Inversion of Control (IOC) and Dependency Injection (DI)

# Story on IOC

An organization conducts a meeting in a hotel for the employees. Employees use disposable glass to drink water and throws the glass into dustbin. After an hour the dustbin is filled up with the disposable cups. This seems to be a critical scenario.

Now let us invert the control.

When somebody wants to drink, the employee has to make a request to a waiter in the hotel and the waiter server the glass of water. When the next employee asks for water, the waiter reuses the same glass and server the water to the employees. Here we get the single instance of glass and waiter the IOC container who serves the water to the employees.

# Principles used in Dependency Injection and IOC

* **Single Responsibility Principle**
* **Open and Closed Principle**
* **Hollywood Principle (Don’t call us we will call you)**
* **Dependency Inversion** – Higher level classes and lower level classes should not be dependent on each other: both should depend upon abstraction

<http://stackoverflow.com/questions/3058/what-is-inversion-of-control>

The Inversion of Control (IoC) and Dependency Injection (DI) patterns are all about removing dependencies from your code.

For example, say your application has a text editor component and you want to provide spell checking. Your standard code would look something like this:

public class TextEditor

{

private SpellChecker checker;

public TextEditor()

{

this.checker = new SpellChecker();

}

}

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

public class TextEditor

{

private ISpellChecker checker;

public TextEditor(ISpellChecker checker)

{

this.checker = checker;

}

}

Now, the client creating the TextEditor class has the control over which SpellChecker implementation to use. We're injecting the TextEditor with the dependency.

We can provide Spell Checker in the form of DLL or Jar file.

**IOC can be thought of Separate what-to-do part from when-to-do part**.

**Good Examples**

**1)**

(The kitchen in your office only serves clean tap water, that is your only choice when you want to drink. Your boss implemented Inversion of Control by setting up a new coffee machine. Now you get the flexibility of choosing either tap water or coffee.)

**2)**

When you use a desktop computer, you have slaved (or say, controlled). You have to sit before a screen and look at it. Using the keyboard to type and using the mouse to navigate. And a badly written software can slave you even more. If you replace your desktop with a laptop, then you somewhat inverted control. You can easily take it and move around. So now you can control where you are with your computer, instead of your computer controlling it.

By implementing Inversion of Control, a software/object consumer get more controls/options over the software/objects, instead of being controlled or having fewer options.

**3)**

**Whitout IoC**: You have a **laptop** computer and you accidentally break the screen. And darn, you find the same brand laptop screen is not anywhere in the market. So you stuck.

**With IoC**: You have a **desktop** computer and you accidentally break the screen. You find you can just grap any brand monitor from the market, and it just works well with your desktop.

Desktop successfully implements the IoC in this case. It just accept any type of monitor, while the laptop does not, it has to require a specific screen to get fixed.

**4)**

Suppose you are an object. And you go to a restaurant:

**Without IoC**: you ask for "apple", and you are always served apple when you ask more.

**With IoC**: You can ask for "fruit". You can get different fruits each time you get served. for example, apple, orange, or water melon.

# Inversion of Control (IOC)

(IoC) is a programming technique in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis. Intraditional programming, the flow of the business logic is determined by objects that are statically assigned to one another. With **inversion of control**, the flow depends on the object graph that is instantiated by the assembler. The binding process is achieved through **dependency injection**, although some argue that the use of a service locator also provides inversion of control.

Inversion of control is a way to design a system where all the modules are thought of abstract entities. In traditional approach/applications, developers used to write business code and framework code. The business code then calls the framework code to accomplish the tasks. Under an IOC model, we invert the model and create a framework that accepts business modules and call them to accomplish tasks.

Inversion of control as a ***design guideline*** serves the following purposes:

1. **There is a decoupling of the execution of a certain task from implementation.**
2. **Every module can focus on what it is designed for.**
3. **Modules make no assumptions about what other systems do but rely on their contracts.**
4. **Replacing modules has no side effect on other modules.**

Inversion of control is a design paradigm with the goal of giving more control to the targeted components of your application, the ones getting the work done.  
Dependency injection is a pattern used to create instances of objects that other objects rely on without knowing at compile time which class will be used to provide that functionality. Inversion of control relies on dependency injection because a mechanism is needed in order to activate the components providing the specific functionality.

Inversion of control serves the following design purposes:

* To [decouple](https://en.wikipedia.org/wiki/Object_decoupling) the execution of a task from implementation.
* To focus a module on the task it is designed for.
* To free modules from assumptions about how other systems do what they do and instead rely on [contracts](https://en.wikipedia.org/wiki/Design_by_contract).
* To prevent [side effects](https://en.wikipedia.org/wiki/Side_effect_(computer_science)) when replacing a module.

# Implementing inversion of control design pattern

In object-oriented programming, there are several basic techniques to implement inversion of control. These are:

1. using a factory pattern
2. using a service locator pattern
3. using a **dependency injection** of any given below type:
   * a constructor injection
   * a setter injection
   * an interface injection

# Dependency Injection

DI is a technique of removing internal dependencies from implementations. DI is an implementation where the concrete relation between different modules is decided at **run-time**. It is a way of injecting properties to an object or an act of wiring properties to an object.

The key benefit of a Dependency Injector is that it allows to plug-in a suitable implementation of a service according to environment and usage.

Inversion of control is used to increase [modularity](https://en.wikipedia.org/wiki/Modularity_(programming)) of the program and make it [extensible](https://en.wikipedia.org/wiki/Extensible_programming).

# How Hollywood Principle works and Who calls whom

At a glance, these patterns are said to be based on the [Hollywood Principle](http://en.wikipedia.org/wiki/Hollywood_Principle), which states: "don't call us, we'll call you". With a canonical approach, you hard code the classes of the objects you want to instantiate in the source of your application, supply parameters to their constructors and manage their interactions. Each object knows at compile time which are the real classes of the objects they need to interact with, and they will call them directly. So, under this point of view, you and your objects are the ones calling Hollywood. To invert this approach, you need some support from a framework which makes your application smart enough to guess which objects to instantiate, how to instantiate them and, in general, how to control their behavior. Instead of working with concrete classes you'll work with abstractions like interfaces or abstract classes, letting your application decide which concrete classes to use and how to satisfy their dependencies on other components. This concept may sound a bit weird in the beginning, but you'll see that this makes a lot of sense.

The best java level technical example will be Logger, it has both Console Logger and File Logger. In case of IOC, both logger can be used as and when required.

# Difference between IOC and NON-IOC

  
let us consider a typical example “**A person is charging phone with a charger**”

public interface IUSBCable {  
 String connect();  
}

public interface ICellPhone {  
 void charge( IUSBCable cable);  
}

**Test Class (NON-IOC)**

public class NokiaCable implements IUSBCable {  
 public String connect() {  
 return "Charging ...";  
 }  
}

public class NokiaPhone implements ICellPhone {  
 public void charge(IUSBCable cable) {  
 //.... Other code  
 String status = cable.connect();  
 System.*out*.println(status);  
 }  
}

public class Person {  
 public static void chargeMobile() {  
 ICellPhone phone = new NokiaPhone();  
 IUSBCable cable = new NokiaCable();  
 phone.charge(cable);  
 }  
 public static void main(String[] args) {  
 *chargeMobile*();   
 }  
}

**IOC Approach – Using Spring**

public class Person {  
 public static void chargeMobile(ICellPhone phone , IUSBCable cable) {  
 phone.charge(cable);  
 }  
  
 public static void main(String[] args) {  
 ApplicationContext context = new ClassPathApplicationContext("beans.xml");  
 ICellPhone phone = (ICellPhone) context.getBean("phone");  
 IUSBCable cable = (IUSBCable) context.getBean("usbCharger");  
 *chargeMobile*(phone,cable);  
 }  
}

# In future, I may change my phone but not necessarily charger. Let us think if a person looses his phone but not the charger, he can use Nokia phone with Samsung charger. Inversion of Control (IOC) and Dependency Injection (DI)

**Story behind IOC**

An organization conducts a meeting in a hotel. Employees participate in that meeting. Employees use disposable glass to drip water. So each employee fills up the glass with water. After an hour the dustbin is filled up with the disposable cups. This is a very typical scenario.

Now let us invert the control.

When somebody wants to drink, the employee has to make a request to a waiter in the hotel and the waiter serves the glass of water. When next employee asks for water, the waiter recuses the same glass and serves the water to the employees.

Here we create a single instance of glass and waiter the IOC container who serves the water to the employees.

**Principles used in Dependency Injection & IOC?**

**Single Responsibility Principle**

**Open and Closed principle**

**Hollywood principle (Don’t call us we will call you)**

**Dependency inversion** - Higher level classes and lower level classes should not be dependent on each other; both should be depend on abstraction.

If you follow these simple two steps, you have done inversion of control:

**Separate what-to-do part from when-to-do part**.

Ensure that when part knows as little as possible about what part; and vice versa.

**Simple Introduction to Dependency Injection**

**A simple introduction**

Scenario 1

You work in an organization where you and your colleagues tend to travel a lot. Generally you travel by air and every time you need to catch a flight, you arrange for a pickup by a cab. You are aware of the airline agency who does the flight bookings, and the cab agency which arranges for the cab to drop you off at the airport. You know the phone numbers of the agencies, you are aware of the typical conversational activities to conduct the necessary bookings. Thus your typical travel planning routine might look like the following :

Decide the destination, and desired arrival date and time. Call up the airline agency and convey the necessary information to obtain a flight booking. Call up the cab agency, request for a cab to be able to catch a particular flight from say your residence (the cab agency in turn might need to communicate with the airline agency to obtain the flight departure schedule, the airport, compute the distance between your residence and the airport and compute the appropriate time at which to have the cab reach your residence) . Pickup the tickets, catch the cab and be on your way.

Now if your company suddenly changed the preferred agencies and their contact mechanisms, you would be subject to the following relearning scenarios.

The new agencies, and their new contact mechanisms (say the new agencies offer internet based services and the way to do the bookings is over the internet instead of over the phone)

The typical conversational sequence through which the necessary bookings get done (Data instead of voice).

Its not just you, but probably many of your colleagues would need to adjust themselves to the new scenario. This could lead to a substantial amount of time getting spent in the readjustment process.

For example, say your application has a text editor component and you want to provide spell checking. Your standard code would look something like this:

public class TextEditor

{

private SpellChecker checker;

public TextEditor()

{

checker = new SpellChecker();

}

}

What we've done here is create a dependency between the TextEditor and the SpellChecker. In an IoC scenario we would instead do something like this:

public class TextEditor

{

private ISpellChecker checker;

public TextEditor(ISpellChecker checker)

{

this.checker = checker;

}

}

Now, the client creating the TextEditor class has the control over which SpellChecker implementation to use. We're injecting the TextEditor with the dependency. It can be spanish spell checker or English spell cheker.

**Inversion of control** is a design paradigm with the goal of giving more control to the targeted components of your application.

I would say "Inversion of Control" is a way to design a system where all the modules are thought of abstract entities. In traditional applications, developers would write business code and framework code. The business code would then call the framework code to accomplish tasks. Under an IoC model, you "invert" that model and create a framework that accepts business modules and calls them to accomplish tasks.

**IOC(Inversion Of Controller)**: Giving control to the container to get instance of object is called Inversion of Control., means instead of you are creating object using new operator, let the container do that for you.

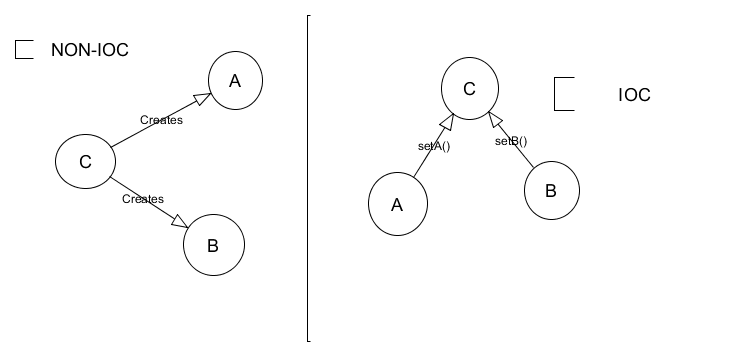
IoC = You give the control to the container to get instance of objects. So instead of you doing "new", let the container do that for you.

**Dependency Injection** is a technique (hard to call it a pattern, really) of removing internal dependencies from implementations by allowing dependent objects to be injected into the class/method by an external caller. "Dependency Injection" is an implementation where the concrete relation between different modules is decided at "run time".

DI(Dependency Injection): Way of injecting properties to an object is called Dependency injection.

DI = The act of "wiring" properties to an object.

Difference Between IOC and DI



Let us consider a typical example “ A person is charging phone with a charger”.

**NON-IOC-Approach**

public class Person {

public static void chargeMobile() {

ICelloPhone phone = new NokiaPhone();

IUSBCable cable = new NokiaCable();

phone.charge(cable);

}

public static void main(String[] args) {

chargeMobile();

}

}

public class NokiaCable Implements IUSBCable {

public String connect() {

return "charging";

}

}

public class NokiaPhone implements ICelloPhone {

public void charge(IUSBCable cable) {

String status = cable.connect();

System.out.println("charging ... ");

}

}

public interface ICelloPhone {

void charage(IUSBCable cable);

}

public interface IUSBCable {

String connect();

}

**IOC – Approach**

public class Person {

public static void chargeMobile(ICelloPhone phone , IUSBCable cable) {

phone.charge(cable);

}

public static void main(String[] args) {

ApplicationContext context = new ClassPathApplicationContext("beans.xml");

ICelloPhone phone = (ICelloPhone) context.getBean("phone");

IUSBCable cable = (IUSBCable) context.getBean("usbCharger");

chargeMobile(phone,cable);

}

}