Setter Injection	Constructor Injection
Performs injection after creating Bean class object so it is bit delayed injection.	Performs injection while creating Bean class object so no delay in injection.
Support cyclic dependency injection.	Does not Support cyclic dependency injection.
<property> Tag is required.	<constructor-arg> Tag is required.
To perform setter injection on “n-properties” in all angle “n-setter” method is required.	To perform constructor injection on “n-properties” in all angle n! Constructor is required.
If all the properties are configured for setter injection then container creates bean class object using 0-param constructor.	If any property is configured for constructor injection container bean class object using parameterized constructor.

Spring AOP	Aspectj AOP
Performs runtime weaving.	Perform compile time weaving.
Generate proxy class at runtime.	Generate proxy class at compile time.it uses AspectJ compiler.
Supports only methods as JoinPoints.	Supports field, constructors, methods as JoinPoints.
Support both static and dynamic pointcut.	Support only static pointcuts.
Does not allow annotations.	Allows annotation.
Advice classes are non POJO classes.	Advice classes are POJO classes.

ses.load()	ses.get()
Only use the load() method if you are sure about object existence.	If you are not sure about object existence the go for get() method.
The load() method will throw an exception if unique id is not found in database.	get() method will return null if unique id is not found in database.
load() method just return proxy by default and it do not hit the database directly until the first proxy is invoked	get() will hit the database immediately.
Performs lazy loading of object.	Performs eager loading of objects.
Generates proxy object.	Do not Generates proxy object.
Use case: **1.** To show personalized offer based on user interest. **2.** To show notification related details in FB in which interested.	**Use case:** **1.** To offer of the day without user permission. **2.** To show job posting.

BeanFactory	ApplicationContext
BaenFactory is also known as **Basic IoC Container.**	ApplicationContext is also known as**Advanced IoC Conatiner.**
Beanfactory uses the **lazy Initialization**means it will create Singlton Bean based on the demand.	ApplicationContext uses e**ager initialization** means it will create all singleton Beans at the time of its own initialization.
BeanFactory creates and manages resources by **explicitely**.	ApplicationContext creates and manages resources by **its own.**
Internationalization is not supported by BeanFactory.	**Internationalization** is supported by ApplicationContext.
Annotation based dependency is not supported by BeanFactory.	ApplicationContext **supports Annotation**by using @PreDestroy @AutoWired.
One of the most popular implementation of BeanFactory interface is **XmlBeanFactory.**	One of the most popular implementation of ApplicationContext Interface is **ClassPathXmlApplicationContext.**
If you are using auto wiring and using BeanFactory then you need to register**AutoWiredBeanPostProcessor.**	If you are using ApplicationContext you can**configure in Xml.**

Update(Object obj)	saveOrUpdate(Object obj)	Merge(Object obj)
Performs object update/record update.	Perform save/insert or update operation on record.	Same as saveOrUpdate()
Update the record without checking the availability of record.	Insert or update the record by checking the availability of the record.	Same as saveOrUpdate()
Generates only update query.	Generate select query and insert/update query.	Same as saveOrUpdate()
Generate update query even through given object data matching with the record available in the table.	Does not generate update query.	Same as saveOrUpdate()
Dees not return any object representing the record that has to be updated.	Same as update()	Returns object of the record that is representing to be update or inserted.

Hibernate	JPA
Hibernate is the actual implementation of JPA.it can be say that Hibernate is superset of JPA with some extra specific functionality.	JPA (java persistence API) is specification for accessing, persisting and managing the data between java object and the relational database. It only a specification API, in other word we can say that JPA does not provide implementation ,it is set of rules and guidelines for developing the interface. Ex: EclipseLink(by oracale) and openJPA

ServletConfig	ServletContext
ServletConfig object represent single servlet.	ServletContext represents whole web Application running on it is common for all servlet.
getServletConfig() method issued to get the config object.	getServletContext() method is used to get the object of context object.
The param-value pairs for ServletConfig object are specified in the <init-param> within <servlet> tags in the web.xml file.	The param-value pairs for ServletContext object are specified in the <context-param> tags in the web.xml file.
The ServletConfig interface is implemented by the servlet container in order to pass configuration information to a servlet.	ServletContext define a set of methods that a Servlet uses to communicate with its servlet container.

GenericServlet	HttpServlet
The GenericServlet is an abstract class that is extended by HttpServlet to provide HTTP protocol-specif methods.	An abstract class that simplifies writing HTTP servlet.it extends the GenericServlet base class provides and framework for handling the HTTP protocol.
The GenericServlet does not include protocol-specific methods for handling request parameters, cookies, sessions and setting response header.	The HttpServlet subclass passes generic service method request to relevant doGet() or doPost() methods.
GenericServlet is not specific to any protocol.	HttpServlet only supports HTTP and HTTPS protocol.

How to use log4j

1. private static Logger logger=Logger.getLogger(ClassName); **//CREATE LOGGER**
2. SimpleLayout layout=new SimpleLayout(); **//CREATE LAYOUT**
3. ConsoleAppender appender=new ConsoleAppender(layout); **//CREATE APPENDER**
4. logger.addAppender(appender); **//ADD APPENDER**
5. logger.setLevel(Level.ALL/DEBUG/ERROR/INFO/FATAL/WARN); **//SPECIFY LOGGER LEVEL**
6. logger.info ("Here logger message will be.") **//logger message that will write in txt file**

How to configure Transaction in Spring(Declarative approach)

```
<!---DataSource Configuration--->
<bean id="txMgr" class="DataSourceTransactionManager">
        <property name="dataSource" ref="dataSource"/>
</bean>

<!---Advice Configuration--->
<tx:advice id="txAdvice" transaction-manager="txMgr">
        <tx:attributes>
                <tx:method name="transferMoney" propagation="REQUIRED" read-only="false"/>
        </tx:attributes>
</tx:advice>

<!---Aop Configuration--->
<aop:config>
        <aop:pointcut id="ptc1" expression="execution(* com.nt.service.bankservice.TransferMoney(..))"/>
        <aop:adviser advice-ref="txAdvice" pointcut-ref="ptc1"/>
</aop:config>
```

How to configure Transaction in Spring(Annotation approach)

```
@Transactional(read-only=”true/false”)
 Public int insert()
{
  ----------------------
}
```

To make it detect we have to configure it in xml file

```
<tx:annotation-driven transaction-manager=”txMgr”/>
```

How to configure hbm.xml file

User.hbm.xml

```
<hibernate-mapping>
   <class name=”User” table=”user”>
    <property name=”userId” column=”uid” type=”java.lang.Integer” length=”10” not-null=”true”>
    <property name=”userName” column=”uname” type=”java.lang.String” length=”100” not-null=”true”>
    <property name=”userPassword” column=”upass” type=”java.lang.String” length=”100” not-null=”true”>
   </class>
</hibernate-mapping>
```

How to write hibernate configuration file

Hibernate.cfg.xml (for oracle)

```
<hibernate-configuration>
  <session-factory>
                    <!-------Connection properties----->
     <property name=”hibernate.connection.driver_class”>oracle.jdbc.driver.OracleDriver</property>
     <property name=”hibernate.connection.url”>jdbc:oracle:thin:@localhost:1521:XE</property>
     <property name=”hibernate.connection.username”>System</property>
     <property name=”hibernate.connection.password”>manager</property>

                    <!-------Hibernate properties------>
     <property name=”hibernate.dialect”>org.hibernate.dialect.Oracle10gDialect</property>
     <property name=”hibernate.show_sql”>false/true</property>
     <property name=”hibernate.hbm2ddl.auto”>create/update/validate/create-drop</property>

                    <!-------Mapping properties------->
     <mapping  resource=”User.hbm.xml”/>

  </session-factory>
</hibernate-configuration>
```

How to Integrate Spring with Hibernate

```
<beans>
                                        <!------DataSource properties----->

        <bean id=”dataSource” class=”org.apache.commons.dbcp.BasicDataSource”>
                <property name=”driverClassName” value=”oracle.jdbc.driver.OracleDriver”>
                <property name=”url” value=”jdbc:oracle:thin:@localhost:1521:XE”>
                <property name=”username” value=”system”>
                <property name=”password” value=”manager”>
        </bean>
                                        <!------SessionFactory Configuration properties----->

        <bean id=”sessionFactory” class=”org.sf.orm.hibernate3.annotation.AnnotationSessionFactoryBean”>
                <property name=”dataSource” ref=”dataSource”/>
                 <property name=”annotatedClasses”>
                        <list>
                                <value>com.ew.hlo.EmployeeHLO</value>
                        </list>
                </property>
                <property name=”hibernateProperties”>
                     <props>
                        <porp key=”hibernate.dialect”>org.hibernate.dialect.oracle10gDialect</prop>
                        <prop key=”show_sql”>true</prop>
                      </props>
                  </property>
        </bean>
                                        <!------HibernateTemplate properties----->

        <bean id=”hibernateTemplate” class=” org.sf.orm.hibernate3. hibernateTemplate”>
                <property name=”sessionFactory” ref=”sessionFactory”/>
        </bean>
                                        <!------TransactionManager properties----->

        <bean id=”TransactionManager” class=” org.sf.orm.hibernate3.HibernateTransactionManager”>
                <property name=”sessionFactory” ref=”sessionFactory”/>
        </bean>
                                        <!------DAO class properties----->

        <bean id=”employeeDao” class=”com.ew.dao.EmployeeDao”>
                <constructor-arg ref=” hibernateTemplate”/>
        </bean>
</beans>
```

Spring Core Configuration

Setter-Based Dependency Injection

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

```
<bean id="bean1" class="com.sms.name">
    <property name="name">
            <value>Ravi</value>
    </property>
</bean>
```

```
<bean id="bean1" class="com.sms.name">
    <property name="name" value="Ravi"/>
</bean>
```

Constructor-Based Dependency Injection

```
public class NameBean{
private String name;
public NameBean(String name){
this.name=name;
}
```

```
<bean id="bean1" class="com.sms.name">
    <constructor-arg>
            <value>Ravi</value>
    </ constructor-arg >
</bean>
```

```
<bean id="bean1" class="com.sms.name">
    <constructor-arg value="Ravi"/>
</bean>
```

How to Inject following

1. Primitive
2. String
3. Reference
4. Collection(List,Set,Map)
5. Properties
6. Null

Primitive

By using value attribute ***or*** <value> tag

String

By using value attribute ***or*** <value> tag

Reference

Using <property> tag and ref attribute

<property ref="beanName"/>

List

```
public class Student{
private String List<String> names;
public void setName(List<String> names){
this.name=name;
}
//to String
```

```
<bean id=”st” class=”com.sms.Student”>
    <property name=”name”>
        <list>
            <value> Deepak </value>
            <value> Neha </value>
            <value> Uttam </value>
            <value> Shipra </value>
        </list>
    </property>
</bean>
```

Set

```
public class Student{
private String Set<String> phones;
public void setName(Set<String> phones){
this.phones=phones;
}
//to String
```

```
<bean id=”st” class=”com.sms.Student”>
    <property name=”name”>
        <set>
            <value>8896224451</value>
            <value> 9919862486</value>
        </set>
    </property>
</bean>
```

Map

```
public class Collage{
private Map<String,String> facultySubject;
private Map<?,?> facultyPhones;
public void setfacultySubject(Map<String,String> facultySubject) {
        this.facultySubject= facultySubject;
}
public void setfacultyPhones(Map<?,?> facultyPhones) {
        this.facultyPhones= facultyPhones;
}
}
//to String
```

```
<bean id="clg" class="com.sms.Collage">
    <property name="facultySubject">
        <map>
            1. <entry key="Raja" value="java"/>
            2. <entry>
                   <key><value>Ravi</value></key>
                   <value>.net</value>
               </entry>
            3. <entry key="Ramesh">
                   <value>PHP</value>
               </entry>
        </map>
    </property>
</bean>
```

Property

```
public class Student{

private Properties stdId;

public void setStdId(Property stdId){

this.stdId=stdId;

}

//to String
```

```
<bean id="st" class="com.sms.Student">
    <property name="stdId">
        <props>
            <prop key="Anil">4451</prop>
        </props>
    </property>
</bean>
```

Null

```
public class User{
private int id;
private String name;
private Date dob;
public User(int id,String name,Date dob){
        this.id=id;
        this.name=name;
        this.dob=dob;
}
//to String
```

```
<bean id="bean1" class="com.sms.name">
        <constructor-arg value="111"/>
        <constructor-arg value="Ravi"/>
        <constructor-arg >
            <value>
                </null>
            </value>
        </constructor-arg >
</bean>
```

How to inject inner bean

1. **Using setter injection**
2. **Using constructor injection**

Constructor

```
public  class  PaymentGateway {
   private  Order order;
   public  PaymentGateway(Order ord){
      this.order = ord;
   }
}
```

```
public  class  Order {
   private  String item;
   public  String getItem() {
      return  item;
   }
   public  void  setItem(String item) {
      this.item = item;
   }
}
```

```
<bean id="paymentGwBean"  class="com.beans.PaymentGateway">
      <constructor-arg>
        <bean class="com.beans.Order">
           <property name="item"  value="Java2Novice"  />
           <property name="price"  value="RS 22.50"  />
           <property name="address"  value="Bangalore"  />
         </bean>
      </constructor-arg>
   </bean>
```

Setter

```
public class Bicycle{
private chain chain;
public void setChain(chain chain){
this.chain=chain;
}
//to string
```

```
public class chain{
private String type;
public String setType(String type){
this.type=type;
}
public void getType(){
          return type;
}}
```

```
<bean id="biCycle"  class="com.beans.Bycycle">
      <property name=”chain”>
        <bean class="com.beans.Chain">
           <property name="type"  value="ralco"  />
         </bean>
      </property>
   </bean>
```

How to Enable Annotation Based Configuration

```
<context:annotation-config/>
```

How to Configure *idref*

```
<bean id=”car” class=”com.beans.Car”>
    <property name=”beanId”>
        <idref bean=”engin”/>
    </property>
</bean>
<bean id=”engin” class=”com.beans.Engine” scope=”prototype”/>
```

How to configure autowiring

```
<bean id=”car” class=”com.beans.car” autowire=”byname/byType/Constructor/Autodetect”>
```

How to configure nested BeanFactories

```
<bean id=”emiCalculator” class=”com.nbf.beans.EMICalculator”>
```

```
<bean id=”CLA” class=” com.nbf.beans.CustomerLoanApprover”>
        <property name=”emiCalculator”>
                <ref parent=” emiCalculator”>
        </property>
</bean>
```

How to configure Bean LifeCycle

1. **Declarative approach**

```
<bean id=”cla” class=” com.beans.CustomerLoanApprover” init-method=”Init” destroy-method=”Destroy”>
```

2. **Programmatic approach**
    1. Implements the interfaces **InitializingBean, DisposableBean**
    2. Override **destroy()** method and **afterPropertySet()**
3. **Annotation based approach**

```
@Named(“motor”)
public class Motor{
        private Engin engine;
                @PostConstruct
                public void init(){}
                @PreDestroy
                public void destroy(){}
}
```

Using P & C Namespace

1. **P Namespace**

In spring 3.0 **p-name space** is introduced to perform setter injection cfg without using <property> tag .It allows to place attributes directly in <bean> having the following syntax.

p:<property name>="value"

p:<property name>-ref="<bean_id>"

p-name space url:*http://www.springframework.org/schema/p*

2. **C Namespace**

Similarly In spring 3.1.1 **c-name space** is introduced to perform constructor injection cfg without using <constructor-arg> tag.

c:<property name>="value"

c:<property name>-ref="<bean_id>"

c-name space url:*http://www.springframework.org/schema/c*

Example

```
<bean id="dt" class="java.util.Date" p:year="115" p:month="10" p:date="20">
<bean id="dept"  class="com.beans.dept" c:deptno="8001" c:name="Accounts" c:dop-ref="dt"/>
```

Dependency Check

```
<bean id="dt" class="java.util.Date" dependency-check="simple/object/all">
        <property  name="name" value="neha"/>
        <property  name="sex" value="female"/>
</bean>
```

In spring 2.5 **" dependency-check** attribute has been removed and now we are using annotation **@Required.**

Aware Injection/Interface Injection/Contextual Dependency Lookup

If underlaying container injects object/value to target class only when target class implements/extends certain interface/class then it is called Aware Injection/Interface Injection/Contextual Dependency Lookup.

- ✓ If target class wants to use dependent object in all the methods then we go for dependency injection.
- ✓ If target class wants to use dependent object in specific methods then we go for dependency lookup.

xxxAware Interfaces are-

1. **BeanNameAware** -to inject current beanid ***(override void setBeanName(String s))***
2. **BeanFactoryAware** -to inject BeanFactory object ***(override void setBeanFactory(BF bf))***
3. **ApplicationContextAware** -to inject ApplicationContext object. ***(override void setApplicationContext(AC ac))***

Static Factory method

```
<bean id="cal" class="java.util.Calender" factory-method="getInstance"/>
<bean id="alarm" class="com.sf.beans.Alarm">
        <property name="time" ref="cal">
</bean>
```

Instance Factory method

```
<bean id=”mapEngineLoacator” class=”com.beans. MapEngineLoacator”/>
<bean id=”indiaMapEngin” factory-bean=” mapEngineLoacator” factory-method=”getMapEngine”/>
```

Factory Bean

In order to work with Factory Bean you need to implement the FactoryBean interface and need to override the of methods ,these are… Factory Bean is use to create the object of given bean.

1. **getObject()**-To create the object of class you want to make as bean
2. **getObjectType()-** to return the object of class type.
3. **isSingleton()**-indicate that object created through the factory is singleton or prototype.

Method Replacement/Method Injection

```
<bean id=”bank” class=”com.beans.Bank”>
        <replaced-method name=”calcIntrAmt” replacer=”mr”>
                <arg-type>float</arg-type>
                <arg-type>float</arg-type>
        </replaced-method>
</bean>
```

Lookup Method injection

How to add RequestHandler

```
RequestHandler rh=getRequestHandler();
Rh.setData(Data);
Rh.handle();
```

Cfg in xml

```
<bean id=”web” class=”com.beans.webContainer”>
        <lookup-method name=”getRequestHandler” bean=”requestHandler”/>
</bean>
<bean id=”requestHandler” class=”com.beans.RequestHandler”>
```

Bean Post Processor

In spring ,we use Bean lifecycle to perform initialization on bean.but using init-method, destroy-method or InitializationBean, disposableBean we do initialization or destruction process for a specific bean which implement these if we have common initialization process which has to applied across all the beans in the configuration,Bean lifecycle can not handle this. we need to use BeanPostProcessor.

1. **postProcessBeforeInitialization(Onject bean,String bName)** (called after the core container has created.)
2. **postProcessAfterInitialization(Onject bean,String bName)** (called after the core container has performed injection.)

Bean Factory Post Processor

It is used to perform post processing logic on BeanFactory after it has been created and before it instantiated the beans then we should go for BeeanFactoryPostProcessor.we can take example of database connectivity as processing logic.to implement this logic we can use PropertyPlaceHolderConfigure is a post processor which will reads the message from the properties file and replace the ${} place holder's of a bean configuration after the BeanFactory has loaded it.

Spring AOP Configuration

Aspect:

The logic that can be applied through various component of application known as **aspect.** this is also known as cross-cuting logic.in other word what type of secondary logic you want to add in various component of application known as **aspect.**

JoinPoint:

The possible point in the class where aspect can be used know as JoinPoint.Spring supports only methods as JoinPoint. i.e. where we can apply the aspect is JoinPoint.

Advice:

When we want to apply the JoinPoint on method is known as advice.as example before method execution or after method execution and etc.

1. **BeforeAdvice**: here advice logic execute before target class method execution.
2. **AfterAdvice**: here advice logic execute after target class method execution but before the control back.
3. **ThrowsAdvice**: here advice logic execute when target class method raise the exception.
4. **AroundAdvice**: here advice logic execute before and after target class method execution.

PointCut:

PointCut is collection of JoinPoint on which we want to aspect.i.e. aspects are configured.

Target Class:

The class on which we want to advice the aspect is known as target class. This target class contains the primary logics.

Weaving:

Mix-up of primary logic and secondary logic is known as weaving. In other word the process of advicing aspect on the target class to build the proxy class is known as weaving.

Proxy Class:

Proxy class is dynamically generated in memory class.

Configuration Approach

1. Spring AOP API()Programmatic Approach)
2. Spring AspectJ(Declarative Approach)
3. AspectJ(Annotation Approach)

Spring AOP API (Programmatic Approach)

AroundAdvice

In order to work with any advice we need to create a **Target class .** then we need **Aspect** representing the advice it is using. our aspect will implement the Interface **MethodInterceptor** and override the method **invoke(MethodInvocation mi)** . mi will have entire method details.

BeforeAdvice

In order to workour aspect will implement the Interface **MethodBeforeAdvice** and override the method **public void before(Method method, Object[] args, Object target).**

AfterThrows

In order to work our aspect will implement the Interface **AfterReturningAdvice** and override the method **public void afterReturning(Object ret, Method method, Object[] args, Object target)**

Example:

Aspect

```
package com.aop.beans;
import java.lang.reflect.Method;
import org.aopalliance.intercept.MethodInterceptor;
import org.aopalliance.intercept.MethodInvocation;
import org.springframework.aop.AfterReturningAdvice;
import org.springframework.aop.MethodBeforeAdvice;
import org.springframework.aop.ThrowsAdvice;
public class LoggingAspect implements MethodInterceptor, MethodBeforeAdvice, AfterReturningAdvice,
ThrowsAdvice {
	// for AROUND ADVICE
	public Object invoke(MethodInvocation mi) throws Throwable {
		Object[] args = mi.getArguments();
		String method = mi.getMethod().getName();
		System.out.println("entering into method:" + method + "(" + args[0] + "," + args[1] + ")");
		Object ret = mi.proceed();
		System.out.println("exiting into method:" + method + "(" + args[0] + "," + args[1] + ")");
		return ret;
	}
	// For BeforeAdvice
	public void before(Method method, Object[] args, Object target) throws Throwable {
		System.out.println("entering into method:" + method);
	}
	// For AfterReturningAdvice
	public void afterReturning(Object ret, Method method, Object[] args, Object target) throws Throwable
{
		if ((Integer) args[1] == 0)
			throw new ArithmeticException("Year can not be zero");
	}
}
```

TargetClass:Maths.java

```
package com.aop.beans;
public class Maths {
	// for AroundAdvice
	public int add(int a, int b) {
		return a + b;
	}
	// for BeforeAdvice
	public int divide(int a, int b) {
		return a / b;
	}
	// for AfterReturningAdvice
	public float calculate(long p, int y, float r) {
		return (p * y * r) / 100;
	}
}
```

Tester Class

```
package com.aop.test;
import org.springframework.aop.framework.ProxyFactory;
import com.aop.beans.LoggingAspect;
import com.aop.beans.Maths;
public class AATest {
	public static void main(String[] args) {
		// TODO Auto-generated method stub
		ProxyFactory pf = new ProxyFactory();
		pf.addAdvice(new LoggingAspect());
		pf.setTarget(new Maths());

		Maths math = (Maths) pf.getProxy();
		// to test AroundAdvice
		System.out.println("Sum: " + math.add(4, 40));

		// to test BeforeAdvice
		System.out.println("Divide: " + math.divide(10, 5));

		// to test AfterReturningAdvice
		System.out.println("Interest: " + math.calculate(10L, 1, 12.0f));
	}
}
```

ThrowsAdvice

AspectClass

```
package com.aop.beans;
import org.springframework.aop.ThrowsAdvice;
public class LoggingAspect implements ThrowsAdvice {
	public void afterThrowing(IllegalArgumentException ie) {
			System.out.println("thrown :"+ie.getMessage());
	}
}
```

Target Class

```
package com.aop.beans;
public class Maths {
	public int willThrow(int i){
		if(i<=0) throw new IllegalArgumentException("invalid param");
		return i+10;
	}
}
```

TesterClass

```
package com.aop.test;
import org.springframework.aop.framework.ProxyFactory;
import com.aop.beans.LoggingAspect;
import com.aop.beans.Maths;
public class AATest {
	public static void main(String[] args) {
		ProxyFactory pf=new ProxyFactory();
		pf.addAdvice(new LoggingAspect());
		pf.setTarget(new Maths());
		Maths math=(Maths)pf.getProxy();
		//to test ThrowsAdvice
		System.out.println("Sum: "+math.willThrow(0));
	}
}
```

Spring AOP API (Programmatic Approach)

Aspect

```
package com.aop.beans;
import org.aspectj.lang.JoinPoint;
import org.aspectj.lang.ProceedingJoinPoint;
import org.springframework.aop.ThrowsAdvice;
public class LoggingAspect implements ThrowsAdvice {
	public Object addLog(ProceedingJoinPoint pjp) throws Throwable {
		Object[] args = pjp.getArgs();
		String method = pjp.getSignature().getName();
		System.out.println("entering into method:" + method + "(" + args[0] + "," + args[1] + ")");
		Object ret = pjp.proceed();
		System.out.println("exiting into method:" + method + "(" + args[0] + "," + args[1] + ")");
		return ret;
	}
	// For BeforeAdvice
	public void divideLog(JoinPoint jp) throws Throwable {
		System.out.println("entering into method:");
	}
	// For AfterReturningAdvice
	public void calculateLog(JoinPoint jp, float intr) throws Throwable {
		if (intr == 0)
			throw new ArithmeticException("Interest can not be zero");
	}
	// for ThrowsAdvice
	public void validateLog(JoinPoint jp, Throwable error)// it is advice
	{
		System.out.println("additional concern");
		System.out.println("Method Signature: " + jp.getSignature());
		System.out.println("Exception is: " + error);
		System.out.println("end of after throwing advice...");
	}
}
```

Target Class

```
package com.aop.beans;
public class Maths {
	// for AroundAdvice
	public int add(int a, int b) {
		return a + b;
	}
	// for BeforeAdvice
	public int divide(int a, int b) {
		return a / b;
	}
	// for AfterReturningAdvice
	public float calculate(long p, int y, float r) {
```

```
                return (p * y * r) / 100;
        }
        // for ThrowsAdvice
        public void validate(int age) throws Exception {
                if (age < 18) {
                        throw new ArithmeticException("Not valid age");
                } else {
                        System.out.println("Thanks for vote");
                }
        }
}
```

Tester Class

```
package com.aop.test;
import org.springframework.context.ApplicationContext;
import org.springframework.context.support.AbstractApplicationContext;
import org.springframework.context.support.ClassPathXmlApplicationContext;
import com.aop.beans.Maths;
public class AATest {
        public static void main(String[] args) {
ApplicationContext context = new ClassPathXmlApplicationContext("com/aop/cfgs/applicationContext.xml");
                Maths e = (Maths) context.getBean("math");
                System.out.println("Sum result is:" + e.add(10, 20));
                System.out.println("division result is:" + e.divide(10, 5));
                System.out.println("Interest result is:" + e.calculate(10L, 2, 12.5f));
                try {
                        System.out.println("Interest result is:" + e.calculate(10L, 2, 0.0f));// give exception
                } catch (ArithmeticException ae) {
                        System.out.println(ae.toString());
                }
                try {
                        e.validate(19);// give exception
                } catch (Exception ae) {
                        System.out.println(ae);
                }
                System.out.println("calling validate again...");
                ((AbstractApplicationContext) context).close();
        }
}
```

ApplicationContext.xml

```
<beans>
<!-- Configure Target class -->
<bean id="math" class="com.aop.beans.Maths" />
<!-- Configure Advice class(SpringBean) -->
<bean id="loggingAspect" class="com.aop.beans.LoggingAspect" />
<aop:config>
<aop:pointcut id="ptc1" expression="execution(* com.aop.beans.Maths.add(..))" />
<aop:pointcut id="ptc2" expression="execution(* com.aop.beans.Maths.divide(..))" />
<aop:pointcut id="ptc3" expression="execution(* com.aop.beans.Maths.calculate(..))" />
<aop:pointcut id="ptc4" expression="execution(* com.aop.beans.Maths.validate(..))" />

<aop:aspect ref="loggingAspect">
<aop:around method="addLog" pointcut-ref="ptc1" />
<aop:before method="divideLog" pointcut-ref="ptc2" />
<aop:after-returning returning="intr" method="calculateLog" pointcut-ref="ptc3" />
<aop:after-throwing throwing="error" method="validateLog"   pointcut-ref="ptc4" />
</aop:aspect>
</aop:config>
</beans>
```

Aspectj (Annotation Approach)

Learn from page no:324 (Spring)

JUNIT

How to write test case for JUNIT

To work with Junit you need to write three class

1. **Main class |Service Class|Target Class (having actual logic)**
2. **TestCase class**
3. **TestRunner class**

Main class |Service Class|Target

```
public class Calculator {
      public static Integer add(Integer a, Integer b) {
        if (a == null) { return null;}
        if (b == null) { return null;}
        return a + b;
    }
}
```

#TestCase class

```
import static org.junit.Assert.*;
import org.junit.*;
public class CalculatorTest {
          @BeforeClass
          public static void setUpClass() {
          }
          @AfterClass
          public static void tearDownClass() {
          }
          @Before
          public void setUp() {
          }
          @After
          public void tearDown() {
          }
          @Test
          public void testAdd() {
              System.out.println("add");
              Integer a = 15;
              Integer b = 15;
              Integer expResult = 30;
              Integer result = Calculator.add(a, b);
              assertEquals(expResult, result);
              assertEquals(null, Calculator.add(15, null));
              assertEquals(null, Calculator.add(null, 15));
              assertEquals(null, Calculator.add(null, null));
          }
}
```

#TestRunner class

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;
public class CalculatorTestRunner {
	public static void main(String[] args) {
		Result result=JUnitCore.runClasses(CalculatorTest.class);
		for(Failure f:result.getFailures())
			System.out.println(f.toString());
	}
}
```

If we have multiple test class to test then we will add all test to testSuits following is the way….

```
import org.junit.runner.RunWith;
import org.junit.runners.Suite;
import org.junit.runners.Suite.SuiteClasses;
import com.nt.add.CalculatorTest;
@RunWith(Suite.class)
@SuiteClasses({CalculatorTest.class})
public class AllTests {
}
```

Web Services

Web Services Configuration

How to create SAX Parser:-

```
SAXParserFactory factory =SAXParserFactory. newInstance() ;
SAXParser parser =factory. newSAXParser () ;
```

How to create DOM parser:-

```
DocumentBuilderFactory factory= DocumentBuilderFactory. newInstance () ;
DocumentBuilder builder =factory. newDocumentBuilder () ;
Document doc=builder. parse(new File("rs. xml")) ;
```

RestFull	SOAP
RestFull uses xml,JSON,text,Html and etc to communicate with the server and client.	SOAP only uses xml to communicate with the server and client.
Restfull supports only server level security.	SOAP web service supports both server and application level security.

Example of Produce and Consume

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;

@Path("product")
public class ProductService {
        @Post
        @Path("/registerProduct")
        @Produces(MediaType.TEXT_PLAIN)
        @Consumes(MediaType.APPLICATION_XML)
        public string registerProduct(String productXML) {
                System.out.println("productXML");
                return "Product registered completed successfully";
        }

}
```

SOAP	REST
SOAP is a protocol.	REST is an architectural style.
SOAP stands for Simple Object Access Protocol.	REST stands for REpresentational State Transfer.
SOAP can't use REST because it is a protocol.	REST can use SOAP web services because it is a concept and can use any protocol like HTTP, SOAP.
SOAP uses services interfaces to expose the business logic.	REST uses URI to expose business logic.
JAX-WS is the java API for SOAP web services.	JAX-RS is the java API for RESTful web services.
SOAP defines standards to be strictly followed.	REST does not define too much standards like SOAP.
SOAP requires more bandwidth and resource than REST.	REST requires less bandwidth and resource than SOAP.
SOAP defines its own security.	RESTful web services inherits security measures from the underlying transport.
SOAP permits XML data format only.	REST permits different data format such as Plain text, HTML, XML, JSON etc.
SOAP is less preferred than REST.	REST more preferred than SOAP.