Give the Examples of Encapsulation, Inheritance, Abstraction And Polymorphism Technical CONCEPTS C#

ChatGPT

Sure, here are examples of each of the four fundamental object-oriented programming concepts (encapsulation, inheritance, abstraction, and polymorphism) in C#:

**Encapsulation:**

Encapsulation is the bundling of data and methods that operate on that data into a single unit. It helps in hiding the internal state of an object from the outside world and only exposing the necessary functionalities. In C#, this is commonly achieved through classes and access modifiers like private, public, protected, etc.

**Example:**

using System;

public class BankAccount

{

private string accountNumber;

private decimal balance;

public void Deposit(decimal amount)

{

balance += amount;

}

public void Withdraw(decimal amount){

if (balance >= amount)

{

balance -= amount;

}

else

{

Console.WriteLine("Insufficient funds");

}

}

public decimal GetBalance()

{

return balance;

}

}

**Inheritance:**

Inheritance is the mechanism by which a new class can be derived from an existing class. The new class inherits properties and behaviors of the existing class and can also add its own unique properties and behaviors. In C#, inheritance is implemented using the ':' symbol.

**Example:**

public class Shape

{

public virtual double Area()

{

return 0;

}

}

public class Rectangle : Shape

{

public double Length { get; set; }

public double Width { get; set; }

public override double Area()

{

return Length \* Width;

}

}

**Abstraction:**

Abstraction is the process of hiding the implementation details and showing only the essential features of the object. It helps in reducing programming complexity and avoiding redundancy. In C#, abstraction is achieved using abstract classes and interfaces.

**Example:**

public abstract class Vehicle

{

public abstract void Start();

public abstract void Stop();

}

public class Car : Vehicle

{

public override void Start()

{

Console.WriteLine("Car started");

}

public override void Stop()

{

Console.WriteLine("Car stopped");

}

}

**Polymorphism:**

Polymorphism allows objects of different types to be treated as objects of a common type. It enables methods to behave differently based on the object they are called on. In C#, polymorphism can be achieved through method overriding and method overloading.

**Example:**

public class Animal

{

public virtual void MakeSound()

{

Console.WriteLine("Some generic sound");

}

}

public class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Woof");

}

}

public class Cat : Animal

{

public override void MakeSound()

{

Console.WriteLine("Meow");

}

}

These examples demonstrate how encapsulation, inheritance, abstraction, and polymorphism are implemented in C# and how they contribute to building robust and flexible object-oriented systems.

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

**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

**1] OUTER APPLY:**

OUTER APPLY is used to apply a table-valued function (TVF) to each row of a table expression.

It returns all rows from the left table expression, even if the TVF returns no results for some rows.

If the TVF doesn't produce any rows for a particular row in the left table expression, the corresponding columns from the TVF result are filled with NULL values.

2] CROSS APPLY:

CROSS APPLY is used to apply a table-valued function (TVF) to each row of a table expression.

It returns only the rows from the left table expression that produce results when applied with the TVF.

If the TVF doesn't produce any rows for a particular row in the left table expression, the corresponding row is not included in the result set.

In simple terms, OUTER APPLY includes all rows from the left table expression, while CROSS APPLY includes only rows that produce results when combined with the TVF.

--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

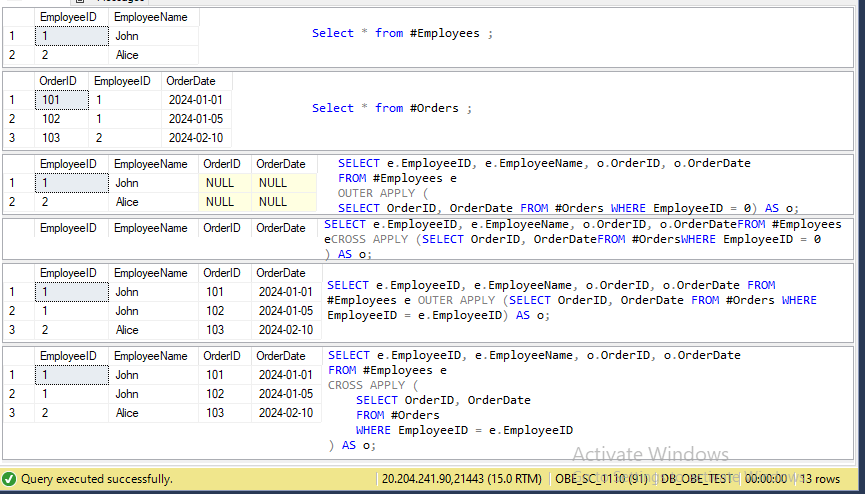
CROSS APPLY (

SELECT top 1 OrderID, OrderDate

FROM #Orders

WHERE EmployeeID = e.EmployeeID

) AS o;



**Normalization** in SQL is a process used to organize data in a relational database efficiently. Here are some simple points to explain normalization to an interviewer:

Definition:Normalization is the process of organizing data in a database to minimize redundancy and dependency.

It involves dividing large tables into smaller ones and defining relationships between them.

Main Goals:Eliminate redundant data: Reduces storage space and improves data integrity.

Minimize data anomalies: Prevents insertion, update, and deletion anomalies that can occur due to redundant data.

Normal Forms:Normalization is typically achieved through a series of normal forms (e.g., First Normal Form, Second Normal Form, etc.).

Each normal form represents a set of rules that a database must follow to be considered normalized.

Example:Consider a table of customer information that includes customer name, address, and phone number.

Instead of storing all this information in a single table, we can divide it into multiple tables such as Customers, Addresses, and Phone Numbers.

By doing this, we eliminate redundancy and create a more efficient database structure.

**Benefits**: improved data integrity: Reduces the risk of inconsistent data.

Better query performance: Smaller, normalized tables can be queried more efficiently.

Easier maintenance: Changes to data are easier to implement and manage.

Normalization Levels:

****Unnormalized Form (UNF)****

**CREATE TABLE Customers (**

**CustomerID INT PRIMARY KEY,**

**FirstName VARCHAR(50),**

**LastName VARCHAR(50),**

**Address VARCHAR(100),**

**City VARCHAR(50),**

**State VARCHAR(50),**

**ZipCode VARCHAR(10),**

**Phone VARCHAR(15)**

**);**

**First Normal Form (1NF)**: Ensures each column contains atomic values(Eg. PhoneNumber , but can be Names(First ,Last Name)) and there are no repeating groups.

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Phone VARCHAR(15)

);

CREATE TABLE Addresses (

AddressID INT PRIMARY KEY,

CustomerID INT,

Address VARCHAR(100),

City VARCHAR(50),

State VARCHAR(50),

ZipCode VARCHAR(10),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

**Second Normal Form (2NF)**: Builds on 1NF by ensuring non-key attributes are fully functionally dependent on the primary key.

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Phone VARCHAR(15)

);

CREATE TABLE Addresses (

AddressID INT PRIMARY KEY,

CustomerID INT,

Address VARCHAR(100),

City VARCHAR(50),

State VARCHAR(50),

ZipCode VARCHAR(10),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

Amount DECIMAL(10, 2),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

**Third Normal Form (3NF)**: Builds on 2NF by removing transitive dependencies between non-key attributes.

CREATE TABLE Customers (

CustomerID INT PRIMARY KEY,

FirstName VARCHAR(50),

LastName VARCHAR(50),

Phone VARCHAR(15)

);

CREATE TABLE Addresses (

AddressID INT PRIMARY KEY,

CustomerID INT,

Address VARCHAR(100),

City VARCHAR(50),

StateID INT,

ZipCode VARCHAR(10),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID),

FOREIGN KEY (StateID) REFERENCES States(StateID)

);

CREATE TABLE States (

StateID INT PRIMARY KEY,

StateName VARCHAR(50)

);

CREATE TABLE Orders (

OrderID INT PRIMARY KEY,

CustomerID INT,

OrderDate DATE,

Amount DECIMAL(10, 2),

FOREIGN KEY (CustomerID) REFERENCES Customers(CustomerID)

);

Considerations:Normalization is not always necessary: Small databases or those with limited data complexity may not require full normalization.

Over-normalization can be detrimental: Excessive normalization can lead to complex database structures and decreased query performance.

**Overall, normalization is a fundamental concept in database design that aims to optimize data storage, improve data integrity, and facilitate efficient data retrieval.**

**REFER**

[DBMS 1NF - javatpoint](https://www.javatpoint.com/dbms-first-normal-form)

**The six guiding principles of RESTful APIs are:**

1] **Client-Server Architecture**

This principle separates the client and server responsibilities. Allowing them to work independently.

Example: In a web application, the client (browser) makes requests to the server (back end) for data or actions. The server processes these requests and sends back responses.

2] **Statelessness** : Each request from the client to the server must contain all the information necessary to understand and fulfill the request.

3]**Cache ability** : To reduce the server performance .

Example: A web page can be cached by the browser to reduce load times for subsequent visits to the same page.

4]**Layered System:**The API can have multiple layers (e.g., load balancers, proxies) between the client and server.

Example: A client sends a request to an API gateway, which forwards the request to the appropriate microservice in a distributed system.

5]**Uniform Interface**: This principle defines a standard way for clients and servers to communicate, promoting simplicity and flexibility.

Example: REST APIs typically use HTTP methods (GET, POST, PUT, DELETE) to perform CRUD operations (Create, Read, Update, Delete) on resources identified by URIs.

**Code on Demand (optional)**: Servers can provide executable code (e.g., JavaScript) to clients, allowing them to extend their functionality.

These principles guide the design and implementation of RESTful APIs, making them scalable, efficient, and interoperable.

Sure, here's a simple example of creating RESTful APIs in ASP.NET MVC using C#, along with a brief explanation of load balancers:

Creating RESTful APIs in ASP.NET MVC:

csharp

Copy code

// UserController.cs

using System.Collections.Generic;

using System.Web.Http;

namespace YourNamespace.Controllers

{

public class UserController : ApiController

{

private static List<User> users = new List<User>

{

new User { Id = 1, Name = "John" },

new User { Id = 2, Name = "Alice" },

new User { Id = 3, Name = "Bob" }

};

// GET: api/user

public IEnumerable<User> Get()

{

return users;

}

// GET: api/user/1

public IHttpActionResult Get(int id)

{

var user = users.Find(u => u.Id == id);

if (user == null)

{

return NotFound();

}

return Ok(user);

}

// POST: api/user

public IHttpActionResult Post([FromBody] User user)

{

users.Add(user);

return CreatedAtRoute("DefaultApi", new { id = user.Id }, user);

}

// PUT: api/user/1

public IHttpActionResult Put(int id, [FromBody] User user)

{

var existingUser = users.Find(u => u.Id == id);

if (existingUser == null)

{

return NotFound();

}

existingUser.Name = user.Name;

return Ok(existingUser);

}

// DELETE: api/user/1

public IHttpActionResult Delete(int id)

{

var user = users.Find(u => u.Id == id);

if (user == null)

{

return NotFound();

}

users.Remove(user);

return Ok(user);

}

}

}

public class User

{

public int Id { get; set; }

public string Name { get; set; }

}

Load Balancers:

Load balancers are networking devices that distribute incoming network traffic across multiple servers.