**Inversion of Control (IOC) and Dependency Injection (DI) – 2025**

* **IOC is the broader concept of inverting control, while DI is a specific technique for managing dependencies within that framework.**
* **IOC is like a boarding school, pupils are thought of objects.**
* **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 is a design paradigm with the goal of giving more control to the targeted components of your application.

**Dependency injection is a pattern where one object(dependent object) receives another object(the dependency) without creating dependency.**

**Why Dependency Injection**

1. **Loose Coupling**
2. **Separation of Concerns**
3. **Inversion of control**

# 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.

# 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

# Difference between IOC and NON-IOC

A graph with numbers and symbols

Description automatically generated with low confidence  
let us consider a typical example “**A person is charging phone with a charger**”

**Inversion of Control (IoC) means that objects do not create other objects on which they rely to do their work**.

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

**Class (NON-IOC)**

public class A {  
}

public class C {  
  
 public void process(A a, B b) {  
 System.*out*.println("A-->"+a);  
 System.*out*.println("B-->"+b);  
 }  
  
 public static void main(String[] args) {  
 A a = new A();  
 B b = new B();  
 new C().process(a, b);  
 }  
}

public class B {  
}

**IOC Approach – Using Spring**

public class C {  
  
 public void process(A a, B b) {  
 System.*out*.println("A-->"+a);  
 System.*out*.println("B-->"+b);  
 }  
  
 public static void main(String[] args) {  
 ApplicationContext context = new ClassPathApplicationContext("beans.xml");  
 A a = (A) context.getBean("a");  
 B b = (B) context.getBean("b");

new C().process(a, b);  
 }  
}

**In the similar fashion, other example is given below.**

**public** **interface** IUSBCable {

String connect();

}

**public** **interface** ICelloPhone {

**void** charge(IUSBCable cable);

}

**public** **class** NokiaCable **implements** IUSBCable {

**public** String connect() {

**return** "Mobile is getting charged...";

}

}

**public** **class** NokiaPhone **implements** ICelloPhone {

**public** **void** charge(IUSBCable cable) { System.***out***.println(cable.connect());

}

}

**Class (NON-IOC)**

public class Person {  
 public static void main(String[] args) {

ICellPhone phone = new NokiaPhone();  
 IUSBCable cable = new NokiaCable();  
 phone.charge(cable);  
 }  
}

**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.

**How will you create your own @Autowire Type using Core Java**

public class B {  
}

public class A {  
  
 public void perform(B b) {  
 System.*out*.println("A is performing for B class-->"+b);  
 }  
}

public class C {  
  
 @Wiring("A")  
 private A a;  
 @Wiring("B")  
 private B b;  
  
 public void process() {  
 a.perform(b);  
 }  
}

**First create an annotation like this**

@Retention(RetentionPolicy.***RUNTIME***)

@Target(ElementType.***FIELD***)

**public** **@interface** **Wiring** {

String value() **default** "somevalue";

}

**Create A Spring Type Container**

public class Container {  
  
 public Object getObject(String pkgName, String className) {  
 Object actualObj = null;  
 try {  
 String pkgClsName = pkgName+"."+className;  
 **actualObj = Beans.*instantiate*(ClassLoader.*getSystemClassLoader*(),pkgClsName);**  
 Field[] flds = actualObj.getClass().getDeclaredFields();  
 for (Field fld : flds) {  
 fld.setAccessible(true);  
 if(fld.isAnnotationPresent(Wiring.class)) {  
 **Wiring wire = fld.getDeclaredAnnotation(Wiring.class);** String annotedValue = wire.value(); *// This is class name* String annotedPkgClsName = pkgName+"."+annotedValue;  
 Object innerObj = Beans.*instantiate*(ClassLoader.*getSystemClassLoader*(), annotedPkgClsName);  
 fld.set(actualObj,innerObj);  
 }  
 }  
 } catch (Exception e) {  
 e.printStackTrace();  
 }  
  
 return actualObj;  
 }  
}

**How to use**

**public** **class** NewTest {

**public** **static** **void** main(String[] args) {

String pkgClsName = "com.ddlab.rnd.type1.Person";

Container container = **new** Container();

Person person = (Person) container.getInstantiatedObject(pkgClsName);

person.chargeMobile();

}

}

Note: java.beans.Beans has method **Beans.*instantiate().***

**How to use Dependency Injection using Service Locator**

**interface Service {  
 void doSomething();  
}**

**class ServiceA implements Service { 🡺  
 @Override  
 public void doSomething() {  
 System.*out*.println("ServiceA is doing something");  
 }  
}**

class ServiceLocator {  
 private static final Map<String, Service> *services* = new HashMap<>();  
  
 static {  
 *services*.put("ServiceA", new ServiceA());  
 *services*.put("ServiceB", new ServiceB());  
 }  
  
 public static Service getService(String serviceName) {  
 return *services*.get(serviceName);  
 }  
}

public class Main {  
 public static void main(String[] args) {  
 Service serviceA = ServiceLocator.*getService*("ServiceA");  
 serviceA.doSomething();  
  
 Service serviceB = ServiceLocator.*getService*("ServiceB");  
 serviceB.doSomething();  
 }  
}

**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.