ADO.NET

Introduzione all'accesso ai database mediante ADO.NET

Anno 2018/2019

Ambiente: .NET 4.0+/C# 4.0+

Indice generale

1 Applicazioni che accedono ai database....................4
1.1 Interfaccia tra applicazioni e DBMS....................4
2 Introduzione ad ADO.NET....................6
2.1 ADO.NET Provider....................6
2.1.1 ADO.NET Namespace....................7
3 Introduzione a Command, Connection e DataReader....................8
3.1 Connection....................8
3.1.1 Stringa di connessione....................8
3.1.2 Creazione di un oggetto Connection....................8
3.1.3 Apertura della connessione....................9
3.1.4 Chiusura della connessione....................9
3.2 Command....................9
3.2.1 Esecuzione di un Command....................9
3.3 DataReader....................10
3.3.1 Accesso ai dati del result set....................10
3.3.2 Accesso associativo, indicizzato e tipizzato alle colonne....................11
4 Programmare con ADO.NET....................12
4.1 Struttura delle applicazioni di accesso ai database....................12
4.2 Gestire globalmente le connessioni....................12
5 Caricamento dati....................13
5.1 Caricamento dei dati in una collezione di oggetti....................13
5.1.1 Caricamento dei corrieri....................13
5.1.2 Uso del metodo CaricaCorrieri()....................14
5.2 Istruzioni SQL parametriche....................14
5.2.1 Creazione di un Command parametrico....................14
5.2.2 Caricare gli ordini evasi dal corriere selezionato....................14
5.2.3 Uso del metodo di caricamento degli ordini....................16
5.3 Associazione dei parametri ai segnaposti corrispondenti....................16
5.3.1 Associazione posizionale (OleDb provider)....................16
5.3.2 Associazione basata sul nome (SQL Server provider)....................17
6 Modifica dei dati del database....................18
6.1 Eliminazione di un corriere....................18

6.1.1 Uso del metodo di eliminazione del corriere....19
6.2 Inserimento di un nuovo corriere....19
6.2.1 Uso del metodo di inserimento corriere....20
6.3 Recuperare il valore della colonna identità....20
6.3.1 Recupero della colonna IdCorriere....21
6.4 Aggiornamento dati....21
6.4.1 Caricamento del corriere da modificare....22
6.4.2 Uso del metodo GetCorriere()....22
6.4.3 Scrittura dei dati aggiornati....23
6.4.4 Uso del metodo ModificaCorriere()....24

1 Applicazioni che accedono ai database

I database sono il cuore di quasi tutte le attività che richiedono la gestione di informazioni: la catalogazione dei libri di una biblioteca, l'organizzazione di un sito di *e-commerce*, la gestione delle prenotazioni per i voli di una compagnia aerea, etc. I dati necessari allo svolgimento di queste attività devono essere resi disponibili agli utilizzatori: utenti, impiegati, dirigenti, etc.

In generale, non è possibile fornire agli utenti l'accesso diretto al database (attraverso un programma come SQL Management Studio, ad esempio); ci sono vari motivi:

1. Al di fuori degli amministratori nessuno dovrebbe avere accesso alla struttura interna del database.
2. Gli utilizzatori (utenti del sito, impiegati, dirigenti) non conoscono il funzionamento di un database relazionale, né conoscono il linguaggio SQL.
3. Di norma, ad ogni tipo di utilizzatore è necessario fornire una prospettiva diversa dei dati. Per l'utente di un sito musicale, il database è rappresentato da un catalogo di titoli, consultabile ma non modificabile. Un impiegato del sito dovrà avere la possibilità di aggiungere o eliminare titoli, verificare l'evasione degli ordini, etc.

In sostanza, la corretta implementazione di un database non garantisce la sua fruibilità. Tale obiettivo è realizzabile mediante applicazioni che si interpongono tra il database e gli utilizzatori. In tal senso, queste vengono definite *applicazioni client* del database, o semplicemente *client*.

1.1 Interfaccia tra applicazioni e DBMS

Nella comunicazione tra una applicazione client e il DBMS vi sono due questioni da affrontare:

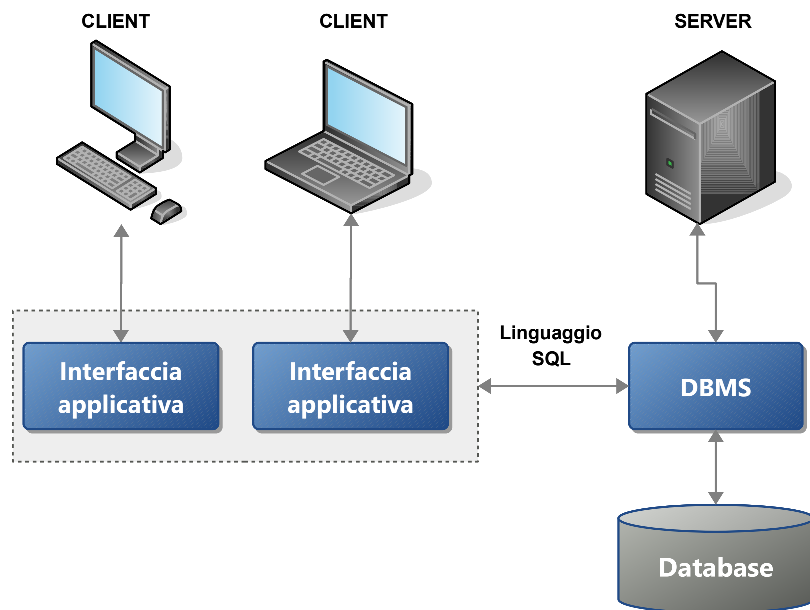
1. il modo in cui viene gestito lo scambio "fisico" dei dati tra applicazione e DBMS;
2. le funzionalità usate dall'applicazione per manipolare i dati memorizzati del database.

Il primo punto riguarda i protocolli di comunicazione utilizzati dal DBMS; qui non è importante. Il secondo è invece fondamentale per la programmazione.

Un programma è in grado di elaborare dati memorizzati in variabili semplici, vettori, collezioni, oggetti definiti dal programmatore, ecc. D'altra parte, i dati del database sono memorizzati in tabelle, secondo una struttura interna propria del DBMS, e manipolabili mediante istruzioni SQL. Ciò che serve è una ***interfaccia applicativa*** (API) tra il programma client e il DBMS che consenta di:

1. Recuperare le informazioni dal database.
2. Inviare dati al database; in generale, consentire all'applicazione di effettuare modifiche sul database.

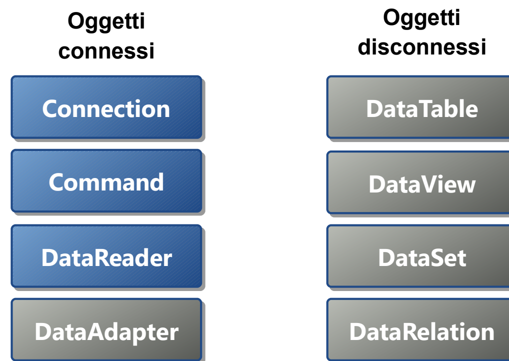
Tale interfaccia applicativa si avvarrà normalmente del linguaggio SQL, poiché è proprio questo il linguaggio più utilizzato dai DBMS.



Esistono vari tipi di interfacce applicative; ADO.NET è una di queste.

2 Introduzione ad ADO.NET

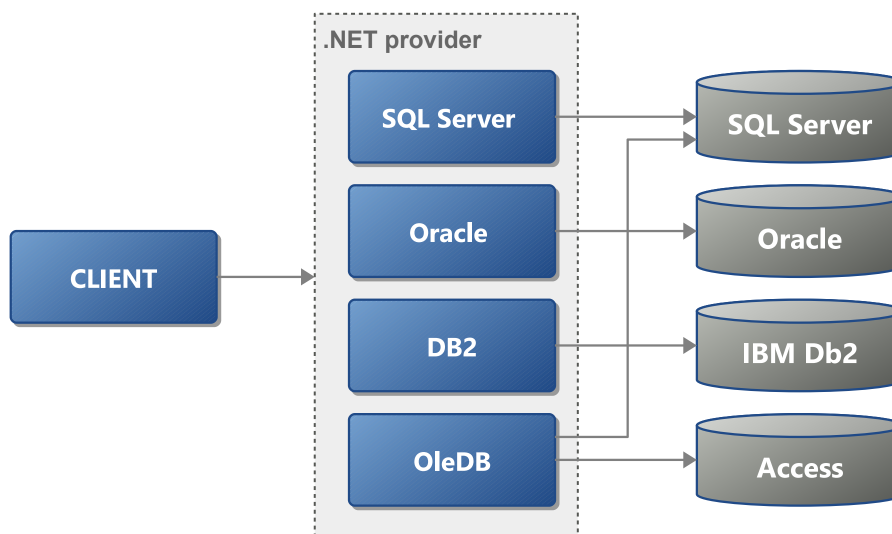
ADO.NET fornisce un insieme di oggetti che mettono l'applicazione in grado di "dialogare" con un DBMS. Gli oggetti sono stati progettati per collaborare tra loro e sono suddivisibili in due categorie: gli *oggetti connessi* e gli *oggetti disconnessi*.



I primi forniscono l'accesso al database; i secondi sono progettati per gestire i dati in memoria, indipendentemente dalla loro provenienza. Questo tutorial tratta soltanto gli oggetti *Connection*, *Command* e *DataReader*.

2.1 ADO.NET Provider

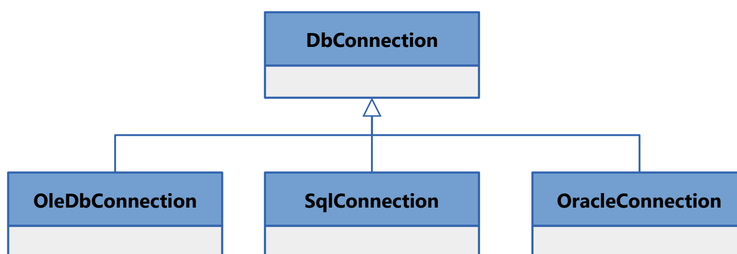
Ogni DBMS adotta un proprio modello di comunicazione con l'esterno, dunque sono necessari degli oggetti appositamente progettati per comunicare con esso: l'insieme di tali oggetti viene definito ***ADO.NET Provider***.



Ciascun provider fornisce una versione specializzata degli oggetti *Connection*, *Command* e *DataReader*. Ad esempio, l'*SQL Server provider* definisce le classi `SqlConnection`, `SqlCommand`, etc; il provider di Oracle definisce `OracleConnection`, `OracleCommand`, e via dicendo.

(Nota bene: un provider può consentire la comunicazione con più DBMS; è questo il caso dell'*OleDb Provider*.)

Le classi specializzate derivano da omologhe classi base, chiamate `DbConnection`, `DbCommand` e `DbDataReader`. Ad esempio:



Nel codice è opportuno utilizzare quanto più possibile le classi base, in modo da rendere il programma facilmente modificabile nel caso in cui fosse necessario adattarlo a diversi DBMS.

2.1.1 ADO.NET *Namespace*

Le classi di ADO.NET sono definite all'interno di vari *namespace*:

1. SQL Server: `System.Data.SqlClient`
2. Oracle: `System.Data.OracleClient`
3. OleDB: `System.Data.OleDb`

Le classi base, comuni a tutti i provider, sono definite in `System.Data.Common`.

3 Introduzione a Command, Connection e DataReader

Segue una breve introduzione agli oggetti *Connection*, *Command* e *DataReader*. Negli esempi viene utilizzato il database **Northwind.mdb** e l'*OleDb Provider*.

3.1 Connection

Un oggetto *Connection* fornisce un canale di comunicazione con il database.

3.1.1 Stringa di connessione

La proprietà fondamentale di un oggetto *Connection* è la *stringa di connessione*; questa definisce le informazioni per connettersi al database: DBMS, nome database e parametri di funzionamento della connessione.

Segue la stringa di connessione necessaria per accedere al database **Northewind.mdb**:

```
Provider=Microsoft.ACE.OLEDB.12.0; Data Source=C:\Database\Northwind.mdb
```

Nota bene: è composta da due parametri: **Provider** e **Data source**. Il primo indica il provider da utilizzare (vedi nota); il secondo definisce il nome del database, completo di percorso.

> **OleDb Provider**
>
> L'uso del provider **Microsoft.ACE.OLEDB.12.0** richiede l'installazione del file **AccessDataBaseEngine.exe**, a 32 o 64 bit in base al SO installato

3.1.2 Creazione di un oggetto Connection

Durante la creazione di un *Connection* è possibile specificare la stringa di connessione.

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

class Program
{
    const string cnStr = @"Provider=Microsoft.ACE.OLEDB.12.0;
                          Data Source=D:\Database\VentoDelNord.mdb";
    static void Main(string[] args)
    {
        DbConnection cn = new OleDbConnection(cnStr);
        ...
    }
}
```

Alternativamente, è possibile usare la proprietà `ConnectionString`:

```
DbConnection cn = new OleDbConnection();
cn.ConnectionString = cnStr;
...
```

3.1.3 Apertura della connessione

La creazione di un oggetto *Connection* non produce alcun accesso al database. Prima di eseguire qualsiasi operazione, la connessione deve essere aperta. A questo scopo esiste il metodo `Open()`:

```
OleDbConnection cn = new OleDbConnection(cnStr);
cn.Open();
...
```

Se non è in grado di stabilire una connessione con il database, `Open()` solleva un'eccezione.

3.1.4 Chiusura della connessione

Una connessione dovrebbe restare aperta soltanto lo stretto necessario per eseguire le operazioni richieste, poi deve essere chiusa mediante `Close()`.

```
DbConnection cn = new OleDbConnection(cnStr);
cn.Open();
// ... una o più operazioni sul database
cn.Close();
...
```

3.2 Command

Qualsiasi operazione sul database implica l'esecuzione di un'istruzione SQL; a questo scopo è necessario usare un *Command*. Per svolgere la propria funzione, un *Command* ha bisogno di due cose: un oggetto *Connection* e l'istruzione SQL da eseguire.

```
DbConnection cn = new OleDbConnection(cnStr);

DbCommand cmd = cn.CreateCommand();
cmd.CommandText = " SELECT Count(*) FROM Clienti WHERE Paese = 'Italia'";
...
```

La prima istruzione crea il *Command* e lo associa all'oggetto *Connection* mediante il quale viene creato. La seconda istruzione imposta l'istruzione SQL da eseguire.

3.2.1 Esecuzione di un Command

La creazione del *Command* non determina la sua esecuzione. Innanzitutto è necessario aprire la connessione, quindi occorre eseguirlo invocando uno dei tre metodi progettati a questo scopo, `ExecuteScalar()`, `ExecuteNonQuery()`, `ExecuteReader()`.

Il codice seguente mostra l'uso del metodo `ExecuteScalar()`, che restituisce un singolo valore

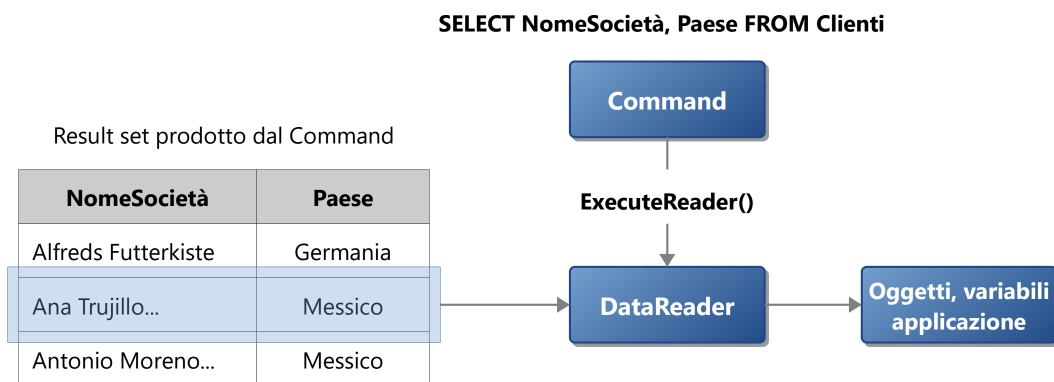
e dunque è adatto per l'esecuzione di query scalari:

```
DbConnection cn = new OleDbConnection(cnStr);

DbCommand cmd = cn.CreateCommand();
cmd.CommandText = "SELECT Count(*) FROM Clienti WHERE Paese = 'Italia'";

cn.Open();
 int numClientiItaliani = (int) cmd.ExecuteScalar();
cn.Close();
```

`ExecuteScalar()` restituisce un `object`, pertanto è necessario convertirlo al tipo appropriato.

3.3 DataReader

Il *DataReader* rappresenta un cursore ***read-only forward-only***, cioè un oggetto che consente di scorrere una riga per volta tutte le righe di un *result set* restituite da un *Command*.



I *DataReader* non devono essere creati esplicitamente, poiché sono restituiti dal metodo `ExecuteReader()`. Il metodo esegue una query sul database e associa un *DataReader* al *result set* prodotto.

```
DbConnection cn = new OleDbConnection(cnStr);

DbCommand cmd = cn.CreateCommand();
cmd.CommandText = "SELECT NomeSocietà, Paese FROM Clienti";

cn.Open();
 DbDataReader dr = cmd.ExecuteReader();
 // ... qui viene usato il DdataReader
```

3.3.1 Accesso ai dati del result set

L'accesso ai dati si ottiene facendo "avanzare" il *DataReader* una riga per volta e indicizzando le colonne. Il metodo `Read()` fa avanzare il *DataReader* alla riga successiva, restituendo *false* quando non ci sono più righe da leggere.

Nell'esempio seguente, da ogni riga viene estratto il campo **NomeSocietà** e memorizzato in una lista di stringhe:

```
...
cn.Open();
DbDataReader dr = cmd.ExecuteReader();
List<string> clienti = new List<string>();

while (dr.Read() == true)
{
    string nomeSocietà = dr["NomeSocietà"].ToString();
    clienti.Add(nomeSocietà);
}
cn.Close();
```

Nota bene:

- Il ciclo termina quando non ci sono più righe da leggere (`Read()` ritorna false).
- L'accesso alla colonna viene eseguito specificandone il nome.
- Al termine della lettura viene chiusa la connessione.

3.3.2 Accesso associativo, indicizzato e tipizzato alle colonne

Il *DataReader* fornisce diversi meccanismi di accesso alle colonne di una riga. Quello usato nell'esempio è di tipo ***associativo non-tipizzato***. Il *DataReader* funziona come un dizionario: il nome della colonna viene usato come chiave per accedere al suo valore, restituito come tipo `object`.

Un secondo meccanismo è quello ***indicizzato non-tipizzato***:, che prevede l'uso di un indice, come se il *DataReader* fosse un vettore o una lista:

```
while (dr.Read() == true)
{
    string nomeSocietà = dr[0].ToString();
    clienti.Add(nomeSocietà);
}
```

(Nota bene: l'ordine delle colonne è quello specificato nella clausola SELECT della query.)

Infine, esiste l'accesso ***indicizzato tipizzato***, con il quale il valore della colonna viene ottenuto mediante un metodo che ritorna un valore del tipo appropriato:

```
while (dr.Read() == true)
{
    string nomeSocietà = dr.GetString(0);
    clienti.Add(nomeSocietà);
}
```

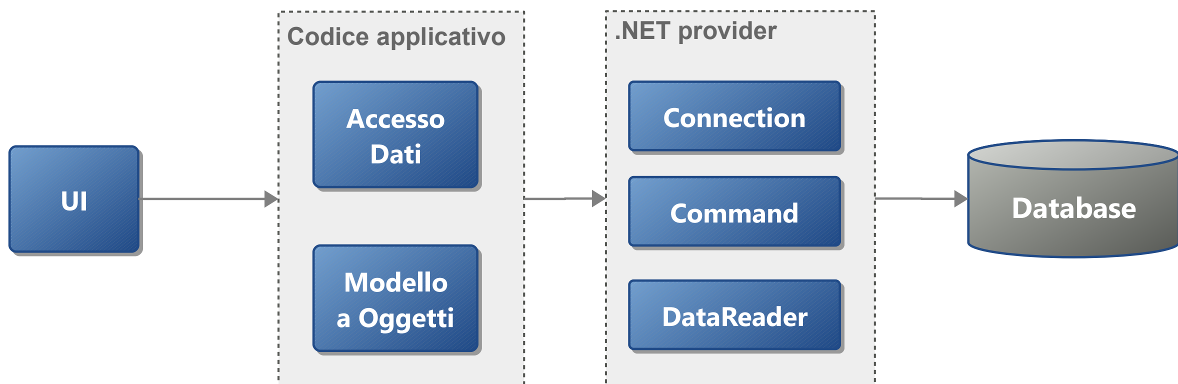
Qualunque sia il metodo utilizzato, se il nome o l'indice fanno riferimento a una colonna inesistente, viene sollevata l'eccezione di `IndexOutOfRange`.

4 Programmare con ADO.NET

Prenderò in considerazione alcuni scenari allo scopo di mostrare in azione le funzionalità degli oggetti precedentemente introdotti. Utilizzerò il più possibile gli oggetti connessi comuni, in modo da rendere il codice facilmente adattabile a DBMS diversi.

4.1 Struttura delle applicazioni di accesso ai database

Normalmente, una tipica applicazione di accesso ai database è strutturata secondo lo schema mostrato in figura.



Il codice di accesso dal database svolge due compiti:

1. Recupera i dati e li carica in memoria, in strutture dati predefinite (array, `List<>`, etc), o progettate dal programmatore. (Modello a oggetti.)
2. Esegue le modifiche sul database.

Gli esempi che seguiranno rispetteranno questo schema. Per semplicità, implementerò l'accesso dati mediante moduli: `GestioneCorrieri` e `GestioneOrdini`.

4.2 Gestire globalmente le connessioni

Le connessioni saranno gestite mediante una classe *helper*.

```
public static class Database
{
    public static string cnStr = "...";
    public static DbConnection CreaConnessione()
    {
        return new OleDbConnection(cnStr);
    }
}
```

Questa soluzione semplifica il codice e rende agevole modificare la stringa di connessione e il *.NET Provider* utilizzato.

5 Caricamento dati

È lo scenario più comune: viene eseguita una query che restituisce un *result set*, accessibile mediante un *DataReader*.

5.1 Caricamento dei dati in una collezione di oggetti

Si costruisce un modello a oggetti che contenga i dati prelevati da una o più tabelle del database.

5.1.1 Caricamento dei corrieri

Il primo passo è definire la classe `Corriere`. Questa corrisponde all'entità **Corriere** del modello E/R e definisce tanti attributi quante sono le colonne della tabella **Corrieri** che si desidera gestire.

```
public class Corriere
{
    public int IdCorriere { get; set; }
    public string NomeSocietà { get; set; }
    public string Telefono { get; set; }
}
```

Nel modulo `GestioneCorrrieri` implemento un metodo che interroga il database e produce una lista dei corrieri:

```
public class GestioneCorrieri
{
    public static List<Corriere> CaricaCorrieri()
    {
        List<Corriere> corrieri = new List<Corriere>();
        DbConnection cn = Database.CreaConnessione();
        DbCommand cmd = cn.CreateCommand();
        cmd.CommandText = "SELECT idCorriere, NomeSocietà, Telefono FROM Corrieri";
        cn.Open();
        DbDataReader dr = cmd.ExecuteReader();
        while (dr.Read() == true)
        {
            Corriere c = new Corriere();
            c.IdCorriere = dr.GetInt32(0);
            c.NomeSocietà = dr.GetString(1);
            c.Telefono = dr.GetString(2);
            corrieri.Add(c);
        }
        cn.Close();
        return corrieri;
    }
}
```

5.1.2 Uso del metodo CaricaCorrieri()

A titolo di esempio, mostro come usare il metodo per ottenere i corrieri e visualizzarli in un *listbox*:

```
List<Corriere> corrieri = GestioneCorrieri.CaricaCorrieri();
lstCorrieri.DisplayMember = "NomeSocietà";
lstCorrieri.DataSource = corrieri;
```

5.2 Istruzioni SQL parametriche

In molti casi, l'istruzione SQL da eseguire dipende dall'input dell'utente. Si parla in questo caso di ***istruzioni SQL parametriche***. Queste contengono uno o più segnaposti, i quali devono essere valorizzati prima che l'istruzione venga eseguita.

Ecco un esempio di query parametrica, per ottenere l'elenco dei clienti appartenenti ad una determinata nazione:

```
SELECT NomeSocietà FROM Clienti
WHERE Paese = @Paese
```

@Paese funge da variabile; il suo valore deve essere specificato prima di eseguire l'istruzione. A questo scopo è necessario l'uso di un ***Command parametrico***.

> **Istruzioni SQL dinamiche**
>
> Essendo una stringa, un'istruzione SQL con parametri può essere costruita anche concatenando le parti fisse con quelle che dipendono dall'input dell'utente.

5.2.1 Creazione di un Command parametrico

Un *Command* parametrico memorizza una collezione di parametri, utilizzati per valorizzare i segnaposti di un'istruzione SQL parametrica. Il tipo dei parametri dipende dal provider utilizzato (`OleDbParameter`, `SqlParameter`, etc); ad ogni modo, negli scenari più semplici è sufficiente il tipo base `DbParameter`.

La creazione di un *Command* parametrico segue la seguente scaletta:

```
// crea il command
// imposta l'istruzione SQL
  // crea il parametro
  // imposta proprietà parametro: nome, tipo, valore...
  // aggiungi parametro alla collezione del command
// ripeti le istruzioni evidenziate per ogni parametro
```

5.2.2 Caricare gli ordini evasi dal corriere selezionato

Supponiamo di volere ottenere gli ordini evasi da un determinato corriere; di ogni ordine interessa soltanto la data e la destinazione.

Definisco innanzitutto la classe `Ordine`, che mappa la tabella **Ordini** del database, specificando però soltanto i campi richiesti:

```
public class Ordine
{
    public DateTime DataOrdine { get; set; }
    public string Destinatario { get; set; }
}
```

La query da eseguire è la seguente:

```
SELECT DataOrdine, Destinatario FROM Ordini
WHERE Corriere = @IdCorriere
```

Nel metodo `OrdiniByCorriere()` utilizzo un Command parametrico; il parametro sarà valorizzato con l'id del corriere:

```
public class GestioneOrdini
{
    public static List<Ordine> OrdiniByCorriere(int idCorriere)
    {
        string sql = " SELECT DataOrdine, Destinatario FROM Ordini" +
                     " WHERE Corriere = @IdCorriere";

        DbConnection cn = Database.CreaConnessione();
        DbCommand cmd = cn.CreateCommand();
        cmd.CommandText = sql;

        DbParameter paId = cmd.CreateParameter();
        paId.ParameterName = "@IdCorriere";
        paId.DbType = DbType.Int32;
        paId.Value = idCorriere;
        cmd.Parameters.Add(paId);

        List<Ordine> ordini = new List<Ordine>();
        cn.Open();
        DbDataReader dr = cmd.ExecuteReader();
        while (dr.Read() == true)
        {
            Ordine o = new Ordine();
            o.DataOrdine= dr.GetDateTime(0);
            o.Destinatario= dr.GetString(1);
            ordini.Add(o);
        }
        cn.Close();
        return ordini;
    }
}
```

Alcune considerazioni:

1. L'uso del tipo `DbParameter` richiede l'impostazione delle singole proprietà. Gli oggetti specializzati (`SqlParameter`, `OldDbParameter`, etc) possono essere impostati già durante la loro creazione;

2. In generale, non tutte le proprietà devono essere valorizzate; dipende dal provider utilizzato. Nel caso specifico sarebbe stato sufficiente impostare la proprietà `Value`.

5.2.3 Uso del metodo di caricamento degli ordini

Si presuppone che l'utente, selezionando un corriere nel *listbox*, possa visualizzare gli ordini in un *datagridview* cliccando su un apposito bottone.

```
private void btnOrdini_Click(object sender, EventArgs e)
{
    Corriere c = lstCorrieri.SelectedItem as Corriere;
    if (c == null)
        return;

    List<Ordine> ordini = GestioneOrdini.OrdiniByCorriere(c.IdCorriere);
    dgwOrdini.DataSource = ordini;
}
```

5.3 Associazione dei parametri ai segnaposti corrispondenti

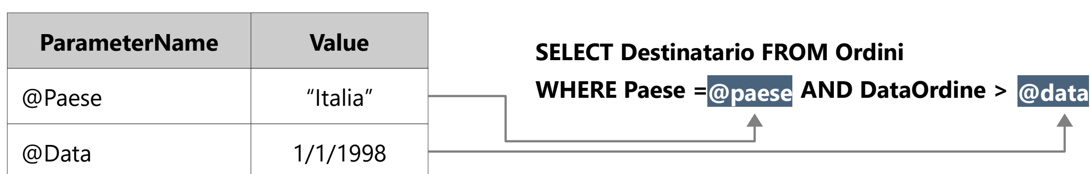
L'associazione dei parametri del *Command* ai segnaposti dell'istruzione SQL può essere di due tipi: ***posizionale*** o ***basato sul nome***. Dipende dal *.NET provider* utilizzato.

5.3.1 Associazione posizionale (OleDb provider)

Viene considerata solo la posizione, dei segnaposti nell'istruzione SQL, dei parametri nel *Command* parametrico. In entrambi i casi, i nomi sono ignorati. Dunque, il primo parametro viene associato al primo segnaposto, il secondo parametro al secondo segnaposto, e così via.

Di seguito vengono forniti due esempi. Nel primo la corrispondenza è corretta.

Command parametrico



ParameterName	Value
@Paese	"Italia"
@Data	1/1/1998

Nel secondo esempio, l'ordine dei parametri nel *Command* è invertito; dunque una data viene associata a un segnaposto stringa e viceversa:

Command parametrico

ParameterName	Value
@Data	1/1/1998
@Paese	"Italia"

Un simile errore non viene rilevato e può produrre risultati imprevedibili.

> **Nomi dei parametri e segnaposti**
>
> Poiché i nomi dei parametri e dei segnaposti vengono ignorato, qualunque nome è corretto. Uno standard utilizzato prevede l'uso del simbolo '?' come segnaposto nelle istruzioni SQL.

5.3.2 Associazione basata sul nome (SQL Server provider)

È il criterio usato dai provider più evoluti, ed è basato unicamente sulla corrispondenza tra i nomi dei parametri e quelli dei segnaposti. Prendendo come riferimento questo criterio, entrambi gli esempi precedenti sono corretti.

6 Modifica dei dati del database

La modifica del database viene effettuata mediante l'esecuzione di istruzioni INSERT, UPDATE o DELETE. Al riguardo, occorre garantire che l'applicazione abbia i privilegi necessari per eseguire le operazioni richieste. Relativamente ai database basati su file (come Access e SQLite) è necessario che l'applicazione abbia l'autorizzazione di scrittura sul file contenente il database.

Per queste istruzioni occorre usare il metodo `ExecuteNonQuery()`. Questo restituisce un valore intero che indica il numero di righe coinvolte, valore che può essere utilizzato per avere una conferma sulla corretta esecuzione dell'operazione.

6.1 Eliminazione di un corriere

Si supponga di voler implementare l'eliminazione di un corriere selezionato dall'utente. Occorre eseguire la seguente istruzione SQL:

```
DELETE FROM Corrieri
WHERE IdCorriere = @IdCorriere
```

Il metodo `EliminaCorriere()` riceve come argomento l'id del corriere da eliminare.

```
public static bool EliminaCorriere(int idCorriere)
{
    string sql = "DELETE FROM Corrieri WHERE IdCorriere = @IdCorriere";
    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    DbParameter paId = cmd.CreateParameter();
    paId.ParameterName = "@IdCorriere";
    paId.DbType = DbType.Int32;
    paId.Value = idCorriere;

    cmd.Parameters.Add(paId);
    cmd.CommandText = sql;
    cn.Open();
    int numRighe = cmd.ExecuteNonQuery();
    cn.Close();
    return numRighe == 1;
}
```

Verificando che il comando abbia effettivamente eliminato una riga, il metodo ritorna *true* se il corriere è stato eliminato, *false* in caso contrario.

Errore nell'esecuzione di istruzione SQL

In questo e nei precedenti esempi non ho preso in considerazione la possibilità che l'operazione possa fallire con un errore. Si tratta di una questione molto importante, che negli scenari realistici non può essere ignorata, ma che qui non considero per mantenere gli esempi di facile comprensione.

Nel caso in oggetto, l'operazione è destinata a fallire se si tenta di eliminare un corriere al quale sono collegati degli ordini. Nel database è presente un vincolo di integrità referenziale tra **Corrieri** e **Ordini** che impedisce questo tipo di modifica. Un programma realistico dovrebbe intercettare questo errore, informando l'utente.

6.1.1 Uso del metodo di eliminazione del corriere

Si presuppone che l'utente selezioni un corriere nel *listbox*:

```
private void btnEliminaCorriere_Click(object sender, EventArgs e)
{
    Corriere c = lstCorrieri.SelectedItem as Corriere;
    if (c == null)
        return;

    GestioneCorrieri.EliminaCorriere(c.IdCorriere);
    MessageBox.Show("Il corriere è stato eliminato");
}
```

6.2 Inserimento di un nuovo corriere

Occorre eseguire la seguente istruzione SQL:

```
INSERT INTO Corrieri(NomeSocietà, Telefono)
VALUES (@NomeSocietà, @Telefono)
```

Nota bene: non ho specificato la colonna **IdContatore**, poiché si tratta di una *colonna identità*, il cui valore viene automaticamente generato dal DBMS.

L'implementazione dell'operazione è realizzata dal metodo `InserisciCorriere()`, il quale riceve il corriere da inserire:

```
public static void InserisciCorriere(Corriere c)
{
    string sql = " INSERT INTO Corrieri(NomeSocietà, Telefono)" +
                 " VALUES (@NomeSocietà, @Telefono)";
    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    DbParameter paNome = cmd.CreateParameter();
    paNome.ParameterName = "@NomeSocietà";
    paNome.DbType = DbType.String;
    paNome.Value = c.NomeSocietà;
    cmd.Parameters.Add(paNome);

```

```
    DbParameter paTel = cmd.CreateParameter();
    paTel.ParameterName = "@Telefono";
    paTel.DbType = DbType.String;
    paTel.Value = c.Telefono;
    cmd.Parameters.Add(paTel);

    cmd.CommandText = sql;
    cn.Open();
    cmd.ExecuteNonQuery();
    cn.Close();
}
```

6.2.1 Uso del metodo di inserimento corriere

Si presuppone che l'utente inserisca nome e telefono mediante dei *textbox*:

```
private void btnInserisciCorriere_Click(object sender, EventArgs e)
{
    Corriere c = new Corriere();
    c.NomeSocietà = txtNomeSocietà.Text
    c.Telefono = txtTelefono.Text
    GestioneCorrieri.InserisciCorriere(c);
    MessageBox.Show("Il corriere è stato inserito");
}
```

6.3 Recuperare il valore della colonna identità

Esistono scenari nei quali si desidera utilizzare immediatamente l'elemento appena inserito per future operazioni. Ad esempio, nel caso precedente si può immaginare di voler aggiungere il nuovo corriere al *listbox* senza dover ricaricare l'intero elenco dal database.

In questi casi sorge il problema di recuperare il valore delle colonne identità, poiché questo è generato automaticamente dal DBMS e dunque sconosciuto all'applicazione.

Ogni DBMS usa delle tecniche specifiche sia per gestire le colonne identità sia per recuperarne i valori. Con SQL Server e Access è possibile farlo eseguendo la query[1]:

```
SELECT @@identity
```

Poiché, il recupero dell'ultimo valore identità generato rappresenta un'operazione di utilità generale, è opportuno implementarla nella classe *helper* `Database`.

```
public static int GetIdentity(DbConnection cn)
{
    DbCommand cmd = cn.CreateCommand();
    cmd.CommandText = "SELECT @@identity";
    bool cnAperta = cn.State == ConnectionState.Open;

    if (cnAperta == false)
    {
```

1 In realtà con SQL Server questa tecnica ha qualche limitazione, che però può essere trascurata negli scenari che stiamo considerando.

```
        cn.Open();
    }

    int identity = (int)cmd.ExecuteScalar();

    if (cnAperta == false)
    {
        cn.Close();
    }
    return identity;
}
```

Alcune osservazioni:

1. Il metodo riceve l'oggetto *Connection* da utilizzare per eseguire il comando. È una scelta opportuna, dato che di solito si desidera recuperare il valore *identity* utilizzando la stessa connessione usata dall'istruzione INSERT.
2. Mediante la proprietà `State` il metodo verifica se la connessione è già aperta; in caso contrario l'apre. Dopo aver eseguito la query, il metodo chiude la connessione, ma soltanto se è stata aperta in precedenza. In caso contrario, lascia la responsabilità al metodo chiamante.

6.3.1 Recupero della colonna IdCorriere

Occorre modificare il metodo `InserisciCorriere()`:

```
public static void InserisciCorriere(Corriere c)
{
    string sql = " INSERT INTO Corrieri(NomeSocietà, Telefono)" +
                 " VALUES (@NomeSocietà, @Telefono)";

    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    // ... creazione del Command parametrico

    cn.Open();
    cmd.ExecuteNonQuery();
    c.IdCorriere = Database.GetIdentity(cn);
    cn.Close();
}
```

6.4 Aggiornamento dati

In molte applicazioni, l'aggiornamento è un'operazione che si volge in due fasi. Nella prima, l'utente seleziona l'entità da modificare; l'applicazione risponde visualizzando i dati attuali e consentendo all'utente la loro modifica. Infine, i dati aggiornati vengono inviati al database, sovrascrivendo quelli originali.

A dirigere questo scambio tra applicazione e database è la chiave primaria dell'elemento da modificare.

6.4.1 Caricamento del corriere da modificare

Occorre eseguire la seguente query:

```
SELECT NomeSocietà, Telefono FROM Corrieri
WHERE IdCorriere = @IdCorriere
```

L'implementazione dell'operazione viene realizzata dal metodo `GetCorriere()`. Questo richiede come argomento l'id del corriere da caricare.

```
public static Corriere GetCorriere(int idCorriere)
{
    string sql = " SELECT NomeSocietà, Telefono FROM Corrieri" +
                 " WHERE IdCorriere = @IdCorriere";

    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    DbParameter paId = cmd.CreateParameter();
    paId.ParameterName = "@IdCorriere";
    paId.DbType = DbType.Int32;
    paId.Value = idCorriere;

    cmd.Parameters.Add(paId);
    cmd.CommandText = sql;
    cn.Open();
    DbDataReader dr = cmd.ExecuteReader();

    Corriere c = null;
    if (dr.Read() == true)
    {
        c = new Corriere();
        c.IdCorriere = idCorriere;
        c.NomeSocietà = dr.GetString(0);
        c.Telefono = dr.GetString(1);
    }
    cn.Close();
    return c;
}
```

Nota bene: il *result set* contiene una o nessuna riga, condizione che viene verificata nella `if()`. Alternativamente, è possibile testare la proprietà `HasRows`, che indica se il *result set* contiene almeno una riga.

6.4.2 Uso del metodo GetCorriere()

Mi limito a mostrare l'invocazione del metodo.

```
// ...qui viene selezionato l'id del corriere
 Corriere c = GestioneCorrieri(idCorriere);
txtNomeSocietà.Text = c.NomeSocietà;
txtTelefono.Text = c.Telefono;
...
```

6.4.3 *Scrittura dei dati aggiornati*

Occorre eseguire la seguente istruzione SQL:

```
UPDATE Corrieri SET
    NomeSocietà = @NomeSocietà,
    Telefono    = @Telefono
WHERE IdCorriere = @IdCorriere
```

L'operazione è realizzata dal metodo `ModificaCorriere()`, che riceve come argomento il corriere da modificare.

```
public static bool ModificaCorriere(Corriere c)
{
    string sql = " UPDATE Corrieri SET" +
                   " NomeSocietà = @NomeSocietà, "+
                   " Telefono = @Telefono "+
                 " WHERE IdCorriere = @IdCorriere";

    DbConnection cn = Database.CreaConnessione();
    DbCommand cmd = cn.CreateCommand();

    DbParameter paNome = cmd.CreateParameter();
    paNome.ParameterName = "@NomeSocietà";
    paNome.DbType = DbType.String;
    paNome.Value = c.NomeSocietà;
    cmd.Parameters.Add(paNome);

    DbParameter paTel = cmd.CreateParameter();
    paTel.ParameterName = "@Telefono";
    paTel.DbType = DbType.String;
    paTel.Value = c.Telefono;
    cmd.Parameters.Add(paTel);

    DbParameter paId = cmd.CreateParameter();
    paId.ParameterName = "@IdCorriere";
    paId.DbType = DbType.Int32;
    paId.Value = c.IdCorriere;
    cmd.Parameters.Add(paId);

    cmd.CommandText = sql;
    cn.Open();
    int numRighe = cmd.ExecuteNonQuery();
    cn.Close();

    return numRighe == 1;
}
```

6.4.4 Uso del metodo ModificaCorriere()

Si può supporre di eseguire la modifica in risposta all'azione di conferma da parte dell'utente, utilizzando lo stesso oggetto corriere precedentemente caricato dal database. Qui mi limito a mostrare l'invocazione del metodo.

```
 // ...la variabile 'c' contiene i dati originali, compreso l'id
c.Nomesocietà = txtNomeSocietà.Text;
c.Telefono = txtTelefono.Text;
 GestioneCorrieri.ModificaCorriere(c);
MessageBox.Show("Il corriere è stato modificato");
...
```