Unit 5

Advanced Concepts in VB.Net

Topic Contents

- **Object Oriented Programming-** Creating Classes , Objects, Fields, Properties, Methods, Events , Constructors and destructors
- **Exception Handling-** Models, Statements
- **File Handling-** Using File Stream Class, File Mode, File Share, File Access Enumerations, Opening or Creating Files with File Stream Class, Reading and Writing Text using StreamReader and StreamWriter Classes
- **Data Access with ADO.Net –** What are Databases? Data Access with Server Explorer, Data Adapter and DataSets, ADO.NET Objects and Basic SQL.

Object-Oriented Programming

Classes and Objects

A class is a blueprint, a template, a specification, a pattern, or any other founding definition of an object. Objects absolutely depend on classes; without classes, you don't have objects. This should be clear from Figure 1, which shows that a class must exist before you have an object, just as the egg must come before you can have a chicken.

Figure 1: Classes are the blueprints of objects

It's easy to create classes and objects in Visual Basic. To create a class, you only need to use the **Class** statement, which, like other compound statements in Visual Basic, needs to end with **End Class**:

```
Public Class DataClass
        ⋮
End Class
```

This creates a new class named **DataClass**. You can create an object of this class, **data**, like this note that you must use the **New** keyword to create a new instance of a class:

```
Dim data As New DataClass()
```

You also can do this like this:

```
Dim data As DataClass = New DataClass()
```

Fields, Properties, Methods, and Events

Fields, Properties, Methods, and Events are called the members of a class. Inside the class, members are declared as either Public, Private, Protected, Friend, or Protected Friend:

- **Public**— Gives variables public access, which means there are no restrictions on their accessibility.
- **Private**— Gives variables private access, which means they are accessible only from within their class, including any nested procedures.
- **Protected**— Gives variables protected access, which means they are accessible only from within their own class or from a class derived from that class. Note that you can use Protected only at class level (which means you can't use it inside a procedure), because you use it to declare members of a class.
- **Friend**— Gives variables friend access, which means they are accessible from within the program that contains their declaration, as well as anywhere else in the same assembly.
- **Protected Friend**— Gives variables both protected and friend access, which means they can be used by code in the same assembly, as well as by code in derived classes.

The fields of a class, also called the class's data members, are much like built-in variables (although they also may be constants). For example, I can declare a field named value to the DataClass class we just saw by declaring a variable with that name:

```
Public Class DataClass
    Public value As Integer
End Class
```

Now I can refer to that field in an object of this class using the familiar object.field syntax of Visual Basic:

```
Dim data As New DataClass()
data.value = 5
```

You also can make fields hold constant values with Const:

```
Public Class Class1
    Public Const Field1 As Integer = 0
        ⋮
End Class
```

Using fields like this can give you direct access to the data stored inside an object, and that's unusual in OOP because you usually want to check the data being stored in your objects to make sure it's legal first. An easy way of guarding access to the data in your objects is to use *properties.*

Properties are retrieved and set like fields, but are handled with the **Property Get** and **Property Set** procedures, which provide more control on how values are set or returned.

Methods represent the object's built-in procedures. For example, a class named **Animal** may have methods named **Sleeping** and **Eating**. You define methods by adding procedures, either Sub routines or functions, to your class; for example, here's how I might implement the **Sleeping** and **Eating** methods:

```
Public Class Animal
    Public Sub Eating()
        MsgBox("Eating...")
    End Sub

    Public Sub Sleeping()
        MsgBox("Sleeping...")
    End Sub
End Class
```

Now I can create a new object of the **Animal** class and call the **Eating** method in the familiar way:

```
Dim pet As New Animal()
pet.Eating()
```

Creating Objects

You can create objects of a class using the **Dim** statement; this statement is used at module, class, structure, procedure, or block level:

```
[ <attrlist> ] [{ Public | Protected | Friend | Protected Friend |
Private | Static }] [ Shared ] [ Shadows ] [ ReadOnly ] Dim
[ WithEvents ] name [ (boundlist) ] [ As [ New ] type ] [ = initexpr ]
```

Here are the parts of this statement:

- ***attrlist***—A list of attributes that apply to the variables you're declaring in this statement. You separate multiple attributes with commas.
- **Public**—Gives variables public access, which means there are no restrictions on their accessibility. **Protected**—Gives variables protected access, which means they are accessible only from within their own class or from a class derived from that class.

- **Friend**—Gives variables friend access, which means they are accessible from within the program that contains their declaration, as well as anywhere else in the same assembly.
- **Protected Friend**—Gives variables both protected and friend access, which means they can be used by code in the same assembly, as well as by code in derived classes.
- **Private**—Gives variables private access, which means they are accessible only from within their declaration context (usually a class), including any nested procedures.
- **Static**—Makes variables static, which means they'll retain their values, even after the procedure in which they're declared ends.
- **Shared**—Declares a shared variable, which means it is not associated with a specific instance of a class or structure, but can be shared across many instances.
- **Shadows**—Makes this variable a shadow of an identically named programming element in a base class. A shadowed element is unavailable in the derived class that shadows it.
- **ReadOnly**—Means this variable can only be read and not written.
- **WithEvents**—Specifies that this variable is used to respond to events caused by the instance that was assigned to the variable.
- ***name***—The name of the variable. You separate multiple variables by commas.
- ***boundlist***—Used to declare arrays; gives the upper bounds of the dimensions of an array variable. Multiple upper bounds are separated by commas. An array can have up to 60 dimensions.
- **New**—Means you want to create a new object immediately. If you use **New** when declaring an object variable, a new instance of the object is created.
- ***type***—The data type of the variable. Can be **Boolean**, **Byte**, **Char**, **Date**, **Decimal**, **Double**, **Integer**, **Long**, **Object**, **Short**, **Single**, or **String** ; or the name of an enumeration, structure, class, or interface.
- ***initexpr***—An initialization expression which is evaluated and the result is assigned to the variable when it is created.

When you create a new object from a class, you use the **New** keyword. You can do that in either of these ways:

```
Dim employee As New EmployeeClass()
Dim employee As EmployeeClass = New EmployeeClass()
```

Creating Constructors

Constructors are special methods that provide control over the **initialization** of Objects. A constructor is nothing more than a subroutine named 'New'. When the class is instantiated, New (constructor) is fired. We can place the startup code just like we do in Form_Load in windows applications and Page_Load in web applications.

There are two types of constructors.

1. Shared constructors
2. Instance constructors

Implementation of Shared Constructors

Shared constructors are used to initialize the **shared variables** of a type. Shared variables are created using the Shared keyword and store values that can be shared by all the instances of a class. Shared constructors have an **implicit** public access. A shared constructor will not run more than once during a single execution of a program.

The following example is an illustration of the shared constructor.

```
Public Class class1
Shared x As Integer

Shared Sub New()
x=0
End Sub
End Class
```

Instance Constructor in Visual Basic

Instance constructors are used to initialize variables that are declared with Dim, Public, Private, Friend, Protected, and Protected Friend keywords. Write the following code in the class module.

```
Public Class ItemClass
 Private ItemCode As String
 Private ItemName As String
 Private ItemDescription As String
 Private ItemCategory As String
 Private ItemPrice As Single
 Private ItemUnit As String

 Public Sub New(ByVal Category As string)
       ItemCategory = Category
```

```
 End Sub

End Class
```

Creating Data Members:

The fields of a class, also called the class's data members, are much like built-in variables.

```
Public Class DataClass
    Public value As Integer
End Class
```

Now I can refer to that field in an object of this class using the familiar *object.field* syntax of Visual Basic:

```
Dim data As New DataClass()
data.value = 5
```

You also can make fields hold constant values with **Const**:

```
Public Class Class1
    Public Const Field1 As Integer = 0
        ⋮
End Class
```

Creating Properties

Using fields like this can give you direct access to the data stored inside an object, and that's unusual in OOP because you usually want to check the data being stored in your objects to make sure it's legal first. An easy way of guarding access to the data in your objects is to use *properties.*

Properties are retrieved and set like fields, but are handled with the **Property Get** and **Property Set** procedures, which provide more control on how values are set or returned.

```
Module Module2
    Private PropertyValue As String
    Public Property Prop1() As String
        Get
            Return PropertyValue
        End Get
        Set(ByVal Value As String)
            PropertyValue = Value
        End Set
    End Property
End Module
```

Note that you can make properties write-only with the **WriteOnly** keyword (and you must omit the **Get** method):

```
Module Module2
    Private PropertyValue As String
    Public WriteOnly Property Prop1() As String
        Set(ByVal Value As String)
            PropertyValue = Value
        End Set
    End Property
End Module
```

You can make properties read-only with the **ReadOnly** keyword (and you must omit the **Set** method):

```
Module Module2
    Private PropertyValue As String
    Public ReadOnly Property Prop1() As String
        Get
            Return PropertyValue
        End Get
    End Property
End Module
```

Creating Events

You can design and support your own events using OOP in Visual Basic, using the **Event** statement:

```
[ <attrlist> ] [ Public | Private | Protected | Friend |
Protected Friend] [ Shadows ] Event eventname[(arglist)]
[ Implements interfacename.interfaceeventname ]
```

Here are the parts of this statement:

- ***attrlist***—Optional. List of attributes that apply to this event. Separate multiple attributes by commas.
- **Public**—Optional. Events declared **Public** have public access, which means there are no restrictions on their use.
- **Private**—Optional. Events declared **Private** have private access, which means they are accessible only within their declaration context.
- **Protected**—Optional. Events declared **Protected** have protected access, which means they are accessible only from within their own class or from a derived class.
- **Friend**—Optional. Events declared **Friend** have friend access, which means they are accessible only within the program that contains the its declaration.
- **Protected Friend**—Optional. Events declared **Protected Friend** have both protected and friend accessibility.
- **Shadows**—Optional. Indicates that this event shadows an identically named programming element in a base class.
- ***eventname***—Required. Name of the event.
- ***interfacename***—The name of an interface.

- ***interfaceeventname***—The name of the event being implemented.

```
Public Class ClickTrack

        Public Event ThreeClick(ByVal Message As String)
        Public Sub Click()
            Static ClickCount As Integer = 0
            ClickCount += 1
            If ClickCount >= 3 Then
                ClickCount = 0
                RaiseEvent ThreeClick("You clicked three times")
            End If
        End Sub
    End Class
```

To raise an event we use the following code:

```
Dim WithEvents tracker As New ClickTrack()
tracker.ThreeClick
```

Exception handling

Overview

Exception handling is crucial since the robustness of software depends on how effectively a program deals with exceptions. Basically exceptions are the run time error.

Why do we need Exception handling?

We inject exception–handling code into our programs to trap errors and "handle" them "gracefully" and to prevent a program from terminating unexpectedly. Some errors don't result in an application crashing. But if we were to allow a program to continue mortally wounded it would present undesirable results to the user or risk trashing data. Instead SEH anticipates the impending disaster and presents a viable rescue plan.

Visual Basic .NET provides us with the facility to catch all types of exceptions. We can code specific handlers to catch only certain types of exceptions, or exceptions generated by related types, or classes.

To summarize, we need exception handling for the following reasons:

- Applications need a facility to catch a method's inability to complete a task.
- To process exceptions caused by a process accessing any functionality in methods, libraries, or classes where those elements are unable to directly handle any exception that arises as a result of the access and the ensuing computation.
- To process exceptions caused by components that are not able to handle any particular exception in their own processing space.

- To ensure that large and complex programs (actually, any Visual Basic program) can consistently and holistically provide program−wide exception handling.

Structured Exception Handling

Structured Exception Handling (SEH) is new to the Visual Basic language. Although it prevents further execution of troublesome code in a manner similar to the familiar **On Error Goto** construct, it is a very different construct to **On Error Goto**. When a program encounters an error in classic VB, all further processing of statements after the violating line is suspended, and the execution is transferred to a region in the current routine where the programmer can deal with the error. That's where the similarity ends.

The concept of SEH is straightforward. Code is optionally enclosed in a block that sets up a guarded perimeter around the code to be executed. This guarded code is everything that falls between the **Try** and **Catch** keywords. Here is an example.

```
13 Try
14 GetTrim = Call(SetFalloff) <- begin protected code
15
16 <-end protected code
17 Catch
18 End Try
```

As soon as an exception is raised, all further execution of the method's code is halted and transferred to a catchment area. The catchment area is the **Catch** block between **Try** and **End Try**.

File Handling

The .NET Framework provides an impressive range of IO namespaces that contain dozens of classes used for writing, reading, and streaming all manner of text, characters, and binary data, as well as file, folder, and path support.

The files you work with in your programs are typically ordered collections of bytes, representing characters on a file system. Streams, on the other hand, are continuous "rivers" of data, writing to and reading from various devices.

Stream is an abstract class and has been extended in a variety of specialized child classes. The **Stream** class and its children provide a facility for handling data, blocks of bytes, without having to care about what happens down in the basement of the operating system.

The following classes derive from **Stream**:

- **BufferedStream** Reads and writes to other **Stream** objects. This class can be derived from and instantiated.
- **FileStream** Bridges a **Stream** object to a file for synchronous and asynchronous read and write operations. This class can be derived from and instantiated.
- **MemoryStream** Creates a **Stream** in memory that can read a block of bytes from a current stream and write the data to a buffer. This class can be derived from and instantiated.
- **CryptoStream** Defines a **Stream** that bridges data objects to cryptographic services. This class can be derived from and instantiated.
- **NetworkStream** Defines a **Stream** that bridges data objects to network services. This class can be derived from and instantiated.

FileStream

FileStream objects can be used to implement all of the standard input, output, and error stream functionality. With these objects, you can read and write to file objects on the file system. With it you can also bridge to various file–related operating system handles, such as pipes, standard input, and standard output. You can use the **FileStream** class to open or create files, and then use other classes, like **BinaryWriter** and **BinaryReader**, to work with the data in the file.

Noteworthy public properties of *FileStream* objects.

Property	Description
CanRead	Determines if the stream supports reading.
CanSeek	Determines if the stream supports seeking.
CanWrite	Determines if the stream supports writing.
Handle	Gets the operating system file handle for the stream's file.
IsAsync	Determines if the stream was opened asynchronously or synchronously.
Length	Gets the length of the stream in bytes.
Name	Gets the name of the file stream passed to the constructor.
Position	Gets/sets the position in this stream.

Noteworthy public methods of *FileStream* objects.

Method	Description
BeginRead	Starts an asynchronous read operation.
BeginWrite	Starts an asynchronous write operation.
Close	Closes a file, making it available in Windows to any other program.
EndRead	Waits for an asynchronous read operation to finish.
EndWrite	Ends an asynchronous write operation, waiting until the operation has finished.
Flush	Flushes all buffers for this stream, writing any buffered data out to its target (such as a disk file).
Lock	Withholds any access to the file to other processes.
Read	Reads a block of bytes.
ReadByte	Reads a byte from the file.
Seek	Sets the current read/write position.
SetLength	Sets the length of the stream.
Unlock	Gives access to other processes to a file that had been locked.
Write	Writes a block of bytes to this stream.
WriteByte	Writes a byte to the current read/write position.

Using the *FileMode* Enumeration

When you open a file with the **FileStream** class, you specify the file mode you want to use—for example, if you want to create a new file, you use the file mode **FileMode.Create**. The various possible file modes are part of the **FileMode** enumeration; you can find the members of this enumeration in Table 5.3.

Table 5.3: Members of the *FileMode* enumeration.

Member	Means
Append	Opens a file and moves to the end of the file (or creates a new file if the specified file doesn't exist). Note that you can only use **FileMode.Append** with **FileAccess.Write**.
Create	Creates a new file; if the file already exists, it is overwritten.

CreateNew	Creates a new file; if the file already exists, an **IOException** is thrown.
Open	Opens an existing file.
OpenOrCreate	Open a file if it exists; or create a new file.
Truncate	Open an existing file, and truncate it to zero length so you can write over its data.

Using the *FileAccess* Enumeration

When you open files with the **FileStream** class, you can specify the file mode (see the previous topic) and *access*. The access indicates the way you're going to use the file-to read from, to write to, or both. To indicate the type of file access you want, you use members of the **FileAccess** enumeration. You can find the members of the **FileAccess** enumeration in Table 5.4.

Table 5.4: Members of the *FileAccess* enumeration.

Member	Means
Read	Gives read access to the file, which means you can read data from the file.
ReadWrite	Gives both read and write access to the file, which means you can both read and write to and from a file.
Write	Gives write access to the file, which means you can write to the file.

Using the *FileShare* Enumeration

When you open a file, you can specify the file-sharing mode you want to use in some of the **FileStream** constructors (you don't have to specify a file-sharing mode with other **FileStream** constructors). For example, if you want to allow other programs to read a file at the same time you're working with it, you use the file-sharing mode **FileShare.Read**. The various possible file-sharing modes are part of the **FileShare** enumeration, and you can find the members of this enumeration in Table 5.5.

Table 5.5: Members of the *FileShare* enumeration.

Member	Means
None	The file cannot be shared. Other processes cannot access it.
Read	The file also may be opened by other processes for reading.

ReadWrite	The file also may be opened by other processes for reading and writing.
Write	The file also may be opened by other processes for writing.

Opening or Creating a File with the *FileStream* Class

When you want to open or create a file, you use the **FileStream** class, which has many constructors, allowing you to specify the file mode (for example, **FileMode.Create**), file access (such as **FileAccess.Write**), and/or the file-sharing mode (such as **FileShare.None**), like this (these are only a few of the **FileStream** constructors):

```
Dim fs As New System.IO.FileStream(String, FileMode)
Dim fs As New System.IO.FileStream(String, FileMode, FileAccess)
Dim fs As New System.IO.FileStream(String, FileMode, FileAccess, FileShare)
```

Example:

Let us create a file named file.txt and opening it for writing with a FileStream object; We will set the file mode to Create to create this new file, and explicitly set the file access to Write so we can write to the file:

```
Imports System
Imports System.IO

Public Class Form1
    Inherits System.Windows.Forms.Form

    'Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button1.Click
        Dim fs As New System.IO.FileStream("file.txt", FileMode.Create, _
            FileAccess.Write)
        ⋮
```

Using the *StreamWriter* Class

After you've opened a file for writing using the **FileStream**, you can create a **StreamWriter** object to write text to the file.

Table 5.6: Noteworthy public properties of *StreamWriter* objects.

Property	Description
AutoFlush	Gets/sets if the **StreamWriter** will flush its buffer after **Write** or **WriteLine** operation.
BaseStream	Gets the base stream for this stream, giving you access to the base stream's

	properties and methods.
Encoding	Gets the character encoding for this stream.

Table 5.7: Noteworthy public methods of *StreamWriter* objects.

Method	Description
Close	Closes the current stream.
Flush	Flushes all buffers for the stream writer, writing any buffered data to the base stream.
Write	Writes data to the stream.

Writing Text with the *StreamWriter* Class

The following example will create a file named file.txt and then write some text in the file.

```
Imports System
Imports System.IO

Public Class Form1
    Inherits System.Windows.Forms.Form

    'Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button1.Click
        Dim fs As New System.IO.FileStream("file.txt", FileMode.Create, _
            FileAccess.Write)
        Dim w As New StreamWriter(fs)
        w.BaseStream.Seek(0, SeekOrigin.End)
        w.WriteLine("Here is the file's text.")
        w.Write("Here is more file text." & ControlChars.CrLf)
        w.WriteLine("And that's about it.")
        w.Flush()
        w.Close()
        :
    End Sub
End Class
```

Using the *StreamReader* Class

You can use the **StreamReader** class to read text data from files; here's the hierarchy of this class:

Table 5.7: Noteworthy public properties of *StreamReader* objects

Property	Means
BaseStream	Holds the underlying stream, giving you access to that stream's properties and

	methods.
CurrentEncoding	Gets the character encoding for the stream reader.

Table 5.8: Noteworthy public methods of *StreamReader* objects.

Method	Means
Close	Closes the stream reader.
DiscardBufferedData	Discards the data in the buffer.
Peek	Looks ahead and returns the next available character (but does not actually read it as **Read** would, so does not advance the read/write position). Returns -1 if there is no more data waiting to be read.
Read	Reads the next character or characters.
ReadLine	Reads a line of text from the stream, returning that data as a string.
ReadToEnd	Reads from the current position to the end of the stream.

Reading Text with the *StreamReader* Class

The following example will read text from the file and then display the text in the textbox.

```
Imports System
Imports System.IO

Public Class Form1
    Inherits System.Windows.Forms.Form

    'Windows Form Designer generated code

    Private Sub Button1_Click(ByVal sender As System.Object, _
        ByVal e As System.EventArgs) Handles Button1.Click

        fs = New System.IO.FileStream("file.txt", FileMode.Open, _
            FileAccess.Read)

        Dim r As New StreamReader(fs)
        r.BaseStream.Seek(0, SeekOrigin.Begin)
        While r.Peek() > -1
            TextBox1.Text &= r.ReadLine() & ControlChars.CrLf
        End While
        r.Close()
    End Sub
End Class
```

Data Access with ADO.NET

What Are Databases?

A database is a collection of related data. By data, we mean known facts that can be recorded and that have implicit meaning. For example, consider the names, telephone numbers, and addresses of the people you know. This is a collection of related data with an implicit meaning and hence is a database. A collection of records—that is, rows of records, where each column is a field— becomes a table. Database is just a collection of one or more tables.

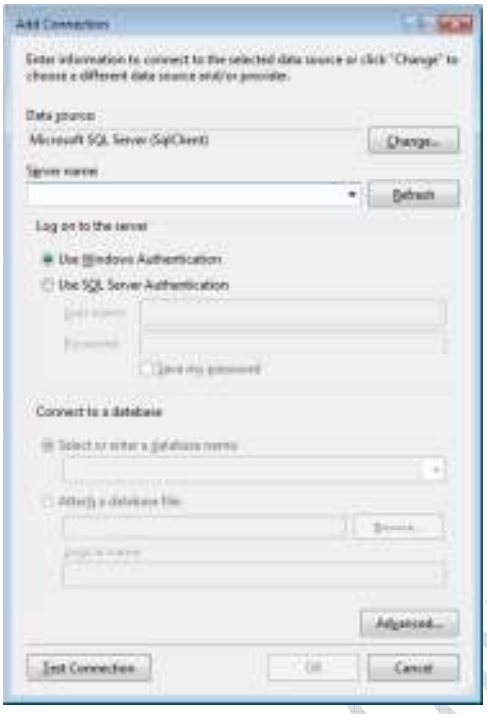
Visual Basic .NET uses ADO.NET (ADO stands for ActiveX Data Objects) as its primary data access and manipulation protocol. There are plenty of objects available in ADO.NET, but at root, they're not difficult to use in practice.

Here's what happens when you use ADO.NET—you first get a connection to a data source, which means using a data provider to access a database. After you have a connection to a data source, you create a *data adapter* to work with that data. The data adapter is what actually applies your SQL statements to a database and causes your datasets to fill with data. Once you have a data adapter, you can generate a *dataset* using that adapter. So those are the three objects that it's essential to know about: data connections to connect to the database, data adapters to execute SQL with, and datasets to store the data—as returned from data adapters—that your code will actually work on.

Accessing Data with the Server Explorer

To work with a database, you need a connection to that database. In Visual Basic, the Server Explorer lets you work with connections to various data sources. To display the Server Explorer if it's not already visible, use the View|Server Explorer menu item, or press Ctrl+Alt+S. 0.This tool lets you create and examine data connections, including connections to Web servers; you can see connections to various databases in the Server Explorer already.

When Visual Basic .NET is installed, it searches your local computer for database servers and adds them to the Server Explorer automatically. To add additional servers to the Server Explorer, you select the Tools|Connect to Server menu item or right-click the Servers node that appears at the bottom of the Server Explorer, and select the Add Server menu item. This opens the Add Server dialog, which lets you enter new database servers by computer name or IP address on the Internet. When you subsequently create data connections, you can specify what server to use, as you see in the drop-down list box in Figure 5.2.



To access data from database just drag the **database** table onto the main form. This automatically creates the **SqlConnection1** and **SqlDataAdapter1** objects you see in the component tray .

Now it's time to generate the dataset that holds the data from the data adapter. To do that, just select the Data|Generate Dataset menu item, or right-click **SqlDataAdapter1** and select the Generate Dataset menu item. This displays the Generate Dataset dialog.

Click the New option to create a new dataset. I'll stick with the default name given to this new dataset object, **DataSet1**; make sure the **database** table checkbox is checked, as well as the "Add this dataset to the designer" checkbox, then click OK. Doing so adds a new dataset, **DataSet11**,

To display the data in the dataset, I'll use a data grid control, and set the data grid's **DataSource** property to **Data11(not DataSet11)** and its **DataMember** property to **table**, which will be displayed automatically as an option when you click that property. This connects the data in the dataset to the data grid.

To fill the dataset with data from the data adapter write the following code in form load event

```
Private Sub Form1_Load(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles MyBase.Load
    DataSet11.Clear()
    SqlDataAdapter1.Fill(DataSet11)
End Sub
```

Accessing Data with Data Adaptors and Datasets

In the previous example, we dragged an entire data table from the Server Explorer to a form, but often you'll want to look at only a few fields in a table, or otherwise customize what you want to do with a table before working with its data. To do that, you can create a data adapter yourself.

To see how this works, just click the Data tab in the toolbox now. In this case, I'll drag an **OleDbDataAdapter** object from the toolbox to the main form. Doing so opens the Data Adapter Configuration Wizard. This wizard will let you customize your data adapter as you want, which usually means we can create the SQL statement using this adapter. Now create a new dataset using this data adapter by clicking Data|Generate Dataset menu item, and connect the new dataset to a data grid using the **DataSource** and **DataMember** properties. Now add a Load Button and write the following code:

```
Private Sub btnLoad_Click(ByVal sender As System.Object, _
    ByVal e As System.EventArgs) Handles btnLoad.Click
    DataSet11.Clear()
    OleDbDataAdapter1.Fill(DataSet11)
End Sub
```

Here's a summary of the steps we took in this example:

1. Create a data connection or use an existing data connection.
2. Drag an **OleDBAdaptor** object onto a form; creates connection and adaptor objects.
3. Use the Data Adapter Configuration Wizard to configure the data adapter and create the SQL you want.
4. Generate a dataset.
5. Bind the dataset to controls.
6. Fill the dataset in code.

Overview of ADO.NET Objects

Here's a list of the most common ADO.NET objects:

- ***Data connection objects***—To start working with a database, you must have a data connection. A data adapter needs a connection to a data source to read and write data, and it uses **OleDbConnection** or **SqlConnection** objects to communicate with a data source.
- ***Data adapters***—Data adapters are a very important part of ADO.NET. You use them to communicate between a data source and a dataset. You typically configure a data adapter

with SQL to execute against the data source. The two types of data adapters are **OleDbDataAdapter** and **SqlDataAdapter** objects.

- ***Command objects***—Data adapters can read, add, update, and delete records in a data source. To allow you to specify how each of these operations work, a data adapter contains command objects for each of them. Data adapters support four properties that give you access to these command objects: **SelectCommand**, **InsertCommand**, **UpdateCommand**, and **DeleteCommand**.
- ***Datasets***—Datasets store data in a disconnected cache. The structure of a dataset is similar to that of a relational database; it gives you access to an object model of tables, rows, and columns, and it contains constraints and relationships defined for the dataset. Datasets are supported with **DataSet** objects.
- ***DataTable objects***—**DataTable** objects hold a data table from a data source. Data tables contain two important properties: **Columns**, which is a collection of the **DataColumn** objects that represent the columns of data in a table, and **Rows**, which is a collection of **DataRow** objects, representing the rows of data in a table.
- ***Data readers***— **DataReader** objects hold a read-only, forward-only (i.e., you can only move from one record to the succeeding record, not backwards) set of data from a database. Using a data reader can increase speed because only one row of data is in memory at a time. See "Using a Data Reader" in Chapter 22
- ***Data views***—Data views represent a customized view of a single table that can be filtered, searched, or sorted. In other words, a data view, supported by the **DataView** class, is a data "snapshot" that takes up few resources.
- ***DataRelation objects***—**DataRelation** objects specify a relationship between parent and child tables, based on a key that both tables share.
- ***DataRow objects***—**DataRow** objects correspond to a particular row in a data table. You use the **Item** property to get or set a value in a particular field in the row. See "Creating Data Rows in Code" in Chapter 22.
- ***DataColumn objects***—**DataColumn** objects correspond to the columns in a table. Each object has a **DataType** property that specifies the kind of data each column contains, such as integers or string values..

Answer the following Question:

1. What is a class? Explain with an example how you create a class.
2. Write a short note on Constructors in VB.net
3. Explain with an example how do you declare and call functions in a class.
4. What is SEH? Explain in detail each keyword with an example.
5. Write a short note on ADO.Net Objects
6. What is server explorer? How do you access data from server explorer?
7. What is data adapter? How do you access data from the data adapter object?