Windows Application Development - C#.NET (23PCA3CT08)

Unit I

Introduction– Dive into Visual C# 2010 Express– Introduction to C# Applications – Introduction to Classes and Objects

Unit II

Control Statements: Part I– Control Statements: Part II– Methods: A Deeper Look– Arrays – Garbage Collection and Destructors, Static Class Members– Delegates, Lambda Expressions, Anonymous Types.

Unit III

Object Oriented Programming: Inheritance– OOP: Polymorphism, Interfaces and Operator Overloading – Exception Handling - Introduction to LINQ and the List Collection– Graphical User Interfaces with Windows Forms: Part 1

Unit IV

Graphical User Interfaces with Windows Forms: Part 2– Database and LINQ – Generics– Collections– GUI with Windows Presentation Foundation.

Unit V

Data Access with .NET ADO.NET Overview– Using Database Connections– Fast Data Access: The Data Reader – Managing Data and Relationships: The DataSet Class – Populating a DataSet – Persisting DataSet Changes – Working with ADO.NET– The DataGrid Control– Data Binding – Visual Studio.Net and Data Access.

UNIT -1
Introduction

C# pronounced as 'C- Sharp'. C# is a simple, modem, object oriented, and type safe programming language derived from C and C++. C# is a purely object-oriented language like as Java. It has been designed to support the key features of .NET framework. C# was developed by Microsoft within its .NET initiative led by Anders Hejlsberg. C# is designed for Common Language Infrastructure (CLI), which consists of the executable code and runtime environment that allows use of various high-level languages on different computer platforms and architectures.

Features of C#

1. Simplicity All the Syntax of java is like C++. There is no preprocessor, and much larger library. C# code does not require header files. All code is written inline.

2. Consistent behavior C# introduced an unified type system which eliminates the problem of varying ranges of integer types. All types are treated as objects and developers can extend the type system simply and easily.

3. Modern programming language

C# supports number of modern features, such as:

- Automatic Garbage Collection
- Error Handling features
- Modern debugging features
- Robust Security features

4. Pure Object- Oriented programming language

In C#, every thing is an object. There are no more global functions, variable and constants..NET Framework and C# lecture Notes Division of Computer and Information Science

It supports all three object oriented features:

- Encapsulation
- Inheritance
- Polymorphism

5. Type Safety Type safety promotes robust programming.

Some examples of type safety are:

• All objects and arrays are initialized by zero dynamically

• An error message will be produced , on use of any uninitialized variable

• Automatic checking of array out of bound and etc.

6. Feature of Versioning Making new versions of software module work with the existing applications is known as

versioning. Its achieve by the keywords new and override.

7. Compatible with other language C# enforces the .NET common language specifications (CLS) and therefore allows interoperation with other .NET language.

The .NET Framework is composed of four main components:
• Common Language Runtime (CLR)
• Framework Class Library (FCL),
• Core Languages (WinForms, ASP.NET, and ADO.NET), and
• Other Modules (WCF, WPF, WF, Card Space, LINQ, Entity Framework, Parallel LINQ, Task Parallel Library, etc.)

Common Language Run Time–

It is a program execution engine that loads and executes the program. It converts the program into native code. It acts as an interface between the framework and operating system. It does exception handling, memory management, and garbage collection. Moreover, it provides security, typesafety, interoperability, and portability. A list of CLR components are given below:

Common Type System

• The Common Type System (CTS) is a standard for defining and types in the .NETframework. CTS defines a collection of data types, which are used and managed by the run time to facilitate cross integration.
• CTS provide the types in the .NET Framework with which .NET applications, components and programming languages so information is shared easily.

Common Language Specification

• The Common Language Specification (CLS) is a fundamental set of language features supported by the Common Language Runtime (CLR) of the .NET Framework.
• CLS is a part of the specifications of the .NET Framework. CLS was designed to support language constructs commonly used by developers and to produce verifiable code, which allows all CLS to ensure the type safety of code.

Structure of C#

C# program consists of the following things.
1. Namespace declaration
2. A Class
3. Class methods
4. Class attributes
5. The Main method
6. Statements and Expressions
7. Comments

Example

```
using System;
namespace HelloWorldApplication
{
class HelloWorld
{
static void Main(string[] args)
{
/* my first program in C# */
Console.WriteLine("Hello World");
}}}
```

• using System

This "using" keyword is used to contain the System namespace in the program. Every program has multiple using statements.

• namespace declaration

It's a collection of classes. The HelloCSharp namespace contains the class prog1HelloWorld.

• class declaration

The class prog1HelloWorld contains the data and method definitions that your program.

• defines the Main method

This is the entry point for all C# programs. The main method states what the class does when executed.

• WriteLine

It's a method of the Console class distinct in the System namespace. This statement causes the message "Hello, World!" to be displayed on the screen.

Important in C#

• C# is case sensitive.
• C# program execution starts at the Main method.
• All C# expression and statements must end with a semicolon (;).
• File name is different from the class name. This is unlike Java.

Data types

The variables in C#, are categorized into the following types:

- **Value types**
- **Reference types**

Value types - variables can be assigned a value directly. They are derived from the class System.ValueType.

The value types directly contain data. Some examples are int, char, and float, which stores numbers, alphabets, and floating point numbers, respectively.

Example:

```
int i = 75;
float f = 53.005f;
double d = 2345.7652;
bool b = true;
```

Reference Types

The pre-defined reference types are object and string, where object - is the ultimate base class of all other types. New reference types can be defined using 'class', 'interface', and 'delegate' declarations. Therefore the reference types are :

Predefined Reference Types

- **Object**
- **String**

User Defined Reference Types

- **Classes**
- **Interfaces**
- **Delegates**

UNIT – 2

Control Statements

Again control statements are divided into three statements *(a)* Loop statements *(b)* Jump statements *(c)* Selection statements

(a) Loop Statements

C# provides a number of the common loop statements:

• while
• do-while
• for
• foreach

while loops

Syntax: while (expression) statement[s]

A 'while' loop executes a statement, or a block of statements wrapped in curly braces, repeatedly until the condition specified by the Boolean expression returns false. For instance, the following code.

```
int a =0; While (a < 3) {
System.Console. WriteLine (a);
a++;
}
```

Produces the following output:

0 1 2

do-while loops

Syntax: do statement [s] while (expression)

A 'do-while' loop is just like a 'while' loop except that the condition is evaluated after the block of code specified in the 'do' clause has been run. So even where the condition is initially false, the block runs once. For instance, the following code outputs '4':

```
Int a = 4;
do
{
System.Console. WriteLine (a);
a++;
} while (a < 3);
```

for loops

Syntax: for (statement1; expression; statement2) statement[s]3

The 'for' clause contains three part. Statementl is executed before the loop is entered.

The loop which is then executed corresponds to the following 'while' loop:

Statementl

while (expression) {statement[s]3; statement2}
'for' loops tend to be used when one needs to maintain an iterator value. Usually, as in the following example, the first statement initializes the iterator, the condition evaluates it against an end value, and the second statement changes the iterator value.

```
for (int a =0; a<5; a++)
{
system.console.writeLine(a);
}
```

foreach loops

syntax:foreach (variablel in variable2) statement[s]

The 'foreach' loop is used to iterate through the values contained by any object which implements the IEnumerable interface. When a 'foreach' loop runs, the given variablel is set in turn to each value exposed by the object named by variable2. As we have seen previously, such loops can be used to access array values. So, we could loop through the values of an array in the following way:

```
int[] a = new int[]{1,2,3};
foreach (int b in a)
system.console.writeLine (b);
```

The main drawback of 'foreach' loops is that each value extracted (held in the given example by the variable 'b') is read-only.

(b) Jump Statements

The jump statements include

- break
- continue
- goto
- return
- throw

break

The following code gives an example - of how it could be used. The output of the loop is the numbers from. 0 to 4.

```
int a = 0;
while (true)
{
system.console.writeLine(a);
a++;
if (a == 5)
break;
}
```

Continue

The 'continue' statement can be placed in any loop structure. When it executes, it moves the program counter immediately to the next iteration of the loop. The following code example uses the 'continue' statement to count the number of values between 1 and 100 inclusive that are not multiples of seven. At the end of the loop the variable y holds the required value.

```
int y = 0;
for (int x=l; x<101; x++)
{
if ((x % 7) == 0)
continue;
y++;
}
```

Goto

The 'goto' statement is used to make a jump to a particular labeled part of the program code. It is also used in the 'switch' statement described below. We can use a'goto' statement to construct a loop, as in the following example (but again, this usage is not recommended):

```
int a = 0;
start:
system.console.writeLine(a);
a++;
if (a < 5)
```

Conditional statements

A conditional statement decides whether to execute code based on conditions. The if statement and the switch statement are the two types of conditional statements in C#.

The if statement

As with most of C#, the **if** statement has the same syntax as in C, C++, and Java. Thus, it is written in the following form:

***if-statement* ::= "if" "(" *condition* ")" *if-body* ["else" *else-body*]**
condition* ::= *boolean-expression
if-body* ::= *statement-or-statement-block
else-body* ::= *statement-or-statement-block

The if statement evaluates its *condition* expression to determine whether to execute the *ifbody*. Optionally, an else clause can immediately follow the *if body*, providing code to execute
when the *condition* is *false*. Making the *else-body* another if statement creates the common *cascade* of if, else if, else if, else if, else statements:

```
using System;
public class IfStatementSample
{
public void IfMyNumberIs()
{
int myNumber = 5;
if ( myNumber == 4 )
Console.WriteLine("This will not be shown because myNumber is not
4.");
else if( myNumber < 0 )
{
Console.WriteLine("This will not be shown because myNumber is not
negative.");
}
else if( myNumber % 2 == 0 )
Console.WriteLine("This will not be shown because myNumber is not
even.");
else
{
Console.WriteLine("myNumber does not match the coded conditions,
so this sentence will be shown!");
}
}
}
```

The switch statement

The switch statement is similar to the statement from C, C++ and Java.
Unlike C, each case statement must finish with a jump statement (which can be break or
goto or return). In other words, C# does not support "fall through" from one case statement to
the next (thereby eliminating a common source of unexpected behaviour in C programs). However "stacking" of cases is allowed, as in the example below. If goto is used, it may refer to a case
label or the default case (e.g. goto case 0 or goto default).
The default label is optional. If no default case is defined, then the default behaviour is to
do nothing.

A simple example:

```
switch (nCPU)
{
case 0:
Console.WriteLine("You don't have a CPU! :-)");
break;
case 1:
Console.WriteLine("Single processor computer");
break;
case 2:
Console.WriteLine("Dual processor computer");
break;
// Stacked cases
case 3:
case 4:
case 5:
case 6:
case 7:
case 8:
Console.WriteLine("A multi processor computer");
break;
default:
Console.WriteLine("A seriously parallel computer");
break;
}
```

• **Arrays**

Object Type is the ultimate base class for all data types in C# Common Type System (CTS). Object is an alias for System.Object class. The object types can be assigned values of any other types, value types, reference types, predefined or userdefined types.

String Type allows you to assign any string values to a variable. The string type is an alias for the System.String class. It is derived from object type.

Ex: String str = "Tutorials Point";

Char to String

```
string s1 = "hello";
char[] ch = { 'c', 's', 'h', 'a', 'r', 'p' };
string s2 = new string(ch);
Console.WriteLine(s1);
Console.WriteLine(s2);
```

Converting Number to String

```
int num = 100;
string s1= num.ToString();
```

Inserting String.NET Framework and C# lecture Notes
Division of Computer and Information Science

```
string s1 = Wel;
string s2 = s1.insert(3,‖come‖);
// s2 = Welcome
string s3 = s1.insert(3,‖don‖);
// s3 = Weldon;
```

Class and Object

Class and Object are the basic concepts of Object Oriented Programming which revolve around the real-life entities.

A **class** is a user-defined blueprint or prototype from which objects are created. Basically, a class combines the fields and methods.

An object consists of :

• **State:** It is represented by attributes of an object. It also reflects the properties of an object.

• **Behavior:** It is represented by methods of an object. It also reflects the response of an object with other objects.

• **Identity:** It gives a unique name to an object and enables one object to interact with other objects.

Example

```
using System; class A { public int x=100; } class mprogram { static void Main(string[] args) { A a = new A();
```

Console.WriteLine("Class A variable x is " +a.x);
} }

Static Class

A static class can only contain static data members, static methods, and a static constructor.It is not allowed to create objects of the static class. **Static classes are sealed,** cannot inherit a static class from another class.

Note: Not allowed to create objects.

.NET Framework and C# lecture Notes

Division of Computer and Information Science

Syntax

static class Class_Name {
// static data members // static method
}

Example

static class Author {
// Static data members of Author public static string A_name = "Ankita"; public static string L_name = "CSharp"; public static int T_no = 84;
// Static method of Author public static void details()
{ Console.WriteLine("The details of Author is:"); } } // Main Method static public void Main() {
// Calling static method of Author Author.details();
// Accessing the static data members of Author Console.WriteLine("Author name : {0} ",
Author.A_name); Console.WriteLine("Language : {0} ", Author.L_name); Console.WriteLine("Total number of articles : {0} ", Author.T_ no); }

Partial Class

A partial class is a special feature of C#. It provides a special ability to implement the functionality of a single class into multiple files and all these files are combined into a single class file when the application is compiled using the partial modifier keyword. The partial modifier can be applied to a class, method, interface or structure.

Advantages:

It avoids programming confusion (in other words better readability).

Multiple programmers can work with the same class using different files.

Even though multiple files are used to develop the class all such files should have a common class name.

Example

Filename: partial1.cs

using System; partial class A {
public void Add(int x,int y) {
Console.WriteLine("sum is {0}",(x+y)); }
}

Filename: partial2.cs using System;

```
partial class A {
public void Substract(int x,int y) {
```

```
Console.WriteLine("Subtract is {0}", (x-y)); }
}
Filename joinpartial.cs class Demo
{
public static void Main() {
A obj=new A(); obj.Add(7,3); obj.Substract(15,12);
} }
```

Member Access Modifiers

Access modifiers provide the accessibility control for the members of classes to outside the class. They also provide the concept of data hiding. There are five member access modifiers provided by the C# Language.

Modifier	***Accessibility***
private	Members only accessible with in class
public	Members may accessible anywhere outside class
protected	Members only accessible with in class and
derived class	
internal	Members accessible only within assembly
protected internal	Members accessible in assembly, derived class
or containing program	

By default all member of class have private accessibility. If we want a member to have any other accessibility, then we must specify a suitable access modifier to it individually.

Example:

```
class Demo {
public int a; internal int x; protected double d;
float m; // private by default }
```

Static classes

Static classes are commonly used to implement a Singleton Pattern. All of the methods,
properties, and fields of a static class are also static (like the WriteLine() method of the System.Console class) and can thus be used without instantiating the static class:

```
public static class Writer
{
public static void Write()
{
System.Console.WriteLine("Text");
}
}
public class Sample
{
public static void Main()
{
Writer.Write();
}
}
```

Garbage Collection (GC)

The Garbage collection is the important technique in the .Net framework to free the unused managed code objects in the memory and free the space to the process.

The garbage collection (GC) is new feature in Microsoft .net framework. When a class that represents an object in the runtime that allocates a memory space in the heap memory. All the behavior of that objects can be done in the allotted memory in the heap.

Microsoft was planning to introduce a method that should automate the cleaning of unused memory space in the heap after the life time of that object. Eventually they have introduced a new technique "Garbage collection". It is very important part in the .Net framework. Now it handles this object clear in the memory implicitly. It overcomes the existing explicit unused memory space clearance.

• Automatic memory management is made possible by **Garbage Collection in .NET Framework**. When a class object is created at runtime, certain memory space is allocated to it in the heap memory. However, after all the actions related to the object are completed in the program, the memory space allocated to it is a waste as it cannot be

used. In this case, garbage collection is very useful as it automatically releases the memory space after it is no longer required.

• Garbage collection will always work on **Managed Heap** and internally it has an Engine which is known as the **Optimization Engine**. Garbage Collection occurs if at least one of multiple conditions is satisfied. These conditions are given as follows:

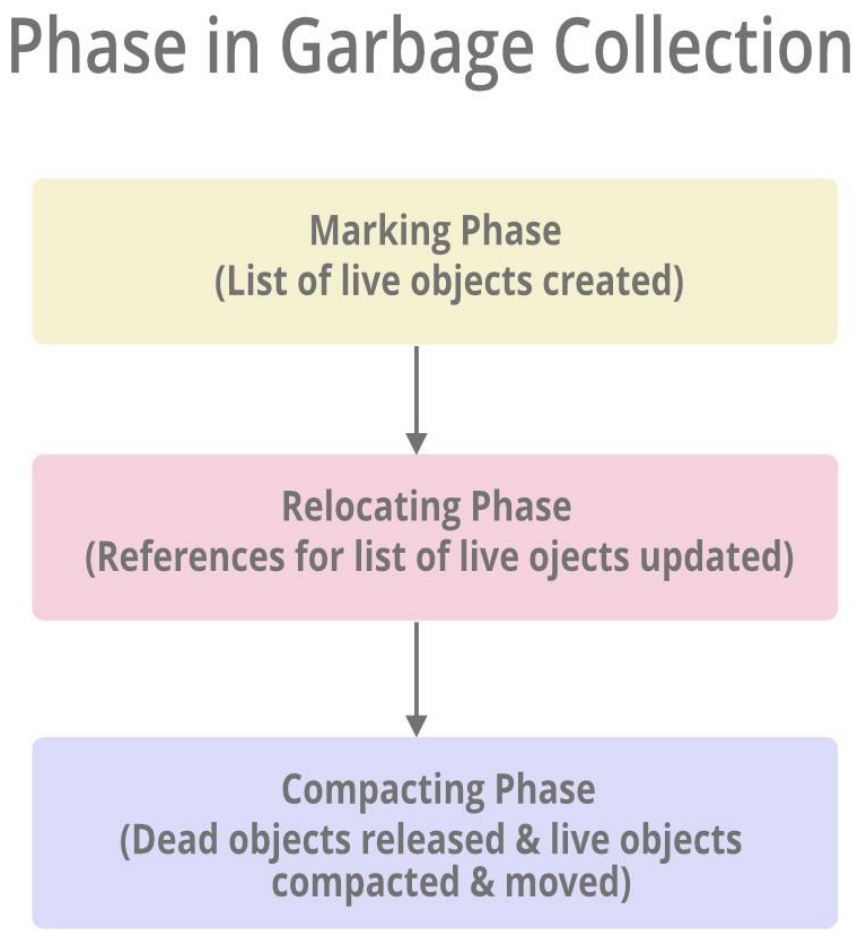
o If the system has low physical memory, then garbage collection is necessary.

o If the memory allocated to various objects in the heap memory exceeds a pre-set threshold, then garbage collection occurs.

o If the *GC.Collect* method is called, then garbage collection occurs. However, this method is only called under unusual situations as normally garbage collector runs automatically.

Phases in Garbage Collection

There are mainly **3** phases in garbage collection. Details about these are given as follows:



1. **Marking Phase:** A list of all the live objects is created during the marking phase. This is done by following the references from all the root objects. All of

the objects that are not on the list of live objects are potentially deleted from the heap memory.
2. **Relocating Phase:** The references of all the objects that were on the list of all the live objects are updated in the relocating phase so that they point to the new location where the objects will be relocated to in the compacting phase.
3. **Compacting Phase:** The heap gets compacted in the compacting phase as the space occupied by the dead objects is released and the live objects remaining are moved. All the live objects that remain after the garbage collection are moved towards the older end of the heap memory in their original order.

Array An array is a group of homogeneous data stored to variables And each data item is called an element of the array.
Syntax :
type [ ] < Name_Array > = new < datatype > [size];
Examples
int[] intArray1 = new int[5]; int[] intArray2 = new int[5]{1, 2, 3, 4, 5}; int[] intArray3 = {1, 2, 3, 4, 5};
Jagged Arrays Jagged **array is a array** of arrays such that member arrays can be of different sizes. In other words, the length of each array index can differ.
Syntax:
data_type[][] name_of_array = new data_type[rows][]
Example:
// Declare the Jagged Array of four elements:
int[][] jagged_arr = new int[4][];
// Initialize the elements
jagged_arr[0] = new int[] {1, 2, 3, 4}; jagged_arr[1] = new int[] {11, 34, 67}; jagged_arr[2] = new int[] {89, 23}; jagged_arr[3] = new int[] {0, 45, 78, 53, 99};
ArrayList is a powerful feature of C# language. It is the nongeneric type of collection which is defined in System.Collections namespace. It is used to create a dynamic array means the size of the array is increase or decrease automatically according to the requirement.
Arraylist in use the program, must be add System.Collections namespace.
Syntax:
ArrayList list_name = new ArrayList();
Example
// Creating ArrayList ArrayList My_array = new ArrayList();

// This ArrayList contains elements // of different types My_array.Add(112.6); My_array.Add("C# program"); My_array.Add(null); My_array.Add('Q'); My_array.Add(1231);
// Access the array list
foreach(var elements in My_array)
{ Console.WriteLine(elements); }
Note: Array List allow add insert remove elements, change element.
Indexers An indexer allows an instance of a class or struct to be indexed as an array. If the user will define an indexer for a class, then the class will behave like a virtual array. Array

access operator i.e ([ ]) is used to access the instance of the class which uses an indexer.
Syntax:
[access_modifier] [return_type] this [argument_list] { get {
// get block code
} set { // set block code } }
Example
public string this[int index] { get { return val[index]; } set { val[index] = value; }
IndexerCreation ic = new IndexerCreation(); ic[0] = "C"; ic[1] = "CPP"; ic[2] = "CSHARP";
Properties
Properties are the special type of class members that provides a flexible mechanism to read, write, or compute the value of a
private field. Properties can be used as if they are public data members, but they are actually special methods called accessors.
Accessors : The block of “set” and “get”
There are different types of properties based on the “get” and set accessors:
Read and Write Properties: When property contains both get and set methods.
Read-Only Properties: When property contains only get method.
Write Only Properties: When property contains only set method.
Auto Implemented Properties: When there is no additional logic in the property accessors and it introduce in C# 3.0.

Delegates

A delegate is a reference type variable that holds the reference to a method. The reference can be changed at runtime.

Delegates are especially used for implementing events and the call-back methods. All delegates are implicitly derived from the System.Delegate class.

A delegate will call only a method which agrees with its signature and return type. A method can be a static method associated with a class or can be instance method associated with an object, it doesn't matter.

Syntax:

[modifier] delegate [return_type] [delegate_name]
([parameter_list]);

.NET Framework and C# lecture Notes

Division of Computer and Information Science

Example

public delegate void addnum(int a, int b);

using System; class TestDelegate { delegate int NumberChanger(int n); static int num = 10; public static int AddNum(int p) { num += p; return num; } public static int MultNum(int q) { num *= q; return num; } public static int getNum() { return num; } static void Main(string[] args) {

//create delegate instances

NumberChanger nc1 = new NumberChanger(AddNum); NumberChanger nc2 = new NumberChanger(MultNum);

//calling the methods using the delegate objects nc1(25);

Console.WriteLine("Value of Num: {0}", getNum()); nc2(5);

Console.WriteLine("Value of Num: {0}", getNum()); Console.ReadKey(); } }

}

Events

Events are user actions such as key press, clicks, mouse movements, etc., or some occurrence such as system generated notifications. Applications need to respond to events when they occur. For example, interrupts. Events are used for inter-process communication.

Delegates with Events

C# and .NET supports event driven programming via delegates.The events are declared and raised in a class and associated with the event handlers using delegates within the same class or some other class. The class containing the event is used to publish the event. This is called the publisher class. Some other class that accepts this event is called the subscriber class. Events use the publisher-subscriber model.

A publisher is an object that contains the definition of the event and the delegate. The event-delegate association is also defined in this object. A publisher class object invokes the event and it is notified to other objects.
A subscriber is an object that accepts the event and provides an event handler. The delegate in the publisher class invokes the method (event handler) of the subscriber class.
To declare an event inside a class, first of all, you must declare a delegate type for the even as:

```
public delegate string BoilerLogHandler(string str);
```

Following are the key points about Event,
• Event Handlers in C# return void and take two parameters.
• The First parameter of Event - Source of Event means publishing object.
• The Second parameter of Event - Object derived from EventArgs.
• The publishers determines when an event is raised and the subscriber determines what action is taken in response.
• An Event can have so many subscribers.

• Events are basically used for the single user action like button click.
• If an Event has multiple subscribers then event handlers are invoked synchronously.

UNIT – 3

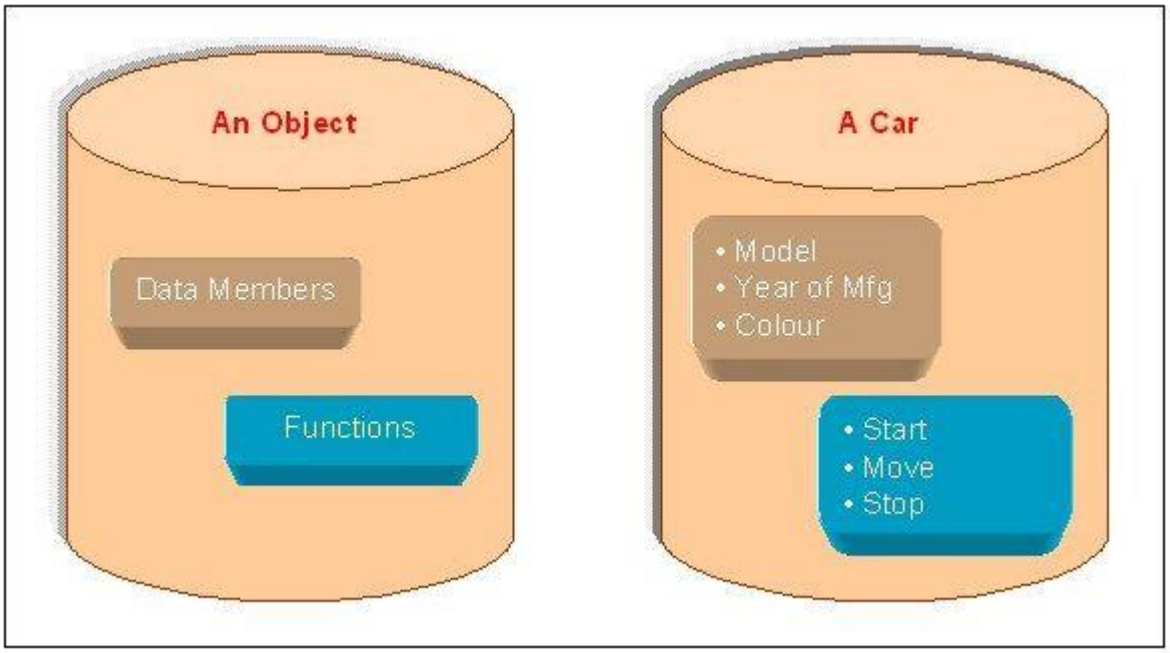
Object Oriented Concept–

Object Oriented Programming (OOP) is a programming model where programs are organized around objects and data rather than action and logic.
OOP allows decomposition of a problem into a number of entities called objects and then builds data and functions around these objects.
1. The software is divided into a number of small units called objects. The data and functions are built around these objects.
2. The data of the objects can be accessed only by the functions associated with that object.
3. The functions of one object can access the functions of another object.
OOP has the following important features.

Lifecycle of an object:
Apart from the normal state of — being in use, ‖ every object includes two important stages:
Construction: When an object is first instantiated it needs to be initialized. This initialization is known as construction and is carried out by a constructor function.
Destruction: When an object is destroyed, there are often some clean - up tasks to perform, such as freeing memory. This is the job of a destructor function. A constructor is used to initialize the data by an object. All objects have a default constructor , which is a parameterless method with the same name as the class itself. A class definition might include several constructor methods with parameters, known as nondefault constructors. Constructors are called using the new keyword: CupOfCoffee myCup = new CupOfCoffee(); The line of code above instantiates a CupOfCoffee object using its default constructor.
Class
A class is the core of any modern Object Oriented Programming language such as C#.
In OOP languages it is mandatory to create a class for representing data.
A class is a blueprint of an object that contains variables for storing data and functions to perform operations on the data.
A class will not occupy any memory space and hence it is only a logical representation of data.
To create a class, you simply use the keyword "class" followed by the class name:

```
class Employee {
}
```

Object



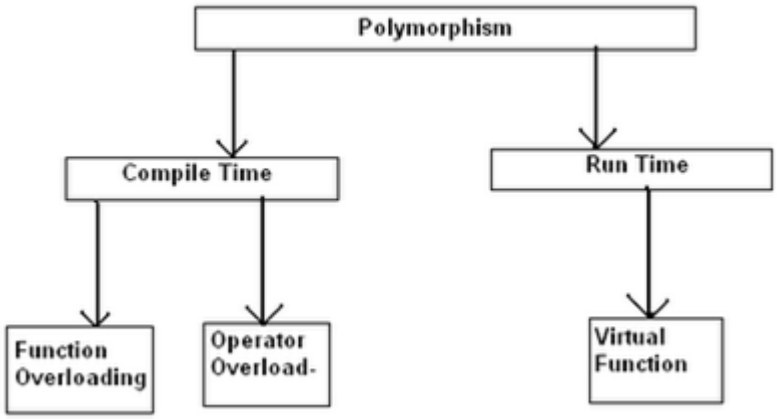
Objects are the basic run-time entities of an object oriented system. They may represent a person, a place or any item that the program must handle.
"An object is a software bundle of related variable and methods." "An object is an instance of a class"
A class will not occupy any memory space. Hence to work with the data represented by the class you must create a variable for the class, that is called an object.
When an object is created using the new operator, memory is allocated for the class in the heap, the object is called an instance and its starting address will be stored in the object in stack memory.
When an object is created without the new operator, memory will not be allocated in the heap, in other words an instance will not be created and the object in the stack contains the value **null**.
When an object contains null, then it is not possible to access the members of the class using that object.

```
1. class Employee
2. {
3.
4. }
```

Syntax to create an object of class Employee:

Polymorphism:

The ability to treat an instance of derived class as an instance of the base class and being able to call the operations of the base class using an instance of the derived class is called Polymorphism. Polymorphism means one name many forms. Polymorphism means one object behaving as multiple forms. One function behaves in different forms



Static or Compile Time Polymorphism

In static polymorphism, the decision is made at compile time. Which method is to be called is decided at compile-time only. Method overloading is an example of this. Compile time polymorphism is method overloading, where the compiler knows which overloaded method it is going to call.

Dynamic or Runtime Polymorphism

Run-time polymorphism is achieved by method overriding. Method overriding allows us to have methods in the base and derived classes with the same name and the same parameters. By runtime polymorphism, we can point to any derived class from the object of the base class at runtime that shows the ability of runtime binding.

Overloadable Operators

C# allows user-defined types to overload operators by defining static member functions using the operator keyword.

The following example illustrates an implementation of the + operator. If we have a Complex class which represents a complex number:

```
public struct Complex
{
public double Real { get; set; }
public double Imaginary { get; set; } }
```

And we want to add the option to use the + operator for this class. i.e.:

Complex a = new Complex() { Real = 1, Imaginary = 2 }; Complex b = new Complex() { Real = 4, Imaginary = 8 }; Complex c = a + b;

We will need to overload the + operator for the class. This is done using a static function and the operator keyword:

```
public static Complex operator +(Complex c1, Complex c2) {
return new Complex
{
Real = c1.Real + c2.Real,
Imaginary = c1.Imaginary + c2.Imaginary };
}
```

Inheritance Inheritance supports the concept of “reusability”
one class is allowed to inherit the features(fields and methods) of another class.
Important terminology:
Super Class: The class whose features are inherited is known as super class(or a base class or a parent class).
Sub Class: The class that inherits the other class is known as subclass(or a derived class, extended class, or child class). The subclass can add its own fields and methods
Reusability: To create a new class and there is already a class that includes some of the code that need to derive new class from the existing class.
1)Single Inheritance
2) Multilevel Inheritance
3) Multiple Inheritance (interface)
4) Hierarchical Inheritance
Example
.NET Framework and C# lecture Notes
Division of Computer and Information Science
Class A { Int x;
Void display()
{ System.Consolw.WriteLine(“x=”+x); }
Class B : A { Display(); }

OPERATOR OVERLOADING: This is something very much similar to the concept of method overloading. Operator overloading allows us to define multiple behaviours to an operator. In operator overloading the behaviour of an operator changes according to the operands types which we use in an expression.
For example ‗+‘ is an overloaded operator, which can be used for both addition as well as concatenation. It works as addition operator when the operator used between numeric operands and works as concatenation operator when the operator used between string operands.
We can also add new behaviour to any existing operators by defining operator method.
Syntax: public static returntype operator op(argumentslist) {
Method body }
Where,
- They must be defined as **public and static**.
- **Return type** specifies the type of result we are expecting when the operator is used between
two operands..

➢ **Operator** is a keyword which represents the operator method for overloading.
➢ **Op** is an any overloadable operator.
➢ The **argument list** is the list of arguments passed.

Example:

```
// C# program to illustrate the
// unary operator overloading
using System;
namespace Calculator {
class Calculator {
        public int number1 , number2;
        public Calculator(int num1 , int num2)
        {
                number1 = num1;
                number2 = num2;
        }
// Function to perform operation
// By changing sign of integers
public static Calculator operator -(Calculator c1)
{
        c1.number1 = -c1.number1;
        c1.number2 = -c1.number2;
        return c1;
}
// Function to print the numbers
public void Print()
{
        Console.WriteLine ("Number1 = " + number1);
        Console.WriteLine ("Number2 = " + number2);
}}
class EntryPoint
{
        // Driver Code
        static void Main(String []args)
        {
                // using overloaded - operator
                // with the class object
                Calculator calc = new Calculator(15, -25);
                calc = -calc;
                // To display the result
                calc.Print();
        }}}
```

(List Collections)

The List<T> is a collection of strongly typed objects that can be accessed by index and having methods for sorting, searching, and modifying list. It is the generic version of the ArrayList that comes under System.Collections.Generic namespace.

List<T> Characteristics

- List<T> equivalent of the ArrayList, which implements IList<T>.
- It comes under System.Collections.Generic namespace.
- List<T> can contain elements of the specified type. It provides compile-time type checking and doesn't perform boxing-unboxing because it is generic.
- Elements can be added using the Add(), AddRange() methods or collection-initializer syntax.
- Elements can be accessed by passing an index e.g. myList[0]. Indexes start from zero.
- List<T> performs faster and less error-prone than the ArrayList.

Creating a List

The List<T> is a generic collection, so you need to specify a type parameter for the type of data it can store. The following example shows how to create list and add elements.

Example: Adding elements in List

```
List<int> primeNumbers = new List<int>();
primeNumbers.Add(1); // adding elements using add() method
primeNumbers.Add(3);
primeNumbers.Add(5);
primeNumbers.Add(7);
var cities = new List<string>();
cities.Add("New York");
cities.Add("London");
cities.Add("Mumbai");
cities.Add("Chicago");
cities.Add(null);// nulls are allowed for reference type list
//adding elements using collection-initializer syntax
var bigCities = new List<string>()
                {
                   "New York",
                   "London",
                   "Mumbai",
                   "Chicago"
                };
```

Interface

C# allows the user to inherit one interface into another interface. When a class implements the inherited interface.

Example

```
using System;
// declaring an interface public interface A {
// method of interface void mymethod1();
void mymethod2(); }
// The methods of interface A
// is inherited into interface B public interface B : A {
// method of interface B void mymethod3();
}
// Below class is inherting // only interface B
// This class must // implement both interfaces class Geeks : B
{
// implementing the method // of interface A
public void mymethod1() { Console.WriteLine("Implement method 1"); }
// Implement the method // of interface B public void mymethod3()
{ Console.WriteLine("Implement method 3"); } }
```

Lambda & anonymous methods

An anonymous method can be assigned wherever a delegate is expected:

```
Func<int, int> square = delegate (int x) { return x * x; }
```

Lambda expressions can be used to express the same thing:

```
Func<int, int> square = x => x * x;
```

In either case, we can now invoke the method stored inside square like this:

```
var sq = square.Invoke(2);
```

Or as a shorthand:

```
var sq = square(2);
```

Notice that for the assignment to be type-safe, the parameter types and return type of the anonymous method must match those of the delegate type:

```
Func<int, int> sum = delegate (int x, int y) { return x + y; } // error Func<int, int> sum = (x, y) => x + y; // error
```

Method Overloading

Method Overloading is the common way of implementing polymorphism. It is the ability to redefine a function in more than one form. A user can implement function overloading by defining two or more functions in a class sharing the same name. i.e. the methods can have the same name but with different parameters list.

Example

```
// adding two integer values. public int Add(int a, int b)
```

{ int sum = a + b; return sum; }
// adding three integer values. public int Add(int a, int b, int c)
{ int sum = a + b + c; return sum; }

Method Overriding

Method Overriding is a technique that allows the invoking of functions from another class (base class) in the derived class. Creating a method in the derived class with the same signature as a method in the base class is called as method overriding.

Three types of keywords for Method Overriding:

1. virtual keyword: This modifier or keyword use within
base class method. It is used to modify a method in
base class for overridden that particular method in the derived class.

2.override: This modifier or keyword use with derived class method. It is used to modify a virtual or abstract method into derived class which presents in base class.

3. base Keyword: This is used to access members of the base class from derived class.

Example

// method overriding using System; // base class public class web { string name = "Method";
// declare as virtual public virtual void showdata()
{ Console.WriteLine("Base class: " + name); } } class stream : web { string s = "Method"; public override void showdata() { base.showdata();
Console.WriteLine("Sub Class: " + s); } }
class mc {
static void Main()
.NET Framework and C# lecture Notes
Division of Computer and Information Science
{ stream E = new stream();
E.showdata(); } }

Error Handling–

An exception is a problem that arises during
exception is a response to an exceptional circumstance that arises while a program is running, such as an attempt to divide by zero.

Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords:
and **throw**.

• **try** - A try block idenƟfies a block of code for which parƟcular exceptions is activated. It is followed by one or more catch blocks.

• **catch** - A program catches an excepƟon with
the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

• **finally** - The finally block is used to execute a given set of statements, whether an exception is thrown or not
open a file, it must be closed whether an exception is raised or not.

• **throw** - A program throws an excepƟon when a problem shows up. This is done using a throw keyword.
the execution of a program. A C#
try
an exception handler at
thrown. For example, if you
, **catch**, **finally**,

Syntax

Assuming a block raises an exception, a method catches an exception using a combination of the try and catch keywords. A try/catch block is placed around the code that might generate an exception. Code within a try/catch block is referred to as protected code, and the syntax for using try/catch looks like the following -

```
try {
// statements causing exception
} catch( ExceptionName e1 ) {
// error handling code
} catch( ExceptionName e2 ) {
// error handling code
} catch( ExceptionName eN ) {
// error handling code
} finally {
// statements to be executed
}
```

Linq to Objects

LINQ to Objects refers to the use of LINQ queries with any IEnumerable collection.

Using LINQ to Objects in C#

A simple SELECT query in Linq

```
static void Main(string[] args) {
string[] cars = { "VW Golf", "Opel Astra", "Audi A4", "Ford Focus", "Seat Leon", "VW Passat", "VW Polo",
"Mercedes C-Class" };
var list = from car in cars
select car;
StringBuilder sb = new StringBuilder(); foreach (string entry in list)
{
sb.Append(entry + "\n"); }
Console.WriteLine(sb.ToString()); Console.ReadLine();
}
```

Windows Forms–

• Windows Forms is a Graphical User Interface(GUI) class library which is bundled in *.Net Framework*.

• Its main purpose is to provide an easier interface to develop the applications for desktop, tablet, PCs.

• It is also termed as the **WinForms**. The applications which are developed by using Windows Forms or WinForms are known as the **Windows Forms Applications** that runs on the desktop computer.

• WinForms can be used only to develop the Windows Forms Applications not web applications. WinForms applications can contain the different type of controls like labels, list boxes, tooltip etc.

• Creating a Windows Forms Application Using Visual Studio 2017

• First, open the Visual Studio then Go to File -> New -> Project to create a new project and then select the language as Visual C# from the left menu. Click on Windows Forms App(.NET Framework) in the middle of current window. After that give the project name and Click OK.

• Here the solution is like a container which contains the projects and files that may be required by the program.

• After that following window will display which will be divided into three parts as follows:

• Editor Window or Main Window: Here, you will work with forms and code editing. You can notice the layout of form which is now blank. You will double click the form then it will open the code for that.

• Solution Explorer Window: It is used to navigate between all items in solution. For example, if you will select a file form this window then particular information will be display in the property window.

• Properties Window: This window is used to change the different properties of the selected item in the Solution Explorer. Also, you can change the properties of components or controls that you will add to the forms.

Unit IV

Graphical User Interfaces with Windows Forms: Part 2– Database and LINQ – Generics– Collections– GUI with Windows Presentation Foundation.

Unit V

Data Access with .NET ADO.NET Overview– Using Database Connections– Fast Data Access: The Data Reader – Managing Data and Relationships: The DataSet Class – Populating a DataSet – Persisting DataSet Changes – Working with ADO.NET– The DataGrid Control– Data Binding – Visual Studio.Net and Data Access.

Database languages

Database languages are special-purpose languages, which allow one or more of the following tasks, sometimes distinguished as sublanguages:

Data control language (DCL) – controls access to data.

Data definition language (DDL) – defines data types such as creating, altering, or dropping and the relationships among them.

Data manipulation language (DML) – performs tasks such as inserting, updating, or deleting data occurrences.

Data query language (DQL) – allows searching for information and computing derived information.

ADO.NET

ADO.NET is the new database technology used in .NET platform. ADO.NET is the next step in the evolution of Microsoft ActiveX Data Objects (ADO). It does not share the same programming model, but shares much of the ADO functionality. The ADO.NET as a marketing term that covers the classes in the System.Data namespace. ADO.NET is a set of classes that expose the data access services of the .NET Framework.

Connected Vs Disconnected

Connected

A connected environment is one in which a user or an application is constantly connected to adata source. A connected scenario offers the following advantages:

• A secure environment is easier to maintain.

Concurrency is easier to control.

• Data is more likely to be current than in other scenarios.

A connected scenario has the following disadvantages:

• It must have a constant network connection.

• Scalability

Disconnected

A disconnected environment is one in which a user or an application is not constantly connected to a source of data. Mobile users who work with laptop computers are the primary users in disconnected environments. Users can take a subset of data with them on a disconnected computer, and then merge changes back into the central data store.

A disconnected environment provides the following advantages:

• You can work at any time that is convenient for you, and can connect to a data source at any time to process requests.

• Other users can use the connection.

• A disconnected environment improves the scalability and performance of applications.

A disconnected environment has the following disadvantages:

• Data is not always up to date.

• Change conflicts can occur and must be resolved.

ADVANTAGES OF ADO.NET

ADO.NET provides the following advantages over other data access models and components:

Interoperability. ADO.NET uses XML as the format for transmitting data from a data source to a local in-memory copy of the data.

Maintainability. When an increasing number of users work with an application, the increased use can strain resources. By using n-tier applications, you can spread application logic across additional tiers. ADO.NET architecture uses local inmemory caches to hold copies of data, making it easy for additional tiers to trade information.

Programmability. The ADO.NET programming model uses strongly typed data. Strongly typed data makes code more concise and easier to write because Microsoft Visual Studio .NET provides statement completion.

Performance. ADO.NET helps you to avoid costly data type conversions because of its use of strongly typed data.

Scalability. The ADO.NET programming model encourages programmers to conserve system resources for applications that run over the Web. Because data is held locally in memory caches, there is no need to retain database locks or maintain active database connections for extended periods.

.NET DATA PROVIDER

A .NET data provider is used for connecting to a database, executing commands, and retrieving results. Those results are either processed directly, or placed in an ADO.NET DataSet in order to be exposed to the user in an ad-hoc manner, combined with data from multiple sources, or remoted between tiers. The .NET data provider is designed to
be lightweight, creating a minimal layer between the data source and your code, increasing performance without sacrificing functionality. The ADO.NET object model includes the following data provider classes:

1. SQL Server .NET Data Provider
2. OLE DB .NET Data Provider
3. Oracle .NET Data Provider
4. ODBC .NET Data Provider
5. Other Native .NET Data Provider

Windows Forms/Applications

The Windows Forms is a collection of classes and types that encapsulate and extend the Win32 API in an organized object model. The .NET Framework contains an entire subsystem devoted to Windows programming called Windows Forms. The primary support for Windows Forms is contained in the System.Windows.Forms namespace. A form encapsulates the basic functionality necessary to create a window, display it on the screen, and receive messages. A form can represent any type of window, including the main window of the application, a child window, or even a dialog box.

The Form Class

Form contains significant functionality of its own, and it inherits additional functionality.

Two of its most important base classes are **System.ComponentModel.Component**, which supports the .NET component model, and **System.Windows.Forms.Control.** The Control class defines features common to all Windows controls.

Creating Windows form Application

Creating a Windows Forms application is largely just a matter of instantiating and extending theWindows Forms and GDI+ classes. In a nutshell, you typically complete the following steps:

1. Create a new project defining the structure of a Windows Forms application.
2. Define one or more Forms (classes derived from the Form class) for the windows
in your application.

3. Use the Designer to add controls to your forms (such as textboxes and checkboxes), and
then configure the controls by setting their properties and attaching event handlers.
4. Add other Designer-managed components, such as menus
or image lists.
5. Add code to your form classes to provide functionality.
6. Compile and run the program

Web Forms/Application

Web Forms are the heart and soul of **ASP.NET**. Web Forms are the User Interface (UI) elements that give Web applications their look and feel. Web Forms are similar to Windows Forms in that they provide properties, methods, and events for the controls that are placed onto them.
Web Forms are made up of two components: the visual portion (the ASPX file), and the code behind the form, which resides in a separate class file.

The Purpose of Web Forms

Web Forms and ASP.NET were created to overcome some of the limitations of ASP. Thesenew strengths include:
• Separation of HTML interface from application logic
• A rich set of server-side controls that can detect the browser
and send out appropriate markup language such as HTML
• Less code to write due to the data binding capabilities of the
new server-side .NET controls
• Event-based programming model that is familiar to
Microsoft Visual Basic programmers
• Compiled code and support for multiple languages, as
opposed to ASP which was interpreted as Microsoft Visual
Basic Scripting (VBScript) or Microsoft Jscript.
•Allows third parties to create controls that provide additional functionality

Exception Handling

An exception is an unwanted or unexpected event, which occurs during the execution of a program i.e at runtime, that disrupts the normal flow of the program's instructions.
This unwanted event is known as Exception.

Errors:

Errors are unexpected issues that may arise during computer program execution.
Errors cannot be handled.
All Errors are exceptions.

Exceptions:

Exceptions are unexpected events that may arise during runtime.
Exceptions can be handled using try-catch mechanisms.
All exceptions are not errors.
Exceptions provide a way to transfer control from one part of a program to another. C# exception handling is built upon four keywords: try, catch, finally, and throw.

try - A try block identifies a block of code for which particular exceptions is activated. It is followed by one or more catch blocks.

catch - A program catches an exception with an exception handler at the place in a program where you want to handle the problem. The catch keyword indicates the catching of an exception.

finally - The finally block is used to execute a given set of statements, whether an exception is thrown or not thrown. For example, if you open a file, it must be closed whether an exception is raised or not.

throw - A program throws an exception when a problem shows up. This is done using a throw keyword.

Exception Hierarchy

All the exceptions are derived from the base class Exception which gets further divided into two branches as ApplicationException and SystemException.
SystemException is a base class for all CLR or program code generated errors.
ApplicationException is a base class for all application related exceptions.
There are different kinds of exceptions which can be generated in C# program:

Divide By Zero exception: It occurs when the user attempts to divide by zero
Out of Memory exceptions: It occurs when then the program tries to use excessive memory
Index out of bound Exception: Accessing the array element or index which is not present in it.
Stackoverflow Exception: Mainly caused due to infinite recursion process

Null Reference Exception : Occurs when the user attempts to reference an object which is of NULL type.
Example

```
public void division(int num1, int num2) {
try {
result = num1 / num2;
}
catch (DivideByZeroException e) {
Console.WriteLine("Exception caught: {0}", e);
}
```

DataReader Object: The DataReader Object **provides a connection oriented data access** to the Data Sources. A **Connection Object** can contain only one DataReader at a time and the connection in the DataReader remains open, also it cannot be used for any other purpose while data is being accessed. When we started to read from a DataReader it should always be open and positioned **prior to the first record**. A DataReader **provides an easy way for the programmer to read the data from a database**. DataReader is a **stream-based, forward read-only** cursor **because it moves forward through the data**. The DataReader not only allows you to move forward through each record of the database, but it also **enables you to parse the data from each column**.
Syntax: COmmand cmd= new Command ("select * from students",Con);
DataReader dr = cmd.ExecuteReader();
DataReader Methods:
Read(): The **Read()** method in the DataReader is used to **read the rows from DataReader** and it always moves forward to a new valid row, if any row exist .
Syntax: DataReader.Raed();
NextResult(): This method is used **to navigate from current table to next table.**
Systax: dr.NextResult();

GetValue(int index): This method is used **for getting the column value from the row** which pointer was pointing by specifying column index position.
Systax: dr.GetValue(0).ToString();
Or
dr[0].ToString();
Or
dr[“sal”].ToString();
GetName(int index): This method returns the **name of the column for specified index**.
Syntax: dr.GetName(0);

DataReader properties:
fieldcount: It **returns the number of columns present** in the result set.
Syntax: dr.fieldcount;
Example: create an ADO.NET application in C#, to retrieve the values from the table using DataReader object.
Procedure:
Step1: place two buttons on form and name it as next and stop. Step2: place two labels for representing column names.
Step3: place two text boxes on form for displaying values from the table.

```
using System;
using System.Collections.Generic; using System.ComponentModel; using System.Data;
using System.Drawing; using System.Linq; using System.Text;
using System.Windows.Forms; using System.Data.OleDb;
namespace WindowsFormsApplication25 {
public partial class Form1 : Form {
public Form1() {
InitializeComponent(); }
OleDbConnection ocon; OleDbDataReader dr;
private void Form1_Load(object sender, EventArgs e) {
ocon = new OleDbConnection("Provider=MSDAORA.1; user id=system; password=raja");
ocon.Open();
OleDbCommand cmd2 = new OleDbCommand("select * from student4", ocon);
dr = cmd2.ExecuteReader(); label1.Text = dr.GetName(0); label2.Text = dr.GetName(1);
}
private void button1_Click(object sender, EventArgs e) {
if (dr.Read()) {
```

```
textBox1.Text = dr.GetValue(0).ToString(); textBox2.Text =
dr.GetValue(1).ToString();
}
else {
MessageBox.Show("no more records");
}
}
private void button2_Click(object sender, EventArgs e) {
ocon.Close();
MessageBox.Show(ocon.State.ToString()); }
}
}
```

Features of DataReader:

1. **Faster access to data from the data source** as it is connection oriented.
2. It **can hold multiple tables in it at a time**. To load multiple tables into a DataReder **pass**
multiple select statements as arguments to command object separated by a semicolon.

Syntax: COmmand cmd= new Command(—select * from students: select * from Teacher‖,Con);

DataReader dr = cmd.ExecuteReader();

Drawbacks of DataReader:

1. As it is connection oriented, **which requires continuous connection** with data source while
we are accessing the data, **so there is a chance of performance degradation if there are**
more number of clients accessing data at the same time.
2. It gives **forward only access** to the data that is it allows going either to next record or table
but not to previous record or table.
3. It is **read only object which will not allow any changes to data** that is present in it.

DataSet:

➢ It is a class present under **System.Data** namespace designed for holding and managing of data on client machines apart from DataReader. DataSet class provides the following features**.**

1. It is designed in disconnected architecture which does not require any permanent
Connection with the data source for holding of data.
2. It allows us to move in any direction that is either top to bottom or bottom to top.
3. It is updatable that is changes can be made to data present in DataSet and those
changes can be sent back to database.
4. DataSet is also capable of holding multiple tables in it.
5. It provides options for searching and storing of data that is present under DataSet.
6. It provides options for establishing relations between the tables that are present under
DataSet.

ACCESSING DATA FROM DATASET: DataSet is a collection of tables where each table is represented as a class **DataTable** and **identified by its index position or name**. Every DataTable again is a **collection of rows and columns** where each row is represented as a class **DataRo**w and **identified by its index position** and each column is represented as a class **DataColumn** and **indentified by its index position or name**.

The object can be created for DataSet class by using the following syntax:

Syntax: **DataSet ds=new DataSet();**

1. Accessing DataTable from DataSet:

Syntax: ds.Tables[index] Or ds.Tables[—tablename‖]

Example: ds.Tables[0]
Or ds.Tables[—emp‖]

2. Accessing DataRow From DataTable

Syntax: ds.datatable.Rows[index]

Example: ds.Tables[0].Rows[0]

3. Accessing DataColumn From DataTable

Syntax: ds.datatable.Columns[index] Or

ds.datatable.Columns[—name‖] **Example:** ds.Tables[0].Column[0]
Or ds.Tables[0].Column[—eno‖]
4. Accessing a cell from DataTable
Syntax: ds.datatable.Rows[row][col]
Example: ds.Tables[0].Rows[0][0]
ds.Tables[0].Rows[0][—eno‖]

Datagridview control: The DataGridView control provides a powerful and flexible way to display data in a tabular format. DataGridView control is updatable which allows us to add, modify or delete records.By using DataSource property we can directly bind DataGridView control to the DataTable. The speciality of DataGridView is any changes made on data which is present on DataGridView, reflects directly into the datatable to which it was bound. Control is directly binded to the datatable by using its DataSource property.

Syntax:	**DataSet ds = new DataSet();dataGridView1.DataSource=ds.Tables[0];**

Example: create an ADO.NET application in C#, to demonstrate dataGridView Control. Procedure:
Step1: Place one DataGridView on the form

```
using System;
using System.Collections.Generic; using System.ComponentModel; using System.Data;
using System.Drawing; using System.Linq; using System.Text;
using System.Windows.Forms; using System.Data.OleDb;
namespace WindowsFormsApplication27 {
public partial class Form1 : Form {
public Form1()
{ InitializeComponent(); }
DataSet dset=new DataSet(); OleDbConnection ocon;
private void Form1_Load(object sender, EventArgs e) {
ocon = new OleDbConnection("Provider=MSDAORA.1;user id=system; password=raja");
OleDbDataAdapter ad = new OleDbDataAdapter(" select * from student4 ", ocon); ad.Fill(dset,"student4");
dataGridView1.DataSource = dset.Tables["student4"];
}
}
}
```

DataView:

- The DataView provides **different views of the data** stored in a DataTable.
- DataView can be used to **sort, filter, and search** the data in a DataTable, additionally we can **add new rows, modify the data in existing rows** and delete **the content** in a DataTable .
- Whatever Changes made to a DataView affect the underlying DataTable automatically, and changes made to the underlying DataTable automatically affect any DataView objects that are viewing the DataTable.
- We can **create DataView in two different ways**. We can use the **DataView Constructor**, or you can **create a reference to the DefaultView Property of the DataTable**.
- The DataView constructor can be empty, or it can take either a DataTable as a single argument, or a DataTable along with filter criteria, sort criteria, and a row state filter.
- **Creating Dataview object as**

Dataview dv=new Dataview(); dv = ds.Tables[0].DefaultView;

- DataVIew filters the data with the help of RowFilter property.

Syntax: dv.RowFilter=”value”;

Step1: place combobox on form.
Step2: place the DataGridView control on the form.

```
using System;
using System.Collections.Generic; using System.ComponentModel; using
System.Data;
using System.Drawing; using System.Linq; using System.Text;
using System.Windows.Forms; using System.Data.OleDb;
namespace WindowsFormsApplication30 {
public partial class Form1 : Form {
public Form1() {
InitializeComponent(); }
OleDbConnection con; OleDbDataAdapter da; DataSet ds;
private void Form1_Load(object sender, EventArgs e) {
con = new OleDbConnection("Provider=MSDAORA.1; user
id=system;password=raja");
da = new OleDbDataAdapter("select * from student4", con); ds = new
DataSet();
da.Fill(ds, "student4");
```

```
dataGridView1.DataSource = ds.Tables["student4"].DefaultView;
comboBox1.DataSource = ds.Tables["student4"]; comboBox1.DisplayMember
= "SID";
}
private void comboBox1_SelectedIndexChanged(object sender, EventArgs e) {
DataView dv = new DataView();
dv = ds.Tables["student4"].DefaultView; dv.RowFilter = "SID=7";
}
}
}
```