Dependency Injection (DI) in C#

What is Dependency Injection (DI) in C#?

Dependency Injection (DI) is a design pattern used to implement Inversion of Control (IoC), in which the control of object creation and dependency management is transferred from the consumer to an external system or framework. It allows for better modularity, testing, and decoupling of components by injecting dependencies instead of allowing classes to instantiate their dependencies directly.

In simpler terms, Dependency Injection means that a class does not create its own dependencies; instead, it gets the necessary dependencies from the outside, making the system more maintainable and testable.

Example without Dependency Injection

Let's first look at a simple example where dependency injection is **not** used. Imagine we have a class `Car` that depends on an `Engine` class.

```
public class Engine
{     public void Start()
      {     Console.WriteLine("Engine started");      }
}
public class Car
{     private readonly Engine _engine;
      public Car()
      {     _engine = new Engine(); // The Car class directly instantiates Engine}
      public void Drive()
      {     _engine.Start();
      Console.WriteLine("Car is driving"); }
}
```

```
public class Program
{      public static void Main()
       {      Car car = new Car();
       car.Drive();
       }
}
```

Explanation:

- The `Car` class directly creates an instance of the `Engine` class (`_engine = new Engine();`).
- This tight coupling makes testing difficult because `Car` is always dependent on the specific implementation of `Engine`.
- This makes it harder to replace `Engine` with a different implementation if needed, or to mock it during testing.

Example with Dependency Injection

Now, let's modify the above example to use dependency injection. We'll use constructor injection, one of the most common DI techniques.

```
public interface IEngine
{      void Start();  }
public class Engine : IEngine
{      public void Start()
       {
       Console.WriteLine("Engine started");
       }
}
```

```
public class Car
{       private readonly IEngine _engine;  // Constructor injection - Engine dependency is injected via the constructor
        public Car(IEngine engine)
        {       _engine = engine;       }
        public void Drive()
        {       _engine.Start();   Console.WriteLine("Car is driving");       }}
public class Program
{       public static void Main()
        {       IEngine engine = new Engine();
        Car car = new Car(engine); // Injecting Engine dependency
        car.Drive();
}}
```

Explanation:

- We created an interface `IEngine` that `Engine` implements.
- The `Car` class now depends on `IEngine` rather than directly instantiating `Engine`. This allows us to inject any implementation of `IEngine` into `Car`.
- In `Main()`, we manually create an instance of `Engine` and inject it into the `Car` constructor.

Benefits of Dependency Injection:

1. **Loose Coupling**: `Car` no longer depends on a concrete implementation of `Engine`. It can work with any class that implements `IEngine`.
2. **Easier Testing**: During unit tests, we can provide a mock implementation of `IEngine`, making it easier to test the `Car` class in isolation.
3. **Flexibility**: The code is now more flexible. We can easily swap out the `Engine` implementation with a different one, such as a `MockEngine` for testing purposes.

Using a DI Container

In larger applications, it can be cumbersome to manually wire up dependencies. That's where Dependency Injection frameworks or containers, like `Microsoft.Extensions.DependencyInjection`, come in handy.

Example with DI Container in ASP.NET Core:

```
using Microsoft.Extensions.DependencyInjection;

public interface IEngine

{      void Start(); }

public class Engine : IEngine

{      public void Start()

       {      Console.WriteLine("Engine started");       }

}
```

```
public class Car
{       private readonly IEngine _engine;
        public Car(IEngine engine)
        {       _engine = engine;       }
        public void Drive()
        {
        _engine.Start();
        Console.WriteLine("Car is driving");
        }
}
```

```
public class Program
{       public static void Main()
        {       // Set up the dependency injection container
        var serviceCollection = new ServiceCollection();
        serviceCollection.AddTransient<IEngine, Engine>();
        serviceCollection.AddTransient<Car>();
        // Build the service provider to resolve dependencies
        var serviceProvider = serviceCollection.BuildServiceProvider();
        // Use the service provider to create an instance of Car
        var car = serviceProvider.GetService<Car>();
        car.Drive();    }}
```

Explanation:

- We use the `ServiceCollection` class to register services (`IEngine` and `Car`).
- `AddTransient<IEngine, Engine>()` tells the DI container that whenever `IEngine` is requested, it should create a new instance of `Engine`.
- `AddTransient<Car>()` registers the `Car` class, which automatically has its dependencies (in this case, `IEngine`) resolved by the container.
- The `ServiceProvider` is then used to create instances of classes with their dependencies automatically injected.

Types of Dependency Injection:

1. **Constructor Injection**: Dependencies are provided through the class constructor. This is the most common and recommended form of DI.
2. **Property Injection**: Dependencies are provided via public properties of the class.
3. **Method Injection**: Dependencies are provided as parameters to a method of the class.

Tasks to Practice Dependency Injection

1. **Task**: Refactor a class `Phone` that directly instantiates a `Battery` class to use dependency injection. Create an interface `IBattery` and inject it into `Phone`.
    - **Solution**:

```
public interface IBattery
{
        void Charge();
}
public class Battery : IBattery
{
        public void Charge()
        {
        Console.WriteLine("Battery is charging");
        }
}
```

```
public class Phone
{       private readonly IBattery _battery;
        // Constructor Injection
        public Phone(IBattery battery)
        {       _battery = battery;     }
        public void UsePhone()
        {       _battery.Charge();
        Console.WriteLine("Phone is in use");   }
}
public class Program
{       public static void Main()
        {       IBattery battery = new Battery();
        Phone phone = new Phone(battery); // Injecting Battery dependency
        phone.UsePhone();       }
}
```

Task: Use a DI container to resolve dependencies for a `Laptop` class that depends on an `IProcessor` interface. Implement the `Processor` class and use a DI container to inject it into `Laptop`.

- **Solution**:

```
using Microsoft.Extensions.DependencyInjection;

public interface IProcessor

{

        void Process();

}

public class Processor : IProcessor

{

        public void Process()

        {

        Console.WriteLine("Processor is running");

        }

}
```

```
public class Laptop
{       private readonly IProcessor _processor;
        public Laptop(IProcessor processor)
        {       _processor = processor; }
        public void Start()
        {
        _processor.Process();
        Console.WriteLine("Laptop started");
        }
}
```

```
public class Program
{     public static void Main()
      {     var serviceCollection = new ServiceCollection();
      serviceCollection.AddTransient<IProcessor, Processor>();
      serviceCollection.AddTransient<Laptop>();
      var serviceProvider = serviceCollection.BuildServiceProvider();
      var laptop = serviceProvider.GetService<Laptop>();
      laptop.Start();       }}
```

Dependency Injection: A design pattern that helps achieve loose coupling between components by injecting dependencies. Makes code more modular, testable, and maintainable.

Types of Lifecycles in C# DI

1. **Singleton**
2. **Scoped**
3. **Transient**

1. Singleton

- **Definition**: A single instance of the service is created and shared across the entire application lifetime.
- **Use Case**: For shared, heavy, or rarely changing objects (e.g., configuration settings, logging services).

Key Characteristics:

- Created once and reused everywhere.
- Same instance is injected into all consumers.

Example:

```
builder.Services.AddSingleton<IMyService, MyService>();
```

How It Works:

```
public class MyService : IMyService

{

	public Guid Id { get; } = Guid.NewGuid();

}



// Both calls will get the same instance

var service1 = app.Services.GetService<IMyService>();

var service2 = app.Services.GetService<IMyService>();

Console.WriteLine(service1.Id == service2.Id); // True
```

2. Scoped

- **Definition**: A new instance of the service is created for each **HTTP request** and shared within the request scope.
- **Use Case**: For services that need to maintain state or share data during the lifetime of a single request (e.g., database contexts).

Key Characteristics:

- Each HTTP request gets a new instance.
- Same instance is reused within the same request.

Example:

```
builder.Services.AddScoped<IMyService, MyService>();
```

How It Works:

```
public class MyService : IMyService
{
        public Guid Id { get; } = Guid.NewGuid();
}


// In the same HTTP request
var service1 = HttpContext.RequestServices.GetService<IMyService>();
var service2 = HttpContext.RequestServices.GetService<IMyService>();
Console.WriteLine(service1.Id == service2.Id); // True

// In a different HTTP request
var service3 = HttpContext.RequestServices.GetService<IMyService>();
Console.WriteLine(service1.Id == service3.Id); // False
```

3. Transient

- **Definition**: A new instance of the service is created **every time it is requested**.
- **Use Case**: For lightweight, stateless, or short-lived services (e.g., utility classes, calculation helpers).

Key Characteristics:

- A new instance is created for every request.
- Not shared between requests or consumers.

Example:

```
builder.Services.AddTransient<IMyService, MyService>();
```

How It Works:

```
public class MyService : IMyService

{

        public Guid Id { get; } = Guid.NewGuid();

}



// Each call gets a new instance

var service1 = app.Services.GetService<IMyService>();

var service2 = app.Services.GetService<IMyService>();

Console.WriteLine(service1.Id == service2.Id); // False
```

Lifecycle	Use Case	Example
Singleton	Shared, rarely changing services.	Configuration, caching, logging.
Scoped	Request-specific services that need to share data within the same HTTP request.	Entity Framework DbContext.
Transient	Stateless, lightweight services or those that should always have a fresh instance.	Utility classes like email senders, validators.

How to Register Lifecycles

In `Program.cs` (or `Startup.cs` in older versions):

```
builder.Services.AddSingleton<IMyService, MyService>(); // Singleton

builder.Services.AddScoped<IMyService, MyService>();     // Scoped

builder.Services.AddTransient<IMyService, MyService>(); // Transient
```

Lifecycle Visualization

Let’s visualize the difference using an example:

Service:

```
public class MyService : IMyService
{
    public Guid Id { get; } = Guid.NewGuid();
}
```

Controller:

```
[ApiController]
[Route("api/[controller]")]
public class TestController : ControllerBase
{
        private readonly IMyService _myService;
        public TestController(IMyService myService)
        {       _myService = myService;
        }
        [HttpGet]
        public IActionResult GetServiceId()
        {       return Ok(_myService.Id);
        }
}
```

Testing:

1. **Singleton**:
   - All requests to `/api/test` return the same `Id`.
2. **Scoped**:
   - Each HTTP request to `/api/test` gets a new `Id`, but the same `Id` is returned within the request.
3. **Transient**:
   - Every time the service is resolved (even within the same request), a new `Id` is generated.

Key Takeaways

1. **Singleton**:
   - Best for global, application-wide data or services.
   - Avoid using for services that depend on per-request state.
2. **Scoped**:
   - Ideal for web applications where the state needs to persist during a single request.
   - Most commonly used for `DbContext`.
3. **Transient**:
   - Best for lightweight, short-lived tasks or stateless operations.
   - Can lead to performance issues if the service is expensive to create.