* API & Micro Service *

* API : (Introduction)

→ API Stands for Application Programming Interface.

→ API's let Software applications communicate with other websites and applications for additional functionality.

Ex: Payment Gateway

phonepay, paytm, Google pay

Defination:

→ APIs are Set of rules & Specifications that allow different S/w applications to interact with each other.

Purpose:

→ They act as a contract, defining how one application can request & receive data (or) functionality from another.

Ex: A payment Gateway might use an API to process payments.

* MicroServices:

Definition

→ MicroServices is an architectural style where an application is built as a collection of small, independent and loosely coupled services, each responsible for a specific business capability.

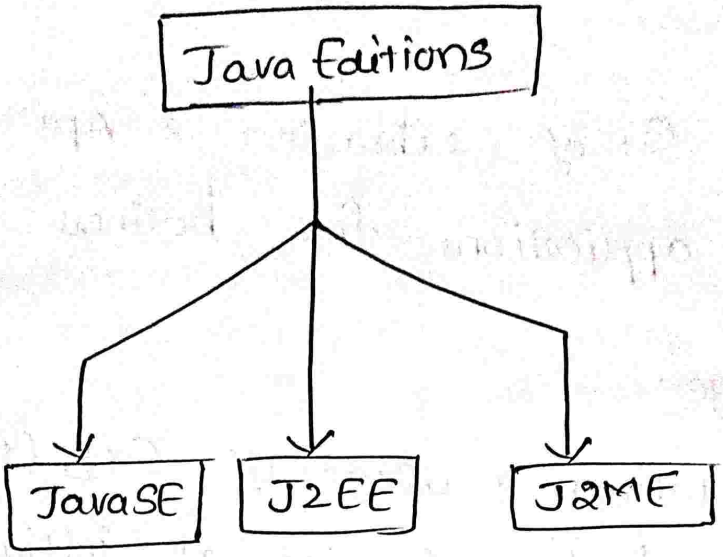
Purpose

→ This approach allows for greater flexibility, Scalability and independent deployment of different parts of an application.

Ex:

→ An e-commerce application might be broken down into microservices for user management, product catalog, order processing and payment processing. Each of these services would expose APIs to communicate with other services.

* Java Editions :



(i) Java Standard Edition : (Java SE)

→ use to develop Standalone applications.

Ex: Desktop (or) Laptop

↳ Calculator Application

↓

Standalone Application

(or) nothing but.

GUI Application

(ii) Java Enterprise Edition (J2EE)

→ use to develop web-App that run on Servers like facebook, Instagram.

(iii) Java Micro Edition (J2ME)

→ use to build applications that run across all basic & Smartphones.

Advantage:

→ It's a core Java with powerful Set of libraries & API's.

→ using those Set of libraries & API's developers can build applications for business computing.

Dis-Advantage:

→ Implementation & usage of EJB (Enterprise Java Bean), is highly complex at initial version of J2EE.

→ poor performance of Entity bean.

Introduction to Spring : (Why Spring)

→ Spring is one of the most popular framework in the java Ecosystem.

→ It is used to build robust, Scalable and maintainable java applications, especially enterprise applications

→ Here's why Spring is so popular in Java development, explained in a simple way:

1. Simplifies Development (Less code, More work)

→ Spring provides a lot of features out of the box, So developers don't have to write everything from Scratch.

→ It uses concepts like Dependency injection (DI) to manage objects automatically - Saving time & Effort.

2. Flexible and Modular

→ Spring is made up of multiple modules (like Spring Core, Spring MVC, Spring Boot, Spring Security etc)

→ you can use only what you need, instead of the entire framework.

3. Support Modern practices :

→ Works great with Rest API's, micro Services & cloud Applications.

→ Spring Boot (apart of Spring) helps you create Stand alone applications quickly with minimal configuration.

4. Built-in Security :

→ Spring Security helps you Secure web applications easily with features like login, ~~rules~~ password encryption, etc authorization etc.

5. Loose coupling with Dependency injection :

→ Objects (beans) are connected using interfaces, not hard - coded - this makes your code flexible, testable & maintainable.

6. Easy Testing :

→ Spring makes unit & integration testing easier with its testing Support.

7. Wide Community & Ecosystem :

→ Spring is widely used in industry, so you will find tons of tutorials, documentation & community Support.

* Spring Boot Introduction:

→ Spring Boot is a framework developed on top of core Spring framework.

→ The main aim of Spring Boot is to let developers to create Spring production grade applications and Services with very less effort.

→ Spring Framework, it includes writing many XML configurations, Server Setting, adding dependencies etc.

→ Spring Boot comes with inbuilt Server, we no longer have to use any external Servers like Tomcat, Glass-fish (or) anything else, So don't need to deploy WAR-files.

→ The main disadvantage is, it will be little tough to migrate existing Spring enterprise applications to to Spring Boot.

→ ~~Remember~~ we have to use either Maven (or) Gradle build tool to work with Spring Boot.

→ Spring Boot provides Command line interface tool to develop /test the Spring Boot applications from the Command prompt easily.

* Spring Framework Introduction:

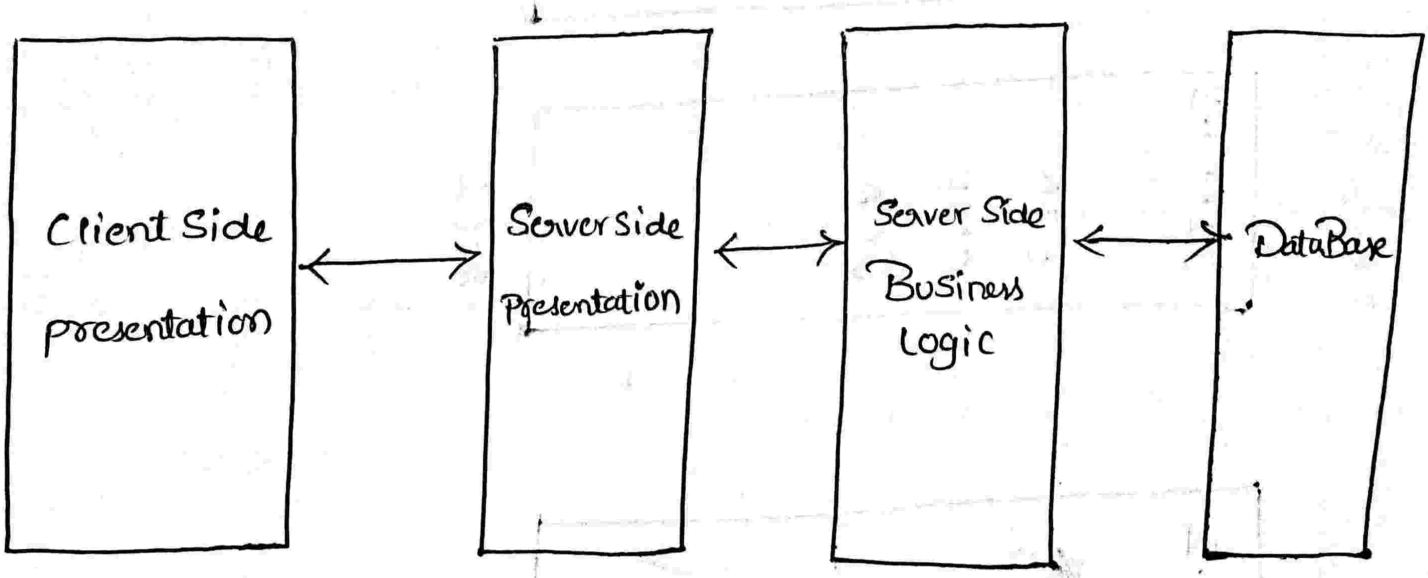
Why Should we use Spring Framework:

→ Very popular Framework for building Java/web Applications.

→ Intially A Simpler And Lightweight Alternative to J2EE.

→ provides A Large Number of Helper classes ...makes things Easier.

What About J2EE:



Versions (J2EE)

* EJB V1 & EJB V2 - Complexity :

→ Early version of EJB (V1 and V2) were extremely complex.

→ Multiple Deployment Descriptors.

→ Multiple Interfaces

→ Poor Performance of Entity Beans.

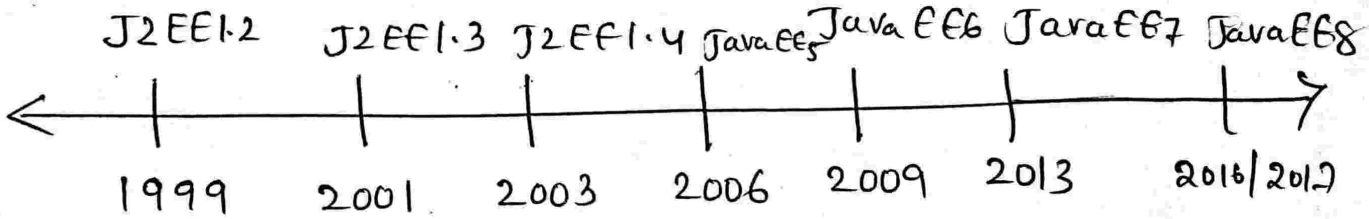
J2EE Development without EJB :

→ Rod Johnson (founder of Spring).

→ Book : J2ee Development without EJB

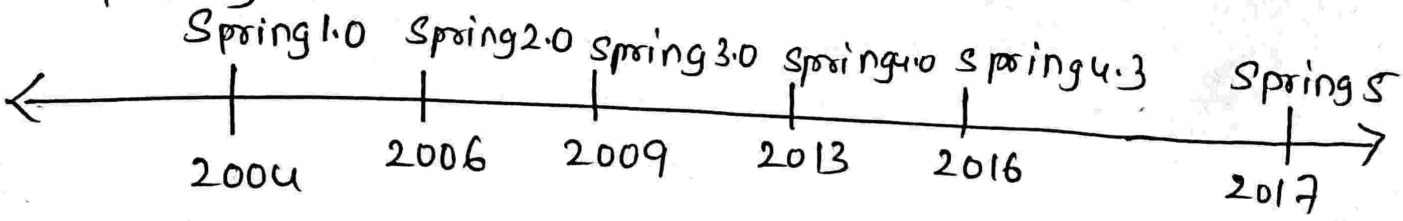
→ Book : J2ee Development with the Spring Framework

Release Time Line :



Java EE

Spring

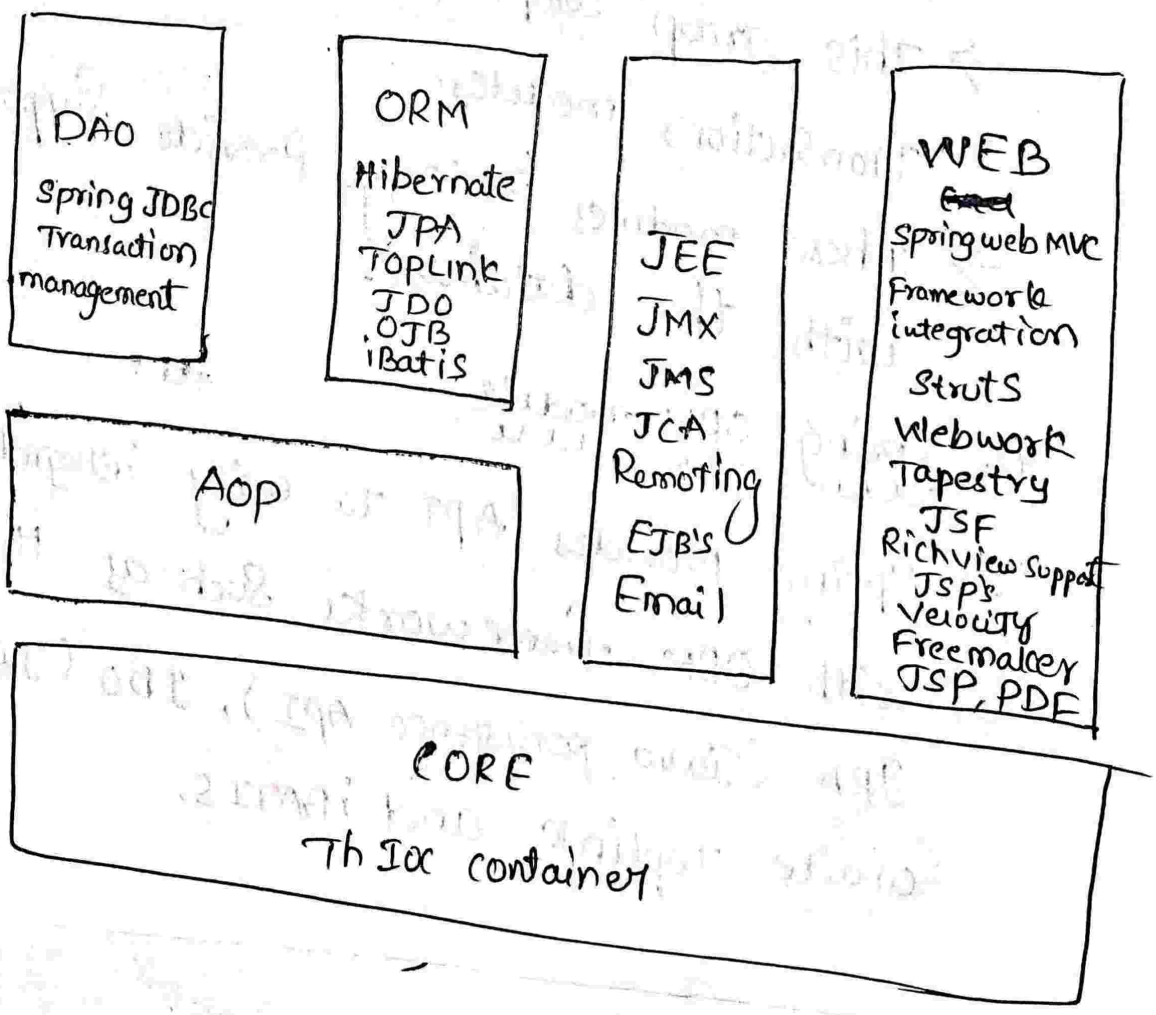


* Spring Modules:

→ Actually in Spring 1.x, the framework has divided into 7 well defined modules. But in 2.x framework is divided into 6 modules only.

1. Spring Core Module
2. Spring Context [J2EE].
3. Spring DAO Module [Spring JDBC]
4. Spring ORM module.
5. Spring Aop [Aspect oriented programming]
6. Spring Web-MVC module.

Spring Architecture



1. Spring Core Module:

→ Spring Core Module is the base for all modules and very important. Let us see one by one module in depth.

→ The Spring core container contains core, beans, context and expression Language (EL) modules.

2. Spring Context [J2EE]

→ This module supports internationalization, EJB, JMS Basic Remoting.

3. Spring DAO Module:

→ This group comprises of JDBC, ORM, OXM, JMS and Transaction modules.

→ These modules basically provide Support to interact with the database.

4. Spring ORM module:

→ Spring provides API to easily integrate Spring with ORM frameworks such as Hibernate, JPA (Java persistence API), JDO (Java Data Objects), oracle Toplink and iBATIS.

5. Spring AOP [Aspect oriented programming]

→ These modules Support aspect oriented programming implementation where you can use Advices, pointcuts etc. to decouple the code.

→ The aspects module provides Support to integration with AspectJ.

→ The instrumentation module provides Support to class instrumentation and class loader implementation.

6. Spring WEB-MVC module:

→ This group comprises of web, web-Servlet, web-Struct and web-portlet.

→ These modules provide Support to create web-application.

* Configuring Ioc container using Java-based configuration in Spring:

→ Instead of using XML configuration, Spring allows you to configure beans using @Configuration and @Bean annotations in Java classes.

→ This approach is type-safe, refactor friendly and wide used in modern Spring applications.

Steps to configure Ioc container with Java Config:

(i) Create a Bean class:

```
import java.lang.*;
public class Student
{
    private int id;
    private String name;
{
    this.id=id;
    this.name = name;
}
Public void display()
{ S.O.pln ("Student ID :"+id + ",Name:"+name);
}
}
```

(ii) Create a configuration class:

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;

@Configuration // Marks this class as configuration for
                Spring Ioc container

Public class AppConfig
{
@Bean // Tells Spring to create a bean and manage it
        inside Ioc container

Public Student studentBean()
{
return new Student (101, "Paramesh");
}
}
```

(iii) Load IOC Container using AnnotationConfigApplicationContext.

```
Public class MainApp
{
  Public Static void main (String args[])
    {
  //Create IOC container and load AppConfig

ApplicationContext context = new AnnotationConfigApplication
                             Context(AppConfig.class);

    //Get bean from container

Student Student = Context.getBean (Student.class);

      Student.display();
    }
}
```

Output:

Student ID : 101, Name : Paramesh

- @Configuration -> Defines a class as Spring Configuration class (like XML).

- @Bean -> Defines a method that returns an object managed by Spring IoC.

- AnnotationConfigApplicationContext -> Loads beans defined in Java-based Configuration.

- Spring IoC Container manages creation, wiring and lifecycle of these beans.

* Introduction to Dependency Injection:

→ Dependency Injection is a design pattern. Used in Software development to achieve loose coupling between classes and their dependencies.

→ A dependency is an object that another object needs to function.

Example

```
import java.lang.*;

Class Car
  {
    Engine engine = new Engine();
  }
```

Note:

↳ Here, Engine is a dependency of Car.

Problem without Dependency Injection:

→ If we create dependencies inside a class directly, the class becomes tightly coupled to that specific dependency implementation.

↳ Hard to change dependency (Example: Petrol Engine → Diesel Engine)

↳ Reduces flexibility.

Dependency Injection Concept:

→ Instead of a class creating its own dependencies we inject them from outside (via constructor, method, or framework).

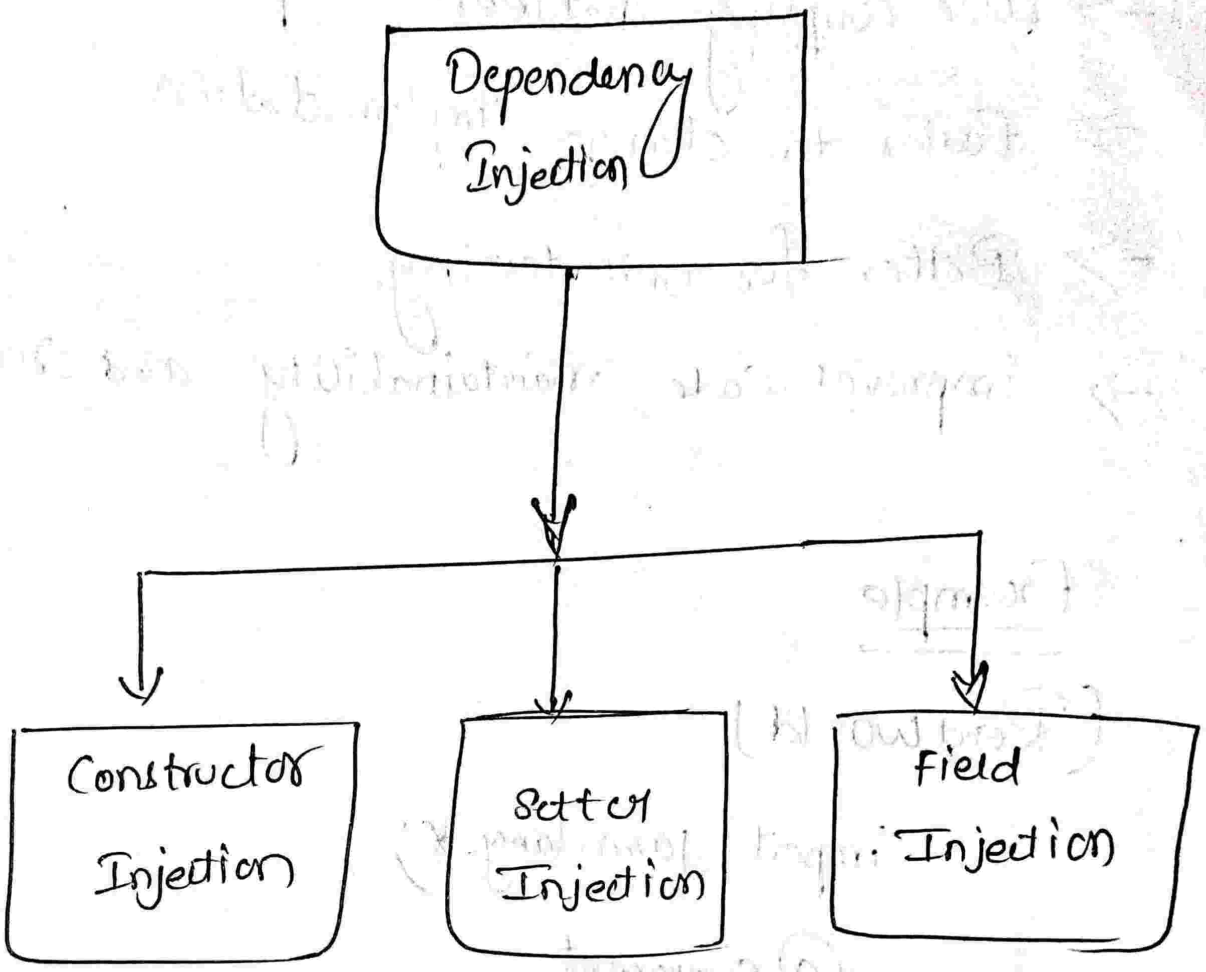
Example with Constructor Injection:

```
import java.lang.*;

Class Car
  {
  private Engine engine;

    Car(Engine engine)
      {
        this.engine = engine;
      }
  }
```

NOTE:

→ Now, Car does not create Engine – Someone else provides it.

Types of Dependency Injection:



Constructor Injection:

→ Dependencies are provided via the Constructor.

Setter Injection:

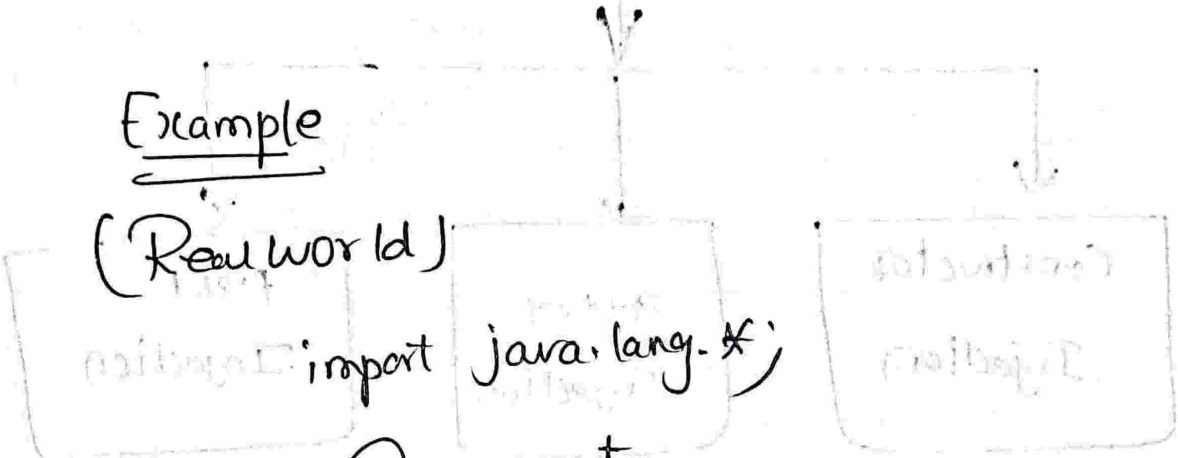
→ Dependencies are set via public setter methods.

Field Injection:

→ Dependencies are injected directly into fields (often used in Spring with @Autowired).

Benefits of Dependency Injection :

→ Loose coupling between components.
→ Easier to change implementation.
→ Better for unit testing.
→ improves code maintainability and readability.



Example

(Real world)

```
import java.lang.*;
@Component
Class Car
{ private final Engine engine;
@Autowired
Car(Engine engine)
{
this.engine;
}
}
```

Note

→ Spring automatically injects the Engine bean into Car.

* Constructor Injection:

→ Constructor injection in Spring Framework is a type of Dependency injection where the Spring IOC Container injects dependencies into a class via its constructor.

→ Constructor injection means passing dependencies to a class through its constructor at the time of object creation.

Example

Step 1: Create a dependency class

Java

```
public class Student
{
    private String name;
    public Student(String name)
    {
        this.name = name;
    }
    public void display()
    {
        System.out.println("Student name is: "+name);
    }
}
```

Step 2: configure in beans.xml

xml

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

<bean id = "StudentBean"  class = "com.example.Student">

    <constructor-arg value = "Kalam" />

    </bean>

    </beans>
```

Step 3:

Java

```
ApplicationContext context =

        new ClassPathXmlApplicationContext("beans.xml");

Student student = (Student) context.getBean("StudentBean");

        student.display();
```

Advantages

→ immutability - once the object is created, fields can't be changed.

→ Promotes good design (helps in writing testable & maintainable code).

Method Injection :

→ Method Injection is a type of Dependency injection in which Spring injects dependencies using Setter methods (or) Custom methods after the object is Constructed.

Example

Java

```
public Class Student
        {
          private String name;
        public void setName(String name)
            {
               this.name = name;
            }
          public void display()
            {
        System.out.println("Name:" +name);
            }
        }
```

XML Configuration (beans.xml):

xml

```
<bean id="StudentBean" class="com.example.Student">
    <property name="name" value="Kalam" />
</bean>
```

* Setter Injection:

→ Setter Injection is one of the Dependency (DI) techniques in the Spring Framework.

→ In this approach, spring injects the dependency (objed) into another objed using setter methods of the dependent class.

How it works:

→ Define a class that depends on another class (dependency)

→ provide a Setter method in the dependent class for that dependency.

→ configure the dependency injection either using XML configuration (or) Annotations.

Example with XML configuration:

Dependency class:

```
public class Student
{ private String name;
  public void setName(String name)
  { this.name = name;
  }
  public void displayInfo()
  { S.o.pln("Student name:" +name);
  } }
```

Spring configuration (applicationContext.xml)

```
<beans xmlns="http://www.springframework.org/schema/be
  xmlns:xsi="http://www.w3.org/2001/XMLSCHEMA-instance
  ....>

  <bean id="studentBean" class="com.example.Student">

    <property name="name" value="kalam"/>

      </bean>
    </beans>
```

Main class:

```
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.ClassPathXml
                                    ApplicationContext;
Public Class MainApp
 {
    Psvm (String arg[])
    {
ApplicationContext context = new
  ClassPathXmlApplicationContext("applicationContext.xml");

Student student = (Student) context.getBean("StudentBean");
      Student.displayInfo(); } }
```

Advantages of Setter Injection:

→ More flexible: you can change dependencies without changing the constructor.

→ Allows partial dependency (you can set only required properties).

→ Makes the bean easier to read and maintain.

Dis-advantages

→ object may remain incomplete if dependencies are not set (null values).

→ Not good for mandatory dependencies (better to use Constructor injection there).

* Auto-Scanning:

→ Auto Scanning in the Spring Framework refers to the automatic detection and registration of Spring managed components (like @Component, @Service, @Repository and @Controller classes) from the classpath.

How it works:

→ Spring Scans Specific packages for classes annotated with Stereotype annotations (@Component, @Service etc)

→ It automatically creates beans for those classes & adds them to the Spring Application Context.

→ You dont need to manually configure each bean.

Required Annotations

→ @Component — Generic Stereotype for any Spring-managed component.

→ @Service — Specialization of @Component for Service layer.

→ @Repository - For DAO layer. (Data Access Object)

→ @Controller - For weblayer (Spring MVC).

Example

1. Java class

```
@Component
public class MyService
{
public void doSomething()
{
System.out.println("Doing Something...");
}
}
```

2. Configuration

Java

```
@Configuration
@ComponentScan(basePackages = "com.example.myapp")
public class AppConfig
{
}
```

→ In this setup, Spring will scan the com.example.myapp package, find MyService, and register it as a bean automatically.

Advantages:

→ Reduces boilerplate code.

→ improve modularity & readability.

→ Helps manage large Applications Efficiently.

without Auto Scanning (Boilerplate code version)

Java

```
public class MyService
        {
        public void work()
          { System.out.println("working....");
          }
        }
```

Java

```
@Configuration
    public class AppConfig
          {
        @Bean
        public Myservice myservice()
            {
            return new MyService(); // This is Boiler code
            }
          }
```

→ Here, you need to manually write a @Bean method for each class. This repeated code is

Example

called Boilerplate code.

with AutoScanning (without Boilerplate code)

Java

```
@component
public class MyService
    {
    public void work()
      {
      System.out.println("working...");
      }
    }
```

Java

```
@configuration
@componentScan(base packages = "com.example")
    public class AppConfig
        {
            // No need to manually define beans
        }
```

→ Now Spring will automatically detect & register MyService as a bean. So you avoid