Let's now learn about polymorphism.

Polymorphism

Polymorphism is one of the important features of OOP that is used to exhibit different forms of any particular procedure. With the help of polymorphism, you can use one procedure in many ways as per your requirements. For example, you can make a procedure for calculating the area of a geometrical figure and can calculate the area of a circle, triangle, or rectangle using the same procedure with different parameters for each geometrical figure.

The advantages of polymorphism are as follows:

- Allows you to invoke methods of a derived class through base class reference during runtime
- Provides different implementations of methods in a class that are called through the same name

Before learning more about polymorphism, first we need to know that an object of a derived class can hold the reference of its base class. Let's take an example where `DerivedClass` class inherits the `BaseClass` class, as shown in Listing 12.7:

Listing 12.7: Inheriting a Class from Another

```
class BaseClass {
  public void BaseMethod() {
        …//Method Body
  }
}
class DerivedClass : BaseClass {
  public void DerivedMethod() {
        …//Method Body
  }
}
```

However, you can write the following statement:

```
BaseClass bc=new DerivedClass();
```

In this code, a reference of the `BaseClass` class is holding an object (`bc`) of type `DerivedClass` class. In this case, we are taking the object of the `DerivedClass` class as an object of the `BaseClass` class. It is possible because we derive the `DerivedClass` class from the `BaseClass` class. Now, we can also write:

```
bc.BaseMethod();
```

But, it is incorrect to write:

```
bc.DerivedMethod(); //It shows error
```

Although, we have created an object of the `DerivedClass` class, but we cannot access the members of the `DerivedClass` class because the apparent class is `BaseClass` not `DerivedClass`.

There are two types of polymorphism, which are as follows:

- Static polymorphism/compile time polymorphism/overloading
- Dynamic polymorphism/run time polymorphism/overriding

Let's learn about the compile time and run time polymorphism in detail.

Compile Time Polymorphism/Overloading

When a compiler compiles a program, it knows the information about the method arguments and accordingly, it binds the appropriate method to an object at the compile time itself. This process is also called compile time polymorphism or early binding. You can implement compile time polymorphism through overloaded methods and operators. Overloaded methods are the methods that have same name but different signatures, such as number, type, and sequence of parameters. There are the following three types of compile time polymorphism:

- Method overloading
- Operator overloading
- Indexer overloading

Let's discuss these in detail.

Method Overloading

Method overloading is a concept in which a method behaves according to the number and type of parameters passed to it. In method overloading, you can define many methods with the same name but different signatures. A method signature is the combination of the method's name along with the number, type, and order of the parameters.

When you call overloaded methods, a compiler automatically determines which method should be used according to the signature specified in the method call. Method overloading is used when methods are required to perform similar tasks but with different input parameters. Note that only the return type does not play any important role in the method overloading, it must take consideration of the number or type of arguments passed to the method.

Let's create an application named `MethodOverload` (also available in the CD) to learn how to overload a method.

Listing 12.8 shows the code of the `MethodOverload` application:

Listing 12.8: Showing the Code of the MethodOverload Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MethodOverload {
  public class Shape {
        public void Area(int Side) {
              int SquareArea = Side * Side;
              Console.WriteLine("The Area of Square is: "+SquareArea);
        }
        public void Area(int Length, int Breadth) {
              int RectangleArea = Length  * Breadth ;
              Console.WriteLine("The Area of Rectangle is: " + RectangleArea);
        }
        public void Area(double Radius) {
              double CircleArea = 3.14 * Radius * Radius;
              Console.WriteLine("The Area of Circle is: "+CircleArea);
        }
        public double Area(double Base, double Height) {
              double TriangleArea = (Base * Height)/2;
              Console.WriteLine("The Area of Triangle is: "+TriangleArea);
              return TriangleArea;
        }
  }
  class MainClass {
        static void Main(string[] args) {
              Shape shape = new Shape();
              shape.Area(15);
              shape.Area(10, 20);
              shape.Area(10.5);
              shape.Area(15.5, 20.4);
              Console.Write("\nPress ENTER to quit...");
              Console.ReadLine();
        }
  }
}
```

In Listing 12.8, the `Area()` method of `Shape` class is overloaded for calculating the areas of `square`, `rectangle`, `circle`, and `triangle` shapes. In the Main() method, the Area() method is called multiple times by passing different arguments.

Figure 12.7 shows the output of Listing 12.8:

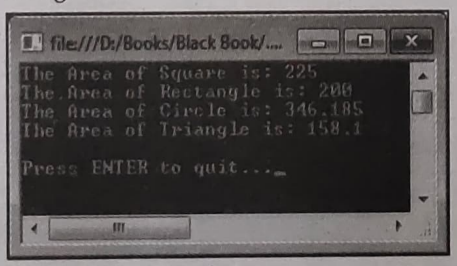

Figure 12.7: Showing the Output of the MethodOverload Application

Operator Overloading

All the operators have specified meaning and functionality, such as the + (plus) operator adds two numerals and the - (minus) operator subtracts one numeral from another. However, in C#, you can change the functionality of an operator by overloading them. For example, you can use the + operator to concatenate two strings. The mechanism of assigning a special meaning to an operator according to user defined data type, such as classes or structs, is known as operator overloading. It is not possible to overload all the operators. Table 12.1 shows a list of operators with their overloading status, which shows whether an operator can be overloaded or not:

Indexer Overloading

In C#, an indexer is used to treat an object as an array. It is used to provide index to an object to obtain value from the object. Implementing an indexer requires you to use brackets ([ ]) with an object to get and set a valu of the object. Indexers are declared as properties, with the difference that in case of indexers, you do not need t provide name to them. You need to use the this keyword to define an indexer.

The following code snippet shows how to define an indexer:

```
public string this[int Position] {
  get {
        return MyData[Position];
  }
  set {
        MyData[Position] = value;
  }
```

In the preceding code snippet, the get property is used to retrieve the index of the MyData member, while the set property is used to assign an index to the member. You can also see that there is no name assign to the indexer.

Similar to any other data members, indexers can also be overloaded. Let's create an application named `IndexerOverload` (also available in the CD) that shows how to overload an indexer. Listing 12.11 shows the code of the `IndexerOverload` application:

Listing 12.11: Showing the Code of the IndexerOverload Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace IndexerOverload {
  class MyClass {
        private string[] MyData;
        private int ArraySize;
        public MyClass(int size) {
              ArraySize = size;
              MyData = new string[size];
              for (int i = 0; i < size; i++) {
                    MyData[i] = "DataValue";
              }
        }
        public string this[int Position]
```

```
        {
                get {
                        return MyData[Position];
                }
                set {
                        MyData[Position] = value;
                }
        }
        public string this[string data] {
                get {
                        int Count = 0;
                        for (int i = 0; i < ArraySize; i++)
                        {
                                if (MyData[i] == data)
                                {
                                        Count=Count+1;
                                }
                        }
                        return Count.ToString();
                }
                set {
                        for (int i = 0; i < ArraySize; i++)
                        {
                                if (MyData[i] == data)
                                {
                                        MyData[i] = value;
                                }
                        }
                }
        }
        class MainClass {
                static void Main(string[] args) {
                        int size = 10;
                        MyClass MyIndexer = new MyClass(size);
                        MyIndexer [9] = "Hello";
                        MyIndexer [3] = "Welcome";
                        MyIndexer [5] = "Good Morning";
                        MyIndexer [7] = "Good Night";
                        MyIndexer ["DataValue"] = "Have a Nice Day";
                        Console.WriteLine("\nIndexer Output\n");
                        Console.WriteLine("-----------------");
                        for (int i = 0; i < size; i++)
                        {
                        Console.WriteLine("MyIndexer[{0}]: {1}", i, MyIndexer [i]);
                        }
                        Console.WriteLine("\nNumber of \"Have a Nice Day\" Entries: {0}",
                        MyIndexer ["Have a Nice Day"]);
                        Console.Write("\nPress ENTER to quit...");
                        Console.ReadLine();
                }
        }
    }
}
```

In Listing 12.11, we have overloaded indexers by passing different type of parameters. We have defined two indexers, with the `int` type and `string` type parameters, named `Position` and `data`, respectively. These indexers are used to display the values of different string data that is passed in the Main() method of the MainClass class.

Figure 12.10 shows the output of Listing 12.11:

Figure 12.10: Showing the Output of the IndexerOverload Application

Runtime Polymorphism/Overriding

Overriding is a feature that allows a derived class to provide a specific implementation of a method that is already defined in a base class. The implementation of method in the derived class overrides or replaces the implementation of method in its base class. This feature is also known as runtime polymorphism because the compiler binds a method to an object during the execution of a program (runtime) instead of during the compilation of the program. When you call a method, the method defined in the derived class is invoked and executed instead of the one in the base class. To invoke the method of a derived class that is already defined in the base class, you need to perform the following steps:

- Declare the base class method as `virtual`
- Implement the derived class method using the `override` keyword

The following code snippet shows how to override a method of a base class in a derived class:

```
class BaseClass {
  public virtual void ShowData() {
        //Method Body
  }
}
class DerivedClass:BaseClass {
  public override void ShowData() {
        //Method Body
  }
}
```

In the preceding code snippet, you can see that in the `BaseClass` class, the `ShowData()` method is marked with the `virtual` keyword and in the `DerivedClass` class, the `ShowData()` method is marked with the `override` keyword.

In C#, you can create virtual methods in a base class which can be overridden in a derived class to change the functionality of the base class method. You can declare a function as virtual in a base class by using the virtual keyword; however, to override this function in a derived class, you need to use the override keyword in derived class. Let's create an application named `MethodOverriding` (also available in the CD) shows an example of overriding a method using the virtual and override keywords. Listing 12.12 shows the code of the `MethodOverriding` application:

Listing 12.12: Showing the Code of the `MethodOverriding` Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MethodOverriding {
  class Program {
        static void Main(string[] args) {
```

```
            Person p = new Person();
            p.setAge(18);
            AdultPerson ap = new AdultPerson();
            ap.setAge(18);
            Console.WriteLine("Person Age: {0}", p.getAge());
            Console.WriteLine("AdultPerson Age: {0}", ap.getAge());
            Console.Write("\nPress ENTER to quit...");
            Console.ReadLine();
        }
    }
    public class Person {
        private int fAge;
        public Person() {
            fAge = 21;
        }
        public virtual void setAge(int age) {
            fAge = age;
        }
        public virtual int getAge() {
            return fAge;
        }
    }
    public class AdultPerson : Person
    {
        public AdultPerson() { }
        override public void setAge(int age)
        {
            if (age > 21)
            base.setAge(age);
        }
    }
}
```

In Listing 12.12, we have created the Person and AdultPerson classes, where the AdultPerson class is the derived class of the Person class. The Person class contains two virtual functions, setAge and getAge. The AdultPerson class is overriding the setAge() method of the base class by using the override keyword.

Figure 12.11 shows the output of Listing 12.12:

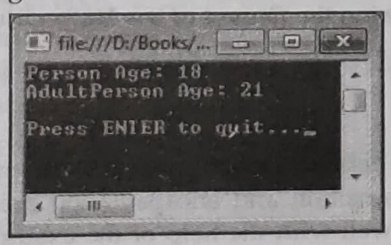

Figure 12.11: Showing the Output of the MethodOverriding Application

Consider, you want to derive a class from a base class and to redefine some methods contained in this base class. In such a case, you cannot declare the base class method as `virtual`. Then how can you override a method without declaring that method as `virtual` in the base class? This can be possible with the `new` operator. The new operator is used to override the base class method in the derived class without using the virtual keyword. The `new` operator tells the compiler that the derived class method hides the base class method.

Let's create an application named `MethodHiding` (also available in the CD) that shows how the derived class method hides the base class method using the `new` keyword. Listing 12.13 shows the code of the `MethodHiding` application:

Listing 12.13: Showing the Code of the `MethodHiding` Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MethodHiding {
  class BaseClass {
```

Figure 13.10: Showing the Output of the FixedDeclare Application

Delegates

A delegate is a special type of object that contains the details of a method rather than data. In C#, a delegate is a class type object, which is used to invoke a method that has been encapsulated into it at the time of its creation. A delegate can be used to hold the reference to a method of any class. It carries three important pieces of information—the name of the method on which it makes calls, the arguments (if any) of this method, and the return value (if any) of this method.

In the subsequent sections, you learn to create and use delegates. Then, you learn about multicast delegates, covariance, contravariance, and anonymous methods with delegates.

Creating and Using Delegates

Suppose you need to create a program that requires data, such as student information, to display it on a website. This data can be retrieved by calling a method without having to know at compile time which method is to be invoked. In this case, you need to create an object of delegate and encapsulate a reference to that method inside the delegate object. Following are the four steps to create and use a delegate in your program:

- Declaring a delegate
- Defining delegate methods

- Creating delegate objects
- Invoking delegate objects

Let's learn to perform these tasks in detail.

Declaring a Delegate

A delegate can be declared using the following syntax:

```
Access-modifier delegate return-type delegate-name(parameter-list);
```

In the preceding syntax, access-modifier is the modifier that controls the accessibility of the delegate and can be `public`, `private`, `protected`, or `internal`. The `delegate` keyword indicates that the declaration belongs to a delegate and the return-type is the return type of the delegate. The delegate-name is the name of the delegate, which can be any valid C# identifier and the parameter-list is the list of parameters that the delegate takes.

Using the preceding syntax, we can declare a delegate, named `Compute`, as follows:

```
public delegate void Compute(int x, int y);
```

Although the syntax of declaring a delegate is similar to that of declaring a method (without method body), the delegate actually represents a class type. As a delegate represents a class type, it can be declared at any place where a class can be defined—outside all classes or inside a class. Delegate types are implicitly sealed; and therefore, deriving a new type from a delegate type is not possible.

Defining Delegate Methods

A delegate method is any method whose signature (number and type of parameters and return type) matches the delegate signature exactly. It can be either a static method or an instance method. We can define a static method, named `Add()`, to be encapsulated into the `Compute` delegate, as follows:

```
public static void Add(int a, int b)
{
  Console.WriteLine("Sum = {0}", a + b);
}
```

Similarly, we can also define an instance method, named `Subtract()`, to be encapsulated into the `Compute` delegate, as follows:

```
public void Subtract(int a, int b)
{
  Console.WriteLine("Difference = {0}", a - b);
}
```

Creating Delegate Objects

You can create an object of a delegate using the following syntax:

```
Delegate-name object-name = new delegate-name(expression);
```

In the preceding syntax, delegate-name is the name of the delegate declared earlier whose object is to be created; object-name is the name of the delegate object; and expression is what you want to encapsulate into the delegate object and it can be the name of a method or an object of a delegate type. If you pass a method or an object of a delegate type to its constructor, its signature should match exactly with the signature of the delegate; otherwise, you receive a compile-time error.

Using the preceding syntax, you can create an object of the `Compute` delegate, named `cmp1`, to hold the reference of the (static) `Add()` method, as follows:

```
Compute cmp1 = new Compute(DelegateTest.Add);
```

Similarly, you can create another object of the `Compute` delegate, named `cmp2`, to hold the reference of the (instance) `Subtract()` method, as follows:

```
DelegateTest dt = new DelegateTest();
Compute cmp2 = new Compute(dt.Subtract);
```

Invoking Delegate Objects

A delegate object is invoked as a method is invoked. The following code snippet shows the syntax to declare a delegate-object:

```
Delegate-object(argument-list)
```

After learning about the control flow statements, let's learn about exception handling.

Exception Handling

Exception is a runtime error that arises because of some abnormal conditions, such as division of a number by zero, passing a string to a variable that holds an integer value. Capturing and handling of runtime errors is one of the important and crucial tasks for any programmer; but before discussing about the runtime errors, let's take a look at compile time errors. Compile time errors are those errors that occur during compilation of a program. It can happen due to bad coding or incorrect syntax. You can correct these compile time errors after looking at the error message that the compiler generates. On the other hand, runtime errors are those errors that occur during the execution of a program and therefore, at that time they cannot be corrected. Therefore, necessary actions need to be taken beforehand to prevent such types of errors. To do so, you should first identify the following two aspects:

- Find out those parts of a program which can cause runtime errors
- How to handle those errors, when they occur

Table 14.1 describes some common exception classes:

Table 14.1: Noteworthy Exception Classes

Exception Class	Description
SystemException	Represents a base class for other exception classes
AccessException	Occurs when a type member, such as a variable, cannot be accessed
ArgumentException	Defines an invalid argument to a method
ArgumentNullException	Occurs if a null argument is passed to a method that does not accept it
ArgumentOutOfRangeException	Occurs if an argument value is not within a range
ArithmeticException	Occurs if an Arithmetic overflow or Arithmetic underflow has occurred
ArrayTypeMismatchException	Occurs if you try to store the incorrect type of object in an array
BadImageFormatException	Occurs when the image file of a DLL file or an executable program is invalid
DivideByZeroException	Occurs if you try to divide a number by zero
FormatException	Occurs if there is an incorrect format of argument
IndexOutOfRangeException	Occurs if the index of an array is out of bound
InvalidCastExpression	Occurs if you try to cast an object to an invalid class
InvalidOperationException	Occurs if a method call is invalid for the current state of the object
MissingMemberException	Occurs if you try to dynamically access a member of a class that does not exist
NotFiniteNumberException	Occurs if a number is invalid or infinite
NotSupportedException	Occurs when a method that is invoked is not supported
NullReferenceException	Occurs if you try to make use of an unassigned reference
OutOfMemoryException	Occurs if an execution stops due to lack of memory
StackOverflowException	Occurs if a stack overflows

In C# 2010, exceptions can be handled by using the following two statements:

- The try…catch…finally statement
- The throw statement

Let's discuss these statements in detail.

The try...catch...finally Statement

C# provides three keywords for exception handling—`try`, `catch`, and `finally`. The `try` block encloses those statements that can cause exception; whereas, the `catch` block encloses the statements to handle the exception, if it occurs. Multiple `catch` blocks can exist for a single `try` block. All these `catch` blocks are used to handle different types of exceptions that are raised inside the `try` block. The statements enclosed in the

finally block are always executed, irrespective of the fact whether an exception occurs or not. There can be only one finally block for a try block.

If any exception occurs in the try block, the program control directly transfers to its respective catch block and later on to the finally block. In C#, catch and finally blocks are not compulsory. In a program, a try block can contain one or multiple catch blocks, only a finally block, or both catch and finally blocks. If no exception occurs inside the try block, the program control is transferred directly to the finally block.

The following code snippet shows the use of the try...catch...finally statement:

```
try
{
  div = 100 / number;
}
catch (DivideByZeroException de)
{
  Console.WriteLine("Exception Occured");
}
finally
{
  Console.WriteLine("Result is : " +div);
  Console.ReadLine();
}
```

In the preceding code snippet, when the statement inside the try block executes, the control may transfer to the catch block depending upon the value of the number variable. Suppose, the value of the number variable is 0, then DivideByZeroException is raised and the program control transfers to the catch block and executes the statements inside the catch block and after that the statements inside the finally block are executed. If the value of the number variable is other than zero, then the program control is not transferred to the catch block, it is then directly transferred to the finally block and statements inside the finally block are executed.

Let's create a console application named MyException (also available in the CD) that shows how to handle an exception. Listing 14.14 shows the code of the MyException application:

Listing 14.14: Showing the Code of the MyException Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MyException
{
    class Program
    {
        static void Main(string[] args)
        {
                int number = 0;
                int div = 0;
                try
                {
                    div = 100 / number;
                }
                catch (DivideByZeroException ex)
                {
                    Console.WriteLine("Exception Occured :" + ex.Message);
                }
                Console.WriteLine("Result is : " + div);
                Console.Write("\nPress ENTER to quit...");
                Console.ReadLine();
        }
    }
}
```

In Listing 14.14, you can see that there are two blocks- try and catch. In the try block, the logic for division of two numbers is given and the catch block handles the exception, in case it occurs.

As you run the `MyException` application, the output appears, as shown in Figure 14.14:

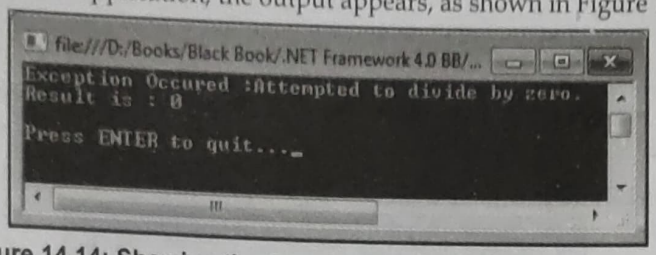

Figure 14.14: Showing the Output of the MyException Application

Figure 14.14 shows that when 100 is divided by 0, the `DivideByZeroException` occurs; and therefore, the program control is transferred to the `catch` block and executes the statement inside the `catch` block and finally displays the result.

The throw Statement

The throw statement is used to raise an exception in case an error occurs in a program. The `throw` statement takes only a single argument to throw the exception. When a `throw` statement is encountered, a program terminates. In C#, it is also possible to throw an exception programmatically. You can use the `throw` keyword to throw an exception. The following code snippet shows an example of the `throw` statement:

```
try
{
throw new DivideByZeroException();
}
catch(DivideByZeroException e)
{
Console.WriteLine("Exception");
}
```

In the preceding code snippet, you can see that we have thrown a new `DivideByZeroException` explicitly.

Let's create a console application named `MyThrowException` (also available in the CD) that shows how to throw an exception. Listing 14.15 shows the code of the `MyThrowException` application:

Listing 14.15: Showing the Code of the `MyThrowException` Application

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
namespace MyThrowException
{
    class Program
    {
        static void Main(string[] args)
        {
            int number;
            Console.Write("Enter a Number:");
            number = int.Parse(Console.ReadLine());
            try
            {

                if (number > 10)
                    throw new Exception("OutofSize");


            }
            catch (Exception e)
            {
                Console.WriteLine("Exception has been Occured");
            }
            finally
            {
                Console.WriteLine("This is the Last Statement");
                Console.Write("\nPress ENTER to quit...");
                Console.ReadLine();
```

that specifies the kind of data each column contains, such as integers or string values.

Describing the Architecture of ADO.NET

ADO.NET consists of two fundamental components: the `DataSet`, which is disconnected from the data source and does not need to know from where the data is retrieved; and the .NET data provider which allows you to connect to the data source and execute the SQL commands against it. The `DataSet` is designed for data access independent of any data source. It can be used in the same way to manipulate data from a traditional data source or from an XML document. The data providers are designed for data manipulation and read only access of data. The architecture of ADO.NET is explained illustratively in Figure 19.1:

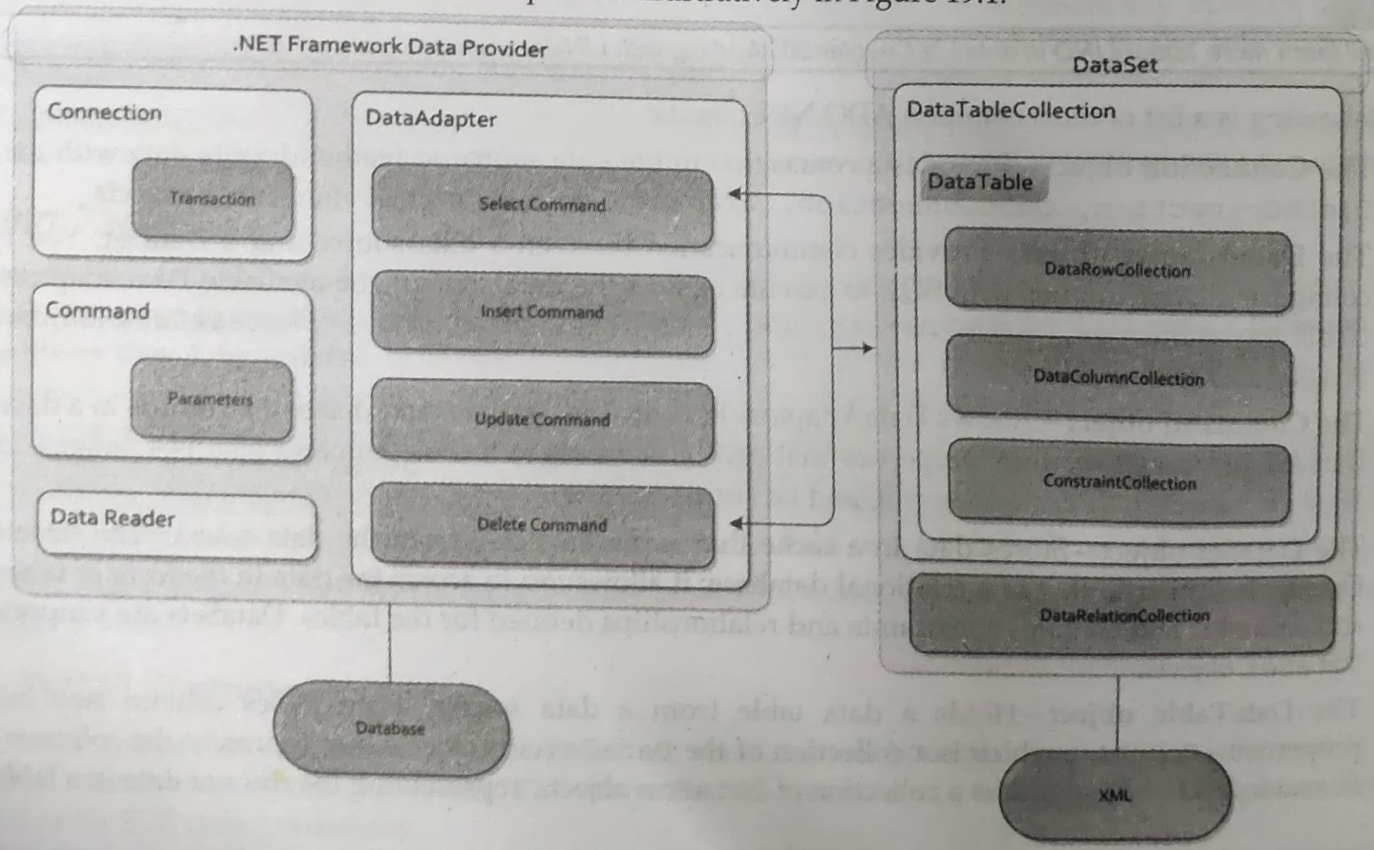

Figure 19.1: Showing the ADO.NET Architecture

Data Providers in ADO.NET

A data provider is a set of related components that work together to provide data in an efficient manner. It is used in ADO.NET for connecting to a database, executing commands, and retrieving results. The results are either processed directly or manipulated between tiers and displayed after combining with multiple sources. The data providers increase performance without compromising on functionality.

Objects of Data Provider

The data provider in ADO.NET consists of the following four objects:

- **Connection**—Creates connection to the data source. The base class for all the `Connection` objects is the `DbConnection` class. The `Connection` object has the methods for opening and closing connection and beginning a transaction. In addition, they have properties for setting the timeout property period of connection. The .NET framework provides two types of `Connection` classes: the `SqlConnection` object, which is designed specifically to connect to Microsoft SQL Server and the `OleDbConnection` object, which is designed to provide connections to a wide range of databases, such as Microsoft Access and Oracle.
- **Command**—Executes a command against the data source and retrieve a `DataReader` or `DataSet`. It also executes the INSERT, UPDATE, or DELETE command against the data source. The base class for all `Command` objects is the `DbCommand` class. The `Command` object is represented by two classes: `SqlCommand` and `OleDbCommand`. The `Command` object provides three methods that are used to execute commands on the database. The `ExecuteNonQuery()` method executes the commands, such as INSERT, UPDATE, or DELETE that have no return value. The `ExecuteScalar()` method returns a single value from a database query. The `ExecuteReader()` method returns a result set by way of the `DataReader` object.
- **DataReader**—Provides a forward-only and read-only connected result set. The base class for all `DataReader` objects is the `DataReader` class. The `DataReader` object cannot be directly instantiated.
- **DataAdapter**—Updates the `DataSet` with data from the data source. The base class for all `DataAdapter` objects is the `DbDataAdapter` class. The `DataAdapter` acts as an intermediary for all the communication between the database and `DataSet`. The `DataAdapter` is used to fill a `DataTable` or `DataSet` with data from the database using the `Fill()` method. The `DataAdapter` commits the changes to the database by calling the `Update()` method. The `DataAdapter` provides four properties that represent the database command: `SelectCommand`, `InsertCommand`, `DeleteCommand`, and `UpdateCommand`.

DataSet

DataSet is a very useful in-memory representation of data and acts as the core of a wide variety of data based applications. A DataSet can be considered as a local copy of the relevant portions of the database. The data in the DataSet can be manipulated and updated independent of the database. You can load the data in the DataSet from any valid source, such as the Microsoft SQLServer database, Oracle database, or Microsoft Access database.

Components of DataSets

The various components that make up a DataSet are listed as follows:

- **DataTable**—Consists of DataRow and DataColumn and stores data in the table row format. The DataTable is the central object of the ADO.NET library and similar to a table in a database. You should note that the DataTable objects are case-sensitive. For example, the EmployeeDetails table is not the same as the employeedetails table. A DataSet can contain two DataTable objects that have the same TableName property and different Namespace property values. The NewRow() method is used to add a row to a DataTable. The maximum number of rows that a DataTable can contain is fixed at 16,777,216. The DataTable also contains a collection of Constraint objects that is used to ensure the integrity of data.
- **DataView**—Represents a customized view of DataTable for sorting, filtering, searching, editing, and navigation. A DataView allows you to create a view on a DataTable to see a subset of data based on a preset condition specified in the RowStateFilter property. A DataView can be used to present a subset of data from the DataTable.
- **DataColumn**—Consists of a number of columns that comprises a DataTable. A DataColumn is the essential building block of the DataTable. The DataType property of DataColumn determines the kind of data that a column holds. You can also bind a control to a particular column in the DataTable.
- **DataRow**—Represents a row in the DataTable. You can use the DataRow object and its properties and methods to retrieve, evaluate, insert, delete, and update the values in the DataTable. You can use the NewRow() method of the DataTable to create a new DataRow and the Add() method to add the new DataRow to the DataTable. You can also delete DataRow by calling the Remove() method.
- **DataRelation**—Allows you to specify relations between various tables. In simple words, a DataRelation is used to relate two DataTable objects to each other through DataCoulmn objects. The relationships are created between matching columns in the parent and child tables. For this, the DataType value of both columns must be the same. The DataRelation objects are contained in a DataRelationCollection, which you can access through the Relations property of the DataSet.