Dependency injection

Dependency Injection (DI) in C# is a design pattern used to implement Inversion of Control (IoC) for resolving dependencies. It allows a class to receive its dependencies from an external source rather than creating them itself, promoting loose coupling and easier testing.

**Simple Explanation**

Imagine you have a coffee machine. To make coffee, it needs a coffee bean grinder. Without DI, the coffee machine would create its own grinder, making it tightly coupled to a specific grinder type. With DI, you provide the grinder to the coffee machine from outside, allowing you to easily swap out the grinder for another type if needed.

**Real-life Example**

Let's say we have a Car class that needs an Engine to run. Without DI, the Car class would create its own Engine, like this:

public class Engine

{

public void Start() => Console.WriteLine("Engine started.");

}

public class Car

{

private Engine \_engine;

public Car()

{

\_engine = new Engine(); // Car creates its own Engine

}

public void Start()

{

\_engine.Start();

Console.WriteLine("Car started.");

}

}

This approach makes it hard to change the Engine type without modifying the Car class.

With DI, you provide the Engine to the Car from outside, like this:

public class Engine

{

public void Start() => Console.WriteLine("Engine started.");

}

public class Car

{

private readonly Engine \_engine;

// Car receives its Engine from outside (injection)

public Car(Engine engine)

{

\_engine = engine;

}

public void Start()

{

\_engine.Start();

Console.WriteLine("Car started.");

}

}

Now, you can inject any Engine into the Car without changing the Car class.

**Using Dependency Injection in C#**

In a real C# application, you typically use a DI container to manage dependencies. Here’s an example using .NET Core's built-in DI container:

1. **Define the interface and classes:**

public interface IEngine

{

void Start();

}

public class GasEngine : IEngine

{

public void Start() => Console.WriteLine("Gas engine started.");

}

public class ElectricEngine : IEngine

{

public void Start() => Console.WriteLine("Electric engine started.");

}

public class Car

{

private readonly IEngine \_engine;

public Car(IEngine engine)

{

\_engine = engine;

}

public void Start()

{

\_engine.Start();

Console.WriteLine("Car started.");

}

}

Configure the DI container:

using Microsoft.Extensions.DependencyInjection;

class Program

{

static void Main(string[] args)

{

var serviceProvider = new ServiceCollection()

.AddTransient<IEngine, GasEngine>() // Register GasEngine as IEngine

.AddTransient<Car>() // Register Car

.BuildServiceProvider();

var car = serviceProvider.GetService<Car>();

car.Start();

}

}

In this setup, ServiceCollection registers the dependencies, and the ServiceProvider resolves them. You can easily switch from GasEngine to ElectricEngine by changing the registration:

var serviceProvider = new ServiceCollection()

.AddTransient<IEngine, ElectricEngine>() // Switch to ElectricEngine

.AddTransient<Car>()

.BuildServiceProvider();

This approach makes your code more flexible, testable, and maintainable.