MCSE 541:Web Computing and Mining

ASP.NET Core-Dependency Injection

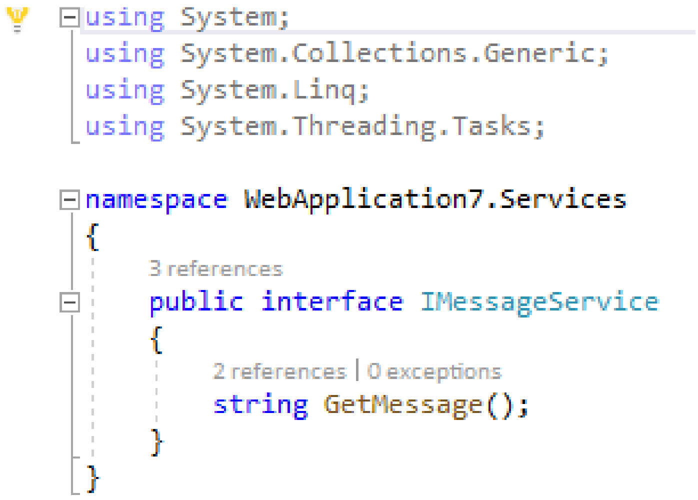
Prof. Dr. Shamim Akhter
Stamford University Bangladesh

Dependency Injection

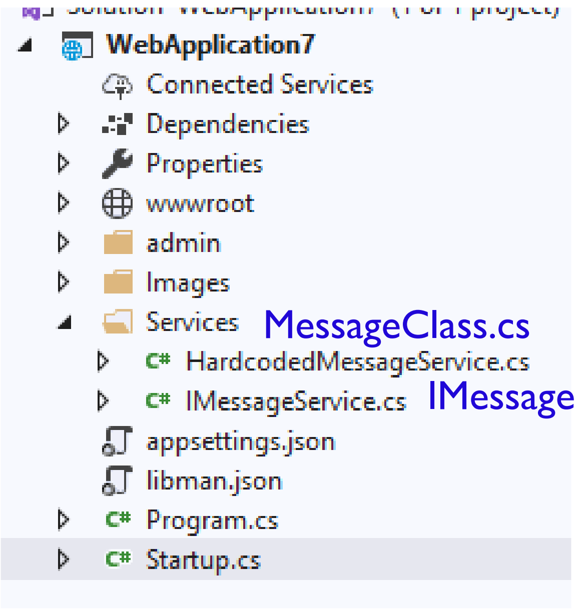
- Dependency injection (also known as DI) is a design pattern
  - in which an object does not create it dependent classes, but asks for it.
- ASP.NET Cores Dependency injection framework
  - Uses DI Container or IoC Container to implement that design pattern (create the object)

Adding an Interface and a New class into Services folder

- Add one interface @ Service folder



```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApplication7.Services
{
    3 references
    public interface IMessageService
    {
        2 references | 0 exceptions
        string GetMessage();
    }
}
```

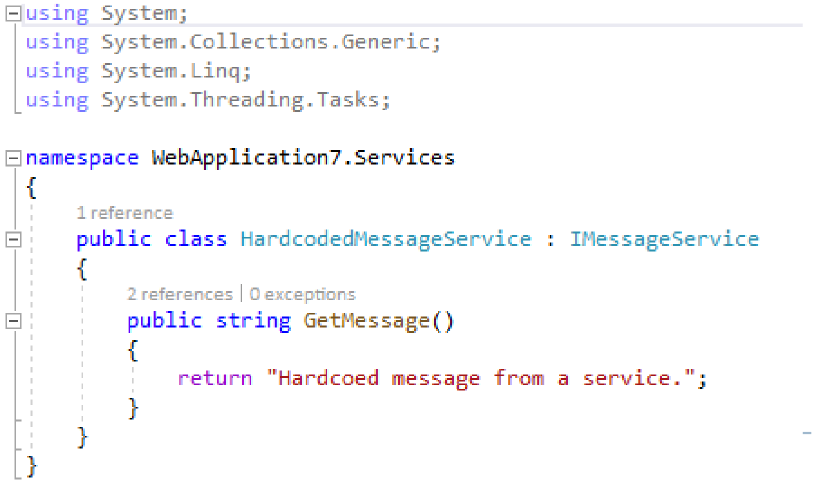


```
Solution 'WebApplication7' (1 of 1 project)
◢ WebApplication7
      Connected Services
   ▷ Dependencies
   ▷ Properties
   ▷ wwwroot
   ▷ admin
   ▷ Images
   ◢ Services
      ▷ C# HardcodedMessageService.cs
      ▷ C# IMessageService.cs
      appsettings.json
      libman.json
   ▷ C# Program.cs
   ▷ C# Startup.cs
```

MessageClass.cs

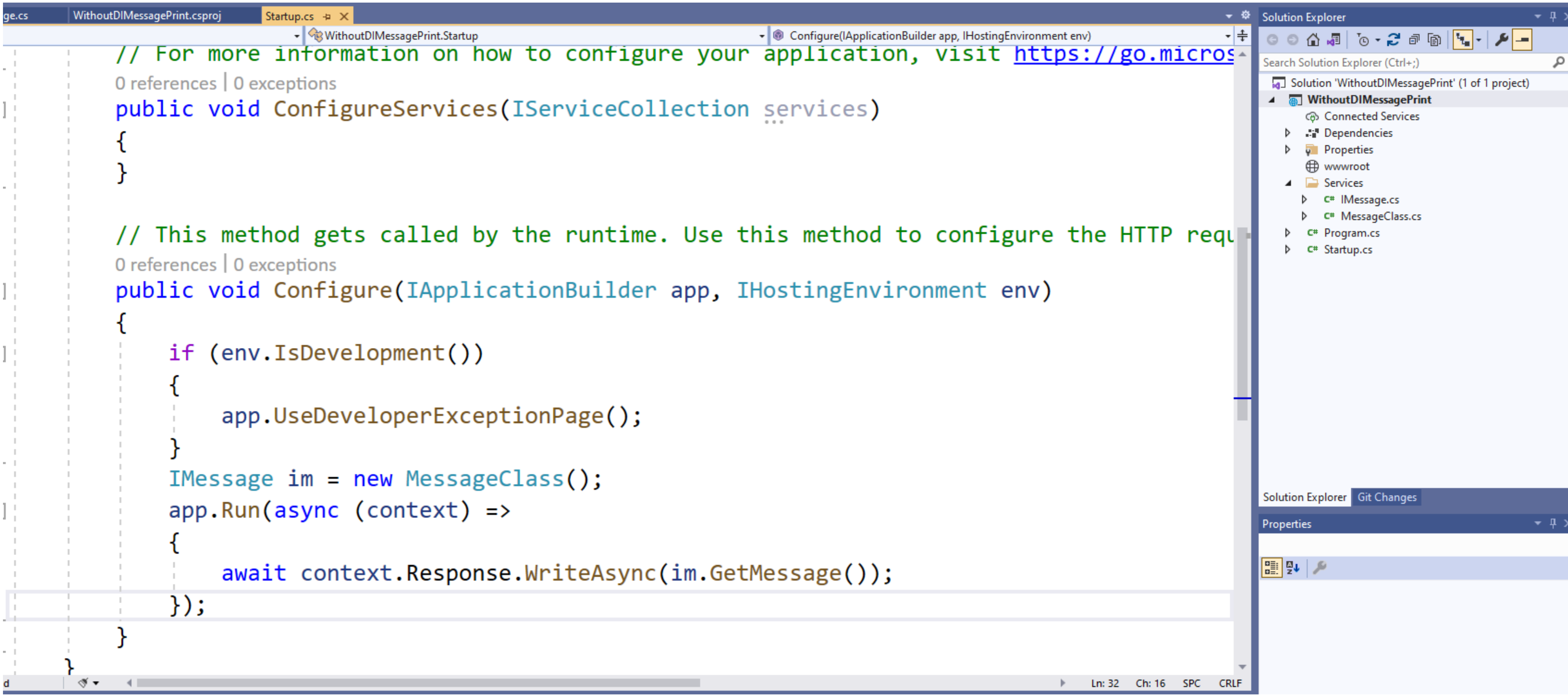
IMessage.cs

- Add one class @HardcodedMessageService



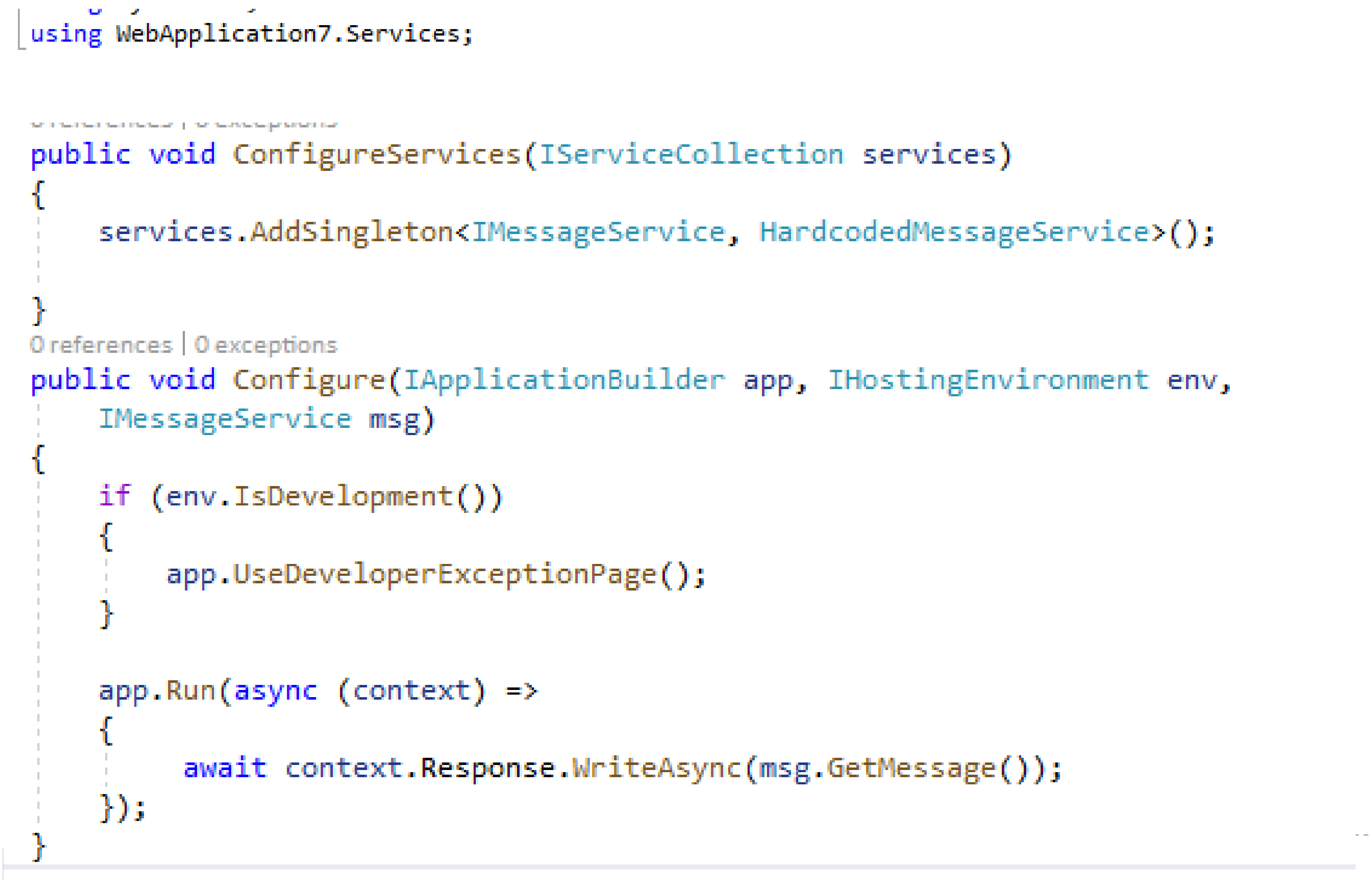
```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Threading.Tasks;

namespace WebApplication7.Services
{
    1 reference
    public class HardcodedMessageService : IMessageService
    {
        2 references | 0 exceptions
        public string GetMessage()
        {
            return "Hardcoed message from a service.";
        }
    }
}
```

Without Dependency Injection



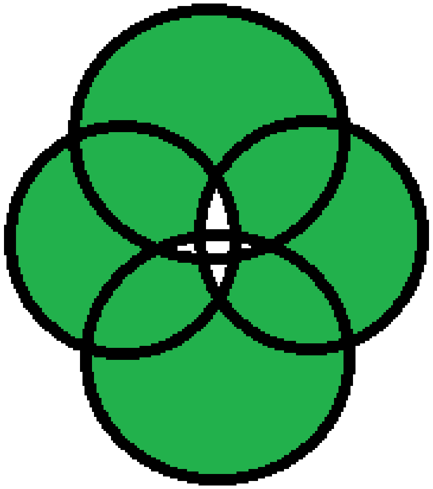
```
        // For more information on how to configure your application, visit https://go.micros
        0 references | 0 exceptions
        public void ConfigureServices(IServiceCollection services)
        {
        }

        // This method gets called by the runtime. Use this method to configure the HTTP requ
        0 references | 0 exceptions
        public void Configure(IApplicationBuilder app, IHostingEnvironment env)
        {
            if (env.IsDevelopment())
            {
                app.UseDeveloperExceptionPage();
            }
            IMessage im = new MessageClass();
            app.Run(async (context) =>
            {
                await context.Response.WriteAsync(im.GetMessage());
            });
        }
    }
```

Configure A Service (With Dependency Injection)



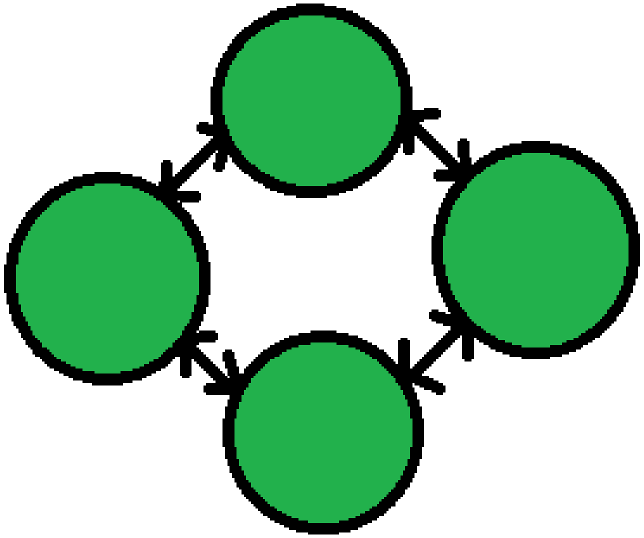
```
using WebApplication7.Services;


public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IMessageService, HardcodedMessageService>();

}
0 references | 0 exceptions
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    IMessageService msg)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
         await context.Response.WriteAsync(msg.GetMessage());
    });
}
```

Tightly Coupling vs Loose Coupling



Tight coupling:

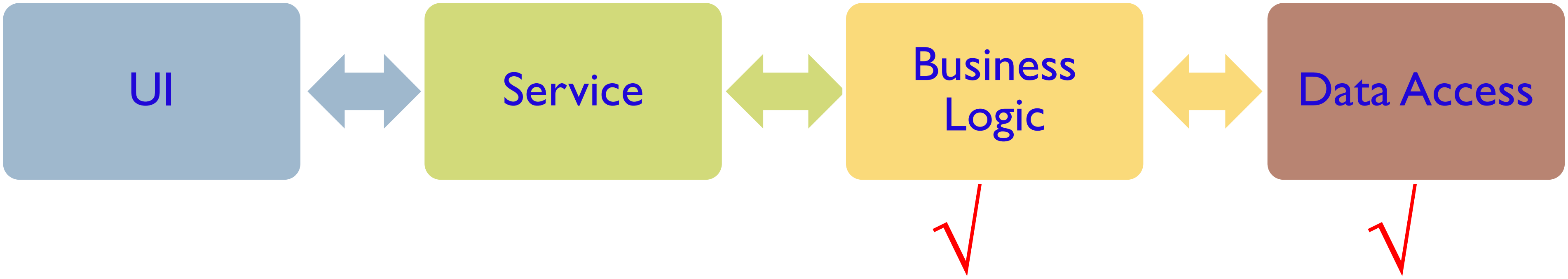
1. More Interdependency
2. More coordination
3. More information flow



Loose coupling:

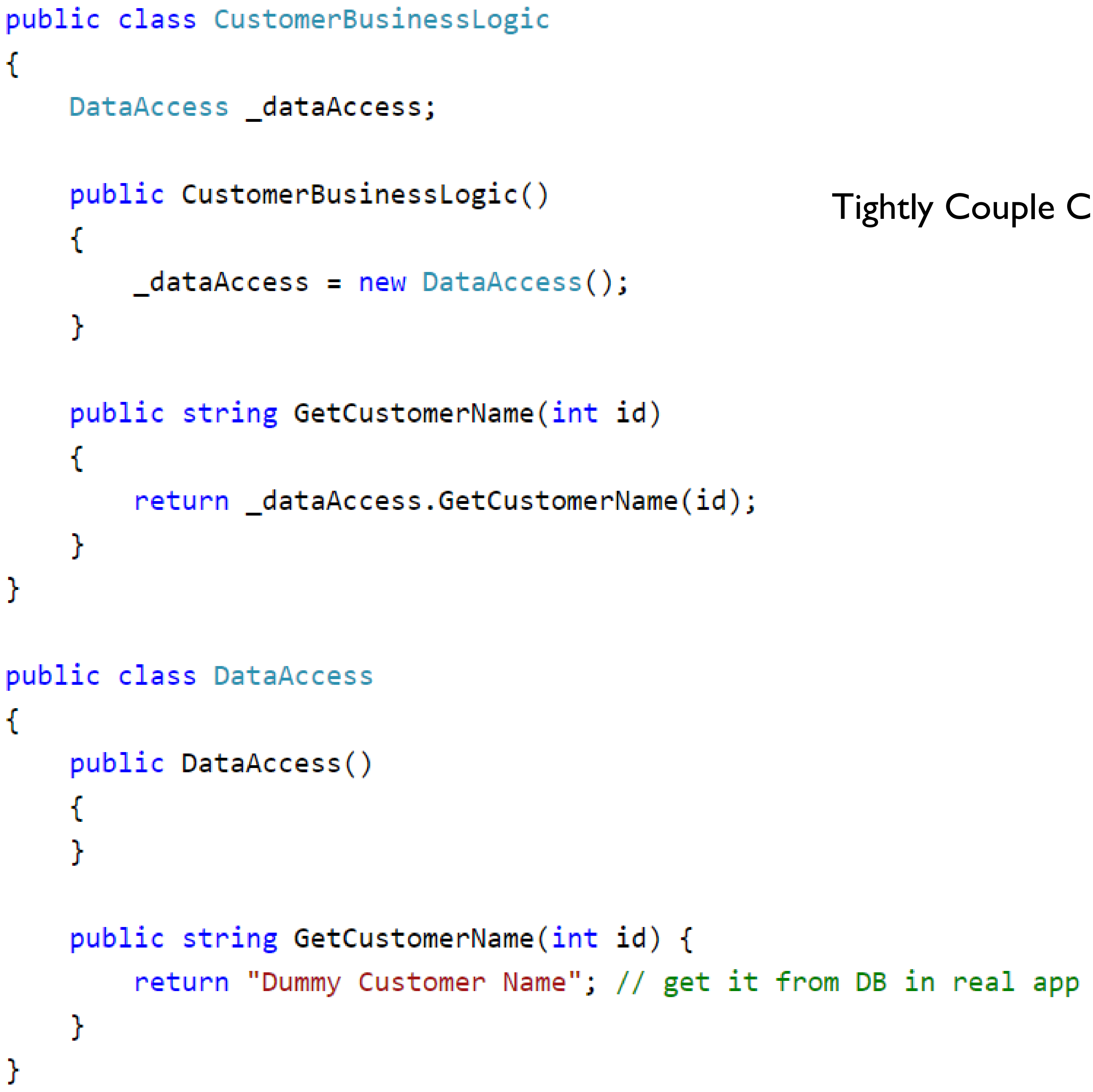
1. Less Interdependency
2. Less coordination
3. Less information flow

N-Tier Architecture



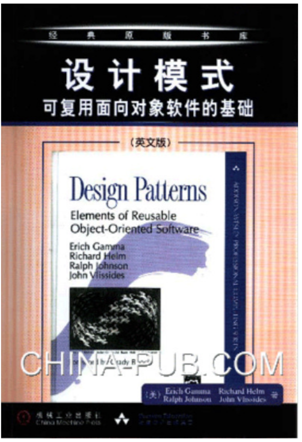
In the typical n-tier architecture, the User Interface (UI) uses Service layer to retrieve or save data. The Service layer uses the `BusinessLogic` class to apply business rules on the data. The `BusinessLogic` class depends on the `DataAccess` class which retrieves or saves the data to the underlying database. This is simple n-tier architecture design. Let's focus on the `BusinessLogic` and `DataAccess` classes to understand IoC.

Tightly Couple Classes

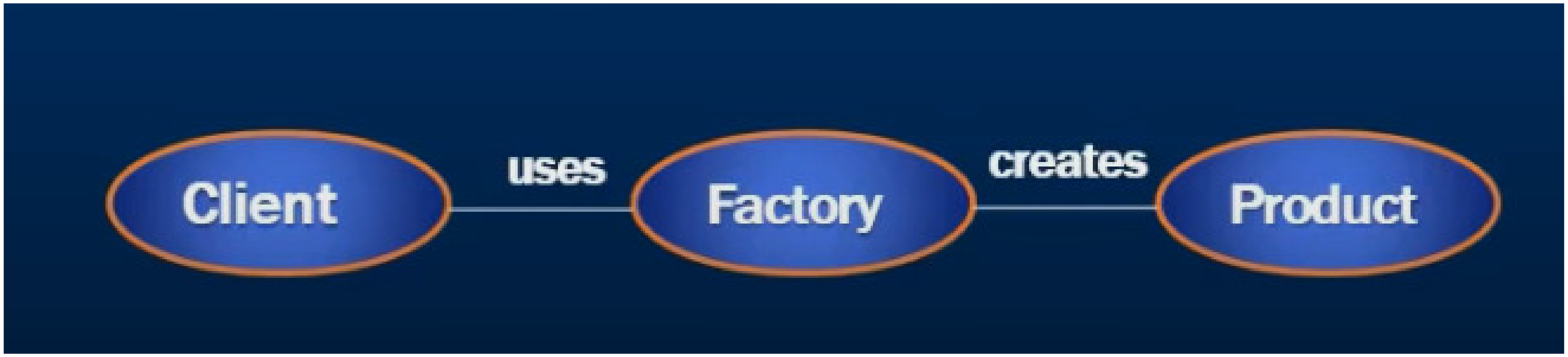


```
public class CustomerBusinessLogic
{
    DataAccess _dataAccess;

    public CustomerBusinessLogic()
    {
        _dataAccess = new DataAccess();
    }

    public string GetCustomerName(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public class DataAccess
{
    public DataAccess()
    {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name"; // get it from DB in real app
    }
}
```

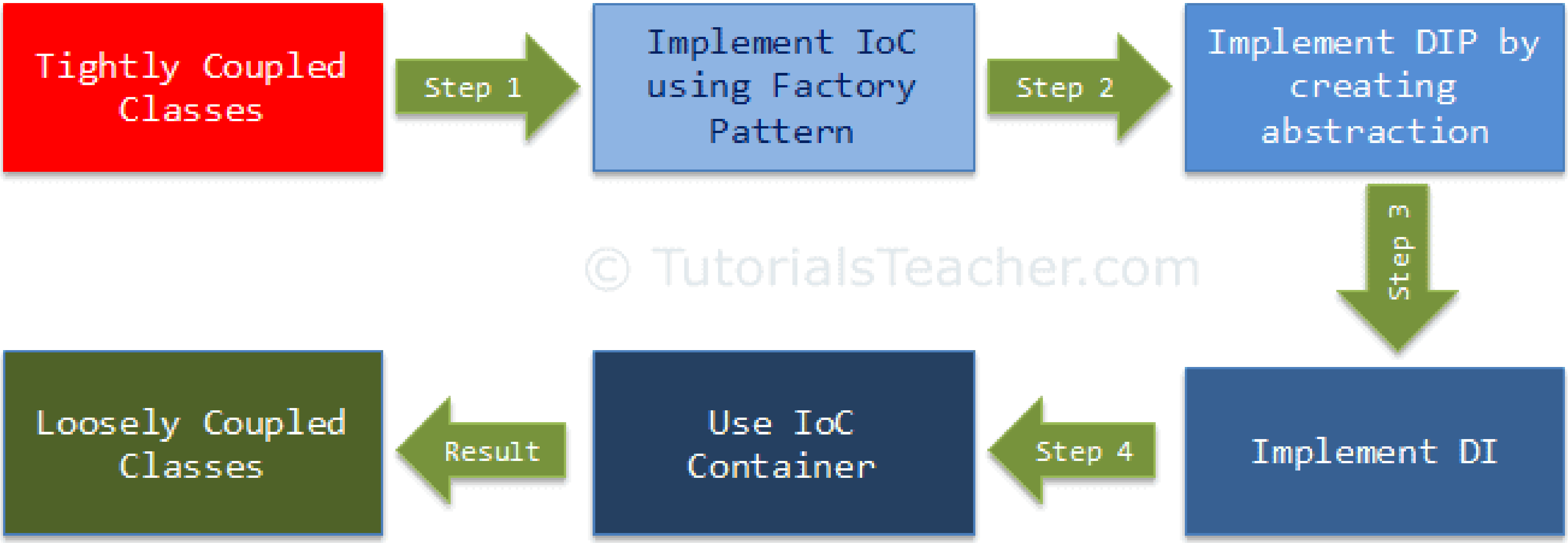
Factory Design Pattern



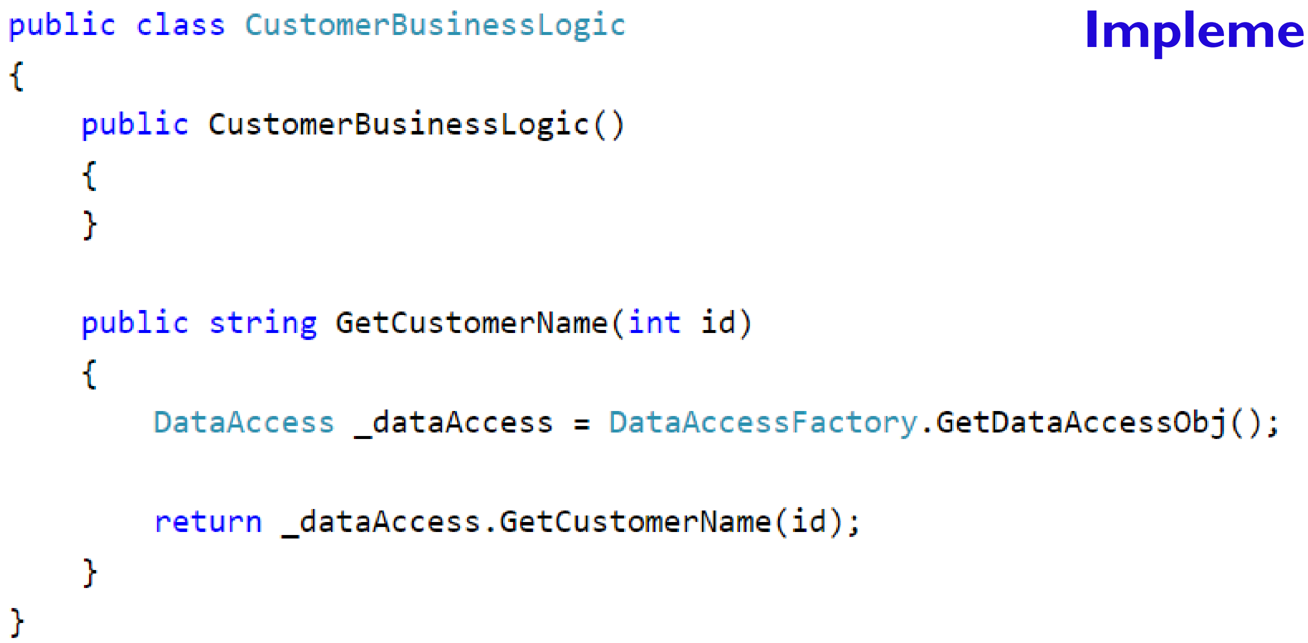
- According to Gang of Four
  - Factory Design Pattern states that **“A factory is an object which is used for creating other objects”**.
  - Factory Design pattern,
    - we create an object without exposing the creation logic to the client
    - and the client will refer to the newly created object using a common interface.



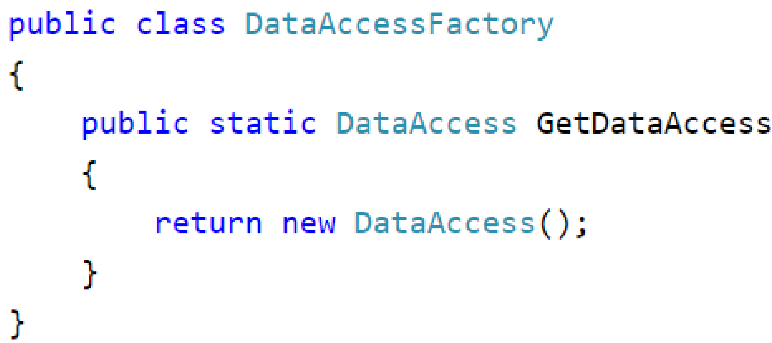
- The following figure illustrates how we are going to achieve loosely coupled design step by step.



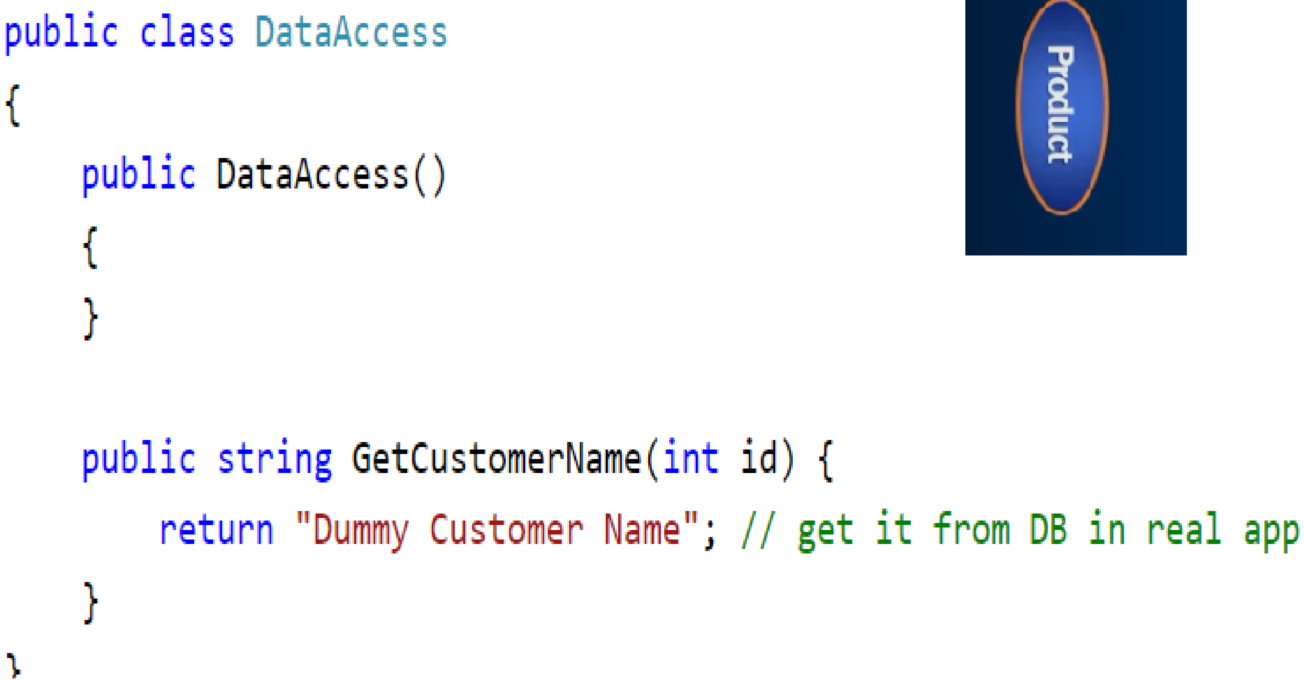
Implementing IoC using factory pattern



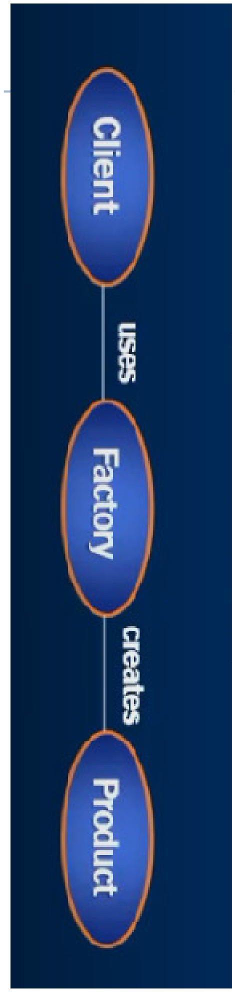
```
public class CustomerBusinessLogic
{
    public CustomerBusinessLogic()
    {
    }

    public string GetCustomerName(int id)
    {
        DataAccess _dataAccess = DataAccessFactory.GetDataAccessObj();

        return _dataAccess.GetCustomerName(id);
    }
}
```



```
public class DataAccessFactory
{
    public static DataAccess GetDataAccessObj()
    {
        return new DataAccess();
    }
}
```

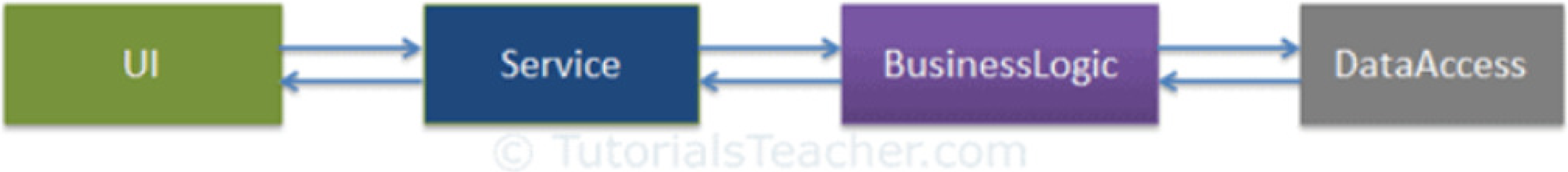


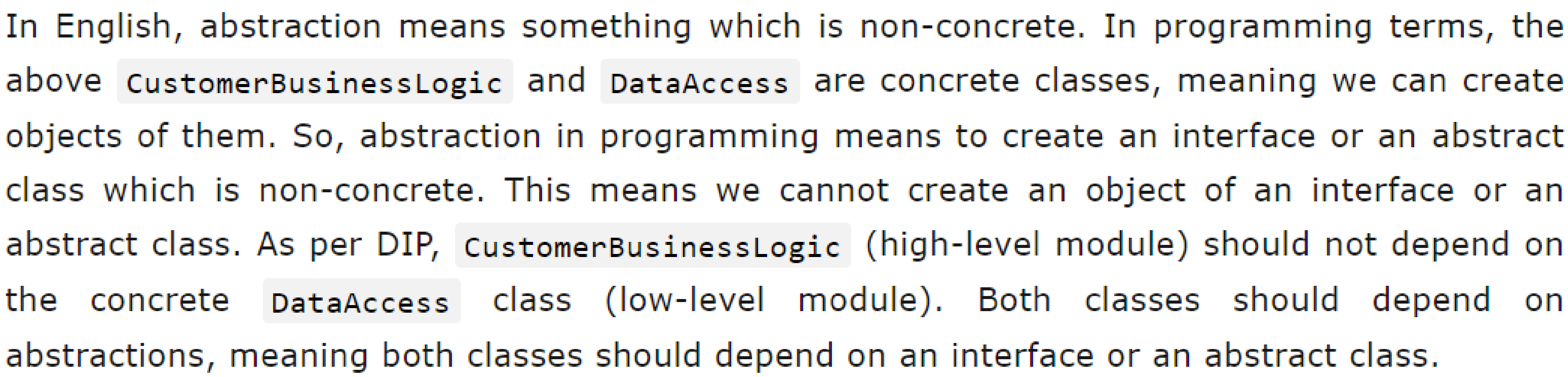
```
public class DataAccess
{
    public DataAccess()
    {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name"; // get it from DB in real app
    }
}
```



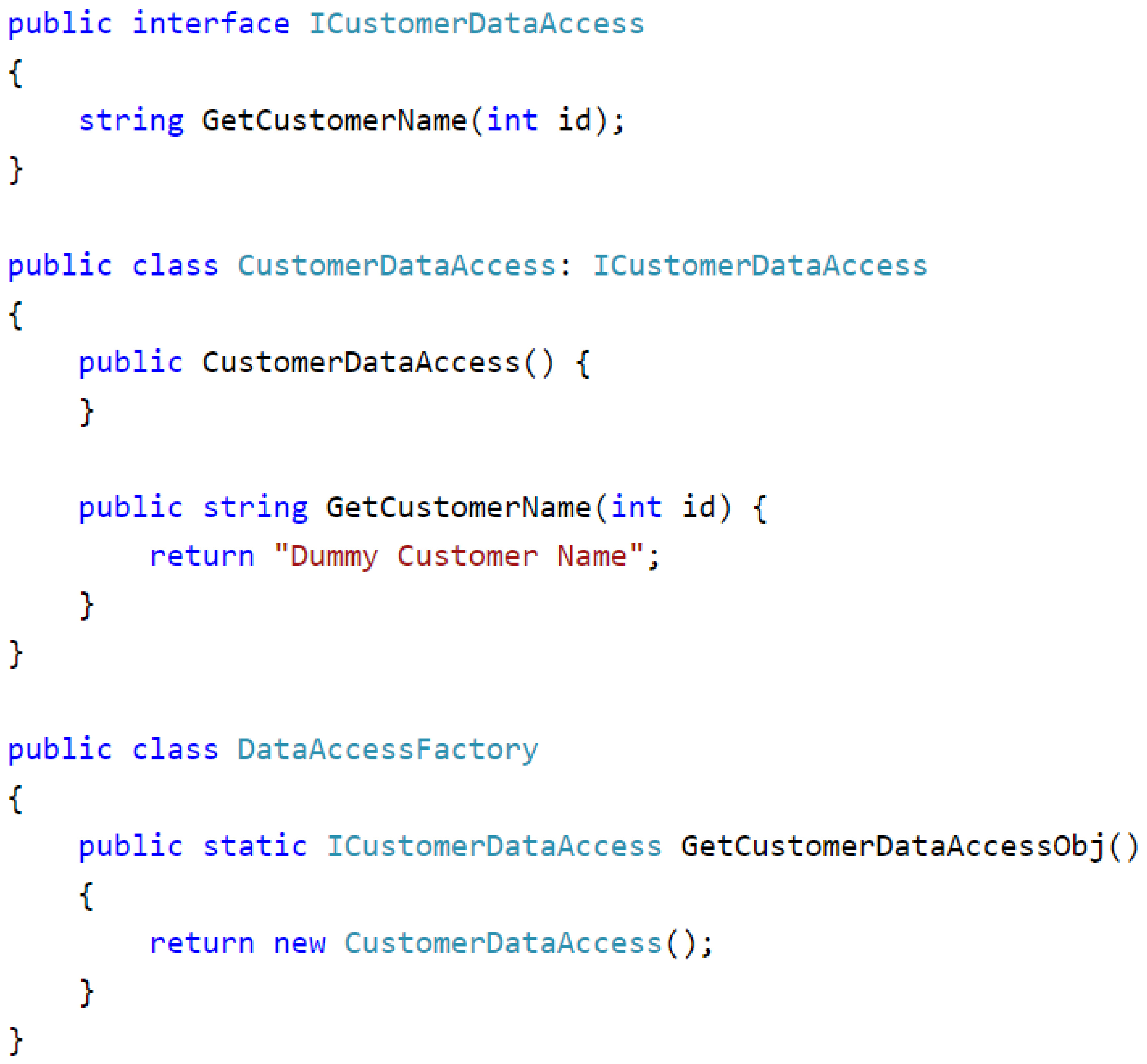
Dependency Inversion Principle

1. High-level modules should not depend on low-level modules. Both should depend on the abstraction.
2. Abstractions should not depend on details. Details should depend on abstractions.



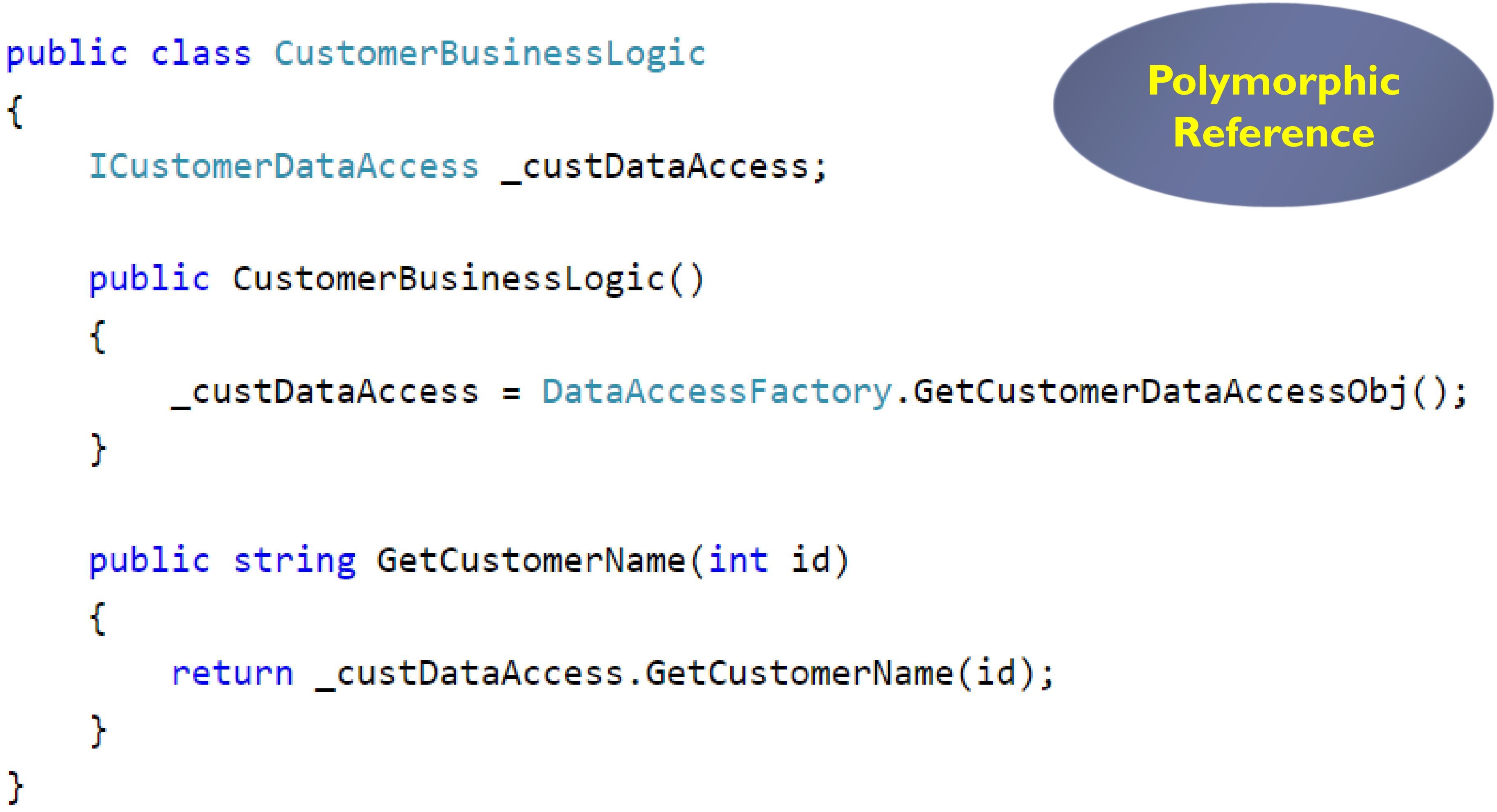

In English, abstraction means something which is non-concrete. In programming terms, the above `CustomerBusinessLogic` and `DataAccess` are concrete classes, meaning we can create objects of them. So, abstraction in programming means to create an interface or an abstract class which is non-concrete. This means we cannot create an object of an interface or an abstract class. As per DIP, `CustomerBusinessLogic` (high-level module) should not depend on the concrete `DataAccess` class (low-level module). Both classes should depend on abstractions, meaning both classes should depend on an interface or an abstract class.

DIP Continue



```
public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}

public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess() {
    }

    public string GetCustomerName(int id) {
        return "Dummy Customer Name";
    }
}

public class DataAccessFactory
{
    public static ICustomerDataAccess GetCustomerDataAccessObj()
    {
        return new CustomerDataAccess();
    }
}
```

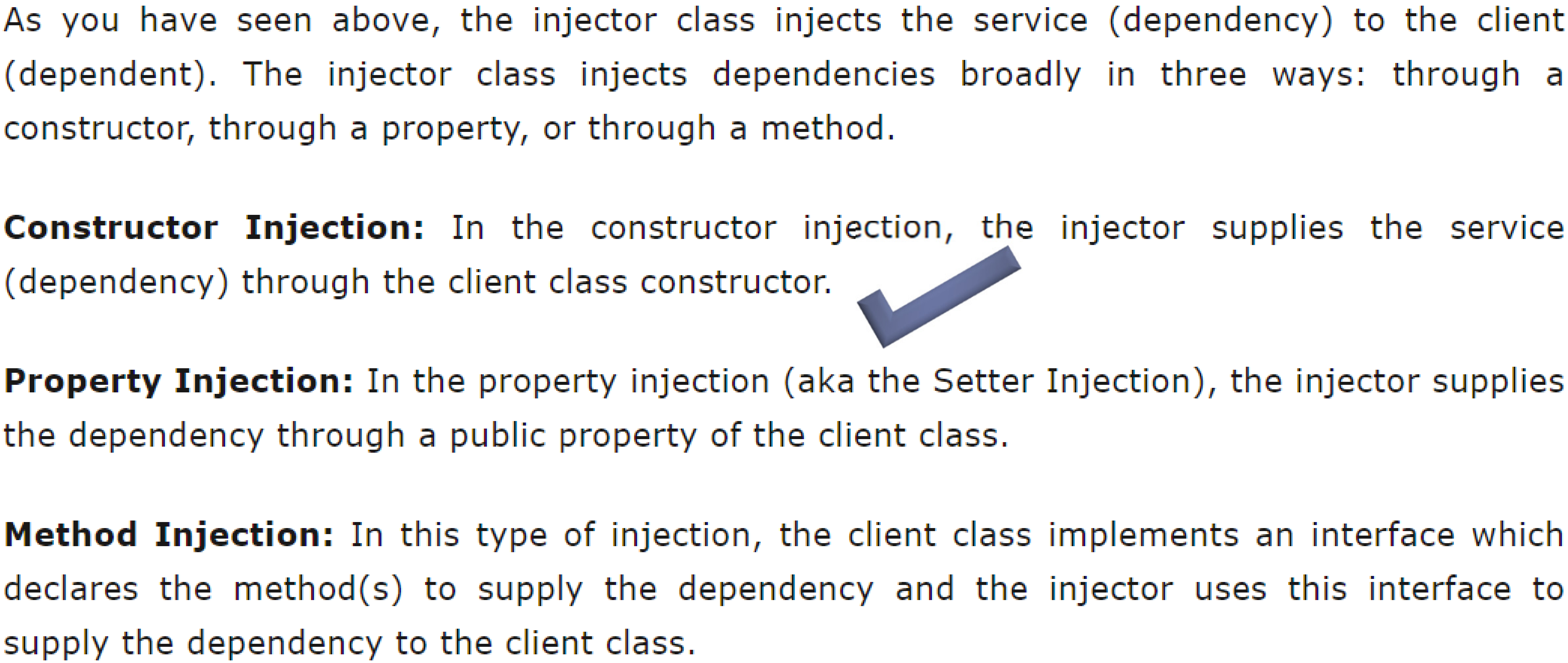
DIP Continue



```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

    public CustomerBusinessLogic()
    {
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

    public string GetCustomerName(int id)
    {
        return _custDataAccess.GetCustomerName(id);
    }
}
```

Polymorphic Reference

Types of Dependency Injection

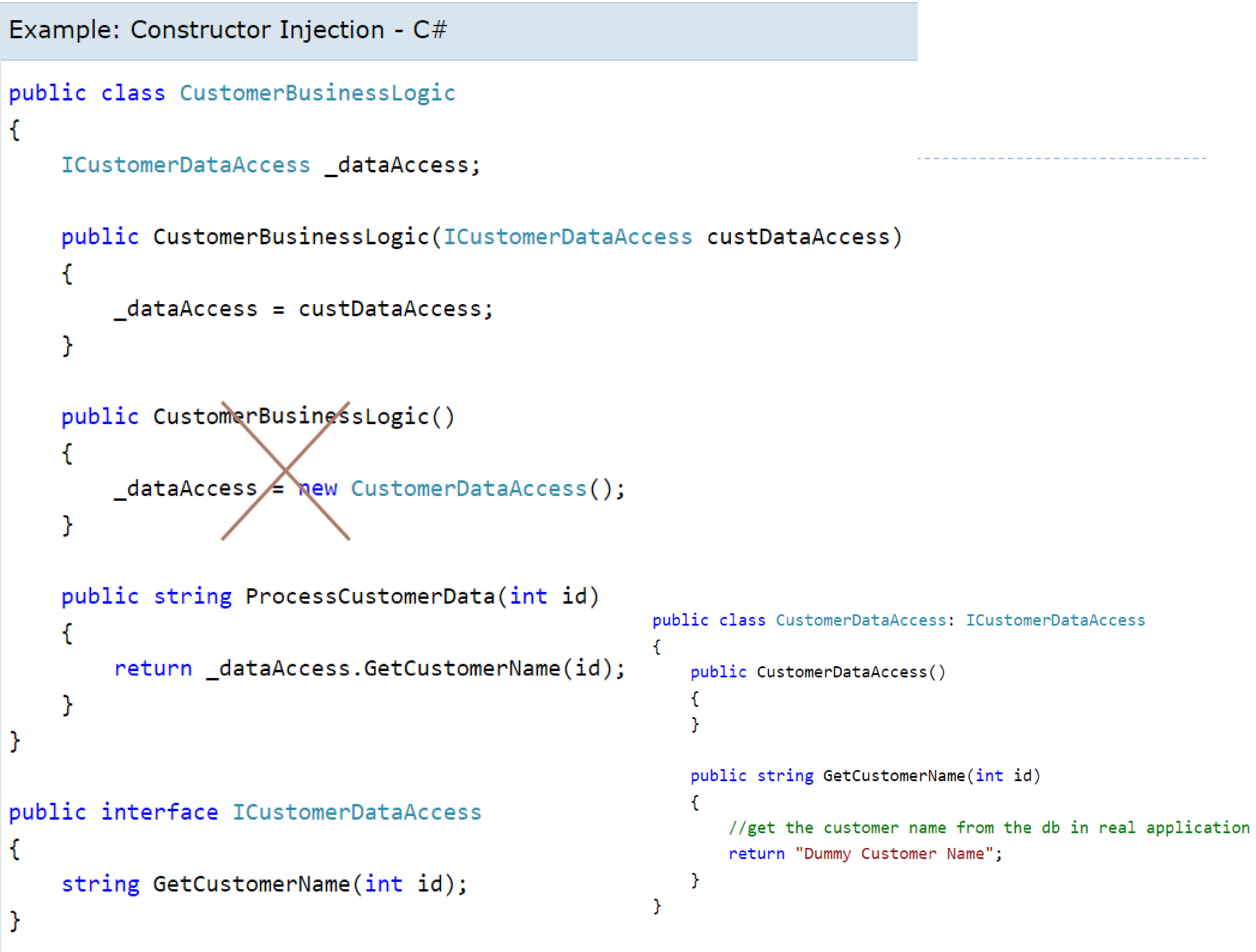


As you have seen above, the injector class injects the service (dependency) to the client (dependent). The injector class injects dependencies broadly in three ways: through a constructor, through a property, or through a method.

Constructor Injection: In the constructor injection, the injector supplies the service (dependency) through the client class constructor.

Property Injection: In the property injection (aka the Setter Injection), the injector supplies the dependency through a public property of the client class.

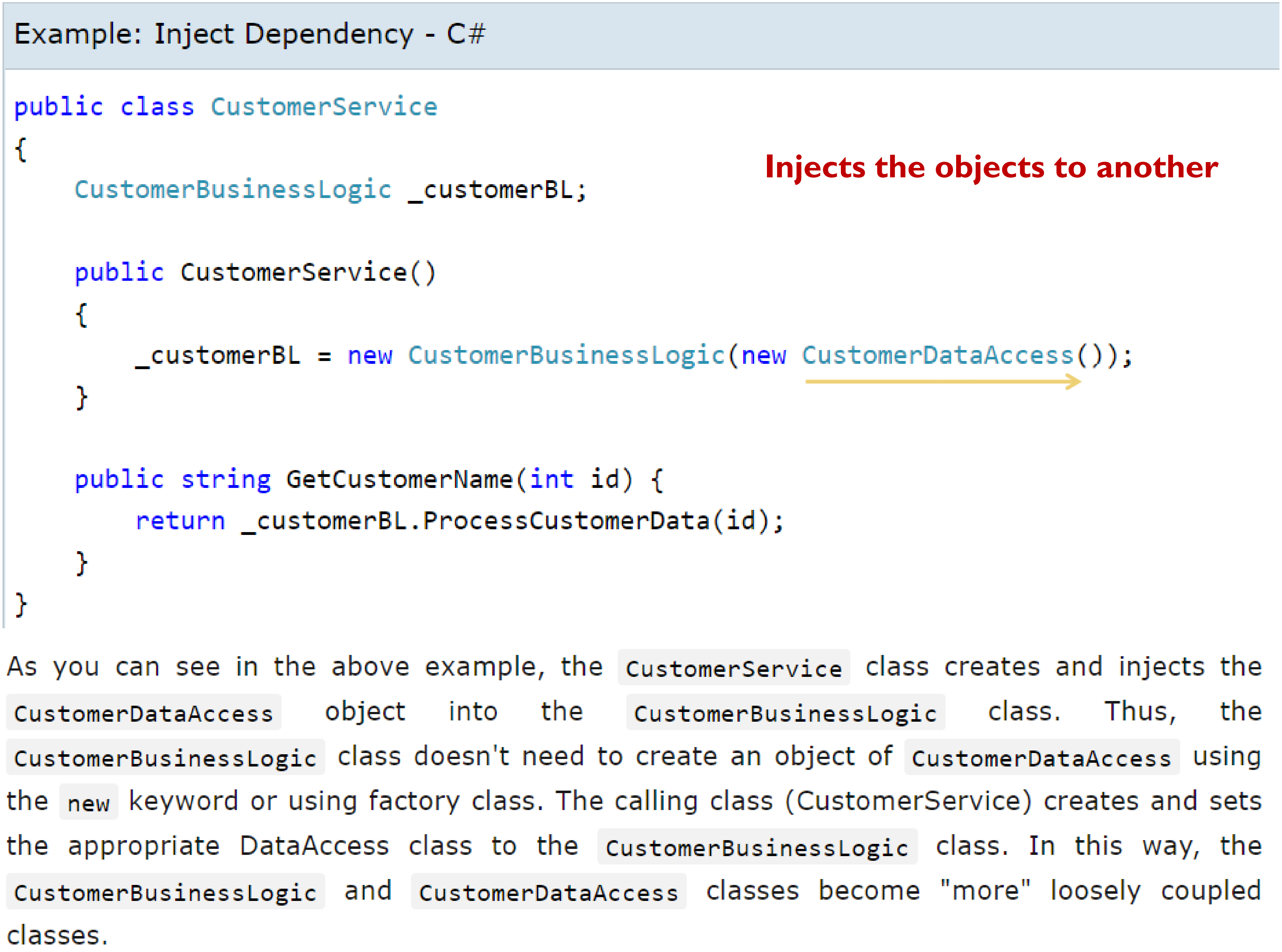
Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.



Example: Constructor Injection - C#

```
public class CustomerBusinessLogic
{
    ICustomerDataAccess _dataAccess;

    public CustomerBusinessLogic(ICustomerDataAccess custDataAccess)
    {
        _dataAccess = custDataAccess;
    }

    ~~public CustomerBusinessLogic()
    {
        _dataAccess = new CustomerDataAccess();
    }~~

    public string ProcessCustomerData(int id)
    {
        return _dataAccess.GetCustomerName(id);
    }
}

public interface ICustomerDataAccess
{
    string GetCustomerName(int id);
}
```

```
public class CustomerDataAccess: ICustomerDataAccess
{
    public CustomerDataAccess()
    {
    }

    public string GetCustomerName(int id)
    {
        //get the customer name from the db in real application
        return "Dummy Customer Name";
    }
}
```



Example: Inject Dependency - C#

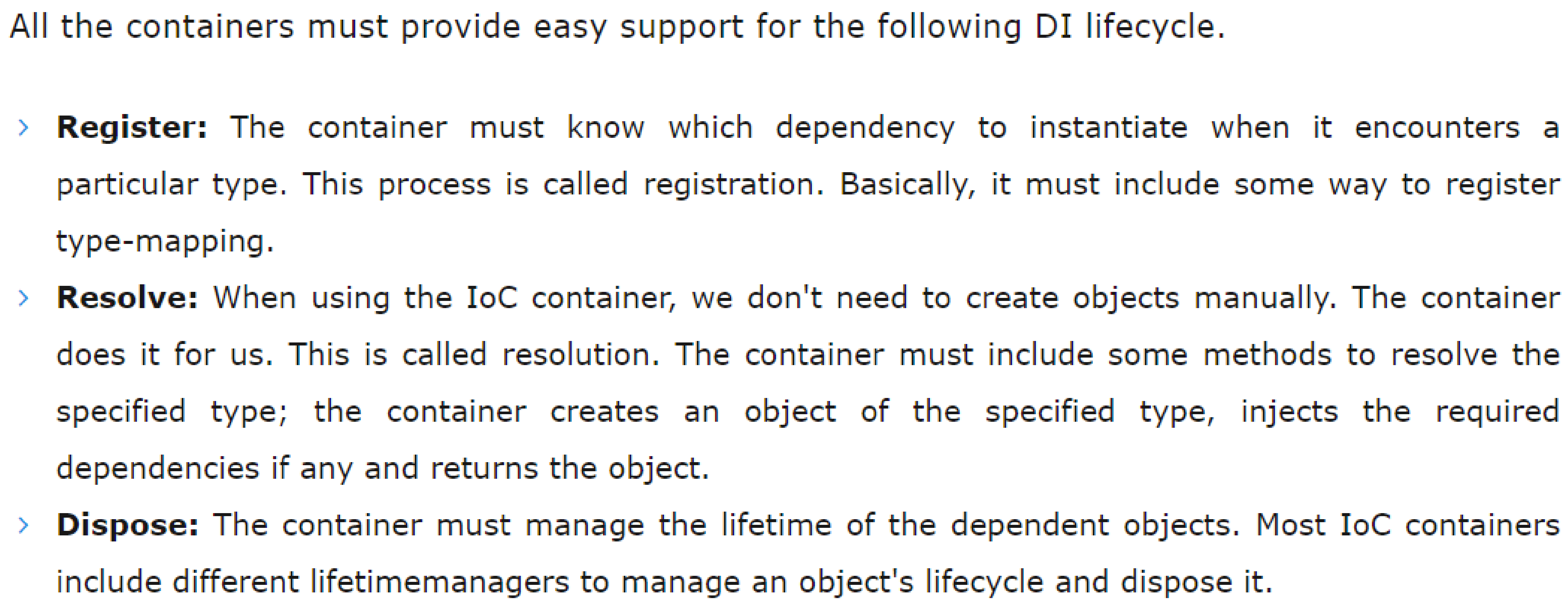
```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

    public CustomerService()
    {
        _customerBL = new CustomerBusinessLogic(new CustomerDataAccess());
    }

    public string GetCustomerName(int id) {
        return _customerBL.ProcessCustomerData(id);
    }
}
```

Injects the objects to another

As you can see in the above example, the `CustomerService` class creates and injects the `CustomerDataAccess` object into the `CustomerBusinessLogic` class. Thus, the `CustomerBusinessLogic` class doesn't need to create an object of `CustomerDataAccess` using the `new` keyword or using factory class. The calling class (CustomerService) creates and sets the appropriate DataAccess class to the `CustomerBusinessLogic` class. In this way, the `CustomerBusinessLogic` and `CustomerDataAccess` classes become "more" loosely coupled classes.

DI Container or IoC container

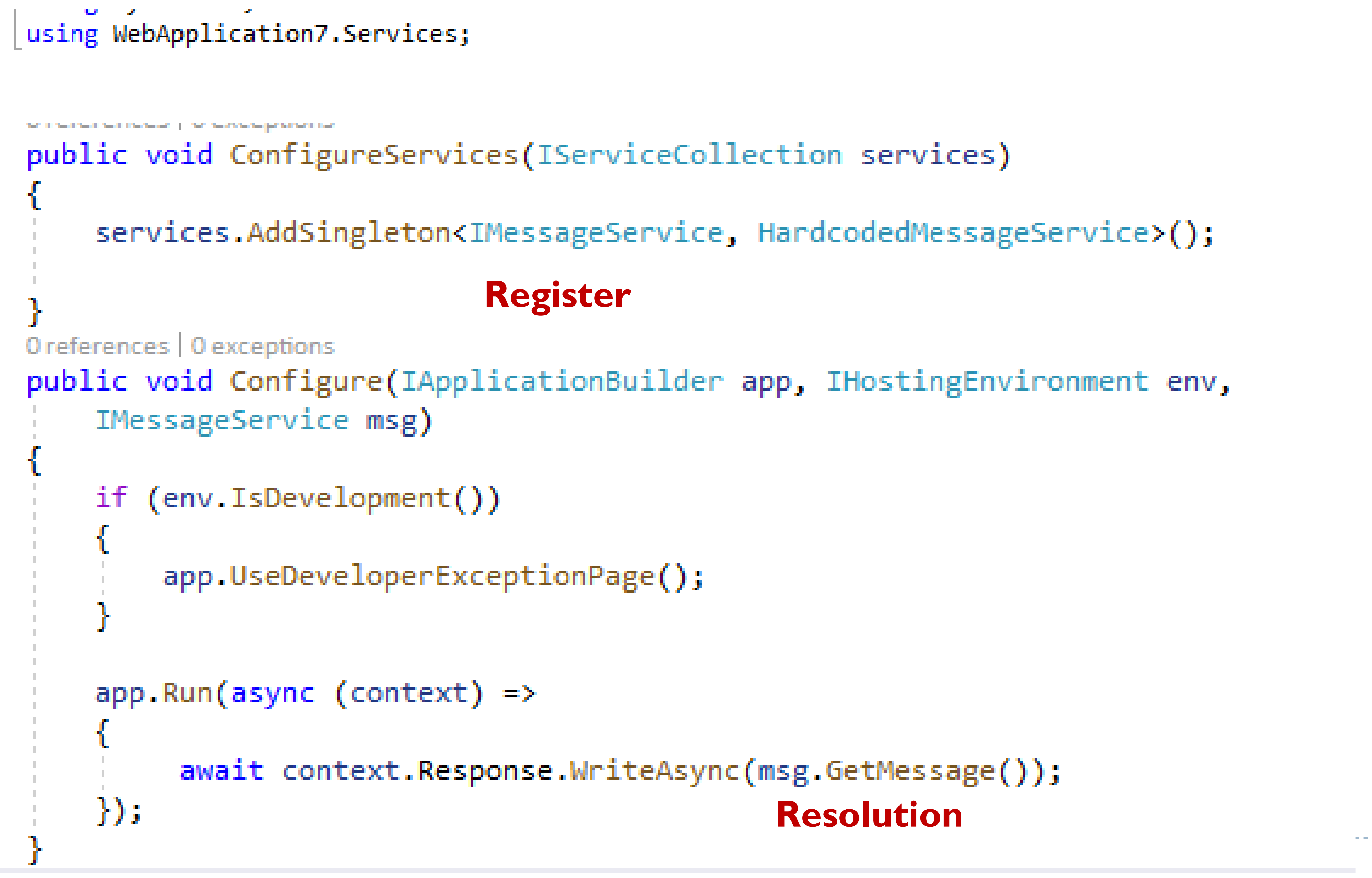
- IoC Container is a framework for implementing automatic dependency injection.
- It manages object creation and it's life-time, and also injects dependencies to the class.



All the containers must provide easy support for the following DI lifecycle.

- **Register:** The container must know which dependency to instantiate when it encounters a particular type. This process is called registration. Basically, it must include some way to register type-mapping.
- **Resolve:** When using the IoC container, we don't need to create objects manually. The container does it for us. This is called resolution. The container must include some methods to resolve the specified type; the container creates an object of the specified type, injects the required dependencies if any and returns the object.
- **Dispose:** The container must manage the lifetime of the dependent objects. Most IoC containers include different lifetimemanagers to manage an object's lifecycle and dispose it.

Configure A Service (With Dependency Injection)



```
using WebApplication7.Services;

public void ConfigureServices(IServiceCollection services)
{
    services.AddSingleton<IMessageService, HardcodedMessageService>();

}
0 references | 0 exceptions
public void Configure(IApplicationBuilder app, IHostingEnvironment env,
    IMessageService msg)
{
    if (env.IsDevelopment())
    {
        app.UseDeveloperExceptionPage();
    }

    app.Run(async (context) =>
    {
         await context.Response.WriteAsync(msg.GetMessage());
    });
}
```

Register

Resolution

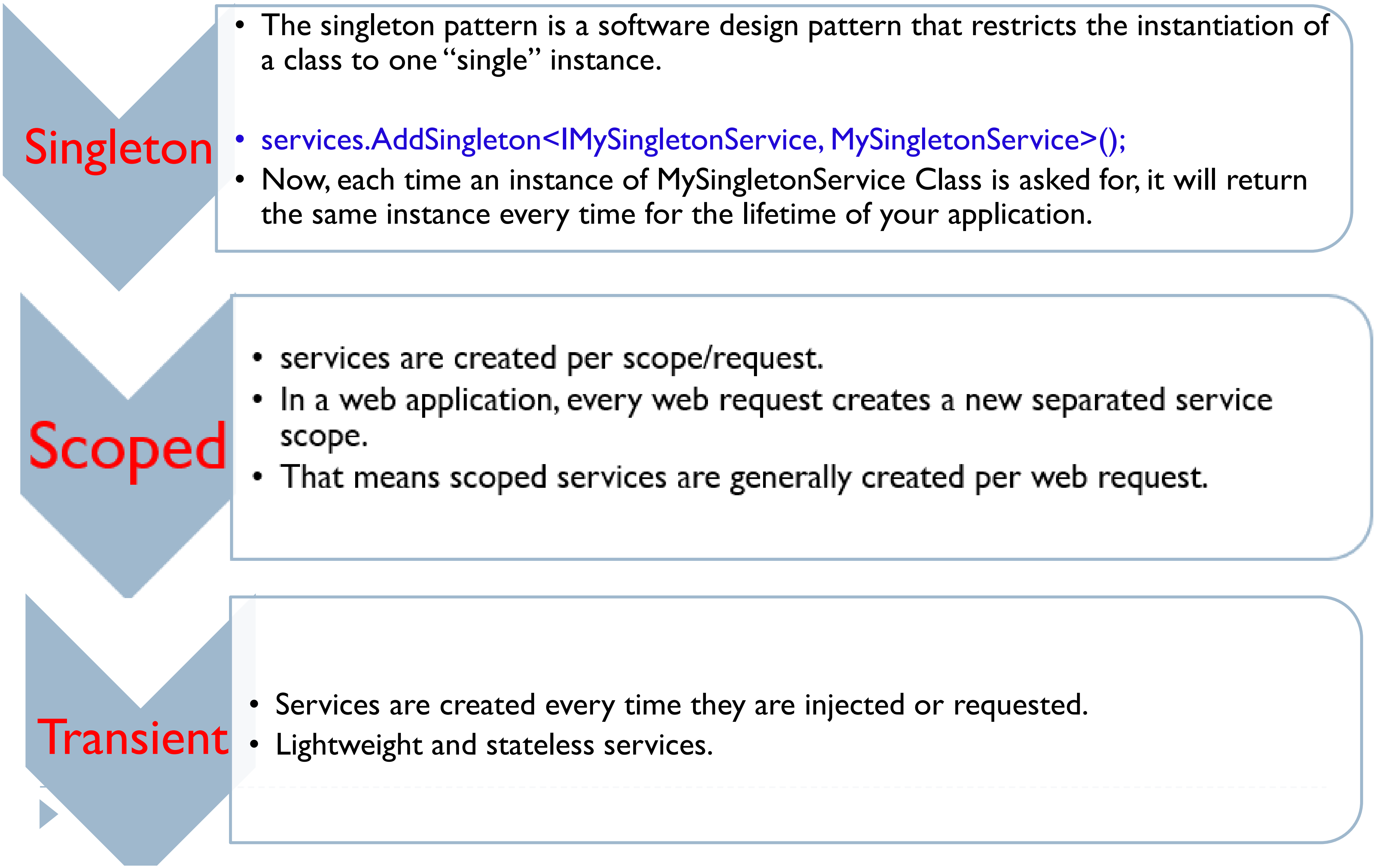
Managing the Service Lifetime

- DI Container must need to decide the life time of the services.
- Service lifetime will be decided during the service registration process.

Most IOC containers allow a “lifetime” to be applied when wiring up dependencies. With the ASP.NET Core Dependency Injection framework, the following life cycles are available.

•Transient – Each time a transient object is requested, a new instance will be created
•Scoped – The same object will be used when requested within the same request
•Singleton – The same object will always be used across all requests

Service Types



Singleton

- The singleton pattern is a software design pattern that restricts the instantiation of a class to one “single” instance.

- services.AddSingleton<IMySingletonService, MySingletonService>();
- Now, each time an instance of MySingletonService Class is asked for, it will return the same instance every time for the lifetime of your application.

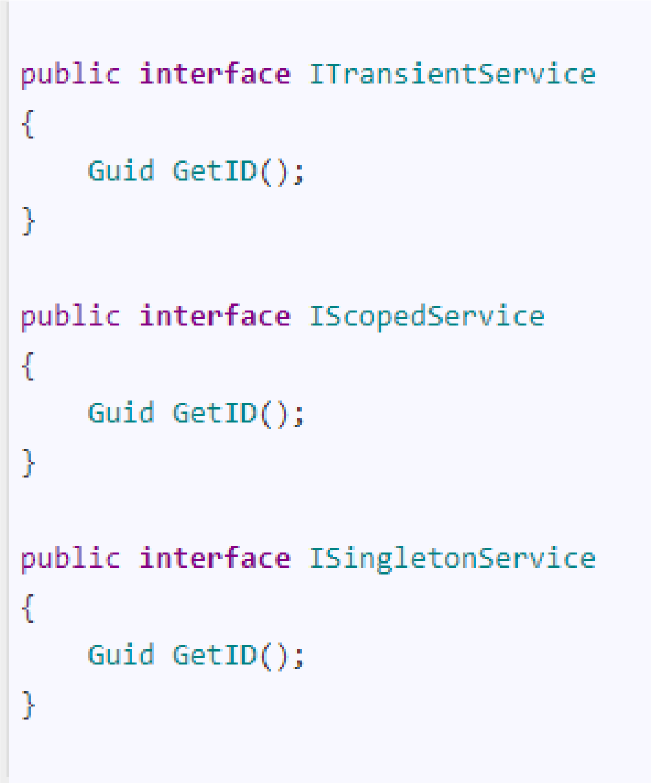
Scoped

- services are created per scope/request.
- In a web application, every web request creates a new separated service scope.
- That means scoped services are generally created per web request.

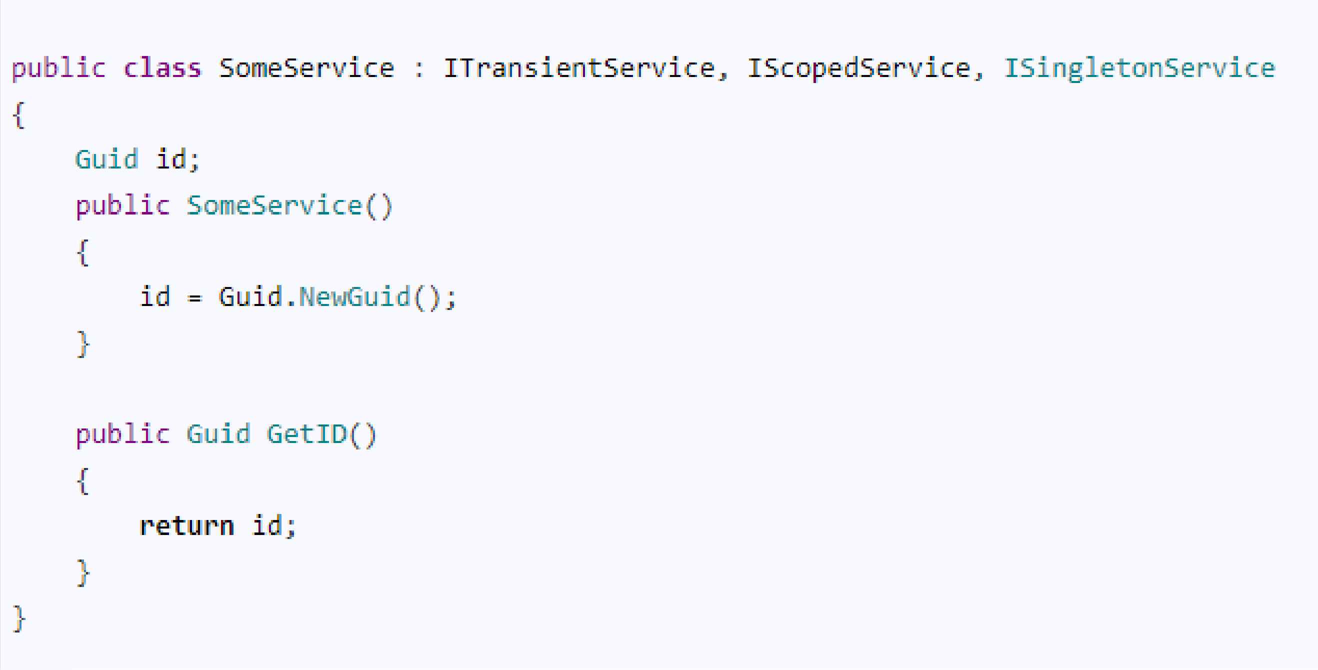
Transient

- Services are created every time they are injected or requested.
- Lightweight and stateless services.

Service Interfaces and Class



```
public interface ITransientService
{
    Guid GetID();
}

public interface IScopedService
{
    Guid GetID();
}

public interface ISingletonService
{
    Guid GetID();
}
```



```
public class SomeService : ITransientService, IScopedService, ISingletonService
{
    Guid id;
    public SomeService()
    {
        id = Guid.NewGuid();
    }

    public Guid GetID()
    {
        return id;
    }
}
```

- A GUID (Global Unique Identifier) is **a 128-bit integer used as a unique identifier.**
- System.Guid id.
- Guid.NewGuid() makes an actual guid with a unique value,
- GUIDs are most commonly written in text as a sequence of hexadecimal digits as such, 3F2504E0-4F89-11D3-9A0C-0305E82C3301

Register the Services

```
services.AddTransient<ITransientService, SomeService>();
```

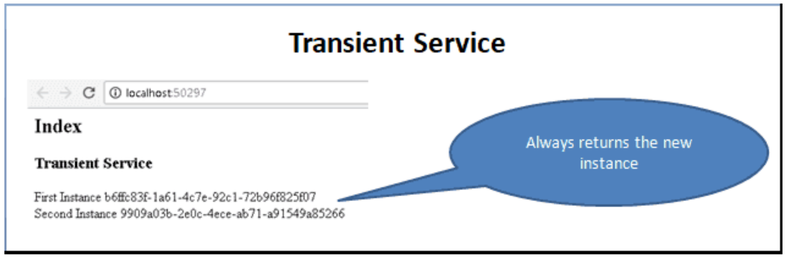
Inject it into Controller

```
public class HomeController : Controller
{
    ITransientService _transientService1;
    ITransientService _transientService2;

    public HomeController(ITransientService transientService1,
                          ITransientService transientService2)
    {
        _transientService1 = transientService1;
        _transientService2 = transientService2;
    }

    public IActionResult Index()
    {

        ViewBag.message1 ="First Instance " + _transientService1.GetID().ToString();
        ViewBag.message2 ="Second Instance "+ _transientService2.GetID().ToString();

        return View();
    }
}
```

View

```
<h3>Transient Service</h3>
@ViewBag.message1
</br>
@ViewBag.message2
```

Transient Service



Index

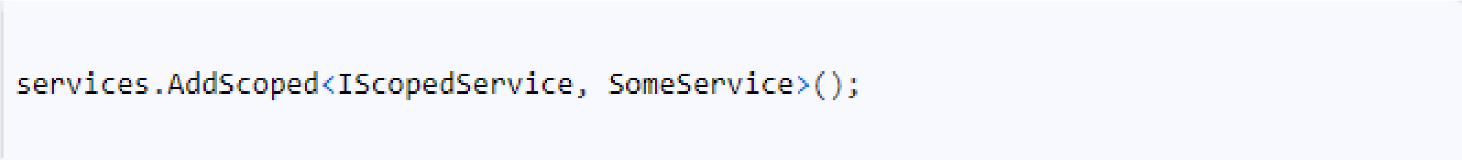
Transient Service

First Instance b6ffc83f-1a61-4c7e-92c1-72b96f825f07
Second Instance 9909a03b-2e0c-4ece-ab71-a91549a85266

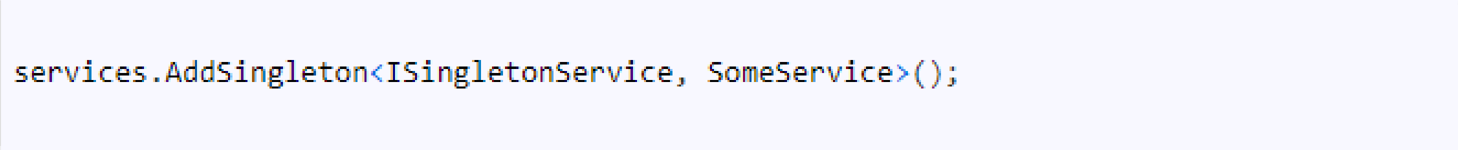
Always returns the new instance

TEKTUTORIALSHUB

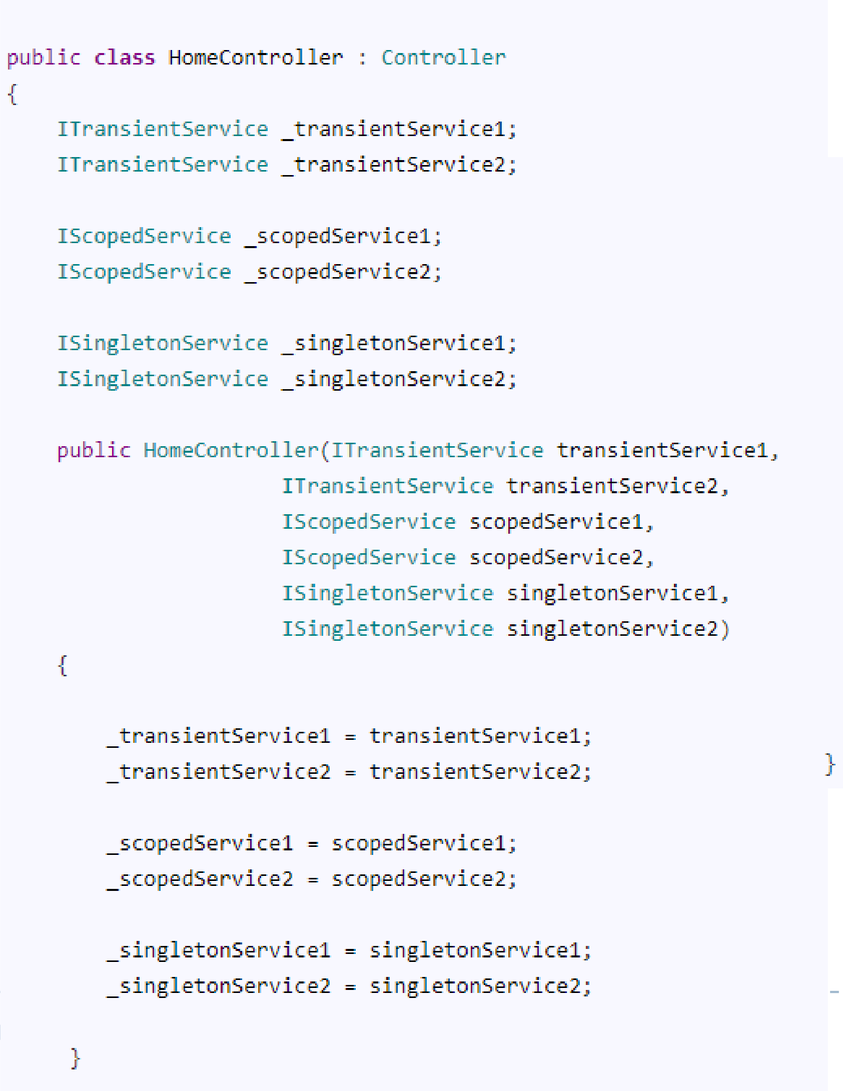
Other Services



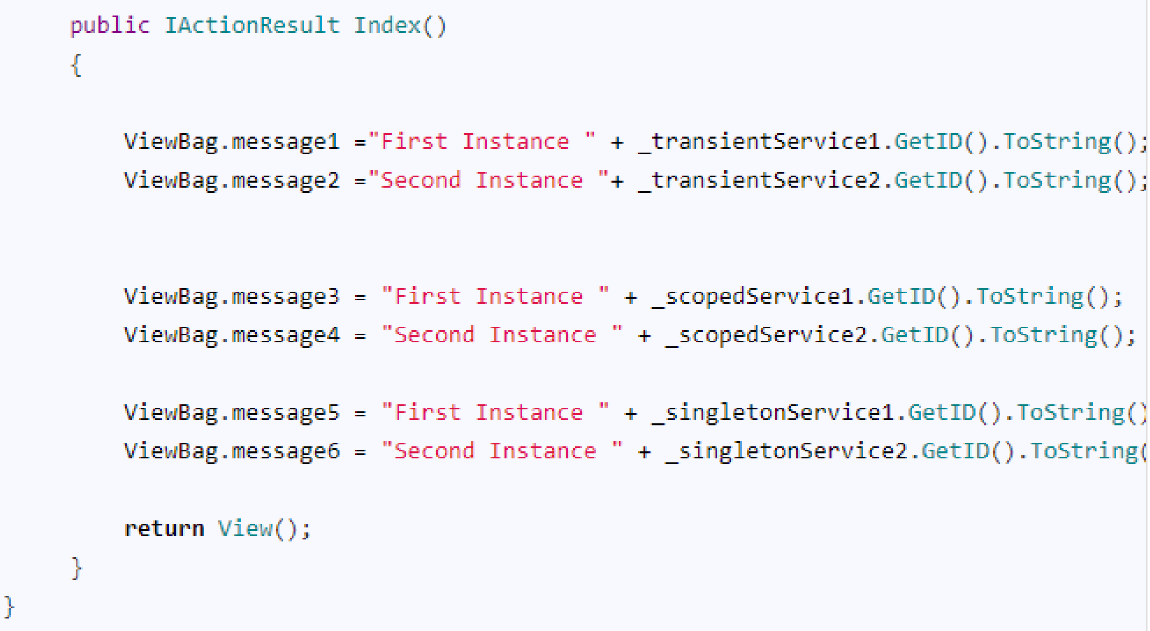
```
services.AddScoped<IScopedService, SomeService>();
```



```
services.AddSingleton<ISingletonService, SomeService>();
```



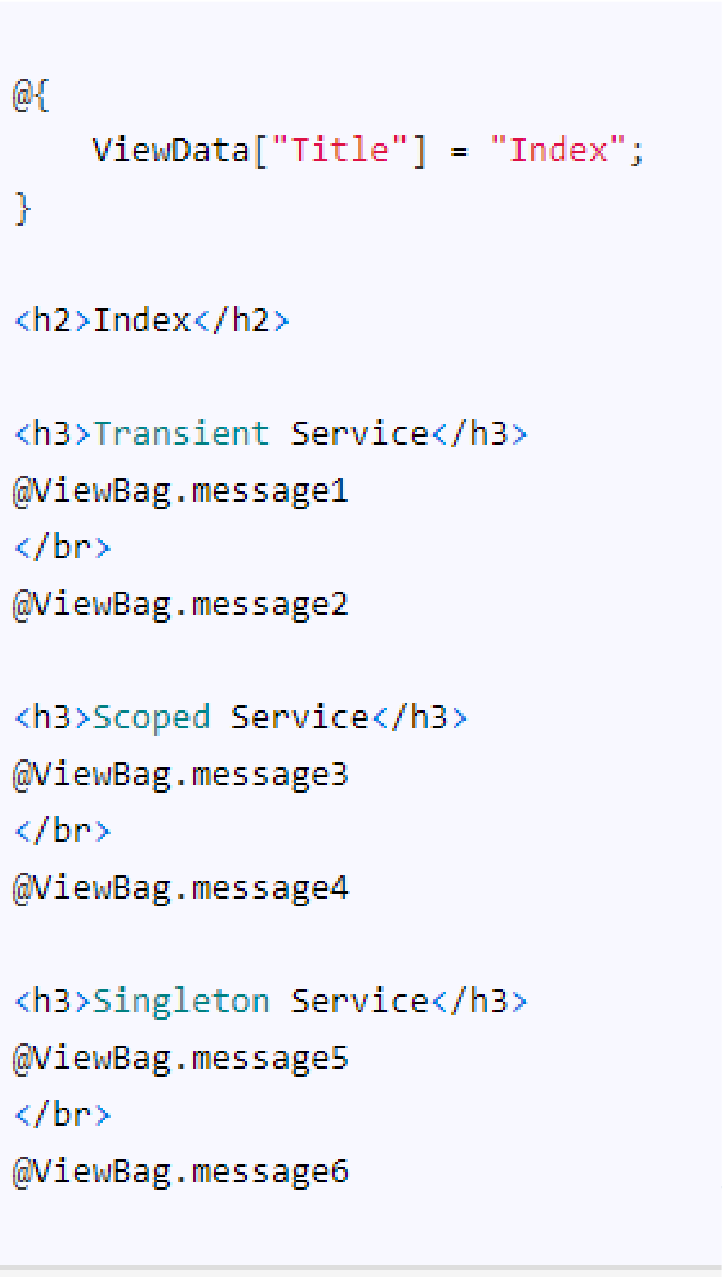
```
public class HomeController : Controller
{
    ITransientService _transientService1;
    ITransientService _transientService2;

    IScopedService _scopedService1;
    IScopedService _scopedService2;

    ISingletonService _singletonService1;
    ISingletonService _singletonService2;

    public HomeController(ITransientService transientService1,
                          ITransientService transientService2,
                          IScopedService scopedService1,
                          IScopedService scopedService2,
                          ISingletonService singletonService1,
                          ISingletonService singletonService2)
    {

        _transientService1 = transientService1;
        _transientService2 = transientService2;

        _scopedService1 = scopedService1;
        _scopedService2 = scopedService2;

        _singletonService1 = singletonService1;
        _singletonService2 = singletonService2;

     }
```



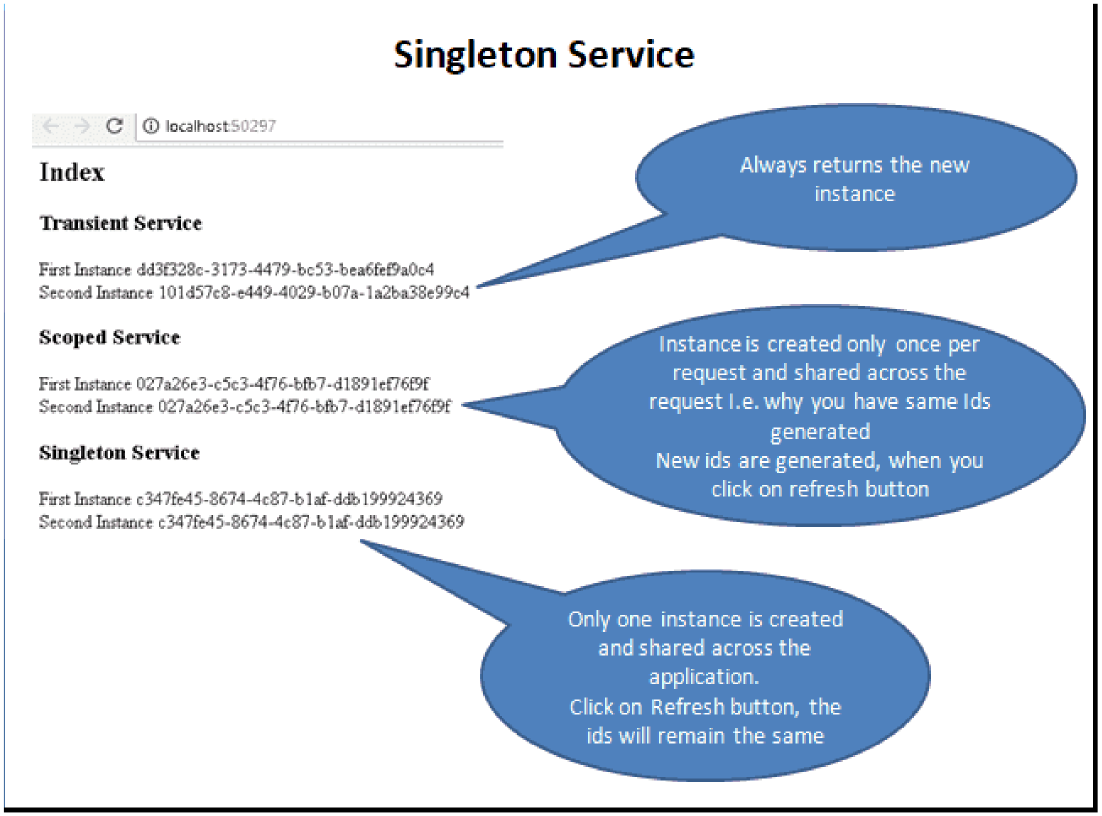
```
    public IActionResult Index()
    {


        ViewBag.message1 ="First Instance " + _transientService1.GetID().ToString();
        ViewBag.message2 ="Second Instance "+ _transientService2.GetID().ToString();


        ViewBag.message3 = "First Instance " + _scopedService1.GetID().ToString();
        ViewBag.message4 = "Second Instance " + _scopedService2.GetID().ToString();

        ViewBag.message5 = "First Instance " + _singletonService1.GetID().ToString()
        ViewBag.message6 = "Second Instance " + _singletonService2.GetID().ToString(

        return View();
    }
}
```

TEKTUTORIALSHUB

View



```
@{
    ViewData["Title"] = "Index";
}

<h2>Index</h2>

<h3>Transient Service</h3>
@ViewBag.message1
</br>
@ViewBag.message2

<h3>Scoped Service</h3>
@ViewBag.message3
</br>
@ViewBag.message4

<h3>Singleton Service</h3>
@ViewBag.message5
</br>
@ViewBag.message6
```



Singleton Service

Index

Transient Service

First Instance dd3f328c-3173-4479-bc53-bea6fef9a0c4
Second Instance 101d57c8-e449-4029-b07a-1a2ba38e99c4

Scoped Service

First Instance 027a26e3-c5c3-4f76-bfb7-d1891ef76f9f
Second Instance 027a26e3-c5c3-4f76-bfb7-d1891ef76f9f

Singleton Service

First Instance c347fe45-8674-4c87-b1af-ddb199924369
Second Instance c347fe45-8674-4c87-b1af-ddb199924369

Always returns the new instance

Instance is created only once per request and shared across the request I.e. why you have same Ids generated
New ids are generated, when you click on refresh button

Only one instance is created and shared across the application.
Click on Refresh button, the ids will remain the same