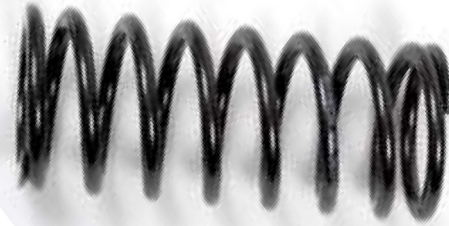
A photograph of a desk setup. On the left, a silver laptop is open, showing its keyboard and trackpad. To the right of the laptop is a spiral-bound notebook with a grid pattern, filled with handwritten notes in blue ink. A blue pen lies on the notebook. In the foreground, there are several sheets of paper, some with printed text and others with handwritten notes. The background is slightly blurred, showing more papers and a smartphone.

# Core Spring 5.x

Rajeev Gupta MTech CS  
Java Trainer & Consultant

# What is Spring?



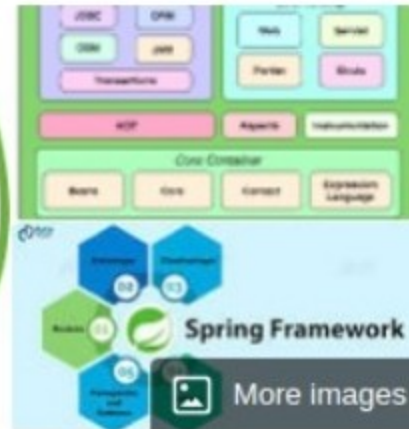
# Agenda

- ▶ 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

# 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](#)



# Spring framework version history

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



# Java EE vs Spring framework



- Java EE and Spring both capable middleware solutions
- Functionally equivalent for the most part, with very different philosophies and focus
- For very vanilla applications, code is almost identical



- Integrated platform with strong defaulting and minimal configuration
- Non-redundant minimalist APIs with specialized roles
- Focus on annotations and type-safety
- Vendor and implementation neutral open standard
- Focused on stable real world use cases, not buzzwords
- Decentralized ecosystem free to innovate via non-standard features, tools, integrations, plug-ins and quality of service

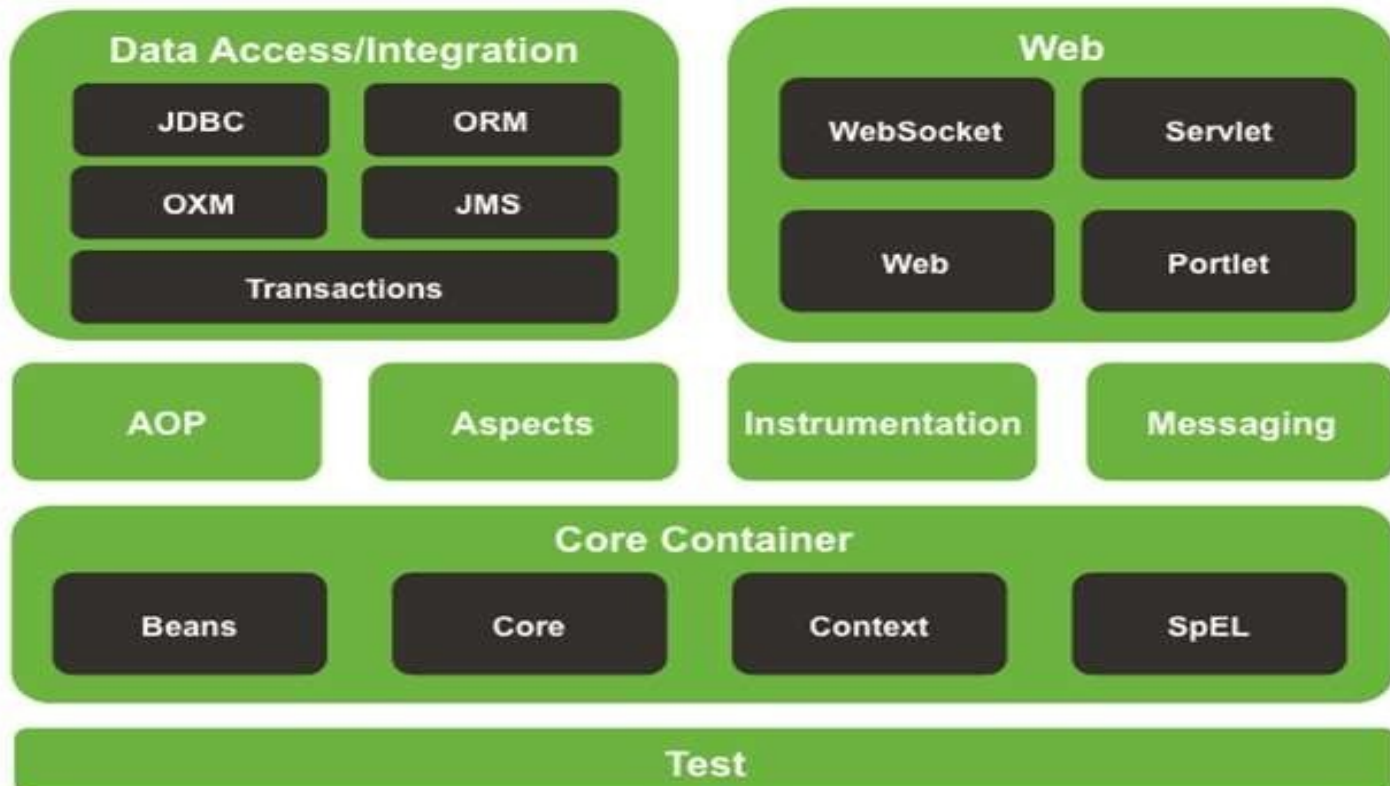
- Explicit configuration, few defaults – often in XML
- More flexibility but more complexity
- Non-standard open source from commercial vendor
- Centralized ecosystem integrating with standard as well as non-standard technologies that are often overlapping
- Portability across runtimes at the risk of implementation lock-in and deployment complexity
- Faster pace of change, especially on volatile edge cases or buzzwords

- No absolute reason to choose between the two
- Balanced competition is good for both developers and vendors
- Excellent integration possibilities as needed

# Spring framework modules



## Spring Framework Runtime





# Java EE vs Spring framework



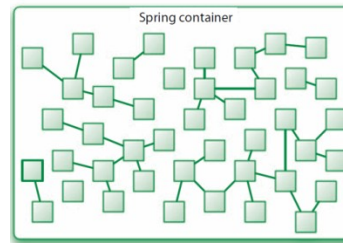
- Similar patterns for validation, WebSocket, JSON, XML, SOAP, remoting, scheduling, caching, etc
- Spring also integrates with EJB 3 but not CDI

# What is spring framework?

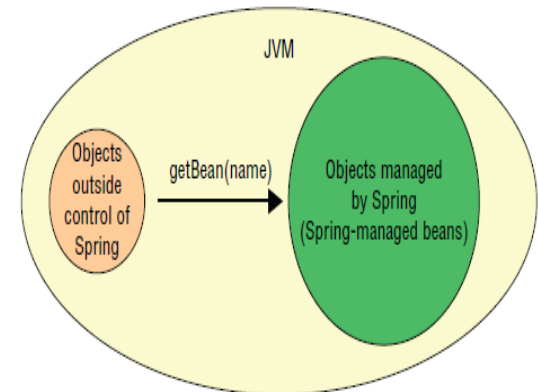
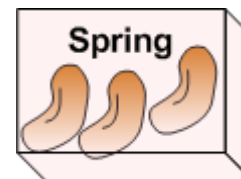
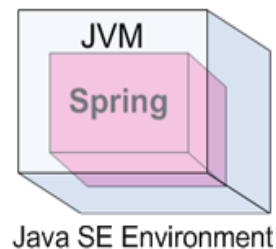
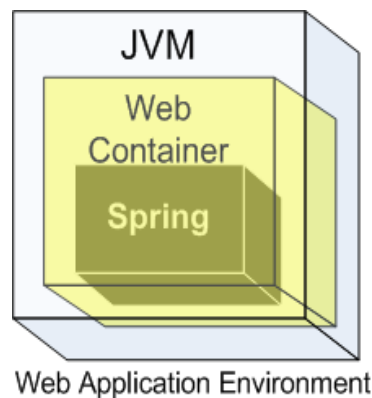
Spring framework is a container that manage life cycle of bean.

It Does 2 primary Jobs:

1. bean wiring
2. bean weaving



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

► Spring framework can be summarized in two ways:

1. Spring as an 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

3. 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 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();
    }
}
```

## Attempt 2:

```
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 3: Supplying vehicle from outside world

```
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();
    }
}
```



# 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();  
    }  
}
```

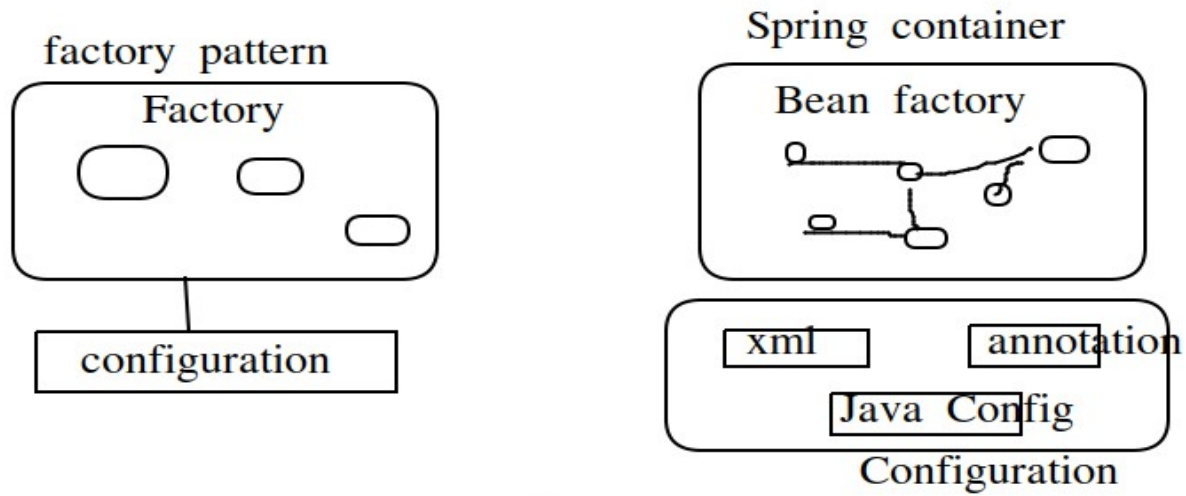
```
<beans xmlns="http://www.springframework.org/schema/beans"  
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"  
    xsi:schemaLocation="http://www.springframework.org/schema/beans  
        http://www.springframework.org/schema/beans/spring-beans.xsd">  
  
    <bean id="passanger" class="com.sample.ex1.Passanger">  
        <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();
```

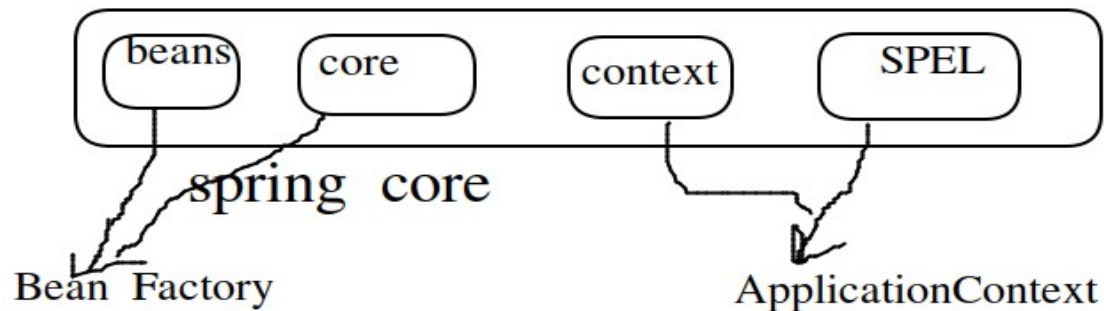
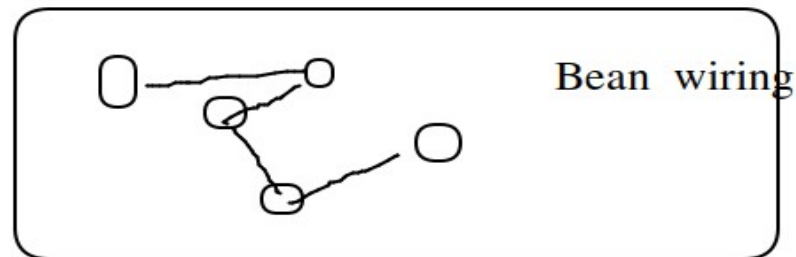
# Different ways of DI in Spring

## Different ways of DI in Spring





SpringContext/IOC container



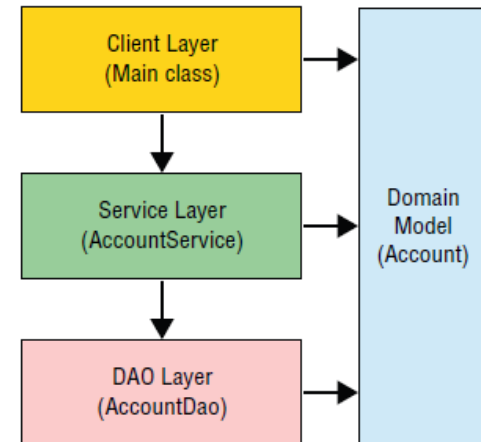
# Bank Application

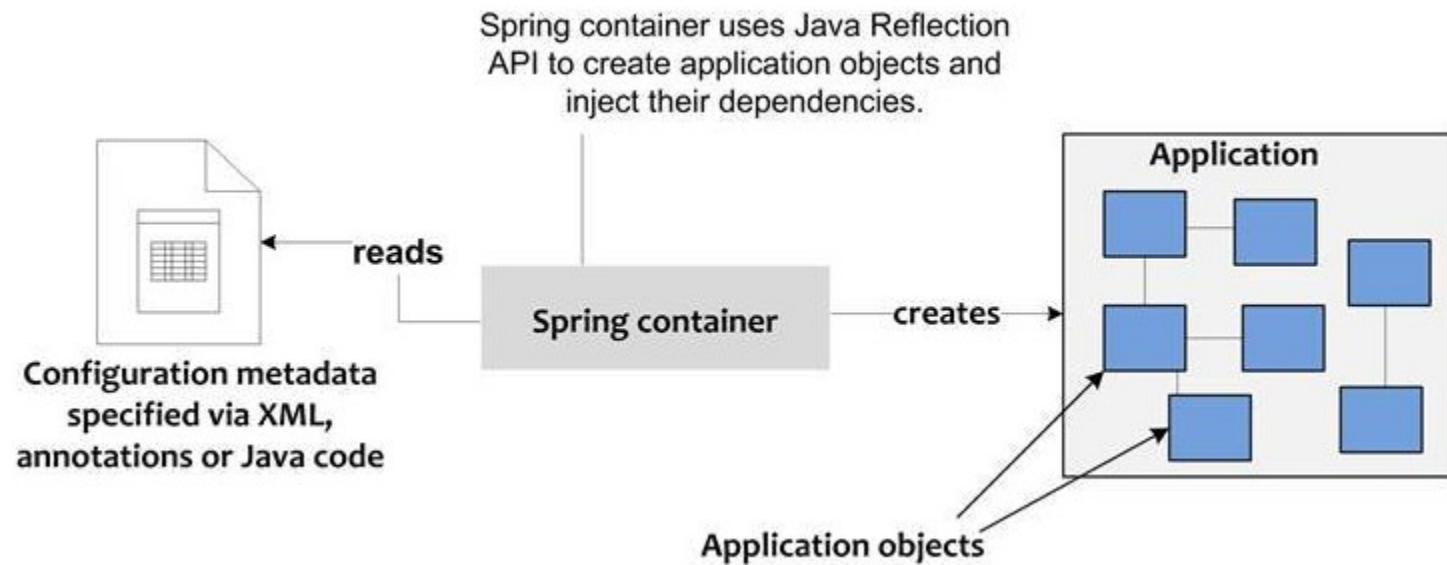
```
public class Account {  
    private int id;  
    private String name;  
    private double balance;  
}
```

```
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"/>
```

## ► c namespace

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

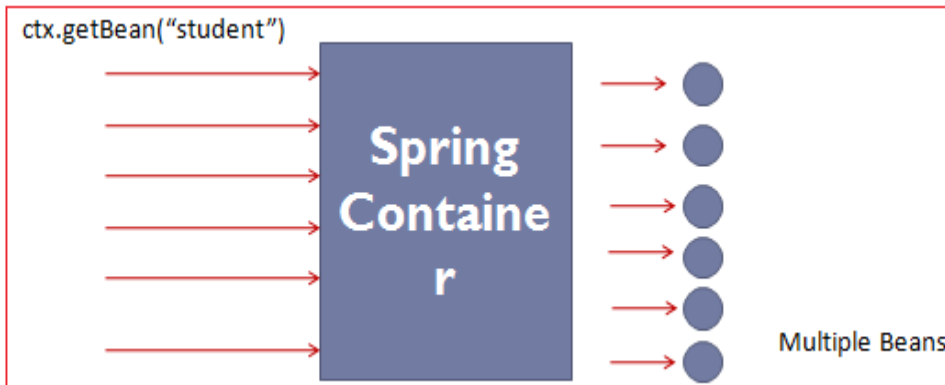
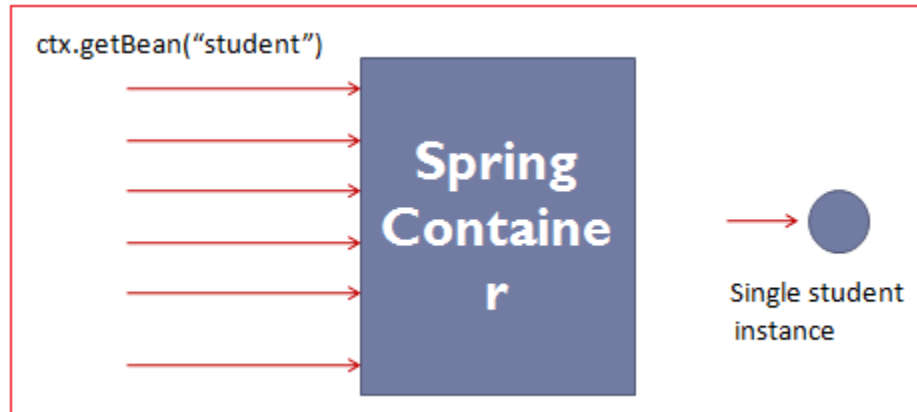
## ► p namespace

```
<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
<code>singleton</code>	Only one instance from a bean definition is created. It is the default scope for bean definitions.
<code>prototype</code>	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.
<code>request</code>	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> .
<code>session</code>	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> .
<code>globalSession</code>	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>
```

# 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>
```

# 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="account2" 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
- ▶ Preferred over the BeanFactory in most situations
- ▶ Most commonly used implementation is the *ClassPathXmlApplicationContext*

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

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

```
FileSystemXmlApplicationContext("c:/foo.xml");
```

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



# 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>
```

## ► Better solution:-

```
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>
```

# Spring bean life cycle

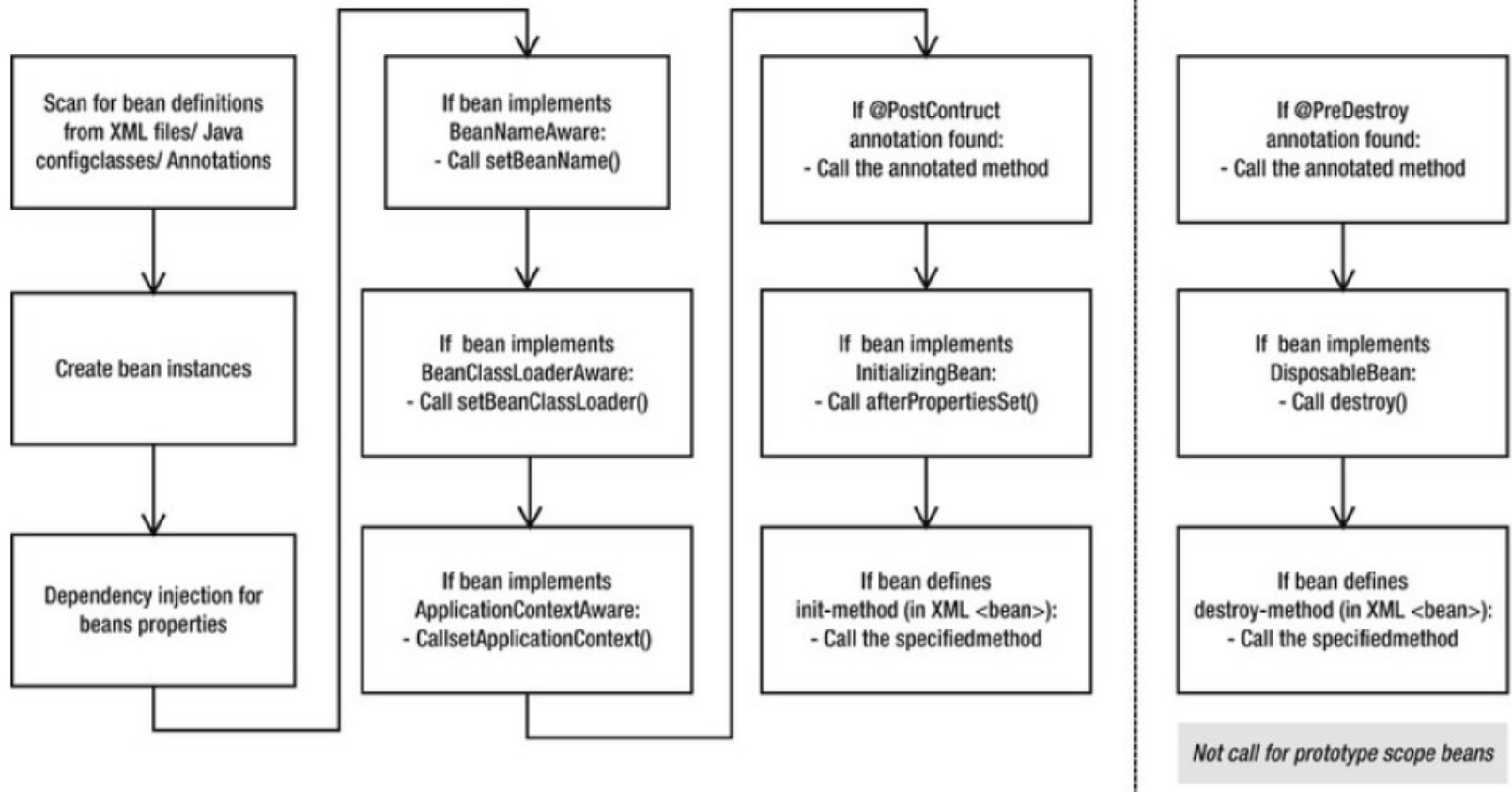


Figure 5-1. Spring beans life cycle

# 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 whereas the others are defined on a per bean basis.**

# 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

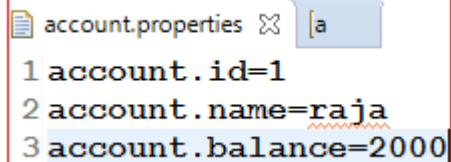
```
public class MyBeanFactoryPostProcessor implements BeanFactoryPostProcessor {  
  
    @Override  
    public void postProcessBeanFactory(ConfigurableListableBeanFactory arg0)  
        throws BeansException {  
        System.out.println("my bean factory post processor is called...");  
    }  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

```
<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;
```

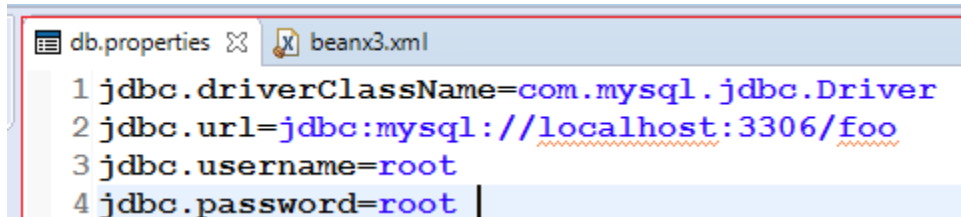
A screenshot of a text editor window titled 'account.properties'. The window contains three lines of text: '1 account.id=1', '2 account.name=raja', and '3 account.balance=2000'. The second line is highlighted with a blue selection bar.

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

# Property Place Holder Configurer:DB Configuration

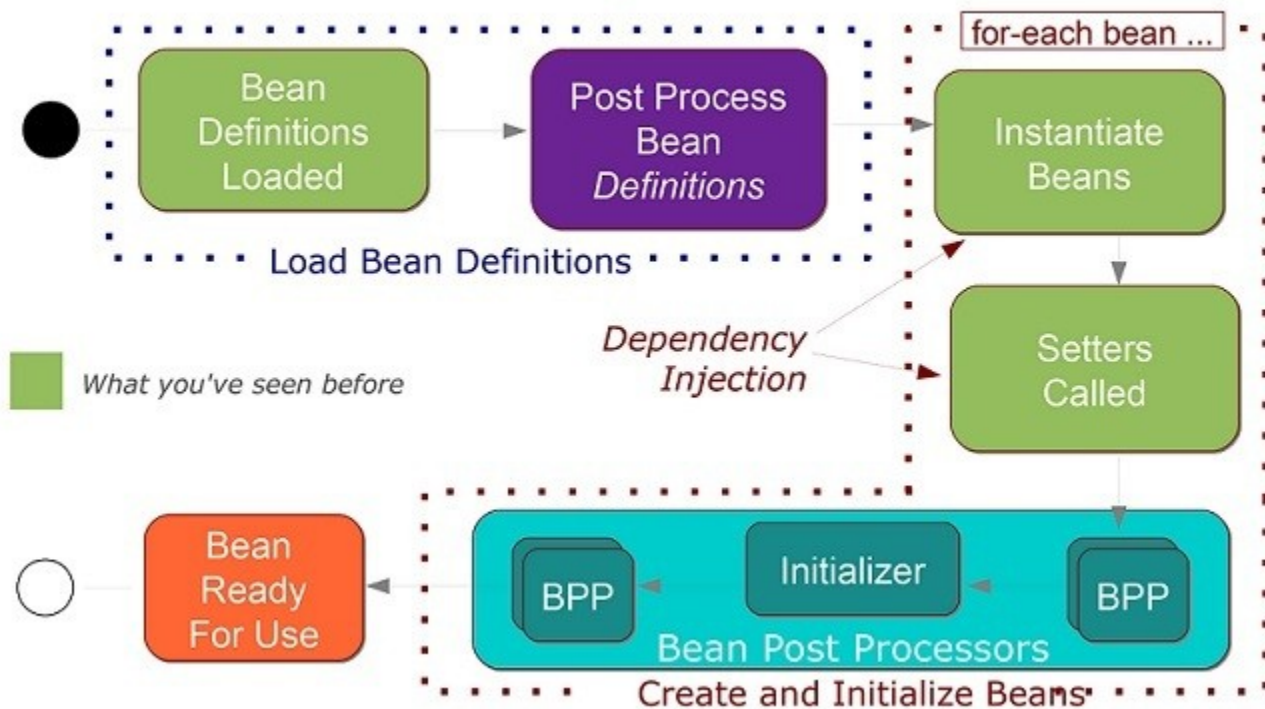
```
<bean id="accountService" class="com.sample.bank.model.service.AccountServiceImp">
    <property name="accountDao" ref="accountDao" />
</bean>
<bean id="accountDao" class="com.sample.bank.model.persistance.AccountDaoImpJdbc">
    <property name="dataSource" ref="dataSource" />
</bean>

<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





```
import org.springframework.beans.factory.annotation.*;  
@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!");  
    }
```

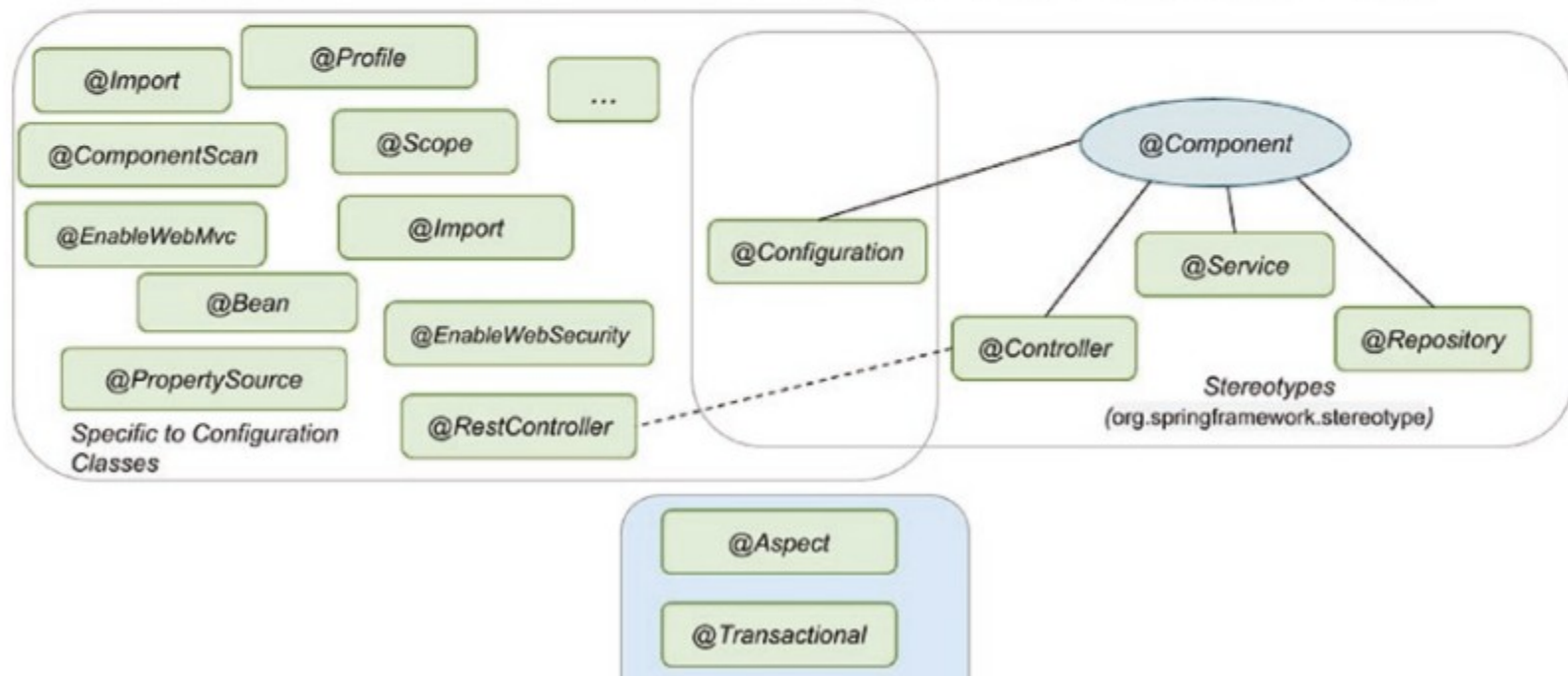
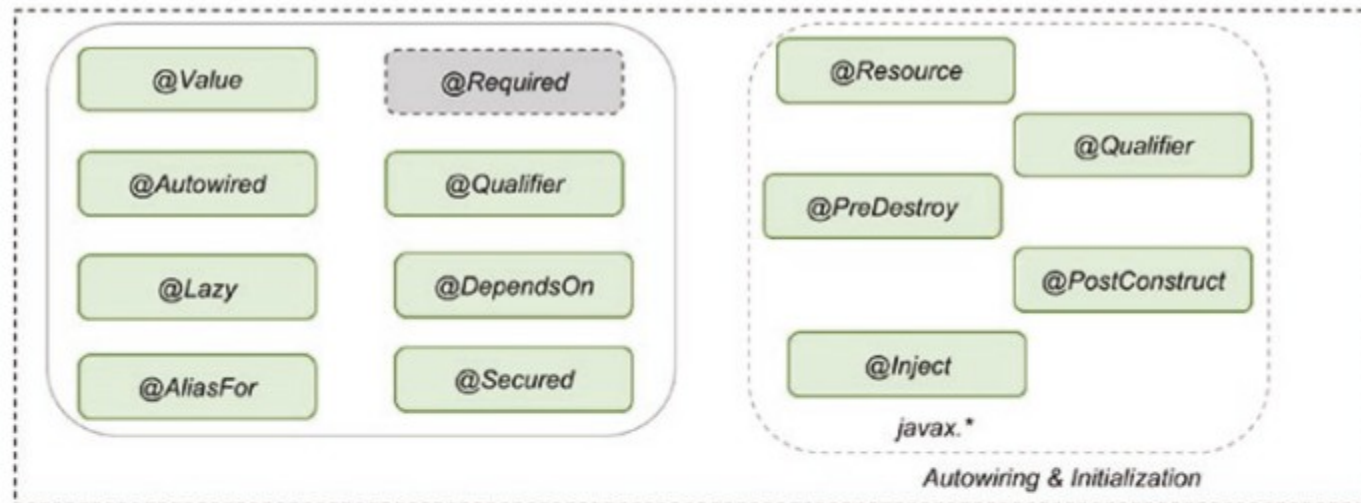
```
}
```

# 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



# 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);
```

# Spring JavaConfig Component Scanning

It will cause Spring to scan the package "com.exmaple" and will discover beans annotated with `@Component` or other stereotype annotations. This is alternative to using `<context:component-scan>`.

```
@Configuration
@ComponentScan("com.example")
public class MyAppConfig{
    //no bean definitions using @Bean methods here
}
```

We can use most of the annoation which we use with `@Bean` methods of `@Configuration` class:

- `@Scope`
- `@Lazy`
- `@DependsOn`
- etc

Declaring a bean component which is to be scanned, we can use any of the followings:

- `@Component`
- `@Controller`
- `@Repository`
- `@Service`

SCAN

Package com.example

```
@Component
public class MyBean1{
    .....
}
```

```
@Component
public class MyBean2{
    @Autowired
    private MyBean1 bean1;
    .....
}
```

Injecting other beans just like before

```
@Component
@Lazy
public class MyBean3{
    .....
}
```

# @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());
```

two profile

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



**Any questions?**

