán (ehhez hasonlókat már korábban is csináltunk, így ezek nem jelenthetnek gondot). Rejtsük el a price tábla adatrácsán a belső azonosítókat mutató oszlopokat. Jelenítsük meg a season_id helyett a szezon nevét. Ehhez használhatunk egy kombinált dobozként megjelenő mezőt. Jelenítsük meg a szezon időszakát is de úgy, hogy vegyük figyelembe az apartman forduló napját (turnday)! A szezon határ ugyanis többnyire nem esik egybe az apartman fordulónapjával, ezért az adott apartmant esetében a szezonhatárt a rákövetkező fordulónapra kell eltolni. Ehhez meg kell határozni, hogy a szezon határ a hét melyik napjára esik (legyen ennek sorszáma h), és ha a fordulónap sorszáma f , akkor a szezonhatár dátumát $(f+7-h)\%7$ -tel kell megnövelni.

Gyermektábla karbantartása

Jelenleg az apartment táblát nem akarjuk módosítani, ezért az azt mutató adatrács ReadOnly tulajdonsága igaz, továbbá kivettük a navigátorsáv „mentés” nyomógombjának Click eseménykezeléséből az apartment tábla visszamentését is. Most helyezzük el a price tábla mentését végző kódrészletet a navigátorsáv „mentés” nyomógombja Click eseménykezelőjének törzsében.

¹ Nem tértünk ki a másik irányra: a fő tábla aktuális sorához tartozó szülő táblabeli sorok megjelenítésére. Ilyen lenne például az, amikor megjelenítjük egy településen található összes épület listáját. Ez ugyanis egy tipikus szűrési feladat, amellyel később foglalkozunk.

```
private void apartmentBindingNavigatorSaveItem_Click(
    object sender, EventArgs e)
{
    this.Validate();
    this.apartmentBindingSource.EndEdit();
    this.priceTableAdapter.Update(apartmentsDataSet.price);
}
```

Dinamikusan változó listájú kombináltdoboz

Vértezzük fel hibaellenőrzéssel a price tábla adatrácsának a szerkesztését úgy, ahogyan ezt a building tábla karbantartásánál láttuk. Itt egy dologra kell figyelni. Egy apartmannál minden szezonhoz pontosan egy árat lehet nyilvántartani. A felvitelnél még megengedhetjük azt, hogy bizonyos szezonok árai ne legyenek megadva, de azt nem, hogy ugyanarra a szezonra többféle ár is legyen: az apartman és a szezon egy összetett egyed kulcs a price táblában. Ennek ellenőrzését elvégezhetjük a price tábla aktuális sorának érvényesítésekor (Validated), de az igazán elegáns megoldás az lenne, ha a szezon megadásakor felbukkanó kombinált doboz, soha nem ajánlana fel olyan szezon neveket, amelyeket az előző megszorítás miatt úgysem lehetne választani.

Zárjuk ki annak a lehetőségét, hogy egy apartman ugyanazon szezonjához egyszerre több árat is meg lehessen adni!

Ezt a korlátozást ott lehet érvényesíteni, amikor a price tábla újabb sorában a season mezőt töltjük ki. Azt könnyű megoldanunk, hogy itt a valóságban nyilvántartott season azonosítók helyett annak neve jelenjen meg. Ennek megvalósításával már több esetben is találkoztunk.

ColumnType:	DataGridViewComboBoxColumn
DataSource:	seasonDataSources
DisplayMember:	name
ValueMember:	season_id
DisplayStyle:	DropDownButton
DisplayStyleForCurrentCellOnly:	true

Az adatforrásnál szereplő seasonBindingSource objektumot úgy hozzuk létre, hogy a DataSource első beállításakor az Other Data Sources/ Project Data Sources/ priceDataSet/ season választást adjuk meg. Ezzel a megoldással a kombináltdobozban egyelőre mindig az összes szezon neve fog megjelenni. Most ezt kellene nekünk megszüntetni úgy, hogy csak azok a szezonok legyenek választhatóak, akik még nem szerepelnek a price tábla adott apartmanhoz tartozó soraiban. A setCurrentFilter() metódus a seasonBindingSource objektum Filter tulajdonságát állítja majd be a megfelelő szűrőfeltételre, míg a szűrés kikapcsolását egyszerűen a seasonBindingSource = "" értékadással végezzük el.

A nehézséget az okozza, hogy amikor egy mező megfelelő kitöltése érdekében beállítunk egy olyan szűrőfeltételt, amely kizárja az adott oszlopban már szereplő szezonokat, akkor ez a szűrés az egész oszlopra fog vonatkozni. Ha az oszlop vagy valamelyik már korábban kitöltött cellája újrarajzolására kerül sor, akkor azok a mezők, ahová korábban már szezon nevet írtunk, üressé válnak, hiszen a szűrőfeltétel nem engedi az értékük megjelenítését. Ezért a szűrést csak akkor szabad bekapcsolni, mielőtt az adott mező kitöltését segítő kombinált doboz legördülő menülistája megjelenik, és a lista lezárása után a szűrést azonnal ki kell kapcsolni.

További probléma, hogy az aktuális cellánál alkalmazott szűrés kizárja az aktuális cellába korábban beírt nevet is. Ezért minden alkalommal, amikor a szűrést bekapcsoljuk a szűrő feltételt gyengíteni kell, hogy az ne zárja ki az aktuális cella értékét (ha van ilyen). Időkimélő megoldás, ha eltároljuk azt a szűrő feltételt (`base_filter`), amely az összes adott oszlopbeli nevet kiszűri, és egy cella szűrésének beállításakor ezt gyengítjük az adott cella értékével. Az alapbeállítást a `setBaseFilter()` metódus végzi.

```
private string base_filter;

private void setBaseFilter()
{
    base_filter = "";
    foreach (priceDataSet.priceRow row in priceDataSet.price)
    {
        base_filter += string.Format(" AND season_id<>'{0}' ",
                                     row.season_id);
    }
    if(base_filter!="") base_filter = base_filter.Substring(4);
}
```

A `base_filter`-t elég egyrészt a `price` tábla betöltésekor (`priceForm_Load()`) kiszámolni, másrészt, amikor a `price` tábla `season` oszlopának valamelyik mezője módosul (`priceDataGridView_RowValidated()`).

```
private void priceForm_Load(object sender, EventArgs e)
{
    priceTableAdapter.Fill(this.priceDataSet.price);
    apartmentTableAdapter.Fill(this.priceDataSet.apartment);

    buildingTableAdapter.Fill(priceDataSet.building);
    cityTableAdapter.Fill(apartmentsDataSet.city);
    seasonTableAdapter.Fill(apartmentsDataSet.season);

    setBaseFilter();
}

private void rentDataGridView_RowValidated(
    object sender, DataGridViewCellEventArgs e)
{
    setBaseFilter();
}
```

A `setCurrentFilter()` metódust kétféleképpen is meglehet hívni. Paraméter nélkül, ilyenkor egyetlen feladata a `seasonBindingSource.Filter` értékének átadni a `base_filter`-t, vagy annak partnernek az azonosítójával (ez egy egész szám), amelyet ki akarunk venni a szűrésből.

```

private void setCurrentFilter(params int[] current_id)
{
    string filter = base_filter;
    if (filter != "" && current_id.Length > 0) filter += " OR ";
    if ( current_id.Length > 0 )
        filter += string.Format(" season_id = '{0}' ",
                                current_id[0]);
    seasonBindingSource.Filter = filter;
}

```

Ezt a metódust mindig meg kell hívni, amikor egy szezonév szerkesztéséhez felnyitjuk a kombinált dobozt, illetve meg kell szüntetni a szűrést, amikor a kombinált doboz lenyíló menüjét lezárjuk. Erre a legelegánsabb megoldás a kombinált doboz DropDown és DropDownClosed eseményeinek kezelése lenne. Az első esetben az adott cella aktuális értékével (ez egy season azonosító) meg kell hívni a setCurrentFilter()-t (ha nincs még értéke a cellának, akkor paraméter nélkül), a második esetben a seasonBindingSource.Filter = "" értékadásra van szükség. Ügyeljünk arra, hogy a szűrés változásakor a kombinált doboz listájában korábban kiválasztott sor megváltozik, ezért azt ideiglenesen meg kell jegyezni és újra beállítani.

```

private ComboBox seasonComboBox;

private void seasonComboBox_DropDown(object sender, EventArgs e)
{
    if (priceDataGridView.CurrentRow.Value == DBNull.Value)
        setCurrentFilter();
    else{
        int i = (int) seasonComboBox.SelectedValue;
        setCurrentFilter(
            (int)priceDataGridView.CurrentRow.Value);
        seasonComboBox.SelectedValue = i;
    }
}

private void seasonComboBox_DropDownClosed
    (object sender, EventArgs e)
{
    int i = (int) seasonComboBox.SelectedValue;
    seasonBindingSource.Filter = "";
    seasonComboBox.SelectedValue = i;
    priceDataGridView.Refresh();
}

```

E fenti két eseménykezelőt nem tudjuk a tervezővel hozzárendelni a megfelelő eseményekhez. A hozzárendelés kódját például a priceDataGridView adatrács EditingControlShowing eseményének kezelőjébe helyezhetjük. Figyeljük meg, hogyan lehet beazonosítani, hogy az adatrács éppen melyik oszlopában állunk.

```
private void priceDataGridView_EditingControlShowing(
    object sender, DataGridViewEditingControlShowingEventArgs e)
{
    if (((DataGridView)sender).CurrentCell.OwningColumn.
        DataPropertyName == "season_id")
    {
        seasonComboBox = priceDataGridView.EditingControl as
            DataGridViewComboBoxEditingControl;
        seasonComboBox.DropDown +=
            new EventHandler(seasonComboBox_DropDown);
        seasonComboBox.DropDownClosed +=
            new EventHandler(seasonComboBox_DropDownClosed);
    }
}
```

Végül még egy apróság. Amikor a kombinált doboz listája lenyílik és túlnyúlik az adatrács belső területén, akkor az adatrács újrarajzolja magát. Mivel ilyenkor be van kapcsolva a szűrő, az újrarajzolás a többi sor season mezőjében megjelenített neveket ideiglenesen törli. Ezt úgy lehet megakadályozni, hogy az újrarajzolás előtti szűrőfeltételt elmentjük, a szűrést kikapcsoljuk, majd az újrarajzolás után visszakapcsoljuk

```
private void priceDataGridView_RowsAdded(
    object sender, DataGridViewRowsAddedEventArgs e)
{
    string filter = seasonBindingSource.Filter;
    seasonBindingSource.Filter = "";
    priceDataGridView.Refresh();
    seasonBindingSource.Filter = filter;
}
```

4. Szűrés

Szűrésről akkor beszélünk, amikor a felhasználói felületen a tárolt adatainknak csak egy részét mutatjuk meg. Egy adattábla szűrésén az adattábla sorainak szűrését értjük. Ilyen szűrésekkel már eddig is találkoztunk, például amikor a price táblának csak apartment táblának aktuálisan kiválasztott sorához (egy adott apartmanhoz) tartozó sorait jelenítettük meg egy adatrácsban (kapcsolt tábla megjelenítése), vagy amikor az előbb bemutatott dinamikusan változó kombinált doboz tartalmát állítottuk be.

Szűrés beállításának helye

Az adattábla szűrését több ponton is meg lehet valósítani.

1. Adatbázisból való betöltéskor

Az adatbázis egy táblájának memóriába töltésekor is lehet már szűrést alkalmazni, ha a betöltést vezérlő parancs (DataCommand) objektumban vagy az adapter (DataAdapter) objektum SelectCommand tulajdonságában egy megfelelő SQL SELECT lekérdezést vagy tárolt eljárást adunk meg. Erre a megoldásra akkor lehet szükség, ha a feladat megoldása során ritkán kell változtatni a leszűrt sorokat.

2. Adattábla sorainak leválogatásával

A memóriában tárolt tábla (DataTable) objektum sorait is leválogathatjuk egy szűrésnek megfelelően, és az eredményt elhelyezhetjük egy másik tábla objektumban vagy adatsorokat tartalmazó tömbben (DataRow[]). Erre írhatunk saját, egyedi kódot, de használhatjuk az adattábla Select metódusának egyikét.

```
DataRow[] Select(string filter),
```

```
DataRow[] Select(string filter, string sorting),
```

```
DataRow[] Select(string filter, string sorting, DataRowView state)
```

Ezekben a filter egy szűrőfeltételt megadó string, amely szintaxisa az SQL SELECT parancsok WHERE záradéka mögött leírt feltételével azonos. Nem megfelelő alakú szűrőfeltétel futás közben dob kivételt.

A sorting a szűrt adatok rendezettségét, a state a leszűrendő sorok státuszát (új, törölt, módosított, változatlan, eredeti, stb.) adja meg.

3. Adatkötésben

Egy adattábla szűrését beállíthatjuk azon kapcsolati (DataSource) objektum Filter tulajdonságával, amelyek az adatok megjelenítéséért felelős vezérlőket kötik az adattáblához. Ez a legrugalmasabb megoldás, amelyet a gyakran változó szűrőfeltételek esetén használunk.

Összetett szűrést beállító egyedi vezérlő

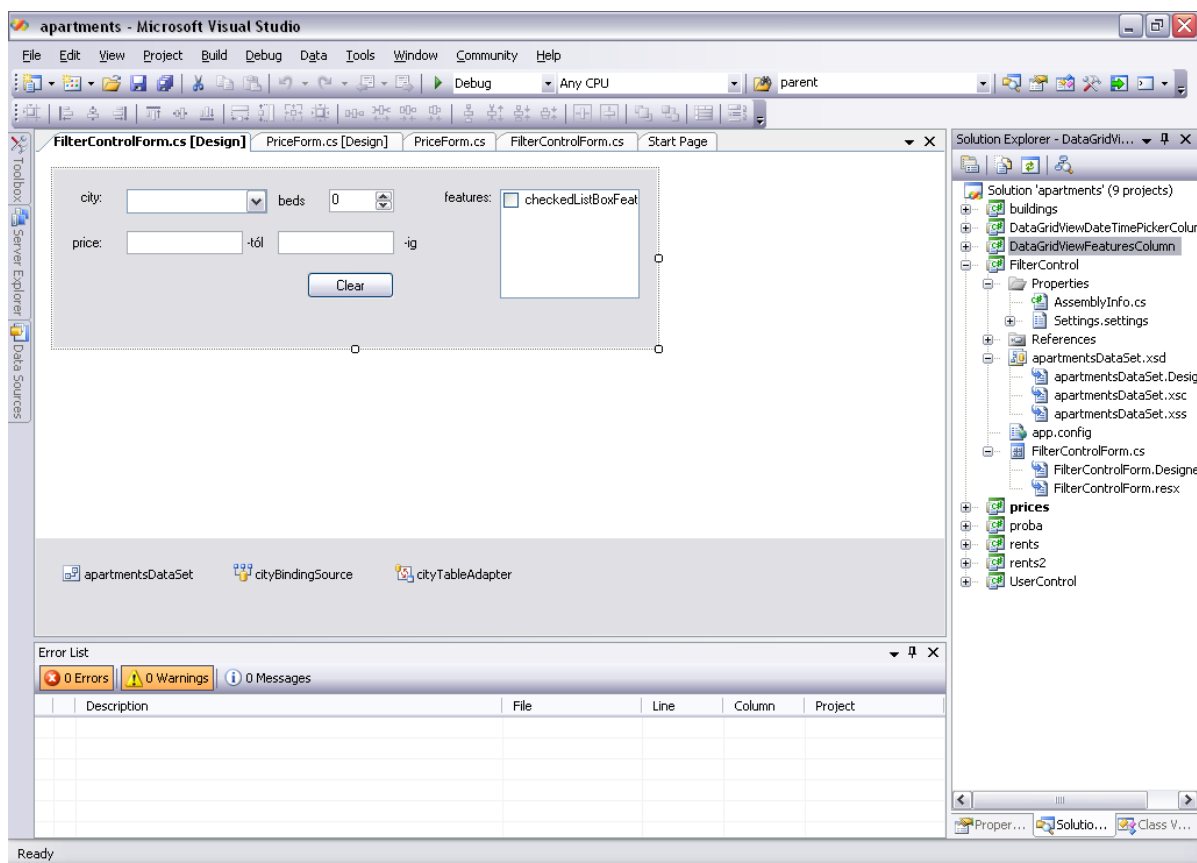
Az, hogy a fenti lehetőségek közül melyikkel élünk, az adott feladaton múlik. Itt elsősorban hatékonysági szempontok (memória igénynek és futási időnek optimalizálása) alapján kell dönteni. A továbbiakban a legutolsó technikát alkalmazzuk egy olyan feladatban, ahol a felhasználó adhat meg egy összetett szűrőfeltételt a táblákat megjelenítő adatrácsokhoz.

Tegyük lehetővé, hogy az apartment tábla és a hozzá tartozó price tábla sorait különböző szempontok szerint szűrve jeleníthessük meg.

Készítsünk először egy olyan egyedi vezérlőt (filtercontrol), amellyel a felhasználó össze tud állítani egy szűrő feltételt. Ehhez ismernünk kell egyrészt azokat a táblákat, amelyeket szűrni akarunk, azokat a szempontokat, amelyek alapján egy tábla szűrését el kell végeznünk és végül azokat a vezérlőket, amelyekkel az egyes szempontok szűrőfeltétele beállítható. Ezeket a vezérlőket elhelyezzük a filtercontrol felületén, lehetővé tesszük, hogy olyan tulajdonságai legyenek, amelyekkel lekérdezhethjük az egyes táblák összetett szűrőfeltételeit, és definiálunk olyan eseményeket, amelyek az egyes szűrőfeltételek megváltozását jelzik.²

Mi most az apartment és price táblák szűrését akarjuk megoldani. Az apartmanokat a település neve, ágyak száma és a jellemzők (features) alapján, a price táblát pedig a bérleti díjaktól-ig határainak megadásával.

Nyissunk egy új Windows Control Library projektet! Alakítsuk ki a felhasználói felületet az ábrának megfelelően. Mindegyik szűrési szemponthoz tartozik egy vezérlő. A címke feliratok utalnak az egyes vezérlők szerepére. A vezérlők egymástól eltérő módon jelzik, hogy őket figyelembe kell-e venni a szűrőfeltétel összeállításánál vagy sem. Többségüknél ezt egy különleges érték (üres sztring, 0 érték) jelzi.



² Ezt a vezérlőt elkészíthetnénk generikus formában is úgy, hogy annak megfelelő paraméterezése révén biztosítsa a paraméterként megadott táblák paraméterként megadott szűrőfeltételeinek összeállítását.

Készítsük el a filtercontrol kódját! Definiáljuk a két eseményt, amelynek kiváltásáról majd mi gondoskodunk. Bevezetjük a *filter* tömböt, amely az egyes szűrő kritériumokat fogja tartalmazni sztring formában. Ha egy szűrő kritérium az üres sztring, akkor az azt jelöli, hogy arra az esetre nem vonatkozik megszorítás. Azt, hogy melyik szűrőkritérium melyik indexen található a filter-ben, a filterCriteria felsorolási típus mutatja.

```
public event EventHandler ApartmentFilterEvent;
public event EventHandler PriceFilterEvent;

private enum filterCriteria { city, bed, priceLow, priceHigh,
                             features };

private string[] filter = new
    string[(int)(filterCriteria.features)+1];
```

Ezek után defináljuk azt a két tulajdonságot, amelyekről az apartment, illetve a price tábla összetett szűrőfeltétele lekérdezhető. Mindkét tulajdonság támaszkodik a composite() metódusra, amely a szűrőkritériumokból állít össze egy szűrőfeltételt. Ezeket a tulajdonságokat akkor érdemes lekérdezni, ha a szűrőfeltétel megváltozását a megfelelő esemény kiváltódása jelzi.

```
public string ApartmentFilter
{
    get
    {
        return composite(filter[(int)filterCriteria.city],
                        filter[(int)filterCriteria.bed],
                        filter[(int)filterCriteria.features]);
    }
}

public string PriceFilter
{
    get
    {
        return composite(filter[(int)filterCriteria.priceLow],
                        filter[(int)filterCriteria.priceHigh]);
    }
}

private string composite(params string[] filters)
{
    string result = "";
    bool l = false;
    foreach(string str in filters)
    {
        if (str!="")
        {
            result += l ? " AND " + str : str;
            l = true;
        }
    }
    return result;
}
```


Az egyes szűrő kritériumokat a megfelelő vezérlők értékének megváltozásakor állítjuk be. Ilyenkor váltjuk ki a megfelelő eseményt is. Üres sztring a szövegdobozban azt jelzi, hogy nem akarunk szűrő kritériumot megadni, tehát a szűrőkritérium is legyen üres. Ellenkező esetben a megfelelő szűrőfeltételt kell beállítani.

```
private void textBoxPriceHigh_TextChanged(...)
{
    if(textBoxPriceHigh.Text != "")
    {
        try{
            int n = int.Parse(textBoxPriceHigh.Text);
            filter[(int)filterCriteria.priceHigh] =
                string.Format("price<='{0}'",n);
        }catch(FormatException){ }
    }
    else filter[(int)filterCriteria.priceHigh] = "";
    PriceFilterEvent(sender, e);
}

private void textBoxPriceLow_TextChanged(...)
{
    if(textBoxPriceLow.Text != "")
    {
        try{
            int n = int.Parse(textBoxPriceLow.Text);
            filter[(int)filterCriteria.priceLow] =
                string.Format("price>='{0}'", n);
        }catch (FormatException) { }
    }
    else filter[(int)filterCriteria.priceHigh] = "";
    PriceFilterEvent(sender, e);
}
```

A számláló vezérlő 0 értéke jelzi azt, hogy nincs szűrés megadva, hiszen nulla darab ágygal rendelkező apartman keresése értelmetlen lenne. Ilyenkor a szűrőkritériumot üres sztringre kell állítani, ellenkező esetben a megfelelő szűrőfeltételt kell beállítani.

```
private void numericUpDownBeds_ValueChanged(...)
{
    if (numericUpDownBeds.Value != 0)
    {
        filter[(int)filterCriteria.bed] =
            string.Format("bed='{0}'",
                numericUpDownBeds.Value.ToString());
    }
    else filter[(int)filterCriteria.bed] = "";
    ApartmentFilterEvent(sender, e);
}

private void numericUpDownBeds_KeyPress(...)
{
    numericUpDownBeds_ValueChanged(sender, e);
}
```

A kombinált dobozban a lehetséges település neveken kívül egy üres sztringet is mutatunk. Ez utóbbi kiválasztásával jelezhetjük azt, hogy nem akarunk ezen szempont szerint szűrni. A település nevek ilyen megjelenítéséhez létre kell hozni azt az adatforrást a city táblára, amit előzőleg kiegészítünk az üres névvel, és ezen forrás alapján generáljuk azt a kapcsolati objektumot, amely a city táblát a kombinált dobozhoz köti.

Egy településnév kiválasztásakor gondoskodni kell arról, hogy a névben szereplő esetleges aposztróf karaktert megkülönböztessük a szűrőfeltételekben szereplő azon aposztróftól, amelyekkel az értékeket kell zárójelezni. Ennek módja a névben szereplő aposztróf duplázása. Az alábbi kódban tehát erről is gondoskodni kell a szűrőkritérium beállítása és a megfelelő esemény kiváltása mellett.

```
private void comboBoxCity_SelectedIndexChanged(...)
{
    if (comboBoxCity.Text != "")
    {
        string str = comboBoxCity.Text;
        int i = str.IndexOf('\\');
        while(i > -1)
        {
            str = str.Insert(i+1, "\\");
            i = str.IndexOf('\\', i+2);
        }
        filter[(int)filterCriteria.city] =
            string.Format("city='{0}'", str);
    }
    else filter[(int)filterCriteria.city] = "";

    if (ApartmentFilterEvent != null)
        ApartmentFilterEvent(sender, e);
}
```

Az ellenőrződoboz-listás vezérlővel úgy lehet szűrni az apartmanokat, hogy azokat a jellemzőket kell beállítani, amelyeket mindenképpen szeretnénk az apartmanban látni. A be nem jelölt jellemzők viszont nem azt jelentik, hogy azokat nem akarjuk az apartmanban, hanem azt, hogy azokról nem kívánunk nyilatkozni. Tehát mindig az összes olyan apartmant meg kell mutatnunk, amely features kódjának (a kód továbbra is egy természetes szám) bitmintájában mindazon helyeken 1-es áll, ahol a szűrővezérlővel beállított jellemzők kódjának bitmintájában is.

```
private int Code()
{
    int code = 0;
    int n = checkedListBoxFeatures.Items.Count - 1;
    foreach (int k in checkedListBoxFeatures.CheckedIndices)
    {
        code |= (1 << n - k);
    }
    return code;
}
```

Készítünk egy olyan felsorolót, amely rendre megadja az összes olyan természetes számot, amelynek bitmintájában az ellenőrződobozlistában beállított helyeken 1-esek állnak.

```
public IEnumerator GetEnumerator()
{
    int code = Code();
    for(int k = 0;
        k < Math.Pow(2, checkedListBoxFeatures.Items.Count); ++k)
    {
        if ((k & code) == code) yield return k;
    }
}
```

Ezt a felsorolót felhasználva szerkesztjük meg a features-re vonatkozó szűrést.

```
private void checkedListBoxFeatures_SelectedValueChanged(...)
{
    string str = "";
    if (Code() != 0)
    {
        foreach (object code in this)
        {
            str += string.Format(" OR features = '{0}'", (int)code);
        }
        str = "(" + str.Substring(3) + ")";
    }
    filter[(int)filterCriteria.features] = str;
    ApartmentFilterEvent(sender, e);
}
```

Szükség lehet még az összes szűrés törlésére. Ezt a Clear gombbal kezdeményezhetjük. Ilyenkor nemcsak a filter tömböt kell kiüríteni, hanem az összes szűrészi szempontot megadó vezérlőt is alapállásba kell hozni.

```
private void buttonClear_Click(...)
{
    for (int i = 0; i <= (int)(filterCriteria.features); ++i)
    {
        filter[i] = "";
    }

    textBoxPriceHigh.Text = "";
    textBoxPriceLow.Text = "";
    comboBoxCity.SelectedValue = 0;
    numericUpDownBeds.Value = 0;
    for (int i=0; i<checkedListBoxFeatures.Items.Count; ++i)
        checkedListBoxFeatures.SetItemChecked(i, false);

    ApartmentFilterEvent(sender, e);
    PriceFilterEvent(sender, e);
}
```

A kezdeti állapot beállítására a konstruktorban kerül sor.

```
public FilterControlForm()
{
    InitializeComponent();

    cityTableAdapter.Fill(apartmentsDataSet.city);
    apartmentsDataSet.city.Rows.Add(0, "");
    object[] list = { "looking sea", "looking green",
                    "terrace", "air condition" };
    checkedListBoxFeatures.Items.AddRange(list);
    buttonClear_Click(buttonClick, new EventArgs());
}
```

Végül térjünk vissza az eredeti projekthez, az apartmanokat és azok árait megjelenítő és karbantartó kapcsolt táblás űrlaphoz! Helyezzük el ennek felhasználói felületén a filtercontrol vezérlőt (ToolBox). Élesítsük ennek mindkét egyedi eseményét!

```
private void filterControlForm1_ApartmentFilterEvent(...)
{
    apartmentBindingSource.Filter =
        filterControlForm1.ApartmentFilter;
}

private void filterControlForm1_PriceFilterEvent(...)
{
    priceBindingSource.Filter = filterControlForm1.PriceFilter;
}
```