Dependency Injection

# Spring DI

In this article, we will discuss dependency injection. The term “Dependency Injection” has been coined by ***Martin Fowler***.

.

By definition,

dependency injection is a software design pattern that implements inversion of control(IOC) contract for resolving dependencies. A dependency is an object(HAS-A/Composition) that can be used (a service). An injection is the passing of a dependency to a dependent object (a client/Top level) that would use it.

The above Definition is bit abstract when you read the first time but once you understand the logic, memorizing definition will suffice to understand DI.

Prior to taking an example Let clear a fundamental term

HAS-A/Composition: In java If any class say Foo contains a reference to another class says Bar, then we can say following

Foo HAS-A Bar.

Foo and bar have a Composition

Foo dependent on Bar.

Foo is dependent Object bar is a dependency.

Say, A new Shop has been Opened which sells “Peter England” shirts. We will design it following way

Code :

**package** com.example.di;

**public** **class** PeterEngland {

**private** String size;

**private** **double** price;

**private** String brand = "Peter England";

**public** String getSize() {

**return** size;

}

**public** **void** setSize(String size) {

**this**.size = size;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

@Override

**public** String toString() {

**return** "PeterEngland [size=" + size + ", price=" + price + ", brand="

+ brand + "]";

}

}

**package** com.example.di;

**import** java.util.Calendar;

**public** **class** Shop {

Calendar calendar = Calendar.*getInstance*();

PeterEngland peterEnglandShirt = **null**;

**public** Shop()

{

peterEnglandShirt = **new** PeterEngland();

peterEnglandShirt.setPrice(1200);

peterEnglandShirt.setSize("XL");

}

**public** **void** displayShirt()

{

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

}

**public** **void** isShopClosed()

{

**if**(Calendar.***SUNDAY*** == calendar.get(Calendar.***DAY\_OF\_WEEK***))

{

System.***out***.println("Shop is closed Today");

**return**;

}

System.***out***.println("Shop is Open Today");

}

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

Shop shop =**new** Shop();

shop.displayShirt();

shop.isShopClosed();

}

}

Output :

PeterEngland [size=XL, price=1200.0, brand=Peter England]

Shop is Open Today

Here A Shop class has been created and as this shop sells PeterEngland shirts, shirt instance is created when shop class is instantiated.

Although this implementation meets the desired output but it has some serious problem

As a developer, you should always avoid those.

**Problem 1**: Shop class has strongly coupled with PeterEngland instance. They can’t be separated so if we want to test “isShopClosed” method still we need to populate PeterEngland instance.

“isShopClosed” method talking about shop is closed or open, so PeterEngland shirt has no relation with this method but still we have to populate PeterEngland, which is unnecessary.

**Problem 2:** This implementation does not welcome future changes say Shop now sells Other brands like Reebok,Addidas etc. To incorporate this change, we need to refactor the whole Shop class, now it will take brand name and base on brand name shirt instances will be created

So Shop has a direct composition with each of the branded shirt. This design is very rigid and fragile by nature. It violates Open Closed principle also.

Code snippet

**public** **void** displayShirt(String brand)

{

if(“P”.equlasIgnoreCase(brand))

System.***out***.println(new PeterEngland());

if(“A”.equlasIgnoreCase(brand))

System.***out***.println(new Adidas());

}

Diagram:

Diagram :1 Shop depends on all kind of branded shirts.

To solve this problem, we can apply IOC design pattern. IOC stands for Inversion of control.

***Inversion of control is a pattern used for decoupling components and layers in the system. The pattern is implemented through injecting dependencies into a component when it is constructed. These dependences are usually provided as interfaces for further decoupling.***

IOC says that we should design in such a way so that Top level component and lower level component both can depend on an interface. Here the magic starts as now both dependent class(Shop) and implementation class(Adidas,PeterEngland) both depends on an interface,

Dependent object doesn’t know about actual implementation and actual implementation does not know who will use them. So it will be very flexible and welcome changes.

So, Now Shop (Top level component) not has a direct relation with PeterEngland or Adidas(implementation) and vice versa but they can talk each other through an interface.

Both depends on Interface here the controls inverse, previously control is in one-way shop depends on branded shirts now Branded shirts also depends on an interface so direction has been inversed so we call it IOC

**Look at the new diagram**

Diagram 2: Shop and (Adidas,PeterEngland ) depends on shirt interface.

Code:

**package** com.example.di;

**public** **interface** Shirt {

**public** **void** setSize(String size);

**public** **void** setPrice(**double** price);

}

**package** com.example.di;

**public** **class** Adidas **implements** Shirt{

**private** String size;

**private** **double** price;

**private** String brand = "Adidas";

**public** String getSize() {

**return** size;

}

**public** **double** getPrice() {

**return** price;

}

@Override

**public** **void** setSize(String size) {

**this**.size=size;

}

@Override

**public** **void** setPrice(**double** price) {

**this**.price=price;

}

**public** String getBrand() {

**return** brand;

}

@Override

**public** String toString() {

**return** "Adidas [size=" + size + ", price=" + price + ", brand=" + brand

+ "]";

}

}

**package** com.example.di;

**public** **class** PeterEngland **implements** Shirt{

**private** String size;

**private** **double** price;

**private** String brand = "Peter England";

**public** String getSize() {

**return** size;

}

**public** **void** setSize(String size) {

**this**.size = size;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

**public** String getBrand() {

**return** brand;

}

@Override

**public** String toString() {

**return** "PeterEngland [size=" + size + ", price=" + price + ", brand="

+ brand + "]";

}

}

**package** com.example.di;

**import** java.util.Calendar;

**public** **class** Shop {

**private** Calendar calendar = Calendar.*getInstance*();

**private** Shirt shirt;

**public** Shop()

{

}

**public** **void** displayShirt()

{

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

}

**public** **void** isShopClosed()

{

**if**(Calendar.***SUNDAY*** == calendar.get(Calendar.***DAY\_OF\_WEEK***))

{

System.***out***.println("Shop is closed Today");

**return**;

}

System.***out***.println("Shop is Open Today");

}

**public** Shirt getShirt() {

**return** shirt;

}

**public** **void** setShirt(Shirt shirt) {

**this**.shirt = shirt;

}

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

Shop shop =**new** Shop();

Shirt peter = **new** PeterEngland();

peter.setPrice(1200);

peter.setSize("XL");

Shirt adi = **new** Adidas();

adi.setPrice(6000);

adi.setSize("XL");

shop.setShirt(peter);

shop.displayShirt();

shop.setShirt(adi);

shop.displayShirt();

shop.isShopClosed();

}

}

Output:

PeterEngland [size=XL, price=1200.0, brand=Peter England]

Adidas [size=XL, price=6000.0, brand=Adidas]

Shop is Open Today

IOC can solve the problems stated above

1. As Shop has a Shirt interface we can easily put a Mock/Dummy shirt then test it easily.

2. This design welcome change with ease just needs to create a new class which implement Shirt interface.

But still Client has to inject the strategy to Shop component. It is client responsibility to provide actual strategy.

***Here Spring takes one step forward***,

***Spring says Spring container will take care of the dependency, a developer does not have to bother about dependency only concentrate on business logic. We call it Dependency Injection,***

***(Spring container injects dependencies for you).***

Basically by DI, (Dependency Injection,) Spring separate the strategy from implementation,

It maintains the strategy in a XML file (Spring configuration xml), Spring container reads instruction from there and wiring Spring beans together.

***Many developers use IOC and DI term interchangeably but IOC is more abstract it does not tell which part has inversed but DI tells you Dependency control has been inversed.***

**There are two types of DI in Spring**

1. **Constructor Injection**: Through constructor parameter, Spring injects dependency bean.

2. **Setter Injection**: By setter method Spring inject the dependency bean;

**Let do the code,**

**Using Spring Constructor Injection:**

**Spring bean XML**:

File name : ShirtDependency.xml

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>

<bean id=*"adidas"* class=*"com.example.di.Adidas"*>

<property name=*"size"* value=*"XL"*/>

<property name=*"price"* value=*"6000"*/>

</bean>

<bean id=*"peterEngland"* class=*"com.example.di.PeterEngland"*>

<property name=*"size"* value=*"XL"*/>

<property name=*"price"* value=*"1200"*/>

</bean>

<bean id=*"shop"* class=*"com.example.di.Shop"*>

<constructor-arg ref=*"adidas"*/>

</bean>

</beans>

**Code** :

**package** com.example.di;

**public** **interface** Shirt {

**public** **void** setSize(String size);

**public** **void** setPrice(**double** price);

}

**package** com.example.di;

**public** **class** Adidas **implements** Shirt{

**private** String size;

**private** **double** price;

**private** String brand = "Adidas";

**public** String getSize() {

**return** size;

}

**public** **double** getPrice() {

**return** price;

}

@Override

**public** **void** setSize(String size) {

**this**.size=size;

}

@Override

**public** **void** setPrice(**double** price) {

**this**.price=price;

}

**public** String getBrand() {

**return** brand;

}

@Override

**public** String toString() {

**return** "Adidas [size=" + size + ", price=" + price + ", brand=" + brand

+ "]";

}

}

**package** com.example.di;

**public** **class** PeterEngland **implements** Shirt{

**private** String size;

**private** **double** price;

**private** String brand = "Peter England";

**public** String getSize() {

**return** size;

}

**public** **void** setSize(String size) {

**this**.size = size;

}

**public** **double** getPrice() {

**return** price;

}

**public** **void** setPrice(**double** price) {

**this**.price = price;

}

**public** String getBrand() {

**return** brand;

}

@Override

**public** String toString() {

**return** "PeterEngland [size=" + size + ", price=" + price + ", brand="

+ brand + "]";

}

}

**package** com.example.di;

**import** java.util.Calendar;

**public** **class** Shop {

**private** Calendar calendar = Calendar.*getInstance*();

**private** Shirt shirt;

**public** Shop(Shirt shirt)

{

**this**.shirt=shirt;

}

**public** **void** displayShirt()

{

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

}

**public** **void** isShopClosed()

{

**if**(Calendar.***SUNDAY*** == calendar.get(Calendar.***DAY\_OF\_WEEK***))

{

System.***out***.println("Shop is closed Today");

**return**;

}

System.***out***.println("Shop is Open Today");

}

}

**package** com.example.di;

**import** org.springframework.context.ApplicationContext;

**import** org.springframework.context.support.ClassPathXmlApplicationContext;

**public** **class** SpringManager {

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

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("configFiles/ShirtDependency.xml");

Shop shop =(Shop) ctx.getBean("shop");

shop.displayShirt();

shop.isShopClosed();

}

}

Output :

Adidas [size=XL, price=6000.0, brand=Adidas]

Shop is Open Today

**By** **Setter Injection:**

All previous class should be same just need to change Shop class and ShirtDependency.xml

**Changed, ShirtDependency.xml**

<?xml version=*"1.0"* encoding=*"UTF-8"*?>

<beans xmlns=*"http://www.springframework.org/schema/beans"*

xmlns:xsi=*"http://www.w3.org/2001/XMLSchema-instance"*

xsi:schemaLocation=*"http://www.springframework.org/schema/beans*

*http://www.springframework.org/schema/beans/spring-beans-3.0.xsd"*>

<bean id=*"adidas"* class=*"com.example.di.Adidas"*>

<property name=*"size"* value=*"XL"*/>

<property name=*"price"* value=*"6000"*/>

</bean>

<bean id=*"peterEngland"* class=*"com.example.di.PeterEngland"*>

<property name=*"size"* value=*"XL"*/>

<property name=*"price"* value=*"1200"*/>

</bean>

<bean id=*"shop"* class=*"com.example.di.Shop"*>

<property name=*"shirt"* ref=*"peterEngland"*/>

</bean>

</beans>

**Changed, Shop class**

**package** com.example.di;

**import** java.util.Calendar;

**public** **class** Shop {

**private** Calendar calendar = Calendar.*getInstance*();

**private** Shirt shirt;

**public** **void** displayShirt()

{

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

}

**public** **void** isShopClosed()

{

**if**(Calendar.***SUNDAY*** == calendar.get(Calendar.***DAY\_OF\_WEEK***))

{

System.***out***.println("Shop is closed Today");

**return**;

}

System.***out***.println("Shop is Open Today");

}

**public** Shirt getShirt() {

**return** shirt;

}

**public** **void** setShirt(Shirt shirt) {

**this**.shirt = shirt;

}

}

Output :

PeterEngland [size=XL, price=1200.0, brand=Peter England]

Shop is Open Today

**Look above example carefully, in ShirtDependect.xml I declare all the bean definition with unique id.**

**Like,**

<bean id=*"shop"* class=*"com.example.di.Shop"*>

<property name=*"shirt"* ref=*"peterEngland"*/>

</bean>

In the case of Constructor injection I use <constructor-arg ref=*"adidas"*/> by this Spring container pass Adidas bean to Shop constructor as ref property is point to Adidas bean ‘s id.

Same in Setter injection I declare <property name=*"shirt"* ref=*"peterEngland"*/> by this Spring container invoke SetShirt() method (setter of the property) and injects PerterEngland bean.

Please note that in case of Setter injection always provide Setter method of the property unless Spring will throw an exception.