Introduction to ADO.NET

Introduction

- **ADO.NET**: *ActiveX Data Objects* ***.NET***

It is the primary data access API (Application Program Interface) of the **.NET** framework.

It provides a set of classes needed for developing database applications with C# or other .NET languages (e.g. VisualBasic).

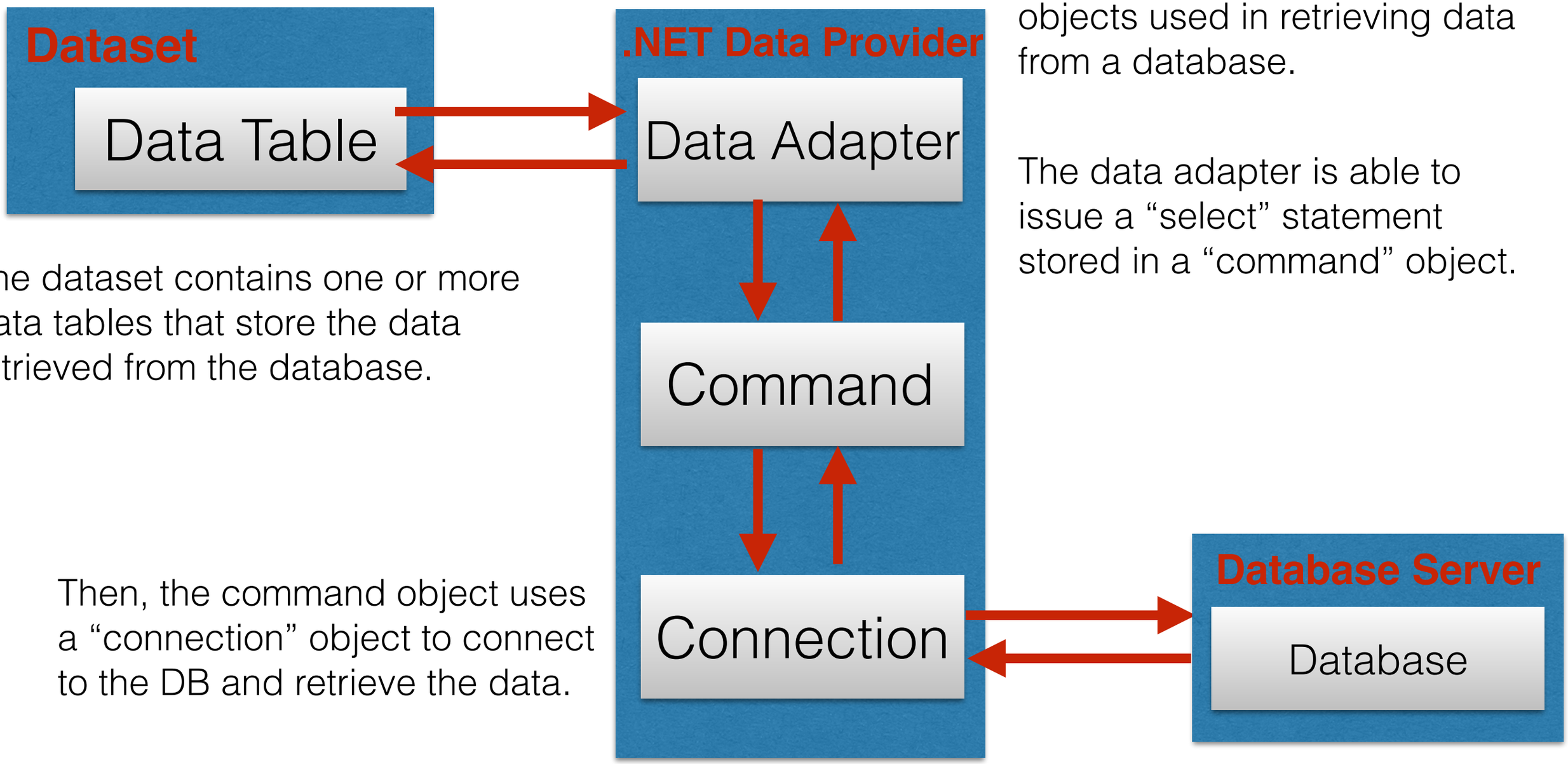
The latest version of ADO.NET is 4.

ADO.NET provides consistent access to data sources such as SQL Server, MySQL, Oracle and XML.
Applications can use ADO.NET to connect to these data sources and retrieve, handle, and update the data that they contain.

ADO.NET separates data access from data manipulation into discrete components that can be used separately or together.

ADO.NET includes data providers for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, placed in an ADO.NET DataSet objects, or combined with data from multiple sources.

ADO.NET Objects Architecture



The data provider provides the classes needed for creating objects used in retrieving data from a database.

The data adapter is able to issue a “select” statement stored in a “command” object.

The dataset contains one or more data tables that store the data retrieved from the database.

Then, the command object uses a “connection” object to connect to the DB and retrieve the data.

Disconnected Data Architecture: When you use a data adapter, the data provider remains connected to the DB only for the time needed to perform the task (update, retrieve, ..). After that, it disconnects from the DB and the application works with her own dataset object. Performance improvement: limited request from the server.

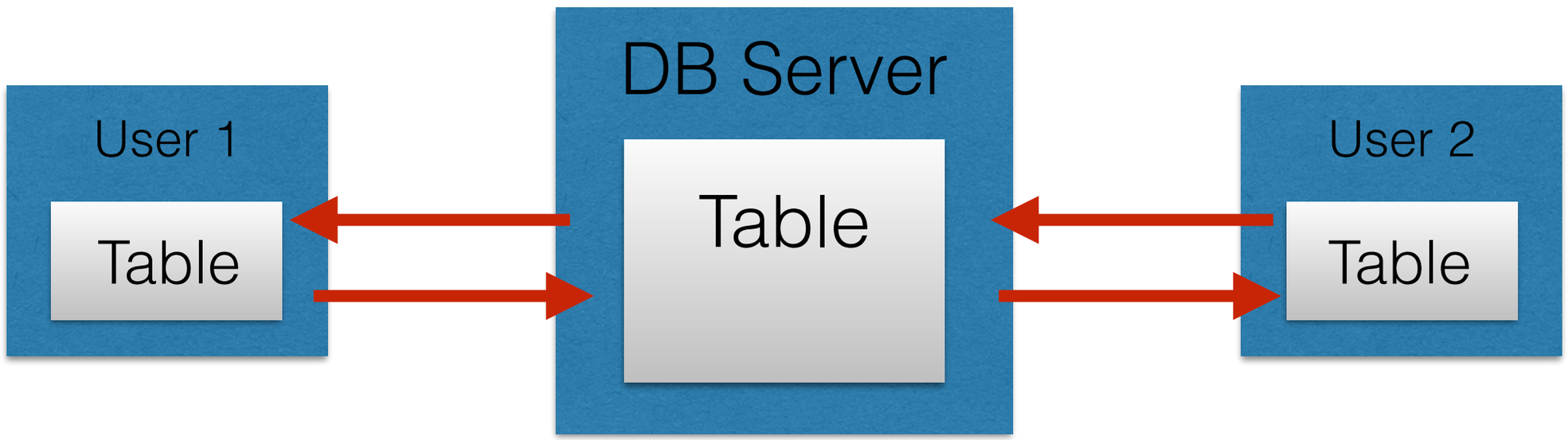
Handling of Concurrency

Client-Server applications naturally pose the problem of concurrency.
More clients can access a database and a specific table at the same time.

Problem:

- User 1 opens a connection, retrieves data and stores them in a local .NET object and then closes the connection. After working on the data, s/he re-opens the connection for updating the table.
- Meanwhile, User 2 completed a similar task.

If the two operations “collide”, in the sense that User 1 modified the table in a way which is inconsistent with the operation of User 2, how will this situation be handled?



Handling of Concurrency

There are two main procedures for concurrency handling:

1) **Optimistic Concurrency**: It this modality, each DB row will be checked for changes since it was retrieved. If a change is detected, a concurrency exception will be thrown. The program must be able to catch the exception and react to it.

2) **“Last in Wins”**: In this modality, no checks are made. The row updated by the last user overwrites the changes of the previous user.

“Last in Wins” easily results in DB corruptions, so optimistic concurrency is used by most programmers.

We will discuss concurrency more in details later.

Handling of Concurrency

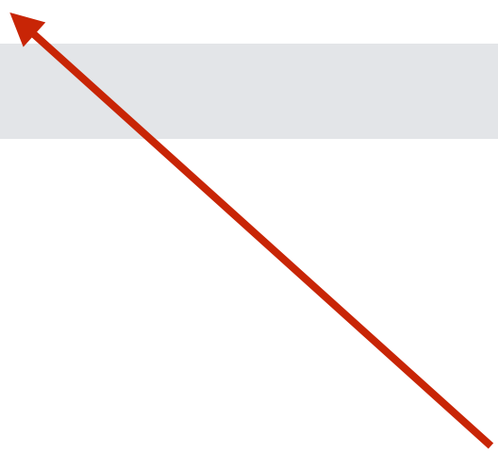
Simple programming techniques for dealing with concurrency are:

- Update the DB frequently, so other users will find your changes.
- Refresh your local dataset frequently so it incorporates changes by other users.
- Retrieve and work on just one row at a time: in this way the probability of collision with another user is reduced.

ADO.NET Data Providers

Object	Description
Connection	Builds the connection to the DB
Command	Represents a single SQL command or stored procedure that can be executed on the DB.
Data Reader	Provides read-only access to the DB
Data Adapter	Provides a link between the command and the connection objects and the dataset object.

Provider	Namespace	Description
SQL Server	System.Data.SqlClient	SQL Server Access
OLE DB	System.Data.OleDb	OLE DB Access
ODBC	System.Data.Odbc	ODBC Access



We are going to use the corresponding provider for **MySQL**.
(Not native in the .NET framework)

The SQLConnection Class

Before working on a DB, you have to establish a connection to it. The SQLConnection class helps you in realizing such a connection with the DB server.
The most important component of SQLConnection is the **connection string** which specifies where the server is, on which DB you would like to work and with which user.

EXAMPLE:

```
string server = “SERVER=140.555.123.321;”;
string database = “DATABASE=databasename;”;
string user = “UID=student;”;
string passwd = “PASSWORD=‘mypasswd’;”;


string connectionstring =
                server+database+user+passwd;
```

Or simply:

```
string connectionstring = @“SERVER= …”;
```

The Connection String

The connection string can be prepared by the user (see previous slide) or can be automatically formed with the aid of a specialized class:

```
System.Data.SqlClient.SqlConnectionStringBuilder bld =
                    new System.Data.SqlClient.SqlConnectionStringBuilder();

                      bld.DataSource = "140.555.123.321";
                      bld.UserID = "username";
                      bld.Password = "passwd";
                      bld.InitialCatalog = "databasename";
                      Console.WriteLine(bld.ConnectionString);//try it
```

The Connection String Methods

Property	Description
`ConnectionString`	This is the connection string
`DataSource`	Address/name of the DB server
`InitialCatalog`	Name of the database
`IntegratedSecurity`	False: user ID and password are included in the connection string. True: Windows integrated security will be used.
`UserID`	Username
`Password`	Password
`PersistSecurityInfo`	False by default. True if the password is returned as part of the connection.
`WorkstationID`	Name of the user's workstation

Not all the properties are mandatory. Using the connection string builder can have some programming advantages: the syntax completions system (IntelliSense) suggests you all the available fields and moreover it is simpler to create connection strings at runtime.

The SQLConnection Class

Basic opening procedure:

```
using MySql.Data.MySqlClient
……
MySqlConnection conn = null;

 try  //try to connect
{
     conn = new MySqlConnection(cstring);
     conn.Open();
     ……
     conn.Close();
}
catch (MySqlException e)
{
      Console.WriteLine(e.ToString());
}
```

Prepare the Database

1) Open an instance of Citrix with the WAMP server
2) Create your DB
3) Create an user and provide the necessary privileges:

```
CREATE USER ‘user’@‘%’ IDENTIFIED BY ‘password’;

GRANT ALL PRIVILEGES ON *.* TO ‘user’@‘%’;

FLUSH PRIVILEGES;
```

Refer to the chosen user and password in your C# code.

Building the first connection with MySQL

1) Open and instance of Citrix with MS Visual Studio

2) Obtain the DLL for MySQL

3) Create a new C# project: eventually it must be a Windows application.

4) Add the DLL to the references in your C# program.

5) Check if you can include the MySQL namespace (see previous slide).

6) Following the C# code in the previous slides, try to realize a connection to your databalse.

7) Add a success/error statement in the two code blocks of “try” and “catch”.

8) Remember to “Close()” your connection at the end.

Pinging the Server

You can try also to “ping” the server from your program:

```
bool pingtest = conn.Ping();
Console.WriteLine("Ping Test = {0}",pt);
```

The Ping() method returns “true” or “false” after having tried the ping.

Related Documentation

For more informations on Data Providers:
https://msdn.microsoft.com/en-us/library/a6cd7c08(v=vs.110).aspx

ADO.NET Architecture:
https://msdn.microsoft.com/en-us/library/27y4ybxw(v=vs.110).aspx

Code examples (for SQL Server)
https://msdn.microsoft.com/en-us/library/dw70f090(v=vs.110).aspx

Project Reminder

1) Start thinking at your final project.

2) Choose a colleague to form a group (if you want to).

3) Is your DB ready?

4) Start building your applications ad Windows applications (graphic GUI).

5) The “connection” phase we have seen today will be eventually placed in an appropriate class: think object-oriented!