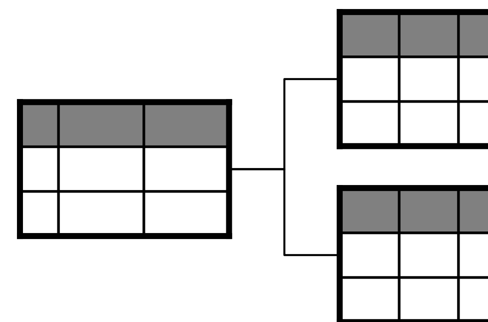
Data Access with ADO.NET

Dr. Herbert Praehofer
Institute for System Software
Johannes Kepler University Linz

Dr. Dietrich Birngruber
Software Architect
TechTalk
www.techtalk.at



© University of Linz, Institute for System Software, 2004
published under the Microsoft Curriculum License

ADO.NET

Introduction

Connection-oriented Access

Connectionless Access

Database Access with DataAdapter

Integration with XML

Preview of ADO.NET 2.0

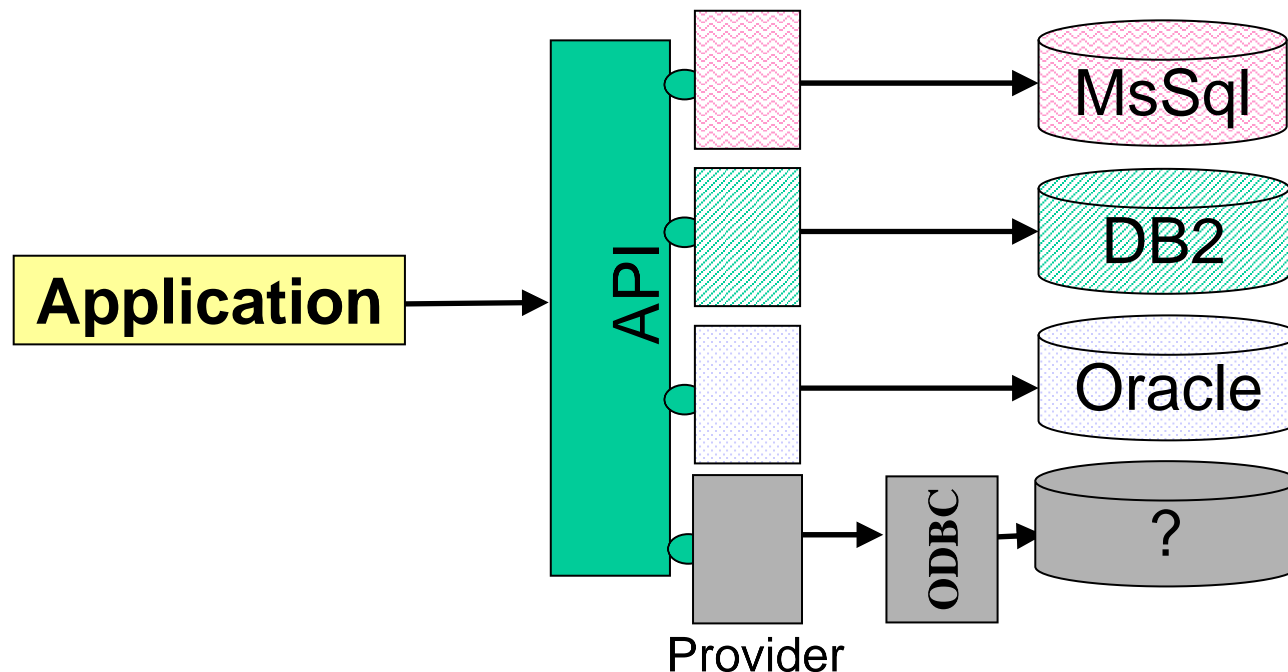
Summary

ADO.NET

- Is the .NET technology for accessing structured data
- Uniform object oriented interface for different data sources
  - relational data bases
  - XML data
  - other data sources
- Designed for distributed and Web applications
- Provides 2 models for data access
  - connection-oriented
  - connectionless

Idea of the Universal Data Access

- Connection of (object-oriented) programming languages and relational data bases
- Uniform programming model and API
- Special implementations for data sources (*providers*)



Data Providers

Microsoft's layered architecture for data access

ADO.NET

SQL Server
Oracle
MySQL

ODBC

OLEDB

SQL-data

MS SQL Server, Oracle,
Jet, Foxpro, ...

Non-SQL-data

Directory Services, Mail,
Text, Video, ...

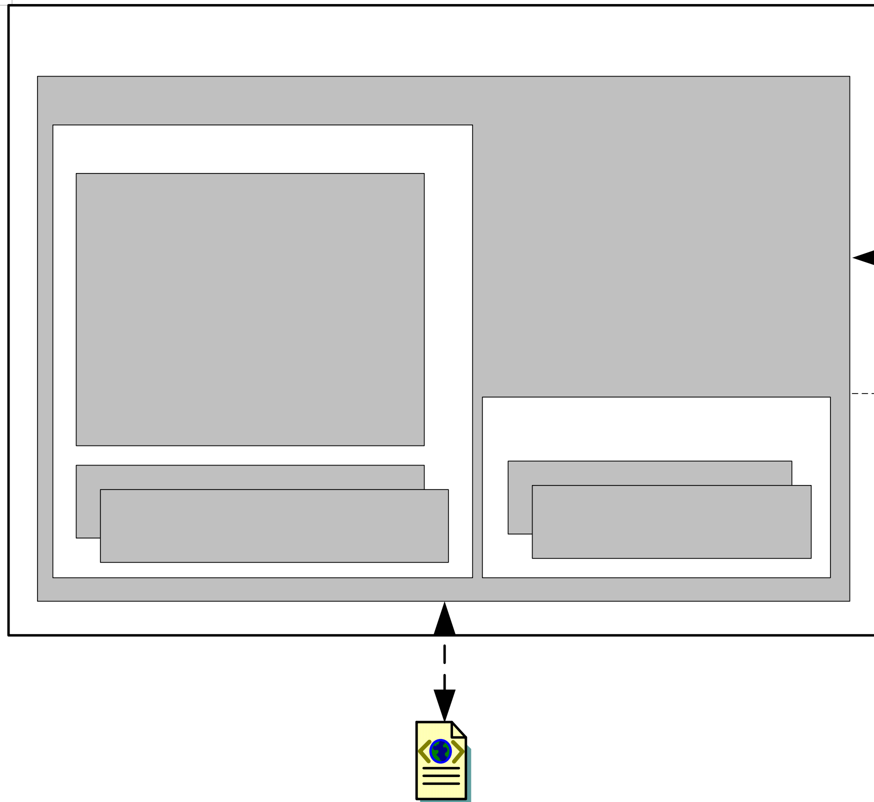
History of Universal Data Access (Microsoft)

- ODBC
  - OLE DB
    - ADO (*Active Data Objects*)
      - ADO.NET

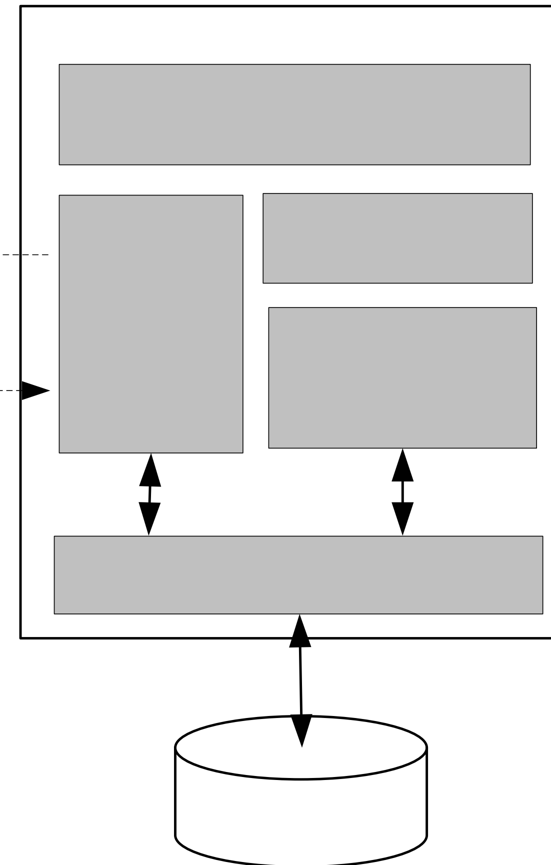
ADO	**ADO.NET**
connection-oriented	connection-oriented + connectionless
sequential access	main-memory representation with direct access
only one table supported	more than one table supported
COM-marshalling	XML-marshalling

Architecture of ADO.NET

connectionless



connection-oriented



ADO.NET Content Cor

Connection-oriented versus Connectionless

- Connection-oriented
  - Keeps the connection to the data base alive
  - Intended for applications with:
    - short running transactions
    - only a few parallel accesses
    - up-to-date data

- Connectionless
  - No permanent connection to the data source
  - Data cached in main memory
  - Changes in main memory ≠ changes in data source
  - Intended for applications with:
    - many parallel and long lasting accesses (e.g.: web applications)

ADO.NET Assembly and Namespaces

Assembly

- System.Data.dll

Namespaces:

– System.Data	general data types
– System.Data.Common	classes for implementing providers
– System.Data.OleDb	OLE DB provider
– System.Data.SqlClient	Microsoft SQL Server provider
– System.Data.SqlTypes	data types for SQL Server
– System.Data.Odbc	ODBC provider (since .NET 1.1)
– System.Data.OracleClient	Oracle provider (since .NET 1.1)
– System.Data.SqlServerCe	Compact Framework

ADO.NET

Introduction

Connection-oriented Access

Connectionless Access

Database Access with DataAdapter

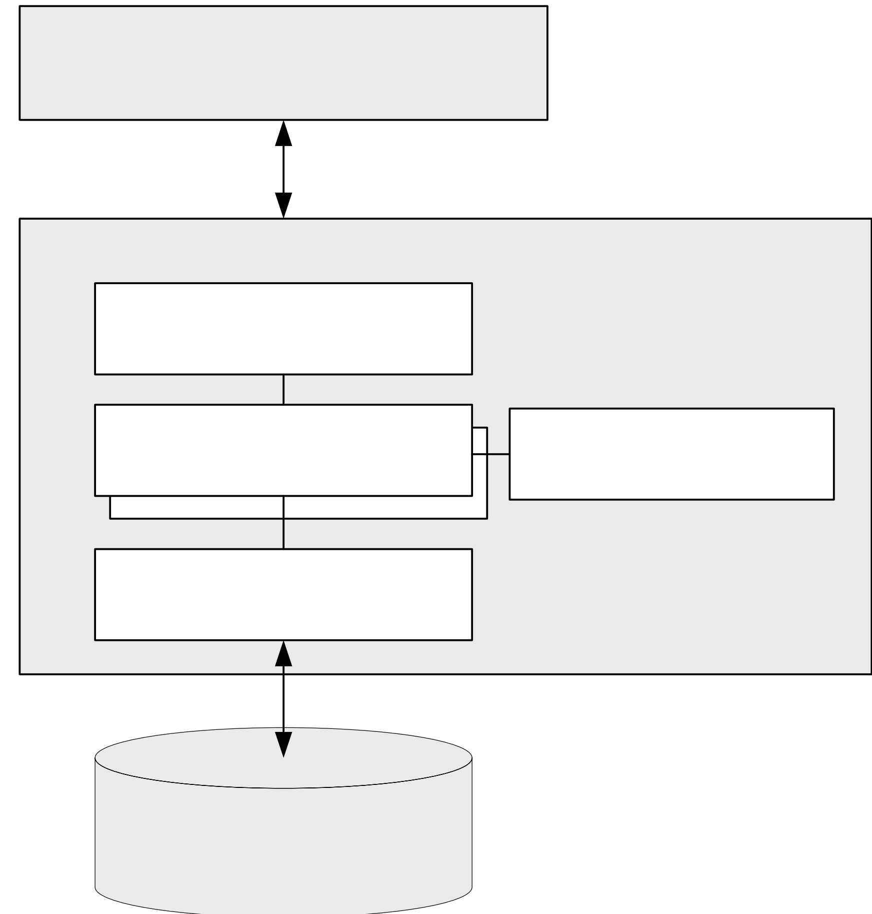
Integration with XML

Preview of ADO.NET 2.0

Summary

Architecture

- DbConnection
  - represents connection to the data source
- DbCommand
  - represents a SQL command
- DbTransaction
  - represents a transaction
  - commands can be executed within a transaction
- DataReader
  - result of a data base query
  - allows sequential reading of rows



Class Hierarchy

- General interface definitions

  IDbConnection

  IDbCommand

  IDbTransaction

  IDataReader

- Special implementations

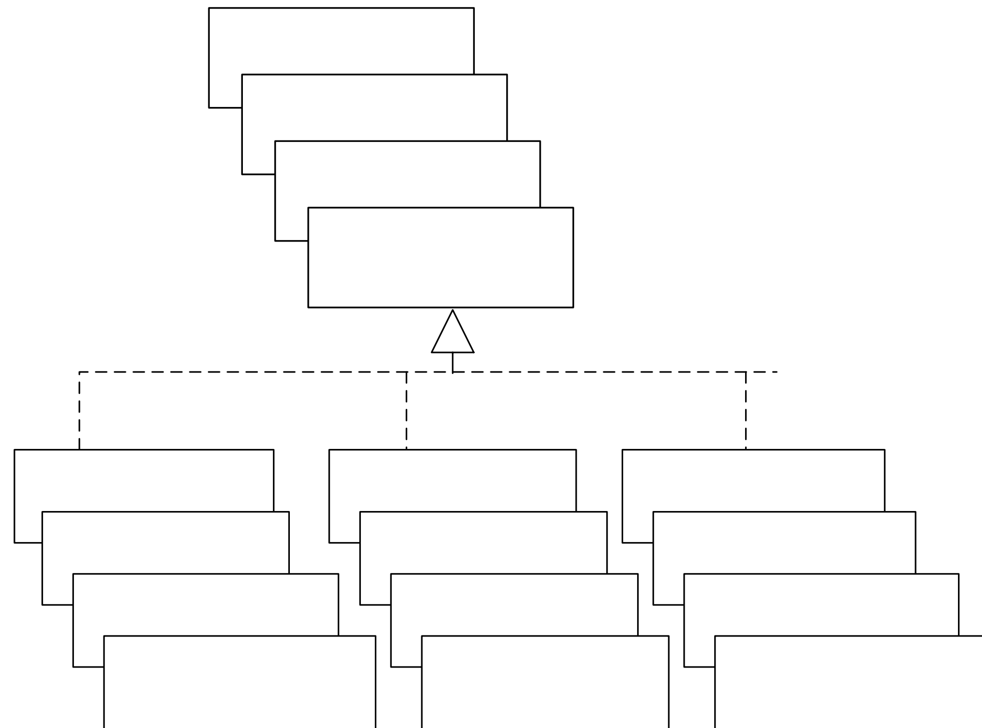
  OleDb: implementation for OLEDB

  Sql: implementation for SQL Server

  Oracle: implementation for Oracle

  Odbc: implementation for ODBC

  SqlCe: implementation for SQL Server CE data base



Example: Northwind Database

Microsoft Example for SQL Server

- Reading of the table Employees
- Output of
  - EmployeesID, LastName, FirstName

  for all rows of table Employees

```
| 1 | Davolio | Nancy
| 2 | Fuller | Andrew
| 3 | Leverling | Janet
| 4 | Peacock | Margaret
| 5 | Buchanan | Steven
| 6 | Suyama | Michael
| 7 | King | Robert
| 8 | Callahan | Laura
| 9 | Dodsworth | Anne
```

Run

Program Pattern for Connection-oriented Data Access

```
1.) Declare the connection
try {
    1.) Request connection to database

    2.) Execute SQL statements

    3.) Process result

    4.) Release Resources
} catch ( Exception ) {
    Handle exception
} finally {
    try {
        4.) Close connection
    } catch (Exception)
        { Handle exception }
}
```

Example: EmployeeReader (1)

```
using System;
using System.Data;
using System.Data.OleDb;

public class EmployeeReader {
    public static void Main() {
```

- Establish connection

```
        string connStr = "provider=SQLOLEDB; data source=(local)\\NetSDK; " +
                         "initial catalog=Northwind; user id=sa; password=; ";
        IDbConnection con = null;                    // declare connection object
        try {
            con = new OleDbConnection(connStr);      // create connection object
            con.Open();                              // open connection
```

- Execute command

```
            //----- create SQL command
            IDbCommand cmd = con.CreateCommand();
            cmd.CommandText = "SELECT EmployeeID, LastName, FirstName FROM Employees";
            //----- execute SQL command;  result is an OleDbDataReader
            IDataReader reader = cmd.ExecuteReader();
```

```
            // continue next page
```

Example: EmployeeReader (2)

- Read and process data rows

```
IDataReader reader = cmd.ExecuteReader();
object[] dataRow = new object[reader.FieldCount];

while (reader.Read()) {
    int cols = reader.GetValues(dataRow);
    for (int i = 0; i < cols; i++) Console.Write("| {0} " , dataRow[i]);
        Console.WriteLine();
}
```

- Close connection

```
        //----- close reader
        reader.Close();
        } catch (Exception e) {
            Console.WriteLine(e.Message);
        } finally {
            try {
                if (con != null)
                    // ----- close connection
                    con.Close();
            } catch (Exception ex) { Console.WriteLine(ex.Message); }
        }
    }
}
```

Interface IDbConnection

- ConnectionString defines data base connection

```
string ConnectionString {get; set;}
```

- Open and close connection

```
void Close();
void Open();
```

- Properties of connection object

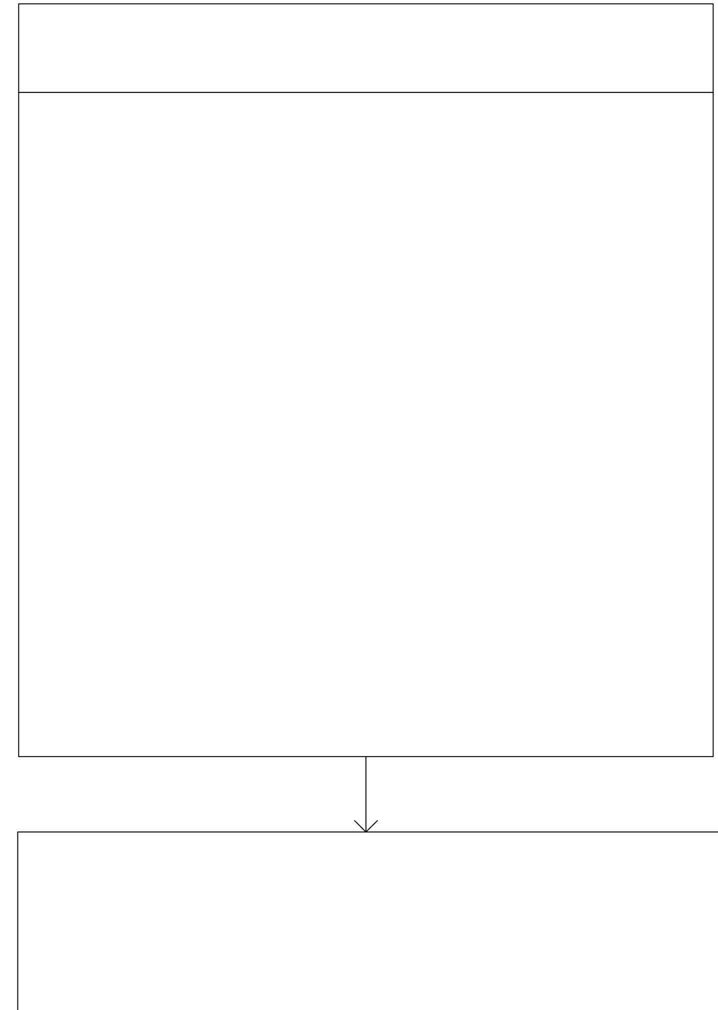
```
string  Database {get;}
int ConnectionTimeout {get;}
ConnectionState State {get;}
```

- Creates Command-Object

```
IDbCommand CreateCommand();
```

- Creates Transaction-Object

```
IDbTransaction BeginTransaction();
IDbTransaction BeginTransaction(IsolationLevel lvl);
```



IDbConnection: Property ConnectionString

- Key-value-pairs separated by semicolon (;)
- Configuration of the connection
  - name of the provider
  - identification of data source
  - authentication of user
  - other database-specific settings

- e.g.: OLEDB:

"**provider**=SQLOLEDB; **data source**=127.0.0.1\\NetSDK;
initial catalog=Northwind; **user id**=sa; **password**=; "

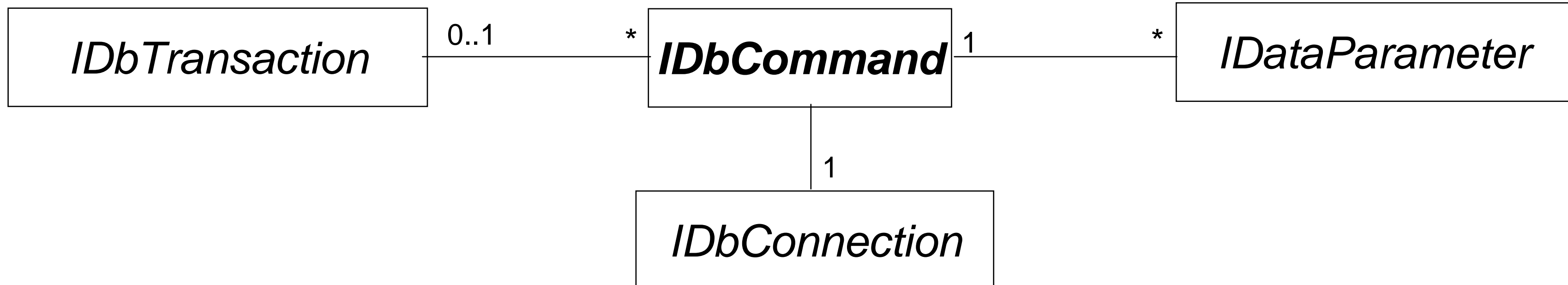
"**provider**=Microsoft.Jet.OLEDB.4.0;**data source**=c:\bin\LocalAccess40.mdb;"

"**provider**=MSDAORA; **data source**=ORACLE8i7; **user id**=OLEDB; **password**=OLEDB; "

- e.g.: MS-SQL-Server:

"**data source**=(local)\\NetSDK; **initial catalog**=Northwind; **user id**=sa;
pooling=false; **Integrated Security**=SSPI; **connection timout**=20;"

Command Objects



- Command objects define SQL statements or stored procedures
- Executed for a connection
- May have parameters
- May belong to a transaction

Interface IDbCommand

- CommandText defines SQL statement or stored procedure

```
string CommandText {get; set;}
```

- Connection object

```
IDbConnection Connection {get; set;}
```

- Type and timeout properties

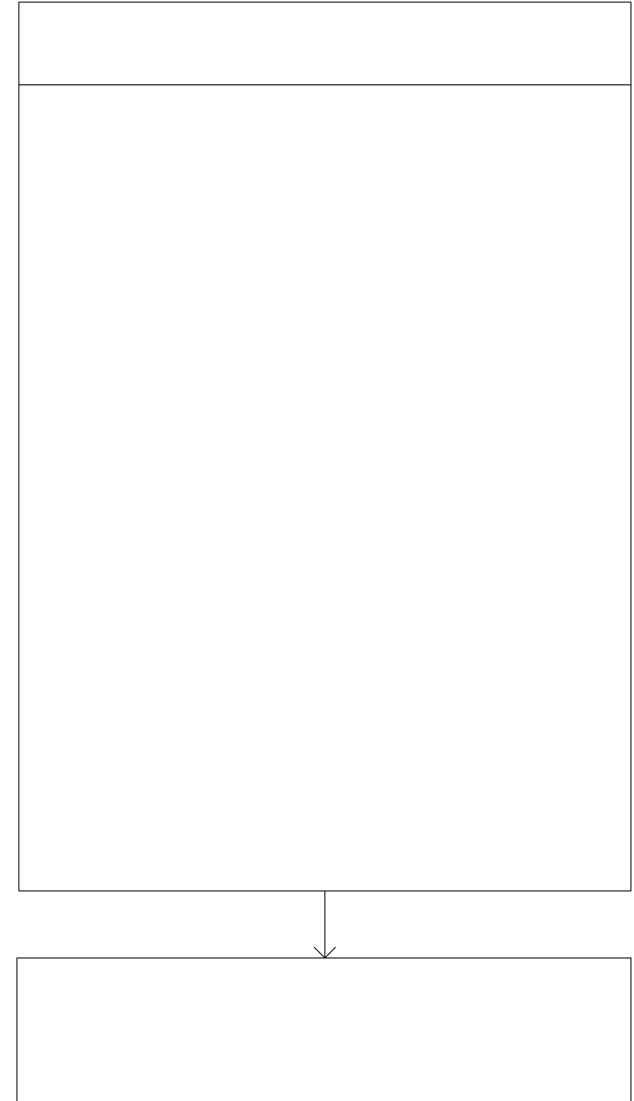
```
CommandType CommandType {get; set;}
int CommandTimeout {get; set;}
```

- Creating and accessing parameters

```
IDbDataParameter CreateParameter();
IDataParameterCollection Parameters {get;}
```

- Execution of command

```
IDataReader ExecuteReader();
IDataReader ExecuteReader(CommandBehavior b);
object ExecuteScalar();
int ExecuteNonQuery();
```



ExecuteReader Method

```
IDataReader ExecuteReader()
IDataReader ExecuteReader( CommandBehavior behavior );
```

```
public enum CommandBehavior {
    CloseConnection, Default, KeyInfo, SchemaOnly,
    SequentialAccess, SingleResult, SingleRow
}
```

- Executes the data base query specified in CommandText
- Result is an IDataReader object

<u>Example:</u>

```
cmd.CommandText =
        "SELECT EmployeeID, LastName, FirstName FROM Employees ";
IDataReader reader = cmd.ExecuteReader();
```

ExecuteNonQuery Method

```
int ExecuteNonQuery();
```

- Executes the non-query operation specified in CommandText
  - UPDATE
  - INSERT
  - DELETE
  - CREATE TABLE
  - …
- Result is number of affected rows

Example:

```
cmd.CommandText = "UPDATE Empls SET City = 'Seattle' WHERE iD=8";
int affectedRows = cmd.ExecuteNonQuery();
```

ExecuteScalar Method

```
object ExecuteScalar();
```

- Returns the value of the 1st column of the 1st row delivered by the database query
- CommandText typically is an aggregate function

Example:

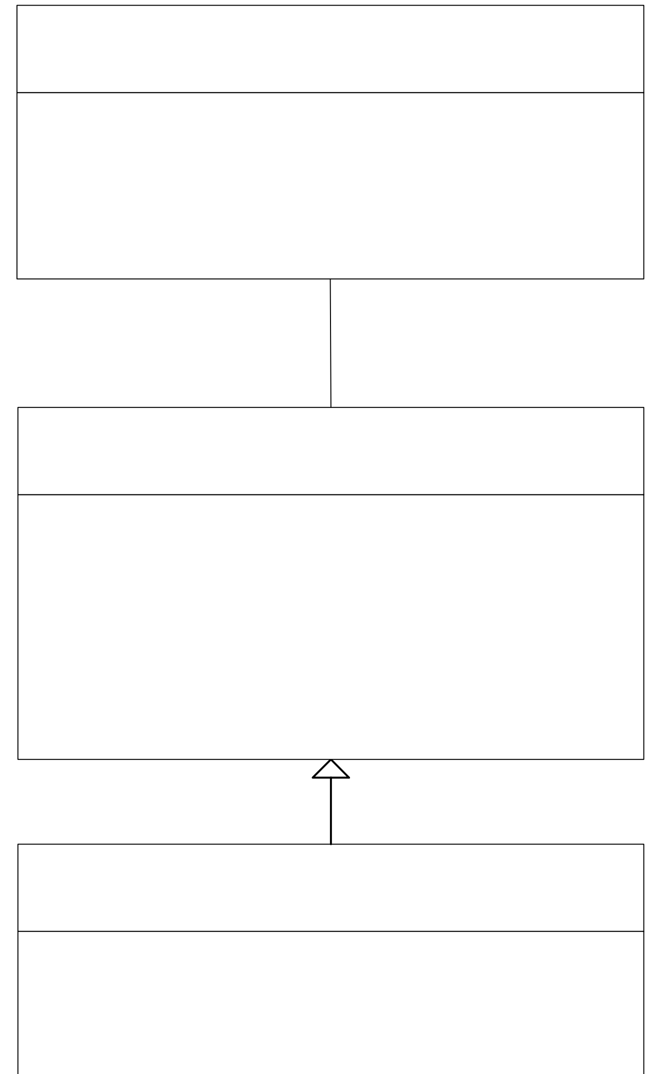
```
cmd.CommandText = " SELECT count(*) FROM Employees ";
int count = (int) cmd.ExecuteScalar();
```

Parameter

- Command objects allow for input and output parameters

```
IDataParameterCollection Parameters {get;}
```

- Parameter objects specify
  - Name: name of the parameter
  - Value: value of the parameter
  - DbDataType: data type of the parameter
  - Direction: direction of the parameter
    - Input
    - Output
    - InputOutput
    - ReturnValue



Working with Parameters

1. Define SQL command with place holders

   OLEDB: Identification of parameters by position (notation: "?")

```
OleDbCommand cmd = new OleDbCommand();
cmd.CommandText = "DELETE FROM Empls WHERE EmployeeID = ?";
```

   SQL Server: Identification of parameters by name (notation: "@*name*")

```
SqlCommand cmd = new SqlCommand();
cmd.CommandText = "DELETE FROM Empls WHERE EmployeeID = @ID";
```

2. Create and add parameter

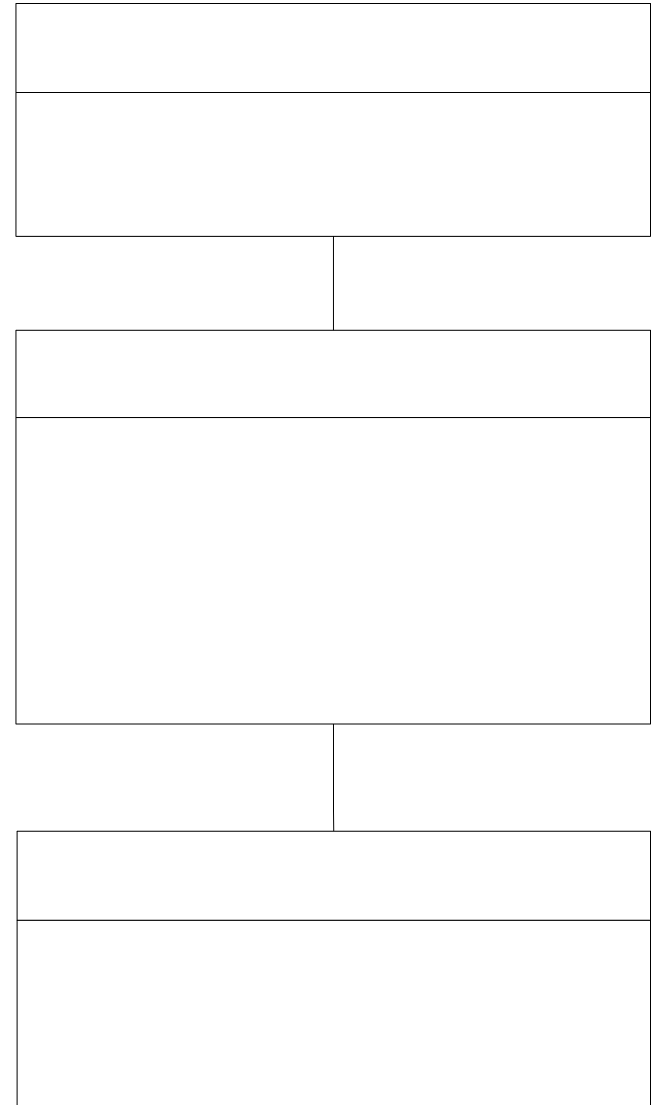
```
cmd.Parameters.Add( new OleDbParameter("@ID", OleDbType.BigInt));
```

3. Assign values and execute command

```
cmd.Parameters["@ID"].Value = 1234;
cmd.ExecuteNonQuery();
```

Transactions

- ADO.NET supports transactions
- Commands are assigned to transactions
- Execution of commands are
  - committed with Commit
  - aborted with Rollback



Working with Transactions (1)

1. Define connection and create Transaction object

```
SqlConnection con = new SqlConnection(connStr);
IDbTranaction trans = null;
try {
     con.Open();
     trans = con.BeginTransaction(IsolationLevel.ReadCommitted);
```

2. Create Command object, assign it to Transaction object, and execute it

```
     IDbCommand cmd1 = con.CreateCommand();
     cmd1.CommandText = "DELETE [OrderDetails] WHERE OrderId = 10258";
     cmd1.Transaction = trans;
     cmd1.ExecuteNonQuery();

     IDbCommand cmd2 = con.CreateCommand();
     cmd2.CommandText = "DELETE Orders WHERE OrderId = 10258";
     cmd2.Transaction = trans;
     cmd2.ExecuteNonQuery();
```

Working with Transactions (2)

3. Commit or abort transaction

```
        trans.Commit();
catch (Exception e) {
        if (trans != null)
                trans.Rollback();
} finally {
        try {
                con.Close();
        }
}
```

Isolation Levels for Transactions

- Define usage of read and write locks in transaction
- ADO.NET transactions allow different isolation levels

```
public enum IsolationLevel {
    ReadUncommitted, ReadCommitted, RepeatableRead, Serializable, ...
}
```

ReadUncommitted	• Allows reading of locked data • *Dirty reads* possible
ReadCommitted (Standard)	• Reading of locked data prohibited • No *dirty reads* but *phantom rows* can occur • *Non-repeatable reads*
RepeatableRead	• Same as *ReadCommitted* but *repeatable reads*
Seriazable	• Serialized access • *Phantom rows* cannot occur

DataReader

- ExecuteReader() returns DataReader object

```
IDataReader ExecuteReader()
IDataReader ExecuteReader( CommandBehavior behavior );
```

- DataReader allows sequential reading of result (row by row)

A	B	C

Interface IDataReader

- Read reads next row

```
bool Read();
```

- Access to column values using indexers

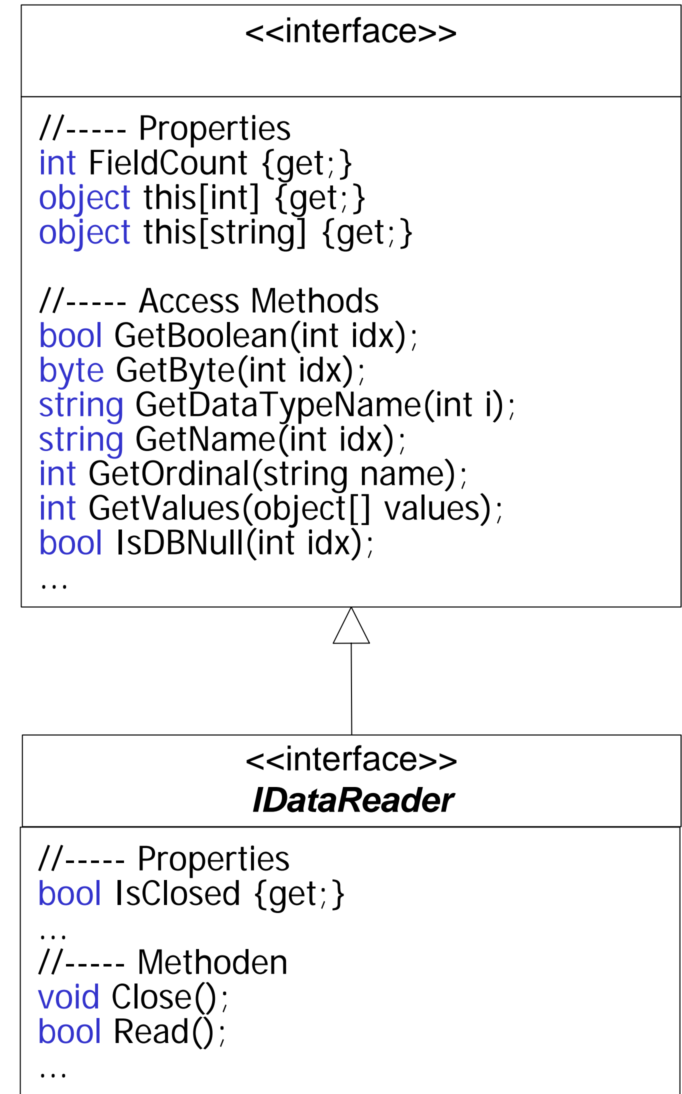
```
object this[int] {get;}
object this[string] {get;}
```

- Typed access to column values using access methods

```
bool GetBoolean(int idx);
byte GetByte(int idx);
...
```

- Getting meta information

```
string GetDataTypeName(int i);
string GetName(int idx);
int GetOrdinal(string name);
...
```



Working with IDataReader

- Create DataReader object and read rows

```
IDataReader reader = cmd.ExecuteReader();
while (reader.Read()) {
```

- Read column values into an array

```
    object[ ] dataRow = new object[reader.FieldCount];
    int cols = reader.GetValues(dataRow);
```

- Read column values using indexers

```
    object val0 = reader[0];
    object nameVal = reader["LastName"];
```

- Read column value using typed access method getString

```
    string firstName = reader.getString(2);
```

- Close DataReader

```
}
reader.Close();
```

ADO.NET

- Introduction
- Connection-oriented Access
- **Connectionless Access**
- Database Access with DataAdapter
- Integration with XML
- Preview of ADO.NET 2.0
- Summary

Motivation and Idea

SSW

- Motivation
  - Many parallel, long lasting access operations
  - Connection-oriented data access too costly

- Idea
  - Caching data in main memory

    ➔ “main memory data base“
  - Only short connections for reading and updates

    ➔ DataAdapter
  - Main memory data base independent from data source

    ➔conflicting changes are possible

Microsoft 3-Tier Architecture

Presentation Tier

Windows Form

MyApp.Exe

Dataset

Web Form

Browser

Dataset

Internet
Intranet

XML

Business Tier (IIS or Web Service)

Dataset

Data adapter

Data connection

Data adapter

Data connection

Business to Business

(BizTalk, for example)

Data Tier

taken from: Introduction to Data Access with ADO.NET, http://msdn.microsoft.com/library/

Architecture of Connectionless Data Access

connectionless

connection-oriented

ADO.NET Content C

DataSet

- Main memory data base
  - relational structure
  - object oriented interface
- DataSet consists of
  - collection of DataTables
  - collection of DataRelations
- DataTables consists of
  - collection of DataTableColumns (= schema definition)
  - collection of DataTableRows (= data)
  - DefaultView (DataTableView, see later)
- DataRelations
  - associate two DataTable objects
  - define ParentTable and ParentColumns and ChildTable and ChildColumns

DataSet Structure

DataSet

.Tables[...] → **DataTable**

schema

.Columns[...] → **DataColumn** **DataColumn**

.Rows[...] → **DataRow**
DataRow

data

.DefaultView → **DataView**

...

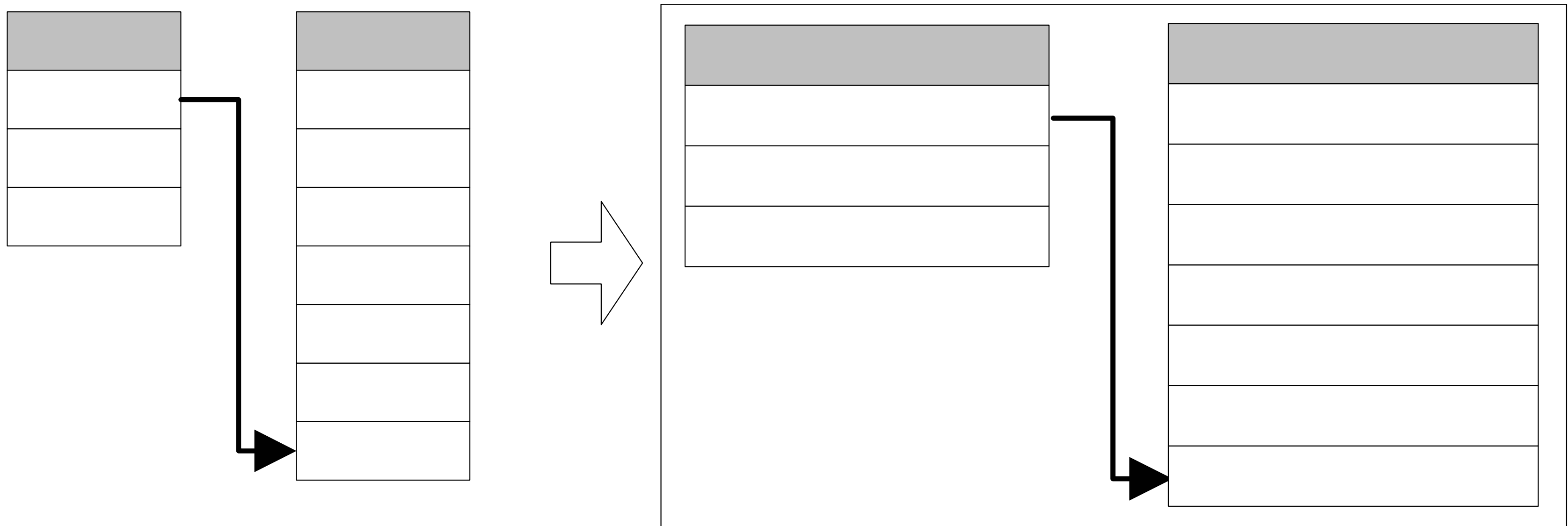
.Relations[...] → **DataRelation**

...

DataSet Class Diagram

DataS

Example: Person Contacts



Implementation steps:

- Define schema
- Define data
- Access data

Concept

Person Contacts: Define Schema (1)

- Create DataSet and DataTable "Person"

```
DataSet ds = new DataSet("PersonContacts");
DataTable personTable = new DataTable("Person");
```

- Define column "ID" and set properties

```
DataColumn col = new DataColumn();
col.DataType = typeof(System.Int64);
col.ColumnName = "ID";
col.ReadOnly = true;
col.Unique = true;                    // values must be unique
col.AutoIncrement = true;             // keys are assigned automatically
col.AutoIncrementSeed = -1;           // first key starts with -1
col.AutoIncrementStep = -1;           // next key = prev. key - 1
```

- Add column to table and set as primary key

```
personTable.Columns.Add(col);
personTable.PrimaryKey = new DataColumn[ ] { col };
```

Person Contacts: Define Schema (2)

- Define and add column "FirstName"

```
col = new DataColumn();
col.DataType = typeof(string);
col.ColumnName = "FirstName";
personTable.Columns.Add(col);
```

- Define and add column "Name"

```
col = new DataColumn();
col.DataType = typeof(string);
col.ColumnName = "Name";
personTable.Columns.Add(col);
```

- Add table to DataSet

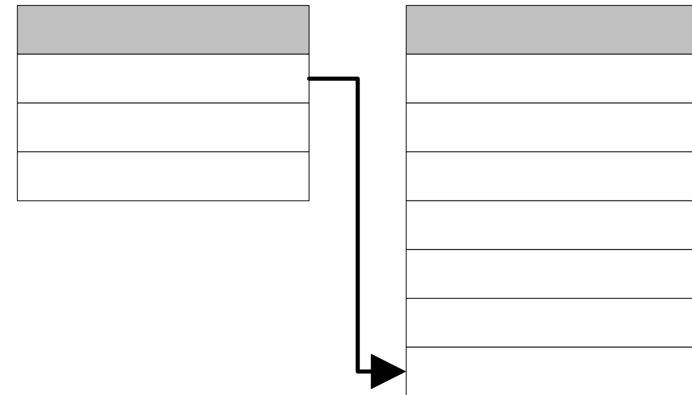
```
ds.Tables.Add(personTable);
```

- Create table "`Contact`" in similar way

```
DataTable contactTable = new DataTable("Contact");
...
ds.Tables.Add(contactTable);
```

Person Contacts: Define Relation

- Create relation PersonHasContacts
- and add it to the DataSet



```
DataColumn parentCol = ds.Tables["Person"].Columns["ID"];
DataColumn childCol = ds.Tables["Contact"].Columns["PersonID"];

DataRelation rel = new DataRelation("PersonHasContacts", parentCol, childCol);
ds.Relations.Add(rel);
```

Person Contacts: Define Data Rows

- Create new row and assign column values

```
DataRow personRow = personTable.NewRow();
personRow[1] = "Wolfgang";
personRow["Name"] = "Beer";
```

- Add row to table "Person"

```
personTable.Rows.Add(row);
```

- Create and add row to table "Contact"

```
DataRow contactRow = contactTable.NewRow ();
contactRow[0] = "Wolfgang";
...
contactRow["PersonID"] = (long)personRow["ID"]; // defines relation
contactTable.Rows.Add (row);
```

- Commit changes

```
ds.AcceptChanges();
```

Person Contacts: Access Data

- Iterate over all persons of personTable and put out the names

```
foreach (DataRow person in personTable.Rows) {
    Console.WriteLine("Contacts of {0}:", person["Name"]);
```

- Access contacts through relation "PersonHasContacts" and print out contacts

```
foreach (DataRow contact in person.GetChildRows("PersonHasContacts")) {
    Console.WriteLine("{0}, {1}: {2}", contact[0], contact["Name"], contact["Phone"] );
}
```

DataSet: Change Management

SSW

- DataSets maintain all changes
- Changes are accepted with acceptChanges
- or discarded with rejectChanges

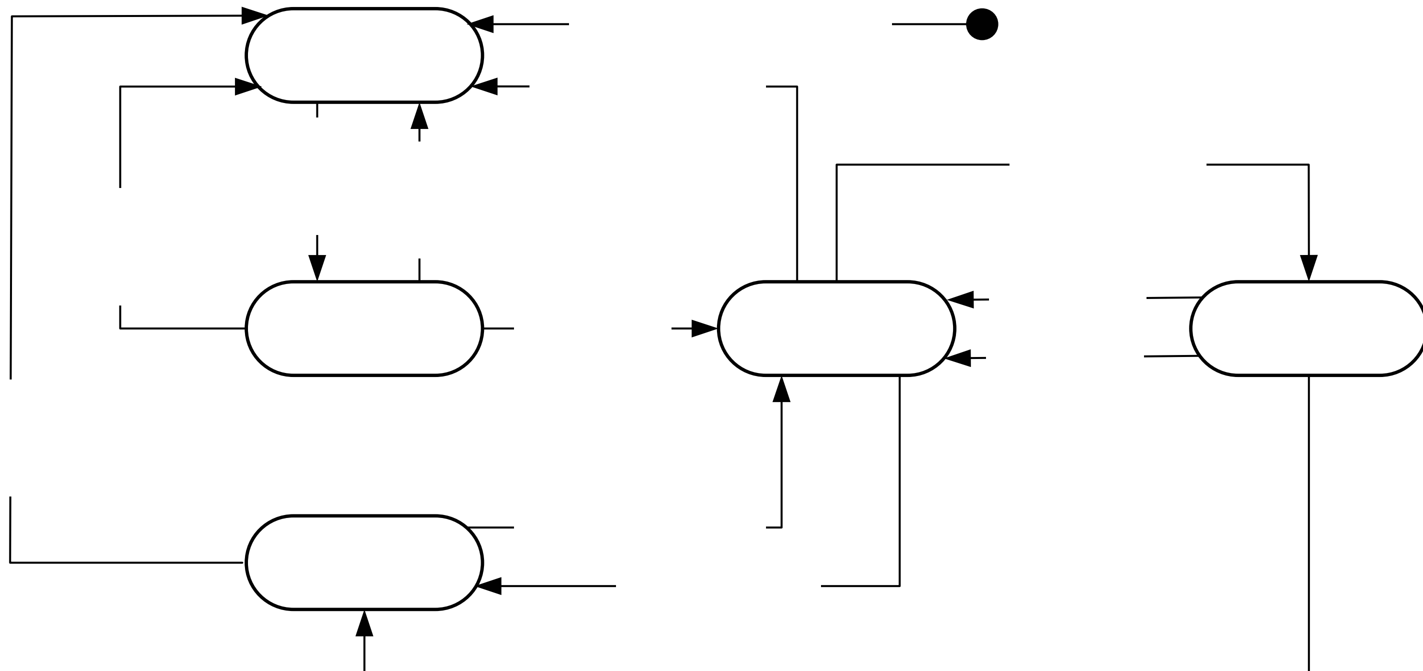
```
    ...
    if (ds.HasErrors) {
        ds.RejectChanges();
    } else {
        ds.AcceptChanges();
    }
}
```

State Diagram of a DataRow object

- DataRow objects have different states

```
public DataRowState RowState {get;}
```

```
public enum DataRowState {
    Added, Deleted, Detached, Modified, Unchanged
}
```



DataRowVersion

DataSets store different versions of data row values:

```
public enum DataRowVersion {
    Current, Original, Proposed, Default
}
```

Current: current values

Original: original values

Proposed: proposed values (values which are currently processed)

Default: standard, based on DataRowState

DataRowState	Default
Added, Modified, Unchanged	Current
Deleted	Original
Detached	Proposed

Example:

```
bool hasOriginal = personRow.HasVersion(DataRowVersion.Original);
if (hasOriginal) {
    string originalName = personRow["Name", DataRowVersion.Original];
}
```

Exception Handling

- ADO.NET checks validity of operations on DataSets
- and throws DataExceptions

```
DataException
        ConstraintException
        DeletedRowInaccessibleExcception
        DuplicateNameException
        InvalidConstraintException
        InvalidExpressionException
        MissingPrimaryKeyException
        NoNullAllowedException
        ReadOnlyException
        RowNotInTableException
        ...
```

DataView

- DataViews support views of tables

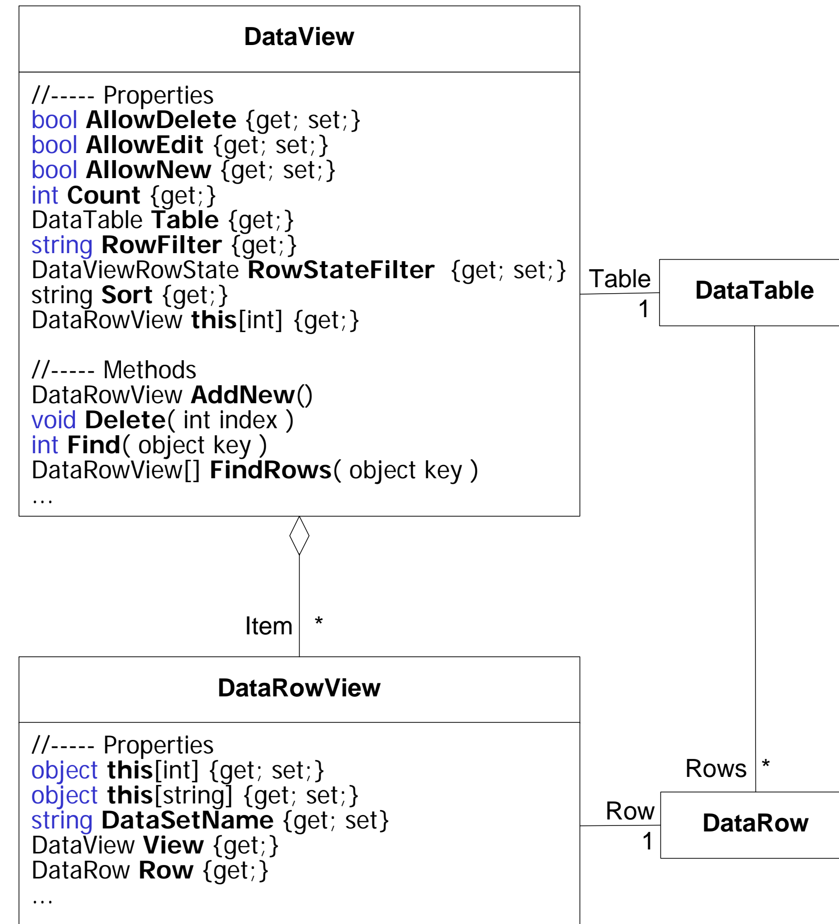
  RowFilter: Filtering based on filter expression

  RowStateFilter: Filtering based on row states

  Sort: Sorting based on columns

- DataView supports
  - changing data rows
  - fast search (based on sorted columns)

- DataView objects can be displayed by GUI elememts
  - e.g. DataGrid



Working with DataView

SSW

- Create DataView object and set filter and sorting criteria

```
DataView a_kView = new DataView(personTable);
dataView.RowFilter = "FirstName <= 'K'";
dataView.RowStateFilter =
        DataViewRowState.Added | DataViewRowState.ModifiedCurrent;
dataView.Sort = "Name ASC";              // sort by Name in ascending order
```

- Display data in DataGrid

```
DataGrid grid = new DataGrid();
...
grid.DataSource = dataView;
```

- Fast search for row based on "Name" column

```
int i = a_kView.Find("Beer");
grid.Select(i);
```

ADO.NET

Introduction

Connection-oriented Access

Connectionless Access

Database Access with DataAdapter

Integration with XML

Preview of ADO.NET 2.0

Summary

Architecture

- DataAdapter for connection to date source
  - Fill: Filling the DataSet
  - Update: Writing back changes
- DataAdapters use Command objects
  - SelectCommand
  - InsertCommand
  - DeleteCommand
  - UpdateCommand

connectionless

connection-oriented

DataAdapter Class Diagram

//----- P
MissingS
MissingN

DataAdapter: Loading Data

- Create DataAdapter object and set SelectCommand

```
IDbDataAdapter adapter = new OleDbDataAdapter();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = new OleDbConnection ("provider=SQLOLEDB; ..." );
cmd.CommandText = "SELECT * FROM Person";
adapter.SelectCommand = cmd;
```

- Read data from data source and fill DataTable "Person"

```
adapter.Fill(ds, "Person");
```

- Accept or discard changes
- Delete DataAdapter object

```
if (ds.HasErrors) ds.RejectChanges();
else ds.AcceptChanges();
if (adapter is IDisposable) ((IDisposable)adapter).Dispose();
```

DataAdapter: Loading Schema and Data

- Create DataAdapter object and set SelectCommand

```
IDbDataAdapter adapter = new OleDbDataAdapter();
OleDbCommand cmd = new OleDbCommand();
cmd.Connection = new OleDbConnection ("provider=SQLOLEDB; ..." );
cmd.CommandText = "SELECT * FROM Person; SELECT * FROM Contact";
adapter.SelectCommand = cmd;
```

- Define action for missing schema and mapping to tables

```
adapter.MissingSchemaAction = MissingSchemaAction.AddWithKey;
adapter.TableMappings.Add("Table", "Person");
adapter.TableMappings.Add("Table1", "Contact");
```

- Read data from data source and fill DataTable "Person"

```
adapter.Fill(ds);
```

- Accept or discard changes; delete DataAdapter object

```
if (ds.HasErrors) ds.RejectChanges();
else ds.AcceptChanges();
if (adapter is IDisposable) ((IDisposable)adapter).Dispose();
```

DataAdapter: Writing Back Changes (1)

- Changes are written back with Update method
- Update-, Insert- and DeleteCommand define how changes are written
- CommandBuilder can create Update-, Insert- und DeleteCommand from SelectCommand automatically (in simple cases)
- Conflict management for updates:
  - comparison of data in DataTable and data source
  - in case of conflict DBConcurrencyException is thrown

DataAdapter: Writing Back Changes (2)

- Create DataAdapter with SELECT expression

```
OleDbConnection con = new OleDbConnection ("provider=SQLOLEDB; …");
adapter = new OleDbDataAdapter("SELECT * FROM Person", con);
```

- Create update commands using CommandBuilder

```
OleDbCommandBuilder cmdBuilder = new OleDbCommandBuilder(adapter);
```

- Call Update and handle conflicts

```
try {
    adapter.Update(ds, tableName);
} catch (DBConcurrencyException) {
    // Handle the error, e.g. by reloading the DataSet
}
adapter.Dispose();
```

DataAdapter: Event Handling

- Two events signaled on updates for each data row
  - OnRowUpdating: just before updating the data source
  - OnRowUpdated: just after updating the data source

```
public sealed class OleDbDataAdapter : DbDataAdapter, IDbDataAdapter
{
    public event OleDbRowUpdatingEventHandler RowUpdating;
    public event OleDbRowUpdatedEventHandler RowUpdated;
    …
}
```

```
public delegate void OleDbRowUpdatedEventHandler( object sender,
                                                   OleDbRowUpdatedEventArgs e );
```

```
public sealed class OleDbRowUpdatedEventArgs : RowUpdatedEventArgs {
    public DataRow Row {get;}
    public StatementType StatementType {get;}
    public UpdateStatus Status {get; set;}
    …
}
```

DataAdapter: Event Handling Example

- Define handler methods

```
private void onRowUpdating(object sender, OleDbRowUpdatedEventArgs args) {
    Console.WriteLn("Updating row for {0}", args.Row[1]);
    ...
}
```

```
private void onRowUpdated(object sender, OleDbRowUpdatedEventArgs args) {
    ...
}
```

- Add delegates to events of DataAdapter

```
OleDbDataAdapter adapter = new OleDbDataAdapter();
...
da.RowUpdating += new OleDbRowUpdatingEventHandler(this.OnRowUpdating);
da.RowUpdated += new OleDbRowUpdatingEventHandler(this.OnRowUpdated);
```

ADO.NET

Introduction

Connection-oriented Access

Connectionless Access

Database Access with DataAdapter

Integration with XML

Preview of ADO.NET 2.0

Summary

Integration DataSets und XML

- DataSets and XML are highly integrated
  - serializing DataSets as XML data
  - XML documents as data sources for DataSets
  - schemas for DataSets defined as XML schemas
  - *strongly typed* DataSets generated from XML schemas
  - access to DataSets using XML-DOM interface

- Integration of DataSets and XML used in distributed systems, e.g., web services
  - (see *Microsoft 3-Tier Architecture*)

Writing and Reading XML Data

- Methods for writing and reading XML data

```
public class DataSet : MarshalByValueComponent, IListSource,
                                    ISupportInitialize, ISerializable {
    public void WriteXml( Stream stream );
    public void WriteXml( string fileName );
    public void WriteXml( TextWriter writer);
    public void WriteXml( XmlWriter writer );
    public void WriteXml( Stream stream, XmlWriteMode m );
    public void ReadXml ( Stream stream );
    public void ReadXml ( string fileName );
    public void ReadXml ( TextWriter writer);
    public void ReadXml ( XmlWriter writer );
    public void ReadXml ( Stream stream, XmlReadMode m );
    ...
}
```

```
public enum XmlWriteMode {DiffGram, IgnoreSchema, WriteSchema}
```

```
public enum XmlReadMode {
    Auto, DiffGram, IgnoreSchema, ReadSchema, InferSchema, Fragment }
```

Example: Writing and Reading XML Data

- Write data to XML file

```
ds.writeXML("personcontact.xml");
```

- Read data from XML
  - with XmlReadMode.Auto a schema is generated automatically

```
DataSet ds = new DataSet();
ds.readXML("personcontact.xml",
             XmlReadMode.Auto);
```

```
<?xml version="1.0" standalone="yes" ?>
<PersonContacts>
-  <Person>
      <ID>1</ID>
      <FirstName>Wolfgang</FirstName>
      <Name>Beer</Name>
   </Person>
-  <Person>
      <ID>2</ID>
      <FirstName>Dietrich</FirstName>
      <Name>Birngruber</Name>
   </Person>
   <Contact>
      <ID>1</ID>
      <FirstName>Dietrich</FirstName>
      <Name>Birngruber</Name>
      <NickName>Didi</NickName>
      <EMail>didi@dotnet.jku.at</EMail>
      <Phone>7133</Phone>
      <PersonID>2</PersonID>
   </Contact>
-  <Contact>
      <ID>2</ID>
      <FirstName>Wolfgang</FirstName>
      <Name>Beer</Name>
      ...
      <PersonID>1</PersonID>
   </Contact>
  </PersonContacts>
```

DataSet and XML Schema

- DataSets allow reading and writing XML schemas
  - WriteXmlSchema: Writes XML schema
  - ReadXmlSchema: Reads XML schema and constructs DataSet
  - InferXmlSchema: Reads XML data and infers schema from the data

```
    ...
    public void WriteXmlSchema ( Stream stream );
    public void WriteXmlSchema ( string fileName );
    public void WriteXmlSchema ( TextWriter writer);
    public void WriteXmlSchema ( XmlWriter writer );

    public void ReadXmlSchema  ( Stream stream );
    public void ReadXmlSchema  ( string fileName );
    public void ReadXmlSchema  ( TextWriter writer);
    public void ReadXmlSchema  ( XmlWriter writer );

    public void InferXmlSchema ( Stream stream, string[] namespaces );
    public void InferXmlSchema ( string fileName, string[] namespaces );
    public void InferXmlSchema ( TextWriter writer, string[] namespaces );
    public void InferXmlSchema ( XmlWriter writer, string[] namespaces );
}
```

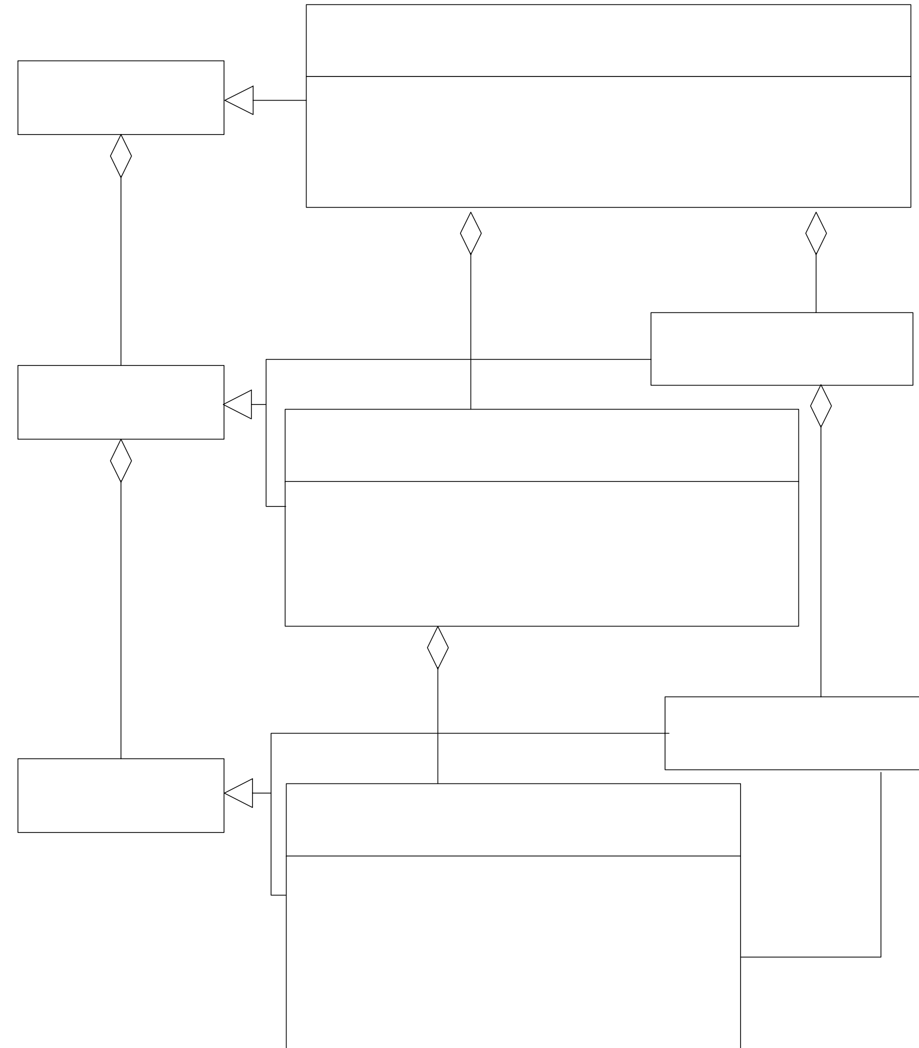
Typed DataSets

- *Typed DataSets* provide typed data access

- Tool xsd.exe generates classes from XML schema

```
> xsd.exe personcontact.xsd /dataset
```

- Classes define properties for typed access to rows, columns, and relations



Example Typed DataSets

- Data access in conventional DataSet

```
DataSet ds = new DataSet("PersonContacts");
DataTable personTable = new DataTable("Person");
...
ds.Tables.Add(personTable);
DataRow person = personTable.NewRow();
personTable.Rows.Add(person);
person["Name"] = "Beer";
...
person.GetChildRows("PersonHasContacts")[0]["Name"] = "Beer";
```

- Data access in typed DataSet

```
PersonContacts typedDS = new PersonContacts();
PersonTable personTable = typedDS.Person;
Person person = personTable.NewPersonRow();
personTable.AddPersonRow(person);
person.Name = "Beer";
...
person.GetContactRows()[0].Name = "Beer";
```

Access to DataSets using XML-DOM

- XmlDataDocument allows access over XML-DOM interface
- Synchronisation of changes in XmlDataDocument and DataSet

Example:

- Create XmlDataDocument object for DataSet objekt
- Change data in DataSet

```
XmlDataDocument xmlDoc = new XmlDataDocument(ds);
...
DataTable table = ds.Tables["Person"];
table.Rows.Find(3)["Name"] = "Changed Name!";
```

- Access changed data from XmlDataDocument object

```
XmlElement root = xmlDoc.DocumentElement;
XmlNode person = root.SelectSingleNode("descendant::Person[ID='3']");
Console.WriteLine("Access via XML: \n" + person.OuterXml);
```

ADO.NET

- Introduction
- Connection-oriented Access
- Connectionless Access
- Database Access with DataAdapter
- Integration with XML
- **Preview of ADO.NET 2.0**
- Summary

ADO.NET 2.0

- Extended interfaces
- Tight coupling with MS SQL Server 9.0 („Yukon“)

New features are (many only available for MS SQL Server 9.0):

- bulk copy operation
- *Multiple Active Result Sets* (*MARS*)
- asynchronous execution of database operations
- batch processing of database updates
- paging through the result of a query
- *ObjectSpaces*

Bulk Copy Operation

- Inserting a large amount of data in one operation (only for MS SQL Server)
- Provided by class SqlBulkCopyOperation

Example

1. Define data source

```
SqlConnection sourceCon = new SqlConnection(conString); sourceCon.Open();
SqlCommand sourceCmd = new SqlCommand("SELECT * FROM Customers",sourceCon);
IDataReader sourceReader = sourceCmd.ExecuteReader();
```

2. Define target

```
SqlConnection targetCon = new SqlConnection(conString); targetCon.Open();
```

3. Copy data from source to target in one operation

```
SqlBulkCopyOperation bulkCmd = new SqlBulkCopyOperation(targetCon);
bulkCmd.DestinationTableName = "Copy_Customers";
bulkCmd.WriteDataReaderToServer(sourceReader);
```

Multiple Active Result Sets (MARS)

- So far only one DataReader for one connection allowed
- ADO.NET 2.0 allows several DataReaders in parallel

```
SqlConnection con = new SqlConnection(conStr);
con.Open();
SqlCommand custCmd = new SqlCommand("SELECT CustomerId, CompanyName " +
    "FROM Customers ORDER BY CustomerId", con);
SqlCommand ordCmd = new SqlCommand("SELECT CustomerId, OrderId, OrderDate " +
    "FROM Orders ORDER BY CustomerId, OrderDate", con);
SqlDataReader custRdr = custCmd.ExecuteReader();
SqlDataReader ordRdr = ordCmd.ExecuteReader();
string custID = null;
while (custRdr.Read()) { // use the first reader
    custID = custRdr.GetString(0);
    while (ordRdr.Read() && ordRdr.GetString(0) == custID ) {  // use the second reader
        ...
    }
}
…
```

Asynchronous Operations

- So far only synchronous execution of commands
- ADO.NET 2.0 supports asynchronous execution mode (similar to asynchronous IO operations)

```
IAsyncResult   BeginExecuteReader (AsyncCallback callback)
IDataReader    EndExecuteReader (AsyncResult result)

IAsyncResult   BeginExecuteNonQuery (AsyncCallback callback)
int            EndExecuteNonQuery (IAsyncResult result)

IAsyncResult   BeginExecuteXmlReader (AsyncCallback callback)
IDataReader    EndExecuteXmlReader (IAsyncResult result)
```

Example Asynchronous Operations

```
...
public class Async {
    SqlCommand cmd; // command to be executed asynchronously
    public void CallCmdAsync() {
        SqlConnection con = new SqlConnection("Data Source=(local)\\NetSDK...");
        cmd = new SqlCommand("MyLongRunningStoredProc", con);
        cmd.CommandType = CommandType.StoredProcedure;
        con.Open();
        // execute the command asynchronously
        cmd.BeginExecuteNonQuery(new AsyncCallback(AsyncCmdEnded), null);
        ...
    }

    // this callback method is executed when the SQL command is finished
    public void AsyncCmdEnded(IAsyncResult result) {
        cmd.EndExecuteNonQuery(result);
        // optionally do some work based on results
        ...
    }
}
```

Batch Processing of Database Updates

- So far rows are updated individually
- With ADO.NET 2.0 several rows can be updated in one batch (only available for MS SQL Server)
- UpdateBatchSize can be specified for DataAdapter

```
void UpdateCategories(DataSet ds, SqlConnection con) {
    // create an adapter with select and update commands
    SqlDataAdapter da = new SqlDataAdapter("SELECT * FROM Categories", con);
    // the command builder creates the missing UPDATE, INSERT and DELETE commands
    SqlCommandBuilder cb = new SqlCommandBuilder(da);
    // set the batch size != 1
    da.UpdateBatchSize = 50;
    …
    // execute the update in batch mode
    da.Update(ds.Tables["Categories"] );
}
```

Paging

- Operation ExecutePageReader allows accessing a subset of rows

```
ExecutePageReader(CommandBehavior b, int startRow, int pageSize)
```

➔Very useful in combination with user interface controls (e.g. DataGrid)

ObjectSpaces

- ObjectSpaces allow mapping of objects and relational data
- Mapping defined in language *OPath* which (based on *XPath*)

Classes of ObjectSpaces

ObjectSpace: for communication with the data source

ObjectSources: list of connections to the data source

ObjectQuery: for reading objects with OPath

ObjectSet: stores the objets (similar to DataSet)

ObjectList and ObjectHolder: collections for delayed reading of objects

Example ObjectSpaces

```
public class Customer { // mapped class
    public string Id; // primary key
    public string Name;
    public string Company;
    public string Phone;
}
public class ObjectSpaceSample {
    public static void Main() {
        // load the mapping and data source information and create the ObjectSpace.
        SqlConnection con = new SqlConnection("Data Source=(local)\\NetSDK; ...");
        ObjectSpace os = new ObjectSpace("map.xml", con);
        // query for objects
        ObjectQuery oQuery = new ObjectQuery(typeof(Customer),"Id >= 'T'", "");
        ObjectReader reader = os.GetObjectReader(oQuery);
        // print result
        foreach (Customer c in reader) {
            Console.WriteLine(c.GetType() + ":");
            Console.WriteLine("Id: " + c.Id);
            Console.WriteLine("Name: " + c.Name);
            Console.WriteLine("Phone: " + c.Phone);
        }
        reader.Close();
        con.Close();
    }
}
```

ADO.NET

Introduction

Connection-oriented Access

Connectionless Access

Database Access with DataAdapter

Integration with XML

Preview of ADO.NET 2.0

Summary

Summary

- Connection-oriented data access model
  - for applications with only a few parallel, short running transactions
  - object-oriented interface abstracts from data source
  - access to database by SQL commands

- Connectionless data access model
  - for applications with many parallel, long running transactions
  - DataSet as main memory data base
  - DataAdapter is used as connector to the data source
  - tight integration with XML
  - well integrated in the .NET Framework (e.g.: WebForms, WinForms)