Spring Boot with Microservices

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

Trainer : Shekher

(Working Professional)

Timing : 11AM – 1PM

Duration : 3 Months

Mode : offline / online

Fee : 8k (without backup videos)

10k (with backup videos(1 year validity))

Pre-requisites: Core JAVA

SQL Basics(CRUD)

Basics of JDBC

Basics of Web

Spring Boot contents:

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

1. Spring Framework basics (Core)
2. Spring Core Annotations
3. Spring Boot Introduction
4. Spring Boot with JDBC
5. Spring Boot with JPA
6. Spring Boot MVC
7. Spring Boot Security
8. Spring Boot REST API
9. Spring Boot REST + JWT security
10. Spring Boot REST + OAUTH2 security
11. Spring Boot Messaging with Kafka
12. Spring Boot Monitoring
13. Spring Boot Batch processing

Microservices contents:

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

1. Monolithic applications
2. Microservices
3. Microservices with Cloud
4. Centralized Configurations
5. Service Registry with Eureka
6. Spring Cloud Gateway
7. Spring Cloud Load balancing
8. Resilience4J
9. Spring Microservices Security
10. Grafana
11. Prometheus
12. Docker
13. Kubernetes(k8s)
14. Deployment on AWS

Spring Framework Basics (Core)

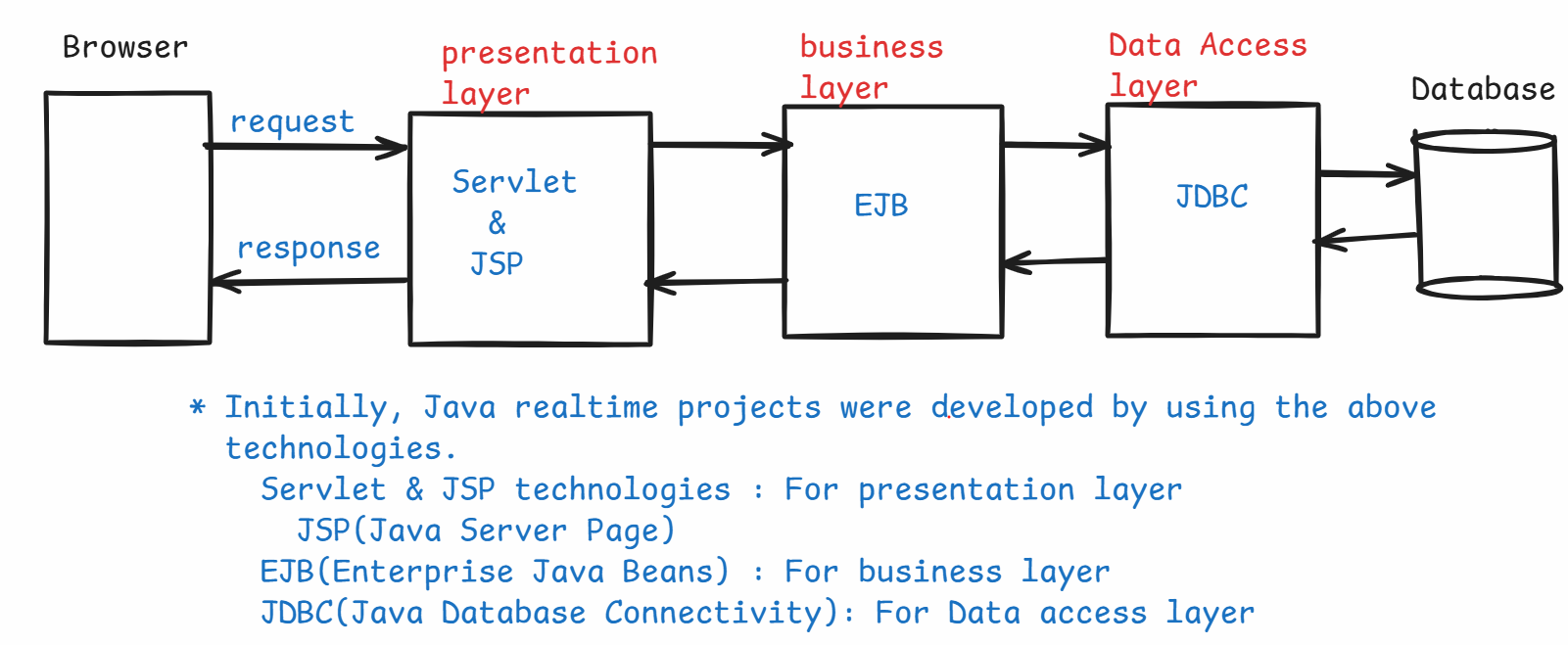
what is a Framework?

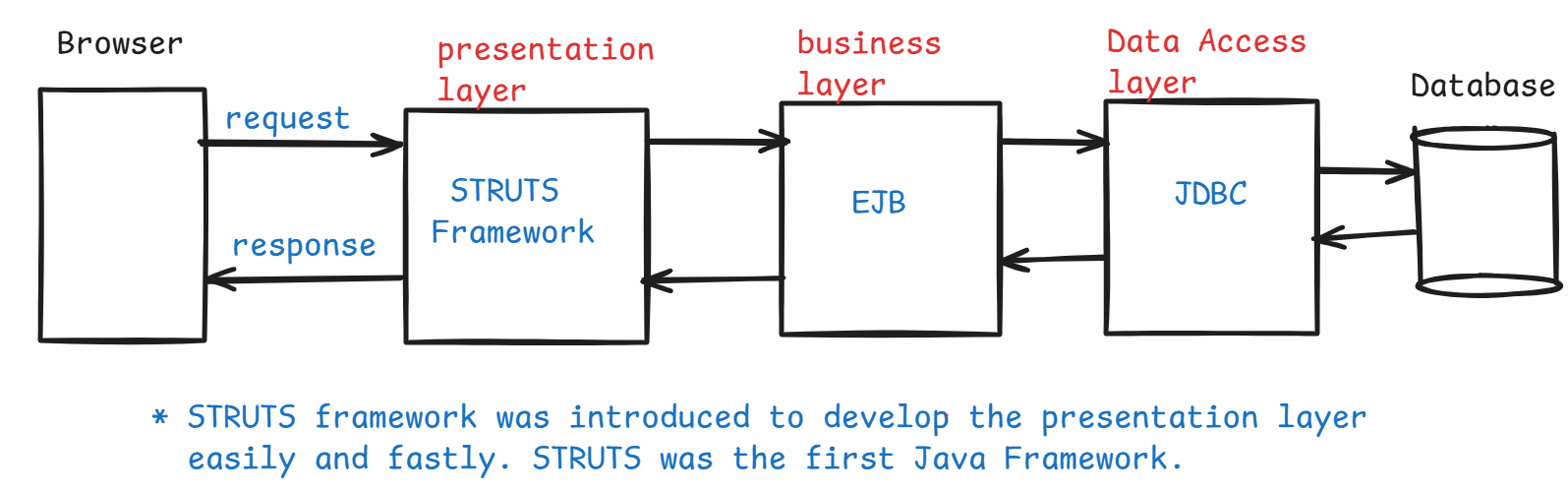
* A Framework is a pre-built structure that provides a foundation to develop the software applications.
* Every project contains some common functionalities and also business functionalities.
* A Framework will provide the common functionalities and the developers are only required to implement the business functionalities.
* In this way, a Framework will help the developers to build the software application rapidly.

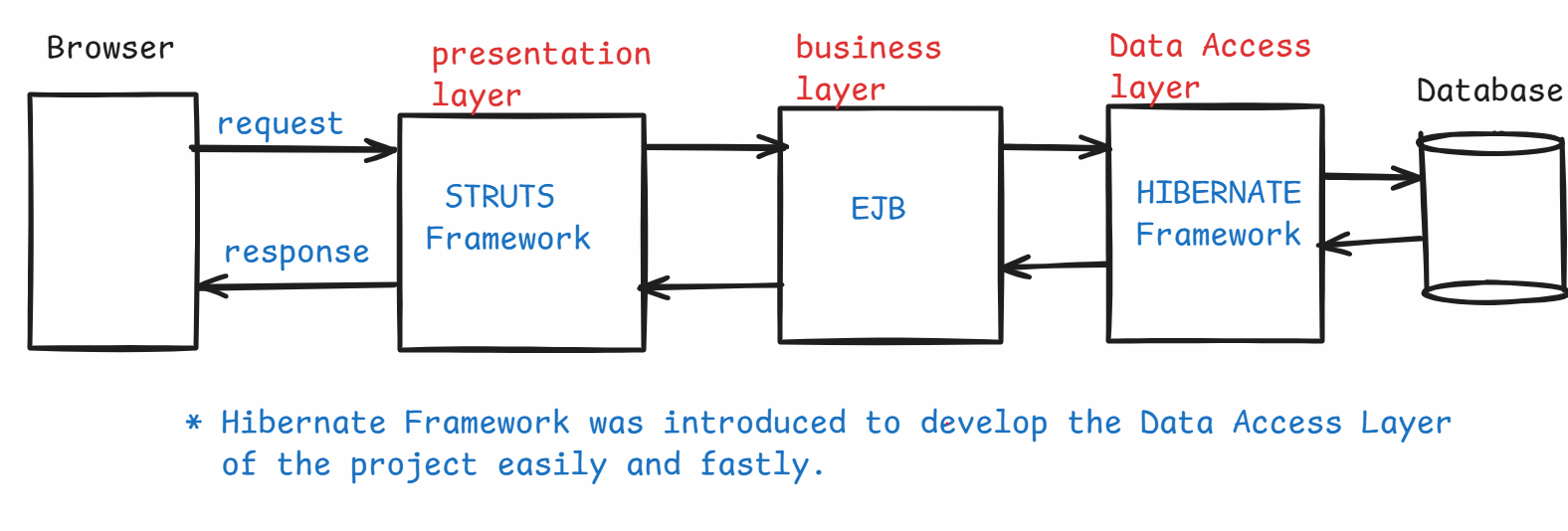
How a Framework is different from a technology?

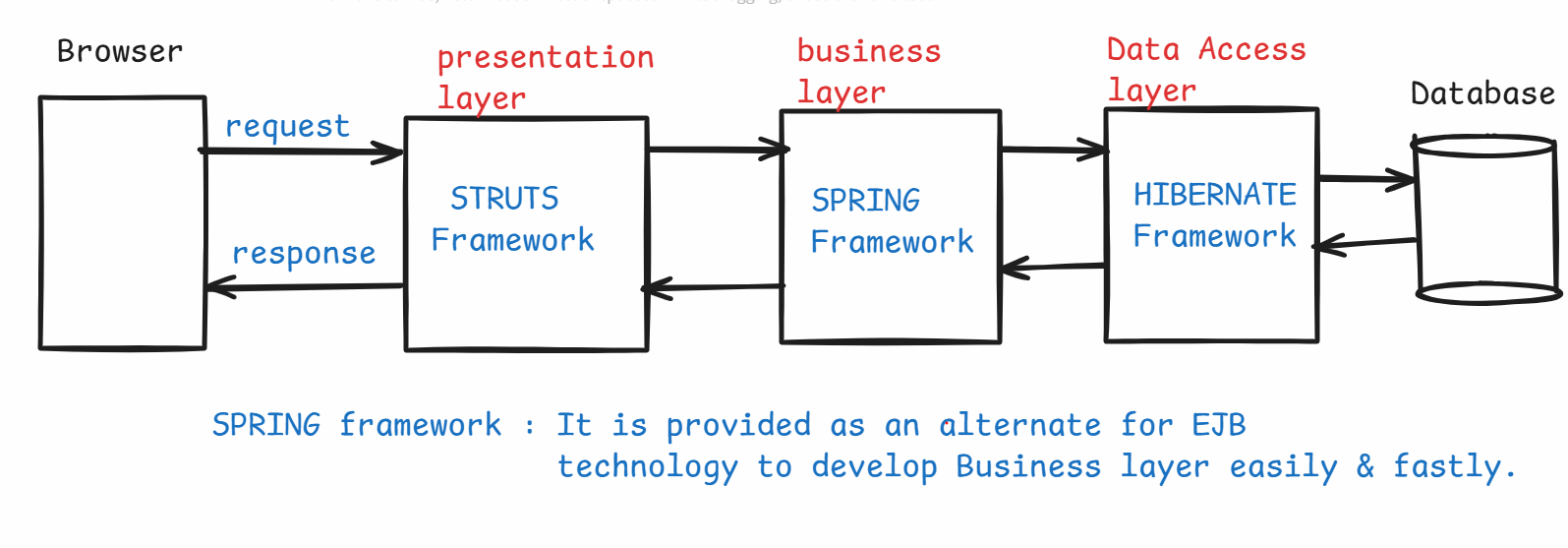
* A framework provides a pre-defined structure to develop the applications.
* A technology only provides the tools and methodologies to develop the applications.
* Suppose, if you want to develop a software which requires 100 lines of code then with technologies, you have to write the entire code by yourself.
* If you use a framework, then almost 50% of code will be provided by the framework only. So, you have to write the remaining code for completing the application.
* It means, with frameworks, you can develop the applications easily and fastly.

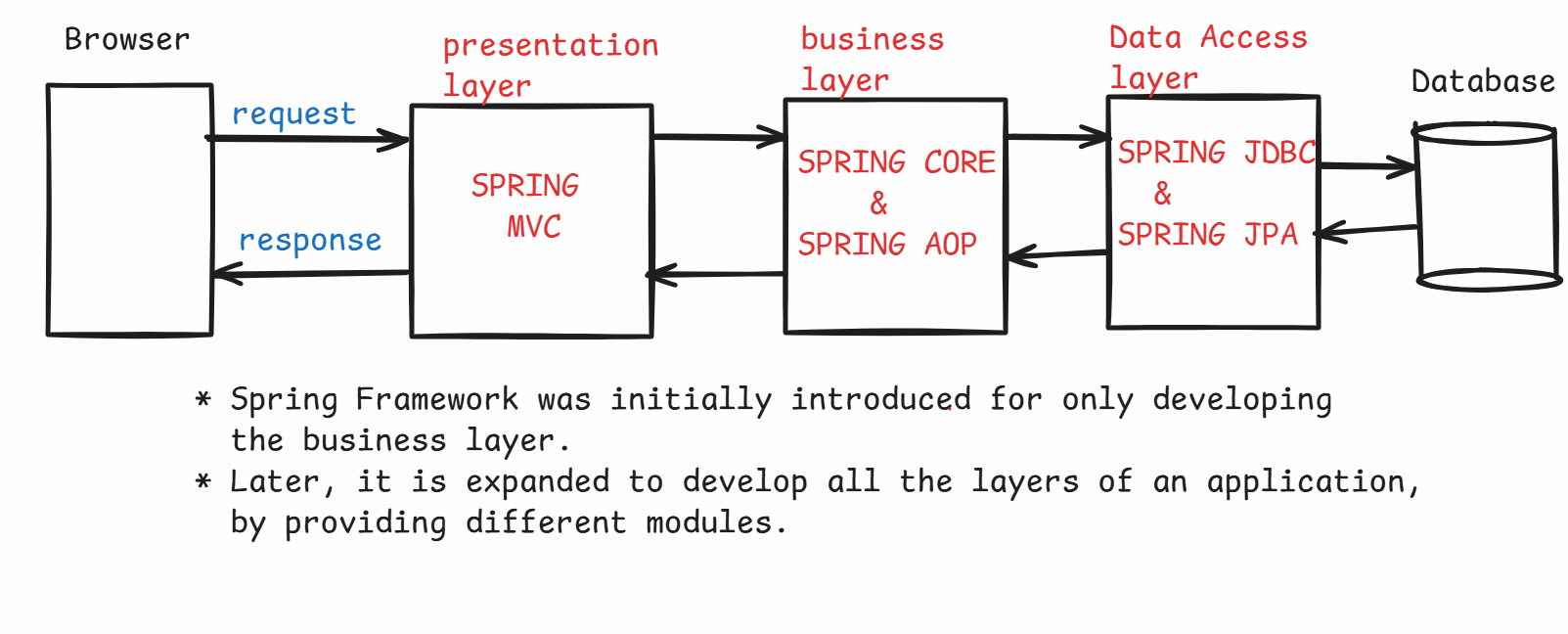
How frameworks are introduced in Java?

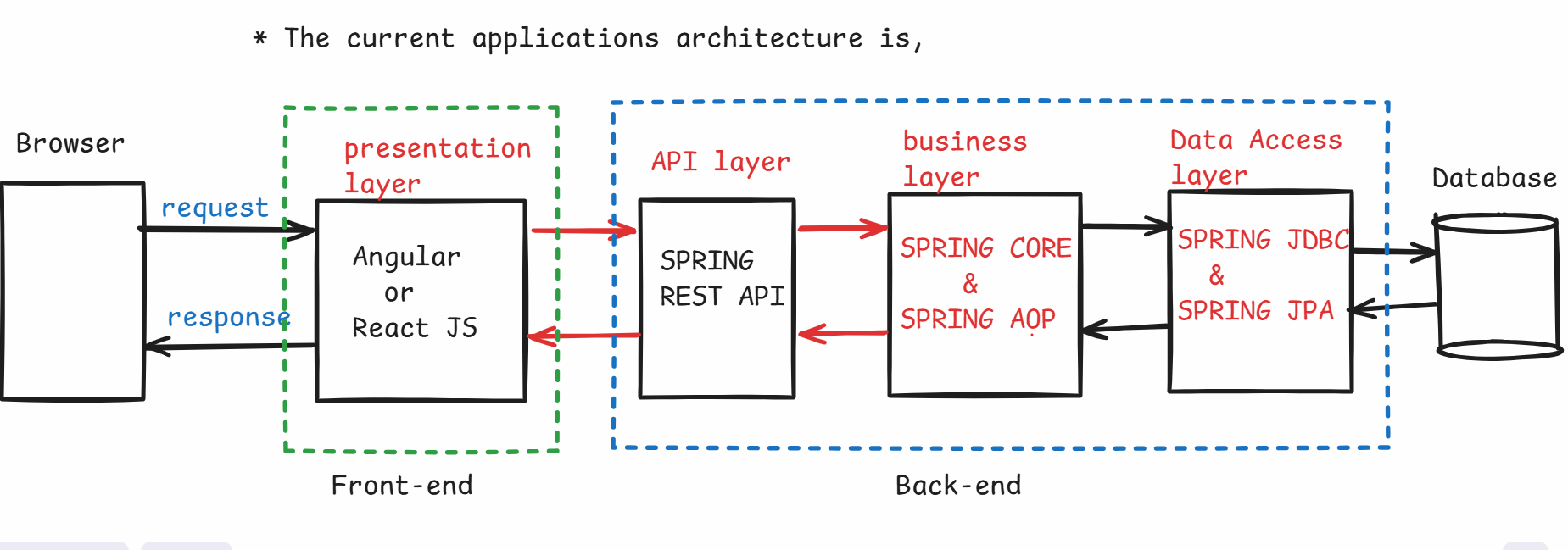












Q) What is IS-A relationship and HAS-A relationship?

A) 1. IS-A relationship is called Inheritance and HAS-A relationship is called Association.

2. For example, Car extends Vehicle, we read it as

Car is-a Vehicle.

3. Association can be Aggregation or Composition.

4. If we create one class object in another class, then there is HAS-A relationship between the classes.

5. For example,

class College {

Department dept = new Department();

}

6. College class has HAS-A relationship with Department class.

7. Aggregation is a weak relationship between the classes and Composition is a strong relationship between the classes.

Dependent class and Dependency class:

* If you create class B object in Class A, then we say that

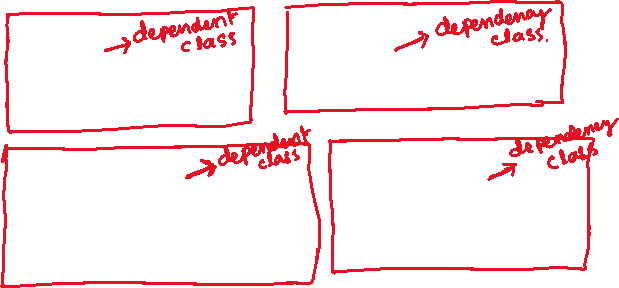
class A is the dependent class and class B is the dependency class.

class A { class B {

B bObj = new B(); //methods

//other methods }

}



class CustomerService { class CustomerRepository

CustomerRepository repo = {

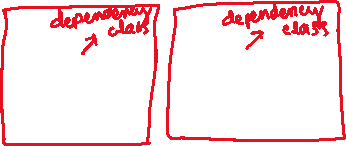
new CustomerRepository(); //methods

//other methods }

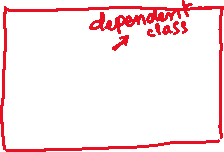
}

* One dependent class can have multiple dependency classes also.

class A { class B { class C {



B bObj = new B(); //methods //methods



C cObj = new C(); } }

//other methods

}

tightly coupled objects:

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

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

class Travel { class Car { class Bike {

Car car = new Car(); //methods //methods

} }

//methods

}

* In the above, Travel class/object is dependent object and Car class/object is dependency object.
* Suppose, Travel object wants to change the dependency object from Car object to Bike object.
* For this, in Travel class, remove the Car object and create Bike object. It means, some code changes are required.
* So, we say that Travel object is tightly coupled with the Car object.

class Travel { class Car { class Bike {

Car car = new Car(); //methods //methods



Bike bike =new Bike(); } }

//methods

}

loosely coupled objects:

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

* If a dependent object wants to change its dependency object, if no code changes are required in the dependent object/class then we can say that dependent object is loosely coupled with the dependency object.
* To provide loose coupling, follow the below rules.

1. create the dependency classes by implementing an interface.
2. create interface reference variable in the dependent class.
3. define a setter method for setting/injecting the dependency object.

ex:

interface Vehicle {

//abstract methods

}

class Car implements Vehicle {

//methods

}

class Bike implements Vehicle {

//methods

}

class Travel {

Vehicle vehicle;

public void setVehicle(Vehicle vehicle)

{

this.vehicle = vehicle;

}

//other methods

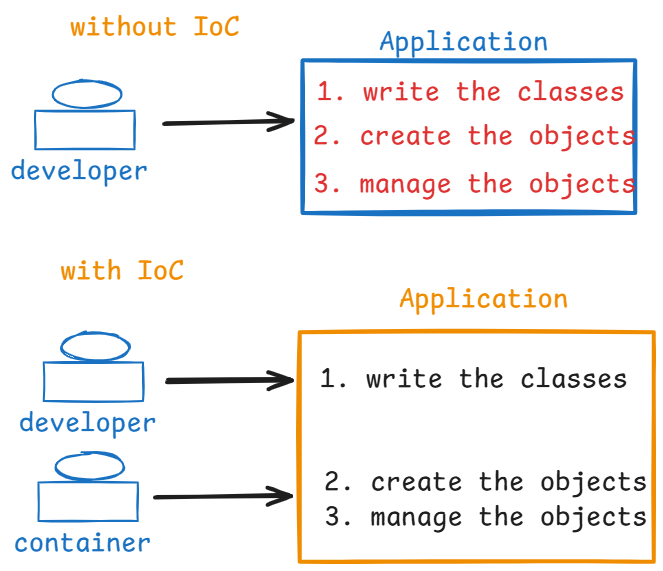
}

* In the above example, we can set the Car object or Bike object to the Travel object, by calling the setter method, without making any code changes. So, the Travel object is loosely coupled with Car object/Bike object.

Inversion of Control(IoC):

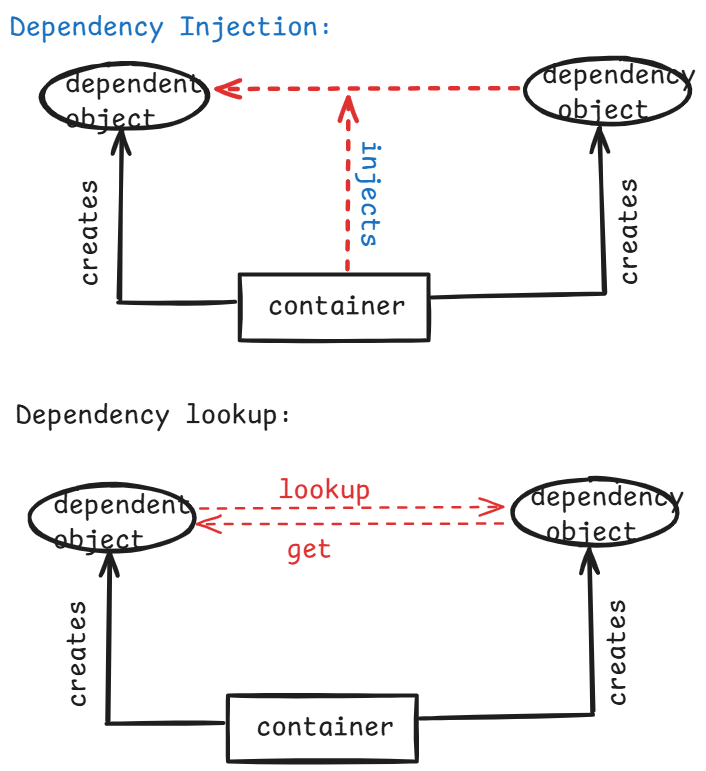
-----------------------

* Inversion of control is a new approach or a design principle, where some responsibilities of a developer are inverted to the another component like a framework.
* The responsibilities are like creating the objects, injecting the dependencies and destroying the objects are inverted to the framework.
* So, instead of a developer is a controlling the objects, the control is inverted to another component like a framework. That’s why it is called inversion of control.



Dependency lookup and Dependency injection:

* Dependency lookup and Dependency injection are two mechanisms two implement the IoC principle. So, we call them as two design patterns.
* In Dependency lookup, the container creates the dependent objects and dependency objects. But, the container doesn’t inject the dependency objects to the dependent objects.
* So, a dependent object should have code internally to search/lookup for its dependency objects and should get them.
* In Dependency injection, the container creates the dependent objects and dependency objects. Also, the container injects the dependency objects to the dependent objects.
* In Spring Framework, the Spring Container is implemented through Dependency Injection pattern.



spring configuration file:

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

* A spring configuration file is an xml file.
* This file is used to provide the meta information/meta data about the spring bean classes to the Spring container.
* A spring configuration file name could be anyname.xml
* The root tag of the xml is, <beans>
* The parent tag is <bean>
* The <bean> is used to configurate a spring bean class.
* If multiple classes are available then you have to use multiple <bean> tags.

config.xml

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

<beans>

<bean id = “sa” class = “com.pack.ServiceA”>

. . . .

. . . .

</bean>

<bean id = “sb” class = “com.pack.ServiceB”>

. . . .

. . . .

</bean>

</beans>

Who is a Spring Container?

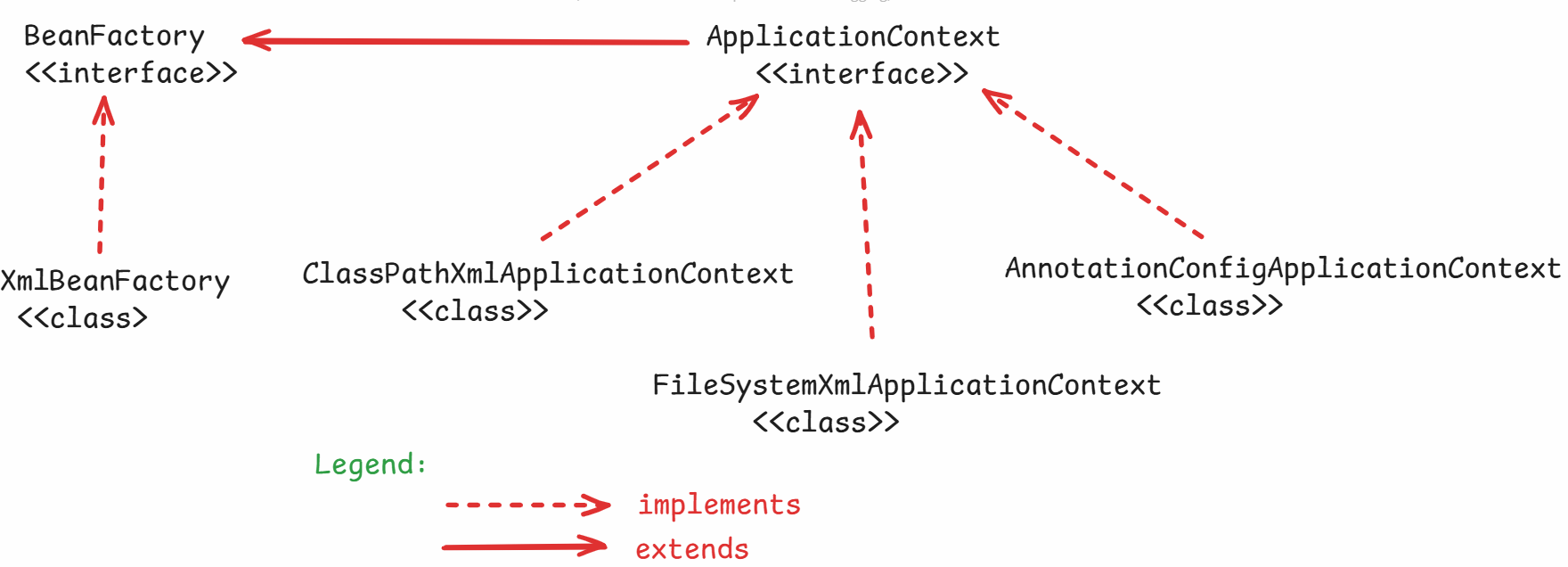
------------------------

* Spring container is nothing but, it is a class provided by spring framework only and it will take care of the responsibilities like, creating the objects, managing the objects, destroying the objects etc..
* In Spring, the implementation of classes of the below interfaces are called as spring containers.
* Spring has provided two types containers.

1. BeanFactory
2. ApplicationContext

* BeanFactory interface was introduced in Spring 1.0 version with an implementation class XmlBeanFactory class.
* ApplicationContext interface was introduced in Spring 1.2 version, it is an extension of BeanFactory and 3 implementation classes are provided.

1. ClassPathXmlApplicationContext
2. FileSystemXmlApplicationContext
3. AnnotationConfigApplicationContext



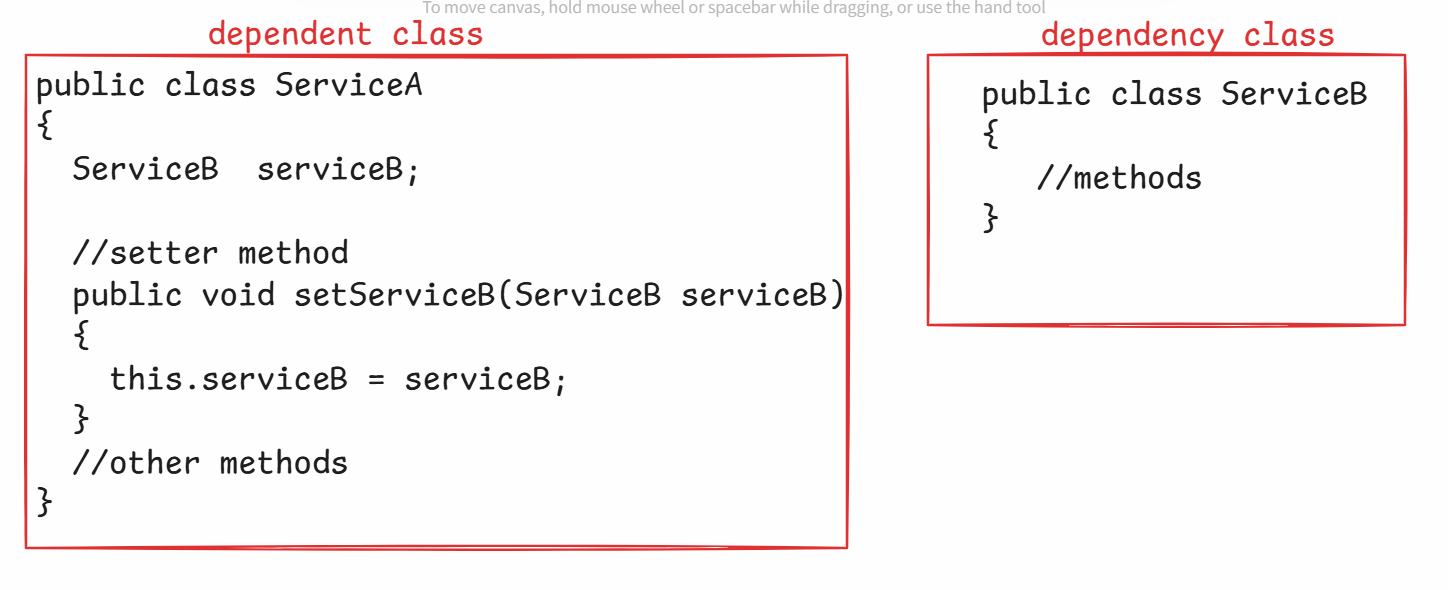
Types of Dependency Injections:

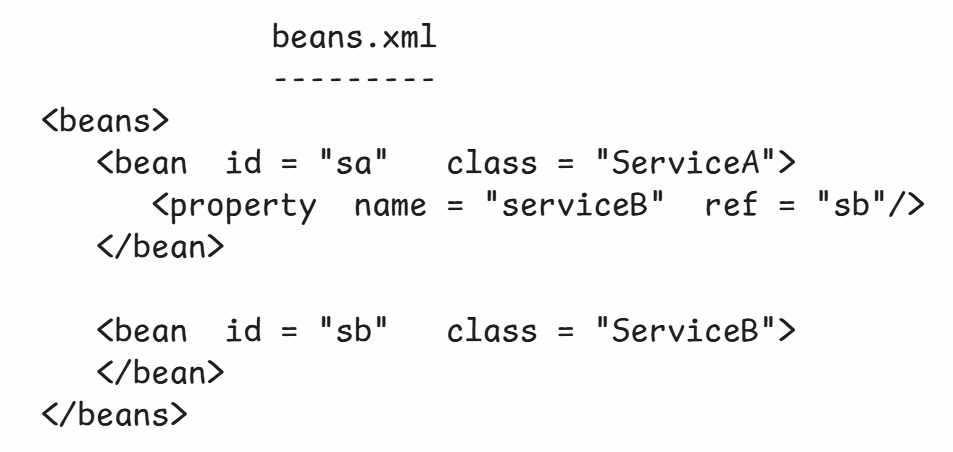
-------------------------------

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

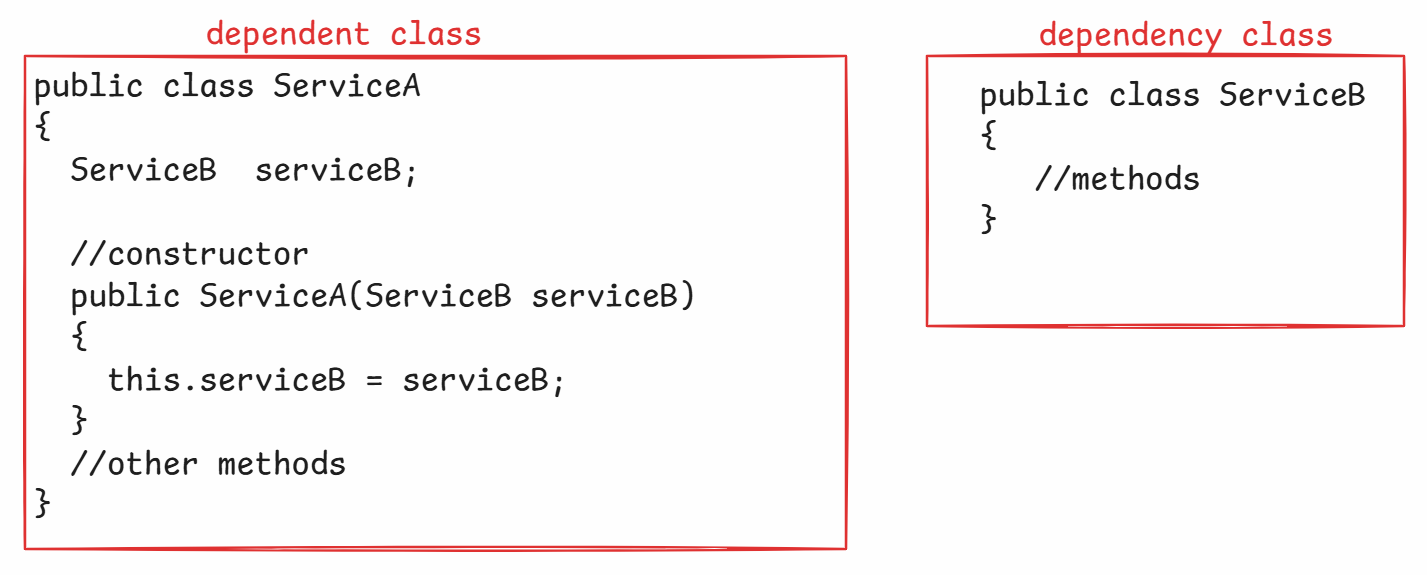
* In setter dependency injection, a dependent object provides a setter method for injecting the dependency object. So, the spring container calls the setter method and injects the dependency.
* In constructor dependency injection, a dependet object provides a constructor for injecting the dependency object. So, the spring container calls the constructor and injects the dependency.
* In <bean> tag, we need to add <property> tag as a child tag, to tell the container that a setter method is provided to inject the dependency object.
* In <bean> tag, we need to add <constructor-arg> tag as a child tag, to tell the container that a constructor is provided to inject the dependency object.

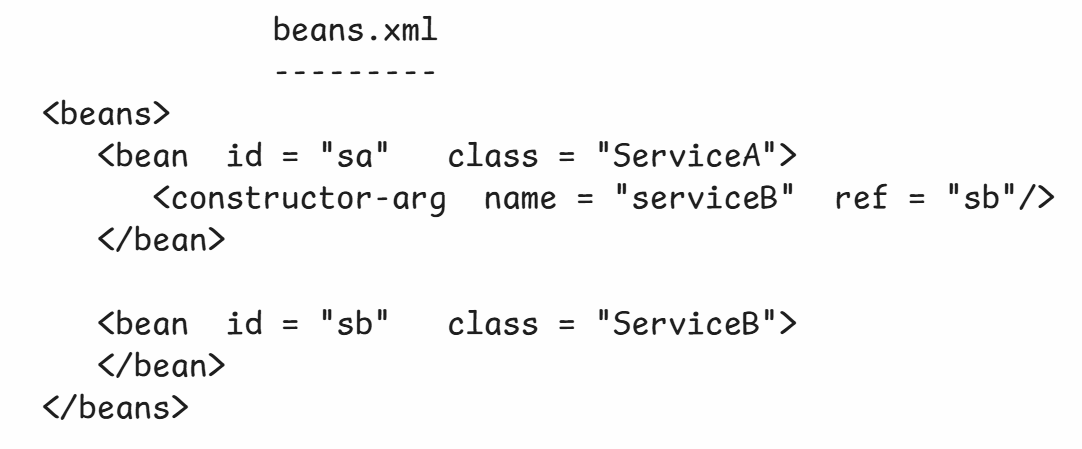
ex1:





ex2:





Maven

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

Build Process of a Java application:

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

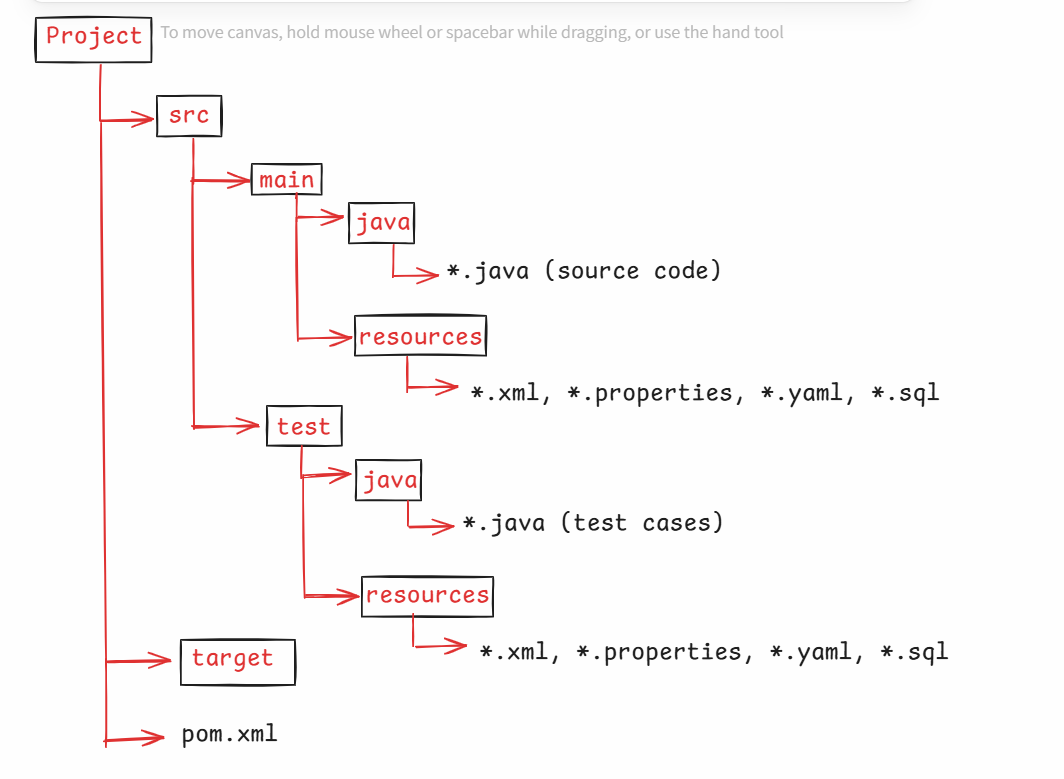
steps:

1. download the required jars to compile the code
2. add the jars to the classpath.
3. compile the source code
4. download the required jars to compile & execute the testcases.
5. add the jars to the classpath.
6. compile the testcases
7. run the testcases.
8. package the application into a jar file.
9. Deploy the project jar file into a server.

* Suppose, after deploying into a server, the test engineers have identified a bug in the application.
* The developer has to modify the code to fix the bug.
* Now, the developer has to repeat the build process steps.
* Suppose, after deploying into a server, a requirement came to add some new features.
* The developer has to modify the code to add the new features.
* Now, the developer has to repeat the build process steps.
* Repeating the build process steps again and again is a complex task for the developers.
* So, to automate this build process, build tools are provided.
* Apache has provided the build tools called Maven and Gradle.

maven project structure:

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



* As a developer, you are not going to setup the project structure manually.
* Maven has provides archetypes to setup the project structure quickly.
* These archetypes are nothing but, they are the templates to create the project structure quickly.

pom.xml(project object model):

* Every maven project contains pom.xml file in the project folder.
* The use of pom.xml file for a developer is to add the dependencies(jars) required for the project.
* In pom.xml file, a developer has to add the required dependencies under <dependencies> tag.
* If you add the dependencies, maven will download them from central repository and will store them into local repository.
* Suppose, if you are developing a spring core application, then you have to add the following dependency in the pom.xml file.

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>6.2.3</version>

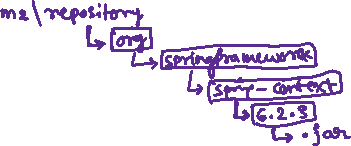
</dependency>

* The local repository will be created in every developers system, when a first maven project is created.
* The location of the local repository is,

C:\Users\Administrator\.m2\repository

* Maven will store the downloaded dependencies in a systamatic way into the local repository.
* First it creates the folders for groupId, then creates artifactId folder, then creates version folders and stores the downloaded jar file in the version folder.

For example:



Maven project parameters:

groupId : It is a unique id of the organization, to identify the projects of that organization.

Mostly, reverse name of the domain name of the organization will be used as group id.

artifactId : It represents the project name

version : version of the project

package : package name of the classes.

Maven commands:

1. compile : compiles the source code
2. test-compile : compiles the test cases
3. test : executes the test cases
4. package : creates the jar file for the project
5. install : installs/stores the jar file into the local repository
6. clean : removes the previously generated artifacts.

* we no need to execute these commands one after the other.
* if we execute package command, then maven completes, compile, test-compile, test and package commands.
* Maven stores the generated artifacts into target folder.
* Before we again build the same project, we clean the previous build.
* when you run clean command, maven deletes all the artifacts from target folder.

Note: If you want to know the spring versions, visit

<https://repo.maven.apache.org/maven2/org/springframework/spring-context/>

First Spring application:

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

1. launch STS IDE 🡪 choose the workspace
2. File 🡪 New 🡪 Maven Project 🡪 Next 🡪 Filter: maven-archetype-quickstart 🡪 choose the groupId: org.apache.maven.archetypes 🡪 next 🡪 Now enter the project parameters :

groupId in.ashokit

artifcatId SpringSetterDemo

version 0.0.1-SNAPSHOT

package com.pack.beans

Finish

1. open pom.xml file, add the below dependecy.

<dependency>

<groupId>org.springframework</groupId>

<artifactId>spring-context</artifactId>

<version>6.2.3</version>

</dependency>

1. expand src/main/java, then package com.pack.beans, then delete App.java
2. Right click on com.pack.beans, create a new class User.

**package** com.pack.beans;

//dependent class

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

PasswordEncoder passwordEncoder;

**public** **void** setPasswordEncoder(PasswordEncoder passwordEncoder) {

**this**.passwordEncoder = passwordEncoder;

}

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

{

System.***out***.println("Login success...");

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

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

}

}

1. Right click on com.pack.beans, create new class PasswordEncoder.

**package** com.pack.beans;

**import** java.util.Base64;

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

//dependency class

**public** **class** PasswordEncoder {

**public** String encodePassword(String password)

{

//creating Encoder object

Encoder encoder = Base64.*getEncoder*();

//converting password into bytes

**byte**[] bytes = password.getBytes();

//encode the password into encoded password

String encodedPassword = encoder.encodeToString(bytes);

**return** encodedPassword;

}

}

1. expand src 🡪 right click on main 🡪 New 🡪 folder 🡪folder Name: resources 🡪 finish
2. Right click on src/main/resources 🡪 New 🡪 File 🡪 filename: beans.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 = "user" class = "com.pack.beans.User">

<property name = "passwordEncoder" ref = "pwd"/>

</bean>

<bean id = "pwd" class = "com.pack.beans.PasswordEncoder">

</bean>

</beans>

1. Right click on src/main/java 🡪new 🡪 package 🡪 com.pack.main 🡪 finish
2. right click on com.pack.main 🡪 New 🡪 class 🡪 name: Main 🡪 select main() checkbox 🡪 finish.

**package** com.pack.main;

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

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

**import** com.pack.beans.User;

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

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

//step1 : start the spring container

ApplicationContext ctx = **new** ClassPathXmlApplicationContext("beans.xml");

//step2: get the bean from the container.

User user = ctx.getBean("user", User.**class**);

//step3: call the method.

user.login("admin", "admin@123");

}

}

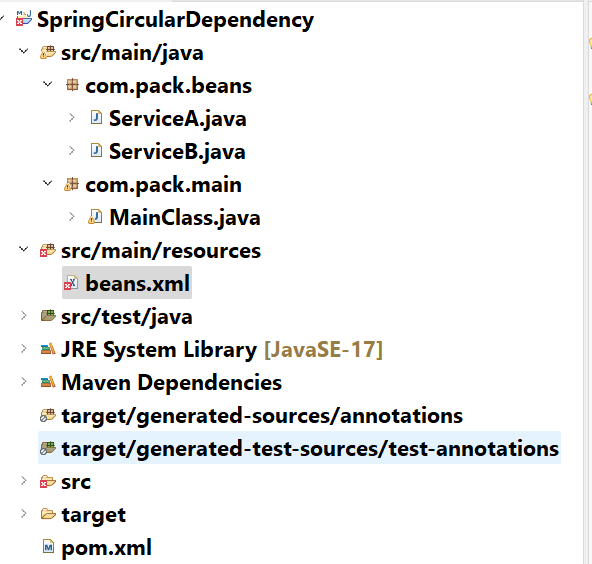
1. right click on project 🡪 RunAs 🡪 Maven Build… 🡪 Goals: package 🡪 Run
2. Right click on project -🡪 refresh
3. Right click on Main.java 🡪 RunAs 🡪 Java Application.

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");

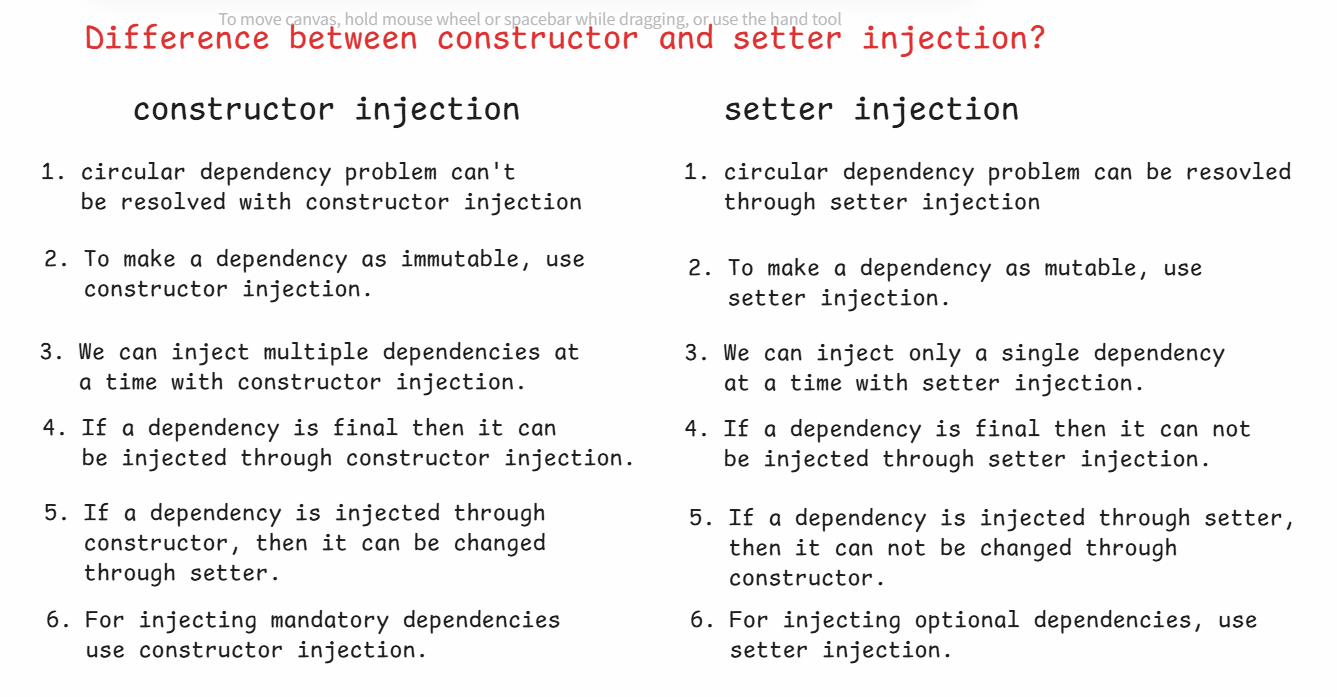
Object obj = context.getBean("sa");

ServiceA servA = (ServiceA)obj;

servA.m1();

}

}



Spring Core 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

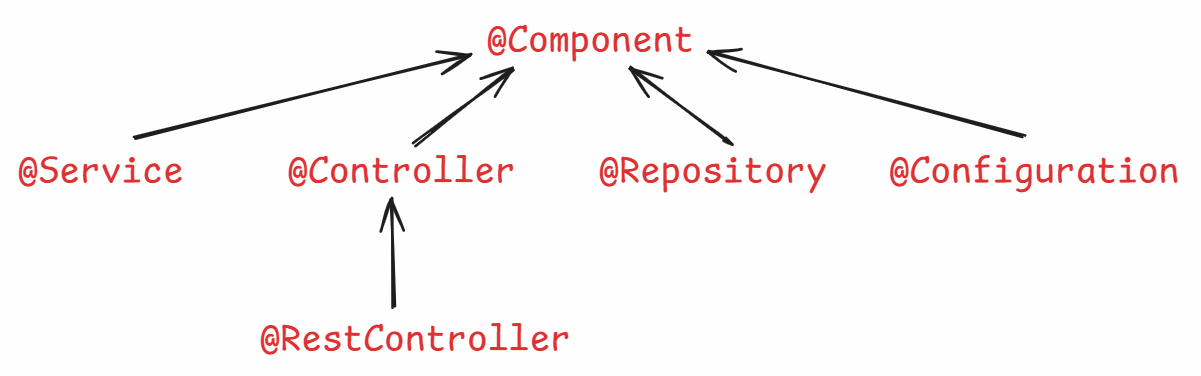
public JdbcTemplate jdbcTemplate() {

//instantiating and initializing JdbcTemplate obj.

return obj;

}

}



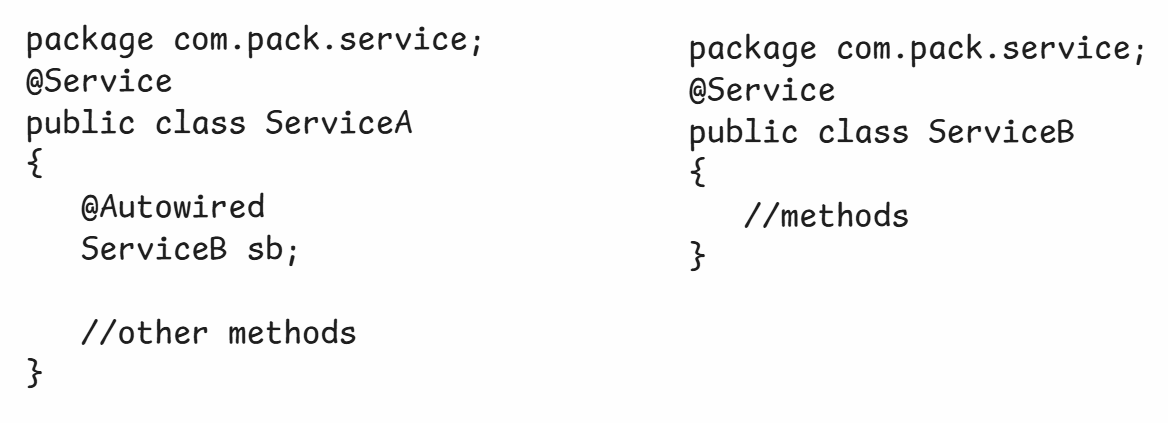
* The above annotations in spring are called stereotype annotations.

Q) what is stereotype annotation?

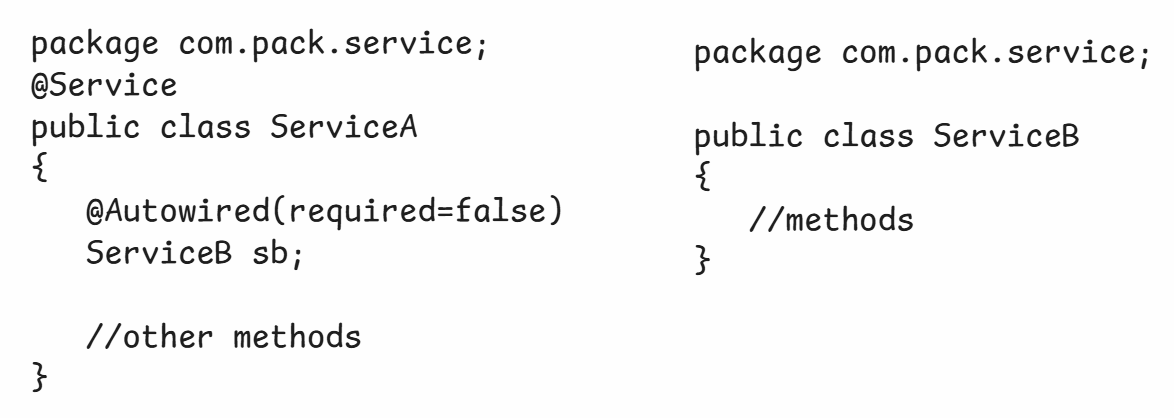
A) stereotype represents a type which tells the role of a class in an application.

@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.

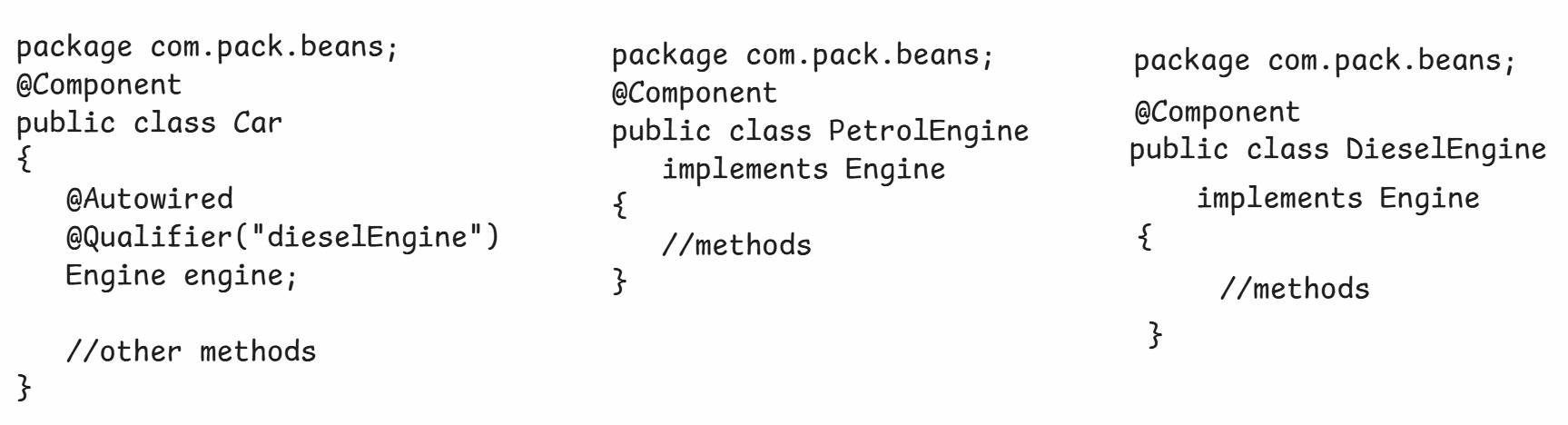


* Suppose, if spring can not find a matching bean for autowiring, then throws UnsatisfiedDependencyException.
* To make autowiring a dependency object as optional, we have to provide a parameter required = false, to the @Autowired annotation.



@Qualifier:

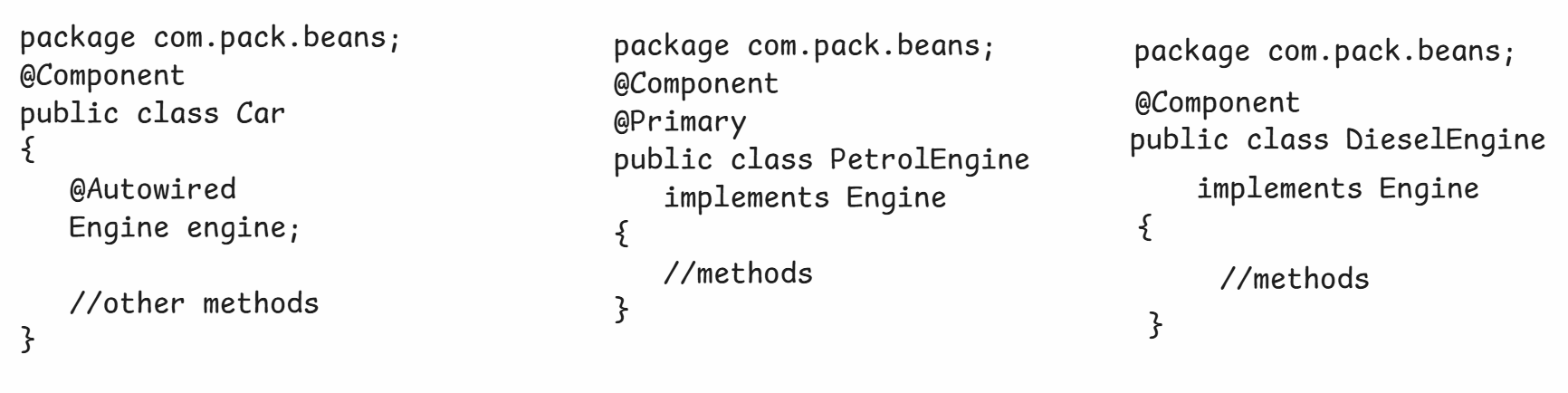
* If multiple matching beans are found by the spring, for autowiring then because of ambiguity, again spring throws UnsatisfiedDependencyException.
* To avoid this exception, we can specify a specific bean name with @Qualifier annotation.



@Primary:

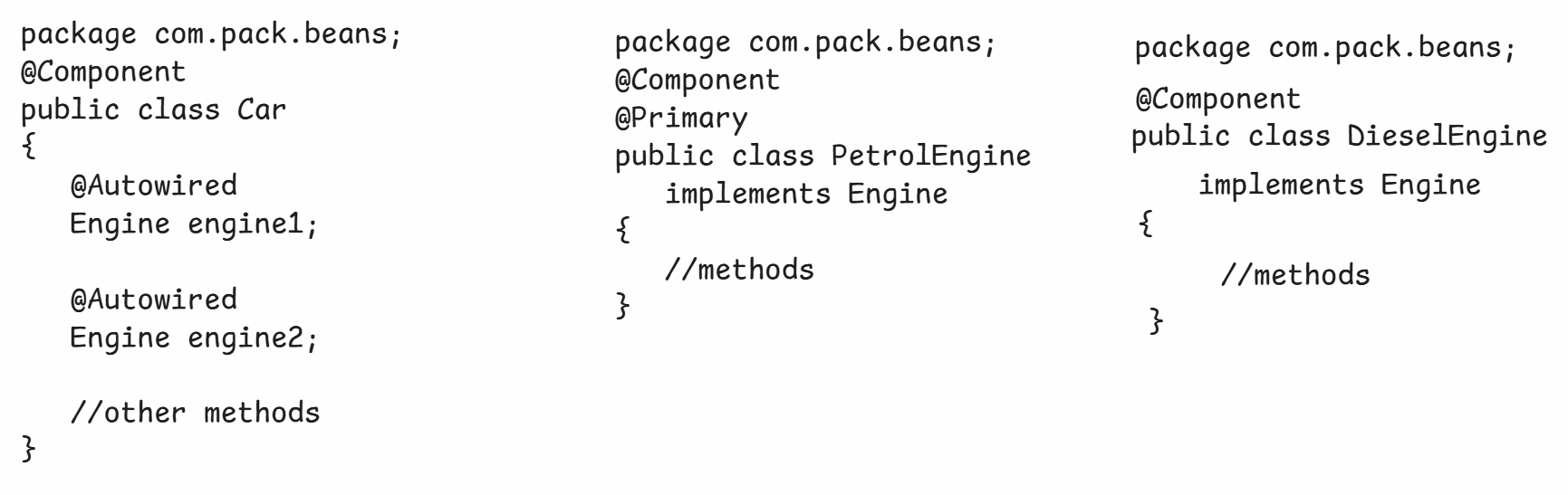
* This annotation can be used at class-level or at method-level.
* This annotation can be used to specify that a bean is a default choice for dependency inection, when mulitple beans are found of the same type by the spring container.
* Suppose, at injection point, if you want to change the bean for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

ex1:



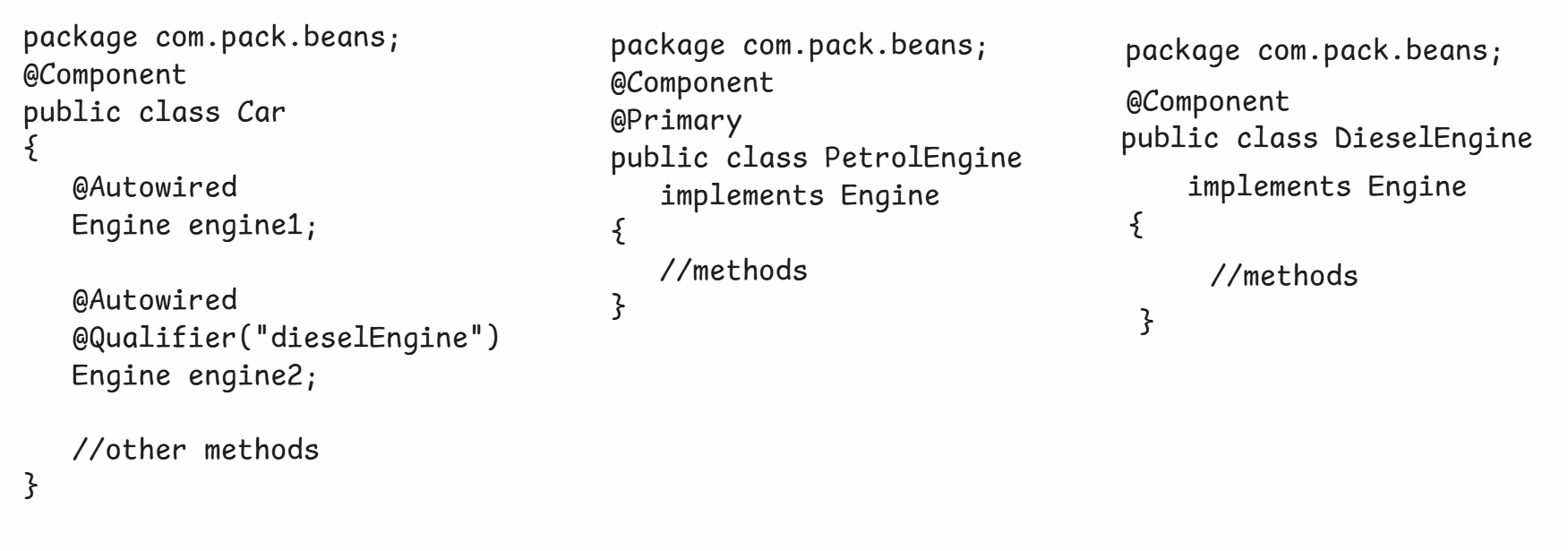
* Here, the spring injects PetrolEngine object as a dependency object to the Car object(dependent object).

ex2:

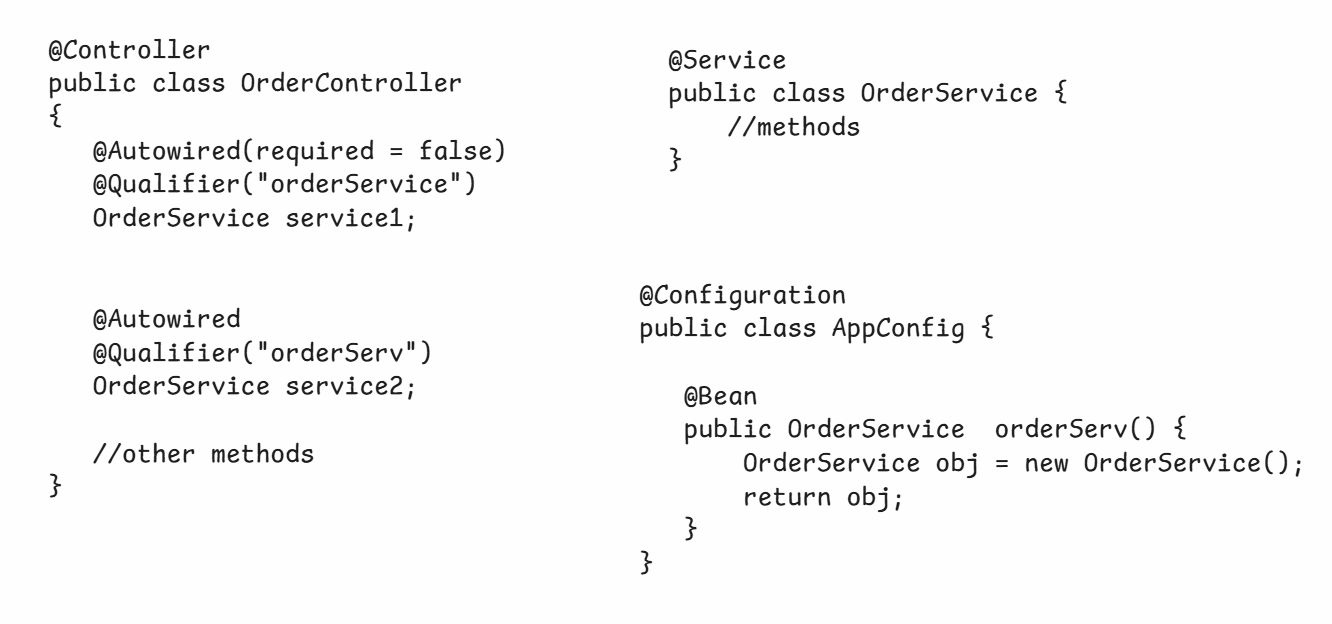


* Here, the spring injects PetrolEngine object as dependency object for both reference variables(engine1, engine2).

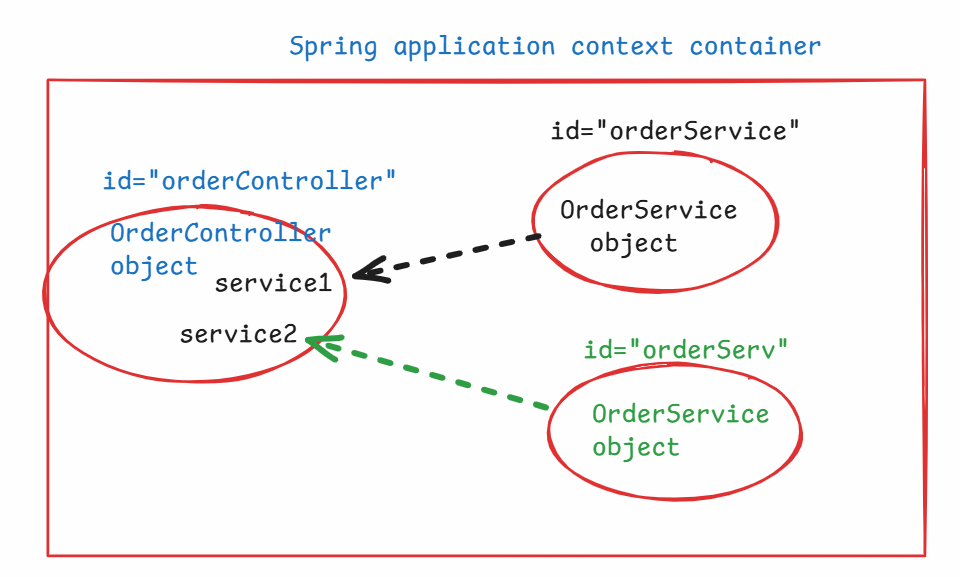
ex3:

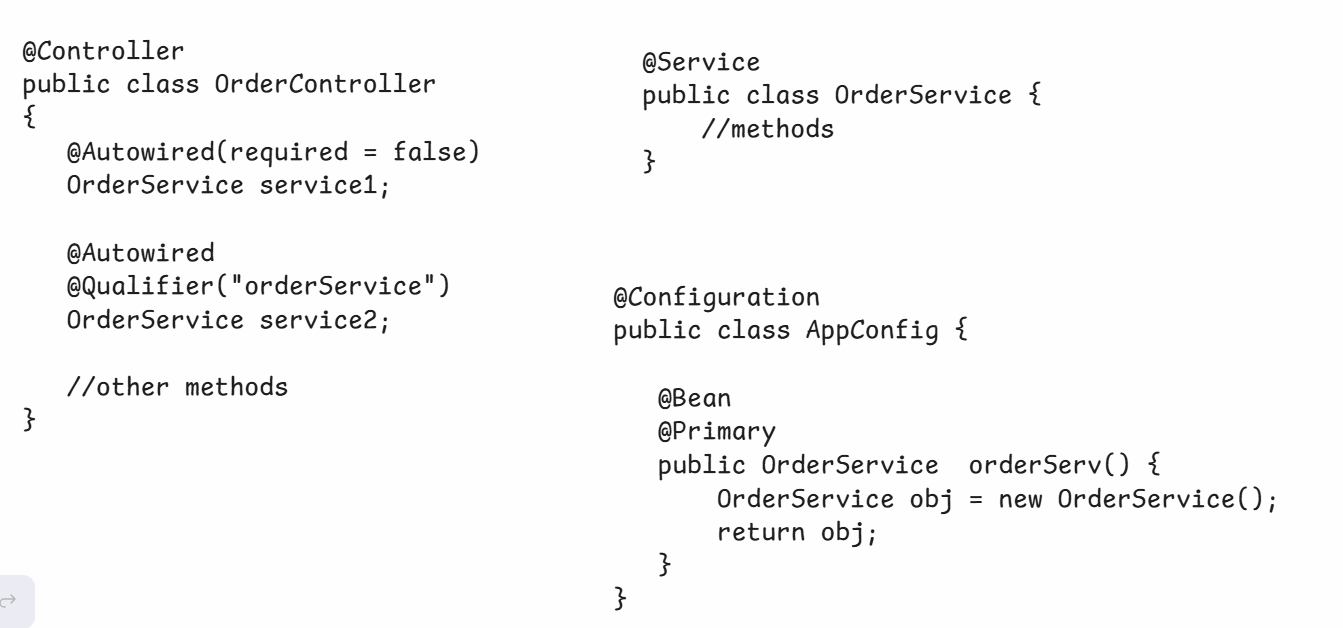


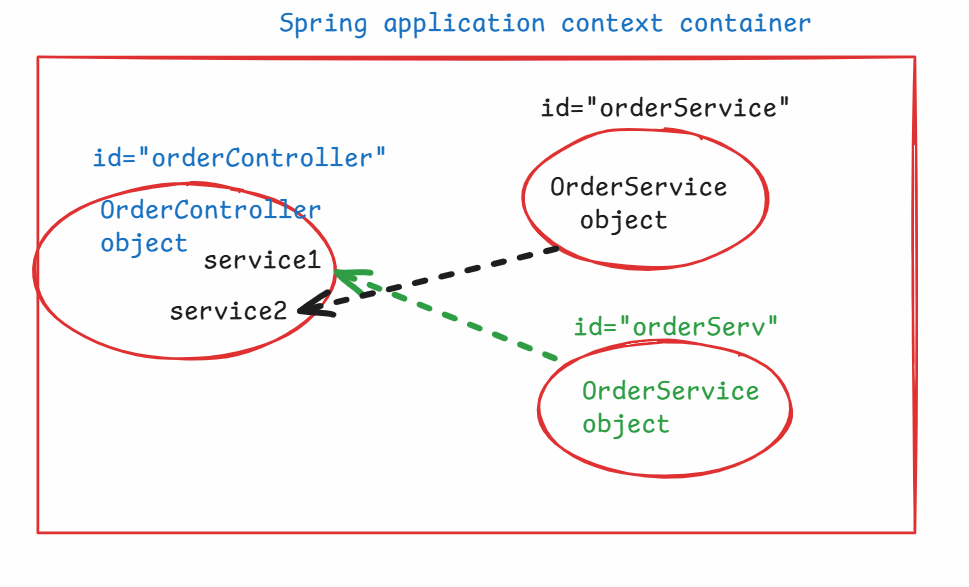
* Here, the spring injects PetrolEngine object as a dependency object for engine1 and DieselEngine object as a dependency object for engine2.
* @Primary is a default choice and @Qualifier is an explicit choice.
* If both are used then @Qualifier gets more priority.
* @Qualifier is used at injection point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

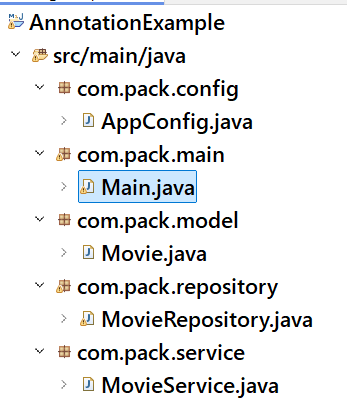












Movie.java

---------

**package** com.pack.model;

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

**private** **final** String name;

**private** **final** **int** year;

**public** Movie(String name, **int** year) {

**super**();

**this**.name = name;

**this**.year = year;

}

**public** String getName() {

**return** name;

}

**public** **int** getYear() {

**return** year;

}

}

MovieRepository.java

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

**package** com.pack.repository;

**import** java.util.ArrayList;

**import** java.util.List;

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

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

**import** com.pack.model.Movie;

@Repository

**public** **class** MovieRepository {

@Autowired

ArrayList<Movie> moviesList;

**public** List<Movie> findByYear(**int** year) {

List<Movie> foundList = **new** ArrayList<>();

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

**if**(movie.getYear() == year)

foundList.add(movie);

}

**return** foundList;

}

}

MovieService.java

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

**package** com.pack.service;

**import** java.util.List;

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

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

**import** com.pack.model.Movie;

**import** com.pack.repository.MovieRepository;

@Service

**public** **class** MovieService {

@Autowired

MovieRepository repository;

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

List<Movie> lst = repository.findByYear(year);

**for**(Movie m : lst) {

System.***out***.println("Name : " + m.getName());

System.***out***.println("Year : " + m.getYear());

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

}

}

}

AppConfig.java

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

package com.pack.config;

import java.util.ArrayList;

import org.springframework.context.annotation.Bean;

import org.springframework.context.annotation.ComponentScan;

import org.springframework.context.annotation.Configuration;

import com.pack.model.Movie;

@Configuration

@ComponentScan("com.pack")

public class AppConfig {

@Bean

public ArrayList<Movie> mList() {

ArrayList<Movie> arrList = new ArrayList<Movie>();

arrList.add(new Movie("Chaava", 2025));

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

arrList.add(new Movie("Yodha", 2024));

arrList.add(new Movie("Kalki", 2024));

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

return arrList;

}

}

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.MovieService;

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

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

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

MovieService service = ctx.getBean(MovieService.**class**);

service.showMovies(2025);

}

}

@Value: @Value in Spring is an annotation used for injecting values into Spring beans. It allows you to inject values from properties files.

ex:

@Component

public class MyBean {

@Value("Hello, Spring!")

private String message;

@Value("42")

private int number;

public void printValues() {

System.out.println(message); // Output: Hello, Spring!

System.out.println(number); // Output: 42

}

}

@PropertySource:

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

@PropertySource is a Spring annotation used to load properties files into the Spring Environment. It allows you to specify external configuration files that contain key-value pairs, which can then be injected into Spring beans using @Value.

ex:

config.properties

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

app.name=MyCustomApp

app.version=2.5.1

AppConfig.java

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

@Configuration

@PropertySource(“config.properties")

public class AppConfig {

}

MyApp.java

----------

@Component

public class MyApp {

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

private String appName;

@Value("${app.version}")

private String appVersion;

public void printAppDetails() {

System.out.println("App Name: " + appName); //op: MyCustomApp

System.out.println("App Version: " + appVersion); //op: 2.5.1

}

}