**IOC Container :**

The IoC container is responsible to instantiate, configure and assemble the objects. The IoC container gets informations from the XML file and works accordingly. The main tasks performed by IoC container are:

* to instantiate the application class
* to configure the object
* to assemble the dependencies between the objects

There are two types of IoC containers. They are:

1. **BeanFactory**
2. **ApplicationContext**

**Dependency Injection in Spring:**

The Dependency Injection is a design pattern that removes the dependency of the programs. In such case we provide the information from the external source such as XML file. It makes our code loosely coupled and easier for testing.

Spring framework provides two ways to inject dependency

* By Constructor
* By Setter method

# Dependency Injection by Constructor

# We can inject the dependency by constructor. The ****<constructor-arg>**** subelement of ****<bean>**** is used for constructor injection

# For Example :

# If there is HAS-A relationship between the classes, we create the instance of dependent object (contained object) first then pass it as an argument of the main class constructor. Here, our scenario is Employee HAS-A Address.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. ------
4. <bean id="a1" **class**="com.javatpoint.Address">
5. <constructor-arg value="ghaziabad"></constructor-arg>
6. <constructor-arg value="UP"></constructor-arg>
7. <constructor-arg value="India"></constructor-arg>
8. </bean>

11. <bean id="e" **class**="com.javatpoint.Employee">
12. **<constructor-arg value="12" type="int"></constructor-arg>**
13. **<constructor-arg value="Sonoo"></constructor-arg>**
14. **<constructor-arg>**
15. <ref bean="a1"/>
16. </constructor-arg>
17. </bean>
19. </beans>

# Setter Injection :

# Like Constructor Injection, we can inject the dependency of another bean using setters. In such case, we use property element. Here, our scenario is Employee HAS-A Address. The ****ref**** attribute of property elements is used to define the reference of another bean.

# For Example:

# **applicationContext.xml**

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans
3. xmlns="http://www.springframework.org/schema/beans"
4. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
5. xmlns:p="http://www.springframework.org/schema/p"
6. xsi:schemaLocation="http://www.springframework.org/schema/beans
7. http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">
9. <bean id="address1" **class**="com.javatpoint.Address">
10. <**property name="addressLine1" value="51,Lohianagar"></property>**
11. **<property name="city" value="Ghaziabad"></property>**
12. **<property name="state" value="UP"></property>**
13. **<property name="country" value="India"></property>**
14. </bean>
16. <bean id="obj" **class**="com.javatpoint.Employee">
17. **<property name="id" value="1"></property>**
18. **<property name="name" value="Sachin Yadav"></property>**
19. **<property name="address" ref="address1"></property>**
20. </bean>
22. </beans>

My Github link of working example on DI.

# <https://github.com/anupamkaushal/randomprojects/tree/master/SpringProject/src/com/spring/di>

# Spring AOP :

**Aspect Oriented Programming** (AOP) compliments OOPs in the sense that it also provides modularity. But the key unit of modularity is aspect than class. A **cross-cutting concern** is a concern that can affect the whole application and should be centralized in one location in code as possible, such as transaction management, authentication, logging, security etc.

AOP breaks the program logic into distinct parts (called concerns). It is used to increase modularity by **cross-cutting concerns**.

#### Why use AOP?

It provides the pluggable way to dynamically add the additional concern before, after or around the actual logic

Suppose there are 10 methods in a class as given below:

1. **class** A{
2. **public** **void** m1(){...}
3. **public** **void** m2(){...}
4. **public** **void** m3(){...}
5. **public** **void** m4(){...}
6. **public** **void** m5(){...}
7. **public** **void** n1(){...}
8. **public** **void** n2(){...}
9. **public** **void** p1(){...}
10. **public** **void** p2(){...}
11. **public** **void** p3(){...}
12. }

There are 5 methods that starts from m, 2 methods that starts from n and 3 methods that starts from p.

**Understanding Scenario** I have to maintain log and send notification after calling methods that starts from m.

**Problem without AOP** We can call methods (that maintains log and sends notification) from the methods starting with m. In such scenario, we need to write the code in all the 5 methods.

But, if client says in future, I don't have to send notification, you need to change all the methods. It leads to the maintenance problem.

**Solution with AOP** We don't have to call methods from the method. Now we can define the additional concern like maintaining log, sending notification etc. in the method of a class. Its entry is given in the xml file.

In future, if client says to remove the notifier functionality, we need to change only in the xml file. So, maintenance is easy in AOP

## AOP Concepts and Terminology

AOP concepts and terminologies are as follows:

#### Join point

Join point is any point in your program such as method execution, exception handling, field access etc. Spring supports only method execution join point.

#### Advice

Advice represents an action taken by an aspect at a particular join point. There are different types of advices:

* **Before Advice**: it executes before a join point.
* **After Returning Advice**: it executes after a joint point completes normally.
* **After Throwing Advice**: it executes if method exits by throwing an exception.
* **After (finally) Advice**: it executes after a join point regardless of join point exit whether normally or exceptional return.
* **Around Advice**: It executes before and after a join point.

#### Pointcut

It is an expression language of AOP that matches join points.

#### Aspect

It is a class that contains advices, joinpoints etc.

**AOP implentation by Spring AspectJ Annotation :**

My Github link of working example AOP.

# <https://github.com/anupamkaushal/randomprojects/tree/master/SpringProject/src/com/spring/aop>

# Flowable Framework :

# The Flowable project provides a core set of open source business process engines that are compact and highly efficient. They provide a workflow and Business Process Management (BPM) platform for developers, system admins and business users.

At its heart is a lightning fast, tried and tested dynamic BPMN process engine, with accompanying DMN decision tables and CMMN case management engines, all written in Java. They are Apache 2.0 licensed open source, with a committed community.

All the engines can run embedded in a Java application, or as a service on a server, a cluster, and in the cloud. They can run as independent engines or services, or integrate together to provide a rich suite for business process management. They integrate perfectly with Spring. With rich Java and REST APIs.

# My GitHub Link for Flowable example :

# <https://github.com/anupamkaushal/randomprojects/tree/master/flowable>

# Above example shows if user requests for holiday what are the possible scenerio.For example manager can reject or accept his/her the holiday request.All these steps can be configured in bpmn.xml file

# Ref Link: http://www.flowable.org/

**An introduction to Camel :**

**What is Camel :**

Apache Camel is an open source integration framework that aims to make

integrating systems easier. Camel provides simple, manageable abstractions for the complex systems you’re integrating . At the core of the Camel framework is a routing engine, or more precisely a routingengine

builder. It allows you to define your own routing rules, decide from which sources to accept messages, and determine how to process and send those messages to other destinations. Camel uses an integration language that allows you to define complex routing rules, akin to business processes. One of the fundamental principles of Camel is that it makes no assumptions about the type of data you need to process. This is an important point, because it gives you, the developer, an opportunity to integrate any kind of system, without the need to

convert your data to a canonical format.

Camel offers higher-level abstractions that allow you to interact with various systems using the same API regardless of the protocol or data type the systems are using.Components in Camel provide specific implementations of the API that target different protocols and data types. Camel comes with support for over 80 protocols and data types. Its extensible and modular architecture allows you to implement and seamlessly plug in support for your own protocols. These architectural choices eliminate the need for unnecessary conversions and make Camel not only faster but also very lean.

Camel doesn’t have container or a reliable message bus, but it can be deployed in one .

**Why Use Camel :**

Camel introduces a few novel ideas into the integration space. Please see below

■ Routing and mediation engine

■ Enterprise integration patterns (EIPs)

■ Domain-specific language (DSL)

■ Extensive component library

■ Payload-agnostic router

■ Modular and pluggable architecture

■ POJO model

■ Easy configuration

■ Automatic type converters

■ Lightweight core

■ Test kit ■ Vibrant community

**Routing and mediation engine :**

The core feature of Camel is its routing and mediation engine. A routing engine will

selectively move a message around, based on the route’s configuration.

**DOMAIN-SPECIFIC LANGUAGE (DSL) :**

Camel’s domain-specific language (DSL) is a major contribution to the integration

space. A few other integration frameworks currently feature a DSL (and some allow

you to use XML to describe routing rules), but unlike Camel their DSLs are based on

custom languages.Camel is unique because it offers multiple DSLs in regular programming

languages such as Java, Scala, Groovy, and it also allows routing rules to be

specified in XML. The purpose of the DSL is to allow the developer to focus on the integration problem

rather than on the tool.

Here are some examples of the DSL using different languages and staying functionally

equivalent:

■ Java DSL

from("file:data/inbox").to("jms:queue:order");

■ Spring DSL

<route>

<from uri="file:data/inbox"/>

<to uri="jms:queue:order"/>

**Routing With Camel**

Router’s function is to selectively move the message forward. In the context of enterprise messaging systems, routing is the process by which a message is taken from an input queue and, based on a set of conditions, sent to one

of several output queues.

In Apache Camel, routing is a more general concept. It’s defined as a step-by-step

movement of the message, which originates from an endpoint in the role of a consumer.

The consumer could be receiving the message from an external service, polling

for the message on some system, or even creating the message itself. This message

then flows through a processing component, which could be an enterprise integration

pattern (EIP), a processor, an interceptor, or some other custom creation. The message

is finally sent to a target endpoint . Camel use URI to communicate over

FTP and JMS. Through URI,You can decide to either send messages to the component configured by

this URI, or to consume messages from it. A Camel endpoint URI consists of three parts: a scheme, a context

path, and a list of options.

***Sending to a JMS queue :*** Camel provides extensive support for connecting to JMS-enabled providers.In JMS, message consumers and producers talk to one another through an intermediary a JMS destination. JMS also provides a ConnectionFactory that clients (like Camel) can use to create a connection with a JMS provider.

There are two types of JMS destinations: queues and topics.

**CONFIGURE CAMEL TO USE A JMS PROVIDER :**

To connect Camel to a specific JMS provider, you need to configure Camel’s JMS component

with an appropriate ConnectionFactory. Apache ActiveMQ is one of the most popular open source JMS providers.

You can create an ActiveMQConnectionFactory that points to the location of the running ActiveMQ broker:

ConnectionFactory **connectionFactory** = new ActiveMQConnectionFactory("vm://localhost");

create your CamelContext, you can add the JMS component as follows:

CamelContext context = new DefaultCamelContext();

context.addComponent("jms",JmsComponent.jmsComponentAutoAcknowledge(**connectionFactory**));

Needed Jar File: camel-core,camel-jms, activemq-core

Now that you’ve configured the JMS component to connect to an actual JMS broker,

it’s time to look at how URIs can be used to specify the destination.

**USING URIS TO SPECIFY THE DESTINATION**

jms:queue:incomingOrders

Using Camel’s Java DSL, you can send a message to the incomingOrders queue by

using the to keyword like this:

.to("jms:queue:incomingOrders")

***Creating routes in Java:***

RouteBuilders are used to create routes in Camel. Each RouteBuilder can create multiple routes.

CamelContext context = new DefaultCamelContext();

context.addRoutes(new RouteBuilder() {

public void configure() throws Exception {

...

}

});

Within the configure method, you define your routes using the Java DSL.

import javax.jms.ConnectionFactory;

import org.apache.activemq.ActiveMQConnectionFactory;

import org.apache.camel.CamelContext;

import org.apache.camel.builder.RouteBuilder;

import org.apache.camel.component.jms.JmsComponent;

import org.apache.camel.impl.DefaultCamelContext;

public class FtpToJMSExample {

public static void main(String args[]) throws Exception {

CamelContext context = new DefaultCamelContext();

ConnectionFactory connectionFactory =

new ActiveMQConnectionFactory("vm://localhost");

context.addComponent("jms",

JmsComponent.jmsComponentAutoAcknowledge(connectionFactory));

context.addRoutes(new RouteBuilder() {

public void configure() {

from("ftp://rider.com/orders"

+ "?username=rider&password=secret")

process(new Processor() {

public void process(Exchange exchange) throws Exception {

System.out.println("We just downloaded: "

+ exchange.getIn().getHeader("CamelFileName"));

}

})

.to("jms:incomingOrders");

}

});

context.start();

Thread.sleep(10000);

context.stop();

}

}

**Creating routes with Spring :**

Spring is the most popular Inversion of Control (IoC) Java container out there. The

core framework allows to you “wire” beans together to form applications. This wiring

is done through an XML configuration file.

<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

http://camel.apache.org/schema/spring

http://camel.apache.org/schema/spring/camel-spring.xsd">

<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">

<property name="connectionFactory">

<bean class="org.apache.activemq.ActiveMQConnectionFactory">

<property name="brokerURL" value="vm://localhost" />

</bean>

</property>

</bean>

<bean id="ftpToJmsRoute" class="camelinaction.FtpToJMSRoute"/>

<camelContext xmlns="http://camel.apache.org/schema/spring">

<routeBuilder ref="ftpToJmsRoute"/>

</camelContext>

</beans>

**<camelContext** xmlns="http://camel.apache.org/schema/spring">

This will automatically start a SpringCamelContext, which is a subclass of the

DefaultCamelContext you used for the Java DSL

public class FtpToJMSRoute extends RouteBuilder {

public void configure() {

from("ftp://rider.com" +

"/orders?username=rider&password=secret")

.to("jms:incomingOrders");

}

}

**USING SPRING DSL:**

<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

http://camel.apache.org/schema/spring

http://camel.apache.org/schema/spring/camel-spring.xsd">

<bean id="jms" class="org.apache.camel.component.jms.JmsComponent">

<property name="connectionFactory">

<bean class="org.apache.activemq.ActiveMQConnectionFactory">

<property name="brokerURL" value="vm://localhost" />

</bean>

</property>

</bean>

<camelContext xmlns="http://camel.apache.org/schema/spring">

<route>

<from

uri="ftp://rider.com/orders?username=rider&password=secret"/>

<to uri="jms:incomingOrders"/>

</route>

</camelContext>

</beans>

Reference link :

http:// manning.com/ibsen

http://code.google.com/p/camelinaction.

# What are Microservices:

# Microservices is a form of service-oriented architecture wherein applications are built as a collection of different smaller services rather than one whole app. Instead of a monolithic app, you have several independent applications that can run on their own and may be created using different coding or programming languages. Big and complicated applications can be made up of simpler and independent programs that are executable by themselves. These smaller programs are grouped together to deliver all the functionalities of the big, monolithic app.

# Each Micro service runs a unique process and communicates through a well-defined, lightweight mechanism to serve a business goal. Each micro service can be a Rest Service which can be implemented through common libararies like Jersey, RestEasy or Spring Rest.

# (2) Jersey : This open source framework supports JAX-RS APIs in Java is very easy to use.

# Github Link for creating Rest APIs through Spring Rest.

# <https://github.com/anupamkaushal/randomprojects/tree/master/farcar/src>