(1) RTP
(2) Has-A Relationship

Interface 21 (alternative for EJBs)

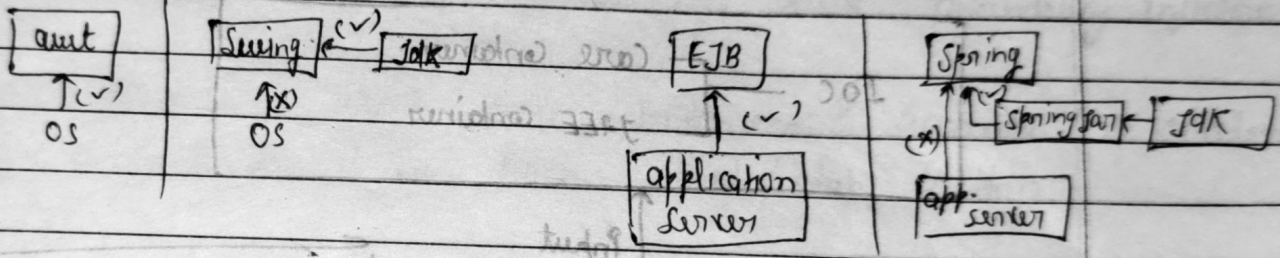
These makes our applications light weight.

⇒ awt are heavy weight because it uses OS libraries.

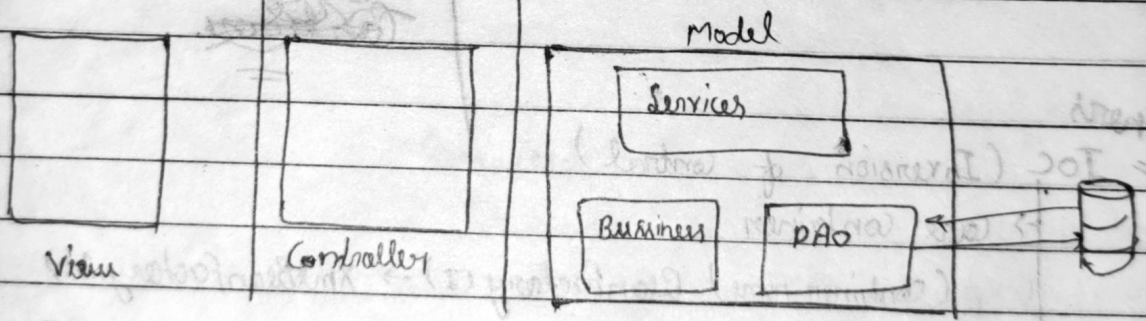
⇒ ~~Spring~~ Swing are light weight because it uses only Jdk libraries.

⇒ EJBs have dependency of application server so it is heavy weight.

⇒ while spring does not have any dependency on application server it uses simple spring jar along with jdk.



MVC

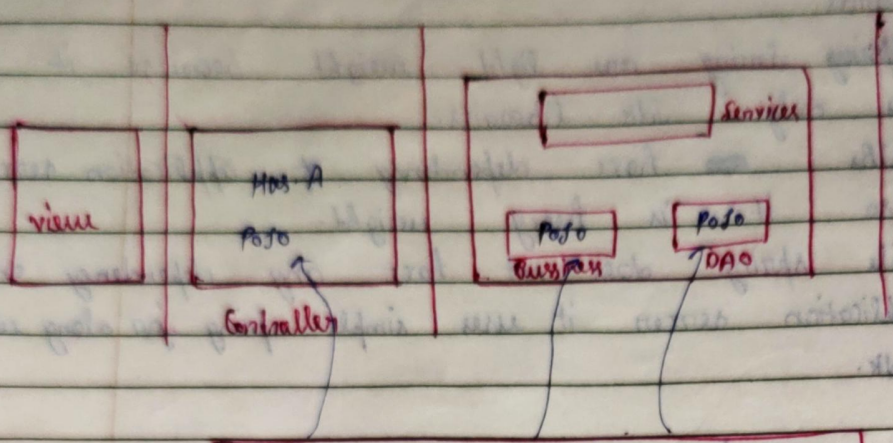


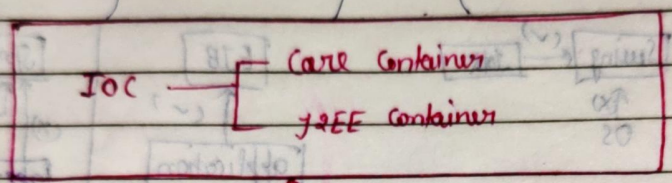
loosely coupled using RTP (Run time Polymorphism)

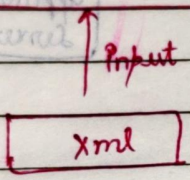
Every layer should be loosely coupled.

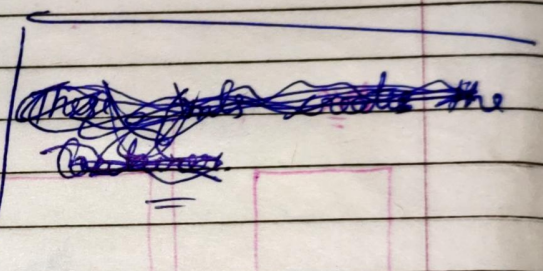
⇒ In case of servlets we need to extends GenericServlet class or HttpServletClass etc. while in case of spring we do not need to extend any other class.

→ spring recommends us using association instead of inheritance.









1.) Containers

IOC (Inversion of control)

→ Core Container

(Container name) BeanFactory (I) → XmlBeanFactory (C)

J2EE Container

(Container name) ApplicationContext (I)

(Container name) ConfigurableApplicationContext (I) → ClassPathXml - ApplicationContext (C)

MVC

Web

WebApplicationContext (I) → WebApplicationContextUtil (C)

(factory class)

similar like DriverManager

Servlet Container (In Tomcat)

(1) It can pass input from web.xml (xml) in case of IoC as a ServletConfig or ServletContext object.

(2) Creates servlet objects (Pojo class) in case of IoC

(3) manages life cycle of servlet (Pojo class) in case of IoC

IoC container works same as servlet containers.

IoC container passes dynamic input from xml to Pojo class, This is called Dependency Injection.

⇒ How to start these containers? Because in servlet container we have start/stop button.

Ans ⇒ These containers are interface so we'll have to create implemented classes.

eg.

```
class Test
{
    p s v m (String[] args)
    {
        new XmlBeanFactory();
        new ClassPathXmlApplicationContext
        new WebApplicationContextUtil.getObject()
    }
}
```

To write this code either should we put init() or service() in it?

Ans - init() method because we'll have to initialize these container once.

while using xml files in spring or Hibernet we must need to put dtd or xsd and main root tag is <beans></beans>.

Spring.xml

```
dtd/xsd
<beans>
    <bean class="" id="t">
    </beans>
</beans>
```

now we need to load this xml into Pojo class with help of driverManager

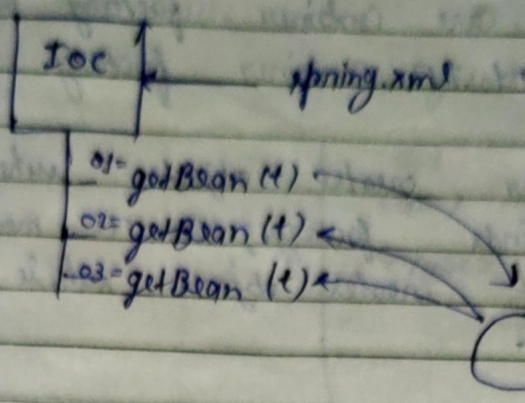
eg

```
class client
{
    p s v main(String[] args)
    {
        Resource r = new classpathResource("bk");
        BeanFactory factory = new XmlBeanFactory(r);
    }
}
```

Spring.xml

here we are using core container

Servlet class is not singleton but tomcat container only create a single servlet object in same way our Test class will be singleton if we try to make many objects it will still call the same object (or assign the same object in all reference variables.)



if in spring.xml singleton = prototype → only one object will be created.

if singleton = false → many objects will be created.

two scopes - (1) singleton ← true or false
(2) prototype

if we don't want to use singleton attribute we use >2.2 dtd. Then we use
scope = "prototype"
or scope = "singleton" by default

if it is web.xml then along with this scope we can add 3 more scopes -
(i) Request scope
(ii) session "
(iii) Context "
— These scopes are valid for web application not for standalone application.

These are att. used in bean tags.

Ques. difference b/w core container factory method and J2EE container factory method?

Ans - core container creates objects when demanded (when it finds factory.getBean() method), while applicationContext creates objects while loading (spring.xml).

\o

XmlBeanfactory is like "without load-on-startup" servlet container and ClasspathXmlApplicationContext is like "with load-on-startup" " "

because in without load-on-startup case at time of first user request ~~creation~~ it will create servlet object.

\o

(better approach) ApplicationContext is called Early (Eager) container.
BeanFactory is called lazy container.

IOC

what These containers do -

(1) will create new instance of POJO class
(2) manages the life cycle of POJO class
(3) Dependency injection into POJO class.

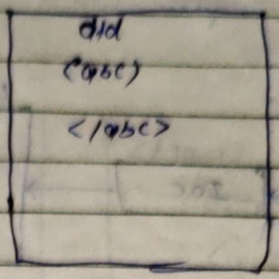
→ IOC container has a parser called SAX.

→ what dtd or xsd do, it tells weather these tags are available or not.

→ what parser does is - ~~[redacted]~~ validate (well formed or not) the xml.

like if we use

spring.xml



so This xml document is well formed but in not valid document.

→ no dtd check - validity of tags
SAX parser - proper formatted or not

Imp if bean scope is singleton only then ApplicationContext will creat pojo object at loading time but if bean scope is prototype then ApplicationContext works same as BeanFactory Container.

Que. So how IoC container create instance for our pojo class?

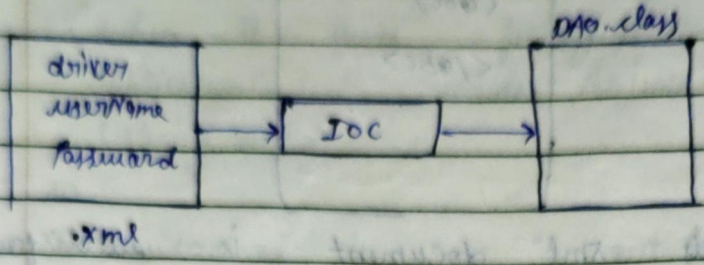
Ans. - IoC container uses -

Class.forName("Test").newInstance();

Imp IoC Container can Create Instances for private Constructor also.

Imp we can inject any dependency via our .xml file.

like if DAO. class requires driver, username, password we can pass it through xml.



⇒ So to take input from IOC, DAO. class must need to have either parameterized constructor or setter method.

(2) Two types of dependency injections - (DI)
(1) setter
(2) parameterized constructor

```
class Test
{
  private String gender;
public void setGender(String g)
  {
    this.gender = g;
  }

public void Hello(String name)
  {
    Sop("Hello" + gender + name);
  }
}
```

spring.xml

```
<beans>
  <bean id="t" class="Test">
    <property name="gender"
              value="Mr."/>
  </bean>
</beans>
```

IOC

→ we can pass value via value attribute or via value child tag.

* property tag is used to call setter method.

→ setter method should be single parameterized. and should not override in xml. (using name attribute to bean name). it will show duplicate value exception.

```
class Test
{
private String gender;
Test (String gender)
{
this.gender = gender;
}

public void Hello (String name)
{
sop ("Hello" + gender + name);
}
}
```

spring.xml

```
<beans>
<bean id="t" class="beans.Test">
<constructor-arg value="123" type="int">
</constructor-arg>
</bean>
</beans>
```

one argument constructor

SRF539047

spring.xml

```
<beans>
<bean id="t" class="beans.Test">
<constructor-arg value="name" type="String"/>
<constructor-arg value=20 type="int"/>
</bean>
</beans>
```

(two argument constructor)

if we define Type attribute then we don't have to worry with order of parameters.

So the constructor would be like:

```
Test (String name, int age)
{
}
```

→ But if there are two argument with same data type then there will be ambiguity.
Then we will use index attribute instead of type. as- <constructor-arg value="name" index="0" />
< " value="age" index="1" />

→ in case of constructor-arg with attribute index we can overload the constructor-arg. as- we can overload age parameter <constructor-arg value="age" index="1" /> } latest value will be assigned to the parameter.
< " value="age" index="1" />

if we have primary data type then we use - value attribute
if we have secondary " " " " " " - ref attribute
or by obj

Using ref

```
class Car
{
private Engine engine;
private String Carname;
// Setters
public void print()
{
Sop ("Carname" + Carname);
Sop("Model"+ engine);
}
}
```

```
class Engine
{
private String model;
// Setters
}
```

Engine.xml

```
<beans>
<bean id="e" class="Engine">
<property name="model" value="2015" />
</bean>
</beans>
```

Car.xml

```
<beans>
<bean id="c" class="Car">
<property name="Carname" value="Audi" />
<property name="engine" ref="e" />
</bean>
</beans>
```

→ Pass by reference

if we want to pass by object not by reference- then in property tag car.xml

```
<property name="engine">
    <bean class="Engine">
        <property name="model"
                  value="2015"/>
    </bean>
</property>
```

in innerBean (pass by object)

```
class Test{
    primitive String name[];
```

⇒ if we want to pass into array. Then we use <list> tag.

```
<value = ... >
<value = ... >
</list>
```

eg.

for primitive arrays

```
<property name="...">
    <list>
        <value= ... >
        <value= ... >
    </list>
</property>
```

⇒ for secondary arrays (using <list> <ref bean="c1"> </list>)

```
<beans>
    <bean id="c" class="Car" />
        <list>
            <ref bean="c1">
            <ref bean="c2">
        </list>
</beans>
```

if we want to pass Collection/arrays (←
<property> <list> </list> </property>

21/12/2019

(3.) Different type of dependent types -

(1) primitive types
(2) Secondary "
(3) Primitive arrays / Secondary arrays
(4) Collections

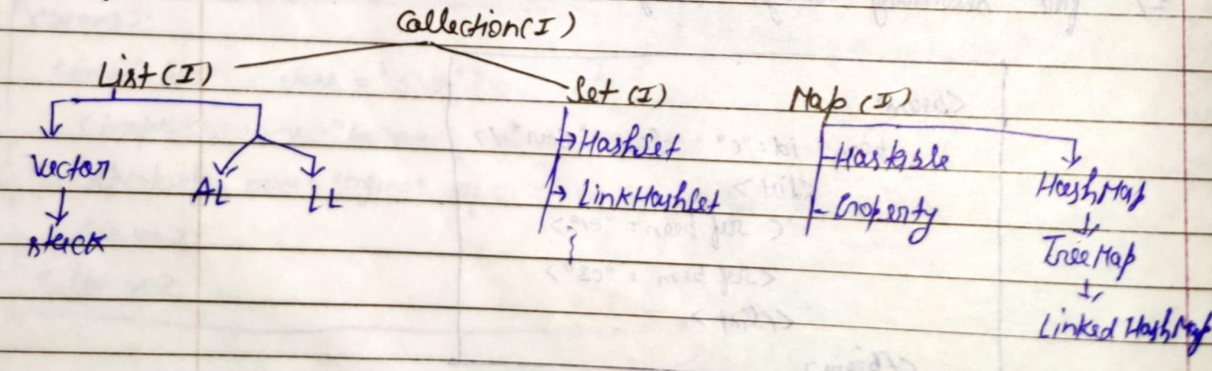
→ for primitive array -

```
<property>
  <list>
    <value = >
    :
  </list>
</property>
```

→ for secondary array -

```
<property>
  <list>
    <ref bean = "--" >
    :
  </list>
</property>
```

→ for collection -



80

```
List l = new Vector();
       = new Stack();
       = new ArrayList();
       = new LinkedList();
```

eg:

```
class Test
{
  private List fruits;
     "    Set Cricketers;
     "    Map Countries;
  // Setter fruits
  // Setter cricketers
  // Setter countries
}
```

Spring.xml

```
<beans>
  <bean id="t" class="Test">
    <property name="fruits">
      <list>
        <value> Apple </value>
        <value> Mango </value>
      </list>
    </property>
    <property name="cricketers">
      <set>
        <value> Sachin </value>
        <value> MSD </value>
      </set>
    </property>
    <property name="countries">
```

by default ArrayList

LinkedHashSet

LinkedHashMap

```
<map>
    <entry key = "IND" value = "delhi" />
    <entry key = "PAK" value = "islamabad" />
</map>
</property>
</bean>
</beans>
```

→ we need to add schema if we want to set specific collection object.

```
<beans xmlns = " beans namespace"
       xmlns : util = "util-namespace" >
  <bean id = "t" class = "Test" >
    <property name = "fruits" >
      <util: list list-class = " java.util.vector " >
        <value> Apple </value>
        <value> mango </value>
      </util:list >
    </property>
      :
      :
```

→ for properties file -

```
class Test
{
  private properties drivers

  //Setter drivers
}
```

```
<beans>
    <bean id="t" class="test">
        <property name="drivers">
            <props>
                <prop key="__">__</prop>
                :
            </props>
        </property>
    </bean>
</beans>
```

but here we can not read external properties file. for that, we'll have to use util namespace

```
<beans xmlns="beans-namespace"
       xmlns:utils="util-namespace">
    <bean id="t" class="test">
        <property name="drivers">
            <util:property location="classpath:resources/drivers.properties"/>
        </property>
    </bean>
</beans>
```

In case of setter DI it is not mandatory to ~~pass~~ inject a property. But in case of Constructor DI injecting properties are mandatory. We can make setter DI also compulsory by adding dependency-check = "simple" (for primitive data types) (in bean tag) "objects" (for secondary) "all" (if both primitive & secondary)

22/12/2019

Sec→IoC part-14 There are 3 types of annotations -

(1.) class level (above class)
(2.) Property level (only for class declaration)
(3.) Method level (above methods)

→ if we want to make some variables required then just place @Required above setter method of those variables.

so we'll have to activate this annotation. for this we'll (IOC Container) need to create one more class object.

@Required
↳ RequiredAnnotationBeanPostProcessor

Dependency for classes

⇒ Depends on - (apply dependency b/w classes) -

Spring.xml

```
<beans>
    <bean id="a" class="A" depends-on="b" />
    <bean id="b" class="B" depends-on="c" />
    <bean id="c" class="C" />
</beans>
```

now first it will create c class object, then B class object, and then A class object.

mutual dependencies are not possible.
↳ means if A depends on B so B can't depended on A.

Another method for DI via setter & Constructor

→ P- namespace // Setter
C- namespace // Constructor

eg:

```
<beans
    xmlns:p="p-namespace">
<bean id="t" class="Test" p:name="ABC"
                          p:Engine-ref="engine">
</beans>
```

Primitive

(for secondry only) Autowiring (automatic dependency injection)

main aim of autowiring

autowire = "byType" Setter DI
= "byName" "
= "Constructor" Constructor DI
= "autodetect" Setter & Constructor both DI
= "no" (by default)

→ <beans
default-autowire = "___" >

→ <bean autowire-candidate = "false"> Here this bean will not support for autowiring.