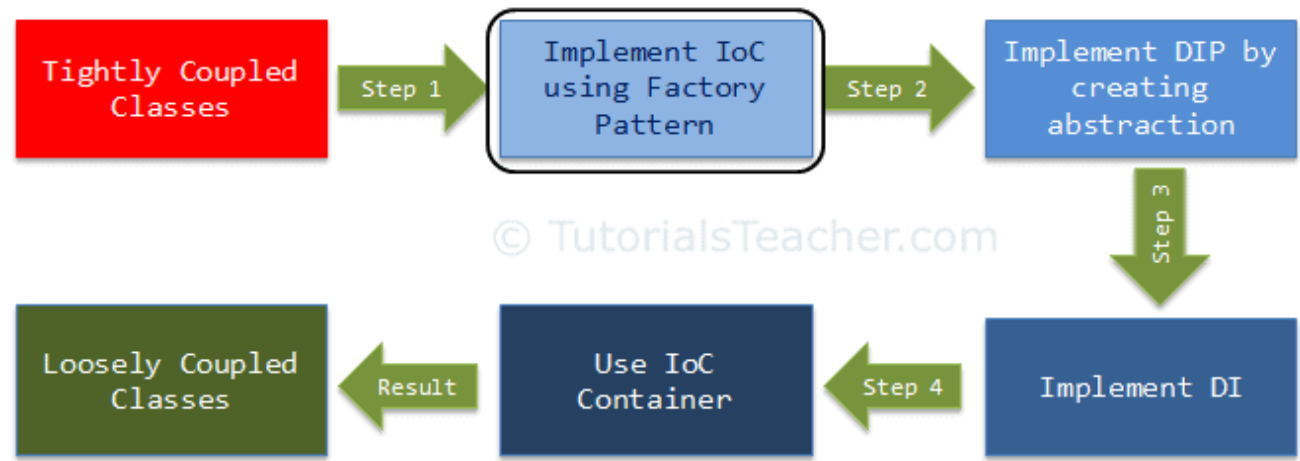


# Inversion of Control

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 - C#

Copy

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

        var wantToSave = Console.ReadLine();

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

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

        var wantToExit = Console.ReadLine();

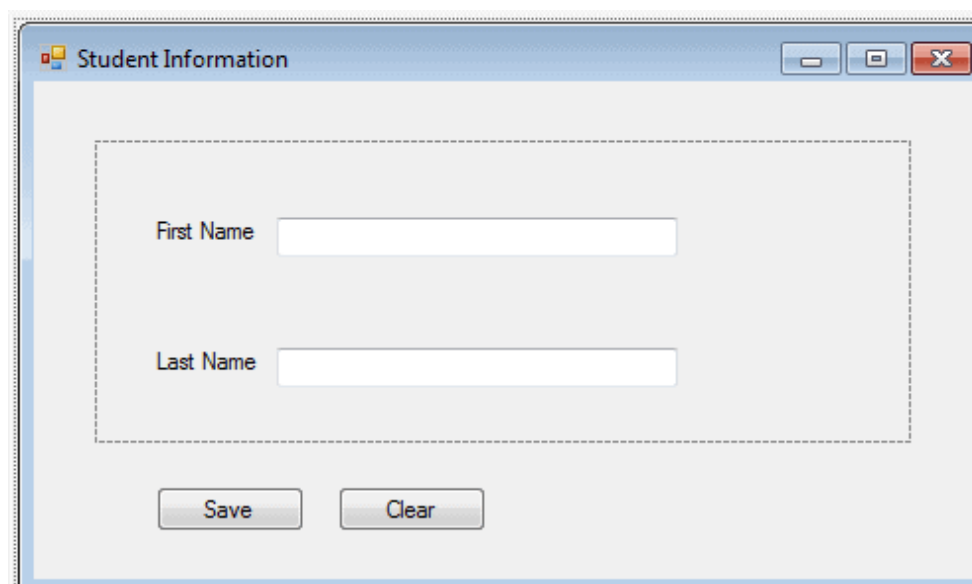
        if (wantToExit.ToUpper() == "Y")
            continueExecution = false;

    }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.

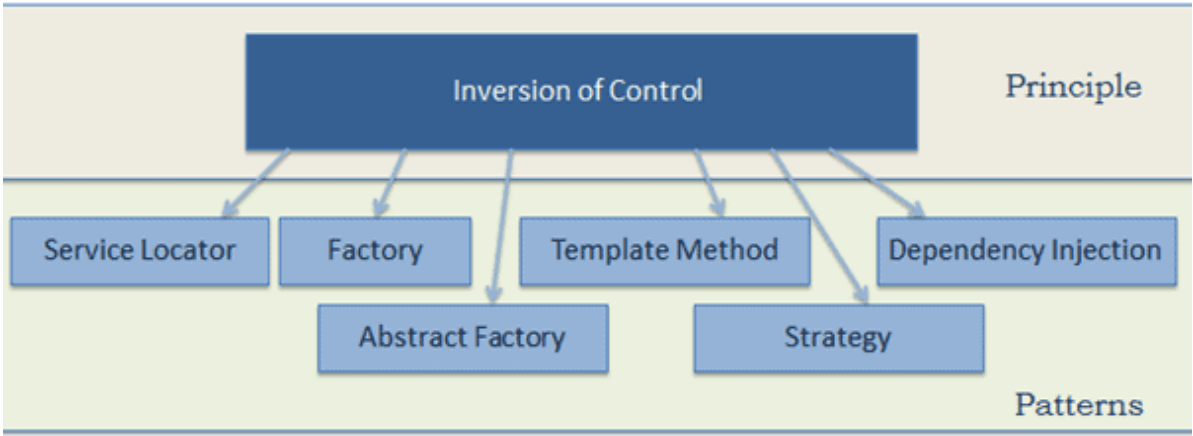
In the above example, `CustomerBusinessLogic` and `DataAccess` are tightly coupled classes because the `CustomerBusinessLogic` class includes the reference of the concrete `DataAccess` class. It also creates an object of `DataAccess` class and manages the lifetime of the object.

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 - C#

Copy

```
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 - C#

Copy

```
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.GetDataAccessObj()` 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).

Let's move to the second step to understand DIP and how it helps in achieving loose coupled design in the next chapter.

 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.