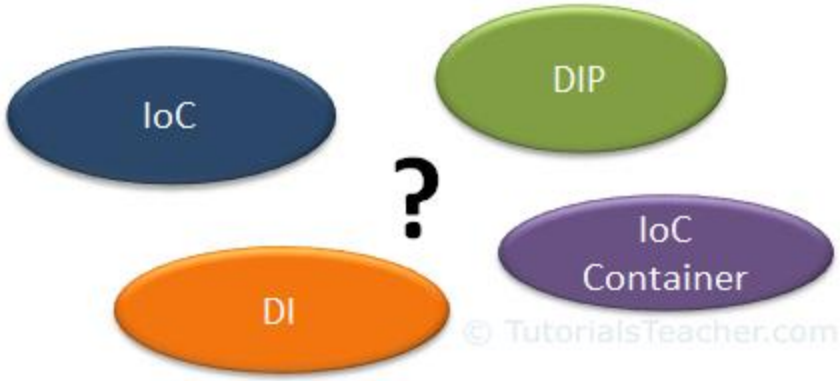
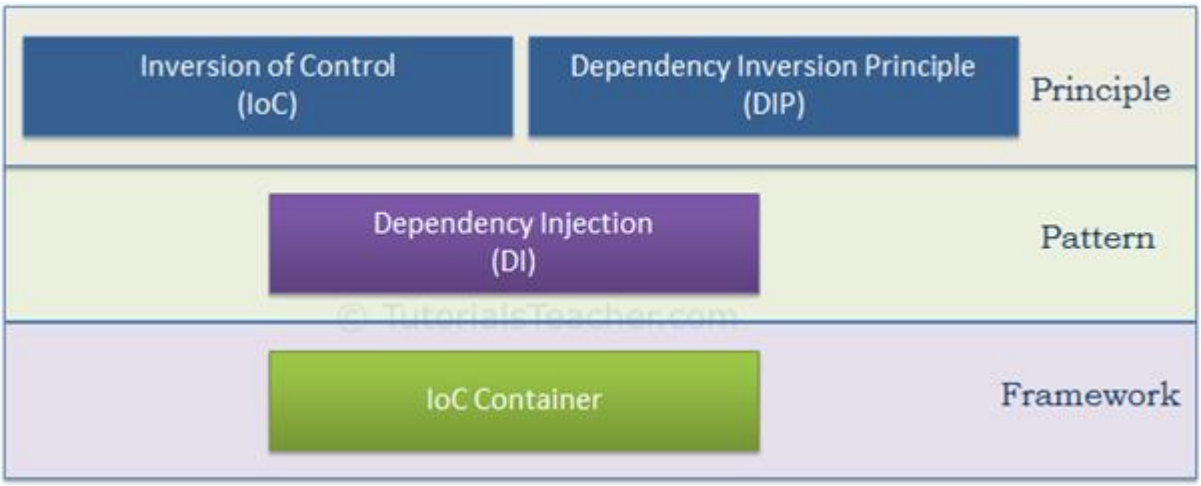
IoC Introduction

The terms Inversion of Control (IoC), Dependency Inversion Principle (DIP), Dependency Injection (DI), and IoC containers may be familiar. But are you clear about what each term means?



Here, you are going to learn about each term, using simple and real-world examples to clear your confusion. Before you go further, it is important to understand the difference between principle and pattern.

Now, let's understand the above buzz words. The following figure clarifies whether they are principles or patterns.



As illustrated in the above figure, IoC and DIP are high level design principles which should be used while designing application classes. As they are principles, they recommend certain best practices but do not provide any specific implementation details. Dependency Injection (DI) is a pattern and IoC container is a framework.

Let's have an overview of each term before going into details.

Inversion of Control

IoC is a design principle which recommends the inversion of different kinds of controls in object-oriented design to achieve loose coupling between application classes. In this case, control refers to any additional responsibilities a class has, other than its main responsibility, such as control over the flow of an application, or control over the dependent object creation and binding (Remember SRP - Single Responsibility Principle). If you want to do TDD (Test Driven Development), then you must use the IoC principle, without which TDD is not possible. Learn about IoC in detail in the next chapter.

Dependency Inversion Principle

The DIP principle also helps in achieving loose coupling between classes. It is highly recommended to use DIP and IoC together in order to achieve loose coupling.

DIP suggests that high-level modules should not depend on low level modules. Both should depend on abstraction.

The DIP principle was invented by Robert Martin (a.k.a. Uncle Bob). He is a founder of the SOLID principles. Learn more about DIP in the DIP chapter.

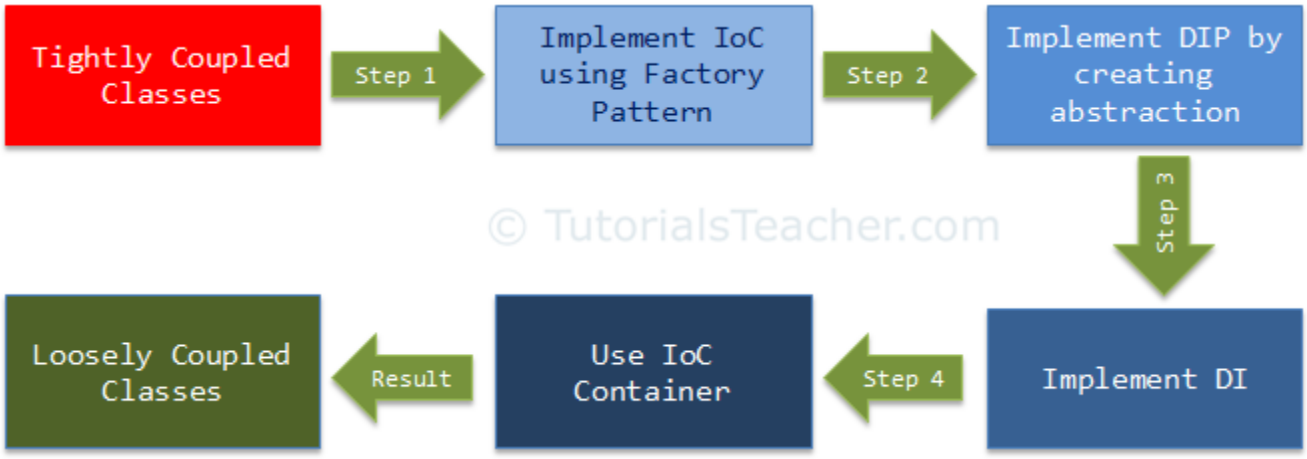
Dependency Injection

Dependency Injection (DI) is a design pattern which implements the IoC principle to invert the creation of dependent objects. We will learn about it in the DI chapter.

IoC Container

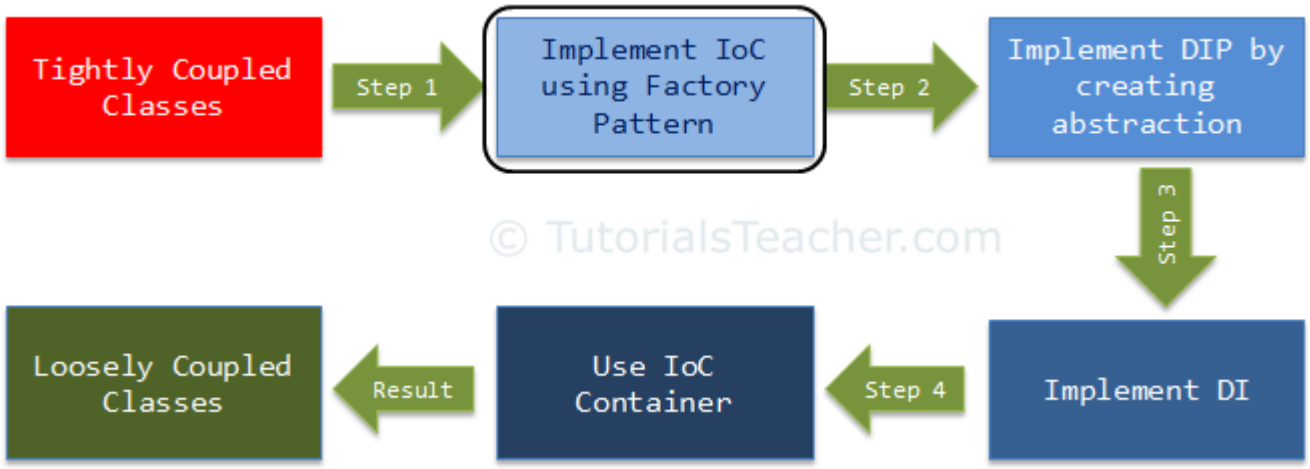
The IoC container is a framework used to manage automatic dependency injection throughout the application, so that we as programmers do not need to put more time and effort into it. There are various IoC Containers for .NET, such as Unity, Ninject, StructureMap, Autofac, etc. We will learn more about this in the IoC Container chapter.

We cannot achieve loosely coupled classes by using IoC alone. Along with IoC, we also need to use DIP, DI and IoC container. The following figure illustrates how we are going to achieve loosely coupled design step by step in the next few chapters.



Inversion of Control

In this chapter, we will learn about IoC and how to implement it. This is the first step towards achieving loose coupled design, as illustrated by the following figure:



Inversion of Control (IoC) is a design principle (although, some people refer to it as a pattern). As the name suggests, it is used to invert different kinds of controls in object-oriented design to achieve loose coupling. Here, controls refer to any additional responsibilities a class has, other than its main responsibility. This include control over the flow of an application, and control over the flow of an object creation or dependent object creation and binding.

IoC is all about inverting the control. To explain this in layman's terms, suppose you drive a car to your work place. This means you control the car. The IoC principle suggests to invert the control, meaning that instead of driving the car yourself, you hire a cab, where another person will drive the car. Thus, this is called inversion of the control - from you to the cab driver. You don't have to drive a car yourself and you can let the driver do the driving so that you can focus on your main work.

The IoC principle helps in designing loosely coupled classes which make them testable, maintainable and extensible.

Let's understand how IoC inverts the different kinds of control.

Control Over the Flow of a Program

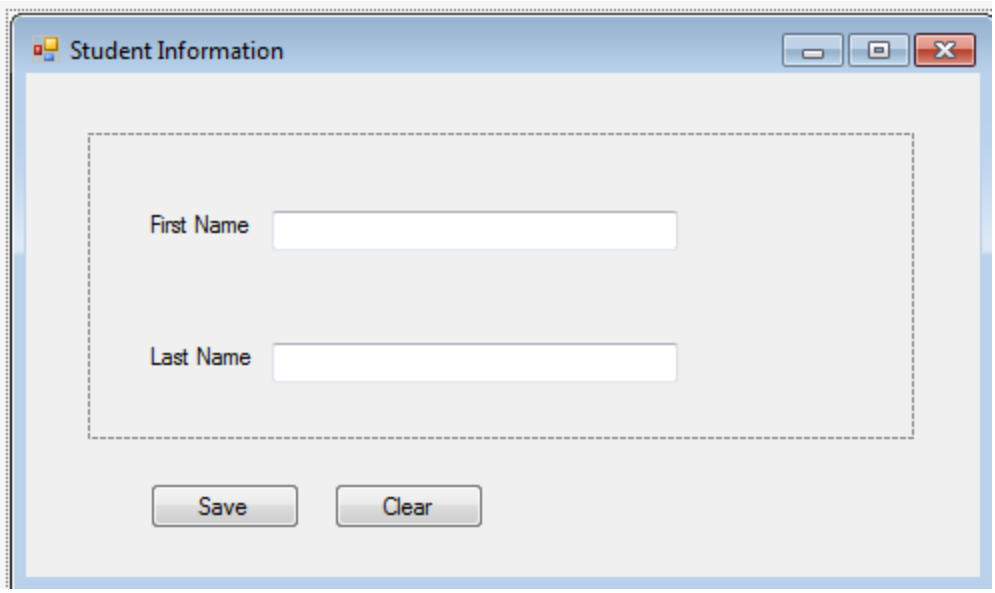
In a typical console application in C#, execution starts from the Main() function. The Main() function controls the flow of a program or, in other words, the sequence of user interaction. Consider the following simple console program.

Example: Program Flow

```
namespace FlowControlDemo
{
    class Program
    {
        static void Main(string[] args)
        {
           bool continueExecution = true;
            do
            {
                Console.Write("Enter First Name:");
                var firstName = Console.ReadLine();

                Console.Write("Enter Last Name:");
                var lastName = Console.ReadLine();

                Console.Write("Do you want to save it? Y/N: ");

                var wantToSave = Console.ReadLine();

                if (wantToSave.ToUpper() == "Y")
                    SaveToDB(firstName, lastName);

                Console.Write("Do you want to exit? Y/N: ");

                var wantToExit = Console.ReadLine();

                if (wantToExit.ToUpper() == "Y")
                    continueExecution = false;

            }while (continueExecution);

        }

        private static void SaveToDB(string firstName, string lastName)
        {
            //save firstName and lastName to the database here..
        }
    }
}
```

In the above example, the Main() function of the program class controls the flow of a program. It takes the user's input for the first name and last name. It saves the data, and continues or exits the console, depending upon the user's input. So here, the flow is controlled through the `Main()` function.

IoC can be applied to the above program by creating a GUI-based application such as the following windows-based application, wherein the framework will handle the flow of a program by using events.



This is a simple example of implementing IoC in the flow of a program.

Control Over the Dependent Object Creation

IoC can also be applied when we create objects of a dependent class. First of all, let's understand what we mean by dependency here.

Consider the following example.

```
public class A
{
    B b;

    public A()
    {
        b = new B();
    }

    public void Task1() {
        // do something here..
        b.SomeMethod();
        // do something here..
    }
```

```
}

public class B {

    public void SomeMethod() {
        //doing something..
    }
}
```

In the above example, class A calls b.SomeMethod() to complete its task1. Class A cannot complete its task without class B and so you can say that "Class A is dependent on class B" or "class B is a dependency of class A".

In the object-oriented design approach, classes need to interact with each other in order to complete one or more functionalities of an application, such as in the above example - classes A and B. Class A creates and manages the life time of an object of class B. Essentially, it controls the creation and life time of objects of the dependency class.

The IoC principle suggests to invert the control. This means to delegate the control to another class. In other words, invert the dependency creation control from class A to another class, as shown below.

```
public class A
{
    B b;

    public A()
    {
        b = Factory.GetObjectOfB ();
    }

    public void Task1() {
        // do something here..
        b.SomeMethod();
        // do something here..
    }
}

public class Factory
{
    public static B GetObjectOfB()
    {
        return new B();
    }
}
```

As you can see above, class A uses Factory class to get an object of class B. Thus, we have inverted the dependent object creation from class A to Factory. Class A no longer creates an object of class B, instead it uses the factory class to get the object of class B.

Let's understand this using a more practical example.

In an object-oriented design, classes should be designed in a loosely coupled way. Loosely coupled means changes in one class should not force other classes to change, so the whole application can become maintainable and extensible. Let's understand this by using typical n-tier architecture as depicted by the following figure:

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.

The following is an example of `BusinessLogic` and `DataAccess` classes for a customer.

```
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
    }
}
```

As you can see in the above example, the `CustomerBusinessLogic` class depends on the `DataAccess` class. It creates an object of the `DataAccess` class to get the customer data.

Now, let's understand what's wrong with the above classes.

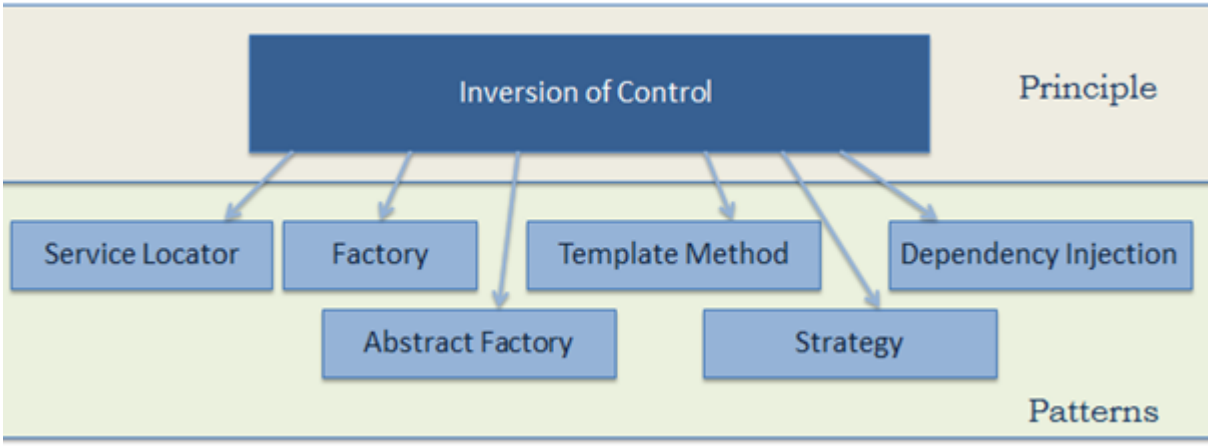
In the above example, `CustomerBusinessLogic` and `DataAccess` are tightly coupled classes because the `CustomerBusinessLogic` class includes the reference of the concrete `DataAccess` class. It also creates an object of `DataAccess` class and manages the lifetime of the object.

Problems in the above example classes:

1. `CustomerBusinessLogic` and `DataAccess` classes are tightly coupled classes. So, changes in the `DataAccess` class will lead to changes in the `CustomerBusinessLogic` class. For example, if we add, remove or rename any method in the `DataAccess` class then we need to change the `CustomerBusinessLogic` class accordingly.
2. Suppose the customer data comes from different databases or web services and, in the future, we may need to create different classes, so this will lead to changes in the `CustomerBusinessLogic` class.
3. The `CustomerBusinessLogic` class creates an object of the `DataAccess` class using the **new** keyword. There may be multiple classes which use the `DataAccess` class and create its objects. So, if you change the name of the class, then you need to find all the places in your source code where you created objects of `DataAccess` and make the changes throughout the code. This is repetitive code for creating objects of the same class and maintaining their dependencies.
4. Because the `CustomerBusinessLogic` class creates an object of the concrete `DataAccess` class, it cannot be tested independently (TDD). The `DataAccess` class cannot be replaced with a mock class.

To solve all of the above problems and get a loosely coupled design, we can use the IoC and DIP principles together. Remember, IoC is a principle, not a pattern. It just gives high-level design guidelines but does not give implementation details. You are free to implement the IoC principle the way you want.

The following pattern (but not limited) implements the IoC principle.



Let's use the *Factory* pattern to implement IoC in the above example, as the first step towards attaining loosely coupled classes.

First, create a simple Factory class which returns an object of the `DataAccess` class as shown below.

Example: DataAccess Factory

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

Now, use this `DataAccessFactory` class in the `CustomerBusinessLogic` class to get an object of `DataAccess` class.

Example: Use Factory Class to Retrieve Object

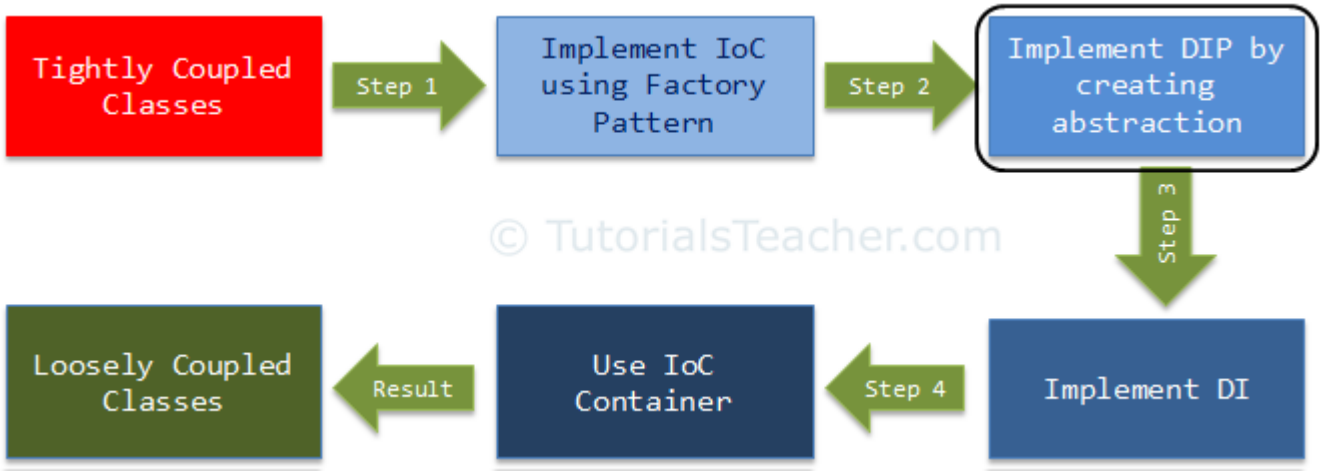
```
public class CustomerBusinessLogic
{

    public CustomerBusinessLogic()
    {
    }

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

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

As you can see, the `CustomerBusinessLogic` class uses the `DataAccessFactory.GetCustomerDataAccessObj()` method to get an object of the `DataAccess` class instead of creating it using the *new* keyword. Thus, we have inverted the control of creating an object of a dependent class from the `CustomerBusinessLogic` class to the `DataAccessFactory` class.

This is a simple implementation of IoC and the first step towards achieving fully loose coupled design. As mentioned in the previous chapter, we will not achieve complete loosely coupled classes by only using IoC. Along with IoC, we also need to use DIP, Strategy pattern, and DI (Dependency Injection).

Dependency Inversion Principle

In the previous chapter, we learned about implementing the IoC principle using the Factory pattern and achieved the first level of loosely coupled design. Here, we will learn how to implement the Dependency Inversion Principle as the second step to achieve loosely coupled classes.



First, let's understand what is Dependency Inversion Principle (DIP)?

DIP is one of the SOLID object-oriented principle invented by Robert Martin (a.k.a. Uncle Bob)

DIP Definition

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.

To understand DIP, let's take an example from the previous chapter, as shown below.

```
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
    }
}
```

In the above example, we implemented the factory pattern to achieve IoC. But, the `CustomerBusinessLogic` class uses the concrete `DataAccess` class. Therefore, it is still tightly coupled, even though we have inverted the dependent object creation to the factory class.

Let's use DIP on the `CustomerBusinessLogic` and `DataAccess` classes and make them more loosely coupled.

As per the DIP definition, a high-level module should not depend on low-level modules. Both should depend on abstraction. So, first, decide which is the high-level module (class) and the low-level module. A high-level module is a module which depends on other modules. In our example, `CustomerBusinessLogic` depends on the `DataAccess` class, so `CustomerBusinessLogic` is a high-level module and `DataAccess` is a low-level module. So, as per the first rule of DIP, `CustomerBusinessLogic` should not depend on the concrete `DataAccess` class, instead both classes should depend on abstraction.

The second rule in DIP is "Abstractions should not depend on details. Details should depend on abstractions".

What is an Abstraction?

Abstraction and encapsulation are important principles of object-oriented programming. There are many different definitions from different people, but let's understand abstraction using the above example.

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.

Now, what should be in the interface (or in the abstract class)? As you can see, `CustomerBusinessLogic` uses the `GetCustomerName()`method of the `DataAccess` class (in real life, there will be many customer-related methods in the `DataAccess` class). So, let's declare the `GetCustomerName(int id)` method in the interface, as shown below.

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

Now, implement `ICustomerDataAccess` in the `CustomerDataAccess` class, as shown below (so, instead of the `DataAccess` class, let's define the new `CustomerDataAccess` class).

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

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

Now, we need to change our factory class which returns `ICustomerDataAccess` instead of the concrete `DataAccess` class, as shown below.

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

Now, change the `CustomerBusinessLogic` class which uses `ICustomerDataAccess` instead of the concrete `DataAccess`, class as shown below.

```
public class CustomerBusinessLogic
```

```
{
    ICustomerDataAccess _custDataAccess;

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

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

Thus, we have implemented DIP in our example where a high-level module (CustomerBusinessLogic) and low-level module (CustomerDataAccess) are dependent on an abstraction (ICustomerDataAccess). Also, the abstraction (ICustomerDataAccess) does not depend on details (CustomerDataAccess), but the details depend on the abstraction.

The following is the complete DIP example discussed so far.

Example: DIP Implementation

```
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()
    {
```

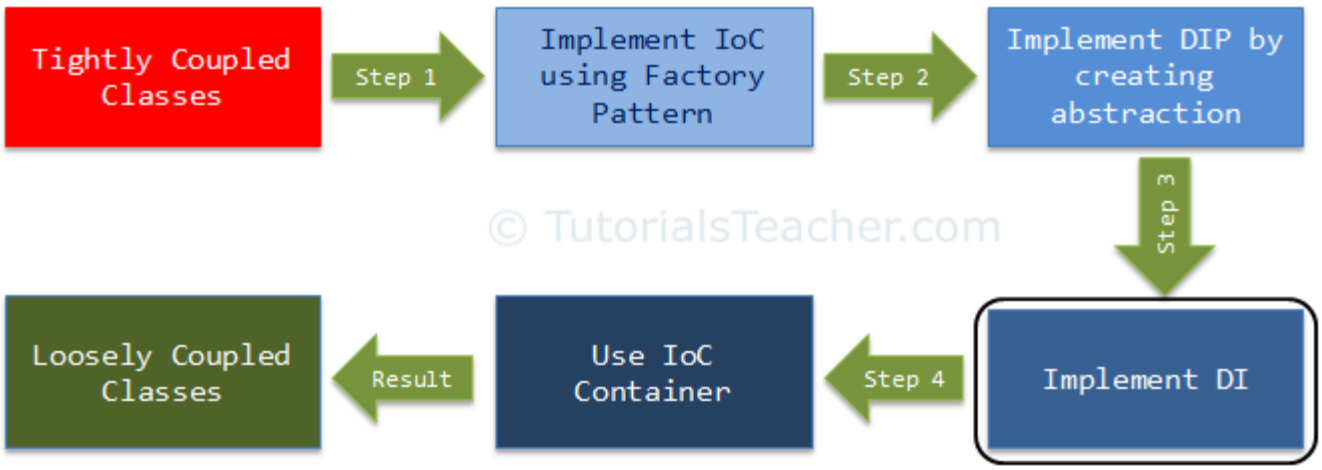
```
        _custDataAccess = DataAccessFactory.GetCustomerDataAccessObj();
    }

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

The advantages of implementing DIP in the above example is that the `CustomerBusinessLogic` and `CustomerDataAccess` classes are loosely coupled classes because `CustomerBusinessLogic` does not depend on the concrete `DataAccess` class, instead it includes a reference of the `ICustomerDataAccess` interface. So now, we can easily use another class which implements `ICustomerDataAccess`with a different implementation.

Still, we have not achieved fully loosely coupled classes because the `CustomerBusinessLogic` class includes a factory class to get the reference of `ICustomerDataAccess`. This is where the Dependency Injection pattern helps us. In the next chapter, we will learn how to use the Dependency Injection (DI) and the Strategy pattern using the above example.

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.



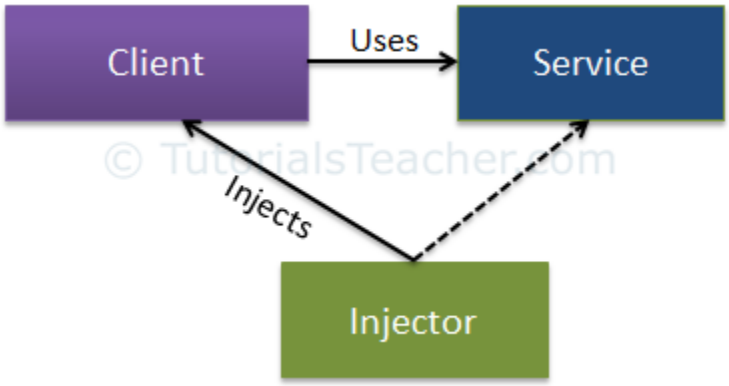
Dependency Injection (DI) is a design pattern used to implement IoC. It allows the creation of dependent objects outside of a class and provides those objects

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:



Dependency Injection

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.

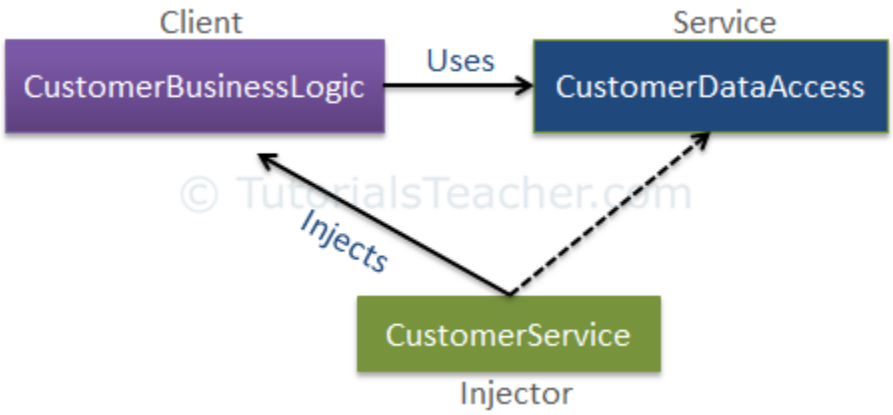
Method Injection: In this type of injection, the client class implements an interface which declares the method(s) to supply the dependency and the injector uses this interface to supply the dependency to the client class.

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

```
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 GetCustomerData(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

```
public class CustomerService
{
    CustomerBusinessLogic _customerBL;

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

    public string GetCustomerName(int id) {
        return _customerBL.GetCustomerName(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

```
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

```
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()` mehtod. 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.