**Dependency Injection (DI) in C#**

**Dependency Injection (DI)** is a design pattern used to achieve **Inversion of Control (IoC)** between classes and their dependencies. In simpler terms, instead of a class creating its own dependencies (e.g., creating objects of other classes it needs to function), these dependencies are **injected** into the class, typically by a framework or an external mechanism.

The goal of DI is to improve **decoupling** between classes, making code more flexible, testable, and maintainable.

Note: Inversion of control (IoC): a design pattern where the responsibility for creating and managing objects is transferred to an IoC container, rather than being controlled directly by application code.

**Why Dependency Injection?**

1. **Loose Coupling**: DI reduces the tight coupling between classes, making the code more modular.
2. **Testability**: Dependencies can be easily replaced with mock objects during testing.
3. **Maintainability**: If a class's dependencies change, you don’t need to modify the class itself.
4. **Readability**: Dependency injection improves the clarity of the code by clearly specifying what dependencies are needed by a class.

**Components of Dependency Injection**

1. **Service (or Dependency)**: The object that the class depends on.
2. **Client**: The class that depends on the service.
3. **Injector**: The code responsible for injecting the dependency into the client.

**Types of Dependency Injection**

1. **Constructor Injection**: Dependencies are provided through the class constructor.
2. **Property Injection (Setter Injection)**: Dependencies are provided via public properties.
3. **Method Injection**: Dependencies are passed through methods.

**1. Constructor Injection**

The most common type of DI in C#. Dependencies are provided through the constructor when the object is created.

**Example:**



**Explanation**:

* The Application class depends on an ILogger instance to log messages.
* Instead of creating an instance of ConsoleLogger inside Application, it's passed via the constructor.
* The ILogger instance (ConsoleLogger) is created in the Main method and injected into the Application class.

**2. Property Injection (Setter Injection)**

In this method, dependencies are injected via public properties.

**Example:**

A screen shot of a computer program

Description automatically generated

**Explanation**:

* The Notification class depends on an IMessageSender to send notifications.
* Instead of passing the dependency via the constructor, it's set using a public property.

**3. Method Injection**

Dependencies are provided as method parameters.

**Example:**

A screen shot of a computer program

Description automatically generated

**Explanation**:

* The PaymentService class depends on an IPaymentProcessor to process payments.
* The dependency is injected into the method MakePayment.

**Dependency Injection in ASP.NET Core**

ASP.NET Core has a built-in **dependency injection container** that manages the lifecycle and injection of services automatically.

**Registering Services:**

In ASP.NET Core, you typically register dependencies in the Startup.cs class in the ConfigureServices method.

A screen shot of a computer

Description automatically generated

Here, ILogger is registered as a service, and the ConsoleLogger implementation will be injected wherever ILogger is required.

**Injection in Controllers:**

A screen shot of a computer program

Description automatically generated

In ASP.NET Core, the ILogger instance is automatically injected into the HomeController by the DI framework.

**Service Lifetimes in Dependency Injection**

When registering services in ASP.NET Core's DI container, you can specify the **lifetime** of the service:

1. **Transient** (AddTransient) — A new instance is created every time the service is requested.
2. **Scoped** (AddScoped) — A new instance is created once per request (in web applications).
3. **Singleton** (AddSingleton) — A single instance is created and shared throughout the application's lifetime.

**Example of registering services:**

A screenshot of a computer program

Description automatically generated

**Advantages of Dependency Injection**

1. **Loosely Coupled Code**: Classes are not responsible for creating their own dependencies, which makes the code easier to maintain and test.
2. **Testability**: DI makes unit testing easier because dependencies can be easily mocked or replaced during tests.
3. **Improved Flexibility**: By injecting interfaces, you can change implementations without modifying the client class.
4. **Better Control**: You can control the lifecycle and reuse of services (singleton, scoped, transient).

**Disadvantages of Dependency Injection**

1. **Complexity**: It adds an extra layer of complexity, especially for simple applications.
2. **Overhead**: Poorly designed DI setups can lead to performance overheads, especially if unnecessary dependencies are injected.

**Conclusion**

Dependency Injection is a powerful technique that fosters **loose coupling**, **maintainability**, and **testability** in C# applications. It shifts the responsibility of creating dependencies to an external entity, thus making the code more modular and flexible. This is especially useful in large applications or when working with frameworks like ASP.NET Core, which provides a built-in DI container.