Frequently and likely asked interview questions on Object-Oriented Programming (OOP) in .NET, tailored for a candidate with 8+ years of experience, along with detailed answers.

**### Question 1: Can you explain the four main principles of Object-Oriented Programming and how they are implemented in .NET?**

\*\*Answer:\*\*

The four main principles of Object-Oriented Programming (OOP) are Encapsulation, Abstraction, Inheritance, and Polymorphism.

1. \*\*Encapsulation:\*\* This is the concept of wrapping data (fields) and methods (functions) that operate on the data into a single unit, known as a class. In .NET, encapsulation is implemented using access modifiers such as `private`, `public`, `protected`, and `internal`. For example:

```csharp

public class Person

{

private string name;

private int age;

public string GetName()

{

return name;

}

public void SetName(string name)

{

this.name = name;

}

public int Age

{

get { return age; }

set { age = value; }

}

}

```

2. \*\*Abstraction:\*\* Abstraction involves hiding the complex implementation details and showing only the necessary features of the object. In .NET, this can be achieved using abstract classes and interfaces.

```csharp

public abstract class Animal

{

public abstract void MakeSound();

}

public class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Bark");

}

}

```

3. \*\*Inheritance:\*\* This is the mechanism by which one class can inherit the properties and methods of another class. In .NET, inheritance is implemented using the `:` symbol.

```csharp

public class Animal

{

public void Eat()

{

Console.WriteLine("Eating");

}

}

public class Dog : Animal

{

public void Bark()

{

Console.WriteLine("Barking");

}

}

```

4. \*\*Polymorphism:\*\* Polymorphism allows methods to do different things based on the object it is acting upon. It can be achieved through method overriding and method overloading in .NET.

```csharp

public class Animal

{

public virtual void MakeSound()

{

Console.WriteLine("Some sound");

}

}

public class Dog : Animal

{

public override void MakeSound()

{

Console.WriteLine("Bark");

}

}

```

**### Question 2: What is the difference between an abstract class and an interface in .NET? When would you use one over the other?**

\*\*Answer:\*\*

An abstract class and an interface are both used to define abstract types in .NET, but they have key differences:

- \*\*Abstract Class:\*\*

- Can contain both abstract methods (without implementation) and concrete methods (with implementation).

- Can have fields, constructors, and destructors.

- Supports access modifiers for its members.

- Can inherit from another class and implement multiple interfaces.

```csharp

public abstract class Shape

{

public abstract void Draw();

public void Show()

{

Console.WriteLine("Showing shape");

}

}

```

- \*\*Interface:\*\*

- Can only contain method signatures (without implementation), properties, events, and indexers.

- Does not have fields, constructors, or destructors.

- All members are implicitly public and abstract.

- Supports multiple inheritance, allowing a class to implement multiple interfaces.

```csharp

public interface IDrawable

{

void Draw();

}

public class Circle : IDrawable

{

public void Draw()

{

Console.WriteLine("Drawing Circle");

}

}

```

\*\*When to use:\*\*

- Use \*\*abstract classes\*\* when you want to provide common base functionality that can be inherited by derived classes.

- Use \*\*interfaces\*\* when you want to define a contract that multiple classes can implement, allowing for more flexible and decoupled designs.

### Question 3: How does method overriding differ from method overloading in .NET?

\*\*Answer:\*\*

- \*\*Method Overriding:\*\*

- Allows a subclass to provide a specific implementation of a method that is already defined in its base class.

- Requires the use of the `virtual` keyword in the base class method and the `override` keyword in the derived class method.

- Method signatures (name and parameters) must be the same.

```csharp

public class BaseClass

{

public virtual void Display()

{

Console.WriteLine("Base Display");

}

}

public class DerivedClass : BaseClass

{

public override void Display()

{

Console.WriteLine("Derived Display");

}

}

```

- \*\*Method Overloading:\*\*

- Allows a class to have multiple methods with the same name but different signatures (different parameter types, number of parameters, or both).

- Does not involve inheritance or overriding.

```csharp

public class MathOperations

{

public int Add(int a, int b)

{

return a + b;

}

public double Add(double a, double b)

{

return a + b;

}

}

```

**### Question 4: What is the SOLID principle in OOP and how do you apply it in .NET?**

\*\*Answer:\*\*

The SOLID principles are a set of five design principles intended to make software designs more understandable, flexible, and maintainable. They are:

1. \*\*Single Responsibility Principle (SRP):\*\* A class should have only one reason to change, meaning it should have only one job or responsibility.

- \*\*Example:\*\* A `User` class should only handle user-related data, while a `UserRepository` class should handle database operations related to users.

2. \*\*Open/Closed Principle (OCP):\*\* Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification.

- \*\*Example:\*\* Use interfaces or abstract classes to allow new functionality through inheritance without modifying existing code.

```csharp

public interface IShape

{

void Draw();

}

public class Circle : IShape

{

public void Draw()

{

Console.WriteLine("Drawing Circle");

}

}

```

3. \*\*Liskov Substitution Principle (LSP):\*\* Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program.

- \*\*Example:\*\* Derived classes should override base class methods in a way that does not break functionality expected by the base class.

```csharp

public class Rectangle

{

public virtual void SetWidth(int width) { }

public virtual void SetHeight(int height) { }

}

public class Square : Rectangle

{

public override void SetWidth(int width)

{

base.SetWidth(width);

base.SetHeight(width);

}

public override void SetHeight(int height)

{

base.SetHeight(height);

base.SetWidth(height);

}

}

```

4. \*\*Interface Segregation Principle (ISP):\*\* Clients should not be forced to depend on interfaces they do not use.

- \*\*Example:\*\* Split large interfaces into smaller, more specific ones.

```csharp

public interface IPrint

{

void Print();

}

public interface IScan

{

void Scan();

}

public class MultiFunctionPrinter : IPrint, IScan

{

public void Print()

{

Console.WriteLine("Printing");

}

public void Scan()

{

Console.WriteLine("Scanning");

}

}

```

5. \*\*Dependency Inversion Principle (DIP):\*\* High-level modules should not depend on low-level modules. Both should depend on abstractions.

- \*\*Example:\*\* Use dependency injection to inject dependencies rather than hardcoding them.

```csharp

public interface ILogger

{

void Log(string message);

}

public class DatabaseLogger : ILogger

{

public void Log(string message)

{

Console.WriteLine("Logging to database: " + message);

}

}

public class UserService

{

private readonly ILogger \_logger;

public UserService(ILogger logger)

{

\_logger = logger;

}

public void CreateUser(string name)

{

// User creation logic

\_logger.Log("User created: " + name);

}

}

```

**### Question 5: How do you implement dependency injection in .NET?**

\*\*Answer:\*\*

Dependency injection (DI) is a design pattern used to achieve Inversion of Control (IoC) between classes and their dependencies. In .NET, DI can be implemented in several ways, but the most common approach is using the built-in DI container provided by ASP.NET Core.

1. \*\*Constructor Injection:\*\* This is the most common form of DI where dependencies are provided through the class constructor.

```csharp

public interface IRepository

{

void Save();

}

public class Repository : IRepository

{

public void Save()

{

Console.WriteLine("Data saved");

}

}

public class Service

{

private readonly IRepository \_repository;

public Service(IRepository repository)

{

\_repository = repository;

}

public void Execute()

{

\_repository.Save();

}

}

```

2. \*\*Registering Services in the DI Container:\*\*

In an ASP.NET Core application, you register services in the `Startup.cs` file.

```csharp

public class Startup

**### Question 6: What are design patterns and can you explain the Singleton pattern and its implementation in .NET?**

\*\*Answer:\*\*

Design patterns are typical solutions to common problems in software design. They are like blueprints that can be applied to solve design problems in various situations.

\*\*Singleton Pattern:\*\*

The Singleton pattern ensures a class has only one instance and provides a global point of access to it. It is useful when exactly one object is needed to coordinate actions across the system.

\*\*Implementation in .NET:\*\*

To implement the Singleton pattern in .NET, you typically make the constructor private, provide a static property to get the instance, and ensure thread safety.

```csharp

public sealed class Singleton

{

private static readonly Lazy<Singleton> instance = new Lazy<Singleton>(() => new Singleton());

private Singleton() { }

public static Singleton Instance

{

get { return instance.Value; }

}

public void DoSomething()

{

Console.WriteLine("Singleton instance method called");

}

}

```

Here, the `Lazy<T>` type ensures that the instance is created in a thread-safe manner the first time it's accessed.

**### Question 7: Can you explain what dependency injection is and the different types of dependency injection in .NET?**

\*\*Answer:\*\*

Dependency injection (DI) is a design pattern used to implement Inversion of Control (IoC), where dependencies are injected into a class rather than the class creating them itself. This promotes loose coupling and enhances testability and maintainability.

\*\*Types of Dependency Injection:\*\*

1. \*\*Constructor Injection:\*\*

- Dependencies are provided through a class constructor.

```csharp

public class Service

{

private readonly IRepository \_repository;

public Service(IRepository repository)

{

\_repository = repository;

}

}

```

2. \*\*Property Injection:\*\*

- Dependencies are set through public properties.

```csharp

public class Service

{

public IRepository Repository { get; set; }

}

```

3. \*\*Method Injection:\*\*

- Dependencies are passed as parameters to a method.

```csharp

public class Service

{

public void SetRepository(IRepository repository)

{

// Use the repository

}

}

```

\*\*Registering Services in ASP.NET Core:\*\*

In ASP.NET Core, you register services in the `Startup.cs` file using the built-in DI container.

```csharp

public class Startup

{

public void ConfigureServices(IServiceCollection services)

{

services.AddTransient<IRepository, Repository>();

services.AddScoped<Service>();

}

}

```

**### Question 8: What is reflection in .NET and how can you use it to dynamically load assemblies?**

\*\*Answer:\*\*

Reflection is the ability of a program to inspect and interact with its own metadata at runtime. It allows for examining the types, properties, methods, and other metadata of assemblies.

\*\*Using Reflection to Load Assemblies:\*\*

1. \*\*Loading an Assembly:\*\*

```csharp

Assembly assembly = Assembly.Load("AssemblyName");

```

2. \*\*Getting Types from the Assembly:\*\*

```csharp

Type[] types = assembly.GetTypes();

foreach (Type type in types)

{

Console.WriteLine(type.FullName);

}

```

3. \*\*Creating Instances Dynamically:\*\*

```csharp

Type myType = assembly.GetType("Namespace.ClassName");

object obj = Activator.CreateInstance(myType);

```

4. \*\*Invoking Methods:\*\*

```csharp

MethodInfo method = myType.GetMethod("MethodName");

method.Invoke(obj, null);

```

Reflection is powerful but should be used judiciously due to performance overhead and potential security risks.

**### Question 9: How do you handle multiple inheritance in .NET?**

\*\*Answer:\*\*

.NET does not support multiple inheritance for classes, but it allows a class to implement multiple interfaces, providing a form of multiple inheritance.

\*\*Example:\*\*

```csharp

public interface IPrinter

{

void Print();

}

public interface IScanner

{

void Scan();

}

public class MultiFunctionDevice : IPrinter, IScanner

{

public void Print()

{

Console.WriteLine("Printing");

}

public void Scan()

{

Console.WriteLine("Scanning");

}

}

```

In this example, `MultiFunctionDevice` class implements both `IPrinter` and `IScanner` interfaces, allowing it to inherit behavior from both.

**### Question 10: Can you explain the concept of extension methods and provide an example of how they are used in .NET?**

\*\*Answer:\*\*

Extension methods allow you to add new methods to existing types without modifying the original type. They are static methods defined in static classes and the first parameter specifies the type to extend with the `this` modifier.

\*\*Example:\*\*

```csharp

public static class StringExtensions

{

public static bool IsPalindrome(this string str)

{

int len = str.Length;

for (int i = 0; i < len / 2; i++)

{

if (str[i] != str[len - i - 1])

return false;

}

return true;

}

}

```

\*\*Usage:\*\*

```csharp

class Program

{

static void Main()

{

string word = "radar";

bool result = word.IsPalindrome();

Console.WriteLine($"Is '{word}' a palindrome? {result}");

}

}

```

In this example, the `IsPalindrome` method is an extension method for the `string` type, allowing it to be called as if it were an instance method of the `string` class.

These advanced questions should help a seasoned .NET developer demonstrate a deep understanding of OOP principles and advanced .NET features during an interview.