Core Spring 5.x

Rajeev Gupta MTech CS
Java Trainer & Consultant

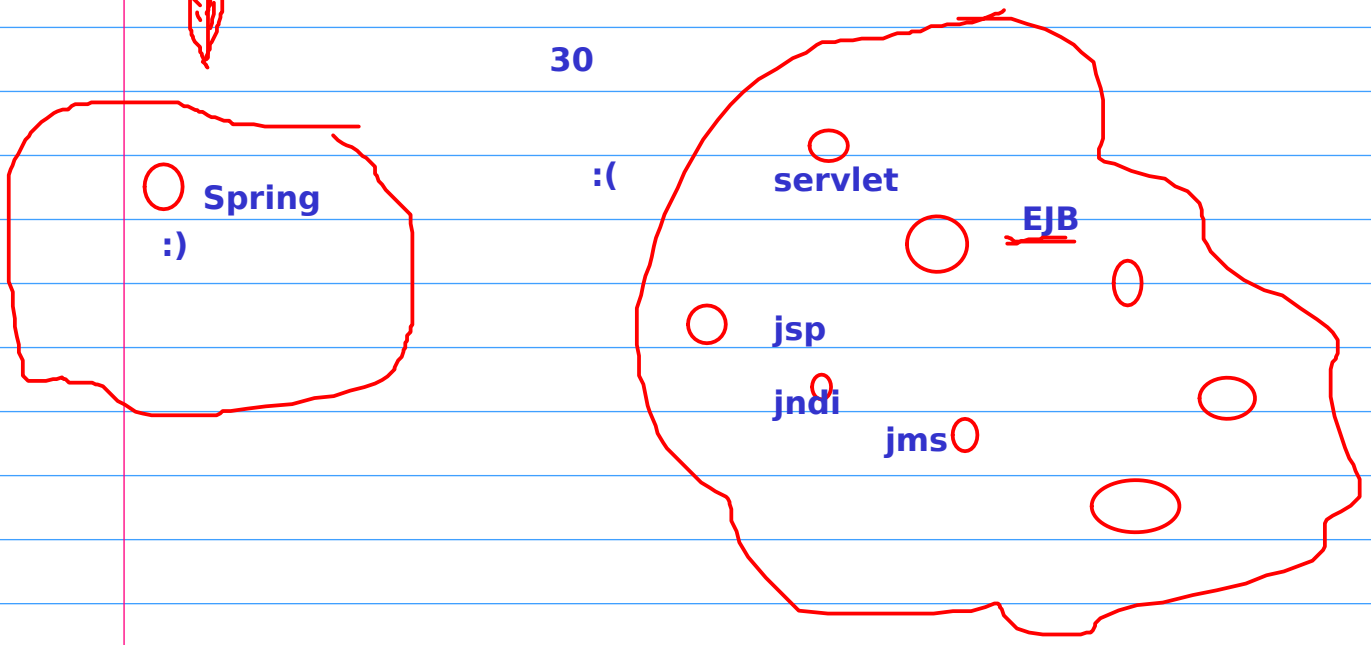
Agenda

- 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

J2EE 1.4

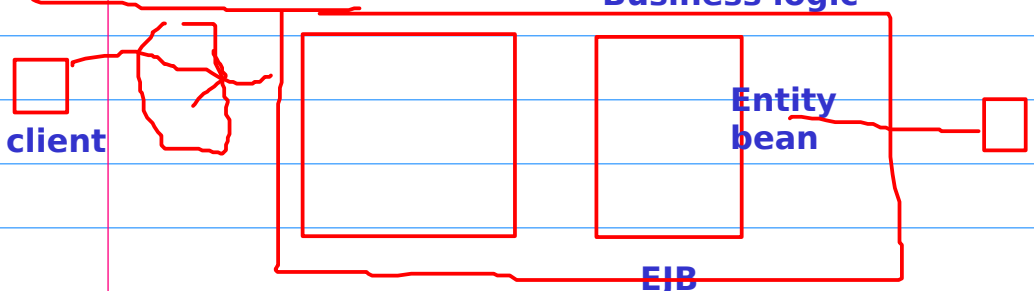
group of spec to create dynmic and dist web app in java

30



Enterprise java bean

Classical J2EE model



Session bean (BL of service layer)

Stateless, statefull

Entity beans: vs hibernate

Messege Drived bean (MDB) JMS server

too much confi

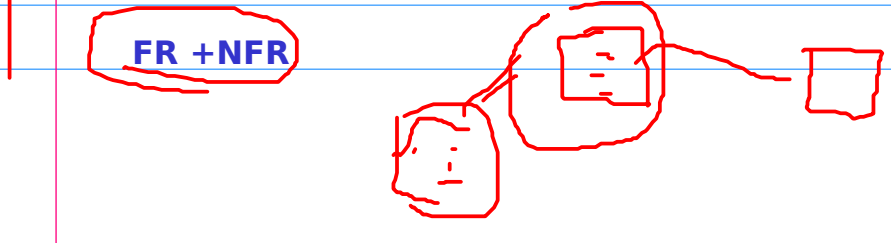
hello world: 2/3 inteface , xml code, life cycle method

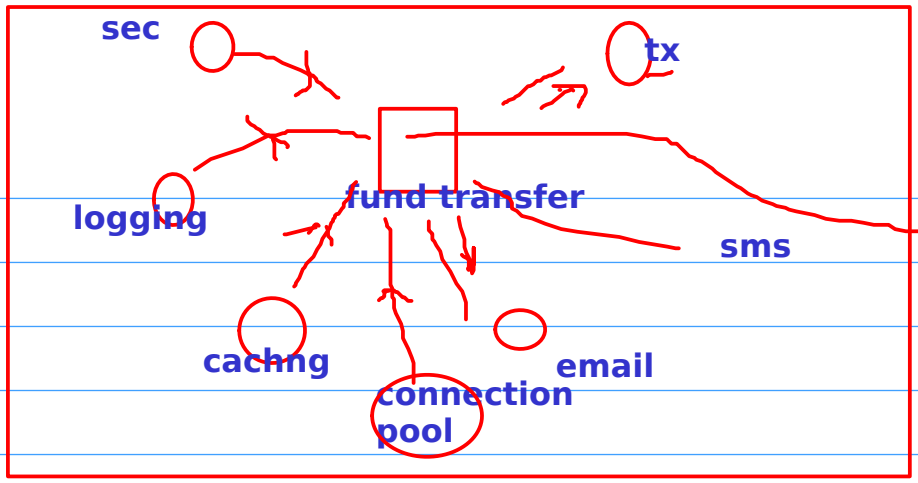
:(

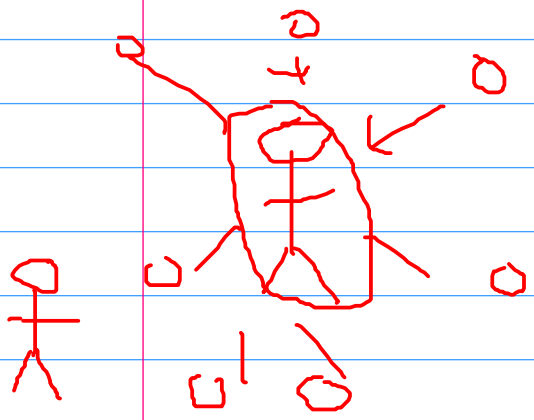
Busness logic: logic for which customer pay u moneny...

Use cases:

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







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" ?

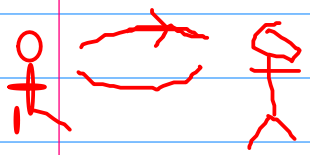
```
class A{
 B b=new B();

}

class BankService{

  BankDao dao=....
  EmailService emailService=...

}
```

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

BL: push model

DI vs IOC pattern

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



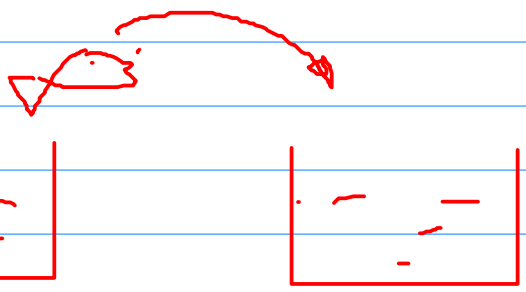
Spring ?

DI, AOP

EJB 3.x: JEE 5

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

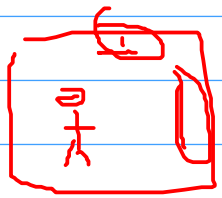
EJB



j2ee

spring

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



Java Naming and Directory Interface JNDI

Connection pool

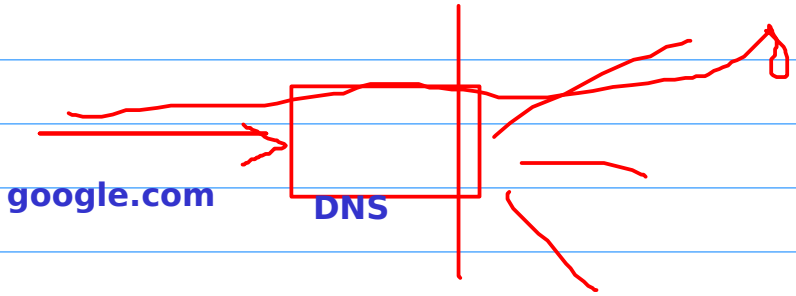
API:
this api is used to register a resouce wiht logical name so that
it can be access in location transpent way?

network programming

dist programming

socket prog
ip/port no

:)



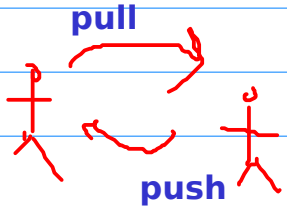
dist programming= network progamming + location transpracny

RMI
EJB
Remoting
web serivce
jndi/DI

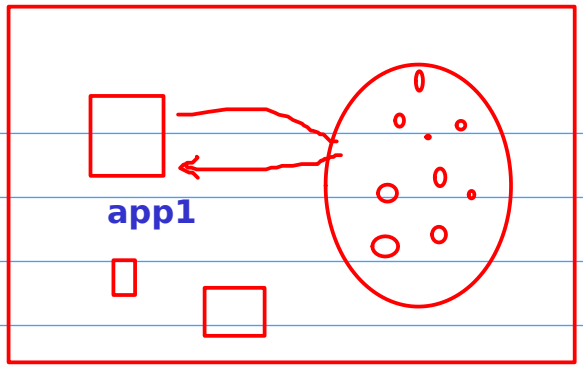
Connection pool vs DriverManager.getConnection()?

improved the performance

hold the readymade jdbc
connection



Connection
Session



tomcat strat

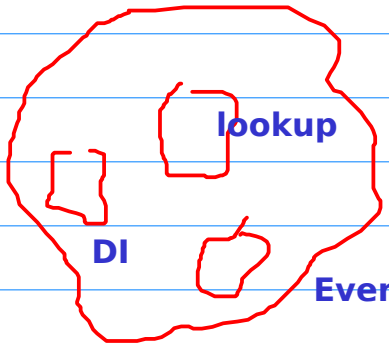
only one conn / per app
Bad design

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

Spring : IOC container

IOC vs DI

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



IOC (inversion of control)

- events,listerns
- framework ,tomcat
- DI

IOC

tomcat:
servlet/jsp

MyServlet ex... HttpServlet{

configre

doGet(... , ...){

}

container

}

lib vs framework

framwork

use

to call my code

mycode

lib

DI: some external agency inject the depency 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

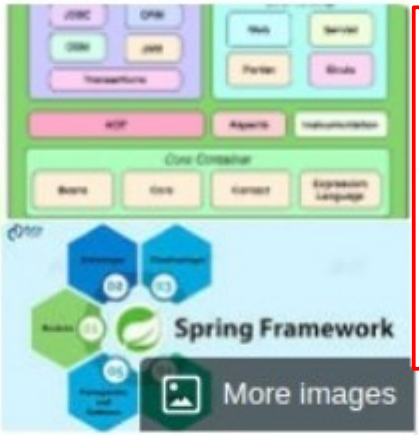
it provide 3 main things:

DI
AOP
reduction of boilerplate code by using template design pattern

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

What is spring? Why spring?



Rod Johnson

Australian technology specialist

Roderick "Rod" Johnson is an Australian computer specialist who created the Spring Framework and co-founded SpringSource, where he served as CEO until its 2009 acquisition by VMware. In 2011 Johnson became Chairman of Neo4j's Board of Directors. At the JavaOne 2012 it was announced that he joined the Typesafe Inc. Wikipedia

Spring Framework

The Spring Framework is an application framework and inversion of control container for the Java platform. The framework's core features can be used by any Java application, but there are extensions for building web applications on top of the Java EE platform. Wikipedia

Stable release: 5.2.7.RELEASE / June 9, 2020; 55 days ago

Preview release: 5.2.0-RC1 / August 5, 2019; 11 months ago

Developed by: Pivotal 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 => configration of bean XML

Spring 2.5 ==> configration with annotations

2009: Spring 3.x==> java configration

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

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

---

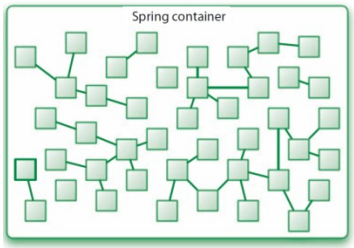
What is spring framework?

IOC

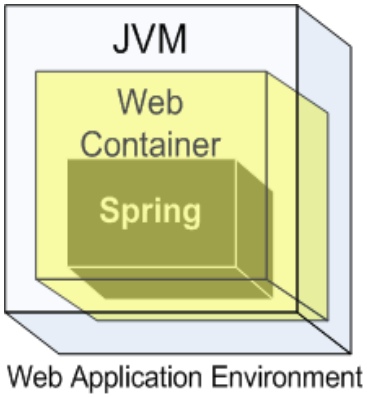
Spring framework is a contrainer that manage life cycle of bean.

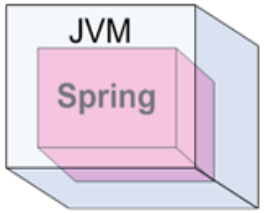
It Does 2 primary Jobs:

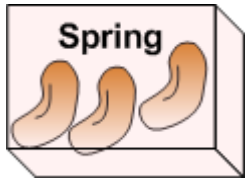
1. bean wiring DI
2. bean weaving AOP

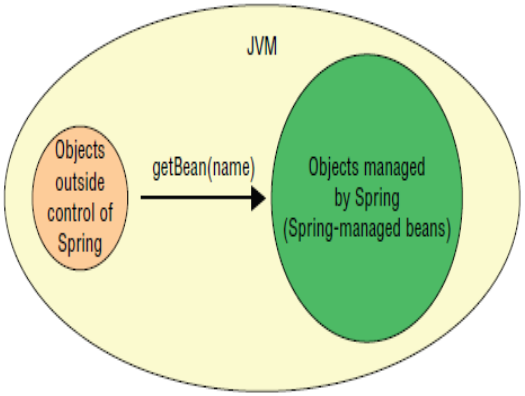


A bean is an object that is instantiated, assembled, and managed by a Spring IoC container.









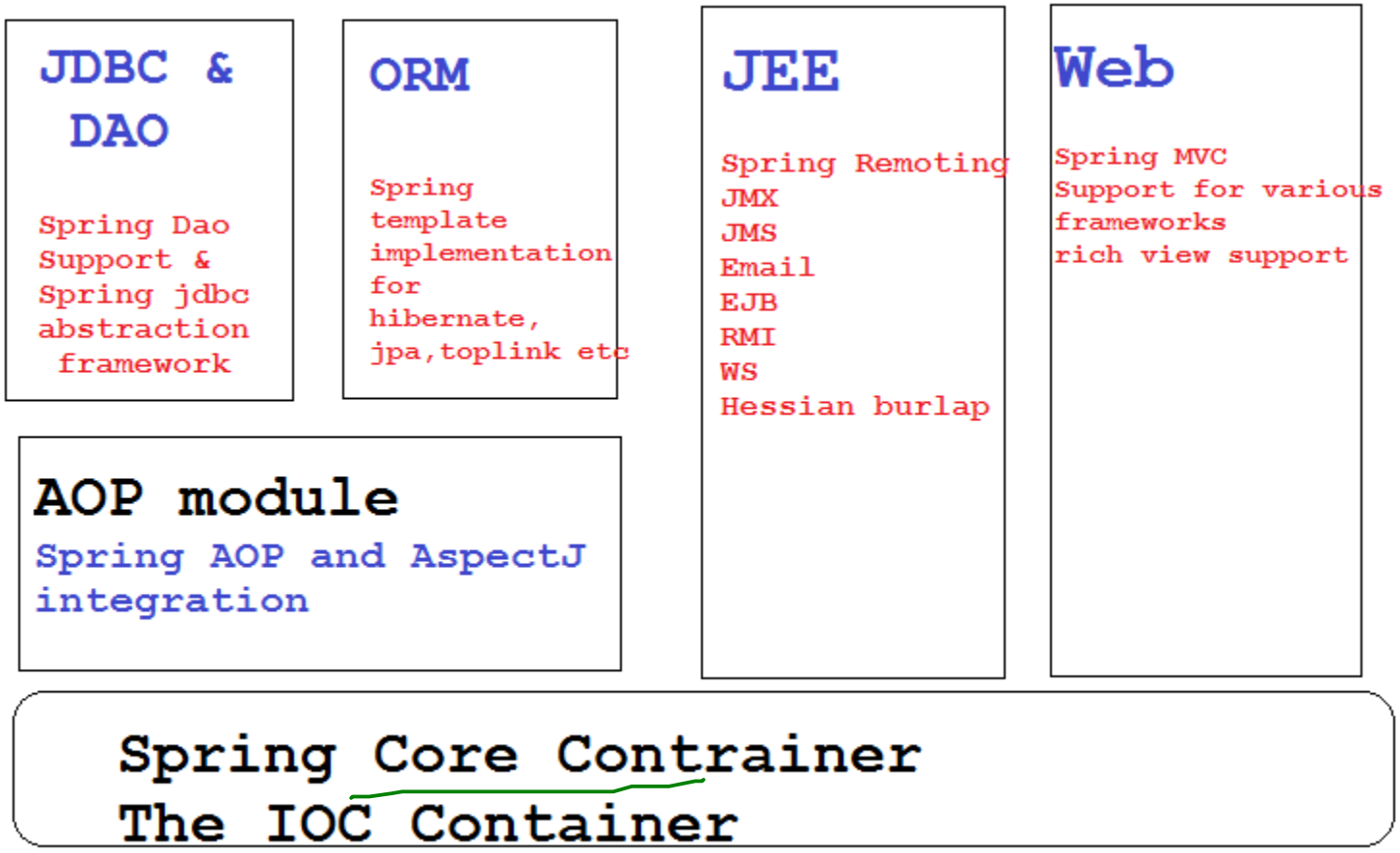
Why Spring framework?

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 and 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



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)

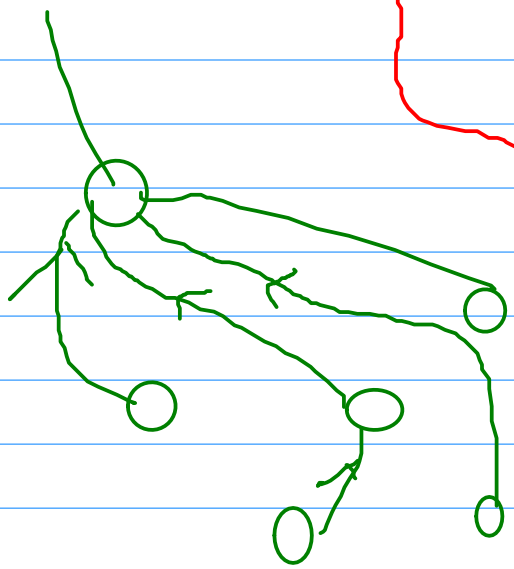
ctx

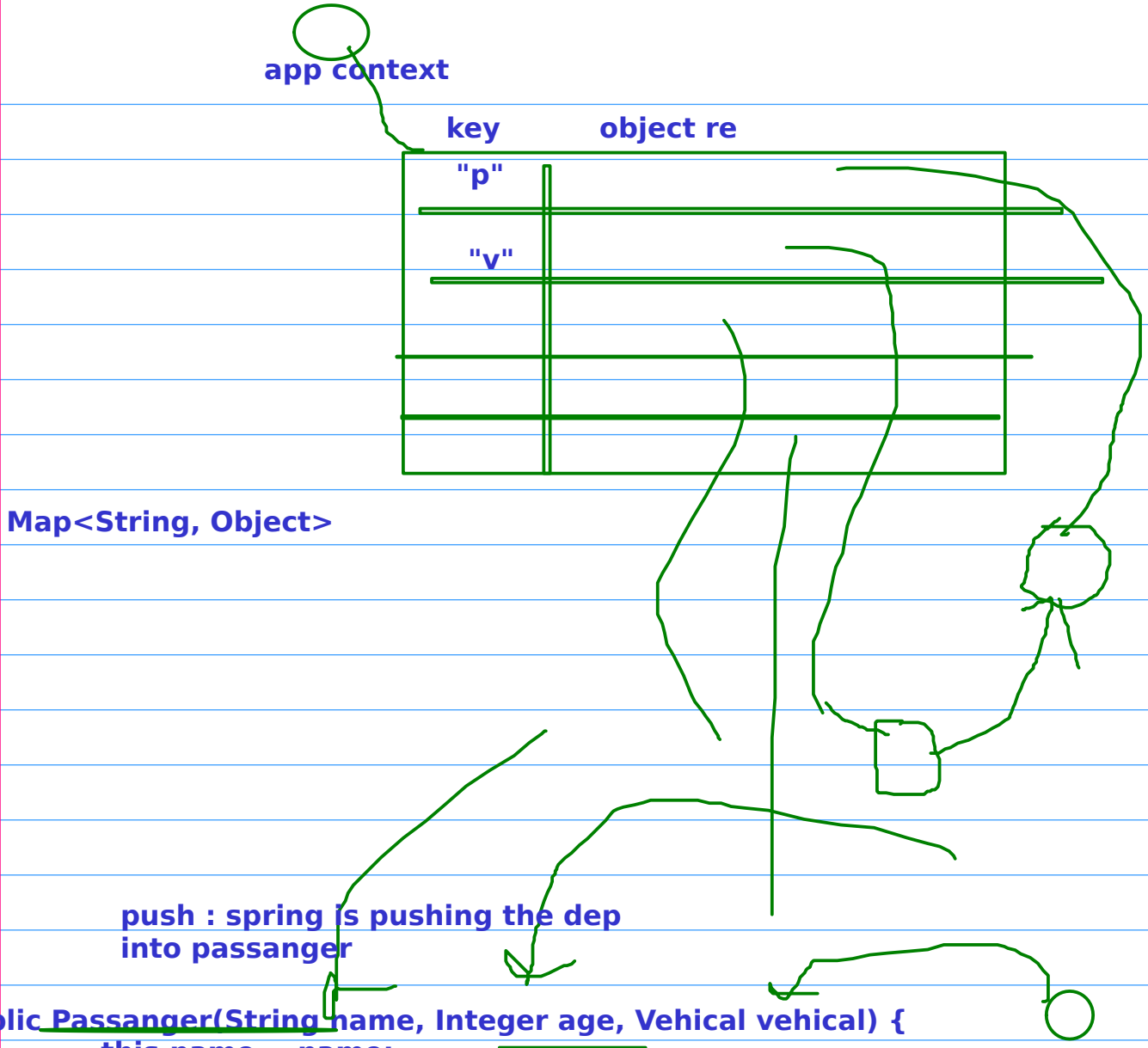
singleton :scope

passanger

"raj"

car





push : spring is pushing the dep
into passanger

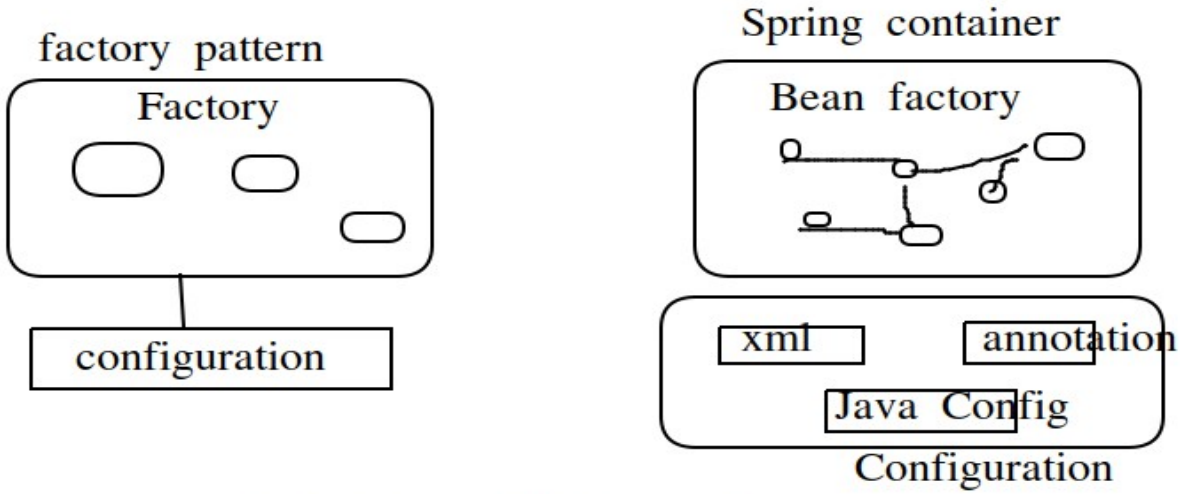
```
public Passanger(String name, Integer age, Vehical vehical) {
        this.name = name;
        this.age = age;
        this.vehical = vehical;
    }
```

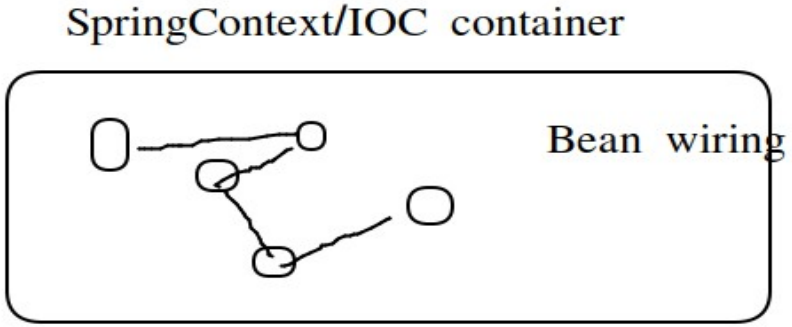
Need Of DI? An passenger need to travel

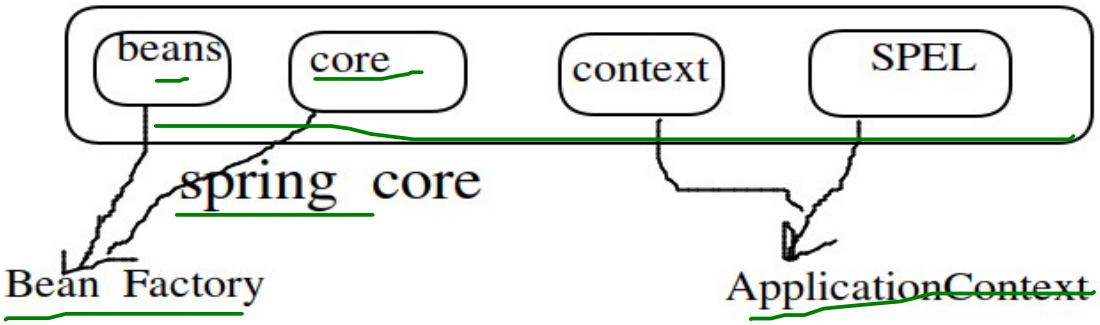
```java
public class Passanger {
    private String name;
    private Vehical vehical;

    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();
    }
}
```

```xml
<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>
```

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

passanger.travel();
```

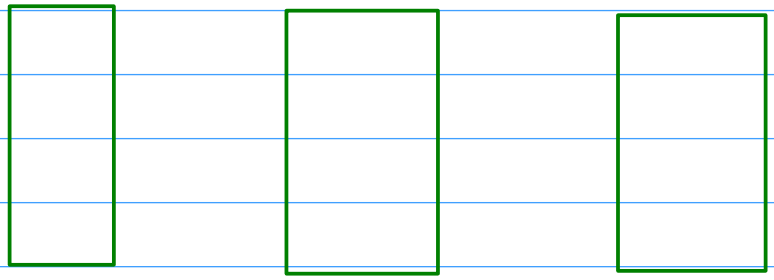

factory pattern
Factory
configuration
Spring container
Bean factory
xml
annotation
Java Config
Configuration


SpringContext/IOC container
Bean wiring


beans
core
context
SPEL
spring core
Bean Factory
ApplicationContext

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

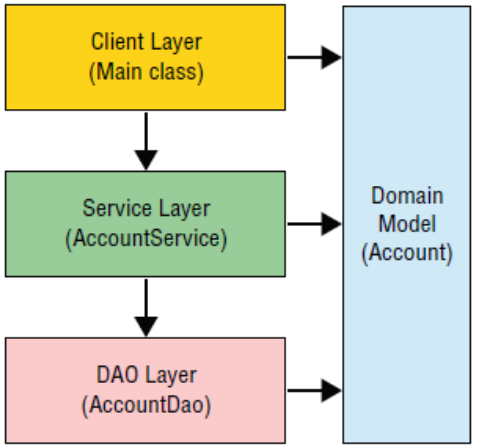
BeanFactory vs ApplicationContext



Bank application
dao
service :
controller

Bank Application

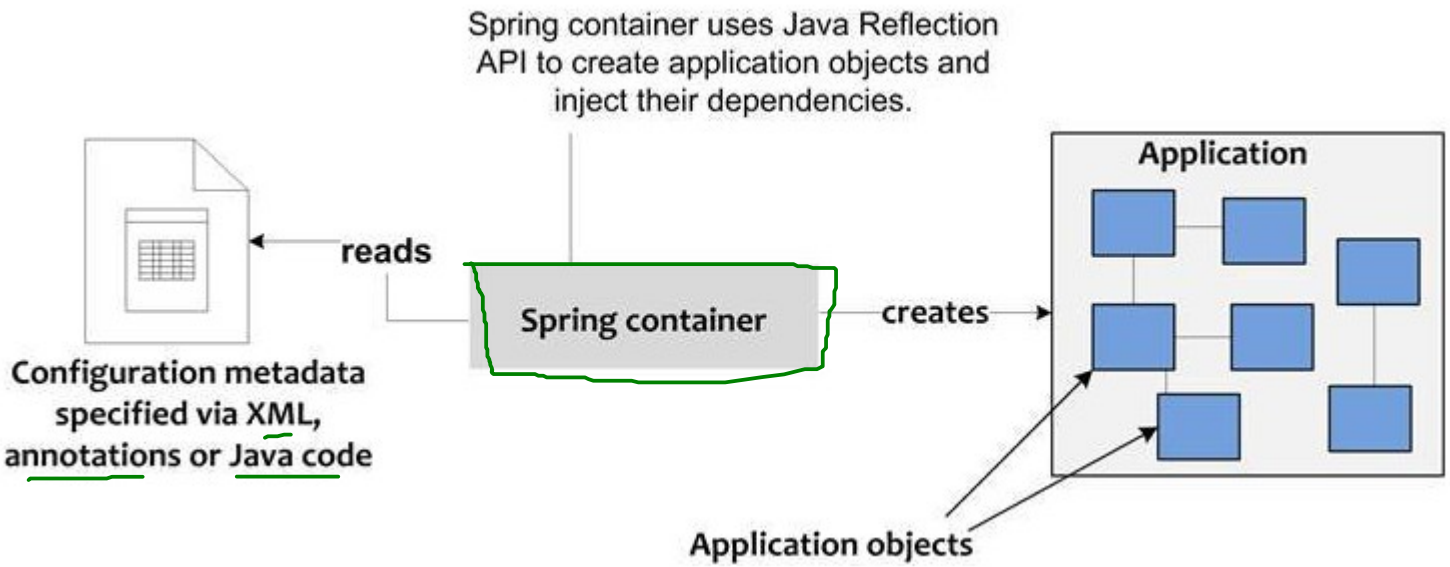
```
public class Account {
    private int id;
    private String name;
    private double balance;
```




```
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 amout);
    public void deposit(int id, double amount);
    public Account getAccount(int id);
}
```


Spring container uses Java Reflection API to create application objects and inject their dependencies.
reads
Spring container
creates
Application
Configuration metadata specified via XML, annotations or Java code
Application objects

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"/>
```

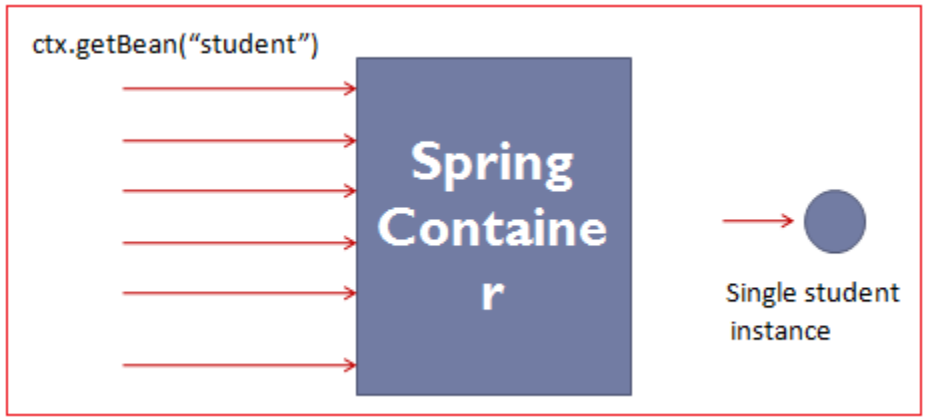
```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp" c:accountDao-ref="accountDao"/>
<bean id="accountDao" class="com.sample.bank.model.persistance.AccountDaoImp"/>
```

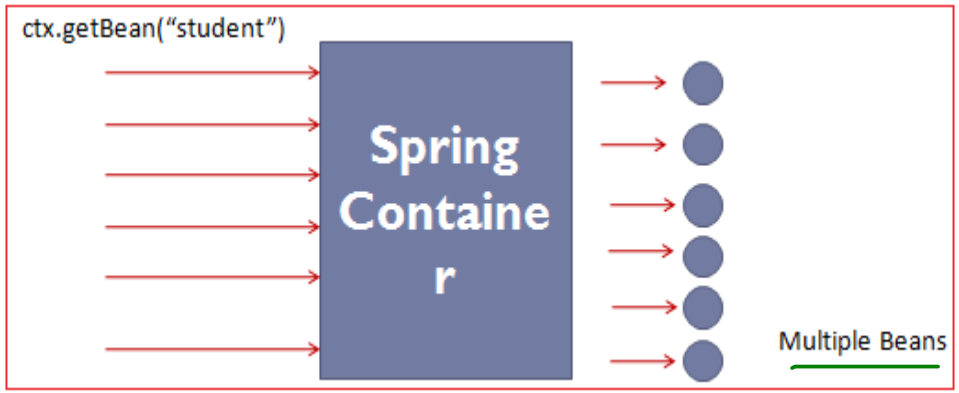
```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp" p:accountDao-ref="accountDao"/>
<bean id="accountDao" class="com.sample.bank.model.persistance.AccountDaoImp"/>
```

Bean Scopes

SCOPE NAME	SCOPE DEFINITION
`singleton`	Only one instance from a bean definition is created. It is the default scope for bean definitions.
`prototype`	Every access to the bean definition, either through other bean definitions or via the `getBean(..)` method, causes a new bean instance to be created. It is similar to the `new` operator in Java.
`request`	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 `ApplicationContexts`.
`session`	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 `ApplicationContexts`.
`globalSession`	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.persistance.AccountDaoImp"/>
```

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

Confusion?

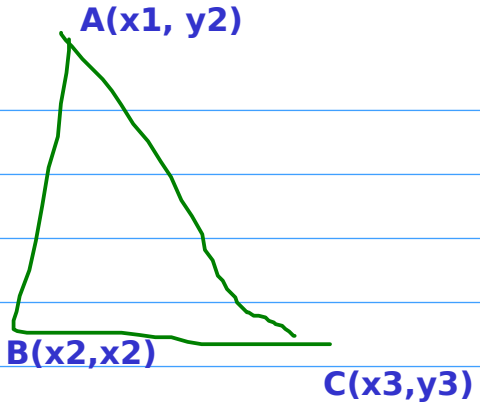
```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp" autowire="byType"/>
<bean id="accountDao1" class="com.sample.bank.model.persistance.AccountDaoImp"/>
<bean id="accountDao2" class="com.sample.bank.model.persistance.AccountDaoImp" autowire-candidate="false"/>
```

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"/> </propert
    <property name="friends" ref="mybestfriends" />
</bean>
```

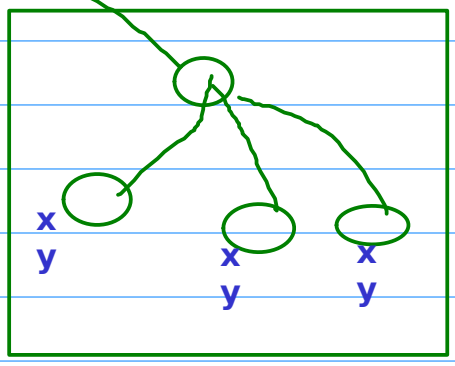
Passanger
Car bike

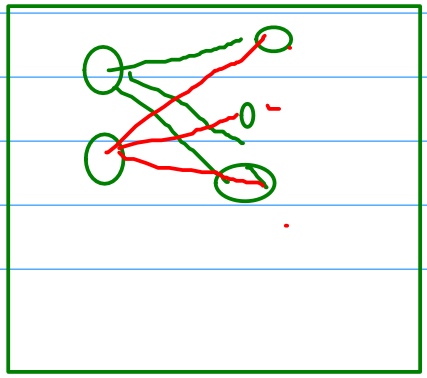


Point

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>
```

topic	done on	done again	note making	r1 discuss	remark
collection	15/2/21	15/2/21			5/10

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.persistance.Account" p:id="1" p:name="raja" p:balance="2000.50"/>
<bean id="account1" class="com.sample.bank.model.persistance.Account" p:id="2" p:name="ravi" p:balance="4000.50"/>

<bean id="accountCollection" class="com.sample.bank.model.persistance.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
- y in most
- ntation is *ontext*

```
String xmlFilePath = "META-INF/beans.xml";

ApplicationContext context = new ClassPathXmlApplicationContext( xmlFilePath );
```

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

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

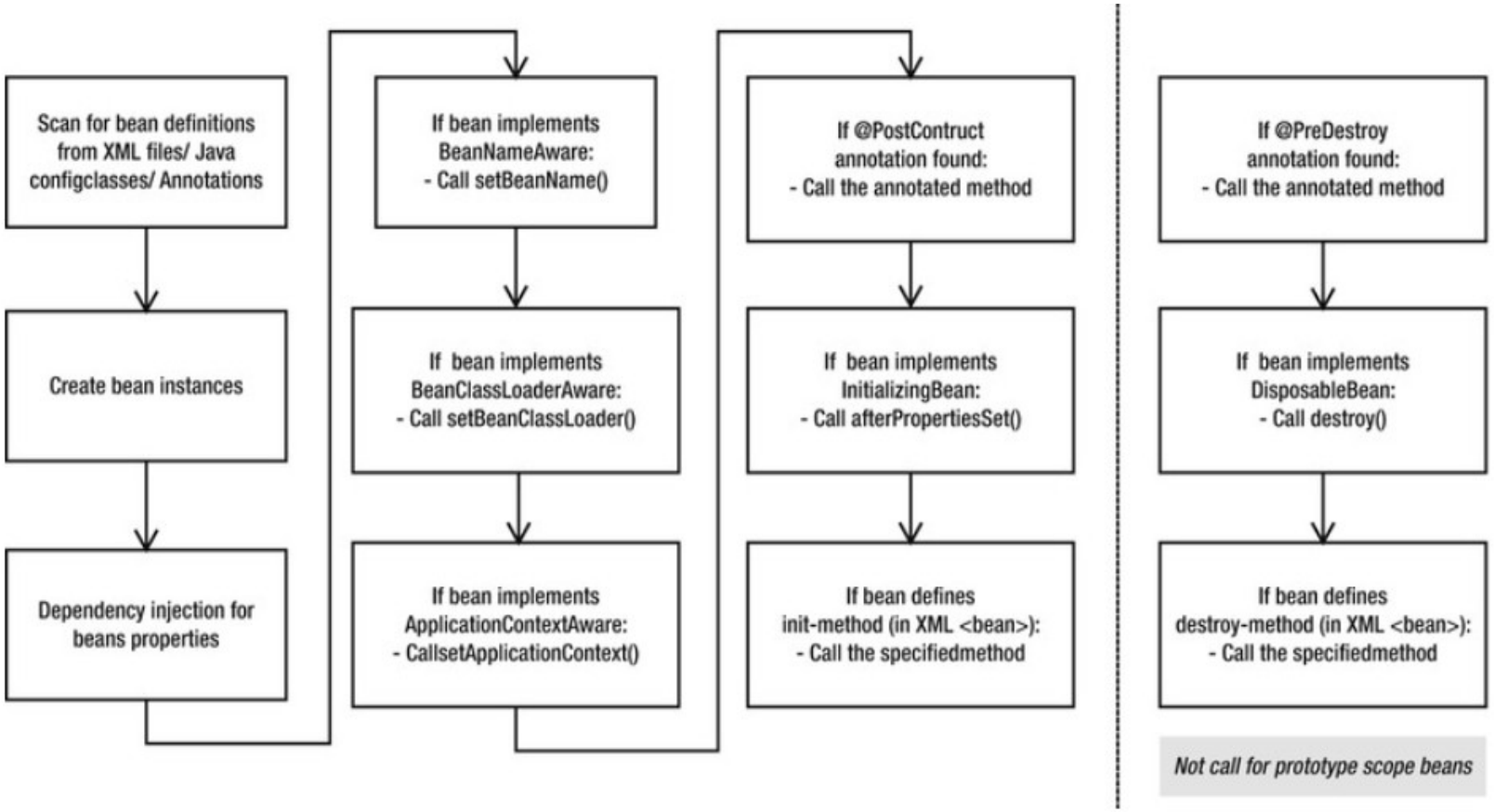


Figure 5-1. Spring beans life cycle

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

```
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"/>
```

> 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 wheras the others are defined on a per bean basis.

BeanFactoryPostProcessor

- BeanFactoryPostProcessor is invoked before bean factory is initilized (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 form properties files before initilization 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

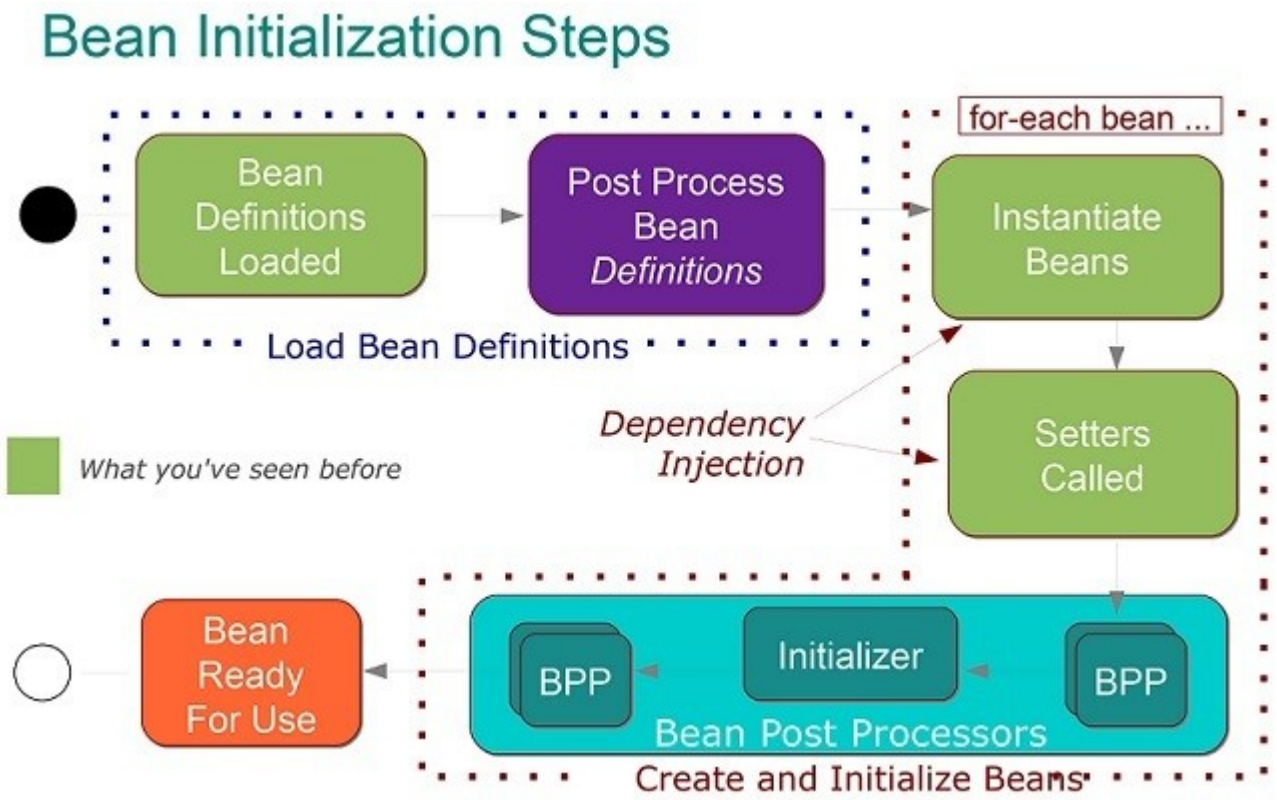
```
1 account.id=1
2 account.name=raja
3 account.balance=2000
```

PropertyPlaceHolderConfigurer:DB Configureation

```xml
<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 | beanx3.xml

```
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
for-each bean ...
Bean Definitions Loaded
Post Process Bean Definitions
Instantiate Beans
Load Bean Definitions
Dependency Injection
Setters Called
What you've seen before
Bean Ready For Use
BPP
Initializer
BPP
Bean Post Processors
Create and Initialize Beans

```java
@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!");
    }

}
```

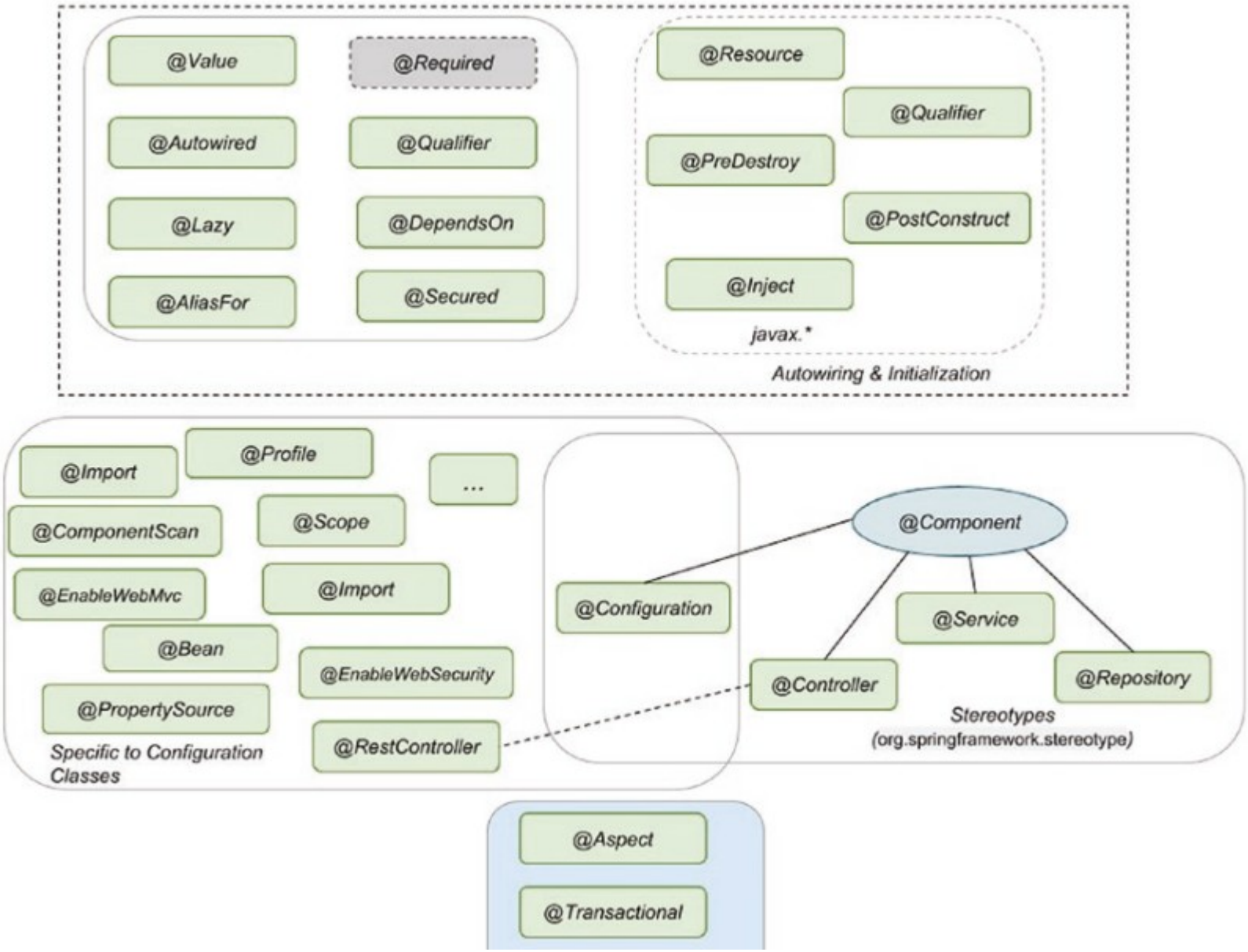
Agenda

- 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
- @Autowire -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 @Resouce
  - @Inject annotation in place of @Autowire


@Value
@Required
@Autowired
@Qualifier
@Lazy
@DependsOn
@AliasFor
@Secured
@Resource
@Qualifier
@PreDestroy
@PostConstruct
@Inject
javax.*
Autowiring & Initialization
@Import
@Profile
...
@ComponentScan
@Scope
@EnableWebMvc
@Import
@Bean
@EnableWebSecurity
@PropertySource
@RestController
Specific to Configuration Classes
@Component
@Configuration
@Service
@Controller
@Repository
Stereotypes
(org.springframework.stereotype)
@Aspect
@Transactional

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

```java
@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;
    }
}
```

```java
AnnotationConfigApplicationContext ctx=new AnnotationConfigApplicationContext(AppConfig.class);

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

@PropertySource & Using Environment to retrieve properties

```java
@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
```

```java
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());
```

wo profile

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