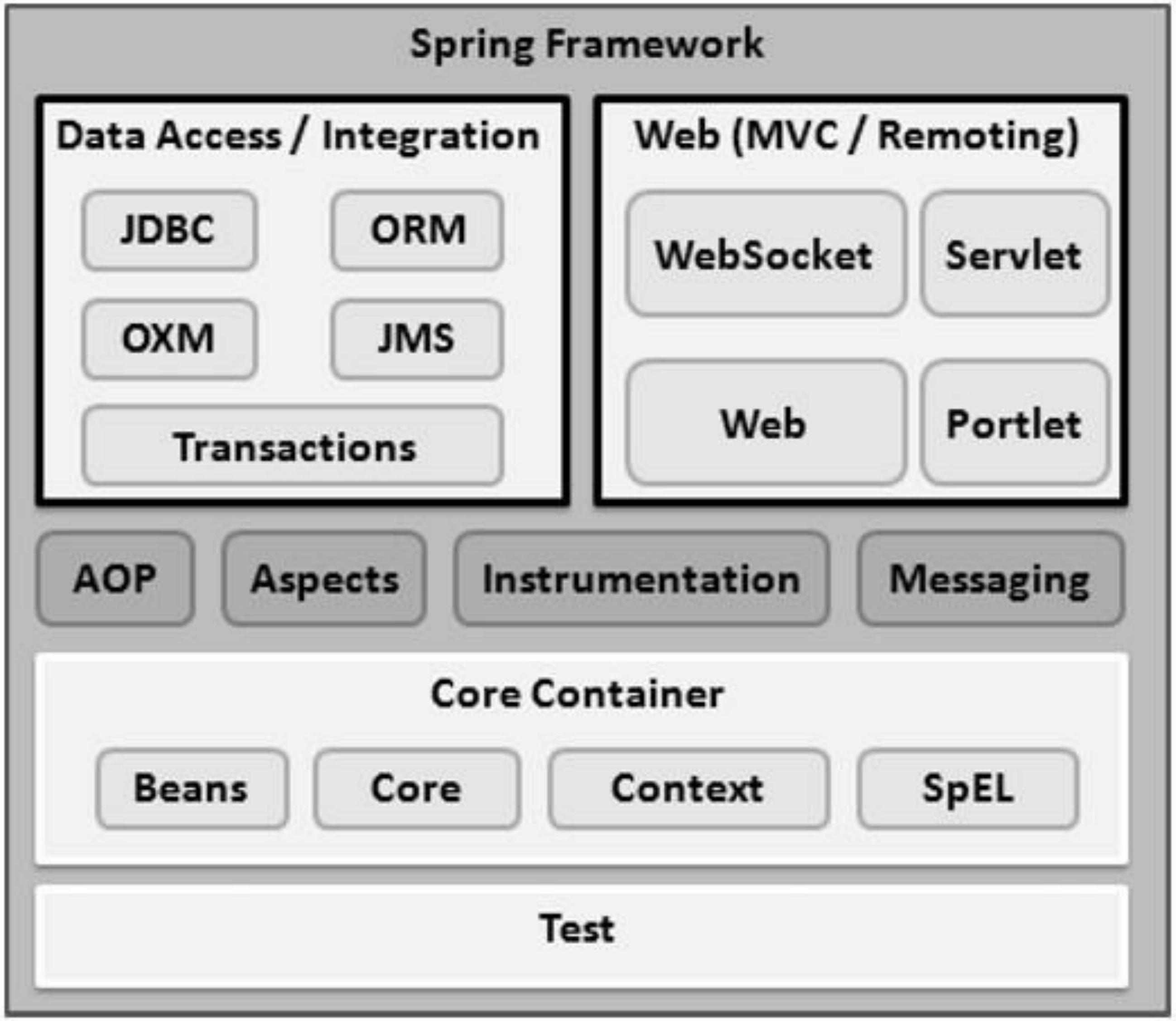
Spring Framework

Objectives

- Purpose:
  - To understand the spring IOC, MVC, JDBC , Database Transactions and AOP.
- Product:
  - Spring3 IOC containers and Dependency Injection
  - Spring3 MVC
  - Spring3 JDBC and transactions
  - Spring3 AOP
- Process:
  - Theory Sessions along with hands on assignments
  - A review at the end of the session and a Quiz.

Table of Contents

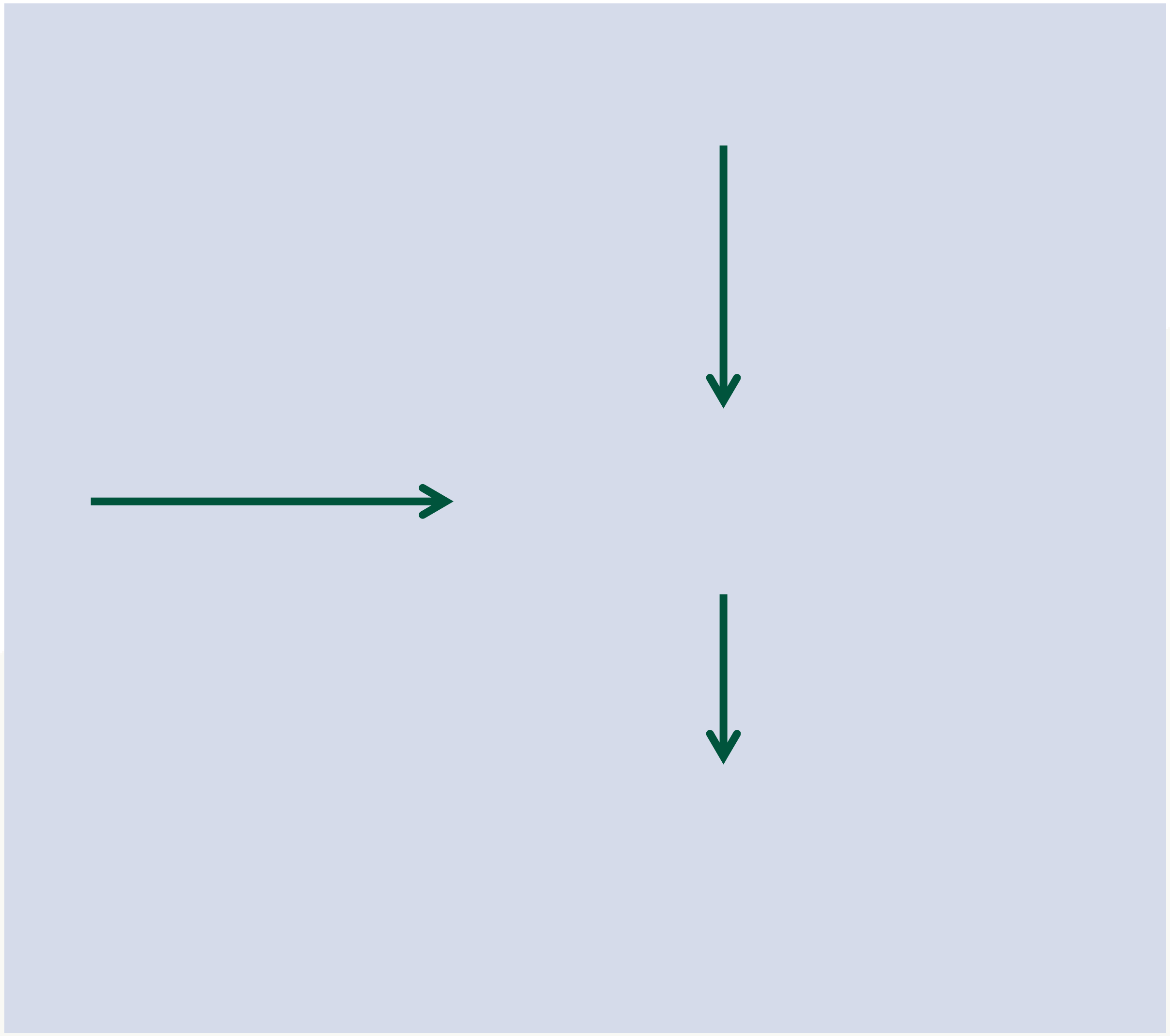
Spring Architecture



Spring IOC containers

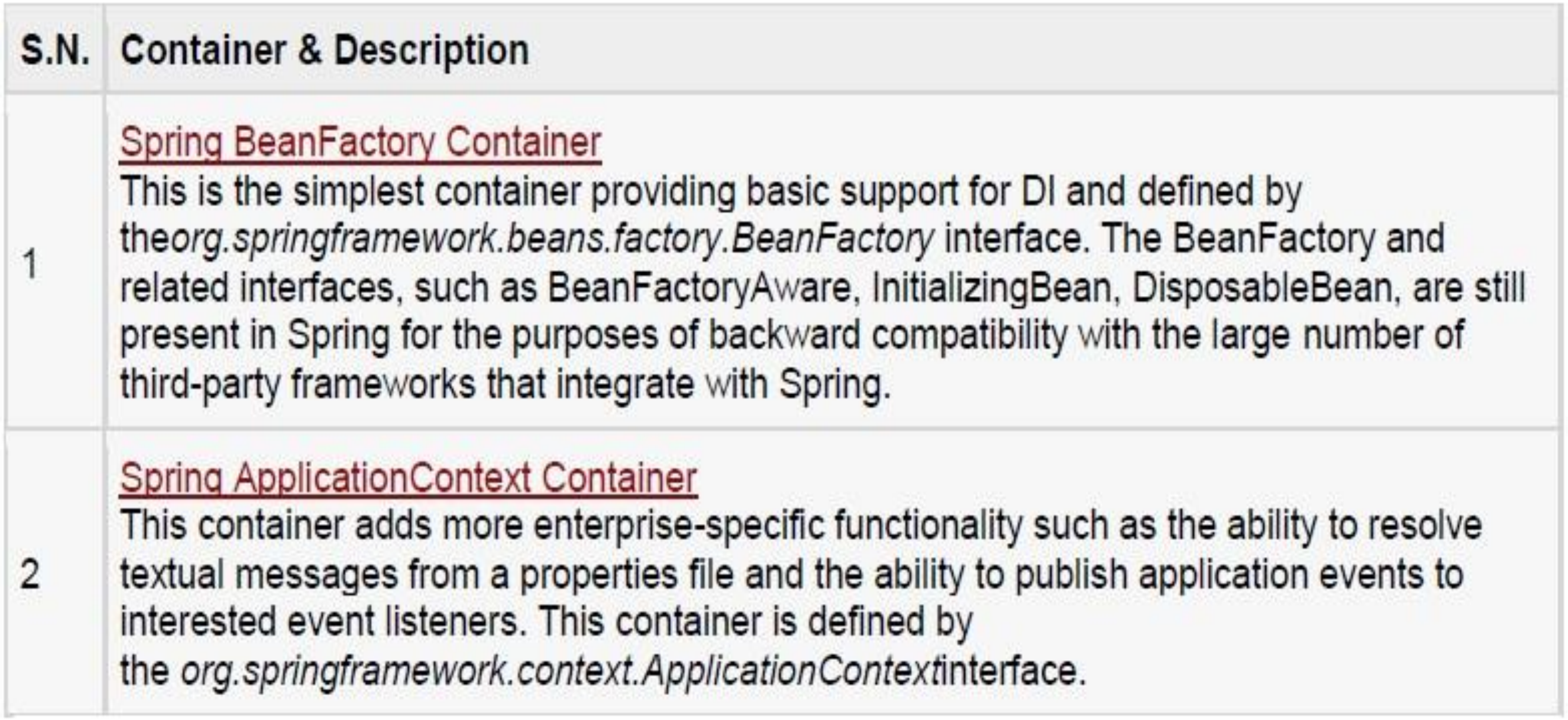
- The Spring container will create the objects, wire them together, configure them, and manage their complete lifecycle from creation till destruction.
- The Spring container uses dependency injection (DI) to manage the components that make up an application.
- The container gets its instructions on what objects to instantiate, configure, and assemble by reading configuration metadata provided.
- The configuration metadata can be represented either by XML, Java annotations, or Java code.
- The Spring IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

Spring IOC containers



Spring IOC containers

Spring provides following two distinct types of containers.



S.N.	Container & Description
1	Spring BeanFactory Container This is the simplest container providing basic support for DI and defined by the*org.springframework.beans.factory.BeanFactory* interface. The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.
2	Spring ApplicationContext Container This container adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners. This container is defined by the *org.springframework.context.ApplicationContext*interface.

Spring IOC containers

Spring BeanFactory Container

- This is the simplest container providing basic support for DI and defined by the org.springframework.beans.factory.BeanFactory interface.

- The BeanFactory and related interfaces, such as BeanFactoryAware, InitializingBean, DisposableBean, are still present in Spring for the purposes of backward compatibility with the large number of third-party frameworks that integrate with Spring.

Spring IOC containers

XmlBeanFactory

XmlBeanFactory factory = new XmlBeanFactory

(new ClassPathResource("Beans.xml"));

Spring ApplicationContext Container

- The Application Context is spring's more advanced container. Similar to BeanFactory it can load bean definitions, wire beans together and dispense beans upon request.
- Additionally it adds more enterprise-specific functionality such as the ability to resolve textual messages from a properties file and the ability to publish application events to interested event listeners.
- This container is defined by the ***org.springframework.context.ApplicationContext*** interface.

Spring IOC containers

The most commonly used ApplicationContext implementations are:

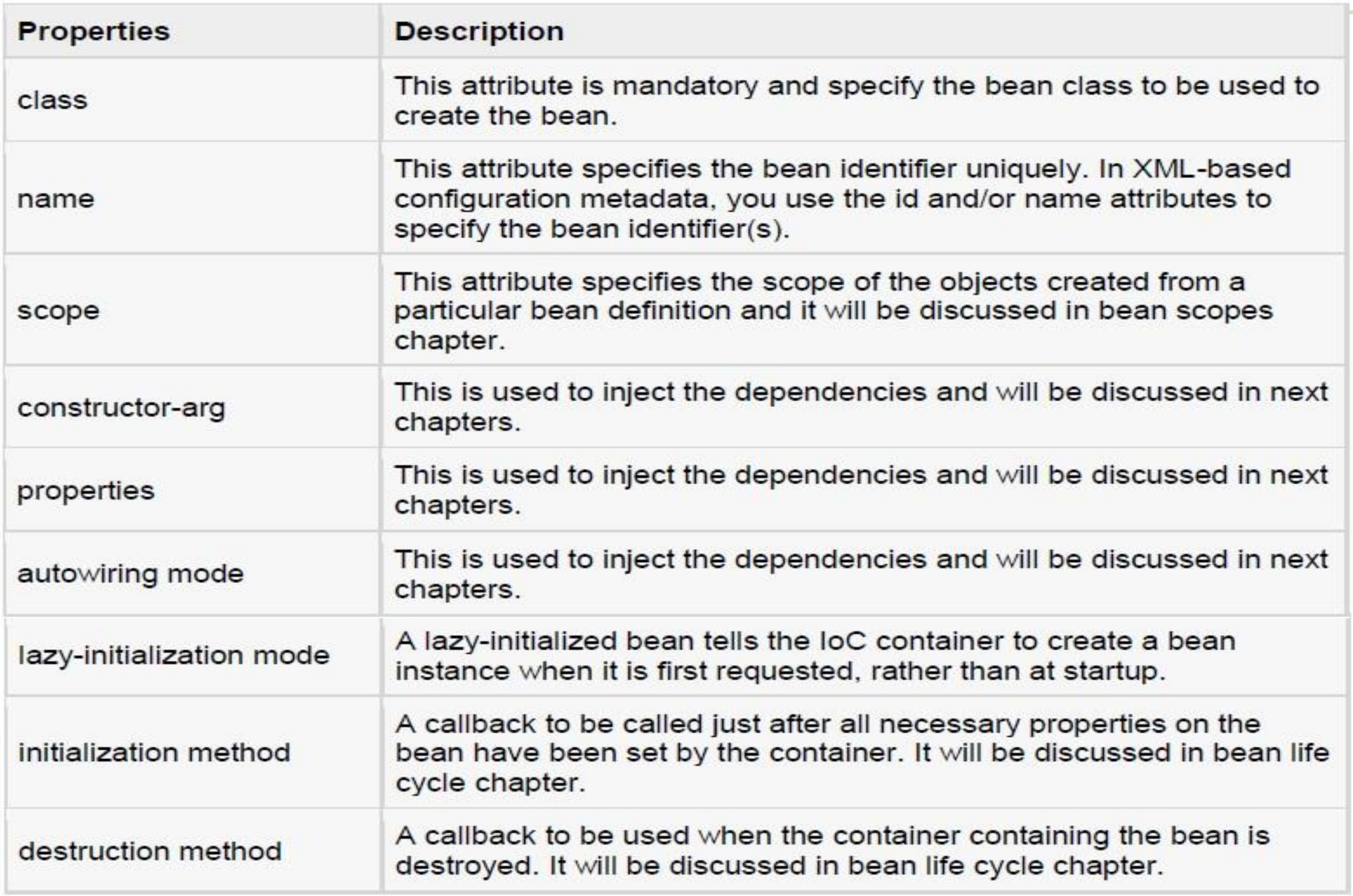
- **FileSystemXmlApplicationContext**
- **ClassPathXmlApplicationContext**
- **WebXmlApplicationContext**

Spring Bean Definition

The bean definition contains the information called configuration metadata which is needed for the container to know the followings:

- How to create a bean
- Bean's lifecycle details
- Bean's dependencies

All the above configuration metadata translates into a set of the following properties that make up each bean definition.



Properties	Description
class	This attribute is mandatory and specify the bean class to be used to create the bean.
name	This attribute specifies the bean identifier uniquely. In XML-based configuration metadata, you use the id and/or name attributes to specify the bean identifier(s).
scope	This attribute specifies the scope of the objects created from a particular bean definition and it will be discussed in bean scopes chapter.
constructor-arg	This is used to inject the dependencies and will be discussed in next chapters.
properties	This is used to inject the dependencies and will be discussed in next chapters.
autowiring mode	This is used to inject the dependencies and will be discussed in next chapters.
lazy-initialization mode	A lazy-initialized bean tells the IoC container to create a bean instance when it is first requested, rather than at startup.
initialization method	A callback to be called just after all necessary properties on the bean have been set by the container. It will be discussed in bean life cycle chapter.
destruction method	A callback to be used when the container containing the bean is destroyed. It will be discussed in bean life cycle chapter.

Spring Bean Definition

Spring Configuration Metadata

Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written. There are following three important methods to provide configuration metadata to the Spring Container:

1. XML based configuration file.
2. Annotation-based configuration
3. Java-based configuration

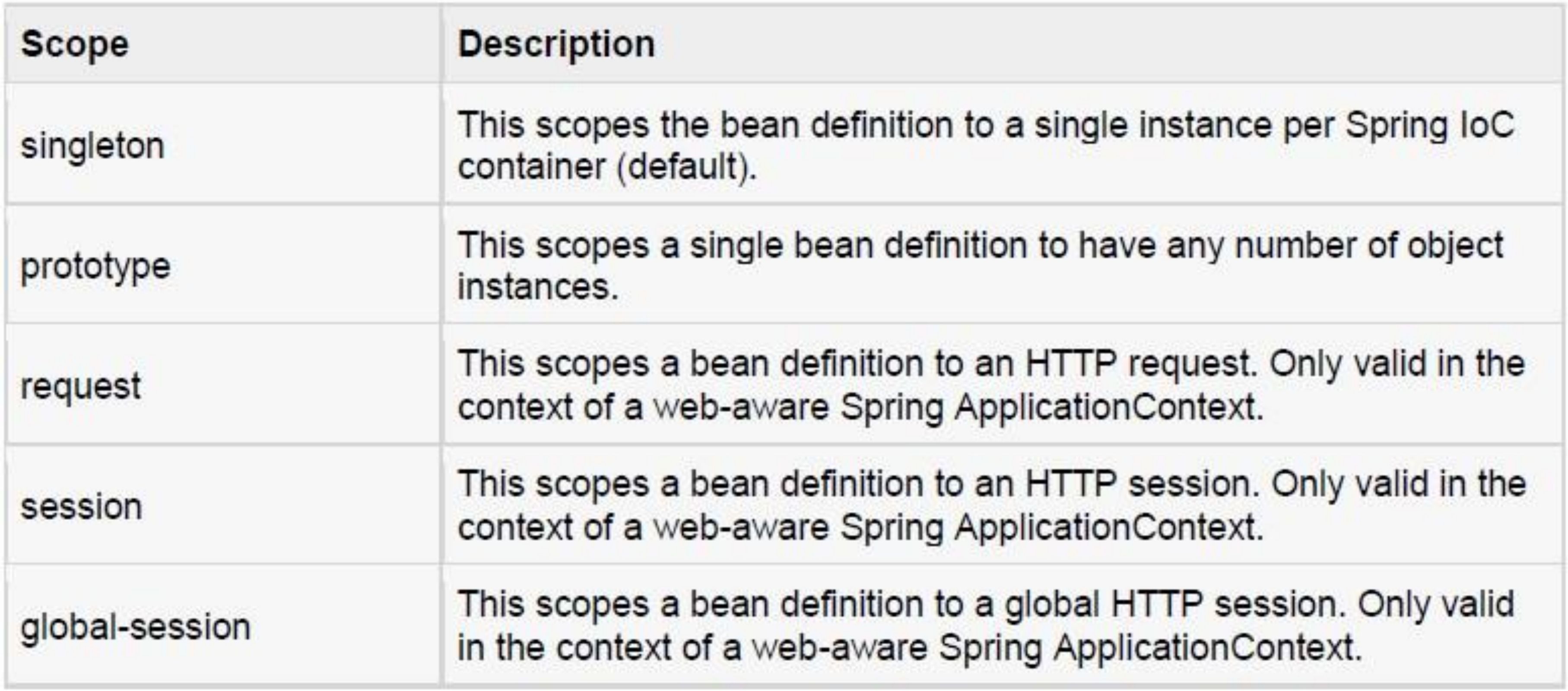
Spring Bean Definition

Beans.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-3.0.xsd">

        <!-- A simple bean definition -->
        <bean id="..." class="...">
        <!-- collaborators and configuration for this bean go here -->
        </bean>

<!-- A bean definition with lazy init set on -->
<bean id="..." class="..." lazy-init="true">
<!-- collaborators and configuration for this bean go here -->
</bean>

        <!-- A bean definition with initialization method -->
        <bean id="..." class="..." init-method="...">
        <!-- collaborators and configuration for this bean go here -->
        </bean>
<!-- A bean definition with destruction method -->
<bean id="..." class="..." destroy-method="...">
<!-- collaborators and configuration for this bean go here -->
</bean>
        <!-- more bean definitions go here -->
</beans>
```

Spring Bean Scopes

The Spring Framework supports following five scopes, three of which are available only if you use a web-aware ApplicationContext.



Scope	Description
singleton	This scopes the bean definition to a single instance per Spring IoC container (default).
prototype	This scopes a single bean definition to have any number of object instances.
request	This scopes a bean definition to an HTTP request. Only valid in the context of a web-aware Spring ApplicationContext.
session	This scopes a bean definition to an HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.
global-session	This scopes a bean definition to a global HTTP session. Only valid in the context of a web-aware Spring ApplicationContext.

Spring Bean Scopes

The singleton scope

If scope is set to singleton, the Spring IoC container creates exactly one instance of the object defined by that bean definition. This single instance is stored in a cache of such singleton beans, and all subsequent requests and references for that named bean return the cached object.

Syntax:

```
<!-- A bean definition with singleton scope -->
<bean id="..." class="..." scope="singleton">
<!-- collaborators and configuration for this bean go here -->
</bean>
```

Spring Bean Scopes

The prototype scope

If scope is set to prototype, the Spring IoC container creates new bean instance of the object every time a request for that specific bean is made. As a rule, use the prototype scope for all state-full beans and the singleton scope for stateless beans.

To define a prototype scope, you can set the scope property to prototype in the bean configuration file, as shown below:

Syntax:

```
<!-- A bean definition with singleton scope -->
<bean id="..." class="..." scope="prototype">
<!-- collaborators and configuration for this bean go here -->
</bean>
```

Spring Bean Life Cycle

- To define setup and teardown for a bean, we simply declare the <bean> with init-method and/ordestroy-method parameters.
- The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation.
- Similarly, destroy-method specifies a method that is called just before a bean is removed from the container.

Spring Bean Life Cycle

Initialization callbacks

The *org.springframework.beans.factory.InitializingBean* interface specifies a single method:

```
void afterPropertiesSet() throws Exception;
```

So you can simply implement above interface and initialization work can be done inside afterPropertiesSet() method as follows:

```
public class ExampleBean implements InitializingBean {
public void afterPropertiesSet() {
// do some initialization work
} }
```

In the case of XML-based configuration metadata, you can use the init-method attribute to specify the name of the method that has a void no-argument signature. For example:

Following is the class definition:

```
<bean id="exampleBean“ class="examples.ExampleBean" init-method="init"/>
```

```
public class ExampleBean {
public void init() {
// do some initialization work
}}
```

Spring Bean Life Cycle

Destruction callbacks

The *org.springframework.beans.factory.DisposableBean interface specifies a single method:*

void destroy() throws Exception;

So you can simply implement above interface and finalization work can be done inside destroy() method as follows:

```
public class ExampleBean implements DisposableBean {
public void destroy() {
// do some destruction work
} }
```

In the case of XML-based configuration metadata, you can use the **destroy-method attribute to specify the name of the method that has a void no-argument signature. For example:**

```
<bean id="exampleBean"  class="examples.ExampleBean" destroy-method="destroy"/>
```

Following is the class definition:

```
public class ExampleBean {
public void destroy() {
// do some destruction work
} }
```

Spring Bean Life Cycle

Default initialization and destroy methods

```
<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-3.0.xsd"
default-init-method="init"
default-destroy-method="destroy">
<bean id="..." class="...">
<!-- collaborators and configuration for this bean go here -->
</bean>
</beans>
```

Spring Bean Post Processors

- The BeanPostProcessor interface defines callback methods that you can implement to provide your own instantiation logic, dependency-resolution logic etc.

- You can also implement some custom logic after the Spring container finishes instantiating, configuring, and initializing a bean by plugging in one or more BeanPostProcessor implementations.

Spring Bean Definition Inheritance

- A bean definition can contain a lot of configuration information, including constructor arguments, property values, and container-specific information such as initialization method, static factory method name, and so on.

- A child bean definition inherits configuration data from a parent definition. The child definition can override some values, or add others, as needed.

- When you use XML-based configuration metadata, you indicate a child bean definition by using the **parent attribute, specifying the parent bean as the value of this attribute.**

Spring Bean Definition Inheritance

Beans.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-3.0.xsd">
<bean id="helloWorld" class=“org.capgemini.HelloWorld">
<property name="message1" value="Hello World!"/>
<property name="message2" value="Hello Second World!"/>
</bean>
<bean id="helloIndia" class="org.capgemini.HelloIndia"
parent="helloWorld">
<property name="message1" value="Hello India!"/>
<property name="message3" value="Namaste India!"/>
</bean>
</beans>
```

Spring Bean Definition Inheritance

Bean Definition Template

You can create a Bean definition template which can be used by other child bean definitions without putting much effort. While defining a Bean Definition Template, you should not specify class attribute and should specify abstract attribute with a value of true as shown below:

- The parent bean cannot be instantiated on its own because it is incomplete, and it is also explicitly marked as abstract.
- When a definition is abstract like this, it is usable only as a pure template bean definition that serves as a parent definition for child definitions.

Spring Bean Definition Inheritance

```
<?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-3.0.xsd">
<bean id="beanTemplate" abstract="true">
<property name="message1" value="Hello World!"/>
<property name="message2" value="Hello Second World!"/>
<property name="message3" value="Namaste India!"/>
</bean>
<bean id="helloIndia" class="org.capgemini.HelloIndia"
parent="beanTemplate">
<property name="message1" value="Hello India!"/>
<property name="message3" value="Namaste India!"/> </bean>
</beans>
```

Dependency Injection

Situation

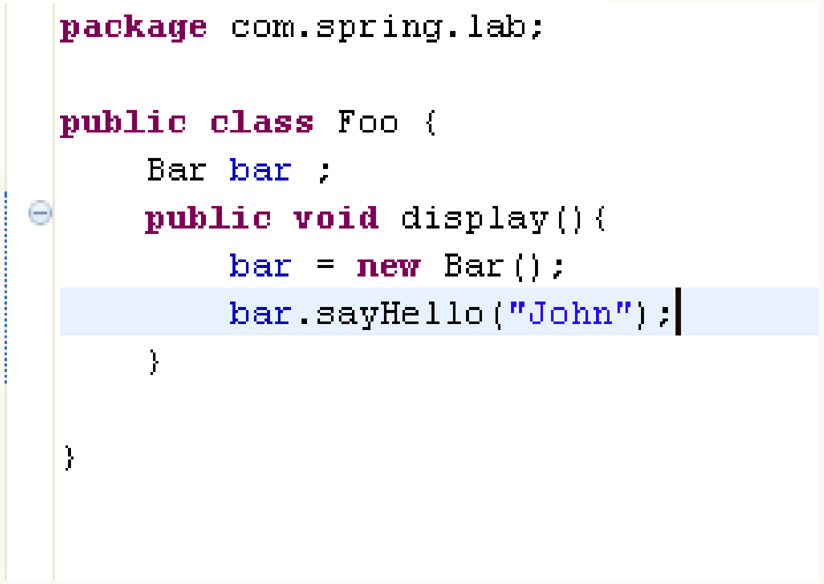
- Assume any user creates an object of class Foo and calls the display method on it. Foo class creates an object of Bar class and calls the sayHello method.

Problem

- Issue with this approach is Foo class not only have to perform its job of executing display method it also needs to maintain the lifecycle of Bar class object which is surely a break in OOP principle!!

Solution

- This is the place where different IOC containers like Spring,PicoContainer or Guice comes into picture and handles the responsibilty of managing the lifecycle of objects and also takes care of object graph.



```
package com.spring.lab;

public class Foo {
    Bar bar ;
    public void display(){
        bar = new Bar();
        bar.sayHello("John");
    }


}
```

Dependency Injection

How to achieve Dependency Injection

- **Constructor-based dependency injection**

  Constructor-based DI is accomplished when the container invokes a class constructor with a number of arguments, each representing a dependency on other class.

- **Setter-based dependency injection.**

  Setter-based DI is accomplished by the container calling setter methods on your beans after invoking a no-argument constructor or no-argument static factory method to instantiate your bean.

Constructor based Injection

Constructor-based **DI**

It is accomplished by the container invoking a constructor with a number of arguments, each representing a dependency. Calling a static factory method with specific arguments to construct the bean is nearly equivalent, and this discussion treats arguments to a constructor and to astatic factory method similarly. The following example shows a class that can only be dependency-injected with constructor injection. Notice that there is nothing *special* about this class, it is a POJO that has no dependencies on container specific interfaces, base classes or annotations.

```
package x.y;

public class Foo {

  public Foo(Bar bar, Baz baz) {
      // ...
  }
}
```

```
<beans>
  <bean id="foo" class="x.y.Foo">
      <constructor-arg ref="bar"/>
      <constructor-arg ref="baz"/>
  </bean>

  <bean id="bar" class="x.y.Bar"/>
  <bean id="baz" class="x.y.Baz"/>

</beans>
```

Other variations of constructor based DI

Constructor argument type matching

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg type="int" value="7500000"/>
<constructor-arg type="java.lang.String" value="42"/>
```

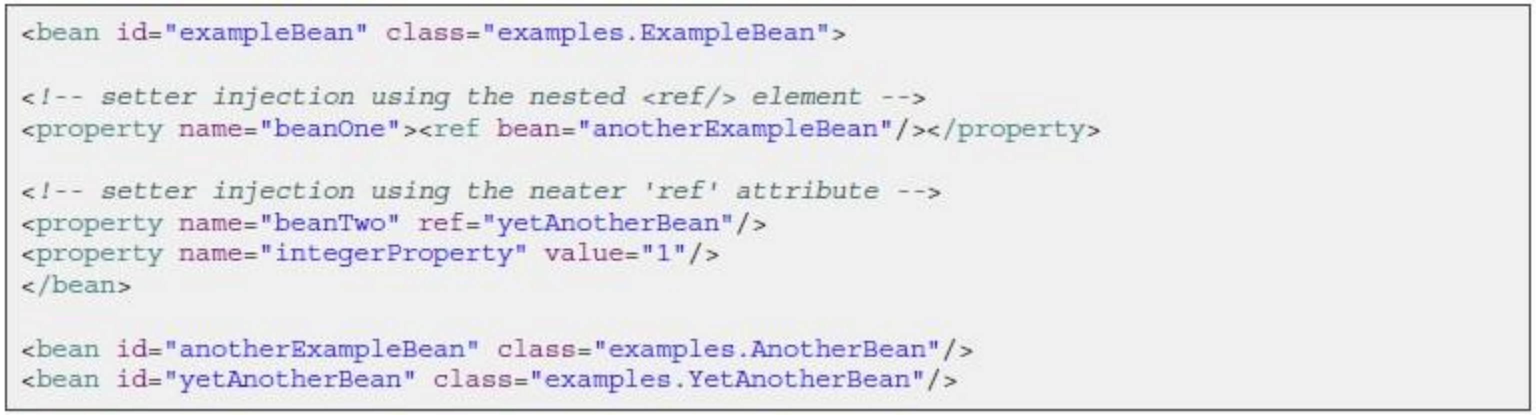
Constructor argument index

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg index="0" value="7500000"/>
<constructor-arg index="1" value="42"/>
</bean>
```

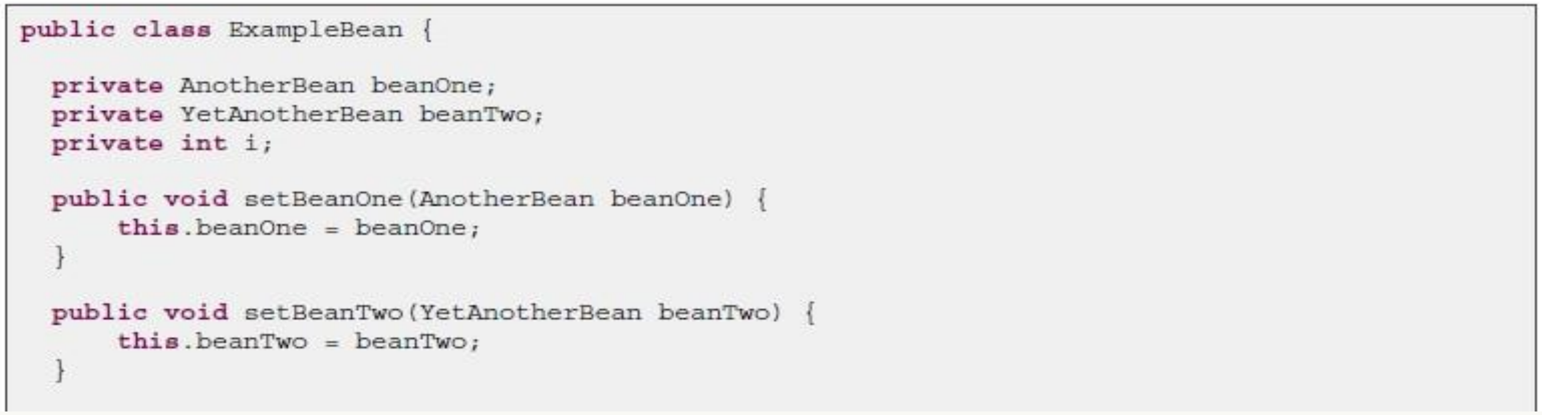
Constructor argument name

```
<bean id="exampleBean" class="examples.ExampleBean">
<constructor-arg name="years" value="7500000"/>
<constructor-arg name="ultimateanswer" value="42"/>
</bean>
```

Setter based Injection

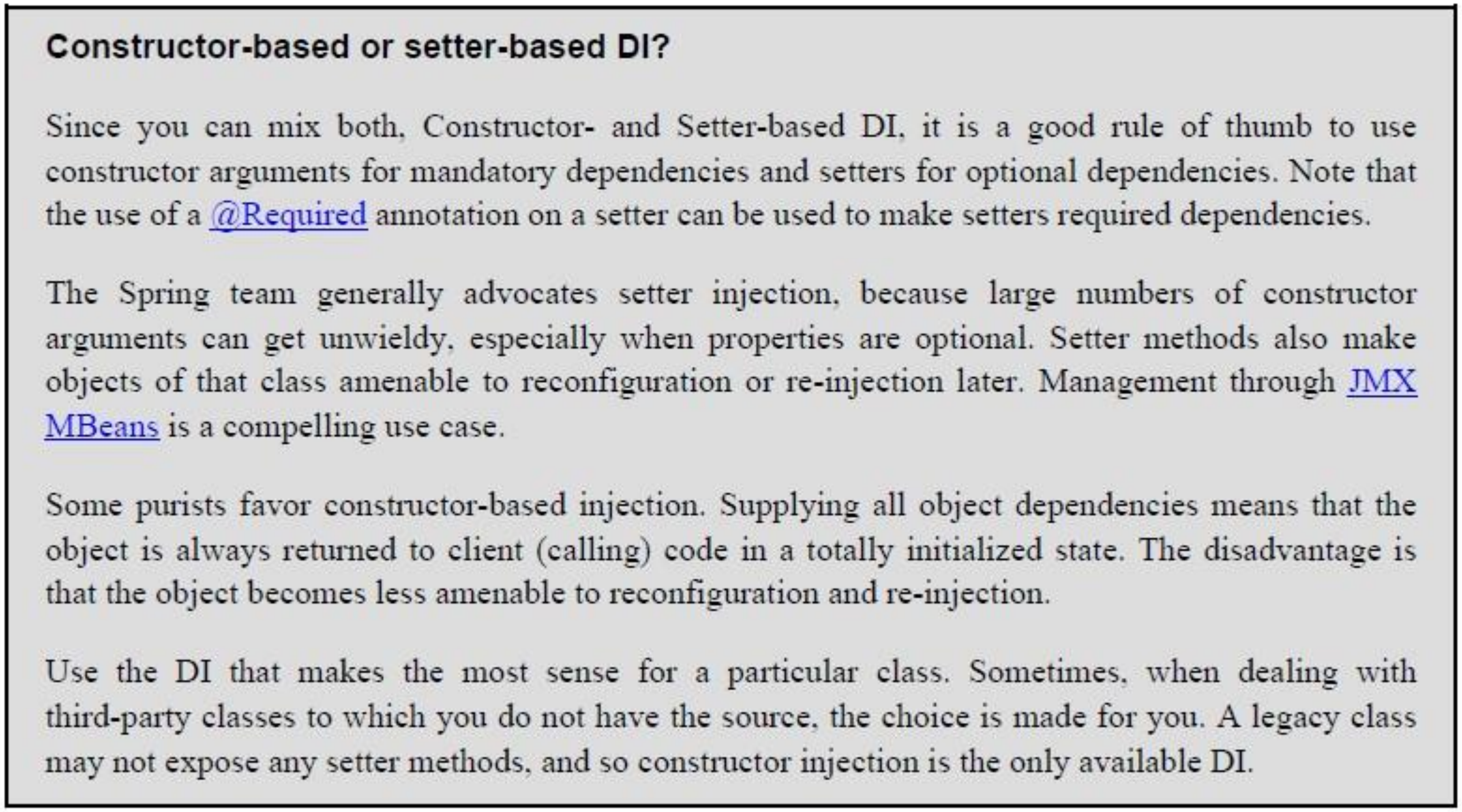


```xml
<bean id="exampleBean" class="examples.ExampleBean">

<!-- setter injection using the nested <ref/> element -->
<property name="beanOne"><ref bean="anotherExampleBean"/></property>

<!-- setter injection using the neater 'ref' attribute -->
<property name="beanTwo" ref="yetAnotherBean"/>
<property name="integerProperty" value="1"/>
</bean>

<bean id="anotherExampleBean" class="examples.AnotherBean"/>
<bean id="yetAnotherBean" class="examples.YetAnotherBean"/>
```



```java
public class ExampleBean {

  private AnotherBean beanOne;
  private YetAnotherBean beanTwo;
  private int i;

  public void setBeanOne(AnotherBean beanOne) {
      this.beanOne = beanOne;
  }

  public void setBeanTwo(YetAnotherBean beanTwo) {
      this.beanTwo = beanTwo;
  }
```

Setter/Constructor based Injection



Constructor-based or setter-based DI?

Since you can mix both, Constructor- and Setter-based DI, it is a good rule of thumb to use constructor arguments for mandatory dependencies and setters for optional dependencies. Note that the use of a @Required annotation on a setter can be used to make setters required dependencies.

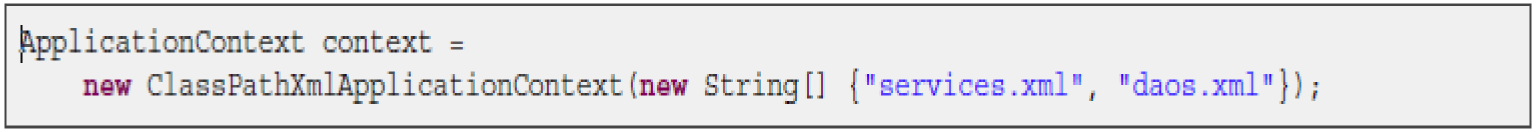
The Spring team generally advocates setter injection, because large numbers of constructor arguments can get unwieldy, especially when properties are optional. Setter methods also make objects of that class amenable to reconfiguration or re-injection later. Management through JMX MBeans is a compelling use case.

Some purists favor constructor-based injection. Supplying all object dependencies means that the object is always returned to client (calling) code in a totally initialized state. The disadvantage is that the object becomes less amenable to reconfiguration and re-injection.

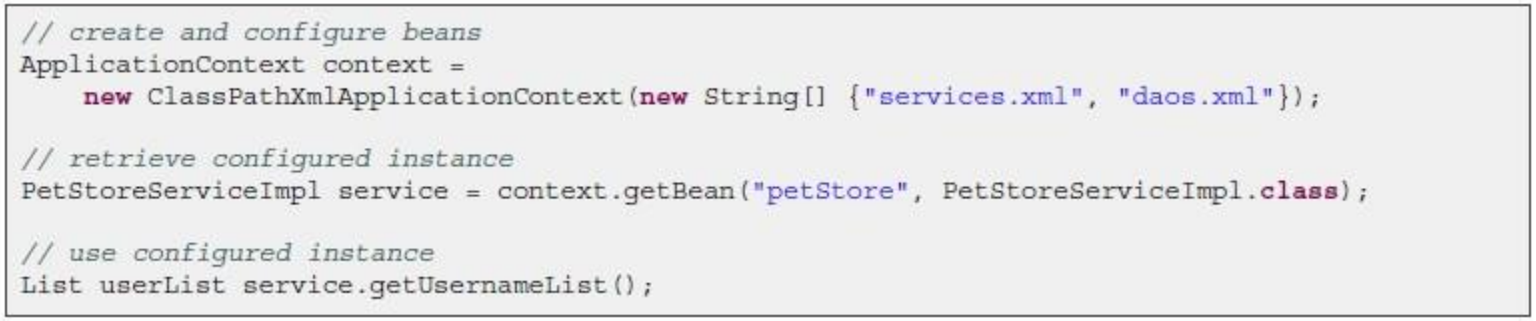
Use the DI that makes the most sense for a particular class. Sometimes, when dealing with third-party classes to which you do not have the source, the choice is made for you. A legacy class may not expose any setter methods, and so constructor injection is the only available DI.

Instantiating a container

Instantiating a Spring IoC container is straightforward. The location path or paths supplied to an ApplicationContext constructor are actually resource strings that allow the container to load configuration metadata from a variety of external resources such as the local file system, from the Java CLASSPATH, and so on.



```
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});
```



```
// create and configure beans
ApplicationContext context =
    new ClassPathXmlApplicationContext(new String[] {"services.xml", "daos.xml"});

// retrieve configured instance
PetStoreServiceImpl service = context.getBean("petStore", PetStoreServiceImpl.class);

// use configured instance
List userList service.getUsernameList();
```

Dependency resolution process

The container performs bean dependency resolution as follows:

1. The ApplicationContext is created and initialized with configuration metadata that describes all the beans. Configuration metadata can be specified via XML, Java code or annotations.

2. For each bean, its dependencies are expressed in the form of properties, constructor arguments, or arguments to the static-factory method if you are using that instead of a normal constructor. These dependencies are provided to the bean, *when the bean is actually created*.

3. Each property or constructor argument is an actual definition of the value to set, or a reference to another bean in the container.

4. Each property or constructor argument which is a value is converted from its specified format to the actual type of that property or constructor argument. By default Spring can convert a value supplied in string format to all built-in types, such as int, long, String, boolean, etc.

The Spring container validates the configuration of each bean as the container is created, including the validation of whether bean reference properties refer to valid beans. However, the bean properties themselves are not set until the bean *is actually created*. Beans that are singleton-scoped and set to be pre-instantiated (the default) are created when the container is created. Scopes are defined in “Bean scopes” Otherwise, the bean is created only when it is requested. Creation of a bean potentially causes a graph of beans to be created, as the bean's dependencies and its dependencies' dependencies (and so on) are created and assigned.

Quiz

Questions

1. Define BeanFactory and ApplicationContext
2. Difference between Singleton and prototype bean
3. Explain Bean Life Cycle
4. Explain DI and how will you achieve the DI in Spring?
5. Discuss which DI is better constructor or setter.

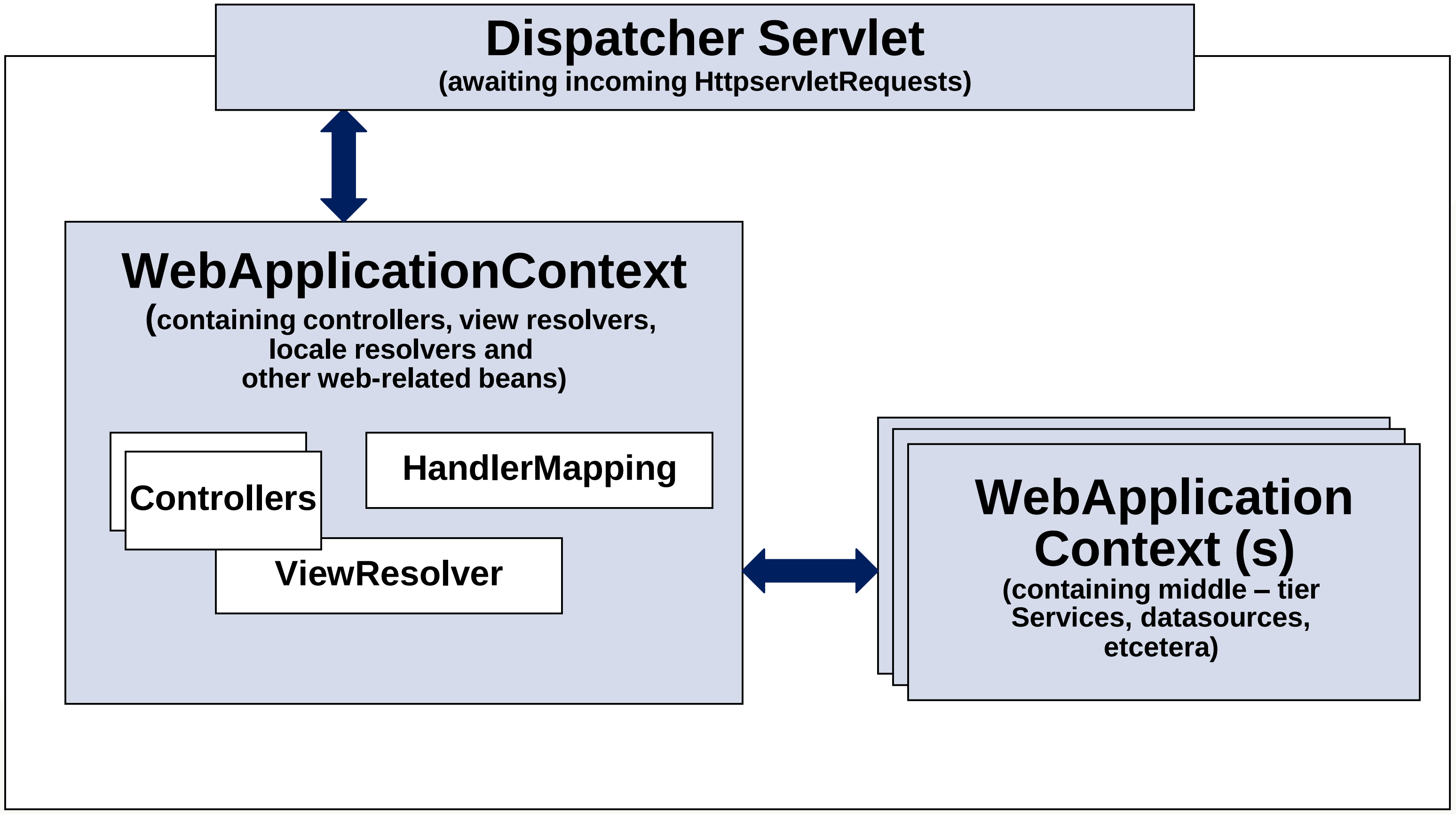
Spring3 Web MVC Framework

- The **Model encapsulates the application data and in general they will consist of POJO.**
- The **View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.**
- The **Controller is responsible for processing user requests and building appropriate model and passes it to the view for rendering.**

Features of Spring web MVC

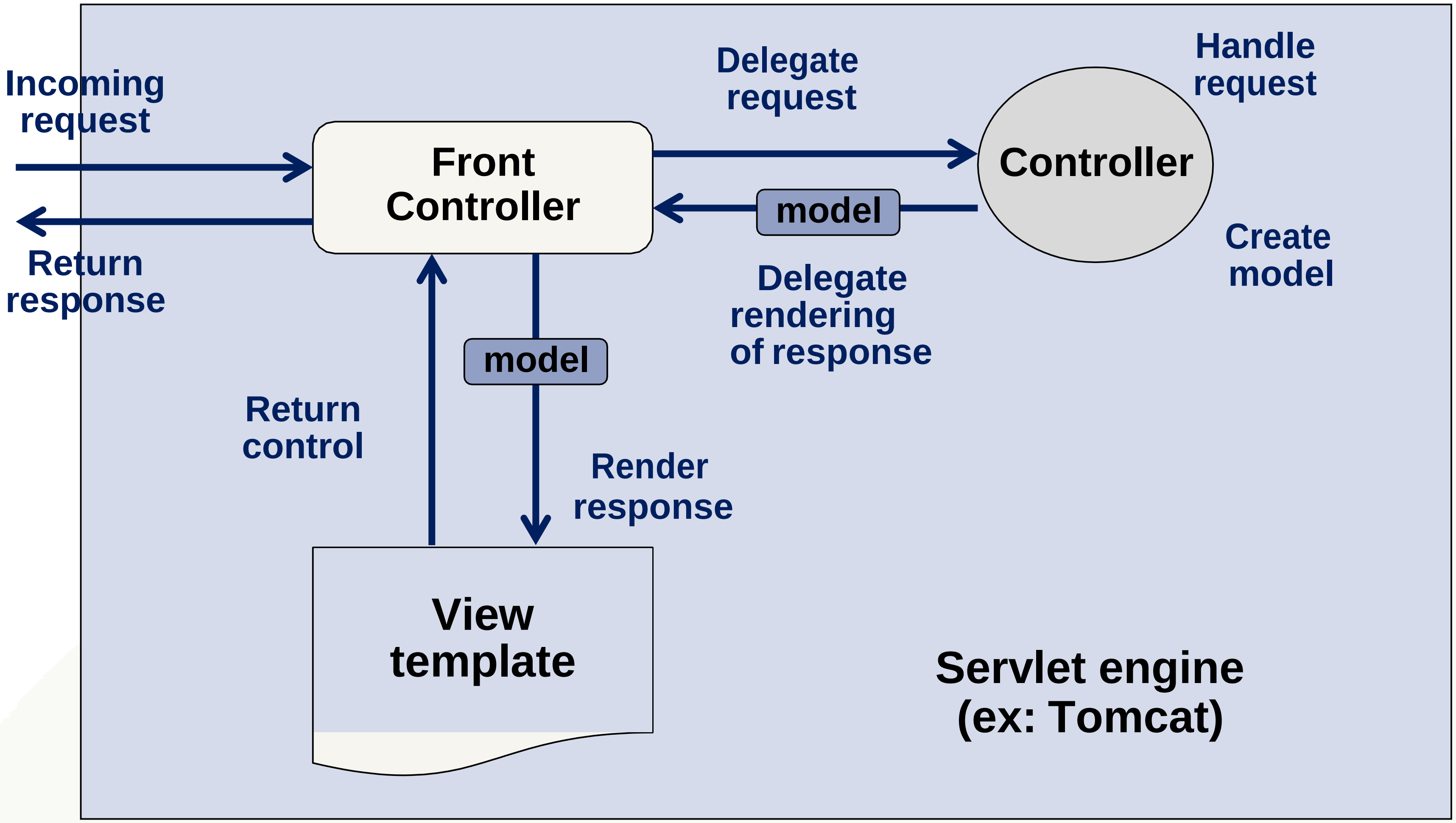
- Clear separation of roles.
- Powerful and straightforward configuration of both framework and application classes as JavaBeans.
- Adaptability, non-intrusiveness, and flexibility.
- Reusable business code, no need for duplication.
- Customizable binding and validation.
- Customizable handler mapping and view resolution.
- Flexible model transfer.
- Customizable locale and theme resolution, support for JSPs with or without Spring tag library, support for JSTL, support for Velocity without the need for extra bridges, and so on.
- A simple yet powerful JSP tag library known as the Spring tag library that provides support for features such as data binding and themes.
- A JSP form tag library, introduced in Spring 2.0, that makes writing forms in JSP pages much easier.
- Beans whose lifecycle is scoped to the current HTTP request or HTTP Session.

The Dispatcher Servlet


Dispatcher Servlet
(awaiting incoming HttpservletRequests)
WebApplicationContext
(containing controllers, view resolvers, locale resolvers and other web-related beans)
Controllers
HandlerMapping
ViewResolver
WebApplication Context (s)
(containing middle – tier Services, datasources, etcetera)

Context hierarchy in Spring Web MVC

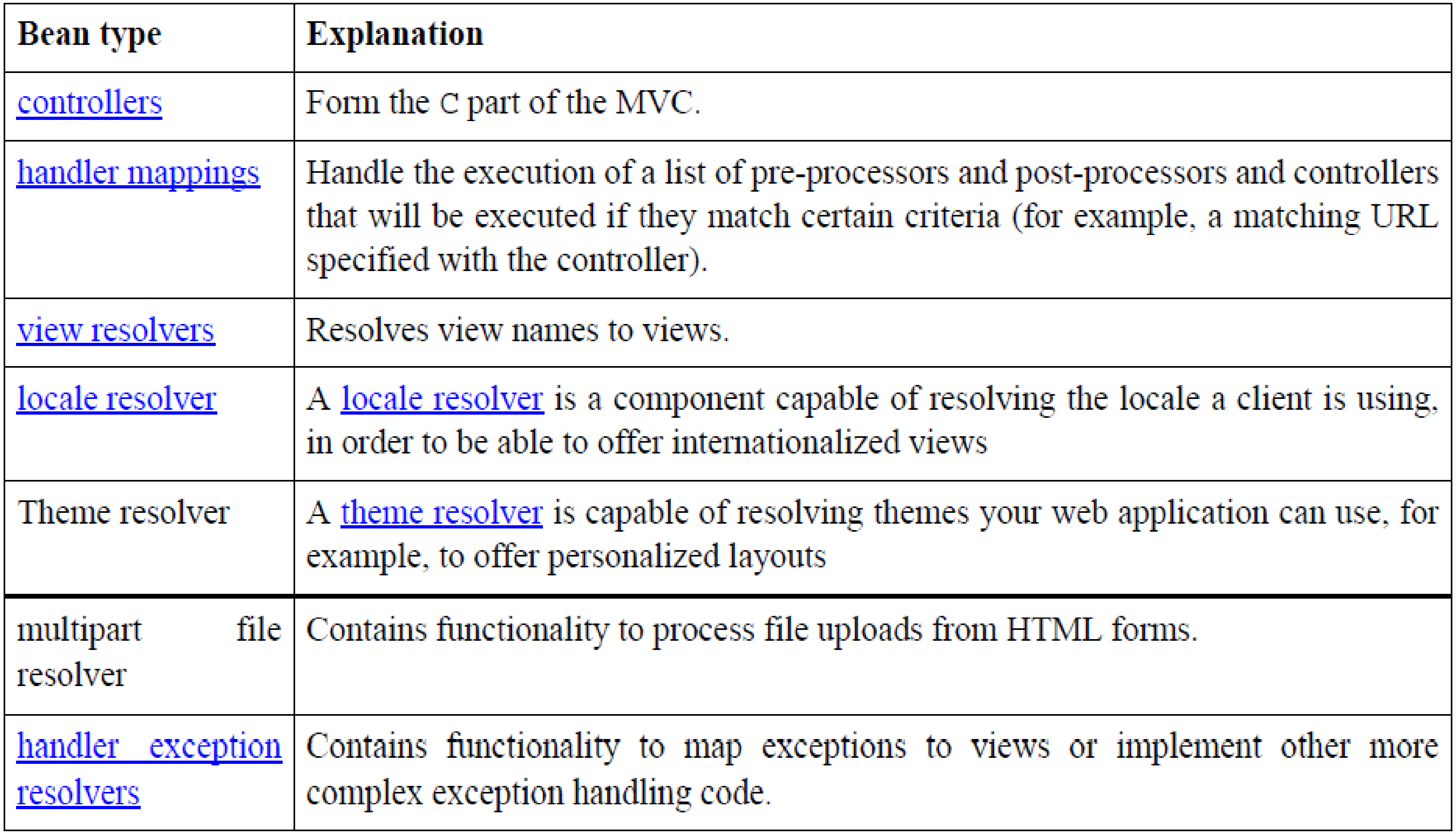
The Dispatcher Servlet (Continue…)



The requesting processing workflow in Spring Web MVC (high level)

The Dispatcher Servlet (Continue…)

Special beans in the WebApplicationContext



Bean type	Explanation
controllers	Form the C part of the MVC.
handler mappings	Handle the execution of a list of pre-processors and post-processors and controllers that will be executed if they match certain criteria (for example, a matching URL specified with the controller).
view resolvers	Resolves view names to views.
locale resolver	A locale resolver is a component capable of resolving the locale a client is using, in order to be able to offer internationalized views
Theme resolver	A theme resolver is capable of resolving themes your web application can use, for example, to offer personalized layouts
multipart file resolver	Contains functionality to process file uploads from HTML forms.
handler exception resolvers	Contains functionality to map exceptions to views or implement other more complex exception handling code.

The Dispatcher Servlet (Continue…)

- Following is the sequence of events corresponding to an incoming HTTP request to *DispatcherServlet:*

1. After recieving an HTTP request, *DispatcherServlet* consults the *HandlerMapping* to call the appropriate Controller.
2. The *Controller* takes the request and calls the appropriate *service methods based on used GET or POST method.* The service method will set model data based on defined business logic and returns view name to the DispatcherServlet.
3. The *DispatcherServlet* will take help from *ViewResolver* to pickup the defined view for the request.
4. Once view is finalized, The *DispatcherServlet passes the model data* to the view which is finally rendered on the browser.

Required Configuration

```
<?xml version="1.0" encoding="UTF-8"?>
<web-app xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns="http://java.sun.com/xml/ns/javaee"
xmlns:web="http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd"
xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
http://java.sun.com/xml/ns/javaee/web-app_2_5.xsd" id="WebApp_ID" version="2.5">
  <display-name>HelloWeb</display-name>
  <welcome-file-list>
      <welcome-file>index.jsp</welcome-file>
  </welcome-file-list>

  <servlet>
  <servlet-name>springmvc</servlet-name>
  <servlet-class>
org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
  </servlet>

  <servlet-mapping>
  <servlet-name>springmvc</servlet-name>
  <url-pattern>*.html</url-pattern>
  </servlet-mapping>
</web-app>
```

Resolving the Request

After you set up a DispatcherServlet, and a request comes in for that specific DispatcherServlet, the DispatcherServlet starts processing the request as follows:

- The WebApplicationContext is searched for and bound in the request as an attribute that the controller and other elements in the process can use. It is bound by default under the key DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE.

- The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request (rendering the view, preparing data, and so on). If you do not need locale resolving, you do not need it.

- The theme resolver is bound to the request to let elements such as views determine which theme to use. If you do not use themes, you can ignore it.

- If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process. (See the section called “Using the MultipartResolver” for further information about multipart handling).

- An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering.

- If a model is returned, the view is rendered. If no model is returned, (may be due to a preprocessor or postprocessor intercepting the request, perhaps for security reasons), no view is rendered, because the request could already have been fulfilled.

Required Configuration (Continue…)

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:context="http://www.springframework.org/schema/context"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
                            http://www.springframework.org/schema/context
http://www.springframework.org/schema/context/spring-context-3.0.xsd">

<!-- It will load all the components from the package org.capgemini -->
<context:component-scan base-package="org.capgemini"/>

<bean id="viewResolver"
class="org.springframework.web.servlet.view.UrlBasedViewResolver">
<property name="viewClass"
value="org.springframework.web.servlet.view.JstlView"/>
<property name="prefix" value="/WEB-INF/jsp/"/>
<property name="suffix" value=".jsp"/>
</bean>

</beans>
```

Defining the controller

```
@Controller
public class HelloWorldController {

        @RequestMapping("/hello")
        public ModelAndView sayHello()
        {
                String msg="Spring3 MVC, Hello World!!!!!";
                return new ModelAndView("hello", "message", msg);
        } }
```

(or)

```
@Controller
public class HelloWorldController {

        @RequestMapping("/hello")
        public ModelAndView sayHello()
        {
                ModelAndView mv=new ModelAndView();
                mv.setViewName(“hello”);
                mv.addObject("message", "Spring3 MVC, Hello World!!!!!“);
                return mv;
        } }
```

Defining the controller (continue…)

- The **@Controller** annotation indicates that a particular class serves the role of a *controller.*
- Spring does not require you to extend any controller base class or reference the Servlet API.
- The dispatcher scans such annotated classes for mapped methods and detects **@RequestMapping** annotations

@RequestMapping

```
@Controller
@RequestMapping("/appointments")
public class AppointmentsController {
private final AppointmentBook appointmentBook;
@Autowired
public AppointmentsController(AppointmentBook appointmentBook) {
this.appointmentBook = appointmentBook;
}
@RequestMapping(method = RequestMethod.GET)
public Map<String, Appointment> get() {
return appointmentBook.getAppointmentsForToday();
}
@RequestMapping(value="/{day}", method = RequestMethod.GET)
public Map<String, Appointment> getForDay(@PathVariable
   @DateTimeFormat(iso=ISO.DATE) Date day, Model model) return
   appointmentBook.getAppointmentsForDay(day);
}
```

(continue…)

@RequestMapping (continue…)

```
@RequestMapping(value="/new", method = RequestMethod.GET)
public AppointmentForm getNewForm() {
return new AppointmentForm();
}
@RequestMapping(method = RequestMethod.POST)
public String add(@Valid AppointmentForm appointment, BindingResult result) {
if (result.hasErrors()) {
return "appointments/new";
}
appointmentBook.addAppointment(appointment);
return "redirect:/appointments";
}
}
```

@RequestMapping (continue…)

- The first usage of @RequestMapping is on the type (class) level, which indicates that all handling methods on this controller are relative to the /appointments path.
- The get() method has a further @RequestMapping refinement: it only accepts GET requests, meaning that an HTTP GET for /appointments invokes this method.
- The post() has a similar refinement, and the getNewForm() combines the definition of HTTP method and path into one, so that GET requests for appointments/new are handled by that method.
- The getForDay() method shows another usage of @RequestMapping: URI templates.
- Use the @PathVariable method parameter annotation to indicate that a method parameter should be
- bound to the value of a URI template variable.

Creating JSP views

```
<html>
<head>
<title>Hello Spring MVC</title>
</head>
<body>
<h2>${message}</h2>
</body>
</html>
```

Here ${message} is the attribute which we have setup inside the Controller. You can have multiple attributes to be displayed inside your view.

Resolving views

The two interfaces that are important to the way Spring handles views are ViewResolver and View.

- The ***ViewResolver*** provides a mapping between view names and actual views.
- The ***View*** interface addresses the preparation of the request and hands the request over to one of the view technologies.

Resolving views (continue…)

ViewResolver	Description
AbstractCachingViewResolver	Abstract view resolver that caches views. Often views need preparation before they can be used; extending this view resolver provides caching.
XmlViewResolver	Implementation of ViewResolver that accepts a configuration file written in XML with the same DTD as Spring's XML bean factories. The default configuration file is /WEB-INF/views.xml.
ResourceBundleViewResolver	Implementation of ViewResolver that uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is views.properties.
ContentNegotiatingViewResolver	Implementation of the ViewResolver interface that resolves a view based on the request file name or Accept header.

Resolving views (continue…)

ViewResolver	**Description**
UrlBasedViewResolver	Simple implementation of the ViewResolver interface that effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.
InternalResourceViewResolver	Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..).
VelocityViewResolver / FreeMarkerViewResolver	Convenient subclass of UrlBasedViewResolver that supports VelocityView (in effect, Velocity templates) or FreeMarkerView ,respectively, and custom subclasses of them.

Resolving views (continue…)

Changing ViewResolvers

```
<bean id="jspViewResolver"
   class="org.springframework.web.servlet.view.InternalResourceViewResolver">
<property name="viewClass"
   value="org.springframework.web.servlet.view.JstlView"/>
<property name="prefix" value="/WEB-INF/jsp/"/>
<property name="suffix" value=".jsp"/>
</bean>
<bean id="excelViewResolver"
   class="org.springframework.web.servlet.view.XmlViewResolver">
<property name="order" value="1"/>
<property name="location" value="/WEB-INF/views.xml"/>
</bean>
<!-- in views.xml -->
<beans>
<bean name="report" class="org.springframework.example.ReportExcelView"/>
</beans>
```

Resolving views (continue…)

Redirecting to views

The RedirectView issues an HttpServletResponse.sendRedirect() call that returns to the client browser as an HTTP redirect. All model attributes are exposed as HTTP query parameters. This means that the model must contain only objects (generally Strings or objects converted to a String representation), which can be readily converted to a textual HTTP query parameter.

The redirect: prefix

The forward: prefix

Resolving views (continue…)

The redirect: prefix

The special redirect: prefix allows you to accomplish this. If a view name is returned that has the prefix redirect:, the UrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

The forward: prefix

It is also possible to use a special forward: prefix for view names that are ultimately resolved by UrlBasedViewResolver and subclasses. This creates an InternalResourceView (which ultimately does a RequestDispatcher.forward()) around the rest of the view name, which is considered a URL.

Handling Exception

Exception	HTTP Status Code
ConversionNotSupportedException	500 (Internal Server Error)
HttpMediaTypeNotAcceptableException	406 (Not Acceptable)
HttpMediaTypeNotSupportedException	415 (Unsupported Media Type)
HttpMessageNotReadableException	400 (Bad Request)
HttpMessageNotWritableException	500 (Internal Server Error)
HttpRequestMethodNotSupportedException	405(Method Not Allowed)
MissingServletRequestParameterException	400(Bad Request)
NoSuchRequestHandlingMethodException	404 (Not Found)
TypeMismatchException	400 (Bad Request)

Handling Exception (continue…)

@ExceptionHandler

```
@Controller
public class SimpleController {
// other controller method omitted
@ExceptionHandler(IOException.class)
public String handleIOException(IOException ex, HttpServletRequest request) {
return ClassUtils.getShortName(ex.getClass());
}
}
```

Questions

1. Compare DispatcherServlet and FrontController.
2. Define the different way of handling @RequestMapping
3. How do you enable context component scan?
4. What is the purpose of ModelView class?
5. Discuss the ViewResolver in spring.
6. How will you handle the exception in Spring MVC. Give one example also.
7. Try to write a code to deal with the form input data. Accept the form details from one page and display that into the next page.

   Note:

   Design a simple form to accept the Employee details(empno,empname,salary,date of joining)

Spring JDBC Framework

- While working with database using plain old JDBC, it becomes cumbersome to write unnecessary code to handle exceptions, opening and closing database connections etc.
- Spring JDBC Framework takes care of all the low-level details starting from opening the connection, prepare and execute the SQL statement, process exceptions, handle transactions and finally close the connection.
- We can just define connection parameters and specify the SQL statement to be executed and do the required work for each iteration while fetching data from the database.
- Spring JDBC provides several approaches and correspondingly different classes to interface with the database.
- One of the most popular approach which makes use of **JdbcTemplateclass of the framework.**
- This is the central framework class that manages all the database communication and exception handling.

Spring JDBC Framework

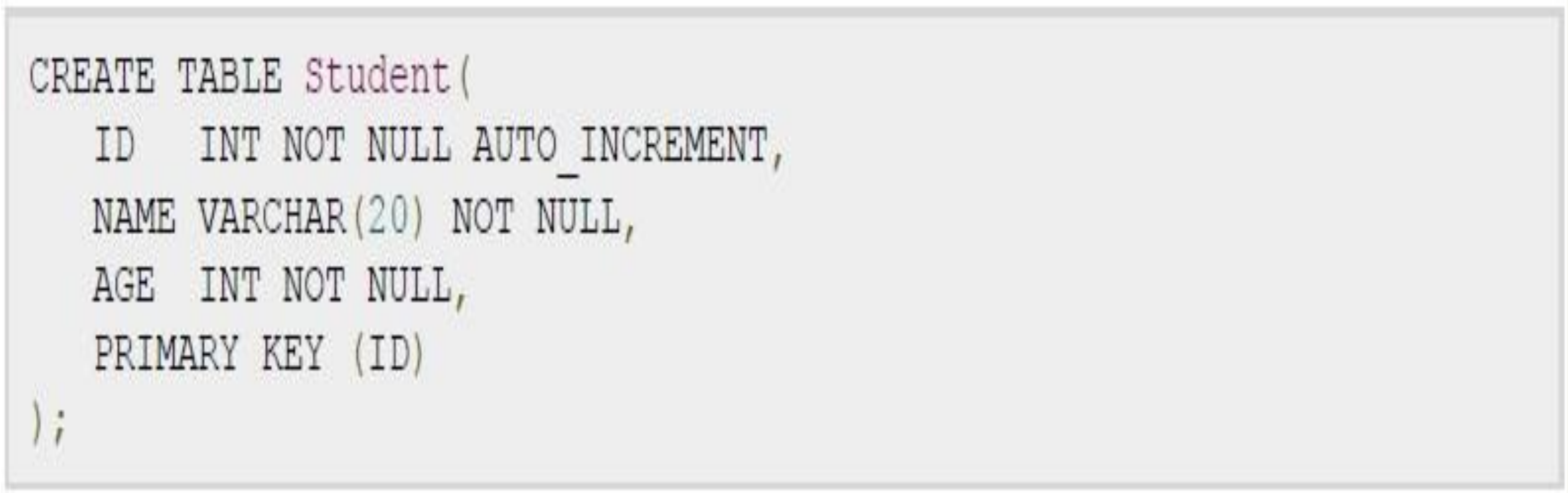
JdbcTemplate Class

- The JdbcTemplate class executes SQL queries, update statements and stored procedure calls, performs iteration over ResultSets and extraction of returned parameter values.
- It also catches JDBC exceptions and translates them to the generic, more informative, exception hierarchy defined in the org.springframework.dao package.
- Instances of the *JdbcTemplate class are threadsafe once configured. So you can configure a single instance of a JdbcTemplate and then safely inject this shared reference into multiple DAOs.*
- A common practice when using the JdbcTemplate class is to configure a *DataSource in your Spring configuration file, and then dependency-inject that shared DataSource bean into your DAO classes, and the JdbcTemplate is created in the setter for the DataSource.*

Spring JDBC Framework

Configuring Data Source

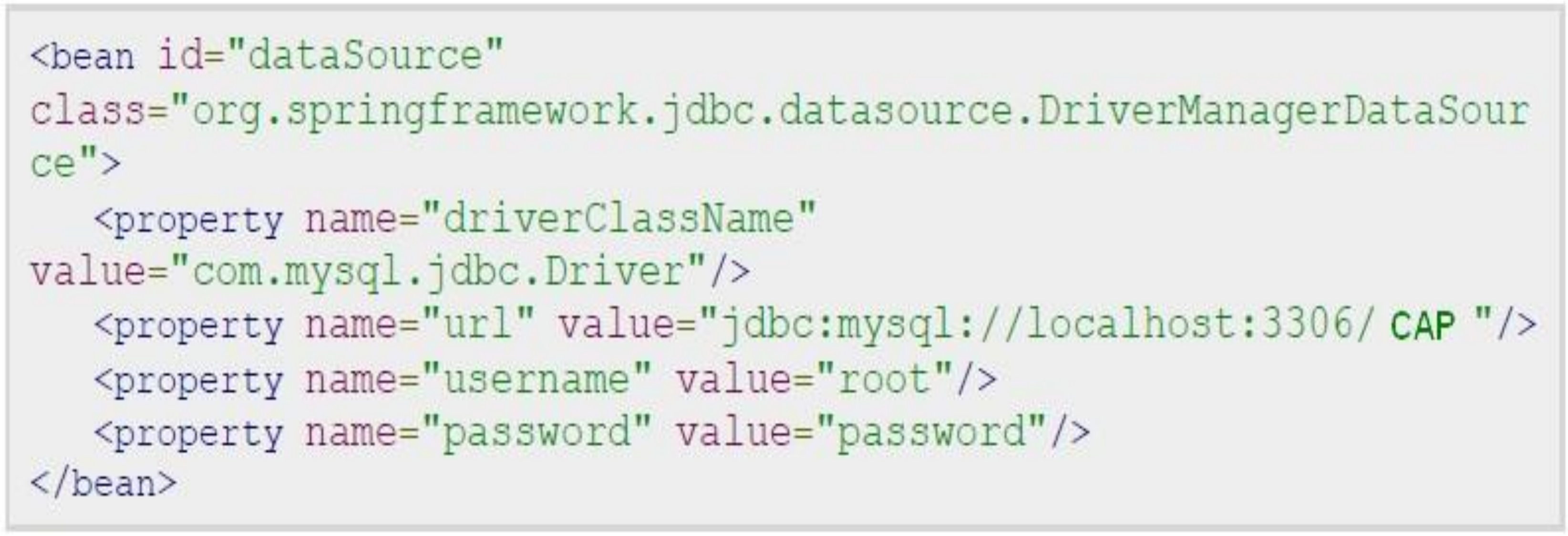
Let us create a database table **Student in our database CAP.**



```
CREATE TABLE Student(
   ID   INT NOT NULL AUTO_INCREMENT,
   NAME VARCHAR(20) NOT NULL,
   AGE  INT NOT NULL,
   PRIMARY KEY (ID)
);
```

Spring JDBC Framework

we need to supply a DataSource to the JdbcTemplate so it can configure itself to get database access. You can configure the DataSource in the XML file with a piece of code as shown below:



```
<bean id="dataSource"
class="org.springframework.jdbc.datasource.DriverManagerDataSour
ce">
   <property name="driverClassName"
value="com.mysql.jdbc.Driver"/>
   <property name="url" value="jdbc:mysql://localhost:3306/ CAP "/>
   <property name="username" value="root"/>
   <property name="password" value="password"/>
</bean>
```

Spring JDBC Framework

Data Access Object (DAO)

- DAO stands for data access object which is commonly used for database interaction.
- DAOs exist to provide a means to read and write data to the database and they should expose this functionality through an interface by which the rest of the application will access them.
- The Data Access Object (DAO) support in Spring makes it easy to work with data access technologies like JDBC, Hibernate, JPA or JDO in a consistent way.

Spring JDBC Framework

Executing SQL statements

Let us see how we can perform CRUD (Create, Read, Update and Delete) operation on database tables using SQL and jdbcTemplate object.

Querying for an integer:

```
String SQL = "select count(*) from Student";
int rowCount = jdbcTemplateObject.queryForInt( SQL );
```

Spring JDBC Framework

Querying for a long:

```
String SQL = "select count(*) from Student";
long rowCount = jdbcTemplateObject.queryForLong( SQL );
```

A simple query using a bind variable:

```
String SQL = "select age from Student where id = ?";
int age = jdbcTemplateObject.queryForInt(SQL, new Object[]{10});
```

Querying for a String:

```
String SQL = "select name from Student where id = ?";
String name = jdbcTemplateObject.queryForObject(SQL, new
  Object[]{10}, String.class);
```

Spring JDBC Framework

Querying and returning an object:

```
String SQL = "select * from Student where id = ?";
Student student = jdbcTemplateObject.queryForObject(SQL,
new Object[]{10}, new StudentMapper());
public class StudentMapper implements RowMapper<Student> {
public Student mapRow(ResultSet rs, int rowNum) throws SQLException
  {
Student student = new Student();
student.setID(rs.getInt("id"));
student.setName(rs.getString("name"));
student.setAge(rs.getInt("age"));
return student;
}
}
```

Spring JDBC Framework

Querying and returning multiple objects:

```
String SQL = "select * from Student";
List<Student> students = jdbcTemplateObject.query(SQL,
new StudentMapper());
public class StudentMapper implements RowMapper<Student> {
public Student mapRow(ResultSet rs, int rowNum) throws SQLException
  {
Student student = new Student();
student.setID(rs.getInt("id"));
student.setName(rs.getString("name"));
student.setAge(rs.getInt("age"));
return student;
}
}
```

Spring JDBC Framework

Inserting a row into the table:

```
String SQL = "insert into Student (name, age) values (?, ?)";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 11} );
```

Updating a row into the table:

```
String SQL = "update Student set name = ? where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{"Zara", 10} );
```

Deleting a row from the table:

```
String SQL = "delete Student where id = ?";
jdbcTemplateObject.update( SQL, new Object[]{20} );
```

Spring JDBC Framework

Executing DDL Statements

You can use the **execute(..) method from *jdbcTemplate to execute any SQL statements or DDL statements. Following is an example to use CREATE statement to create a table:***

```
String SQL = "CREATE TABLE Student( " +
"ID INT NOT NULL AUTO_INCREMENT, " +
"NAME VARCHAR(20) NOT NULL, " +
"AGE INT NOT NULL, " +
"PRIMARY KEY (ID)); "
jdbcTemplateObject.execute( SQL );
```

Spring JDBC Framework

SQL Stored Procedure in Spring

- The **SimpleJdbcCall class can be used to call a stored procedure with IN and OUT parameters. You can use this approach while working with either of the RDBMS like Apache Derby, DB2, MySQL, Microsoft SQL Server, Oracle, and Sybase.**
- To understand the approach let us take our Student table which can be created in MySQL TEST database with the following DDL:

```
CREATE TABLE Student(
ID INT NOT NULL AUTO_INCREMENT,
NAME VARCHAR(20) NOT NULL,
AGE INT NOT NULL,
PRIMARY KEY (ID)
);
```

Spring JDBC Framework

Consider the following code block. It contains both IN and OUT parameters.

```
DELIMITER $$
DROP PROCEDURE IF EXISTS `CAP`.`getRecord` $$
CREATE PROCEDURE `CAP`.`getRecord` (
IN in_id INTEGER,
OUT out_name VARCHAR(20),
OUT out_age INTEGER)
BEGIN
SELECT name, age
INTO out_name, out_age
FROM Student where id = in_id;
END $$
DELIMITER ;
```

Spring JDBC Framework

Following is the implementation class file **StudentJDBCTemplate.java for the defined DAO interface StudentDAO:**

```
public class StudentJDBCTemplate implements StudentDAO {
private DataSource dataSource;
private SimpleJdbcCall jdbcCall;
public void setDataSource(DataSource dataSource) {
this.dataSource = dataSource;
this.jdbcCall = new SimpleJdbcCall(dataSource).
withProcedureName("getRecord");
}
```

Spring JDBC Framework

```
…..
//Specify the stored procedure parameters here.
public Student getStudent(Integer id) {
SqlParameterSource in = new MapSqlParameterSource().
addValue("in_id", id);
Map<String, Object> out = jdbcCall.execute(in);
Student student = new Student();
student.setId(id);
student.setName((String) out.get("out_name"));
student.setAge((Integer) out.get("out_age"));
return student;
}
….
…..
….
```

Questions

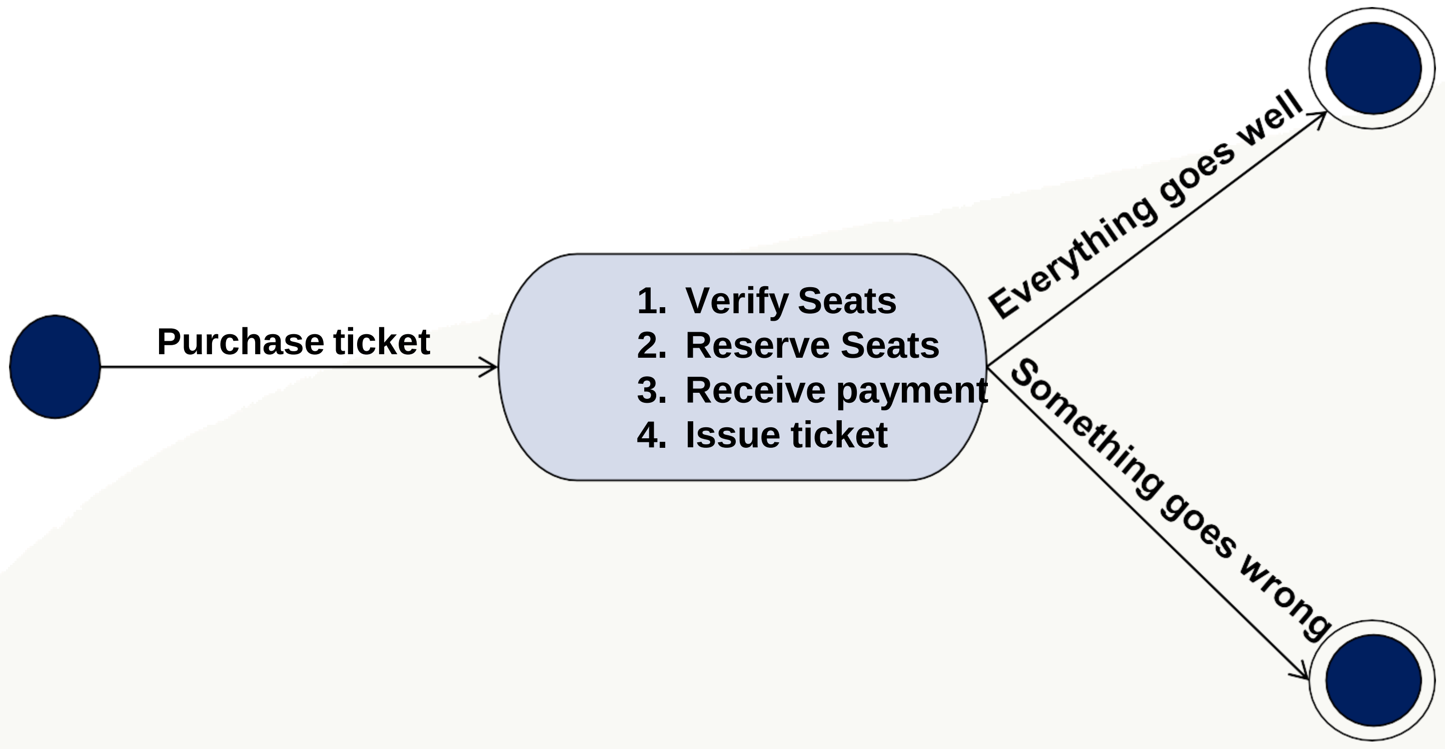
1. What JDBC Framewrok?
2. How to configure the DataSource in xml file?
3. What is JDBC Template?
4. Give the code block to query the record from the database?
5. Explain about SimpleJDBCCall

Spring Transaction Management

A database transaction is a sequence of actions that are treated as a single unit of work. These actions should either complete entirely or take no effect at all. Transaction management is an important part of and RDBMS oriented enterprise applications to ensure data integrity and consistency.

Spring Transaction Management

Example for Transaction



Spring Transaction Management

ACID properties:

- **Atomicity**: A transaction should be treated as a single unit of operation which means either the entire sequence of operations is successful or unsuccessful.
- **Consistency**: This represents the consistency of the referential integrity of the database, unique primary keys in tables etc.
- **Isolation**: There may be many transactions processing with the same data set at the same time, each transaction should be isolated from others to prevent data corruption.
- **Durability**: Once a transaction has completed, the results of this transaction have to be made permanent and cannot be erased from the database due to system failure.

Spring Transaction Management

- A real RDBMS database system will guarantee all the four properties for each transaction. The simplistic view of a transaction issued to the database using SQL is as follows:

  - **Begin** the transaction using begin transaction command.
  - Perform various **deleted, update or insert** operations using SQL queries.
  - If all the operation are successful then perform **commit** otherwise **rollback** all the operations.

Spring Transaction Management

Spring framework provides an abstract layer on top of different underlying transaction management APIs. The Spring's transaction support aims to provide an alternative to EJB transactions by adding transaction capabilities to POJOs. Spring supports both programmatic and declarative transaction management. EJBs requires an application server, but Spring transaction management can be implemented without a need of application server.

Spring Transaction Management

Local vs. Global Transactions

- Local transactions are specific to a single transactional resource like a JDBC connection, whereas global transactions can span multiple transactional resources like transaction in a distributed system.

- Local transaction management can be useful in a centralized computing environment where application components and resources are located at a single site, and transaction management only involves a local data manager running on a single machine. Local transactions are easier to be implemented.

Spring Transaction Management

Programmatic vs. Declarative

Spring supports two types of transaction management:

- **Programmatic transaction management:** This means that you have manage the transaction with the help of programming. That gives you extreme flexibility, but it is difficult to maintain.

- **Declarative transaction management:** This means you separate transaction management from the business code. You only use annotations or XML based configuration to manage the transactions.

Spring Transaction Management

Spring Transaction Abstractions

The key to the Spring transaction abstraction is defined by the*org.springframework.transaction.PlatformTransactionManager interface, which is as follows:*

```
public interface PlatformTransactionManager {
TransactionStatus getTransaction(TransactionDefinition definition);
throws TransactionException;
void commit(TransactionStatus status) throws TransactionException;
void rollback(TransactionStatus status) throws TransactionException;
}
```

Spring Transaction Management

Description about the methods

Method	Description
TransactionStatus getTransaction(TransactionDefinition definition)	This method returns a currently active transaction or create a new one, according to the specified propagation behavior.
void commit(TransactionStatus status)	This method commits the given transaction, with regard to its status.
void rollback(TransactionStatus status)	This method performs a rollback of the given transaction.

Spring Transaction Management

The *TransactionDefinition is the core interface of the transaction support in Spring and it is defined as below:*

```
public interface TransactionDefinition {
int getPropagationBehavior();
int getIsolationLevel();
String getName();
int getTimeout();
boolean isReadOnly();
}
```

Spring Transaction Management

Method	Description
int getPropagationBehavior()	This method returns the propagation behavior. Spring offers all of the transaction propagation options familiar from EJB CMT.
int getIsolationLevel()	This method returns the degree to which this transaction is isolated from the work of other transactions.
String getName()	This method returns the name of this transaction.
int getTimeout()	This method returns the time in seconds in which the transaction must complete.
boolean isReadOnly()	This method returns whether the transaction is read-only.

Spring Transaction Management

Following are the possible values for isolation level:

- **TransactionDefinition.ISOLATION_DEFAULT** This is the default isolation level.
- **TransactionDefinition.ISOLATION_READ_COMMITTED** Indicates that dirty reads are prevented; non-repeatable reads and phantom reads can occur.
- **TransactionDefinition.ISOLATION_READ_UNCOMMITTED** Indicates that dirty reads, non-repeatable reads and phantom reads can occur.
- **TransactionDefinition.ISOLATION_REPEATABLE_READ** Indicates that dirty reads and non-repeatable reads are prevented; phantom reads can occur.
- **TransactionDefinition.ISOLATION_SERIALIZABLE** Indicates that dirty reads, non-repeatable reads and phantom reads are prevented.

Spring Transaction Management

Following are the possible values for propagation types:

- **TransactionDefinition.PROPAGATION_MANDATORY** Support a current transaction; throw an exception if no current transaction exists.
- **TransactionDefinition.PROPAGATION_NESTED** Execute within a nested transaction if a current transaction exists.
- **TransactionDefinition.PROPAGATION_NEVER** Do not support a current transaction; throw an exception if a current transaction exists.
- **TransactionDefinition.PROPAGATION_NOT_SUPPORTED** Do not support a current transaction; rather always execute non-transactionally.
- **TransactionDefinition.PROPAGATION_REQUIRED** Support a current transaction; create a new one if none exists.
- **TransactionDefinition.PROPAGATION_REQUIRES_NEW** Create a new transaction, suspending the current transaction if one exists.
- **TransactionDefinition.PROPAGATION_SUPPORTS** Support a current transaction; execute non-transactionally if none exists.
- **TransactionDefinition.TIMEOUT_DEFAULT** Use the default timeout of the underlying transaction system, or none if timeouts are not supported.

Spring Transaction Management

The *TransactionStatus interface provides a simple way for transactional code to control transaction execution and query transaction status.*

```
public interface TransactionStatus extends SavepointManager {
boolean isNewTransaction();
boolean hasSavepoint();
void setRollbackOnly();
boolean isRollbackOnly();
boolean isCompleted();
}
```

Spring Transaction Management

Method	Description
boolean hasSavepoint()	This method returns whether this transaction internally carries a savepoint, that is, has been created as nested transaction based on a savepoint.
boolean isCompleted()	This method returns whether this transaction is completed, that is, whether it has already been committed or rolled back.
boolean isNewTransaction()	This method returns true in case the present transaction is new.
boolean isRollbackOnly()	This method returns whether the transaction has been marked as rollback-only.
void setRollbackOnly()	This method sets the transaction rollback-only

Spring Transaction Management

Programmatic Transaction Management

- Programmatic transaction management approach allows you to manage the transaction with the help of programming in your source code. That gives you extreme flexibility, but it is difficult to maintain.
- Before we begin, it is important to have at least two database tables on which we can perform various CRUD operations with the help of transactions. Let us take **Student table, which can be created in MySQL CAP database with the following DDL:**

Spring Transaction Management

```
CREATE TABLE Student(
ID INT NOT NULL AUTO_INCREMENT,
NAME VARCHAR(20) NOT NULL,
AGE INT NOT NULL,
PRIMARY KEY (ID) );
```

Second table is **Marks in which we will maintain marks for students based on years. Here SID is the foreign key for Student table.**

```
CREATE TABLE Marks(
SID INT NOT NULL,
MARKS INT NOT NULL,
YEAR INT NOT NULL);
```

Spring Transaction Management

- ***PlatformTransactionManager*** *directly to implement programmatic approach to implement transactions.*

- *To start a new transaction we need to have a instance of* ***TransactionDefinition*** *with the appropriate transaction attributes. For this example we will simply create an instance of* ***DefaultTransactionDefinition*** *to use the default transaction attributes.*

- Once the TransactionDefinition is created, we can start our transaction by calling ***getTransaction()****method, which returns an instance of* ***TransactionStatus****.*

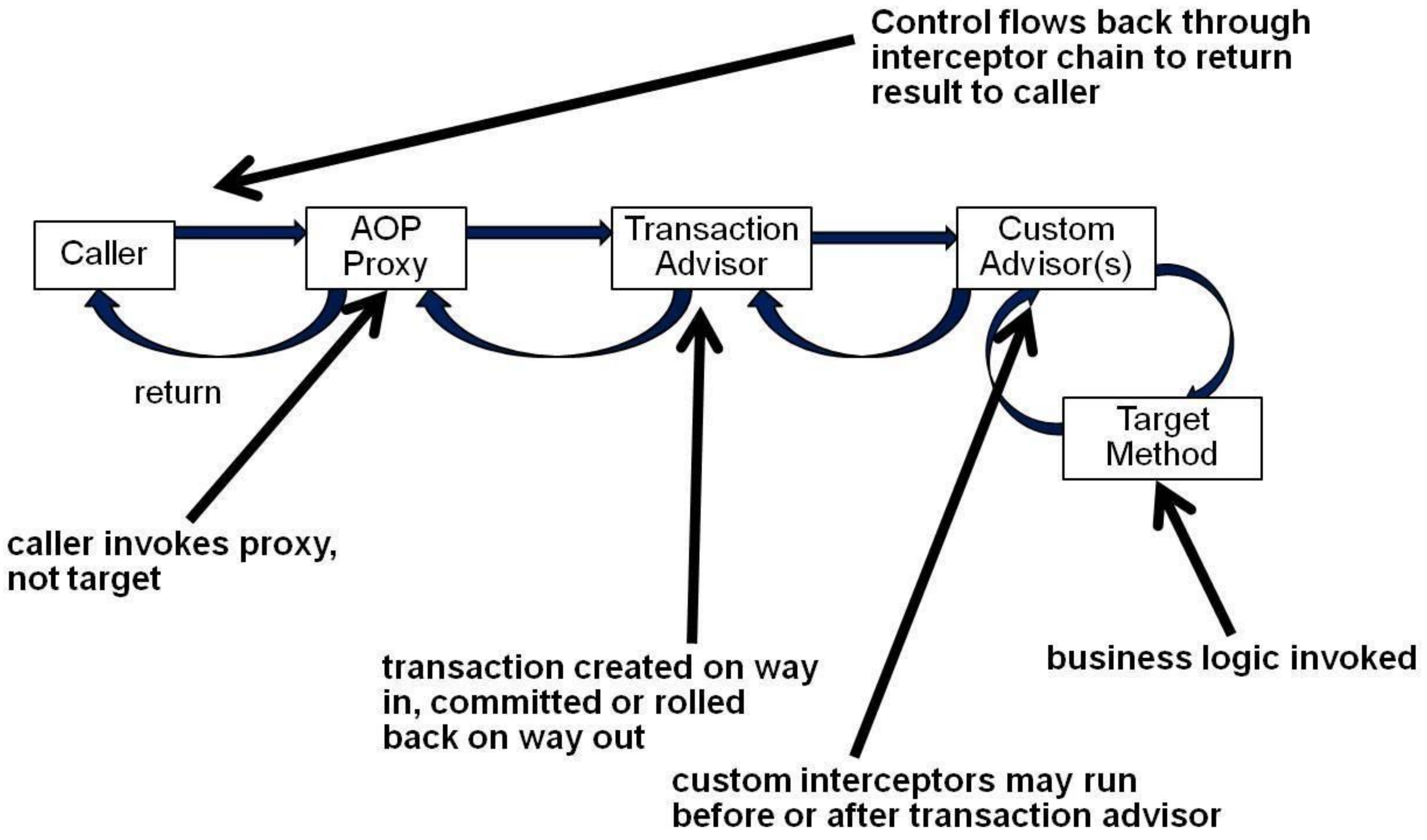
- *The TransactionStatus objects helps in tracking the current status of the transaction and finally, if everything goes fine, you can use* ***commit()*** *method of PlatformTransactionManager to commit the transaction, otherwise you can use* ***rollback()*** *to rollback the complete operation.*

Spring Transaction Management

Declarative Transaction Management

- We use <tx:advice /> tag, which creates a transaction-handling advice and same time we define a pointcut that matches all methods we wish to make transactional and reference the transactional advice.
- If a method name has been included in the transactional configuration then created advice will begin the transaction before calling the method.
- Target method will be executed in a *try / catch block.*
- If the method finishes normally, the AOP advice commits the transaction successfully otherwise it performs a rollback.

Spring Transaction Management


Control flows back through interceptor chain to return result to caller
Caller
AOP Proxy
Transaction Advisor
Custom Advisor(s)
Target Method
return
caller invokes proxy, not target
transaction created on way in, committed or rolled back on way out
business logic invoked
custom interceptors may run before or after transaction advisor

Spring Transaction Management

Let us see how above mentioned steps work but before we begin, it is important to have at least two database tables on which we can perform various CRUD operations with the help of transactions. Let us take **Student table, which can be created in MySQL TEST database with the following DDL:**

```
CREATE TABLE Student(
ID INT NOT NULL AUTO_INCREMENT,
NAME VARCHAR(20) NOT NULL,
AGE INT NOT NULL,
PRIMARY KEY (ID) );
```

Second table is **Marks in which we will maintain marks for students based on years. Here SID is the foreign key for Student table.**

```
CREATE TABLE Marks(
SID INT NOT NULL,
MARKS INT NOT NULL,
YEAR INT NOT NULL);
```

Spring Transaction Management

Following is the configuration file **Beans.xml:**

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:tx="http://www.springframework.org/schema/tx"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/tx
http://www.springframework.org/schema/tx/spring-tx-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd">
<!-- Initialization for data source -->
```

Spring Transaction Management

```
<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/CAP"/>
<property name="username" value="root"/>
<property name="password" value="cohondob"/>
</bean>
<tx:advice id="txAdvice" transaction-manager="transactionManager">
<tx:attributes>
<tx:method name="create"/>
</tx:attributes>
</tx:advice>
```

Spring Transaction Management

```
<aop:config>
<aop:pointcut id="createOperation"
expression="execution(* com.tutorialspoint.StudentJDBCTemplate.create(..))"/>
<aop:advisor advice-ref="txAdvice" pointcut-ref="createOperation"/>
</aop:config>
<!-- Initialization for TransactionManager -->
<bean id="transactionManager"
class="org.springframework.jdbc.datasource.DataSourceTransactionManager">
<property name="dataSource" ref="dataSource" />
</bean>
<!-- Definition for studentJDBCTemplate bean -->
<bean id="studentJDBCTemplate"
class="com.capgemini.StudentJDBCTemplate">
<property name="dataSource" ref="dataSource" />
</bean>
</beans>
```

Questions

1. Difference between programmatic and Declarative transaction.
2. Difference between Local Vs Global
3. Describe ACID properties
4. Discuss Transaction management used in EJB and Spring framework.

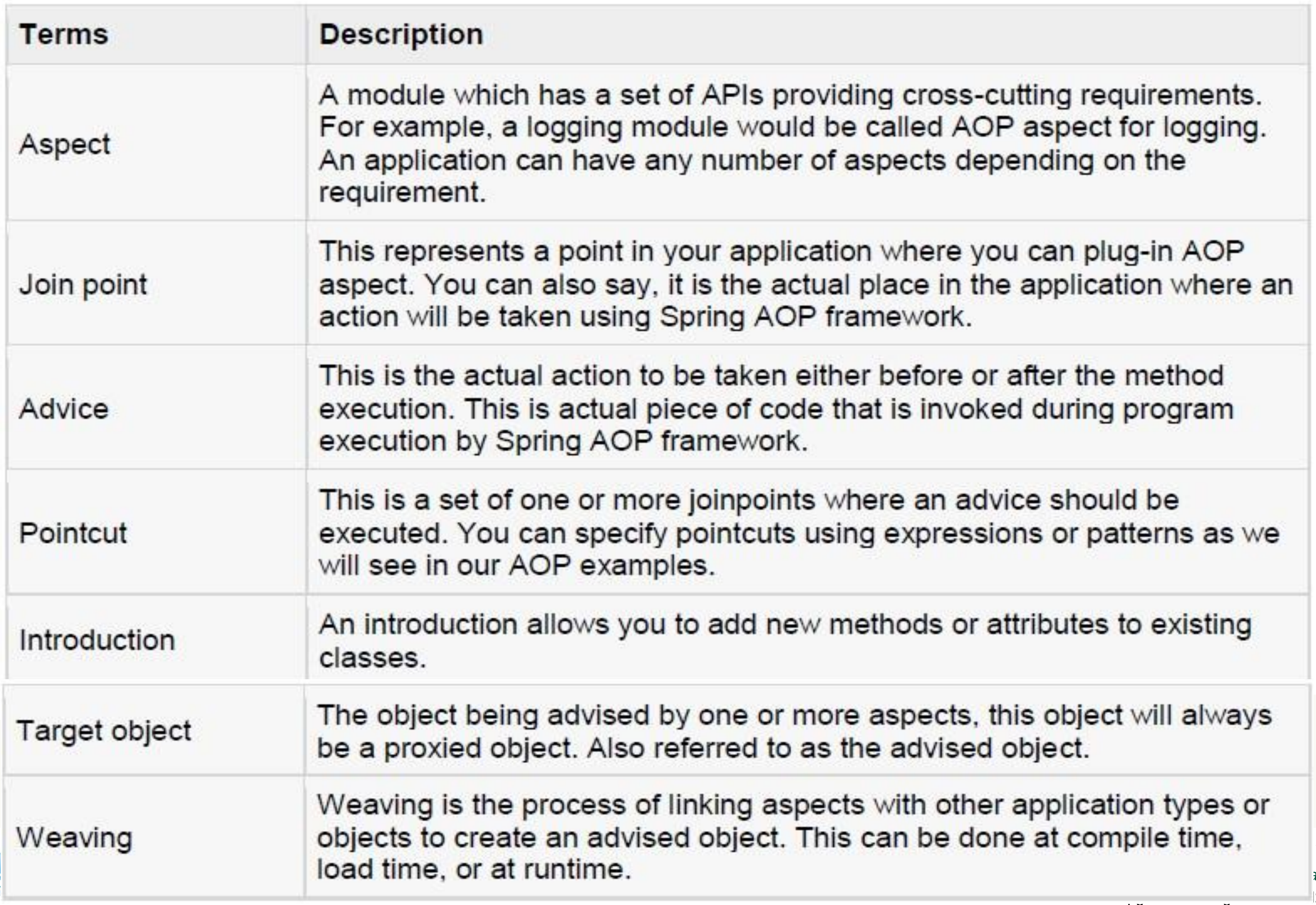
AOP in SpringFramework

- Breaking down program logic into distinct parts called so-called concerns.

- **Cross-cutting concerns and these cross-cutting concerns are conceptually separate from the application's business logic.**

- Examples of aspects like logging, auditing, declarative transactions, security, and caching etc.

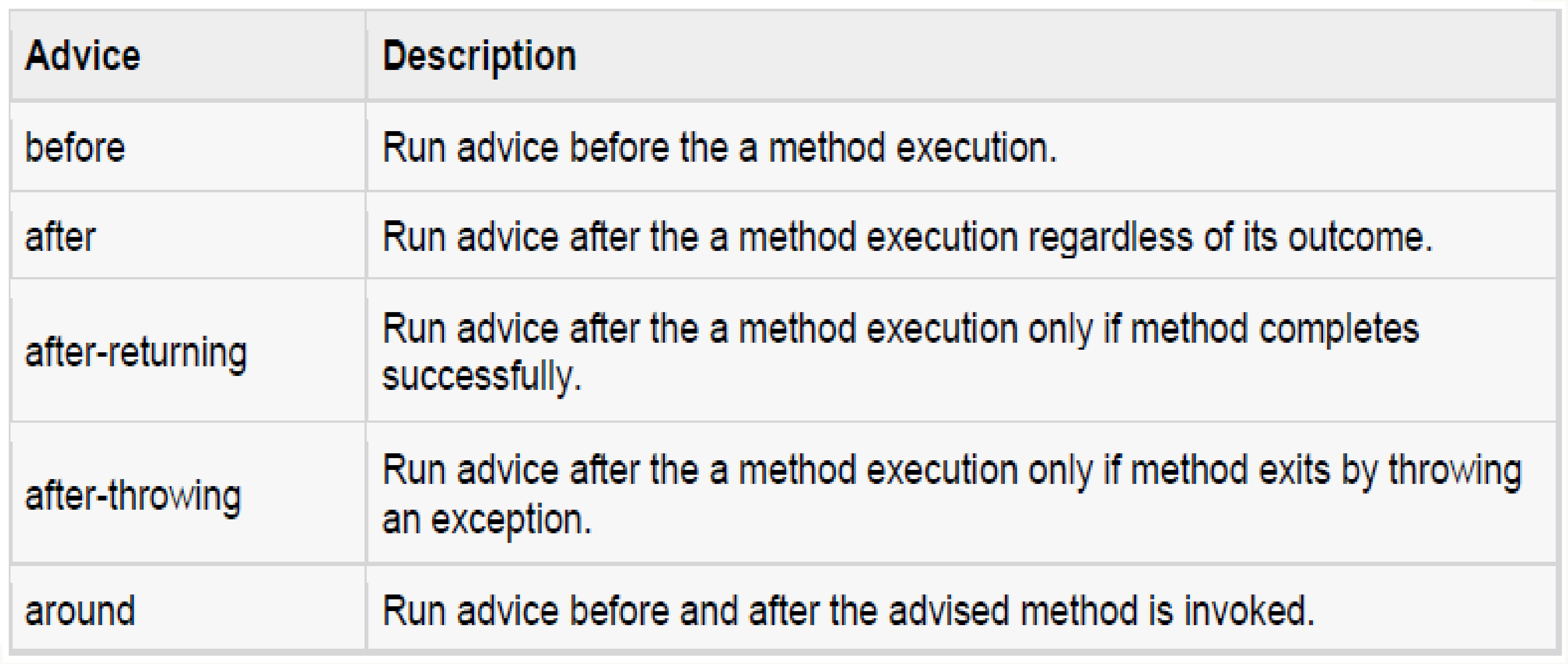
AOP Terminologies



Terms	Description
Aspect	A module which has a set of APIs providing cross-cutting requirements. For example, a logging module would be called AOP aspect for logging. An application can have any number of aspects depending on the requirement.
Join point	This represents a point in your application where you can plug-in AOP aspect. You can also say, it is the actual place in the application where an action will be taken using Spring AOP framework.
Advice	This is the actual action to be taken either before or after the method execution. This is actual piece of code that is invoked during program execution by Spring AOP framework.
Pointcut	This is a set of one or more joinpoints where an advice should be executed. You can specify pointcuts using expressions or patterns as we will see in our AOP examples.
Introduction	An introduction allows you to add new methods or attributes to existing classes.
Target object	The object being advised by one or more aspects, this object will always be a proxied object. Also referred to as the advised object.
Weaving	Weaving is the process of linking aspects with other application types or objects to create an advised object. This can be done at compile time, load time, or at runtime.

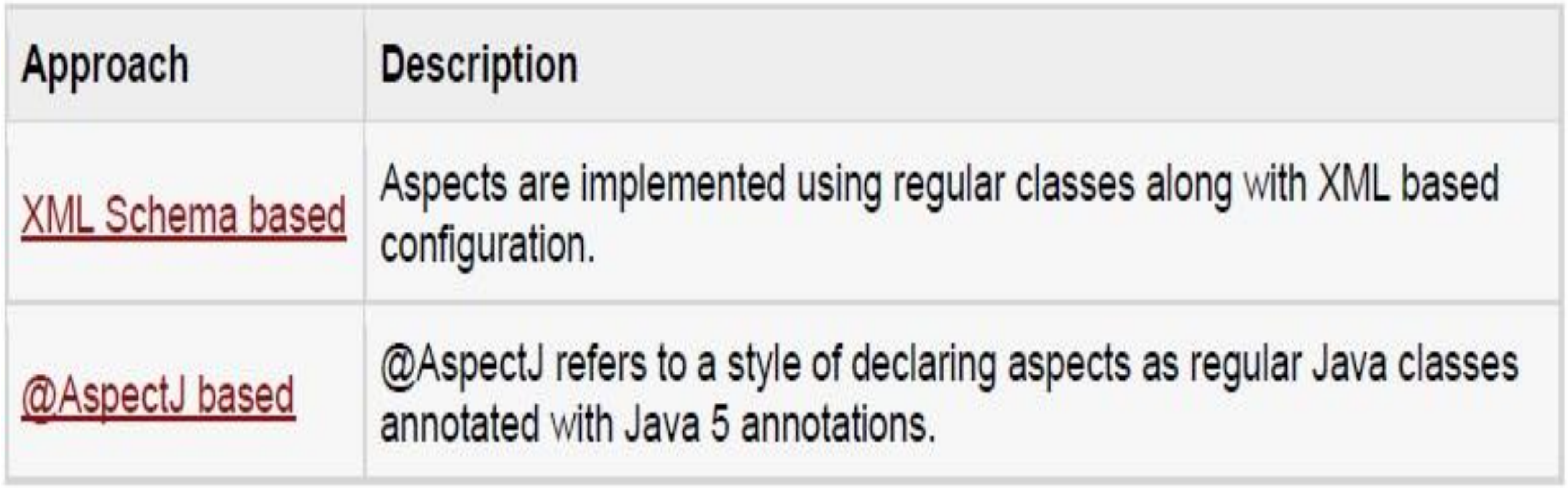
Types of Advice

Spring aspects can work with five kinds of advice mentioned below:



Advice	Description
before	Run advice before the a method execution.
after	Run advice after the a method execution regardless of its outcome.
after-returning	Run advice after the a method execution only if method completes successfully.
after-throwing	Run advice after the a method execution only if method exits by throwing an exception.
around	Run advice before and after the advised method is invoked.

Custom Aspects Implementation



Approach	Description
XML Schema based	Aspects are implemented using regular classes along with XML based configuration.
@AspectJ based	@AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.

XML Schema Based AOP with Spring

We need to import the spring-aop schema as described below:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:aop="http://www.springframework.org/schema/aop"
xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
http://www.springframework.org/schema/aop
http://www.springframework.org/schema/aop/spring-aop-3.0.xsd ">
<!-- bean definition & AOP specific configuration -->
</beans>
```

Declaring an aspect

An aspect is declared using the **<aop:aspect>** element, and the backing bean is referenced using the **ref** attribute as follows:

```
<aop:config>
<aop:aspect id="myAspect" ref="aBean">
...
</aop:aspect>
</aop:config>
<bean id="aBean" class="...">
...
</bean>
```

Declaring a pointcut

- A pointcut helps in determining the join points (ie methods) of interest to be executed with different advices. While working with XML Schema based configuration, pointcut will be defined as follows:

```
<aop:config>
<aop:aspect id="myAspect" ref="aBean">
<aop:pointcut id="businessService"
expression="execution(* com.xyz.myapp.service.*.*(..))"/>
...
</aop:aspect>
</aop:config>
<bean id="aBean" class="...">
...
</bean>
```

Declaring a pointcut (continue…)

- The example defines the pointcutnamed 'businessService' that will match the execution of getName() method available in Student class under the package org.capgemini.

```
<aop:config>
<aop:aspect id="myAspect" ref="aBean">
<aop:pointcut id="businessService"
expression="execution(* org.capgemini.Student.getName(..))"/>
...
</aop:aspect>
</aop:config>
<bean id="aBean" class="...">
…
</bean>
```

@AspectJ Based AOP with Spring

- @AspectJ refers to a style of declaring aspects as regular Java classes annotated with Java 5 annotations.
- The @AspectJ support is enabled by including the following element inside your XML Schema-based configuration file.

```
<aop:aspectj-autoproxy/>
```

Declaring an aspect

- Aspects classes are like any other normal bean and may have methods and fields just like any other class, except that they will be annotated with @Aspect as follows:

```
package org.capgemini;
import org.aspectj.lang.annotation.Aspect;
@Aspect
public class AspectModule {
}
```

- They will be configured in XML like any other bean as follows:

```
<bean id="myAspect" class="org.capgemini.AspectModule">
<!-- configure properties of aspect here as normal -->
</bean>
```

Declaring a pointcut

- A pointcut helps in determining the join points (ie methods) of interest to be executed with different advices. While working with @AspectJ based configuration, pointcut declaration has two parts:
  - A pointcut expression that determines exactly which method executions we are interested in.
  - A pointcut signature comprising a name and any number of parameters. The actual body of the method is irrelevant and in fact should be empty.

Declaring a pointcut (continue…)

Example

- The following example defines a pointcut named 'businessService' that will match the execution of every method available in the classes under the package org.capgemini.myapp.service:

```
import org.aspectj.lang.annotation.Pointcut;
@Pointcut("execution(* org.capgemini.myapp.service.*.*(..))") // expression
private void businessService() {} // signature
```

- The following example defines a pointcut named 'getname' that will match the execution of getName() method available in Student class under the package org.capgemini:

```
import org.aspectj.lang.annotation.Pointcut;
@Pointcut("execution(* org.capgemini.Student.getName(..))")
private void getname() {}
```

Declaring advices

```
@Before("businessService()")
public void doBeforeTask(){
...
}

@After("businessService()")
public void doAfterTask(){
...
}

@AfterReturning(pointcut = "businessService()", returning="retVal")
public void doAfterReturnningTask(Object retVal){
// you can intercept retVal here.
...
}

@AfterThrowing(pointcut = "businessService()", throwing="ex")
public void doAfterThrowingTask(Exception ex){
// you can intercept thrown exception here.
...
}

@Around("businessService()")
public void doAroundTask(){ ... }
```

Questions

1. Define AOP?
2. Discuss different types of Advices
3. Difference between XML schema based AOP and @AspectJ based AOP.

Recap

DI

IOC

AOP

Merits

Framework

Architecture

MVC

JDBC

Transactions

Thank You For Your Time