**Inversion of Control (IOC) and Dependency Injection (DI) – 2022**

**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 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.**

# 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.**

**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**

**First create an annotation like this**

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

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

**public** **@interface** **WiringType** {

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

}

**Create an object with that annotation type**

**public** **class** Person {

@WiringType("NokiaPhone")

**private** ICelloPhone phone;

@WiringType("NokiaCable")

**private** IUSBCable cable;

**public** **void** chargeMobile() {

phone.charge(cable);

}

}

**Create A Spring Type Container**

**public** **class** Container {

**public** Object getInstantiatedObject(String pkgClsName) {

Object actualObj = **null**;

**try** {

**actualObj = Beans.*instantiate*(ClassLoader.*getSystemClassLoader*(), pkgClsName);**

Field[] fields = actualObj.getClass().getDeclaredFields();

**for** (Field field : fields) {

field.setAccessible(**true**);

**boolean** flag = field.isAnnotationPresent(WiringType.**class**);

**if** (flag) {

WiringType type = field.getDeclaredAnnotation(WiringType.**class**);

String implClassName = type.value();

String packageName = actualObj.getClass().getPackageName();

String invokeClsName = packageName + "." + implClassName;

Object obj = Beans.*instantiate*(ClassLoader.*getSystemClassLoader*(), invokeClsName);

**field.set(actualObj, obj);**

}

}

} **catch** (ClassNotFoundException | IOException | IllegalArgumentException | IllegalAccessException 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();

}

}

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

**public** **interface** MessagingService {

String getMessageBody();

}

**public** **class** EmailService **implements** MessagingService {

**public** String getMessageBody() {

**return** "email body message";

}

}

**public** **class** SMSService **implements** MessagingService {

**public** String getMessageBody() {

**return** "sms body message";

}

}

**public** **class** **InitialContext** {

**public** Object lookup(String serviceName) {

**if** (serviceName.equalsIgnoreCase("EmailService")) {

**return** **new** EmailService();

} **else** **if** (serviceName.equalsIgnoreCase("SMSService")) {

**return** **new** SMSService();

}

**return** **null**;

}

}

**public** **class** Cache {

**private** Map<String, MessagingService> serviceMap = **new** HashMap<>();

**public** MessagingService getService(String serviceName) {

**return** serviceMap.get(serviceName);

}

**public** **void** addService(String serviceName, MessagingService newService) {

serviceMap.put(serviceName, newService);

}

}

**public** **class** ServiceLocator {

**private** **static** Cache *cache* = **new** Cache();

**public** **static** MessagingService getService(String serviceName) {

MessagingService service = *cache*.getService(serviceName);

**if** (service != **null**) {

**return** service;

}

InitialContext context = **new** InitialContext();

MessagingService service1 = (MessagingService) context.lookup(serviceName);

*cache*.addService(serviceName, service1);

**return** service1;

}

}

**To Test**

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

MessagingService service = ServiceLocator.*getService*("EmailService");

String email = service.getMessageBody();

System.***out***.println(email);

}

**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.