**МІНІСТЕРСТВО ОСВІТИ І НАУКИ УКРАЇНИ**

**ХАРКІВСЬКИЙ НАЦІОНАЛЬНИЙ ЕКОНОМІЧНИЙ УНІВЕРСИТЕТ**

**Методичні рекомендації**
**до виконання лабораторних робіт**
**з навчальної дисципліни**

# "ОРГАНІЗАЦІЯ БАЗ ДАНИХ І ЗНАНЬ (ADO.NET)"

**для студентів напряму підготовки**
**"Комп'ютерні науки"**
**денної форми навчання**

**Харків. Вид. ХНЕУ, 2010**

Затверджено на засіданні кафедри інформаційних систем.
Протокол № 3 від 02.12.2009 р.

М54 Методичні рекомендації до виконання лабораторних робіт з навчальної дисципліни "Організація баз даних і знань (ADO.NET)" для студентів напряму підготовки "Комп'ютерні науки" денної форми навчання / укл. М. Ю. Лосєв, О. В. Тарасов, В. В. Федько. – Харків : Вид. ХНЕУ, 2010. – 88 с. (Укр. мов.)

Наведено опис лабораторних робіт, спрямованих на формування вмінь та навичок використання новітніх технологій взаємодії з базами даних. Виконання робіт сприятиме підвищенню компетенції в галузі розроблення багаторівневих застосувань.

Рекомендовано для студентів напряму підготовки "Комп'ютерні науки" денної форми навчання.

## Загальні положення

Методичні рекомендації призначені для виконання лабораторних робіт з навчальної дисципліни "Організація баз даних і знань (ADO.NET)".

Перед виконанням кожної роботи необхідно вивчити відповідний лекційний матеріал і звернути особливу увагу на загальні положення, які передують опису лабораторних завдань.

Наведені приклади програм слід розглядати лише як один із можливих варіантів розв’язання задачі.

Методичні рекомендації містять опис восьми лабораторних робіт за третім модулем. Кожен розділ, який відповідає окремій лабораторній роботі, складається з таких підрозділів:

мета роботи й вимоги до теоретичної та практичної підготовки, що необхідна для виконання лабораторної роботи;

рекомендації щодо підготовки до виконання лабораторної роботи, основні теоретичні відомості, необхідні для її виконання;

суть роботи – загальна постановка завдання до лабораторної роботи (необов'язково);

варіанти завдань і контрольні запитання.

Для проведення всіх лабораторних робіт використовується єдина конфігурація програмно-апаратних засобів: персональний комп’ютер типу IBM-PC з процесором не нижче Pentium III, операційна система Windows XP або Windows 7, середовище візуальної розробки програм Microsoft Visual Studio .NET.

Під час виконання лабораторних робіт студент повинен продемонструвати:

творчий, індивідуальний підхід до розробки проектів (програмного коду та інтерфейсу користувача);

кваліфіковане використання існуючого програмного забезпечення;

навички програмування мовою високого рівня C#.

Студент повинен вміти перетворити свою програму в програмний продукт, використовувати якісний аналіз програми, виконувати оцінку отриманих результатів. Значна увага звертається на пояснення до окремих фрагментів програми.

***Типовий порядок виконання роботи*:**

уважно ознайомитися з методичними рекомендаціями до конкретної лабораторної роботи (теоретичними відомостями, прикладами, формулюванням завдань);

створити заготівлю проекту, скористатися для цього майстром створення застосування Microsoft Visual Studio;

заповнити отриману заготівлю проекту конкретним змістом відповідно до запропонованого завдання;

усунути всі помилки, що виникли на етапі компіляції початкового тексту програми;

виконати програму в покроковому режимі;

підсумковий запуск застосування виконати за допомогою виконуваного модуля;

відповісти на контрольні запитання;

виконати експериментальну частину роботи згідно з завданням;

оформити звіт і здати викладачеві.

**Звіт з лабораторної роботи повинен містити:**

1. Титульний аркуш:

назва дисципліни;

тема лабораторної роботи;

дата виконання роботи;

П.І.Б. студента, курс, номер групи;

П.І.Б., посада викладача.

2. Аркуш змісту (нумерований перелік назв пунктів роботи згідно з п. 3 із зазначенням номерів сторінок).

3. Опис виконаних завдань:

а) умова завдання;

б) опис архітектури програми (склад, структура класів, зв‘язки між ними, алгоритми тощо);

в) вихідний код програми;

г) приклади результатів роботи програми на тестових вихідних даних;

4. Висновки з лабораторної роботи з урахуванням усіх завдань, що були виконані:

аналіз отриманих результатів за кожним пунктом завдання;

аналіз результатів тестування програм;

ступінь відповідності розроблених программ постановці завдання;

інша інформація.

**Вимоги до оформлення звіту:**

1. Звіт з лабораторної роботи виконується друкованим способом на аркушах формату А4 відповідно до державних стандартів.

2. Аркуші звіту з’єднуються скріпками або іншим загальноприйнятим способом.

## Лабораторна робота 1. Розробка програм створення і управління з'єднанням з джерелом даних

**Мета лабораторної роботи:**

1. Отримання практичних навичок створення з'єднань із джерелом даних.

2. Набуття вмінь та навичок використання класів провайдерів даних ADO.NET, контейнерів стандартної бібліотеки .Net Framework.

3. Удосконалення навичок роботи з інтегрованим середовищем розробки Visual C# і довідковою системою Microsoft Developer Network (MSDN).

**Перед виконанням лабораторної роботи студент повинен знати:**

1. Основи використання бібліотеки Windows Forms.

2. Організацію системи створення й керування з'єднанням в ADO.NET.

3. Принципи обробки виключень в C#-програмі.

4. Структуру контейнерних класів стандартної бібліотеки .Net Framework і принципи їхнього використання.

**Після виконання лабораторної роботи студент повинен вміти:**

1. Самостійно розробляти прості C#-програми з графічним інтерфейсом користувача для створення з'єднання з базою даних і керування пулінгом.

2. Використовувати основні бібліотеки .Net Framework під час розробки програм.

### Основні відомості про підключення до джерела даних

Для того щоб почати працювати із джерелом даних, необхідно до нього підключитися. Оскільки підключені об'єкти мають працювати безпосередньо з базою даних, вони зазвичай реалізують код,

характерний для конкретної СУБД. Автономні ж об'єкти за своєю сутністю не повинні знати про базу даних. Тому логічно припустити, що вони можуть спільно використовуватися різними базами. Виявляється, більшість підключених об'єктів реалізовано в рамках того, що називається постачальниками даних.

Підключені об'єкти розділяються в ADO.NET за конкретними реалізаціями для різних СУБД. Наприклад, для підключення до бази даних Microsoft SQL Server є спеціальний клас SqlConnection. Насправді всі класи, що належать до SQL Server, перебувають у тому самому просторі імен System.Data.SqlClient. Аналогічно, всі класи, що належать до Oracle, знаходяться у просторі імен System.Data.OracleClient. Ці окремі реалізації для конкретних СУБД називаються постачальниками даних .NET (.NET data provider).

Постачальника для конкретного джерела даних в ADO.NET можна визначити як набір класів у просторі імен, що створені спеціально для роботи з конкретним джерелом даних. Таким чином, постачальники даних є зв'язуванням між застосуванням і джерелом даних. Причому як застосування може виступати будь-який компонент, XML Web-служба або звичайна програма.

Існують такі постачальники даних:

SQL Server .NET Data Provider – провайдер, що призначений для роботи з базами даних Microsoft SQL Server;

OLE DB .NET Data Provider – провайдер, що призначений для роботи із джерелами даних SQL Server, Oracle, Access;

ODBC .NET Data Provider – провайдер, що забезпечує доступ до джерел даних через їхнього ODBC-драйвера;

ORACLE .NET Data Provider – провайдер, що забезпечує доступ до баз даних Oracle.

За згодою постачальники, що входять до складу .NET Framework, містяться у власному просторі імен, що перебуває в просторі імен System. Data. У табл. 1.1 перераховано простори імен, що входять до складу .NET Framework.

Будь-яка Windows-машина може містити кілька інстальованих постачальників даних. В .NET Framework у файлі Machine.Config є розділ DbProviderFactories. У ньому можна визначити різних постачальників даних, до яких є доступ.

Таблиця 1.1

**Простори імен**

| Джерело даних | Простір імен постачальників |
|---|---|
| Microsoft SQL Server 7.0 і вище | System. DataSqlClient |
| Oracle 8.1. 6 і вище | System. DataOracleClient |
| Підтримка SqlXml в SQL Server | System. DataSqlXml |
| Будь-яке джерело даних ODBC | System. DataODBC |
| Будь-яке джерело даних OleDb | System. DataOleDb |

ADO.NET надає основні й допоміжні класи, які дозволяють керувати доступом до даних. Перелічимо основні класи, які містять провайдери даних. При цьому будемо використовувати спільні класи для всіх провайдерів:

**Connection** – застосовується для створення з'єднання із джерелом даних. Якщо використовується SQL Data Provider, цей клас називається SqlConnection**,** якщо ж OLE DB .NET Data Provider – цей клас називається OleDbConnection**;**

**Command** – призначається для виконання команд джерелом даних (SqlCommand й OleDbCommand відповідно до провайдера);

**DataReader** – використовується для читання даних з їхньою однобічною вибіркою;

**DataAdapter** – вживається для читання даних й зберігання змін, що зроблені в них.

Допоміжні класи розглядатимуться в процесі виконання таких лабораторних робіт.

Об'єкт підключення до бази даних можна одержати, створюючи нові екземпляри відповідних класів підключення. Наприклад:

**SqlConnection connect = new SqlConnection ();**

До викладеного вище додатково потрібно вказати об'єкт підключення (до якої бази даних треба під’єднатися), які права доступу використати тощо. Ці параметри можна задати за допомогою властивості ConnectionString об'єкта SqlConnection:

```
SqlConnection connect = new SqlConnection();
string connectionString =
  "Data Source=(local)/Initial Catalog=Rtest;Integrated
Security=SSPI";
connect.ConnectionString = connectionString;
```

Цей код готує об'єкт підключення до SQL Server до його відкриття на локальній машині за допомогою Windows-аутентифікації й до підключення до бази даних з ім'ям **Rtest**.

Зараз, маючи підготовлений у такий спосіб об'єкт для відкриття підключення до зазначеної бази даних, потрібно лише викликати метод Open().

```
SqlConnection connect = new SqlConnection(
"Server=(local);Database=Rtest;Integrated Security=SSPI");
 try
{
connect.Open ();
if (connect.State == ConnectionState.Open)
{
Console.WriteLine("Підключення успішно відкрите");
} }
catch (Exception) {
if (connect.State != ConnectionState.Open)
{
Console.WriteLine("Неможливо відкрити підключення");
} }
Finally {
if (connect.State == ConnectionState.Open)
connect.Close();        // Метод Close() закриває з'єднання
}
```

У рядках підключення можна зазначити множину різних параметрів. Для успішного встановлення підключення до бази даних кожний із цих параметрів потрібно правильно назвати і вказати для нього значення. Але якщо відсутній легкий інтуїтивний спосіб конструювання рядка підключення, то не тільки важко запам'ятати точне ім'я кожного параметра, але й легко пропустити різні можливості конфігурування для об'єктів підключення кожного постачальника даних.

В ADO.NET ця проблема вирішується за допомогою класу DbConnectionStringBuilder. Об'єкт DbConnectionStringBuilder суворо типізує різні значення, що складають рядок підключення. Він дозволяє уникнути тривіальних програмістських помилок, а також полегшує керування інформацією в рядку підключення.

Кожний постачальник даних повинен містити клас, що є спадковим від DbConnectionStringBuilder, полегшуючи створення рядка підключення. Клас OracleClient містить клас OracleConnection StringBuilder, а SqlClient – клас SqlConnectionStringBuilder. Розглянемо цей принцип на прикладі класу SqlConnectionStringBuilder.

```
static void Main(string[ ] args) {
SqlConnectionStringBuilder conBuilder = new
SqlConnectionStringBuilder ();
conBuilder.DataSource = "(local)";
conBuilder.InitialCatalog = "Rtest";
conBuilder.IntegratedSecurity = true;
using (SqlConnection connect =
new SqlConnection(conBuilder.ToString ())) {
try {
connect.Open();
if (connect.State == ConnectionState.Open)
{
Console.WriteLine ("Підключення успішно відкрите "-); "
Console.WriteLine("Використано рядок підключення: "
 +  connect.ConnectionString); } }
catch (Exception) {
if (connect.State != ConnectionState.Open) {
Console.WriteLine("Неможливо відкрити підключення ");
Console.WriteLine("Використано рядок підключення: "
+ connect.ConnectionString); } } }
//Автоматичний виклик звільнення підключення гарантує його закриття
Console.WriteLine ("Для продовження натисніть будь-яку клавішу");
Console.Read(); }
```

Рядок підключення до бази даних може розташовуватися в конфігураційному файлі app.config застосування, а саме в елементі **<connectionStrings>.**

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>
  <configSections>
  </configSections>
  <connectionStrings>
```

```
    <add name="lab1"
connectionString= "Provider=Microsoft.Jet.OLEDB.4.0;
Data Source=|DataDirectory|\db1.mdb"
        providerName="System.Data.OleDb" />
  </connectionStrings>
</configuration>
```

Зберігання такої інформації у файлі конфігурації вважається зручним, оскільки рядок з'єднання відсутній у коді програми й цей файл можна зашифрувати. Для підключення до бази даних у такий спосіб необхідно додати посилання на компонент System.Configuration (рис. 1.1).

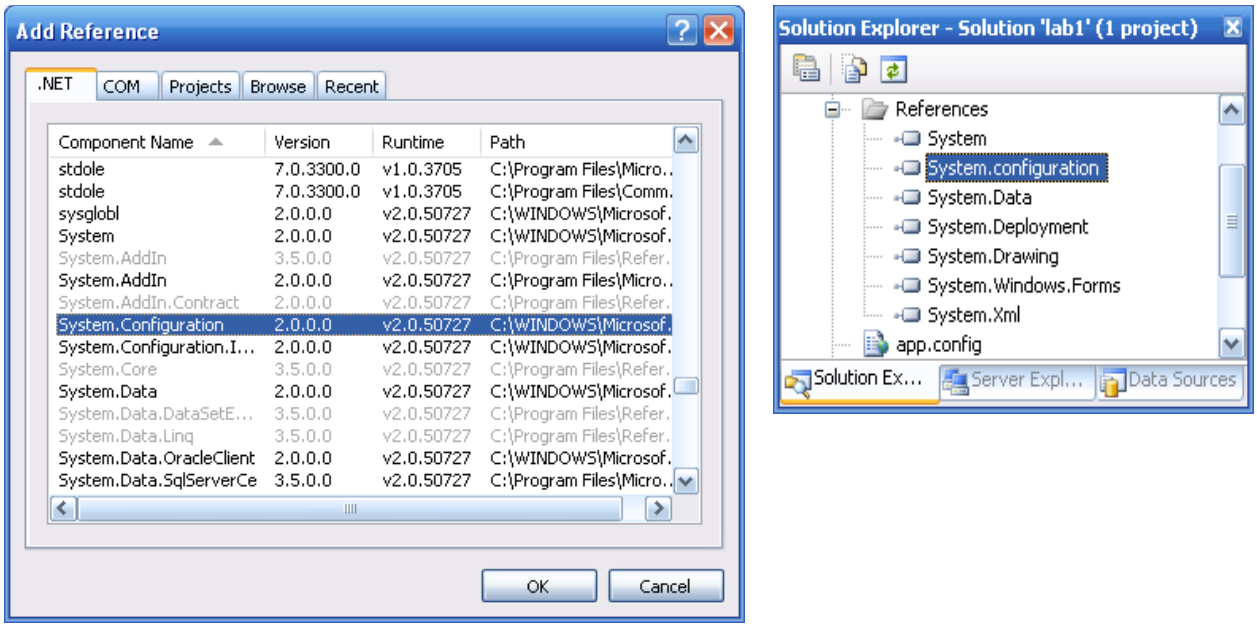


Рис. 1.1. **Додавання посилання на компонент System.Configuration**

Рядок підключення можна взяти з конфігураційного файла за допомогою класу ConfigurationManager і його властивості ConnectionStrings. Нижче наведено код з використанням компонента connectionToolStrip–MenuItem_Click.

```
private void connectionToolStripMenuItem_Click(object sender,
EventArgs e)
    {
       OleDbConnection connection = new OleDbConnection();
       string str =
ConfigurationManager.ConnectionStrings["lab1"].ToString();
```

```
            connection.ConnectionString = str;
            try
            {
               connection.Open();
            }
            catch (Exception ex)
            {
               MessageBox.Show(
"Помилка під час створення підключення\n" + ex.ToString());
            }
            connection.Close();
         }
```

В .NET кожний постачальник даних пропонує спеціальний клас System.Data.Common.DbProviderFactory. Для одержання типу, що є похідним від DbProviderFactory і відповідним для постачальника даних, простір імен System.Data.Common пропонує тип класу DbProviderFactories. Методом GetFactory() можна одержати унікальний об'єкт цього класу для заданого постачальника даних. Нижче наведено приклад підключення за допомогою зазначених класів й об'єкта OpenFileDialog.

```
      private void dBProviderFactoryToolStripMenuItem_Click(object
sender, EventArgs e)
      {
         OpenFileDialog f = new OpenFileDialog();
         f.Filter = "db files (*.mdb)|*.mdb|All files (*.*)|*.*";
         if (f.ShowDialog() == DialogResult.OK)
         {
            DbConnection connection = null;
            DbProviderFactory factory =
DbProviderFactories.GetFactory("System.Data.OleDb");
//Одержання джерела постачальника даних SQL
//             DbProviderFactory sqlfactory =
//DbProviderFactories.GetFactory("System.Data.SqlClient");
//Одержання джерела постачальника даних Oracle
//             DbProviderFactory sqlfactory =
```

```
//DbProviderFactories.GetFactory("System.Data.OracleClient");
            connection = factory.CreateConnection();
            connection.ConnectionString =
"Provider=Microsoft.Jet.OLEDB.4.0;Data Source=" + f.FileName;
            connection.Open();
            MessageBox.Show(connection.State.ToString());
            connection.Close();
          }
      }
```

За допомогою методу GetFactoryClasses() класу DbProviderFactories можна одержати список провайдерів, що встановлені на комп'ютері. Нижче наведено приклад розв'язання такої задачі.

```
private void button_Click(object sender, EventArgs e)
      {
         DataTable factoryClassesTable =
DbProviderFactories.GetFactoryClasses();
        string str = "";
         foreach (DataRow factoryClass in factoryClassesTable.Rows)
         {
         str += "Ім'я: " + factoryClass["Name"];
         str += "\nОпис: " + factoryClass["Description"];
        str += "\nІнваріантне ім'я: " + factoryClass["InvariantName"] +
"\n\n";
          }
          MessageBox.Show(str);
      }
```

Microsoft Visual Studio дозволяє автоматизувати процес створення з'єднання з базою даних. Щоб додати нове джерело даних, зверніться до меню Data діалогового вікна Microsoft Visual Studio (рис.1.2). Середовище Visual Studio дає можливість легко й швидко створювати класи, що розроблені спеціально для даних із БД, веб-сервісу або колекції об'єктів. Щоб запустити майстра Data Source Configuration, потрібно вибрати команду Add New Data Source.

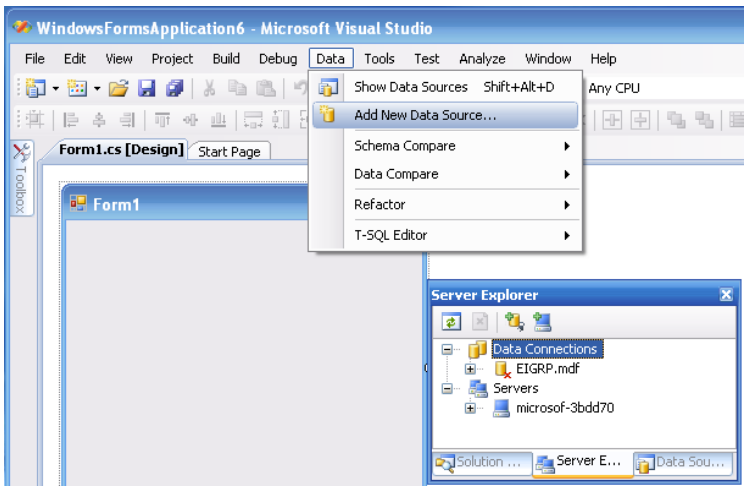


Рис. 1.2. **Додавання нового джерела даних в Visual Studio 2005**

За допомогою цього майстра можна швидко створити клас, що розрахований на зберігання множини таблиць з інформацією із БД. У першому вікні майстра, що зображено на рис. 1.3, з'явиться запитання про тип джерела даних. Виберіть елемент *Database* і клацніть кнопку Next.

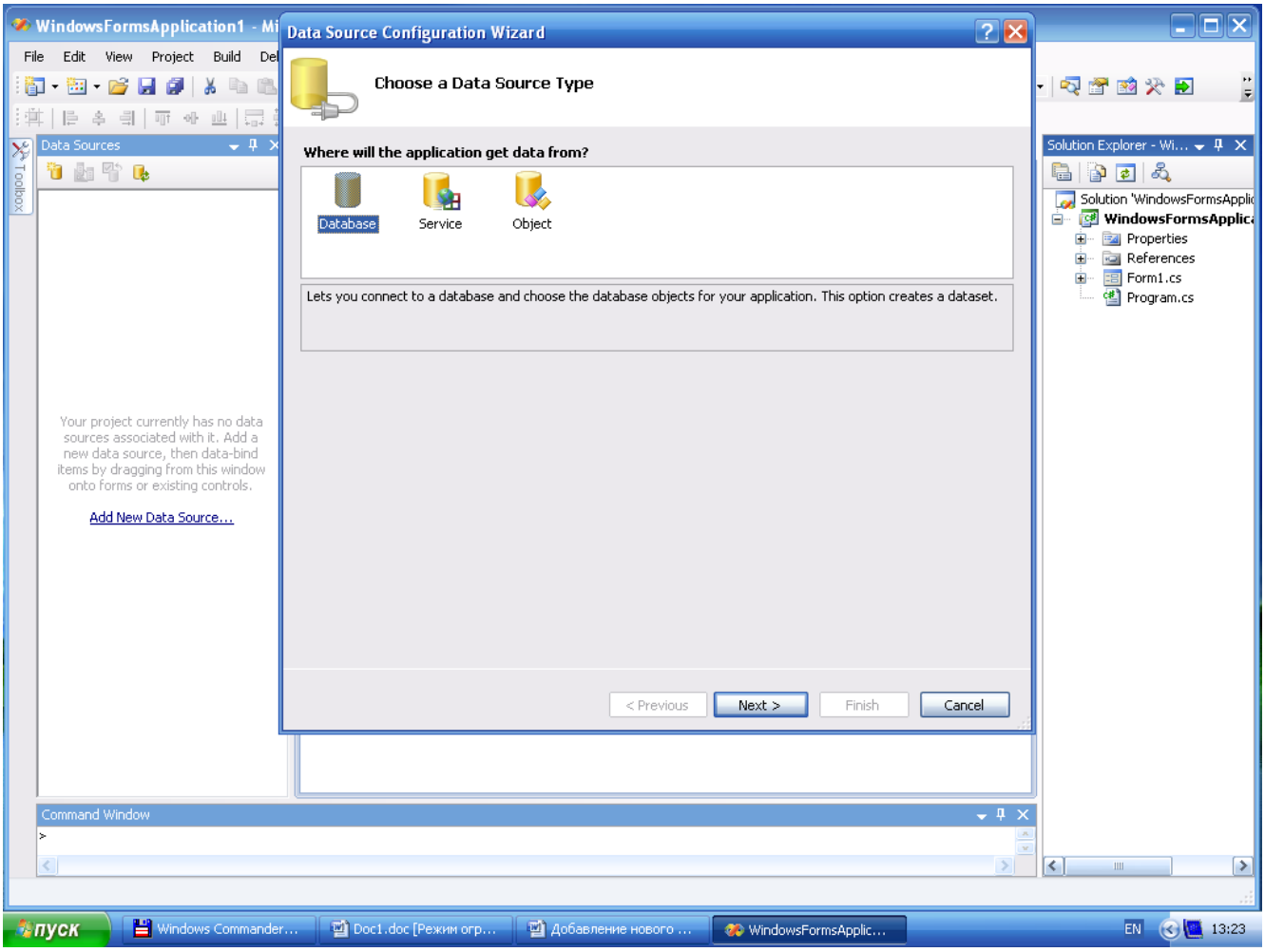


Рис. 1.3. **Вибір бази даних як джерела даних**

Коли Visual Studio уперше починає підказувати користувачеві, як виконати з'єднання, на екрані з'являється діалогове вікно Change Data Source (рис. 1.4). В ньому ставиться запитання про тип БД. Це вікно за замовчуванням звертається до БД Microsoft SQL Server, але в ньому є варіанти під'єднання до інших БД, таких як Microsoft SQL Server Database File, Microsoft Access Database File, Oracle Database і деяких інших.

Визначившись з типом БД, його можна встановити за замовчуванням і тим самим виключити появу згаданого діалогового вікна в майбутньому. Для цього потрібно встановити прапорець Always Use This Selection і клацнути кнопку ОК. Однак, навіть вибравши цю опцію, можна повернутися до того самого діалогового вікна й вибрати інший тип БД. Після натискання кнопки OK Visual Studio відобразить діалогове вікно, що спеціально створене для уточнення інформації про обраний тип БД. Передбачається, що користувач підключається до БД SQL Server.

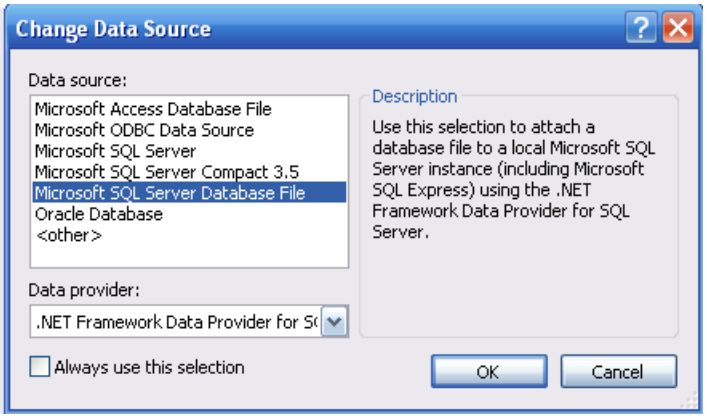


Рис. 1.4. **Вибір типу бази даних у вікні Change Data Source**

Щоб створити нове підключення, клацніть кнопку New Connection. Відкриється діалогове вікно, що підкаже, як підключитися до потрібної БД (рис. 1.5).

Як тільки буде обрано підключення до БД SQL Server, на екрані з'явиться діалогове вікно Add Connection (рис. 1.6). У ньому відображається інформація про місцезнаходження бази даних, про мандат на підключення до неї, а також про вихідний каталог, який необхідно використати.

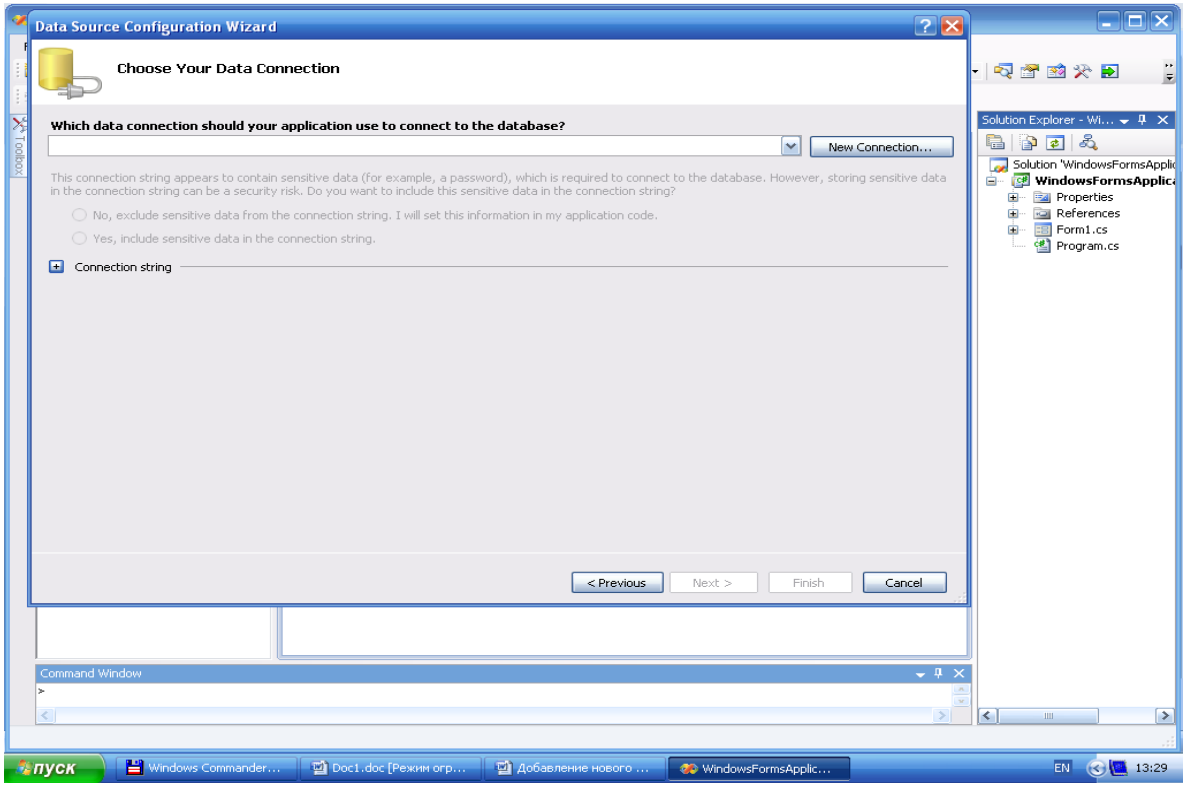


Рис. 1.5. **Вибір підключення до потрібної БД**

Набравши ***.\SQLExpress*** у полі Server Name цього вікна, вказують, що виконується підключення до платформи SQL Server Express, яка встановлена на поточному комп'ютері. Знак крапки позначає скорочення роздільника. Він розшифровується як "локальна машина". Можна було б указати ім'я комп'ютера або конструкцію *(local),* однак знак крапки швидше набирається й займає менше місця в коді. Знаком зворотної похилої риски ім'я машини відокремлюється від імені екземпляра SQL Server. У нашому разі це ім'я екземпляра SQL Server Express – SQLExpress.

Діалогове вікно Add Connection побудовано так, що підключення до БД SQL Server передбачається виконувати за допомогою засобів аутентифікації Windows. За замовчуванням в установках SQL Server 2005 встановлено єдиний параметр, що дає можливість підключитися, використовуючи його поточний мандат у Windows. Його використання позбавляє від необхідності вводити ім'я користувача й пароль.

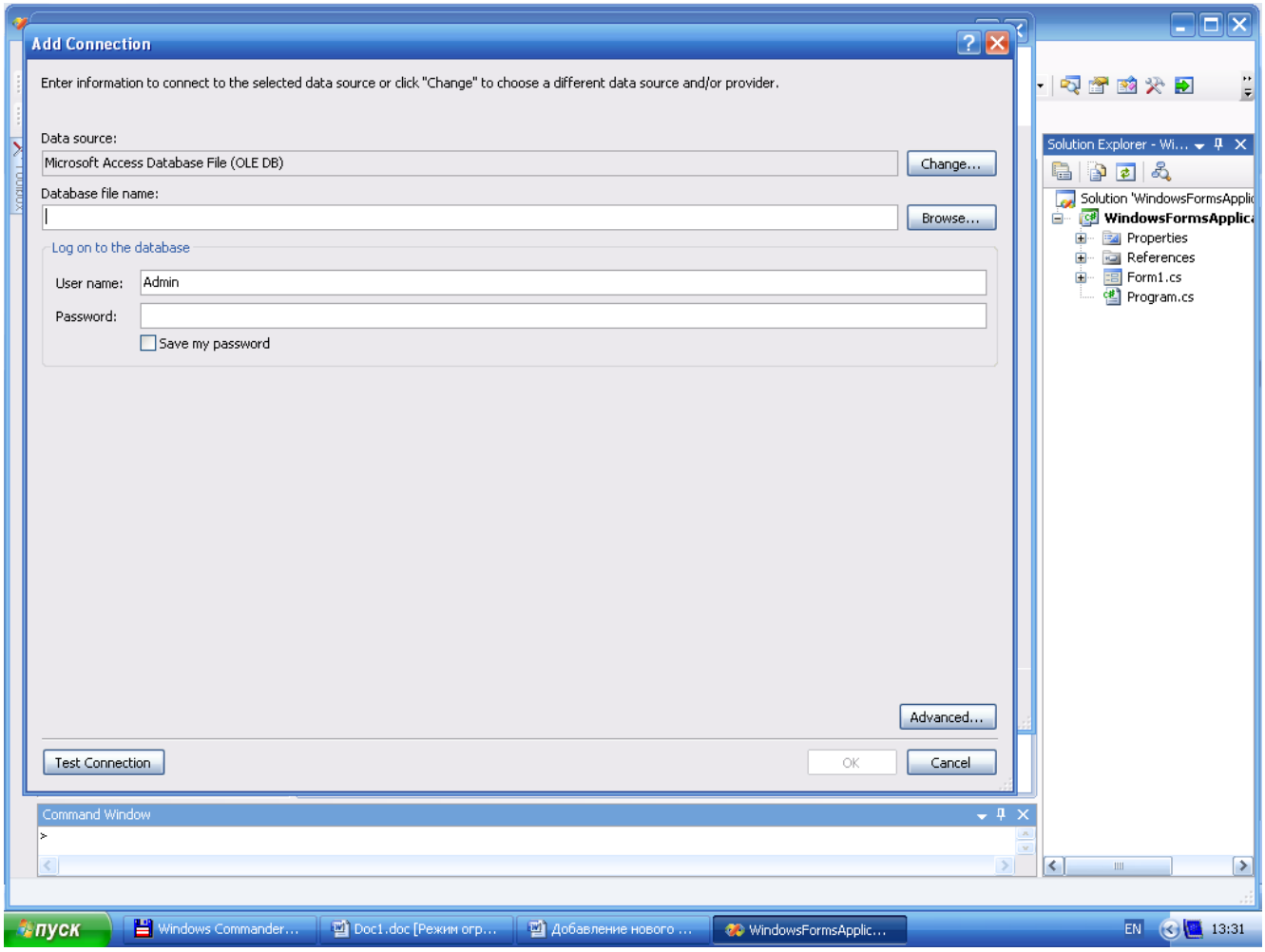


Рис. 1.6. **Введення інформації про базу даних**

Щоб увійти в систему іншим способом, який часто називають аутентифікацією SQL Server, виберіть перемикач Use SQL Server Authentication у вікні Add Connection, після чого введіть ім'я користувача й пароль.

За допомогою опцій у діалоговому вікні можна перевірити рядок підключення, клацнувши для цього кнопку зі знаком плюс, і видалити з неї такі уразливі дані, як, наприклад, інформацію про пароль (Рис.1.8).

Після того як буде вказано всю інформацію про з'єднання, майстер Data Source Configuration відобразить діалогове вікно, що подано на рис. 1.9. В ньому майстер поцікавиться, чи зберігати рядок підключення у файлі конфігурації застосування. І хоча в даному підході є багато за й проти, одна з ключових переваг полягає в тому, що файл конфігурації є простим способом відокремити інформацію про підключення від іншої частини коду.

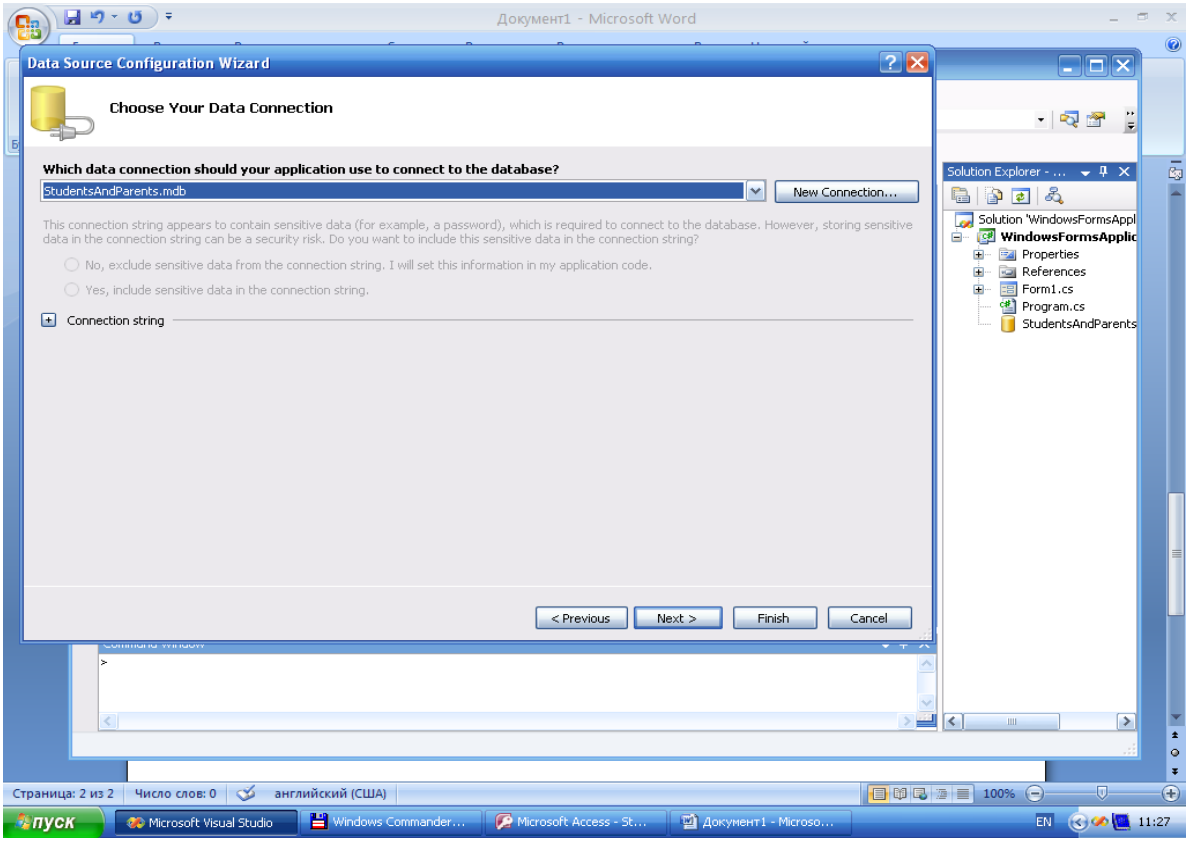


Рис. 1.7. **Результат створення підключення до бази даних**

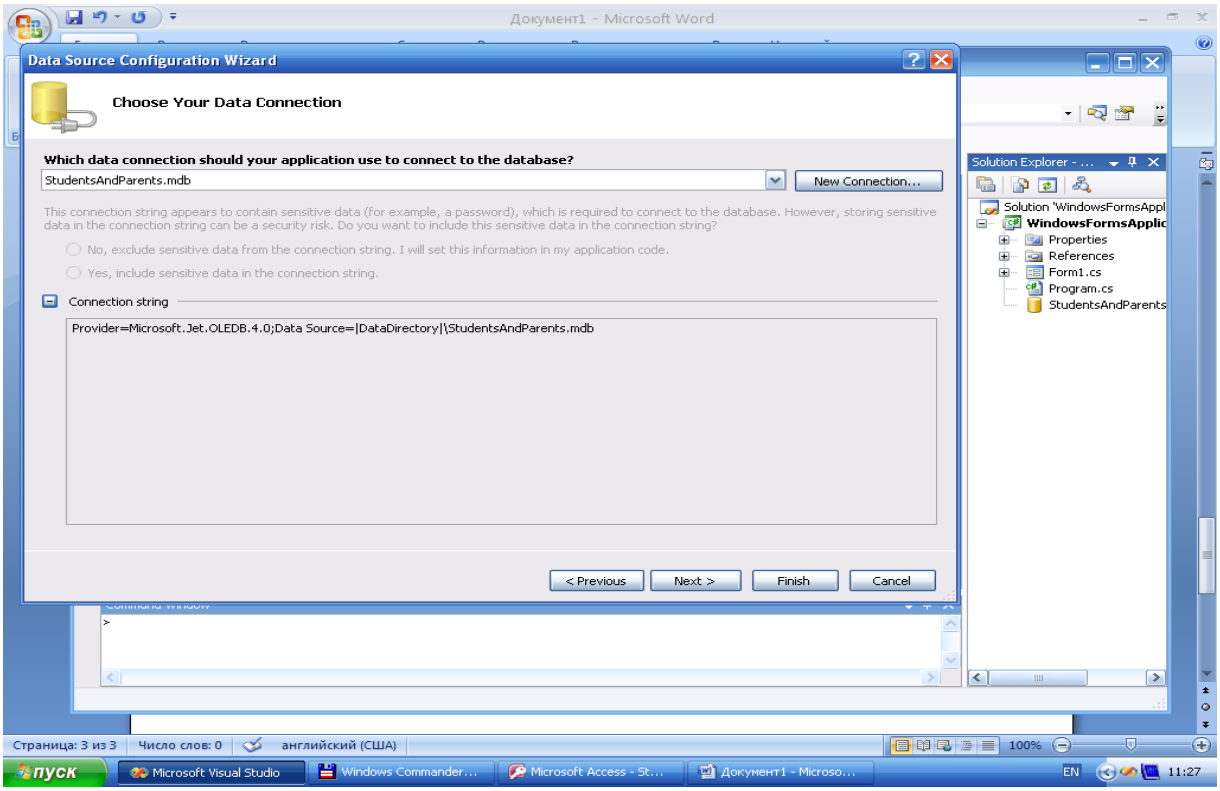


Рис.1.8. **Перевірка рядка підключення**

Компіляція рядка підключення в застосуванні має низку недоліків. Розміщення БД може змінитися після того, як застосування буде розгорнуто. Збереження ж рядка підключення в ресурсі, незалежному від коду застосування, робить саме застосування більш гнучким. Якщо в діалоговому вікні встановити прапорець Yes, save the connection string as, то середовище Visual Studio збереже рядок підключення й ім'я, що введе користувач, у файлі конфігурації застосування для проекту. Крім того, воно додасть логічні оператори в код проекту, щоб одержувати й використати цей рядок підключення для взаємодії з базою даних.

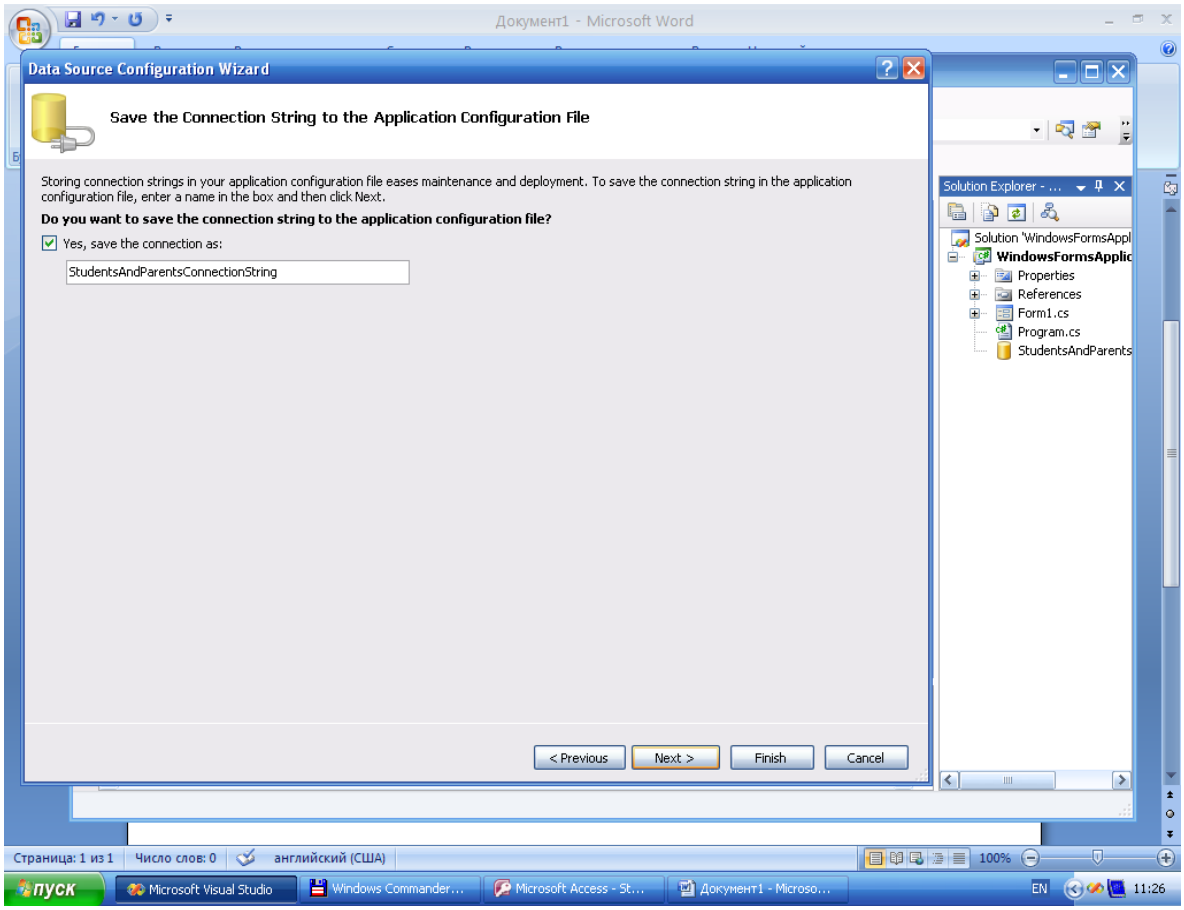


Рис. 1.9. **Збереження рядка підключення у файлі конфігурації застосування**

Кожний постачальник даних .NET із складу ADO.NET реалізує пул з'єднань. Коли знадобиться нове з'єднання, постачальник .NET переглядає передані реквізити (розміщення БД, ім'я користувача тощо) і шукає в пулі відкрите з'єднання з аналогічними параметрами підключення. Якщо з'єднання знайдено, постачальник передає його користувачу. У протилежному разі він створює й повертає нове з'єднання. Коли знищується об'єкт Connection, постачальник даних .NET не закриває реальне з'єднання

із БД. Він позначає об'єкт як знищений, але поміщає з'єднання в пул. Якщо протягом заданого періоду часу (за замовчуванням – 60 секунд) з'єднання не використовується повторно, постачальник даних .NET закриває його.

Пул з'єднань включено за замовчуванням. Що робити, якщо не треба поміщати з'єднання в пул? Клас OleDbConnection реалізує метод ReleaseConnectionPool, який можна використати разом з методом Collect об'єкта. Він здійснює глобальний збір сміття, щоб насправді закрити фізичне з'єднання з базою даних. Однак є більш витончений спосіб. У рядок підключення OLE DB треба додати такий атрибут:

**OLE DB Services= -4;**

При цьому постачальник даних OLE DB .NET відзначає, що з'єднання не потрібно поміщати в пул. Викликавши метод Close об'єкта OleDbConnection, по-справжньому закривають з'єднання з базою даних.

Якщо використовують об'єкт SqlConnection, у рядок підключення можна додати такий атрибут, який вказує постачальникові .NET, що з'єднання не потрібно поміщати в пул:

**Pooling=False; .**

У процесі створення застосування може виникнути ситуація, при якій не можна використати з'єднання із старого пулу. У цьому разі треба змінити рядок підключення так, щоб можна було створити пул, не вплинувши на застосування. Найпростіший спосіб полягає в додаванні одного пробілу до рядка підключення.

## Завдання для лабораторної роботи

1. Визначити доступних провайдерів, що встановлені на комп’ютері. Виконати підключення до бази даних з використанням цих провайдерів (SQL Server, Ole Db й ін.) з індикацією стану підключень.

2. Розробити застосування, що забезпечує підключення до бази даних з використанням класів DBConnectionStringBuilder, DBProviderFactory й елемента <connectionString> файла конфігурації.

3. Визначити економію часу при виконанні з'єднання з використанням пулу.

## Додаткові завдання

1. Передбачте з'єднання з книгою Excel за допомогою відповідної кнопки на формі *Рядки ConnectionString.* Рядок з'єднання має формат:

Provider=Microsoft.Jet.OLEDB.4.0;Data Source=C:\MyExcel.xls;
Extended Properties="Excel 8.0;HDR=Yes;IMEX=1";
де HDR=Yes  означає, що в першому рядку таблиці містяться назви стовпчиків, IMEX=1 означає, що всі дані таблиці сприймаються як текстові, у протилежному разі параметр не вказується.

2. Створити застосування, в якому підключення вибирається як у майстрі.

3. Створити застосування, в якому під час підключення провайдер вибирається автоматично за розширенням файла джерела даних.

4. *Створити застосування, що визначає, чи дійсно закриті з'єднання, чи воно просто поміщене в пул.

5. *Визначити, чи є щойно створене з'єднання новим чи використовується повторно з пулу з'єднань.

### Контрольні запитання

1. Провайдери даних в ADO.NET.
2. Способи підключення до бази даних.
3. Класи, специфічні для бази даних.
4. Параметри рядка з'єднань.
5. Стан з'єднання.
6. Пул з'єднань.
7. Параметри пулу з'єднань.

## Лабораторна робота 2. Розробка програм виконання операцій у з'єднаному середовищі

**Мета лабораторної роботи:**

1. Отримання практичних навичок із створення й запуску командних об'єктів.

2. Набуття вмінь та навичок використання класів провайдерів даних ADO.NET, контейнерів стандартної бібліотеки .Net Framework.

3. Удосконалення навичок роботи з інтегрованим середовищем розробки Visual C# і довідковою системою Microsoft Developer Network (MSDN).

**Перед виконанням лабораторної роботи студент повинен знати:**

1. Основи використання бібліотеки Windows Forms.
2. Організацію системи створення й керування з'єднанням в ADO.NET.
3. Принципи обробки виключень в C#-програмі.
4. Структуру контейнерних класів стандартної бібліотеки .Net Framework і принципи їхнього використання.

**Після виконання лабораторної роботи студент повинен вміти:**

1. Самостійно розробляти прості C#-програми з графічним інтерфейсом користувача для створення з'єднання з базою даних і керування обробкою інформації.
2. Використовувати основні бібліотеки .Net Framework під час розробки програм.

## Основні відомості про виконання операцій у з'єднаному середовищі

Після встановлення підключення до бази даних взаємодія з нею полягає у виконанні запиту. При цьому необхідно вказати, які дані потрібні застосуванню й одержати результати. Для виконання запиту використовується об'єкт команди. Узагальнений об'єкт команди в ADO.NET подано класом **DbCommand.**

Залежно від провайдера даних, що використовується, необхідно вибрати клас. Його об'єкт буде застосовуватися для передачі команд джерелу даних. Нижче подано список доступних класів:

**System.Data.SqlClient.SqlCommand** – призначений для роботи з SQL Server7.0 або вище;

**System.Data.OleDb.OleDbCommand** – призначений для роботи з базами даних на основі провайдера OLE DB;

**System.Data.OracleClient.OracleCommand** – використовується в провайдері даних Oracle.NET і призначений винятково для роботи із джерелами даних, що керуються системою Oracle;

**System.Data.Odbc.OdbcCommand** – використовується разом з ODBC .NET провайдерами даних і призначений для роботи із джерелами через ODBC-драйверів.

Нижче подано список властивостей, які знадобляться під час створення й роботи з командними об'єктами. Ці властивості дозволяють використовувати будь-якого провайдера даних:

**(Name)** – властивість містить ім'я командного об'єкта, за допомогою якого здійснюється доступ до нього в програмному коді. Вона доступна тільки з вікна властивостей командного об'єкта, якщо використовується візуальний дизайнер у Visual Studio. NET для додавання об'єкта за технологією Drag and Drop.

**Connection** – властивість, що передає командному об'єкту ім'я з'єднання, через яке він буде працювати із джерелом даних, тобто містить посилання на об'єкт Connection.

**CommandType** – властивість, яка може мати одне з трьох значень: **Text, StoredProcedure** або **TableDirect.** Значення **Text,** що установлюється за замовчуванням, визначає, що властивість **CommandText** буде містити звичайну команду SQL. Значення **StoredProcedure** вказує на те, що властивість **CommandText** містить ім'я збереженої процедури в базі даних. Значення **TableDirect** означає, що властивість **CommandText** зберігає ім'я таблиці або кілька імен таблиць бази даних, з яких будуть вибиратися дані після викликання методу **Execute.**

**CommandText** – властивість, що містить або SQL-вираз, або ім'я збереженої процедури, або імена таблиць.

**Parameters** – колекція, що призначена для зберігання параметрів.

**CommandTimeout** – властивість, в якій вказують час очікування результату при виконанні командного об'єкта. Час зазначають в секундах і за замовчуванням становить 30 секунд.

Існує кілька конструкторів, що дозволяють створювати об'єкти **Command.** Розглянемо приклади конструювання об'єктів на основі SQL Server .NET провайдера даних. Перший конструктор класу **SqlCommand** не має параметрів:

```
SqlConnection conn = new SqlConnection(connectionString);
SqlCommand myCommand = new SqlCommand();
string commandText = "SELECT * FROM Товари";
 myCommand.Connection = conn;
 myCommand.CommandText = commandText;
 // або Command.CommandText = "SELECT * FROM Товари";
```

Другий конструктор класу **SqlCommand** сприймає рядок підключення як параметр:

```
SqlConnection conn = new SqlConnection(connectionString);
 SqlCommand myCommand = new SqlCommand(commandText);
//SqlCommand myCommand =
// new SqlCommand("SELECT * FROM Товари");
 myCommand.Connection = conn;
```

Третій конструктор класу **SqlCommand** як параметри сприймає рядок підключення й текст команди:

```
SqlConnection conn = new SqlConnection(connectionString);
SqlCommand myCommand = new SqlCommand(commandText, conn);
// SqlCommand myCommand =
// new SqlCommand("SELECT * FROM Товари", conn);
```

У четвертому варіанті конструктора класу **SqlCommand** до параметрів додано об'єкт транзакції:

```
SqlConnection conn = new SqlConnection(connectionString);
commandText = "SELECT * FROM Товари";
SqlTransaction trans = conn.BeginTransaction();
SqlCommand myCommand = new SqlCommand(commandText, conn,
trans);
```

Об'єкт **Command** можна створити за допомогою методу **CreateCommand()** об'єкта **Connection:**

```
string comText = "SELECT * FROM Товари";
SqlConnection conn = new SqlConnection(connectionString);
conn.Open();
 SqlCommand myCommand;
 myCommand = conn.CreateCommand();
 myCommand.CommandText = comText;
conn.Close();
```

Після встановлення необхідних властивостей, можна запустити на виконання команду, що зберігається в об'єкті Command. Для цього використовують один із чотирьох методів:

**ExecuteScalar** – метод, що призначений для виконання команд, результатом яких є одне значення;

**ExecuteReader** – метод, який використовують для виконання команд, що повертають множину рядків;

**ExecuteNonQuery** – метод, що дозволяє виконати команду, яка обновляє базу даних або змінює її структуру. Вона повертає число змінених рядків;

**ExecuteXmlReader** – виконує команду, що повертає результат у вигляді XML. Він застосовується тільки в SqlCommand-об'єкті.

Використання цих методів розглянемо на прикладах.

Метод ExecuteScalar об'єкта Command застосовується для запитів, що повертають одне значення. Часто для цього використовуються агрегатні функції COUNT, MIN, MAX. Нижче наведено код застосування, що визначає кількість товарів у таблиці "Товари":

```
using System;
using System.Data.SqlClient;
namespace Example
{
  class Class1
  {
    [STAThread]
    static void Main(string[ ] args)
    {
      SqlConnection conn = new SqlConnection();
      conn.ConnectionString = @"Data Source=.\SQLEXPRESS;
AttachDbFilename=|DataDirectory|\ПродажТоварів.mdf;Integrated
Security=True; Connect Timeout=30;User Instance=True";
      conn.Open();
      SqlCommand myCommand = conn.CreateCommand();
      // SqlCommand myCommand = new SqlCommand();
      // myCommand.Connection = conn;
      myCommand.CommandText = "SELECT COUNT (*) FROM
Товари";
      string kilkistTovariv =
Convert.ToString(myCommand.ExecuteScalar());
      conn.Close();
      Console.WriteLine("Кількість товарів: " +
       kilkistTovariv);
    }
  }  }
```

Можна застосовувати цей метод кілька разів:

```
  myCommand.CommandText = "SELECT COUNT (*) FROM Товари";
  string kilkistTovariv =
Convert.ToString(myCommand.ExecuteScalar());
```

```
  myCommand.CommandText = "SELECT MAX (Ціна) FROM Товари";
  string MaxPrice = Convert.ToString(myCommand.ExecuteScalar());
  myCommand.CommandText = "SELECT MIN (Ціна) FROM Товари";
  string MinPrice = Convert.ToString(myCommand.ExecuteScalar());
  myCommand.CommandText = "SELECT AVG (Ціна) FROM Товари";
  string AvgPrice = Convert.ToString(myCommand.ExecuteScalar());
  conn.Close();
  Console.WriteLine("Кількість товарів: " + kilkistTovariv +
    "\nнайдорожчий товар, ціна: " + MaxPrice +
    "\nнайдешевший товар, ціна: " + MinPrice +
    "\nсередня ціна товарів: " + AvgPrice);
 }
```

Методом ExecuteNonQuery() зручно виконувати оновлення, додавання й видалення рядків у таблицях бази даних. У наведеному нижче фрагменті коду оновлюється прізвище в рядку, де код клієнта дорівнює 1, і здійснюється перевірка виконання оновлення даних:

```
        SqlConnection conn = new SqlConnection();
        conn.ConnectionString = @"Data Source=.\SQLEXPRESS;
AttachDbFilename=|DataDirectory|\ПродажТоварів.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True";
        conn.Open();
        SqlCommand myCommand = conn.CreateCommand();
        myCommand.CommandText = "UPDATE Клієнти
         SET Прізвище = 'Петренко'   WHERE КодКлієнта = 1";
        myCommand.ExecuteNonQuery();
        conn.Close();
     int zmineno = myCommand.ExecuteNonQuery();
     if (zmineno !=0)
     {
       Console.WriteLine ("Зміни внесено");
     }
     else
     {
       Console.WriteLine("Зміни не внесено !!! ");
     }
```

Метод ExecuteNonQuery застосовується також для виконання запитів, що належать до категорії DDL мови SQL. Мова визначення

даних (Data Definition Language, DDL) дозволяє створювати й змінювати структуру об'єктів бази даних, наприклад, створювати й видаляти таблиці. Основними операторами цієї мови є CREATE, ALTER, DROP. В результаті виконання запитів DDL не повертаються дані. Саме тому можна застосовувати метод ExecuteNonQuery. Наведемо приклад виконання таких операцій. Для цього у властивість CommandText запишемо новий текст команди, що створює в базі "ПродажТоварів" нову таблицю "ВідгукиКлієнтів":

**myCommand.CommandText = "CREATE TABLE** ВідгукиКлієнтів
**(КодВідгуку INT NOT NULL, КодКлієнта INT NOT NULL, ЗмістВідгуку VARCHAR(50)";**

При додаванні нового стовпчика "Дата" рядок Command Text повинен мати такий вигляд:

**myCommand.CommandText = "ALTER TABLE** ВідгукиКлієнтів **ADD Дата datetime)";**

Для видалення таблиці "ВідгукиКлієнтів" виконуємо команду, що містить такий рядок CommandText:

**myCommand.CommandText = "DROP TABLE** ВідгукиКлієнтів**";**

Часто необхідно вводити дані в процесі виконання програми, оскільки немає змоги заздалегідь вказати, які будуть значення змінних. Для розв’язання таких задач, які виникли ще на самому початку розробки мови SQL, запропоновано параметризовані запити. В них невідомі значення замінюються параметрами. Наприклад:

**myCommand.CommandText = "UPDATE Клієнти SET**
**Прізвище = @Prizvische WHERE КодКлієнта = @KlientID";**

Тут @ **Prizvische** (зверніть увагу, пишеться без лапок!) – параметр для невідомого значення прізвища, @**KlientID** – параметр для невідомого значення коду клієнта. Тепер можна прив'язувати параметри до тексту, що вводить користувач у компонентах форми:

```
//уводиться користувачем у поле txtBoxPrizvische:
    string Prizvische = Convert.ToString(this.txtBoxPrizvische.Text);
//уводиться користувачем у поле txtBoxKlientID:
    int KlientID = int.Parse(this.txtBoxKlientID.Text);
    conn = new SqlConnection();
    conn.ConnectionString = @"Data Source=.\SQLEXPRESS;
```

```
AttachDbFilename=|DataDirectory|\ПродажТоварів.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True";
    conn.Open();
    SqlCommand myCommand = conn.CreateCommand();
    myCommand.CommandText = "UPDATE Клієнти
     SET Прізвище = @Prizvische WHERE КодКлієнта = @KlientID";
//Додаємо параметр @ Prizvische в колекцію параметрів
//об'єкта myCommand
    myCommand.Parameters.Add("@Prizvische",
     SqlDbType.NVarChar, 50);
//Установлюємо значення параметра @ Prizvische
//яке дорівнює значенню змінної Prizvische
    myCommand.Parameters["@Prizvische"].Value = Prizvische;
//Додаємо параметр @KlientID в колекцію параметрів
//об'єкта myCommand
    myCommand.Parameters.Add("@KlientID", SqlDbType.Int, 4);
//Установлюємо значення параметра @KlientID
// яке дорівнює значенню змінної KlientID
    myCommand.Parameters["@KlientID "].Value = KlientID;
    int zmineno = myCommand.ExecuteNonQuery();
```

Дуже часто виникає необхідність виконувати запити, які повертають не одне значення, а запис із бази даних або цілу множину записів. Доступ до множини записів здійснюється за допомогою спеціального курсору. Він дозволяє переміщатися послідовно обраними записами й одержувати дані з кожного запису. Цей курсор реалізовано за допомогою спеціального об'єкта **DataReader.** Для зчитування записів необхідно створити об'єкт **DataReader** і використовувати метод **ExecuteReader** командного об'єкта. Для переміщення записами вживають метод **Read,** що дозволяє читати наступний запис. У разі, коли записів більше немає, метод **Read** повертає значення **false.**

Розглянемо приклад, у якому відбувається зчитування інформації з таблиці **Products:**

```
//створюємо з'єднання
    SqlConnection conn=new SqlConnection() ;
//встановлюємо рядок зв'язку
 conn.ConnectionString=@"Data Source=.\SQLEXPRESS;
```

```
AttachDbFilename=|DataDirectory|\ПродажТоварів.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True";
//створення командного об'єкта
SqlCommand conn =
    new SqlCommand("SELECT * FROM Клієнти", conn);
    connSql.Open();
//виконуємо ExecuteReader
        SqlDataReader reader=conn.ExecuteReader();
//друкуємо таблицю
    while (reader.Read()) {
    Console.WriteLine("{0}\t{l}\t{2}",
            reader["KlientID"] .ToString() ,
            reader ["Prizvische"].ToString(),
//закриваємо DataReader
        reader.Close();
        conn.Close ();
```

Поточне з'єднання з базою недоступне іншим методам, поки об'єкт DataReader залишається відкритим. Тому його необхідно закрити відразу після закінчення роботи, щоб звільнити ресурси.

## Завдання для лабораторної роботи

1. За допомогою командних об'єктів створити таблицю в базі даних, яка задана викладачем.

2. Забезпечити відображення даних таблиці на формі в наочному вигляді.

3. Створити команди оновлення, додавання й видалення даних з таблиці. Забезпечити виконання цих операцій з використанням параметрів.

## Додаткові завдання

1.* Визначити кількість оброблених записів у кожній таблиці під час виконання пакетного запиту.

2.* Застосування виконує запит до бази даних, результати якого можуть отримуватися із затримкою. Необхідно передбачити забезпечення виконання яких-небудь дій на головній формі застосування протягом цієї затримки.

### Контрольні запитання

1. Створення командних об'єктів.
2. Властивості командних об'єктів.
3. Запуск командних об'єктів.
4. Використання методу ExecuteNonQuery.
5. Використання параметризованих запитів у командних об'єктах.

## Лабораторна робота 3. Розробка програм виконання операцій у роз'єднаному середовищі

**Мета лабораторної роботи:**

1. Отримання практичних навичок створення наборів даних.
2. Набуття вмінь та навичок використання класів адаптерів даних ADO.NET.
3. Отримання практичних навичок відображення даних у застосуваннях і виконання операцій ведення бази даних.
4. Удосконалення навичок роботи з інтегрованим середовищем розробки Visual C# і довідковою системою Microsoft Developer Network (MSDN).

**Перед виконанням лабораторної роботи студент повинен знати:**

1. Основи використання бібліотеки Windows Forms.
2. Організацію системи створення й керування з'єднанням в ADO.NET в роз'єднаному середовищі.
3. Принципи обробки виключень в C#-програмі.

**Після виконання лабораторної роботи студент повинен уміти:**

1. Самостійно розробляти прості C#-програми з графічним інтерфейсом користувача для створення з'єднання з базою даних і керування обробкою інформації.
2. Використовувати основні бібліотеки .Net Framework під час розробки програм.

## Основні відомості про виконання операцій в роз'єднаному середовищі

Роз’єднаний режим роботи застосування припускає використання підключення до бази даних тільки для одержання необхідної інформації. В процесі обробки даних з'єднання між застосуванням і базою припиняється. Будь-які зміни в отриманих даних не позначаються на вмісті бази даних. Для забезпечення такої можливості в ADO.NET передбачено кілька класів.

Клас DataSet розроблено як автономне сховище даних. Він складається з набору таблиць, кожна з яких містить множину рядків і стовпчиків даних. У класі DataSet можна визначити зв'язки між таблицями.

У табл. 3.1 наведено властивості, які становлять структуру даних DataSet: Tables, Relations й ExtendedProperties.

Таблиця 3.1

**Клас DataSet і його основні складові**

| Властивість | Опис |
|---|---|
| Tables | DataSet. Tables – об'єкт типу System. Data. DataTableCollection, що може містити об'єкти System. Data. DataTable. Кожний DataTable містить набір табличних даних. У свою чергу, у кожного DataTable є колекції з іменами Columns й Rows, які можуть містити відповідно нуль або більше об'єктів DataColumn або DataRow |
| Relations | Властивість DataSet.Relations – об'єкт типу System. Data.DataRelationCollection, що може містити об'єкти System.Data.DataRelation. Об'єкт DataRelation визначає відношення предок/нащадок між двома об'єктами DataTable на основі значень зовнішніх ключів |
| Extended Properties | DataSet. ExtendedProperties – об'єкт типу System. Data .PropertyCollection, що може містити властивості, які визначені користувачем. Колекцію ExtendedProperties можна використовувати для зберігання користувальницьких даних, що належать до DataSet, таких як час створення об'єкта |

Існує два конструктори, що дозволяють створювати об'єкти типу DataSet. Перший конструктор сприймає як параметр рядок, що містить ім'я створюваного об'єкта. Ім'я для об'єкта типу DataSet необхідне, щоб гарантувати наявність його в документі XML для елемента верхнього рівня в XML-схемі.

**DataSet data=new DataSet("Northwind");**

**Console.WriteLine("Об'єкт {0} створено",data.DataSetName);**

Другий конструктор дозволяє створити об'єкт DataSet без параметрів, наприклад:

**DataSet data=new DataSet();**

У цьому разі новий об'єкт матиме ім'я NewDataSet.

Клас DataTable реалізує всі стандартні інтерфейси, які реалізовані в DataSet. Тому він може виконувати ті ж функції, що й DataSet. Клас DataTable містить колекції Columns, Rows і Constraints. Колекції Columns й Constraints разом утворюють схему для DataTable, а колекція Rows містить дані. Ці три колекції описано в табл. 3.2.

Таблиця 3.2

**Колекції класу DataTable**

| Властивість | Опис |
|---|---|
| Columns | Екземпляр класу System. Data. DataColumnCollection, що може містити об'єкти DataColumn. Об'єкти DataColumn визначають такі властивості кожного стовпчика з DataTable: ім'я, тип збережених даних і первинний ключ |
| Rows | Екземпляр класу System. Data. DataRowCollection, що складається з об'єктів DataRow. Об'єкти DataRow містять реальні дані таблиці DataTable, що визначені в колекції DataTable. Columns. Кожний об’єкт DataRow має один елемент для кожного DataColumn з колекції Columns |
| Constraints | Екземпляр класу System.Data.ConstraintCollection, що складається з об'єктів System. Data.ForeignKeyConstrant або об'єктів System. Data. UniqueConstraint. Перші визначають дію, яка виконується над стовпчиком у відношенні первинний ключ – зовнішній ключ у разі зміни або видалення рядка, а другі використовуються для забезпечення унікальності значень в даному стовпчику |

Створимо об'єкт DataTable для таблиці clientsTable:

**DataTable clientsTable = dsTests.Tables.Add("Клієнти");**

або

**DataTable clientsTable = new DataTable("Клієнти");**

**dsTests.Tables.Add(clientsTable);**

Для того, щоб додати об'єкт типу DataTable до колекції Tables об'єкта типу DataSet, використовується метод Add:

**DataSet dataSet = new DataSet();**

**DataTable clientsTable =new DataTable("Клієнти ");**

**dataSet.Tables.Add(clientsTable);**

Клас DataColumn вживається для визначення імені й типу даних стовпчика в DataTable. Новий об'єкт DataColumn можна створити або за допомогою конструктора DataColumn, або викликавши метод Add () властивості-колекції DataTable.Columns:

// Додавання стовпчика за допомогою конструктора

**DataColumn myColumn = new DataColumn("ID", typeof(System.Int32));**

// Додавання стовпчика за допомогою DataTable

**productsTable.Columns.Add("ID", Type.GetType("System.Int32"));**

У наступному фрагменті коду створюється новий об'єкт DataTable. Після створення об'єкта визначається схема таблиці за допомогою додавання до колекції Columns двох нових стовпчиків:

// Створення нового DataTable

**DataTable clientsTable = new DataTable("Clients");**

// Створення схеми таблиці clientsTable

**clientsTable.Columns.Add("ID", typeof(System.Int32));**

**clientsTable.Columns.Add("Name", typeof(System.String));**

Після створення DataTable і визначення стовпчиків можна перейти до заповнення таблиці даними. При цьому в колекцію DataTable.Rows типу DataRowCollection додаються нові об'єкти DataRow. Щоб створити в DataTable новий рядок, спочатку необхідно викликати метод DataTable.NewRow (), що повертає об'єкт DataRow, який задовольняє поточній схемі DataTable. Потім потрібно встановити значення кожного стовпчика з DataRow, а після цього викликати метод DataTable.Rows.Add(), передавши як його єдиний аргумент щойно створений об'єкт DataRow:

```
// Створення нового DataRow з тією же схемою, що й DataTable
    DataRow clientRow = clientsTable.NewRow();
// Встановлення значень стовпчиків
    clientRow.Item["ID"] = 1;
    clientRow.Item["Name"] = "Петренко";
// Додавання DataRow до DataTable
    clientsTable.Rows.Add(clientRow);
```

В об'єктній моделі ADO.NET важливе місце посідає клас DataAdapter. Він забезпечує зв'язок між автономними об'єктами класів DataSet, DataTable тощо з підключеними до бази даних об'єктами DbCommand. Об’єкт DataAdapter не тільки дозволяє заповнити об'єкт DataSet або DataTable із джерела даних, але й надає зручний механізм внесення змін у базу даних.

Для створення об'єкта DataAdapter можна використати чотири конструктори, які наведено в табл. 3.3.

Таблиця 3.3.

**Конструктори об'єкта DataAdapter**

| № | Вид | Опис |
|---|---|---|
| 1 | public DataAdapter () | Конструктор за замовчуванням |
| 2 | public DataAdapter (string commandText, string connectionString) | Перший параметр є властивістю CommandText об'єкта Command. У свою чергу об'єкт Command подається властивістю SelectCommand об'єкта DataAdapter. Другий параметр – це рядок підключення до бази даних ConnectionString |
| 3 | public DataAdapter (string commandText, Connection connection) | Перший параметр – це властивість CommandText об'єкта Command, другий – об'єкт класу Connection |
| 4 | public DataAdapter (Command command) | Як параметр передається про ініціалізований об'єкт класу Command |

Приклади створення об'єктів DataAdapter розглянемо під час використання його методів. Об'єкт DataAdapter має чотири методи, що наведені в табл. 3.4.

Таблиця 3.4

**Методи об'єкта DataAdapter**

| Метод | Опис |
|---|---|
| Fill | Виконує запит, що зберігається у властивості SelectCommand |
| FillSchema | Одержує інформацію про схему для запиту у властивості SelectCommand |
| GetFillParameters | Повертає масив з параметрами для властивості SelectCommand |
| Update | Передає в базу даних зміни, що зберігаються в об'єкті DataSet (DataTable або DataRows) |

Створимо застосування, в якому на формі відображаються дані таблиці "Товари" для виконання оновлення, додавання й видалення рядків. Для цього створимо проект Windows Forms мовою C# з ім'ям ***appТовари***. Потім скопіюємо файл бази даних ***ТовариСервер.mdb*** у папку appТовари\bin\Debug нового застосування.

Зараз створимо клас ***myPublic*** для зберігання величин, які є спільними для всього проекту. Для цього необхідно виконати команду **Проект – Додати клас**. У вікні **Додавання нового елемента** виберіть значок **Клас** й уведіть ім'я ***myPublic.cs***, а потім клацніть кнопку **Додати**. У вікні, що відкрилося, ведіть такий код класу:

```
class myPublic
 {
  //Рядок з'єднання
  public static string connString =
@"Provider=Microsoft.Jet.OLEDB.4.0;Data
Source=|DataDirectory|\ТовариСервер.mdb";
 }
```

Додамо до застосування нову форму з елементом DataGridView для відображення записів таблиці ***Товари***.

У вікні коду форми визначимо потрібні простори імен:

```
using System;
using System.Data;
using System.Text;
using System.Windows.Forms;
using System.Data.OleDb;
```

У тому самому вікні введемо такий опис спільних об'єктів:

```
//Спільні об'єкти
  OleDbConnection conn;   //З'єднання
  DataSet ds;             //Набір даних
  OleDbDataAdapter da;    //Адаптер даних
  BindingSource bindingТовары = new BindingSource();
//Прив'язка ds до dataGridView
  private bool formТоварыIsLoaded = false; //Завантажено форму?
  private bool rowТоварыIsAdding = false;//Операція додавання запису?
```

Додамо код оброблювача події "Завантаження форми", щоб відображати в елементі DataGridView всі записи таблиці ***Товари*** під час відкриття форми

```
private void formТовары_Load(object sender, EventArgs e)
{
  //Створюємо з'єднання OleDbConnection
  conn = new OleDbConnection();
  conn.ConnectionString = myPublic.connString;
  //Створюємо командний об'єкт
  OleDbCommand cmd = new OleDbCommand();
  cmd.Connection = conn;
  cmd.CommandType = CommandType.Text; //За замовчуванням
  //Запит для одержання набору товарів
  cmd.CommandText = "SELECT * FROM Товари";
  //Створюємо DataSet
  ds = new DataSet("dsТовари");
  //Будуємо таблицю Товари
  DataTable tableТовары = new DataTable("Товари");
  DataColumnCollection cols = tableТовары.Columns;
  //Ключове поле таблиці
  DataColumn column = cols.Add("Код_товару", typeof(System.Int32));
  column.AutoIncrement = true;
  column.AutoIncrementSeed = -1;
  column.AutoIncrementStep = -1;
//Негативні значення ключового поля обрано для того, щоб вони
  //не збіглися з ключовим полем у таблиці бази даних
//під час додавання рядків.
```

```
//Цей приклад докладно розглянуто в лабораторній роботі №4.
  //Інші стовпчики
  cols.Add("Товар", typeof(System.String)).MaxLength = 25;
  cols.Add("Ціна", typeof(System.Decimal));
  cols.Add("Ціна_закупівлі", typeof(System.Decimal));
  //Задаємо первинний ключ
  tableТовары.PrimaryKey = new DataColumn[ ] { cols["Код_товару"]};
  //Додаємо таблицю Товари в DataSet
  ds.Tables.Add(tableТовари);
  //Заповнюємо таблицю Товари в DataSet даними із БД
  //Створюємо OleDbDataAdapter
  da =new OleDbDataAdapter(cmd.CommandText,myPublic.connString);
  //Заповнюємо таблицю Товари в ds даними з бази методом Fill()
  da.Fill(ds.Tables["Товари"]);
  //Відображаємо таблицю на формі, прив'язуючи її до
  //dataGridViewТовари
  bindingТовари.DataSource = tableТовари;
  dataGridViewТовари.DataSource = bindingТовари;
  //Повідомляємо про закінчення завантаження форми
  formТоварыIsLoaded = true;
}
```

В існуюче застосування додамо функцію для зміни даних про обраний товар. Вибір запису, що змінюється, виконується в елементі DataGridView на формі ***Товари***. Після зміни даних в обраному полі з'являється діалогове вікно із запитом на збереження зроблених змін. Тут можна або погодитися, або скасувати зміни. У першому разі рядок зі зміненими даними переноситься в базу даних.

Для цього у вікні властивостей DataGridView знайдіть подію CellValueChanged і двічі клацніть на ній. У вікні коду введіть оператори тіла оброблювача події dataGridView**Товари**_CellValueChanged.

```
private void dataGridViewТовари_CellValueChanged(objectsender,
DataGridViewCellEventArgs e)
{
  //Зміна даних
  //При додаванні запису також відбувається зміна даних,
  //але при додаванні Код_товару негативний.
```

```
//Щоб переконатися, що маємо справу зі зміною, перевіряємо
// значення поля Код_товару
string strКод_товару =
dataGridViewтовары.Rows[e.RowIndex].Cells[0].Value.ToString();
int intКод_товару = int.Parse(strКод_товару);
if (intКод_товару > 0)
{
  //Запит на підтвердження збереження зміненого запису
  DialogResult result =
  MessageBox.Show("Зберегти зміну?", "Змінений запис",
  MessageBoxButtons.YesNo, MessageBoxIcon.Warning,
  MessageBoxDefaultButton.Button2);
  if (result.ToString() == "Yes")
  {
    //Вибираємо змінений запис із DataGridView
    int i = e.RowIndex;
    DataGridViewRow row=dataGridViewТовари.Rows[i];
    //Формуємо поля для запиту
    string Код_товару = row.Cells["Код_товару"].Value.ToString();
    string Товар = "'" + row.Cells["Товар"].Value.ToString() + "'";
    string Ціна = row.Cells["Ціна"].Value.ToString().Replace(",", ".");
    string Ціна_закупівлі =
          row.Cells["Ціна_закупівлі"].Value.ToString().Replace(",", ".");
    //Формуємо запит на зміну
    string SQLUpdate="UPDATE Товари SET Товар = " + Товар +
      ", Ціна =" + Ціна + ", Ціна_закупівлі =" + Ціна_закупівлі +
      " WHERE Код_товару= " + Код_товару;
    //Вказуємо запит адаптеру
    da.UpdateCommand = new OleDbCommand(SQLUpdate, conn);
    //Завершуємо процес редагування
    bindingтовары.EndEdit();
    //Надсилаємо зміни в БД
    da.Update(ds,"Товари");
  }
  else
  {
```

```
//Відмовляємося від зроблених змін
      int i = e.RowIndex;
      ds.Tables["Товари"].Rows[i].RejectChanges();
    }
  }
}
```

В існуюче застосування уведемо функцію для додавання даних про новий товар. Вибір змінюваного запису виконується в елементі DataGridView на формі ***Товари*** (позначений символом *). Після уведення даних при переході на інший рядок з'являється діалогове вікно із запитом на збереження доданих даних. Тут можна або погодитися, або скасувати зміни. У першому разі рядок з новими даними переноситься в базу даних.

Для цього у вікні властивостей DataGridView знайдіть подію **RowsAdded** і двічі клацніть на ній. У вікні коду введіть оператори тіла оброблювача події **dataGridViewТовари_RowsAdded**. Він матиме такий вигляд:

```
private void dataGridViewТовари_RowsAdded(object sender,
DataGridViewRowsAddedEventArgs e)
{
  //Додавання даних
  if (formТоварыIsLoaded)
  {
    //Повідомляємо, що почався процес додавання запису.
    //Він закінчиться, коли користувач покине рядок у DataGridView.
    rowТоварыIsAdding = true;
  }
}
```

У вікні властивостей D**ataGridView** знайдіть подію **RowLeave** і двічі клацніть на ній. У вікні коду введіть оператори тіла оброблювача події **dataGridViewТовари_RowLeave**. Він матиме такий вигляд:

```
private void dataGridViewТовари_RowLeave(object sender,
DataGridViewCellEventArgs e)
{
  //Продовжуємо процес
  if (rowТоварыIsAdding)   {
  //Додавання даних
```

```
//Завершуємо процес уведення даних у DataGridView
  dataGridViewТовари.EndEdit();
//Запит на підтвердження збереження доданого запису
  DialogResult result =
    MessageBox.Show("Зберегти додавання?", "Доданий запис",
    MessageBoxButtons.YesNo, MessageBoxIcon.Warning,
    MessageBoxDefaultButton.Button2);
  if (result.ToString() == "Yes")
  {
//Вибираємо доданий запис
    int i = e.RowIndex;
    DataGridViewRow row = dataGridViewТовари.Rows[i];
//Формуємо поля для запиту
    string Товар = "'" + row.Cells["Товар"].Value.ToString() + "'";
    string Ціна = row.Cells["Ціна"].Value.ToString().Replace(",", ".");
    string Ціна_закупівлі =
    row.Cells["Ціна_закупівлі"].Value.ToString().Replace(",", ".");
//Формуємо запит на додавання
    string SQLInsert = "INSERT INTO Товари " +
      "(Товар, Ціна, Ціна_закупівлі)" +
      "VALUES("+Товар+","+ Ціна+","+ Ціна_закупівлі+")";
//Вказуємо запит адаптеру
    da.InsertCommand= new OleDbCommand(SQLInsert, conn);
//Завершуємо процес редагування й додаємо запис у БД
    bindingТовари.EndEdit();
    da.Update(ds, "Товари");
  }
  else
  {
//Відмовляємося від додавання запису
    int i = e.RowIndex;
    ds.Tables["Товари"].Rows[i].RejectChanges();
  }
//Повідомляємо про закінчення процесу додавання запису
  rowТовариIsAdding=false;
} }
```

Об'єктна модель ADO.NET надає засоби динамічної генерації команд оновлення даних з використанням об'єкта SqlCommandBuilder. При цьому досить тільки сформувати запит на вибірку даних Select. На основі цього запиту об'єкт SqlCommandBuilder звертається до бази даних за іменами таблиці й стовпчиків, а також за відомостями про ключові поля. Команди Update, Insert, Delete генеруються, якщо виконуються всі такі умови:

запит повертає дані тільки з однієї таблиці;

у таблиці визначено первинний ключ;

первинний ключ є в результатах запиту.

Розглянемо використання об'єкта SqlCommandBuilder на прикладі попередньої задачі. Для цього в області описів спільних об'єктів класу ***formТовари*** закоментуйте опис змінних formТоварыIsLoaded й rowТоварыIsAdding.

Вставте перед закоментованими рядками опис об'єкта ***OleDbCommand–Builder***. Після цього область описів спільних об'єктів класу набуде такого вигляду:

```
//Спільні об'єкти
OleDbConnection conn;   //З'єднання
DataSet ds;             //Набір даних
OleDbDataAdapter da;    //Адаптер даних
BindingSource bindingТовари = new BindingSource();
//Прив'язка ds до dataGridView
OleDbCommandBuilder cb;
//Будівельник запитів на зміну,  додавання
//і видалення для адаптера
```

У коді оброблювача події **formТовари_Load** виконайте такі зміни:

закоментуйте останній рядок зі змінною formтоварыIsLoaded.

після створення об'єкта ***da*** вставте оператор створення об'єкта ***cb*** (CommandBuilder).

У результаті код оброблювача події form**Товари**_Load має такий вигляд:

```
private void formТовари_Load(object sender, EventArgs e)
{

  //З'єднання OleDbConnection
  conn = new OleDbConnection();
```

```
conn.ConnectionString = myPublic.connString;
//Командний об'єкт
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = conn;
cmd.CommandType = CommandType.Text; //За замовчуванням
//Список товарів
//Запит для одержання набору товарів
cmd.CommandText = "SELECT * FROM Товари";
//Створюємо DataSet
ds = new DataSet("dsХліб");
//Будуємо таблицю Товари
DataTable tableТовари = new DataTable("Товари");
DataColumnCollection cols = tableТовари.Columns;
//Ключове поле
DataColumn column = cols.Add("Код_товару", typeof(System.Int32));
column.AutoIncrement = true;
column.AutoIncrementSeed = -1;
column.AutoIncrementStep = -1;
//Інші стовпці
cols.Add("Товар", typeof(System.String)).MaxLength = 25;
cols.Add("Ціна", typeof(System.Decimal));
cols.Add("Ціна_закупівлі", typeof(System.Decimal));
//Задаємо первинний ключ
tableТовари.PrimaryKey = new DataColumn[ ] { cols["Код_товару"]};
//Додаємо таблицю Товари до DataSet
ds.Tables.Add(tableТовари);
//Заповнюємо таблицю Товари в DataSet даними із БД
//Створюємо OleDbDataAdapter
da =new OleDbDataAdapter(cmd.CommandText,myPublic.connString);
cb = new OleDbCommandBuilder(da);
//Заповнюємо таблицю Товари в ds даними з бази
da.Fill(ds.Tables["Товари"]);
//Відображаємо таблицю на формі, прив'язуючи її до
dataGridViewТовари
bindingТовари.DataSource = tableТовари;
dataGridViewТовари.DataSource = bindingТовари;
```

```
 }
```

Додати на форму ***Товари*** кнопку ***Зберегти***, що зберігає в базі даних всі зміни даних.

```
private void buttonЗберегти_Click(object sender, EventArgs e)
{
 //Завершуємо процес редагування й додаємо запис в БД
 bindingТовари.EndEdit();
 da.Update(ds, "Товари");
}
```

Додамо оброблювач події form**Товари**_FormClosing, щоб нагадати користувачеві про те, що залишилися не збереженими дані. Для цього у вікні властивостей форми ***Товари*** знайдіть подію **FormClosing** і двічі клацніть на ньому. У вікні коду введіть оператори тіла оброблювача події **formТовари_FormClosing**. Він матиме такий вигляд:

```
private void formТовари_FormClosing(object sender,
FormClosingEventArgs e)
{
 if (ds.HasChanges())
 {
  //Запит на підтвердження збереження змін
  DialogResult result =
   MessageBox.Show("Зберегти всі зміни в таблиці?", "Товари",
   MessageBoxButtons.YesNo, MessageBoxIcon.Warning,
   MessageBoxDefaultButton.Button2);
  if (result.ToString() == "Yes")
  {
   buttonЗберегти_Click(sender, e);
  }
  else
  {
   //Відмовляємося від видалення запису
   ds.Tables["Товари"].RejectChanges();
  }
}
```

**Завдання для лабораторної роботи**

Створити застосування, що виконує оновлення, додавання й видалення рядків таблиці, яка задана викладачем.

Створити застосування, що виконує функції з використанням будівельника запитів CommandBuilder.

**Додаткові завдання**

1. В процесі роботи застосування виконується оновлення даних таблиці. У деяких запитах оновлення даних можливе виникнення помилок. Одержати всю інформацію про помилки, що доступна після невдалого оновлення бази даних без аварійного завершення застосування.

2. В процесі налагодження застосування потрібно забезпечити перегляд, як поточних значень змінених рядків таблиці, так і їхніх початкових значень.

3. Відобразити обчислюване значення для кожного рядка локальної таблиці, виконати фільтрацію даних таблиці за середнім значенням отриманого стовпчика.

4. Відобразити підсумкові значення в обраному стовпчику таблиці, виконати фільтрацію даних таблиці за отриманим значенням.

5. В процесі роботи застосування користувач редагує дані таблиць. Для ведення протоколу роботи користувача потрібно відобразити всі зміни даних у таблицях.

**Контрольні запитання**

1. Принцип роботи застосування у роз'єднаному середовищі.
2. Об'єкти, які використовують для роботи застосувань в роз'єднаному середовищі.
3. Створення командних об'єктів
4. Параметри командних об'єктів.
5. Створення об'єкта DataSet.
6. Створення об'єктів DataColumn.
7. Призначення об'єкта DataAdapter.
8. Створення об'єкта DataAdapter.

## Лабораторна робота 4. Розробка програм виконання операцій з ієрархічними даними.

**Мета лабораторної роботи:**

1. Отримання практичних навичок з відображення ієрархічних даних у застосуваннях і виконання операцій з ними.

2. Удосконалення навичок роботи з інтегрованим середовищем розробки Visual C# і довідковою системою Microsoft Developer Network (MSDN).

**Перед виконанням лабораторної роботи студент повинен знати:**

1. Основи використання бібліотеки Windows Forms.
2. Організацію системи встановлення зв'язків між таблицями.
3. Принципи прив'язування даних до елементів інтерфейсу.

**Після виконання лабораторної роботи студент повинен уміти:**

1. Самостійно розробляти прості застосування для роботи з ієрархічними даними.

2. Використовувати основні бібліотеки .Net Framework під час розробки програм.

## Основні відомості про роботу застосувань з ієрархічними даними

У процесі створення застосування розробник обов'язково зіткнеться з ситуацією, коли буде потрібно вивести або програмно звернутися до даних зі зв'язаних таблиць бази даних. Користувачеві необхідно забезпечити зручне переміщення між різними таблицями з інформацією. Крім того, їм необхідно редагувати дані. Застосуванням найчастіше доводиться збирати агрегатні відомості, а при зміні одного запису виконувати зміни в інших, пов'язаних з ним записах (наприклад, коли вилучено замовлення, виникає потреба у видаленні всіх пов'язаних з ним товарів).

Для пошуку дочірніх записів, що пов'язані з батьківським записом в іншому об'єкті *DataTable,* достатньо викликати метод *GetChildRows* об'єкта *DataRow* і передати йому ім'я об'єкта *DataRelation,* що визначає відношення між об'єктами *DataTable.* Замість імені можна також передати сам об'єкт *DataRelation.* Метод *GetChildRows* повертає зв'язані дані у вигляді масиву об'єктів *DataRow.* Нижче наведено стандартний приклад виведення вмісту записів з батьківської таблиці й, пов'язаних з ними записів з дочірньої таблиці.

```
//Додаємо об'єкт DataRelation, що зв'язує дві таблиці;
        rel = ds.Relations.Add("Customers_Orders".
        ds.Tables["Customers"].Columns["CustomerlD"],
        ds.Tables["Orders"].Columns["CustomerlD"]);
//Заповнюємо об'єкт DataSet
        string strConn, strSQL ;
        str.Conn = @"DataSource=.\SQLExpress;"+
        "Initial Catalog=Northwind;Integrated Security=True;";
        strSQL = "SELECT CustomerlD,   CompanyName FROM
        Customers " + " WHERE Customerl LIKE   'A%';" +
        "SELECT Orderl, Customerl, OrderDate FROM Orders " +
        " WHERE CustomerlD LIKE    'A%' " ;
        SqlDataAdapter da = new SqlDataAdapter(strSQL,strConn);
        da.TableMappings.Add("Table",   "Customers");
        da.TableMappings.Add( "Table1",   "Orders");
```

В останніх двох рядках встановлюємо відповідність між таблицями "Table" бази даних і таблицями Customers,Orders об'єкта DataSet. Чому таблиця Customers, що ми витягаємо, тут називається "Table"? Справа в тому, що в об'єкта DataAdapter немає можливості визначити, як називається таблиця в базі, і він підставляє можливу назву "Table".

```
        da.Fill(ds);
// Переглядаємо записи про клієнтів
        foreach (DataRow rowCustomer in
        ds.Tables[ "Customers"].Rows)
        { Console.WriteLine("{0}  -  {1}",   rowCustomer["CustomerlD"],
        rowCustomer[ "CompanyName"]);
// Переглядаємо замовлення,   що розміщені цими клієнтами
foreach (DataRow rowOrder in rowCustomer.GetChildRows(rel))
        Console.WriteLine(" {0} {1:MM/dd/yyyy}",   row0rder["OrderlD"],
        rowOrder["OrderDate"]);
        Console.WriteLine();
```

Процес оновлення даних у зв'язаних таблицях істотно ускладнюється, якщо вони використовуються декількома застосуваннями при наявності ключового поля типу AutoIncrement в Sql Server або "Лічильник" в Access. У базі даних цей стовпчик уважається стовпчиком ідентифікації. Тому під час додавання нового рядка, його заповнює СУБД, а не автономне

застосування. Автономний користувач, не може точно визначити наступне значення ідентифікаційного номера, тому що інший користувач може використати значення, яке ви намагаєтеся занести в DataTable. Тому в застосуваннях необхідно генерувати фіктивне значення первинного ключа під час оновлення запису. Після завершення оновлення формувати справжнє значення первинного ключа. У лабораторній роботі №3 такі способи вже використовувалися під час створення ключового поля Код_товару:

```
DataColumn column = cols.Add("Код_товару", typeof(System.Int32));
column.AutoIncrement = true;
column.AutoIncrementSeed = -1;
column.AutoIncrementStep = -1;
```

Негативні значення гарантують відмінність ключа в застосуванні від його значення в таблиці бази даних.

Розглянемо приклад застосування, що виконує операцію додавання записів у базу даних, що складається із двох таблиць "Товар" й "Продаж". Вони мають ключові поля типу AutoIncrement. Схема бази даних наведена на рис. 4.1.


Рис. 4.1. **Схема бази даних "Продаж товарів"**

Додамо в код форми потрібні простори імен й опис спільних об'єктів. При цьому створимо об'єкти для організації зв'язку з базою даних, об'єкти DataSet, DataTable, DataColumn, DataRelation і два об'єкти SqlDataAdapter (по одному для кожної таблиці).

```
using System;
using System.Collections.Generic;
using System.ComponentModel;
using System.Data;
using System.Drawing;
using System.Linq;
using System.Text;
```

```
using System.Windows.Forms;
using System.Data.SqlClient;
using System.Data.Common;
namespace Вставка
{    public partial class Form1 : Form
   {
       string connectionString = @"Data Source=.\SQLEXPRESS;
AttachDbFilename=|DataDirectory|\Продаж_Товарів.mdf;Integrated
Security=True;Connect Timeout=30;User Instance=True";
       DataTable ТоварTable, ПродажTable;
       DataSet ds = new DataSet();
       SqlConnection MyConn = new SqlConnection();
        MyConn.ConnectionString = connectionString;
       DataColumn parentCol, childCol;
       DataRelation relat;
       SqlDataAdapter sqlDA;
       SqlDataAdapter sqlDA1;
       public Form1()
       {
          InitializeComponent();
          createTable();
       }
```

У наведеному вище коді є звертання до методу createTable(), у якому необхідно створити схеми таблиць "Товар" й "Продаж", з огляду на особливості ключових полів і зв'язків між ними. Частина коду цього методу наведена нижче:

```
       public void createTable()
       {
// Створення схеми для таблиці
// Будуємо батьківську таблицю
      ТоварTable = new DataTable("Товар");
      DataColumn dc = null;
      dc = new DataColumn("Код_Товару");
      dc.Unique = true;
      dc.AutoIncrement = true;
```

```
dc.AutoIncrementSeed = -1;
dc.AutoIncrementStep = -1;
dc.DataType = typeof(System.Int32) ;
 ТоварTable.Columns.Add(dc);
   dc = new DataColumn("Товар");
     animalsTable.Columns.Add(dc);
     ds.Tables.Add(animalsTable);

// Будуємо дочірню таблицю
ПродажTable = new DataTable("Продаж");
   dc = new DataColumn("Код_продажу");
   dc.Unique = true;
   dc.AutoIncrement = true;
   dc.AutoIncrementSeed = -1;
   dc.AutoIncrementStep = -1;
   dc.DataType = typeof(System.Int32);
   myanimTable.Columns.Add(dc);
   dc = new DataColumn("Код_товару");
   dc.DataType = typeof(System.Int32);
   myanimTable.Columns.Add(dc);
   dc = new DataColumn("Кількість");
   dc.DataType = typeof(System.Int32);
   myanimTable.Columns.Add(dc);
   ds.Tables.Add(myanimTable);
   // Створення відносин
   parentCol = ds.Tables["Товар"].Columns["Код_товару"];
   childCol = ds.Tables["Продаж"].Columns["Код_товару"];
    relat = new DataRelation("Продаж_Товар",parentCol,childCol);
   ds.Relations.Add(relat);
```

Створимо на формі два компоненти DataGridView з іменами dgView, dgView1 і виведемо на форму вміст двох таблиць:

// Прив'язка таблиці за даними, хоча в ній ще немає рядків

```
        dgView.DataSource = ТоварTable;
        dgView1.DataSource = ПродажTable;
```
Нові дані будуть додаватися в компонентах DataGridView і передавати їх для уведення в таблиці бази даних краще використовуючи параметри. У наступному фрагменті коду наведено опис параметрів:
```
        SqlParameter param = null;
// Параметри для 1-ої таблиці
        param = new SqlParameter("@Код_товару", SqlDbType.Int);
        param.Direction = ParameterDirection.Input;
        param.SourceColumn = "Код_товару";
        insertCommand.Parameters.Add(param);
    param = new SqlParameter("@Товар", SqlDbType.VarChar, 50);
        param.SourceColumn = "Товар";
        insertCommand.Parameters.Add(param);
// Параметри для 2-ої таблиці
        param = new SqlParameter("@Код_продажу", SqlDbType.Int);
        param.Direction = ParameterDirection.Input;
        param.SourceColumn = "Код_продажу";
        insertCommand1.Parameters.Add(param);
        param = new SqlParameter("@Код_товару", SqlDbType.Int);
        param.SourceColumn = "Код_товару";
        insertCommand1.Parameters.Add(param);
        param = new SqlParameter("@Кількість", SqlDbType. Int);
        param.SourceColumn = "Кількість";
        insertCommand1.Parameters.Add(param);
```
Виконання всіх операцій з даними будемо здійснювати з використанням збережених процедур. Вибір такого способу взаимодії з даними пояснюється насамперед тим, що при виконанні команди Insert застосуванню потрібно повернути значення первинного ключа, справжнє його значення відомо в першу чергу базі даних. Тепер приступимо до створення команд, що забезпечують відображення на формі даних з таблиць. У наведеному нижче коді для доступу до даних батьківської й дочірньої таблиці використовуються збережені процедури UP_Select й UP_Select1 відповідно:
```
// Створення команди SELECT батьківської таблиці
      sqlDA = new SqlDataAdapter("UP_Select", connectionString);
      sqlDA.SelectCommand.CommandType =
```

```
        CommandType.StoredProcedure;
        sqlDA.SelectCommand.CommandText = "UP_Select";
// Створення команд SELECT дочірньої таблиці
        sqlDA1 = new SqlDataAdapter("UP_Select1", connectionString);
        sqlDA1.SelectCommand.CommandType =
        CommandType.StoredProcedure;
        sqlDA1.SelectCommand.CommandText = "UP_Select1";
```

Перш ніж використати застосування, треба створити збережені процедури в базі даних. Текст збережених процедур наведено нижче:

```
ALTER PROCEDURE UP_Select
@Код_товару int=null
AS        SET NOCOUNT ON
if @Код_товару is not null
begin select Код_товару, Товар from Товар
where Код_товару =@ Код_товару
return 0 end
select Код_товару, Товар from Товар
        return 0
ALTER PROCEDURE UP_Select1
@Код_продажу int=null
AS        SET NOCOUNT ON
if @ Код_продажу is not null
begin select Код_продажу, Код_товару, Кількість from Продаж  where
Код_продажу =@ Код_продажу
return 0 end
select * from Продаж
return 0
```

За аналогією з попередніми командами Select створимо команди Insert додавання рядків у таблиці бази даних за допомогою збережених процедур UP_INSERT й UP_INSERT1. Код команд додавання рядків наведено нижче:

```
// Створення команд INSERT батьківської таблиці
        SqlCommand insertCommand = new SqlCommand();
        insertCommand.CommandType =
CommandType.StoredProcedure;
        insertCommand.CommandText = "UP_INSERT";
        insertCommand.Connection = new
SqlConnection(connectionString);
```

```
        insertCommand.UpdatedRowSource = UpdateRowSource.Both;
        sqlDA.InsertCommand = insertCommand;
// Створення команд INSERT дочірньої таблиці
        SqlCommand insertCommand1 = new SqlCommand();
        insertCommand1.CommandType =
CommandType.StoredProcedure;
        insertCommand1.CommandText = "UP_INSERT1";
        insertCommand1.Connection = new
SqlConnection(connectionString);
        insertCommand1.UpdatedRowSource = UpdateRowSource.Both;
        sqlDA1.InsertCommand = insertCommand1;      }
```

У наведеному вище фрагменті програми треба звернути увагу на такі рядки:

```
        insertCommand.UpdatedRowSource = UpdateRowSource.Both;
        insertCommand1.UpdatedRowSource = UpdateRowSource.Both;
```

Ці рядки визначають процес оновлення джерела даних після виконання команди **Insert.** Те, як результати виклику методу **Update()** об'єкта **DataAdapter** будуть застосовуватися до об'єкта DataRow, визначає властивість **UpdatedRowSource**. Можливі значення властивості **UpdateRowSource**:

**Both** – вставленому або оновленому рядку в наборі даних співставляються дані як з першого повернутого рядка, так і з вихідного параметра;

**FirstReturnedRecord** – вставленому або оновленому рядку в наборі даних співставляються дані з першого повернутого рядка;

**None** – повертають значення і параметри ігноруються. Це значення встановлено за замовчуванням, коли дана команда генерується об'єктом **CommandBuilder**;

**OutputParameters** – вставленому або оновленому рядку в наборі даних співставляються дані з вихідного параметра.

У такий спосіб для об'єкта DataAdapter визначається команда вставлення з параметрами. Тепер у джерело даних можна додавати нові рядки, а значення стовпчиків "Код_товару" й "Код_продажу", що згенеровані джерелом даних для цих рядків, можна було б одержувати через вихідний параметр або через перший запис, який повертають збережені процедури **UP_INSERT й UP_INSERT1**. Слід зазначити, що параметри "Код_товару" й "Код_продажу" в процедурах обов'язково

повинні описуватися із службовим словом **output.** Нижче наведено код збережених процедур **UP_INSERT й UP_INSERT1:**

```
ALTER PROCEDURE UP_INSERT
@Код_товару int output,
@Товар VARCHAR(50)
AS
SET  NOCOUNT  ON
INSERT  Товар(Товар)VALUES(@Товар)
if   @@rowcount=0
return 1
set @Код_товару = Scope_Identity()
 select Scope_Identity() Код_товару
return 0
ALTER PROCEDURE UP_INSERT1
@ Код_продажу int output,
@Код_товару int,
@Кількість int
AS
SET  NOCOUNT  ON
INSERT  Продаж(Код_товару,Кількість)
VALUES(@Код_товару,@Кількість)
if   @@rowcount=0
return 1
set @Код_продажу = Scope_Identity()
 select Scope_Identity() N
return 0
```

Збережена процедура виконує такі функції:

обчислює значення **AccountNumber**, що становить наступне число;

виконує вставку нового рядка;

вибирає значення первинного ключа з тільки що уведеного рядка за допомогою функції SQL Server **SCOPE_IDENTITY()**.

У результаті прооблених дій отримано код, що створює всі необхідні об'єкти для виконання операції додавання рядка. Тепер треба використати їх. Створимо метод UpdateData(), що призначений безпосередньо для додавання рядків. Нижче наведено код цього методу:

```
private void UpdateData()
{
   try
```

```
        {
sqlDA.Update(ds.Tables[0].Select(null, null, DataViewRowState.Added));
sqlDA1.Update(ds.Tables[1].Select(null, null,
DataViewRowState.Added));
        }
        catch (Exception ex)
        {
           MessageBox.Show(ex.Message);
        }
}
```

У даному фрагменті коду використовується перевантажений метод Update() об'єкта SqlDataAdapter. Він додає масив рядків, що відбираються за їхнім станом (DataViewRowState.Added).

У розроблювальному застосуванні залишилося створити елементи керування процесом оновлення даних. Для цього потрібно додати на форму дві кнопки "Завантажити дані" й "Зберегти зміни". Код, що виконує відповідні операції, наведено нижче:

```
private void Завантажити_Click(object sender, EventArgs e)
{
   animalsTable.Rows.Clear();
   myanimTable.Rows.Clear();
   sqlDA.Fill(ТоварTable);
   sqlDA1.Fill(ПродажиTable);
}
private void Зберегти_Click(object sender, EventArgs e)
{
   UpdateData();
   ds.AcceptChanges();
}
```

У запропонованому прикладі розглядається тільки операція додавання рядків у базу даних, що складається із двох таблиць. Для виконання операцій видалення й відновлення рядків необхідно створити відповідні команди в методі **createTable()** і збережені процедури в базі даних, а потім у методі **UpdateData()** виконати команди **sqlDA.Update** й **sqlDA.Update1** для вилучених (Deleted) і оновлених (Updated) рядків. При цьому, треба звернути увагу на послідовність виконання цих операцій у батьківській і дочірній таблицях. Нагадаємо, що операції

додавання й відновлення рядків у базі даних виконуються "зверху вниз", а видалення рядків "знизу нагору".

### Завдання для лабораторної роботи

1. Створити базу даних в Sql Server (Oracle або Access), що містить не менше чотирьох таблиць. Схема бази даних видається викладачем.
2. Розробити застосування, що забезпечує відображення даних таблиць на формі в наочному вигляді. При виділенні запису в батьківській таблиці на формі повинні відображатися всі залежні записи дочірньої таблиці.
3. Застосування має виконувати оновлення, додавання й видалення даних з таблиць. Забезпечити виконання цих операцій з використанням параметрів і збережених процедур.
4. Виконати тестування, використовуючи кілька екземплярів застосування для паралельної роботи при оновленні рядків у базі даних.

### Контрольні запитання

1. Методи об'єкта DataAdapter.
2. Створення відносин між таблицями.
3. Призначення функції **Scope_Identity() .**
4. Використання процедур, що зберігаються при оновленні даних.
5. Особливості оновлення таблиць з ключовим полем типу автоінкремент.

## Лабораторна робота 5. Розробка програм взаємодії з базами даних з використанням транзакцій

**Мета лабораторної роботи:**

1. Придбання практичних навиків використання транзакцій при оновленні даних в ієрархічно зв'язаних таблицях.
2. Дослідження рівнів ізоляції транзакцій.
3. Вдосконалення навиків роботи з інтегрованим середовищем розробки Visual C# і довідковою системою Microsoft Developer Network (MSDN).

**Перед виконанням лабораторної роботи студент повинен знати:**

1. Основи використання бібліотеки Windows Forms.

2. Організацію функціонування СУБД в розрахованому на багато користувачів середовищі.

**Після виконання лабораторної роботи студент повинен уміти:**

1. Самостійно розробляти прості застосування для роботи в розрахованому на багато користувачів середовищі.

2. Застосовувати транзакції при реалізації операцій оновлення бази даних.

3. Правильно вибирати і встановлювати режими ізоляції даних у транзакціях.

## Основні відомості про роботу застосувань з використанням транзакцій

Транзакцією називається виконання послідовності команд (SQL-конструкцій) в базі даних, яка або фіксується при успішному виконанні кожної команди, або відміняється при невдалому виконанні хоч би однієї команди.

ADO.NET забезпечує як підключений, так і автономний доступ до даних і підтримує транзакції в обох цих режимах. У підключеному режимі типова послідовність дій у транзакції така:

відкривається підключення до бази даних за допомогою методу **Open()** об'єкта підключення;

починається транзакція за допомогою методу **BeginTransaction()** об'єкта підключення, цей метод повертає об'єкт транзакції, який надалі використовується для фіксації або відкоту транзакції, всі зміни, виконані будь-якими запитами до виклику методу **BeginTransaction()**, фіксуються в базі даних відразу ж після їх виконання;

за допомогою об'єкта команди виконується команда SQL, для цієї мети можна використовувати більше одного об'єкта команди, але тільки якщо у властивості Transaction всіх цих об'єктів вказаний допустимий об'єкт транзакції;

транзакція фіксується або відкатується за допомогою методу **Commit()** або **Rollback()** об'єкта транзакції;

підключення до бази даних закривається.

Нижче наведений приклад використання транзакції:

```
static void Main(string[] args)
{
```

```
SqlConnection conn = new SqlConnection();
conn.ConnectionString = "integrated security=SSPI;
data source=\".\"; persist security info=False; initial catalog=firm";
conn.Open();
SqlCommand myCommand = conn.CreateCommand();
//Створюєм транзакцію
myCommand.Transaction = conn.BeginTransaction();
try
{
myCommand.CommandText = "INSERT INTO Клієнти (Кодклієнта,
Прізвище, Ім'я, Побатькові) VALUES (6, 'Тихоміров', 'Андрій',
'Борисович')";
myCommand.ExecuteNonQuery();
myCommand.CommandText = "INSERT INTO
Информацияотуристах(Кодклієнта, Серіяпаспорта, Місто,
Держава, Телефон, Індекс)  VALUES (6, 'НК 8444567',
apos;Харків', 'Україна', 3235867, 497538)";
myCommand.ExecuteNonQuery();
// Підтверджуємо транзакцію
 myCommand.Transaction.Commit();
Console.WriteLine("Передача даних успішно завершена ");
 }
catch(Exception ex)
{
// Відхиляємо транзакцію
myCommand.Transaction.Rollback();
Console.WriteLine("При передачі даних відбулася помилка: " +
ex.Message);
 }
finally
{
conn.Close();
} } } }
```

В автономному режимі дані (звичайно одна або більш таблиць) вибираються в об'єкт DataSet. У цьому режимі звичайна послідовність дій така:

відкриття підключення до бази даних;
вибірка потрібних даних в об'єкт DataSet;
закриття підключення;
обробка даних в об'єкті DataSet;
знову відкриття підключення до бази даних;
початок транзакції;
призначення об'єкта транзакції відповідним командам адаптера даних;
занесення в базу даних змін з DataSet;
закриття підключення.

Відповідно до перерахованих дій код методу Update() (з лабораторної роботи №4) при використанні транзакції виглядатиме таким чином:

```
public void Update()
{
   SqlConnection con = new SqlConnection(connectionString);
            con.Open();
//Створюємо транзакцію
        tran = con.BeginTransaction();
try
{
 da.Update(ds.Tables[0].Select(null, null,
DataViewRowState.Added));
 da1.Update(ds.Tables[1].Select(null, null,
DataViewRowState.Added));
//Підтверджуємо транзакцію
          tran.Commit();
  }
     catch (SqlException ex)
        {
//Відхиляємо транзакцію
          tran.Rollback();
   MessageBox.Show("Відбувся відкіт операції!", "Помилка!");
          }
          con.Close();
 }
```

Основні правила, яких необхідно дотримуватися при розробці застосувань з використанням транзакцій:

1. Після початку транзакції по якому-небудь підключенню всі

запити, що використовують це підключення, повинні відноситися до цієї транзакції. Наприклад, усередині транзакції можуть бути виконані два запити **INSERT** і один **UPDATE**. І якщо у середині транзакції спробувати виконати запит **SELECT** без транзакції, то ви одержите повідомлення про помилку, вказуюче, що транзакція ще не завершена.

2. У одного об'єкта підключення у будь-який момент часу може бути тільки одна чекаюча завершення транзакція. Після виклику методу **BeginTransaction** об'єкта підключення неможливо ще раз викликати **BeginTransaction**, поки для цієї транзакції не буде виконана фіксація або відкіт. У таких ситуаціях ви одержите повідомлення про те, що паралельні транзакції не підтримуються. Щоб позбавиться цієї помилки, для виконання запиту потрібно використовувати інше підключення.

При виконанні транзакцій декількома користувачами однієї бази даних можуть виникати такі проблеми:

Dirty reads – "брудне" читання;

on-repeatable reads – неповторюване читання;

Phantom reads – читання фантомів.

Для вирішення цих проблем розроблені чотири рівні ізоляції транзакції:

**Read uncommitted** – транзакція може прочитувати дані, з якими працюють інші транзакції, застосування цього рівня ізоляції може привести до всіх перерахованих проблем;

**Read committed** – транзакція не може прочитувати дані, з якими працюють інші транзакції, застосування цього рівня ізоляції виключає проблему "брудного" читання;

**Repeatable read** – транзакція не може прочитувати дані, з якими працюють інші транзакції, інші транзакції також не можуть прочитувати дані, з якими працює ця транзакція, застосування цього рівня ізоляції виключає всі проблеми, окрім читання фантомів;

**Serializable** – транзакція повністю ізольована від інших транзакцій, застосування цього рівня ізоляції повністю виключає всі проблеми.

Вибір рівня ізоляції можна здійснити за допомогою коду:

```
SqlConnection conn = new SqlConnection(conn_str);
    conn.Open();
```

```
        if (radioButton1.Checked == true)
        {
tran =
conn.BeginTransaction(System.Data.IsolationLevel.ReadCommitted);
        }
        if (radioButton2.Checked == true)
        {
tran =
conn.BeginTransaction(System.Data.IsolationLevel.ReadUncommitted);
        }
        if (radioButton3.Checked == true)
        {
tran =
conn.BeginTransaction(System.Data.IsolationLevel.RepeatableRead);
        }
        if (radioButton4.Checked == true)
        {
tran = conn.BeginTransaction(System.Data.IsolationLevel.Serializable);
        }
```

Для виконання цього коду на формі повинні бути присутніми чотири компоненти RadioButton.

У процесі розробки застосувань часто виникає потреба створення вкладеної транзакції, використовуючи постачальник даних .NET для SQL Server, але команда Begin(), необхідна для цього, є тільки у постачальника даних .NET для OLE DB. Постачальник даних .NET для SQL Server не має вбудованої підтримки вкладених транзакцій. Тому треба імітувати вкладені транзакції за допомогою точок збереження, управляти часом життя класу SqlTransaction і реалізувати необхідну обробку виключень. Код застосування виконує додавання запису в таблицю бази даних використовуючи транзакцію. У даному прикладі передбачається, що для введення значень параметрів необхідно на формі створити такі компоненти: товарTextBox, ценаTextBox, товар2TextBox, цена2TextBox. Нижче приведений код даного прикладу:

```
private DataTable dt;
 private SqlDataAdapter da;
 //
```

```
private void TransactionForm_Load(object sender, System.EventArgs
e)
{
// наповнюємо даними таблицю "Товари"
String sqlQuery = "SELECT Код_товара, Товар, Ціна FROM Товари";
da = new SqlDataAdapter(sqlQuery,conn);
dt = new DataTable("Товари");
da.FillSchema(dt. SchemaType.Source);
da.Fill(dt);
// прив'язуємо таблицю до елементу екранної форми
dataGrid.DataSource = dt.DefaultView:
}
private void insertButton_Click(object sender. System.EventArgs e)
{
string sqlQuery = "INSERT Товари "
+ "(Товар, Ціна) VALUES " + "(@Товар, @Ціна)";
// починаємо транзакцію
conn.Open();
SqlTransaction tran = conn.BeginTransaction();
// створюємо в транзакції команду з параметрами
SqlCommand cmd = new SqlCommand(sqlQuery, conn, tran);
cmd.Parameters.Add(new SqlParameter("@Товар",
SqlDbType.NVarChar, 15));
cmd.Parameters.Add(new SqlParameter("@Ціна", SqlDbType.Int));
try
{
// вставляємо запис №1 у таблицю
if (categoryNamelTextBox.Text.Trim().Length == 0)
// якщо стовпець "Товар" порожній, привласнюємо йому значення null
cmd.Parameters[ "@Товар"].Value = DBNull .Value;
else
cmd.Parameters["@Товар"].Value = товарTextBox.Text:
cmd.Parameters["@Ціна"].Value = цінаTextBox.Text;
cmd.ExecuteNonQuery(); }
catch (Exception ex)
{
```

```
// виникло виключення – проводимо відкіт транзакції
tran.Rollback();
MessageBox.Show(ex.Message +"Відкіт транзакції.");
conn.Close();
return;
}
tran.Save("SavePointReturn");
try
{
// вставляємо запис №2 у таблицю
if (товар2TextBox.Text.Trim().Length == 0)
cmd.Parameters["@Товар"].Value = DBNull.Value;
else
cmd.Parameters["@Товар"].Value = товар2TextBox.Text;
cmd.Parameters["@Ціна"].Value = ціна2TextBox.Text;
cmd.ExecuteNonQuery();
// якщо все гаразд до цього етапу, зберігаємо транзакцію
tran.Commit();
MessageBox.Show("Транзакція збережена!");
}
catch (SqlException ex)
{
tran.Rollback("SavePointReturn");
tran.Commit();
MessageBox.Show(ex.Message +
" Відкіт транзакції Збережений тільки запис №1");
}
finally
{  conn.Close();  }
// оновлюємо дані на формі
da.Fill(dt);
}
```

У класу SqlTransaction постачальника даних .NET для SQL Server є метод Save(), що встановлює в транзакції точку збереження, до якої можна згодом провести частковий відкіт транзакції замість повної її відміни з поверненням до початкової точки. Точці збереження дається ім'я, яке указується в єдиному аргументі методу Save(). Переобтяжений

варіант методу Rollback() класу SqlTransaction приймає як аргумент ім'я точки збереження, до якої слід провести відкіт.

## Завдання для лабораторної роботи

1. На основі застосування, виконаного в лабораторній роботі №3, створити застосування, що використовує транзакції. Операції, що здійснюються в транзакціях, виконуються у відповідній наочній області (що задається викладачем) по аналогії з алгоритмом:

операція "вставка" запису батьківської таблиці повинна обов'язково забезпечувати додавання не менше одного запису дочірньої таблиці, інакше повинен виконуватися відкіт транзакції;

операція "оновлення" запису батьківської таблиці повинна обов'язково забезпечувати оновлення не менше одного запису дочірньої таблиці, інакше повинен виконуватися відкіт транзакції;

операція "видалення" запису батьківської таблиці повинна обов'язково забезпечувати видалення всіх зв'язаних записів дочірньої таблиці в діалоговому режимі, інакше повинен виконуватися відкіт транзакції, наприклад:

додайте MessageBox, який для кожного дочірньому запису, що видаляється, питає підтвердження видалення (оновлення), якщо, хоча б для одного запису натиснуто No, транзакція відкатується.

2. У кожній з операцій передбачити можливість використання проміжних точок відкоту транзакцій.

3. У застосуванні передбачити можливість установки рівнів ізоляції транзакцій. Досліджувати результати роботи застосування для різних рівнів ізоляції транзакцій, передбачивши всі можливі випадки аварійного завершення роботи застосування.

## Контрольні питання

1. Опис і створення об'єкта Transaction.
2. Властивості транзакцій.
3. Види транзакцій.
4. Методи класу Transaction.
5. Рівні ізоляції транзакцій в ADO.NET.
6. Проміжні точки відкоту транзакцій.

## Лабораторна робота 6. Розробка програм взаємодії з базами даних з використанням розподілених транзакцій і принципів паралелізму

**Мета лабораторної роботи:**

1. Придбання практичних навиків використання розподілених транзакцій при оновленні даних в ієрархічно зв'язаних таблицях.

2. Вдосконалення навиків роботи з інтегрованим середовищем розробки Visual C# і довідковою системою Microsoft Developer Network (MSDN).

**Перед виконанням лабораторної роботи студент повинен знати:**

1. Основи використання бібліотеки Windows Forms.

2. Організацію функціонування СУБД в розрахованому на багато користувачів середовищі.

**Після виконання лабораторної роботи студент повинен уміти:**

1. Самостійно розробляти прості застосування для роботи в розрахованому на багато користувачів середовищі з використанням розподілених транзакцій.

2. Застосовувати принципи паралелізму при реалізації операцій оновлення бази даних.

### Основні відомості про роботу застосувань з використанням розподілених транзакцій

Транзакція, що охоплює декілька ресурсів, називається розподіленою транзакцією. Наприклад, фінансові переліки між рахунками банків є розподілена транзакція.

Найбільш важливі учасники в розподілених транзакціях це:

диспетчера ресурсів (resource manager – RM), що безпосередньо виконують всю роботу і що повідомляють про успішне або невдале її завершення;

диспетчер транзакцій або координатор розподілених транзакцій (Distributed Transaction Coordinator – DTC).

Координатор транзакцій, який поставляється з Windows,

називається координатором розподілених транзакцій Microsoft (Microsoft Distributed Transaction Coordinator – MSDTC) є служба Windows, яка виконує координацію транзакцій для розподілених застосувань.

Стандартний алгоритм виконання розподіленої транзакції:

застосування починає транзакцію, запрошуючи її у MSDTC, це застосування звичайно називається ініціатором;

потім застосування пропонує RM виконувати свою роботу у складі цієї транзакції, і диспетчери ресурсів реєструються в диспетчері транзакцій як частина цієї ж транзакції, звичайно це називається включенням або занесенням (enlisting) в транзакцію;

якщо все нормально, то застосування фіксує транзакцію, якщо щось не так, кожен крок може виконати відкіт.

У будь-якому випадку, координатор транзакції координує роботу всіх RM, щоб забезпечити, що або всі вони завершилися успішно і виконали потрібну роботу, або всі вони відмінили результати своєї роботи.

Дуже зручним інструментом для роботи з розподіленими транзакціями являється клас TransactionScope простору імен System.Transactions. Наприклад, відкриття з'єднання в рамках класу TransactionScope спричинить за собою автоматичну вказівку цього з'єднання в транзакції.

Виклик методу Commit класу TransactionScope підтверджує виконувану транзакцією дію. При виклику методу Dispose цього класу (неявно в кінці блоку Using або явно) виконувана в транзакції непідтверджена дія автоматично відміняється.

Як працювати з безліччю з'єднань в класі TransactionScope і підтверджувати зміни розглянемо на прикладі додавання рядків у різні таблиці, причому додавання кожного рядка виконується по окремому підключенню до бази даних. Код прикладу приведений нижче:

```
static void Main(string[] args)
{
try {
using (TransactionScope myTransaction = new TransactionScope
())
{
using (SqlConnection connection1 =
```

```
new SqlConnection(connectionStringl))
{
connectionl.Open();
SqlCommand myCommand = connection1.CreateCommand();
myCommand.CommandText =
"Insert into Кредит(Сума_кредиту) Values (100) ";
myCommand.ExecuteNonQuery();
}
Console.WriteLine("Перша операція транзакції завершена" + ",
переходимо до другої." );
Console.ReadLine();
using (SqlConnection connection2 =

new SqlConnection(connectionString2))
{
connection2.Open ();
SqlCommand myCommand = connection2.CreateCommand ();
myCommand.CommandText =
"Insert into Дебіт(Сума_дебіту) Values (100)" ;
myCommand.ExecuteNonQuery ();
}
myTransaction.Complete() ;
} }
catch (System.Exception ex)
{
Console.WriteLine(ex.ToString());
 }
 }
```

У приведеному прикладі транзакція оформляється за допомогою блоку using. Застосування створює новий екземпляр TransactionScope, який визначає частину коду, що включається в транзакцію. І весь код між конструктором TransactionScope і викликом Dispose для цього екземпляра TransactionScope заноситься в створювану транзакцію.

Крім того, один з конструкторів TransactionScope дозволяє запропонувати рівень ізоляції для окремих RM з транзакції. Саме запропонувати, але не визначити, оскільки при необхідності RM може і

проігнорувати цю пропозицію і реалізувати вищий рівень ізоляції. Потім, за допомогою блоку using починають роботу різні RM – в нашому випадку два об'єкти SqlConnection, що підключаються до двох різних баз даних. У такий спосіб можна створити декілька компонентів, які включають себе в транзакцію MSDTC. Кожний з них викликає метод SetComplete при успішному завершенні або SetAbort при невдалому. Виклик SetComplete повідомляє координатора транзакцій, що зміни повністю готові і чекають інших. Якщо хто-небудь в ланцюжку викличе SetAbort, то всі оновлення відміняються, і жодна зміна не фіксується.

Після компіляції і запуску застосування можна встановити точку переривання в кінець виконання команди першого підключення ExecuteNonQuery. Якщо в цій крапці перейти в SQL Server Management Studio і виконати команду SQL:

```
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED
BEGIN TRANSACTION
SELECT Сума_кредиту FROM Кредит
COMMIT
```

,

то SQL Server 2005 повідомить, що поки вставлений тільки один рядок. Врахуйте, що ця вставка залежить від другої вставки, тобто ви бачите результат "брудного" читання.

Хай розподілена транзакція виконується на високому рівні ізоляції на зразок Serializable. Якщо відкрити аплет Control Panels^Administrative Tools^Component Services (Панель управління ^Средства адміністрування ^ Служба компонентів) і відкрити на лівій панелі список транзакцій, то під рядком координатора розподілених транзакцій (Distributed Transaction Coordinator) не буде нічого, поки не почне працювати підключення connection2.

При покроковому виконанні коду в режимі відладки, як тільки буде пройдений рядок connection2.0pen, в списку транзакцій DTC з'явиться транзакція. Це важливе поняття, яке звичайно називається підвищуване занесення (promotable enlistment) транзакції.

У процесі роботи безлічі користувачів з базою даних між вибіркою і збереженням рядків можуть відбутися деякі зміни:

оновлюваний рядок може бути видалений іншим користувачем;

у рядку, що додається, є відношення по зовнішньому ключу з іншим

рядком, який на той час може бути видалений іншим користувачем;

оновлюваний рядок вже оновлений іншим користувачем, але він не змінив стовпець, який потрібно змінити.

Розробнику необхідно перерахувати різні ситуації і вирішити, який підхід годиться краще всього в застосуванні, що розробляється. При цьому необхідно поклопотатися про методологію виявлення конфліктів і стратегії рішення паралелізму.

Один з найбільш ефективних інструментів для вирішення конфліктів паралелізму це перевірка відміток часу. У СУБД Microsoft SQL Server є стовпці типу timestamp – відмітки часу. Стовпець типу timestamp змінюється кожного разу при виконанні над рядком операції DML.

Запит UPDATE, що використовує стовпець timestamp, виглядає так:

**Update Товари Ціна = 60 where**

**Код_товару = 1 and TimeStamp = 0x00000000000007DI**

Щоб використовувати в таблиці стовпець Timestamp, його потрібно визначити під час створення таблиці:

**CREATE TABLE SomeTable (identifier int PRIMARY KEY, timestamp)**

Якщо для оновлення необхідно використовувати транзакцію, то для правильної роботи, додатку слід застосувати низький рівень ізоляції транзакцій, такий як ReadCommitted.

Тип даних timestamp (тимчасова мітка) автоматично генерує 8-байтове двійкове значення, що гарантовано є унікальним у межах бази даних.

Тип даних timestamp у мові T-SQL не співпадає з типом timestamp, визначеним у стандарті SQL-92. Тип timestamp, визначений у стандарті SQL-92, еквівалентний типу datetime в мові T-SQL.

У SQL Server для типу timestamp був введений синонім rowversion, і саме цей синонім рекомендується по можливості використовувати в операторах визначення даних. Відмінність між типом даних timestamp і типом-синонімом rowversion полягає в тому, що тип timestamp не вимагає імені стовпця, а тип rowversion – вимагає.

У таблиці може бути тільки один стовпець типу timestamp. Кожного разу, коли відбувається вставка або оновлення рядка із стовпцем timestamp, в нього записується поточне значення тимчасової мітки з бази даних, що повертається функцією @@DBTS. Використовування стовпця тимчасової мітки як частину ключа, особливо первинного ключа не рекомендується, оскільки при кожній вставці або оновленні доведеться перебудовувати індекс, що завдасть удару по продуктивності застосування.

Значення стовпця timestamp не зв'язане з часом і датою вставки або оновлення даних. Якщо потрібно зафіксувати час вставки або оновлення записів, то можна сформувати в таблиці стовпець з типом datetime і створити процедури вставки і оновлення, що встановлюють його значення.

### Завдання для лабораторної роботи

1. У різних філіалах фірми є бази даних, які реалізовані в SQL Server і Access (або Oracle) відповідно з однаковою схемою. Заповнення баз даних повинне виконуватися синхронно. Треба розробити застосування, що виконує одночасне оновлення, додавання і видалення рядків з таблиць обох баз даних. У базах даних повинна бути однакова інформація. Для забезпечення паралельної роботи декількох застосувань рекомендується використовувати стовпець timestamp.
2. Для перевірки правильності внесення даних у таблиці обох баз розробити застосування, яке періодично контролювало відповідність даних. У результаті перевірки в базах даних повинна бути збережена однакова інформація. При цьому наявність надмірної інформації або втрата даних недопустима. У звіті адміністратору повинні бути відображені відмінності, виявлені при перевірці.
3. Виконати тестування, використовуючи декілька екземплярів застосування для паралельної роботи при оновленні рядків у базах даних.

### Контрольні питання

1. Поняття розподіленої транзакції.
2. Функції диспетчера ресурсів при виконанні розподіленої транзакції.
3. Функції диспетчера транзакцій при виконанні розподіленої транзакції.
4. Привести приклади конфліктних ситуацій, які виникають в процесі роботи декількох користувачів з базою даних.
5. Способи боротьби з конфліктами в процесі роботи декількох користувачів з базою даних.

## Лабораторна робота 7. Розробка програм взаємодії з базами даних з використанням збережених процедур

**Мета лабораторної роботи:**

1. Придбання практичних навичок використання процедур, що зберігаються, при обробці даних в ієрархічно зв'язаних таблицях.

2. Вдосконалення навиків роботи з інтегрованим середовищем розробки Visual C# і довідковою системою Microsoft Developer Network (MSDN).

**Перед виконанням лабораторної роботи студент повинен знати:**

1. Основи використання бібліотеки Windows Forms.

2. Організацію функціонування СУБД в розрахованому на багато користувачів середовищі.

**Після виконання лабораторної роботи студент повинен уміти:**

1. Самостійно розробляти прості застосування для роботи в розрахованому на багато користувачів середовищі з використанням процедур, що зберігаються.

2. Застосовувати транзакції при реалізації операцій оновлення бази даних з використанням процедур, що зберігаються.

## Основні відомості про роботу застосувань з використанням процедур, що зберігаються

Процедура, що зберігається, – це одна або декілька SQL-конструкцій, які записані в базі даних. Основні причини використання процедур, що зберігаються, полягають в такому:

повністю заборонити доступ для звичайних запитів і забезпечити стабільність і надійність роботи;

швидке виконання запитів;

розбиття великих завдань на малі модулі;

зменшення навантаження на мережу.

Основні правила звернення до процедур, що зберігаються, використання параметрів під час виклику процедур, що зберігаються, і приклади їх застосування для оновлення даних у таблицях розглядалися в лабораторній роботі №4.

У справжній лабораторній роботі розглядаються питання сумісного використання транзакцій в ADO.NET і SQL Server за участю процедур, що зберігаються.

В процесі розробки застосувань особлива увага повинна

приділятися створенню надійного коду, який попереджає конфліктні ситуації, що виникають при оновленні даних. При використанні окремої програмної одиниці декількома додатками також можуть виникнути конфлікти оновлення даних. Такими програмними одиницями можна вважати процедури, що зберігаються в базі даних. Однієї з найпоширеніших помилок, що виникають в процесі відладки програм з використанням процедур, є невідповідність параметрів зухвалих модулів, що викликаються. Тому розглянемо приклад визначення параметрів процедури, що зберігається, в процесі виконання програми.

Рішення даної задачі можна виконати використовуючи метод **DeriveParameters()** об'єкта **CommandBuilder.** Код прикладу приведений нижче:

```
// Простори імен, змінні і константи
        using System;
        using System.Configuration;
        using System.Data;
        using System.Data.SqlClient;
private void parameters_Click(object sender. System.EventArgs e)
{
string procedureName = "UP_INSERT";
// створюємо таблицю, куди будуть поміщені результати
        DataTable dt = new DataTable();
// створюємо команду для процедури, що зберігається
        string connectionString = @"Data Source=.\SQLEXPRESS;
       AttachDbFilename=|DataDirectory|\
       Продажи_Товаров.mdf;Integrated Security=True;
       Connect Timeout=30;User Instance=True";
        SqlConnection conn = new SqlConnection();
        conn.ConnectionString = connectionString;
        SqlCommand cmd = new SqlCommand(procedureName, conn);
       cmd.CommandType = CommandType.StoredProcedure:
// одержуємо параметри
       conn.Open();
       SqlCommandBuilder.DeriveParameters(cmd);
        conn.Close();
```

```
// визначаємо стовпці таблиці з результатами
        dt.Columns.Add("Name");
        dt.Columns.Add ("Direction");
        dt.Columns.Add("SqlType");
// прочитуємо результати в таблицю з об'єкта Command
        foreach (SqlParameter param in cmd.Parameters)
dt.Rows.Add(new
object[]{param.ParameterName,param.Direction.ToString(),
param.SqlDbType.ToString()});
        dataGrid.CaptionText = "Параметри процедури, що зберігається" +
        procedureName + "одержані за допомогою
        CommandBuilder.DeriveParameters";
        dataGrid.DataSource = dt.DefaultView;
}
```

У приведеному прикладі створюється об'єкт **Command** для виконання процедури, що зберігається. Імена процедури, що зберігається, і підключення задаються в конструкторі об'єкта **Command**. Після відкриття підключення викликається метод **DeriveParameters()**, що записує інформацію про параметри процедури, що зберігається, в колекцію Parameters. Інформація про параметри витягується з колекції і виводиться на екран. В тому випадку, якщо вказана процедура, що зберігається, не існує, то генерується виключення **InvalidOperationException**.

У Microsoft SQL Server для визначення параметрів процедури, що зберігається, можна також застосовувати системні процедури, що зберігаються. Нижче приведений код рішення даної задачі, в якому використовується процедура **sp_sproc_columns**:

```
// будуємо команду, одержуючу параметри процедури, що зберігається
// за допомогою системної процедури SQL Server, що зберігається
        SqlCommand cmd = new SqlCommand("sp_sproc_columns",
        conn); cmd.CommandType = CommandType.StoredProcedure;
        SqlParameter param = cmd.Parameters.Add("@procedure_name",
        SqlDbType.NVarChar, 50);
        param.Value = procedureName;
// наповнюємо таблицю з результатами
        SqlDataAdapter da = new SqlDataAdapter(cmd);
```

```
da.Fill(dt);
dataGrid.CaptionText = "Параметри процедури, що зберігається" +
procedureName + "одержані з допомогою sp_proc_columns" ; }
//прив'язуємо таблицю до  елементу DataGrid
dataGrid.DataSource = dt.DefaultView; }
```

Приведений вище спосіб специфічний для Microsoft SQL Server. Системна процедура, що зберігається **sp_proc_columns** повертає інформацію про параметри однієї або декількох процедур, що зберігаються. На відміну від методу DeriveParameters(), з її допомогою не можна автоматично наповнити колекцію Parameters інформацією про параметри процедури, що зберігається. Зате вона повертає більше інформації, ніж метод **DeriveParameters()**, і дозволяє за один прийом одержувати відомості про декілька процедур, що зберігаються. Вона також підтримує фільтрацію і не вимагає створення об'єкта Command для процедури, що зберігається. Процедура **sp_proc_columns** повертає набір рядків, які відповідають параметрам процедури, що зберігається.

Для підвищення надійності програмного забезпечення в процесі створення процедур, що зберігаються, часто використовуються транзакції. Код застосування також може містити транзакцію. Таким чином, виникає завдання сумісного використання транзакцій в ADO.NET і SQL Server за участю процедур, що зберігаються. Нижче розглянутий приклад рішення цієї задачі при виконанні операції додавання запису в таблицю за допомогою процедури UP_Insert.

Процедура, що зберігається UP_Insert містить транзакцію, яка відновлює початковий стан при виникненні помилки (**@@error<>**), відсутності доданих рядків (**@@rowcount=0**) або ознака відкоту транзакції відмінна від 0 (**@Rollback=1**). Нижче приведений код процедури, що зберігається:

```
CREATE PROCEDURE UP_Insert
@Код_товара int output,
@Товар nvarchar(15),
@Ціна int,
@Rollback bit = 0
AS        SET NOCOUNT ON
begin tran
```

```
insert Товари( Товар, Ціна)
Values  (@Товар, @Ціна)
if @@error<>0 or @@rowcount=0 or @Rollback=1
begin
rollback tran
set @Код_товара = -1
return 1
end
commit tran
set @Код_товара = Scope_Identity()
select @Код_товара
return 0
```

Код застосування виконує додавання запису в таблицю бази даних використовуючи транзакцію і аналізує результат роботи транзакції в процедурі, що зберігається:

```
private SqlDataAdapter da;
private DataTable dt;
// створюємо підключення
string connectionString = @"Data Source=.\SQLEXPRESS; AttachDbFilename=|DataDirectory|\ Продажі_Товарів.mdf;Integrated Security=True;Connect Timeout=30;User Instance=True";
SqlConnection conn = new SqlConnection();
conn.ConnectionString = connectionString;
private void AdoSqlTranForm_Load(object sender, System.EventArgs e)
{
// наповнюємо даними таблицю
string sqlQuery = "SELECT Код_товара, Товар, Ціна " +
"FROM Товари";
da = new SqlDataAdapter(sqlQuery,conn);
dt = new DataTable("Товари");
da.Fill Schema(dt.SchemaType.Source);
da.Fill(dt);
// прив'язуємо таблицю до табличного елементу DataGrid
dataGrid.DataSource = dt.DefaultView;
}
private void insertButton_Click(object sender. System.EventArgs e)
```

```
        {  // починаємо транзакцію
        conn.Open();
        SqlTransaction tran = conn.BeginTransaction();
// створюємо в транзакції команду з параметрами
        SqlCommand cmd = new SqlCommand("UP_Insert ", conn, tran);
        cmd.CommandType = CommandType.StoredProcedure;
        cmd.Parameters.Add( "@Код_товара", SqlDbType.Int), Direction =
        ParameterDirection.Output;
        cmd.Parameters.Add( "@Товар", SqlDbType.NVarChar, 15);
        cmd.Parameters.Add( "@Цена", SqlDbType. Int);
        cmd.Parameters.Add( "@Rollback", SqlDbType.Bit);
```

У даному прикладі передбачається, що для введення значень параметрів необхідно, щоб на формі були створені такі компоненти: **товарTextBox**, **ціна TextBox**, **rollbackCheckBox**.

```
        try
{
 // встановлюємо значення параметрів
        if(товарTextBox.Text.Trim(). Length == 0)
        cmd.Parameters["@Товар"].Value = DBNull.Value;
        else
        cmd.Parameters["@Товар"].Value = товарTextBox.Text;
        cmd.Parameters["@Цена"].Value = ценаTextBox.Text;
cmd.Parameters["@Rollback"].Value=rollbackCheckBox.Checked ? 1:0;
// додавання запису в таблицю
        cmd.ExecuteNonQuery();

// додавання успішно виконане – зберігаємо транзакцію
        tran.Commit();
        MessageBox.Show("Транзакція збережена.");
 }
        catch (SqlException ex)
        {  bool spRollback = false;
        foreach (SqlError err in ex.Errors)    {
 // перевіряємо, чи відбувся відкіт транзакції в процедурі
        if(err.Number == 266), що зберігається
        {  MessageBox.Show(ex.Message,
 "Відбувся відкіт транзакції СУБД в процедурі, що зберігається",
```

```
MessageBoxButtons.OK, MessageBoxIcon.Error);
        spRollback = true;
        break; } }
        if (spRollback)
       {
// відкоту транзакції в СУБД не було, помилка
// SqlException проводимо відкіт транзакції
        tran.Rollback();
        MessageBox.Show(ex.Message); } }
       catch (Exception ex)
        {
// інше виключення – проводимо відкіт транзакції
        tran.Rollback();
        MessageBox.Show(ex.Message); }
        finally  {  conn.Close();  }
       da.Fill(dt);
       }
```

SQL Server повертає код помилки 266, якщо при завершенні процедури що зберігається, число транзакцій не співпадає з тим, яке було при її ініціалізації. Це число повертається функцією **@@TRANCOUNT**. При виникненні помилки СУБД відправляє повідомлення клієнту і це не впливає на виконання процедури, що зберігається.

При виклику процедури, що зберігається, з ручної транзакції .NET у момент входу в процедуру число транзакцій дорівнює 1. За допомогою SQL-оператора **BEGIN TRAN** у процедурі, що зберігається, створюється вкладена транзакція, яка збільшує кількість транзакцій до 2. Якщо ця транзакція зберігається командою **COMMIT TRAN**, число транзакцій знову зменшується до 1.

Дуже важливо відзначити, що збереження внутрішніх транзакцій не звільняє ресурси і не проводить незворотних змін, а також не впливає на зовнішні транзакції. Крім того, якщо дається команда **ROLLBACK**, проводиться відкіт усіх транзакцій, вкладених у транзакцію верхнього рівня, і число транзакцій зменшується до 0. Спроба зберегти або відмінити транзакцію з коду .NET після того, як був проведений її відкіт у процедурі, що зберігається, приведе до виникнення виключення

**InvalidOperationException**, оскільки транзакція вже відмінена.

У прикладі, при виконанні процедури що зберігається, перехоплюються виключення і перевіряються на відповідність помилці SQL Server номер 266 (невідповідність між числом транзакцій на вході і виході з процедури, що зберігається, в результаті відкоту транзакції в процедурі, що зберігається). Якщо процедура, що зберігається, провела відкіт транзакції, код .NET не виконує відкіт. Усі інші помилки, що виникають при виконанні процедури, що зберігається, приводять до відкоту транзакції кодом .NET.

### Завдання для лабораторної роботи

1. Розробити застосування забезпечуюче відображення параметрів всіх процедур бази даних, що зберігаються, за допомогою методу DeriveParameters() і системної процедури, що зберігається sp_sproc_columns. Виконати порівняння цих способів рішення задачі.

2. Розробити застосування, що виконує оновлення, додавання і видалення даних з таблиць бази даних з використанням транзакції. Забезпечити виконання цих операцій з використанням процедур, що зберігаються, кожна з яких містить транзакцію.

3. Виконати тестування, використовуючи декілька екземплярів застосування для паралельної роботи при оновленні рядків у базі даних.

### Контрольні питання

1. Дати визначення процедури, що зберігається.
2. Виклик процедури, що зберігається.
3. Параметри процедури, що зберігається.
4. Методи визначення параметрів процедури, що зберігається, в процесі виконання програми.
5. Особливості сумісного використання транзакцій ADO.NET і SQL Server.

## Лабораторна робота 8. Розробка програм створення таблиць і пошуку даних XML

**Мета лабораторної роботи:**

1. Придбання практичних навичок використання наборів даних XML і XSD файлів, що строго типізуються.

2. Придбання практичних навиків для роботи з XML і XSD файлами.

3. Вдосконалення навиків роботи з інтегрованим середовищем розробки Visual C# і довідковою системою Microsoft Developer Network (MSDN).

**Перед виконанням лабораторної роботи студент повинен знати:**

1. Основи використання бібліотеки Windows Forms.
2. Організацію функціонування СУБД в розрахованому на багато користувачів середовищі.

**Після виконання лабораторної роботи студент повинен уміти:**

Самостійно розробляти прості застосування для роботи з XML і XSD файлами.

## Основні відомості про роботу застосувань з використанням XML-файлів

XML-файл – це звичайний текстовий файл, який має певну структуру. ADO.NET дає можливість безпосередньо працювати з даними в XML-форматі. Цей формат став настільки поширений, що важко знайти місце, де не було б можливостей по обробці XML-файлів.

Дані в XML-файлі можуть бути структуровані за допомогою XML-схеми або XSD- документа (XML Structure Definitions).

XML схеми містять таку інформацію:

опис зв'язків між полями даних, це дозволяє зберігати інформацію про зв'язки в реляційній базі даних;

опис обмежень на поля даних, що дозволяє зберігати інформацію про первинні ключі і унікальність полів реляційної моделі;

специфікацію типу даних для кожного індивідуального елементу і атрибуту в XML-документі, який створюється на основі XML -схеми.

Таким чином, XML-схема – це XML-документ, що визначає структуру інших XML-документів за допомогою опису структур і типів елементів, які можуть бути використані в цих документах.

Розглянемо простий приклад XML-схеми:

```
<xs:element name="Ім’я">
 <xs:complexType>
      <xs:attribute name="Товар" type="xs:string" />
      <xs:attribute name="Час" type="xs:time" />
      <xs:attribute name="Ознака" type="xs:boolean" />
      <xs:attribute name="Ціна" type="xs:decimal" />
         </xs:complexType>
 </xs:element>
```

XML-код використовує приведену вище схему:

```
<Ім’я Товар="Хліб"
      Час="12:00"
      Ознака="true"
      Ціна="3.85" />
```

Схема указує, що елемент з ім'ям <Ім’я> може існувати в пов'язаному з нею XML-документі, і що цей елемент повинен мати чотири атрибути різних типів.

Використовуючи XSD-документ можна пересилати дані з DataSet в XML-форматі. На основі XML-документа можна створити об'єкт DataSet, але для цього необхідно знати структуру даних, яка і міститься в XSD-схемі. Таким чином, при передачі даних по мережі, використовується два файли – XML, який містить дані, і XSD, який містить структуру даних, зв'язки між ними, а також обмеження.

Об'єкти типу DataSet, для яких визначена структура, називаються такими, що типізуються. Для створення таких об'єктів DataSet слід додати в проект клас, успадкований від DataSet, який слід згенерувати на основі інформації зі схеми. Після чого можна створювати об'єкти нового класу, які і будуть об'єктами DataSet, що типізуються.

У .NET є два підходи до створення тих, що строго типізуються DataSet:

створити візуально у конструкторі за допомогою засобів Visual Studio .NET;

користуватися утилітами командного рядка і аргументами компілятора.

Створення того, що строго типізується DataSet в Visual Studio .NET здійснюється в такій послідовності:

створити нове застосування;

клацніть правою кнопкою на імені проекту, виберіть пункт Add^Add New Item, а потім виберіть значок DataSet і введіть в поле його імені;

перетягніть з палітри інструментів на порожнє місце у вікні конструктора два елементи DataTable, з цієї миті можна користуватися візуальним інтерфейсом, щоб додати стовпці для того, що кожного знаходиться у вікні DataTable;

створіть стовпці відповідно до стовпців, які є в таблицях бази даних (рис. 4.1);

на завершення створіть первинний ключ з ім'ям KeyКод_Товара для стовпця Код_Товара таблиці Товар.

тепер є дві повністю відособлені і не зв'язані між собою таблиці, для створення відношення перетягніть з палітри інструментів у вікно конструктора елемент Relation, протягаючи його над якими-небудь елементами таблиці "Продажу", коли ви відпустите кнопку, з'явиться діалог, в якому буде запропоновано ввести додаткову інформацію про відношення (keyref); змініть ім'я keyref на KeyКод_ТовараRef, оскільки в одній з таблиць вже є ключ і в обох таблицях є стовпці з однаковими іменами, діалог заповнюється всією інформацією, необхідною для формування батьківського відношення після закриття діалогового вікна згенерує відповідний код;

у вікні Properties (Властивості) змініть властивість targetName-space схеми на ІМ’Я_СХЕМИ.xsd, а attributeFormDefault на (Default).

Тепер конструктор виглядатиме приблизно так, як показано на рис. 4.1.

При використанні утиліт командного рядка і аргументів компілятора створення того, що строго типізується DataSet здійснюється в такій послідовності:

створити новий каталог з ім'ям і скопіювати в нього створений файл ИМ’Я_СХЕМИ.xsd, який містить визначення таблиці "Товар", таблиці "Продажу", а також батьківського відношення keyref;

написати пакетний файл для створення того, що строго типізується DataSet, ім'я файла CreateDs.cmd, а його вміст приведений нижче:

**xsd /d /1:CS ТовариDataSet.xsd**

пакетний файл викликає утиліту XSD. exe, яка генерує файли схем або класів з вказаного джерела, щоб цей пакетний файл зміг працювати, потрібно виконати його з того ж каталога, де знаходиться xsd;

файл приймає схему і або перетворює її в класи, або створює з неї що типізується DataSet (аргумент /d);

версія /1 (C#) дозволяє вказати мову, на якій потрібно згенерувати той, що строго типізується DataSet.

Розглянемо приклад створення XML і XSD-документів для бази даних, що мають схему, представлену на рис. 4.1.

Елементи **<keyref>** в XML-схемі дозволяють оголошувати посилання усередині документа. Ця можливість по суті аналогічна відносинам по зовнішньому ключу в реляційних СУБД. Розглянемо приклад формування відносин на основі даних, приведених нижче:

```
<xs:element name="ТоварыDataSet" msdata:IsDataSet="true">
<xs:complexType>
 <xs:choice maxOccurs="unbounded">
 <xs:element name="Товари">
 <xs:complexType>
 <xs:sequence>
<xs:element name="Код товара" type="xs:integer" minOccurs="1./>
<xs:element name="Товар" type="xs:string" />
 <xs:element name="Ціна" type="xs:integer" minOccurs="0" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 <xs:element name="Продажі">
 <xs:complexType>
 <xs:sequence>
<xs:element name="Код товара" type="xs:integer" minOccurs="1" />
<xs:element name="Кількість" type="xs:integer" minOccurs="1" />
 <xs:element name="Вартість" type="xs:integer" minOccurs="0" />
 </xs:sequence>
 </xs:complexType>
 </xs:element>
 </xs:choice>
 </xs:complexType>
```

У даному прикладі створюються дві нові таблиці для нашого DataSet, які називаються "Товари" і "Продажі" відповідно. У кожній з таблиць елементом схеми є стовпець певного типу.

Визначимо поле "Код_товара" як унікальне. Нижче показаний приклад визначення унікального поля:

```
<xs:unique
msdata:ConstraintName="Код" name="UniqueКодConstr" >
 <xs:selector xpath=".//Товари" />
<xs:field xpath="Код_товара" />
</xs:unique>
```

Це оголошення показує, що ми вимагаємо унікальність даних у колонці "Код_товара", таблиці "Товари" і це обмеження має ім'я "Код".

Тепер за допомогою елементів <key> і <keyref> створимо відношення по зовнішньому ключу між таблицями "Товари" і "Продажи".

Для цього як зовнішній ключ у таблиці "Товари" – стовпець "Код товара":

```
<xs:key name="KeyТовар">
 <xs:selector xpath=".//Товари" />
 <xs:field xpath="Код_товара" />
 </xs:key>
```

.

Елемент <keyref > указує, що створюється обмеження по зовнішньому ключу для стовпця "Код_товара" таблиці "Продажу", яке посилається на елемент <key> з ім'ям Key Товар:

```
<xs:keyref name="KeyТоварRef" refer="KeyТовар">
 <xs:selector xpath=".//Продажі" />
 <xs:field xpath="Код_товара" />
 </xs:keyref>
 </xs:element>
```

Приведемо приклад таблиці "Товари"у файлі ПродажіТоварів.xml:

```
<ТовариDataSet xmlns="urn: ТовариDataSet.xsd">
 <Товари>
 < Код_товара >1</ Код_товара >
 <Товар> Хліб </Title>
 <Ціна>3</Ціна>
 </Товари>
 <Товари>
 < Код_товара >2</ Код_товара >
 <Title> Булка </Title>
```

**<Ціна>2</Ціна>**
**</Товари>**

Кожен рядок даної таблиці визначається элементом <Товари>. Приведемо приклад вмісту таблиці "Продажі" у файлі ПродажіТоваров.xml, у якій кожен рядок визначається елементом <Продажі>:

**< Продажі >**
**<Код_товара>1</Код_товара>**
**<Кількість>5</Кількість>**
**< Вартість >15</ Вартість >**
**</Продажі>**
**<Продажі>**
**< Код_товара >2</ Код_товара >**
**< Кількість >20</ Кількість >**
**< Вартість >40</ Вартість >**
**</Продажі>**
**<Продажі>**
**< Код_товара >2</ Код_товара >**
**< Кількість >4</ Кількість >**
**< Вартість>8</ Вартість >**
**</Продажи>**
**<Продажи>**
**<Код_товара>2</Код_товара>**
**<Кількість>16</Кількість>**
**< Вартість >32</ Вартість >**
**</Продажі>**
**</ТовариDataSet>**

Тепер можна завантажити XSD-файл в об'єкт DataSet, щоб створити його структуру у вигляді таблиць і зв'язків, а також різного роду обмежень.

Метод ReadXmlSchema може приймати як параметри шлях до файла, що містить схему XML, об'єкти класів TextReader або XmlReader, а також об'єкт типу Stream.

**DataSet.ReadXmlSchema(string I Stream I TextReader | XmlReader);**

Для перевірки правильності завантаження структури використовуються такі властивості:

Tables – використовується для доступу до колекції таблиць;

Relations – використовується для доступу до колекції зв'язків;

Tables.Count – повертає число таблиць;

Tables-[index].TableName – повертає ім'я указаннной таблиці;

Tables[tablename | index].Columns[index] – дозволяє дістати доступ до списку колонок у заданій таблиці;

Tables[tablename | index].Columns[index].ColumnName – дозволяє одержати ім'я колонки;

Tables[tablename | index]. Columns [index]. DataType – містить тип даних колонки;

Tablesftablename | index].Columns.Count – повертає число колонок таблиці.

З допомогою засобів ADO.NET можна зробити висновки про структуру за XML-файлом. Якщо ви маєте XML-файл, але не маєте схеми, то ви можете створити схему, базуючись на структурі XML-файла. В цьому випадку схема не завжди відповідатиме дійсності, оскільки багато даних можна трактувати по-різному.

Наступним етапом після завантаження схеми є завантаження даних в об'єкт типу DataSet. Для читання даних з XML-файла використовується метод класу DataSet - ReadXml.

**DataSet.ReadXml (string I Stream I TextReader | XmlReader [XmlReadMode]);**

Другий параметр і визначає поведінку методу ReadXml по відношенню до схеми, яка може розташовуватися усередині XML-файла. Значення типу XmlReadMode за замовчуванням містить Auto і може приймати одне із значень:

Auto – це значення рівносильно значенню ReadSchema, якщо XML або DataSet містять схему, інакше використовується InferSchema;

DiffGram – використовується для злиття даних з XML-файла і DataSet, дані будуть додані в DataSet, якщо їх структура співпадає, інакше буде виключення;

Fragment – це значення дозволяє прочитувати дані, які були вибрані з бази даних за допомогою спеціальної команди FOR XML, яка використовується спільно з SELECT, це значення працює тільки для SQL

Server;

IgnoreSchema – ігнорує вбудовану в XML схему і просто прочитує дані, всі дані, які не відповідатимуть структурі DataSet, ігноруються;

InferSchema – ігнорує будь-яку схему на початку XML і створює схему на основі даних;

ReadSchema – завантажує схему, а потім дані, якщо схема вже існує, то до неї будуть додані нові таблиці, якщо таблиця вже існує в DataSet, то буде виключення для кожної такої таблиці, якщо схеми немає і DataSet не містить структури, то дані не будуть прочитані.

Наведемо приклад завантаження DataSet з файлів ПродажіТоварів.xsd і ПродажіТоварів.xml. :

```
DataSet ТовариDataSet = new DataSet();
ТовариDataSet.ReadXmlSchema( "ПродажіТоварів.xsd" );
ТовариDataSet.ReadXml( " ПродажіТоварів.xml");
Console.WriteLine( "Створені відносини:" );
foreach (DataRelation relation in ТовариDataSet.Relations) {
Console.WriteLine(relation.RelationName);
 }
Console.WriteLine();
```

Далі виконується перебір всіх рядків у DataSet, і для кожного рядка виводиться значення стовпця "Товар":

```
foreach (DataRow row in ТовариDataSet.Tables[ "Товари" ].Rows)
{
Console.WriteLine("{0}", row["Товар"]);
// Вибірка дочірніх рядків за допомогою відношення KeyТоварRef
foreach (DataRow rrow in xRow.GetChildRows( "KeyТоварRef "))
{
Console.WriteLine (" {0}", rrow[ "Кількість" ]);
}
}
```

Метод GetChildRows() одержує на основі відношення список дочірніх рядків даного рядка. При цьому можна вказати або ім'я відношення, або об'єкт DataRelation.

Для створення схем з об'єкта типу DataSet використовують два методи:

WriteXmlSchema() – зберегти структуру об'єкта DataSet в XSD-файлі,

потоці або в об'єкті типу Reader;

GetXmlSchema() – зберегти інформацію про схему в об'єкт типу String, є окремим випадком WriteXmlSchema.

Нижче приведений приклад збереження схеми, яка створена на основі даних XML-файла:

```
string xmlfile= @"c:\ПродажіТоварів.xml";
string schema = @"c:\Друга_схема.xsd";
DataSet data=new DataSet ();
data.ReadXml(xmlfile,XmlReadMode.InferSchema);
data.WriteXmlSchema(schema);
Console.WriteLine(data.GetXmlSchema());
```

Для запису даних використовується метод WriteXml(), загальний вид якого приведений нижче:

```
public void WriteXml (string | Stream | TextReader | XmlReader
[XmlWriteMode]);
```

Параметр XmlWriteMode може приймати одне з таких значень:

IgnoreSchema – записує тільки дані;

WriteSchema – записує дані і схему;

DiffGram – створює XML-файл у формі DiffGram, цей формат дозволяє зберігати в XML-файлі зміни, які були зроблені з об'єктом типу DataSet.

Нижче приведений приклад збереження даних і схеми, яка створена на основі даних XML-файла:

```
string xmlfile= @"c:\ ПродажіТоварів.xml";
string xmlfilenew = @"c:\Новий_файл.xml";
DataSet data=new DataSet();
data.ReadXml(xmlfile, XmlReadMode.InferSchema);
data.WriteXml(xmlfilenew,XmlWriteMode.WriteSchema);
```

## Завдання для лабораторної роботи

1. Створити базу даних у форматі XML за завданням лабораторної роботи №4.

2. Розробити застосування забезпечуючи відображення даних таблиці на формі в наочному вигляді. При виділенні запису в батьківській таблиці на формі повинні відображатися всі залежні записи дочірньої таблиці.

3. Застосування повинне виконувати оновлення, додавання і видалення даних з таблиць.

## Контрольні питання

1. DataSet, що строго типізується.
2. Створення DataSet, що строго типізується.
3. Створення відносин між таблицями в XML-схемі.
4. Методи читання даних з XML і XSD-файлів.
5. Методи збереження даних у XML і XSD-файлах.

# Рекомендована література

1. C# и платформа .Net / Э. Троелсен; – пер. с англ. – СПб. : Питер, 2007 – 796 с.

2. Троелсен Э. Язык программирования C# 2005 и платформа .Net 2.0 / Э. Троелсен; – пер. с англ. – М. : Издательский дом "Вильямс", 2007 – 1168с.

3. .NET Framework. Секреты создания Windows-приложений, / С. С. Байдачный; – М. : СОЛОН-Пресс, 2004. – 496 с.

4. Программирование на Microsoft ADO.NET. Мастер – класс / Д. Сеппа; пер. с англ. – СПб. : Питер, 2007. – 764с.