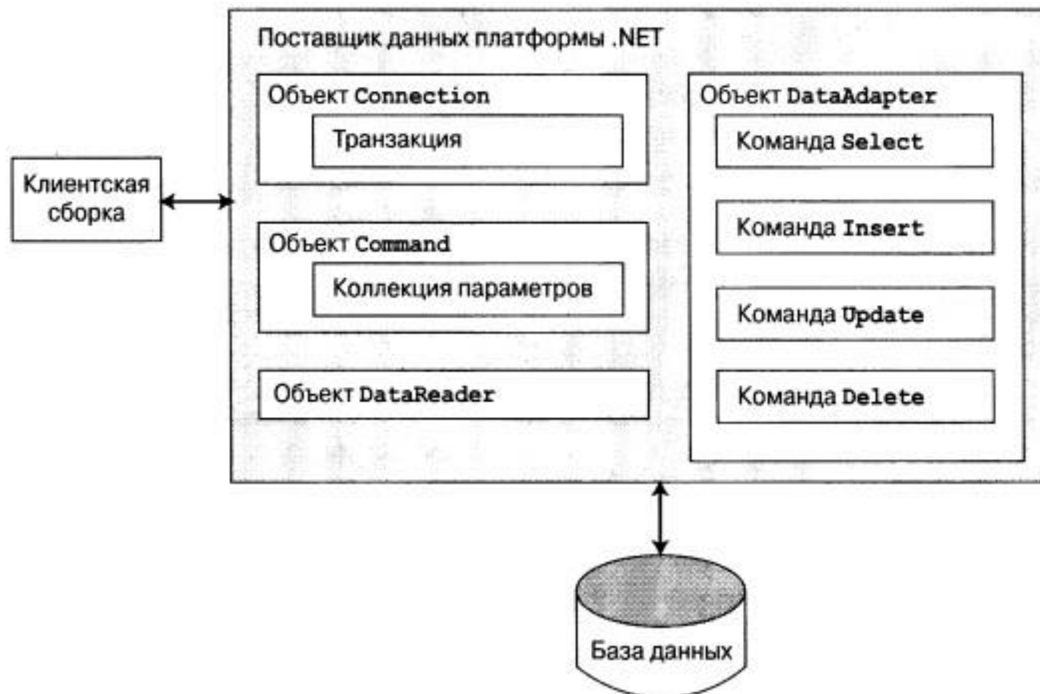
Поставщики данных ADO.NET

В ADO.NET имеются различные поставщики данных, каждый из которых оптимизирован для взаимодействия с конкретной СУБД.

Поставщик данных	Пространство имен	Сборка
OLEDB	System.Data.OleDb	System.Data.dll
Microsoft SQL Server	System.Data.SqlClient	System.Data.dll
Microsoft SQL Server Mobile	System.Data.SqlServerCe	System.Data.SqlServerCe.dll
ODBC	System.Data.Odbc	System.Data.dll



Поставщики данных ADO.NET

Независимо от используемого поставщика данных, каждый из них определяет набор классов, обеспечивающих основную функциональность.

Тип объекта	Базовый класс	Соответствующие интерфейсы	Назначение
Connection	DbConnection	IDbConnection	Позволяет подключаться к хранилищу данных и отключаться от него. Кроме того, объекты подключения обеспечивают доступ к соответствующим объектам транзакций
Command	DbCommand	IDbCommand	Представляет SQL-запрос или хранимую процедуру. Кроме того, объекты команд предоставляют доступ к объекту чтения данных конкретного поставщика данных
DataReader	DbDataReader	IDataReader, IDataRecord	Предоставляет доступ к данным только для чтения с помощью курсора на стороне сервера
DataAdapter	DbDataAdapter	IDataAdapter, IDbDataAdapter	Передает наборы данных между вызывающим процессом и хранилищем данных. Адаптеры данных содержат подключение и набор из четырех внутренних объектов команд для выборки, вставки, изменения и удаления информации в хранилище данных
Parameter	DbParameter	IDataParameter, IDbDataParameter	Представляет именованный параметр в параметризованном запросе
Transaction	DbTransaction	IDbTransaction	Инкапсулирует транзакцию в базе данных

Основные объекты поставщиков данных ADO.NET

Для взаимодействия с файлом данных Access, можно использовать поставщик данных OLE DB или ODBC.

Поставщик данных OLE DB нужен только для взаимодействия с какой-нибудь СУБД, для которой нет специального поставщика данных .NET. Однако поставщик OLE DB неявно взаимодействует с различными COM-объектами, что может снизить производительность приложения.

Поставщик данных Microsoft SQL Server предоставляет прямой доступ к хранилищам данных Microsoft SQL Server и только к хранилищам данных SQL Server версии 7.0 или больше.

Поставщики System.Data.Odbc и System.Data.SqlClientCe обеспечивают взаимодействие с ODBC-подключениями и доступ к SQL Server версии Mobile

В предыдущих версиях платформы .NET имелась сборка System.Data.OracleClient.dll, которая предоставляла поставщик данных для взаимодействия с базами данных Oracle.

Oracle поставляет собственную сборку .NET, которая разработана на тех же общих принципах, что и поставщики данных, предоставляемые Microsoft. Эту сборку можно загрузить с официального веб-сайта Oracle http://www.oracle.com

Пространство имен System.Data

System.Data содержит типы, представляющие различные примитивы баз данных (например, таблицы, строки, столбцы и ограничения), а также общие интерфейсы, реализованные объектами поставщиков данных.

Тип	Назначение
Constraint	Ограничение для данного объекта DataColumn
DataColumn	Один столбец в объекте DataTable
DataRelation	Отношение "родительский-дочерний" между двумя объектами DataTable
DataRow	Одна строка в объекте DataTable
DataSet	Находящийся в памяти кэш данных, который состоит из любого количества взаимосвязанных объектов DataTable
DataTable	Табличный блок данных, находящихся в памяти
DataTableReader	Позволяет обращаться с DataTable как с примитивным курсором (доступ к данным только для чтения и только в прямом направлении)
DataView	Специализированное представление DataTable для сортировки, фильтрации, поиска, редактирования и навигации
IDataAdapter	Определяет основное поведение объекта адаптера данных
IDataParameter	Определяет основное поведение объекта параметра
IDataReader	Определяет основное поведение объекта чтения данных
IDbCommand	Определяет основное поведение объекта команды
IDbDataAdapter	Расширяет IDataAdapter для получения дополнительных возможностей объекта адаптера данных
IDbTransaction	Определяет основное поведение объекта транзакции

Роль интерфейса IDbConnection

Тип IDbConnection реализован объектом подключения поставщика данных. Этот интерфейс определяет набор членов, применяемых для настройки подключения к конкретному хранилищу данных.

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

Перегруженный метод BeginTransaction() предоставляет доступ к объекту транзакции поставщика.

Роль интерфейса IDbTransaction

Члены, определенные в IDbTransaction, позволяют программным образом взаимодействовать с сеансом транзакций и соответствующим хранилищем данных.

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

Роль интерфейса IDbCommand

Интерфейс IDbCommand реализуется объектом команды поставщика данных, позволяющим программно работать с операторами SQL, хранимыми процедурами и параметризованными запросами. Также объекты команды обеспечивают доступ к типу чтения данных поставщика данных с помощью перегруженного метода ExecuteReader()

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

Роль интерфейсов IDbDataParameter и IDataParameter

Функциональность интерфейсов IDbDataParameter и IDataParameter позволяет использовать параметры в командах SQL (в том числе и в хранимых процедурах) с помощью особых объектов параметров ADO.NET вместо жестко закодированных строковых литералов.

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

Интерфейс IDbDataParameter расширяет интерфейс IDataParameter с целью получения дополнительных возможностей.

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

Роль интерфейсов IDbDataAdapter и IDataAdapter

Адаптеры данных используются для выборки и занесения наборов данных DataSet в конкретное хранилище данных.

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

Интерфейс IDataAdapter определяет основную функцию типа адаптера данных: возможность пересылать объекты DataSet между вызывающим процессом и непосредственным хранилищем данных с помощью методов Fill() и Update()

```
public interface IDataAdapter
{
    MissingMappingAction MissingMappingAction { get; set; }
    MissingSchemaAction MissingSchemaAction { get; set; }
    ITableMappingCollection TableMappings { get; }
    int Fill(System.Data.DataSet dataSet);
    DataTable[] FillSchema(DataSet dataSet, SchemaType schemaType);
    IDataParameter[] GetFillParameters();
    int Update(DataSet dataSet);
}
```

Роль интерфейса IDbReader

IDataReader представляет общие функции, поддерживаемые конкретным объектом чтения данных. Получив от поставщика данных ADO.NET тип, совместимый с IDataReader, можно просматривать результирующий набор, но только в прямом направлении и только просматривать.

```
public interface IDataReader : IDisposable, IDataRecord
{
    int Depth { get; }
    bool IsClosed { get; }
    int RecordsAffected { get; }
    void Close();
    DataTable GetSchemaTable();
    bool NextResult();
    bool Read();
}
```

Роль интерфейса IDbRecord

В интерфейсе IDataRecord определено значительное количество членов, позволяющих сразу извлекать из потока строго типизированные значения, а не приводить к нужному типу обобщенный тип System.Object, который получен от перегруженного метода индексатора из объекта чтения данных

```
public interface IDataRecord
{
    int FieldCount { get; }
    object this [string name] { get; }
    object this [int i] { get; }
    bool GetBoolean(int i);
    byte GetByte(int i);
    char GetChar(int i);
    DateTime GetDateTime(int i);
    Decimal GetDecimal (int i);
    float GetFloat(int i);
    short GetInt16(int i);
    int GetInt32(int i);
    long GetInt64(int i);
    ...
    bool IsDBNull(int i);
}
```

Метод IDataReader.IsDBNull() позволяет программным способом узнать, установлено ли в конкретном поле значение null, прежде чем выбрать значение из объекта чтения данных.

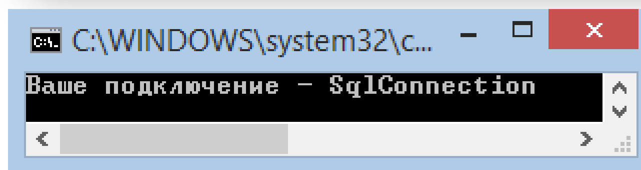
Абстрагирование поставщиков данных с помощью интерфейсов

Несмотря на то, что точные имена реализованных типов отличаются в различных поставщиках данных, в программах они используются однотипно – в этом вся прелесть полиморфизма на основе интерфейсов.

```
// Список возможных поставщиков.
enum DataProvider
{ SqlServer, OleDb, Odbc, Oracle, None }
// Этот метод возвращает конкретный объект подключения
//на основе значения перечисления DataProvider.
static IDbConnection GetConnection(DataProvider dp)
{
    IDbConnection conn = null;
    switch (dp)
    {
        case DataProvider.SqlServer:
            conn = new SqlConnection();
            break;
        case DataProvider.OleDb:
            conn = new OleDbConnection();
            break;
        case DataProvider.Odbc:
            conn = new OdbcConnection();
            break;
    }
    return conn;
}
```

Преимущество работы с обобщенными интерфейсами из System.Data (т.е. с абстрактными базовыми классами из System.Data.Common) состоит в том, что при этом имеется гораздо больше возможностей для создания гибкой кодовой базы, которую со временем можно развивать.

```
static void Main(string[] args)
{
    // Получение конкретного подключения.
    IDbConnection myCn = GetConnection(DataProvider.SqlServer);
    Console.WriteLine("Ваше подключение - {0}", myCn.GetType().Name);
    // Открытие, использование и закрытие подключения...
    Console.ReadLine();
}
```



Конфигурационные файлы приложения

Для повышения гибкости приложений ADO.NET можно использовать клиентский файл *.config, элемент <appSettings> которого может содержать произвольные пары ключ/значение.

```
<configuration>
   <appSettings>
      <!-- Это значение ключа отображается на одно из значений перечисления -->
      <add key="provider" value="SqlServer"/>
   </appSettings>
</configuration>
```

Можно изменить метод Main(), чтобы программно получить соответствующий поставщик данных.

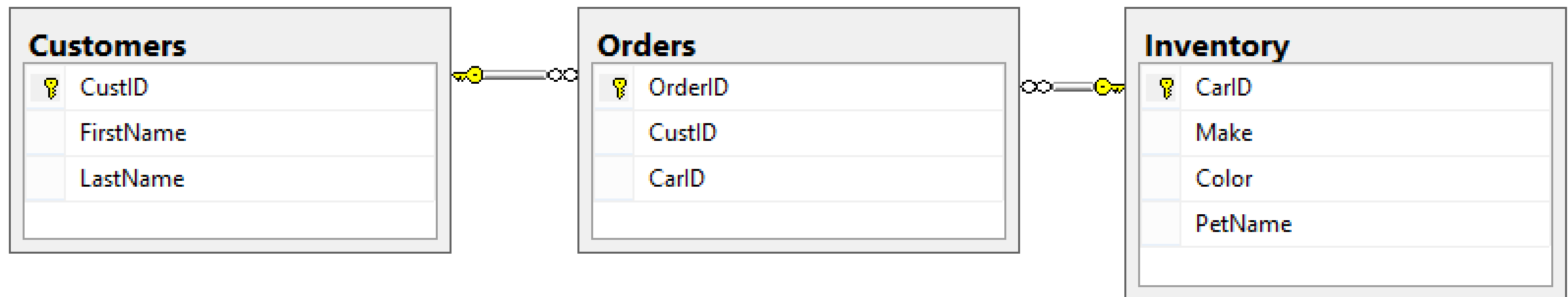
```
static void Main(string[] args)
{
    // Чтение ключа поставщика.
    string dataProvString = ConfigurationManager.AppSettings["provider"];
    // Преобразование строки в перечисление.
    DataProvider dp = DataProvider.None;
    if (Enum.IsDefined(typeof(DataProvider), dataProvString))
        dp = (DataProvider)Enum.Parse(typeof(DataProvider), dataProvString);
    else
        Console.WriteLine("К сожалению, поставщик отсутствует.");
    // Получение конкретного подключения.
    IDbConnection myCn = GetConnection(dp);
    if (myCn != null)
        Console.WriteLine("Ваше подключение - {0}", myCn.GetType().Name);
    // Открытие, использование и закрытие подключения...
    Console.ReadLine();
}
```

По сути создается фабрика объектов подключений, позволяющая изменить поставщик без необходимости перекомпиляции кодовой базы. Нужно лишь изменить файл *.config

Для использования типа ConfigurationManager необходимо поместить ссылку на сборку System.Configuration.dll и импортировать пространство имен System.Configuration.

База данных AutoLot

База данных AutoLot содержит три взаимосвязанных таблицы: Inventory, Orders и Customers, в которых хранятся различные данные о заказах гипотетической компании по продаже автомобилей.



	CustID	FirstName	LastName
1	1	Dave	Brenner
2	2	Matt	Walton
3	3	Steve	Hagen
4	4	Pat	Walton

	OrderID	CustID	CarID
1	1000	1	1000
2	1001	2	32
3	1002	3	888
4	1003	4	2911

	CarID	Make	Color	PetName
1	32	VW	Black	Zippy
2	83	Ford	Rust	Rusty
3	872	Saab	Black	Mel
4	888	Yugo	Yellow	Clunker
5	1000	BMW	Black	Bimmer
6	1011	BMW	Green	Hank
7	2911	BMW	Pink	Pinky

Добавим хранимую процедуру, которая по идентификатору автомобиля будет возвращать его дружественное имя

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

Модель фабрик поставщиков данных ADO.NET

Фабрика поставщиков данных .NET позволяет создать единую кодовую базу с помощью обобщенных типов доступа к данным.

Все поставщики данных, разработанные Microsoft, содержат класс, порожденный от System.Data.Common.DbProviderFactory.

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

Модель фабрик поставщиков данных ADO.NET

Для получения типа, порожденного от DbProviderFactory, непосредственно для вашего поставщика данных в пространстве имен System.Data.Common имеется класс DbProviderFactories. С помощью метода GetFactory() можно получить конкретный объект DbProviderFactory для указанного поставщика данных.

```
static void Main(string[] args)
{
    // Получение генератора для поставщика данных SQL.
    DbProviderFactory sqlFactory =
         DbProviderFactories.GetFactory("System.Data.SqlClient");
    //...
}
```

Фабрику можно получить не с помощью жестко закодированного строкового литерала, а, например, прочитать эту информацию из клиентского файла *.config. После получения генератора для поставщика данных можно получить связанные с ним объекты данных

Модель фабрик поставщиков данных ADO.NET

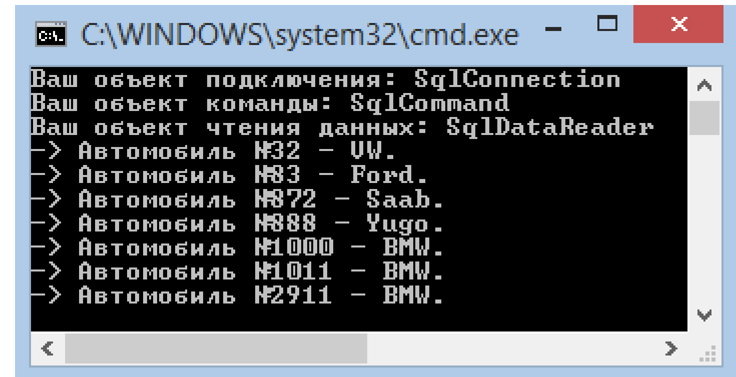
Чтобы получить тип фабрики для необходимого поставщика данных в метод DbProviderFactories.GetFactory() нужно передать значение provider, считанный из конфигурационного файла.

```
static void Main(string[] args)
{
    // Получение строки подключения и поставщика из *.config.
    string dp = ConfigurationManager.AppSettings["provider"];
    string cnStr = ConfigurationManager.AppSettings["cnStr"];
    // Получение генератора поставщика.
    DbProviderFactory df = DbProviderFactories.GetFactory(dp);
    // Получение объекта подключения.
    using (DbConnection cn = df.CreateConnection())
    {
        Console.WriteLine("Ваш объект подключения: {0}", cn.GetType().Name);
        cn.ConnectionString = cnStr;
        cn.Open();
        // Создание объекта команды.
        DbCommand cmd = df.CreateCommand();
        Console.WriteLine("Ваш объект команды: {0}", cmd.GetType().Name);
        cmd.Connection = cn;
        cmd.CommandText = "Select * From Inventory";
        // Вывод данных с помощью объекта чтения данных.
        using (DbDataReader dr = cmd.ExecuteReader())
        {
            Console.WriteLine("Ваш объект чтения данных: {0}", dr.GetType().Name);
            while (dr.Read())
                Console.WriteLine("-> Автомобиль №{0} - {1}.",
                dr["CarID"], dr["Make"].ToString());
        }
    }
    Console.ReadLine();
}
```

Модель фабрик поставщиков данных ADO.NET

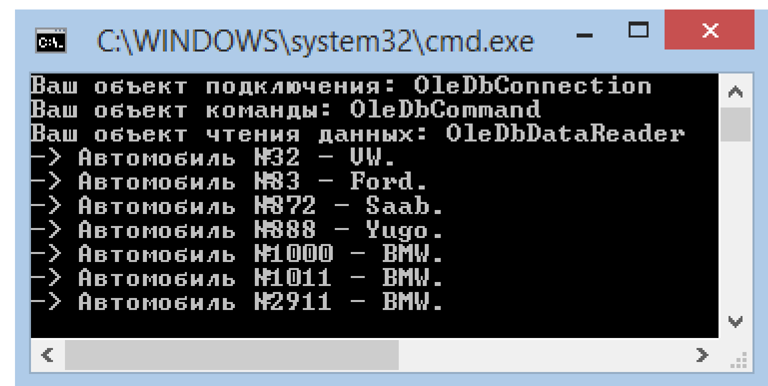
В файле *.config в качестве поставщика данных указан System.Data.SqlClient

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



В файле *.config в качестве поставщика данных указан System.Data.OleDb

```
<?xml version="1.0" encoding="utf-8"?>
<configuration>
  <appSettings>
    <!-- Поставщик -->
    <add key="provider" value="System.Data.OleDb" />
    <!-- Строка подключения -->
    <add key="cnStr" value=
        "Provider=SQLOLEDB; Data Source=(local)\SQLEXPRESS;
        Integrated Security=SSPI; Initial Catalog=AutoLot"/>
  </appSettings>
</configuration>
```



Потенциальный недостаток модели фабрик поставщиков данных

Хотя это действительно очень мощная модель, все же нужно проверить, что в кодовой базе используются только типы и методы, общие для всех поставщиков как потомков абстрактных базовых классов.

Поэтому при разработке кодовой базы следует ограничиться членами из DbConnection, DbCommand и других типов из пространства имен System.Data.Common.

Но такой обобщенный подход не позволит непосредственно задействовать дополнительные возможности конкретной СУБД.

Если все же потребуются вызовы специфических членов конкретного поставщика (например, SqlConnection), то это можно сделать с помощью явного преобразования типа

```
using (DbConnection cn = df.CreateConnection())
{
    Console.WriteLine("Ваш объект подключения: {0}", cn.GetType().Name);
    cn.ConnectionString = cnStr;
    cn.Open();
    if (cn is SqlConnection)
    {
        // Вывод используемой версии SQL Server.
        Console.WriteLine(((SqlConnection)cn).ServerVersion);
    }
}
```

Элемент <connectionString>

В конфигурационных файлах приложения может быть определен элемент <connectionStrings>.

В этом элементе можно задать любое количество пар имя/значение, которые программа может прочитать в память с помощью индексатора ConfigurationManager.ConnectionStrings.

Одним из преимуществ данного подхода по сравнению с использованием элемента <appSettings> и индексатора ConfigurationManager.AppSettings является то, что при этом можно определить несколько строк подключения для одного приложения.

```
<configuration>
  <appSettings>
    <!-- Поставщик -->
    <add key="provider" value="System.Data.SqlClient" />
  </appSettings>
  <!-- Строки подключения -->
  <connectionStrings>
    <add name ="AutoLotSqlProvider" connectionString = "Data Source=(local)\SQLEXPRESS;
      Integrated Security=SSPI; Initial Catalog=AutoLot"/>
    <add name ="AutoLotOleDbProvider" connectionString = "Provider=SQLOLEDB; Data Source=(local)\SQLEXPRESS;
      Integrated Security=SSPI; Initial Catalog=AutoLot"/>
  </connectionStrings>
</configuration>
```

```
static void Main(string[] args)
{
    string dp = ConfigurationManager.AppSettings["provider"];
    string cnStr = ConfigurationManager.ConnectionStrings["AutoLotSqlProvider"].ConnectionString;
}
```

Подключенный уровень ADO.NET

Подключенный уровень ADO.NET позволяет взаимодействовать с базой данных с помощью объектов подключения, чтения данных и команд конкретного поставщика данных.

Первое, что нужно сделать при работе с поставщиком данных – это установить сеанс с источником данных с помощью объекта подключения, порожденного от DbConnection.

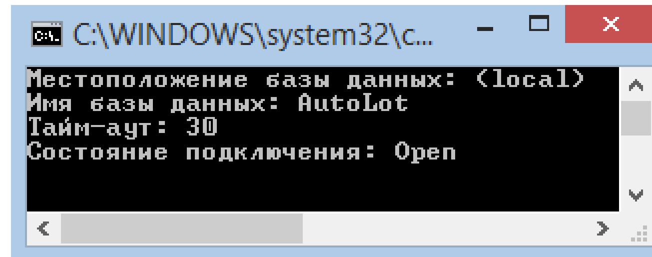
Члены типа DbConnection

Член	Назначение
BeginTransaction()	Используется для начала транзакции базы данных
ChangeDatabase()	Изменяет базу данных для открытого подключения
ConnectionTimeout	Свойство только для чтения. Возвращает время ожидания при установке подключения, после которого ожидание прекращается и выдается сообщение об ошибке (по умолчанию 15 секунд). Для изменения этого времени нужно изменить в строке подключения сегмент Connect Timeout (например, Connect Timeout=30)
Database	Свойство только для чтения. Содержит имя базы данных, с которой связан объект подключения
DataSource	Свойство только для чтения. Содержит местоположение базы данных, с которой связан объект подключения
GetSchema()	Этот метод возвращает объект DataTable, содержащий информацию схемы из источника данных
State	Свойство только для чтения. Содержит текущее состояние подключения в виде одного из значений перечисления ConnectionState

Работа с объектами подключения

Свойства типа DbConnection предназначены в основном только для чтения и поэтому нужны, если требуется получить характеристики подключения во время выполнения. Для переопределения стандартных значений необходимо будет изменить саму строку подключения.

```
static void ShowConnectionStatus(DbConnection cn)
{
    // Вывод различных сведений о текущем объекте подключения.
    Console.WriteLine("Местоположение базы данных: {0}", cn.DataSource);
    Console.WriteLine("Имя базы данных: {0}", cn.Database);
    Console.WriteLine("Тайм-аут: {0}", cn.ConnectionTimeout);
    Console.WriteLine("Состояние подключения: {0}\n", cn.State.ToString());
}
static void Main(string[] args)
{
    using (SqlConnection cn = new SqlConnection())
    {
        cn.ConnectionString =
            @"Data Source=(local);" +
            "Integrated Security=SSPI; Initial Catalog=AutoLot; Connect Timeout=30";
        cn.Open();
        ShowConnectionStatus(cn);
    }
}
```



Работа с объектами ConnectionStringBuilder

Поставщики данных ADO.NET, разработанные Microsoft, поддерживают объекты построителей строк соединения, которые позволяют устанавливать пары имя/значение с помощью строго типизированных свойств.

```
static void Main(string[] args)
{
    // Создание строки подключения с помощью объекта построителя.
    SqlConnectionStringBuilder cnStrBuilder =
    new SqlConnectionStringBuilder();
    cnStrBuilder.InitialCatalog = "AutoLot";
    cnStrBuilder.DataSource = @"(local)\SQLEXPRESS";
    cnStrBuilder.ConnectTimeout = 30;
    cnStrBuilder.IntegratedSecurity = true;
    using (SqlConnection cn = new SqlConnection())
    {
        cn.ConnectionString = cnStrBuilder.ConnectionString;
        cn.Open();
        ShowConnectionStatus(cn);
    }
    Console.ReadLine();
}
```

Создается экземпляр SqlConnectionStringBuilder, устанавливаются его свойства, и выбирается внутренняя строка из свойства ConnectionString.

После наполнения объекта начальными строковыми данными можно изменить отдельные пары имя/значение с помощью соответствующих свойств

```
static void Main(string[] args)
{
    string cnStr = @"Data Source=(local)\SQLEXPRESS;" +
         "Integrated Security=SSPI; Initial Catalog=AutoLot";
    SqlConnectionStringBuilder cnStrBuilder =
         new SqlConnectionStringBuilder(cnStr);
    // Изменение значения тайм-аута.
    cnStrBuilder.ConnectTimeout = 5;
}
```

Работа с объектами команд

Тип SqlCommand (порожденный от DbCommand) представляет собой объектно-ориентированное представление SQL-запроса, имени таблицы или хранимой процедуры.

Тип команды указывается свойством CommandType, которое принимает значения из перечисления CommandType

```
public enum CommandType
{
    StoredProcedure,
    TableDirect,
    Text // Значение по умолчанию.
}
```

При создании объекта команды можно сразу задать SQL-запрос, передав его с помощью параметра конструктора или непосредственно свойства CommandText. Кроме того, при создании объекта команды необходимо указать подключение, которое будет в нем применяться. Это тоже можно сделать либо через параметр конструктора, либо с помощью свойства Connection.

```
// Создание объекта команды с помощью аргументов конструктора.
string strSQL = "Select * From Inventory";
SqlCommand myCommand = new SqlCommand(strSQL, cn);
// Создание еще одного объекта команды с помощью свойств.
SqlCommand testCommand = new SqlCommand();
testCommand.Connection = cn;
testCommand.CommandText = strSQL;
```

Работа с объектами команд

Члены типа DbCommand

Член	Назначение
CommandTimeout	Выдает или устанавливает время ожидания при выполнении команды до прекращения попытки и генерации ошибки. По умолчанию равно 30 секунд
Connection	Выдает или устанавливает объект DbConnection, используемый данным DbCommand
Parameters	Выдает коллекцию типов DbParameter, используемых для параметризованного запроса
Cancel()	Отменяет выполнение команды
ExecuteReader()	Выполняет SQL-запрос и возвращает объект DbDataReader поставщика данных, позволяющий доступ к результату запроса только для чтения в прямом направлении
ExecuteNonQuery()	Выполняет не запросный SQL (например, вставка, обновление, удаление или создание таблицы)
ExecuteScalar()	Облегченная версия метода ExecuteReader(), созданная специально для одноэлементных запросов (наподобие получения количества записей)
Prepare()	Создает подготовленную (или скомпилированную) версию команды в источнике данных. Подготовленные запросы выполняются несколько быстрее, и их имеет смысл использовать, если требуется многократное выполнение одного и того же запроса (обычно каждый раз с различными параметрами)

Работа с объектами чтения данных

Самым простым и быстрым способом получения информации из хранилища данных является тип DbDataReader, реализующий интерфейс IDataReader.

Объекты чтения данных представляют поток данных, допускающий только чтение в прямом направлении, и возвращают каждый раз по одной записи.

Поэтому объекты чтения данных применяются только для выдачи SQL-запросов на выборку информации из хранилища данных.

Объекты чтения данных удобны, если нужно быстро просмотреть большой объем данных без необходимости их представления в памяти.

Например, если запросить из таблицы 20000 записей, чтобы сохранить их в текстовом файле, то хранение этой информации в DataSet будет излишней затратой памяти, т.к. DataSet полностью хранит результат запроса в памяти.

Объекты чтения данных (в отличие от объектов адаптеров данных) поддерживают открытое подключение к источнику данных, пока сеанс не будет явно закрыт.

Работа с объектами чтения данных

- Объект чтения данных можно получить из объекта команды с помощью вызова ExecuteReader().
- Объект чтения данных представляет текущую запись, прочитанную из базы данных.
- Доступ к конкретному столбцу возможен либо по имени, либо по целочисленному индексу, начиная с нуля.

```
static void Main(string[] args)
{
    // Создание открытого подключения.
    using (SqlConnection cn = new SqlConnection())
    {
        cn.ConnectionString = @"Data Source=(local)\SQLEXPRESS;" +
                "Integrated Security=SSPI;Initial Catalog=AutoLot";
        cn.Open();
        // Создание объекта команды SQL.
        string strSQL = "Select * From Inventory";
        SqlCommand myCommand = new SqlCommand(strSQL, cn);
        // Получение объекта чтения данных с помощью ExecuteReader().
        using (SqlDataReader myDataReader = myCommand.ExecuteReader())
        {
            // Просмотр всех результатов.
            while (myDataReader.Read())
            {
                Console.WriteLine("-> Make: {0}, PetName: {1}, Color: {2}.",
                    myDataReader["Make"].ToString().Trim(),
                    myDataReader["PetName"].ToString().Trim(),
                    myDataReader["Color"].ToString().Trim());
            }
            myDataReader.Close();
        }
    }
}
```

Метод Read() позволяет определить, когда достигнут конец данных

Сразу после завершения обработки записей вызывается метод Close(), чтобы освободить объект подключения

Работа с объектами чтения данных

Объекты чтения данных могут получать множественные результирующие наборы с помощью одного объекта команды.

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

После получения объекта чтения данных можно просмотреть все записи результирующего набора с помощью метода NextResult(). Автоматически возвращается всегда первый результирующий набор.

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

В этом фрагменте индексатор объекта чтения данных перегружен, чтобы он мог принимать либо string (имя столбца), либо int (порядковый номер столбца). Это позволяет прояснить логику объекта чтения и избежать применения жестко закодированных строковых имен.

Объект чтения данных может обрабатывать только SQL-операторы Select и не может использоваться для изменения существующей таблицы базы данных с помощью запросов Insert, Update или Delete.

Проект AutoLotDAL

1. Пустой проект

```
using System;
using System.Collections.Generic;
using System.Text;
//Будем применять поставщик SQL Server;
//но для большей гибкости можно воспользоваться
//и фабрикой поставщиков ADO.NET.
using System.Data;
using System.Data.SqlClient;
namespace AutoLotConnectedLayer
{
    public class InventoryDAL
    {
    }
}
```

2. Добавление логики подключения

```
public class InventoryDAL
{
    // Этот член будет использоваться всеми методами.
    private SqlConnection sqlCn = null;
    public void OpenConnection(string connectionString)
    {
        sqlCn = new SqlConnection();
        sqlCn.ConnectionString = connectionString;
        sqlCn.Open();
    }
    public void CloseConnection()
    {
        sqlCn.Close();
    }
}
```

3. Добавление логики вставки

```
public void InsertAuto(int id, string color, string make, string petName)
{
    // Формирование и выполнение оператора SQL.
    string sql = string.Format("Insert Into Inventory" +
    " (CarID, Make, Color, PetName) Values" +
    " ('{0}', '{1}', {2}', '{3}')", id, make, color, petName);
    // Выполнение с помощью нашего подключения.
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        cmd.ExecuteNonQuery();
    }
}
```

Вставка новой записи в таблицу Inventory сводится к форматированию SQL-оператора Insert (в зависимости от введенных пользователем данных) и вызову метода ExecuteNonQuery() с помощью объекта команды.

Проект AutoLotDAL

Можно предложить перегруженную версию, которая позволяет вызывающему методу передать объект строго типизированного класса, представляющий данные для новой строки.

```
public class NewCar
{
    public int CarID { get; set; }
    public string Color { get; set; }
    public string Make { get; set; }
    public string PetName { get; set; }
}
```

Создание оператора SQL с помощью конкатенации строк может оказаться опасным с точки зрения безопасности (существуют атаки вставкой в SQL). Текст команды лучше создавать с помощью параметризованного запроса.

```
public void InsertAuto(NewCar car)
{
    // Формирование и выполнение оператора SQL.
    string sql = string.Format("Insert Into Inventory" +
        "(CarID, Make, Color, PetName) Values" +
        "('{0}', '{1}', '{2}', '{3}')", car.CarID, car.Make, car.Color, car.PetName);
    // Выполнение с помощью нашего подключения.
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        cmd.ExecuteNonQuery();
    }
}
```

Определение классов, представляющих записи в реляционной базе данных – распространенный способ создания библиотеки доступа к данным.

Проект AutoLotDAL

4. Добавление логики удаления

```
public void DeleteCar(int id)
{
    // Получение идентификатора автомобиля перед его удалением.
    string sql = string.Format("Delete from Inventory where CarID = '{0}'", id);
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        try
        {
            cmd.ExecuteNonQuery();
        }
        catch (SqlException ex)
        {
            Exception error = new Exception("К сожалению, эта машина заказана!", ex);
            throw error;
        }
    }
}
```

Область try/catch обрабатывает возможную ситуацию, когда выполняется попытка удаления автомобиля, уже заказанного кем-то из таблицы Customers.

5. Добавление логики изменения

```
public void UpdateCarPetName(int id, string newPetName)
{
    // Получение идентификатора автомобиля и нового дружественного имени для него.
    string sql = string.Format("Update Inventory Set PetName = '{0}' Where CarID = '{1}'", newPetName, id);
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        cmd.ExecuteNonQuery();
    }
}
```

Проект AutoLotDAL

6. Добавление логики выборки

```
public List<NewCar> GetAllInventoryAsList()
{
    // Здесь будут находиться записи.
    List<NewCar> inv = new List<NewCar>();
    // Подготовка объекта команды.
    string sql = "Select * From Inventory";
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        SqlDataReader dr = cmd.ExecuteReader();
        while (dr.Read())
        {
            inv.Add(new NewCar
            {
                CarID = (int)dr["CarID"],
                Color = (string)dr["Color"],
                Make = (string)dr["Make"],
                PetName = (string)dr["PetName"]
            });
        }
        dr.Close();
    }
    return inv;
}
```

Класс DataTable содержит данные в виде коллекции строк и столбцов. Эти коллекции можно заполнять программным образом, но в типе DataTable имеется метод Load(), который может автоматически заполнять их с помощью объекта чтения данных.

```
public DataTable GetAllInventoryAsDataTable()
{
    // Здесь будут находиться записи.
    DataTable inv = new DataTable();
    // Подготовка объекта команды.
    string sql = "Select * From Inventory";
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        SqlDataReader dr = cmd.ExecuteReader();
        // Заполнение DataTable данными из объекта чтения и зачистка.
        inv.Load(dr);
        dr.Close();
    }
    return inv;
}
```

Работа с параметрами

- Параметризованные запросы позволяют рассматривать параметры SQL как объекты и помогают сократить количество опечаток.
- Параметризованные запросы обычно выполняются значительно быстрее запросов в виде строковых литералов, поскольку они анализируются только один раз.
- Параметризованные запросы защищают от атак внедрением в SQL.

Основные члены типа DbParameter

Свойство	Назначение
DbType	Выдает или устанавливает тип данных из параметра, представляемый в виде типа CLR
Direction	Выдает или устанавливает вид параметра: только для ввода, только для вывода, для ввода и для вывода или параметр для возврата значения
IsNullable	Выдает или устанавливает, может ли параметр принимать пустые значения
ParameterName	Выдает или устанавливает имя DbParameter
Size	Выдает или устанавливает максимальный размер данных для параметра (полезно только для текстовых данных)
Value	Выдает или устанавливает значение параметра

Работа с параметрами

```
public void InsertAuto(int id, string color, string make, string petName)
{
    // "Заполнители" в SQL-запросе.
    string sql = string.Format("Insert Into Inventory " +
        "(CarID, Make, Color, PetName) Values " +
        "(@CarID, @Make, @Color, @PetName)");
    // У этой команды будут внутренние параметры.
    using (SqlCommand cmd = new SqlCommand(sql, this.sqlCn))
    {
        // Заполнение коллекции параметров.
        SqlParameter param = new SqlParameter();
        param.ParameterName = "@CarID";
        param.Value = id;
        param.SqlDbType = SqlDbType.Int;
        cmd.Parameters.Add(param);
        param = new SqlParameter();
        param.ParameterName = "@Make";
        param.Value = make;
        param.SqlDbType = SqlDbType.Char;
        param.Size = 10;
        cmd.Parameters.Add(param);
        param = new SqlParameter();
        param.ParameterName = "@Color";
        param.Value = color;
        param.SqlDbType = SqlDbType.Char;
        param.Size = 10;
        cmd.Parameters.Add(param);
        param = new SqlParameter();
        param.ParameterName = "@PetName";
        param.Value = petName;
        param.SqlDbType = SqlDbType.Char;
        param.Size = 10;
        cmd.Parameters.Add(param);
        cmd.ExecuteNonQuery();
    }
}
```

- Для поддержки параметризованных запросов объекты команд ADO.NET поддерживают коллекцию отдельных объектов параметров.
- По умолчанию эта коллекция пуста, но в нее можно занести любое количество объектов параметров, которые соответствуют параметрам-заполнителям в SQL-запросе.
- Если нужно связать параметр SQL-запроса с членом коллекции параметров некоторого объекта команды, надо поставить перед параметром SQL символ @ (по крайней мере, при работе с Microsoft SQL Server, хотя не все СУБД поддерживают это обозначение)

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

7. Выполнение хранимой процедуры

```csharp
public string LookUpPetName(int carID)
{
    string carPetName = string.Empty;
    // Задание имени хранимой процедуры.
    using (SqlCommand cmd = new SqlCommand("GetPetName", this.sqlCn))
    {
        cmd.CommandType = CommandType.StoredProcedure;
        // Входной параметр.
        SqlParameter param = new SqlParameter();
        param.ParameterName = "@carID";
        param.SqlDbType = SqlDbType.Int;
        param.Value = carID;
        param.Direction = ParameterDirection.Input;
        cmd.Parameters.Add(param);
        //По умолчанию параметры считаются входными, но все же для ясности:
        param.Direction = ParameterDirection.Input;
        cmd.Parameters.Add(param);
        // Выходной параметр.
        param = new SqlParameter();
        param.ParameterName = "@petName";
        param.SqlDbType = SqlDbType.Char;
        param.Size = 10;
        param.Direction = ParameterDirection.Output;
        cmd.Parameters.Add(param);
        // Выполнение хранимой процедуры.
        cmd.ExecuteNonQuery();
        // Возврат выходного параметра.
        carPetName = ((string)cmd.Parameters["@petName"].Value).Trim();
    }
    return carPetName;
}
```

Объект команды может представлять оператор SQL (по умолчанию) или имя хранимой процедуры. Если необходимо сообщить объекту команды, что он должен вызывать хранимую процедуру, то нужно передать имя этой процедуры и установить в свойстве CommandType значение CommandType.StoredProcedure

Транзакции баз данных

Транзакция – это набор операций в базе данных, которые должны быть либо все выполнены, либо все не выполнены. Транзакции применяются для обеспечения безопасности, достоверности и непротиворечивости данных в таблице.

Классический пример транзакции – процесс перевода денежных средств с одного банковского счета на другой. Например, если понадобилось перевести $500 с депозитного счета на текущий счет, то нужно выполнить в режиме транзакции следующие шаги:

- банк должен снять $500 с вашего депозитного счета;
- затем банк должен добавить $500 на ваш текущий счет.

Объекты транзакции, которые имеются в поставщиках данных ADO.NET порождены от DBTransaction и реализуют интерфейс IDbTransaction

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

Метод Commit() вызывается, если все операции в базе данных завершились успешно. При этом все ожидающие изменения фиксируются в хранилище данных.

Метод Rollback() можно вызвать при возникновении исключения времени выполнения, чтобы сообщить СУБД, что все ожидающие изменения следует отменить и оставить первоначальные данные без изменений.

Проект AutoLotDAL

8. Добавление метода транзакции в InventoryDAL

```
// Новый член класса InventoryDAL.
public void ProcessCreditRisk(bool throwEx, int custID)
{
    // Вначале выборка имени по идентификатору клиента.
    string fName = string.Empty;
    string lName = string.Empty;
    SqlCommand cmdSelect = new SqlCommand(
    string.Format("Select * from Customers where CustID = {0}",
custID), sqlCn);
    using (SqlDataReader dr = cmdSelect.ExecuteReader())
    {
        if (dr.HasRows)
        {
            dr.Read();
            fName = (string)dr["FirstName"];
            lName = (string)dr["LastName"];
        }
        else
            return;
    }
    // Создание объектов команд для каждого шага операции.
    SqlCommand cmdRemove = new SqlCommand(string.Format("Delete from
Customers where CustID = {0}", custID), sqlCn);
    SqlCommand cmdInsert = new SqlCommand(string.Format("Insert Into
CreditRisks" + "(CustID, FirstName, LastName) Values ({0}, '{1}',
'{2}')", custID, fName, lName), sqlCn);
```

```
    // Получаем из объекта подключения.
    SqlTransaction tx = null;
    try
    {
        tx = sqlCn.BeginTransaction();
        // Включение команд в транзакцию.
        cmdInsert.Transaction = tx;
        cmdRemove.Transaction = tx;
        // Выполнение команд.
        cmdInsert.ExecuteNonQuery();
        cmdRemove.ExecuteNonQuery();
        // Имитация ошибки.
        if (throwEx)
        {
            throw new ApplicationException("Ошибка базы данных!
Транзакция завершена неудачно.");
        }
        // Фиксация.
        tx.Commit();
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
        // При возникновении любой ошибки выполняется откат транзакции.
        tx.Rollback();
    }
}
```