Understanding POJO and Java Bean:

--------------------------------

POJO – Plain Old Java Object

* POJO and Java Bean are the Java classes.
* POJO class is a normal java class created without any special requirements. Just it should follow Java naming conventions.
* A POJO class does not extend any parent class, does not implement any interface and does not use any annotations.
* A POJO class can have,

1. variables/fields
2. constructors
3. gettes and setters(optional)
4. other methods(optional)

ex:

public class Employee



{

int empid;

String ename;

public Employee(int empid, String ename)

{

this.empid = empid;

this.ename = ename;

}

public void show()

{

//logic

}

}

Java Bean : It is a Java class created with special requirements.

The requirements are,

1. variables must be private
2. no-argument constructor is mandatory
3. setter and getter methods for the variables is mandatory
4. class must implement java.io.Serializable interface.

ex:

public class Customer implements Serializable {

private long customerId;

private String customerName;

public Customer() {

customerId=101;

customerName = “John”;

}

//Getters and Setters

}

Note: Actually a pojo class does not implement any interface. But one exception is that, it can implement Seriliazable interface.

Q) what is the difference between POJO and Java Bean class?

A) 1. POJO is a class created without any special requirements.

Java Bean is a class created with special requirements.

2. In POJO class, variables can be

private/protected/public/default.

In Java Bean class, variables must be private

3. In POJO class, no-argument constructor is not mandatory.

In Java Bean class, no-argument constructor is mandatory.

4. In POJO class, Getters and Setters are optional.

In Java Bean class, Getters and Setter are mandatory.

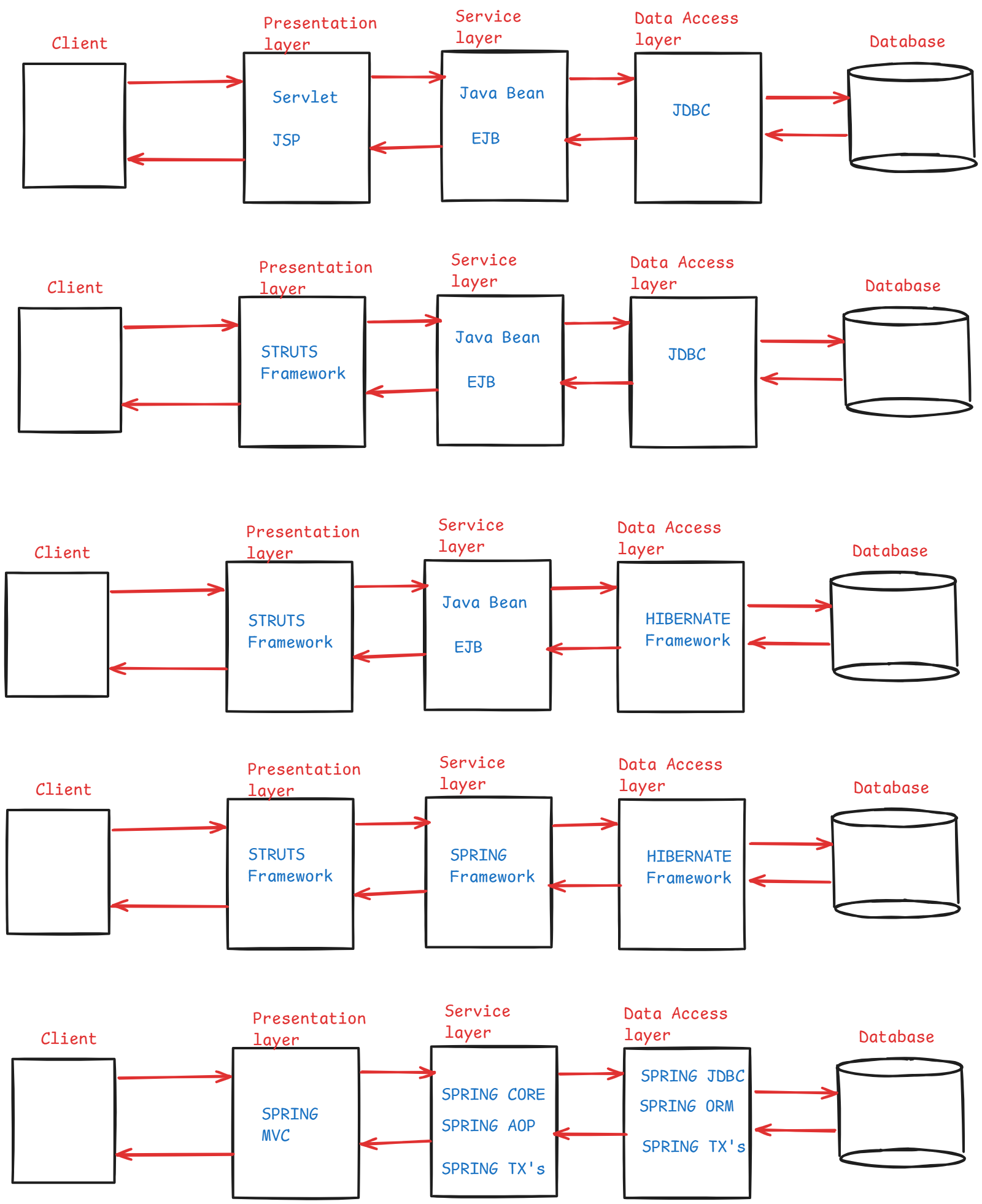
Note: All Java Bean classes are POJO classes, but all POJO classes are not the Java Bean classes.

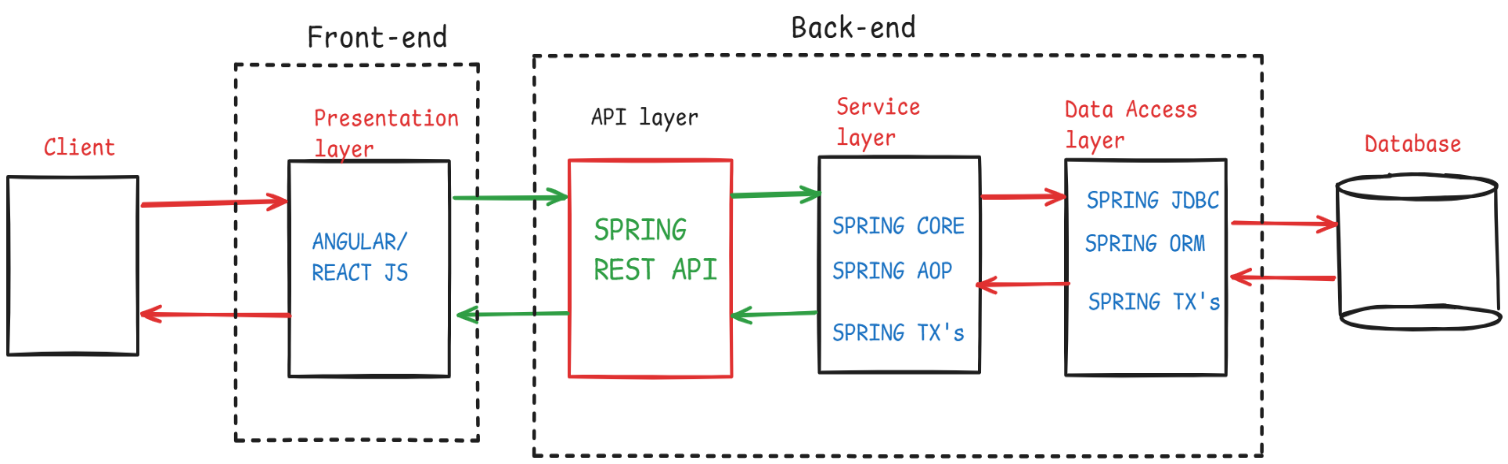
why should we use frameworks to develop the applications?

* While developing the software projects, the projects contain some common functionalities and also some business functionalities.
* The business functionalities are different from one project to another project.
* If we develop the software projects, without using frameworks then you have to write the code for implementing common functionalities and also for implementing the business functionalities.
* By using the frameworks, if you develop then, you have to put focus only on business functionalities, because the framework will automatically provides the common functionalities.
* So, by using frameworks if you develop the applications, then you can complete them and deliver them faster to the clients.
* some examples of the common functionalities are,

1. Databinding and validations
2. Database connectivity
3. Security management
4. Caching
5. Dependency injection
6. Transaction managment

Software Application Architecture:





Spring Core module

IS-A and HAS-A:

--------------

* When you create a child class by extending a parent class then that relationship is called IS-A relationship.

ex:

public class Product {

//variables

//methods

}

public class Mobile extends Product {

//variables

//methods

}

* When you create one class object in another class as a data member then that relationship is called HAS-A relationship.

ex:

public class User { public class PasswordEncoder {

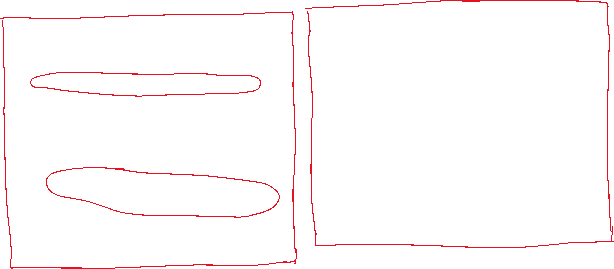
private String username; void encode(String password)

private String password; {

PasswordEncoder encoder; //logic

public User(String username, }

String password) { void decode(String password)



this.username=username; {

this.password=password; //logic

encoder =

new PasswordEncoder(); }

} }

//other methods

}

* Here User object has HAS-A relationship with PasswordEncoder object.
* One object can have HAS-A relationship with multiple objects also.

ex: public class EmailNotification {

//methods

}

public class SMSNotification {

//methods

}

public class NotificationManager

{

private String emailid;

private long mobile;

EmailNotification email = new EmailNotification();

SMSNotification sms = new SMSNotification();

//other methods

}

* Here, NotificationManager object has HAS-A relationship with EmailNotification and SMSNotification objects.

who is dependent class and who is dependency class?

. Suppose, we have two classes called A and B, and if you create class B object inside class A, then class A is called the dependent class and class B is called the dependency class.

public class A { public class B {

B b = new B(); //methods

} }

. Suppose, we have two classes called PaymentController and PaymentService, and if you create PaymentService class object inside the PaymentController class, then PaymentController class is called dependent class and PaymentService class is called as the dependency class.

public class PaymentController { public class PaymentService {

PaymentSerice service = //methods

new PaymentService(); }

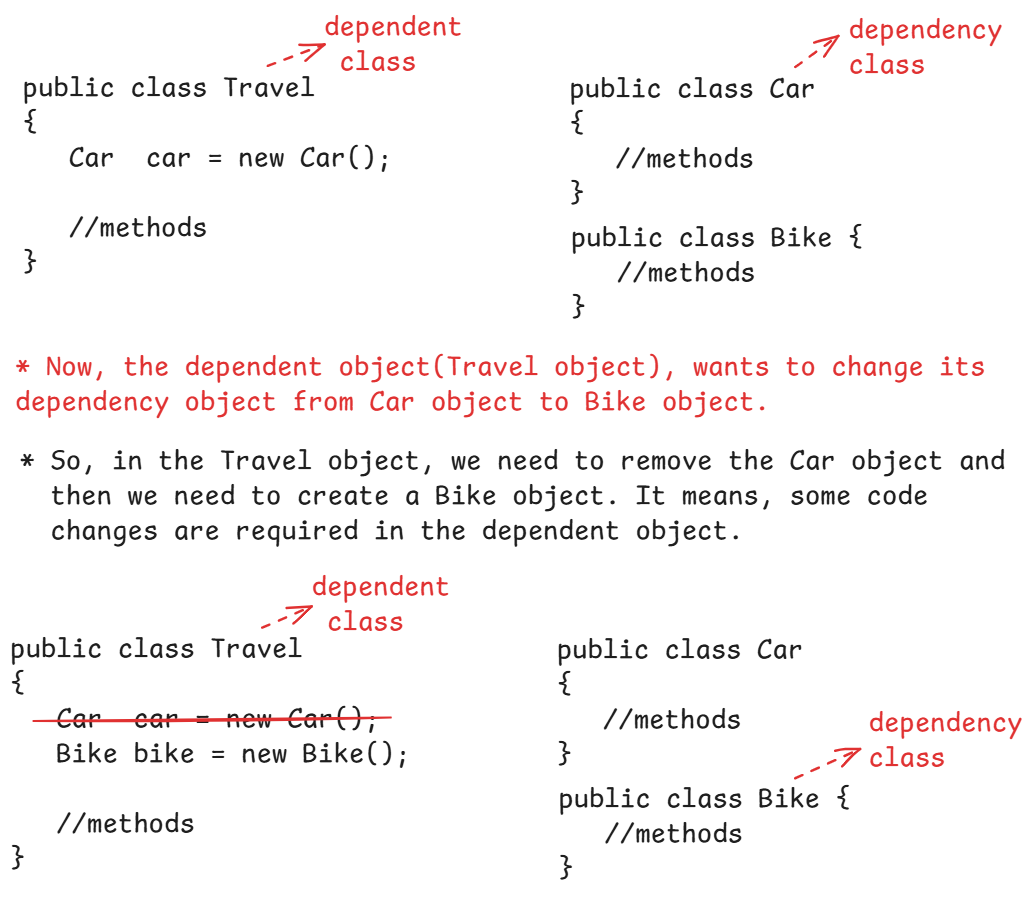
}

tightly coupled objects?

. When a dependent object wants to change its dependency object, if code changes are required in the dependent object, then we say that the dependent object is tightly coupled with the dependency object.

* When a dependent class is creating its dependency class object, inside the class itself, then the objects are tightly coupled.

Example:



loosely coupled objects?

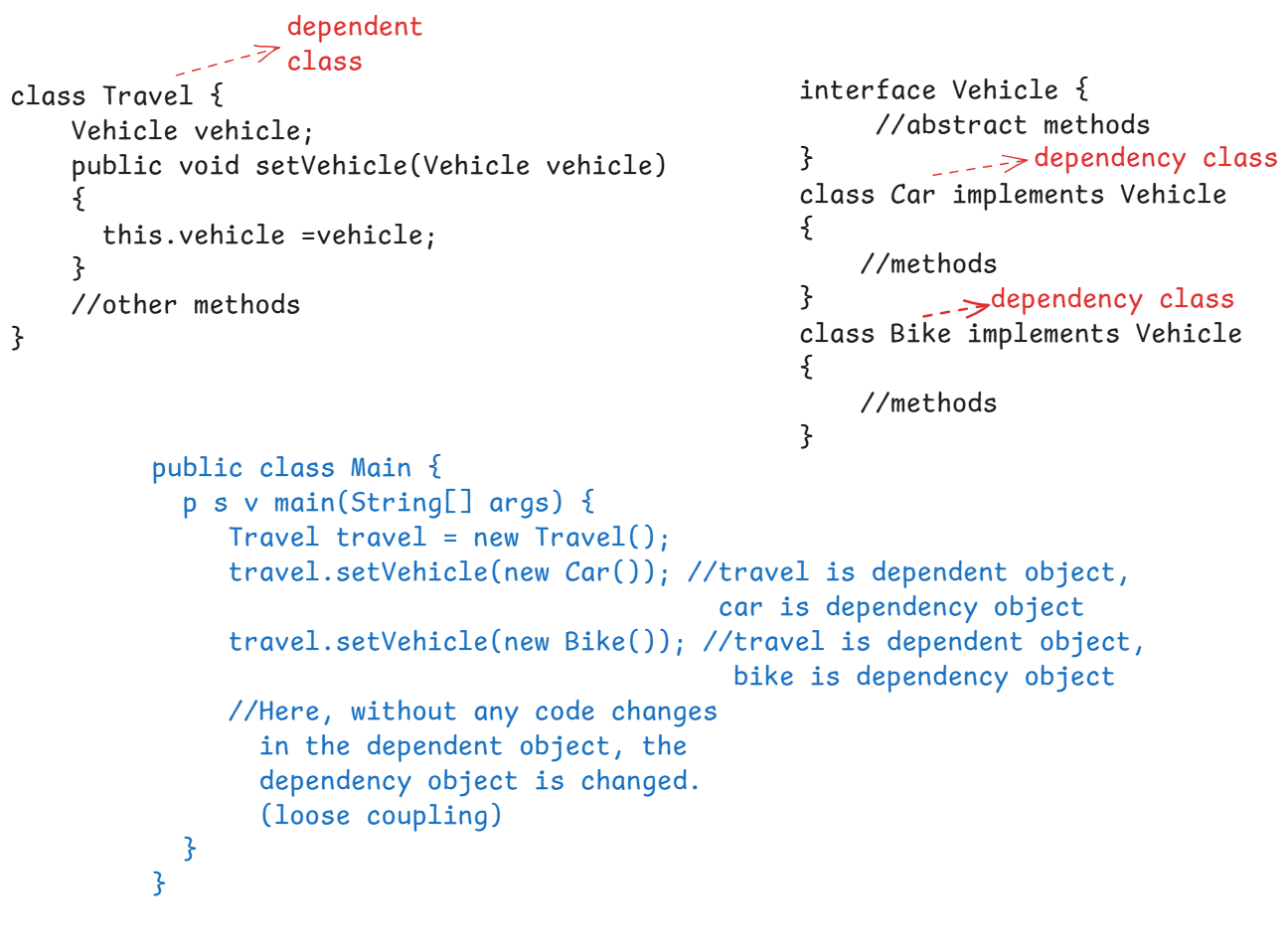
. When a dependent object wants to change its dependency object, if no code changes are required in the dependent object, then we say that the dependent object is loosely coupled with the dependency object.

. The rules to follow to provide loose coupling b/w the objects are,

1. The dependency classes must implement an interface.
2. In the dependent class, create interface reference variable

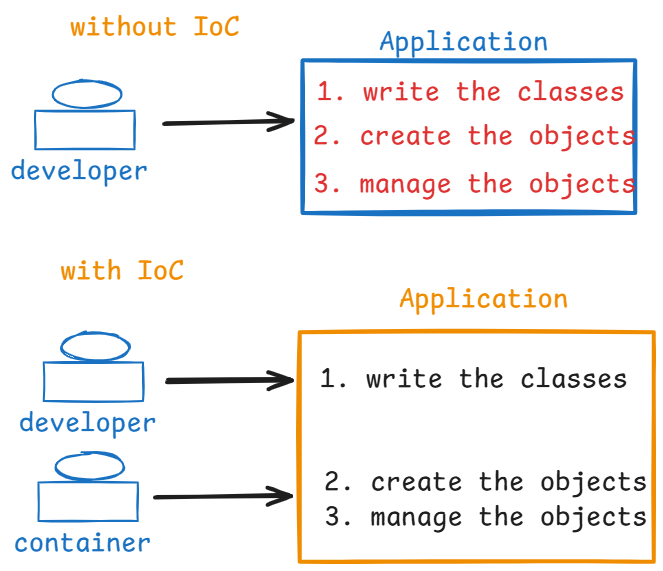
as a data member;

1. In the dependent class, define a constructor or a setter method for injecting the dependency object.



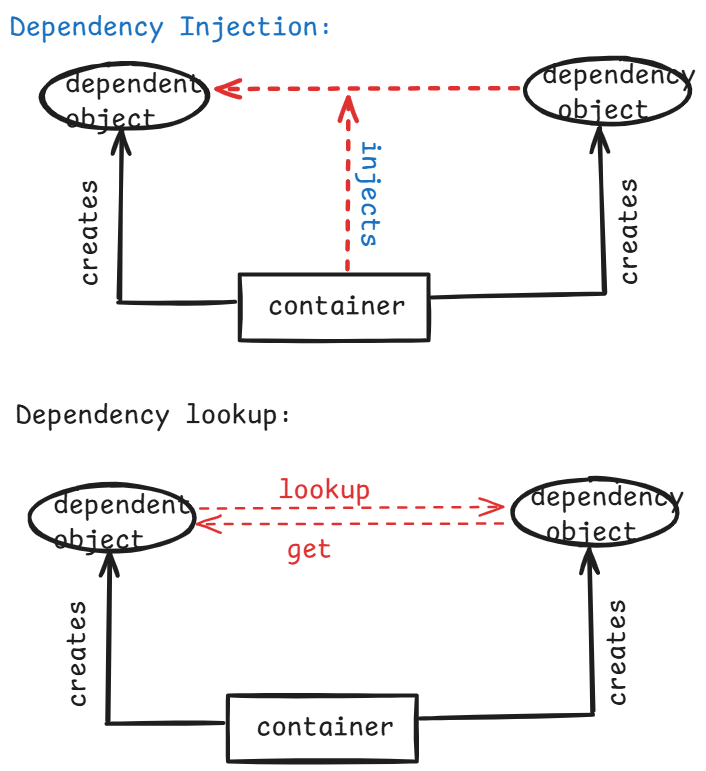
What is Inversion of Control(IoC)?

* Inversion of Control is a design principle, where the responsibilities like creating the objects and managing the objects are inverted from the programmer to the framework or a container.
* Here, the programmer is only responsible for developing the classes with the required functionalities. The objects creation and managing them will be taken care by a container.



What is Dependency Injection and Dependency Lookup?

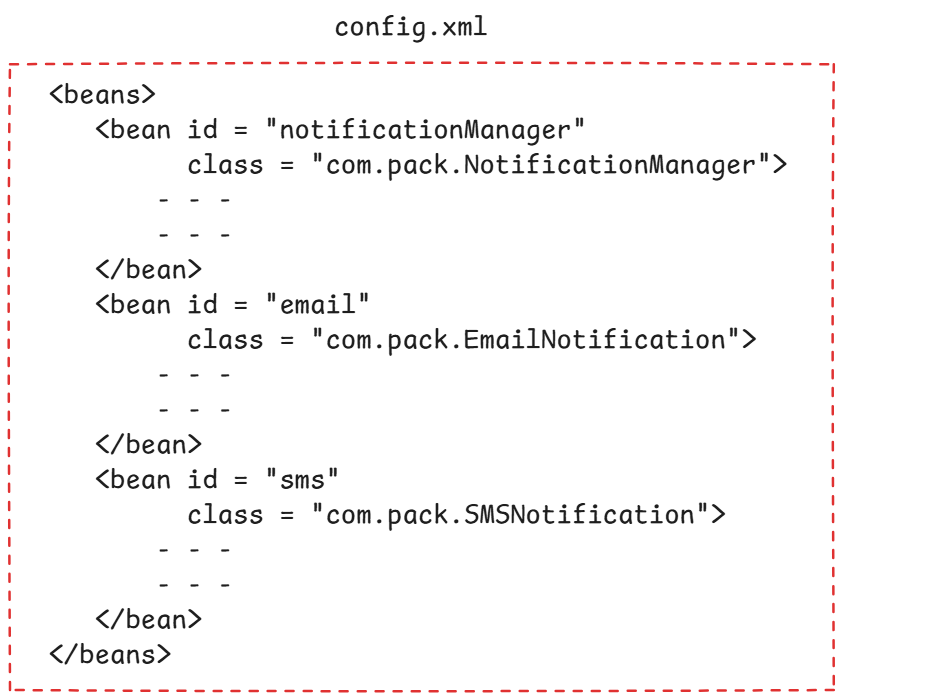
* Dependency Injection and Dependency Lookup are the two mechanisms to implement IoC design principle. So, we call them as design patterns.
* In Dependency Injection, the container/framework creates the dependent object and dependency object. Also, injects the dependency object to the dependent object.
* In Dependency Lookup, the container/framework creates the dependent object and dependency object. But, it doesn’t inject the dependency object to the dependent object. So, the dependent object only will serach/look for its dependency object and it will get it.
* In Spring, the container is implemented with Dependency Injection design pattern.



Spring Configuration file:

-------------------------

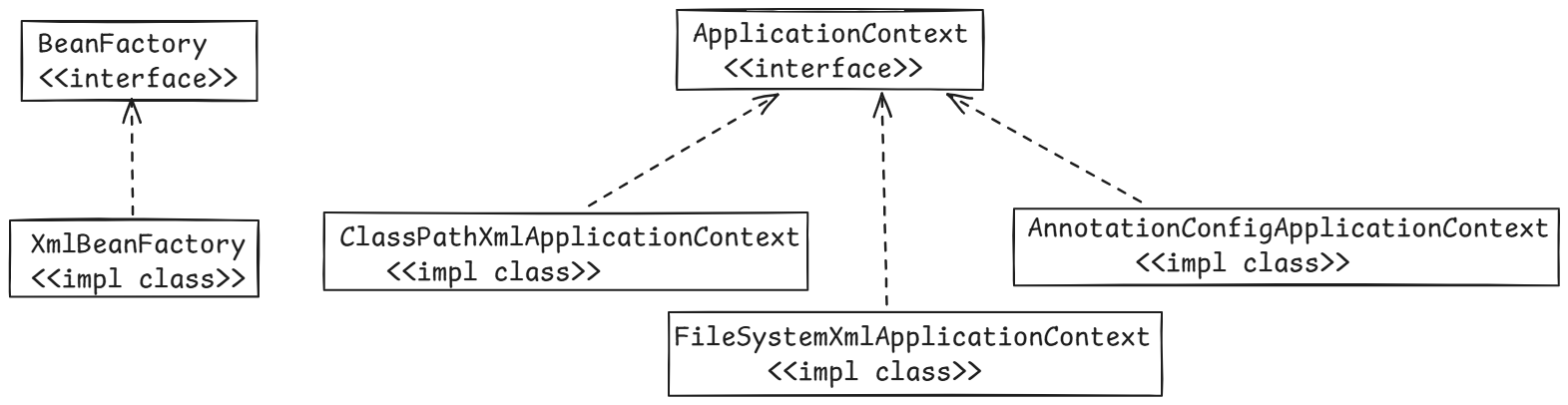
* As a programmer, we need to provide the necessary details to the spring container, to create the objects and manage the objects.
* The details can be provided to the container by creating an xml file. This is called a configuration file.
* The xml file name can be <anyname>.xml
* The root element of this xml file is <beans>, and parent tag in the xml is, <bean>.
* <bean> tag has two important attributes called id and class.
* We can define one or more configuration files in spring.
* A sample spring configuration file looks like below.



Types of Spring containers:

---------------------------

* A spring container is a class, which implements one of the below two interfaces. So, we say that there are 2 types of spring containers.
* 1. BeanFactory interface
* 2. ApplicationContext interface



* A spring container can be started by creating an object of a container class.
* The container class is nothing but it is an implementation class of either BeanFactory interface or ApplicationContext interface.

For example:

BeanFactory factory = new XmlBeanFactory(“config.xml”);

(or)

ApplicationContext context = new ClassPathXmlApplicationContext(“config.xml”);

Note:

1. BeanFactory container is a basic spring container which only supports xml-based configurations.
2. ApplicationContext container is an extension of BeanFactory container, which supports both xml-based configurations and also annotation-based configurations.
3. ApplicationContext is a child interface of BeanFactory.

Maven

=====

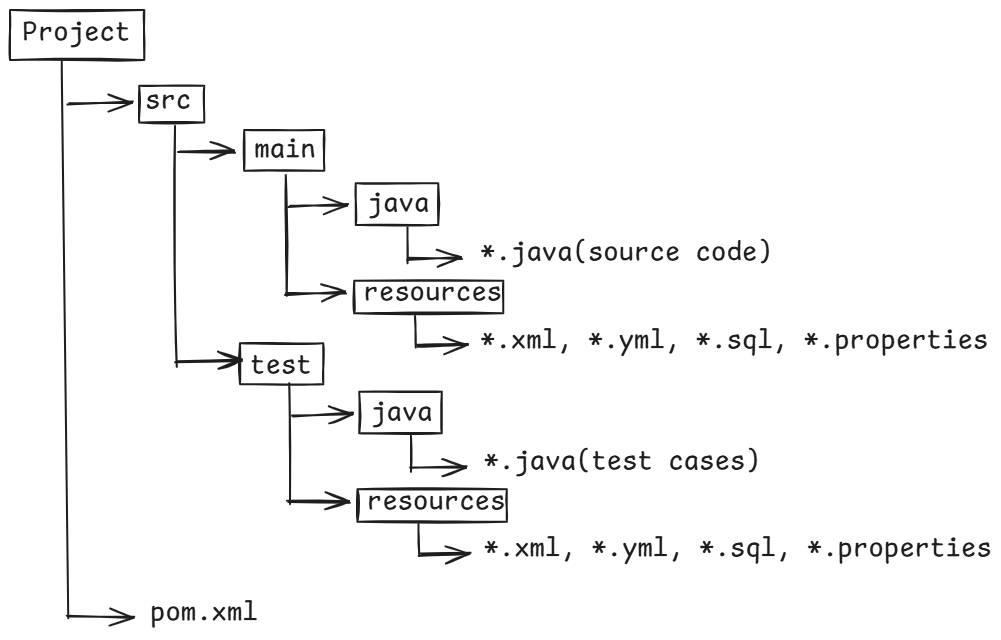
BuildProcess?

* Build process means, converting the code written by the programmer into an executable application.
* The steps involved in this process are,

1. download/fetch the jars to compile the code from internet.
2. add the jars to the classpath
3. compile the source code
4. download/fetch the jars to compile the test code from internet.
5. add these jars to the classpath
6. compile and execute the test cases.
7. package the application into a jar/war file for execution.

* After the application jar/war file is deployed into a server, if any issue/bug is identified then the developer has to fix the issue/bug and again has to perform the above steps.
* To automate this build process steps, we use a build tool.
* Maven and Gradle are the build tools used in Java applications and these are provided by Apache.

Maven project structure:



pom.xml 🡪 project object model

* In pom.xml file, we have to configure the project dependencies(jars).
* To configure a dependency, we have to use <dependency> tag.
* Under <dependency> tag, we have to write three child tags.

<groupId> -- a unique id of an organization

<artifactId> -- name of the jar file

<version> -- version of the jar file

ex:

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>5.2.2</version>

</dependency>

maven repositories:

------------------

1. central repository
2. local repository

* Maven maintains all the dependencies/libraries required for developing Java projects with different technologies, tools and frameworks with different versions in a place called central repository.
* The url of maven central repository is

<https://repo.maven.apache.org/maven2>

* When you are creating a new maven project from your IDE,

maven will download the required plugins and dependencies from central repository.

* The downloaded files will be stored into a local repository.
* In every developer’s machine, maven local repository will be created.
* The location of maven local repository is,

C:\Users\Administrator\.m2\repository

* Before downloading any dependencies from central repository, Maven will first check for them in the local repository.
* If exists in local repository, then Maven will reuse them from the local repository.
* If not exists, then Maven will download from the central repository, stores them into local repository for use.

Maven archetypes:

----------------

* Maven archetypes are project templates, which helps the developers to quickly setup the project structure by without creating the folders manually.
* If developers setup the project structure manually, then it will take time and also errors may occur, if the folders are not created properly.
* That’s why maven has provided archetypes.
* Maven has provided many archetypes to create project structures for different requirements.
* The two important archetypes are,

maven-archetype-quickstart --- used for creating console java applications.

maven-archetype-webapp --- used for creating web applications.

Maven commands:

------------

* To build a project, we can execute maven commands either from the IDE or from Command Prompt.
* To execute the commands in IDE, we have to right click on the project -🡪 RunAs 🡪 maven build…
* To execute from the command prompt, download the maven software(zip) file from maven.apache.org and extract the file.
* add bin folder of maven to the path variable.
* To set the maven path,

1. on windows search, type environment and select edit system environment variable
2. click on Environment variables button
3. choose path variable then click edit
4. click new button and copy and paste the maven’s bin path

(C:\Users\WINDOWS\Downloads\apache-maven-3.9.9-bin\apache-maven-3.9.9\bin)

1. ok 🡪 ok 🡪 ok

* mvn compile --- compile the source code files, the generated files are stored into target folder.
* mvn test-compile ----- compile the test case files, the generated files are stored into target folder.
* mvn test – execute the test case files.
* mvn package --- package the application into a jar file, the generated file is stored into target folder.
* mvn install ---- copies the generated jar file into local repository.

Note:

1.we no need to execute the commands in the order. We can directly execute either package or install command.

2. To clean the previous build, it means to remove all the files and folders from the target directory, execute

> mvn clean

3. To clean the previous build, and also to create a new build at a time, then execute

> mvn clean package

First spring application:

========================

1. start STS IDE 🡪 choose the workspace 🡪 launch
2. File menu 🡪 New 🡪 Maven Project 🡪 Next 🡪 Filter: maven-archetype-quickstart 🡪 choose the archetype 🡪 Next 🡪Enter project parameters:

GroupId: in.ashokit

ArtifactId: SpringWelcomeApp

version : 0.0.1-SNAPSHOT

package: com.ashokit.bean

finish

1. Expand the project folder 🡪 expand src/main/java 🡪 expand com.ashokit.bean 🡪 delete App.java file.
2. Expand src folder 🡪 right click on main folder 🡪 new 🡪 folder 🡪 folder name: resources 🡪 finish.
3. right click on the package com.ashokit.bean 🡪 new 🡪 class 🡪 name: WelcomeBean 🡪 finish.

**package** com.ashokit.bean;

**public** **class** WelcomeBean {

**public** String sayWelcome() {

**return** "Welcome to Spring applications!!";

}

}

1. right click on src/main/resources -🡪 new 🡪 file 🡪 filename: config.xml 🡪 finish

<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.xsd">

<bean id = "welcomeBean" class = "com.ashokit.bean.WelcomeBean">

</bean>

</beans>

1. open pom.xml file and add the below dependency.

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>6.2.2</version>

</dependency>

1. right click on src/main/java 🡪 new 🡪 package 🡪 com.ashokit.main 🡪 finish.
2. right click on com.ashokit.main package 🡪 new 🡪 class 🡪 name: Solution 🡪 check the main checkbox 🡪 finish.

**package** com.ashokit.main;

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

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

**import** com.ashokit.bean.WelcomeBean;

**public** **class** Solution {

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

//step-1: start the spring container

ApplicationContext context = **new** ClassPathXmlApplicationContext("config.xml");

//step-2: get the bean(object) from the container

Object obj = context.getBean("welcomeBean");

WelcomeBean wb = (WelcomeBean)obj;

//step-3: call the method

String msg = wb.sayWelcome();

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

}

}

1. right click on the project(SpringWelcomeApp) 🡪 RunAs 🡪 Maven build… 🡪 Goal: package 🡪 run
2. refresh the project
3. right click on Solution.java 🡪 RunAs 🡪 Java Application

Types of Dependency injections:

1. constructor injection
2. setter injection
3. interface injection

* If a dependency object is injected to the dependent object(target object) through the constructor defined in the dependent object, then it is called constructor injection.
* If a dependency object is injected to the depenent object(target object) through the setter method defined in the dependent object, then it is called setter injection.

For example:

public class EmpController (dependent class/target class)

{

EmpService empService;

public EmpController(EmpService empService)

{

this.empService = empService;

}

//other methods

}

public EmpService (dependency class)

{

//methods

}

* The dependent object is providing a constructor for injecting the dependency object. So, it is constructor injection.

For example:

public class EmpController (dependent class/target class)

{

EmpService empService;

public void setEmpService(EmpService empService)

{

this.empService = empService;

}

//other methods

}

public EmpService (dependency class)

{

//methods

}

* The dependent object is providing a setter method for injecting the dependency object. So, it is setter injection.
* In the spring configuration file, to tell the spring container that a constructor is defined to inject a dependency, we have to use <constructor-arg> tag under <bean> tag.
* In the spring configuration file, to tell the spring container that a setter method is defined to inject a dependency, we have to use <property> tag under <bean> tag.
* In spring, a dependent class can have two types of dependencies.
* 1. primitive type dependencies
* 2. reference type dependencies
* <constructor-arg> tag has attributes,

name

index

ref

value

* <property> tag has attributes,

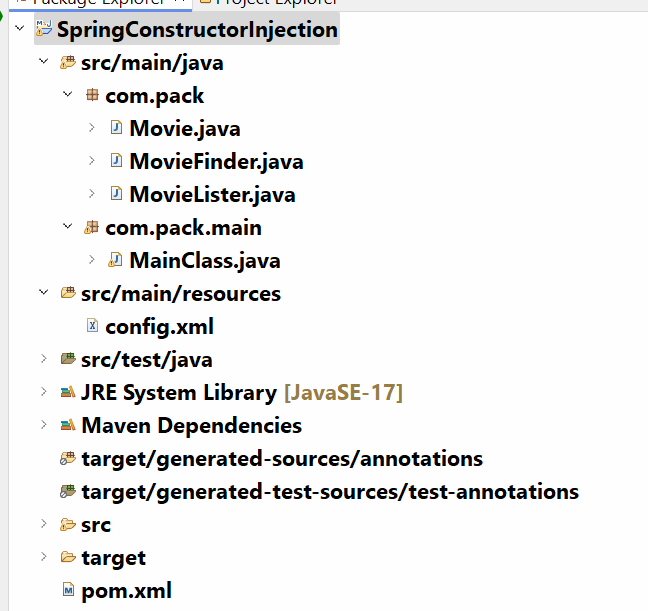
name

ref

value

Constructor injection example:

------------------------------



Movie.java

==========

**package** com.pack;

**public** **class** Movie {

**private** **final** **int** releasedYear;

**private** **final** String movieName;

**public** Movie(**int** releasedYear, String movieName)

{

**this**.releasedYear = releasedYear;

**this**.movieName = movieName;

}

**public** **int** getReleasedYear() {

**return** releasedYear;

}

**public** String getMovieName() {

**return** movieName;

}

}

MovieLister.java

--------------

**package** com.pack;

**import** java.util.List;

//Dependent class

**public** **class** MovieLister {

MovieFinder movieFinder;

**public** MovieLister(MovieFinder movieFinder) {

**this**.movieFinder = movieFinder;

}

**public** **void** showMovies(**int** year) {

List<Movie> list = movieFinder.getMovies(year);

**for**(Movie movie : list) {

System.***out***.println("Released Year : " + movie.getReleasedYear());

System.***out***.println("Movie Name : " + movie.getMovieName());

System.***out***.println("=========================");

}

}

}

MovieFinder.java

----------------

package com.pack;

import java.util.ArrayList;

import java.util.List;

//Dependency class

public class MovieFinder {

ArrayList<Movie> moviesList;

public MovieFinder()

{

moviesList = new ArrayList<>();

moviesList.add(new Movie(2025, "Game Changer"));

moviesList.add(new Movie(2025, "Daku Maharaj"));

moviesList.add(new Movie(2024, "Pushpa2"));

moviesList.add(new Movie(2024, "Lucky Baskar"));

moviesList.add(new Movie(2024, "Devara"));

}

public List<Movie> getMovies(int year) {

List<Movie> found = new ArrayList<>();

//for each loop

for(Movie movie : moviesList)

{

if(movie.getReleasedYear() == year)

found.add(movie);

}

return found;

}

}

config.xml

<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.xsd">

<bean id = "mLister" class = "com.pack.MovieLister">

<constructor-arg name = "movieFinder" ref = "mFinder"/>

</bean>

<bean id = "mFinder" class = "com.pack.MovieFinder">

</bean>

</beans>

MainClass.java

package com.pack.main;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pack.MovieLister;

public class MainClass {

public static void main(String[] args) {

ApplicationContext context = new ClassPathXmlApplicationContext("config.xml");

Object obj = context.getBean("mLister");

MovieLister movieLister = (MovieLister)obj;

movieLister.showMovies(2025);

}

}



User.java

**package** com.example.beans;

**public** **class** User {

**private** **final** String username;

**private** **final** String password;

**public** User(String username, String password) {

**super**();

**this**.username = username;

**this**.password = password;

}

**public** String getUsername() {

**return** username;

}

**public** String getPassword() {

**return** password;

}

}

PasswordService.java

**package** com.example.beans;

**import** java.util.Base64;

**import** java.util.Base64.Decoder;

**import** java.util.Base64.Encoder;

//dependency class

**public** **class** PasswordService {

**public** String encodePassword(String password)

{

Encoder encoder = Base64.*getEncoder*();

**return** encoder.encodeToString(password.getBytes());

}

**public** String decodePassword(String password)

{

Decoder decoder = Base64.*getDecoder*();

**byte**[] b = decoder.decode(password);

**return** **new** String(b);

}

}

UserService.java

**package** com.example.beans;

**import** java.util.HashMap;

//dependent class

**public** **class** UserService {

PasswordService pwdService;

HashMap<String, User> usersMap = **new** HashMap<>();

//setter method

**public** **void** setPwdService(PasswordService pwdService) {

**this**.pwdService = pwdService;

}

**public** **void** saveUser(String username, String password)

{

//encode the password

String encodedPassword = pwdService.encodePassword(password);

System.***out***.println("In saveUser() method....");

System.***out***.println("password : " + password);

System.***out***.println("encoded password : " + encodedPassword);

//create User class object

User user = **new** User(username, encodedPassword);

//store the User object to HashMap collection

usersMap.put(username, user);

System.***out***.println("User is saved.");

}

**public** **void** fetchUser(String username) {

//fetch the user from the HashMap collection

User user = usersMap.get(username);

**if**(user != **null**)

{

String password = pwdService.decodePassword(user.getPassword());

System.***out***.println("Username : " + username);

System.***out***.println("Password : " + password);

}

**else**

{

System.***out***.println(username + " doesn't exist!!!");

}

}

}

beans.xml

<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.xsd">

<bean id = "uService" class = "com.example.beans.UserService">

<property name = "pwdService" ref = "pService"/>

</bean>

<bean id = "pService" class = "com.example.beans.PasswordService">

</bean>

</beans>

Main.java

package com.example.main;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.example.beans.UserService;

public class Main {

public static void main(String[] args) {

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

Object obj = context.getBean("uService");

UserService userService = (UserService)obj;

System.out.println("calling saveUser()....");

userService.saveUser("ashokit", "ashokit@123");

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

System.out.println("calling fetchUser() ...");

userService.fetchUser("ashokit");

}

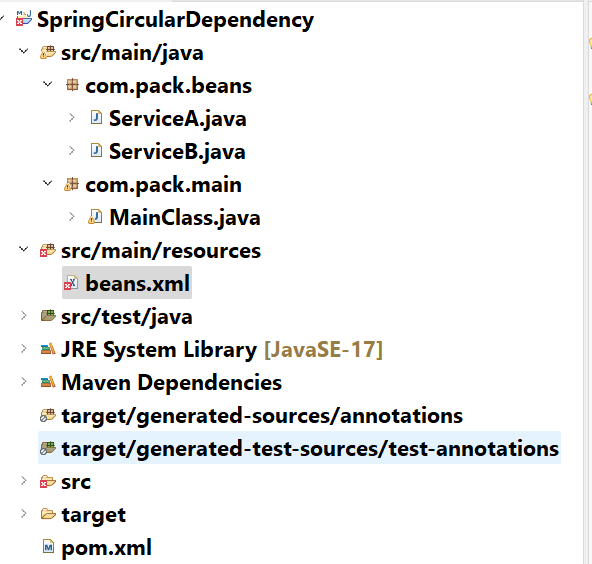
}

Circular dependency:

------------------

* Suppose if we have two components ServiceA and ServiceB, where ServiceA depends on ServiceB and ServiceB depends on ServiceA, then it is circular dependency.
* If both the components have defined a constructor for injecting its dependency object, then we will get BeanCreationException.
* The solution to this exception is,

1. define a setter method to inject the dependency object, in one of the components.
2. configure the component which has setter method above the other components in the xml file.



ServiceA.java

------------

**package** com.pack.beans;

**public** **class** ServiceA {

ServiceB serviceB;

//setter method

**public** **void** setServiceB(ServiceB serviceB)

{

**this**.serviceB = serviceB;

}

**public** **void** m1() {

System.***out***.println("In ServiceA :: m1()");

serviceB.f1();

}

**public** **void** m2() {

System.***out***.println("In ServiceA :: m2()");

}

}

ServiceB.java

------------

**package** com.pack.beans;

**public** **class** ServiceB {

ServiceA serviceA;

//constructor

**public** ServiceB(ServiceA serviceA)

{

**this**.serviceA = serviceA;

}

**public** **void** f1() {

System.***out***.println("In ServiceB :: f1()");

}

**public** **void** f2() {

System.***out***.println("In ServiceB :: f2()");

serviceA.m2();

}

}

beans.xml

--------

<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.xsd">

<bean id = "sa" class = "com.pack.beans.ServiceA">

<property name = "serviceB" ref = "sb"/>

</bean>

<bean id = "sb" class = "com.pack.beans.ServiceB">

<constructor-arg name = "serviceA" ref = "sa"/>

</bean>

</beans>

MainClass.java

package com.pack.main;

import org.springframework.context.ApplicationContext;

import org.springframework.context.support.ClassPathXmlApplicationContext;

import com.pack.beans.ServiceA;

public class MainClass {

public static void main(String[] args) {

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

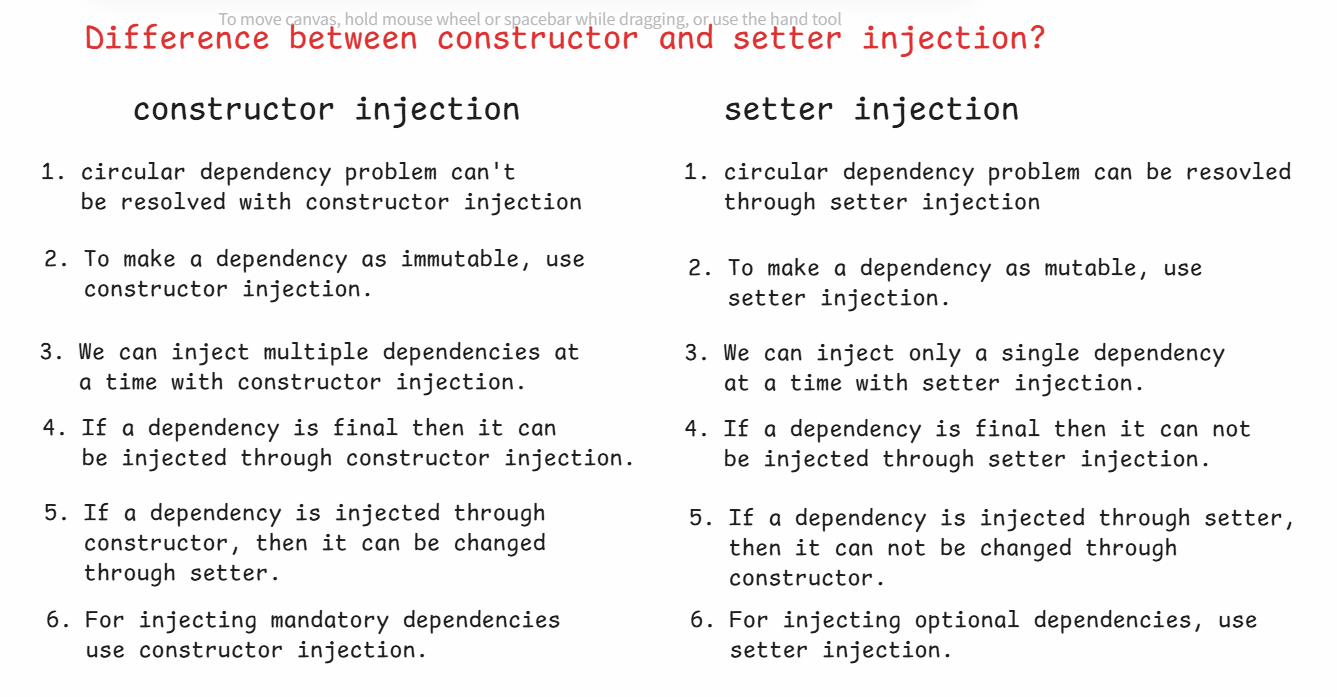
Object obj = context.getBean("sa");

ServiceA servA = (ServiceA)obj;

servA.m1();

}

}



Bean autowiring:

--------------

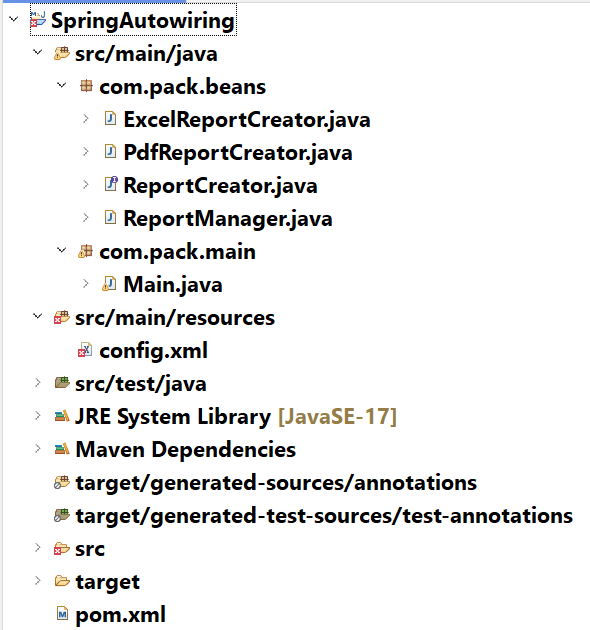
* wiring a bean is nothing but connecting a dependent object with its dependencies.
* In Bean autowiring, the Spring Container automatically injects the dependency objects to the dependent object, without providing any configuration the xml file using <constructor-arg> or <property> tags.
* This Bean autowiring facility is only provided for object type dependencies.
* To enable bean autowiring on a class, add autowire attribute to the <bean> tag.
* The default value of autowire attribute is “no”. It means, by default autowiring on a bean is disabled.
* To enable, you can specify one of the below three values for the autowire attributes. These are called autowiring strategies.

1. byName
2. byType
3. constructor

* when autowire=”byName”, the spring container will search for a bean whose id matches with the reference variable to be injected.
* If found, then the container injects the dependency object by calling setter method.
* If not found, then the container doesn’t inject the dependency object.
* when autowire=”byType”, the spring container will search for a bean whose class matches with the datatype of the reference variable to be injected.
* If found, then the container injects the dependency object by calling setter method.
* If not found, then the container doesn’t inject the dependency object.
* when autowire=”constructor”, the spring container will search for a bean whose class matches with the datatype of the reference variable to be injected.
* If found, then the container injects the dependency object by calling the constructor.
* If not found, then the container throws BeanCreationException.

primary attribute:

* If multiple matching bean classes are found then Spring container throws UnsatisfiedDependencyInjection.
* To resolve it, we should provide higher priority for a bean class with primary = “true” attribute.



ReportCreator.java

**package** com.pack.beans;

**public** **interface** ReportCreator {

**void** createReport(**int** pages);

}

PdfReportCreator.java

**package** com.pack.beans;

**public** **class** PdfReportCreator **implements** ReportCreator {

@Override

**public** **void** createReport(**int** pages) {

System.***out***.println("PdfReportCreator :: created report with : "+ pages + " pages");

}

}

ExcelReportCreator.java

**package** com.pack.beans;

**public** **class** ExcelReportCreator **implements** ReportCreator {

@Override

**public** **void** createReport(**int** pages) {

System.***out***.println("ExcelReportCreator :: created report with : "+pages+ " pages");;

}

}

ReportManager.java

**package** com.pack.beans;

**public** **class** ReportManager {

ReportCreator reportCreator;

**private** **int** pages;

**public** ReportManager(**int** pages) {

**this**.pages = pages;

}

**public** ReportManager(ReportCreator reportCreator) {

**this**.reportCreator = reportCreator;

}

**public** **void** setReportCreator(ReportCreator reportCreator)

{

**this**.reportCreator = reportCreator;

}

**public** **void** printReport()

{

reportCreator.createReport(pages);

}

}

config.xml

<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.xsd">

<bean id = "reportManager" class = "com.pack.beans.ReportManager" autowire = "byType">

<constructor-arg name = "pages" value = "15"/>

</bean>

<bean id = "pdfCreator" class = "com.pack.beans.PdfReportCreator" primary = "true" />

<bean id = "excelCreator" class = "com.pack.beans.ExcelReportCreator" />

</beans>

Main.java

**package** com.pack.main;

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

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

**import** com.pack.beans.ReportManager;

**public** **class** Main {

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

ApplicationContext context = **new** ClassPathXmlApplicationContext("config.xml");

ReportManager rm = context.getBean(ReportManager.**class**);

rm.printReport();

}

}

==================================================================

Initializing and Disposing a bean:

* Initializing a bean refers to executing some logic automatically by the container, immediately after injecting the bean dependencies.
* Disposing a bean refers to executing some cleanup logic automatically by the container, before removing a bean from the container.
* Spring framework has provided, InitializingBean and DisposableBean interfaces to define the logics.
* InitializingBean interface has provided afterPropertiesSet() method to define the initialization logic.
* DisposableBean interface has provided destroy() method to define the cleanup resources logic.
* To define initialization and disposing logics without implementing the spring provided interfaces, we can define custom methods. (user-defined methods).
* A custom initialization method can be specified to the container, with init-method attribute.
* A custom destroy method can be specified to the container, with destroy-method attribute.

ex:

public class DBOperations implements InitializingBean, DisposableBean {

private String driver;

private String url;

private String username;

private String password;

//setter methods

@Override

public void afterPropertiesSet() {

//initialization logic

//open a connection with database

}

//other methods

@Override

public void destroy() {

//cleanup logic

//close a connection with database

}

}

ex2:

public class DBOperations {

private String driver;

private String url;

private String username;

private String password;

//setter methods

//cusom init method

public void setUp() {

//initialization logic

//open a connection with a database

}

//other methods

//custom destroy method

public void tearDown() {

//cleanup logic

//close a connection with a database

}

}

* In <bean> tag of xml, you have to add the init-method and destory-method attributes.

<bean id=”db” class=”DBOperations” init-method=”setUp”

destroy-method=”tearDown”>

Interface Injection:

---------------------

* In interface injection, the Spring Container injects a dependency, by calling a method which is provided by an interface.
* Spring has a limited support for interface injection. It performs interface injection only for the below interfaces.

1. BeanNameAware
2. ApplicationContextAware

* BeanNameAware interface has an abstract method setBeanName()
* ApplicationContextAware interface has an abstract setApplicationContext()
* By calling BeanNameAware’s setBeanName() method, the container injects the id/name of the bean.
* By calling ApplicationContextAware’s setApplicationContext() method, the container injects ApplicationContext object to the bean.

Bean Life cycle:

--------------

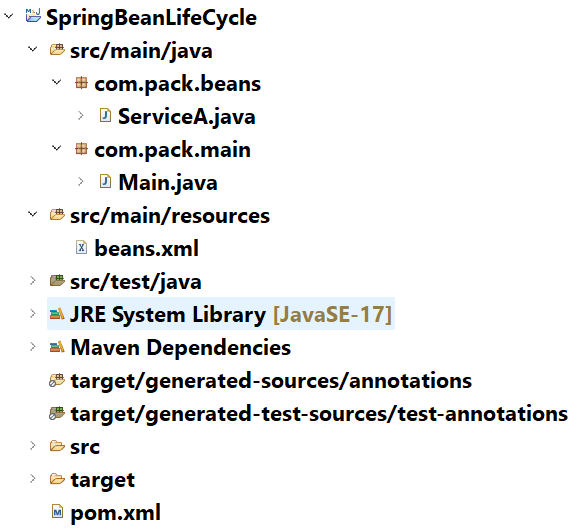
1. creation
2. initialization
3. service
4. destroy

* By loading a bean definition from the xml file, the spring container creates an object for that bean class.
* After creating the object, the spring container will execute the below steps to complete the initialization.

1. dependencies are injected.
2. If BeanNameAware is implemented, then calls setBeanName() method, injects the bean id/name.
3. If ApplicationContextAware is implemented, then calls setApplicationContext() method, injects the ApplicationContext.
4. If InitializingBean is implemented, then calls afterPropertiesSet() method.
5. If custom init method is configured, then executes it.

* Now, the bean is ready to provide the services to the client.
* It means, a client class invokes the services/methods of the bean.
* When a Spring Container is going to shutdown, it will remove the beans from the container. During this, the container performs destroy operations.

1. If DisposableBean interface is implemented, then calls destory() method.
2. if custom destroy method is configured, then executes it.



ServiceA.java

------------

package com.pack.beans;

import org.springframework.beans.factory.BeanNameAware;

import org.springframework.beans.factory.DisposableBean;

import org.springframework.beans.factory.InitializingBean;

import org.springframework.context.ApplicationContext;

import org.springframework.context.ApplicationContextAware;

public class ServiceA implements BeanNameAware, ApplicationContextAware, InitializingBean, DisposableBean {

private String message;

public ServiceA() {

System.out.println("ServiceA :: constructor");

}

public void setMessage(String message)

{

this.message = message;

System.out.println("Dependency is injected");

}

public void printMessage() {

System.out.println("message ----> " + message);

}

@Override

public void setBeanName(String name) {

// TODO Auto-generated method stub

System.out.println("BeanNameAware's setBeanName()");

}

@Override

public void setApplicationContext(ApplicationContext applicationContext) {

// TODO Auto-generated method stub

System.out.println("ApplicationContextAware's setApplicationContext()");

}

@Override

public void afterPropertiesSet() {

// TODO Auto-generated method stub

System.out.println("InitiazingBean's afterPropertiesSet()");

}

@Override

public void destroy() {

// TODO Auto-generated method stub

System.out.println("DisposableBean's destroy()");

}

public void setUp() {

System.out.println("custom init method : setUp()");

}

public void tearDown() {

System.out.println("custom destroy method : tearDown()");

}

}

beans.xml

========

<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.xsd">

<bean name = "sa" class = "com.pack.beans.ServiceA" init-method = "setUp" destroy-method = "tearDown">

<property name = "message" value = "Welcome to LifeCycle"/>

</bean>

</beans>

Main.java

----------

**package** com.pack.main;

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

**import** com.pack.beans.ServiceA;

**public** **class** Main {

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

ClassPathXmlApplicationContext context = **new** ClassPathXmlApplicationContext("beans.xml");

ServiceA servA = context.getBean(ServiceA.**class**);

servA.printMessage();

context.close();

}

}

========================================================

Spring Annotations

===============================

@Component :

\* It is a class-level annotation.

\* This annotation is used to specify that a class is a

spring managed bean class.

\* During components auto scanning, spring will identify

the classes with @Component annotation as spring beans

and registers them with the spring application context.

\* A class with @Component annotation will be identified as

a general purpose spring bean, could be used in mulitple

layers.

Ex:

@Component

public class Validator {

//methods

}

. The above class is registerd in the spring application context with an id as “validator”.

@Component(value = “id1”)

public class Validator {

//methods

}

. The above class is registered in the spring application context with an id as “id1”.

@Service :

\* It is a class level annotation.

\* This is used to specify that a class is a service layer

or business logic layer class.

\* @Service is derived from @Component.

\* During components auto scanning, spring will identify

the classes with @Service annotation as spring beans

and registers them with the spring application context.

Ex:

@Service

public class EmployeeService {

//methods

}

. The above class is registered in the spring application context with id as “employeeService”.

@Repository :

\* It is a class level annotation.

\* This is used to specify that a class is a Data access

layer or persistence layer class.

\* @Repository is derived from @Component.

\* During components auto scanning, spring will identify

the classes with @Repository annotation as spring beans

and registers them with the spring application context.

Ex:

@Repository

public class EmployeeDao {

//methods

}

. The above class is registered in spring application context with id as “employeeDao”.

@Controller:

\* It is a class level annotation.

\* This is used to specify that a class is a presentation

layer or User Interface layer class.

\* @Controller is derived from @Component.

\* During components auto scanning, spring will identify

the classes with @Controller annotation as spring beans

and registers them with the spring application context.

Ex:

@Controller

public class EmployeeController {

//methods

}

. The above class is registered in the application context with id as “employeeController”.

@RestController :

\* It is a class level annotation.

\* This is used to specify that a class is a API layer

class.

\* @RestController is derived from @Controller.

\* During components auto scanning, spring will identify

the classes with @RestController annotation as spring

beans and registers them with the spring application

context.

Ex:

@RestController

public class EmployeeApi {

//methods

}

@Configuration:

\* It is a class level annotation.

\* It is used to mark that a class is a Java configuration

class.

\* @Configuration is derived from @Component.

\* This annotation indicates that a class is a source for

one or more @Bean methods.

\* @Bean indicates that a method produces an object(bean),

that should be stored into the spring container.

\* @Bean is a method level annotation.

\* If you want to explicitly create an object, initialize

an object and then if you want to push it into spring

container, then you have to create a @Bean method.

\* Mostly we create @Bean methods for creating and

initializing the objects for pre-defined classes.

Because, spring container can create and initialize the

objects for user-defined classes only.

Ex:

@Configuration

public class AppConfig {

@Bean

public JdbcTemplate jdbcTemplate() {

//instantiating and initializing JdbcTemplate obj.

return obj;

}

}

@ComponentScan:

* It is a class-level annotation.
* It is used to specify the base package(s) for components auto scanning.
* If you specify a base package then spring will search/scan for the spring bean components from the base package and it will reach out to sub packages also.
* This annotation is used at configuration class.
* It is repeatable annotation, which means, we can repeat the same annotation more than once at the class level.

ex1:

@Configuration

@ComponentScan(value = “com.account”)

@ComponentScan(value = “org.loan”)

public class AppConfig {

}

ex2:

@Configuration

@ComponentScan(value = {“com.account”, “org.loan”})

public class AppConfig {

}

* To start the spring container(ApplicationContext), we have to instantiate AnnotationConfigApplicationContext class.
* To the AnnotationConfigApplicationContext object, we should provide Class object of configuration class as a parameter.

ex:

ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);

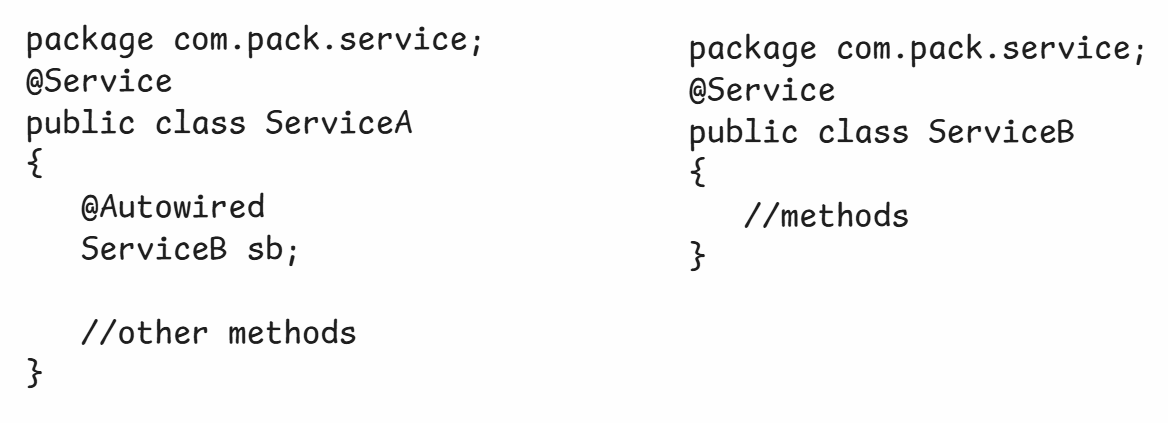
* If more than one configuration class exists, then we should provide Class[] as a parameter.

ex:

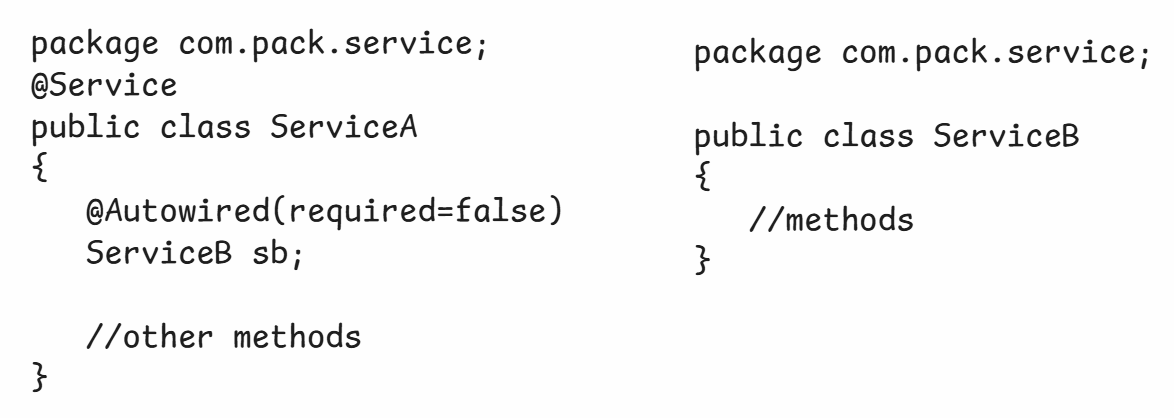
ApplicationContext context = new AnnotationConfigApplicationContext(new Class[]{ AppConfig1.class, AppConfig2.class});

@Autowired :

* This annotation can be used at field level, constructor level or setter level.
* In spring annotations, we got another type of dependency injection called, field injection.
* Field injection means, spring will directly inject the dependency object to the field, by without needing a constructor or a setter method.
* Suppose, if we use @Autowired at field level, then spring performs field injection.
* For field injection, spring internally uses reflection api.

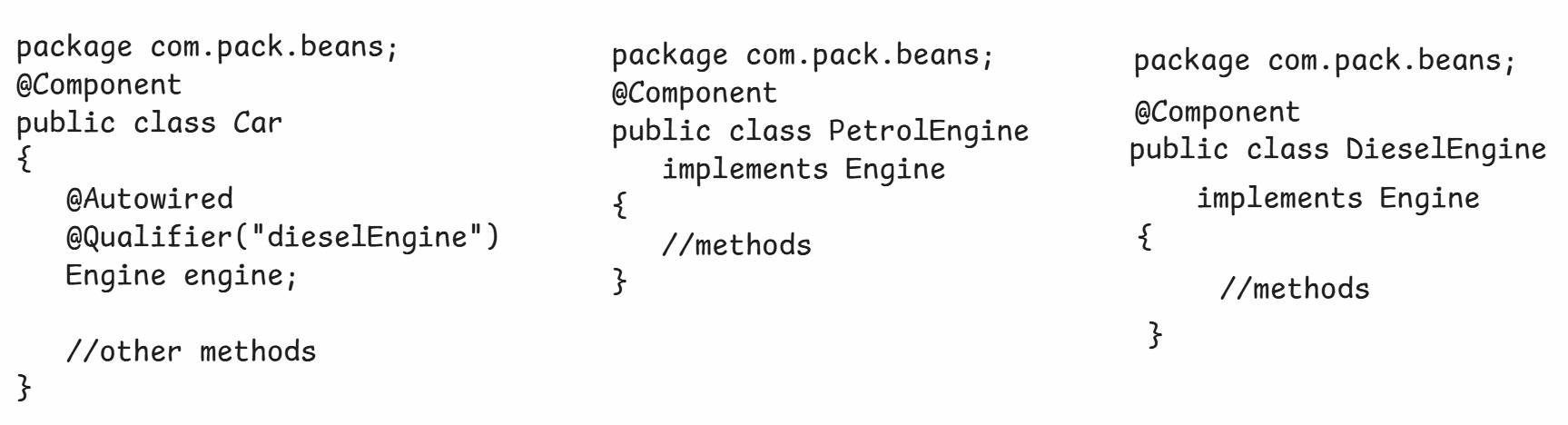


* Suppose, if spring can not find a matching bean for autowiring, then throws UnsatisfiedDependencyException.
* To make autowiring a dependency object as optional, we have to provide a parameter required = false, to the @Autowired annotation.



@Qualifier:

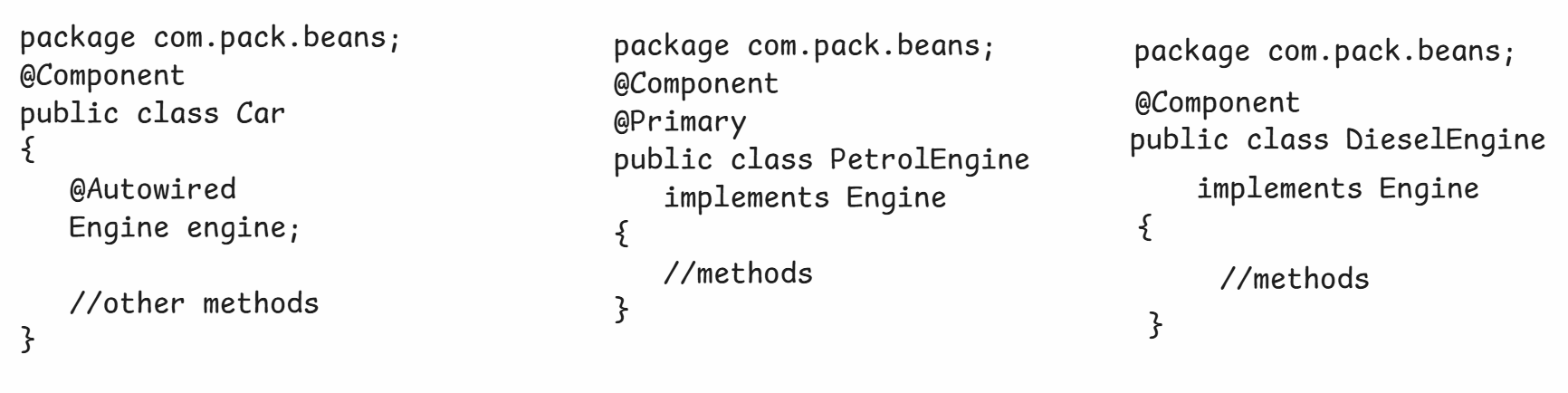
* If multiple matching beans are found by the spring, for autowiring then because of ambiguity, again spring throws UnsatisfiedDependencyException.
* To avoid this exception, we can specify a specific bean name with @Qualifier annotation.



@Primary:

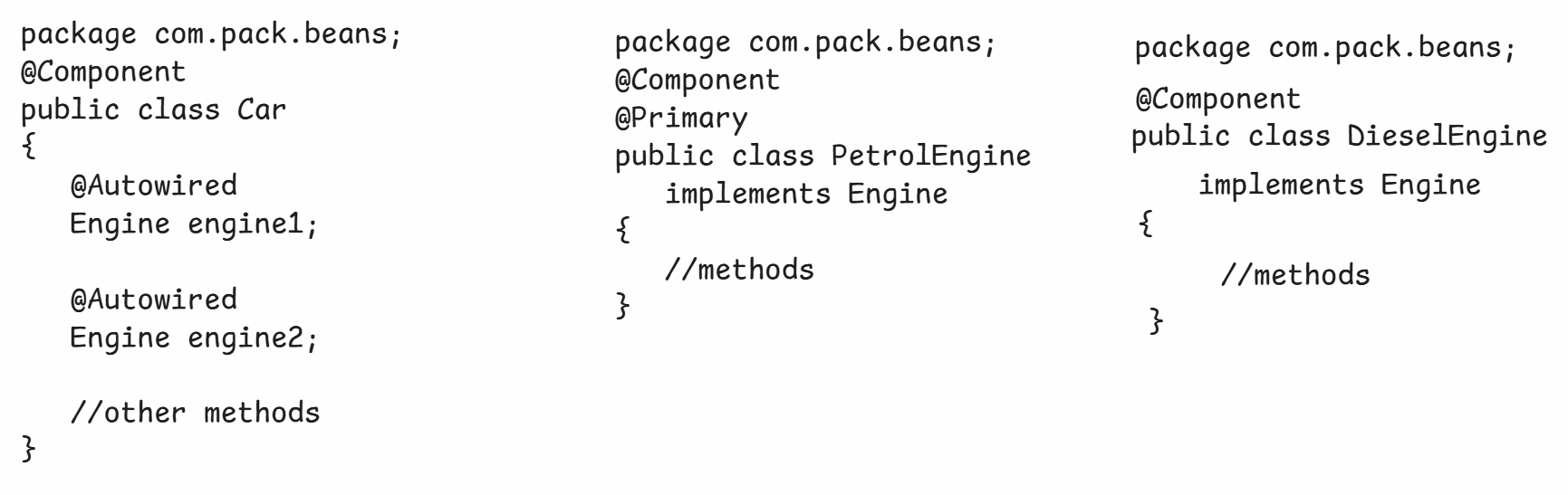
* This annotation can be used at class-level or at method-level.
* This annotation can be used to specify that a bean is a default choice for dependency inection, when mulitple beans are found of the same type by the spring container.
* Suppose, at injection point, if you want to change the bean for dependency injection, then you have to use @Qualifier explicitly.

ex1:



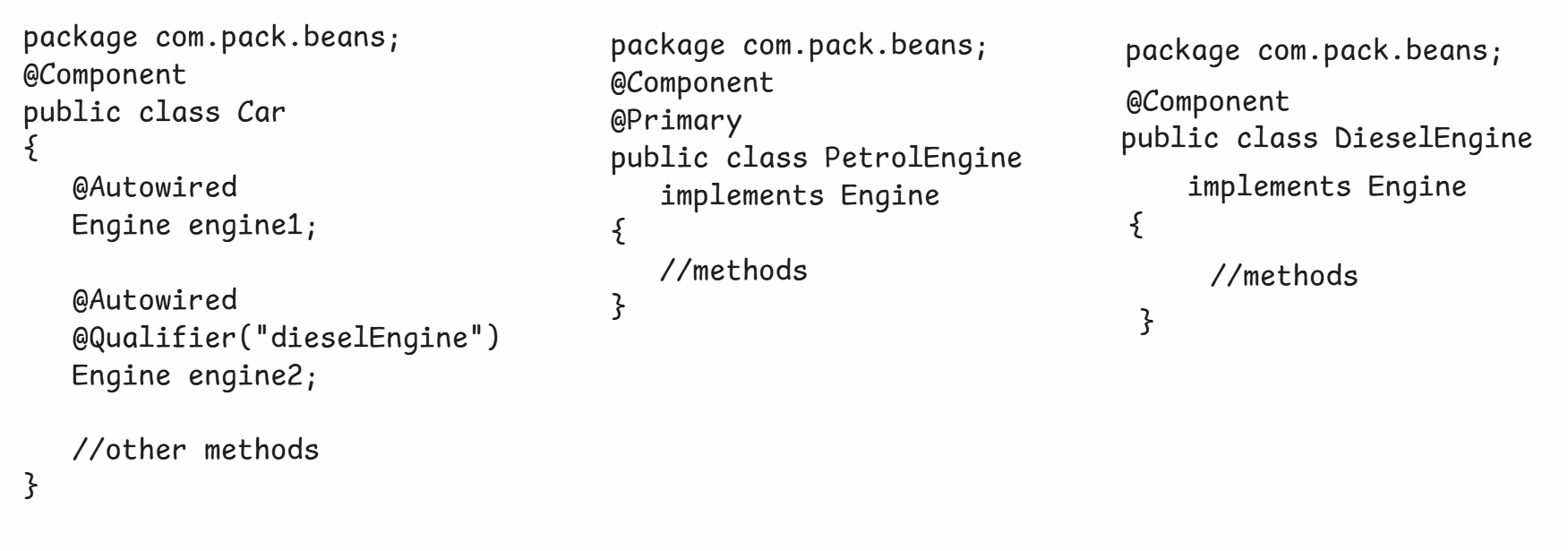
* Here, the spring injects PetrolEngine object as a dependency object to the Car object(dependent object).

ex2:

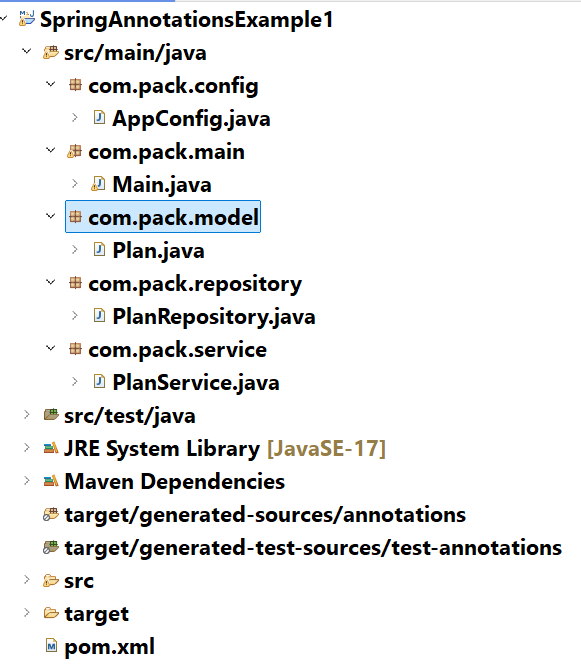


* Here, the spring injects PetrolEngine object as dependency object for both reference variables(engine1, engine2).

ex3:



* Here, the spring injects PetrolEngine object as a dependency object for engine1 and DieselEngine object as a dependency object for engine2.
* @Primary is a default choice and @Qualifier is an explicit choice.
* If both are used then @Qualifier gets more priority.
* @Qualifier is used at injection point and @Primary is used to declaration level.



Plan.java

--------

**package** com.pack.model;

**public** **class** Plan {

**private** String planId;

**private** String description;

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

**private** String validity;

**public** Plan(String planId, String description, **double** price, String validity) {

**super**();

**this**.planId = planId;

**this**.description = description;

**this**.price = price;

**this**.validity = validity;

}

**public** String getPlanId() {

**return** planId;

}

**public** **void** setPlanId(String planId) {

**this**.planId = planId;

}

**public** String getDescription() {

**return** description;

}

**public** **void** setDescription(String description) {

**this**.description = description;

}

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

**return** price;

}

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

**this**.price = price;

}

**public** String getValidity() {

**return** validity;

}

**public** **void** setValidity(String validity) {

**this**.validity = validity;

}

@Override

**public** String toString() {

**return** "Plan [planId=" + planId + ", description=" + description + ", price=" + price + ", validity=" + validity

+ "]";

}

}

PlanRepository.java

---------------

**package** com.pack.repository;

**import** java.util.HashMap;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Repository;

**import** com.pack.model.Plan;

@Repository

**public** **class** PlanRepository {

@Autowired

HashMap<String, Plan> plansMap;

**public** Plan findById(String planId) {

**return** plansMap.get(planId);

}

**public** HashMap<String, Plan> findAll() {

**return** plansMap;

}

}

PlanService.java

------------

**package** com.pack.service;

**import** java.util.HashMap;

**import** org.springframework.beans.factory.annotation.Autowired;

**import** org.springframework.stereotype.Service;

**import** com.pack.model.Plan;

**import** com.pack.repository.PlanRepository;

@Service

**public** **class** PlanService {

@Autowired

PlanRepository repository;

**public** Plan getPlanById(String planId) {

**return** repository.findById(planId);

}

**public** HashMap<String, Plan> getAllPlans() {

**return** repository.findAll();

}

}

AppConfig.java

-------------

**package** com.pack.config;

**import** java.util.HashMap;

**import** org.springframework.context.annotation.Bean;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

**import** com.pack.model.Plan;

@Configuration

@ComponentScan(value = "com.pack")

**public** **class** AppConfig {

@Bean

**public** HashMap<String, Plan> hashMap() {

HashMap<String, Plan> hm = **new** HashMap<>();

hm.put("p-101", **new** Plan("p-101", "1GB/day, 100SMS/day", 399, "28 Days"));

hm.put("p-102", **new** Plan("p-102", "1.5GB/day, 100SMS/day", 599, "36 Days"));

hm.put("p-103", **new** Plan("p-103", "2GB/day, 100SMS/day", 799, "56 Days"));

**return** hm;

}

}

Main.java

---------

**package** com.pack.main;

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

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

**import** com.pack.config.AppConfig;

**import** com.pack.service.PlanService;

**public** **class** Main {

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

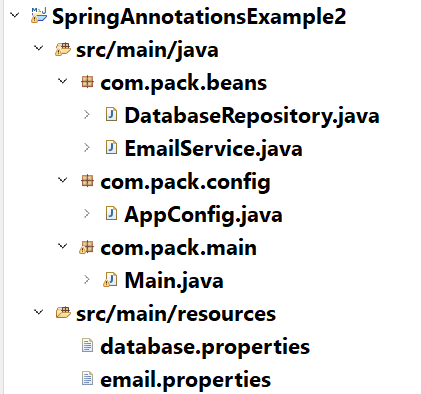
ApplicationContext context = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

PlanService service = context.getBean(PlanService.**class**);

System.***out***.println(service.getPlanById("p-102"));

}

}



EmailService.java

---------------

**package** com.pack.beans;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.stereotype.Service;

@Service

**public** **class** EmailService {

@Value("${smtp.host}")

**private** String smtpHost;

@Value("${smtp.port}")

**private** **int** smtpPort;

@Value("${to.address}")

**private** String toAddress;

@Value("${from.address}")

**private** String fromAddress;

**public** **void** sendMail() {

System.***out***.println("Email sent to : " +toAddress + " from : " + fromAddress

+ " with smtpHost : " + smtpHost + " and smtpPort : " + smtpPort);

}

}

DatabaseRepository.java

----------------------

**package** com.pack.beans;

**import** org.springframework.beans.factory.annotation.Value;

**import** org.springframework.stereotype.Repository;

@Repository(value = "dbRepository")

**public** **class** DatabaseRepository {

@Value("${driver.name}")

**private** String driverClassName;

@Value("${database.port}")

**private** String port;

@Value("${database.username}")

**private** String username;

@Value("${database.password}")

**private** String password;

**public** **void** showDatabaseProperties() {

System.***out***.println("Driver classname : " + driverClassName);

System.***out***.println("Database port : " + port);

System.***out***.println("username : " + username);

System.***out***.println("password : " + password);

}

}

AppConfig.java

-------------

**package** com.pack.config;

**import** org.springframework.context.annotation.ComponentScan;

**import** org.springframework.context.annotation.Configuration;

**import** org.springframework.context.annotation.PropertySource;

@Configuration

@ComponentScan("com.pack")

@PropertySource("database.properties")

@PropertySource("email.properties")

**public** **class** AppConfig {

}

Main.java

-------

**package** com.pack.main;

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

**import** org.springframework.context.annotation.AnnotationConfigApplicationContext;

**import** com.pack.beans.DatabaseRepository;

**import** com.pack.beans.EmailService;

**import** com.pack.config.AppConfig;

**public** **class** Main {

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

ApplicationContext ctx = **new** AnnotationConfigApplicationContext(AppConfig.**class**);

EmailService emailService = ctx.getBean(EmailService.**class**);

emailService.sendMail();

System.***out***.println("========================");

DatabaseRepository dbRepository = ctx.getBean(DatabaseRepository.**class**);

dbRepository.showDatabaseProperties();

}

}

database.properties

-----------------

# database properties

driver.name=com.mysql.jdbc.Driver

database.port=3306

database.username=root

database.password=root

email.properties

---------------

# email properties

smtp.port=9761

smtp.host=smtp.google.com

to.address=John@gmail.com

from.address=ashokit@gmail.com

Singleton design pattern:

----------------------

