Contents

Unit 1: Language Preliminaries

Unit outcome: By the end of this unit on Language Preliminaries for .NET, students will gain a foundational grasp of essential .NET concepts, equipping them to create robust, maintainable applications. They will understand the .NET framework's architecture, including the CLR and FCL, and the compilation process. Core language constructs and object-oriented principles, such as inheritance and polymorphism, will enable students to write organized, extensible code. They will explore advanced types, delegates, events, partial classes, and generics for modular programming. The unit also covers file I/O, LINQ, and lambda expressions for data handling, error handling, attributes, and asynchronous programming to support efficient, responsive applications.

What is a framework?

A framework is a foundation or structure that makes building applications easier. Instead of starting from scratch, a framework provides ready-made tools and components that can be used to build applications faster. It works like a template that can be adapted and modified to fit a project's needs.

Writing code from scratch is time-consuming and can lead to mistakes and bugs. It is necessary to make sure that the code is clean, well-tested, and easy for other developers to understand. This can be hard to do without help. Frameworks make things easier by providing a ready-made structure that reduces errors. They offer a general template that can be adjusted to fit project needs, making it easier for others to understand and work with the code since they are already familiar with how the framework works.

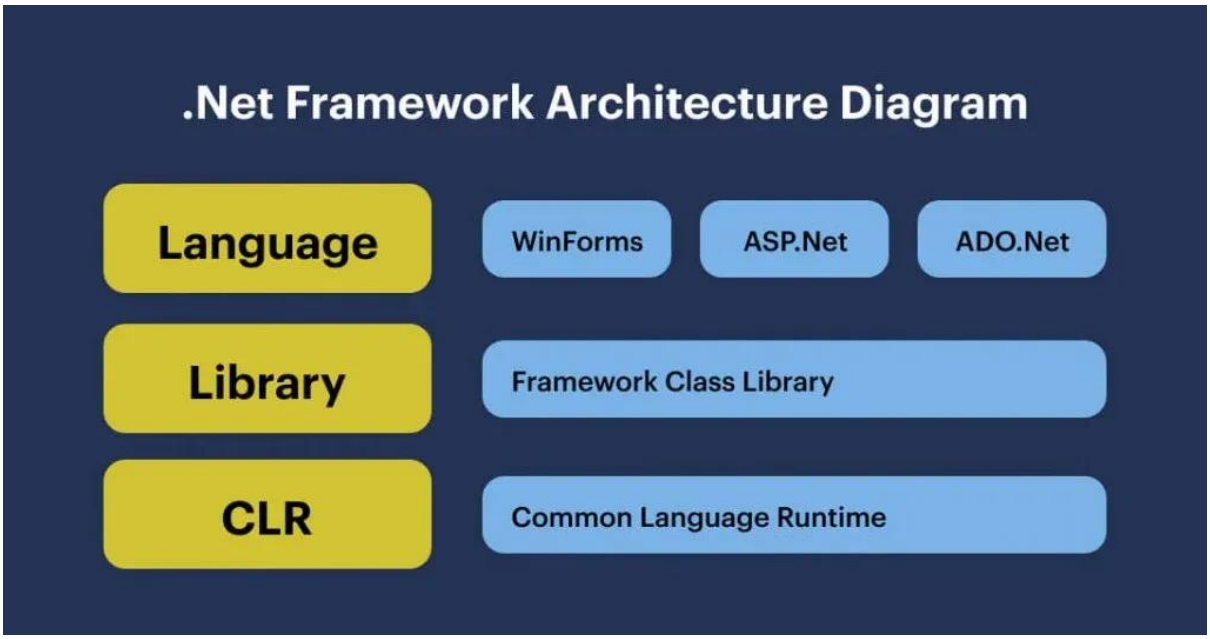
Frameworks are built using specific programming languages. Some popular frameworks include Django, Ruby on Rails, React, Angular, Node, Laravel and many more. These frameworks help developers create powerful, feature-rich software quickly and efficiently.

Introduction to .Net framework

.NET Framework is a software development platform developed by Microsoft. The framework was meant to create an application which would run on Windows platform. The first version of .NET framework is 1.0 which was released in the year 2002. And the latest version of .NET

framework is 4.8.1 which was released in August 2022. .NET framework can be used to create form based, console-based Application and web-based application. Framework also supports various programming languages such as Visual Basic, C#, ASP, J#.

Components/Architecture of .Net Framework



1. Common Language Runtime(CLR): It is an important part of a .NET framework that works like a virtual component of the .NET Framework to executes the different languages program like c#, Visual Basic, etc. A CLR also helps to convert a source code into the byte code, and this byte code is known as CIL (Common Intermediate Language) or MSIL (Microsoft Intermediate Language). After converting into a byte code, a CLR uses a JIT compiler at run time that helps to convert a CIL or MSIL code into the machine or native code.

2. Framework Class Library(FCL): It provides the various system functionality in the .NET Framework, that includes classes, interfaces and data types, etc. to create multiple functions and different types of applications such as desktop, web, mobile application, etc. In other words, it can be defined as, it provides a base on which various applications, controls and components are built in .NET Framework.

3. Applications
    a. WinForms: Used for developing form-based applications.
    b. ASP.NET: Used for developing web based applications

c. ADO.NET: Used for developing applications to interact with databases.

Design Principle of .NET Framework

1. Interoperability
   Interoperability in .NET means it can work well with code, libraries, and applications that were written using different programming languages or designed for different systems. This lets us combine old or external code with your new .NET code so you don't have to start from scratch.
   Example: Let's say we have a library of code written in C++ that performs special scientific calculations, and you want to use this library in a new C# application you're building in .NET. Instead of rewriting all that C++ code in C#, .NET lets you use that C++ library directly. This is very useful when you're working with other systems, or when companies want to update older systems without rebuilding them from the ground up.

2. Portability
   Portability in .NET means that we can create an application that can run on different operating systems (like Windows, macOS, or Linux) and different types of devices without needing to rewrite the code. .NET achieves this by using versions like .NET Core and .NET 5+, which are designed to work cross-platform.

3. Security
   Security in .NET means that the framework has built-in tools to help protect applications from unauthorized access, hacking, and other security threats. It also helps keep sensitive data safe. .NET includes features like user permissions (so only certain users can access certain parts of an app), encryption (to protect data), and sandboxing (running code in a restricted way).

4. Memory Management
   Memory management in .NET means that the framework automatically handles the memory application needs. It allocates memory when we create objects (data structures, variables, etc.) and frees it up when they're no longer needed. This is done by a system called the Garbage Collector (GC). This way, app uses memory efficiently, and you don't need to manually manage memory.

5. Simplified Deployment
   Simplified deployment means .NET makes it easy to share, install, or update applications on other computers without complicated setups. There are multiple ways to deploy applications, such as XCOPY deployment (simple file copy), Global Assembly Cache (GAC) (for shared code libraries), and ClickOnce (a one-click installation for users).

Compilation and Execution of .Net Applications

The compilation and execution process of .NET applications involves several steps that convert source code into an executable program running in the .NET environment. Here's an overview of each stage:

1. Source Code Compilation: When a developer writes code in a .NET-supported language like C#, it is first compiled by a language-specific compiler (e.g., csc.exe for C#) into an Intermediate Language (IL), also known as Microsoft Intermediate Language (MSIL). This compilation step results in a binary file, typically a .dll (library) or .exe (executable), containing IL code.
2. Just-In-Time (JIT) Compilation: When the .NET application is run, the Common Language Runtime (CLR) loads the IL code and performs Just-In-Time (JIT) compilation. The JIT compiler translates IL into native machine code. This native code is then executed directly by the operating system, resulting in better performance as the code runs.
3. Execution by the CLR: Once JIT compilation is complete, the CLR manages the execution of the application. It provides essential runtime services such as memory management, garbage collection, security, and exception handling. The CLR's managed environment ensures that resources are allocated efficiently, and that the application is protected against certain vulnerabilities like memory leaks.
4. Cross-Platform Execution with .NET Core and Beyond: For applications developed using .NET Core or .NET 5+, the compilation process supports cross-platform compatibility. These versions include runtime libraries and JIT compilers for different operating systems (Windows, macOS, Linux), allowing the same IL code to run on any supported platform.
5. Ahead-of-Time (AOT) Compilation (Optional): In certain cases, .NET applications can also undergo Ahead-of-Time (AOT) compilation, which compiles the IL directly into native code before runtime, removing the need for JIT compilation. AOT is often used for scenarios that require reduced startup time or environments where JIT isn't feasible, such as mobile or some IoT devices.

The .NET compilation and execution pipeline takes high-level code, converts it into a platform-neutral IL, and finally, during execution, compiles it into native machine code optimized for the host environment. This two-step compilation enables cross-language compatibility, optimized execution, and efficient resource management, making .NET applications robust and adaptable across platforms.

Basic Structure of .Net Application

Common Language Constructs

1. Data Types
2. Variables
3. Conditional Statements
    a. If statements
    b. If then Else statements
    c. If then ElseIf statements
    d. Select Case statements
    e. Switch case
4. Looping Statements
    a. For loop
    b. While loop
    c. Do while loop
    d. ForEach loop
5. Jump Statements
    a. Break
    b. Continue
    c. Goto
    d. return
6. Functions(Methods)
    a. Access Modifiers

Constructor

a constructor is a special method used to initialize objects of a class. It is automatically called when an instance of the class is created. A constructor has the same name as the class and does not have a return type. If no constructor is explicitly defined, C# provides a default constructor that takes no arguments.

Simple example of Constructor:

```
using System;

class Car
{
        // Fields
        private string model;
        private int year;

        // Constructor to initialize the Car object
        public Car(string model, int year)
        {
        this.model = model;
        this.year = year;
        }

        // Method to display car details
        public void DisplayInfo()
        {
        Console.WriteLine($"Model: {model}, Year: {year}");
        }
}

class Program
{
        // Main method - entry point of the program
        static void Main()
        {
        // Create a Car object using the constructor
        Car myCar = new Car("Tesla Model S", 2023);

        // Display car information
        myCar.DisplayInfo();
        }
}
```

Types of Constructor

a. Default Constructor

   A default constructor in C# is a constructor that does not take any parameters and is used to initialize an object with default values. If no constructor is explicitly defined in a class, C# automatically provides a default constructor that initializes fields to their default values (e.g., null for reference types, 0 for numeric types). If we define our own constructor without parameters, we are explicitly providing a default constructor.

Example:

```
using System;

class Car
{
        // Fields
        private string model;
        private int year;

        // Default constructor
        public Car()
        {
        model = "Unknown";
        year = 0;
        Console.WriteLine("Default constructor called.");
        }

        // Method to display car details
        public void DisplayInfo()
        {
        Console.WriteLine($"Model: {model}, Year: {year}");
        }
}

class Program
{
        static void Main()
        {
        // Create a Car object using the default constructor
        Car myCar = new Car();

        // Display car information
        myCar.DisplayInfo();
        }
}
```

b. Parameterized Constructor

A parameterized constructor in C# is a constructor that takes one or more parameters to initialize an object with specific values when it is created. Which allows passing values to the constructor at the time of object creation, enabling more flexible and customized initialization.

Example:

```
using System;

class Car
{
        private string model;
        private int year;

        // Parameterized constructor
        public Car(string model, int year)
        {
        this.model = model;
        this.year = year;
        }

        // Method to display car details
        public void DisplayInfo()
        {
        Console.WriteLine($"Model: {model}, Year: {year}");
        }
}

class Program
{
        static void Main()
        {
        // Create a Car object using the parameterized constructor
        Car myCar = new Car("Tesla Model 3", 2023);

        // Display car information
        myCar.DisplayInfo();
        }
}
```

c. Copy Constructor

A copy constructor in C# is a constructor that creates a new object as a copy of an existing object of the same class. It is used to initialize a new object with the values from another object of the same type.

The copy constructor takes an instance of the same class as a parameter. It creates a new object with the same values as the object passed to it. It is typically used for shallow copying (for reference types) or deep copying (for objects containing references to other objects).

Example:

```
using System;

class Car
{
	private string model;
	private int year;

	// Constructor to initialize a new Car object
	public Car(string model, int year)
	{
	this.model = model;
	this.year = year;
	}

	// Copy constructor - creates a new Car object from an existing one
	public Car(Car otherCar)
	{
	this.model = otherCar.model;
	this.year = otherCar.year;
	Console.WriteLine("Copy constructor called.");
	}

	// Method to display car details
	public void DisplayInfo()
	{
	Console.WriteLine($"Model: {model}, Year: {year}");
	}
}

class Program
{
	static void Main()
	{
	// Create a Car object using the parameterized constructor
	Car myCar = new Car("Tesla Model S", 2023);

	// Create a new Car object using the copy constructor
	Car copyCar = new Car(myCar);

	// Display information for both cars
	Console.WriteLine("Original Car:");
	myCar.DisplayInfo();
```

```
        Console.WriteLine("\nCopied Car:");
        copyCar.DisplayInfo();
        }
}
```

d. Private Constructor

A private constructor in C# is a constructor that is declared with the private access modifier, meaning it cannot be called directly from outside the class. This type of constructor is used to restrict object creation and is often used in design patterns like Singleton, or in cases where we don't want an object to be instantiated directly but need to control its instantiation in other ways.

Example:

```
using System;

class Car
{
        private string model;
        private int year;

        // Private constructor
        private Car(string model, int year)
        {
        this.model = model;
        this.year = year;
        Console.WriteLine("Private constructor called.");
        }

        // Static method to create a Car object
        public static Car CreateCar(string model, int year)
        {
        return new Car(model, year);
        }

        // Method to display car details
        public void DisplayInfo()
        {
        Console.WriteLine($"Model: {model}, Year: {year}");
        }
}
```

```
class Program
{
        static void Main()
        {
        // Create a Car object using the static method (not directly through the
constructor)
        Car myCar = Car.CreateCar("Tesla Model X", 2024);

        // Display car information
        myCar.DisplayInfo();
        }
}
```

e. Static Constructor

A static constructor in C# is a constructor that is used to initialize the type itself, rather than instances of the class. It is called automatically when the type is first accessed, and it runs only once, regardless of how many instances of the class are created. A static constructor does not take any parameters and is used to initialize static members or perform actions that are necessary for the type itself.

Key points:

- A static constructor does not accept any parameters.
- It is called automatically by the runtime when the class is first accessed (either when an instance is created or when a static member is accessed).
- It can only access and initialize static fields and properties, not instance members.
- It is executed only once, regardless of how many instances of the class are created.

Examples:

```
using System;

class Car
{
        private string model;
        private int year;

        // Static field
        private static int carCount;
```

```
        // Static constructor (called once when the class is first accessed)
        static Car()
        {
        carCount = 0;
        Console.WriteLine("Static constructor called. carCount initialized.");
        }

        // Regular constructor
        public Car(string model, int year)
        {
        this.model = model;
        this.year = year;
        carCount++;  // Increment the static field each time a car is created
        Console.WriteLine($"Car created: {model} - {year}");
        }

        // Static method to get the count of cars
        public static int GetCarCount()
        {
        return carCount;
        }

        // Method to display car details
        public void DisplayInfo()
        {
        Console.WriteLine($"Model: {model}, Year: {year}");
        }
}

class Program
{
        static void Main()
        {
        // Accessing the static constructor
        Console.WriteLine("Before creating any Car objects:");

        // Creating Car objects (which will trigger the static constructor)
        Car car1 = new Car("Tesla Model S", 2023);
        Car car2 = new Car("BMW X5", 2022);

        // Displaying information about the car count
        Console.WriteLine($"\nTotal number of cars: {Car.GetCarCount()}");

        // Display car information
```

```
            car1.DisplayInfo();
            car2.DisplayInfo();
            }
    }
```

Properties

In C#, properties are members that provide a mechanism to read, write, or compute the values of private fields. Properties combine the concepts of fields and methods and serve as a way to encapsulate data. Properties are usually accessed like fields but are defined with methods called getters and setters. This encapsulation provides control over the value being assigned or returned.Properties are a fundamental feature of object-oriented programming (OOP) in C#, and they help in maintaining the principles of encapsulation by controlling access to private fields.

It has two methods: get and set. The get method is used for getting the value of a field while the set method is used for setting value for the field.

A property in C# can be defined with the following syntax:

```
public Type PropertyName
{
        get { return field; }
        set { field = value; }
}
```

- type: The data type of the property (e.g., int, string).
- PropertyName: The name of the property (e.g., Age, Name).
- get: The accessor that retrieves the value of the property.
- set: The accessor that assigns a value to the property.

Example of Property:

```
private int age;  // private field

public int Age
{
        get { return age; }
        set { age = value; }
}
```

Types of Property:

1. Read Only Property

A read-only property is a property that only has a getter and no setter. This means the property can only be read (accessed) but cannot be modified from outside the class once it's set, typically at the time of initialization.

Example:

```
using System;

public class Person
{
        // Private field (backing field)
        private string _name;

        // Read-only property for Name
        public string Name
        {
        get { return _name; }
        }

        // Initializing the field via constructor
        public Person()
        {
        _name = "John Doe";  // Set value of the private field
        }
}

class Program
{
        static void Main()
        {
        Person person = new Person();

        // Accessing the read-only property
        Console.WriteLine($"Name: {person.Name}");
        }
}
```

2. Write Only Property
   A write-only property in C# is a property that only has a setter and no getter. This means the property can only be assigned a value, but you cannot read its value directly from outside the class.

Example:

```
using System;

public class Person
{
	// Private field (backing field)
	private string _password;

	// Write-only property for Password
	public string Password
	{
	set { _password = value; }  // Only setter, no getter
	}

	// Constructor to set the write-only property
	public Person(string password)
	{
	Password = password;  // Set password via write-only property
	}

	// Method to demonstrate internal use of the password
	public void DisplayPasswordLength()
	{
	Console.WriteLine($"Password length is: {_password.Length}");
	}
}

class Program
{
	static void Main()
	{
	// Creating object and passing value for Password through the constructor
	Person person = new Person("MySecretPassword");

	// Accessing the password indirectly via a method
	person.DisplayPasswordLength(); // Prints the length of the password

	// The following line will result in a compile-time error:
```

```
        // Console.WriteLine(person.Password);  // Error: 'Password' is write-only
        }
}
```

3. Read and Write property:
   A read-write property in C# is a property that has both a getter (to retrieve the value) and a setter (to modify the value). This allows us to read and update the property value from outside the class.

   Example: Research Yourself.

Arrays

An array in C# is a collection of elements of the same data type stored in a contiguous memory location. It allows developers to store multiple values in a single variable, with elements accessed using an index that starts from 0. Arrays in C# have a fixed size, meaning the number of elements must be defined at the time of creation, and they are homogeneous, requiring all elements to be of the same type. Arrays provide an efficient way to manage and manipulate collections of data, and C# offers built-in methods like Array.Sort() and Array.Length for easy operations.

To declare an array, the syntax is ***type[] arrayName***, where type specifies the data type of elements, and arrayName is the variable name. The array can be initialized using new type[size], where size defines the number of elements.
For example,
int[] numbers = new int[5];
creates an integer array with five elements, which can be assigned values individually using numbers[0] = 10;.
Alternatively, arrays can be initialized directly with predefined values, such as
int[] numbers = { 10, 20, 30, 40, 50 };

Two Dimensional Array

A two-dimensional array in C# is a data structure that stores elements in a grid-like or tabular format, using rows and columns. It is often referred to as a matrix. Each element in the array is accessed using two indices: one for the row and the other for the column. Two-dimensional arrays are particularly useful for representing data like tables, matrices, or game boards. The array is declared using two sets of square brackets ([,]) to indicate rows and columns.

To declare a two-dimensional array, we need to specify the data type, followed by the array name, and then initialize it with the number of rows and columns.

For example, **int[,] array = new int[3, 2];**
creates a two-dimensional integer array with 3 rows and 2 columns. Elements can be assigned values individually using their indices, such as array[0, 1] = 10;,
or during declaration using nested braces, such as
int[,] array = { { 1, 2 }, { 3, 4 }, { 5, 6 } };.

Example:

```
using System;

class Program
{
    static void Main()
    {
        // Declare and initialize a 2D array
        int[,] numbers = {
            { 1, 2 },
            { 3, 4 },
            { 5, 6 }
        };

        // Display the elements in the 2D array
        Console.WriteLine("Elements in the 2D array:");
        for (int i = 0; i < numbers.GetLength(0); i++) // Loop through rows
        {
            for (int j = 0; j < numbers.GetLength(1); j++) // Loop through columns
            {
                Console.Write(numbers[i, j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Note: The GetLength(dimension) method can be used to retrieve the size of specific dimensions, where dimension is 0 for rows and 1 for columns.

Jagged Arrays

A jagged array in C# is an array of arrays, where each "inner" array can have a different length, unlike a two-dimensional array which has a fixed number of rows and columns. It is called "jagged" because the lengths of its subarrays can differ, forming an irregular or non-uniform

structure. Jagged arrays are particularly useful when dealing with collections of data where the number of elements in each row is not consistent.

To declare a jagged array, we use two sets of square brackets ([][]) and initialize it with the appropriate number of rows, where each row is itself an array.
For example, ***int[][] jaggedArray = new int[3][];*** creates a jagged array with 3 rows, but the number of elements in each row can vary.

You can also directly initialize a jagged array with specific values, such as
int[][] jaggedArray =
{
new int[] {1, 2},
new int[] {3, 4, 5},
new int[] {6}
};
where the first row has 2 elements, the second row has 3 elements, and the third row has 1 element.

Example:

```
using System;

class Program
{
    static void Main()
    {
        // Declare and initialize a jagged array
        int[][] jaggedArray = {
            new int[] { 1, 2 },
            new int[] { 3, 4, 5 },
            new int[] { 6 }
        };

        // Display the elements in the jagged array
        Console.WriteLine("Elements in the jagged array:");
        for (int i = 0; i < jaggedArray.Length; i++) // Loop through rows
        {
            for (int j = 0; j < jaggedArray[i].Length; j++) // Loop through each row's columns
            {
                Console.Write(jaggedArray[i][j] + " ");
            }
            Console.WriteLine();
        }
    }
}
```

Indexers

An indexer in C# is a special type of property that allows an object to be accessed using array-like syntax. It is defined using ***this*** keyword, making it possible to treat an instance of a class as a virtual array. Indexers simplify the process of accessing and modifying data stored within objects by allowing parameters, such as an index, to be passed directly. They consist of get and set accessors, which define the logic for retrieving and assigning values. The primary purpose of indexers is to provide an intuitive and clean way to work with data collections encapsulated within an object.

For example, consider a class Department that stores a collection of employees in an internal array. By implementing an indexer, employees can be set and retrieved using an index, just like with an array. The get accessor retrieves the employee at a given index, while the set accessor assigns a value (an employee object) at that index.

Example:

```
using System;

class Employee
{
    public int ID { get; set; }
    public string Name { get; set; }

    public Employee(int id, string name)
    {
        ID = id;
        Name = name;
    }

}

class Department
{
    private Employee[] employees = new Employee[5]; // Internal array to store employees

    // Indexer to set and get employees using an index
    public Employee this[int index]
    {
        get
        {
            if (index >= 0 && index < employees.Length)
            {
                return employees[index];
```

```
            }
            else
            {
                throw new IndexOutOfRangeException("Invalid index");
            }
        }
        set
        {
            if (index >= 0 && index < employees.Length)
            {
                employees[index] = value;
            }
            else
            {
                throw new IndexOutOfRangeException("Invalid index");
            }
        }
    }
}

class Program
{
    static void Main()
    {
        Department department = new Department();

        // Setting employees using the indexer
        department[0] = new Employee(1, "Alice");
        department[1] = new Employee(2, "Bob");
        department[2] = new Employee(3, "Charlie");

        // Getting employees using the indexer
        for (int i = 0; i < 3; i++)
        {
            Console.WriteLine(department[i]);
        }

            // Modifying an employee using the indexer
        Console.WriteLine("\nModifying employee at index 1...");
        department[1] = new Employee(4, "David");

        // Displaying updated employees
        Console.WriteLine("\nUpdated Employees in the Department:");
        for (int i = 0; i < 3; i++)
```

```
        {
            Console.WriteLine(department[i]);
        }

    }
}
```

Inheritance

Inheritance is a key concept in object-oriented programming that lets us create a new class based on an existing one. The new class inherits the properties and methods of the existing class and can also introduce its own properties and methods. Inheritance helps in reusing code, making code maintenance easier, and organizing the code more effectively.

Base Class (Parent): The class that provides properties and methods for other classes to inherit.
Derived Class (Child): The class that inherits features from the base class, and can also add its own properties and methods or modify the inherited ones.

Types of Inheritance:

1. Single Inheritance

   Single inheritance is a type of inheritance where a derived class (child class) inherits properties and methods from only one base class (parent class). This means that the derived class can access the members (fields, properties, methods) of the base class and can also define its own members or override the inherited ones.

   In single inheritance, the derived class is limited to inheriting from just one parent class, which helps to keep the hierarchy simple and avoids complexity.

   Example:

```
using System;

// Base class (Parent)
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}

// Derived class (Child)
class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("Dog is barking.");
    }
}

class Program
{
    static void Main()
    {
        // Create an object of the derived class
        Dog dog = new Dog();

        // Inherited method from Animal class
        dog.Eat();

        // Method specific to Dog class
        dog.Bark();
    }
}
```

2. Multi-Level Inheritance

   Multi-level inheritance is a type of inheritance where a class is derived from another class that is already a derived class. In simple terms, it’s like a chain of classes where one class inherits from another, and that second class also inherits from a third class.

   In this type of inheritance, each class in the chain can add its own features (properties and methods) while still inheriting the ones from its parent class.

Example:

```
using System;

// Base class (Parent)
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}

// Derived class 1 (Child of Animal)
class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("Dog is barking.");
    }
}

// Derived class 2 (Grandchild of Animal)
class Puppy : Dog
{
    public void Play()
    {
        Console.WriteLine("Puppy is playing.");
    }
}

class Program
{
    static void Main()
    {
        // Create an object of the grandchild class
        Puppy puppy = new Puppy();

        // Methods inherited from Animal and Dog classes
        puppy.Eat();   // From Animal
        puppy.Bark();  // From Dog
        puppy.Play();  // From Puppy
    }
```

```
}
```

3. Hierarchical Inheritance

   Hierarchical inheritance occurs when multiple classes inherit from a single parent class. In this type of inheritance, one base class is shared by two or more derived classes. Each of the derived classes can add its own unique features (properties and methods) while still inheriting the common properties and methods from the base class.
   Example:

```
using System;

// Base class (Parent)
class Animal
{
    public void Eat()
    {
        Console.WriteLine("Animal is eating.");
    }
}

// Derived class 1 (Child)
class Dog : Animal
{
    public void Bark()
    {
        Console.WriteLine("Dog is barking.");
    }
}

// Derived class 2 (Child)
class Cat : Animal
{
    public void Meow()
    {
        Console.WriteLine("Cat is meowing.");
    }
}

class Program
{
    static void Main()
    {
```

```
        // Create an object of the Dog class
        Dog dog = new Dog();
        dog.Eat();   // Inherited from Animal
        dog.Bark();  // Specific to Dog

        // Create an object of the Cat class
        Cat cat = new Cat();
        cat.Eat();   // Inherited from Animal
        cat.Meow();  // Specific to Cat
    }
}
```