Spring

Topics

Sl.No.	Topic
1	Spring Core
2	Spring DAO
3	Spring Transactions
4	Spring Web – Servlets, JSP,
5	Spring MVC
6	Spring AOP

Spring Introduction

- A J2EE Framework developed by **Rod Johnson** to make building applications easier.
- The programming model is built upon **Inversion of Control** and **Aspect-Oriented Programming**. Inversion of Control allows classes to be loosely coupled and dependencies written in xml or annotated.
- The **user code need not have Spring references** all over the place as Spring integrates using these techniques by placing specific frameworks where needed and manage the business objects & their dependencies
- Spring provides a framework to integrate all kinds of Java technologies/API's and makes it possible to use them with simple POJO's. The framework that allows developer to pick and choose features to use from.

Features

- ❖Inversion of Control
- ❖Dependency Injection
- ❖Foundational support for JDBC, JPA, JMS
- ❖Spring MVC web application and RESTful web service framework
- ❖Seamless integration with other frameworks
- ❖Aspect-Oriented Programming including Spring's declarative transaction management
- ❖Much more...

Background

➢ Tight Coupling

- In O.O. design, **Coupling** refers to the degree of direct knowledge that one element has of another. In other words, how often do **changes in class A** force **related changes in class B.**

- Tight coupling means the **two classes often change together**. In other words, if A knows more than it should about the way in which B was implemented, then A and B are tightly coupled.

- This scenario arises when a **class assumes too many responsibilities**, or **when one concern is spread over many classes** rather than having its own class.

contd..

```
class Employee
 {
   int  eid;
   String name
   Address address;
  Employee()
  {
   eid= 0;
   name= " ";
   address=new Address();
  }
}
```

```
class Employee
{
  int  eid;
  String name
  Address address;
  Employee ( int i1, String s, Address
                   address)
   {  eid= i1;
      name= s;
      this.address=address;
    }
}
```

contd..

➢ Look-up problem

- ❖ Applications developed using J2EE consist of many components, belonging to different frame works either within Java/J2EE or exterior. Communication between components is thru servers of different vendors.
- ❖ These components offer transaction management, multithreading, security, etc.

E.g. when a component C1 requires another component C2, then C1 itself is responsible for looking up the C2 as it depends upon.

- ❖ The **look–up happens by name**, so the name of the dependency is hardcoded in the component (code or deployment descriptor).
- ❖ The result is heavy weight application which includes services even when not needed and developers–architects spend lot of time in **setting up configurations** using **xml files**.

contd..

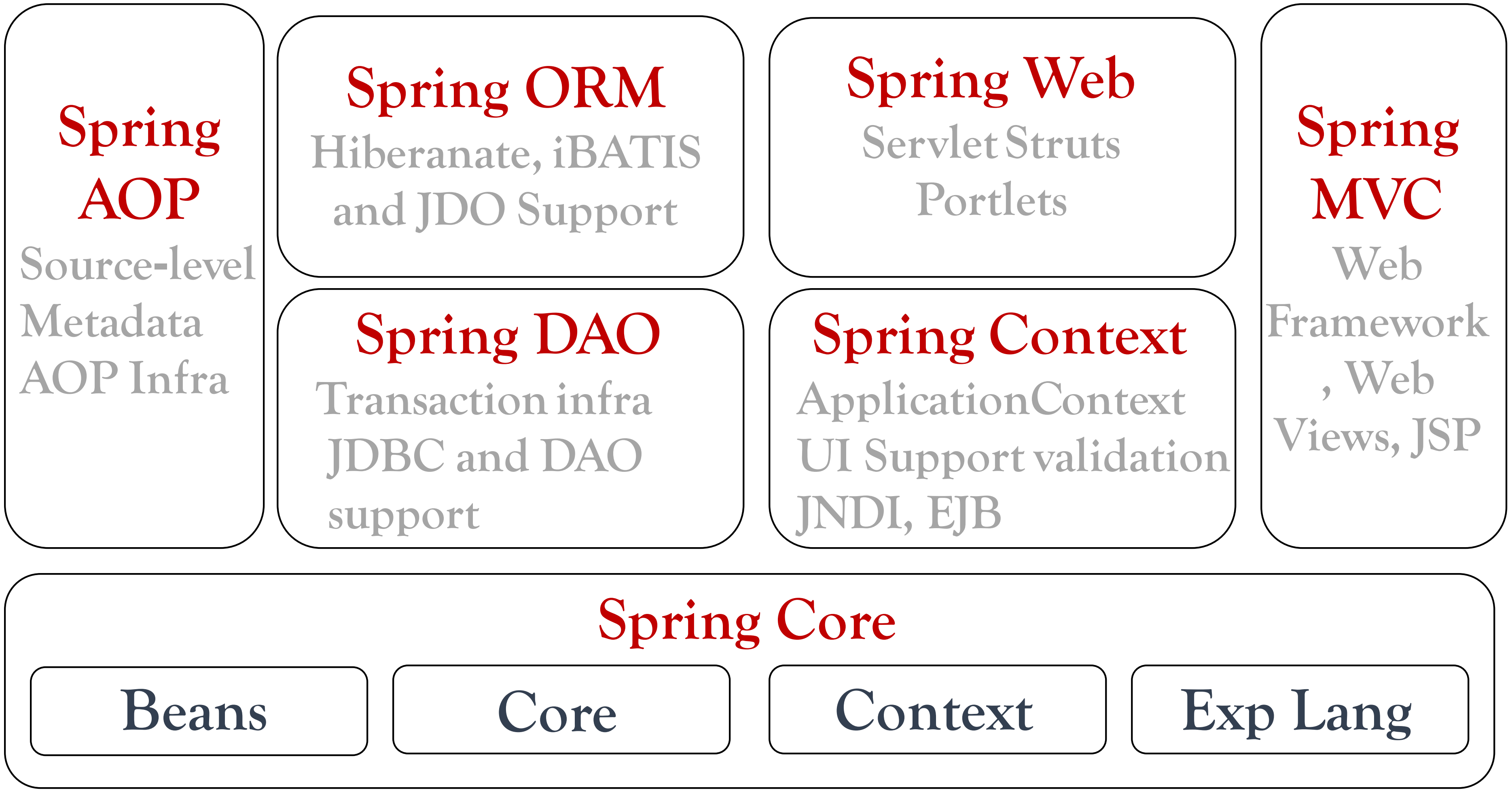
- **Problem with Testing**
  - ✓ The code can't be run in a unit test, since it presumes all kinds of services present, normally provided by the J2EE application server i.e. it is not testable outside of the container.
  - ✓ Deploying a J2EE application in a container and starting it up can be a very time-consuming task, so not very ideal if test cases need to run unit tests regularly.
- ❖ Developer needs to comply to several seemingly arbitrary rules, and write code that is not so Object Oriented.
- ❖ The classes have to be implemented in a very specific way, which coupled the business logic to the J2EE classes.
- ❖ The code less extensible, and not very test friendly as it uses specific J2EE classes and interfaces.

contd..

➢ **Solution –**

- ❖ Solution is to write just POJOs, without the J2EE standards overhead.
- ❖ No overhead to implement any interfaces or extend from other classes,
- ❖ Clean implementation and design of domain with regular Java classes.

Spring Modules


Spring AOP
Source-level Metadata AOP Infra
Spring ORM
Hiberanate, iBATIS and JDO Support
Spring Web
Servlet Struts Portlets
Spring MVC
Web Framework , Web Views, JSP
Spring DAO
Transaction infra JDBC and DAO support
Spring Context
ApplicationContext UI Support validation JNDI, EJB
Spring Core
Beans
Core
Context
Exp Lang

Spring Core

Inversion of Control

- **IOC introduction –**
  - Developing an application, involves **dependencies** between and on components, services, classes etc. which are '**wired**' together on the spot where needed.
  - The disadvantage of this approach is that when user needs to use a **different implementation** of the **dependency**, its necessary for code change. Also involves huge cost if change is necessitated in implementation context of the environment.
  - E.g. to use a different authentication service during development as in production. It is not really convenient to change code every time a production artifact is made, or unit-tests are run.

contd..

- So the wiring of these dependencies is taken out of the code, and an external party manages the wiring, namely the container. Hence the name **Inversion of Control**, i.e. let something from outside control how dependencies are wired together.
- It is also **Dependency Injection** because the container '**injects**' the necessary dependencies instead of letting the developer manage them.
- In principle, a framework isn't needed to inject dependencies into the code, but most applications built by using inversion of control use a framework of some type to carry out the dependency injection.
- These read configuration information and then use the Java reflection API or byte-code manipulation to invoke the appropriate methods on the code to inject the dependencies.

contd..

- Although this behavior is not innate in the dependency injection approach, it is so widespread.
- Unfortunately, it leads directly to the one real disadvantage that containers such as Spring have over hard-coding of dependencies: that they lose some of the advantages of *static type checking*.
- The configuration information will not be read until runtime; therefore, any incompatible type information given in the configuration will not cause errors to be produced until runtime.

Dependency Injection

- In an application, Java classes should be as independent as possible of each other.
- The mutual independence increase the possibility to reusability and test them independent of other classes while doing unit testing.
- Dependency Injection (or sometime called wiring) helps in connecting these classes together and same time keeping them independent.
- With Dependency injection
  - ✓ Beans define their dependencies through constructor arguments or properties
  - ✓ The container provides the injection at runtime

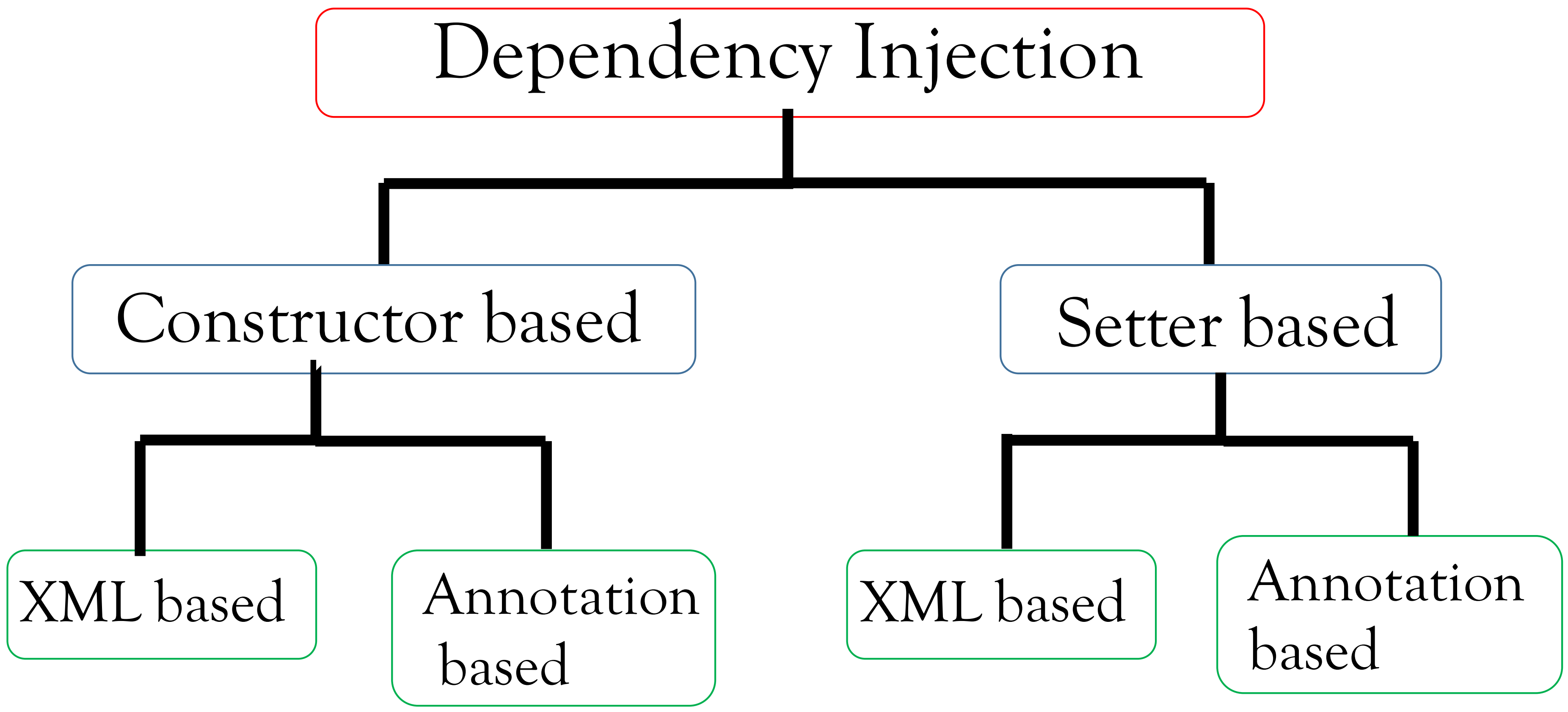
contd..

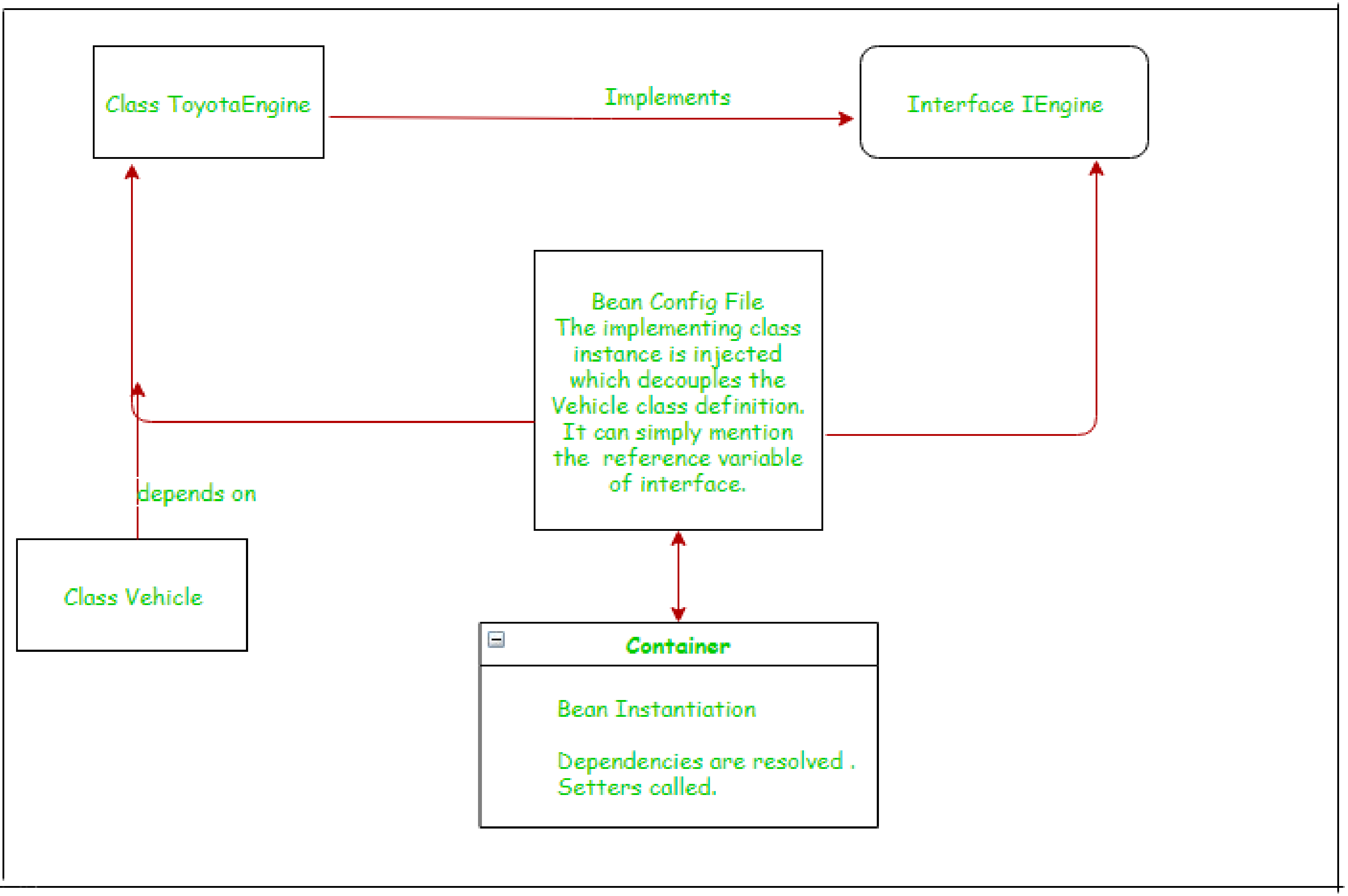
- Decouples **object creators** and **locators** from application logic
- Easy to maintain, reuse, and conduct **Unit Testing**.
- Dependency Injection is a technique that allows us to remove dependency of one component/class on other component/class.
- **Dependency** is an object on which a class depends. **Injection** refers to the process of injecting a dependency (object) into the dependent class.
- The main idea behind the pattern is to allow a class to use object of another class without actually bothering on how to create that object.

contd..

- Dependency Injection (DI) is a software design pattern that implements inversion of control for resolving dependencies.
- An injection is the passing of a dependency to a dependent object that would use it.
- DI is a process whereby objects define their dependencies. The other objects they work with—only through constructor arguments or arguments to a factory method or property—are set on the object instance after it is constructed or returned from a factory method.
- The container then injects those dependencies, and it creates the bean. This process is named Inversion of Control (IoC) (the bean itself controls the instantiation or location of its dependencies by using direct construction classes or a Service Locator). DI refers to the process of supplying an external dependency to a software component.

contd..

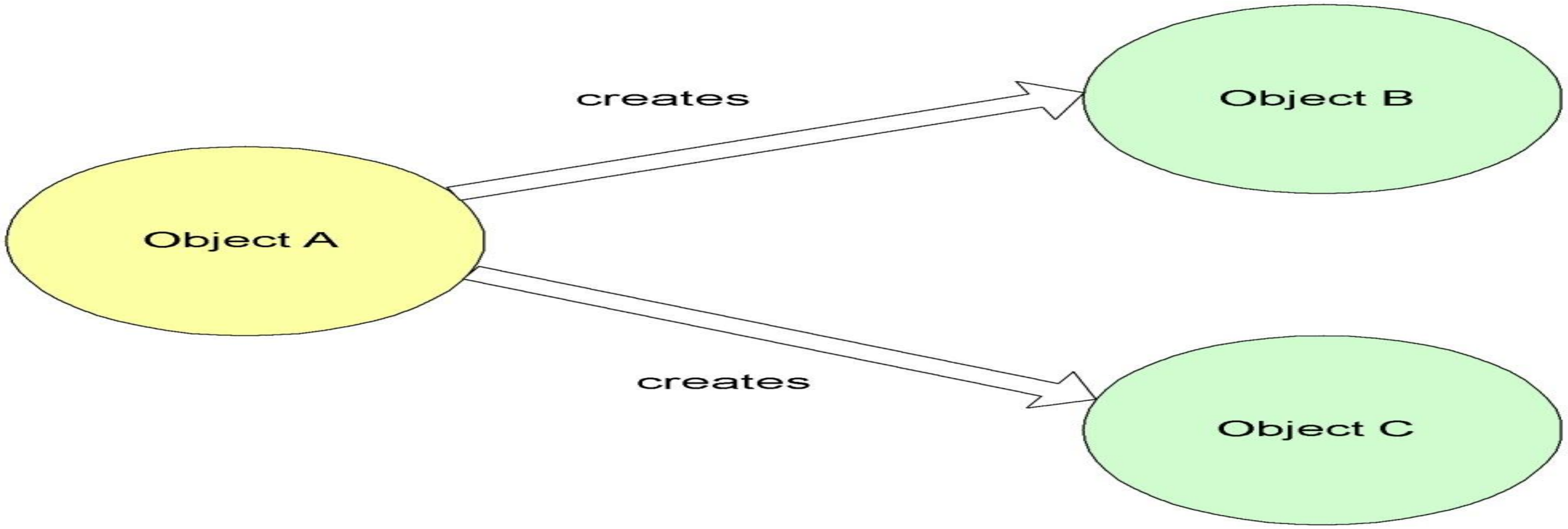

Dependency Injection
Constructor based
Setter based
XML based
Annotation based
XML based
Annotation based


Class ToyotaEngine
Implements
Interface IEngine
Bean Config File
The implementing class instance is injected which decouples the Vehicle class definition. It can simply mention the reference variable of interface.
depends on
Class Vehicle
Container
Bean Instantiation
Dependencies are resolved . Setters called.

Constructor based DI

❖ In constructor based DI –

- ✓ the dependency is being injected into the class through a **Class Constructor**. As flow of control gets "inverted" because the dependencies are effectively delegated to some external system, the concept is termed Dependency Injection (DI).
- ✓ the container invokes a class constructor with a number of arguments, each representing a dependency on other class.



XML based constructor DI

- ❖ The **<constructor-arg>** sub element of **<bean>** is used for constructor injection.
- ❖ The default is type is String, in case no data type is mentioned
- ❖ Collection values can also be injected by constructor. There are three elements inside the **constructor-arg** element.
  - ✓ list
  - ✓ set
  - ✓ map
- ❖ Each collection can have string based and non-String based values.

Annotation based constructor DI

- The bean configuration can be moved into the component class itself by using annotations on the relevant class, method, or field declaration.
- If both are used then XML configuration will override annotations because XML configuration will be injected after annotations.
- The annotations based configuration is turned off by default so need to turn it on by entering into spring XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns: ............>
    <context:annotation-config/>
    <!-- beans declaration goes here -->
</beans>
```

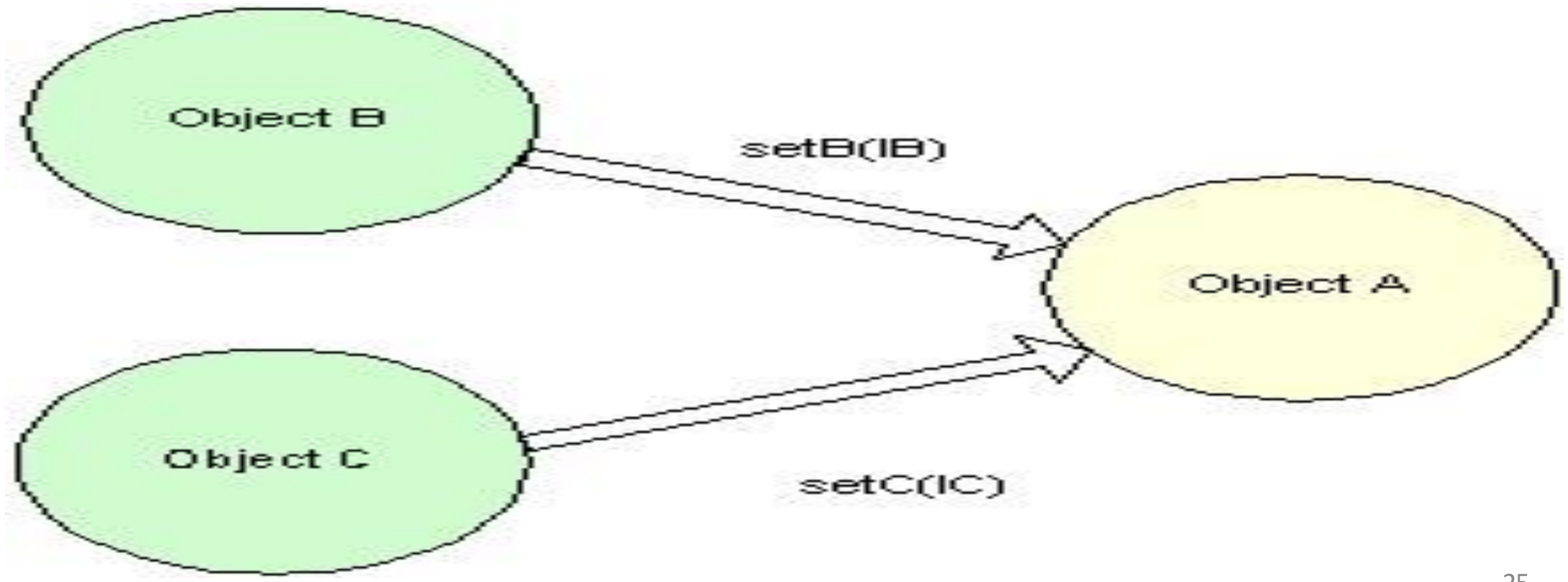
contd..

❖ Some important annotations in Spring –

1. @Bean – Applied to the method. Declares the bean object on the method which creates the bean object.
2. @Configuration – Specified on the class. Specifies the configuration of the bean objects.
3. @Required – The @Required annotation applies to bean property setter methods.
4. @Autowired – The @Autowired annotation is applied to bean property setter methods, non-setter methods, constructor and properties. @Autowired can be used to remove the confusion by specifiying which exact bean will be wired.
5. @Qualifier – The @Qualifier annotation is used along with @AutoWired if same bean is to be declared more than once.

Setter-method based DI

- ❖ In Setter–method based DI,
  - ✓ The other method of injecting dependency is through **setter methods** of the class where user creates instance and this instance will be used to call setter methods to access properties.
  - ✓ The container calls setter methods on the beans after invoking a no-argument constructor or no–argument static factory method to instantiate the bean.



XML based setter-method DI

- The **<property>** sub element of **<bean>** is used for constructor injection.
- The default is type is String, in case no data type is mentioned
- Collection values can also be injected by constructor. There are three elements inside the **property** element.
  - ✓ list
  - ✓ set
  - ✓ map
- Each collection can have string based and non–String based values.

Annotation based setter-method DI

- The bean configuration can be moved into the component class itself by using annotations on the relevant class, method, or field declaration.
- If both are used then XML configuration will override annotations because XML configuration will be injected after annotations.
- The annotations based configuration is turned off by default so need to turn it on by entering into spring XML file

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns: ............>
    <context:annotation-config/>
    <!-- beans declaration goes here -->
</beans>
```

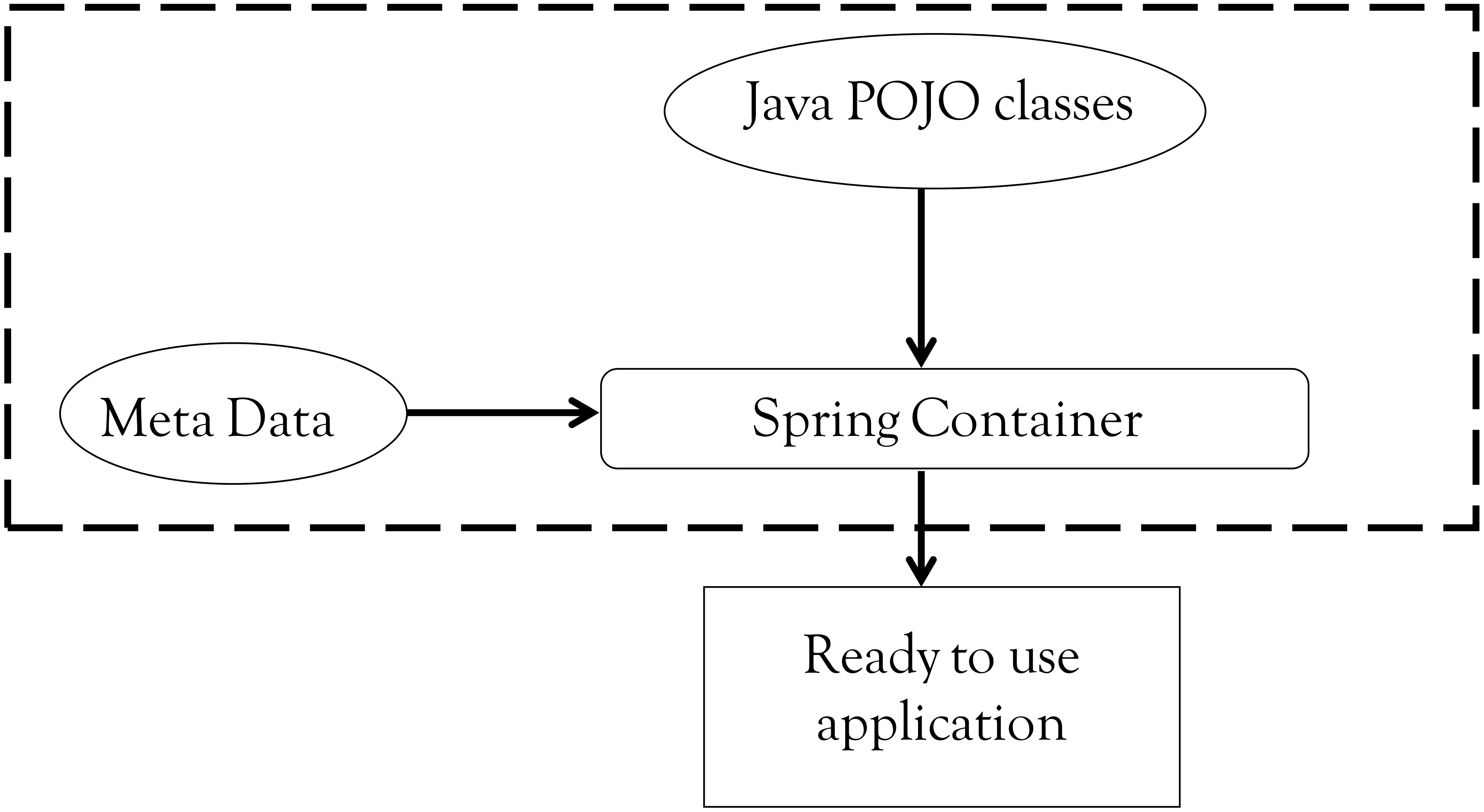
contd..

- Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written.
- The following are three ways to provide configuration metadata to the Spring Container:
  - ✓ XML based configuration file.
  - ✓ Annotation-based configuration
  - ✓ Java-based configuration

Spring Container

- ❖The Spring container is the core of the Spring Framework, works on principle of D.I. The container –
  - ✓ creates objects called as Spring Bean,
  - ✓ wire them together,
  - ✓ configure them, and
  - ✓ manage their complete lifecycle from creation till destruction.
- ❖The container gets its instructions by reading configuration metadata (either xml or annotations)
- ❖The IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.

contd..


Java POJO classes
Meta Data
Spring Container
Ready to use application

contd..

Sl.No.	Name	ver	Description
1.	BeanFactory		defined by the ***org.springframework.beans.factory.BeanFactory interface.*** simplest container providing basic support for DI
2.	Application Context		defined by the ***org.springframework.context.ApplicationContext*** interface 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.

contd..

Sl.No	Name	ver	Description
1.	XmlBean Factory		this container reads the configuration metadata from a xml file and uses it to create a fully configured system or application.
2.	FileSystemXmlApplicationContext		this container loads the definitions of the beans from an XML file. The full path of the XML bean configuration file is given to the constructor.
3.	ClassPathXmlApplication Context		this container loads the definitions of the beans from an XML file. The container will look bean configuration XML file in CLASSPATH.
4.	WebXmlApplicationContext		this container loads the XML file with definitions of all beans from within a web application.

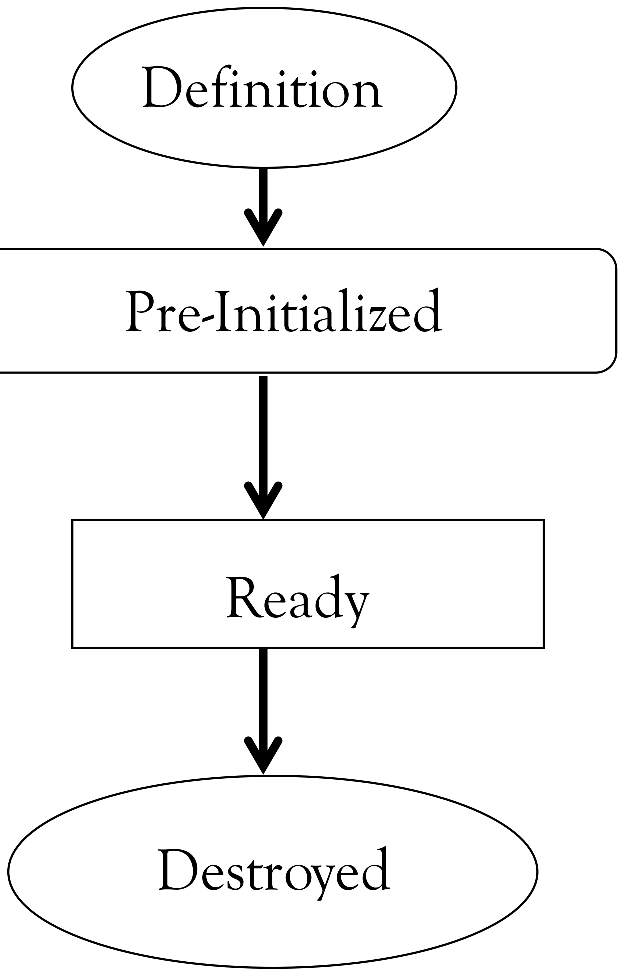
contd..

- XML based Containers
- Annotation based Containers

Spring Bean

- ❖ Spring bean is an object that is instantiated, assembled, and managed by a Spring IoC container.
- ❖ The bean definition contains the information called **configuration metadata** which is needed for the container to know the following :
  - ✓ **how to create a bean**
  - ✓ **bean's lifecycle details**
  - ✓ **bean's dependencies**
- ❖ The configuration metadata in the form of xml is provided to the container.
- ❖ Bean configuration details are specified in the spring bean configuration file and the beans that will be injected in other beans are specified using the attribute ref or inner bean.

Bean Life Cycle



contd..

- When a bean is instantiated, it may be required to perform some initialization to get it into a usable state and when the bean is no longer required and is removed from the container, some cleanup may be required.
- There is list of the activities that take place behind the scenes between the time of bean instantiation and its destruction,
- To define setup and teardown for a bean, declare the <bean> with **init-method** and/or **destroy-method** parameters.
- The init-method attribute specifies a method that is to be called on the bean immediately upon instantiation and destroy-method specifies a method that is called just before a bean is removed from the container.
- It would be reasonable to assume that both of these dependencies should be provided via injection.

contd..

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

Properties	**Description**
class	mandatory and specifies the bean class to be used to create the bean.
name	specifies the bean identifier uniquely. In xml-based configuration metadata, id and/or name attributes used to specify the bean identifier(s).
scope	specifies the scope of the objects created from a particular bean definition
constructor-arg	used to inject the dependencies
properties	used to inject the dependencies
autowiring mode	used to inject the dependencies

contd..

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.
destruction method	A callback to be used when the container containing the bean is destroyed.

Bean Scopes

❖ The Spring Framework supports following five scopes.

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.

AutoWiring

- The autowiring feature is such with which there is no need for ref attribute and provide bean references explicitly. The auto-wire mode automatically wires the beans.
- The autowire mode can be configured using "autowire" attribute

```
<bean id="bean_id" class="bean_class" autowire="default |
                    byname | byType | constructor | autodetect " />
```

- The different autowiring modes supported by Spring are –
  a. default or no
  b. byName
  c. byType
  d. constructor
  e. autodetect

contd..

Sl. No	Mode	Description
1.	by Name	When autowiring a property in bean, property name is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property else a error is raised.
2.	by Type	When autowiring a property in bean, property's class type is used for searching a matching bean definition in configuration file. If such bean is found, it is injected in property else a error is raised.
3.	constr uctor	Autowiring by constructor is similar to byType, but applies to constructor arguments. In autowire enabled bean, it will look for class type of constructor arguments, and then do a autowire bytype on all constructor arguments.
4.	auto detect	Autowiring by autodetect uses either of two modes i.e. constructor or byType modes. First it will try to look for valid constructor with arguments, If found the constructor mode is chosen. If there is no constructor defined in bean, or explicit default no-args constructor is present, the autowire byType mode is chosen.

contd..

- **Autowiring** feature internally uses setter or constructor injection.
- Autowiring can't be used to inject primitive and String valuesAd
- It requires the **less code** because we don't need to write the code to inject the dependency explicitly.
- No control of programmer.
- The default mode in traditional XML based configuration is no. The default mode in Java based @Autowired is byType.

Spring Expression Language

- A powerful expression language, used to wire values into bean's properties. It's similar to other ELs, supporting querying and manipulating an object graph at runtime.
- the spring expression language is enclosed with in

  "*#{expression language}*"
- The expression language supports the following functionality
  - Literal expressions
  - Boolean and relational operators
  - Regular expressions
  - Class expressions
  - Accessing properties, arrays, lists, maps
  - Method invocation
  - Calling constructors

contd..

- Relational operators
- Assignment
- Bean references
- Array construction
- Inline lists
- Inline maps
- Ternary operator
- Variables
- User defined functions
- Collection projection
- Collection selection
- Templated expressions

Spring DAO

Data Access

- Spring offers support for the following APIs and frameworks:
  - JDBC
  - Java Persistence API (JPA)
  - Java Data Objects (JDO)
  - Hibernate
  - Common Client Interface (CCI)
  - iBATIS SQL Maps
  - Oracle TopLink

contd..

- Various data access libraries supported by Spring have quite different implementations, still they do tend to have similar usage patterns.
- Spring takes advantage of this by providing sets of tailored support classes to aid in the building of data access logic, and specifically to aid in building DAO implementations.
- When building a DAO for a supported database access mechanism, Spring provides helper classes to aid in implementation. These usually include a template class and a DAO support class

contd..

- JDBC technology exceptions are checked, so use of try, catch blocks in the code at various places which increases the complexity of the application. Leads to writing a lot of repetitive code to perform the database operations e.g. write loading driver, connection, creating statement lot of times
- If developer opens the connection with database, developer only is responsible to close that connection. Else may get some connection issues
- JDBC framework throws error codes of the database, when ever an exception is raised. All java programmers may or may not know these codes. So the application is gonnabe database dependent

contd..

- ❖Spring framework provides one abstraction layer on top of existing JDBC technology, called as Spring-JDBC. Developers work with this abstraction layer and that layer internally uses JDBC.
- ❖Spring-JDBC layer take care about connection management and error managements, and programmers will concentrate on their logics, etc.
- ❖Spring framework provides an exception translator and it translates the checked exceptions obtained using JDBC to un-checked exceptions of Spring type and finally the un-checked exceptions are thrown to developer.
- ❖Opening and closing the database connection is taken care by the spring framework.

contd..

- Spring framework uses DataSource interface to obtain the connection with database internally.
- The two implementation classes of DataSource interface are –
  1. org.springframework.jdbc.datasource.DriverManagerDataSource
  2. org.apache.commons.dbcp.BasicDataSource
- The above **2** classes are suitable for Spring application at developing stage. In real time developers uses connection pooling service provided by the application server.
- DriverManagerDataSource is equal to DriverManager class, Spring framework internally opens a new connection and closes the connection for each operation done on the database.
- BasicDataSource is given the apache, and this is better than DriverManagerDataSource because BasicDataSource having inbuilt connection pooling implementation.

contd..

❖ **JdbcTemplate –**

❖ The JdbcTemplate class is given in package org.springframework.jdbc.core.* and provides methods for executing the SQL commands on a database

❖ JdbcTemplate class follows template design pattern, where a template class accepts input from the user and produces output to the user by hiding the interval details, provides the 3 methods –

1. execute()
2. update()
3. query().

❖ execute() and update() are for non-select operations on the database, and query() is for select operations on the database.

❖ JdbcTemplate class depends on DataSource object only. There are both setter, constructor injections in JdbcTemplate class for inserting DataSource object.

contd..

```
<bean id=”id1”
class=”org.springframework.datasource.DriverManagerDataSource”>
      <property name=”driverClassName” value=”” />
     <property name=”url” value=”” />
     <property name=”username” value=”” />
     <property name=”password” value=”” />
 </bean>
 <bean id="id2" class="org.springframework.jdbc.core.JdbcTemplate">
      <constructor-arg ref="id1" />
 <bean>
```

contd..

```
<bean id=”id1”
class=”org.springframework.datasource.DriverManagerDataSource”>
      <property name=”driverClassName” value=”” />
     <property name=”url” value=”” />
     <property name=”username” value=”” />
     <property name=”password” value=”” />
 </bean>
 <bean id="id2" class="org.springframework.jdbc.core.JdbcTemplate">
      <property name="dataSource" ref="id1" />
 <bean>
```

Spring Jdbc API

- Row Mapper – An interface used by **JdbcTemplate** for mapping rows of a ResultSet on a per-row basis. ... **RowMapper** objects are typically stateless and thus reusable; used to implementing row-mapping logic in a single place. Usage

Step 1 – Create a JdbcTemplate object using a configured datasource.

Step 2 – Create a StudentMapper object implementing RowMapper interface.

Step 3 – Use JdbcTemplate object methods to make database operations while using POJO class object.

E.g. String SQL = "select * from Emp";

List <Emp> elist= jdbcTemplateObject.query(SQL, new EmpMapper());

jdbcTemplateObject – EmpJDBCTemplate object to read student records from database.

EmpMapper – EmpMapper object to map emp records to Emp objects.

contd..

- BeanRowMapper – RowMapper implementation that converts a row into a new instance of the specified mapped target class. The mapped target class must be a top-level class and it must have a default or no-arg constructor.
- Column values are mapped based on matching the column name as obtained from result set meta-data to public setters for the corresponding properties.
- The names are matched either directly or by transforming a name separating the parts with underscores to the same name using "camel" case.
- Mapping is provided for fields in the target class for many common types, e.g.: String, boolean, Boolean, byte, Byte, short, Short, int, Integer, long, Long, float, Float, double, Double, BigDecimal, java.util.Date, etc.

contd..

- The queryForObject() method executes an SQL query and returns a result object. The result type is specified in the arguments.

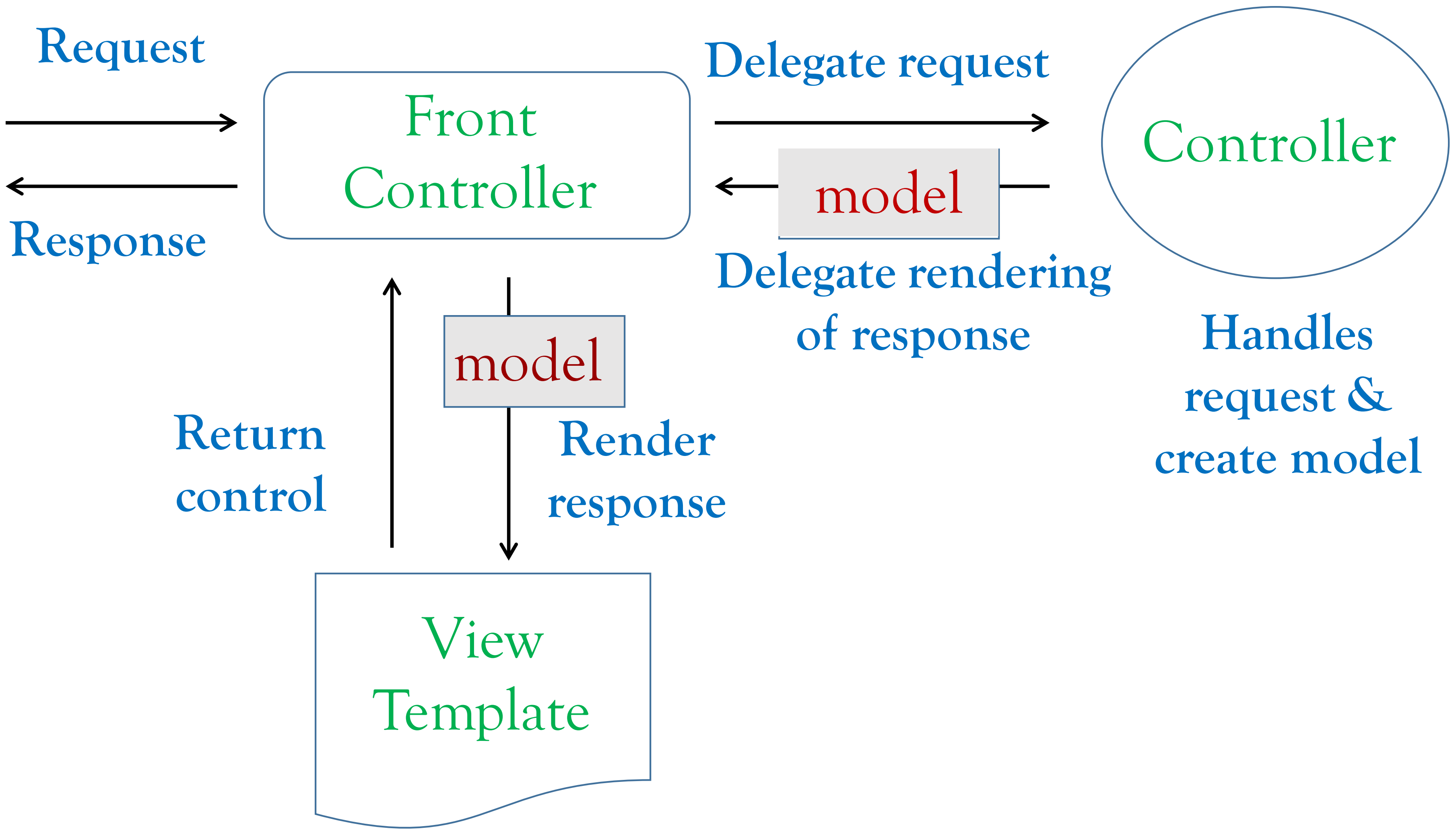
Spring Web-MVC

Front Controller Pattern

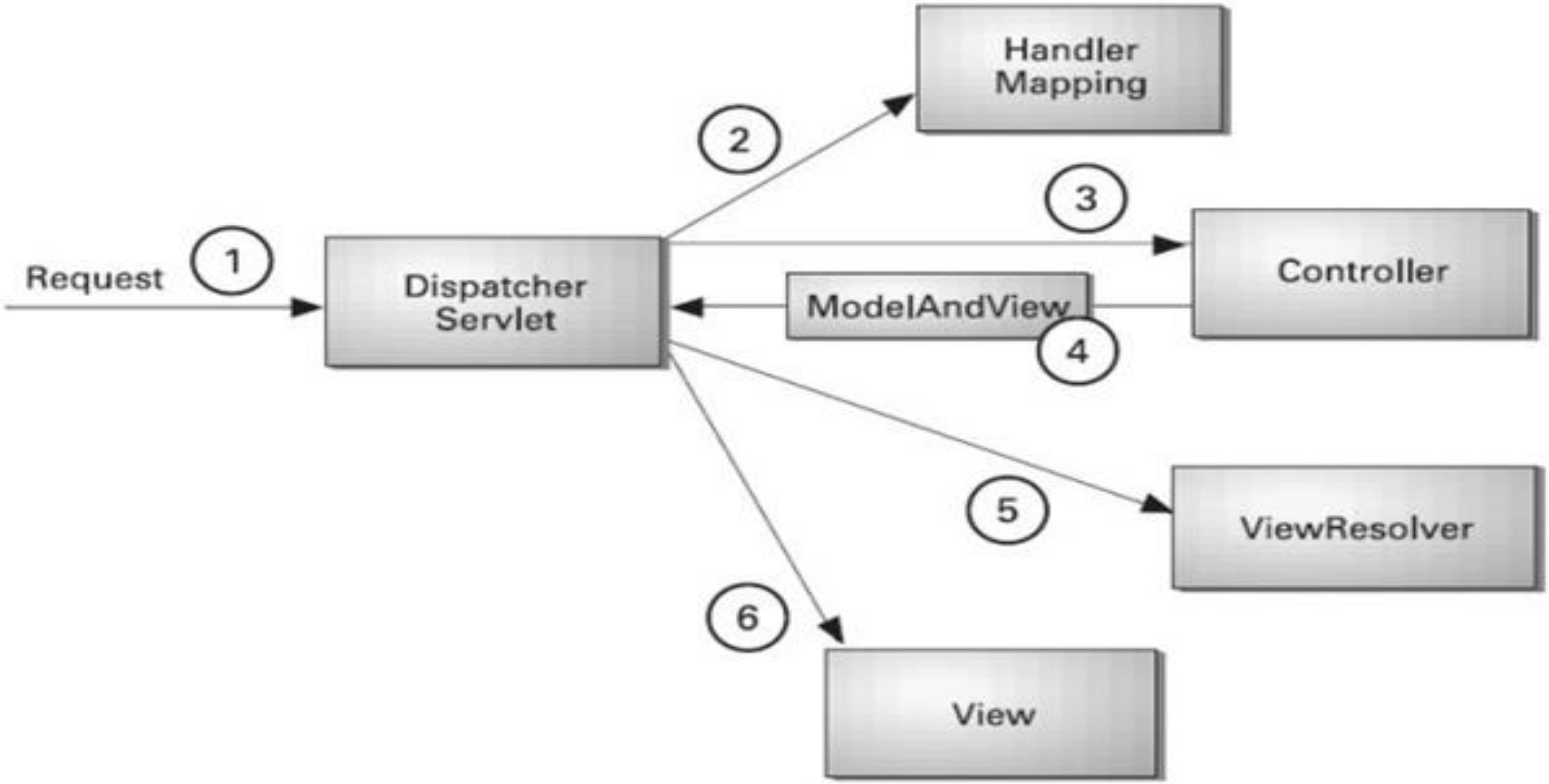
- ✓ **Controller :** is the initial contact point for handling all requests in the system; the controller may then delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval.
- ✓ **View:** represents and displays information to the client; the view retrieves information from a model. Helpers support views by encapsulating and adapting the underlying data model for use in the display.
- ✓ **Dispatcher:** responsible for view management and navigation, managing the choice of the next view to present to the user, and providing the mechanism for vectoring control to this resource.
- ✓ **Helper :** responsible for helping a view or controller complete its processing; helpers gather data required by the view and store the intermediate model, in which case the helper is sometimes referred to as a value bean.

contd..

- High level diagram depicting Spring Web Flow



contd..


Request
1
Dispatcher Servlet
2
Handler Mapping
3
Controller
ModelAndView
4
5
ViewResolver
6
View

SpringMVC

- Spring MVC framework is request-driven, designed around a central Servlet that dispatches requests to controllers and offers other functionality that facilitates the development of web applications.
- Request routing is completely controlled by the Front Controller.
- Proven pattern shown in Core J2EE Patterns
- Components of Spring MVC are –
  - ✓ DispatcherServlet
  - ✓ Controller
  - ✓ Model
  - ✓ View

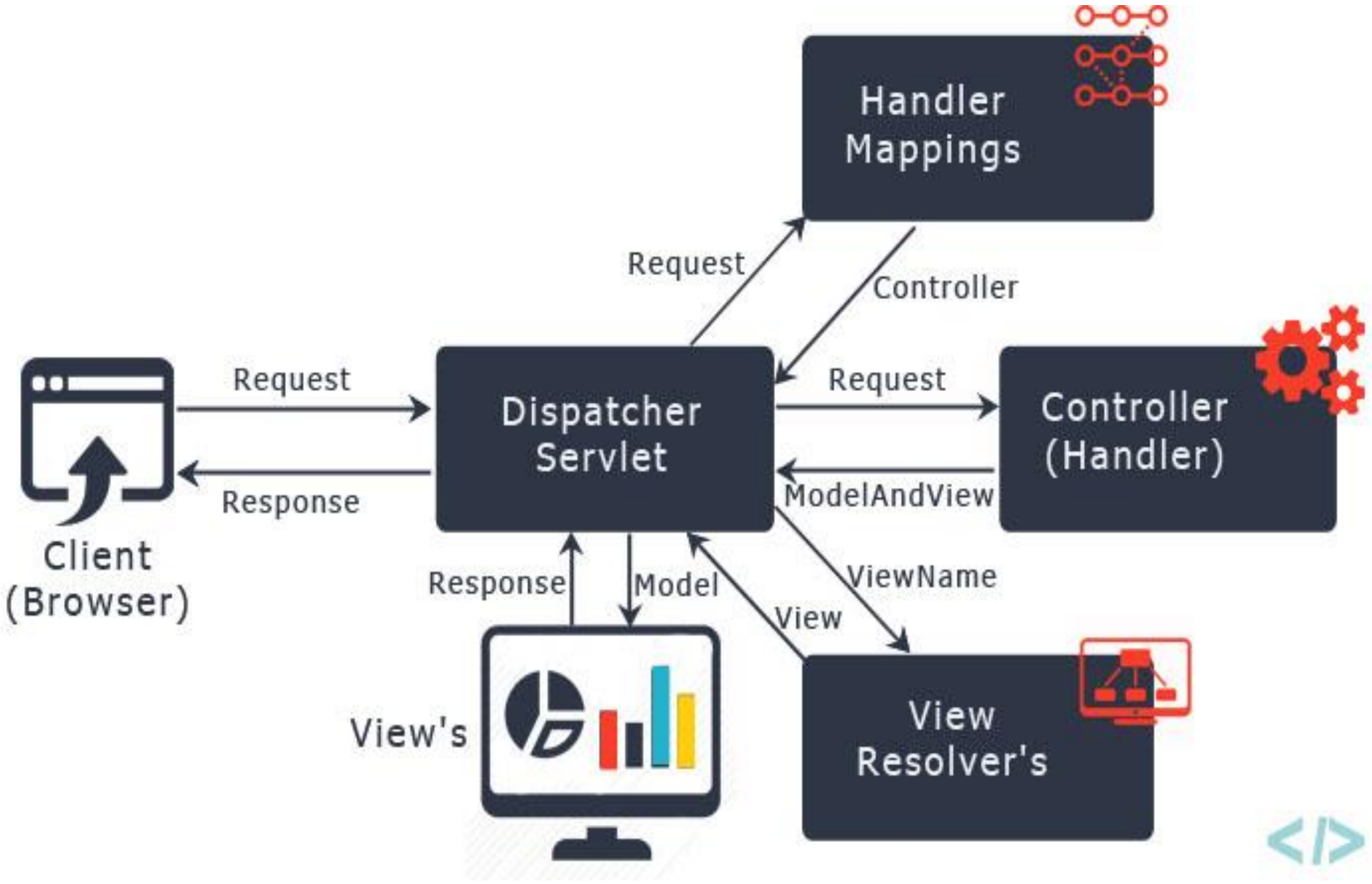
contd..

- When a client request is given to DispatcherServlet, it performs the following operations:
- Types of Phases :
  - ✓ Prepare the request context
  - ✓ Locate the handler
  - ✓ Execute interceptors with prehandler methods
  - ✓ Invoke handler
  - ✓ Execute interceptors with post hanlder methods
  - ✓ Handle Exceptions
  - ✓ Render the view
  - ✓ Execute interceptors after completion methods

contd..

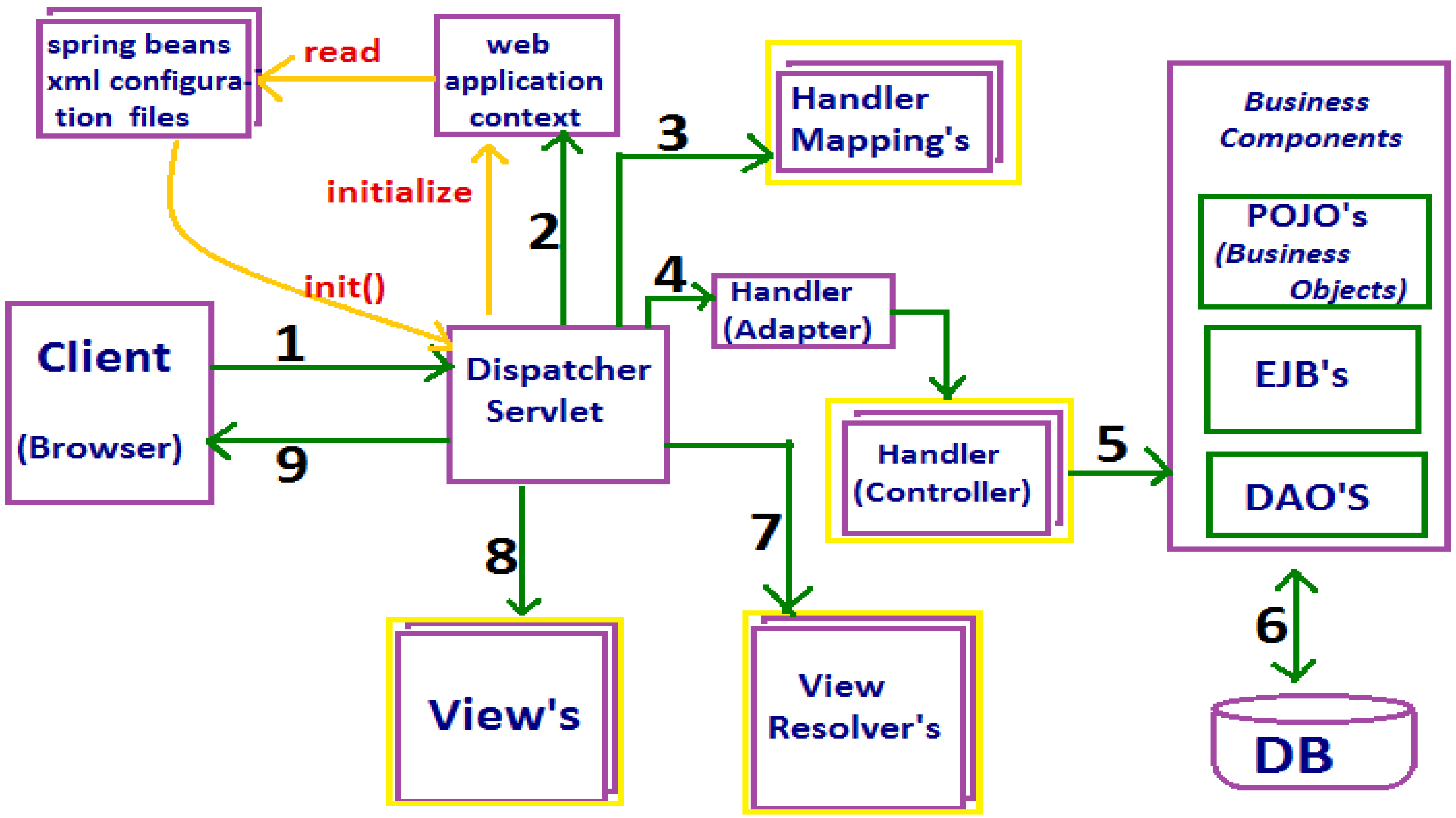
- View
  - ✓ Responsible for rendering output
  - ✓ View name resolution is highly configurable through file extension or
  - ✓ Accept header content type negotiation, through bean names, a properties file, or even a custom ViewResolver implementation.
- Model
  - ✓ is a Map interface, which allows for the complete abstraction of the view technology.
  - ✓ Also possible to integrate directly with template based rendering technologies such as JSP,
  - ✓ The model Map is simply transformed into an appropriate format, such as JSP request attributes,

contd..


Handler Mappings
Request
Controller
Client (Browser)
Request
Response
Dispatcher Servlet
Request
ModelAndView
Controller (Handler)
Response
Model
View
ViewName
View's
View Resolver's

contd..

- Detailed diagram depicting Spring Web Flow



Spring Web MVC flow

- In summary, here is the flow of an HTTP request in Java application created using the Spring MVC framework:

1) The client sends an HTTP request to a specific URL

2) DispatcherServlet of Spring MVC receives the request

2) It passes the request to a specific controller depending on the URL requested
using @Controller and @RequestMapping annotations.

3) Spring MVC Controller then returns a logical view name and model to DispatcherServlet.

contd..

4) DispatcherServlet consults view resolvers until actual View is determined to render the output

5) DispatcherServlet contacts the chosen view (like Thymeleaf, Freemarker, JSP) with model data and it renders the output depending on the model data

6) The rendered output is returned to the client as a response

contd..

1. The incoming request is intercepted by the DispatcherServlet that works as the front controller.
2. The DispatcherServlet gets an entry of handler mapping from the XML file and forwards the request to the controller.
3. The controller returns an object of ModelAndView.
4. The DispatcherServlet checks the entry of view resolver in the XML file and invokes the specified view component.

DispatcherServlet

- **DispatcherServlet** is at the heart of Spring MVC. – Spring's Front Controller implementation, completely integrated with the Spring IoC container.
- **DispatcherServlet** inherits **HttpServlet** base class and is declared in the web.xml of the application.
- On initialization of a DispatcherServlet, Spring MVC looks for a file named *[servlet-name]-servlet.xml* in the WEB-INF directory and creates the beans defined there. Any definitions of beans with the same name in the global scope are overridden.

contd..

- The DispatcherServlet dispatches requests to handlers, with
  - ✓ configurable handler mappings,
  - ✓ view resolution,
  - ✓ locale,
  - ✓ timezone and theme resolution as well as support for uploading files.
- The **requests** needed to be **handled** by the DispatcherServlet should be mapped by using a URL mapping. The default handler is based on the **@Controller** and **@RequestMapping** annotations.
- Individual Controllers are used to handle many different URLs

contd..

- **DispatcherServlet Processing Sequence**

- On getting a request for a specific DispatcherServlet, request processing starts as follows:

1. 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.

2. a. 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, etc)

b. The **theme resolver** is bound to the request to let elements such as views determine which theme to use.

contd..

c. If a **multipart file resolver** is specified, 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.

3. An appropriate **handler** is searched for. If a handler is found, the execution chain associated with the handler – **preprocessors, postprocessors, & controllers** is executed in order to prepare a **model** or rendering.
4. If a model is returned, the **view** is rendered. If no model is returned, (may be a pre or postprocessor intercepting the request,), then view is not rendered, as the request could already have been fulfilled.

HandlerMapping

❖ **Handler mappings**

✓ HandlerMapping automatically looks for **@RequestMapping** annotations on all **@Controller** beans.

✓ The HandlerMapping classes extending from AbstractHandlerMapping have the following properties –

1. **defaultHandler** – used when this handler mapping does not result in a matching handler.
2. **order** – based on the value of the order property. Spring sorts all handler mappings in the context and applies the first matching handler
3. **alwaysUseFullPath – if** true, the full path within the current Servlet context is used to find an appropriate handler; else the path within the current Servlet mapping is used. (default is false)
4. **urlDecode** Defaults to true, the HttpServletRequest always exposes the Servlet path in decoded form.

@RequestMapping

- **@RequestMapping** The org.springframework.web.bind.annotation.RequestMapping annotation is used to map web requests onto specific handler classes and/or handler methods. @RequestMapping can be applied to the **controller class** as well as **methods.**

1. **@RequestMapping with Class:**

```
@Controller
@RequestMapping("/home")
public class HomeController
    { }
```

/home is the URI for which this controller will be used. This concept is very similar to servlet context of a web application.

contd..

2. **@RequestMapping** with **Method** – used with method to provide the URI pattern for which handler method will be used.

```
@RequestMapping(value="/method0")
@ResponseBody
 public String method0()
   { return "method0"; }
```

Above annotation can be written as @RequestMapping("/method0")

@ResponseBody is to send the String response for this web request.

3.@RequestMapping with **Multiple URI** – a single method can handle multiple URIs,
@RequestMapping(value={"/method1","/method1/second"}) @ResponseBody public String method1(){ return "method1"; } Its possible to create String array for the URI mappings for the handler method.

contd..

4. **@RequestMapping** with **HTTP Method** – In situations where different operations are to be performed based on the HTTP method used, even though request URI remains same. The @RequestMapping method variable can narrow down the HTTP methods for which this method will be invoked.

```
@RequestMapping (value="/method2",
                            method=RequestMethod.POST )
@ResponseBody  public String method2()
                { return "method2"; }
@RequestMapping (value="/method3",
                method={RequestMethod.POST,RequestMethod.GET})
@ResponseBody  public String method3()  { return "method3"; }
```

contd..

5. @RequestMapping with **Headers** – the headers that should be present to invoke the handler method can be specified.

```
@RequestMapping(value="/method5",
                        headers={"name=Rohit", "id=1001"})
   @ResponseBody
    public String method5() { return "method5"; }
```

6.@RequestMapping with **Produces and Consumes** – @RequestMapping provides **produces** and **consumes** variables which can specify the request content-type for which method will be invoked and the response content type.

The header Content-Type and Accept are used to find out request contents and what is the mime message it wants in response.

```
@RequestMapping( value="/method6",
produces={"application/json","application/xml"}, consumes="text/html")
@ResponseBody  public String method6() { return "method6"; }
```

contd..

7.@RequestMapping with **@PathVariable**: RequestMapping annotation can be used to handle dynamic URIs where one or more of the URI value works as a parameter. Even regular expression can be specified for URI dynamic parameter to accept only specific type of input.

The @PathVariable annotation maps the URI variable to one of the method arguments.

```
@RequestMapping(value="/method7/{id}")
@ResponseBody
    public String method7(@PathVariable("id") int id)
        { return "method7 with id="+id; }
@RequestMapping(value="/method8/{id:[\\d]+}/{name}")
@ResponseBody public String method8(@PathVariable("id") long id,
@PathVariable("name") String name)
{ return "method8 with id= "+id+" and name="+name; }
```

contd..

8. @RequestMapping with **@RequestParam for URL parameters – @RequestParam** is used to retrieve the URL parameter and map it to the method argument.

```
@RequestMapping (value="/method9")

@ResponseBody

public String method9(@RequestParam("id") int id)

        { return "method9 with id= "+id; }
```

For this method to work, the parameter name should be "id" and it should be of type int.

contd..

9.@RequestMapping default method – If value is empty for a method, it works as default method for the controller class.

@RequestMapping()
@ResponseBody
public String defaultMethod(){ return "default method"; }

As seen above, the /home is mapped to HomeController, this method will be used for the default URI requests.

10.@RequestMapping fallback method – a fallback method for the controller class is created to catch all the client requests even though there are no matching handler methods. It is useful in sending custom 404 response pages to users when there are no handler methods for the request.

@RequestMapping("*")

@ResponseBody public String fallbackMethod(){ return "fallback method"; }

Controller

❖ **Controller**

- ✓ user created component for handling requests.
- ✓ they interpret user input and transform it into a model that is represented to the user
- ✓ encapsulates navigation logic and delegate to the service objects for business logic
- ✓ responsible for preparing a model Map with data and selecting a view name
- ✓ can also write directly to the response stream and complete the request.
- ✓ provide access to the application behavior that is defined through a service interface.
- ✓ **@Controller** annotation indicates that a particular class serves the role of a controller.

contd..

- ✓ Spring implements a controller in a very abstract way, which enables developer to create a wide variety of controllers.
- ✓ annotations used are @RequestMapping, @RequestParam, @ModelAttribute;
- ✓ Controllers implemented in this style do not have to extend specific base classes or implement specific interfaces.
- ✓ Controllers do not usually have direct dependencies on Servlet APIs, although can be easily configured to access Servlet facilities.
- ✓ The @Controller annotation acts as a stereotype for the annotated class, indicating its role. The dispatcher scans such annotated classes for mapped methods and detects @RequestMapping annotation.

contd..

- ✓ The annotated controller beans can be defined explicitly, using a standard Spring bean definition in the dispatcher's context.
- ✓ Auto–detection is also possible. The @Controller stereotype aligned with Spring general support provides for detecting component classes in the classpath and auto–registering bean definitions for them.
- ✓ Add component scanning to the configuration by using the *spring-context* schema.

Model

- **Model**
- The model can supply attributes used for rendering views.
- To provide a view with usable data, simply add this data to its *Model* object.
- The core Spring packages for Model are –
  1. org.springframework.ui.Model,
  2. org.springframework.ui.ModelMap &
  3. org.springframework.web.servlet.ModelView
- Additionally, maps with attributes can be merged with *Model* instances:

contd..

1. org.springframework.ui.Model – the Model can supply attributes used for rendering views.
   - ❖ To provide a view with usable data, simply add this data to its *Model* object.
   - ❖ Additionally, maps with attributes can be merged with *Model* instances:
2. org.springframework.ui.ModelMap – *ModelMap* is also used to pass values to render a view.
   - ❖ *ModelMap* provides for a collection of values to be passed and treat these values as if they were within a *Map*.
3. org.springframework.web.servlet.ModelView – The final interface to pass values to a view is the *ModelAndView*.
   - ❖ This interface allows us to pass all the information required by Spring MVC in one return.

contd..

4. **<u>ModelAndView</u>** – Holder for both Model and View in the web MVC framework.
   - ❖ This class merely holds both to make it possible for a controller to return both model and view in a single return value.
   - ❖ Represents a model and view returned by a handler, to be resolved by a DispatcherServlet.
   - ❖ The view can take the form of a String view name which will need to be resolved by a ViewResolver object;
   - ❖ alternatively a View object can be specified directly. The model is a Map, allowing the use of multiple objects keyed by name.

contd..

- **The Model ModelMap (ModelAndView)**
- The ModelMap class is essentially a Map that can make adding objects that are to be displayed in (or on) a View adhere to a common naming convention.
- The ModelAndView class uses a ModelMap class that is a custom Map implementation that automatically generates a key for an object when an object is added to it.

contd…

❖**Validation**

- ✓Spring provides a **Validator** interface that can be used for validation in all layers of an application.
- ✓In Spring MVC it can configured for use as a global Validator instance, to be used whenever an @Valid or @Validated controller method argument is encountered, and/or
- ✓as a local Validator within a controller through an @InitBinder method. Global and local validator instances can be combined to provide composite validation.

contd..

❖ There are three important configurations.

a. **annotation-driven** tells DispatcherServlet to look for Controller classes using @Controller annotation.

b. **context:component-scan** tells DispatcherServlet where to look for controller classes.

c. **InternalResourceViewResolver** bean configuration to specify location of view pages and suffix used. Controller class methods return name of the view page and then suffix is added to figure out the view page to use for rendering the response.

Views

❖ Resolving views

✓Spring provides view resolvers, which enable user to render models in a browser without tying user to a specific view technology.

✓Out of the box, Spring enables you to use JSPs, Velocity templates and XSLT 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.

contd..

- ✓Implementing Controllers, all handler methods in the Spring Web MVC controllers must resolve to a logical view name, either explicitly (e.g., by returning a String, View, or ModelAndView) or
- ✓implicitly (i.e., based on conventions). Views in Spring are addressed by a logical view name and are resolved by a view resolver. Spring comes with quite a few view resolvers.