**Difference between Concrete Class , Abstract Class And Interface ?**

Sure, let's break down the differences between concrete classes, abstract classes, and interfaces:

1. ****Concrete Class****:
   * A concrete class is a regular class that can be instantiated directly to create objects.
   * It may contain fields, properties, methods, constructors, and other members.
   * Concrete classes provide complete implementations for all of their methods.
   * Objects of concrete classes can be created using the **new** keyword.

Example:

public class Animal {

public string Name { get; set; }

public void MakeSound() {

Console.WriteLine("Animal makes a sound");

}

}

1. ****Abstract Class****:
   * An abstract class is a class that cannot be instantiated directly and is marked with the **abstract** keyword.
   * It may contain abstract methods (methods without a body) as well as regular methods and fields.
   * Abstract classes serve as blueprints for other classes and provide a common base for related classes to inherit from.

Concrete subclasses must provide implementations for all abstract methods.

Example:

public abstract class Shape {

public abstract double CalculateArea();

}

public class Circle : Shape {

public double Radius { get; set; }

public override double CalculateArea() {

return Math.PI \* Radius \* Radius;

}

}

---------------------------------------------------------------

}

1. ****Interface****:
   * An interface is a reference type in C# that defines a contract for behavior without providing any implementation.
   * It contains only method signatures, properties, indexers, and events (without any method bodies).
   * Classes can implement multiple interfaces, allowing for multiple inheritance of behavior.
   * Interfaces are useful for defining common behavior across unrelated classes.

Example:

public interface IPlayable {

void Play();

}

public class VideoPlayer : IPlayable {

public void Play() {

Console.WriteLine("Playing video...");

}

}

public class MusicPlayer : IPlayable {

public void Play() {

Console.WriteLine("Playing music...");

}

}

In summary, concrete classes provide full implementations, abstract classes provide partial implementations with the ability to define abstract methods, and interfaces define contracts for behavior without providing any implementation. Each serves different purposes and can be used based on the requirements of the application.

**Static And Not Static Class In C#**  
  
In C#, a static class is a class that cannot be instantiated and can only contain static members (fields, methods, properties). It's often used to create utility classes or classes that provide functionality without needing to maintain state.

A non-static class, on the other hand, can be instantiated, meaning you can create objects (instances) of that class. Non-static classes can contain both static and non-static members. They are typically used to represent objects that have state and behavior.

Here's a brief comparison:

Static class:

* Cannot be instantiated.
* Can only contain static members.
* Typically used for utility functions, extension methods, or classes that provide functionality without needing instance-specific data.

Non-static class:

* Can be instantiated to create objects (instances).
* Can contain both static and non-static members.
* Typically used to represent objects that have state and behavior, where each instance may have different data.

In summary, static classes are useful for grouping related utility methods or constants, while non-static classes are used to model real-world entities with state and behavior.

----------------------------------------------------------------------------------------------

What is the use of static methods in non static class ?

Static methods in a non-static class serve several purposes:

Utility Functions: They can provide utility functions that are related to the class but don't require an instance of the class to operate. These methods can be called directly on the class itself without needing to create an instance.

Factory Methods: Static methods can be used as factory methods to create instances of the class or to return specific instances based on certain criteria.

Initialization Logic: They can be used for initialization logic that is common to all instances of the class. These methods can perform setup tasks or initialize static data that's shared among all instances.

Helper Methods: Static methods can serve as helper methods that perform tasks related to the class but don't require access to instance-specific data. They can encapsulate common behavior that's reusable across different instances of the class.

In summary, static methods in a non-static class provide functionality that is associated with the class itself rather than with specific instances of the class. They can be used for various purposes such as utility functions, factory methods, initialization logic, or helper methods.

--CROSS APPLY AND OUTER APPLY

--In summary, OUTER APPLY includes all rows from the outer table (It acts like LEFT JOIN),

--while CROSS APPLY only includes matching rows (It acts like INNER JOIN).

-- Sample Employees table

CREATE TABLE #Employees (

EmployeeID INT PRIMARY KEY,

EmployeeName VARCHAR(50)

);

-- Sample Orders table

CREATE TABLE #Orders (

OrderID INT PRIMARY KEY,

EmployeeID INT,

OrderDate DATE

);

-- Sample data for demonstration

INSERT INTO #Employees VALUES (1, 'John'), (2, 'Alice');

INSERT INTO #Orders VALUES (101, 1, '2024-01-01'), (102, 1, '2024-01-05'), (103, 2, '2024-02-10');

----------------------------

Select \* from #Employees ;

Select \* from #Orders ;

SELECT e.EmployeeID, e.EmployeeName, o.OrderID, o.OrderDate

FROM #Employees e

OUTER APPLY (

SELECT top 1 OrderID, OrderDate

FROM #Orders

WHERE EmployeeID = 0

) AS o;

SELECT e.EmployeeID, e.EmployeeName, o.OrderID, o.OrderDate

FROM #Employees e

CROSS APPLY (

SELECT top 1 OrderID, OrderDate

FROM #Orders

WHERE EmployeeID = e.EmployeeID

) AS o;

