SPRING CORE

* What is spring ?

1) Spring is a dependency injection framework to make java application loosely coupled.

2) Spring frame work makes the easy development of Java EE application. It provides various modules such as spring mvc, spring jdbc, spring security, spring core this makes development easy.

3) It was developed by Rod Johnson in 2003

* What is Dependency Injection ?

1) It is a design pattern.

```
class Ramu                          class Geeta
{                                   {
    Geeta obj;                          public void dowork()
    public void dowork()                {
    {
                                        }
    }                               }
}
```

Ramu is dependent on geeta to complete the work

Ramu needs obj of geeta.

Because of this it is tightly coupled.

* Spring will create the obj of Geeta & will inject in Ramu at run time instead of using new keyword of obj creation this will make the code loosly coupled.

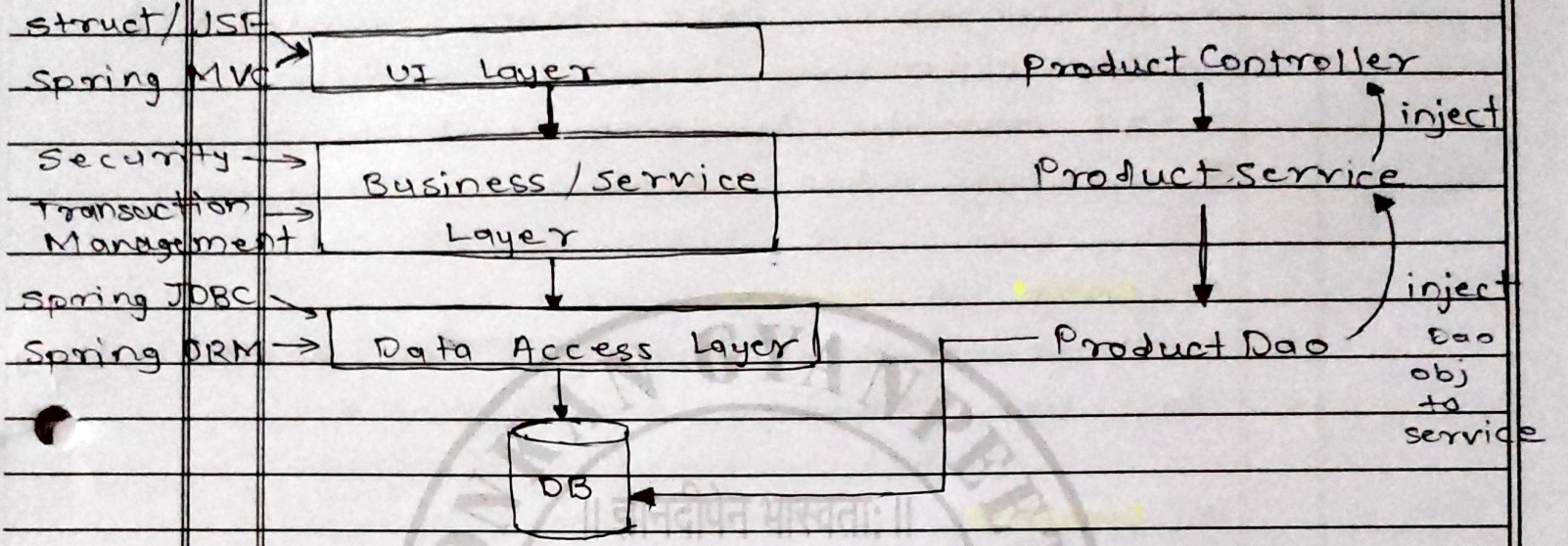
* Dependency Injection

It is a design pattern that removes the dependency from the programming code so that it can be easy to manage and test the application.

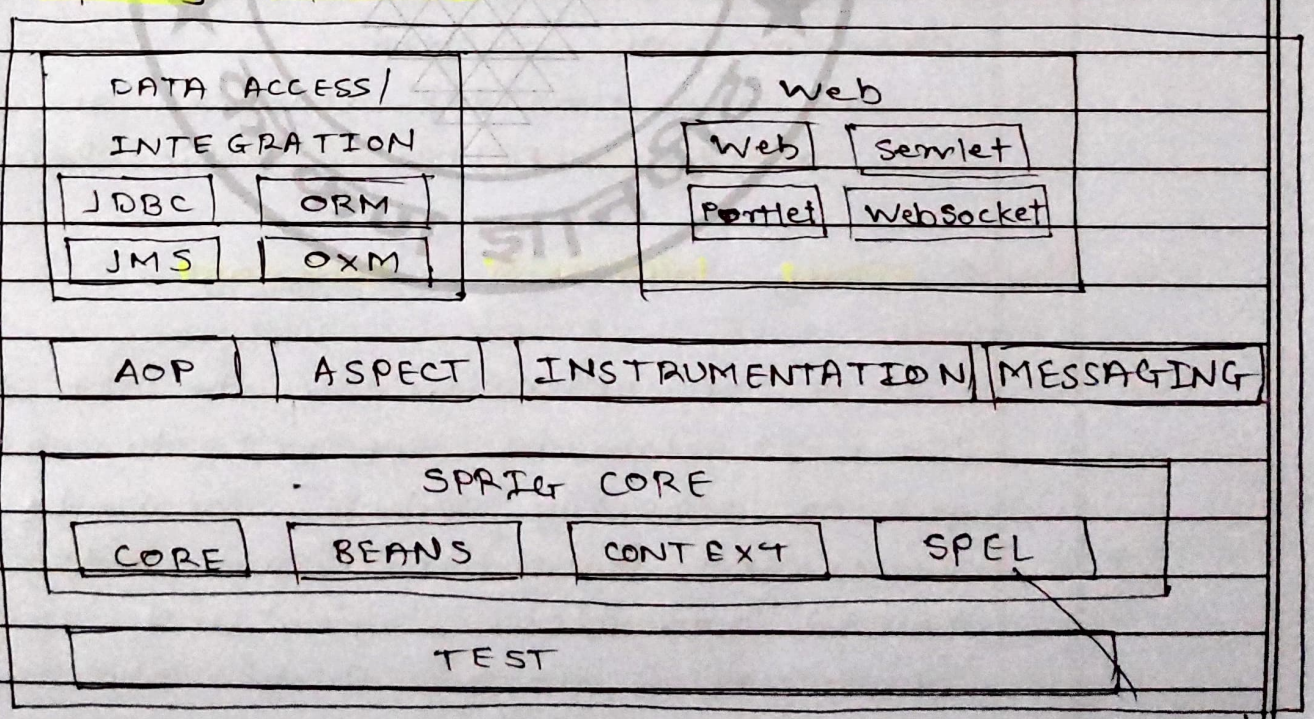
Dependency Injection makes our program loosely coupled.

KGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCE

* Spring and JEE



* Spring Modules



SPRING FRAMEWORK

* **Spring Core Container / Module**

* It containes core, beans, context & spring expression language modules.
* It provides fundamental basic stuff such as IoC, dependency Injection.

* **Core & Beans**

These modules provide IoC & Dependency Injection features.

* **Context**

Context inherite features from bean & also have internationalization, Event propogation, Resource Loading & Transparent creation of context.

And also provides features of JavaEE such as EJB, JMS, Basic Remoting.

* **Spring Expression Language**

It is an extension to the EL defined in JSP. It provides support to setting & getting property values, method invocation, accessing collections and indexers, named variables, logical and arithmetic operators, retrival of objects by name etc.

* AOP, Aspects & Instrumentation

These modules support aspect oriented programming implementation where you can use Advices, Pointcuts, Interseptors etc to decouple the code.

The Instrumentation module provides support to class instrumentetion and classLoader implementations.

* Messaging

It provides foundation to messaging appl^n.

* Data Access / Integration

It conteins JDBC, ORM, OXM, JMS and transaction modules. These modules' provides support to interact with the database.

* Web

It conteins Web, Servlet, Struts & portlet. These modules provide support to create web application.

* Test

It provides support of testing with Junit & TestNG.

* Spring IOC Container

The IOC container is responsible to instantiate, configure & assemble the object. The IOC container gets info from xml file and works accordingly

1) Creating the objects
2) Holding these objects in memory
3) & injecting them in another obj as required

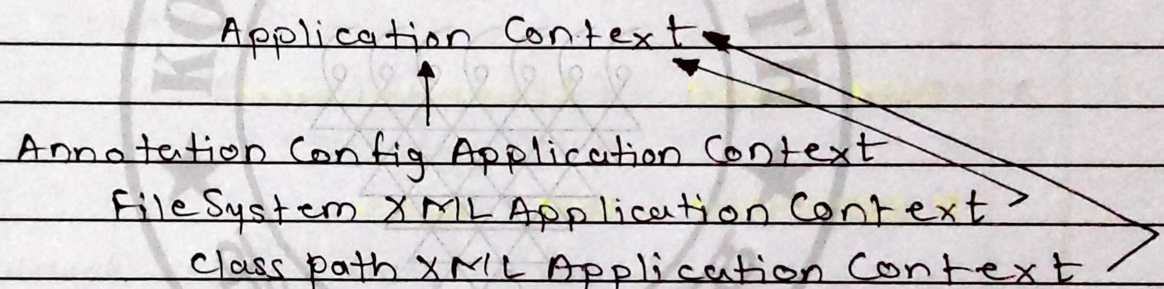
complete life cycle of the objects from creation to destruction is maintained by IOC container. we provid beans (POGO) & xml configuration.

IOC Container Type :-

1) Bean Factory
2) Application Context

BeanFactory and ApplicationContext interfaces acts as the IOC container. Appl^n Context interface is built on top of the BeanFactory interface. It adds extra functionality such as simple integration with AOP, message resource handling, event propagation, application layer specific context for web appl^n.

So it is better to use Application Context.



* Spring framework Provide Two types of Dependency Injection
1) Setter Injection (Setter Method)
2) Constructor Injection

1) Setter Injection / Property Injection

It will inject the dependency using setter methods.

The <property> subelement of <bean> is used for setter injection.

2) Constructor Injection

It will inject the dependency using constructor.

The <constructor-arg> subelement of <bean> is used for constructor injection.

* Configuration File

It is a XML file where we declare beans and its dependency.

* Data Types (Dependencies)

1) Primitive Data Types

byte, short, char, int, float, double, long, boolean

2) Collection Type

List, Set, Map & Properties

3) Reference Type

other class object (user defined data type)

* Process - Setter Injection

1) Create maven Project
2) Adding dependencies → Spring core, Context 5.2.3
3) Creating Java class (Beans)
4) Creating Configuration file → config.xml
5) Setter Injection
6) Main Class : which can pull the object

config.xml (option 1)

```
<beans>
    <bean class = "com.ycp.Student" name = "s1">
        <property name = "studentID">
            <value> 01 </value>
        </property>
        <property name = "studentName">
            <value> Akhilesh </value>
        </property>
        <property name = "studentCity">
            <value> Mumbai </value>
        </property>
    </bean>
</beans>
```

main class

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

Student s1 = (Student) context.getBean("s1");
System.out.println(s1);
```

---

config.xml (option 2)

```
<beans>
    <bean class="com.ycp.Student" nam = "s2">
        <property name="studentID"
                  value="02" />
        <property name="studentName"
                  value="Amey" />
        <property name="studentCity"
                  value="Pune" />
    </bean>
</beans>
```

---

config.xml (Option 3) with p schema

```
<beans>
    <bean class="com.ycp.Student name="s3"
          p:studentID="03" p:studentName="Raj"
          p:studentCity="Nashik" />
</beans>

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

* **Collection Type**

* **List**

```
<bean   class = "   "   name = "   ">
    < property   name = " ListName" >
        < list >
            < value > 10 < /value >
            < value > 15 < / value >
            < value > 100 < / value >
            < value > 5 < / value >
            < null / >
        < / list >
    < / property
< / bean >
```

* **Set**

```
< bean   class = "   "   name = "   ">
    < property   name = " SetName" >
        < set >
            < value > 10 < / value >
            < value > 15 < /value >
            < value > 100 < / value >
            < value > 5 < / value >
        < / set >
    </property >
< / bean >
```

KGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCE

* **Map**

```
<bean   class = "  "   name "   " >
  <property  name = "MapName" >
     <map>
        <entry  key = "java"  value = "2M" />
        <entry  key = "python"  value = "1M" />
        <entry key = "C++"  value = "2M" />
     </map>
  </property>
</bean>
```

* **Properties**

```
<bean  class = "   name = "  " >
  <property  name = "PropertiesName" >
     <props>
        <prop  key = "name" > Akhilesh </prop>
        <prop  key = "age" > 22 </prop>
        <prop  key = "City" > Mumbai </prop>
     </props>
  </property>
</bean>
```

---

* **Referance Type**

```
< bean >
   <property  name = "A" >
      <ref  bean = "B" />
   </property>
</bean>
```

```
   <property  name = "A"  ref = "B" />   option 2
```

```
<bean  class = "  "  name = "  "  p:ob-ref = "B" />   option 3
```

* **Constructor Injection**

Process

1) Create Maven project
2) Adding dependencies → spring core, context
3) Creating Java class (Bean)
4) Creating configuration file → config.xml
5) Constructor Injection
6) Main class : which can pull the object

config.xml

```
<beans>
    <bean class = "com.ycp.Student" name = "s1">
        <constructor-arg value = "Akhilesh"/>
        <constructor-arg value = "22"/>
    </bean>
</beans>
```

```
<constructor-arg value = " " type = "int"/>
```

* To reslove ambuigity. (Type)

* Life Cycle Methods

spring provide two important methods to every bean

1> public void init ()
2> public void destroy ()

1> public void init () -

It initialises the code, Loading config, connecting db, webservice etc.

2> public void destroy () -

Used for clean up code.

* We can change the name of these method but signature must be same.

PTO

KGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCE

* Life Cycle

spring bean →
- ◯ → create obj of spring bean
- (20) → instantiate the obj / set the properties

Configuration XML File →
- init() → Database Con^n etc
- Read & Use the bean
- destroy() → clean up code
- ◯ → destroy the obj

* Configure Technique

1> XML 2> Spring Interface 3> Annotation

1> XML Process -

① Write init() & destroy() in bean (class)
② Configure it in xml
③ Use AbstractApplicationContext & registerShutdownHook() to call destroy().

1> xml

```
<beans>
  <bean class = " " name = " " init-method = " " destroy-method = " ">
    <property name = " " value = " " />
  </bean>
</beans>
```

2) Spring Interface

Implementing bean Lifecycle using interfaces

1) Initializing Bean
2) Disposable Bean

Process

1) Bean (class) implements InitializingBean, DisposableBean
2) InitializingBean provides afterPropertiesSet() -
   we will write init code in that method
3) DisposableBean provides destroy()
   we will write destroy code in that method

* No need to write init-method & destroy-method in bean tag in xml.

3) Annotations

Implementing bean Lifecycle using Annotations

1) @PostConstruct
2) @PreDestroy

These two annotations are part of JavaEE and after java these are removed so we have to add the javax.annotation dependency in pom.xml file to use annotation.

& to enable the annotation we will use <context:annotation-config /> in cofig.xml

① Write init() & destroy() method in bean with any name & use @PostConstruct annotation before init() & @PreDestroy annotation before destroy.

② Add javax.annotation dependancy in pom.xml if java vers is greater than 11 or equal to 11.

③ Add <context:annotation-config /> tag in config.xml to enable all the annotation.

Autowiring in Spring

* Feature of spring framework in which spring container inject the dependencies automatically.

* Autowiring can't be used to inject primitive and string values. It works with reference only.

Autowiring

XML

Autowiring Modes

1) no
2) by Name
3) by Type
4) constructor
5) autodetect (deprecated)

Annotations

@Autowired

* Autowiring Advantages

1) Automatic
2) less code

* Autowiring disadvantages

1) No control of programmer
2) It can't be used for primitive & String values.

1) Autowire by XML using byName

config.xml

```
<bean class=" " name=" " autowire="byName" />
```

Process -

① Create bean (class) with setter, getter & default & parameterized constructors
② create config.xml
③ Add bean tag for dependent bean (Address)
④ Add bean tag for Emp bean with autowire attribute as "byName"

Note :- Data member name & value of name in xml should be same for autowire byName

or

ref name should be same as bean name in xml.

2) Autowire by XML using byType

config.xml

<bean class=" " name=" " autowire="byType"/>

Process -

① Create bean (class) with getter, setter & default & parameterized constructors.
② create config.xml
③ Add bean tag for Address bean
④ Add bean tag for Emp bean with autowire attribute as "byType"

Note :- Gives error if two or more beans of same type is present in config.xml

3) Autowire by XML using constructor

<bean class=" " name=" " autowire="constructor"/>

Process -

① Create bean (class) with getter, setter, default & parameterized constructor.
② Create config.xml
③ Add bean tag for Address
④ Add bean tag for Emp bean with autowire attribute as "constructor"

Note :- It will check name of ref & bean to autowire

KGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCEKGCE

4) Autowire by Annotation (@ Autowired)

1) Autowired using property (Data member)
(by Type)

```
public class Emp {
    @Autowired
    private Address address;

}
```

* It uses byType

2) Autowired using setter method

```
public class Emp {
    private Address address;

    @Autowired
    public void setAddress(Address address)
    { this.address = address; }
}
```

3) Autowired using constructor

```
public class Emp {
    private Address address;
    @Autowired
    public Emp(Address address) {
        this.address = address
    }
}
```

* Autowired + Qualifier

Qualifier - It is used to specify the name of bean to be inject in obj.

```
public class Emp {
    @Autowired
    @Qualifier ("address") ← (Name of bean)
    private Address address ;
}
```

* Qualifier is used when more than two beans with same type is present in the config. to elemenate the exception.

* Standalone Collections

* It is used when we want to use same collections in multiple beans.
* we will have to use util schema to declare standalone collections outside bean.

Process -
① Create bean (Class)
② Create config.xml with util schema
③ Use util:list, util:set, util:map etc. with id & class.
④ Pass the id of collection in ref of other bean to use those collections.
⑤ @Value ("#{id}") use this annotation in bean

tag - list :

```
<util:list   list-class = "java.util.LinkedList"
                    id = "myList" >
    <value> Amey </value>
    <value> Prathamesh </value>
    <value> Pratik </value>
    <value> Darshan </value>
</util:list>
```

tag - Map :

```
<util:map   map-class = "java.util.HashMap"
                id = "myMap" >
    <entry  key = "Java"   value = "10000" />
    <entry  key = "C++"    value = "8000" />
    <entry  key = "Python" value = "5000" />
</util:map>
```

tag - Properties :

```
<util:properties  id = "dbconfig" >
    <prop  key = "driver" > com.mysql.jdbc.Driver </prop>
    <prop  key = "username"> Akhilesh </prop>
    <prop  key = "password"> mysql123 </prop>
    <prop  key = "url" > mysql:jdbc://localhost:3306/
                        database </prop>
</util:properties>
```

* Stereotype Annotations

Stereotype Annotations are used to create spring beans automatically in the application context.

@ Component Annotation is the main Annotation.

① Add <context:component-scan base-package = " " /> tag in config.xml

② Use @ component before class

```
@ Component
public class Student {
    @ Value ("Akhilesh Dalvi")
    private String studentName;
    @ Value ("Mumbai")
    private String studentCity;

getter , setter , constructors , tostring
}
```

@ value is used for dependency injection or we can say to inject values in obj.

Bean Scope

1) Singleton (Default scope)
2) prototype
3) request - (Web application)
4) session - (Web application)
5) globalsession - (Portlet application)

1) Singleton scope

If scope is set to singleton the spring Ioc container creates exactly one instance of the object defined by that bean defination. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object

```
<bean class = " " name = " " scope = " " />
```

```
@Component
@Scope (" ")
public class Student{

}
```

2) Prototype Scope

If scope is prototype then the spring IoC container creates a new bean instance of the the object every time a request for that specific bean is made.

Note :- Use the prototype for all state-full beans & singleton for stateless beans.

```
<bean class="" name="" scope="prototype"/>
```

```
@Component
@Scope("prototype")
public class Student {

}
```