Core Spring Framework Notes

rgupta.mtech@gmail.com

Agenda:

Understanding spring framework
Introduction to DI
Implementing DI application using xml, annotation and java code
setter/contructor injection, Scopes, c and p namespace
xml configuration
Spring bean life cycle, understanding Factory Post Processors
Spring EL basics,spring annotation in details, Idea about JSR 250, 330 annotations
Using Environment to retrieve properties
What are Profiles? ,Activating profiles

Lab: creating bank application for transfering fund from account a to b, applying DI to achive loose coupling

what is Spring Framework?

Spring is container that does two jobs "bean wiring" and "bean weaving"

Dependency injection

Aspect-oriented programming

Boiler-plate reduction.

Where it fits?

Presentation/UI Layer<===> Controller Layer <===>Business Layer<===>Data Access Layer<==> DB

Spring Modules

==> DI
==> AOP

==> DAO support: jdbc , ORM
==> Spring MVC
==>Spring JEE support
.....
.....

Bean factory vs ApplicationContext

XmlBeanFactory factory = new XmlBeanFactory (new ClassPathResource("foo.xml"));

ApplicationContext aC = new ClassPathXmlApplicationContext("foo.xml");

Maven hello world example

Passanger ----------> Vehical

Approach:
---
1. using xml
2. using annotations
3. using java configuration

bean.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"
		xsi:schemaLocation=" http://www.springframework.org/schema/beans
		http://www.springframework.org/schema/beans/spring-beans.xsd" default-init-
method="myInit"
		default-destroy-method="myDestroy">

	<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="p" class="com.Passanger">
			<property name="vehical" ref="vehical" />
```

```
	</bean>
	<bean id="vehical" class="com.Car" />
</beans>
```

using annotation

```
<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"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans.xsd
	http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-4.0.xsd">

	<context:annotation-config/>
	<context:component-scan base-package="com.spring.core.first2"/>
</beans>
```

using java

```
@Configuration
public class AnnotionConfiguration {
	@Bean
	public Passanger passangerService(){
		Passanger passanger=new Passanger();
		passanger.setVehical(vehicalService());
		return passanger;
	}
	@Bean
	public Vehical vehicalService(){
		Vehical vehical=new Car();
		return vehical;
	}
}
```

using bean

```
AnnotationConfigApplicationContext ctx = new AnnotationConfigApplicationContext(
				AnnotionConfiguration.class);
Passanger passanger=ctx.getBean("passangerService", Passanger.class);

passanger.travel();
```

Banking application
==========

=> need to transfer fund from accountA to accountB

ui layer<----> service layer <----> dao layer <---> db

```
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>();

	{
		accouts.put(1, new Account(1, "raja", 5000));
		accouts.put(2, new Account(2, "ravi", 1000));
	}

	@Override
	public void update(Account account) {
		accouts.put(account.getId(), account);
	}

	@Override
	public Account find(int id) {
		return accouts.get(id);
	}

}
```

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

```
}


public class AccountServiceImp implements AccountService {

	private AccountDao accountDao;


	public void setAccountDao(AccountDao accountDao) {
		this.accountDao = accountDao;
	}

	@Override
	public void transfer(int from, int to, int amout) {
		Account fromAccount=accountDao.find(from);
		Account toAccount=accountDao.find(to);

		fromAccount.setBalance(fromAccount.getBalance()-amout);
		toAccount.setBalance(toAccount.getBalance()+amout);

		accountDao.update(fromAccount);
		accountDao.update(toAccount);

	}

	@Override
	public void deposit(int id, double amount) {
		Account account=accountDao.find(id);
		account.setBalance(account.getBalance()+amount);
		accountDao.update(account);
	}

	@Override
	public Account getAccount(int id) {
		// TODO Auto-generated method stub
		return accountDao.find(id);
	}

}
```

3 ways to do configuraiton:
---

1. Using xml based configuration

-> setter injection

```
<bean id="accountService" class="com.service.AccountServiceImpl">
	<property name="accountDao" ref="accountDao"/>
```

```
</bean>

<bean id="accountDao" class="com.persistance.AccountDaoInMemoryImpl">

</bean>
```

Constructor Injection

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

2. Using annotation based configuration

```
@Repository
@Service
@Controller
```

3. Using Java based configuration

```
@Configuration
public class AccountConfiguration {

	@Bean
	public AccountService accountService() {
		AccountServiceImpl bean = new AccountServiceImpl();
		bean.setAccountDao(accountDao());
		return bean;
	}

	@Bean
	public AccountDao accountDao() {
		AccountDaoInMemoryImpl bean = new AccountDaoInMemoryImpl();
		//depedencies of accountDao bean will be injected here...
		return bean;
	}
}
```

Main

```
AnnotationConfigApplicationContext ctx = new
AnnotationConfigApplicationContext(AccountConfiguration.class);

AccountService service=ctx.getBean("accountService", AccountService.class);
```

Overriding bean definations

If we create a bean defination with a name that is already given to some other bean defination then second bean.

```
public class Foo {
	private String name;

	public String getName() {
		return name;
	}

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

}
```

```
@Configuration
public class FooConf1 {

	@Bean
	public Foo foo(){
		Foo foo1=new Foo();
		foo1.setName("aaa");
		return foo1;
	}
}

@Configuration
public class FooConf2 {

	@Bean
	public Foo foo(){
		Foo foo1=new Foo();
		foo1.setName("bbb");
		return foo1;
	}
}
```

Now test code

```
		AnnotationConfigApplicationContext context = new
AnnotationConfigApplicationContext(
					FooConf1.class, FooConf2.class);

		Foo foo=context.getBean("foo",Foo.class);
		System.out.println(foo.getName());
```

output: bbb

Annotation mapping in detail

@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
@Beans
@Configuration
@PostConstructs- Life cycle post
@PreDestroy- Life cycle pre

@Component
public class Foo{.....} is same as <bean id="foo" class="Foo"/>

EL Expression Language in XML to make configuration in XML more dynamic:
==========

==> EL can be used in spring configuration to make it more dynamic
==> Expression Syntex: #{}

==> Using EL apart form using operatiors you can
	also invoke methods and get the result

```
<bean id="t1" class="com.ex7.Traveler">
	<property name="travelerName" value="#{'raj'}"/>
	<property name="vehical" value="#{v}"/>

</bean>
```

```
<bean id="v" class="com.ex7.Car"/>

<bean id="t2" class="com.ex7.Traveler">
	<property name="travelerName" value="#{t1.getTravelerName()}"/>
	<property name="vehical" value="#{t1.myMethod()}"/>
</bean>
```

JSR 250 annotations

```
@Resource
@PostConstruct/ @PreDestroy
@Component
```

JSR 330 annotations

```
@Named annotation in place of @Resouce
@Inject annotation in place of @Autowire

@Named
public class CustomerDAO {
	public void save() {
		System.out.println("CustomerDAO save method...");
	}
}


@Named
public class CustomerService {
	@Inject
	CustomerDAO customerDAO;

	public void save() {
		System.out.println("CustomerService save method...");
		customerDAO.save();

	}

}
```

JSR-330 Limitations

There are some limitations on JSR-330 if compare to Spring :

@Inject has no required attribute to make sure the bean is injected successful.

No equivalent to Spring @Value, @Required or @Lazy.

xml mapping in detail:

Autowiring:aka shortcut
==========

==> Default mode: Auto-Wiring "no"

=> byName, byType, constructor

Account problem example:

Rather then :

```
<bean id="accountService" class="com.spring.core.second.AccountServiceImp">
        <property name="accountDao" ref="accountDao"/>
</bean>
```

We can write it like this:

```
<bean id="accountDao" class="com.spring.core.second.AccountDaoImp" />
```

Confusion?

```
<bean id="accountService" class="com.spring.core.second.AccountServiceImp" autowire="byName"/>

<bean id="accountDao" class="com.spring.core.second.AccountDaoImp" />
<bean id="accountDaoJdbc" class="com.spring.core.second.AccountDaoImpUsingJdbc"/>
```

Example 2:

```
<bean id="accountService" class="com.spring.core.second.AccountServiceImp" autowire="byType"/>

<bean id="accountDao" class="com.spring.core.second.AccountDaoImp" />
<bean id="accountDaoJdbc" class="com.spring.core.second.AccountDaoImpUsingJdbc"/>
```

Solution?

```
	<bean id="accountService" class="com.spring.core.second.AccountServiceImp"
autowire="byType"/>

	<bean id="accountDao" class="com.spring.core.second.AccountDaoImp" />
	<bean id="accountDaoJdbc" class="com.spring.core.second.AccountDaoImpUsingJdbc"
autowire-candidate="false"/>
```

Autowiring with a java based bean configuration

```
@Configuration
public class AccountConfiguration {

	@Bean(autowire=Autowire.BY_NAME)
	public AccountService accountService(){
		AccountServiceImp service=new AccountServiceImp();
		//service.setAccountDao(accountDao());
		return service;
	}

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

load multiple Spring bean configuration file

```
	ApplicationContext context = new ClassPathXmlApplicationContext
		(new String[] {"Spring-Common.xml", "Spring-Connection.xml","Spring-
ModuleA.xml"});
```

```
project-classpath/Spring-Common.xml
project-classpath/Spring-Connection.xml
project-classpath/Spring-ModuleA.xml
```

better solution?
==========

```
<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-2.5.xsd">

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

</beans>
```

ApplicationContext context = new ClassPathXmlApplicationContext(Spring-All-Module.xml);

Collection mapping Spring

```
public class Point{
	private int x,y;
}


pubic class Figure
{
	private List<Point> points;

	//....
	//....
}
```

Different Collection elements supported by Spring :

List: <list> - </list>

Set: <set> - </set>

Map: using-- <map> - </map>

Properties: using-- <props> - </props>

Now how to map it using list?

```
<bean id="triangle" class="Triangle">
	<property name="points">
		<list>
```

```
                        <ref bean="pointA"/>
                        <ref bean="pointB"/>
                        <ref bean="pointC"/>
                    </list>
                </property>

        </bean>
```

Now how to map it using set?

Using property?

```
  <property name="addressProp">
    <props>
      <prop key="one">INDIA</prop>
      <prop key="two">Pakistan</prop>
      <prop key="three">USA</prop>
      <prop key="four">USA</prop>
    </props>
   </property>

</bean>
```

Using map?

```
        <bean id="hank" class=".....">
        <propertyname="instruments">
                <map>
                        <entry key="GUITAR"value-ref="guitar"/>
                        <entrykey="CYMBAL"value-ref="cymbal"/>
                        <entrykey="HARMONICA"value-ref="harmonica"/>
                </map>
        </property>
        </bean>
```

Bean Scopes

==> Singleton (default)

==> prototype

==> request:new bean per servlet request

==> session: new bean per session

==> Global session:new bean per global HTTP session (portal)

```
<bean class="com.Point" id="zeroPoint" scope="singleton">
        <property name="x" value="0"></property>
        <property name="y" value="0"></property>
</bean>
```

```
@Service
@Scope("singleton")
public class Point {
    private int x;
    private int y;
```

Bean Lifecycle and Callbacks

Registering shut downhook:

Change from applicationContext==>AbstractApplicationContext

and then call context.registerShutdownHook();

way to notice life cycle of the bean
---

1. using implements InitializingBean , DisposableBean
2. Way annotation
3. u can use init-method="myinit" and destroy-method="mydestroy"

1. using implements InitializingBean , DisposableBean

```
public class Triangle implements InitializingBean , DisposableBean{
@Override
public void afterPropertiesSet() throws Exception {
  //To do some initialization works here
}

  @Override
public void destroy() throws Exception {
  //To do some Destruction works here
 }
   }
```

2. use amnotation

```
public class Triangle  {
	@PostConstruct
	public void myInit() {  }

	@PreDestroy
	public void myInit(){

	}
 }
```

BeanPostProcessor

=> Can be used to extends spring functionality

=> Lets say we need to put functionality just after init of bean for all configured bean in container?

Configuration of BeanPostProcessor

Step 1: Create an class

```
public class DisplayNameBeanPostProcessor implements BeanPostProcessor
{
	@Override
	public Object postProcessAfterInitialization(Object bean, String beanName) throws BeansException
	{
		System.out.println("In After bean Initialization method. Bean name is "+beanName);
		return bean;
	}

	@Override
	public Object postProcessBeforeInitialization(Object bean, String beanName) throws BeansException
	{
		System.out.println("In Before bean Initialization method. Bean name is "+beanName);
		return bean;
	}
}
```

Step 2: configure it in bean.xml

```
<bean class="com.DisplayNameBeanPostProcessor"></bean>
```

thats it!!!!

BeanFactoryPostProcessor

BeanFactoryPostProcessor:

=> is called before init of beanfactory (hence ApplicationContext)
=> then singleton bean

Example configuration

Step 1;

create BeanFactoryPostProcessorImp class

```
public class BeanFactoryPostProcessorImp implements BeanFactoryPostProcessor{

	@Override
	public void postProcessBeanFactory(ConfigurableListableBeanFactory fac)
		throws BeansException {
		System.out.println("BeanFactoryPostProcessorImp  is created.....");

	}

}
```

Step 2;

Configure BeanFactoryPostProcessorImp class

```
<bean class="BeanFactoryPostProcessorImp"></bean>
```

thats it!!!

Using properties files

What if we want to specify values of points from properties file rather then giving then in beans.xml

steps:

Step 1:

create an propeties file;

account.properties

account.id=1
account.name=raja
account.balance=2000

Step 2:

configure pointC as:

```
<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>
```

Step 3:

Write

```
<bean class="org.springframework.beans.factory.config.PropertyPlaceholderConfigurer" >
        <property name="location" value="classpath:account.properties"/>
</bean>
```

PropertyPlaceholderConfigurer enable spring to refer location specified in property tage
read shap.properties file and replace placeholder accordingly ....

Step 4:
run and observe output.

Example of BeanFactoryPostProcessor :

=> Spring provide PropertyPlaceholderConfigurer is used to read values
form properties files before initilization of beanfactory

=>if we want to execute some code after initilization of factory itself we can use BeanFactoryPostProcessor

=> Spring includes a number of pre-existing bean factory post-processors, such as

PropertyResourceConfigurer
PropertyPlaceHolderConfigurer

=> These bean factory post-processor, is used to externalize some property values from a BeanFactory definition,
into another separate file in Java Properties format.

This is useful to allow to developer to declare some key property as properties file.

As given below example show the database connection related information
in the following property file.

db.properties

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

NOw we can configure datasoucre as:

```
<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>
```

ApplicationContextAware

```
public class MyApplicationContextAware implements ApplicationContextAware{

        @Override
        public void setApplicationContext(ApplicationContext ctx)
                        throws BeansException {

                if(ctx instanceof AbstractApplicationContext)
                        ((AbstractApplicationContext) ctx).registerShutdownHook();
        }
```

}

Using Environment to retrieve properties

How to get database configuration info:?
---
db.properties

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

```
@Configuration
@ComponentScan(basePackages = { "com.demo.*" })
@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;
	}
}
```

testing:

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

		Connection con = (Connection) ctx.getBean("getConnection");

		if(con!=null)
			System.out.println("done");
```

What are Profiles? ,Activating profiles

=> @Profile allow developers to register beans by condition

Hello World:

```
	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");
			return foo;
```

```
		}
	}



	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());


	=> Eg: We require to use caching in our book application we want to support two
		profile "dev" and "production"

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



	code:
	--------

public class Book {
	private int id;
	private String isbn;
	private String title;
	private String author;




public interface BookDao {
		public Book getBookByIsbn(String isbn);
	}




@Repository(value = "bookDaoImp")
public class BookDaoImp implements BookDao {

	@Cacheable(value = "getBookById", key = "#isbn")
	@Override
	public Book getBookByIsbn(String isbn) {
		System.out.println("find book method is running....");
		// simulate slow method
```

```
		try {
			Thread.sleep(5000);
		} catch (InterruptedException ex) {
		}
		System.out.println("find book method is done....");
		return new Book(1, isbn, "java is in action", "rajiv");
	}

}
```

```
//-Dspring.profiles.active=dev
```

Now there are two configurations:

```
@Configuration
@Profile("dev")
public class CacheConfigDev {

	@Bean
	public CacheManager concurrentMapCacheManager() {
		return new ConcurrentMapCacheManager("getBookById");
	}

}
```

```
@Configuration
@Profile("live")
public class CacheConfigLive {

	@Bean
	public CacheManager cacheManager() {
		return new EhCacheCacheManager(ehCacheCacheManager().getObject());
	}

	@Bean
	public EhCacheManagerFactoryBean ehCacheCacheManager() {
		EhCacheManagerFactoryBean cmfb = new EhCacheManagerFactoryBean();
		cmfb.setConfigLocation(new ClassPathResource("ehcache.xml"));
		cmfb.setShared(true);
		return cmfb;
	}

}
```

```
	<ehcache xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
	xsi:noNamespaceSchemaLocation="ehcache.xsd" updateCheck="true"
	monitoring="autodetect" dynamicConfig="true">

	<!-- <diskStore path="java.io.tmpdir" /> -->
	<diskStore path="c:\\cache" />

	<cache name="getBookById"
		maxEntriesLocalHeap="10000"
		maxEntriesLocalDisk="1000"
		eternal="false"
		diskSpoolBufferSizeMB="20"
		timeToIdleSeconds="300" timeToLiveSeconds="600"
		memoryStoreEvictionPolicy="LFU"
		transactionalMode="off">
		<persistence strategy="localTempSwap" />
	</cache>

</ehcache>
```

configuration:

```
@Configuration
@EnableCaching
@ComponentScan({ "com.demo.*" })
public class AppConfig {
}
```

Main:

```
	System.setProperty(AbstractEnvironment.ACTIVE_PROFILES_PROPERTY_NAME,
"dev");
		ApplicationContext context = new
AnnotationConfigApplicationContext(AppConfig.class);

		BookDao dao=context.getBean("bookDaoImp", BookDao.class);
		dao.getBookByIsbn("1");
		dao.getBookByIsbn("1");

		((ConfigurableApplicationContext) context).close();
```

profile example:

```
<beans profile="dev,test">
		<bean id="dataSource"
			class="org.springframework.jdbc.datasource.SingleConnectionDataSource">
			<property name="driverClassName" value="org.h2.Driver" />
			<property name="url" value="jdbc:h2:mem:test" />
			<property name="username" value="sa" />
			<property name="password" value="" />
		</bean>
	</beans>
	<beans profile="prod">
		<bean id="dataSource"
			class="org.springframework.jdbc.datasource.DriverManagerDataSource">

			<property name="driverClassName" value="com.mysql.jdbc.Driver" />
			<property name="url" value="jdbc:mysql://localhost:3306/foo" />
			<property name="username" value="root" />
			<property name="password" value="root" />
		</bean>
```

-Dspring.profiles.active

spring.profiles.active context-param in web app

https://www.mkyong.com/spring/spring-profiles-example/
https://www.mkyong.com/spring/spring-propertysources-example/

```
	<bean id="dataSource"
class="org.springframework.jdbc.datasource.SingleConnectionDataSource">
		<property name="driverClassName" value="org.h2.Driver"/>
		<property name="url" value="jdbc:h2:mem:test"/>
		<property name="username" value="sa"/>
		<property name="password" value=""/>
	</bean>
```

```
	<bean class="org.apache.commons.dbcp.BasicDataSource"
		destroy-method="close" id="dataSource">
		<property name="driverClassName" value="com.mysql.jdbc.Driver" />
		<property name="url" value="mysql://localhost:3306/foo " />
```

```
        <property name="username" value="root" />
        <property name="password" value="root" />
    </bean>
```

Spring Expression language (SpEL) #{} or ${}

Need of SpEL?

=> Mostly beans declared for DI are static and statically defined.
=> there may be a requirement to perform dependency injection dynamically at runtime using SpEL

What we can achive using SpEL?

=> Refer to other beans by id attribute, Refer to the properties and invoke methods defined in other beans
=> Refer to the static constants and invoke static methods
=> Perform Mathematical operations on values
=> Perform Relational and Logical comparisons
=> Perform Regular Expression Matching

Refer to other beans by id attribute,
Refer to the properties and invoke methods defined in other beans
==========

Ex:Consider :We have a book collection and we want to initilize our book lib with it:

```
public class BookListCollection {
    private List<String> books;

    public List<String> getBooks() {
        return books;
    }

    public void setBooks(List<String> books) {
        this.books = books;
    }

    public String getSecondBook() {
        return getBooks().get(1);
    }
```

```
	}


	public class BookLib {
		private List<String> books;
		private String secondBook;
		//getter setter
	}
```

configuration:

```
	<bean id="bookListCollection"
class="com.training.model.persistance.BookListCollection">
		<property name="books">
			<list>
				<value>head first core java</value>
				<value>head first EJB</value>
				<value>head first Servlet JSP</value>
			</list>
		</property>
	</bean>

	<bean id="bookLib" class="com.training.model.persistance.BookLib">
		<property name="books" value="#{bookListCollection.books}"/>
		<property name="secondBook" value="#{bookListCollection.getSecondBook()}"/>
	</bean>
```

Refer to the static constants and invoke static methods
==========

The key syntax for accessing static methods are:

=> All Spring Expresions should be declared inside ${...}

=> Static class is referred by using T(...)

=> Members and methods of a bean are accessed using the dot (.) notation

Ex:

```
	public class RandomNumberGenerator {
		private Double randomNumber;
```

```
		public Double getRandomNumber() {
			return randomNumber;
		}

		public void setRandomNumber(Double randomNumber) {
			this.randomNumber = randomNumber;
		}

	}



	<bean id="randomNumberGenerator"
class="com.training.model.persistance.RandomNumberGenerator">
		<property name="randomNumber" value="#{T(java.lang.Math).random()}" />
	</bean>
```

Perform Mathematical operations on values

The key syntax for performing mathematical operations:

=> All Spring Expresions should be declared inside ${...}

=> Members and methods of a bean are accessed using the dot (.) notation

=> Standard mathematical operations such as +, -, *, /, % etc. are used on numerical properties

Example:

```
	public class Rectangle {
		private Integer length;
		private Integer breadth;
		//getter setter

	}


	public class PerimeterCalculator {
		private Integer perimeter;

		public Integer getPerimeter() {
			return perimeter;
		}

		public void setPerimeter(Integer perimeter) {
			this.perimeter = perimeter;
		}
```

```
	}



	<bean id="rectangle" class="com.training.model.persistance.Rectangle">
		<property name="length" value="5" />
		<property name="breadth" value="4" />

	</bean>

	<bean id="perimeterCalculator"
class="com.training.model.persistance.PerimeterCalculator">
		<property name="perimeter" value="#{2*(rectangle.length + rectangle.breadth)}" />
	</bean>
```

Perform Relational and Logical comparisons
==========

The key syntax for performing Relational and Logical comparisons:

=> All Spring Expresions should be declared inside ${...}

=> Members and methods of a bean are accessed using the dot (.) notation

=> Standard relational operations such as <, <=, ==, >=, > etc are used on numerical properties
Logical operations such as and, or, not should be used.

Example:

The result is 'passed' if the student has more than 40 marks in every subject
else the result is 'failed'.

If the result is 'passed' then the result message is 'Congratulations: You have passed!'
or else the message is 'Sorry: You have failed.'.

```
public class MarkSheet {

private String studentName;

private Integer marksInMathematics;
private Integer marksInPhysics;
private Integer marksInChemistry;


public class ExaminationResult {
```

```
private Boolean hasPassed;
private String resultMessage;
```

```
	<bean id="markSheet" class="com.training.model.persistance.MarkSheet">
		<property name="studentName" value="Alba" />
		<property name="marksInMathematics" value="90" />
		<property name="marksInPhysics" value="85" />
		<property name="marksInChemistry" value="80" />
	</bean>

	<bean id="passedMessage" class="java.lang.String">
		<constructor-arg value="Congratulations: You have passed!" />
	</bean>

	<bean id="failedMessage" class="java.lang.String">
		<constructor-arg value="Sorry: You have failed." />
	</bean>

	<bean id="examinationResult" class="com.training.model.persistance.ExaminationResult">
		<property name="hasPassed"
			value="#{markSheet.marksInMathematics >= 40 and
markSheet.marksInPhysics >= 40 and markSheet.marksInChemistry >= 40}" />
		<property name="resultMessage"
			value="#{markSheet.marksInMathematics >= 40 and
markSheet.marksInPhysics >= 40 and markSheet.marksInChemistry >= 40 ?
passedMessage:failedMessage}" />
	</bean>
```

Perform Regular Expression Matching
==========

The key syntax for performing Regular Expression Matching:

=> All Spring Expresions should be declared inside ${...}

=> Members and methods of a bean are accessed using the dot (.) notation

=> Matching with regular expression is done using the matches keyword (very similar to java regex)

Example:

The sample program is based on an email validator that validates whether the email address has a valid format.

=> We will create the Person class with members as name and email.

=> We will then create the EmailValidator class with member emailValid

```
public class Person {

	private String name;
	private String email;
}


public class EmailValidator {
	private Boolean emailValid;

	public Boolean getEmailValid() {
		return emailValid;
	}

	public void setEmailValid(Boolean emailValid) {
		this.emailValid = emailValid;
	}

}


<bean id="person" class="com.training.model.persistance.Person">
	<property name="name" value="Alba" />
	<property name="email" value="alba.bach@cmail.com"></property>
</bean>

<bean id="emailValidator" class="com.training.model.persistance.EmailValidator">
	<property name="emailValid"
		value="#{person.email matches '[\w]+.[\w]+@[\w]+.com'}" />
</bean>
```

SpEL with collection
==========

```
public class Student {
	private String name;
	private Integer marks;
```

```
public class StudentListAccessor {

	private Student thirdStudentInList;
	private List<Student> failedStudents;
	private List<String> studentNames;


	Discussion:
	----------
	accessing individual elements of List:
	---------------------------------------
	#{studentList[2]}	: Will access 3rd element of the collection

	performing filter operations on List
	------------------------------------
	#{studentList.?[marks lt 40]}
		filter out students in the List with marks less than 40 the using .?[] operator
	Populate the value of 'failedStudents'

	performing projections onto a List
	----------------------------------
	#{studentList.![name]}
		project the student list onto another list containing only the names of all students
using the .![] operator
	Populate the value of 'studentNames' property


	<bean id="student1" class="com.training.model.persistance.Student">
		<property name="name" value="Zorro" />
		<property name="marks" value="70" />
	</bean>

	<bean id="student2" class="com.training.model.persistance.Student">
		<property name="name" value="Bach" />
		<property name="marks" value="50" />
	</bean>

	<bean id="student3" class="com.training.model.persistance.Student">
		<property name="name" value="Cindy" />
		<property name="marks" value="30" />
	</bean>

	<bean id="student4" class="com.training.model.persistance.Student">
		<property name="name" value="Alba" />
		<property name="marks" value="80" />
	</bean>

	<bean id="student5" class="com.training.model.persistance.Student">
		<property name="name" value="Danny" />
		<property name="marks" value="20" />
```

```
	</bean>

	<bean id="studentList" class="java.util.ArrayList">
		<constructor-arg>
			<list>
				<ref bean="student1" />
				<ref bean="student2" />
				<ref bean="student3" />
				<ref bean="student4" />
				<ref bean="student5" />
			</list>
		</constructor-arg>
	</bean>

	<bean id="studentListAccessor"
class="com.training.model.persistance.StudentListAccessor">
		<property name="thirdStudentInList" value="#{studentList[2]}" />
		<property name="failedStudents" value="#{studentList.?[marks lt 40]}" />
		<property name="studentNames" value="#{studentList.![name]}" />
	</bean>
```

http://howtodoinjava.com/spring/spring-core/13-best-practices-for-writing-spring-configuration-files/
https://www.mkyong.com/spring3/spring-3-and-jsr-330-inject-and-named-example/
https://www.mkyong.com/spring/spring-profiles-example/