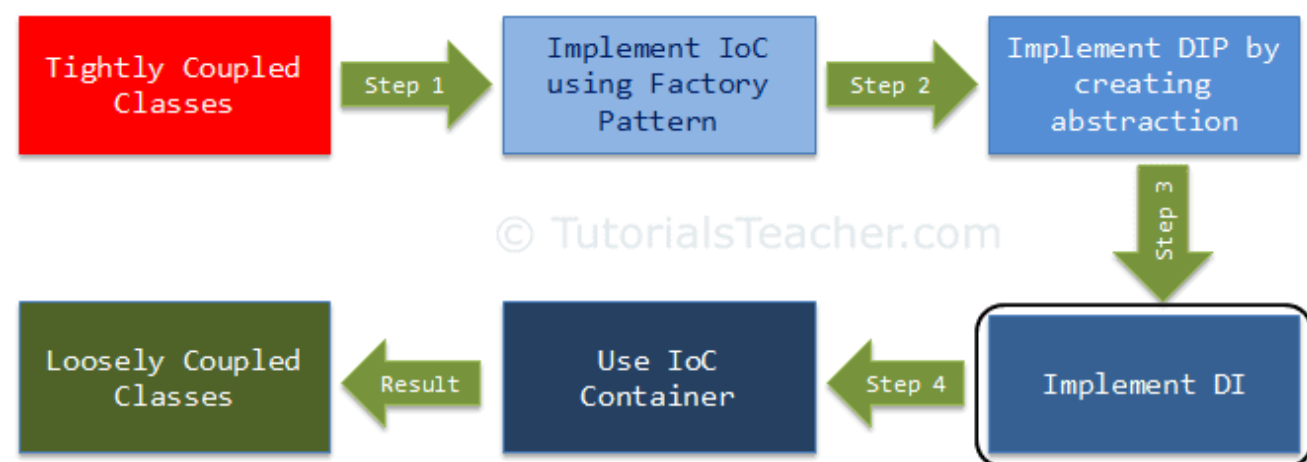


## Dependency Injection

In the previous chapter, related to DIP, we created and used abstraction to make the classes loosely coupled. Here, we are going to implement Dependency Injection and strategy pattern together to move the dependency object creation completely out of the class. This is our third step in making the classes completely loose coupled.

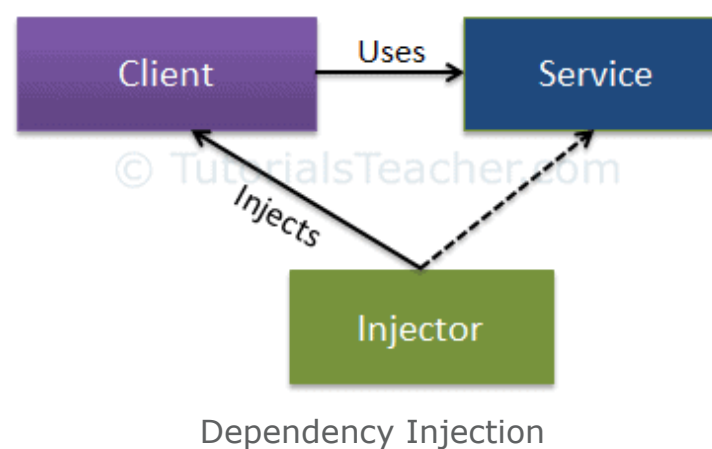


Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects to a class through different ways. Using DI, we move the creation and binding of the dependent objects outside of the class that depends on them.

The Dependency Injection pattern involves 3 types of classes.

1. **Client Class:** The client class (dependent class) is a class which depends on the service class
2. **Service Class:** The service class (dependency) is a class that provides service to the client class.
3. **Injector Class:** The injector class injects the service class object into the client class.

The following figure illustrates the relationship between these classes:



As you can see, the injector class creates an object of the service class, and injects that object to a client object. In this way, the DI pattern separates the responsibility of creating an object of the service class out of the client class.

### 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.

Let's take an example from the previous chapter to maintain the continuity. In the previous section of DIP, we used Factory class inside the `CustomerBusinessLogic` class to get an object of the `CustomerDataAccess` object, as shown below.

```
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();
    }
}

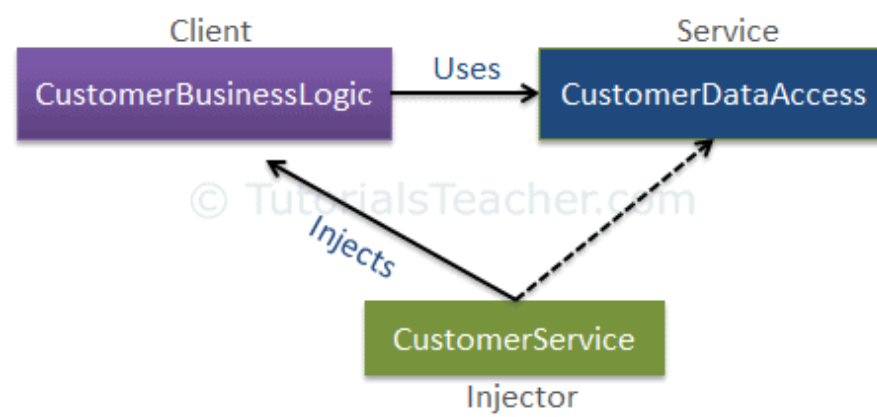
public class CustomerBusinessLogic
{
    ICustomerDataAccess _custDataAccess;

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

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

The problem with the above example is that we used `DataAccessFactory` inside the `CustomerBusinessLogic` class. So, suppose there is another implementation of `ICustomerDataAccess` and we want to use that new class inside `CustomerBusinessLogic`. Then, we need to change the source code of the `CustomerBusinessLogic` class as well. The Dependency injection pattern solves this problem by injecting dependent objects via a constructor, a property, or an interface.

The following figure illustrates the DI pattern implementation for the above example.



Dependency Injection

As you see, the **CustomerService** class becomes the injector class, which sets an object of the service class (**CustomerDataAccess**) to the client class (**CustomerBusinessLogic**) either through a constructor, a property, or a method to achieve loose coupling. Let's explore each of these options.

## Constructor Injection

As mentioned before, when we provide the dependency through the constructor, this is called a constructor injection.

Consider the following example where we have implemented DI using the constructor.

### Example: Constructor Injection - C#

[Copy](#)

```

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";
    }
}
  
```

In the above example, `CustomerBusinessLogic` includes the constructor with one parameter of type `ICustomerDataAccess`. Now, the calling class must inject an object of `ICustomerDataAccess`.

#### Example: Inject Dependency - C#

[Copy](#)

```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

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

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

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.

## Property Injection

In the property injection, the dependency is provided through a public property. Consider the following example.

#### Example: Property Injection - C#

[Copy](#)

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

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

    public ICustomerDataAccess DataAccess { get; set; }
}

public class CustomerService
{
    CustomerBusinessLogic _customerBL;

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

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

As you can see above, the `CustomerBusinessLogic` class includes the public property named `DataAccess`, where you can set an instance of a class that implements `ICustomerDataAccess`. So, `CustomerService` class creates and sets `CustomerDataAccess` class using this public property.

## Method Injection

In the method injection, dependencies are provided through methods. This method can be a class method or an interface method.

The following example demonstrates the method injection using an interface based method.

### Example: Interface Injection - C#



```
interface IDataAccessDependency
{
    void SetDependency(ICustomerDataAccess customerDataAccess);
}

public class CustomerBusinessLogic : IDataAccessDependency
{
    ICustomerDataAccess _dataAccess;

    public CustomerBusinessLogic()
    {
    }

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

    public void SetDependency(ICustomerDataAccess customerDataAccess)
    {
        _dataAccess = customerDataAccess;
    }
}

public class CustomerService
{
    CustomerBusinessLogic _customerBL;

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

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

In the above example, the `CustomerBusinessLogic` class implements the `IDataAccessDependency` interface, which includes the `SetDependency()` method. So, the injector class `CustomerService` will now use this method to inject the dependent class (`CustomerDataAccess`) to the client class.

Thus, you can use DI and strategy pattern to create loose coupled classes.

So far, we have used several principles and patterns to achieve loosely coupled classes. In professional projects, there are many dependent classes and implementing these patterns is time consuming. Here the IoC Container (aka the DI container) helps us. Learn about the IoC Container in the next chapter.

Learn more about the IoC and Dependency Injection [here](#).

 Share

 Tweet

 Share


 Whatsapp

[< Previous](#)

[Next >](#)

### TUTORIALSTEACHER.COM

TutorialsTeacher.com is optimized for learning web technologies step by step. Examples might be simplified to improve reading and basic understanding. While using this site, you agree to have read and accepted our terms of use and [privacy policy](#).

 [feedback@tutorialsteacher.com](mailto:feedback@tutorialsteacher.com)

### TUTORIALS

[ASP.NET Core](#)  
[ASP.NET MVC](#)  
[IoC](#)  
[Web API](#)  
[C#](#)  
[LINQ](#)  
[Entity Framework](#)

[AngularJS 1](#)  
[Node.js](#)  
[D3.js](#)  
[JavaScript](#)  
[jQuery](#)  
[Sass](#)  
[Https](#)

### E-MAIL LIST

Subscribe to TutorialsTeacher email list and get latest updates, tips & tricks on C#, .Net, JavaScript, jQuery, AngularJS, Node.js to your inbox.

Email address

GO

We respect your privacy.

[HOME](#) [PRIVACY POLICY](#) [TERMS OF USE](#) [ADVERTISE WITH US](#)

© 2020 TutorialsTeacher.com. All Rights Reserved.