Spring Core

Basics

* We need spring jar and common logging jar for initial setup.
* If we want to use annotations instead of xml then we have to include cglib.jar and asam.jar.
* Ways to initialize spring bean.
* *ApplicationContext context=new ClassPathXml ApplicationContext(springconfigurationfilename);*
* *ApplicationContext context=new AnnotationConfig ApplicationContext(anotationconfigclassname.Class);*
* Using BeanFactory.

1. Resource resource=new ClassPathResource(springconfigurationfilename);
2. BeanFactory=new XmlBeanFactory(resource);

* If we want to make any class as config then use @Configuration before class name.
* If we want to make any method as bean the use @Bean(name=’somename’) before method name.
* If you want to include one xml config file to another xml file then use below syntax.

<beans>

<import resource=”beanfile1.xml”/>

<import resource=”beanfile1.xml/>

</beans>

* If you want to include one config class to other config class then use below syntax.

@Configuration

@Import({*anotationconfigclassname*.class, *anotationconfigclassname*.class })

public class AppConfig {

}

* Whenever we use constructor injection, always give “type” attribute with respect to each property.

For example-

*<constructor-arg type="java.lang.String">*

*<value>188</value>*

*</constructor-arg>*

* We can inject values to bean properties by using “p” schema. For that we have to include following lines in at the top of xml file.

*xmlns:p=”*[*http://www.springframework.org/schema/p*](http://www.springframework.org/schema/p)*”*

For example- Here name is an attribute of “*FileNameGenerator* “class.

<bean id="FileNameGenerator" class="com.mkyong.common.FileNameGenerator" p:name="mkyong"/>

* Whenever bean is used for some special purpose, it is always preferable to use inner bean.

For example –

<property name="person">

<bean class="com.mkyong.common.Person">

<property name="name" value="mkyong" />

<property name="address" value="address1" />

<property name="age" value="28" />

</bean>

</property>

* There are 5 types of bean scopes.

1. singleton – Return a single bean instance per Spring IoC container
2. prototype – Return a new bean instance each time when requested
3. request – Return a single bean instance per HTTP request. \*
4. session – Return a single bean instance per HTTP session. \*
5. globalSession – Return a single bean instance per global HTTP session. \*

* 1. Without annotation:

<bean id="customerService" class="com.mkyong.customer.services.CustomerService" scope="prototype"/>

* 1. With annotation:
     + - Write following annotations at the top of bean.

@Service

@Scope("scope name")

* + - * Write following configurations in xml file.

xmlns:context=”<http://www.springframework.org/schema/context>”

“http://www.springframework.org/schema/context

<http://www.springframework.org/schema/context/spring-context-2.5.xsd>”

<context:component-scan base-package="package name" />

* Usage of collections

1. List

<property name="lists">

<list>

<value></value>

<ref bean=”/>

</list>

</property>

1. Set

<property name="sets">

<set>

<value></value>

<ref bean=”/>

</set>

</property>

1. Map

<property name="maps">

<map>

<entry key=”” value=””/>

<entry key-ref=”” value-ref=””/>

</map>

</property>

1. Properties

<property name="propert">

<props>

<prop key=””>somevalue</prop>

</props>

</property>

1. ListFactoryBean – this class provides developers a way to create concrete List Collection class in Spring bean configuration file.
2. Without util:list :



1. With util:list :
2. Add following lines in xml configuration.





1. Add following code



1. SetFactoryBean- this class provides developers a way to create concrete Set Collection class in Spring bean configuration file.
2. Without util:set :



1. With util:set:
2. Add following lines in xml configuration.





1. Add following code



1. MapFactoryBean: this class provides developers a way to create concrete Map Collection class in Spring bean configuration file.
2. Without util:map:



1. With util:map:
2. Add following lines in xml configuration.





1. Add following code



* Injecting Date in Spring

1. Factory bean
2. We will first declare the format using SimpleDateFormat().



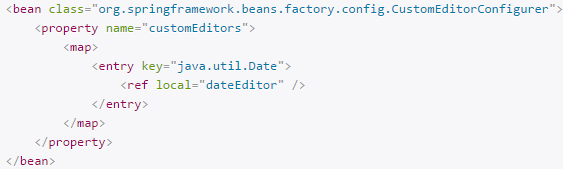
1. For converting String to Date, we have to call parse() method.



1. CustomDateEditor
2. We have to declare the format and parse is automatically built in it that convert String to Date.

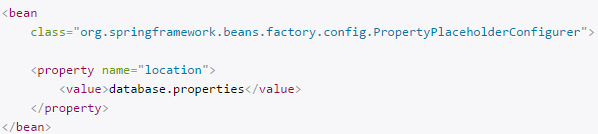


1. We have to declare “CustomEditorConfigurer” to make spring convert bean properties whose type is java.util.Date.



* Injecting Properties using PropertyPlaceholderConfigurer.

1. Declare the location of properties file.



1. For accessing values of property file. We have to use ${variablename}.



* Inheritance in Spring:

1. For inheriting parent class bean we have to use “parent” attribute in bean declaration.



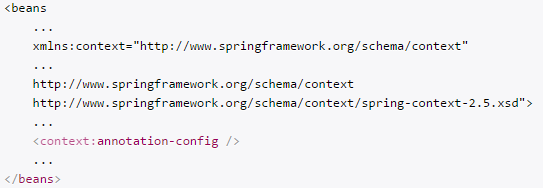
1. For making parent bean as abstract. Please use “abstract” attribute.



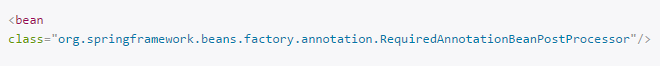
* @Required annotation

There are 2 ways to implements @Required annotation.

1. Add Spring context and <context:annotation-config /> in bean configuration file.

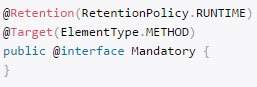


1. Include ‘RequiredAnnotationBeanPostProcessor’ directly in bean configuration file.

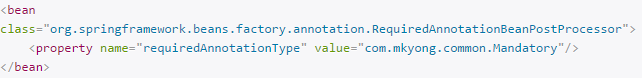


Note: Like @Required annotation we can also make custom annotation.

1. Create @Mandatory interface



1. Include your new @Mandatory annotation in ‘RequiredAnnotationBeanPostProcessor’ class.



* InitializingBean and DisposableBean

1. These are marker interfaces that perform certain action on initialization and destruction of beans.
2. For bean implemented *InitializingBean*, it will run *afterPropertiesSet()* after all bean properties have been set.
3. For bean implemented *DisposableBean*, it will run *destroy()* after Spring container is released the bean.
4. The *ConfigurableApplicationContext.close()* will close the application context, releasing all resources and destroying all cached singleton beans.



* Init-method and destroy-method

1. We can use ***init-method*** and ***destroy-method*** as attribute in bean configuration file for bean to perform certain actions upon initialization and destruction.



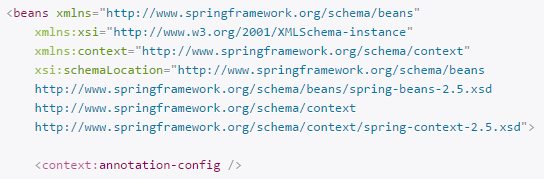
* @PostConstruct and @PreDestroy

We have to use these annotations with methods that you want to initialize and destroy and it can enable it by following 2 ways.

1. CommonAnnotationBeanPostProcessor



1. <context:annotation-config />



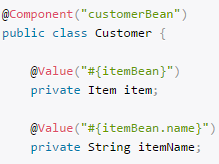
* Spring EL(SpEL)

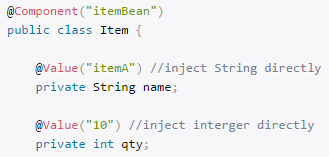
1. In XML

The SpEL are enclosed with #{ SpEL expression }

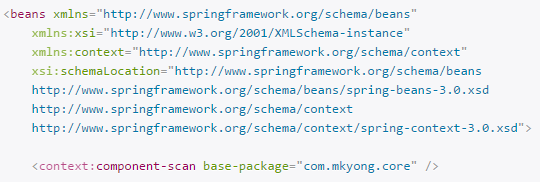


1. EL in Annotation
2. We have to use @Value annotation with class properties.

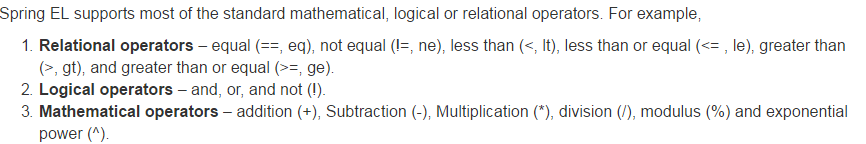




1. Enable auto component scanning.



1. SpEL operators.

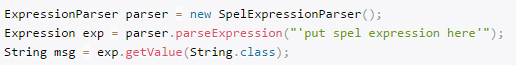


Note: We should use textual representation of operators (>, <, >=,<=) like (gt,lt,ge,le) in XML.

1. SpEL supports Regular expressions using a simple keyword “matches”.



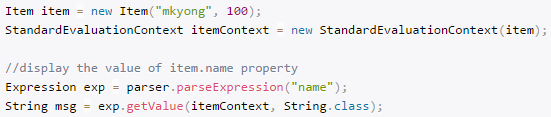
1. *ExpressionParser* interface in SpEl



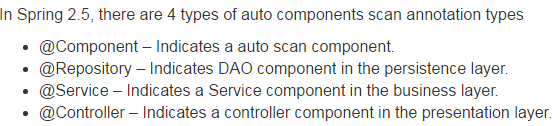
Example:



1. *StandardEvaluationContext* class in SpEL – it is used for processing bean classes.



* Auto scanning components



1. Filter component – include
2. Scan and register components’ name which matched defined “regex”, even the class is not annotated with @Component.

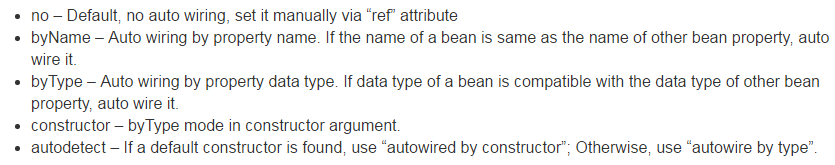


1. Filter component – exclude
2. Exclude specified components, to avoid Spring to detect and register it in Spring container.

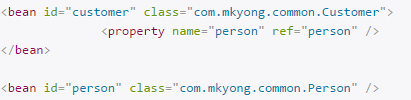


* Autowiring in Spring

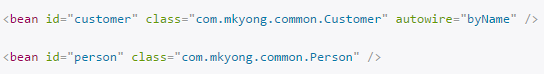
1. In spring there are 5 autowiring modes.



1. Auto-Wiring ‘no’



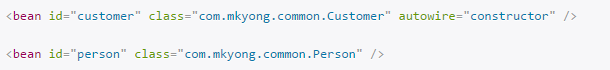
1. Auto-Wiring ‘byName’



1. Auto-Wiring ‘byType’



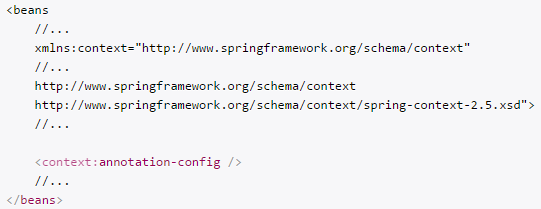
1. Auto-Wiring ‘constructor’



1. Auto-Wiring ‘autodetect’



1. @Autowired
2. We can use it with fields, contructors, setter methods.
3. The @Autowired annotation is auto wire the bean by matching data type.
4. We can enable it by following 2 ways.
5. Include <context:annotation-config />



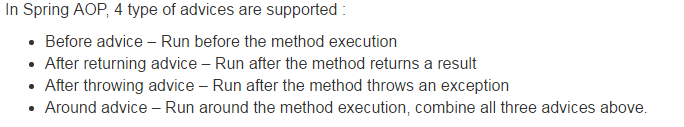
1. Include AutowiredAnnotationBeanPostProcessor



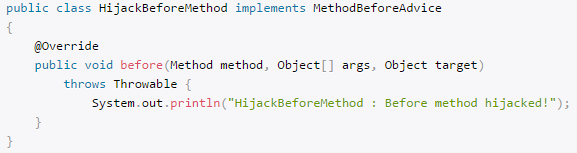
1. By default @Autowired perform dependency checking. If you want to disable that dependency checking please use @Autowired(required=false).
2. As we know that @Autowired is used as ‘byType’ so in bean config file(xml) if there are more than one bean of same type than there is a conflict which bean will be autowired. So here we use @Qualifier(“beanid”).

* Spring AOP(Aspect oriented programming)

1. When a method is execute, Spring AOP can hijack the executing method, and add extra functionality before or after the method execution.

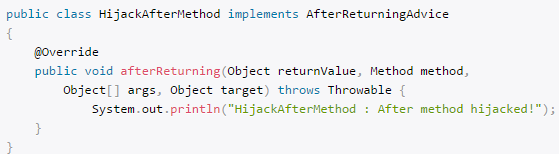


1. We have to include following jar files in project classpaths.
   * aspectjrt.jar
   * aspectjweaver.jar
   * aspectj.jar
   * aopalliance.jar
2. Types of advices
3. Before Advice



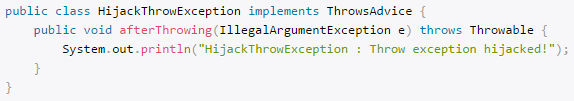


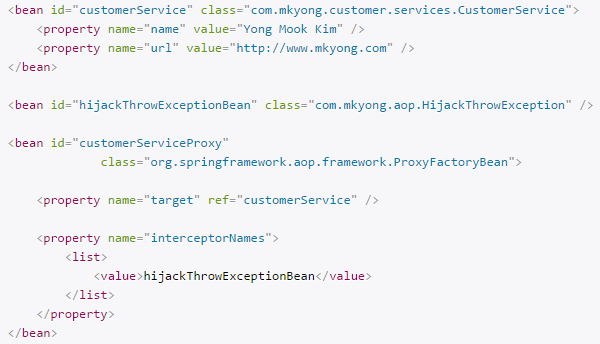
1. After returning advice





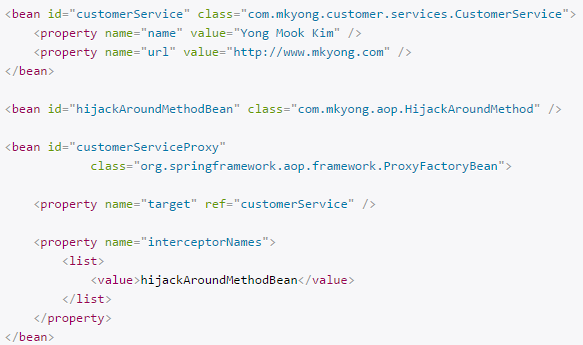
1. After throwing advice



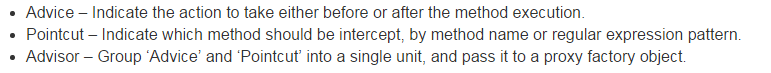


1. Around advice

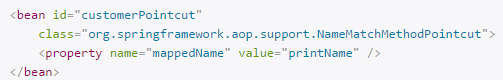




1. Difference between Advice,Pointcut,Advisor.



1. Pointcuts
2. NameMatchMethodPointcut
3. Intercept a printName() method via ‘pointcut’ and ‘advisor’. Create a NameMatchMethodPointcut pointcut bean, and put the method name you want to intercept in the ‘mappedName‘ property value.



1. Create a DefaultPointcutAdvisor advisor bean, and associate both advice and pointcut.



1. Replace the proxy’s ‘interceptorNames’ to ‘customerAdvisor’.



**Note:** Spring comes with PointcutAdvisor class to save your work to declare advisor and pointcut into different beans, you can use NameMatchMethodPointcutAdvisor to combine both into a single bean.

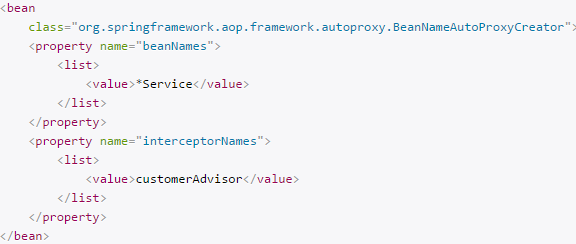


1. RegexpMethodPointcutAdvisor



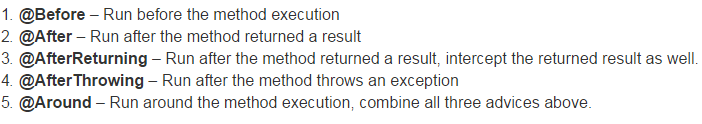
Remaining steps will be same as above.

1. BeanNameAutoProxyCreator
2. Without this, we have to create a separate bean for each class.
3. In auto proxy mechanism, you just need to create a BeanNameAutoProxyCreator, and include all your beans (via bean name, or regular expression name) and ‘advisor’ into a single unit.

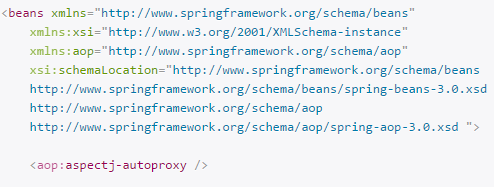


* AOP + AspectJ

1. Annotations.
2. Supports following annotations.



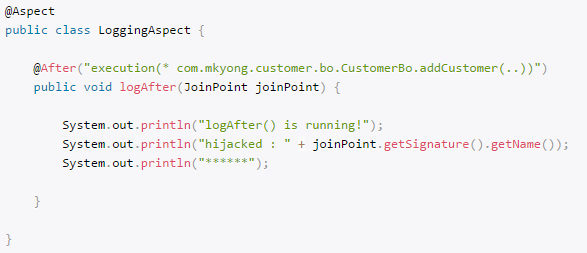
1. We can enable AspectJ by following config.



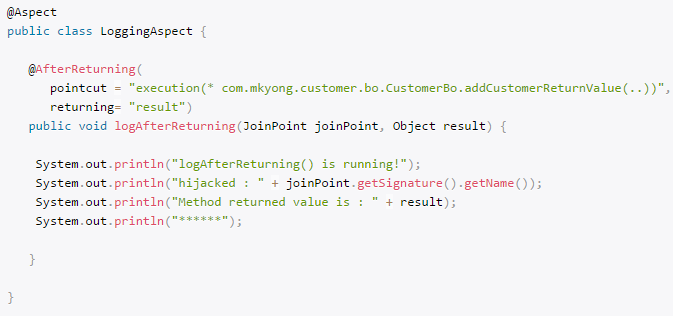
1. Annotations examples
2. @Before



1. @After



1. @AfterReturning



Note: To intercept returned value, the value of the “returning” attribute (result) need to be same with the method parameter (result).

1. @AfterThrowing



1. @Around



1. XMLs
2. <aop:before> = @Before



1. <aop:after> = @After



1. <aop:after-returning> = @AfterReturning



1. <aop:after-throwing> = @AfterThrowing



1. <aop:after-around> = @Around



* JDBC

1. In simple Spring JDBC application we have to follow following steps.
2. In XML file, create an object for DriverManagerDataSource which is responsible for all setting up driver name, database URL, username and password of database.



1. Refer this bean in your main class object.



1. Call getConnection() method by the object of DriverManagerDataSource and proceed with remaining steps of JDBC.



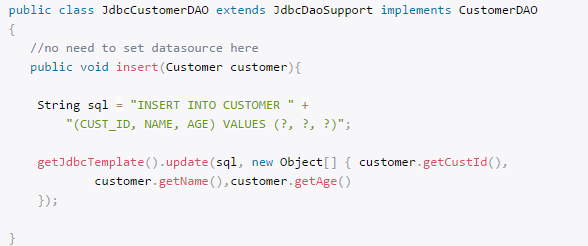
1. JdbcTemplate
2. Without JdbcTemplate, we have to create many redundant codes (create connection, close connection, handle exception) in all the DAO database operation methods – insert, update and delete. It just not efficient, ugly, error prone and tedious.
3. With JdbcTemplate, we save a lot of typing on the redundant codes, becuase JdbcTemplate will handle it automatically.

Example of Inserting Object to DB.

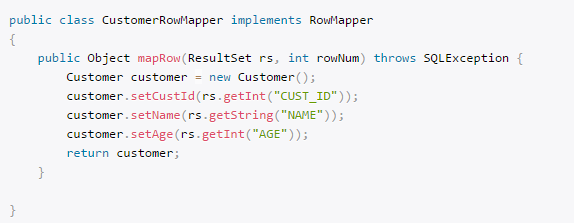


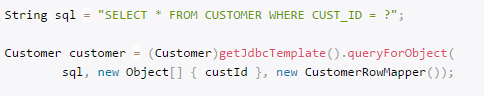
1. JdbcDaoSupport

By extended the JdbcDaoSupport, set the datasource and JdbcTemplate in our class is no longer required, we just need to inject the correct datasource into JdbcCustomerDAO. And we can get the JdbcTemplate by using a getJdbcTemplate() method.



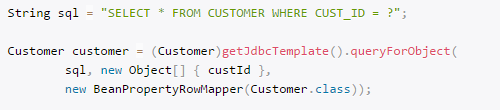
1. Query for Single Row (JdbcTemplate)
2. Custom RowMapper



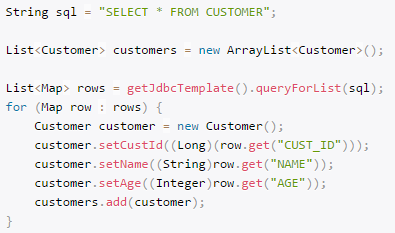


1. BeanPropertyRowMapper

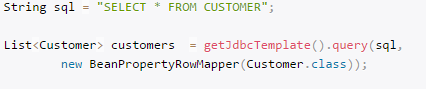
It can maps a row’s column value to a property by matching their names. Just make sure both the property and column has the same name, e.g property ‘custId’ will match to column name ‘CUSTID’ or with underscores ‘CUST\_ID’.



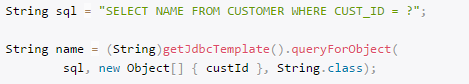
1. Query for multiple rows (JdbcTemplate)
2. RowMapper not supported in queryForList(). We have to map it manually.



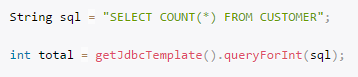
1. BeanPropertyRowMapper



1. Querying for a Single Value (JdbcTemplate)
2. Single column name



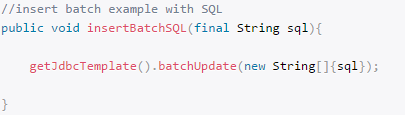
1. Total number of rows



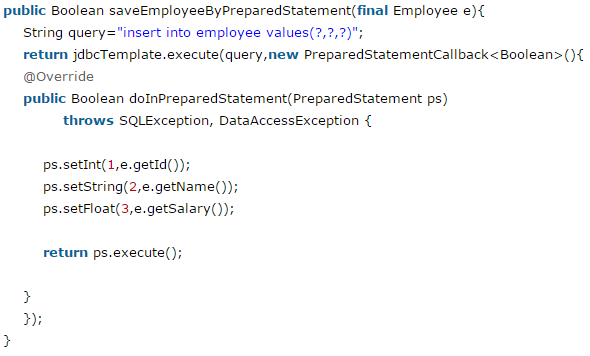
1. JdbcTemplate batchUpdate()
2. We can execute multiple statements together. It compiled once but execute multiple times.



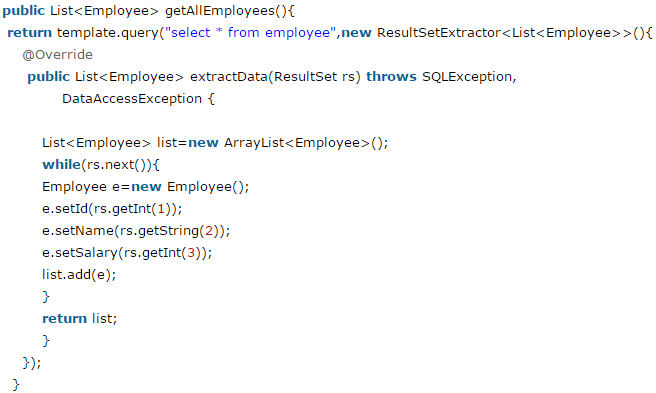
1. We can execute any SQL directly.



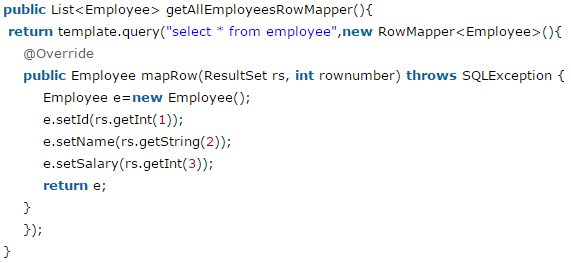
1. PreparedStatement in Spring JdbcTemplate



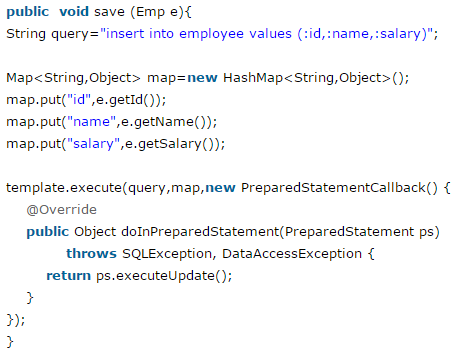
1. ResultSetExtractor



1. RowMapper



1. NamedParameterJdbcTemplate
2. Extend *NamedParameterJdbcDaoSupport* in the class in which you want to use Named Parameter queries.
3. Spring provides another way to insert data by named parameter. In such way, we use names instead of ?(question mark). So it is better to remember the data for the column.



Note: Here in place of ‘template’ we can use getNamedParameterJdbcTemplate().