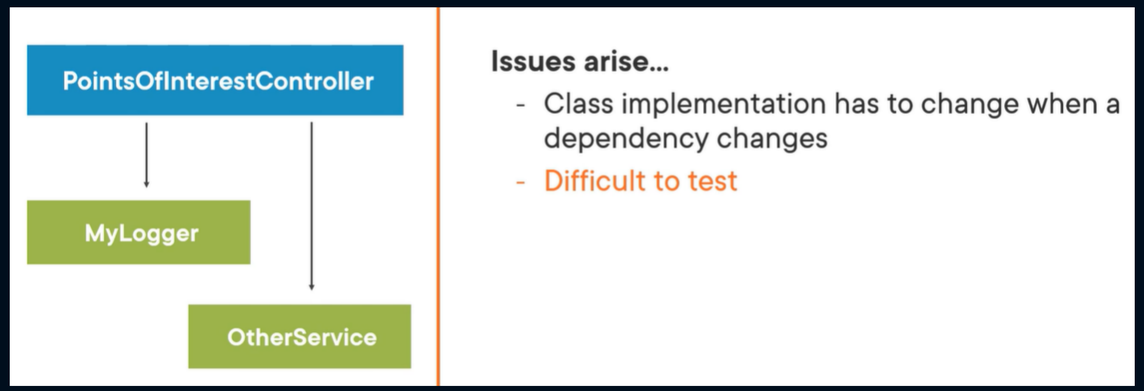
# Inversion of control

IoC delegates the function of selecting a concrete implementation type for a class dependencies to an external component.

DI is a specialization of IoC which uses an object (container) to initialize objects and provide the required dependencies to the object




Depending on concrete implementations

```
public class PointsOfInterestController :
Controller
{

    private
    ILogger<PointsOfInterestController>
    _logger;

    public PointsOfInterestController(
    ILogger<PointsOfInterestController>
    logger)
    {
        _logger = logger;
    }

...

}
```

◂ Interface, not concrete implementation

◂ Constructor injection

Dependency injection

# Dependency injection

**In the context of C#, dependency injection is commonly achieved using a dependency injection container or framework, such as Microsoft's built-in DI container or popular third-party libraries like Autofac or Ninject**

**Dependencies:** Dependencies are the objects or services that a class requires to perform its functionality.

**DI Container:** The dependency injection container is responsible for managing the creation and lifetime of objects and injecting their dependencies.

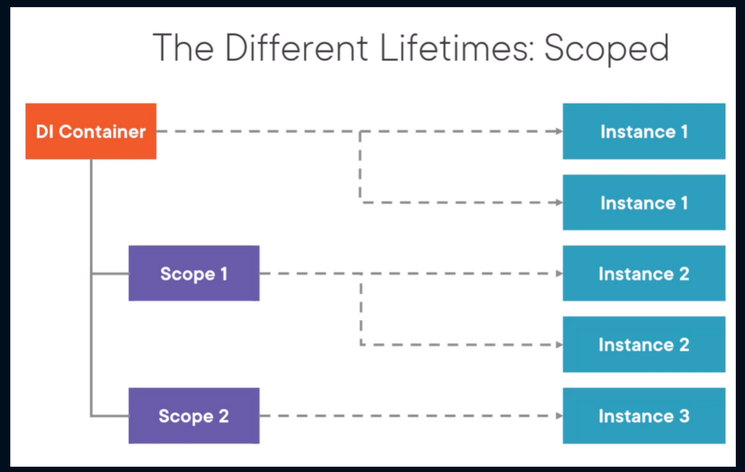
**Registration:** Dependencies need to be registered with the dependency injection container. This typically involves specifying the concrete types that should be instantiated when a certain interface or base class is requested

**Lifetime management:** Dependency injection containers provide options for managing the lifetime of objects. Common lifetime options include transient (new instance every time), singleton (single instance for the entire application), and scoped (single instance per request or unit of work).

**Types of injection:** Constructor Injection: Dependencies are injected through a class's constructor.
Property Injection: Dependencies are injected through public properties of a class.
Method Injection: Dependencies are injected through methods of a class.

# Lifetime of injected services


A new instance is created every time a type is requested

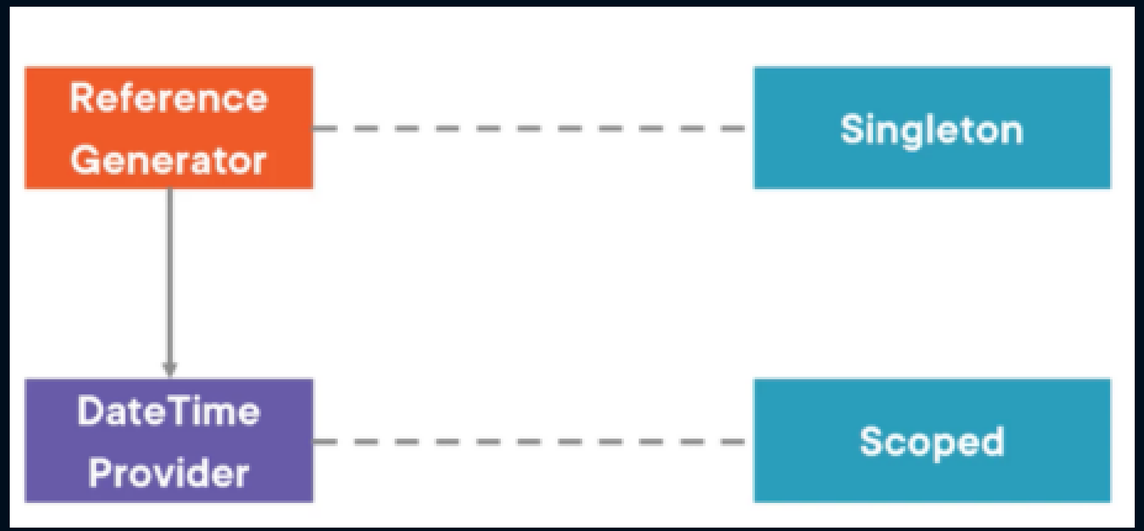



A new instance is created once and reused from then onwards

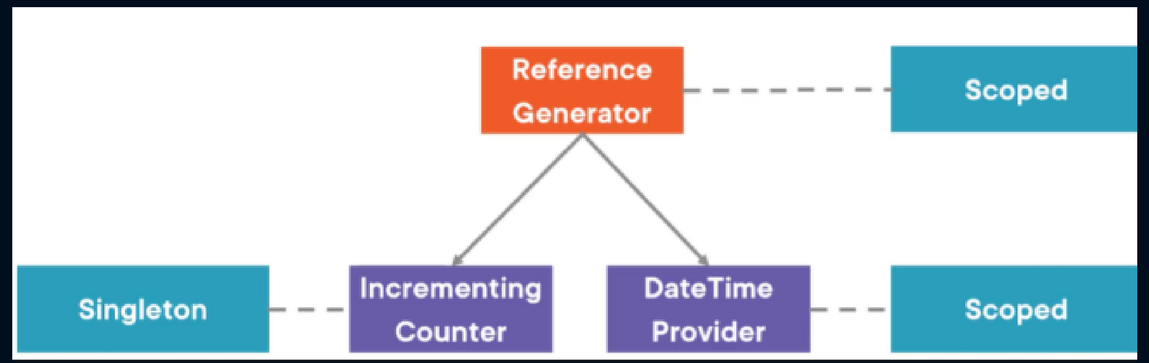

A new instance is created every time a type is requested

# Dependency captivity

Improper Lifetime Management: a cause of hanging state is improper management of the lifetime of injected dependencies. For example, if a long-lived singleton instance is injected into a short-lived scoped instance, it can lead to unexpected state issues.




Dependency captivity




Fixing dependency captivity

# Dependency injection and service locator

```
public CsvProductSource(
    IOptions<CsvProductSourceOptions> productSourceOptions,
    IPriceParser priceParser,
    IImportStatistics importStatistics)
{
    ...
}
```

Dependency Injection

**A class declares its dependencies, and they get provided somehow. A DI container is one way.**

## Dependency injection

A class declares its dependencies, ad they get provided somehow. A DI Container is one way

```
public CsvProductSource(IServiceProvider serviceProvider)
{
    _productSourceOptions = serviceProvider.getService<Ioptions<...>>;
    _priceParser = serviceProvider.getService<IPriceParser>;
    _importStatistics = serviceProvider.getService<IImportStatistics>;
}
```

Service Locator Using a DI Container

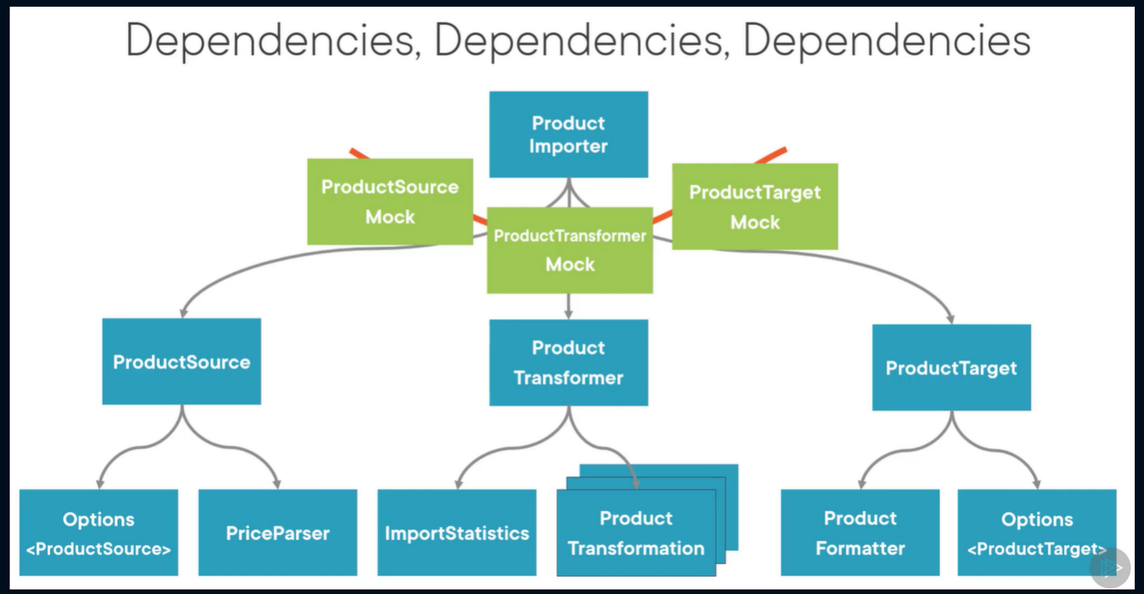
**You misuse the DI container as a Service Locator.**

## Service locator

A class references a central services repository and requests the services it needs from there

# Unit testing

Dependency injection plays a crucial role in unit testing, as it allows for better testability and isolation of dependencies. Dependency injection works hand in hand with mocking frameworks, which provide convenient ways to create and manage test doubles. These frameworks often integrate well with dependency injection containers and allow for seamless integration of test doubles into the codebase.




Mocking dependencies

```
[Theory]
[InlineData(0)]
0 references
public async Task WhenItReadsNProductsFromSource_ThenItWritesNProductsToTarget(int numberOfProducts)
{
    var productSource = new Mock<IProductSource>();
    var productTransformer = new Mock<IProductTransformer>();
    var productTarget = new Mock<IProductTarget>();
    var importStatistics = new Mock<IImportStatistics>();

    var subjectUnderTest = new ProductImporter(productSource.Object, productTransformer.Object,
        productTarget.Object, importStatistics.Object);

    var productcounter = 0;

    productSource
        .Setup(x => x.hasMoreProducts())
        .Callback(() => productcounter++)
        .Returns(() => productcounter <= numberOfProducts);
```

Example using MOQ

# DI considerations

Do implement IDisposable in your classes as needed

Don't register IDisposable types as transient, instead instantiate them using a factory

Don't resolve IDisposable types in the root container

Avoid (custom) scopes and scoped IDisposables when you can

Scoped and Singleton lifetimes will dispose the resources/ services used correctly.

For transient types, consider the case when more and more are created but disposed only much later when the container or scope is disposed.

Lifetimes are coupled to the service type, not the implementing type

```
services.AddSingleton<ImportStatistics>();
services.AddSingleton<IGetImportStatistics>((serviceProvider) =>
{
    return serviceProvider.GetRequiredService<ImportStatistics>();
});
services.AddSingleton<IWriteImportStatistics>((serviceProvider) =>
{
    return serviceProvider.GetRequiredService<ImportStatistics>();
});
```

Consider the previous example when you need to inject a implementing type to multiple services, and you need to maintain the same lifetime for all of them.

# Idempotency and safety

Safety indicates that it does not change the state of the system.
The convention has been established that GET and HEAD methods SHOULD NOT have the significance of taking an action other than retrieval. These methods ought to be considered 'safe'.


Non idempotent method (POST)


Safety requests

# Idempotency and safety

Idempotent methods have the property of 'idempotence' in that (aside from error or expiration issues) the side effects of N > O identical requests is the same as for a single request. The effect on the state of the system is the same whether the request is handled once or many times.

| Method | Idempotent? | Safe? |
|---|---|---|
| GET | Yes | Yes |
| POST | No | No |
| PUT | Yes | No |
| DELETE | Yes | No |
| PATCH | Yes | No |
| HEAD | Yes | Yes |

Safety and Idempotency

# Demo

**We will check the following:**

Lifetime of injected services

Extension methods for injections

You can check the following Repository for some examples:

C# fundamentals

You can check the following Repository(ies) for some examples:

ASP NET Core 6 Web API

# Homework

**Exercise/ Homework**

This is a continuation of the previous exercise.

Modularize your application by using interfaces and services, and follow an Domain Centric Architecture

Register your services and use Dependency injection