Day 1: Intro to Spring Framework

1) Spring employs four key strategies:

- o Lightweight and minimally invasive development with plain old
  Java objects (POJOs)
- o Loose coupling through dependency injection and interface
  orientation
- o Declarative programming through aspects
- o Boilerplate code reduction through aspects and templates

2)POJO : Spring almost never forces you to implement a Spring-specific interface or extend a Spring-specific class

```
        package com.synechron;
    public class HelloWorldBean
    {
            public String sayHello()
            {
                    return "Hello World";
             }
        }
```

3)DI : Dependency Injection is a technique through which, the dependent objects are given to the components so that components need not create dependent objects themselves

```
public class Employee  {
        private Address addr;
        public Employee(Address addr)
           {
              this.addr = addr
           }
   }
```

4)AOP : AOP is a programming technique that promotes separation of business logic from the system concerns such as logging, transaction management, security, persistence etc.

5) Eliminating boilerplate code with templates

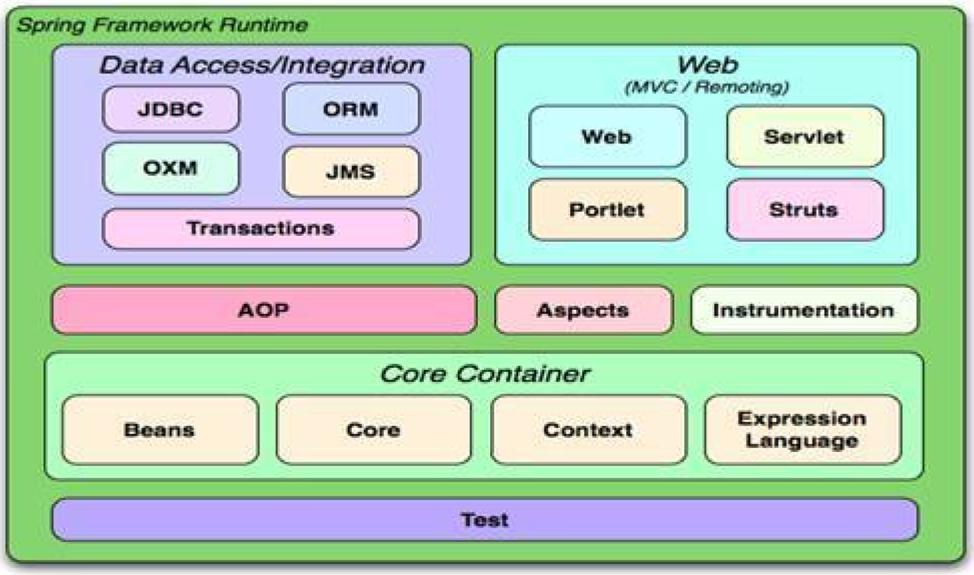
- ❑ Boiler plate code -the code that you often have to write over and over again to accomplish common tasks. E.g- Jdbc Code

Spring seeks to eliminate boilerplate code by encapsulating it in templates. Spring's JdbcTemplate makes it possible to perform database operations without all of the ceremony required by traditional JDBC

6) Spring Container : In a Spring-based application, your application objects will live within the Spring container.

The container will create the objects, wire them together, configure them, and manage their complete lifecycle from born to dead

7) Spring Architecture :

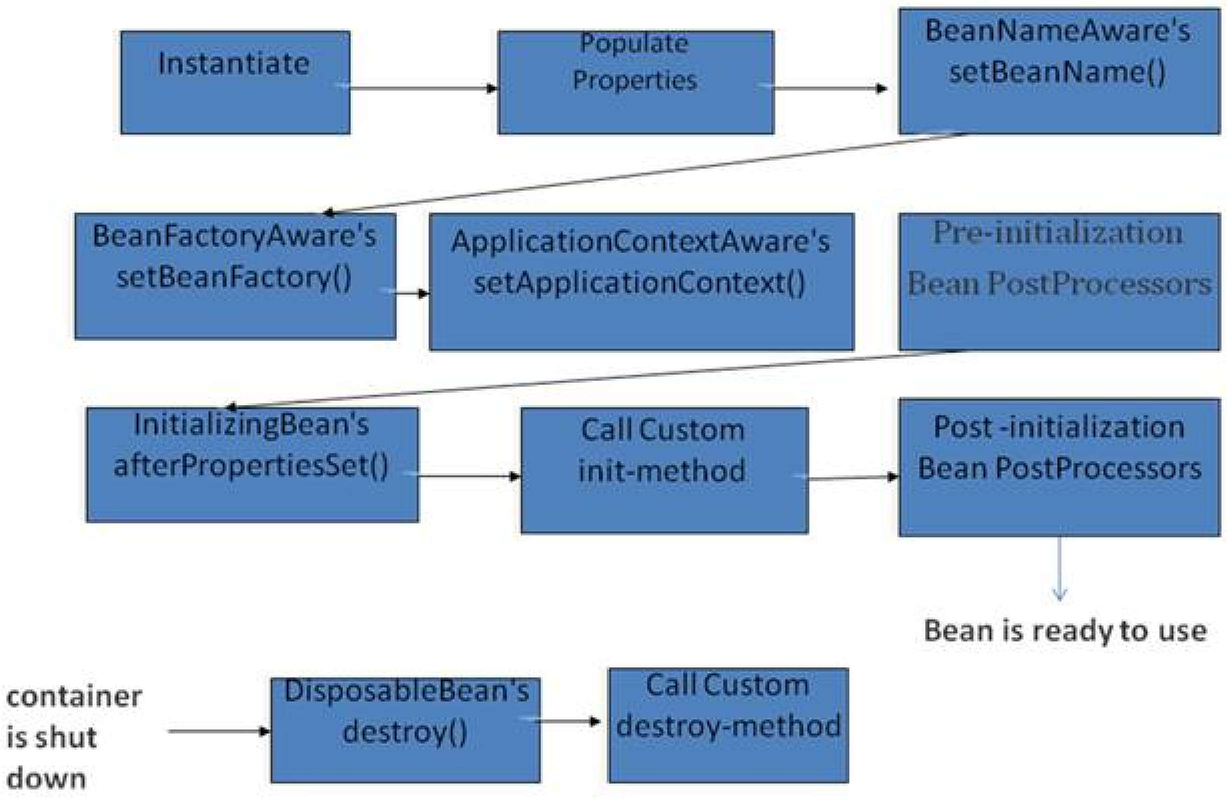


8)Hands on: Assignment no 1: Hello world demo

9)Spring Container

- ❑ There’s no single Spring container. Spring comes with several container implementations that can be categorized into two distinct types.
- ❑ ***Bean factories*** (defined by the org.springframework.beans.factory.BeanFactory interface) are the simplest of containers, providing basic support for DI.
- ❑ ***Application contexts* (**defined by the org.springframework.context.ApplicationContext interface) ApplicationContext is a sub-interface of BeanFactory. It adds easier integration with Spring's AOP features; message resource handling (for use in internationalization), event publication and application-layer specific contexts such as the WebApplicationContext for use in web applications.

10) Life Cycle of a Bean in Spring Container


Instantiate
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
ApplicationContextAware's setApplicationContext()
Pre-initialization Bean PostProcessors
InitializingBean's afterPropertiesSet()
Call Custom init-method
Post -initialization Bean PostProcessors
Bean is ready to use
container is shut down
DisposableBean's destroy()
Call Custom destroy-method

11)Types of Injections

- Setter Injection

```
<bean id="emp" class="com.emp.Employee">
  <property name=“fname” value=“a” />
 <property name=“lname” value=“b” />
</bean>
```

- Constructor Injection

```
<bean id="emp" class="com.emp.Employee">
  <constructor-arg value=“Joy" />
   <constructor-arg value=“Roy" /> </bean>
```

- Collection Injection

<list> Wiring a list of values, allowing duplicates

<set> Wiring a set of values, ensuring no duplicates
<map>Wiring a collection of name-value pairs where name and
value can be of any type
<props>Wiring a collection of name-value pairs where the name
and value are both Strings

12) Referencing other beans

- ❑ A bean is a dependency of another bean is expressed by the fact the one bean is set as a property of another.
- ❑ It is achieved with the <ref/> element in XML based configuration metadata of beans

```
<bean id="addr" class="com.emp.Address " />
<bean id="emp" class="com.emp.Employee ">
 <property name="address" ref="addr" />
 </bean>
```

13) Bean Scope

- ❑ singleton : Scopes the bean definition to a single instance per Spring container (default).
- ❑ prototype : Allows a bean to be instantiated any number of times (once per use).
- ❑ request : Scopes a bean definition to an HTTP request. Only valid when used with a web -capable Spring context (such as with Spring MVC).
- ❑ session : Scopes a bean definition to an HTTP session. Only valid when used with a web-capable Spring context (such as with Spring MVC).

- ❑ global- session : Scopes a bean definition to a global HTTP session. Only valid when used in a portlet context.

<bean id=*"helloWorld"* class=*"com.pk1.HelloWorld"* *scope="singleton"*>

14)Bean Inheritance

```
  <bean id="helloWorld" class="com.pk1.HelloWorld">
     <property name="message1" value="Hello World!"/>
     <property name="message2" value="Hello Second 
World!"/>
   </bean>

   <bean id="helloIndia" class="com.pk1.HelloIndia" 
parent="helloWorld">
      <property name="message1" value="Hello India!"/>
       <property name="message3" value="Namaste 
India!"/>
    </bean>
```

15) Hands on

Day 2: Minimizing XML Configuration in Spring

1) Four kinds of autowiring

- byName
- byType
- Constructor
- Autodetect

byName—Attempts to match all properties of the autowired bean with beans that have the same name (or ID) as the properties.

byType—Attempts to match all properties of the autowired bean with beans whose types are assignable to the properties.

constructor—Tries to match up a constructor of the autowired bean with beans whose types are assignable to the constructor arguments.

Autodetect- If you want to autowire your beans, but you can't decide which type of autowiring to use, have no fear. You can set the autowire attribute to autodetect to let Spring make the decision for you.

2) Wiring with Annotation

- ❑ **@Autowired**
- ❑ **@Qualifier**
- ❑ **@Inject**

@Autowired -When Spring sees that you've annotated setter method with @Autowired it'll try to perform byType autowiring on the method. You can annotate @autowired at property/setter method/constructor.

@Qualifier - Suppose you have two beans of same type .In that event, there’s no way for @Autowired to choose which one you really want.

So, instead of guessing, a NoSuchBeanDefinitionException will be thrown and wiring will fail.

To help @Autowired figure out which bean you want, you can accompany it with Spring’s @Qualifier annotation.

@Inject - This annotation is an almost complete drop-in replacement for Spring’s @Autowired annotation. So, instead of using the Spring-specific @Autowired annotation, you might choose to use @Inject on the
instrument property:

Just like @Autowired, @Inject can be used to autowire properties, methods, and constructors.

3) What is Autodiscovery?

The <context:component-scan> element does everything that <context:annotation-config> does, plus it configures Spring to automatically discover beans and declare them for you.

What this means is that most (or all) of the beans in your Spring application can be declared and wired without using <bean>

@Component—

A general-purpose stereotype annotation indicating that the class is a Spring component

- ❑ For example, suppose that our application context has the guitar beans in it. We can eliminate the explicit <bean> declarations from the XML configuration by using <context:component-scan> and annotating the Guitar class with @Component.

```
@Component
public class Guitar implements Instrument
{
            public void play()
            { System.out.println("Strumstrumstrum"); }
}
```

By default, the bean’s ID will be generated by camel-casing the class name. In the case of Guitar that means that the bean ID will be guitar.

4) Hands on

Spring - Java Based Configuration

- ❑ Java based configuration option enables you to write most of your Spring configuration without XML but with the help of few Java-based annotations explained below

- ❑ @Configuration & @Bean Annotations:
  Annotating a class with the @Configuration indicates that the class can be used by the Spring IoC container as a source of bean definitions. The @Bean annotation tells Spring that a method annotated with @Bean will return an object that should be registered as a bean in the Spring application context. The simplest possible @Configuration class would be as follows:

```
    @Configuration
 public class HelloWorldConfig {
   @Bean
   public HelloWorld helloWorld(){      return new
HelloWorld();   }
  }
```

Above code will be equivalent to the following XML configuration:

```
 <beans>   <bean id="helloWorld" class="HelloWorld"
/></beans>
```

Aspect Oriented Programming

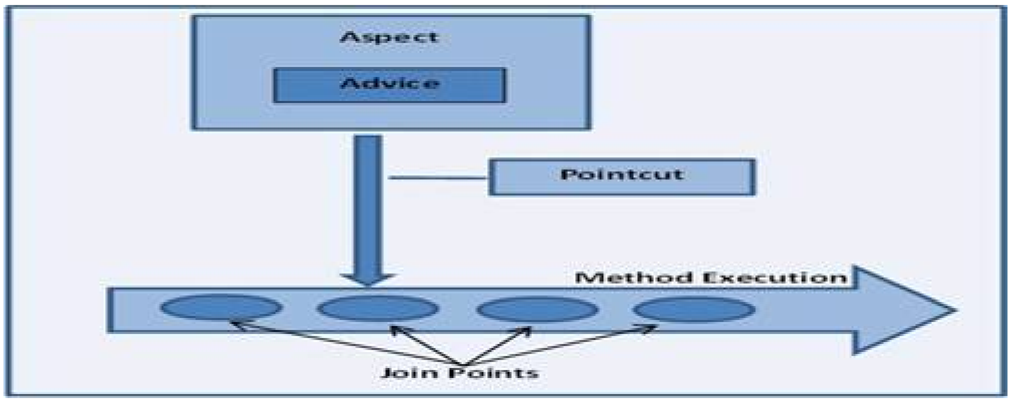
1)Basics of AOP

AOP is a programming technique that promotes separation of business logic from cross-cutting concerns.

A cross-cutting concern can be described as any functionality that affects multiple points of an application. E.g. logging or security.

2) AOP terminology

- ❑ **Advice** : The job of an aspect is called *advice*. Advice defines both the *what* and the *when* of an aspect.
- ❑ **Join Points** : Your application may have thousands of opportunities for advice to be applied. These opportunities are known as join points.
  A *join point* is a point in the execution of the application where an aspect can be plugged in.
- ❑ **Pointcuts** : A pointcut definition matches one or more join points at which advice should be woven.
  If advice defines the *what* and *when* of aspects, then pointcuts define the *where*.
- ❑ **Aspects** : An *aspect* is the merger of advice and pointcuts. Taken together, advice and pointcuts define everything there is to know about an aspect — what it does and where and when it does it.



- **Target :** A *target* is the object that is being advised.
- **Proxy :** A *proxy* is the object created after applying advice to the target object.

3) *Weaving* is the process of applying aspects to a target object to create a new proxied object.

Compile time
Class load time
Runtime

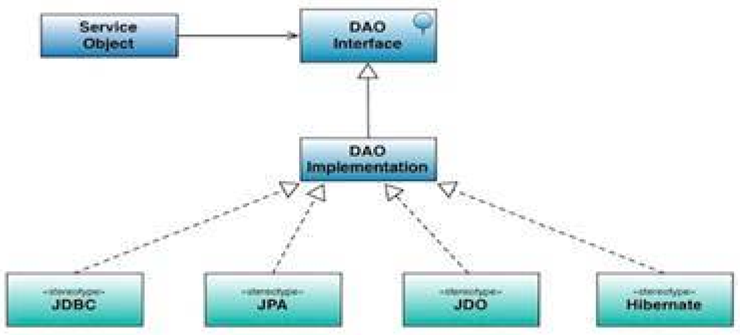
4) Spring aspects can work with five kinds of advice:

- **Before**—The advice functionality takes place before the advised method is invoked.
- **After**—The advice functionality takes place after the advised method completes, regardless of the outcome.
- **After-returning**—The advice functionality takes place after the advised method successfully completes.
- **After-throwing**—The advice functionality takes place after the advised method throws an exception.
- **Around**—The advice wraps the advised method, providing some functionality before and after the advised method is invoked.

5) *Declaring aspects in XML -Demo*
6) Annotating aspects –Demo

Day 3: Data Access using JDBC in Spring

1)**DAO:** *DAO* stands for *data access object*, which perfectly describes a DAO’s role in an application. DAOs exist to provide a means to read and write data to the database.



2) Data access templates :

Spring separates the fixed and variable parts of the data access process into two distinct classes: *templates* and *callbacks*.

- ❑ Templates manage the fixed part of the process
- ❑ Callback manage custom data access code.

3) *JDBC driver-based data source*

```
<bean id="dataSource“
class="org.springframework.jdbc.datasource.DriverManager
DataSource/SingleConnectionDataSource ">
                        <property name="driverClassName“
value=“ "/>
                        <property name=“url“ value=“ "/>
                        <property name=“username“ value=“ "/>
                        <property name=“password“ value=“ "/>
  </bean>
```

4) Demo

For Spring MVC refer below link

http://www.tutorialspoint.com/spring/spring_web_mvc_framework.htm