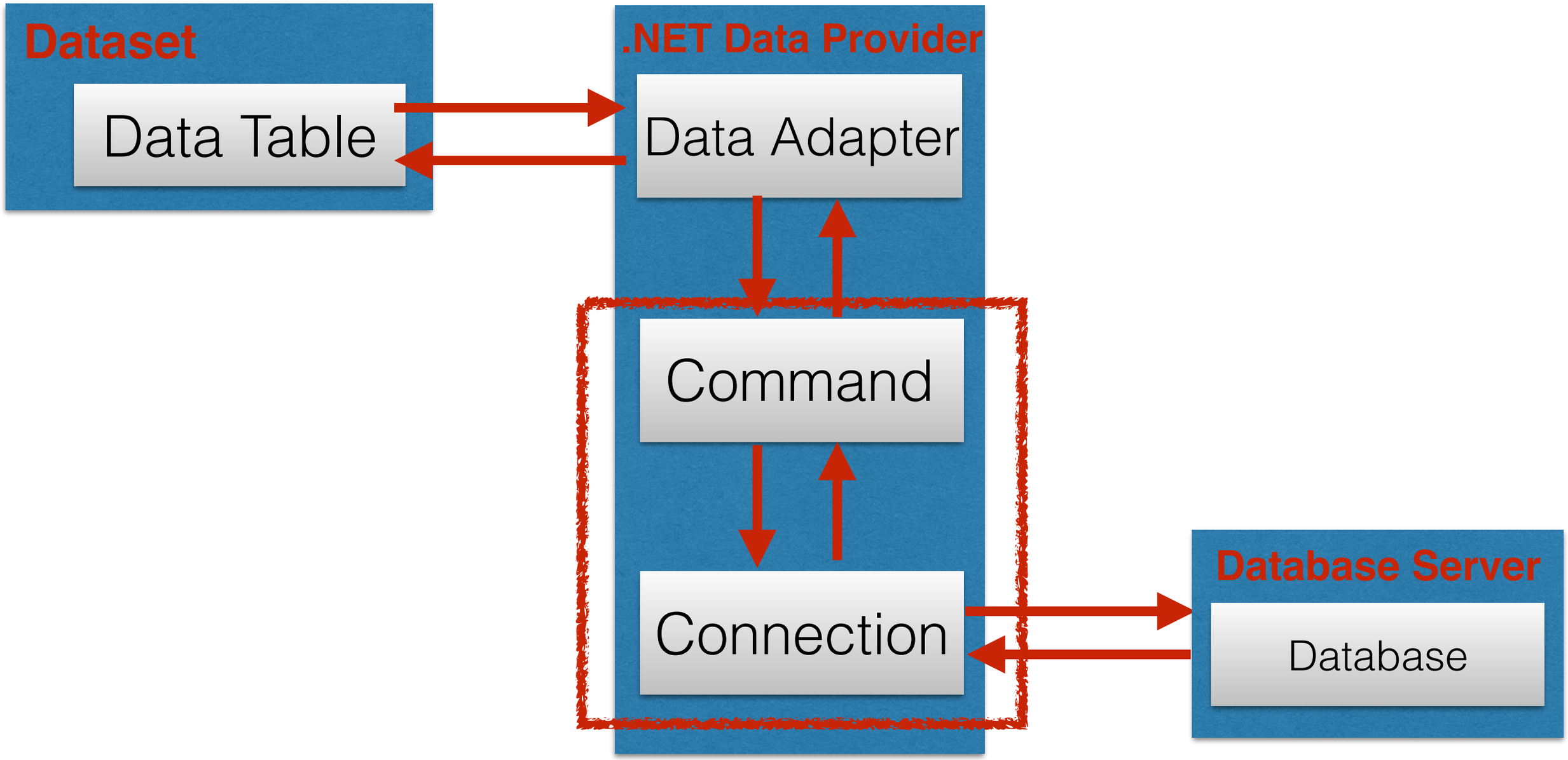
ADO.NET
Command Class

ADO.NET Objects Architecture



The SqlCommand Class

After an established connection, for executing a SQL statement on a SQL server, one has to create an **SqlCommand** object which will contain the statement.
With a connection to MySql, we use the **MySqlCommand** object.

The MySqlCommand is build from the MySqlConnection object and the command string:

```
MySqlCommand cmd = new MySqlCommand(cmdstring,conn);
```

Alternatively, you can assign connection and command with the associated methods:

```
MySqlCommand cmd = new MySqlCommand();
cmd.Connection = conn;
string cmdstring = “SELECT Customers FROM Table ORDER BY Name”;
cmd.CommandText = cmdstring;
```

SqlCommand Class: Properties and Methods

Property	Description
`Connection`	Connection object
`CommandText`	SQL statement string
`CommandType`	States how to interpret the command
`Parameters`	Parameters of the command

Method	Description
`ExecuteReader()`	Executes the query and returns the result as an SqlDataReader object.
`ExecuteScalar()`	Executes a query and returns the first column of the first row of the resulting data.
`ExecuteNonQuery()`	Executes a query and returns an integer indicating the numbers of rows affected.

Property	Description
`Text`	Default: SQL statement
`StoredProcedure`	stored procedure
`TableDirect`	name of a table (OLE database only)

Handling answers to a SQL query

Answers to many SQL commands are e.g. database tables: this is for sure the case of a SELECT statement.
Data is returned in the form of a **Data Reader** object.

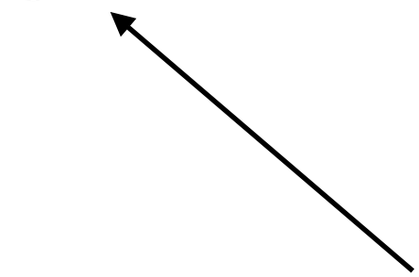
```
MySqlDataReader reader =
            cmd.ExecuteReader(CommandBehavior.CloseConnection);
```

Reader Methods:
Close()
NextResult()
Read()

Command Behavior enumeration members:
- CloseConnection
- Default
- SingleResult
- SingleRow

Simple DataReader Example

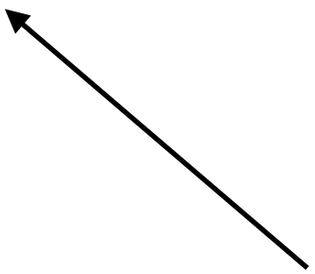
```
……
connection.Open();

……

MySqlDataReader reader =
            cmd.ExecuteReader(CommandBehavior.CloseConnection);

while (reader.Read()){

  Console.WriteLine(reader.GetString(0));

}

reader.Close();
```



Specifies the column position

Another DataReader Example: save data into an object

```
……
connection.Open();

……

MySqlDataReader reader =
            cmd.ExecuteReader(CommandBehavior.CloseConnection);

List<Person> personslist = new List<Person>();

while (reader.Read()){ //Read returns a bool
  Person p = new Person();
  p.FirstName = reader[“FirstName”].ToString();
  p.LastName  = reader[“LastName”].ToString();
  personslist.Add(p);
}

reader.Close();
```



Column name in the DB

DataReader Summary

- Before calling the data reader, the connection must be opened.

- The data reader is opened automatically when created. While the data reader is open, no other data readers can be opened on the same connection (although Oracle is an exception in this regard).

- When the reader is created, it is positioned BEFORE the first row of the result. To read the first row, you have to execute Read().

- It is possible to execute more SELECT statements (separated by a semicolon in the command string) and feed all the results into the same reader. You can then Read() them and for jumping to the next SELECT result you invoke the NextResult() method

EXERCISE: Try a multiple SELECT query + NextResult()

Data Reader Efficiency

In general, looking up columns by position number is faster than by name. This procedure is prone to errors and moreover the column positions might change if the query changes.

To practically solve this and other issues connected to column lookup, you can use the **column methods**:

Method	Description
`GetOrdinal(name)`	Gets the position of the “name” column (starts form 0)
`GetBoolean(position)`	Gets the value of the column at “position” as Boolean
`GetDateTime(position)`	“"
`GetDecimal(position)`	“"
`GetInt16(position)`	“"
`GetInt32(position)`	“"
`GetInt64(position)`	“"
`GetString(position)`	“"

Example

```
string s = “SELECT * FROM T WHERE Id>10;”;
MySqlCommand cmd = new MySqlCommand(s,connection);
connection.Open();

MySqlDataReader r = cmd.ExecuteReader();

int id_ord = r.GetOrdinal(“ID”);
int name_ord = r.GetOrdinal(“Name”);

List l = new List<Person>();
while (r.Read()){
    Person p = new Person();
    p.ID = r.GetInt16(id_ord);
    p.FirstName = r.GetString(name_ord);
    list.Add();
}

r.Close();
connection.Close();
```

Class Design for Storing Results: Example

```
public class Person
{
    private int ID;
    private string FirstName;
    private string FamilyName;

    public Person();

    public int ID {
        get {return ID;}
        set {ID = value;}
    }
    public string FirstName {
        get {return FirstName;}
        set {FirstName = value;}
    }
    public string FamilyName {
        get {return FamilyName;}
        set {FamilyName = value;}
    }
}
```

Queries with No Data Return

Queries which do not return data are called **action queries**.
Action queries are for example:

- Update
- Delete
- Insert

statements.

The command to use is:

`ExecuteNonQuery`

which returns an integer indicating the number of rows affected by the command.

Action Query Example

```
MySqlCommand del = new MySqlCommand();
del.Connection = connection;
del.CommandText = “DELETE FROM Person” +
                  “WHERE ID>100”;
try {
   connection.Open();
   int rows = del.ExecuteNonQuery();
   MessageBox.Show(rows + “ rows deleted”);
}
```

```
catch (MySqlException e)
{
   MessageBox.Show(“Connection Error #”
            + e.Number + “ ” + e.Message);
}

finally
{
   connection.Close();
}
```

Queries returning Scalar Values

Some queries return just a single value which is sometimes called **aggregate value**.
Such a value can be for example:

- a value extracted from a single column
- a calculated value (SUM, AVERAGE, COUNT, …)
- any other number which can be obtained from a DB.

The appropriate command in such cases is:

`ExecuteScalar`

EXAMPLE:

```
cmd.CommandText = "SELECT AVERAGE(gpa) " +
                  "AS AverageGPA FROM Students";

try{
    …
    float avgGPA = (float)cmd.ExecuteScalar();
    …
}
…
```

Parametric Queries in (My)SQL

A parameter is a variable used in a SQL statement. Parameters allow the creation of statements able to retrieve or change data based on a variable's value.
Queries with parameters are usually called parameterized queries (or prepared statements.).

- They are useful for decoupling the query string from the query values.
- Improve security (SQL injection problem)
- Improve speed (query caching).

EXAMPLE:

```
set @id:=123;
SELECT * FROM Students u WHERE u.ID = @id;
```

Create Parameters

After writing a parameterized query, you need to define the parameters.
This is done in the following way:

```
MySqlParameter par = new MySqlParameter();
par.ParameterName = “@id”;
par.Value = someID;
```

or:

```
MySqlParameter par = new MySqlParameter(“@id”,someID);
```

Create a Parametric Query

```
//Create the parameter
MySqlParameter par = new MySqlParameter();
par.ParameterName = “@id”;
par.Value = someID;

//Create the Command
MySqlCommand cmd = new MySqlCommand();
cmd.Connection = connection;

//Add the parameter
cmd.Parameters.Add(“@Student_ID”);

//Add a parameter with value
cmd.Parameters.AddWithValue(“@Par_Name”,Par_Value);

//Change an existing parameter value
cmd.Parameters[“@Student_ID”].Value = some_id;
```

OUTPUT Parameters

Output parameters can store data from the DB and sometimes this is preferable than using a DataReader.
Output parameters are found in the SELECT clause.

EXAMPLE:

```
set @Name;
SELECT Name=@Name FROM Students u WHERE u.ID = 10;
```

Such query is realized in this way:

```
MySqlParameter par = new MySqlParameter(
                          "@Name",
                          MySqlDbType.VarChar,   //var type
                          100);                  //var size

par.Direction = ParameterDirection.Output;
```

We are going to use this in the context of stored procedures.

Using Multiple Parameters (NOTE the MODULARITY)

```
public static bool UpdateStudent(Student astudent)
{
    MySqlConnection conn = StudentsDB.GetConnection();
    string query = “UPDATE Students SET
                    ID = @ID
                    Name = @Name,
                    Surname = @Surname
                    ……”;

    MySqlCommand cmd = new MySqlCommand(query,conn);
    cmd.Parameters.AddWithValue(“@ID”,astudent.ID);
    cmd.Parameters.AddWithValue(“@Name”,astudent.Name);
    cmd.Parameters.AddWithValue(“@Surname”,astudent.Surname);

    try {
           conn.Open()
           int rows = cmd.ExecuteNonQuery();
           if (rows>0) return true; else return false;
       }
    catch (MySqlException e){ throw e; }
    finally {conn.Close();}

}
```

Stored Procedures

Stored Procedures are DB objects (stored on the DB server, not on the client) which contains one or more SQL statements (see previous slides).
The compile-and-optimize phase is run only the first time, therefore the execution of a SP is faster than a normal query.

Creating Stored Procedures will use some machinery we have seen in parametric queries.

EXAMPLE (MySQL):

```
mysql> delimiter //

mysql> CREATE PROCEDURE GetStudent(p1 INT)
    -> BEGIN
    ->   SET @x = p1;
    ->   SELECT p1,Name FROM Students
    -> END
    -> //

mysql> delimiter ;
mysql> CALL GetStudent(1000);
```

Stored Procedures

```
try
{
    conn.Open();

    string str = “some_name";
    MySqlCommand cmd = new MySqlCommand(str, conn);
    cmd.CommandType = CommandType.StoredProcedure;

    cmd.Parameters.AddWithValue("@name", "John");

    MySqlDataReader rdr = cmd.ExecuteReader();
    while (rdr.Read())
    {
        Console.WriteLine(rdr[0] + " --- " + rdr[1]);
    }
    rdr.Close();
}
catch (Exception ex)
{
    Console.WriteLine(ex.ToString());
}
```