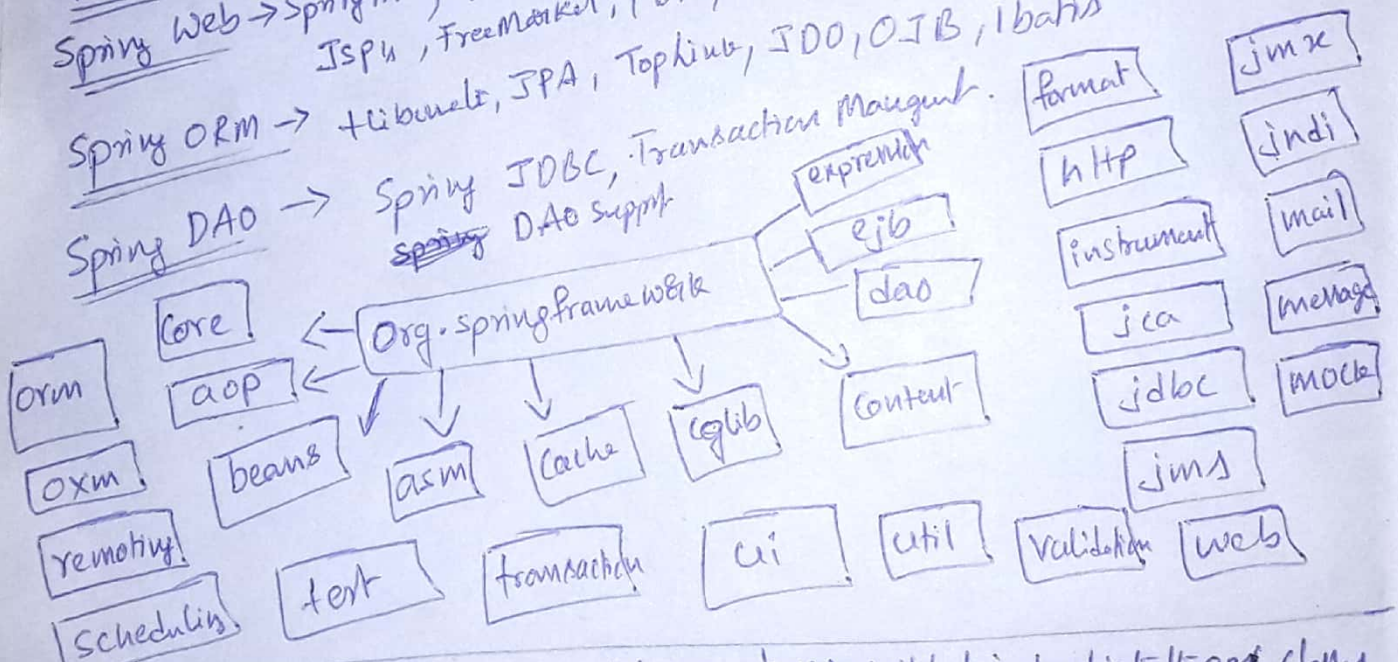
Spring FRWRK :- Light-weight → it doesn't require at many code changes to incorporate them with our F/W
→ - EJB - is heavyweight F/W.

Non-Invasive → means it doesn't force a programmer to extend or implement their class from any predefined class or interfaces given by Spring API, in Struts we used to extends Action class right that's why struts is said to be invasive.

Advantages :- ① predefined templates, ex:- Spring F/W provided templates for JDBC, Hibernate, JPA etc.
② Loose Coupling
③ Easy to test
④ Lightweight
⑤ Fast Development
⑥ powerful abstraction :- it provides powerful abstractions to J2EE Specification - JMS, JDBC, JPA and JTA.
⑦ Declarative Support :- Caching, validation, transactions, formatting.

No need to write the code for exception handling, creating Connection, Creating Statement, Commit transaction, closing Connection, Spring will care all the things you need to execute the query only

Spring Modules :- Spring core → Supporting utilities, Bean Container [Bean, Core, Context, SpEL]
(Core container)
Spring AOP → Source-level metadata, AOP infrastructure [AOP, Aspects, instrumentation, Messaging]
Spring J2EE → Jmx, JMS, JCA, Remoting, EJB's, Email
Spring Web → Spring MVC F/W Integration with Struts, webwork, Tapestry, JSF, Velocity
JSPs, FreeMarker, POI, JasperReports, Excel, Spring Portlet MVC.
Spring ORM → Hibernate, JPA, Toplink, JDO, OJB, ibatis
Spring DAO → Spring JDBC, Transaction Manager. DAO support



IOC - Container → It's a programming technique → responsible to instantiate the application classes
→ to configure the objects → to assemble the dependencies b/w the objects.
2 types → BeanFactory (I) [console based application] org.springframework.beans.factory
→ ApplicationContext [web based application] org.springframework.context

BeanFactory (I) → XmlBeanFactory (org.springframework.beans.factory.xml)

```
Resource rs = new ClassPathResource("applicationContext.xml");
BeanFactory bf = new XmlBeanFactory(rs);
Student st = (Student) bf.getBean("stuff Bean");
```

Resource → public interface Resource extends InputStreamSource (org.springframework.core.io)
which either a given ClassLoader or a given class for loading resources.

ApplicationContext: [org.springframework.context] [Web based Applications]

implementation classes: ClassPathXmlApplicationContext, FileSystemXmlApplicationContext

AnnotationConfigApplicationContext → annotated standalone applications

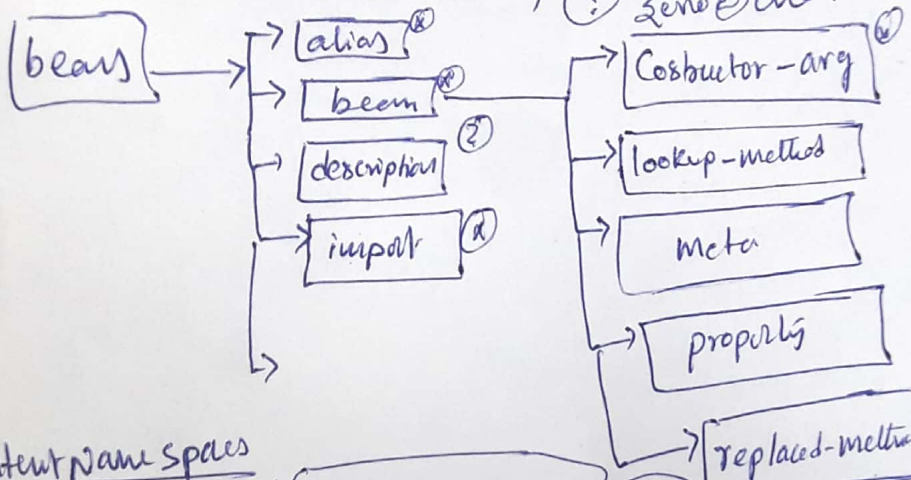
AnnotationConfigWebApplicationContext → web applications.
XmlWebApplicationContext

```
ApplicationContext ap = new ClassPathXml
Student st = (Student) ap.getBean("student");
```

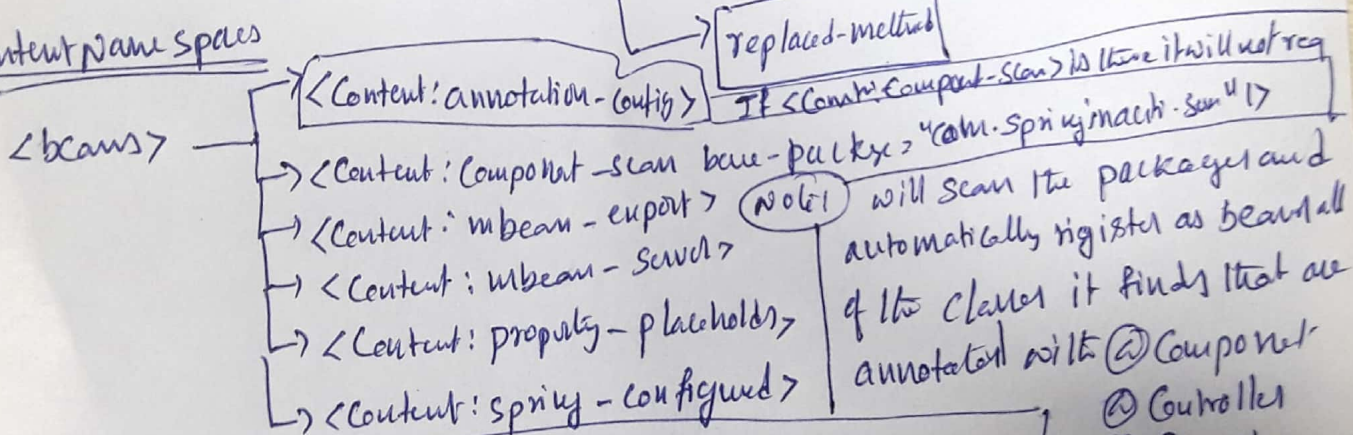
Spring Bean Configuration → Spring configuration file (xml) to configure beans.

XML Bean Configuration:- → If Spring MVC, need some code in web.xml file to load automatically.

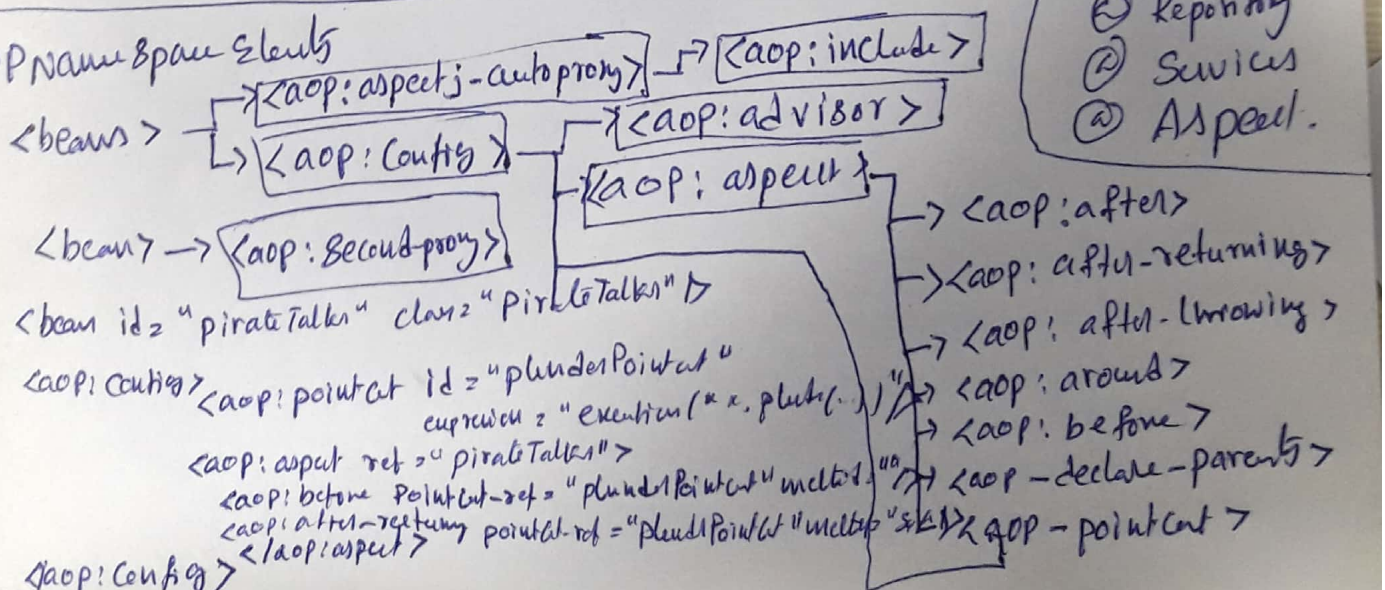
(R) Required xml elem, (*) Zero or more, (?) Zero or one



Context Name spaces

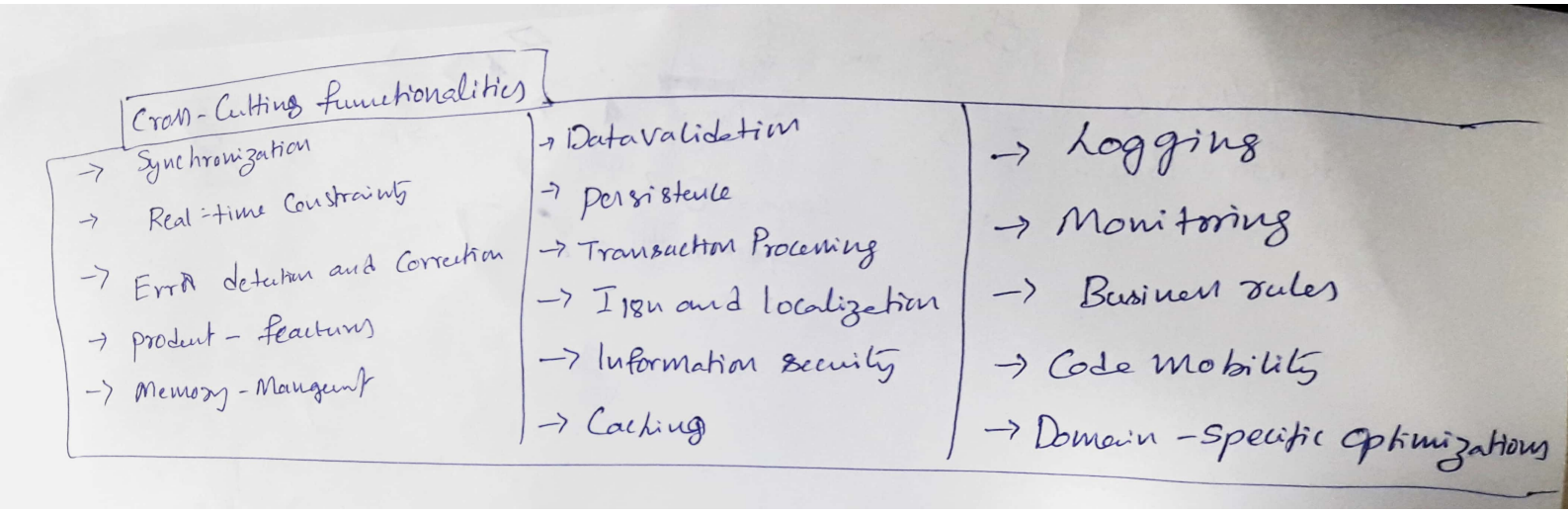


AOP Name space elements



Ex `<bean id="pirateTalks" class="PirateTalks" />`

```
<aop:config>
  <aop:pointcut id="plunderPointcut"
      expression="execution(* *.plunder(..))" />
  <aop:aspect ref="pirateTalks">
    <aop:before pointcut-ref="plunderPointcut" method="..." />
    <aop:after-returning pointcut-ref="plunderPointcut" method="..." />
  </aop:aspect>
</aop:config>
```



Spring Bean life Cycle →

1) Instantiate → Within Ioc Container a Spring bean is created using class Constructor.

2) populate properties → Using its dependency injection, Spring populates all of its properties as specified in the bean definition.

3) SetBeanName() → BeanNameAware, If bean implements the BeanNameAware Interface, the factory calls setBeanName() passing the bean's ID.

Eg
```
public class AwareBean implements BeanNameAware, BeanFactoryAware
{
    public void setBeanName(String beanName)
    {
        S.O.P(beanName);
    }
```

Creates a bean.xml
```
<beans>
  <bean id="awareBean"
        class="AwareBean" />
</beans>
```

4) SetBeanFactory() → BeanFactoryAware → Spring Container Passes its BeanFactory to setBeanFactory()

```
public void setBeanFactory(BeanFactory beanFactory) throws BeansException
{
    S.O.P();
}
```

5) Pre Initialization – this stage is also called the Bean Post Process. If there bean are any BeanPostProcessors associated with that their postProcessBeforeInitialization() methods will be called

Eg
```
public class MyBeanPostProcessor implements BeanPostProcessor
{
    public Object postProcessAfterInitialization(Object bean, String beanName)
            throws BeansException
    {
        return bean;
    }
    public Object postProcessBeforeInitialization(Object bean, String beanName)
            throws BeansException
    {
        return bean;
    }
```

Post Constructor / After init-method / Before Constructor / Before-Init-method

Spring-Config.xml
```
<beans>
  <bean id="book" class="Can.Book" init-method="myPostConstruct"
        destroy-method="myPreDestroy">
    <property name="bookName" value="Math" />
  </bean>
  <bean class="MyBeanPostProcessor" />
</beans>
```

6) Initialize bean – After @PostConstruct, the InitializingBean.afterPropertiesSet()

Eg
```
class BookBean implements InitializingBean
    void afterPropertiesSet()
        S.O.P(InitializingBean.afterPropertiesSet();");
```

If Bean has Init-method declaration, its specified initialization method is called

7) → the Spring Container calls their postProcessAfterInitialization() method

BeanPostProcessor.postProcessAfterInitialization() is called.

8) Ready-to-Use – Now to be used by the application.

9) Destroy → After the DisposableBean.destroy() method is called on the bean.

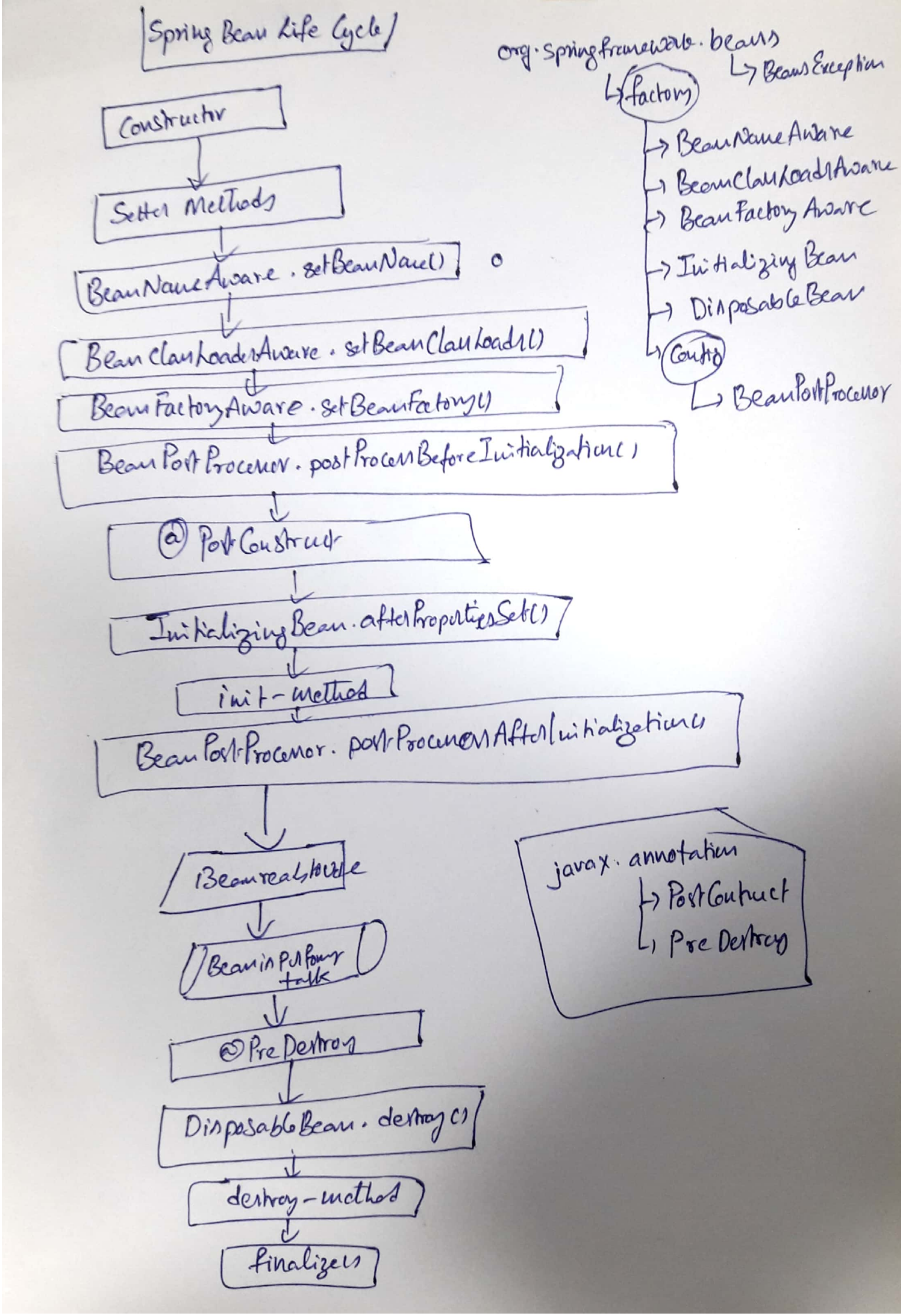
Eg
```
class Book implements DisposableBean
    public void destroy()
        S.O.P(Destroy);
```

→ If a custom destroy() method is defined, destroy-method attribute of bean in XML file method will be called just before a bean is removed from the Container.

→

10) finalize() of a object is called → It will be called Before garbage Collection.

Eg
```
class Book
    protected void finalize()
    {
    }
```



Spring AOP

Cross-Cutting Concerns :- There are generic functionality needed in many places in application.

Eg Logging and Tracking
Transaction Management
Security
Caching
Error Handling
Performance Monitoring
Custom Business Rules

Core AOP Concepts

Join point :- point in the execution of a program such as a method call @ Exception Thrown.

PointCut :- the expression that selects one @ more join points.

Advise :- code to be executed at each selected join point

Aspect :- (in a class) in which Cutting pointcuts and advise

Weaving :- Technique which aspect are combined with main code

@Aspect annotation in Spring

How to enable @AspectSupport

include in spring config file `<aop:aspectj-autoproxy />`

@Aspect → org.aspectj.lang.annotation.Aspect

Required jars

aspectjrt.jar
aspectjweaver.jar
aspectj.jar

Declaring an aspect

① @Aspect
public class LoggingAspect {
}

Key need to configure in Spring Bean class xml file as other bean.

```
<bean class="com.rj.LoggingAspect"
      id="logAspect" />
```

Declaring a pointcut

→ A pointcut expression that determines exactly which method executions we are interested in.
→ A pointcut signature comprising a name and any number of parameters. The actual body of the method is irrelevant and in fact should be empty.

① **How to apply pointcut execution under matched under Package of all classes**

@Pointcut("execution(* com.rj.aspect.test.service.*.*(..))");
private void businessServices() {} // signature.

(2) expression for determining exactly which method executions are required.

@Aspect
public class LoggingAspect {
@Pointcut("execution(* com.rj.service.Employe.addEmployee(..))");
public void addEmployee() {}

(3) Combining pointcut expressions can be combined using && || !

@Aspect
-> it is possible to refer to pointcut expression by name.
-> any publicOperations
-> only with in the specific module
-> any matches if a method execution represents any public methods in the trading module.

(1) for any public @ private @ protected method expression call
@Pointcut("execution(public * *(..))")
private void anyPublicOperation() {}
name which matches and

(2) a method execution is in the particular module.
@Pointcut("within(com.rj.Service..*)")
private void inTrading() {}

(3) tradingOperation (which matches if a method execution represents any public methods in trading module).

@pointcut("anyPublicOperation() && inTrading()")
private void tradingOperation() {}

Spring AOP users are likely to use the **execution** pointcut designator

Its format of an execution expression is:

```
execution(modifiers-pattern? ret-type-pattern declaring-type-pattern?name-
                                                    pattern(param-pattern)
                                                    throws-pattern?)
```

→ @Pointcut("execution(public **(..)) → execution of any public method only.

→ @Pointcut("execution(* set*(..)) → any method with a name beginning with "set"

→ @Pointcut("execution(* com.cmbd.Service*.*(..))") → package

↳ this will execution of any method defined by Service ~~package~~

→ @Pointcut("execution(* com.cmbd.service.IService.*(..))")

↳ execution of any method defined by the IService interface.

→ @Pointcut("execution(* com.xyz.service..*.*(..))

↳ the execution of any method define in the service @ Sub-package.

→ Any join point (method execution only in Spring AOP) with in the given pkg.

@within(com.xyz.service.*)

→ any join point (method execution only in Spring AOP) within the service @ Sub-package.

@within(com.xyz.service..*)

→ any join point (method execution only in Spring AOP) where the proxy implements the AccountService interface → this(com.xyz.service.AccountService)

→

@target

@args

@annotation

@bean

Advise :- Code to be executed at each selected joinpoint.

@Before → org.aspectj.lang.annotation.Before

→ In Spring AOP before advice is that execute before a joinpoint. i.e. a method which annotated with AspectJ @Before annotation run exactly before the all methods matching with pointcut expression.

Pointcut expressions

(1) expression for before advise is valid for all public method (@Before joinpoint(pointcut expression)) with any arguments of any type and any type of all classes in the com.cudd.service package

@Before ("execution(* com.cudd.service.*.*(..))")

(2) expression for before advise is valid for all public method and whose name are getTransaction() with taking two (2) more argument type and type of all classes in the com.cudd.service package

@Before ("expression(* com.cudd.service.*.getTransaction(*,*,*))")

Named pointcut

(1) Declare Named pointcut

@Pointcut ("execution(* com.cudd.service.*.*(..))")
public void logForAllMethods(){}

(2) Apply @Before advice on named pointcut

@Before ("logForAllMethod()")
public void beforeAdviceForAllMethod(JoinPoint jp) throws Throwable{
S.O.P (jp.getSignature.getName");
}

@After - Advise annotation :- Advice to be executed regardless of the means by which a joinpoint exits (normal (or) exceptional return)

@After returning advice :- Advise to be executed after a joinpoint completes normally : for if a method returns without throwing an exception.

@AfterReturning

@AfterThrowing advice :- to be executed if a method exits by throwing an exception.

@Around advice :- to be perform custom behaviour before and after the method invocation. it is responsible for choosing to proceed to the joinpoint (or) shortcut the joinpoint - (or) executing by returning its own return value (or) throwing an exception.

@AfterReturning :- @AfterReturning (pointcut = "com.endd.Bank.getBalance()", returning = "retval")

public void doAccountCheck(object retval){

}

@AfterThrowing :- @AfterThrowing ("com.endd.dao.dataAccessOperation()", throwing = "ex")

public void doRecoveryActions(DataAccessException ex)

@After :- (finally) advice

@After ("com.endd.dao.dataAccessOperation()")

public void doRollback(){

@Around

@Around ("com.endd.app.SystemArchitecture.businessService()")

public object doBasicProfiling(ProceedingJoinPoint pjp) throws Throwable{

Spring Bean Configuration file

1. <beans Schema definition>
2. <util details>
3. <jee tag details (Java Enterprise Edition) - related Configuration such as - JNDI, EJB
4. <language support declaration> → JRuby, Groovy, Beanshell
5. <jms schema declarations>
6. <tx transaction Schema declarations.
7. <aop schema declarations>
8. <context Schema definitions>
   - → <property-placeholder/>
   - → <annotation-config/>
   - → <component-scan/>
   - → <load-time-weaver/>
   - → <spring-configured/>
   - → <mbean-export/>
9. <tool schema declaration>
10. </beans> Schema End declaration.

① XML Schema-based Configuration :- <beans schema definition>

XSD → XML Schema Definition

→ XML schema is XML-based (and more powerful) alternative to DTD (Document Type Definition)

<?xml version="1.0" encoding="UTF-8"?>

<beans xmlns="http://www.springfw.org/"

xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"

xsi:schemaLocation="http://www.spfw.org/schema/beans (1) http://www.spfw.org/schema/beans/spring-beans-3.0.xsd (2)">

<!---->

Note: xmlns="uri" → indicates that the elements and datatypes used in the schema come from the "uri" namespace. ie xmlns:beans → should suffix beans <

</beans>

xmlns:xsi :- xmlns:xsi="uri" → this declaration tells the schema-validator that all the elements used in this XML document are declared in the "uri" namespace.

xsi:schemaLocation :- you can use the schemaLocation attribute. This attribute has two values, separated by a space. ① the first value is the namespace to use.

② the second value is the location of the XML schema to use for that namespace. (xsd file)

② beans declaration :

→ Dependency Injection in Spring

→ It is a design pattern allows us to remove the hard-coded dependencies and make our application loosely coupled, extendable and maintainable.

→ We can implement DI in java to move the dependency resolution from compile-time to runtime.

In Spring → two ways are there ① By Constructor

② By Setter method.

Constructor Injection — XML Config :- <constructor-arg>

```
public class Emp {
    private int id;
    private String name;
    private A a;
    private B b;
    public Emp() {...} // default Constructor

    public Emp(int id) { this.id = id; }
    public Emp(String name) { this.name = name; }
    public Emp(int id, String name, A a, B b) {
        this.id = id;   this.a = a;
        this.name = name;  this.b = b;
    }
    void show() { S.o.p(id + "..." + name); }
```

applicationContext.xml

```
<beans>
  <bean id="emp" class="com.core.Emp">
    Constructor param name
      <constructor-arg name="id" value="10" />
      <constructor-arg name="name" value="ti" />
```

Using type attribute → datatype

```
      <constructor-arg type="int" val="10" />
      <constructor-arg type="String" val="ti" />
```

Using ref attribute → Object dependency

```
      <constructor-arg ref="aClass" />
      <constructor-arg ref="bClass" />
  <bean id="aClass" class="com.core.A" />
  <bean id="bClass" class="com.core.B" />
</beans>
```

Using index attribute

```
<constructor-arg index="0" val="10" />
<constructor-arg index="1" value="ti" />
```

Constructor Injection in Collections

① Straight values (primitives, Strings, and soon) → <property />

Eg

```
<bean id="emp" class="com.core.Emp">
    <property id="id" value="10" />
    <propert id="name" value="ti" />
</bean>
```

Reference to other beans :- to specify other dependent Object through

ref	bean, local & parent	they we can declare in <constructor-arg /> <property /> &

<ref bean="someOtherBeanObjectId" /> → Specifying the other dependent bean id
refers to any bean in the same Container (config xml file)
parent Container (parent xml file)
regardless of weather it in in same xml file

<ref local="someOtherBeanId ref" /> → from Spring 4.0 schema will not support.
→ purpose of this is refers in within the Container (same xml config file)

Setter Injection — We can inject the dependency by Setter method also.

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

1. primitive and String-based values
2. Dependent Object (Contained Object)
3. Collection values.

Employee.java

```
private int id;
private String name;
private List<Phone> ph;
```

Phone.java

```
private phoneNo;
private empNo;
```

Set<Phone> ph;
Map<Employee, List<phone>> ph;

Set / get

```
<bean id="emp" class="com.ends.Employee">
  <property name="id" value="101" />
  <property name="name" value="RJ" />
  <property name="phSet">
    <set>
      <ref bean="ph1" />
      <ref bean="ph2" />
    <set>
  </property>
  <property name="phList">
    <list>
      <ref bean="ph1" />
      <ref bean="ph2" />
    </list>
  </property>
  <property name="mapPhone">
    <map>
      <entry key-ref="Key-Id" value-ref="objectMapId" />
      <entry key-ref="Key-Id2" value-ref="objectMapId2" />
    </map>
  </property>
</bean>
```

Spring Bean Scope :- xml Configuration → Singleton scope is Spring IOC container default scope.

Singleton , Prototype , request , session , globalsession, Application

Singleton :- This is the default scope, there will be only one instance in the Entire Spring Container, or in the entire java application.

```
Eg  <bean id="dataSource"  class="com.cndd.DBdataSou"
                           scope="Singleton" >
    </bean>
```

Prototype Multi-instance type, Container will create a new bean instance for every request to a bean by client.

```
Eg  <bean id="CustomerSevice"  class="Com.cndd.Sevice"
                               Scope="Prototype" />
    </bean>
```

Request :- Return a single bean instance per HTTP request. When a web action request an instance of this bean, it will be created by Spring Container, and then saved in HttpServletRequest object. When the request Complete, the bean will be out of scope and wait for garbage Collection.

```
Eg  <bean id="customerSevice"  class="com.cndd.CutomrSevi"
                               Scope="request" >
    </bean>
```

Session → Return a Single bean instance per HTTP Session. This bean Scope type is valid only in web appl. also. Defined with this scope web action request this bean, creates an instance of it then save in the "HttpSession" object. When the Session timeout - @Invalidate, the bean in invalidate also.

```
<bean id="customSe" class="cndd.servi.Sevice"
                    Scope="session" >
</bean>
```

globalsession (same like session) :- Return a Single bean instance per global HTTP Session. When you use portlet Container to create portlet application, there are a number of portlets in it. Each portlet will save variables in their own session by default. how to share a global variable ~~in the session~~ object to all the portlet in the portlet application.

⑥ application This is also for web application only. This kind of bean will exist in the web application context. One webapp will have only one instance.

How to specify Bean Scope

① Xml Configuration xml file.

```
<bean id="beanId" class="com.dev.HelloWorld"
      scope="singleton" />
```

singleton
prototype
session
application
request
globalSession.

② Via java Annotation

Condition① Want to use annotation to define, you should first add below xml bean definition by Scanning annotations.

```
<Context:Component-scan base-package="com.evidd.web.*" />
```

③ Then you need to use annotations such as @Component on the corresponding java class to indicate that it needs to be added as a bean definition to Corresponding Container.

@ Scope annotation.

Eg @Component / @Service

```
@Service("prototype")
public class HelloWorld{
}
```

Spring bean Autowiring Modes

→ This feature of spring FW enables you to inject the object-dependency implicitly. It internally uses setter & constructor injection.

Syntax:

```
<bean id="objectRefId" class="com.endd.web.Customer"
      autowire="byName" />
```

- byType
- constructor
- autodetect
- no

→ no :- Default, no autowiring, set it manually via "ref" attribute.

→ byName :- Autowiring by property name. If the name of a bean is same as the name of the bean property, autowire it.

→ byType :- Auto wiring by Property data type. If data type of a bean is compatible with the data type of other bean property, autowire it.

→ Constructor :- byType mode in constructor argument.

→ autodetect :- If a default constructor is found, use "autowired by Constructor" Otherwise, use "autowire byType".

Ex:

```
public class Customer {
    private Person person;
    public Customer(Person person) {
        this.person = person;
    }
    public setPerson(Person person) {
        this.Person = person;
    }
}
```

① Auto_wiring → "no"

```
<bean id="customer" class="com.Cust">
    <property name="person">
        <ref="person" /> </bean>
```

② Autowiring "byName"

```
setPerson(Person pe)
<bean id="customer" class="com.endd.Cust"
      autowire="byName" />
<bean id="person" class="person" />
```

③ Autowire - byType :-

```
private Person person;
```

④ Autowire byConstructor :-

```
public Customer(Person pe)
```

⑤ autodetect :- it autoinjects by using constructor injection.

Spring - bean - lazy initialization mode

In general all the beans are initialized at startup of application
To stop such unnecessary initialization we use lazy initialization
which create the bean instance when it is first required.

Spring Configuration file

```
<bean id="studentBean" class="com.cmdd.Student">
  <property name="name" value="satya" />
</bean>
<bean id="addressBean" class="com.cmdd.Address"
      lazy-init="true">
  <property name="city" value="New Delhi" />
</bean>
```

Annotation Configuration

@Lazy annotation in spring is used with @Configuration.

```
@Configuration
public class AppConfig {
    @Bean
    @Lazy("true")
    public A a() {
        return new A();
    }
    @Bean
    public B b() {
        return new B();
    }
}
```

```
AnnotationConfigApplicationContext ctx =
    new AnnotationConfigApplicationContext();
ctx.register(AppConfig.class);
ctx.refresh();
ctx.getBean(A.class);
```

Spring Annotations – Context Configuration Annotations (It will eliminate the Spring Bean Configuration mapping file)

Annotation	Used (when we can apply)	Description
@Configurable	Type (class level)	Used with <context:springconfig> in applicationContext.xml
@Bean	Type, the bean declaration with @Bean	Typically used to inject the property into Domain object.
@Scope	Type	Specifies the scope of a Bean, either Singleton, prototype request, session, global session, some custom scope
@Autowired	Constructor (CI injection) Field (setter injection) Method (SI)	Declaring a Constructor → class A{ @Autowire public A(){ field → @Autowire public B; Method (Setter Method) → @Autowired setB(B b); this.b = b;
@Required	Method (setter)	Ex class Disclosure{ private ODDIM; private ODDIM(); @Required void setODDIM(ODDIM oddm) this.ODDIM = oddm;

by default it is false if not mentioned this annotation it considers not required

@Order	Type, Method, Field	Defines ordering; as an alternative to implement the org.spfw.core.Ordered Interface

Ex Interface Rank

@Component @Order(1) class RankOne implements Rank

@Component @Order(2) ~~class RankTwo implements Rank~~

@Component public class Results{ @Autowired private List list;

In Bean file Component Scan ~~@Events required~~

<context:component-scan base-package = "com.ende.Ren" />

@Qualifier	Field, Parameter Type, Annotation Type	if we declare @Qualifier("beanName Id") it will pick right bean ~~name~~ picked by the spring container and assign it to corresponding property.

Ex: @Component

```
public class PaymentGateway {
    @Autowired
    @Qualifier("oracleOrderBean")
    private Order order;
```

Stereotyping Annotations :-

The annotations are used to stereotype classes with regard to the application tier (tier) that they belong to.
→ classes that are annotated with one of these annotations will automatically be registered in the Spring application Context if

```
<Context:Component-scan base-package="com.end.web.Controller
                                      Service
                                      Dao
                                      HelperService
```

in this Spring xml Configuration.

@Component	@Controller	@Repository, @Service

	Use	Description
@Component :-	Type (class level)	Generic stereotype annn. for any Spring managed Component.

```
eg: @Component
    public class HelperBean {
        @Autowire
        IServiceImpl iserv;
    }
```

```
Context:
<Component-scan base-package="com.end.Help
```

	Use	Description
@Controller :-	Type (class level)	Used for spring mvc fw.

```
eg: @Controller
    public class LoginController {
    }
```

```
<Context:Component-scan base-package="Com.end.
Cont"/>
```

	Use	Description
@Repository :-	Type (class level)	This we can use as DAO tier Layer class.

```
eg: @Repository
    public class DaoAcm {
        @Autowired
        SpringBean fun;
    }
```

```
<Context:Component-Scan base-package="Com.end.
Dao"/>
```

	Use	Description
@Service :-	Type (class level)	To make any class as Service tier

```
eg: @Service
    public class LoginService {
    }
```

```
<Context:Component-Scan base-package=
"com.end.Ser"/>
```

Spring – Transaction-Management

ACID → Atomicity – Consistency – Isolation – Durability

Atomicity → A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful (or) unsuccessful.

Consistency → A transaction either creates a new and valid state of data (or) if any failure occurs, returns all data to its state before the transaction was started.

Isolation → A transaction in process and not yet committed must remain isolated from any other transaction.

Durability →
- remain isolated from any other transaction.
- ↳ there may be many transactions processing with the same data set at the same time. Each transaction should be isolated from other to prevent data corruption.
- ↳ once a transaction has completed, the result of this transaction has to be made permanent and cannot be erased from the DB due to system failure.

Spring Transaction Management

A Transaction is a unit of work in which either all operations must execute @ none of them.

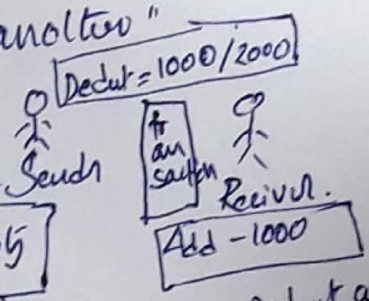
Sequence of action that will be performed to complete dB operation and its management is known as Transaction Management.

All these action in Combination will be treated as ONE action only So that DB doesn't fall in inconsistent mode ever.

Scenario "Transfering Amount from one account to another"

ⓐ Deduct the balance from the Sender's account

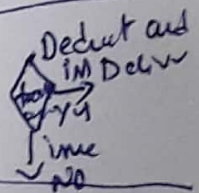
ⓑ Add the amount to the recivers account



ACID → Atomic Consistent Isolation Durability

ⓐ Atomic → Atomicity makes sure that either all operations with in transaction must be Success @ None of them.

Conditions :-



ⓑ Consistent :- This property makes sure that data should be in Consistent state once the transaction is Completed.

ⓒ Isolated :- this property allows multiple users to access the same set of data and each user's processing should be isolated from others.

ⓓ Durable :- Result of the transaction should be permanent once the transaction is Completed to avoid any loss of data.

2-types of J2EE-Transaction Manager ① Global Transaction.
② Local Transaction.

Global Transaction

→ Use to work with multiple transaction resources like → RDBMS @ Message Queue (Pros)

→ Managed by Applications Server (WebSphere, Weblogic) using JTA (Cons)

→ JNDI is required to use JTA

→ Code can not be reused as JTA is available at Server level (Cons)

Eg Example of Global Transaction
EJB Container Managed Transaction

② Local Transaction

→ Use to work with Specific resource (transaction associated with JDBC)
→ Can not work across multiple transaction resource opposite to Global transaction (Jaus)
→ Most of web application uses only single resources hence it is best option to use in normal app.

Spring Transaction Management Support → ① Programatic Transactions
② Declarative Transactions

① programatic Transactions :- → transaction management code like :-
Commit when everything is successful & rolling back if anything goes wrong is clubbed with the business logic.

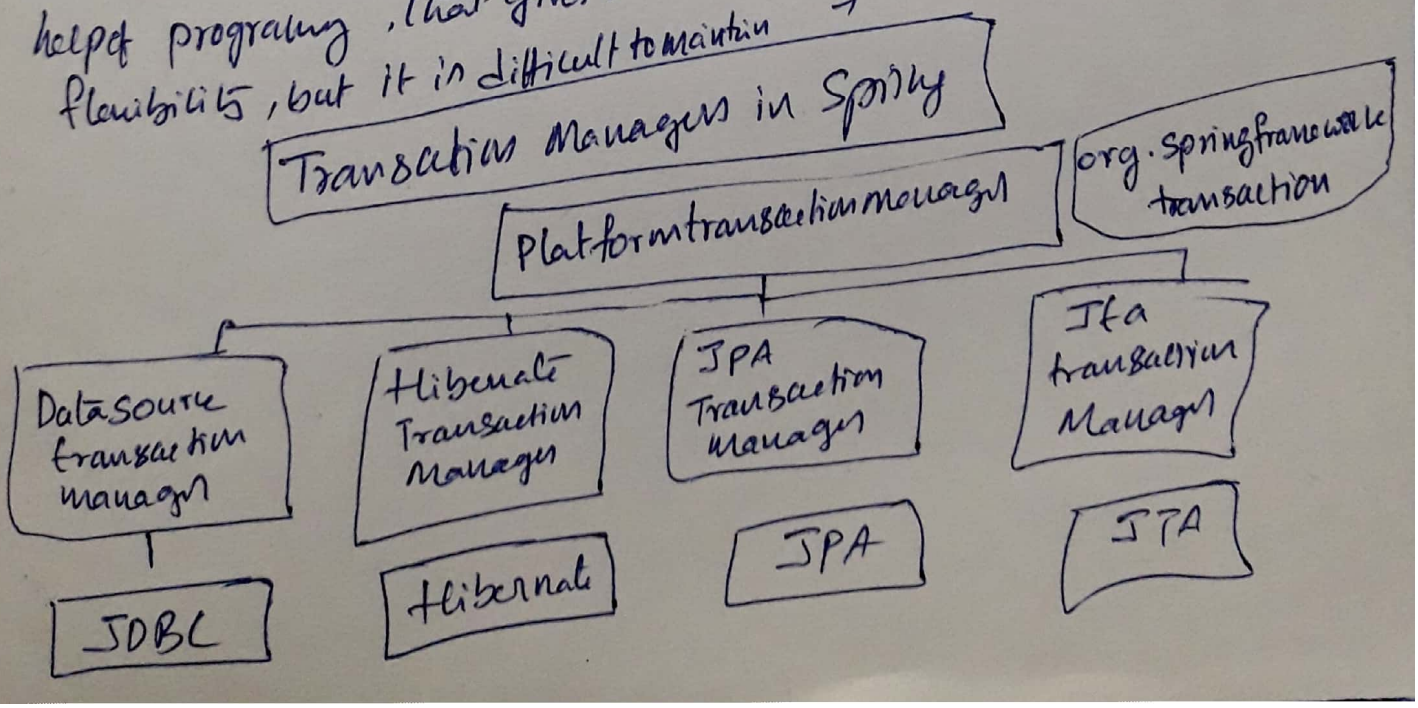
② Declarative Transactions :- DT separates transaction management code from business logic. Spring Supports DT using transaction advise (AOP).

programatic Transaction

→ A fine Control on transaction Boundries
→ to manage the transaction with the help of programing, that gives extreme flexibility, but it is difficult to maintain

Declarative Transaction

→ provides a great Configurability using Configuration files & annotations.



* HibernateTransaction manager :- When your application using Hibernate

Sample Config <bean id = "transactionManager"
class = "org.springframework.hibernate3.HibernateTransactionManager">
<property name = "sessionFactory" ref = "sessionFactory" />
</bean>

* DataSource Transaction manager :- Used for JDBC persistence mechanism.

Sample Config <bean id = "transactionManager"
class = "org.springframework.jdbc.datasource.DataSourceTransactionManager">
<property name = "dataSource" ref = "dataSource" />
</bean>

* JdO Transaction manager :- Used for Java data object transaction manager.

Syntax <bean id = "transactionManager"
class = "org.springframework.orm.jdo.JdoTransactionManager">
<property name = "persistanceManagerFactory" ref = "persistanceManagerFactory" />
</bean>

* JtaTransaction manager :- Used for Java transaction API transactions.

Syn <bean id = "transactionManager"
class = "org.springframework.transaction.jta.JtaTransactionManager">
<property name = "transactionManagerName" ref = "java:/TransactionManager" />
</bean>

Programatic

It is less preferable
more flexible TM through Code

Declarative

It is more preferable
less flexible TM through config file @ annotation.
It is preffrable in case of Cross cutting Concern, AOT can be modularized with AOP approach.

Spring Transaction Abstractions

→ The key to the Spring transaction abstraction is defined by the org.springframework.transaction.PlatformTransactionManager interface.

```
public interface PlatformTransactionManager {
    TransactionStatus getTransaction(TransactionDefinition) throws
    void commit(TransactionStatus status) throws TransactionException;
    void rollback(TransactionStatus status) throws TransactionException.
```

note:- getTransaction → this method returns a currently active transaction
@ creates a new one, according to the specified propagation behaviour.

① TransactionDefinition is the interface of the transaction Support-

```
public interface TransactionDefinition {
    int getPropagationBehaviour(); // this method returns Propagation
    int getIsolationLevel();       // spring offers all of Tran. Propagation options
    String getName();              // familiar from EJB CMT.
    int getTimeOut();              // this method returns the name of transaction.
    boolean isReadOnly();          // returns the time in seconds in which tran must
                                   // complete.
                                   // returns whether the transaction is read-only.
```

(N) TransactionStatus interface :- a simple way for transactional code to control transaction execution and query transaction status.

```
public interface TransactionStatus extends SavepointManager {
    boolean isNewTransaction();  // returns true in case the present tray is new.
    boolean hasSavepoint();  // nested transaction based on a savepoint
    void setRollbackOnly();  // sets transaction as rollback only.
    boolean isRollbackOnly();  returns transaction is marked as rollback only.
    boolean isCompleted();  // returns tray in completed i.e: wether it has
                                already been committed @ rollback.
```

* Propagation types :-

(1) TransactionDefinition.PROPAGATION_MANDATORY
(Supports a current Tray.; throws an exp if no current Trs exists.)

(2) TransactionDefinition.PROPAGATION_NESTED
→ execute with in a nested transaction if a current transaction exists.

(3) TransactionDefinition.PROPAGATION_NEVER
Do not support a current transaction; throw an exception if a current transaction exists.

(4) TransactionDefinition.PROPAGATION_NOT_SUPPORTED
Does not support a current transaction, rather always execute non transactionally.

(5) TransactionDefinition.PROPAGATION_REQUIRED
Supports a current transaction; creates a new one if non exists

(6) TransactionDefinition.PROPAGATION_REQUIRES_NEW

Creates a new transaction, suspending the current transaction if one exist.

(7) TransactionDefinition.PROPAGATION_SUPPORTS

Supports a current transaction; executes non-transactionally if none exists.

Isolation levels

(1) TransactionDefinition.ISOLATION_DEFAULT

→ this is the default isolation level.

(2) TransactionDefinition.ISOLATION_READ_COMMITTED

Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.

(3) TransactionDefinition.ISOLATION_READ_UNCOMMITTED

Indicates that dirty reads, non-repeatable reads, and phantom reads can occur.

(4) TransactionDefinitions.ISOLATION_REPEATABLE_READ

Indicates that dirty read and non-repeated reads are prevented; phantom reads can occur.

(5) TransactionDefinition.ISOLATION_SERIALIZABLE

Indicates that dirty reads, non repeatable reads and phantom reads are prevented.

2.2.1 Programmatic transaction management:

The Spring Framework provides two means of programmatic transaction management.

a. Using the TransactionTemplate (Recommended by Spring Team):

Let's see how to implement this type with the help of below code (taken from Spring docs with some changes).

Please note that the code snippets are referred from Spring Docs.

Context Xml file:

```
<!-- Initialization for data source -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverMana
<property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
<property name="username" value="root"/>
<property name="password" value="password"/>
</bean>

<!-- Initialization for TransactionManager -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.Da
<property name="dataSource"  ref="dataSource" />
</bean>

<!-- Definition for ServiceImpl bean -->
<bean id="serviceImpl" class="com.service.ServiceImpl">
<constructor-arg ref="transactionManager"/>
</bean>
```

Service Class:

```
public class ServiceImpl implements Service
{
  private final TransactionTemplate transactionTemplate;

  // use constructor-injection to supply the PlatformTransactionManager
  public ServiceImpl(PlatformTransactionManager transactionManager)
  {
this.transactionTemplate = new TransactionTemplate(transactionManager);
  }

  // the transaction settings can be set here explicitly if so desired henc
  //This can also be done in xml file
  this.transactionTemplate.setIsolationLevel(TransactionDefinition.ISOLATIO

  // and so forth...

  public Object someServiceMethod()
  {
    return transactionTemplate.execute(new TransactionCallback()
    {
     // the code in this method executes in a transactional context
     public Object doInTransaction(TransactionStatus status)
     {
     updateOperation1();
        return resultOfUpdateOperation2();
     }
   });
}}
```

If there is no return value, use the convenient TransactionCallbackWithoutResult class with an anonymous class as follows:

```
transactionTemplate.execute(new TransactionCallbackWithoutResult()
{
protected void doInTransactionWithoutResult(TransactionStatus status)
{
  updateOperation1();
   updateOperation2();
}
});
```

- Instances of the TransactionTemplate class are thread safe, in these instances do not maintain any conversational state.
- TransactionTemplate instances do however maintain configuration state, so while a number of classes may share a single instance of a TransactionTemplate, if a class needs to use a TransactionTemplate with different settings (for example, a different isolation level), then you need to create two distinct TransactionTemplate instances.

b. Using a PlatformTransactionManager implementation directly:

Let's see this option again with the help of code.

```
<!-- Initialization for data source -->
<bean id="dataSource" class="org.springframework.jdbc.datasource.DriverMana
<property name="driverClassName" value="com.mysql.jdbc.Driver"/>
<property name="url" value="jdbc:mysql://localhost:3306/TEST"/>
<property name="username" value="root"/>
<property name="password" value="password"/>
</bean>

<!-- Initialization for TransactionManager -->
<bean id="transactionManager" class="org.springframework.jdbc.datasource.Da
<property name="dataSource"  ref="dataSource" />
</bean>
```

```
public class ServiceImpl implements Service
{
 private PlatformTransactionManager transactionManager;

 public void setTransactionManager( PlatformTransactionManager transactionM
 {
   this.transactionManager = transactionManager;
 }

DefaultTransactionDefinition def = new DefaultTransactionDefinition();

// explicitly setting the transaction name is something that can only be do
def.setName("SomeTxName");
def.setPropagationBehavior(TransactionDefinition.PROPAGATION_REQUIRED);

TransactionStatus status = txManager.getTransaction(def);

try
{
// execute your business logic here
}
catch (Exception ex)
{
txManager.rollback(status);
throw ex;
}

txManager.commit(status);

}
```

Now, before going to next way of managing transaction i.e Declarative Transaction Management lets see how to choose which type of transaction management to go for.

Choosing between **Programmatic** and **Declarative Transaction Management**:

- Programmatic transaction management is good only if you have a small number of transactional operations. *(Most of the times, this is not the case.)*
- Transaction name can be explicitly set only using Programmatic transaction management.
- Programmatic transaction management should be used when you want explicit control over managing transactions.
- On the other hand, if your application has numerous transactional operations, declarative transaction management is worthwhile.
- Declarative Transaction management keeps transaction management out of business logic, and is not difficult to configure.

2.2.2 Declarative Transaction (Usually used almost in all scenarios of any web application)

Step 1: Define a transaction manager in your spring application context xml file.

```
bean id="txManager" class="org.springframework.jdbc.datasource.DataSourceTransac
```

```
1 <tx:annotation-driven transaction-manager="txManager"/>
```

Step 2: Turn on support for transaction annotations by adding below entry to your spring application context XML file.

OR add @EnableTransactionManagement to your configuration class as below:

```
1 @Configuration
2 @EnableTransactionManagement
3 public class AppConfig
4 {
5  ...
6 }
```

Spring recommends that you only annotate concrete classes (and methods of concrete classes) with @Transactional annotation as compared to annotating interfaces.

The reason for this is if you put an annotation on the Interface Level and if you are using class-based proxies (proxy-target-class="true") or the weaving-based aspect (mode="aspectj"), then the transaction settings are not recognized by the proxying and weaving infrastructure .i.e Transactional behaviour will not be applied.

Step 3: Add the @Transactional annotation to the Class (or method in a class) or Interface (or method in an interface).

```
1 <tx:annotation-driven proxy-target-class="true">
```

Default configuration: proxy-target-class="false"

- The @Transactional annotation may be placed before an interface definition, a method on an interface, a class definition, or a public method on a class.
- If you want some methods in the class (annotated with @Transactional) to have different attributes settings like isolation or propagation level then put annotation at method level which will override class level attribute settings.
- In proxy mode (which is the default), only 'external' method calls coming in through the proxy will be intercepted. This means that 'self-invocation', i.e. a method within the target object calling some other method of the target object, won't lead to an actual transaction at runtime even if the invoked method is marked with @Transactional.

@Transactional (isolation=Isolation.READ_COMMITTED)

- The default is Isolation.DEFAULT
- Most of the times, you will use default unless and until you have specific requirements.
- Informs the transaction (tx) manager that the following isolation level should be used for the current tx. Should be set at the point from where the tx starts because we cannot change the isolation level after starting a tx.

DEFAULT

Use the default isolation level of the underlying database.

READ_COMMITTED

A constant indicating that dirty reads are prevented; non-repeatable reads and phantom reads can occur.

READ_UNCOMMITTED

This isolation level states that a transaction may read data that is till uncommitted by other transactions.

REPEATABLE_READ

A constant indicating that dirty reads and non-repeatable reads are prevented; phantom reads can occur.

SERIALIZABLE

A constant indicating that dirty reads, non-repeatable reads, and phantom reads are prevented.

What do these Jargons dirty reads, phantom reads, or repeatable reads mean?

- **Dirty Reads**: Transaction 'A' writes a record. Meanwhile Transaction 'B' reads that same record before Transaction A commits. Later Transaction A decides to rollback and now we have changes in Transaction B that are inconsistent. This is a dirty read. Transaction B was running in READ_UNCOMMITTED isolation level so it was able to read Transaction A changes before a commit occurred.
- **Non-Repeatable Reads**: Transaction 'A' reads some record. Then Transaction 'B' writes that same record and commits. Later Transaction A reads that same record again and may get different values because Transaction B made changes to that record and committed. This is a non-repeatable read.
- **Phantom Reads:** Transaction 'A' reads a range of records. Meanwhile Transaction 'B' inserts a new record in the same range that Transaction A initially fetched and commits. Later Transaction A reads the same range again and will also get the record that Transaction B just inserted. This is a phantom read: a transaction fetched a range of records multiple times from the database and obtained different result sets (containing phantom records).

@Transactional(timeout=60)

Defaults to the default timeout of the underlying transaction system.

Informs the tx manager about the time duration to wait for an idle tx before a decision is taken to rollback non-responsive transactions.

@Transactional(propagation=Propagation.REQUIRED)

If not specified, the default propagational behavior is REQUIRED.

Other options are REQUIRES_NEW, MANDATORY, SUPPORTS, NOT_SUPPORTED, NEVER, and NESTED.

REQUIRED

- Indicates that the target method can not run without an active tx. If a tx has already been started before the invocation of this method, then it will continue in the same tx or a new tx would begin soon as this method is called.

REQUIRES_NEW

- Indicates that a new tx has to start every time the target method is called. If already a tx is going on, it will be suspended before starting a new one.

MANDATORY

- Indicates that the target method requires an active tx to be running. If a tx is not going on, it will fail by throwing an exception.

SUPPORTS

- Indicates that the target method can execute irrespective of a tx. If a tx is running, it will participate in the same tx. If executed without a tx it will still execute if no errors.
- Methods which fetch data are the best candidates for this option.

NOT_SUPPORTED

- Indicates that the target method doesn't require the transaction context to be propagated.
- Mostly those methods which run in a transaction but perform in-memory operations are the best candidates for this option.

NEVER

- Indicates that the target method will raise an exception if executed in a transactional process.
- This option is mostly not used in projects.

- This option is mostly not used in projects.

@Transactional (rollbackFor=Exception.class)

- Default is rollbackFor=RunTimeException.class
- In Spring, all API classes throw RuntimeException, which means if any method fails, the container will always rollback the ongoing transaction.
- The problem is only with checked exceptions. So this option can be used to declaratively rollback a transaction if Checked Exception occurs.

@Transactional (noRollbackFor=IllegalStateException.class)

- Indicates that a rollback should not be issued if the target method raises this exception.

*Now the last but most important step in transaction management is the **placement of @Transactional annotation**. Most of the times, there is a confusion where should the annotation be placed: at Service layer or DAO layer?*

@Transactional: Service or DAO Layer?

- The Service is the best place for putting @Transactional, service layer should hold the detail-level use case behavior for a user interaction that would logically go in a transaction.
- There are a lot of CRUD applications that don't have any significant business logic for them having a service layer that just passes data through between the controllers and data access objects is not useful. In these cases we can put transaction annotation on Dao.
- So in practice you can put them in either place, it's up to you.
- Also if you put @Transactional in DAO layer and if your DAO layer is getting resused by different services then it will be difficult to put it on DAO layer as different services may have different requirements.
- If your service layer is retrieving objects using Hibernate and let's say you have lazy initializations in your domain object definition then you need to have a transaction open in service layer else you will face LazyInitializationException thrown by the ORM.
- Consider another example where your Service layer may call two different DAO methods to perform DB operations. If your first DAO operation failed then other two may be still passed and you will end up inconsistent DB state. Annotating Service layer can save you from such situations.