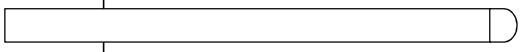
Laboratorio di Ingegneria del Software L-A

ADO.NET



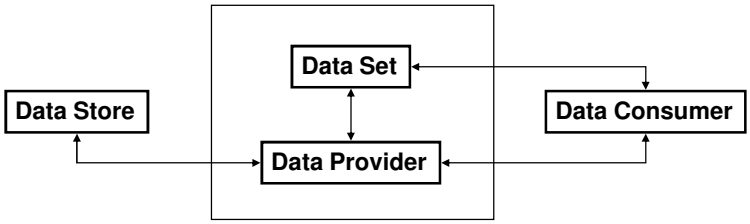
ADO.NET

- Tutte le applicazioni moderne hanno bisogno di accedere ad un qualche tipo di “base di dati”
  - Database server (SQL Server, Oracle, DB2, ecc.)
  - Documenti XML
  - File di testo (txt, csv, ecc.)
  - File indicizzati (.mdb, .dbf, ecc.)
- La modalità di accesso ai dati dovrebbe essere il più indipendente possibile
  - dal tipo di base di dati
  - dal tipo di applicazione
    - Applicazioni Windows
    - Pagine web dinamiche
    - Web Service

Laboratorio di Ingegneria del Software L-A

ADO.NET

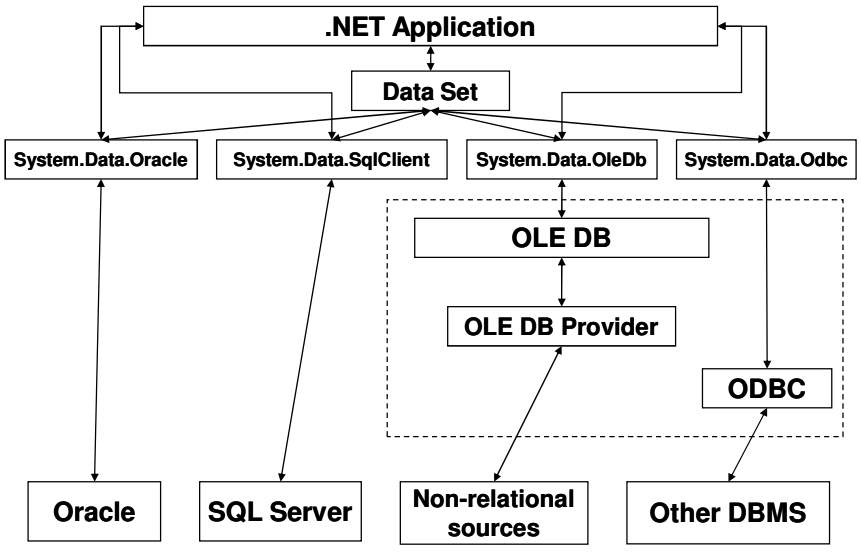
- ActiveX Data Objects
- Insieme di tipi (classi, strutture, interfacce, …) divisi, dal punto di vista funzionale, in due gruppi fondamentali
  - **Data Set** – tipi che gestiscono la rappresentazione dei dati
  - **Data Provider** – tipi che gestiscono la comunicazione con un *data store*



8.3

Laboratorio di Ingegneria del Software L-A

ADO.NET

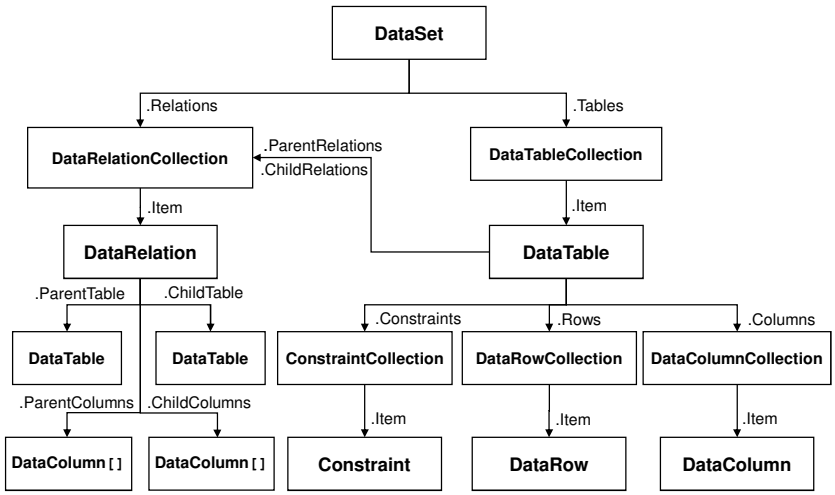


8.4

Data Set

- Permette di gestire i dati come se si disponesse di un semplice database relazionale
  - Residente in memoria di lavoro
  - Disconnesso dalla sorgente dati esterna
- Comprende vari tipi di oggetti:
  - **DataSet** – rappresenta il database in memoria e comprende:
    - Un insieme di tabelle
    - Un insieme di relazioni tra tabelle
  - **DataTable** – rappresenta una tabella
  - **DataColumn** – rappresenta una colonna di una tabella
  - **DataRow** – rappresenta una riga di una tabella
  - **Constraint** – rappresenta un vincolo di integrità su una tabella
  - **DataRelation** – rappresenta una relazione tra due tabelle

Data Set Object Model



Laboratorio di Ingegneria del Software L-A

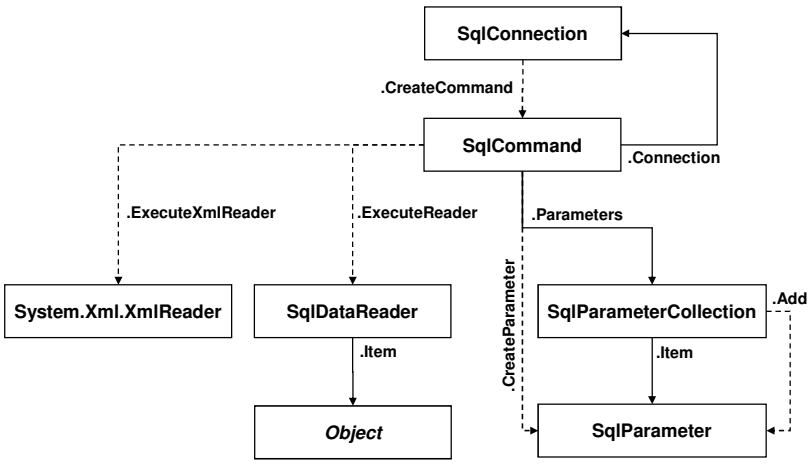
Data Provider

- Permette di ottenere/inviare dati da/a un qualsiasi tipo di sorgente dati
  - Relazionale
  - XML
  - *Custom*
- Specifico per una sorgente dati
- Comprende vari tipi di oggetti:
  - **Connection** – rappresenta la connessione fisica con la sorgente dati
  - **Command** – rappresenta un comando da eseguire sulla sorgente dati (ad es. un'istruzione SQL)
  - **DataReader** – permette di ottenere in modo veloce, *read-only* e *forward-only* i dati dalla sorgente dati
  - **DataAdapter** – fornisce il meccanismo di interazione tra la connessione e il data set

8.7

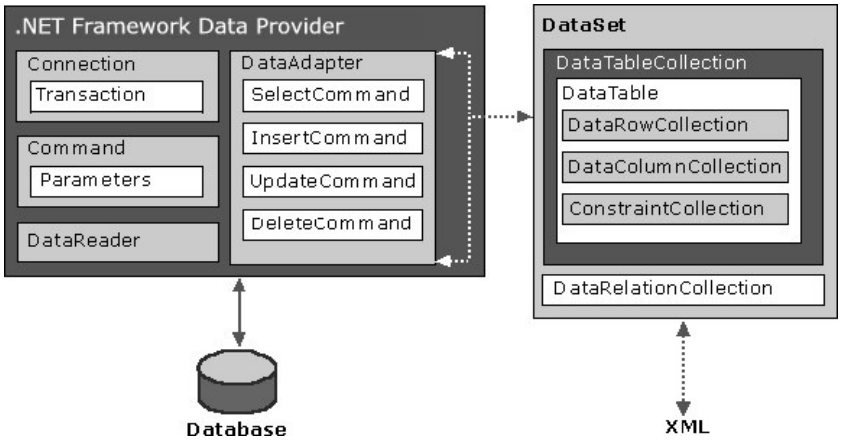
Laboratorio di Ingegneria del Software L-A

SqlClient .NET
Data Provider Object Model



8.8

Architettura di ADO.NET



`System.Data.DataSet`

- Non dipende dalla sorgente dati
- Le sue tabelle
  - possono essere popolate con dati provenienti da sorgenti differenti
  - utilizzano tipi di dati .NET, indipendenti dalla sorgente
- Può essere costruito
  - completamente in automatico
  - completamente via codice
  - parte in automatico e parte via codice
- Può essere utilizzato come *data store* indipendente

System.Data.DataSet

- Può essere "***typed***" o "***untyped***"
- Nel caso di *untyped* **DataSet**, si scrive:
  - **Biblioteca.Tables["Libri"].Rows[0]["Id"]**
- Nel caso di *typed* **DataSet**, si può anche scrivere:
  - **Biblioteca.Libri[0].Id**
- Per creare un *untyped* **DataSet**:
  - Drag dalla toolbox del DataSet
  - Inserimento delle tabelle
- Per creare un *typed* **DataSet**:
  - Add new item → Data Set
  - Drag dalla toolbox di un Element per ogni tabella

System.Data.DataTable

- Il vero contenitore dei dati in memoria
- La sua struttura è descritta da una collezione di **DataColumn**
  - proprietà **Columns**
- I suoi dati sono contenuti in una collezione di **DataRow**
  - proprietà **Rows**
- Consente la definizione di una chiave primaria, composta da una o più colonne
  - proprietà **PrimaryKey**
- Consente la creazione di colonne calcolate
  - proprietà **Expression** di **DataColumn**

System.Data.DataTable

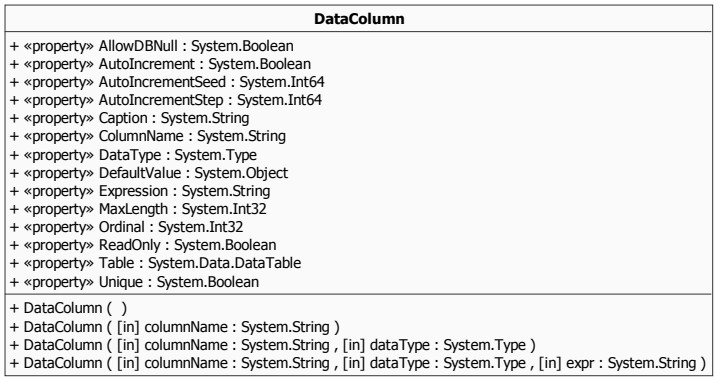
```
//  Creazione di una nuova DataTable
DataTable table = new DataTable("Libri");
//  Inserimento della DataTable nel DataSet
Biblioteca.Tables.Add(table);
//  Definizione della struttura della DataTable
DataColumn columnId = new DataColumn("Id", typeof(int));
table.Columns.Add(columnId);
table.Columns.Add("Titolo", typeof(string));
table.PrimaryKey = new DataColumn[] { columnId };
//  Aggiunta di una riga alla DataTable
DataRow dataRow = table.NewRow();
dataRow["Id"] = 1;
dataRow["Titolo"] = "Primo libro";
table.Rows.Add(dataRow);
```


System.Data.DataTable

- **`UniqueConstraint`** consente di definire **vincoli di univocità** su una o più colonne
  - `dt.Constraints.Add(new UniqueConstraint( dt.Columns[…]));`
- **`ForeignKeyConstraint`** consente di definire ***foreign key*** su una o più colonne
  - `dt.Constraints.Add(new ForeignKeyConstraint( dtmaster.Columns[…],dt.Columns[…]));`
  - Le proprietà **`DeleteRule`** e **`UpdateRule`** consentono di ottenere la *Cascading Referential Integrity*
  - La proprietà **`AcceptRejectRule`** consente di propagare le modifiche fatte su una tabella “master” a una o più “details”

Laboratorio di Ingegneria del Software L-A

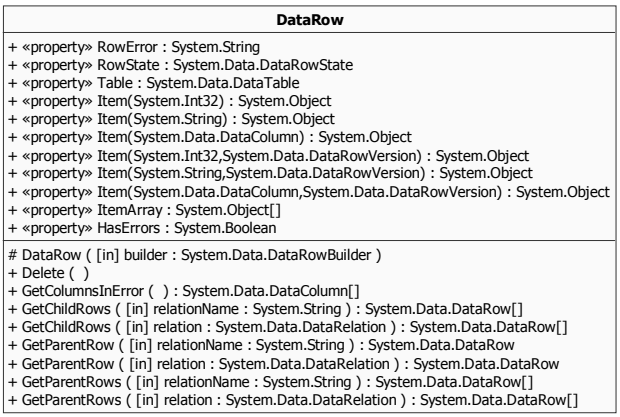
System.Data.DataColumn



8.15

Laboratorio di Ingegneria del Software L-A

System.Data.DataRow



8.16

Creazione di un `DataSet` da codice

- Il **`DataSet`** è ottimo per costruire strutture dati in grado di mantenere in cache locale le informazioni
  - Creo il **`DataSet`**
    ```
    DataSet custDS = new DataSet("CustomerOrders");
    ```
  - Creo una o più **`DataTable`**
    ```
    DataTable ordersTable = new DataTable("Orders");
    ordersTable.Columns.Add("OrderID", typeof(Int32));
    ordersTable.Columns.Add("Quantity", typeof(Int32));
    ordersTable.Columns.Add("CustID", typeof(Int32));
    ordersTable.PrimaryKey =
     new DataColumn[] {ordersTable.Columns["OrderID"]};
    …
    ```
  - Aggiungo al **`DataSet`** le **`DataRelation`** tra le tabelle
    ```
    custDS.Relations.Add("CustOrders",
     custDS.Tables["Customers"].Columns["CustID"],
     custDS.Tables["Orders"].Columns["CustID"]);
    ```


`DataRelation` e navigazione tra i dati

- Le **`DataRelation`**, oltre a gestire l'integrità referenziale, consentono di navigare tra i dati
- Da una **`DataRow`** di una tabella è possibile ottenere le righe della tabella collegata mediante i metodi
  - **`GetChildRows`**
  - **`GetParentRow`**
  - **`GetParentRows`**

```
DataRelation dataRel = custDS.Relations.Add("CustOrders",
   custDS.Tables["Customers"].Columns["CustID"],
   custDS.Tables["Orders"].Columns["CustID"]);
foreach (DataRow custRow in custDS.Tables["Customers"].Rows)
{
  Console.WriteLine(custRow["CustID"]);
  foreach (DataRow orderRow in custRow.GetChildRows(dataRel))
    Console.WriteLine(orderRow["OrderID"]);
}
```

Modifiche ai dati

- Ogni **`DataRow`** ha uno stato che è interrogabile mediante la proprietà **`RowState`**
- **`DataRow dataRow = dataTable.NewRow();`** crea una nuova **`DataRow`** in uno stato **`Detached`**
- **`dataTable.Add(dataRow);`** modifica lo stato di **`dataRow`** in **`Added`**
- **`dataRow["Name"] = "Pippo";`** modifica lo stato di **`dataRow`** in **`Modified`**
- **`dataRow.Delete()`** modifica lo stato di **`dataRow`** in **`Deleted`**

«enumeration»
DataRowState

Detached = 1
Unchanged = 2
Added = 4
Deleted = 8
Modified = 16

«enumeration»
DataRowVersion

Original = 256
Current = 512
Proposed = 1024
Default = 1536

- Una **`DataTable`** può mantenere più versioni della stessa **`DataRow`**
- Le modifiche ad una **`DataRow`** vengono eseguite sulla versione **`Current`** della **`DataRow`**

Modifiche ai dati

- Se durante la modifica di una riga si è riscontrato un errore, è possibile associare un messaggio di errore
  - all'intera riga
    `dataRow.RowError = messaggioDiErrore`
  - ad una colonna specifica
    `dataRow.SetColumnError(colonna,messaggioDiErrore)`
- È possibile verificare se una riga contiene degli errori
  - **`dataRow.HasErrors`**
- È possibile ottenere i messaggi di errore associati
  - all'intera riga
    `dataRow.RowError`
  - ad una colonna specifica
    `dataRow.GetColumnError(colonna)`
- È possibile cancellare tutti gli errori associati ad una riga
  - **`dataRow.ClearErrors()`**

Modifiche ai dati

- Le modifiche rimangono nella cache locale e verranno propagate alla sorgente dati (ad es. al database) mediante **`DataAdapter`**
- Il metodo **`AcceptChanges`** di **`DataSet`**, **`DataTable`** e **`DataRow`**
  - Esegue la ***commit*** di tutte le modifiche nella cache locale
  - Riporta allo stato **`Unchanged`** le righe inserite e modificate
  - Rimuove le eliminate.
- Il metodo **`RejectChanges`** di **`DataSet`**, **`DataTable`** e **`DataRow`**
  - Esegue un ***rollback*** delle modifiche
  - Le righe tornano allo stato originale

Selezionare e ordinare i dati

- Mediante il metodo **`Select`** è possibile
  - selezionare le righe di una **`DataTable`**,
  - eventualmente, ordinare le righe selezionate e
  - ottenere come risultato un'array di **`DataRow`**

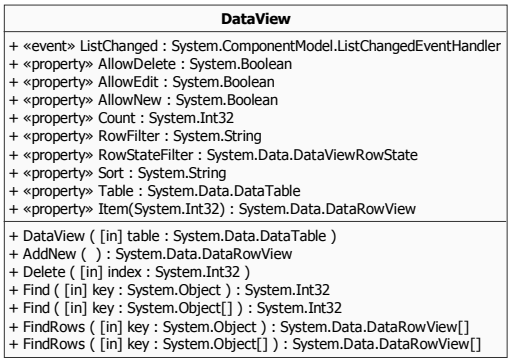
```
DataRow[] Select(string filterExpression)
DataRow[] Select(string filterExpression,string sort)

libri.Select("id > 10", "titolo DESC")
```

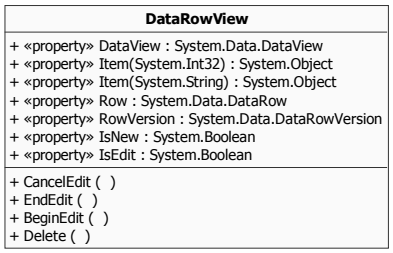
System.Data.DataView

- Costruita su una **DataTable** per
  - sorting, filtering, searching, editing, and navigation
- Permette di selezionare le righe della **DataTable**, ma non permette di selezionare le colonne
- Utilizzata principalmente per poter mostrare i dati della stessa tabella in modo diverso
  - Data Binding e Ordinamenti
- Contiene una collezione di oggetti di tipo **DataRowView**
- Permette di inserire, modificare e cancellare righe della **DataTable** sulla quale è costruita

System.Data.DataView



System.Data.DataRowView



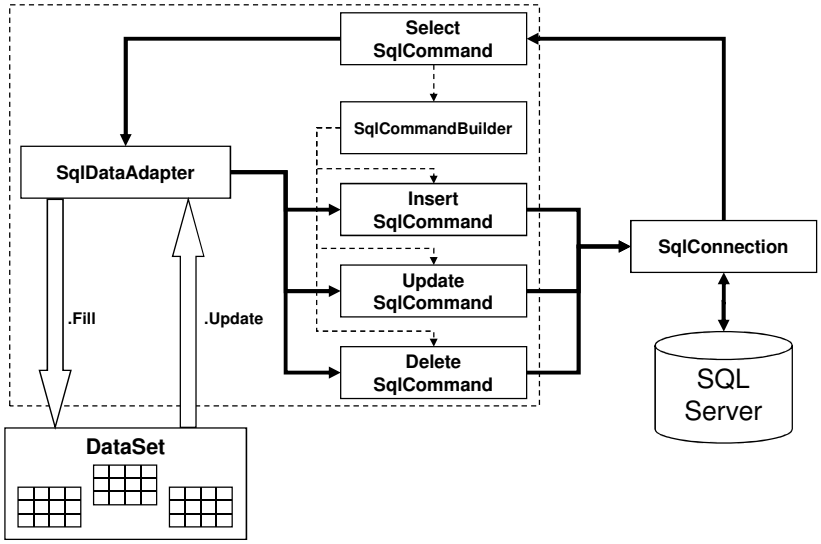
DataAdapter

- Fornisce il meccanismo di interazione tra
  - la parte "connessa" alla sorgente dei dati
  - la parte "disconnessa" del **`DataSet`**
- È costituito
  - da un **insieme di comandi** e
  - da una **connessione** alla sorgente dati

  che vengono utilizzati per
  - riempire il **`DataSet`** con i dati letti dalla sorgente
  - aggiornare la sorgente per tenere conto di operazioni di
    - inserimento
    - modifica
    - cancellazione

    effettuate sul **`DataSet`**
- Tipicamente si utilizza un **`DataAdapter`** per **`DataTable`**

Architettura del `DataAdapter`



Metodo `Fill`

- Accetta come parametri
  - un **`DataSet`** e il nome di una tabella, oppure
  - una **`DataTable`**
- Retrieves rows from the data source using the **`SELECT`** statement specified by an associated **`SelectCommand`** property
  - The connection object associated with the **`SELECT`** statement must be valid, but it does not need to be open
  - If the connection is closed before **`Fill`** is called, it is opened to retrieve data, then closed
  - If the connection is open before **`Fill`** is called, it remains open
- Then adds the rows to the specified destination **`DataTable`** object in the **`DataSet`**, creating the **`DataTable`** object if it does not already exist
  - When creating a **`DataTable`** object, the **`Fill`** operation normally creates only column name metadata. However, if the **`MissingSchemaAction`** property is set to **`AddWithKey`**, appropriate primary keys and constraints are also created

Metodo `Update`

- Accetta come parametri
  - un **`DataSet`** e il nome di una tabella, oppure
  - una **`DataTable`**
- Propaga al database le modifiche effettuate sulla **`DataTable`**
  - Invoca rispettivamente il comando **`INSERT`**, **`UPDATE`**, o **`DELETE`** per ogni riga inserita, modificata o cancellata dalla **`DataTable`**
- Per poter invocare il metodo **`Update`**, è necessario avere già creato le tre istanze di **`Command`** con **`INSERT`**, **`UPDATE`** e **`DELETE`**
  - automaticamente in design
  - mediante un **`CommandBuilder`**
  - manualmente da codice

Esempio

```
DataSet ds = new DataSet(dataSetName);
SqlConnection nwindConn = new SqlConnection(
   "Data Source=localhost; Integrated Security=SSPI;
   Initial Catalog=Northwind");
SqlDataAdapter custDA = new SqlDataAdapter(
   "SELECT * FROM Customers", nwindConn);
SqlCommandBuilder custCB = new SqlCommandBuilder(custDA);
custDA.MissingSchemaAction = MissingSchemaAction.AddWithKey;
//  Crea la tabella, aggiunge i vincoli e la riempie
custDA.Fill(ds, "Customers");
//  Modifico la tabella "Customers"
…
//  Eseguo l'update - senza custCB, l'update fallirebbe
custDA.Update(ds, "Customers");
```

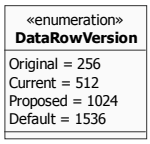
Esempio

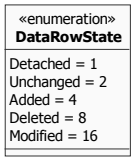
```
//  Phoenix - Metodo di configurazione di un sqlDataAdapter
//  La configurazione avviene in modo (semi-)automatico
SqlCommandBuilder commandBuilder =
   new SqlCommandBuilder(sqlDataAdapter);
SqlCommand insertCommand = commandBuilder.GetInsertCommand();
if(InsertCommandText != null)
   insertCommand.CommandText = InsertCommandText;
SqlCommand updateCommand = commandBuilder.GetUpdateCommand();
if(UpdateCommandText != null)
   updateCommand.CommandText = UpdateCommandText;
SqlCommand deleteCommand = commandBuilder.GetDeleteCommand();
if(DeleteCommandText != null)
   deleteCommand.CommandText = DeleteCommandText;
commandBuilder.DataAdapter = null;
commandBuilder.Dispose();
sqlDataAdapter.InsertCommand = insertCommand;
sqlDataAdapter.UpdateCommand = updateCommand;
sqlDataAdapter.DeleteCommand = deleteCommand;
```

8.31

Come funziona `Update`

- Di ogni riga la **`DataTable`** tiene traccia
  - sia dei valori originali delle colonne
  - sia dei valori correnti delle colonne
- Ogni **`DataRow`** tiene traccia
  - del suo stato
- Il **`DataAdapter`** è in grado di
  - individuare le righe inserite, modificate o cancellate
  - inviare gli aggiornamenti pendenti, utilizzando il comando corrispondente

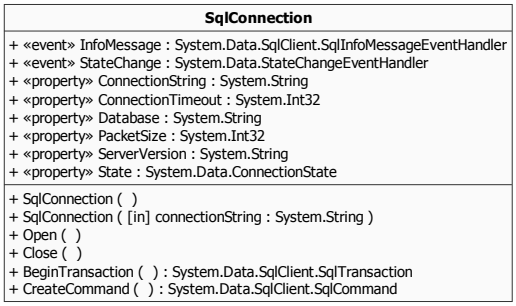




8.32

SqlConnection

- Rappresenta una connessione fisica verso SQL Server
- Viene passato a oggetti **`SqlCommand`** o **`SqlDataAdapter`**
- Permette di creare **transazioni locali**
  - Il metodo **`BeginTransaction`** restituisce un oggetto **`SqlTransaction`** che rappresenta il contesto della transazione

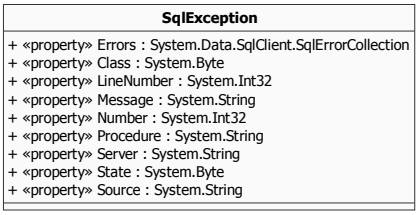


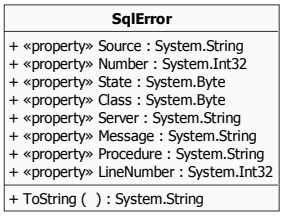
SqlConnection

- Per utilizzare la modalità “connessa” occorre aprire e chiudere la connessione
  - Implicitamente con **`SqlCommand`** e **`SqlDataReader`**
  - Esplicitamente con i metodi **`Open`**/**`Close`** (o **`Dispose`**)
- Ogni oggetto **`SqlConnection`** mantiene riferimenti a risorse ***unmanaged*** che vanno rilasciate appena possibile pertanto, in caso di apertura esplicita:
  - non basarsi sul GC
  - chiudere esplicitamente la connessione appena possibile utilizzando il metodo **`Close`** o il metodo **`Dispose`**
  - quando possibile, utilizzare la *keyword* **`using`**
- Utilizza *connection pooling*
  - per migliorare la scalabilità
  - abilitato di default, ma può essere configurato

Gestione degli errori

- Eventuali errori che avvengono durante l'esecuzione di operazioni sulla sorgente dati vengono intercettati e convertiti in eccezioni
- Ogni provider implementa le proprie eccezioni
- **`SqlException`**
  - Contiene almeno una istanza di tipo **`SqlError`**
  - **`SqlError`** restituisce le informazioni specifiche dell'errore

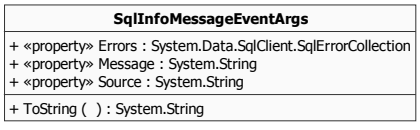




8.35

Gestione dei messaggi informativi

- Eventuali messaggi informativi generati dalla sorgente dati possono essere intercettati gestendo l'evento **`InfoMessage`**
- **`SqlInfoMessageEventHandler`**
- **`SqlInfoMessageEventArgs`**



8.36

Esempio

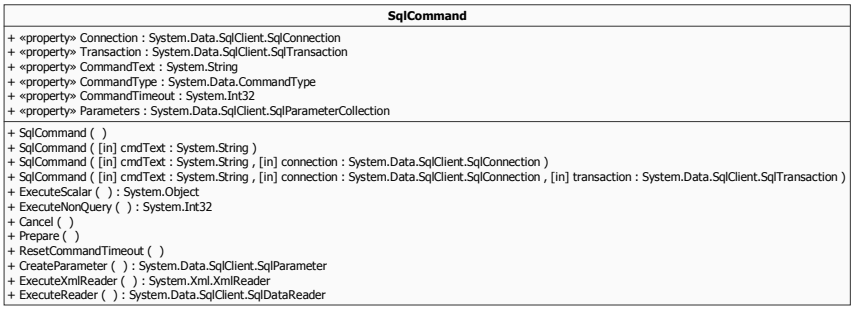
```
using (SqlConnection conn = new SqlConnection("…"))
{
  // Associo un handler all'evento InfoMessage
  conn.InfoMessage += new SqlInfoMessageEventHandler(myHandler);
  try
  {
    conn.Open();
    …
  }
  catch (SqlException e)
  {
    for (int i = 0; i < e.Errors.Count; i++)
    {
      //  Accedo alle informazioni sull'errore i-esimo
      … e.Errors[i].ToString() …
    }
  }
}
```


Esempio

```
public static void myHandler(object conn,
   SqlInfoMessageEventArgs e)
{
  for (int i = 0; i < e.Errors.Count; i++)
  {
    //  Accedo al messaggio informativo i-esimo
    … e.Errors[i].ToString() …
  }
}
```

SqlCommand

- Rappresenta un comando da eseguire su SQL Server
- Opera attraverso una connessione ad esso associata
  - in fase di creazione dell'oggetto o successivamente
- Permette di eseguire operazioni di tipo diverso
  - query ad-hoc
  - stored procedure



SqlCommand

- Sfrutta tutte le caratteristiche esposte da SQL Server
  - Es. esecuzione di query FOR XML
- Implementa l'interfaccia **`IDbCommand`**
  - **`ExecuteReader`**
    - Da utilizzare quando è previsto un result set come ritorno
  - **`ExecuteScalar`**
    - Da utilizzare per aggregazioni o risultati di calcoli
    - Restituisce la prima colonna della prima riga
  - **`ExecuteNonQuery`**
    - Ottimizzato per query che non restituiscono result set ma solo parametri di ritorno o numero di record modificati
  - **`ExecuteXmlReader`**
    - Da utilizzare insieme alla clausola FOR XML

Esempio

```
int rowsAffected;
using (SqlConnection sqlConn = new SQLConnection(myConnString))
{
  try
  {
    //  Creo il comando
    SqlCommand sqlCmd = new SqlCommand();
    //  specifico il tipo di comando
    sqlCmd.CommandType = CommandType.Text;
    sqlCmd.CommandText =
     "Insert Customers (Alias,CustomerName) Values ('myAlias','myName')";
    sqlCmd.Connection = sqlConn;
    //  apro la connessione
    sqlConn.Open();
    //  eseguo il comando, vengono restituite le righe inserite
    rowsAffected = sqlCmd.ExecuteNonQuery();
  }
  catch (SqlException e)
  {
    //  gestisco l'eccezione …
  }
}
```


`SqlCommand` e query parametriche

- Utilizzare la collezione **`Parameters`** per passare i parametri di input e di output
  - nei comandi
  - nelle chiamate a stored procedure
- I parametri sono nominali

Esempio

```
// Create a new command object
SqlCommand sqlCmd = new SqlCommand();
// Specify the stored procedure and connection
sqlCmd.CommandType = CommandType.StoredProcedure;
sqlCmd.CommandText = "InsertCustomer";
sqlCmd.Connection = sqlConn;
// Define and add a parameter to the parameters collection
SqlParam param = sqlCmd.Parameters.Add(
   new SqlParameter("@p", SqlDbType.NVarChar, 100);
param.Direction = ParameterDirection.Input;
// Set the parameter value
param.Value = "xyz";
// Add remaining parameters
   …
// Open the connection
sqlConn.Open();
// Execute the command
rowsAffected = sqlCmd.ExecuteNonQuery();
```


`SqlDataReader`

- Fornisce funzionalità di accesso *read-only* e *forward-only* su uno *stream* di record restituiti dall'esecuzione di un comando sul database
- Viene creato al momento dell'esecuzione del metodo **`ExecuteReader`** di un oggetto **`SqlCommand`**
- Il metodo **`Read`**
  - legge il record successivo nello *stream*
  - restituisce **`false`** alla fine del *result set*
- Esiste un solo record alla volta in memoria
  - Aumenta performance e scalabilità delle applicazioni se usato correttamente

SqlDataReader

- Accesso ai valori nelle singole colonne mediante
  - indice o nome del campo
  - metodi **Get** specifici per i tipi di dato contenuti
    - **GetDateTime**
    - **GetDouble**
    - **GetInt32**
    - ecc.

```
SqlDataReader myReader = myCommand.ExecuteReader();
while (myReader.Read())
{
  … myReader.GetInt32(0) … //  valore del 1° campo
  … myReader.GetString(1) … //  valore del 2° campo
}
myReader.Close();
```