📘 Dependency Injection (DI) in .NET

🧠 What is Dependency Injection?

Dependency Injection (DI) is a design pattern used to achieve Inversion of Control (IoC) between classes and their dependencies. Rather than a class instantiating its dependencies itself, it receives them from an external source, usually the framework's DI container.

💡 Key Concepts:

- **Dependency**: A service or object a class depends on (e.g., a logger, database service).
- **Dependent**: The class that uses the dependency.
- **Injection**: The act of passing the dependency into the class.

⚠️ Without Dependency Injection (Hardcoded Dependency)

```
public class Logger
{
    public void Log(string message)
    {
        Console.WriteLine("LOG: " + message);
    }
}

public class ReportService
{
    private Logger _logger = new Logger();

    public void GenerateReport()
    {
        _logger.Log("Generating report...");
    }
}
```

❌ Problems:

- **Tight coupling** between `ReportService` and `Logger`.
- **Hard to test** (e.g., no easy way to substitute a mock logger).
- **Low flexibility** (cannot swap logger without modifying the code).

✅ With Dependency Injection

Step 1: Define an Interface

```
public interface ILogger
{
    void Log(string message);
}
{
    void Log(timestamp time_of_action);
}
{
    void Log(string message);
}

{
    void Log(string message);
}
```

Step 2: Implement the Interface

```
public class ConsoleLogger : ILogger
{
    public void Log(string message)
    {
        Console.WriteLine("LOG: " + message);
    }
}
```

Step 3: Inject the Dependency

```
public class ReportService
{
    private readonly ILogger _logger;

    // Constructor Injection
    public ReportService(ILogger logger)
    {
        _logger = logger;
    }

    public void GenerateReport()
    {
        _logger.Log("Generating report...");
    }
}
```

Step 4: Configure DI in .NET (Program.cs)

```
builder.Services.AddScoped<ILogger, ConsoleLogger>();
builder.Services.AddScoped<ReportService>();
```

🧪 Types of Dependency Injection

1️⃣ Constructor Injection (Recommended)

```
public class ReportService
{
    private readonly ILogger _logger;

    public ReportService(ILogger logger)
    {
        _logger = logger;
    }

    public void GenerateReport()
    {
        _logger.Log("Report generated.");
    }
}
```

Explanation:

- The `ILogger` interface is passed to the constructor.
- This enforces that the class cannot exist without its dependency.
- Most preferred because it makes the dependency mandatory and the class immutable.

2️⃣ Property Injection

```
public class ReportService
{
    public ILogger Logger { get; set; }

    public void GenerateReport()
    {
        Logger?.Log("Report generated via property injection.");
    }
}
```

Explanation:

- The dependency is set via a public property.
- Optional dependency: the class can work even if the property is not set, hence should be used with caution.
- Useful in scenarios where dependency may not always be required.

3️⃣ Method Injection

```
public class ReportService
{
    public void GenerateReport(ILogger logger)
    {
        logger.Log("Report generated via method injection.");
    }
}
```

Explanation:

- The dependency is passed directly into the method that needs it.
- Useful when the dependency is needed only in specific methods.
- Promotes short-lived or transient use of services.

🔄 Comparison: Without vs With DI

Without DI	With DI
Tight coupling	Loose coupling
Hard to test	Easy to test
Difficult to maintain	Easy to extend or replace services
Hardcoded implementations	Flexible and configurable

🚀 Advantages of Using DI

- Enables better **testability** through mocking.
- Promotes **loose coupling** and **separation of concerns**.
- Makes code **cleaner, maintainable, and flexible**.
- Enhances **scalability** of large applications.

Summary

1. **Start with an Interface**: Define what the dependency should do.
2. **Create Implementations**: One or more classes that implement the interface.

3. **Inject the Interface**: Use constructor injection in your classes.
4. **Register in DI Container**: Configure in `Program.cs` or `Startup.cs`.

Dependency Injection is a fundamental part of building scalable, testable .NET applications.