
Core Spring 5.x

Rajeev Gupta MTech CS
Java Trainer & Consultant

why spring is called non-invasic
:The Spring Framework doesn't force
the programmer to inherit any class
or implement any interface.
That is why it is said non-invasive.

class Account

class AccountDaoImpl extends

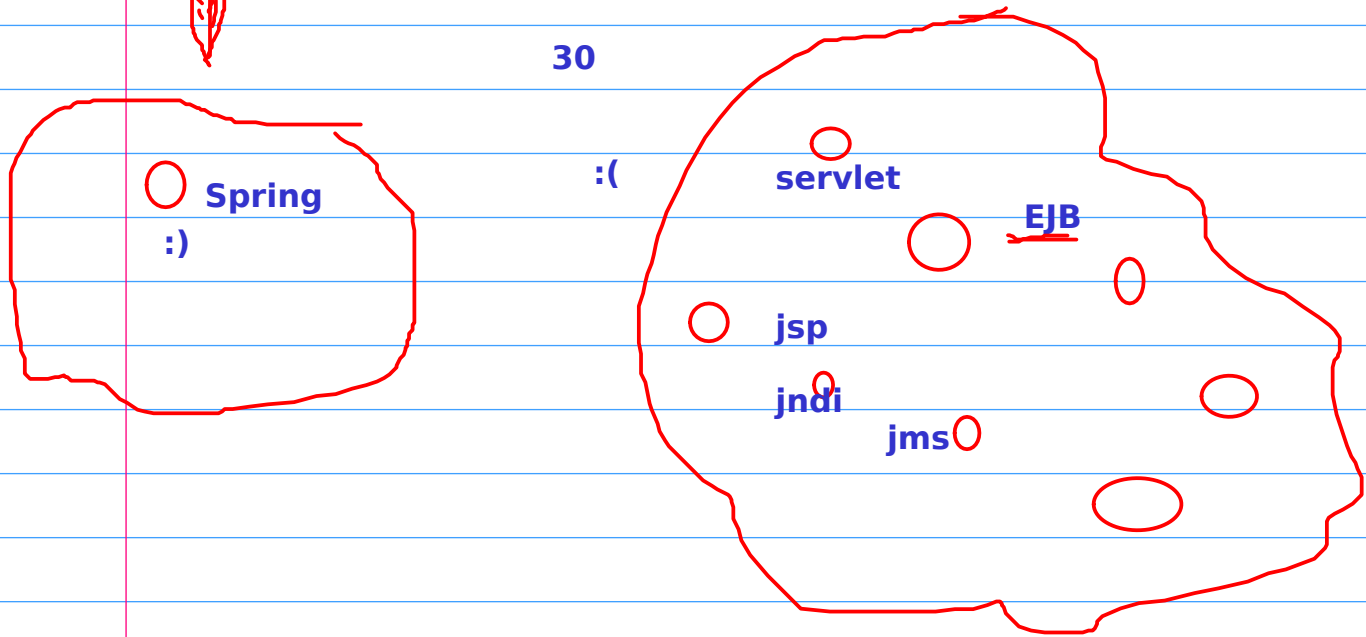
Agenda

c, p, annotation

- ▶ Introduction to Spring framework
- ▶ Dependency Injection using xml
 - ▶ Constructor, setter injection
 - ▶ C and p namespace
 - ▶ Scopes
 - ▶ Autowire
 - ▶ Collection mappings
 - ▶ Bean factory vs application context
 - ▶ Splitting configuration in multiple files
 - ▶ Bean life cycle
- ▶ Dependency Injection using annotation
 - ▶ @Autowired , @Qualifier, @Resource, @PostConstruct , @PreDestroy, @Service, @Repository
- ▶ Dependency Injection using java configuration
 - ▶ AnnotationConfigApplicationContext
 - ▶ @Configuration, @Bean, @Import, @Scope
 - ▶ @PropertySources
 - ▶ Using Environment to retrieve properties
- ▶ Using Java configuration
 - ▶ What are Profiles?
 - ▶ Activating profiles

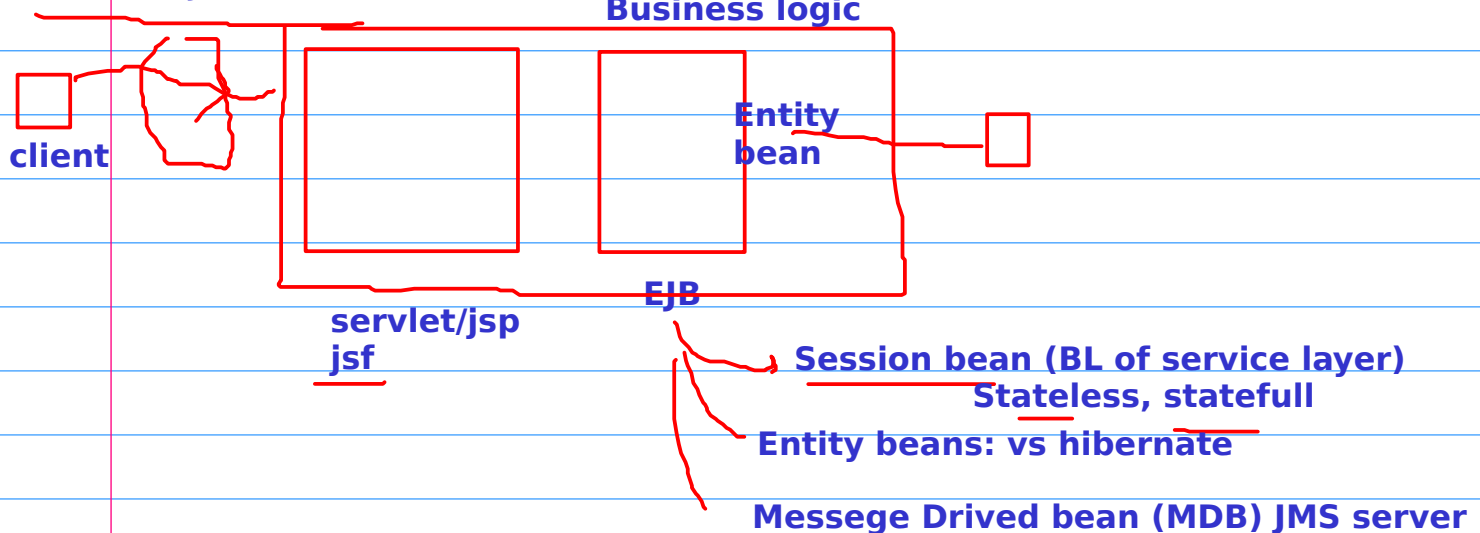
J2EE 1.4

group of spec to create dynamic and dist web app in java



Enterprise java bean

Classical J2EE model



too much confi

hello world: 2/3 interface, xml code, life cycle method

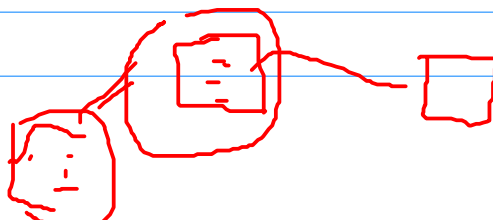
:(

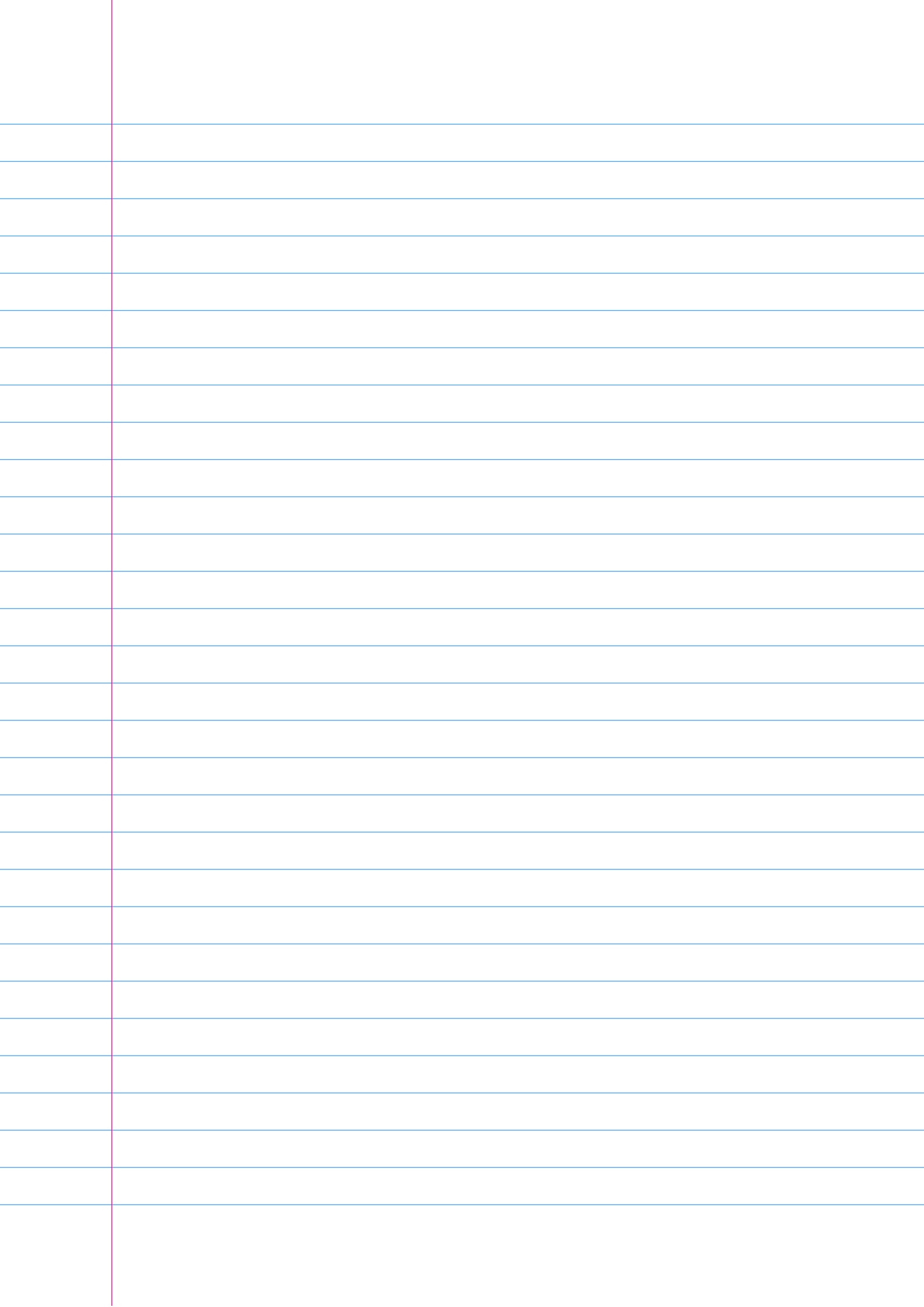
Business logic: logic for which customer pay u money...

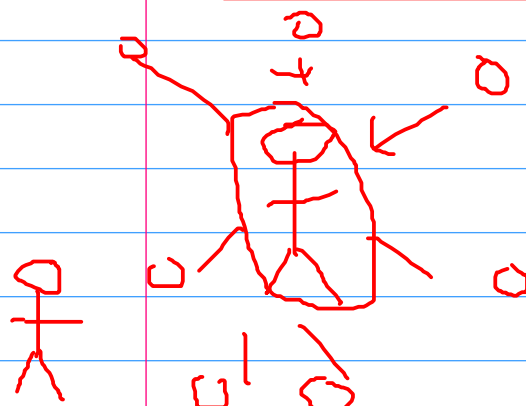
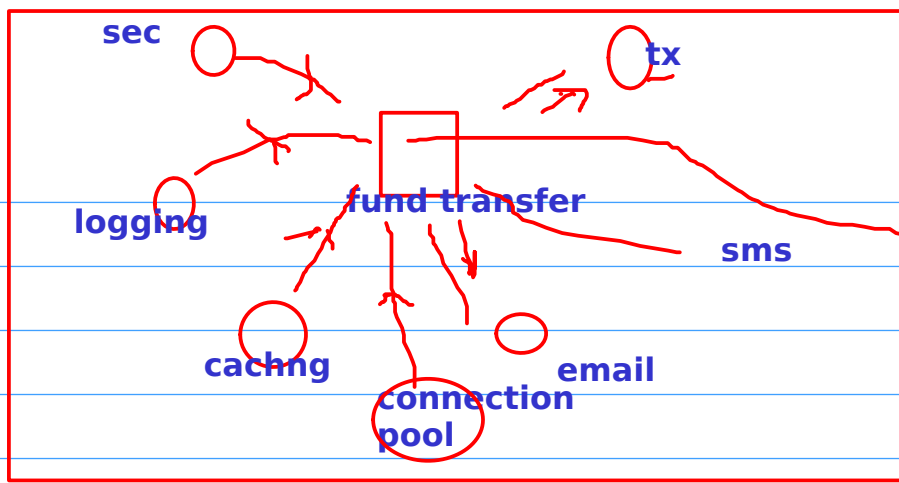
Use cases:

a customer can transfer money from his acc to another acc, provided he has money and valid transaction, it should be secure, performance should be, tx (acid) logging should...

FR + NFR







EJB 2.x (J2EE 1.4)

ejb ie BL need some services

that can be provided by container
EJB container(weblogic server)* heavy?

BL : pull model

"dependency" ?

```
class A{
  B b=new B();
}
```

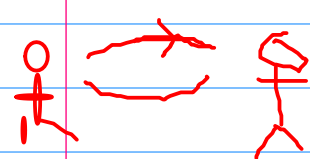
jndi api: is a way to access a remote services by ejb/ servlet
=> complicated, unit test, coupling

```
class BankService{
  BankDao dao=...
  EmailService emailService=...
}
```

BL: push model

DI vs IOC pattern

=> rod johnson
Spring framework started as lightweight IOC container that can manage life cycle of Business bean
it provide various dependencies to that BB using DI (push) approach

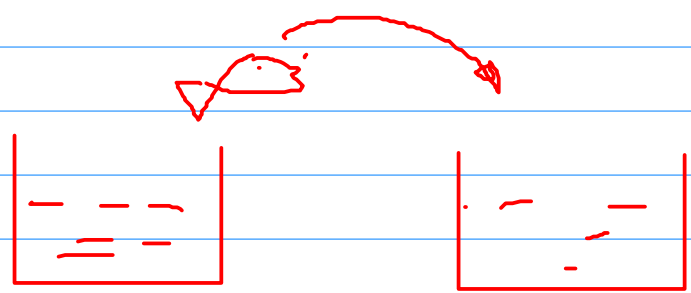


Spring ?

DI, AOP

EJB 3.x: JEE 5

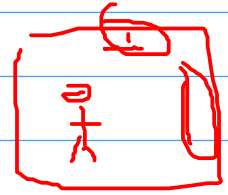
CDI
(Context DI)
Interceptor (not as powerful as AOP)
EJB



j2ee

spring

AOP (Aspect oriented problem) is a way to deal with Cross cutting concern it and it can be applied with DI design pattern



Java Naming and Directory Interface

JNDI

Connection pool

API:

this api is used to register a resource with logical name so that it can be accessed in location transparent way?

network programming

dist programming

socket prog
ip/port no

:)

google.com

DNS

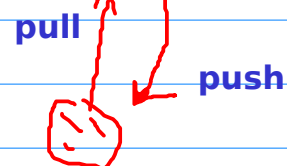
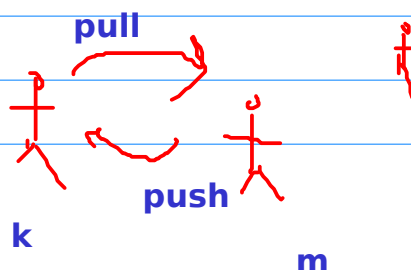
dist programming = network programming + location transparency

RMI
EJB
Remoting
web service
jndi/DI

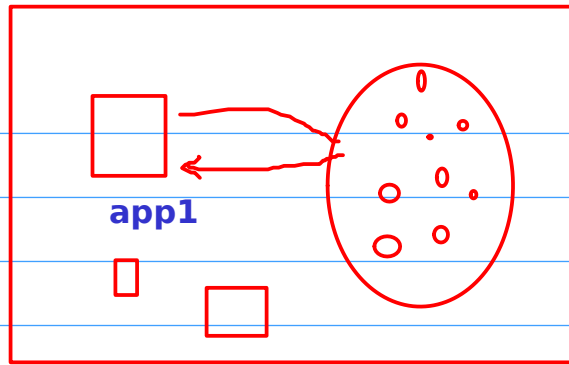
Connection pool vs DriverManager.getConnection()?

improved the performance

hold the readymade jdbc connection



**Connection
Session**



tomcat strat

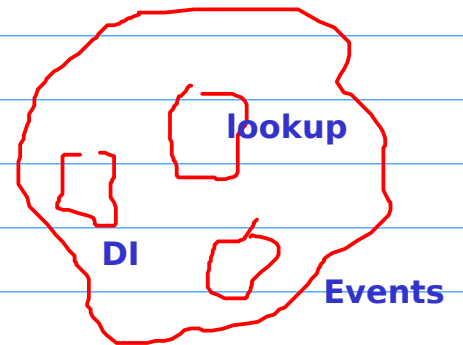
only one conn / per app
Bad design

DI : design pattern, which is used to inject the dependency by some external framework

Spring : IOC container

IOC vs DI

DI is one form of active IOC
(teaching is a way to earn money)



IOC (inversion of control)

tomcat:
servlet/jsp

events, listerns
framework ,tomcat
DI

IOC

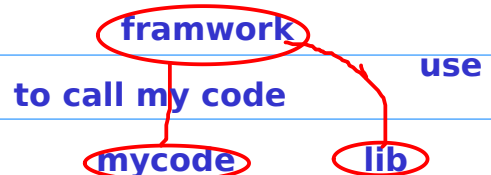
MyServlet ex... HttpServlet{

lib vs framework

configure

doGet(... ,){
}

container
}



DI: some external agency inject the dependency to ur component

What is spring?

Spring is a container that manage life cycle of business bean (POJO) and it use DI concept
it is also called lightweight framework (less heavy as compare to ejb model)
App server > tomcat

IOC container

it provide 3 main things:

DI

AOP

reduction of boilerplate code by using template design pattern

DI

AOP

- Reduce the boilerplate code?

- (template dp)

jdbcTemplate, JPATemplet, jms....

Agenda

- ▶ **Introduction to Spring framework**
- ▶ Dependency Injection using xml
 - ▶ Constructor, setter injection
 - ▶ C and p namespace
 - ▶ Scopes
 - ▶ Autowire
 - ▶ Collection mappings
 - ▶ Bean factory vs application context
 - ▶ Splitting configuration in multiple files
 - ▶ Bean life cycle
- ▶ Dependency Injection using annotation
 - ▶ @Autowired , @Qualifier, @Resource, @PostConstruct , @PreDestroy, @Service, @Repository
- ▶ Dependency Injection using java configuration
 - ▶ AnnotationConfigApplicationContext
 - ▶ @Configuration, @Bean, @Import, @Scope
 - ▶ @PropertySources
 - ▶ Using Environment to retrieve properties
- ▶ Using Java configuration
 - ▶ What are Profiles?
 - ▶ Activating profiles



Developed by: Divotal Software

bean wiring? how spring will inject one thing into another

for helping spring we need to configure this: xml , anno , java configure

Spring framework:

2003-06: Spring 1.x, Spring 2.x => configuration of bean XML

Spring 2.5 ==> configuration with annotations

2009: Spring 3.x==> java configuration

2013: Spring 4.0=> spring boot + all rest previous features

2017: Spring 5 => very interesting concept ie called reactive spring

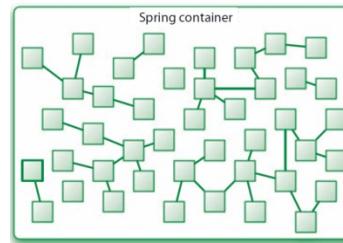
What is spring framework?

IOC

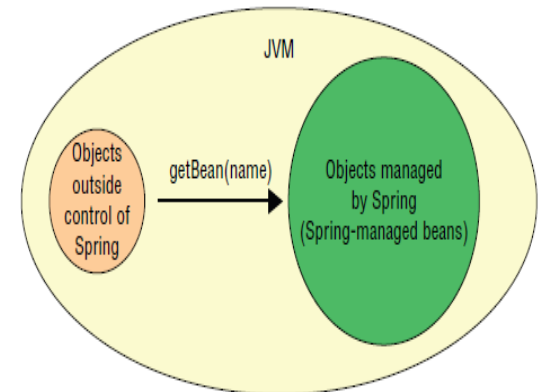
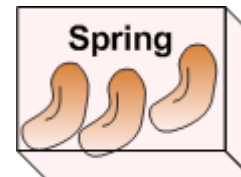
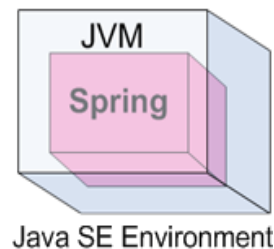
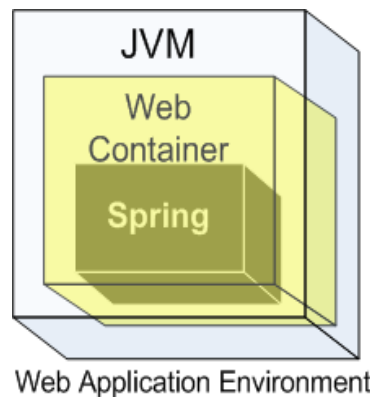
Spring framework is a container that manage life cycle of bean.

It Does 2 primary Jobs:

1. bean wiring **DI**
2. bean weaving **AOP**
3. reduction of boilerplate code



A bean is an object that is instantiated, assembled, and managed by a Spring IoC container.



Why Spring framework?

Spring Framework is focused on simplifying enterprise Java development through

- ✓ dependency injection
- aspect-oriented programming
- ▶ • boiler-plate code reduction using template design pattern

1. Spring as a container

1. Light weight container that do not need any installation, configurations start/stop activities.
2. Just collection of some jars

2. Spring an framework provides API

1. To integrate various technologies

Some of Spring modules

JDBC & DAO

Spring Dao
Support &
Spring jdbc
abstraction
framework

ORM

Spring
template
implementation
for
hibernate,
jpa, toplink etc

JEE

Spring Remoting
JMX
JMS
Email
EJB
RMI
WS
Hessian burlap

Web

Spring MVC
Support for various
frameworks
rich view support

AOP module

Spring AOP and AspectJ
integration

Spring Core Contrainer
The IOC Container

Need Of DI? An passenger need to travel

► Attempt 1:

```
public class Car {  
    public void move() {  
        System.out.println("Moving in a car!");  
    }  
}
```

```
public class Passanger {  
    private String name="raja";  
    private Car car=new Car();  
  
    public void travel(){  
        System.out.println("Passanger named:"+name);  
        car.move();  
    }  
}
```

Attempt 2:

```
public class Passanger {  
    private String name;  
    private Vehical vehical = new Car();  
  
    public void setName(String name) {  
        this.name = name;  
    }  
    public void setVehical(Vehical vehical) {  
        this.vehical = vehical;  
    }  
  
    public void travel() {  
        System.out.println("Passanger named:" + name);  
        vehical.move();  
    }  
}
```

```
public class Passanger {  
    private String name;  
    private Vehical vehical;  
  
    public void setName(String name) {  
        this.name = name;  
    }  
    public void setVehical(Vehical vehical) {  
        this.vehical = vehical;  
    }  
  
    public void travel() {  
        System.out.println("Passanger named:" + name);  
        vehical.move();  
    }  
}
```

```
public class Main {  
  
    public static void main(String[] args) {  
        Vehical vehical=new Car();  
        Passanger passanger=new Passanger();  
        passanger.setName("raja");  
  
        //WE NEED TO PASS VEHICAL TO PASSANGER, OTHERWISE NULL PE  
        passanger.setVehical(vehical);  
  
        passanger.travel();  
    }  
}
```


passanger

Vehical
Car
Bike

read , parsing (xml*)

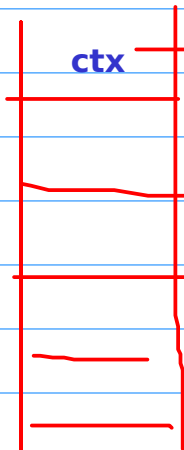
```
<bean id="p" class="com.demo.Passanger">  
  <property name="name" value="raj"/>  
  <property name="vehical" ref="v"/>  
</bean>  
<bean id="v" class="com.demo.Car"/>
```

ApplicationContext ctx=new ClassPathXmlApplicationContext("hello.xml");

Passanger passanger=(Passanger) ctx.getBean("p");

passanger.travel();

ctx object (container)

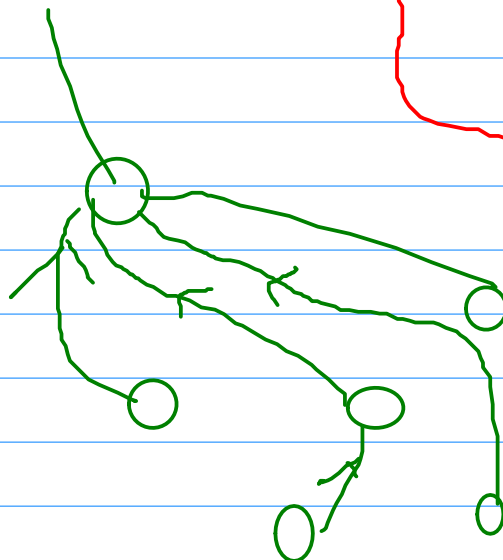


singleton :scope

passanger

"raj"

car





Map<String, Object>

push : spring is pushing the dep into passanger

```
public Passenger(String name, Integer age, Vehical vehical) {  
    this.name = name;  
    this.age = age;  
    this.vehical = vehical;  
}
```

Need Of DI? An passenger need to travel

```
public class Passanger {
    private String name;
    private Vehical vehical;

    public void setName(String name) {
        this.name = name;
    }
    public void setVehical(Vehical vehical) {
        this.vehical = vehical;
    }

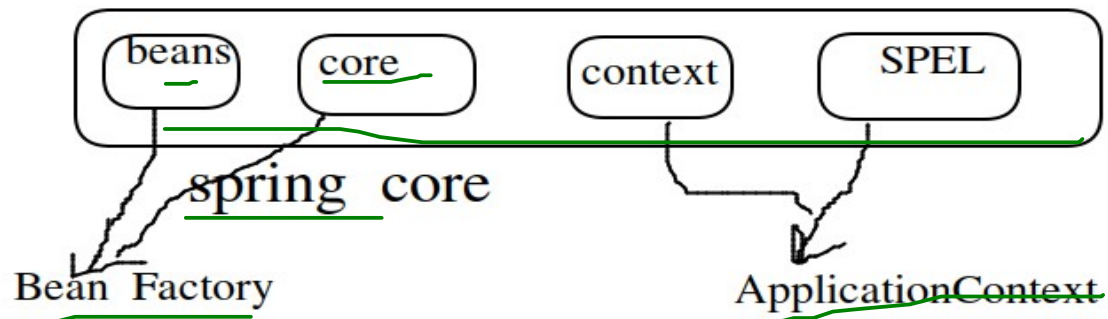
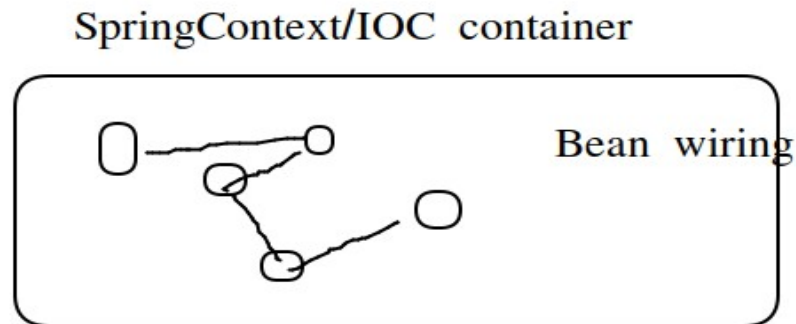
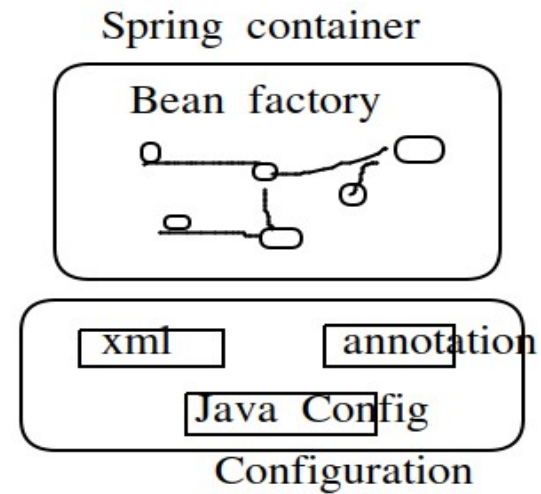
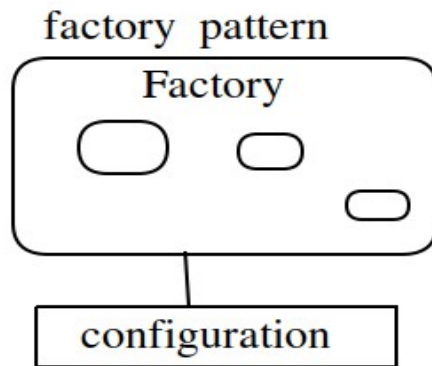
    public void travel() {
        System.out.println("Passanger named:" + name);
        vehical.move();
    }
}
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://www.springframework.org/schema/beans
        http://www.springframework.org/schema/beans/spring-beans.xsd">

    <bean id="passanger" class="com.sample.ex1.Passanger">
        <property name="name" value="raja"/>
        <property name="vehical" ref="v"/>
    </bean>
    <bean id="v" class="com.sample.ex1.Car"/>
</beans>
```

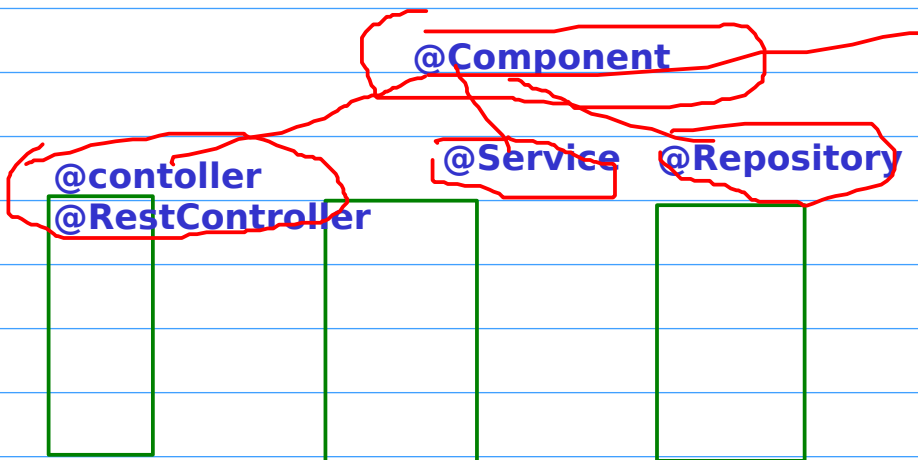
```
ApplicationContext context=new ClassPathXmlApplicationContext("beans.xml");
Passanger passanger=context.getBean("passanger", Passanger.class);

passanger.travel();
```



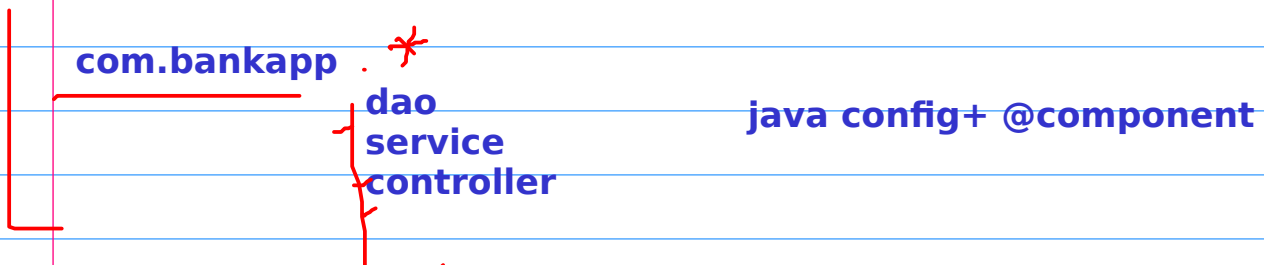
`ApplicationContext ctx=new ClassPathXmlApplicationContext("hello.xml");`

BeanFactory vs ApplicationContext



Bank application
dao
service :
controller

1. configuration using xml
2. annotation
3. java config



@Bean vs @Component *

`@Bean` should be used to configure infrastructure bean

`@Component` is used to configure business bean*

BL

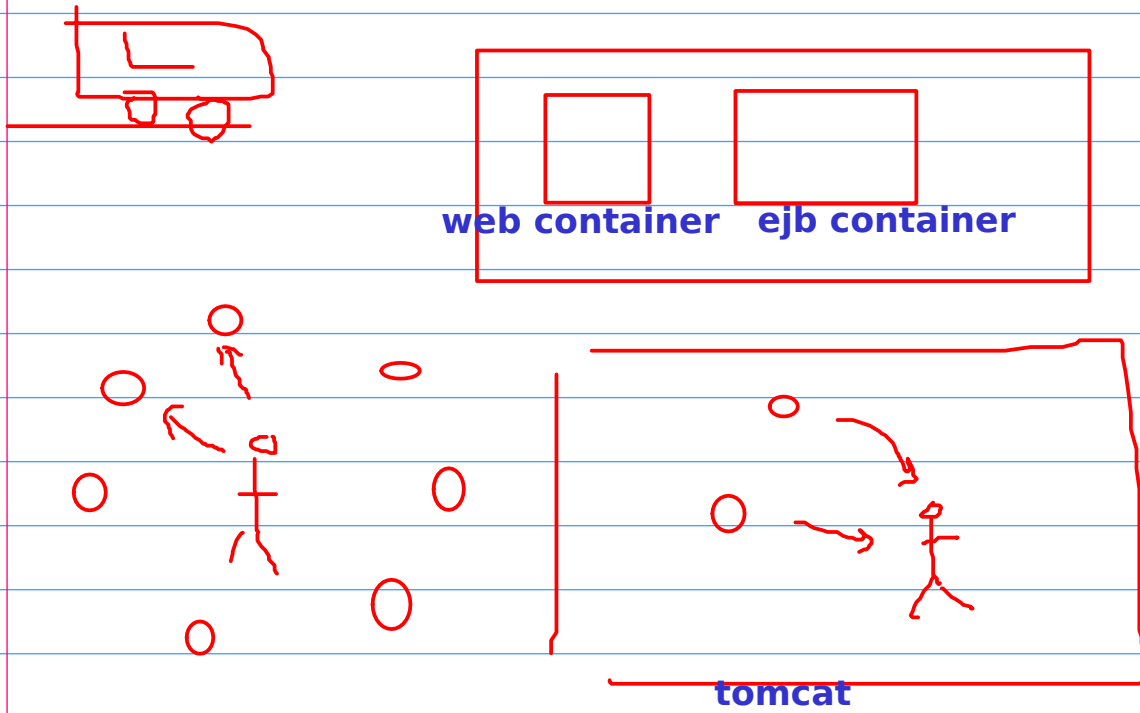
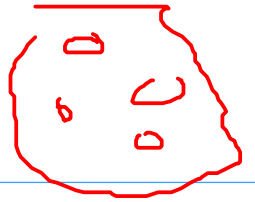
accountDao
accountService

database connection
hib conn
jms cont

Spring is lightweight container , ejb is a heavy container

EJB cant run on tomcat (servlet jsp)

EJB container: app server (glassfish, weblogic)





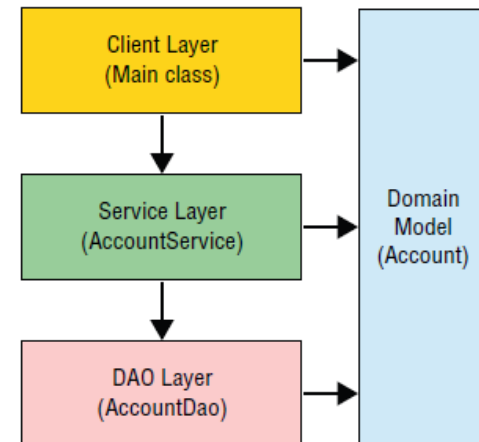
Bank Application

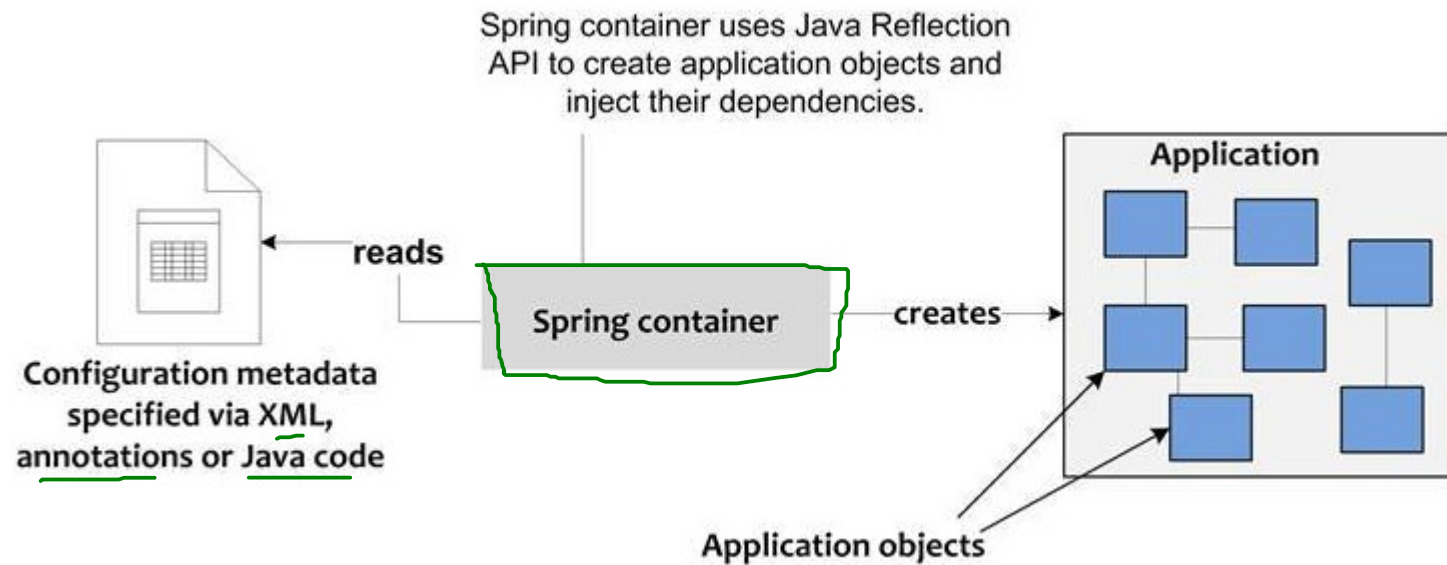
```
public class Account {  
    private int id;  
    private String name;  
    private double balance;  
}
```

```
public interface AccountDao {  
    public void update(Account account);  
    public Account find(int id);  
}
```

```
public class AccountDaoImp implements AccountDao {  
  
    private Map<Integer, Account> accouts = new HashMap<Integer, Account>();  
  
    public void update(Account account) {..}
```

```
public interface AccountService {  
    public void transfer(int from, int to, int amount);  
    public void deposit(int id, double amount);  
    public Account getAccount(int id);  
}
```





Agenda

- ▶ Introduction to Spring framework
- ▶ **Dependency Injection using xml**
 - ▶ **Constructor, setter injection**
 - ▶ **C and p namespace**
 - ▶ **Scopes**
 - ▶ **Autowire**
 - ▶ **Collection mappings**
 - ▶ **Bean factory vs application context**
 - ▶ **Splitting configuration in multiple files**
 - ▶ **Bean life cycle**
- ▶ Dependency Injection using annotation
 - ▶ @Autowired , @Qualifier, @Resource, @PostConstruct , @PreDestroy, @Service, @Repository
- ▶ Dependency Injection using java configuration
 - ▶ AnnotationConfigApplicationContext
 - ▶ @Configuration, @Bean, @Import, @Scope
 - ▶ @PropertySources
 - ▶ Using Environment to retrieve properties
- ▶ Using Java configuration
 - ▶ What are Profiles?
 - ▶ Activating profiles

Dependency Injection Using XML

▶ Setter Injection

```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp">
    <property name="accountDao" ref="accountDao"/>
</bean>
<bean id="accountDao" class="com.sample.bank.model.persistance.AccountDaoImp"/>
```

▶ Constructor Injection

```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp">
    <constructor-arg ref="accountDao"/>
</bean>
<bean id="accountDao" class="com.sample.bank.model.persistance.AccountDaoImp"/>
```

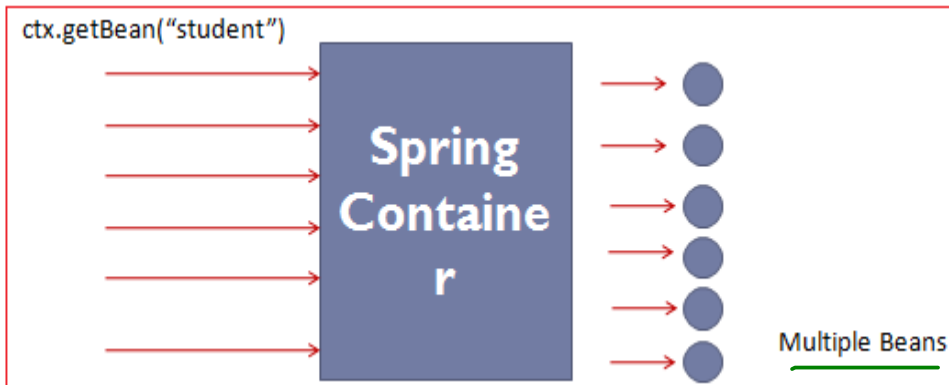
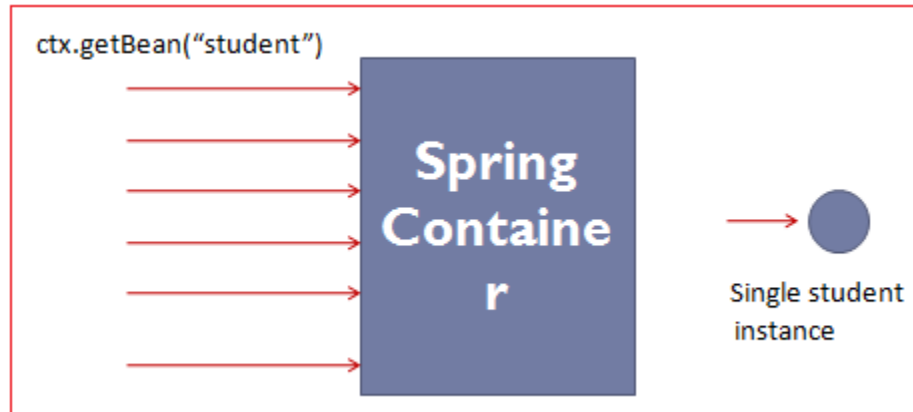
```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp" c:accountDao-ref="accountDao"/>
<bean id="accountDao" class="com.sample.bank.model.persistance.AccountDaoImp"/>
```

```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp" p:accountDao-ref="accountDao"/>
<bean id="accountDao" class="com.sample.bank.model.persistance.AccountDaoImp"/>
```

Bean Scopes

SCOPE NAME	SCOPE DEFINITION
<u>singleton</u>	Only one instance from a bean definition is created. It is the default scope for bean definitions.
<u>prototype</u>	Every access to the bean definition, either through other bean definitions or via the <code>getBean(..)</code> method, causes a new bean instance to be created. It is similar to the <code>new</code> operator in Java.
<u>request</u>	Same bean instance throughout the web request is used. Each web request causes a new bean instance to be created. It is only valid for web-aware <code>ApplicationContexts</code> .
<u>session</u>	Same bean instance will be used for a specific HTTP session. Different HTTP session creations cause new bean instances to be created. It is only valid for web-aware <code>ApplicationContexts</code> .
<u>globalSession</u>	It is similar to the standard HTTP Session scope (described earlier) and applies only in the context of portlet-based web applications.

Singleton vs Prototype



Autowiring

- ▶ AKA shortcut.
 - ▶ Default mode: Auto-Wiring "no"
 - ▶ Type of auto wiring: byName, byType

```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp" autowire="byName"/>  
<bean id="accountDao" class="com.sample.bank.model.persistence.AccountDaoImp"/>
```

```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp" autowire="byType"/>  
<bean id="accountDao1" class="com.sample.bank.model.persistence.AccountDaoImp"/>  
<bean id="accountDao2" class="com.sample.bank.model.persistence.AccountDaoImp"/>
```

Confusion?

```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp" autowire="byType"/>  
<bean id="accountDao1" class="com.sample.bank.model.persistence.AccountDaoImp"/>  
<bean id="accountDao2" class="com.sample.bank.model.persistence.AccountDaoImp" autowire-candidate="false"/>
```

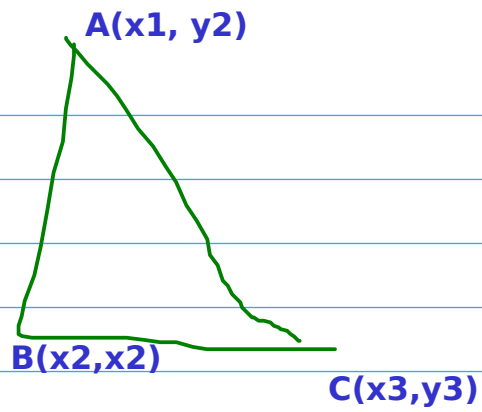
Collection Mapping

- ▶ Collection mapping supported by Spring :
 - ▶ List:<list> - </list>
 - ▶ Set:<set> - </set>
 - ▶ Map:<map> - </map>
 - ▶ Properties:<props> - </props>

```
<util:list list-class="java.util.LinkedList"
  id="mybestfriends">
  <value>Aman</value>
  <value>Ramn</value>
  <value>Ankit</value>
  <value>Rohit</value>
</util:list>

<bean class="com.springcore.standalone.collections.Person"
  name="person1">
  <!-- <property name="friends"> <ref bean="mybestfriends"/> </property>
  <property name="friends" ref="mybestfriends" />
</bean>
```

Passanger
Car bike



Point

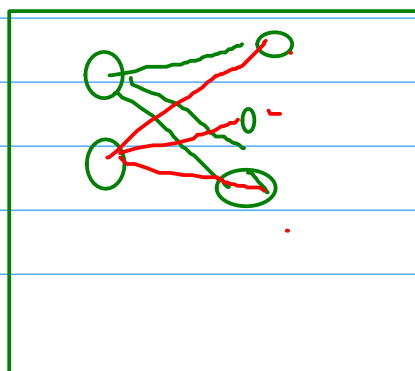
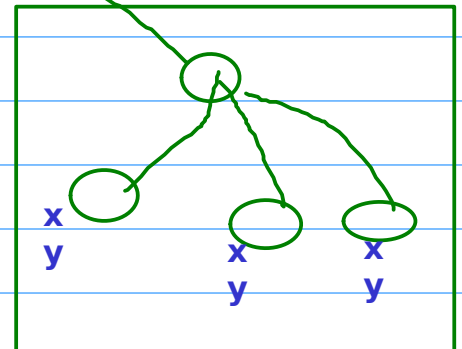
Shape

```
<bean id="tri" class="com.demo.Triangle">  
  <property name="pointA" ref="p1" />  
  <property name="pointB" ref="p2" />  
  <property name="pointC" ref="p3" />  
</bean>
```

```
<bean id="p1" class="com.demo.Point">  
  <property name="x" value="12" />  
  <property name="y" value="12" />  
</bean>
```

```
<bean id="p2" class="com.demo.Point">  
  <property name="x" value="1" />  
  <property name="y" value="-12" />  
</bean>
```

```
<bean id="p3" class="com.demo.Point">  
  <property name="x" value="5" />  
  <property name="y" value="7" />  
</bean>
```



List Collection Mapping

```
public class Account {
    private int id;
    private String name;
    private double balance;

    public class AccountCollection {
        private List<Account> accounts;

        public List<Account> getAccounts() {
            return accounts;
        }
    }
}
```

```
<bean id="account1" class="com.sample.bank.model.persistance.Account" p:id="1" p:name="raja" p:balance="2000.50"/>
<bean id="account1" class="com.sample.bank.model.persistance.Account" p:id="1" p:name="raja" p:balance="2000.50"/>

<bean id="accountCollection" class="com.sample.bank.model.persistance.AccountCollection">
    <property name="accounts">
        <list>
            <ref bean="account1"/>
            <ref bean="account2"/>
        </list>
    </property>
</bean>
```

[illegible]

Map Collection mapping

```
public class Account {  
    private int id;  
    private String name;  
    private double balance;
```

```
public class AccountCollection {  
    private Map<Integer, Account> accountMap;  
  
    public Map<Integer, Account> getAccountMap() {  
        return accountMap;  
    }  
}
```

```
<bean id="account1" class="com.sample.bank.model.persistence.Account" p:id="1" p:name="raja" p:balance="2000.50"/>  
<bean id="account2" class="com.sample.bank.model.persistence.Account" p:id="2" p:name="ravi" p:balance="4000.50"/>  
  
<bean id="accountCollection" class="com.sample.bank.model.persistence.AccountCollection">  
    <property name="accountMap">  
        <map>  
            <entry key="1" value-ref="account1"/>  
            <entry key="2" value-ref="account2"/>  
        </map>  
    </property>  
</bean>
```

BeanFactory

- ▶ Provides basic support for dependency injection
- ▶ Lightweight
- ▶ *XMLBeanFactory most commonly used implementation*

```
Resource xmlFile = new ClassPathResource( "META-INF/beans.xml" );  
  
BeanFactory beanFactory = new XmlBeanFactory( xmlFile );
```

```
MyBean myBean = (MyBean) beanFactory.getBean( "myBean" );
```

ApplicationContext

- ▶ Built on top of the BeanFactory
- ▶ Provides more enterprise-centric functionality
 - ▶ Internationalization of messages
 - ▶ AOP, transaction management

```
String xmlFilePath = "META-INF/beans.xml";  
ApplicationContext context = new ClassPathXmlApplicationContext( xmlFilePath );
```

```
MyBean myBean = (MyBean) context.getBean( "myBean" );
```

y in most

entation is
ontext

Splitting configuration in multiple file

▶ bean1.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- some configuration... -->

</beans>
```

bean2.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:c="http://www.springframework.org/schema/c"
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd">

    <!-- some configuration... -->

</beans>
```

```
ApplicationContext ctx = new ClassPathXmlApplicationContext(
    new String[] { "beans1.xml", "beans2.xml" });
```

```
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:c=
       xmlns:p="http://www.springframework.org/schema/p"
       xsi:schemaLocation="http://www.springframework.org/schema/bea
http://www.springframework.org/schema/beans/spring-beans.xsd":

    <import resource="common/Spring-Common.xml" />
    <import resource="connection/Spring-Connection.xml" />
    <import resource="moduleA/Spring-ModuleA.xml" />

</beans>
```

Bean wiring :

✓ xml

- annotation

- java config

@Primary

```
@Component(value = "car")
public class Car implements Vehical {
    @Override
    public void move() {
        System.out.println("moving in a car");
    }
}
```

```
@Component
public class Passanger {
```

```
    private Vehical vehical;
```

```
    @Autowired
    setVehical(Vehical v){
```

```
    }
```

```
} ctr injection
```

```
setter injection
```

```
@Component(value = "bike")
public class Bike implements Vehical {
    @Override
    public void move() {
        System.out.println("moving on a bike");
    }
}
```

older concept:

```
class Passanger{
```

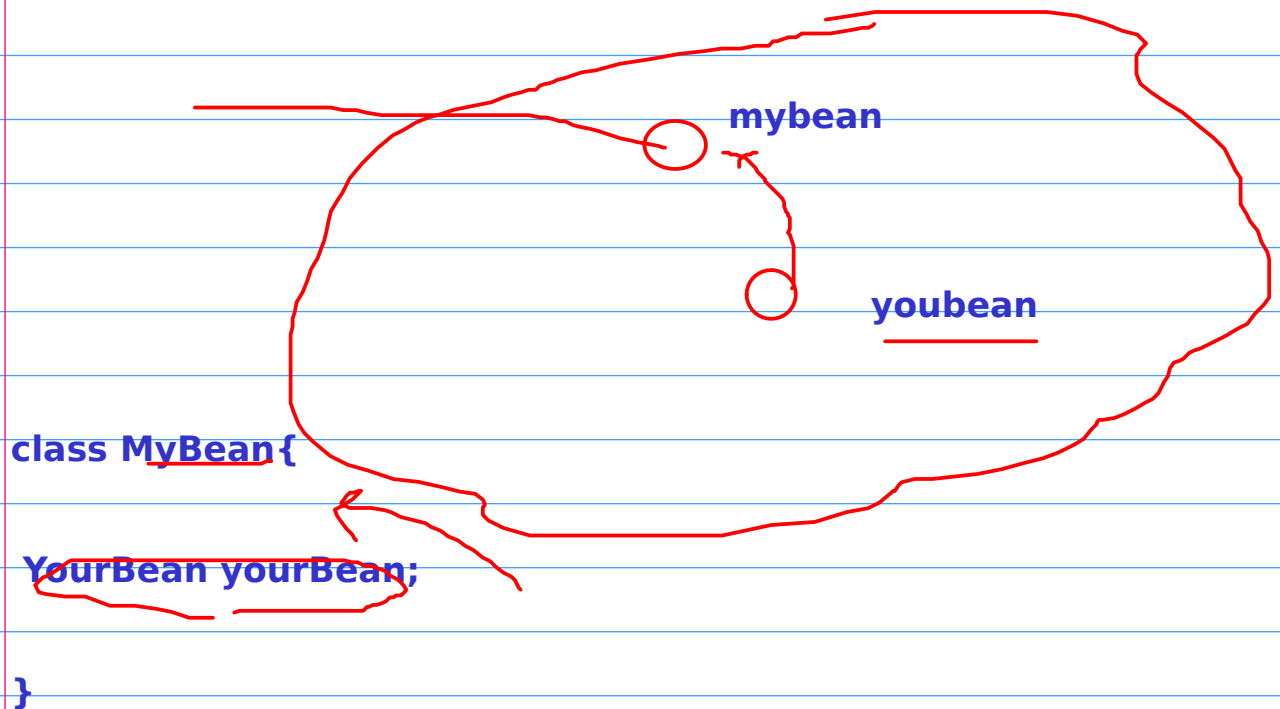
```
}
```

Spring bean life cycle:

=> xml

=> java config

=> annotation



1. spring will default ctr of mybean
2. it try to called life cycle init of it
3. then spring will try to finish life cyle of yourbean
3. then yourbean is injected to mybean
5. findally my bean creation is finished

Spring bean life cycle

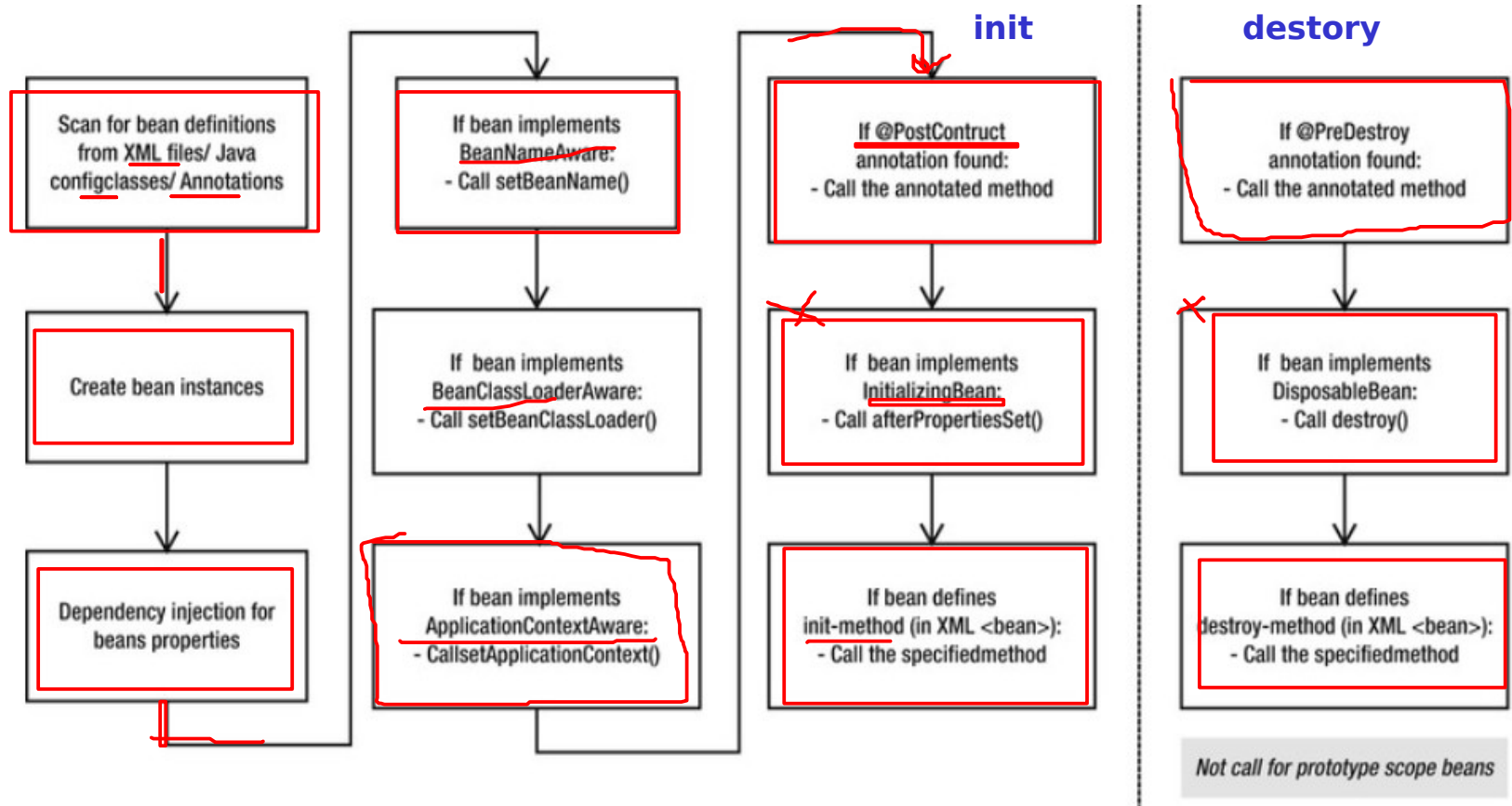


Figure 5-1. Spring beans life cycle

XXXXAware

Beans Life cycle

1. Spring instantiates the bean.
2. Spring injects values and bean references into the bean's properties.
3. If the bean implements `BeanNameAware`, Spring passes the bean's ID to the `setBeanName()` method.
4. If the bean implements `BeanFactoryAware`, Spring calls the `setBeanFactory()` method, passing in the bean factory itself.
5. If the bean implements `ApplicationContextAware`, Spring calls the `setApplicationContext()` method, passing in a reference to the enclosing application context.
6. If the bean implements the `BeanPostProcessor` interface, Spring calls its `postProcessBeforeInitialization()` method.
7. If the bean implements the `InitializingBean` interface, Spring calls its `afterPropertiesSet()` method. Similarly, if the bean was declared with an `init-method`, then the specified initialization method is called.
8. If the bean implements `BeanPostProcessor`, Spring calls its `postProcessAfterInitialization()` method.
9. At this point, the bean is ready to be used by the application and remains in the application context until the application context is destroyed.
10. If the bean implements the `DisposableBean` interface, Spring calls its `destroy()` method. Likewise, if the bean was declared with a `destroy-method`, the specified method is called.

BeanPostProcessor

aware vs XXPostProcessor

per bean they work for every bean
reg in ur project

```
public class DisplayBeanNamePostProcessor implements BeanPostProcessor {  
  
    @Override  
    public Object postProcessBeforeInitialization(Object bean, String beanName)  
        throws BeansException {  
        System.out.println("in before init method bean name is :" + beanName);  
        return bean;  
    }  
  
    @Override  
    public Object postProcessAfterInitialization(Object bean, String beanName)  
        throws BeansException {  
        System.out.println("in after init method bean name is :" + beanName);  
        return bean;  
    }  
}  
  
<bean class="com.sample.ex1.DisplayBeanNamePostProcessor"/>
```

BeanPostProcessor

Foo
ctr

#####

init()

#####

Bar
ctr

#####

init()

#####

BeanFactoryPostProcessor

only once at the time
of loading of all the
bean by beanfactory

Ex: `propertyPlaceholderConfigurer`

property file reading

The BeanPostProcessor interface defines callback methods that you can implement to provide your own (or override the container's default) instantiation logic, dependency-resolution logic, and so forth. If you want to implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean, you can plug in one or more BeanPostProcessor implementations.

So in essence the method `postProcessBeforeInitialization` defined in the BeanPostProcessor gets called (as the name indicates) before the initialization of beans and likewise the `postProcessAfterInitialization` gets called after the initialization of the bean.

The difference to the `@PostConstruct`, `InitializingBean` and custom `init` method is that these are defined on the bean itself. Their ordering can be found in the [Combining lifecycle mechanisms](#) section of the spring documentation.

So basically the BeanPostProcessor can be used to do custom instantiation logic for several beans whereas the others are defined on a per bean basis.

di
jdbc
spring hibernate
spring aop
spring mvc
spec sec

...

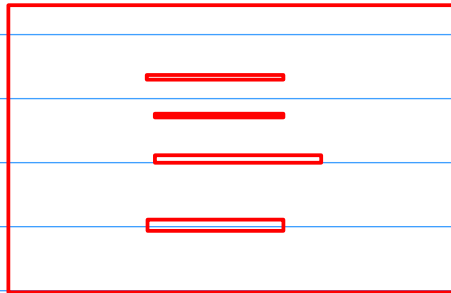
...

...

Spring "parent pom"

if u want to delay a project
choose java!

BOM



BeanFactoryPostProcessor

- ▶ BeanFactoryPostProcessor is invoked before bean factory is initialized (and before any bean is initialized)
- ▶ There are many BeanFactoryPostProcessor available by default in spring framework such as PropertyPlaceholderConfigurer
- ▶ PropertyPlaceholderConfigurer is used to read values from properties files before initialization of

```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
  
    @Override  
    public void postProcessBeanFactory(ConfigurableListableBeanFactory arg0)  
        throws BeansException {  
        System.out.println("my bean factory post processor is called...");  
    }  
  
}
```

```
<bean class="com.sample.ex1.MyBeanFactoryPostProcessor"/>
```

PropertyPlaceholderConfigurer

```
<bean id="account" class="com.sample.ex1.Account">
    <property name="id" value="${account.id}"/>
    <property name="name" value="${account.name}"/>
    <property name="balance" value="${account.balance}"/>
</bean>
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" >
    <property name="location" value="classpath:account.properties"/>
</bean>
```

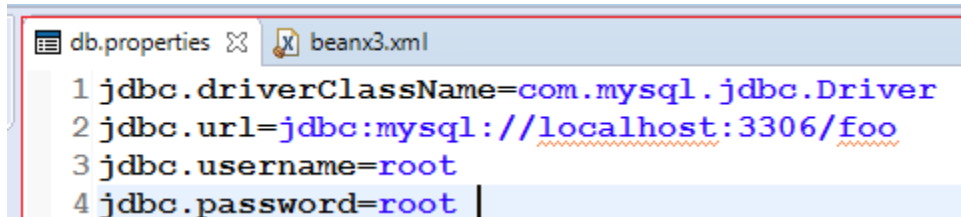
```
public class Account {
    private int id;
    private String name;
    private double balance;
```

```
account.properties [a
1 account.id=1
2 account.name=raja
3 account.balance=2000
```

PropertyPlaceholderConfigurer:DB Configuration

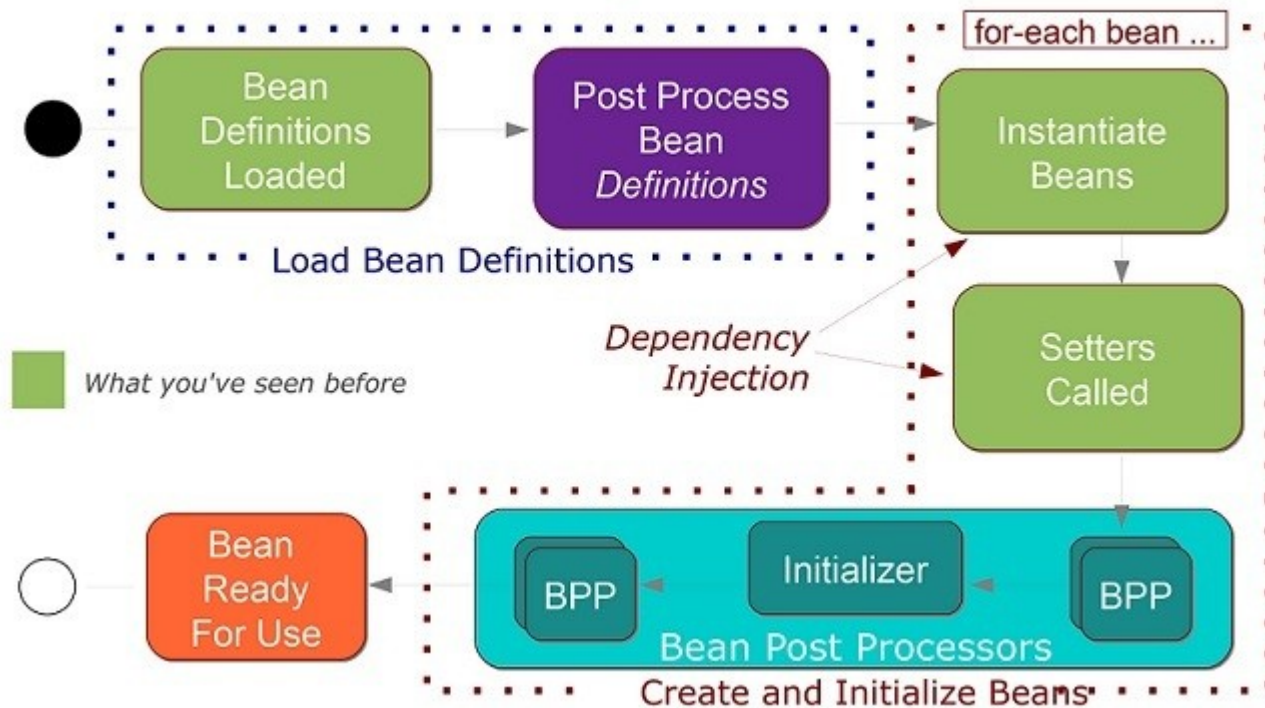
```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp">
    <property name="accountDao" ref="accountDao" />
</bean>
<bean id="accountDao" class="com.sample.bank.model.persistance.AccountDaoImpJdbc">
    <property name="dataSource" ref="dataSource" />
</bean>

<bean class="org.apache.commons.dbcp.BasicDataSource"
    destroy-method="close" id="dataSource">
    <property name="driverClassName" value="${jdbc.driverClassName}" />
    <property name="url" value="${jdbc.url}" />
    <property name="username" value="${jdbc.username}" />
    <property name="password" value="${jdbc.password}" />
</bean>
<bean
    class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer">
    <property name="location" value="db.properties"></property>
</bean>
```



```
db.properties
1 jdbc.driverClassName=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/foo
3 jdbc.username=root
4 jdbc.password=root
```

Bean Initialization Steps



```
@Component
```

```
public class MyBeanFFPP implements BeanFactoryPostProcessor{
```

```
    @Override
```

```
    public void postProcessBeanFactory(ConfigurableListableBeanFactory factory) throws BeansException {
```

```
        BeanDefinition beanDefinition=factory.getBeanDefinition("foo");
```

```
        MutablePropertyValues propertyValue=beanDefinition.getPropertyValues();
```

```
        propertyValue.addPropertyValue("foo", "a new foo value!");
```

```
    }
```

```
}
```



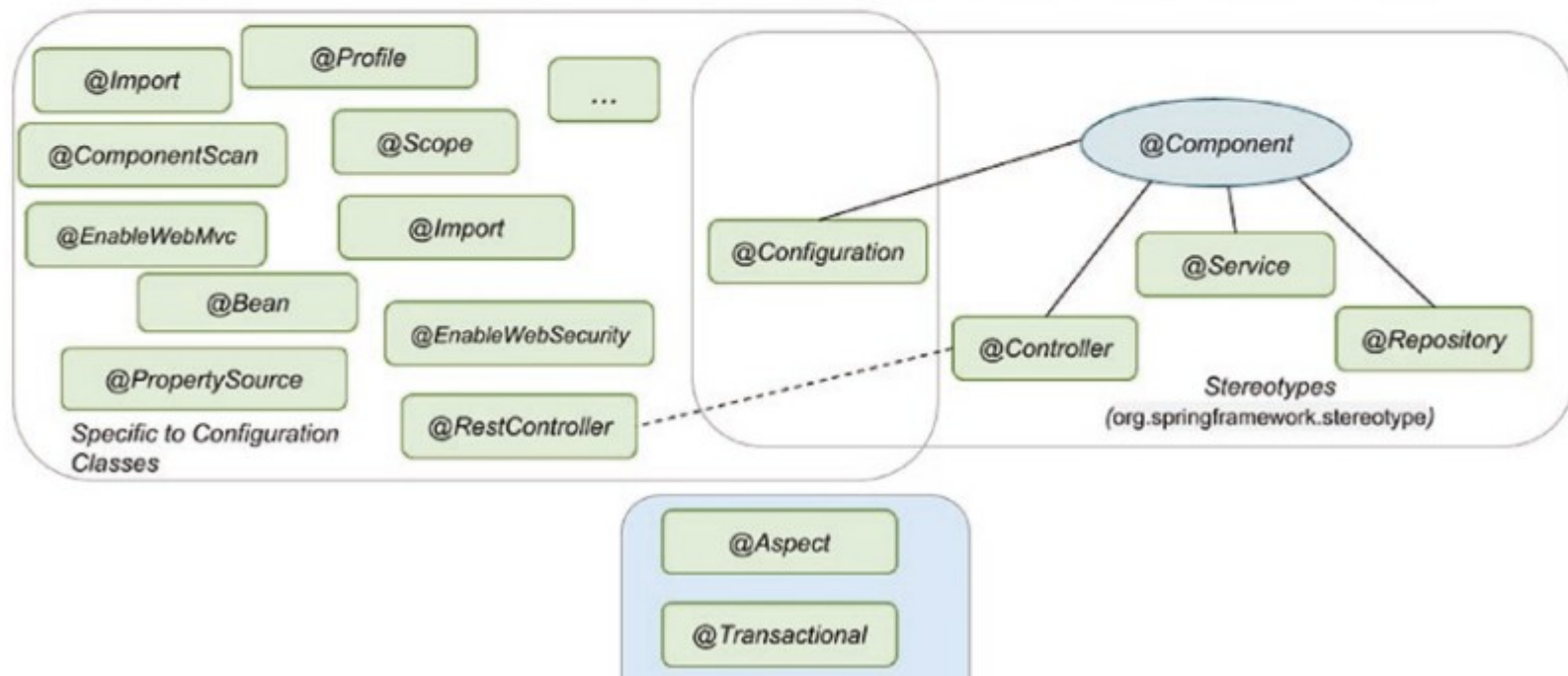
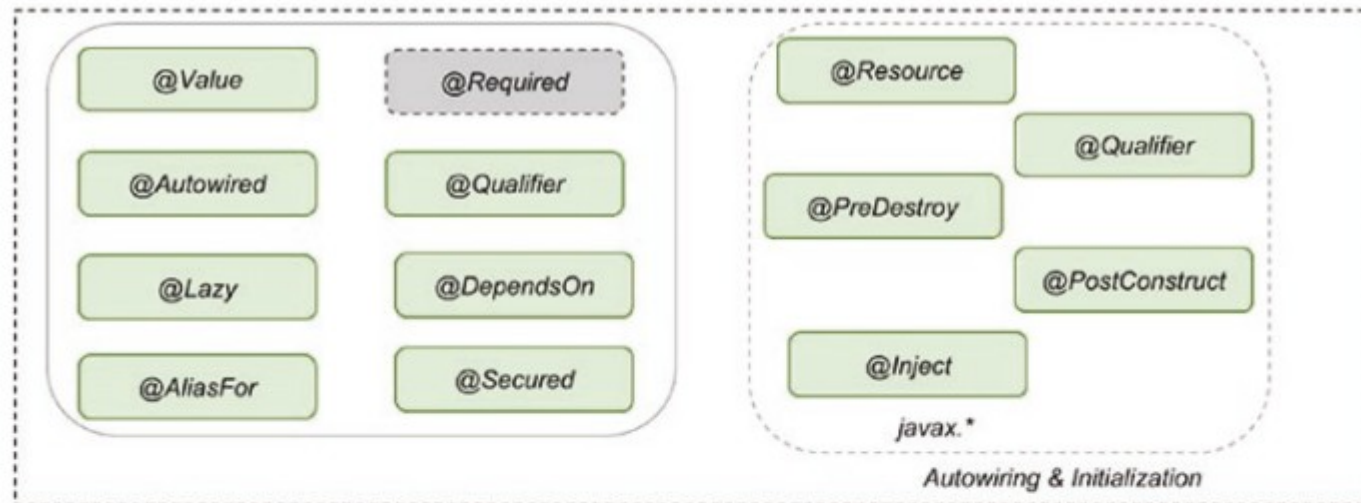
Agenda

- ▶ Introduction to Spring framework
- ▶ Dependency Injection using xml
 - ▶ Constructor, setter injection
 - ▶ C and p namespace
 - ▶ Scopes
 - ▶ Autowire
 - ▶ Collection mappings
 - ▶ Bean factory vs application context
 - ▶ Splitting configuration in multiple files
 - ▶ Bean life cycle
- ▶ **Dependency Injection using annotation**
 - ▶ **@Autowired , @Qualifier, @Resource, @PostConstruct , @PreDestroy, @Service, @Repository**
- ▶ Dependency Injection using java configuration
 - ▶ AnnotationConfigApplicationContext
 - ▶ @Configuration, @Bean, @Import, @Scope
 - ▶ @PropertySources
 - ▶ Using Environment to retrieve properties
- ▶ Using Java configuration
 - ▶ What are Profiles?
 - ▶ Activating profiles

Dependency Injection using annotations

- ▶ @Value - to inject a simple property
- ▶ @Autowired -to inject a property automatically
- ▶ @Component:@Controller @Service and @Repository
- ▶ @Qualifier - while autowiring, fix the name to an particular bean
- ▶ @Required - mandatory to inject, apply on setter
- ▶ @PostConstructs- Life cycle post
- ▶ @PreDestroy- Life cycle pre

- ▶ JSR 250 Annotations:
 - ▶ @Resource, @PostConstruct/ @PreDestroy, @Component
- ▶ JSR 330 Annotations:
 - ▶ @Named annotation in place of @Resource
 - ▶ @Inject annotation in place of @Autowired



Agenda

- ▶ Introduction to Spring framework
- ▶ Dependency Injection using xml
 - ▶ Constructor, setter injection
 - ▶ C and p namespace
 - ▶ Scopes
 - ▶ Autowire
 - ▶ Collection mappings
 - ▶ Bean factory vs application context
 - ▶ Splitting configuration in multiple files
 - ▶ Bean life cycle
- ▶ Dependency Injection using annotation
 - ▶ @Autowired , @Qualifier, @Resource, @PostConstruct , @PreDestroy, @Service, @Repository
- ▶ **Dependency Injection using java configuration**
 - ▶ **AnnotationConfigApplicationContext**
 - ▶ **@Configuration, @Bean, @Import, @Scope**
 - ▶ **@PropertySources**
 - ▶ **Using Environment to retrieve properties**
- ▶ Using Java configuration
 - ▶ What are Profiles?
 - ▶ Activating profiles

DI using Java Configuration

```
@Configuration
@ComponentScan(basePackages={"com.sample.bank.*"})
@Scope(value="prototype")
public class AppConfig {

    @Bean(autowire=Autowire.BY_TYPE)
    @Scope(value="prototype")
    public AccountService accountService() {
        AccountServiceImp accountService=new AccountServiceImp();
        //accountService.setAccountDao(accountDao());
        return accountService;
    }

    @Bean
    public AccountDao accountDao() {
        AccountDao accountDao=new AccountDaoImp();
        return accountDao;
    }
}

AnnotationConfigApplicationContext ctx=new AnnotationConfigApplicationContext(AppConfig.class);

AccountService s=ctx.getBean("accountService", AccountService.class);
s.transfer(1, 2, 100);
```

@PropertySource & Using Environment to retrieve properties

```
@Configuration
@ComponentScan(basePackages={"com.sample.bank.*"})
@PropertySource("classpath:db.properties")
public class AppConfig {
    @Autowired
    private Environment env;

    private Connection con;

    @Bean
    public Connection getConnection() {

        try{
            Class.forName("com.mysql.jdbc.Driver");
        } catch (ClassNotFoundException ex) {
            ex.printStackTrace();
        }
        try{
            con=DriverManager.getConnection(env.getProperty("jdbc.url"),
                env.getProperty("jdbc.username"),
                env.getProperty("jdbc.password"));
        } catch (SQLException ex) {
            ex.printStackTrace();
        }
        return con;
    }
}
```

```
1 jdbc.driverClassName=com.mysql.jdbc.Driver
2 jdbc.url=jdbc:mysql://localhost:3306/foo
3 jdbc.username=root
4 jdbc.password=root
```

```
AnnotationConfigApplicationContext ctx=new AnnotationConfigApplicationContext(AppConfig.class);
Connection con = (Connection) ctx.getBean("getConnection");
if (con!=null)
    System.out.println("done");
```

Agenda

- ▶ Introduction to Spring framework
- ▶ Dependency Injection using xml
 - ▶ Constructor, setter injection
 - ▶ C and p namespace
 - ▶ Scopes
 - ▶ Autowire
 - ▶ Collection mappings
 - ▶ Bean factory vs application context
 - ▶ Splitting configuration in multiple files
 - ▶ Bean life cycle
- ▶ Dependency Injection using annotation
 - ▶ @Autowired , @Qualifier, @Resource, @PostConstruct , @PreDestroy, @Service, @Repository
- ▶ Dependency Injection using java configuration
 - ▶ AnnotationConfigApplicationContext
 - ▶ @Configuration, @Bean, @Import, @Scope
 - ▶ @PropertySources
 - ▶ Using Environment to retrieve properties
- ▶ **Using Java configuration**
 - ▶ **What are Profiles?**
 - ▶ **Activating profiles**

Profile Using Java configuration

- ▶ What are Profiles?
 - ▶ @Profile allow developers to register beans by condition

```
public class Foo {  
    private String name;  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
}
```

```
@org.springframework.context.annotation.Configuration  
public class Configuration {  
  
    @Bean  
    @Profile("test")  
    public Foo testFoo(){  
        Foo foo=new Foo();  
        foo.setName("test");  
        return foo;  
    }  
  
    @Bean  
    @Profile("dev")  
    public Foo devFoo(){  
        Foo foo=new Foo();  
        foo.setName("dev");  
    }  
}
```

```
System.setProperty(AbstractEnvironment.ACTIVE_PROFILES_PROPERTY_NAME, "dev");  
ApplicationContext context = new AnnotationConfigApplicationContext(AppConfig.class);  
Foo foo = context.getBean(Foo.class);  
System.out.println(foo.getName());
```

no profile

- ▶ If profile "dev" is enabled, return a simple cache manager
ConcurrentMapCacheManager
- ▶ If profile "production" is enabled, return an advanced cache manager –
EhCacheCacheManager