CS708 Lecture Notes

Visual Basic.NET Programming

Object-Oriented Programming

VB.NET Data Access Technologies

(Part II)

(Lecture Notes 4B)

Prof. Abel Angel Rodriguez

SECTION I. .NET DATABASE ACCESS TECHNOLOGIES 3

1.0 Review of Application Architecture and Business Objects Data Access 3
1.1 Review of Application Architecture and Business Objects Data Access Requirements 3

1.2 Introduction to Data Access Technolgies 10
1.2 Introduction 10
1.3 What is a Data Source? 10

2.0 Microsoft's Data Access Technologies 11
2.1 OLE DB 11
2.2 ADO.NET 11
2.3 ADO.NET OBJECT MODEL 11
2.4 Available ADO.NET Data Providers 16
2.5 Using ADO in a NutShell 18

3.0 Using ADO.NET in Your Applications 23
3.1 Data Access Using ADO.NET Library 23
3.2 The ADO.NET Connection Class 23
3.3 The ADO.NET Command Class 36
3.4 The ADO.NET DataReader Class 43
3.5 The ADO.NET Parameters Class 58
Summary – Using the Parameters Collection 72
3.6 SUMMARY – Data Access using OleDBDataReader, IN-LINE SQL & SQL SERVER Database 74
Examples 7 - Executing Non-Parameter SELECT Query (IN-LINE SQL) 75
Example 8 - Executing Parameterized SELECT Query 76
Example 9 - Executing SELECT Query that Return Multiple Records 77
Example 10 - Executing Parameterized UPDATE Query 78
Example 11 - Executing Parameterized INSERT Query 79
Example 12 - Executing Parameterized DELETE Query 80

4.0 Using ADO.NET DataSet Class 81
4.1 REVIEW OF Data Access Using DataReader Object 81
4.2 Data Access Using DataSet Object 81
4.3 DataAdapter Class 82
DataAdapter Class – Properties & Methods 83
Using the DataAdapter Class Object 83
4.4 DataSet Class 85
DataSet Class – Properties & Methods 88
Using the DataSet Class Object 88
Example 13 - Executing SELECT Query using DataSet that returns One or Multiple Records 88
Example 14 - Executing SELECT Query that Return TWO Tables using DataSet 91
Example 15 - SELECT Query that Return TWO Tables using DataSet (METHOD II) 94

Section I. .NET Database Access Technologies

1.0 Review of Application Architecture and Business Objects Data Access

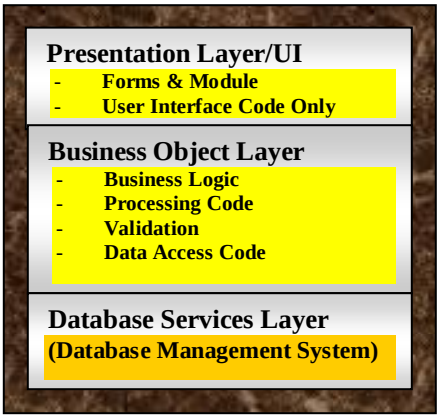
1.1 Review of Application Architecture and Business Objects Data Access Requirements

- Before we begin to understand the details of accessing a data source or database, lets review the basic application architecture and business objects or classes data access requirements.

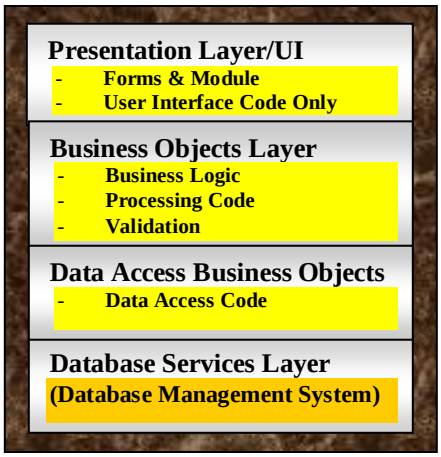
Review of Application Architecture

- The application architecture targeted for ***client/server*** applications is as follows:

Basic 3 Layer Architecture



Basic 4 Layer Architecture



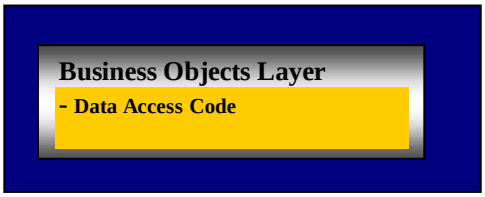
- This architecture is design for the following ***client/server*** architectures:

  - One-Tier Client/Server
  - Two-Tier Client Server
  - N-Tier Client/Server
  - Web-based Client/Server

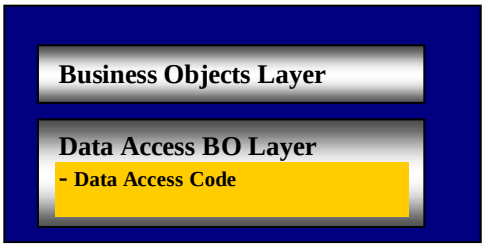
Data Access Design Objectives:

- Our objectives is to implement the Business Objects layers based on either the 3 Layer or 4 Layer architecture:

3 Layer Architecture Business Objects Layer



4 Layer Architecture Business Objects Layers



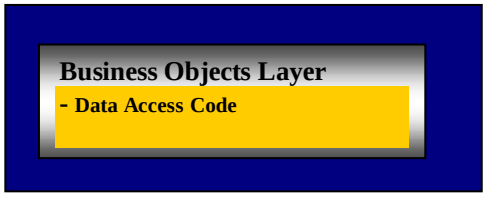
Implementing the Data Access Objectives:

- It is important to decide where to place the **Data Access code** or ***SQL Statements/Stored Procedure calls*** that will ***Load***, ***update***, ***insert*** and ***delete*** the *Objects* to the database.
- Where to place the Data Access Code depends on the application architecture being used (3-Layer or 4 Layer).
- The following methods are available:

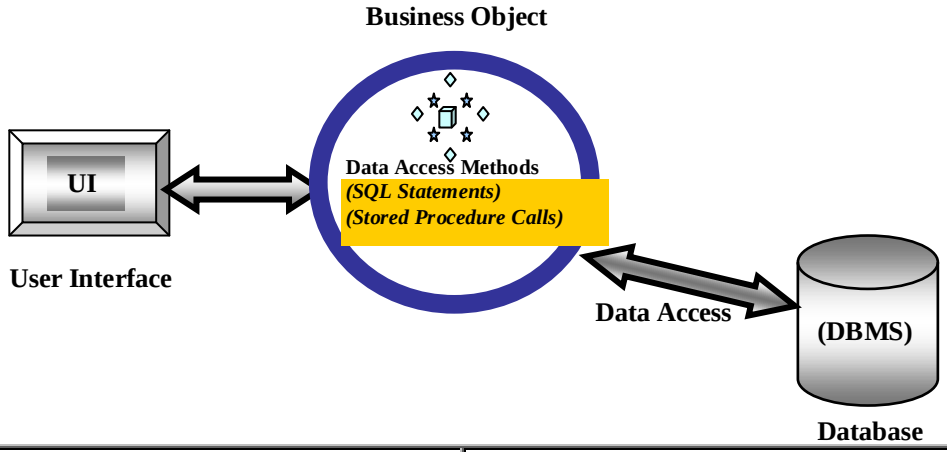
Method I – Business Objects Perform Their Own Data Access (Basic 3-Layered Architecture)

- This method is based on the 3-Layered application architecture:

3 Layer Architecture Business Objects Layer



- In this method it is the Business Objects that handle their own data access
- The *Unanchored* or *Distributed* Business Objects *save*, *update*, *insert*, & *delete* themselves to the database. They contain the queries or *stored procedure* calls to interact with the database:



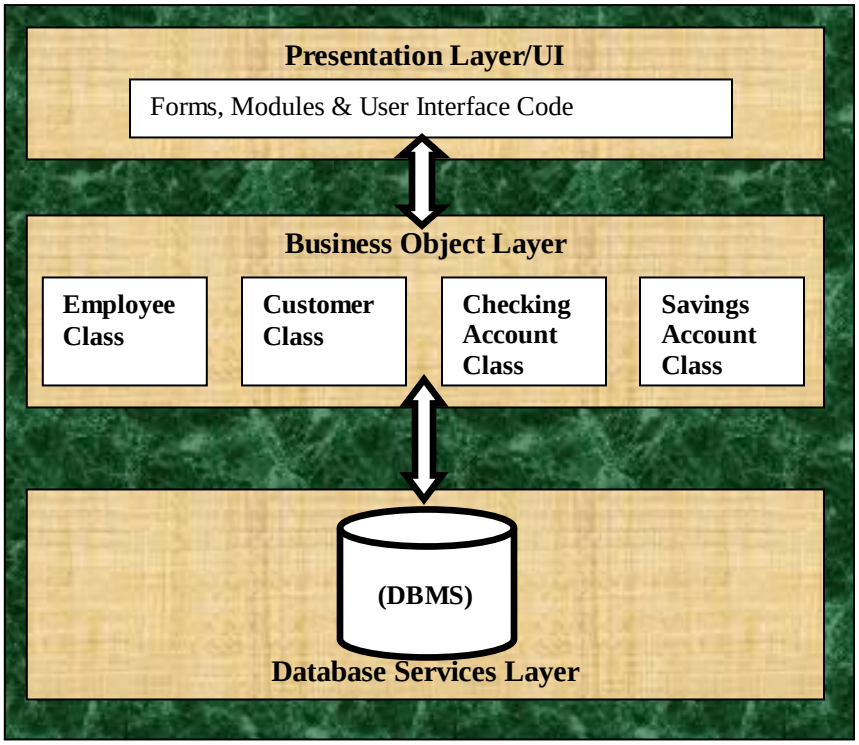
Advantages/Characteristics	Disadvantages
▪ ***Simple***. BO handle themselves ▪ Object is one package with everything we need, thus we have full encapsulation.	▪ Not scalable for our multi-tiered Client/Server architectures.

Example of using this Method:

- Supposed you were creating a client/server Banking Management System or (ATM) application. In this application the following Business Object classes may be required:
  - ***Employee*** Class – represents the employee and performs data access
  - ***Customer*** Class – represents the customers and performs data access
  - ***CheckingAccount*** Class – represents the checking account and performs data access
  - ***SavingsAccount*** Class –represents the savings account and performs data access

- Note you need a total of **4** classes.

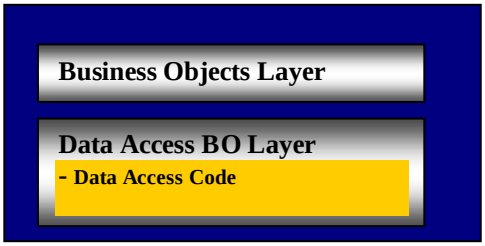
- For the ATM example, the diagram below illustrates the distributed architecture and data access hierarchy.



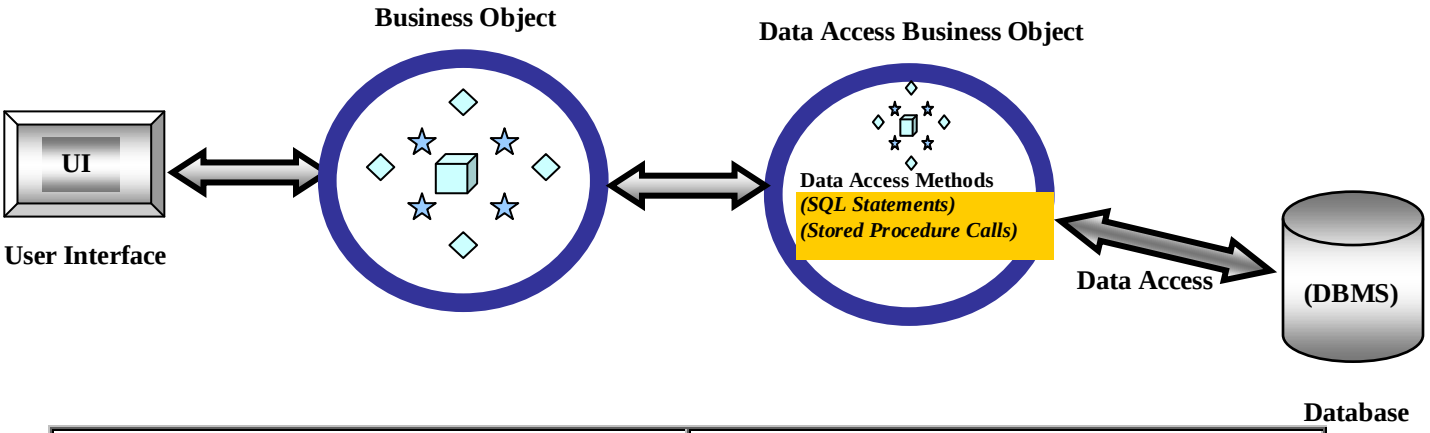
Method II – Data Access Business Objects Handle the Data Access

- This method is based on the 4-Layered Application Architecture Business Objects Layers:

4 Layer Architecture Business Objects Layers



- In this method the Business Object rely on another specialize Business Object to manage or *save, update, insert* & *delete* their data access
- These Data Access Business Objects can be A*nchored* and contain the ***SQL Statements*** or ***Stored Procedures calls*** to interact with the database:



Advantages/Characteristics	Disadvantages
▪ Business Objects are light-weight. Less complex since Data Access BOs contain queries ▪ ***Scalable***. Fits our client/server architectures	▪ Object not one single package but broken up into two separate classes. ▪ Will need one Data Access Object for every type of business objects ▪ Business Object rely on Data Access BOs ▪ More complex to program

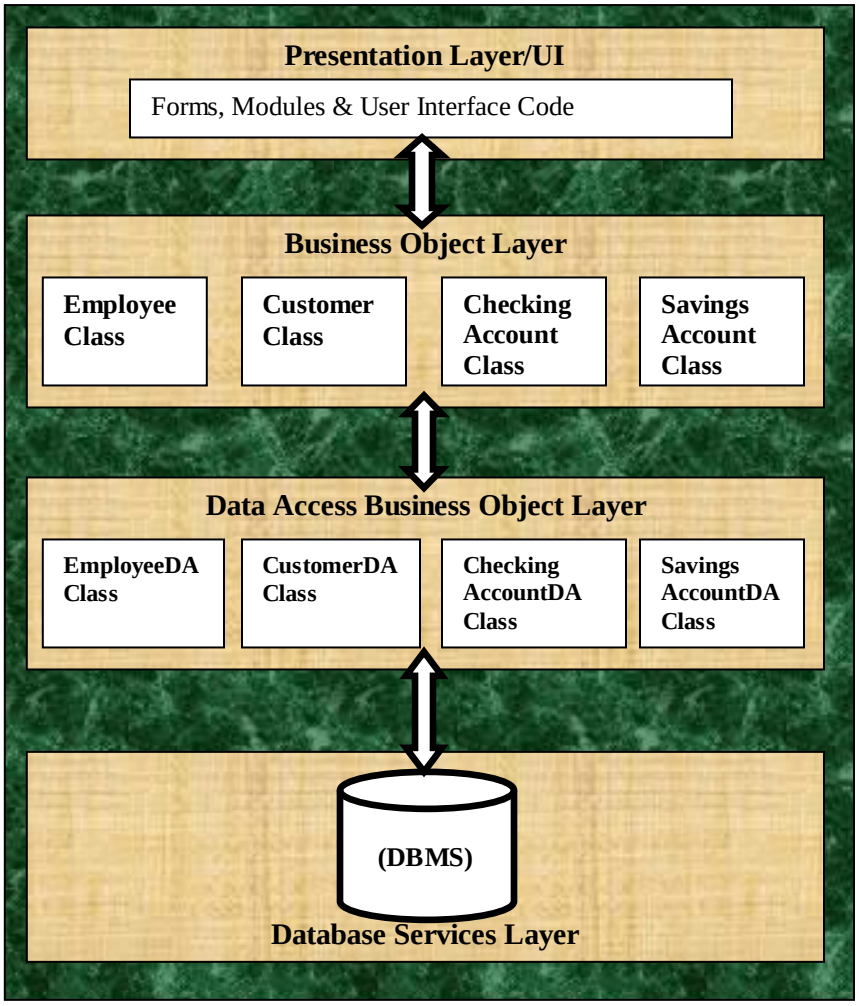
Example of using this Method:

- Supposed you were creating a client/server Banking Management System or (ATM) application. In this application the following Business Object classes may be required:
  - ***Employee*** Class – represents the employee
  - ***Customer*** Class – represents the customers
  - ***CheckingAccount*** Class – represents the checking account
  - ***SavingsAccount*** Class –represents the savings account.

- Now you will need one Data Access Business Object Class for each of the Business Objects as follows:
  - ***EmployeeDataAccess*** Class – performs data access for the Employee Class
  - ***CustomerDataAccess*** Class – performs data access for the Customers Class
  - ***CheckingAccountDataAccess*** Class – performs data access for the CheckingAccount Class.
  - ***SavingsAccountDataAccess*** Class – performs data access for the SavingsAccount Class.

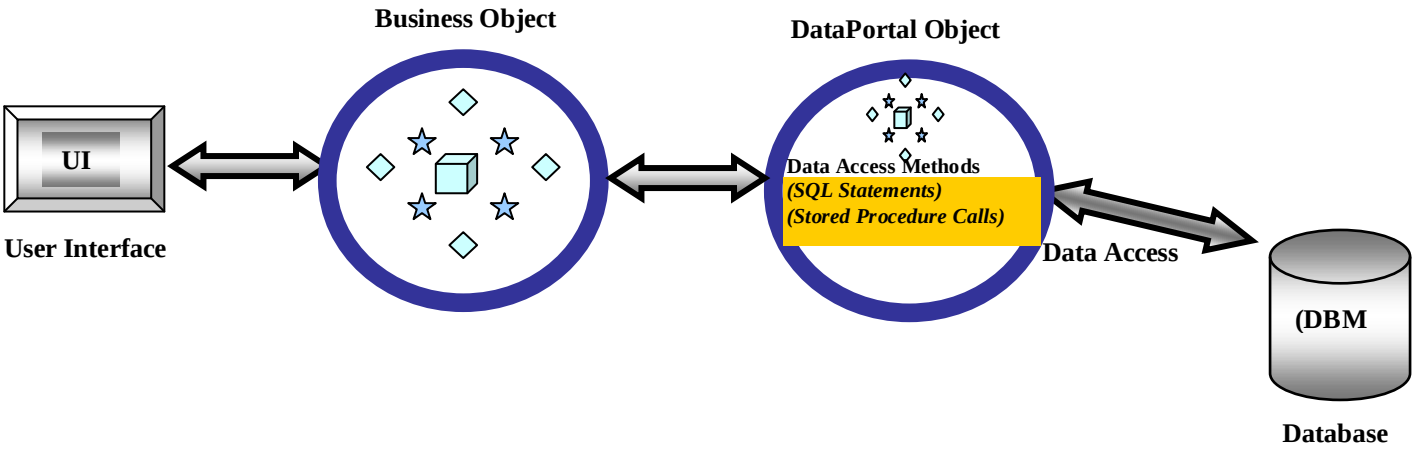
- Note you need a total of **8** classes.

- For the ATM example, the diagram below illustrates the distributed architecture and data access hierarchy.



Method III – General Purpose DataPortal Layer Handle the Data Access (Common Practice)

- In this method the Business Object rely on a general DATAPORTAL Object or Layer to manage or save, update insert & delete their data access
- The DATAPORTAL is usually ***Anchored*** and contain the SQL Statements or queries and interact with the database:



Advantages/Characteristics	Disadvantages
▪ Business Objects are light-weight. Less complex since DataPortal contains queries ▪ ***Scalable***. Fits our client/server architectures ▪ Object partially a single package and encapsulated ▪ One **DataPortal** for all BO objects. ▪ Could have a **DataPortal** for each type of Database SQL, Oracle etc.	▪ No data access code in Business Objects. ▪ Business Objects will always rely on **DataPortal** ▪ Data portal will contain methods to handle each of the individual business objects it will handle. ▪ Need to constantly modify DataPortal object by adding new data access methods every time you add a new class to the project.

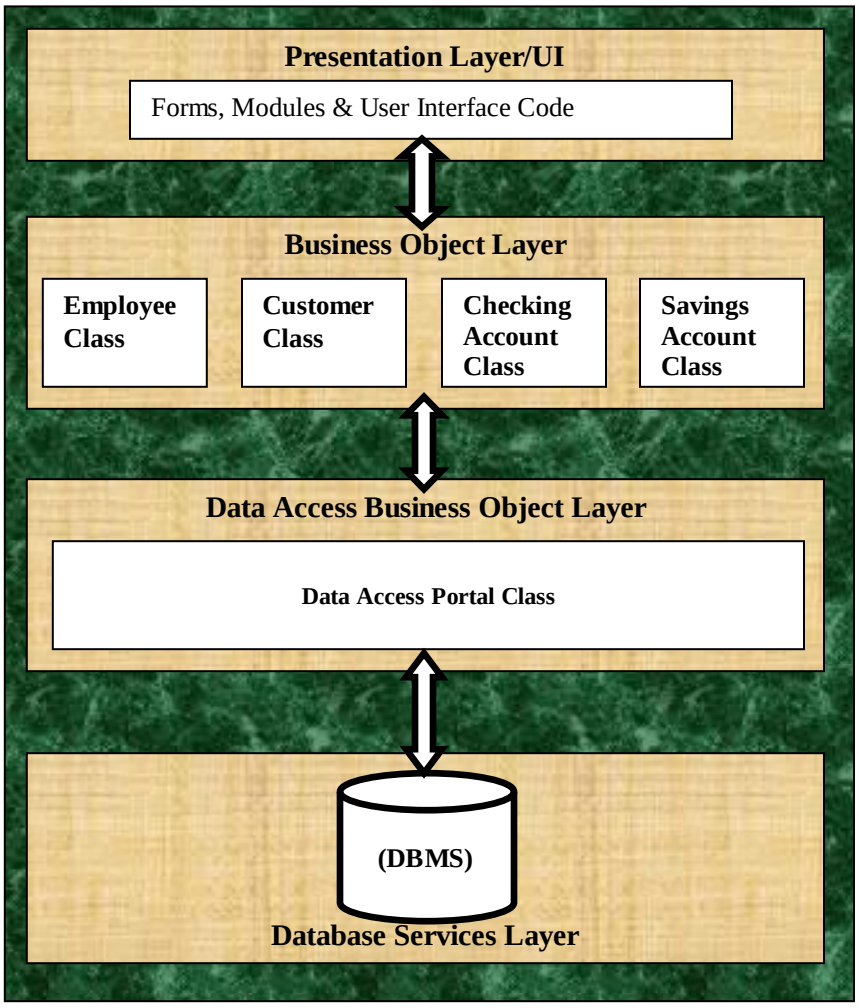
Example of using this Method:

- Supposed you were creating a client/server Banking Management System or (ATM) application. In this application the following Business Object classes may be required:
  - ***Employee*** Class – represents the employee
  - ***Customer*** Class – represents the customers
  - ***CheckingAccount*** Class – represents the checking account
  - ***SavingsAccount*** Class –represents the savings account.

- Now you will need one Data Access Business Object Class for each of the Business Objects as follows:
  - ***DataPortal*** Class – performs data access for all Business Objects (*Employee*, *Customer*, *CheckingAccount* & *SavingsAccount*)
    - The DataPortal Class will contain the following data access methods (*Load*, *Update*, *Insert*, *Delete*) for the ***Employee Class***
    - The DataPortal Class will contain the following data access methods (*Load*, *Update*, *Insert*, *Delete*) for the ***Customer Class***
    - The DataPortal Class will contain the following data access methods (*Load*, *Update*, *Insert*, *Delete*) for the ***CheckingAccount Class***
    - The DataPortal Class will contain the following data access methods (*Load*, *Update*, *Insert*, *Delete*) for the ***SavingsAccount Class***

- Note you need a total of **4** Business classes and **1** ***DataPortal*** Class. Nevertheless, the ***DataPortal*** Class may contain 16 data access methods, 4 for each class that requires its service.

- For the ATM example, the diagram below illustrates the distributed architecture and data access hierarchy.


Presentation Layer/UI
Forms, Modules & User Interface Code
Business Object Layer
Employee Class
Customer Class
Checking Account Class
Savings Account Class
Data Access Business Object Layer
Data Access Portal Class
(DBMS)
Database Services Layer

1.2 Introduction to Data Access Technolgies

1.2 Introduction

- Data Access Technologies refers to available methods or technologies that allow you to access a variety of data.
- These technologies allow us to use programs created in programming languages such as Visual Basic, C++, Java etc. to connect to a ***Database*** & other ***Data Source***.

1.3 What is a Data Source?

- When we think of a ***Database***, we think of programs such as Access, Oracle & SQL Server.
- But there is much more to data than storing it in a Database. Think about email, word-processing, documents, spreadsheets, graphics etc. These examples of data are stored in different format, which can be know or unknown to us.

Definitions of Data Source & Data Access Technologies

- ***Data Source*** - A source of data, that can be in a database or some other source
- ***Access Technologies*** - The technology that allows us to access this data, ***Data Source***.

- Today's technologies offer a variety ***Data Source***, for example:

  - Raw Data taken from a text file
  - Data from traditional Database Applications such as **Oracle** or ***MS SQL Server***.
  - Data formatted for direct access by proprietary programming languages or desktop databases like *MS Access*, *MS Visual Basic or MS FoxPro*
  - Data from Messaging Systems stores or Workflow Applications such as *MS Exchange* or *Lotus Notes*
  - Data formatted for access by applications such as *MS Excel*, *MS Outlook*, *Word Processing* & many others
  - Data from communication programs that identify all users logged onto a network.

- These applications will need data with different ***format*** and each application provides a different ***interface*** for manipulating its data.

1.3 .NET Data Access Technologies

- There are many access technologies available, but we will briefly describe some, but concentrate on a few.
- The most common access technologies are:

  - **ADO.NET** – Microsoft ActiveX Data Objects
  - **OLE DB** – Microsoft's Object Linking and Embedding for Databases
  - **ODBC** - Open Database Connectivity (Know standard. Used by Microsoft, Unix, etc.)

2.0 Microsoft's Data Access Technologies

2.1 OLE DB

- **OLE DB** is Microsoft's universal ***data access technology***.
- **OLE DB** - Object Linking and Embedding for Databases
- **OLE DB is a library or interface that allows transparent data access to a variety of data sources.**
- On top of OLEDB Microsoft has created another layer to simplify using OLEDB, this layer is called **ADO.NET**.

2.2 ADO.NET

- ***ADO.NET*** is Microsoft latest technology for accessing data.
- ***ADO.***NET is a set of libraries that are included in the Microsoft .NET Framework that help you communicate with various data sources.
- ***ADO.***NET is set of Objects derived from ***OLE DB***, which allow your applications to connect to a variety of data sources.
- **ADO will be the data access technology of choice for this course**!
- ADO is a powerful library with many features and functionalities. There are many ways to use ADO. We will only cover some of the basic features required to retrieve, delete, update and insert data to a database.

2.3 ADO.NET OBJECT MODEL

- To truly understand & program ***ADO***, you need to understand the ADO Object Model.
- Basically ADO is divide into two separate Libraries:

.NET Data Provider Component (Connected Objects) :

- The *.NET Data Provider Objects* communicate directly with your database to manage the connections, transactions, retrieval and submittal of changes to the database.
- Since these objects directly communicate with the database and establishes a connection they are know as ***connected*** objects.
- This library includes everything you need to access, store, modify, delete and manage data from a database. Therefore you can perform all data Access using the objects in this library

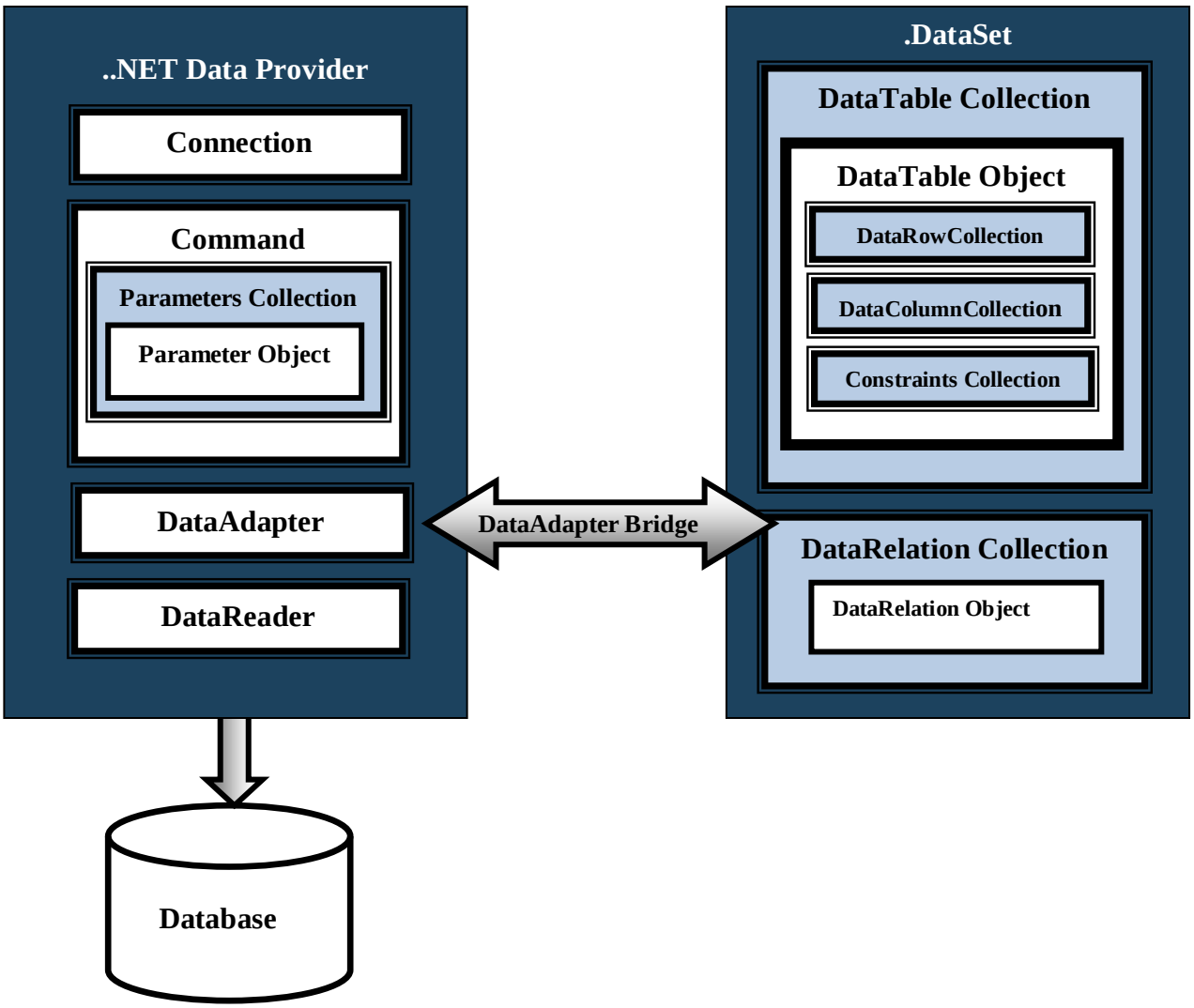
DataSet Component (Disconnected Objects) :

- The ***DataSet*** library objects are used as another mechanism to store the data retrieved from a database. But these objects provide a more powerful data storage mechanism than what is offered in the Provider Object.
- Similar to the *Provider* Objects, these objects store data, but the data is ***disconnected*** from the database. That means once is populated with data, the data is offline. This is a major performance benefit.
- The **DataSet** Objects provide a more complex and powerful storage mechanism, which stores the following type of data:
  - **Data** – Actual data or records from the database. No only can you store the result of a query or a table, but you can store one or multiple tables from the database.
  - **Metadata** – Description of the data such as the data type of the columns, properties, size etc.
  - **Relationship** – Relationships between the tables retrieved from the database

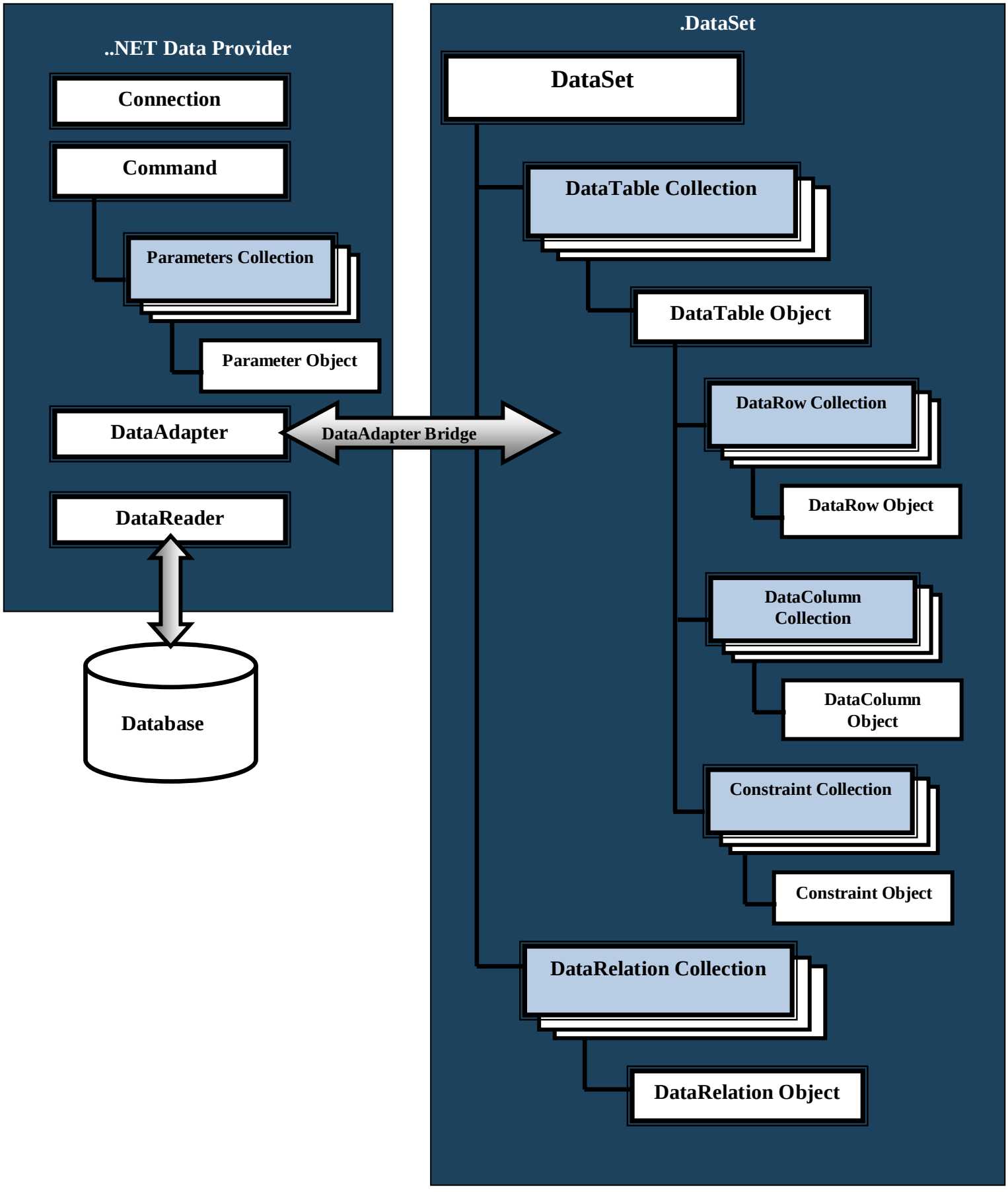
- **As you can see from the data being stored, these Objects can store one or more tables as well as the database schema. You can actually store the entire database locally.**
- **You can also UPDATE, INSERT & DELETE in the DataSet and COMMIT TO DATABASE**
- Therefore if desired, you can query the database once, get all the data you need, work with the data offline and submit changes to the local DataSet. Once you are finished, you can then submit all changes to the main database.
- This mechanism offers the benefit of instead of connection to a database many times, you can get it all locally, work offline, and commit all changes when you are done. This is a big performance benefit.
- FULL SUPPORT FOR DATA BINDING! Data Binding allow us to bind or append data from table directly to Form or WEB Controls, such as Text Box, ListBox, ComboBox etc.

ADO Object Model

- Lets take a look at the ***ADO Object Model*** or architecture:



- Another view of the ***ADO Object Model*** hierarchy:



- The **.NET Data Provider** is composed of 5 major Objects. The objects descriptions is as follows:

.NET Data Provider	***Description***
Data Provider Component	▪ Data Access Library containing the objects shown in the Provider diagram or objects described in this table. ▪ ADO.NET Provider comes in different versions. The providers are as follows: - **ODBC** provider: Connect to ODBC data sources etc. - **OLEDB** Provider: Connect to Access, SQL Server & Oracle - **SQL** Provider: Exclusively connect to SQL Server database - **ORACLE** Provider: Exclusively connect to Oracle database ❖ **This means that there are 4 versions of the architecture shown in diagrams or classes addressed in this table. Four variations for different types of databases**
Connection	▪ This object handles the connection to the data source. ▪ Here you specify the necessary software drivers, security settings and location of the data source you are to connect to.
Command	▪ This object handles or executes the **queries** and **stored procedures**.
Parameter Collection	▪ This Collection Object resides inside the Command Object. It is a child object of the Command Object ▪ This Collection stores object of type ***Parameter***. Each parameter object represents and stores a parameter to be passed to queries
Parameter	▪ These are the objects store by the ParameterCollection ▪ Object of this type are used to store QUERY PARAMTERS. Parameters are the variables or values used in parameterized queries. For example: **SELECT * FROM Customer WHERE Customer_ID = *@CustomerID*** - Here the ***@CustomerID*** represents a value that can be passed from a Form or variable etc.
DataAdapter	▪ This object acts as a *bridge* between the ***Providers*** and Database and the disconnected object the ***DataSet***. ▪ It fetches the results of queries and *fills* or populates the ***DataSet*** or a ***DataTable*** Object so you can work with data offline. This is done via a method inside the *DataAdpater* named ***Fill()***. ▪ The *DataAdapter* object actually exposes a number of properties from the Command Object for *Selecting*, *Updating*, *Inserting* and *Deleting* data to the database. ▪ The *DataAdapter* object populates tables into the **DataSet** or a **DataTable** Object and also submits changes from the DataSet Object to the database. It is a *bridge* between these two libraries.
DataReader	▪ This object has similar functionality as the DataSet objects and that is it stores the result of a query. ▪ Except this Object is a Forward-Only, Read-Only stream of data that bypasses the DataSet Object to directly communicate with the database. ▪ **Read-Only –** You can only read data, no updates or modification is supported ▪ **Forward-Only –** You can examine the results of a query one row at a time, when you move forward to the next row the contents of the previous row are discarded. ▪ Supports minimal features, but is fast and lightweight. ▪ This OBJECT IS BEST FOR QUICK AND FAST DATA ACCESS and IDEAL FOR OBJECT ORIENTED PROGRAMMING.

- The **DataSet** structure contains 5 major Objects. The objects descriptions is as follows:

DataSet	***Description***
DataSet Object	▪ Store the results of a query. Can also DELETE, UPDATE AND INSERT the records in the DataSet and COMMIT to DATABASE. ▪ This object stores a collection of ***DataTable*** Objects and a Collection of DataRelationship Objects. ▪ Each *DataTable* Object stores other collections and objects to manage the data retrieved from the database. ▪ Each *DataRelationship* Object stores information concerning the relationship between the tables, primary & foreign keys that link the tables etc. ▪ The explanation to the other collections and objects is listed below. ▪ Data from queries is stored in the ***DataSet*** object via the .NET Data Provider DataAdapter Object **Fill()** method
DataTable	▪ This object lets you stores and examine the data returned from a query. ▪ The rows and columns returned from a query are stored in two Collections named *DataRow* Collection and *DataColumn* Collection. Both of these child objects will be described below. ▪ Data from queries is stored in the ***DataTable*** object via the .NET Data Provider DataAdapter Object **Fill()** method. ▪ YOU CAN USE THE DATATABLE OBJECT DIRECTLY WITHOUT THE DATASET IF DESIRED for simply database retrieval ▪ **Once data is fetched from the database and stored in the ***DataTable*** object, the connection to the database is closed. You can then examine the content of the ***DataTable*** object totally disconnected from the database without creating any network traffic.
DataColumn Collection	▪ A collection storing ***DataColumn*** Objects. Each of these objects store information about one Column in the table. ▪ Each ***DataColumn*** Object corresponds to one Column in the table. ▪ The ***DataColumn*** object DOES NOT store DATA! It only stores information about the structure of the column or METADATA, such as data type, properties size, format etc.
DataRow Collection	▪ A collection storing ***DataRow*** Objects. Each of these objects store information about the Row in the table. ▪ Each ***DataRow*** Object corresponds to one Row in the table. ▪ The ***DataRow*** object STORES the actual DATA return from a query. ▪ You examine the content of each DataRow object in the collection to retrieve and analyzed the data. ▪ You can use a ***For..Each..Next*** loop to iterate through the collection and access the data.
Constraints Collection	▪ A collection storing ***Contraints*** Objects. Each of these objects store information about the constraints or rules placed on columns or multiple columns in the table stored in the *DataSet*.
DataRelation Collection	▪ A collection storing ***DataRelation*** Objects. Each of these objects store information about the relationship between the tables. ▪ Also information about the primary & foreign keys that link the tables. ▪ In addition this object enforces referential integrity.

2.4 Available ADO.NET Data Providers

- The previous section illustrated the object model diagram for ADO.NET and two tables describing each of the ADO.NET classes
- We now understand that ADO.NET is divided into two parts, the .NET PROVIDER and DATA SET:

I. ADO.NET DATA PROVIDER CLASSES:

- Connection
- Command
- Parameters Collection
- Data Adapter
- Data Reader

II. DATA SET CLASSES:

- DataTable Collection
- DataTable Classes
- DataRow Collection
- DataColumn Collection
- Constraint Collection
- DataRelation Collection

- NOTE THAT THE CLASS NAMES SHOWN FOR THE DATA PROVIDER (Connection, Command, Data Adapter & Data Reader) ARE THE NOT THE ACTUAL NAMES OF THE DATA PROVIDER CLASSES!
- These are not the actual names for the objects that exist in the ADO.NET library.
- The ADO.NET DATA PROVIDER library is made up of several .NET Data Provider libraries, specifically tailored for different databases in the market.
- Example of 4 such providers that ship with the .NET Framework are:

1. **SQL Client** – Contains all the ADO.NET libraries customized to connect to a Microsoft SQL Server database ONLY. Best performance for SQL Server database access.

2. **OleDB Client** – Use for other databases that support OleDB such as Microsoft Access, Oracle, SQL Server as well, etc.

3. **Oracle Client** – Exclusively dedicated for Oracle databases only. Gives best performance if connecting to Oracle database only.

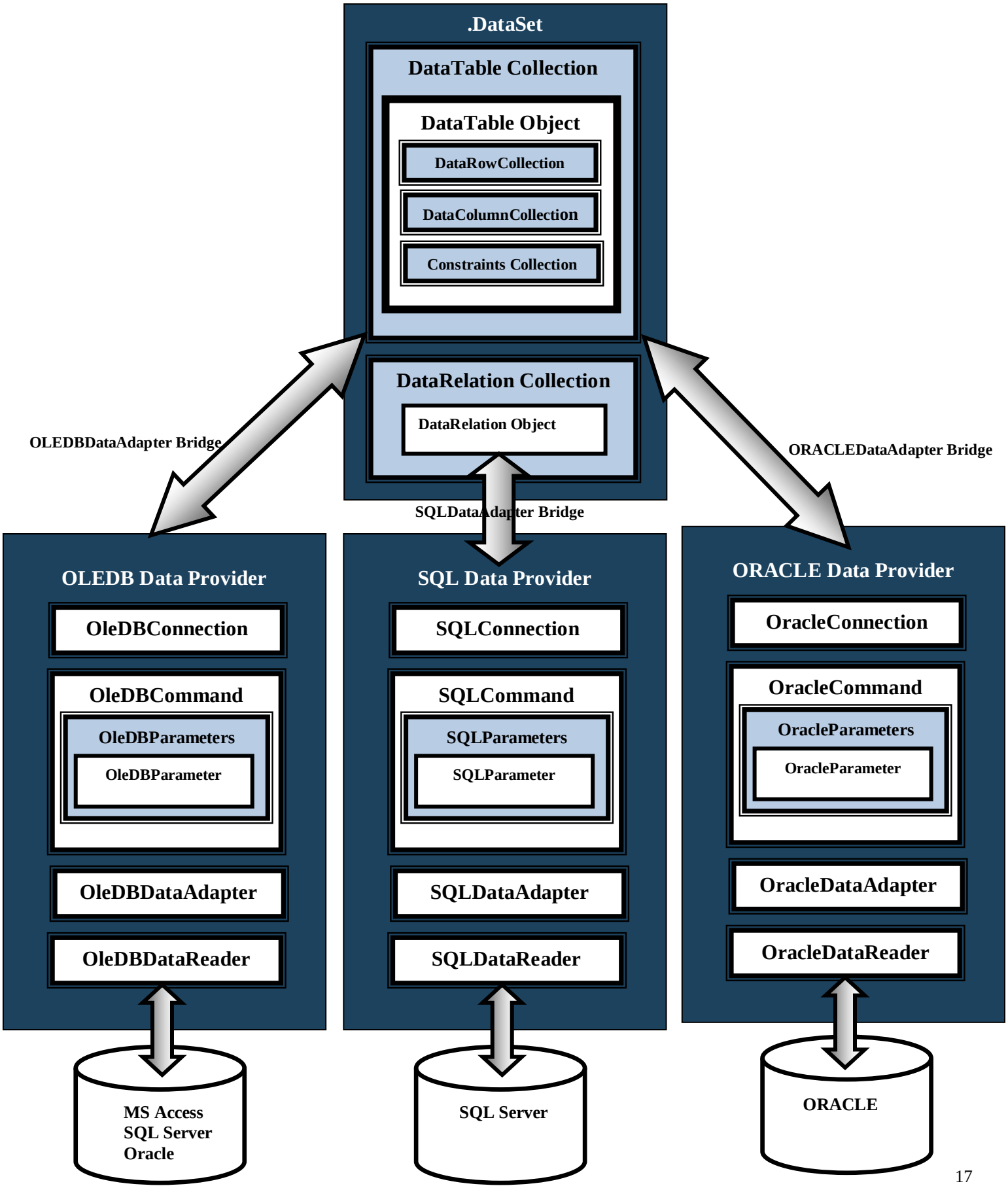
4. **ODBC Client** – Exclusively dedicated for ODBC databases.

- What this means is that there are (Connection, Command, Data Adapter, & Data Reader classes for each of these ADO.NET DATA PROVIDER.
- For each of the providers, a unique name is used for the classes as follows for the SQL, OLEDB & ORACLE CLIENTS:

1. **SQL Client**:	2. **OleDBClient**:	3. **OracleClient**:
○ **SQLConnection**	○ **OleDBConnection**	○ **OracleConnection**
○ **SQLCommand**	○ **OleDBCommand**	○ **OracleCommand**
○ **SQLParameters**	○ **OleDBParameters**	○ **OracleParameters**
○ **SQLDataAdapter**	○ **OleDBDataAdapter**	○ **OracleDataAdapter**
○ **SQLDataReader**	○ **OleDBDataReader**	○ **OracleDataReader**

- Note that there is only one **DataSet** and **DataTable** Library used by ALL of the providers
- Note that you can purchase vendor specific providers
- IN SUMMARY, THE FIRST THING YOU NEED TO DO IS DECIDE WHICH PROVIDER TO USE WHEN CREATING YOUR APPLICATION DATA ACCESS CODE.
- THIS WILL DEPEND ON THE DATABASE YOU ARE USING AND THE REQUIREMENTS
- ONCE THIS DECISION IS MADE, YOU CREATE THE CLASSES FOR THE SPECIFIC PROVIDER.

- SQL, OLEDB & ORACLE Providers & DataSet architecture:



2.5 Using ADO in a NutShell

Using the ADO.NET Library

- OK, we saw the ADO.NET Class Library. How do we use it?
- Well, these are classes, which means that Microsoft already created them for us, therefore following our OOP rules, we need to do the following:

1. **Create Objects of the ADO.NET classes**
   a. **Import the ADO.NET libraries into your program**

2. **Read MSDN LIBRARY for description of Properties & methods of ADO.NET Classes**
3. **Use the Objects to perform data access:**
   a. **Set & Get properties**
   b. **Call methods**
   c. **Etc.**

Which Objects or Class do we use? In what order?

- But which Objects do we create first? How do I use the library?
- From the description of the library, we definitely need the following objects:

  - **Connection** object – To establish the connection
  - **Command** Object – To execute the query
  - **Command.Parameters Collection** – To store parameters of a query. Note that this object resides inside the Command Object

- Now we need to decide where we are going to store the results of a query, and the choices are:

Option 1: (DATAREADER)	**Option 2: (DATASET)**	**Option 3: (DATATABLE)**
▪ **DataReader** object: - Fast, reliable storage mechanism. - Connected: Always maintains a connection with data source. - Forward-read only, data is discarded after reading it. - Limited, but recommended for most data access needs. - IF YOU SIMPLY RETURNING ONE OR A FEW RECORDS, THE DATAREADER IS IDEAL. - FULL DATA BINDING SUPPORT	▪ **DataAdapter** object: - *Bridge* to the **DataSet** or **DataTable** Object. Manages the operations performed on the DataSet Object. - *Fills* or populates the DataSet Object with data. - *Updates* or *commits* changes within the **DataSet** Object to the Data Source. ▪ **DataSet** object: - Powerful, reliable storage mechanism. - Extensive functionality. - Disconnected (Great for performance). - Can store entire tables, etc. Recommended when you need to store and manage large number of records or tables. - More complex to program! - FULL DATA BINDING SUPPORT	▪ **DataAdapter** object: - *Bridge* to the **DataSet** or **DataTable** Object. Manages the operations performed on the DataSet Object. - *Fills* or populates the DataSet Object with data. - *Updates* or *commits* changes within the **DataTable** Object to the Data Source. ▪ **DataTable** object: - If you need the features of a DATASET, but simply returning a table, you can also use a **DataTable** Object by itself to store your data off-line instead of a DATASET. The *DataSet* is a complex structure which takes up resources, therefore for returning only a table, you can use a **DATATABLE** object only. - FULL DATA BINDING SUPPORT

Putting it all together

- So, to use ADO.NET we will be doing the following:

Step 1: Decide Which Provider to Use:

- Options:
  - **SQL Client** – Use this provider if the connections will always be to a Microsoft SQL Server database ONLY
  - **OleDB Client** – Use this provider if you are connection to a Microsoft Access, Oracle, SQL Server, or other types of data sources.
  - **Oracle Client** – Exclusively dedicated for Oracle databases.

Step 2: Import the ADO.NET Data Provider library NAMESPACE

- For **OLEDB Client** Provider type, import the library:

```
Imports System.Data.OleDb  'OLEDB Data Provider
```

- For **SQL CLIENT** Provider type, import the library:

```
Imports System.Data.SQLClient SQL Data Provider
```

- For **ORACLE CLIENT** Provider type, import the library:

```
Imports System.Data.OracleClient Oracle Data Provider
```

```
Imports System.Data
Imports System.Data.OleDb  'OLEDB Provider
```

Step 3: Create ADO.NET Data Provider Objects & Use them by Calling Methods and Properties

- Create Objects of the following ADO.NET classes:

Option 1 – OLEDB Client	Option 2 – SQL Client	Option 3 – ORACLE Client
▪ **OLEDBConnection** ▪ **OLEDBCommand** ▪ **OLEDBParameters**	▪ **SQLConnection** ▪ **SQLCommand** ▪ **SQLParameters**	▪ **ORACLEConnection** ▪ **ORACLECommand** ▪ **ORACLE**Parameters

- Use the Objects, by calling their *properties* and *methods*.
  - **Connection Object** – Call methods/Properties to connect to database or data source of choice
  - **Command Object** – Call methods/Properties to execute Query.
  - **Parameters Collection & Parameter Object** – Call methods/Properties to prepare parameters passed to Queries used by command object.

Step 3: Decide which ADO.NET Object(s) to Store the Result of the Query & Call Methods/Properties to manage data.

1. Decide where to store your query results. The choice of implementation are as follows:

Option 1 – Data Access that requires simple and fast Data Access	**Option 2 – Data Access that returns MULTIPLE TABLES, manage large number of records, require off-line/disconnected performance, etc.**	**Option 3 – Data Access that returns ONE TABLE, manage large number of records, require off-line/disconnected performance, etc.**
▪ If the data access requires simple, fast & reliable data access, such as returning ONE RECORD, returning a few number of records, we are not modifying or making a lot of changes to many tables, don't require disconnected performance etc. ▪ If you will simply retrieve the data, use it and discard it, then you should use this option ▪ IDEAL FOR OBJECT-ORIENTED PROGRAMMING ▪ You will need the following Object(s): - **DataReader** Object	▪ If the data access requires management of a large number of records, such as updating, inserting, deleting etc., on many tables. Then this option is best. ▪ For example if the part of the application requires that the user works with a lot of records or is going to make bulk changes to a large number of records, then this option is best since it allows you to store and modify large number of records or tables locally and commit them in one step. ▪ This option requires more complex programming. ▪ You will need the following Object(s): - **DataAdapter** object - **DataSet** object	▪ If the data access requirements are the same as the DataSet, but you are only returning ONE TABLE, you can create ONE DATASET object. ▪ For example if the part of the application requires that the user works with a lot of records or is going to make bulk changes to a large number of records, but only one table is being returned, this option is best. ▪ You will need the following Object(s): - **DataAdapter** object - **DataTable** object

2. Import the ADO.NET Data component NAMESPACE:

```
Imports System.Data 'Required for DATASET & DATATABLE classes
```

3. Once you have decided on where to store your data, either a **DataReader, DataSet** or single **DataTable** Object, you create the objects and use the Object(), by calling properties and methods to retrieve/manage the stored data as follows:

Option 1 – DATAREADER Object	**Option 2 – DATASET** Object	**Option 2 – DATATABLE** Object
▪ ***OLEDB Client*** provider create: - **OLEDBDataReader** object - Call methods/Properties to retrieve and manage the data ▪ ***SQL Client*** provider create: - **SQLDataReader** object - Call methods/Properties to retrieve and manage the data	▪ ***OLEDB Client*** provider create: **1. OLEDBDataAdapter** object - Call methods/Properties to *populate*, *update*, *Insert* & *Delete* records in **DataSet**. In addition *commit changes to database* **2. DataSet** object - Call methods/Properties to retrieve and manage the data ▪ ***SQL Client*** provider create: **1. SQLDataAdapter** object - Call methods/Properties to *populate*, *update*, *Insert* & *Delete* records in **DataSet**. In addition *commit changes to database*. **2. DataSet** object - Call methods/Properties to retrieve and manage the data	▪ ***OLEDB Client*** provider create: **1. OLEDBDataAdapter** object - Call methods/Properties to *populate*, *update*, *Insert* & *Delete* records in **DataTable**. In addition *commit changes to database* **2. DataTable** object - Call methods/Properties to retrieve and manage the data ▪ ***SQL Client*** provider create: **1. SQLDataAdapter** object - Call methods/Properties to *populate*, *update*, *Insert* & *Delete* records in **DataSet**. In addition *commit changes to database*. **2. DataTable** object - Call methods/Properties to retrieve and manage the data

Final words & Summary

- For any of the data access options we need a connection, command, & parameters objects. But depending on which storage mechanism we decide on, we either create a **DataReader** or **DataAdapter/DataSet/DataTable** objects.
- For fast and simple data access, using the **DataReader** to quickly store and retrieve our data is the way to go.
- For more complex data access such as working with many tables etc., the **DataAdapter/DataSet** objects is best.

- The table below is a listing of the required ADO.NET Objects for OLEDB provider use:

Option 2A – OLEDB Provider Using DATAREADER	Option 2B – OLEDB Provider using DATASET	Option 2C – OLEDB Provider Using DATATABLE
- **OLEDBConnection** Object - **OLEDBCommand** Object - **OLEDBParameters** Collection Object - **OLEDBDataReader** Object	- **OLEDBConnection** Object - **OLEDBCommand** Object - **OLEDBParameters** Collection Object - **OLEDBDataAdapter** Object - **DataSet Object**	- **OLEDBConnection** Object - **OLEDBCommand** Object - **OLEDBParameters** Collection Object - **OLEDBDataAdapter** Object - **DataTable** Object

- The table below is a final listing of the available ADO.NET Objects required for using the SQL Provider:

Option 1A – SQL Provider Using DATAREADER	Option 1B – SQL Provider using DATASET	Option 1C – SQL Provider Using DATATABLE
- **SQLConnection** Object - **SQLCommand** Object - **SQLParameters** Collection Object - **SQLDataReader** Object	- **SQLConnection** Object - **SQLCommand** Object - **SQLParameters** Collection Object - **SQLDataAdapter** Object - **DataSet** Object	- **SQLConnection** Object - **SQLCommand** Object - **SQLDBParameters** Collection Object - **DataTable** Object

- The table below is a listing of required objects for ORACLE client available in ADO.NET:

Option 2A – ORACLE Provider Using DATAREADER	Option 2B – ORACLE Provider using DATASET	Option 2C – ORACLE Provider Using DATATABLE
- **ORACLEConnection** Object - **ORACLECommand** Object - **ORACLEParameters** Collection Object - **ORACLEDataReader** Object	- **ORACLEConnection** Object - **ORACLECommand** Object - **ORACLEParameters** Collection Object - **ORACLEDataAdapter** Object - **DataSet** Object	- **ORACLEConnection** Object - **ORACLECommand** Object - **ORACLEParameters** Collection Object - **ORACLEDataAdapter** Object - **DataTable** Object

3.0 Using ADO.NET in Your Applications

3.1 Data Access Using ADO.NET Library

- In this section I will demonstrate how to use the ADO.NET objects within our Class Objects to perform data access.
- Be aware that there are many ways to use the ADO.NET library objects. ADO.NET objects can indirectly call each other without having to create them individually.
- In other words, you can create each of the 5 main ADO.NET objects to perform the data access, or with just creating a connection object you can call methods that create command objects to handle the SQL command etc. So instead of creating 5 objects you only needed to create one or a connection object and you were able to indirectly create a *Command* Objects, *DataReader* object etc., all from within the Connection Object.
- The point here is that there are many ways to perform the data access using the ADO.NET model. I will show you one of them.

Approach to Learning Data Access in this Course

- The approach I am going to take is as follows:

  - Select each of the required ADO.NET object; show a table with some of the properties and method available to this object.
  - You can use the table as reference to build the code.
  - Show the code required to perform the data access.
  - In the following code, tables & examples, I will only show properties and methods for the OLEDB provider since in this course we will be connection to both MS Access and SQL Server. Nevertheless, keep in mind that the SQL Server Data Provider is available and each of these library or classes have properties and methods that are very similar in name to the OLEDB, but are implemented differently to target an SQL Server Database.

3.2 The ADO.NET Connection Class

- We need a connection object to connect to a database. The table below again provides a description of the ADO.NET Connection object:

.NET Data Provider	*Description*
Connection	▪ This object handles the connection to the data source. ▪ Here you specify the necessary software drivers, security settings and location of the data source you are to connect to.

- What I will do next is list a table with some of the important properties and methods of the Connection Class.
- Remember that the .NET Library comes equipped with a provider or library for the type of database you want to connect. As you know there are two .NET Data Providers that ship with the .NET Framework, one for SQL Server (SQL Provider), the other for MS Access, Oracle and other OLEDB compliant database (OLEDB Provider). Point here is that there are three to four connection class available, one for SQL Server and the other for MS Access etc:

❖ **SQL Client** – Contains all the ADO.NET libraries to connect ONLY to SQL Server database:
  - **SQLConnection**

❖ **OleDBClient** – Use to connect to SQL Server and other databases that support OleDB, such as Microsoft Access, Oracle etc.
  - **OleDBConnection**

❖ **OracleClient** – Use to connect to SQL Server and other databases that support OleDB, such as Microsoft Access, Oracle etc.
  - **OracleConnection**

- In my code, I will only show the OLEDB provider, nevertheless the SQL provider code is identical.

Connection Class – Properties & Methods

- Table below lists some important properties, methods and constructor of the **Connection Class**:

Public Constructors

Constructore	Description & Examples
OleDbConnection Constructors: Default Constructor: **Public Sub New()** Parameterized Constructor passing connection string: **Public Sub New(ByVal** *connectionString* **As** String**)**	▪ Overloaded. Initializes or creates a new instance or Object of the **OleDbConnection** class via the default constructor. Example: `Dim objConn As New OleDbConnection()` ▪ Parameterized constructor to pass a connection string when creating object of this class. (More on this later). Example: `Dim objConn As New OleDbConnection(strConn)`

Public Properties

Property	Description
ConnectionString	Gets or sets the string used to open a database.
State	▪ Gets the current state of the connection. Connection state can be open or closed. ▪ Value return is of a special type defined within the ADO.NET library. The enumerated type called **ConnectionState**. Example of two values of this type is: - **ConnectionState.Open** - **ConnectionState.Closed** ▪ You can test the State Property of a Connection Object against these value to verify if a connection is open. For example: `If objConnection.State <> ConnectionState.Open Then` `  'go ahead and open the connection since` `  'it's not already opened` `End If`

Public Methods

Method	Description
Open	Opens a database connection with the property settings specified by the **ConnectionString**.
Close	Closes the connection to the data source. This is the preferred method of closing any open connection.
CreateCommand	Creates and returns an **Command** object associated with the **Connection**. We can create a Command Object here if we like.
Dispose	Releases the resources

- In the table below I will list some of the Exceptions that are raised when we use the connection object. You can use Try-Catch-Block statement to trap for these exceptions:

Exception Handling

Exceptions Class	Description
OleDBException	▪ Exception object that catch connection-level errors that occur while trying to open a connection. ▪ Note that this class is for the OLEDB provider. If you were using the SQL or Oracle provider the class you would need is: **SQLException** **OracleException**
InvalidOperationException	▪ This is a general Exception object to trap when the database connection is already open.

Using the Connection Object

- Steps to add code to use the Connection Object in your applications:

Step 1: Verify that the ADO.NET Data Provider NAMESPACE are imported into your code

- For **OLEDB Client** Provider type, import the library:

```
Imports System.Data
Imports System.Data.OleDb  'OLEDB Provider
```

- For **SQL CLIENT** Provider type, import the library:

```
Imports System.Data
Imports System.Data.SQLClient SQL Data Provider
```

Step 2: Prepare a Connection String

Dynamic Connection String

- In the next step you will need to use a connection string. So I will briefly explain how to do this now.
- A connection string is a text string which contains CONNECTION INFORMATION required to connect to a database, such as SERVER NAME, DATABASE NAME, PATH ETC.
- A connection string varied base on database types, SQL Server, MS Access and Oracle.
- A connection string uses a **SETTING=VALUE** pair combinations separated by a semicolon as follows:

Setting1 =Value1;Setting2=Value2;Setting3=Value3……"

- Assuming you create a variable ***strConn*** to store the connection string, the syntax is as follows:

```
strConn = Setting1 = Value1;Setting2=Value2;Setting3=Value3……"
```

- The **Settings** and **Values** will be different and will depend on the following:
  - Which .NET Provider you have chosen: *SQL Client*, *OleDB Client* or *Oracle Client*.
  - The technology such as device drivers etc.
  - Type of data you are connection to
  - Whether you want security such as username & password in the string or a reference to a file containing the encrypted username & password etc.

- You will need to read the documentation in order to determine the various settings and values. Nevertheless the SYNTAX OR RULE for a particular provider is the same, therefore you can simply copy paste a connection string and modify it to your preference.

Microsoft ACCESS OLEDB Connection String SYNTAX:

- In this section we describe the Connection String Syntax for MS ACCESS & SQL SERVER:
- The **VALUES** of the Connection String **SETTINGS** are:

SETTING	VALUE DESCRIPTION	EXAMPLE
Provider	**OleDBClient provider Driver**	`Microsoft.Jet.OleDB.4.0;Data`
Data Source	Path to MDB file	`C:\CS708\DB\video.mdb`
User ID	[Optional] Username required for security	`DBHRUser01`
Password	[Optional] Password required for security	`DBPassword01`

- The **OleDBClient** .NET provider connection syntax to an **Access Database**:
  - The connection string syntax with security is as follows:

Provider = ***Provider_Name***;**Data Source=Database *Path*;**
User ID=*Username*;Password=*Password*

  - The connection string syntax without security:

Provider = ***Provider_Name***;**Data Source=Database *Path***

  - In most cases just the Provider and Data Source settings/value are required for a connection string. Example of a connection string that connects to an **MS Access 2000** database without security:

```
Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\CS708\DB\video.mdb
```

  - Example of a connection string that connects to an **MS Access 2000** database with security:

```
Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\CS708\DB\video.mdb;
User=DBHRUser01;Password=DBPassword01
```

Microsoft SQL SERVER OLEDB Connection String SYNTAX:

- The database connection syntax for an SQL SERVER varied depending on which type of Authentication feature we implement. The options are as follows:

1. Windows Authentication:

- Applications connecting to the database are authenticated via Windows Network Operating System, or WINDOWS ACTIVE DIRECTORY
- In the Windows Authentication Mode, access is granted based on a security token assigned during successful domain (or local server) logon by a Windows account, which subsequently requests access to SQL Server resources
- In Windows Authentication Mode ONLY WINDOWS ACCOUNTS can be granted login rights in SQL Server.
- The ***Windows Administrators*** use ACTIVE DIRECTORY USERS & COMPUTERS to CREATE USERACCOUNTS (Username/Password) which can authenticate to the SQL SERVER.
- The ***Database Administrators*** must configure SQL server to use Windows Authentication and give permission to the Windows Account.
- This method has many advantages; one is that you can grant access to the SQL SERVER to all members of a WINDOWS GROUP.
- RECOMMENDED BEST PRACTICE!!! Does not required username & password in the connection string. PREFERED METHOD OF AUTHENTICATE FOR SECURITY CONPLIANCE!!!!

2. Mixed Mode (Database Authentication & Windows Authentication):

- Applications connecting to the database are authenticated via WINDOWS ACTIVE DIRECTORY or DATABASE SECURITY Mechanism

Database Authentication:

- Users connected to the database are authenticated via a Username/Password passed in the connection string.
- The Database Administrator uses the SQL SERVER Security Console and create a User Account (Username/Password) to authenticate the connection.
- Verification of credentials stored and maintained by the SQL Server
- SECURITY RISK PASSING USERNAME & PASSWORD in connection string. You can encrypt the string, but nevertheless there are security risks.

Windows Authentication:

- This mode supports windows authentication as well.

I. DATABASE AUTHENTICATION Connection String SYNTAX:

- The **VALUES** of the Connection String **SETTINGS** for **OLEDB SQL SERVER** are:

SETTING	DESCRIPTION	VALUE EXAMPLES
Provider	**SQLOLEDB provider Driver**	**SQLOLEDB**
Data Source	▪ Name of SERVER where database resides. ▪ If using SQL SERVER 2005 EXPRESS, syntax: ServerName\SQLEXPRESS	▪ AppServer01 ▪ AppServer01\SQLEXPRESS
Database (or Initial Catalog)	Name of DATABASE	HRDatabase
User ID	Username required for database authentication security	Username
Password	Password required for database authentication security	Passord

OLEDB CLIENT:

- The **OLEDB Client** .NET provider connection syntax to an SQL Server Database:

  - The connection string handles the connection to the data source and it's syntax is as follows:

    Provider = *Provider*;Data Source=*ServerName*;Database=DatabaseName;User ID=*Username*;Password=*Password*

  - Example of a connection string that connects to an SQL Server Database with security:

```
Provider=SQLOLEDB;DataSource=AppServer01;Database=HRDatabase;User
ID=Username;Password=Password
```

  - Second syntax using "***Initial Catalog***" instead of "***Database***":

    Provider = *Provider*;Data Source=*ServerName*;Database=Initial Catalog;User ID=*Username*;Password=*Password*

  - Example of a connection string using Initial Catalog:

```
Provider=SQLOLEDB;DataSource=AppServer01;Initial Catalog=HRDatabase;User
ID=Username;Password=Password
```

  - **OLEDB Client SQL SERVER EXPRESS** connection string syntax:

    Provider = *Provider*;Data Source=*ServerNameSQLEXPRESS*;Database=DatabaseName;**
    User ID=*Username*;Password=*Password*

  - Example of a connection string that connects to an SQL Server Database with security:

```
Provider=SQLOLEDB;DataSource=AppServer01\SQLEXPRESS;Database=HRDatabase;User
ID=Username;Password=Password
```

SQL CLIENT:

- The **SQL Client** .NET provider connection syntax to an SQL Server Database:
  - The connection string handles the connection to the data source and it's syntax is as follows:

Data Source=*ServerName*;Database=DatabaseName;User ID=*Usename*;Password=*Password*

  - Example of a connection string that connects to an SQL Server Database:

```
DataSource=AppServer01;Database=HRDatabase;User ID=Username;Password=Password
```

  - Second syntax using "***Initial Catalog***" instead of "***Database***":

Data Source=*ServerName*;Database=Initial Catalog;User ID=*Username*;Password=*Password*

  - Example of a connection string using Initial Catalog:
  - Second example using "***Initial Catalog***" instead of "***Database***":

```
DataSource=AppServer01;Initial Catalog=HRDatabase;User ID=sa;Password=DBPassword
```

  - SQL SERVER EXPRESS The connection string syntax:

Data Source=*ServerNameSQLEXPRESS*;Database=DatabaseName;User ID=*Username*;Password=*Password***

  - Example of a connection string that connects to an SQL Server Database with security:

```
DataSource=AppServer01\SQLEXPRESS;Database=HRDatabase;User
ID=Username;Password=Password
```

II. WINDOWS AUTHENTICATION Connection String SYNTAX:

- The **VALUES** of the Connection String **SETTINGS** for **OLEDB SQL SERVER** are:

SETTING	DESCRIPTION	VALUE EXAMPLES
Provider	**SQLOLEDB provider Driver**	**SQLOLEDB**
Data Source	▪ Name of SERVER where database resides. ▪ If using SQL SERVER 2005 EXPRESS, syntax: ServerName\SQLEXPRESS	▪ AppServer01 ▪ AppServer01\SQLEXPRESS
Database (or Initial Catalog)	Name of DATABASE	HRDatabase
Integrated Security	▪ When **TRUE** or **SSPI**, Windows account credentials used for authentication (SSPI recommended) ▪ When FALSE, User ID & Password specified in connection string	SSPI

OLEDB CLIENT:

- The **OLEDB Client** .NET provider connection syntax for SQL Server WINDOWS AUTHENTICATAION:

  - Syntax for OLEDB connection string with Windows Authentication:

    Provider = *Provider*;Data Source=*ServerName*;Database=DatabaseName;Integrated Security=SecurityType

  - Example of an OLEDB connection string that connects to an SQL Server Database with Windows Authentication:

```
Provider=SQLOLEDB;Data Source=AppServer01;Database=HRDatabase;Integrated
Security=SSPI
```

  - Second syntax using "***Initial Catalog***" instead of "***Database***":

    Provider = *Provider_Name*;Data Source=*ServerName*;Database=Initial Catalog;Integrated Security=SecurityType

  - Example of a connection string using Initial Catalog:

```
Provider=SQLOLEDB;Data Source=AppServer01;Initial Catalog=HRDatabase;Integrated
Security=SSPI
```

  - **OLEDB Client SQL SERVER EXPRESS** connection string syntax:

    Provider = *Provider*;Data Source=*ServerNameSQLEXPRESS*;Database=DatabaseName;**
    Integrated Security=SecurityType

  - Example of a connection string that connects to an SQL Server Database with security:

```
"Provider=SQLOLEDB;DataSource=AppServer01\SQLEXPRESS;Database=HRDatabase;
Integrated Security=SSPI
```

SQL CLIENT:

- The **SQL Client** .NET provider connection syntax for SQL Server WINDOWS AUTHENTICATAION:

  - Syntax for SQL Client connection string with Windows Authentication:

    Data Source=*ServerName*;Database=DatabaseName;Integrated Security=SecurityType

  - Example of an OLEDB connection string that connects to an SQL Server Database with Windows Authentication:

```
Data Source=AppServer01;Database=HRDatabase;Integrated Security=SSPI
```

  - Second syntax using "***Initial Catalog***" instead of "***Database***":

    Data Source=*ServerName*;Database=Initial Catalog;Integrated Security=SecurityType

  - Example of a connection string using Initial Catalog:

```
Data Source=AppServer01;Initial Catalog=HRDatabase;Integrated Security=SSPI
```

  - **SQL Client SQL SERVER EXPRESS** connection string syntax:

    Data Source=*ServerNameSQLEXPRESS*;Database=DatabaseName;Integrated Security=SecurityType**

  - Example of a connection string that connects to an SQL Server Database with security:

```
DataSource=AppServer01\SQLEXPRESS;Database=HRDatabase; Integrated Security=SSPI
```

Step 3: Declare the Connection Object, Connection String & Opening the Connection

- In your code window, modules, routines create the connection string, connection object & assign the string to the object.

- Remember that the .NET Library comes equipped with a provider or library for OLE DB Client, SQL Client and Oracle Client version of this Class:

  - **OleDBClient – OleDBConnection**

  - **SQL Client – SQLConnection**

  - **Oracle Client – OracleConnection**

- My sample code will focus on OLE DB.

- ADO.NET provides several methods to implement assigning a ***connection string*** to a ***connection object***. I will list two methods:

Method I – Creating Connection Object using Default Constructor:

- In this method we will use the ***properties*** and ***methods*** in the CONNECTION Object
- We will perform the following steps:
  1. Create the connection string
  2. Create the Connection object using *Default Constructor*
  3. Use Connection Object ***ConnectionString*** property to assign the connection string to the Connection Object.
  4. Open the Connection
  5. **Perform Data Access HERE……Use other ADO.NET object to execute queries and retrieve data etc**.
  6. Close the connection
  7. Dispose of the connection
  8. Destroy the connection object

- Code is as follows:

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"

'Step 2-Create Connection object using Defalut Constructor
Dim objConn As New OleDbConnection()

'Step 3-Assign Connection String to Connection object ConnectionString Property
objConn.ConnectionString = strConn

'Step 4-Open the Connection
objConn.Open()

'Perform Data Access here, such as creating command, DataReader or DataSet etc.…….

'Step X-When finished with the Data Access, Close the Connection
objConn.Close()

'Step Y-It is a good idea to dispose or release all memory associated with the connection
object when finished.
objConn.Dispose()

'Step Z-Finally, destroy the object.
objConn = Nothing
```

Method II – Using the Parameterized Constructor:

- In this method we will simplify the code by using the **Connection Object's Parameterized Constructor**.
- The constructor contains the necessary code to automatically assign the Connection String to the *ConnectionString* Property.
- We perform the following steps:

1. Create the connection string
2. Create the Connection object, pass the ***Connection String*** as argument to the Constructor.
3. Open the Connection
4. Perform Data Access……Use other ADO.NET object to execute queries and retrieve data etc.
5. Close the connection
6. Dispose of the connection
7. Destroy the connection object

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()

'Perform Data Access here, such as creating command, DataReader or DataSet etc.…….

'Step X-When finished with the Data Access, Close the Connection
objConn.Close()

'Step Y-It is a good idea to dispose or release all memory associated with the connection
object when finished.
objConn.Dispose()

'Step Z-Finally, destroy the object.
objConn = Nothing
```

- Note that using this method we save one step!

Method III – (BEST PRACTICE!) Using the Exceptions:

- Now we demonstrate using Exception handling using Try-Catch block to trap errors generated by the **Connection Object**.
- We will enclose the connection open statement within Try-Catch blocks to trap for the possible connection errors:
- We perform the following steps:

1. Create the connection string
2. Create the Connection object, pass the ***Connection String*** as argument to the Constructor.
3. Begin trapping errors
4. Open the Connection
5. Catch connection related exceptions
6. Catch open connection exception
7. End trapping section
8. Perform Data Access……Use other ADO.NET object to execute queries and retrieve data etc.
9. Close the connection
10. Dispose of the connection
11. Destroy the connection object

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Begin Error Trapping via Try statement
Try
      'Step 4-Open the Connection
      objConn.Open()

'Perform Data Access here, such as creating command, DataReader or DataSet etc.…….

'Step 5-Trap for all connection related exceptions
Catch objOleDbExError As OleDbException
'Step 6-Handle error accordingly.  If this is within a class, you may have to
'raise or throw and exception, or from a form, simply display a message box Example:

      MessageBox.Show (objOleDbExError.Message)

'Step 7-Trap for the Open Connection exception
Catch objInvalidEx As InvalidOperationException
'Step 8-Handle error accordingly.  If this is within a class, you may have to
'raise or throw and exception, or from a form, simply display a message box Example:

      MessageBox.Show (objInvalidEx.Message)

'Step 9-End Error trapping
Finally
   'Step X-When finished with the Data Access, Close the Connection
     objConn.Close()

   'Step Y-It is a good idea to dispose or release all memory associated with the
   'connection object when finished.
     objConn.Dispose()

   'Step Z-Finally, destroy the object.
     objConn = Nothing

 End Try
```

Method IV – Testing for an open connection using State Property:

- You can also us the Connection Object **State** Property to verify if a connection is already open as follows:

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Begin Error Trapping via Try statement
Try

'Step 4-Open the Connection
If objConn.State <> ConnectionState.Open Then

      'Step 4-Open the Connection
      objConn.Open()

'Perform Data Access here, such as creating command, DataReader or DataSet etc…….

Else
      'Step 6-Handle accordingly.  Throw an exception if you are inside a class, call a
      message box etc. Example:

      MessageBox.Show ("Connection Already open")

End If

'Step 5-Trap for all connection related exceptions
Catch objOleDbExError As OleDbException
'Step 6-Handle error accordingly.  If this is within a class, you may have to
raise or throw and exception, or from a form, simply display a message box Example:

      MessageBox.Show (objOleDbExError.Message)

'Step 9-End Error trapping
Finally
   'Step X-When finished with the Data Access, Close the Connection
     objConn.Close()

   'Step Y-It is a good idea to dispose or release all memory associated with the
   'connection object when finished.
     objConn.Dispose()

   'Step Z-Finally, destroy the object.
     objConn = Nothing

 End Try
```

3.3 The ADO.NET Command Class

- The ADO.NET **Command** Class performs the following functions:

.NET Data Provider	*Description*
Command	▪ This object handles or executes the **queries** and **stored procedures**.

- As previously, we will list the properties and methods of this class.
- The .NET Library comes equipped with a Command Class for OLE DB Client, SQL Client and Oracle Client version of this Class:
  - **OleDBClient – OleDBCommand**
  - **SQL Client – SQLCommand**
  - **Oracle Client – OracleCommand**
- My sample code will focus on OLE DB.

Command Class – Properties & Methods

- Table below lists some important properties, methods and constructor of the **Command Class**:

Public Constructors

OleDbCommand Constructors: Default Constructor: **Public Sub New()** Parameterized Constructors passing Query String: **Public Sub New(ByVal *cmdText* As String)** Parameterized Constructors passing Query String & Connection Object: **Public Sub New(ByVal *cmdText* As String, ByVal *connection* As OleDbConnection)**	▪ Overloaded. Initializes or creates a new instance or Object of the **OleDbCommand** class. Example using default: `Dim objCmd As New OleDbCommand()` ▪ Parameterized constructor can pass a query string when creating an object of this class, so object already contains the query and we don't have to call it's property to set it. Example: `Dim objCmd As New OleDbCommand(strSQL)` ▪ Parameterized constructor can pass a query string & connection object, therefore the object already will have a connection when created. Example: `Dim objCmd As New OleDbCommand(strSQL, objConn)`

Public Properties

Property	Description
CommandText	▪ Gets or sets the SQL statement or stored procedure to execute at the data source. ▪ The ***SQL Query string*** or ***stored procedure*** name is assigned to this property
CommandType	▪ Gets or sets a value indicating how the **CommandText** property is interpreted. ▪ This property dictates whether the value of the **CommandText** is a text **SQL Statement** or a **Stored Procedure**. ▪ Value set by this property follow a special *enumerated* type defined within the ADO.NET library. The enumerated type is called **CommandType**. Example of three values of this type is: ▪ **CommandType.Text –** (Default Value) Indicates the SQL Statement or query is a full text In-Line SQL Statement ▪ **CommandType.StoredProcedure–** Indicates the SQL Statement or Query is a stored procedure is to be executed ▪ **CommandType.TableDirect–** Indicates the Query contains the NAME of the TABLE from which data will be retrieved. ▪ You can test the State Property of a Connection Object against these value to verify if a connection is open. For example: `'Executes an SQL Query` `objCom.CommandType = CommandType.Text` `'Executes a stored procedure` `objCom.CommandType = CommandType.StoredProcedure`
ConnectionString	Gets or sets the Connection object to be used by this instance of the **Command** object.
Parameters	▪ Gets the OleDbParameterCollection. ▪ This collection stores the unknown variables or parameters of a query. More on this in later section.
Transaction	Gets or sets the transaction in which the **OleDbCommand** executes.

Public Methods

Method	Description
ExecuteNonQuery	▪ Executes ACTION QUERIES or SQL statement, such as UPDATE, INSERT AND DELETE against the Connection ▪ Returns the number of **rows affected**.
ExecuteReader	▪ Executes SELECT queries to STORE IN DATAREADER OBJECT. ▪ Method internally builds an OleDbDataReader object and returns it. ▪ This method is only used when using a DataReader object to store the results of a query.
ExecuteScalar	Executes the query, and returns the first column of the first row in the result set returned by the query. Extra columns or rows are ignored.
Dispose	Releases the resources
Cancel	Attempts to cancel the execution of the **Command object**.

Using the Command Class Object

- Steps to add code to use the Command Object in your applications:

Step 1: Verify that the ADO.NET Provider and Data mechanism are imported into your code

- For **OLEDB Client** Provider type, import the library:

```
Imports System.Data
Imports System.Data.OleDb  'OLEDB Provider
```

- For **SQL CLIENT** Provider type, import the library:

```
Imports System.Data
Imports System.Data.SQLClient SQL Data Provider
```

Pre-Step 2: Preparing the SQL String

- The SQL String contains the **Query** or **Stored Procedure** to be executed.

IN-LINE SQL Query:

- In-line SQL refers to actual SQL Queries embedded within your code.
- The complete SQL query string is compiled and part of your program. For example:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = 111"
```

- IN-LINE have the following advantages and disadvantages:

Advantages	Disadvantages
▪ Queries are easier to implement ▪ SQL Statement resides within your application ▪ No need to learn any specific Database Management System special languages, such as Store Procedures etc. ▪ Connection string can be kept in an external file, so changes can only be made in one location.	▪ Creating a complex query can result in errors building the actual string in VB.NET due to the concatenation of the query, commas, punctuation, special characters etc (, & “”) ▪ Bad for performance. Complete SQL statement is sent to the Database via the network. ▪ More processing in client, this closer to FAT-CLIENT systems. ▪ POOR FOR SECURITY. ▪ NOT BEST PRACTICE!!!!!!

- In this pre-step, I will show you how to create your IN-LINE SQL string.
- You need to create the SQL query correctly. Since you are using VB code, you need to create it as a string. In addition you will need to use concatenation symbols such as & to create your query.
- Lets look at the following examples:

Example 1:

- Supposed you want to execute the following query with the literal value being an INTEGER assigned in the WHERE clause:

```
SELECT * FROM Customer WHERE Customer_ID = 111
```

- You need to simply assign this string to a variable as follows:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = 111"
```

Example 2:

- Now the value assigned in the WHERE Clause is an INTEGER value but within a parameter contained within a variable say intID, then you will need to create the string using the PARAMETER as follows:

```
'Assume somewhere in your program the following statement is made:
Dim intID As Integer = 111


'Else where the In-line query string is being created using a parameter
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & intID
```

- Creating this string in VB is pretty easy with no issues since inside the variable strSQL, the complete string looks as follows:

```
"SELECT * FROM Customer WHERE Customer_ID = 111"
```

Example 3:

- Supposed now that the literal value being assigned to the WHERE Clause is a *string*. This changes things since as you know you need to enclose string within quotes or (‘) or double quotes depending on the database. Lets look at the query we want to execute assuming Microsoft Access Database:

```
SELECT * FROM Customer WHERE Customer_ID = '111'
```

- You need to create the string as follows:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = '111'"
```

- Again this was a simple string to build in VB

Example 4:

- Supposed now that the value being assigned to the WHERE Clause is a *string* within a variable. The value is stored in a String Variable say ***strID*** or a TEXTBOX etc and can be unknown. Now things get a bit complicated.
- You will need to use the **&** operator to build the string including the characters (double quotes (“)) required to enclosed the string value. The query will look as follows:

```
'Assume somewhere in your program
'The variable is assigned the following string:
Dim strID As String = "111"


Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = '" & strID & "'"
```

- Note that we needed to build our string and take into account the (') character required to enclosed any string in a Microsoft Access or SQL Server Query
- Building these kinds of strings can become very complex and prone to syntax errors.

❖ **Note that using the Parameter Object of ADO.NET Provider will eliminate having to create these complex VB.NET String code to build our queries. The Parameter object takes care of these details**

Step 3: Declare the Command Object, Create Query String, and Assign Connection Object

- Now we proceed by creating a **Command Object** to handle our Query and execution.
- Again ADO.NET provides several methods to implement this and I will list three methods:

Creating and Preparing the Command Object

Method I – Creating Command Object using Default Constructor

- In this method we will use the properties and methods in the COMMAND Object
  1. Create the SQL query string
  2. Create the Command Object using *Default Constructor*
  3. Then use Connection Object ***CommandText*** property to assign the SQL string to the Command Object.
  4. Assign the previously created connection object to the command object.

```
'Step 1-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & intID

'Step 2-Create Command object
Dim objCmd As New OleDbCommand()

'Step 3-Assign SQL string to CommandObject.CommandText Property
objCmd.CommandText = strSQL

'Step 4-Assign the Connection Object
objCmd.Connection = objConn

'Step X-Execute the query.
 'This step will be shown in sections to follow.

'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

Method II – Creating Command Object Using Parameterized Constructor:

- In this method we will simplify the code by using the Command Object's Parameterized Constructor assign the SQL string.
- The constructor contains the necessary code to automatically assign the SQL string to it's *CommandText* Property. Steps:
  1. Create the SQL query string
  2. Create the **Command Object**; pass the ***SQL string*** as arguments to the **Constructor**.

```
'Step 1-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & intID

'Step 2-Create Command object, pass string as arguments
Dim objCmd As New OleDbCommand(strSQL)

'Step 3-Assign the Connection Object
objCmd.Connection = objConn

'Step X-Execute the query.
 'This step will be shown in sections to follow.

'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

- Note that using this method we eliminate one step.

Method III – Creating Command Object Using Parameterized Constructor for Query String & Connection object:

- In this method we will simplify the code by using the Command Object's Constructor assign the SQL string and the connection object.
- The constructor contains the necessary code to automatically assign the Connection Object and the SQL string.
- We perform the following steps:

1. Create the SQL query string
2. Create the **Command** Object; pass the *SQL string* and Connection Object as arguments to the **Constructor**.

```
'Step 1-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & intID

'Step 2-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)

'Step X-Execute the query.
 'This step will be shown in sections to follow.

'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

- Note that using this method we save **two** steps.

Executing Queries and Storing Results in a DATAREADER

- Executing a query will depend on where you want the query results to be placed.
- As you have learned, the results can be placed in a ***DataReader*** Object, ***DataSet*** or the ***DataTable*** Object via the ***DataAdapter*** Object.
- In this section we will concentrate on the DATAREADER Object. The *DataSet* and *DataTable* options will be discussed in later lecture.
- Depending on which object will manage the data, you would use on of the three FUNCTION Methods for executing queries: **ExecuteReader()**, **ExecuteNonQuery()** and **Execute Scalar()**.

  - Note that in order to explain these methods, we need to create other objects such as DataReader object. The details to the DataReader Object will be explained in it's own section, for now we will get a preview of using the DataReader object

Executing the Query and Storing results in the DataReader Object.

- If you are going to use a **DataReader** Object to store the results, than you would use the ***ExecuteReader()*** Method.
- Steps:

Method I:

- In this method we will execute the query using the COMMAND Object methods and assign the result to a DATAREADER Object:
  1. Because the result of the ***ExecuteReader()*** FUNCTION returns a DataReader POINTER, create a ***reference*** to a DATAREADER Object. Note that we only create a reference. The keyword **NEW is NOT USED**, this is a **POINTER ONLY**!
  2. Execute the query using the Command Object ***ExecutreReader()*** method and assign the results to the DATAREADER Object. Note that the *ExecuteReader()* method returns a *DataReader* Object, that is why you need to pointer reference.

```
'Step 1-Create DATAREADER object
Dim objDR As OleDbDataReader

'Step 2-Execute query and assign results to DataReader object
objDR = objCmd.ExecuteReader
```

Method II:

- Of course we can do this all in one step to save some code:
  1. Create a ***reference*** to a DATAREADER Object assign the execution of the query using the Command Object ***ExecutreReader()*** method.

```
'Step 1-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

- ❖ Now you need to extract the data stored in the DataReader Object. We will see this when explaining the DataReader Object.

Executing a Non-Row Returning Query (Action Queries)

- ❑ Non-Row returning queries or action queries are queries that modify the database, such as **UPDATE**, **INSERT** & **DELETE** queries.
- ❑ Examples of these queries are:
  - Delete Query with hard-coded values:

```
Dim strSQL As String = "DELETE FROM Customers WHERE Customer_ID = 111"
```

  - Queries with variables containing integers:

```
Dim strSQL As String = "DELETE FROM Customers WHERE Customer_ID = " & intID
```

- ❑ The COMMAND Object contains a method named ***ExecuteNonQuery()*** that you will use for these types of queries.
- ❑ The ***ExecuteNonQuery()*** FUNCTION returns the number of rows affected by the action.
- ❑ You can create a variable to store this data and use it as you see fit.
  1. Create a variable to store the return integer number of rows affected by the action query.
  2. Execute the query using the Command Object ***ExecutreNoQuery()*** method.

```
'Step 1-Create variable to store number of rows affected
Dim intRowsAffected As Integer
```

```
'Step 2-Execute query and assign results to DataReader object
intRowsAffected = objCmd.ExecuteNonQuery
```

- ❖ You can test the variable storing the number of rows affected to verify that the correct rows were modified.

3.4 The ADO.NET DataReader Class

- The ADO.NET **DataReader** Class performs the following functions:

.NET Data Provider	*Description*
DataReader	▪ This object has similar functionality as the ***DataSet*** objects and that is it stores the result of a query. ▪ Except this Object is a Forward-Only, Read-Only stream of data that directly communicate with the database. ▪ **Read-Only –** You can only read data, no updates or modification is supported ▪ **Forward-Only –** You can examine the results of a query one row at a time, when you move forward to the next row the contents of the previous row are discarded. ▪ Supports minimal features, but is fast and lightweight.

- As previously, we will list the properties and methods of this class.
- Again the .NET Library comes equipped with a provider or library for SQL Server and OLE DB:
- The .NET Library comes equipped with a Command Class for OLE DB Client, SQL Client and Oracle Client:

  - **OleDBClient – OleDBDataReader**
  - **SQL Client – SQLDataReader**
  - **Oracle Client – OracleDataReader**

- My sample code will focus on OLE DB.

DataReader Class – Properties & Methods

- Table below lists some important properties, methods of the **DataReader Class**. Note that the DataReader Class has NO Constructor since we only create a reference of it not an object:

Public Properties

Property	Description
Item Getting data from column by index: **Overloads Public Property Item(ByVal** *index* **As** Integer **) As** Object Getting data from column by String or Column name: `Overloads Public Property Item(ByVal name As String) As Object`	▪ Gets the value of a column in the result set ▪ You use this property to get the data resulted from the query and stored in the ***DataReader*** object. The data is retrieved by column. Each call to this property will return the data stored in the cell of the column specified ▪ You can specified the column by index, 0, 1, 2 etc. Example extracting the data in the first column: `intIDNumber = objDR.Item(0)` ▪ You can specified the column by string using the name of the column. Example extracting the data in the first column: `intIDNumber = objDR.Item("Customer_ID")`
HasRows	▪ Returns a Boolean value (True/False) indicating whether the DataReader Objects contains one or more rows. ▪ True indicates there are rows, false indicates empty or no rows returned.
FieldCount	▪ Gets the number of columns in the current row
RecordsAffected	▪ Gets the number of rows changed, inserted, or deleted by execution of the SQL statement.

Basic Public Methods

Method	Description
Read	▪ Advances the **OleDbDataReader** to the next record. ▪ You use this method to navigate from record to record within the DataReader Object
NextResult	▪ Used when executing multiple queries. Multiple or batch SQL Statements return multiple tables into a **DataReader**. Advances the data reader to the next result or table.
Close	Close the **DataReader** Object.

Special Public Methods to Read Native Data Types from DataReader

- The ITEM PROPERTY is used to extract data from the DataReader as is. We don't know what data is being returned. The DATAREADER class provides specific method to retrieve the data in the specific native mode of the database. The table below lists these GETXXX() methods that return the data in a specific data type. Note that NO DATA CONVERTION IS DONE BY THESE METHODS SO, DON'T GET CONFUSED AND THINK IT CONVERTS THE DATA. They simply return it as the native type of the data in database and gives better performance internally. THE VARIABLE WAITING FOR THE DATA HAS TO BE IN THE CORRECT TYPE. CONVERTION MAY BE REQUIRED TO MATCH THE TARGET DATA TYPE
- Examples:

```
txtAge.Text = CStr(objDR.GetInt32(5)) 'Integer data being returned, but must
                                      'be converted to string for text box

txtBirthDate.Text = CStr(objDR.GetDateTime(2)) 'DateTime returned, converted
                                               'to string for text box
```

Method	Description
GetBoolean	Gets the value of the specified column **INDEX** as a Boolean.
GetByte	Gets the value of the specified column **INDEX** as a byte.
GetChar	Gets the value of the specified column **INDEX** as a character.
GetDateTime	Gets the value of the specified column **INDEX** as a DateTime object.
GetDecimal	Gets the value of the specified column **INDEX** as a Decimal object.
GetDouble	Gets the value of the specified column **INDEX** as a double-precision floating point number.
GetFloat	Gets the value of the specified column **INDEX** as a single-precision floating point number.
GetGuid	Gets the value of the specified column **INDEX** as a globally-unique identifier (GUID).
GetInt16	Gets the value of the specified column **INDEX** as a 16-bit signed integer.
GetInt32	Gets the value of the specified column **INDEX** as a 32-bit signed integer.
GetInt64	Gets the value of the specified column **INDEX** as a 64-bit signed integer.
GetName	Given the INDEX of a column, it Gets the name of the specified column.
GetOrdinal	Gets the column INDEX, given the name of the column.
GetString	Gets the value of the specified column **INDEX** as a string.

Using the DataReader Class Object

- We got a taste of how to use the ***DataReader*** object in the previous section. The **DataReader** Object works in conjunction with the **Command** Object as we saw in the previous section.
- The job of the **DataReader** object is to store the results of a query. As you recall, the results of an SQL query is a table. Therefore after execution of a query, the data stored inside the **DataReader** object is in the form of a table.
- Using the ***DataReader*** Object involves the following:
  - Store the results of the execution of a query carried out by the **Command** Object
  - Navigate from record to record, extracting the data via the **Read()** method and **Item** Property.
  - Closing the Reader when done.

Review of Steps to Execute the Query and Assign Results to DataReader

- Steps to add code to use the **DataReader** Object in your applications are as follows:

Step 1: Verify that the ADO.NET Provider and Data mechanism are imported into your code

```
Imports System.Data
Imports System.Data.OleDb  'OLEDB Provider
```

Step 2: Create Connection Object & Command Object & associated properties and methods

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

```
'Step 4-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & intID

'Step 5-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

Step 3: Declare the DataReader POINTER, and Store data from a query execution via Command Object

- There are two methods to declaring and using the ***DataReader*** object to store the result of a query:

Method I:

- In this method we will execute the query using the COMMAND Object methods and assign the result to a DATAREADER Object:
  1. The ***ExecutreReader()*** function returns a DATAREADER POINTER. Therefore, create a ***reference*** or ***POINTER*** to a DATAREADER Object. Note that we only create a reference. The keyword **NEW is NOT** used because we DON'T CREATE A DATAREADER OBJECT but a DATAREADER POINTER!
  2. Execute the query using the Command Object ***ExecutreReader()*** function and assign the results to the DATAREADER Object. Note that the *ExecuteReader()* method returns a *DataReader* Object.

```
'Step 6-Declare DATAREADER object Reference Only
Dim objDR As OleDbDataReader
```

```
'Step 7-Execute query and assign results to DataReader object
objDR = objCmd.ExecuteReader
```

Method II:

- Of course we can do this all in one step to save some code:
  1. Create a ***reference*** to a DATAREADER Object assign the execution of the query using the Command Object ***ExecutreReader()*** method.

```
'Step 6-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

Extracting Data From DataReader

Step 4: Extracting the Data from the DataReader Object

- Now that we have the data inside the **DataReader** Object in the form of a table. We need to extract it.
- To do this, we need the combination of the following:
  1. **HasRows** property to determine if the query returned data
  2. **Read()** Method to navigate from row to row
  3. Now we have a choice of Properties of Methods to get the data out of the DataReader:
     i. **Item property** to extract the data within a row, by column.
        - Option 1 – Takes Column NAME as argument and returns the data
        - Option 2 (BEST PRACTICE) – Takes the Column INDEX as argument and returns the data. This method has better performance and recommended when using the **Item** Property

     BEST PRACTICE!

     ii. **GetXXX(INDEX) methods** to extract the data within a row, by column **INDEX**
        - Takes and **INDEX** as argument and returns the native data type of the data

Example of Query that returns one Record Only

- The following example is for a query returns only one record.
- We will show two options, one using column names, the other using column index to extract the data

Method I: Using Column Names

- In this method we will follow the steps listed above, but we will use the name of the columns on the resulting table to extract the data. when using the Item property:
  1. We create the ***DataReader*** Object and execute the query.
  2. Test to verify that there are records to extract
  3. Navigate through the records using Read() method and extract the data using the Item property.
  4. Close the ***DataReader***
  5. Dispose of the ***DataReader***

```
'Step 1-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
'Step 2-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

    'Step 3-Call Read() Method to point and read the first record
    objDR.Read()

    'Step 3b-Extract data from a row.  Use Item property to get the column data
    'Note that the name of the column of the database table is used
    intIDNumber = objDR.Item("Customer_ID")
    strName = objDR.Item("Name")
    dBirthDate = objDR.Item("BirthDate")
    strAddress = objDR.Item("Address")
    strPhone = objDR.Item("PhoneNumber")
```

```
      'Step 3c-Now that you have the data for a record or row, do what you want
      'with the data, pass it to methods, display it, assign it etc.
      'Process(intIDNumber, strName, dBirthDate, strAddress, strPhone)

Else
      MessageBox.Show("No Customers found")

End If
```

```
'Step 4-Close object.
objDR.Close()

'Step 5-Destroy the object.
objDR = Nothing
```

```
'Step XYZ-You need to close and Dispose of the connection and command objects.
'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

Method II: Using Index numbers (BEST PRACTICE) FASTER PERFORMANCE!

- In this method the steps are the same, but we use a more efficient way of extracting the data using the **Item** property. Instead of using the *names* of the columns on the resulting table, we use an **Index** from 0 to ***objRec.FieldCount* – 1** property.
- Using index instead of the column names is actually more efficient and offers better for performance.
  1. We create the DataReader Object and execute the query.
  2. Test to verify that there are records to extract
  3. Navigate through the records using Read() method and extract the data using the Item property.
  4. Close the DataReader
  5. Dispose of the DataReader

```
'Step 1-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
'Step 2-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

'Step 3-Navigate through the DataReader Object by reading row-by-row

      'Step 3-Call Read() Method to point and read the first record
      objDR.Read()

      'Step 3b-Extract data from a row.  Use Item property to get the column data
      'Note that we are using index numbers starting from 0 for each column.
      intIDNumber = objDR.Item(0)
      strName = objDR.Item(1)
      dBirthDate = objDR.Item(2)
      strAddress = objDR.Item(3)
      strPhone = objDR.Item(4)

      'Step 3c-Now that you have the data for a record or row, do what you want
      'with the data, pass it to methods, display it, assign it etc.
      'Process(intIDNumber, strName, dBirthDate, strAddress, strPhone)
Else
      MessageBox.Show("No Customers found")

End If
```

```
'Step 4-Close object.
objDR.Close()

'Step 5-Destroy the object.
objDR = Nothing
```

```
'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

Method IIi: Using Index numbers with OPTION STRICT ON!

- If Option Strict = ON, then we need to convert t the data type of the data being return from the database:

```
'Step 1-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
'Step 2-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

'Step 3-Navigate through the DataReader Object by reading row-by-row

      'Step 3-Call Read() Method to point and read the first record
      objDR.Read()

      'Step 3b-Extract data from a row.  Use Item property to get the column data
      'Note that we are using index numbers starting from 0 for each column.
      intIDNumber = CInt(objDR.Item(0))
      strName = CStr(objDR.Item(1))
      dBirthDate = CDate(objDR.Item(2))
      strAddress = CStr(objDR.Item(3))
      strPhone = CStr(objDR.Item(4))

      'Step 3c-Now that you have the data for a record or row, do what you want
      'with the data, pass it to methods, display it, assign it etc.
      'Process(intIDNumber, strName, dBirthDate, strAddress, strPhone)
Else
      MessageBox.Show("No Customers found")

End If
```

```
'Step 4-Close object.
objDR.Close()

'Step 5-Destroy the object.
objDR = Nothing
```

```
'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

Method II: Using GETXXX() FUNCTIONS (BEST PRACTICE & RECOMMENDED)

- Using the ITEM PROPERTY has the disadvantage that we need to CONVERT THE DATATYPE USING data conversion methods such as CStr(), CInt() etc.
- The DataReader provides GETXXX() methods that return the native data types already converted.
- As with the Item Property, we use an **Index** from 0 to ***objRec.FieldCount* – 1** as argument to the Methods.
- The steps are the same as the previous examples, but this time we use the GETXXX() methods to extract data.

```
'Step 1-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
'Step 2-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

'Step 3-Navigate through the DataReader Object by reading row-by-row

      'Step 3-Call Read() Method to point and read the first record
      objDR.Read()

      'Step 3b-Extract data from a row.  Use Item property to get the column data
      'Note that we are using index numbers starting from 0 for each column.
      intIDNumber = objDR.GetValue(0) 'Must use GetValue for MS Access Number type
      strName = objDR.GetString(1)
      dBirthDate = objDR.GetDateTime(2)
      strAddress = objDR.GetString(3)
      strPhone = objDR.GetString(4)

      'Step 3c-Now that you have the data for a record or row, do what you want
      'with the data, pass it to methods, display it, assign it etc.
      'Process(intIDNumber, strName, dBirthDate, strAddress, strPhone)
Else
      MessageBox.Show("No Customers found")

End If
```

```
'Step 4-Close object.
objDR.Close()

'Step 5-Destroy the object.
objDR = Nothing
```

```
'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

Example of Query that returns many Records

- The following example is for a query returns ONE or MORE records.
- The results or table is store in the DataReader
- In this case we need to navigate through each of the records stored in the **DataReader**.
- Again we will show examples of using *column names*, *column index* or *GETXXX()* functions to extract the data

Method I: Using Column Names

- In this method we will follow the steps listed above, but we will use the name of the columns on the resulting table to extract the data. when using the Item property:

```
'Step 1-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader

'Step 2-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

'Step 3-Navigate through the DataReader Object calling Read() method and read row-by-row
      While objDR.Read()

            'Step 3b-Extract data from a row.  Use Item property to get the column data
            'Note that the name of the column of the database table is used
            intIDNumber = CInt(objDR.Item("Customer_ID"))
            strName = CStr(objDR.Item("Name"))
            dBirthDate = CDate(objDR.Item("BirthDate"))
            strAddress = CStr(objDR.Item("Address"))
            strPhone = CStr(objDR.Item("PhoneNumber"))


            'Step 3c-Now that you have the data for a record or row, do what you want
            'with the data, pass it to methods, display it, assign it etc.
             MessageBox.Show(intIDNumber & strName & dBirthDate & strAddress & strPhone)


      End While
Else
      MessageBox.Show("No Customers found")
End If

'Step XYZ-You need to close and Dispose of the connection and command objects.
'Step 4-Close object.
objDR.Close()

'Step 5-Destroy the object.
objDR = Nothing

'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

Method II: Using Index numbers

- In this example again we use the INDEX number as argument to the **Item** property. Instead of using the *names* of the columns.

```
'Step 1-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader

'Step 2-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

'Step 3-Navigate through the DataReader Object by reading row-by-row
      While objDR.Read()

            'Step 3b-Extract data from a row.  Use Item Property to get the column data
            'Note that the name of the column of the database table is used
             intIDNumber = CInt(objDR.Item(0))
             strName = CStr(objDR.Item(1))
             dBirthDate = CDate(objDR.Item(2))
             strAddress = CStr(objDR.Item(3))
             strPhone = CStr(objDR.Item(4))

            'Step 3c-Now that you have the data for a record or row, do what you want
            'with the data, pass it to methods, display it, assign it etc.
             MessageBox.Show(intIDNumber & strName & dBirthDate & strAddress & strPhone)

      End While

Else
      MessageBox.Show("No Customers found")

End If

'Step XYZ-You need to close and Dispose of the connection and command objects.
'Step 4-Close object.
objDR.Close()

'Step 5-Destroy the object.
objDR = Nothing

'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

Method III: Using GETXXX() FUNCTIONS

- In this example again we use the GETXXX() functions provided by the DataReader using INDEX number as argument to return the data from a record:

```
'Step 1-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader

'Step 2-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

'Step 3-Navigate through the DataReader Object by reading row-by-row
      While objDR.Read()

            'Step 3b-Extract data from a row.  Use Item Property to get the column data
            'Note that the name of the column of the database table is used
            intIDNumber = objDR.GetValue(0) 'Must use GetValue for MS Access Number type
            strName = objDR.GetString(1)
            dBirthDate = objDR.GetDateTime(2)
            strAddress = objDR.GetString(3)
            strPhone = objDR.GetString(4)

            'Step 3c-Now that you have the data for a record or row, do what you want
            'with the data, pass it to methods, display it, assign it etc.
             MessageBox.Show(intIDNumber & strName & dBirthDate & strAddress & strPhone)

      End While

Else
      MessageBox.Show("No Customers found")

End If

'Step XYZ-You need to close and Dispose of the connection and command objects.
'Step 4-Close object.
objDR.Close()

'Step 5-Destroy the object.
objDR = Nothing

'Step Y-It is a good idea to dispose or release all memory associated with the object.
objCmd.Dispose()

'Step Z-Finally, destroy the object.
objCmd = Nothing
```

Summary of Data Access Code – Query Without Using Paramters (MS ACCESS DATABASE)

- At this point, we have all the data access code we need to execute a query without a parameter
- These are queries of the following types:
  - Query with hard-coded values:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = 111"
```

  - Queries with variables containing integers:

```
'Else where the In-line query string is being created using a parameter
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & intID
```

  - Queries with hard-coded value strings:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = '111'"
```

  - Queries with variables containing strings B

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = '" & strID & "'"
```

Data Access Query without Exception Handling

- The data access code with NO Try-Catch:

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

```
'Step 4-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & txtIDNum

'Step 5-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
'Step 6-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
'Step 7-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

    'Step 8a-Call Read() Method to point and read the first record
    objDR.Read()

    'Step 8b-Extract data from a row.  Use Item method to get the column data
    intIDNumber = CInt(objDR.Item(0))
    strName = CStr(objDR.Item(1))
    dBirthDate = CDate(objDR.Item(2))
    strAddress = CStr(objDR.Item(3))
    strPhone = CStr(objDR.Item(4))

    'Step 8c-Now that you have the data for a record or row, do what you want
    MessageBox.Show(intIDNumber & strName & dBirthDate & strAddress & strPhone)
```

```
Else
      'Step 9-No data returned, Record not found.  Do what you want here!
      MessageBox.Show("No Customers found")

End If
```

```
'Step 10-Teminate Command Object
objCmd.Dispose()
objCmd = Nothing
'Step 11- Teminate DataReader Object.
objDR.Close()
objDR = Nothing
'Step 12-Terminate the Connection Object
objConn.Close()
objConn.Dispose()
objConn = Nothing
```

Data Access Query with TRY-CATCH Exception Handling (BEST PRACTICE)

- The data access code with Try-Catch:

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)
```

```
'Step 3-Begin Error Trapping via Try statement
Try
      'Step 4-Open the Connection
       objConn.Open()

       'Step 5-Create SQL string.  Variable txtIDNum contains the ID
       Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & txtIDNum

       'Step 6-Create Command object, pass string and connection object arguments
       Dim objCmd As New OleDbCommand(strSQL, objConn)

       'Step 7-Create DATAREADER object & Execute Query
       Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
       'Step 8-Test to make sure there is data in the DataReader Object
       If objDR.HasRows Then

          'Step 8a-Call Read() Method to point and read first record
           objDR.Read()

           'Step 8b-Extract data from a row.
         intIDNumber = CInt(objDR.Item(0))
         strName = CStr(objDR.Item(1))
         dBirthDate = CDate(objDR.Item(2))
         strAddress = CStr(objDR.Item(3))
         strPhone = CStr(objDR.Item(4))

           'Step 8c- you have the data for a record or row, do what you want
           MessageBox.Show(intIDNumber & strName & dBirthDate & strAddress & strPhone)
```

```
        Else
             'Step 9-No data returned, Record not found.  Do what you want here!
             MessageBox.Show("No Customers found")

        End If

        'Step 10-Teminate Command Object
        objCmd.Dispose()
        objCmd = Nothing
        'Step 11- Teminate DataReader Object.
        objDR.Close()
        objDR = Nothing

'Step 12-Trap for all connection related exceptions
Catch objOleDbExError As OleDbException
'Step 13-Handle error accordingly.  If this is within a class, you may have to
'throw and exception, or from a form, display a message box etc. Example:

        MessageBox.Show(objOleDbExError.Message)

             'Step 14-Trap for the Open Connection exception
Catch objInvalidEx As InvalidOperationException
'Step 15-Handle error accordingly.  If this is within a class, you may have to
'throw and exception, or from a form, simply display a message box Example:

       MessageBox.Show(objInvalidEx.Message)

             'Step 16-End Error trapping
Finally
    'Step 17-Terminate the Connection Object
    objConn.Close()
    objConn.Dispose()
    objConn = Nothing
End Try
```

Summary of Data Access Code For Action Queries Without Using Paramters (MS ACCESS)

- Now let's summarize the code required to execute an action query. These are UPDATE, DELETE & INSERT queries.
- These queries return no value, nevertheless, the ADA.NET Command Object method will return the number of ROWS AFFECTED.
- Examples of these queries are:

  - Delete Query with hard-coded values:

```
Dim strSQL As String = "DELETE FROM Customers WHERE Customer_ID = 111"
```

  - Queries with variables containing integers:

```
Dim strSQL As String = "DELETE FROM Customers WHERE Customer_ID = " & intID
```

Action Query with Try-Catch

- The data access code with Try-Catch:

```
'Step 1-Create Connection, assign Connection to string & open it
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"
Dim objConn As New OleDbConnection(strConn)
```

```
'Step 2-Begin Error Trapping via Try statement
Try
```

```
    objConn.Open()

    'Step 3-Create Command, Query, assing query, and assign connection
    Dim strSQL As String = "DELETE FROM Customers WHERE Customer_ID = " & strIDNum

    'Step 4-Create Command object, pass string and connection object arguments
    Dim objCmd As New OleDbCommand(strSQL, objConn)

     'Step 5-Execute Non-Row Query Test result and throw exception if failed
     Dim intRecordsAffected As Long = objCmd.ExecuteNonQuery()
     If intRecordsAffected <> 1 Then
        'Step 6-take appropriate action
         MessageBox.Show("Error INSERTING Record")
     End If

     'Step X-Terminate Command Object
     objCmd.Dispose()
     objCmd = Nothing
```

```
'Step 7-Trap for all connection related exceptions
Catch objOleDbExError As OleDbException
'Step 13-Handle error accordingly.  If this is within a class
'throw and exception, or from a form, simply display a message box Example:

      essageBox.Show(objOleDbExError.Message)

            'Step 8-Trap for the Open Connection exception
Catch objInvalidEx As InvalidOperationException
'Step 9-Handle error accordingly.  If this is within a class
'throw and exception, or from a form, simply display a message box Example:
      MessageBox.Show(objInvalidEx.Message)
```

```
'Step 10-End Error trapping
Finally

      'Step Y-Terminate the Connection Object
      objConn.Close()
      objConn.Dispose()
      objConn = Nothing

End Try
```

3.5 The ADO.NET Parameters Class

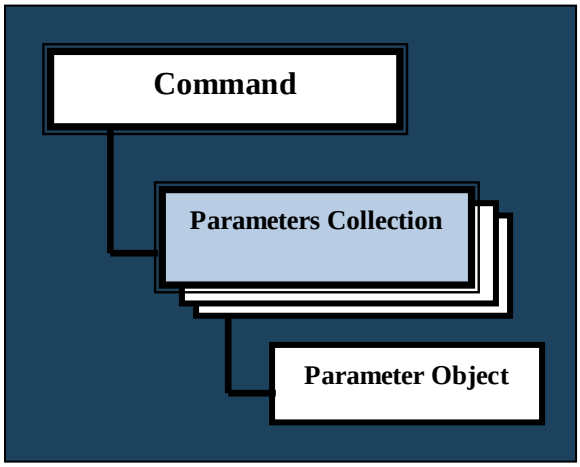
- The ADO.NET **Parameters Collection** Class resides inside the **Command Object** and performs the following functions:

.NET Data Provider	*Description*
Parameter Collection	▪ This Collection Object resides inside the Command Object. It is a child object of the Command Object ▪ This Collection stores object of type ***Parameter***. Each parameter object represents and stores a parameter to be passed to queries
Parameter	▪ These are the objects store by the ParameterCollection ▪ Object of this type are used to store QUERY PARAMTERS. Parameters are the variables or values used in parameterized queries. For example: **SELECT * FROM Customer WHERE Customer_ID = *@CustomerID*** - Here the ***@CustomerID*** represents a value that can be passed from a Form or variable etc. ▪ Each Parameter Object represents a parameter, which is passed into the query.

- Again, remember that the .NET Library comes equipped with a provider or library for SQL Server and OLE DB, so we will have the following classes available to us depending on which database we want to use:

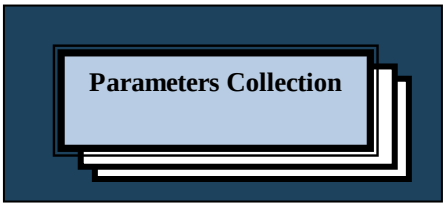
  - **OleDBClient – OleDBParameterCollection & OleDBParameter**
  - **SQL Client – SQLParameterCollection & SQLParameter**
  - **Oracle Client – OracleParameterCollection & OracleParameter**

- Object Model structure:



OLEDBParameter Collection Class – Properties & Methods

- The **OleDBParemterCollection Class** is a collection class of the ILIST Collection type or INDEX based collection. This is similar to the ARRAYLIST Collection:



- This collection stores objects of the ***OleDBParameter*** Class.
- Table below lists some important properties, methods of the **OleDBParemterCollection Class**:

Public Properties

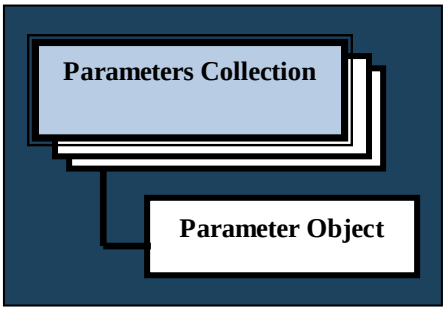
Property	Description
Item	▪ Gets or sets the **OleDbParameter** with a specified attribute ▪
Count	▪ Gets the number of **OleDbParameter** objects in the collection.

Public Methods

Method	Description
Add **Overloaded version of ADD, takes Name & Data Type:** **Overloads Public Function Add(String, OleDbType) As OleDbParameter** **Overloaded version of ADD, takes Name, Data Type & Size:** **Overloads Public Function Add(String, OleDbType, Integer) As OleDbParameter**	▪ Adds an **OleDbParameter** to the **OleDbParameterCollection**. ▪ Example of the first OVERLOADED VERSIONS OF ADD: `objCmd.Parameters.Add("@Customer_ID", OleDbType.Integer)` ▪ Example of the second OVERLOADED VERSIONS OF ADD: `objCmd.Parameters.Add("@Customer_ID", OleDbType.Integer,20)`
Remove	▪ Removes the specified OleDbParameter from the collection
Contains	▪ Gets a value indicating whether an OleDbParameter exists in the collection
Clear	▪ Removes all items from the collection.

OLEDBParameter Class – Properties & Methods

- Tthe **OleDBParemeter Class** is a child of **OleDBParemterCollection**. Objects of this class contain the storage mechanism (Properties) to store parameters passed to queries. Each Parameter Object represents one query parameter:



- Table below lists some important properties & methods of the **OleDBParemeter Class**:

Public Properties

Property	Description
OleDBType	▪ Gets or sets the OleDbType or DATA Type of the parameter. IMPORTANT PROPERTY! ▪ This property takes values of a specialize ENUMERATED TYPE. ▪ SEE TABLE BELOW FOR A LISTING OF THE AVAILABLE ENUMERATED DATA TYPE FORMATS used by this PROPERTY.
ParamterName	▪ Gets or sets the name of the OleDbParameter
Value	▪ Gets or sets the value of the parameter. IMPORTANT PROPERTY
Size	▪ Gets or sets the maximum size, in bytes, of the data within the column

Public Methods

Method	Description
For our DATA ACCESS NEEDS, we don't need at this time the methods provided by this class.	

OleDBType Enumerated Type Members

- The OleDBType Property GETs or SETs the Data Type of the Parameter.
- This Property contains an Enumerated Data Structure storing a list of Data Type identifiers
- The Data Type is based on the following table:

Members

Member name	Description
Binary	A stream of binary data (DBTYPE_BYTES). This maps to an Array of type Byte.
Boolean	A Boolean value (DBTYPE_BOOL). This maps to Boolean.
Char	A character string (DBTYPE_STR). This maps to String.
Currency	A currency value ranging from -2^{63} (or -922,337,203,685,477.5808) to 2^{63} -1 (or +922,337,203,685,477.5807). This maps to Decimal.
Date	Date data, stored as a double (DBTYPE_DATE). This maps to DateTime.
Decimal	A fixed precision and scale numeric value between -10^{38} -1 and 10^{38} -1 (DBTYPE_DECIMAL). This maps to Decimal.
Double	A floating point number within the range of -1.79E +308 through 1.79E +308 (DBTYPE_R8). This maps to Double.
Guid	A globally unique identifier (or GUID) (DBTYPE_GUID). This maps to Guid.
Integer	A 32-bit signed integer (DBTYPE_I4). This maps to Int32.
LongVarChar	A long string value (OleDbParameter only). This maps to String.
LongVarWChar	A long null-terminated Unicode string value (OleDbParameter only). This maps to String.
Numeric	An exact numeric value with a fixed precision and scale (DBTYPE_NUMERIC). This maps to Decimal.
Single	A floating point number within the range of -3.40E +38 through 3.40E +38 (DBTYPE_R4). This maps to Single.
VarChar	A variable-length stream of non-Unicode characters (OleDbParameter only). This maps to String.
Variant	A special data type that can contain numeric, string, binary, or date data, as well as the special values Empty and Null (DBTYPE_VARIANT). This type is assumed if no other is specified. This maps to Object.
VarNumeric	A variable-length numeric value (OleDbParameter only). This maps to Decimal.

Using the OleDBParameterCollection Object & OleDbParameter Object

- OK, now let's learn to use the ParemeterCollection. Like any collection we will be adding parameter objects using the ADD() method, getting Parameter objects using the ITEM PROPERTY etc.
- N
- The job of the **DataReader** object is to store the results of a query. As you recall, the results of an SQL query is a table. Therefore after execution of a query, the data stored inside the **DataReader** object is in the form of a table.

What is a Parameter and Understanding the Parameters Collection

What is a Parameter

- The Parameters Collection stores the PARAMTER Objects that are passed to queries. But, what are the parameters?
- To understand this, we need to go back and look at how queries are created and put together as a string in VB.NET. Let's look at the examples we used earlier in the Command Object section.
- The first example is a straight forward query where the value being searched is coded into the query:

Example 1:

- Supposed you want to execute the following query:

```
SELECT * FROM Customer WHERE Customer_ID = 111
```

- Note that the value 111 is hard-coded into the query and is a constant. This mean every time this query is executed, it will only affect the user whose ID = 111.
- Creating the string in VB.NET to be executed by the Command Object looks as follows:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = 111"
```

Example 2:

- In reality applications are not just written for one user, but for any user. You want to be able to search information for any user whose ID is passed via a Form or control.
- The value that is sent to the query is unknown and is usually contained within a variable say *intID*, for example the query string would look as follows:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & intID
```

- The variable *intID*, is a **PARAMETER** passed to the query.

Example 3:

- Supposed now that the value is a *string*. This changes things since as you know you need to enclose string within single quotes or (‘) depending on the database. For Microsoft Access & SQL Server Databases a string has to be enclosed in single quotes. For example, the following query contains a hard-coded value string:

```
SELECT * FROM Customer WHERE Customer_ID = ‘111’
```

- You need to create the string as follows:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ‘111’"
```

- Again there are no issues creating this string. This is a simple statement

Example 4:

- Now assuming the value is a string, but the value is unknown and is contained within a variable say ***intID***, then things get a bit complicated and difficult to create the string.
- You will need to use the **&** keyword to build the string including the characters required to enclosed the string value. The string will look as follows:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ‘" & intID & "’"
```

- Note that we needed to build our string and take into account the (‘) character required to enclosed any string in a Microsoft Access Query. Creating these strings can get complicated, messy and prone to errors.

Example 5:

- To show how complex creating the string can be, let’s look at the string required for an INSERT statement for a customer.
- The INSERT statement has the following syntax:

INSERT ***INTO*** *Table Name* ***(****column1, column2, column3, column n****)***
VALUES ***(****value1, value2, value3, value n****)***

- Using the Customer example, if we wanted to add a new customer to the CUSTOMER table the query will look as follows in when executed from the Database:

INSERT INTO *Customer* ***(***Customer_ID, LastName, FirstName, BirthDate, Address, Gender, PhoneNumber***)***
VALUES ***(****111, ‘Smith’, ‘Joe’, #12/12/1965#,’333 Jay Street’,’M’,’718 260 5555’****)***

- Creating this string in VB.NET will required the following complex string:

```
Dim strSQL As String

strSQL = "INSERT INTO Customer(Customer_ID, LastName, FirstName, BirthDate,
Address,Gender, PhoneNumber)" _
& "VALUES (" & intIDNumber & ",'" & strLastName & "','" & strFirstName & "',#" _
& dBirthDate & "#,'" & strAddress & "','" & strGender & "','" & strPhone & "')"
```

- **IMPORTANT!** Note the complexity of creating this string. You need to make sure that all the single quotes and other required syntax characters are properly handled in the string. The possibility for errors here are very high. Imagine if the table contained many columns etc. The string would be quite complex to build!
- **This is where the PARAMTER COLLECTION AND PARAMTER OBJECTS COME IN HANDY!!**
- **Using the PARAMTERS COLLECTION, we DON’T NEED TO CREATE COMPLEX VB STRINGS.**

Understanding the Parameters Collection

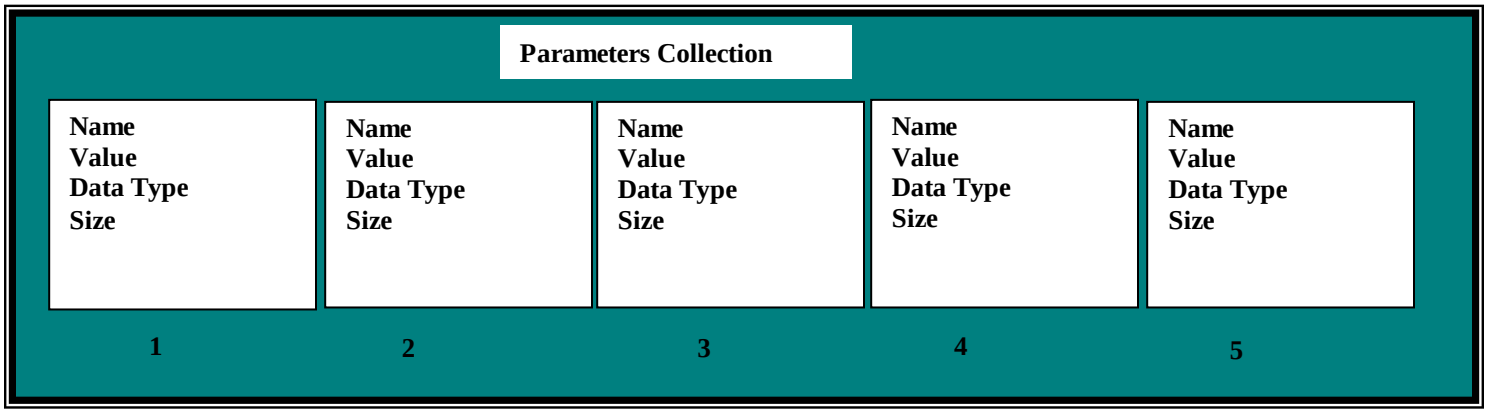
- Using the ***Parameters Collection***, we can avoid having to worry about creating such complex string when passing parameters to queries.
- In order to understand the **Parameter Collection** we first need to understand the object stored by the Collection, that is the **OleDBParemeter** Object.
- The **OleDBParemeter** objects stores the necessary information for each parameter.
- The important properties to this Class are follows:

  - ParameterName
  - Value
  - OleDBType – Data Type that the parameter represents
  - Size of data type

- The **Parameters Collection** stores each of the **OleDBParemeter** objects. Since it is a collection, it had the methods and properties such as:

  - Item
  - Count
  - Add
  - Remove
  - Etc.

- The diagram below illustrates this concept:



USING QUERY MARKERS to FORMAT QUERIES

- To use the Parameters Collection, you need to use special QUERY MARKERS to represent each of the parameters in the query.
- There are two markers:

  - ?
  - @COLUMNNAME

- Which markers you use depends on the following factors:

  - Which Data Provider you are using: **OLEDB PROVIDER** or **SQL CLIENT PROVIDER**
  - Which type of query **IN-LINE QUERY** OR **STORED PROCEDURE**
  - Database: SQL Server, Oracle or MS Access

- The rules are as follows:

Provider	Database	Query Type	Database	Query Marker	Example
OLEDB Provider	SQL Server	In-Line SQL	SQL Server	?	Select * From Customer WHERE Customer_ID = ?
		Stored Procedure	SQL Server	@ColumnName	Select * From Customer WHERE Customer_ID = @Customer_ID
	MS Access	In-Line SQL	MS Access	? or @ColumnName	Select * From Customer WHERE Customer_ID = ? or Select * From Customer WHERE Customer_ID = @Customer_ID
		Stored Procedure	MS Access	@ColumnName	Select * From Customer WHERE Customer_ID = @Customer_ID
SQL Provider	SQL Server	In-Line SQL	SQL Server	@ColumnName	Select * From Customer WHERE Customer_ID = @Customer_ID
		Stored Procedure	SQL Server	@ColumnName	Select * From Customer WHERE Customer_ID = @Customer_ID

Example Using OLEDB Provider & SQL SERVER Database

- For example in the SELECT query used earlier using **IN-LINE QUERY**:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = '" & strIDNum & "'"
```

- We replace the variable intID with the marker **?**:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ?"
```

  - Note that we don't have to worry about the formatting of the single quotes etc.

- If we were using **STORED PROCEDURES**, then we would use a named parameter and it would look as follows:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = @CustomerID"
```

- Let's look at the INSERT query. This is what it would look like without the **Paremeter** Class:

```
Dim strSQL As String

  strSQL = "INSERT INTO Customer(Customer_ID, LastName, FirstName, BirthDate,
  Address,Gender, PhoneNumber)" _
  & "VALUES (" & intIDNumber & ",'" & strLastName & "','" & strFirstName & "',#" _
   & dBirthDate & "#,'" & strAddress & "','" & strGender & "','" & strPhone & "')"
```

- Using the maker ?, the query looks as follows:

```
Dim strSQL As String

strSQL = "INSERT INTO Customer(Customer_ID, LastName, FirstName, BirthDate,
Address,Gender, PhoneNumber)" _
& "VALUES (?,?,?,?,?,?,?)"
```

- Using the maker @, the query looks as follows:

```
Dim strSQL As String

strSQL = "INSERT INTO Customer(Customer_ID, LastName, FirstName, BirthDate,
Address,Gender, PhoneNumber)" _
& "VALUES (@Customer_ID,@LastName,@FirstName,@BirthDate, @Address, @Gender,
@PhoneNumber)"
```

  - Notice how much simpler these strings are compare to the one without using the Parameters Collection. Less chance for errors.

Using the Parameters Collection

- Now we know the components of the Parameters collection:

  - ***OleDBParameter*** Object – Object stored inside the collection. Represents a Parameter. Contains the following IMPORTANT properties:
    - **Name**
    - **Value**
    - **Data Type that the parameter represents**
    - **Size of data type**

  - Methods of the ***Parameters Collection***:
    - **Item – *Important Property!***
    - Count
    - **Add – *Important Method!***
    - Remove
    - Etc.

- We also know that we need to Format the Query using the marker **?** or **@COLUMNNAME**.
- Next we need to know how to put it together.
- The Parameters Collection will contain an **oleDBParameter** object that represents each of the parameters in the query. This means that each of the **oleDBParameter** objects is mapped to each of the parameters in the query.

- You need the do the following steps:

1. FORMAT THE SQL STATEMENT USING THE **?** OR **@COLUMN_NAME** MARKERS**:**
   - `Option 1 for IN-LINE SQL: "SELECT * FROM Customer WHERE Customer_ID = ?"`
   - `Option 2 for STORED PROCEDURES: "SELECT * FROM Customer WHERE Customer_ID = @CustomerID"`

2. Create the **COMMAND OBJECT**
3. Use the Command Object ***CommandType*** property to tell the Command Object that you will be executing a ***TEXT query, stored procedure*** or ***TableDirect***. Text is the default.

4. Call COMMAND.PARAMTERS.**Add(Name,DataType, Sizet)** method to add to the collection each required **parameter object** as follows:
   - NO NEED TO CREATE A PARAMETER OBJECTS, YOU CAN DO IT ALL VIA THE COLLECTION ADD Method ONLY.
   - The ADD() METHOD of the Collection, accepts parameters data and creates the PARAMTER OBJECT INTERNALLY.
   - In the ADD method, you provide the arguments that will SET the following PROPERTIES of the internally created PARAMETER OBJECT:
     - Name (Can be any name, common practice is to use the COLUMN_NAME
     - Data Type
     - SIZE (OPTIONAL)

   - Examples:

```
objCmd.Parameters.Add("@Customer_ID", OleDbType.Integer)

objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar,20)
```

   - **IMPORTANT! You will need a Collection.ADD() method for every single PARAMTER MARKER in the query**
   - **IMPORTANT!! The PARAMETER OBJECTS via the ADD() method, MUST BE ADDED IN THE ORDER IN WHICH THEY ARE LOCATED IN THE SQL STATEMENT!**
   - Note that the VALUE PROPERTY of the PARAMETER OBJECT will be set using the COLLECTION ITEM PROPERTY (Shown below)

5. Set the **VALUE** PROPERTY of EACH ***oleDBParemeter*** object (calls to ADD) using the COMMAND.PARAMETERS.ITEM(KEY) PROPERTY to the VARIABLE representing the marker:

   - `objCmd.Parameters.Item("@Customer_ID").Value = intID.`

Parameters Collection ADD Method in detail

- The most important method of the Parameters Collection is the **ADD()** method.

  - The **Add()** method is used to add each of the parameters needed.
  - This method is overloaded and can take several arguments. I will show a typical implementation:

```
Public Function Add(ByVal parameterName As String, ByVal Type As OleDbType)
```

  - ***ParemeterName***: Column name or name of parameter.
  - ***Type***: Data Type. Must use one of the **oleDBType** Enumerator type

```
Public Function Add(ByVal parameterName As String, ByVal Type As OleDbType, ByVal Size As Integer )
```

  - ***ParemeterName***: Column name or name of parameter. This can be any name, but we will use the @COLUMNNAME for this name
  - ***Type***: Data Type. Must use one of the **oleDBType** Enumerator type
  - ***Size***: Size of data in database

- The table below is a snapshot of the listing of these data type you must choose. Note that this appears automatically in the Editing window. So there is no need to memorize this table:

Members

Member name	Description
Binary	A stream of binary data (DBTYPE_BYTES). This maps to an Array of type Byte.
Boolean	A Boolean value (DBTYPE_BOOL). This maps to Boolean.
Char	A character string (DBTYPE_STR). This maps to String.
Currency	A currency value ranging from -2^{63} (or -922,337,203,685,477.5808) to $2^{63}-1$ (or +922,337,203,685,477.5807). This maps to Decimal.
Date	Date data, stored as a double (DBTYPE_DATE). This maps to DateTime.
Decimal	A fixed precision and scale numeric value between $-10^{38}-1$ and $10^{38}-1$ (DBTYPE_DECIMAL). This maps to Decimal.
Double	A floating point number within the range of -1.79E +308 through 1.79E +308 (DBTYPE_R8). This maps to Double.
Guid	A globally unique identifier (or GUID) (DBTYPE_GUID). This maps to Guid.
Integer	A 32-bit signed integer (DBTYPE_I4). This maps to Int32.
LongVarChar	A long string value (OleDbParameter only). This maps to String.
LongVarWChar	A long null-terminated Unicode string value (OleDbParameter only). This maps to String.
Numeric	An exact numeric value with a fixed precision and scale (DBTYPE_NUMERIC). This maps to Decimal.
Single	A floating point number within the range of -3.40E +38 through 3.40E +38 (DBTYPE_R4). This maps to Single.
VarChar	A variable-length stream of non-Unicode characters (OleDbParameter only). This maps to String.
Variant	A special data type that can contain numeric, string, binary, or date data, as well as the special values Empty and Null (DBTYPE_VARIANT). This type is assumed if no other is specified. This maps to Object.
VarNumeric	A variable-length numeric value (OleDbParameter only). This maps to Decimal.

- To use the **ADD()** method in our code simply call the parameter collection and call **ADD()**. Remember that the Parameters collection is a member of the command object.
- For example, supposed we have the following query string with the ? marker:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ?"
```

- The call to **ADD()** method of the ***parameters collection*** to queries using the **?** marker is:

```
'Calling the Parameters collection Add method to add a parameter and type:
objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar)
```

- Example using **@COLUMNNAME** Marker the query looks as follows:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = @Customer_ID"
```

- The call to **ADD()** method of the ***Parameters Collection*** object to map **@COLUMNNAME** this marker:

```
'Calling the Parameters collection Add method to add a parameter and type:
objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar)
```

  - Note that the SAME PARAMTER NAME @Customer_ID can be used for both Marker type ? or @COLUMNNAME

SETTING THE VALUE OF THE PARAMTER OBJECT: Parameters Collection ITEM Property and the OleDBParemeter Object VALUE Property

- The next important steps are the **ITEM Property** of the *Parameters* Collection, and the **VALUE Property** of the *oleDBParameter* object stored inside the Collection.
  - We need to modify the parameter just added with the **ADD** method. Therefore we need the **Item** property to find the parameter and set its **VALUE** property.
  - We use the **ITEM** property passing the unique key or name given to the parameter to find the *oleDBParameter* object.
  - Then we call the *oleDBParameter* Object's **VALUE** property and set it to the variable that will map to the marker **?**

- This is what the code looks like for MS ACCESS OR SQL SERVER:

```
'Setting the Value property of the parameter object
objCmd.Parameters.Item("@Customer_ID").Value = strID
```

- This code can be shorten by COMBINING THE ADD METHOD AND ITEM PROPERTY using a one step method (**PREFERRED SYNTAX**):

```
'Add and setting the value in one step and type:
objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar).Value = strID
```

- Syntax with SIZE property included:

```
'Add and setting the value in one step and type:
objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar, 20).Value = strID
```

Putting it all together

- So the steps are:

1. Format the QUERY USING MARKERS **?** or **@COLUMNNAME**
2. For each MARKER PARAMTER IN QUERY, add a call to the ParameterCollection.**Add()** method in the order in which they are found in the query. You must indicate a **name** or the parameter such as @COLUMNNAME, **data type**, & Size.
   - **IMPORTANT! The order of the ADD statements MUST match the order in which the parameters are listed in the query, otherwise the query will NOT work.**
3. Repeat for every parameter

QUERIES WITH ? MARKERS:

- Again, supposed we have the following query, with the marker **?** that represents say a variable name ***intID***:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ?"
```

- The complete code to add and prepare one parameter is as follows:

```
'Calling the Parameters collection Add method to add a parameter Object and type:
objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar)

'Setting the Value property of the parameter object
objCmd.Parameters.Item("@Customer_ID").Value = strID
```

- This code can be shorten to the following form in one step:

```
'Add and setting the value in one step and type:
objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar).Value = strID
```

- **IMPORTANT!** You will need one of these code statement for every parameter in your query

QUERIES WITH @COLUMNNAME MARKERS:

- Again, supposed we have the following query, with the marker **@COLUMNNAME** that represents say a variable name ***intID***:

```
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =
                        @Customer_ID"
```

- The complete code to add and prepare one parameter is as follows:

```
'Calling the Parameters collection Add method to add a parameter Object and type:
objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar)

'Setting the Value property of the parameter object
objCmd.Parameters.Item("@Customer_ID").Value = strID
```

- This code can be shorten to the following form in one step (**PREFERRED SYNTAX**):

```
'Add and setting the value in one step and type:
objCmd.Parameters.Add("@Customer_ID", OleDbType.Integer).Value = strID
```

- **IMPORTANT!** You will need one of these code statement for every parameter in your query

Example 6:

- Let's look at another example. Suppose we want to implement the **INSERT** query from earlier example. we saw that the if we do NOT use parameters the complex string we need to process would look as follows:

```
Dim strSQL As String

  strSQL = "INSERT INTO Customer(Customer_ID, LastName, FirstName, BirthDate,
  Address,Gender, PhoneNumber)" _
  & "VALUES (" & intIDNumber & ",'" & strLastName & "','" & strFirstName & "',#" _
   & dBirthDate & "#,'" & strAddress & "','" & strGender & "','" & strPhone & "')"
```

QUERY with ? MARKERS:

- Using the marker **?**, we DON'T need such complex string, we simply use **?** as follows:

```
Dim strSQL As String

strSQL = "INSERT INTO Customer(Customer_ID, LastName, FirstName, BirthDate,
Address,Gender, PhoneNumber)" _
& "VALUES (?,?,?,?,?,?,?)"
```

- Add parameters to the Parameters Collection. REMEMBER THAT THE ORDER MUST BE THE SAME AS IT APPEARS IN THE QUERY:

```
'Add Paramter to Pareameters Collection and set value for each parameter
objCmd.Parameters.Add("Customer_ID", OleDbType.Integer)
objCmd.Parameters.Item("Customer_ID").Value = intIDNumber
'Short Syntax
objCmd.Parameters.Add("LastName", OleDbType.Char).Value = strLastName
objCmd.Parameters.Add("FirstName", OleDbType.Char).Value = strFirstName
objCmd.Parameters.Add("BirthDate", OleDbType.Date).Value = dBirthDate
objCmd.Parameters.Add("Address", OleDbType.Char).Value = strAddress
objCmd.Parameters.Add("Gender", OleDbType.Char).Value = strGender
objCmd.Parameters.Add("PhoneNumber", OleDbType.Char).Value = strPhone
```

QUERIES with @COLUMNAME MARKERS:

- Using the marker **@ColunnName**:

```
Dim strSQL As String

strSQL = "INSERT INTO Customer(Customer_ID, LastName, FirstName, BirthDate,
Address,Gender, PhoneNumber)" _
& "VALUES
(@Customer_ID,@LastName,@FirstName,@BirthDate,@Address,@Gender,@PhoneNumber)"
```

- Add parameters to the Parameters Collection. REMEMBER THAT THE ORDER MUST BE THE SAME AS IT APPEARS IN THE QUERY:

```
'Add Paramter to Pareameters Collection and set value for each parameter
objCmd.Parameters.Add("@Customer_ID", OleDbType.Integer).Value = intIDNumber
objCmd.Parameters.Add("@LastName", OleDbType.Char).Value = strLastName
objCmd.Parameters.Add("@FirstName", OleDbType.Char).Value = strFirstName
objCmd.Parameters.Add("@BirthDate", OleDbType.Date).Value = dBirthDate
objCmd.Parameters.Add("@Address", OleDbType.Char).Value = strAddress
objCmd.Parameters.Add("@Gender", OleDbType.Char).Value = strGender
objCmd.Parameters.Add("@PhoneNumber", OleDbType.Char).Value = strPhone
```

Summary – Using the Parameters Collection

- To use the *Parameters Collection*, you need to use special markers to represent the parameters. The markers as follows:
  - Microsoft Access Only: Use a question mark (**?**)
  - SQL Server & Microsoft Access: Use named parameters using the **@ColumnName** symbol – Ex. **@CustomerID**
- Since the second marker syntax can be used for both SQL Server and MS Access, creating our parameterized query using this marker makes our program scalable. We will use this syntax throughout my examples.
  - **IMPORTANT! The order of the parameter statements MUST be the same as they appear in the Query. Otherwise the query will NOT work. This can be tricky since executing the query with the wrong order may not yield an exception, thus you may think the query worked when it did not.**
- Steps to use the Parameters Collection Class in your applications:

Step 1: We create the Connection Object OLEDB PROVIDER

MS Access:

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

MS SQL Server:

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=SQLOLDB;Data Source=Server01;Database =NYCTCDB;User
ID=sa;Password=password"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

Step 2: Create the Query and use a Symbol ? or @ to represent the parameters

FOR ? PARAMETER MARKERS:

- We use the marker **?** character as a place holder for the Parameters instead of a variable in the query

```
'Step 4-Create SQL string.  Use a ? as a place holder for the value of the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ?"
```

FOR @COLUMNNAME PARAMETER MARKERS:

- We use the marker **@ColumnName** character as a place holder for the Parameters instead of a variable in the query

```
'Step 4-Create SQL string.  Use a ? as a place holder for the value of the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = @Customer_ID"
```

Step 3: Create the Command Object and initialize it with connection and query

- Create the command object. Use any of the methods shown previously

```
'Step 5-Create Command object, pass Query string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

Step 4: Use the Parameters Collection. Call the ADD() method & Set the Value to the Parameter Object to the variable

- Call Add() and set value

FOR ? & @COLUMNNAME PARAMETER MARKERS:

```
'Step 6-Add Parameter to Collection and Set Value to variable
objCmd.Parameters.Add("Customer_ID", OleDbType.Integer).Value = intID
```

FOR ? & @COLUMNNAME PARAMETER MARKERS:

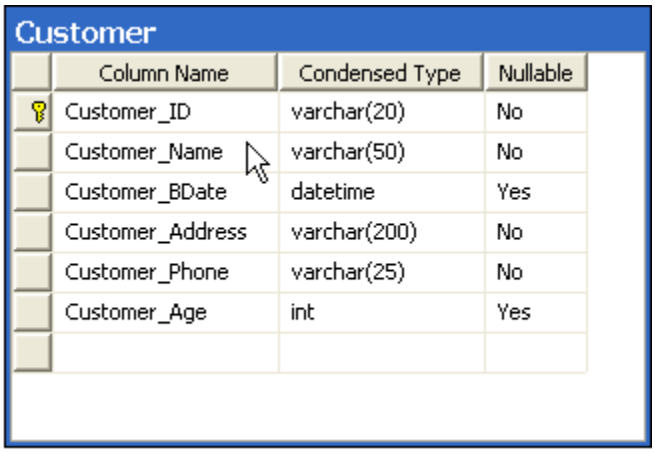
```
'Step 6-Add Parameter to Collection and Set Value to variable
objCmd.Parameters.Add("@Customer_ID", OleDbType.Integer).Value = intID
```

Step 5: Repeat Step 4 for every Parameter in QUERY in the order in which they appear

- Repeat Step 4

3.6 SUMMARY – Data Access using OleDBDataReader, IN-LINE SQL & SQL SERVER Database

- In the previous sections, we analyzed and learned how to use ADO.NET Data Provider to perform data access.
- Our examples so far have targeted MS ACCESS database.
- Now, let's summarize all the code required to perform data access using the following requirements:
  1. **DataReader** as our storage mechanism to store the results of the query.
  2. We will use **Microsoft SQL Server 2005 EXPRESS**
  3. Use the following CUSTOMER table in a database named **smallbusinesDB** for our summary code:



  4. use DATABASE AUTHENTICATION in the examples to connect to the SQL database
  5. We will also set OPTION STRICT ON in our code so that proper data type conversion is done
  6. We will use the following set of variables to store our RECORD from the database:

     - **`strIDNumber`**
     - **`strName`**
     - **`dBirthDate`**
     - **`strAddress`**
     - **`strPhone`**
     - **`intAge`**

  7. We will also NOT IMPLEMENT EXCEPTIONS AT THIS TIME

- The following examples cover the various data access functionalities such as select, insert, update and delete, etc.

Examples 7 - Executing Non-Parameter SELECT Query (IN-LINE SQL)

- Problem Statement:
  - Execute a query to return the record from the *Customer* Table of an Access Database for the customers whose ID Number is entered in a Form using the text box **txtIDNum**. Don't use the parameters collection. Display the results in a message box.

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"
'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

```
'Step 4-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID =" & txtIDNum

'Step 5-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
'Step 6-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
'Step 7-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then

      'Step 8a-Call Read() Method to point and read the first record
      objDR.Read()

      'Step 8b-Extract data from a row.  Use Item method to get the column data
                strIDNumber = CStr(objDR.Item(0))
                strName = CStr(objDR.Item(1))
                dBirthDate = CDate(objDR.Item(2))
                strAddress = CStr(objDR.Item(3))
                strPhone = CStr(objDR.Item(4))
                intAge = CInt(objDR.Item(5))

      'Step 8c-Now that you have the data for a record or row, do what you want
     MessageBox.Show(strIDNumber & strName & dBirthDate & strAddress & strPhone & intAge)

Else
      'Step 9-No data returned, Record not found.  Do what you want here!
      MessageBox.Show("No Customers found")

End If
```

```
'Step 10-Teminate Command Object
objCmd.Dispose()
objCmd = Nothing
'Step 11- Teminate DataReader Object.
objDR.Close()
objDR = Nothing
'Step 12-Terminate the Connection Object
objConn.Close()
objConn.Dispose()
objConn = Nothing
```

Example 8 - Executing Parameterized SELECT Query

- Problem Statement:
  - Execute a query to return the record from the *Customer* Table of an Access Database for the customers whose ID Number is entered in a Form using the text box **txtIDNum**. User the Parameters Collection to manage the parameters. Display the results in a message box.

```
'Step 1-Create Connection string
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

```
'Step 4-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ?"

'Step 5-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
'Step 6-Add Parameter to Collection & Set Value to variable storing data
objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar).Value = txtIDNum.txt
```

```
'Step 7-Create DATAREADER object & Execute Query
Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
'Step 8-Test to make sure there is data in the DataReader Object
If objDR.HasRows Then
      'Step 9a-Call Read() Method to point and read the first record
      objDR.Read()

      'Step 9b-Extract data from a row.  Use Item method to get the column data
                strIDNumber = CStr(objDR.Item(0))
                strName = CStr(objDR.Item(1))
                dBirthDate = CDate(objDR.Item(2))
                strAddress = CStr(objDR.Item(3))
                strPhone = CStr(objDR.Item(4))
                intAge = CInt(objDR.Item(5))

      'Step 9c-Now that you have the data for a record or row, do what you want
    MessageBox.Show(strIDNumber & strName & dBirthDate & strAddress & strPhone & intAge)
Else
      'Step 10-No data returned, Record not found.  Do what you want here!
      MessageBox.Show("No Customers found")

End If
```

```
'Step 11-Teminate Command Object
objCmd.Dispose()
objCmd = Nothing
'Step 12- Teminate DataReader Object.
objDR.Close()
objDR = Nothing
'Step 13-Terminate the Connection Object
objConn.Close()
objConn.Dispose()
objConn = Nothing
```

Example 9 - Executing SELECT Query that Return Multiple Records

- Problem Statement:
  - Execute a query to return more than one record from the *Customer* Table of an Access Database. User the Parameters Collection to manage the parameters.

```
'Step 1-Creates Connection Object, assigns connection string
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"

Dim objConn As New OleDbConnection(strConn)
objConn.Open()
```

```
            'Step 2-Create SQL string
            Dim strSQL As String = "SELECT * FROM Customers"

            'Step 3-Create Command object, pass string and connection object as arguments
            Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
            'Step 4-Create DATAREADER object & Execute Query
            Dim objDR As OleDbDataReader = objCmd.ExecuteReader
```

```
            'Step 5-Test to make sure there is data in the DataReader Object
            If objDR.HasRows Then

                'Step 6-Iterate through DataReader one record at a time.
                Do While objDR.Read

      'Step 7-Extract data from a row.  Use Item method to get the column data
                strIDNumber = CStr(objDR.Item(0))
                strName = CStr(objDR.Item(1))
                dBirthDate = CDate(objDR.Item(2))
                strAddress = CStr(objDR.Item(3))
                strPhone = CStr(objDR.Item(4))
                intAge = CInt(objDR.Item(5))


      'Step 8-Now that you have the data for a record or row, do what you want
    MessageBox.Show(strIDNumber & strName & dBirthDate & strAddress &  strPhone & intAge)
                Loop

            Else
                'Step 9-No data returned, Record not found.  Do what you want here!
                MessageBox.Show("No Customers found")

            End If
```

```
            'Step X-Terminate Command Object
            objCmd.Dispose()
            objCmd = Nothing

           'Step Y- Terminate DataReader Object.
            objDR.Close()
            objDR = Nothing

            'Step Z-Terminate the Connection Object
            objConn.Close()
            objConn.Dispose()
            objConn = Nothing
```

Example 10 - Executing Parameterized UPDATE Query

- Problem Statement:
  - Execute a query to UPDATE a record in the *Customer* Table of an Access Database for the customers whose ID Number is entered in a Form using the text box **txtIDNum**. The values modified are obtained from the Form as well. Display whether the UPDATE failed using a message box.

```
'Step 1-Create Connection & open it.  Alternate Syntax
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"

Dim objConn As New OleDbConnection(strConn)
objConn.Open()
```

```
        'Step 2-Create Query and use Parameter Markers
        strSQL = "UPDATE Customer SET Customer_Name=?, Customer_BDate=?," & _
                                      "Customer_Address=?,Customer_Phone=?," & _
                                      "Customer_Age=? " & _
                 "WHERE Customer_ID=?"
        'Step 3-Create Command object, pass string and connection object as arguments
        Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
        'Step 4-Add Parameter to Collection & Set Value
    objCmd.Parameters.Add("@Customer_Name", OleDbType.VarChar).Value = strName
    objCmd.Parameters.Add("@Customer_BDate", OleDbType.Date).Value = dBirthDate
    objCmd.Parameters.Add("@Customer_Address", OleDbType.VarChar).Value = strAddress
    objCmd.Parameters.Add("@Customer_Phone", OleDbType.VarChar).Value = strPhone
    objCmd.Parameters.Add("@Customer_Age", OleDbType.Integer).Value = intAge
    objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar).Value = strIDNumber
```

```
        'Step 5-Execute Non-Row Query Test result and throw exception if failed
        Dim intRecordsAffected As Long = objCmd.ExecuteNonQuery()
```

```
        'Step 6-Test result and throw exception if failed
        If intRecordsAffected <> 1 Then
            MessageBox.Show("Error Updating Record")
        End If

        'Step X-Terminate Command Object
        objCmd.Dispose()
        objCmd = Nothing
        'Step Y-Terminate the Connection Object
        objConn.Close()
        objConn.Dispose()
        objConn = Nothing
```

Example 11 - Executing Parameterized INSERT Query

- Problem Statement:
  - Execute a query to INSERT a NEW record in the *Customer* Table of an Access Database. The values added are obtained from the Form. Display whether the INSERT was successful using a message box.

```
'Step 1-Create Connection, assign Connection to string & open it
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"

Dim objConn As New OleDbConnection(strConn)
objConn.Open()
```

```
        'Step 2-Create Query and use Parameter Markers
        strSQL = "INSERT INTO Customer(Customer_ID,Customer_Name," & _
                                      "Customer_BDate," & _
                                      "Customer_Address," & _
                                      "Customer_Phone,Customer_Age)" & _
                 "VALUES(?,?,?,?,?,?)"
        'Step 3-Create Command object, pass string and connection object as arguments
        Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
        'Step 4-Add Paramter to Pareameters Collection and set value for each parameter
    objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar).Value = strIDNumber
    objCmd.Parameters.Add("@Customer_Name", OleDbType.VarChar).Value = strName
    objCmd.Parameters.Add("@Customer_BDate", OleDbType.Date).Value = dBirthDate
    objCmd.Parameters.Add("@Customer_Address", OleDbType.VarChar).Value = strAddress
    objCmd.Parameters.Add("@Customer_Phone", OleDbType.VarChar).Value = strPhone
    objCmd.Parameters.Add("@Customer_Age", OleDbType.Integer).Value = intAge
```

```
        'Step 5-Execute Non-Row Query Test result and throw exception if failed
        Dim intRecordsAffected As Long = objCmd.ExecuteNonQuery()
```

```
        If intRecordsAffected <> 1 Then
                MessageBox.Show("Error INSERTING Record")
        End If
```

```
        'Step X-Terminate Command Object
        objCmd.Dispose()
        objCmd = Nothing

        'Step Y-Terminate the Connection Object
        objConn.Close()
        objConn.Dispose()
        objConn = Nothing
```

Example 12 - Executing Parameterized DELETE Query

- Problem Statement:
  - Execute a query to DELETE a record in the *Customer* Table of an Access Database for the customers whose ID Number is entered in a Form using the text box **txtIDNum**. Display whether the UPDATE failed using a message box.

```
'Step 1-Create Connection, assign Connection to string & open it
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"

Dim objConn As New OleDbConnection(strConn)
objConn.Open()
```

```
        'Step 2-Create Command, Query, assing query, and assign connection
        Dim strSQL As String = "DELETE FROM Customer WHERE Customer_ID = ?"

        'Step 3-Create Command object, pass string and connection object as arguments
        Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
        'Step 4-Add Parameter to Collection & Set Value
       objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar).Value = strIDNumber
```

```
        'Step 5-Execute Non-Row Query Test result and throw exception if failed
        Dim intRecordsAffected As Long = objCmd.ExecuteNonQuery()
```

```
        If intRecordsAffected <> 1 Then
                MessageBox.Show("Error INSERTING Record")
        End If
```

```
        'Step X-Terminate Command Object
        objCmd.Dispose()
        objCmd = Nothing

        'Step Y-Terminate the Connection Object
        objConn.Close()
        objConn.Dispose()
        objConn = Nothing
```

4.0 Using ADO.NET DataSet Class

4.1 REVIEW OF Data Access Using DataReader Object

- In the previous section we went over the ADO.NET Classes required to perform Data Access using the **DataReader** Object to store the results of a query.
- To execute SELECT Queries we needed the following objects:

  - **Connection** object – *Establish connection*
  - **Command** Object – *Execute Query*
    - [Optional] **Parameters Collection** – *Better management of query parameters*
  - **DataReader** Object – *Store the results of the query*

- To execute INSERT, UPDATE & DELETE Queries we need the following objects only:

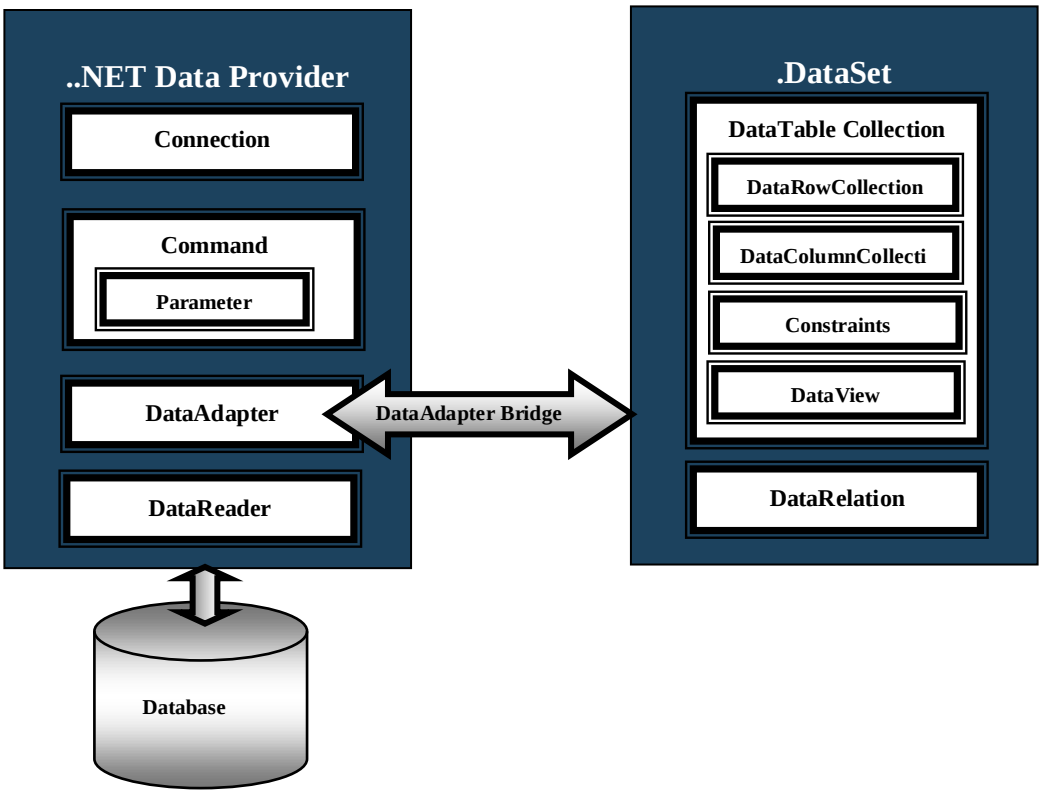
  - **Connection** object – *Establish connection*
  - **Command** Object – *Execute Query*
    - [Optional] **Parameters Collection** – *Better management of query parameters*

- The main point here is that for select queries we can store our results in a **DataReader** Objects, if we wanted a fast and simple way to perform data access.
- Now we will look at a more powerful mechanism to store the results of a query by using the ***DataSet*** Class.
- Using a ***DataSet*** Object provides the feature of *Disconnected* Data Access, which is one of the new ADO.NET most recognized features

4.2 Data Access Using DataSet Object

- In this section we will look at storing the results of a SELECT query using the **DataSet** Object.
- The objects required will be as follows:

  - **Connection** object – *Establish connection*
  - **Command** Object – *Execute Query*
    - [Optional] **Parameters Collection** – *Better management of query parameters*
  - **DataAdapter** – *Bridge to DataSet object. Executes query via Command Object and Populates or fills the DataSet*
  - **DataSet** Object – *Store results of the query. Modify & update records. Update database with updated values etc.*



4.3 DataAdapter Class

- The ADO.NET DataAdapter Class performs the following functions:

.NET Data Provider	*Description*
DataAdapter	▪ This object acts as a *bridge* between the ***Database*** and the disconnected object the ***DataSet***. ▪ It fetches the results of queries and *fills* or populates the ***DataSet*** Object so you can work with data offline. This is done via a method inside the *DataAdpater* named ***Fill()***. ▪ The *DataAdapter* object actually exposes a number of properties from the Command Object for *Selecting*, *Updating*, *Inserting* and *Deleting* data to the database. ▪ The *DataAdapter* object populates tables into the DataSet Object and also submits changes from the DataSet Object to the database. It is a *bridge* between these two libraries.

- Remember that the .NET Library comes equipped with a provider or library for SQL Server and OLE DB, so we will have the following classes available to us depending on which database we want to use:

I. **SQL Client** – Contains all the ADO.NET libraries to connect ONLY to SQL Server database:
   a. **SQLDataAdapter**

II. **OleDBClient** – Use for other databases other than SQL Server that support OleDB, such as Microsoft Access, Oracle etc.
   a. **OleDBDataAdapter**

- My sample code will focus on OLE DB.

DataAdapter Class – Properties & Methods

- Table below lists some important properties, methods and constructor of the **DataAdapter Class**:

Public Constructors

OleDbDataAdapter Constructor	▪ Overloaded. Initializes a new instance of the **OleDbDataAdapter** class. ▪ You can actually pass a query string when creating an object of this class. In addition you can pass a connection or a command object. In short, there are several constructor methods available.

Public Properties

Property	Description
	▪
SelectCommand	▪ Gets or sets an SQL statement or stored procedure used to select records in the data source. ▪ You can also assign a Command Object with the query to this parameter as well.
	▪

Public Methods

Method	Description
Fill **OleDbDataAdapter.Fill (DataSet)** **OleDbDataAdapter.Fill (DataTable)** **OleDbDataAdapter.Fill (DataSet, String)**	▪ Populates the **DataSet** Object with records, or tables resulting from the query. Adds or refreshes rows in the DataSet to match those in the data source using the DataSet name, and creates a DataTable named "Table." Adds or refreshes rows in a specified range in the DataSet to match those in the data source using the DataSet, DataTable, and IDataReader names. Adds or refreshes rows in the DataSet to match those in the data source using the DataSet and DataTable names.
Dispose	Releases the memory.

Using the DataAdapter Class Object

- The **DataAdapter** object acts as a bridge between the disconnected and the connected halves of the ADO.NET object model. It can do the following:
  - *DataAdapter* pulls data from the database and populates the **DataSet** Object.
  - *DataAdapter* can take updates from your **DataSet** and submit them to the database.
  - *DataAdpater* works with disconnected data in the **DataSet**, after it fills the **DataSet**.

- Note that we assume that at this point you have already created your **Connection** and **Command** Objects as shown previously.
- Steps to add code to use the **DataAdapter** Object in your applications are as follows:

Step 1: Verify that the ADO.NET Provider and Data mechanism are imported into your code

```
Imports System.Data
Imports System.Data.OleDb  'OLEDB Provider
```

Step 2: We assume that you have already created the Connection and Command objects

- Connection object already created and opened.
- Command object already created

```
'Step 1-Create Connection and open
Dim strConn As String = "Provider=Microsoft.Jet.OleDB.4.0;Data Source=C:\DB\video.mdb"
Dim objConn As New OleDbConnection(strConn)
objConn.Open()

'Step 4-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customers

'Step 5-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

Step 3: Create the Data Adapter Object & Initializing with Command Object

- Now we proceed by creating a DataAdapter Object.

```
'Step 6-Create DataAdpater object
Dim objDA As New OleDbDataAdapter(objCmd)
```

Step 4: Execute Query and Populate DataSet Object. Use Fill() method

- In order to demonstrate the use of the **DataAdapter**, we need to also use a DataSet object.
- We will use some basic properties and methods of the **DataSet** Object to explain the DataAdapter object, but details on DataSet object will be explain in the **DataSet** object section.
- In this step we will call the **DataAdapter.Fill()** method to populate the DataSet object with the results of the query.

```
'Step 7-Create DataAdpater object
Dim objDS As New DataSet()

'Step 8-Call Fill method, pass DataSet object as argument
objDA.Fill(objDS)
```

4.4 DataSet Class

- The ADO.NET DataSet Class performs the following functions:

.NET Data Provider	*Description*
DataSet Object	▪ Stores the results of a query. ▪ Very powerful Class, can store entire tables as well as several tables or entire database locally. Think of the implication, entire database stored locally, no need to traverse network to get data. ▪ This object stores a *Collection* of ***DataTable*** Objects and a *Collection* of **DataRelationship** Objects. ▪ ***DataTable*** – Object that stores a Collection named **DATAROW** and a Collection named **DATACOLUMN** and other objects to manage the data retrieved from the database. ▪ ***DataRelantionship*** – This Collection stores *DataRelationship* Objects with information concerning the relationship between the tables, primary & foreign keys that link the tables etc. ▪ The explanation to the other collections and objects is listed below.

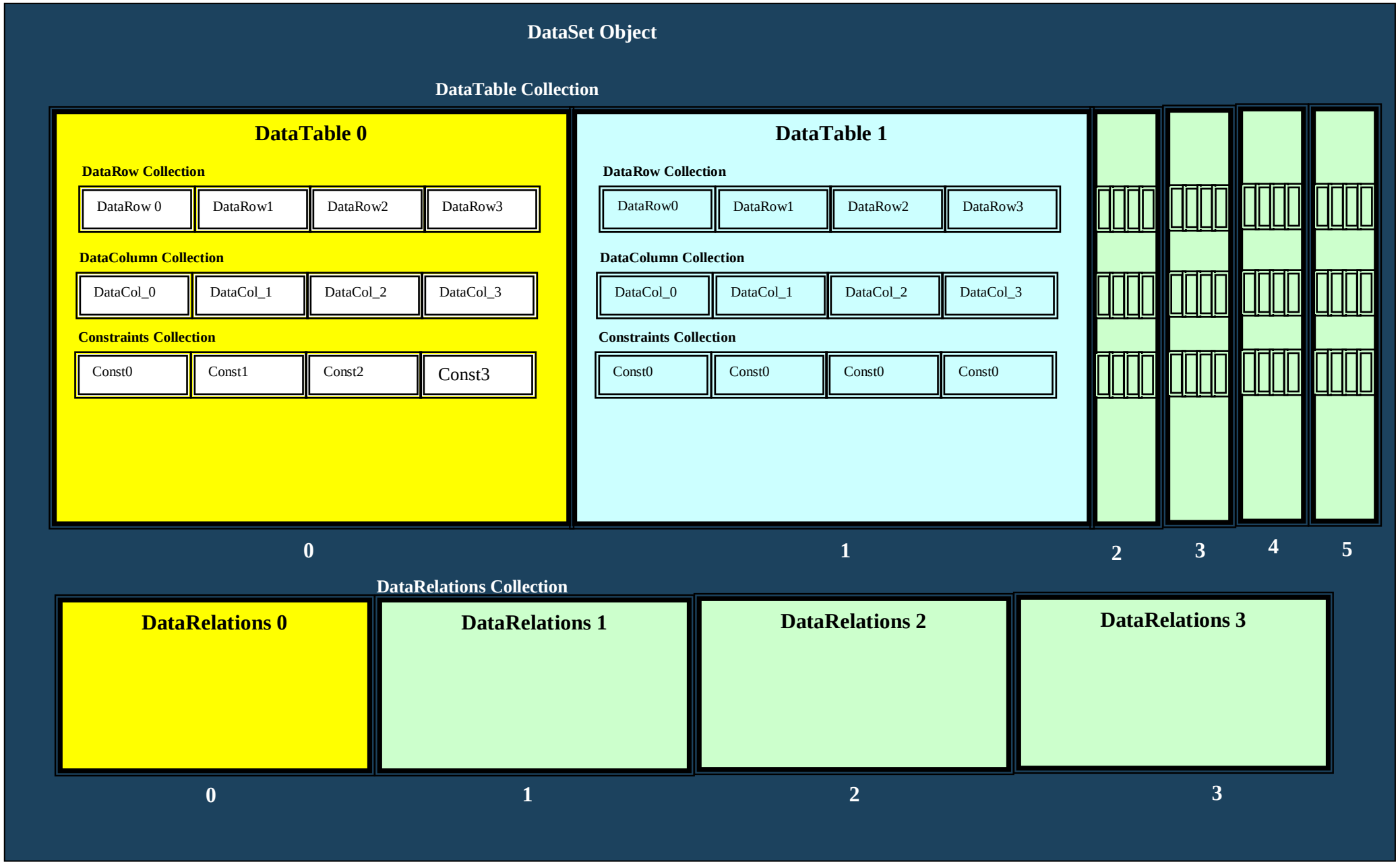
DataSet Members	*Description*
DataTable Collection	▪ Collection that Stores **DataTable** Objects. ▪ The explanation to the *DataTable* object is listed below.
DataRelationship Collection	▪ Collection that Stores **DataRelationship** Objects. ▪ The explanation to the **DataRelationship** object is listed below.

DataTable Object Members	*Description*
DataTable Object	▪ This object lets you stores and examine the data returned from a query. It represents the result of query ▪ The rows and columns returned from a query are stored in two Collections named ***DataRow*** Collection and ***DataColumn*** Collection. Both of these child objects will be described below.
DataColumn Collection	▪ A collection storing ***DataColumn*** Objects. Each of these objects store information about one Column in the table. ▪ Each ***DataColumn*** Object corresponds to one Column in the table. ▪ The ***DataColumn*** object DOES NOT store DATA! It only stores information about the structure of the column or METADATA, such as data type, properties size, format etc.
DataRow Collection	▪ A collection storing ***DataRow*** Objects. Each of these objects store information about the *Row* in the table. ▪ Each ***DataRow*** Object corresponds to one Row in the table. ▪ The ***DataRow*** object STORES the actual DATA return from a query. ▪ You examine the content of each *DataRow* object in the collection to retrieve and analyzed the data. ▪ You can use a ***For..Each..Next*** loop to iterate through the collection and access the data.

Constraints Collection	▪ A collection storing ***Contraints*** Objects. Each of these objects store information about the constraints or rules placed on columns or multiple columns in the table stored in the *DataSet*.
DataView	▪ This object is used to view the data in different ways. ▪ If you want to sort by column, filter rows by criteria etc. ▪ Multiple views of the same data etc.
DataRelation Collection	▪ A collection storing ***DataRelation*** Objects. Each of these objects store information about the relationship between the tables. ▪ Also information about the primary & foreign keys that link the tables. ▪ In addition this object enforces referential integrity.

DataRelations Collection Member	***Description***
DataRelationship Object	▪ ***DataRelation*** Objects store information about the relationship between the tables. ▪ Also information about the primary & foreign keys that link the tables. ▪ In addition this object enforces referential integrity

- I understand this is a complex data model, so I came up with a diagram below that may help you visualize what the DataSet Class looks like


DataSet Object
DataTable Collection
DataTable 0
DataRow Collection
DataRow 0
DataRow1
DataRow2
DataRow3
DataColumn Collection
DataCol_0
DataCol_1
DataCol_2
DataCol_3
Constraints Collection
Const0
Const1
Const2
Const3
DataTable 1
DataRow Collection
DataRow0
DataRow1
DataRow2
DataRow3
DataColumn Collection
DataCol_0
DataCol_1
DataCol_2
DataCol_3
Constraints Collection
Const0
Const0
Const0
Const0
0
1
2
3
4
5
DataRelations Collection
DataRelations 0
DataRelations 1
DataRelations 2
DataRelations 3
0
1
2
3

DataSet Class – Properties & Methods

- Table below lists some important properties, methods and constructor of the **DataAdapter Class**:

Public Constructors

DataSet Constructor	▪ Overloaded. Initializes a new instance of the **OleDbDataSet** class. ▪

Public Properties

Property	Description
DataSetName	▪ Gets or sets the name of the current **DataSet**. ▪
Tables	▪ Gets the collection of tables contained in the **DataSet**

Public Methods

Method	Description

Using the DataSet Class Object

- The **DataSet** object stores the result of your query in a complex and flexible data structure.
- In this course, we will simply show the basic use of this class for fetching and retrieving data only.
- In order to use the DataSet Object, you need to keep in mind the diagram show above. Primary the following
  - ***DataSet*** contains a collection of **DataTable** objects.
  - ***DataTable*** object contains a Collection name **DataRow** that hold **Row** objects that store the rows of a query.
  - ***DataTable*** object contains a Collection name **DataColumn** that hold **Column** objects that store the metadata of each column
    - Note that for simple data access, you only need the **DataTable**, **DataRow** collection and Row objects.

Example 13 - Executing SELECT Query using DataSet that returns One or Multiple Records

- Problem Statement:
  - Execute a query to return the ***one or multiple records*** from the *Customer* Table of the SQL Database for the customers whose ID Number is is available via a variable **strIDNumber**. User the Parameters Collection to manage the parameters. Use DataSet object to store and manage the data retrieved from the database. Display the resultant records in a message box, indicate if record is empty as well.

- Steps to create data access code to use the **DataAdapter** & **DataSet** Objects in your applications are as follows:

Step 1: Verify that the ADO.NET Provider and Data mechanism are imported into your code

```
Imports System.Data
Imports System.Data.OleDb  'OLEDB Provider
```

Step 2: We create the Connection, Command object and set Parameters as necessary.

- Code:

```
'Step 1-Create Connection, assign Connection to string & open it
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

```
'Step 4-Create SQL string.  We assume here that the variable intID contains the ID
Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ?"

'Step 5-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
'Step 6-Add Parameter to Collection & Set Value to variable storing data
       objCmd.Parameters.Add("@Customer_ID", OleDbType.VarChar).Value = strIDNumber
```

Step 3: We create DataAdapter, DataSet Object and execute query and populate DataSet.

- Code:

```
'Step 7-Create DataAdpater object
Dim objDA As New OleDbDataAdapter(objCmd)

'Step 8-Create DataAdpater object
Dim objDS As New DataSet()

'Step 9-Call Fill method, pass DataSet object as argument
objDA.Fill(objDS)
```

Step 4: Retrieving the Data Stored in the DataTable Data Row Collection.

- Now we need to search the internal objects of the **Tables** Collection to extract our results:

```
'Step 11-Get first Table object from DataTable Collection
Dim objDT As DataTable = objDS.Tables.Item(0)

'Step 12-Create Row object to hold the results of the search
Dim objRow As DataRow

'Step 13-Test to make sure there is data in the in Data Rows Collection
'Note that the Rows (DataRows) collection is a child of the DataTable.
If objDT.Rows.Count <> 0 Then

    'Step 14-Iterate through collection and extract data from a row.
    'Use Item method to get the column data
    For Each objRow In objDT.Rows
      'Step 14b-Extract data from a row.  Use Item method to get the column data
                strIDNumber = CStr(objDR.Item(0))
                strName = CStr(objDR.Item(1))
                dBirthDate = CDate(objDR.Item(2))
                strAddress = CStr(objDR.Item(3))
                strPhone = CStr(objDR.Item(4))
                intAge = CInt(objDR.Item(5))


      'Step 15-Now that you have the data for a record or row, do what you want
    MessageBox.Show(strIDNumber & strName & dBirthDate & strAddress & strPhone & intAge)

    Next

 Else
            'Step 16-No data returned, Record not found.  Do what you want here!
           MessageBox.Show("No Customers found")

 End If
```

Step 5: Clean up

- Now clean up and destroy objects:

```
'Step 17-Terminate objects
objCmd.Dispose()
objCmd = Nothing
objDA.Dispose()
objDA = Nothing
objDT.Dispose()
objDT = Nothing
objRow = Nothing
objConn.Close()
objConn.Dispose()
objConn = Nothing
```

Example 14 - Executing SELECT Query that Return TWO Tables using DataSet

- Problem Statement:
  - Execute two queries to return TWO TABLES. Return the *Customer* & Product Tables. User the Parameters Collection to manage the parameters. Use a DataSet to store the multiple tables

- The code to handle multiple records is identical as the previous code. Since a collection is being used to store the Rows, the For Each Loop will either return one row or multiple rows.
- Steps to create data access code to use the **DataAdapter** & **DataSet** Objects in your applications are as follows:

Step 1: Verify that the ADO.NET Provider and Data mechanism are imported into your code

```
Imports System.Data
Imports System.Data.OleDb  'OLEDB Provider
```

Step 2: We create the Connection, Command object and set Parameters as necessary.

- Code:

```
'Step 1-Create Connection, assign Connection to string & open it
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

```
'Step 4-Create SQL string.  Here we have 2 QUERIES, to search 2 Tables
            Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ?;" & _
                                   "SELECT * FROM Product WHERE Product_ID = ?"
'Step 5-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
'Step 6-Add Parameter to Collection & Set Value to variable storing data
objCmd.Parameters.Add("Customer_ID", OleDbType.Integer).Value = txtIDNum.txt
```

Step 3: We create DataAdapter, DataSet Object and execute query and populate DataSet.

- Code:

```
'Step 7-Create DataAdpater object
Dim objDA As New OleDbDataAdapter(objCmd)

'Step 8-Create DataAdpater object
Dim objDS As New DataSet()

'Step 9-Call Fill method, pass DataSet object as argument
objDA.Fill(objDS)
```

Step 4: Retrieving the Data Stored in the DataTable Data Row Collection.

- Now we need to iterate through the FIRST TABLE, table(0) to extract the ROWS for the first table in the **DataTable** Collection:

```
'Step 11-Get first Table object from DataTable Collection
Dim objDT As DataTable = objDS.Tables.Item(0)

'Step 12-Create Row object POINTER to point to each ROW OBJECT
Dim objRow As DataRow

'Step 13-Test to make sure there is data in the in Data Rows Collection
'Note that the Rows (DataRows) collection is a child of the DataTable.
If objDT.Rows.Count <> 0 Then

    'Step 14-Iterate through collection and extract data from a row.
    'Use Item method to get the column data
    For Each objRow In objDT.Rows
      'Step 14b-Extract data from a row.  Use Item method to get the column data
                strIDNumber = CStr(objDR.Item(0))
                strName = CStr(objDR.Item(1))
                dBirthDate = CDate(objDR.Item(2))
                strAddress = CStr(objDR.Item(3))
                strPhone = CStr(objDR.Item(4))
                intAge = CInt(objDR.Item(5))


      'Step 15-Now that you have the data for a record or row, do what you want
    MessageBox.Show(strIDNumber & strName & dBirthDate & strAddress & strPhone & intAge)

    Next

 Else
            'Step 16-No data returned, Record not found.  Do what you want here!
            MessageBox.Show("No Customers found")

 End If
```

Step 5: Retrieving the Data Stored in the DataTable Data Row Collection.

- Now we need to iterate through the FIRST TABLE, table(0) to extract the ROWS for the first table in the **DataTable** Collection:

```
'Step 11-POINT OR GET SECOND Table object from DataTable Collection
 objDT objDS.Tables.Item(1)

'Step 13-Test to make sure there is data in the in Data Rows Collection
If objDT.Rows.Count <> 0 Then

    'Step 14-Iterate through collection and extract data from a row.
    For Each objRow In objDT.Rows
      'Step 14b-Extract data from a row.  Use Item method to get the column data
                strProduct_ID = CStr(objDR.Item(0))
                strProduct_Name = CStr(objDR.Item(1))
                strProduct_Description = CStr(objDR.Item(2))
                dProduct_Date = CDate(objDR.Item(3))
                decProduct_Cost = CStr(objDR.Item(4))
                decProduct_Sale_Price = CStr(objDR.Item(5))

      'Step 15-Now that you have the data for a record or row, do what you want
 MessageBox.Show(strProduct_ID & strProduct_Name & strProduct_Description & dProduct_Date
& decProduct_Cost & decProduct_Sale_Price)

    Next

 Else
            'Step 16-No data returned, Record not found.  Do what you want here!
            MessageBox.Show("No Products found")

 End If
```

Step 6: Clean up

- Now clean up and destroy objects:

```
'Step 17-Terminate objects
objCmd.Dispose()
objCmd = Nothing
objDA.Dispose()
objDA = Nothing
objDT.Dispose()
objDT = Nothing
objRow = Nothing
objConn.Close()
objConn.Dispose()
objConn = Nothing
```

Example 15 - SELECT Query that Return TWO Tables using DataSet (METHOD II)

- Problem Statement:
  - We will show a different implementation of the previous example. Execute two queries to return TWO TABLES. Return the *Customer* & Product Tables. User the Parameters Collection to manage the parameters. Use a DataSet to store the multiple tables

- The code to handle multiple records is identical as the previous code. Since a collection is being used to store the Rows, the For Each Loop will either return one row or multiple rows.
- Steps to create data access code to use the **DataAdapter** & **DataSet** Objects in your applications are as follows:

Step 1: Verify that the ADO.NET Provider and Data mechanism are imported into your code

```
Imports System.Data
Imports System.Data.OleDb  'OLEDB Provider
```

Step 2: We create the Connection, Command object and set Parameters as necessary.

- Code:

```
'Step 1-Create Connection, assign Connection to string & open it
Dim strConn As String = "Provider=SQLOLEDB;Data Source=SATURN12\SQLEXPRESS;" & _
                        "Database=smallbusinessdb;User ID=dbuser01;Password=dbpassword01"

'Step 2-Create Connection object, Pass the string as an argument to the constructor
Dim objConn As New OleDbConnection(strConn)

'Step 3-Open the Connection
objConn.Open()
```

```
'Step 4-Create SQL string.  Here we have 2 QUERIES, to search 2 Tables
            Dim strSQL As String = "SELECT * FROM Customer WHERE Customer_ID = ?;" & _
                                   "SELECT * FROM Product WHERE Product_ID = ?"
'Step 5-Create Command object, pass string and connection object as arguments
Dim objCmd As New OleDbCommand(strSQL, objConn)
```

```
'Step 6-Add Parameter to Collection & Set Value to variable storing data
objCmd.Parameters.Add("Customer_ID", OleDbType.Integer).Value = txtIDNum.txt
```

Step 3: We create DataAdapter, DataSet Object and execute query and populate DataSet.

- Code:

```
'Step 7-Create DataAdpater object
Dim objDA As New OleDbDataAdapter(objCmd)

'Step 8-Create DataAdpater object
Dim objDS As New DataSet()

'Step 9-Call Fill method, pass DataSet object as argument
objDA.Fill(objDS)

 'Step 5-Name tables based on order of query
 objDS.Tables(0).TableName = "Customer"
 objDS.Tables(1).TableName = "Product"
```

Step 4: Retrieving the Data Stored in the DataTable Data Row Collection.

- Now we need to iterate through the FIRST TABLE, table(0) to extract the ROWS for the first table in the **DataTable** Collection:

```
'Step 11-Get first Table object from DataTable Collection
Dim objDT As DataTable = objDS.Tables.Item(0)

'Step 12-Create Row object POINTER to point to each ROW OBJECT
Dim objRow As DataRow

'Step 13-Loop through DataTables Collection to access each table(i)
For Each objDT In objDS.Tables
    'Step 14-Loop throgh data row collection to access row in table(i)
     For Each objRow In objDataTable.Rows

         'Step 15-Access Table(0) and display data
          If objDT.TableName = "Customer" Then
               If objDT.Rows.Count <> 0 Then
                  'Step 16-Extract data from a row.
                   strIDNumber = CStr(objDR.Item(0))
                   strName = CStr(objDR.Item(1))
                   dBirthDate = CDate(objDR.Item(2))
                   strAddress = CStr(objDR.Item(3))
                   strPhone = CStr(objDR.Item(4))
                   intAge = CInt(objDR.Item(5))
               Else
                   'Step 17-Display no record found
                   MessageBox.Show("No Customers found")
               End If
           End If
           'Step 18-Access Table(1) and display data
           If objDT.TableName = "Product" Then
                If objDT.Rows.Count <> 0 Then
                  'Step 19-Extract data from a row.
                  strProduct_ID = CStr(objDR.Item(0))
                  strProduct_Name = CStr(objDR.Item(1))
                  strProduct_Description = CStr(objDR.Item(2))
                  dProduct_Date = CDate(objDR.Item(3))
                  decProduct_Cost = CStr(objDR.Item(4))
                  decProduct_Sale_Price = CStr(objDR.Item(5))
                Else
                    'Step 20-Display no record found
                     MessageBox.Show("No Products found")
                End If
            End If
     Next
Next
```

Step 5: Clean up

- Now clean up and destroy objects:

```
'Step 17-Terminate objects
objCmd.Dispose()
objCmd = Nothing
objDA.Dispose()
objDA = Nothing
objDT = Nothing
objRow = Nothing
objConn.Close()
objConn.Dispose()
objConn = Nothing
```