Spring - ver 5

Topics

Sl.No.	Topic
1	Spring Core
2	Spring DAO
3	Spring MVC

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.

- In an application, classes should be as **independent** as possible as mutual independence increase the possibility to **reusability and unit test** them **independent** of other classes.

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;
    }
}
```

Problem with look–up and Unit testing

- **J2EE applications** consist of components offering transaction management, multithreading, security, etc belonging to different frame works within Java/J2EE or exterior.
- **Inter–component communication** is thru servers of different vendors.
- If component C1 requires component C2, then C1 itself is responsible for looking up the C2 as it depends upon. The **look–up** happens **by–name**, i.e. **name of the dependency is hardcoded** in the component. This approach result in **heavy weight application**.
- Services are included even when **not needed**. Developers–architects spend lot of time in **setting up configurations** using **xml files**.
- The code **cannot be unit tested**, as all kinds of services (provided by the J2EE server) are assumed to be present.
- 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.

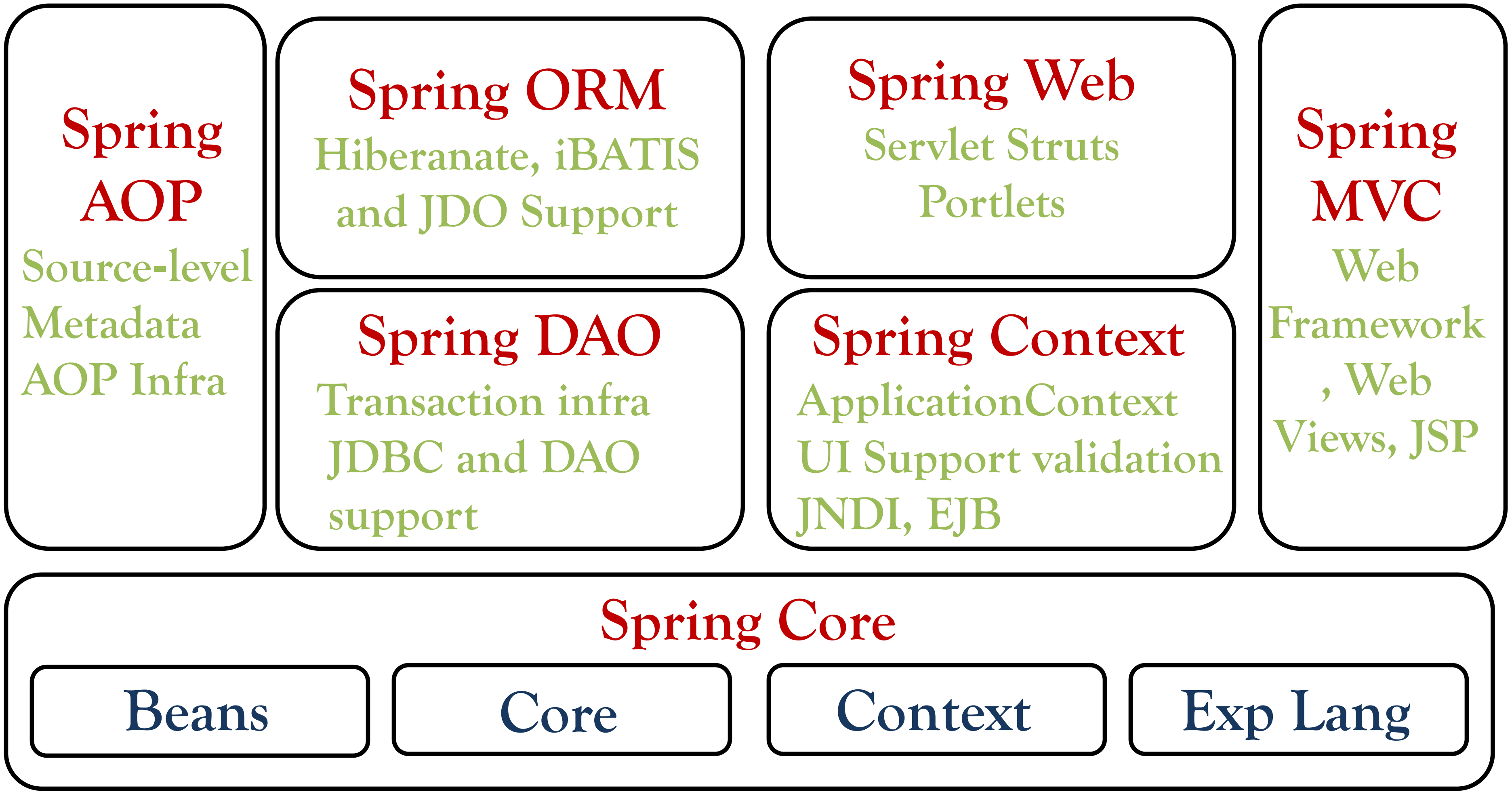
contd..

- **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 Introduction

- ❖ A J2EE Framework developed by **Rod Johnson.**
- ❖ 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 to manage the business objects & their dependencies with simple POJO's.
- ❖ Spring framework offers –
  - ✓ Inversion of Control, Dependency Injection and AOP including Spring's declarative management.
  - ✓ Foundational support for JDBC, JPA, JMS providing easy integration with other frameworks
  - ✓ Spring MVC web application and RESTful web service framework.

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

- Developing an application, involves **dependencies** between and on components, services, classes etc. which are '**wired**' together on the spot where needed.
- When user needs a **different implementation** of the **dependency,** it costs huge more so if change is in implementation context.
- With IOC, an external party the **CONTAINER** manages the wiring from outside. Hence the name **Inversion of Control.**
- It is also termed **Dependency Injection** because the container '**injects**' the dependencies instead of developer managing them.
- The framework **reads configuration information** and invokes the **appropriate methods** on the code to inject the dependencies.
- 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.

Sample application to demo D.I.

❖ public class EmailService

```
{
    public void sendEmail(String message, String receiver)
    { //logic to send email
        System.out.println("Email sent to "+receiver+ " with
        Message="+message);
     }
}
```

❖ public class MyApplication

```
{  private EmailService email = new EmailService();
  public void processMessages(String msg, String rec)
        {    //do some msg validation, manipulation logic etc
    this.email.sendEmail(msg, rec);
  }   }
```

contd..

```
❖ public class MyLegacyTest
  {
     public static void main(String[] args)
      {
        MyApplication app = new MyApplication();
        app.processMessages("Hi Pankaj","pankaj@abc.com");
      }
  }
```

Limitations of Code Logic

- ❖ MyApplication class is responsible to initialize the email service and then use it. This leads to hard-coded dependency.

- ❖ To switch to some other advanced email service in the future, it will require code changes in MyApplication class. This makes application hard to extend and if email service is used in multiple classes then that would be even harder.

- ❖ To extend the application to provide an additional messaging feature, such as SMS or Facebook message, will require to write another application for that. This will involve code changes in application classes and in client classes too.

- ❖ Testing the application will be very difficult since the application is directly creating the email service instance. There is no way we can mock these objects in our test classes.

Concept of Dependency and Injection

- Dependency Injection in Java requires at least the following:

1. Service components should be designed with base class or interface. It's better to prefer interfaces or abstract classes that would define contract for the services.
2. Consumer classes should be written in terms of service interface.
3. Injector classes that will initialize the services and then the consumer classes.

contd..

- MessageService - the contract for service implementations.

```
❖   public interface MessageService
     {  void sendMessage(String msg, String rec);
        }

❖   public class EmailServiceImpl implements MessageService
     {  @Override
        public void sendMessage(String msg, String rec)
         {  //logic to send email
            System.out.println("Email sent to "+rec+ " with
                                              Message="+msg);
         }
       }
```

contd..

```
public class SMSServiceImpl implements MessageService
  {  @Override
     public void sendMessage(String msg, String rec)
     { //logic to send SMS
        System.out.println("SMS sent to "+rec+ " with
                                   Message="+msg);
      }
  }
```

contd..

- public interface Consumer

```
 {  void processMessages(String msg, String rec);
     }
```

- public class MyDIApplication implements Consumer

```
 { private MessageService service;
   public MyDIApplication(MessageService svc)
    { this.service=svc; }
 @Override
    public void processMessages(String msg, String rec)
    { //do some msg validation, manipulation logic etc
        this.service.sendMessage(msg, rec);
    }
 }
```

contd..

-
  ```
  public interface MessageServiceInjector
  {   public Consumer getConsumer();
    }
  ```

-
  ```
  public class EmailServiceInjector implements MessageServiceInjector
  {   @Override
         public Consumer getConsumer()
          {    return new MyDIApplication(new EmailServiceImpl());
            }  }
  ```

-
  ```
  public class SMSServiceInjector implements MessageServiceInjector
  {  @Override
      public Consumer getConsumer()
        {   return new MyDIApplication(new SMSServiceImpl());
            }  }
  ```

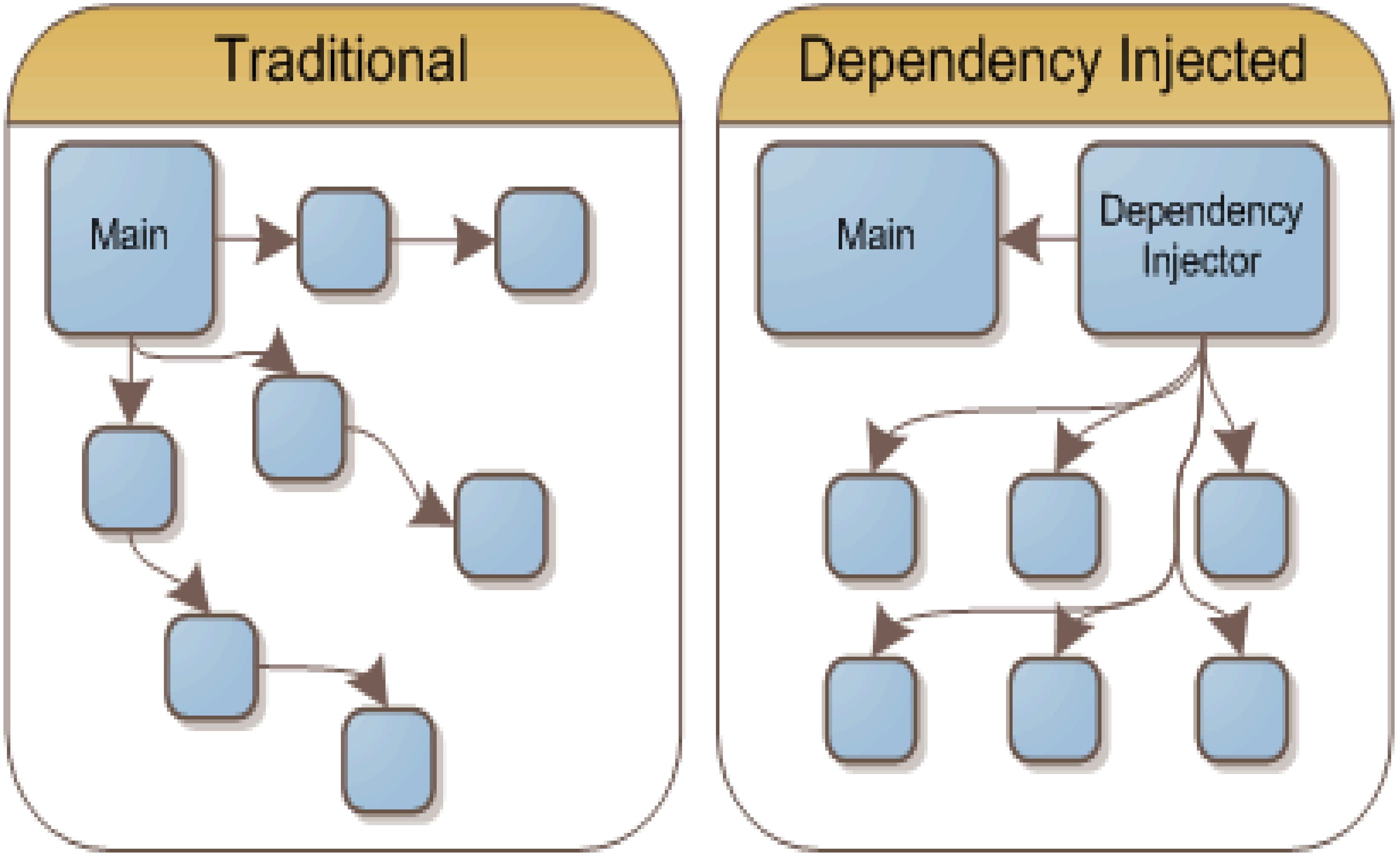
contd..

```
public class MyMessageDITest
 {
    public static void main(String[] args)
     {  String msg = "Hi Pankaj";
        String email = "pankaj@abc.com";
        String phone = "4088888888";
        MessageServiceInjector injector = null;
        Consumer app = null;
        //Send email
        injector = new EmailServiceInjector();
        app = injector.getConsumer();
        app.processMessages(msg, email);
        //Send SMS
        injector = new SMSServiceInjector();
        app = injector.getConsumer();  app.processMessages(msg, phone); }}
```

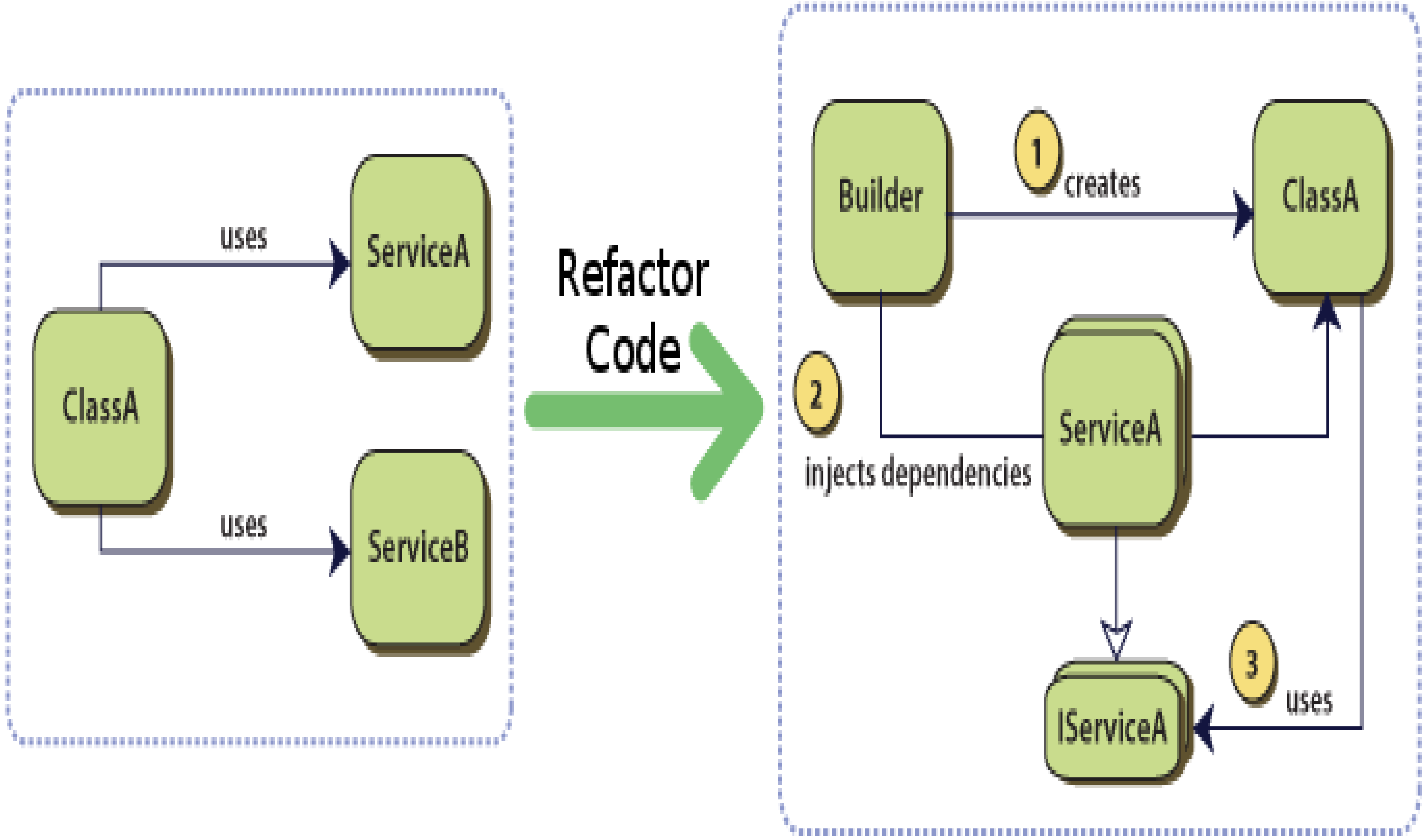
contd..

- Application classes are responsible only for using the service.
- Service classes are created in injectors.
- If the application is to be further extended to allow facebook messaging, then only write Service classes and injector classes.
- So dependency injection implementation is solved the problem with hard-coded dependency and helped in making our application flexible and easy to extend.
- This approach helps in testing our application class by mocking the injector and service classes.

Diagrammatic Representation



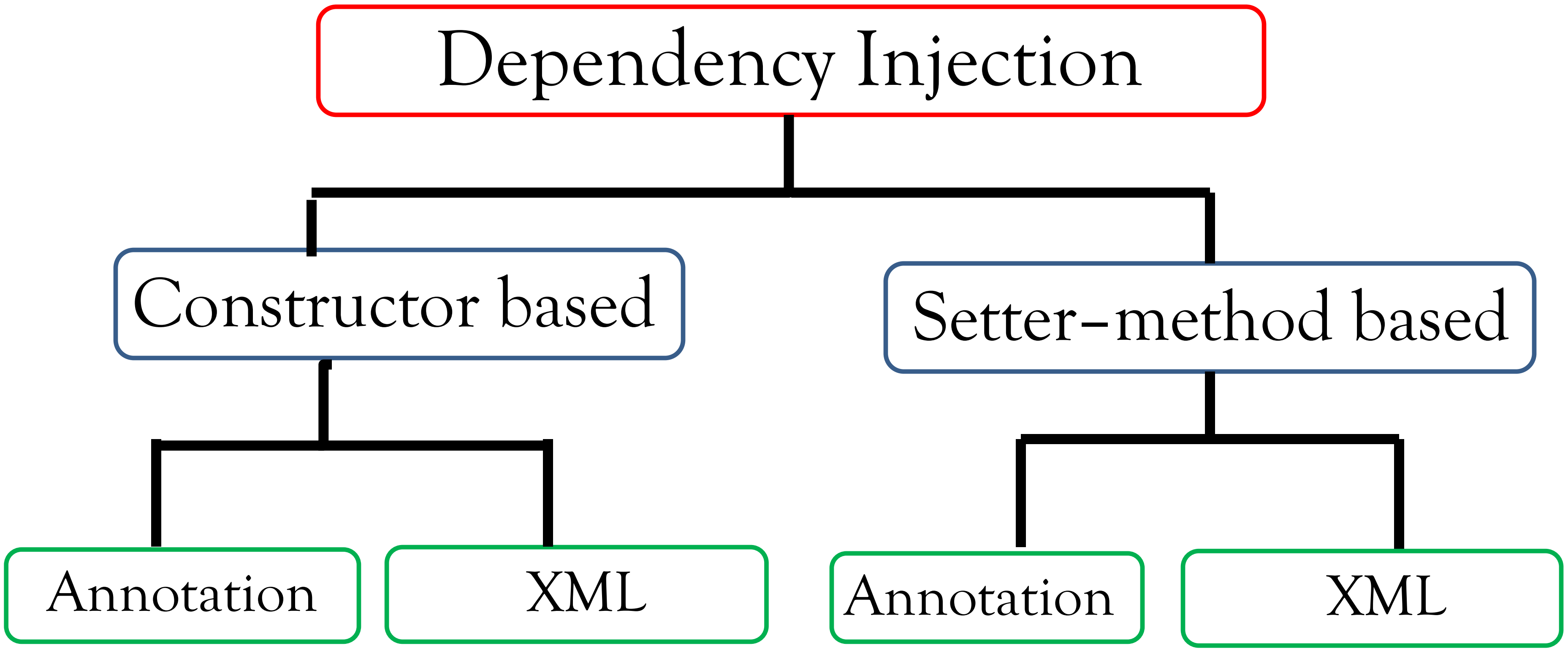
contd..


uses
ServiceA
ClassA
uses
ServiceB
Refactor Code
Builder
1
creates
ClassA
2
injects dependencies
ServiceA
IServiceA
3
uses

Dependency Injection

- **Dependency Injection (DI)** implements the design pattern called **Inversion of Control** for resolving dependencies.
- **Dependency** is an **object** on which a class depends. **Injection** refers to the process of injecting a dependency (object) into the dependent class.
- The container creates a bean, controls the instantiation or location of its dependencies by using direct construction classes or a Service Locator.
- D.I. allows Beans to define their dependencies through constructor or properties and container provides the injection at runtime thereby reduces boilerplate code.
- D.I. provides for Separation of Concerns as it decouples **object creators** and **locators** from **application logic** hence connects together classes and same time keep independent.
- Configurable components makes application easily extendable
- Unit testing is easy with mock objects

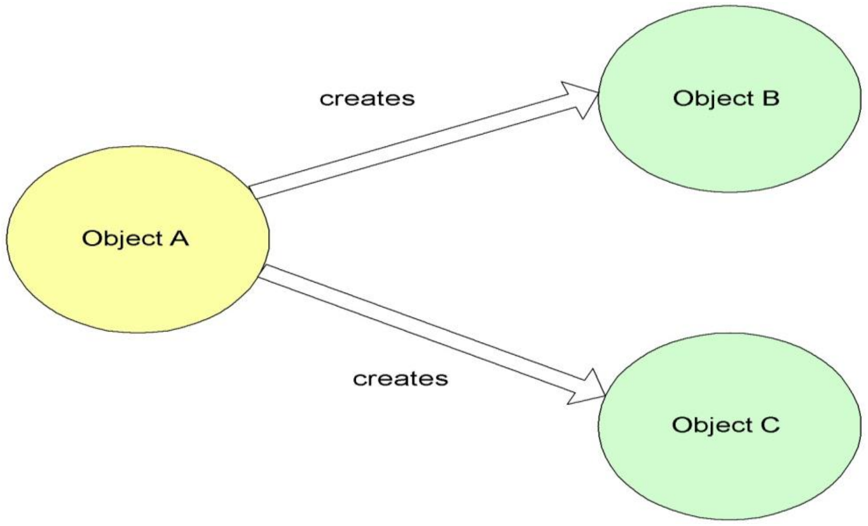
contd..


Dependency Injection
Constructor based
Setter-method based
Annotation
XML
Annotation
XML

DI types

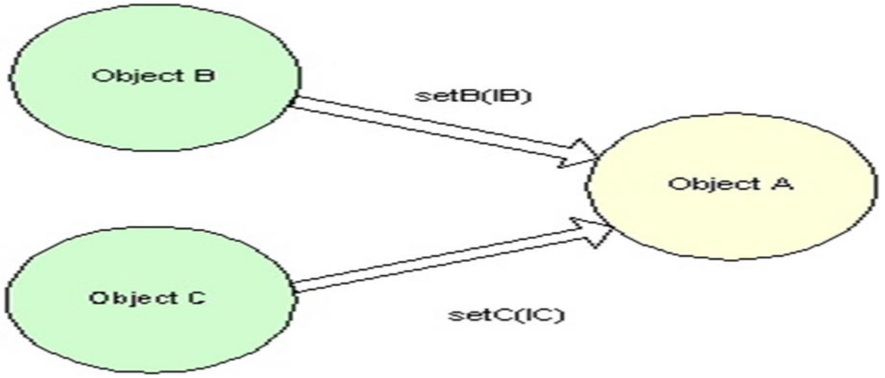
❖ In Constructor based DI –

- ✓ the dependency is injected into the class through **constructor**.
- ✓ the container invokes the class constructor with a number of arguments, each representing a dependency on other class.



❖ In Setter–method based DI,

- ✓ dependency is injected through **setter methods** of the class.
- ✓ user creates instance and the instance is used to call setter methods to access properties.
- ✓ 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 DI

- <bean> is the root tag of the xml configuration having two attributes **id** and **class**. id specifies the unique name of the object and class mentions the fully qualified class name.
- <bean> has two subelements viz.. <constructor–arg> and <property> used for constructor and setter method injection respectively. The default is type is **String**, in case no data type is mentioned
- Collection values can also be injected using these three elements.
  list, set, map.
- <!-- A simple bean definition -->

  <bean id = "..." class = "..." lazy-init = "true"init-method = "..."
    destroy-method = "...">
  <!-- collaborators and configuration for this bean go here -->
  </bean>

Annotation based DI

- The bean configuration can be moved into the component class itself by using annotations on the relevant class, method, or field declarations.
- 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..

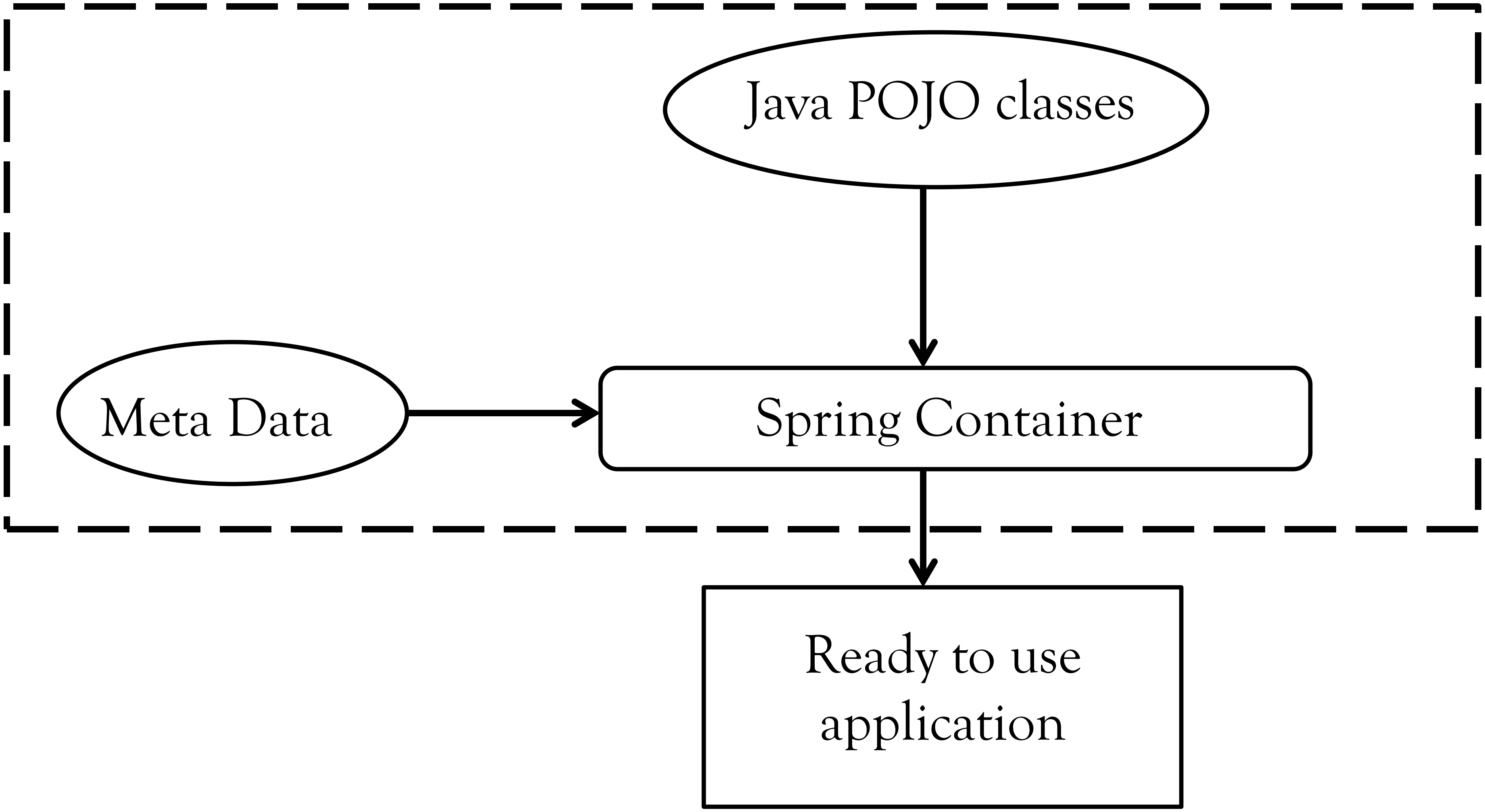
❖ Some important annotations in Spring –

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.

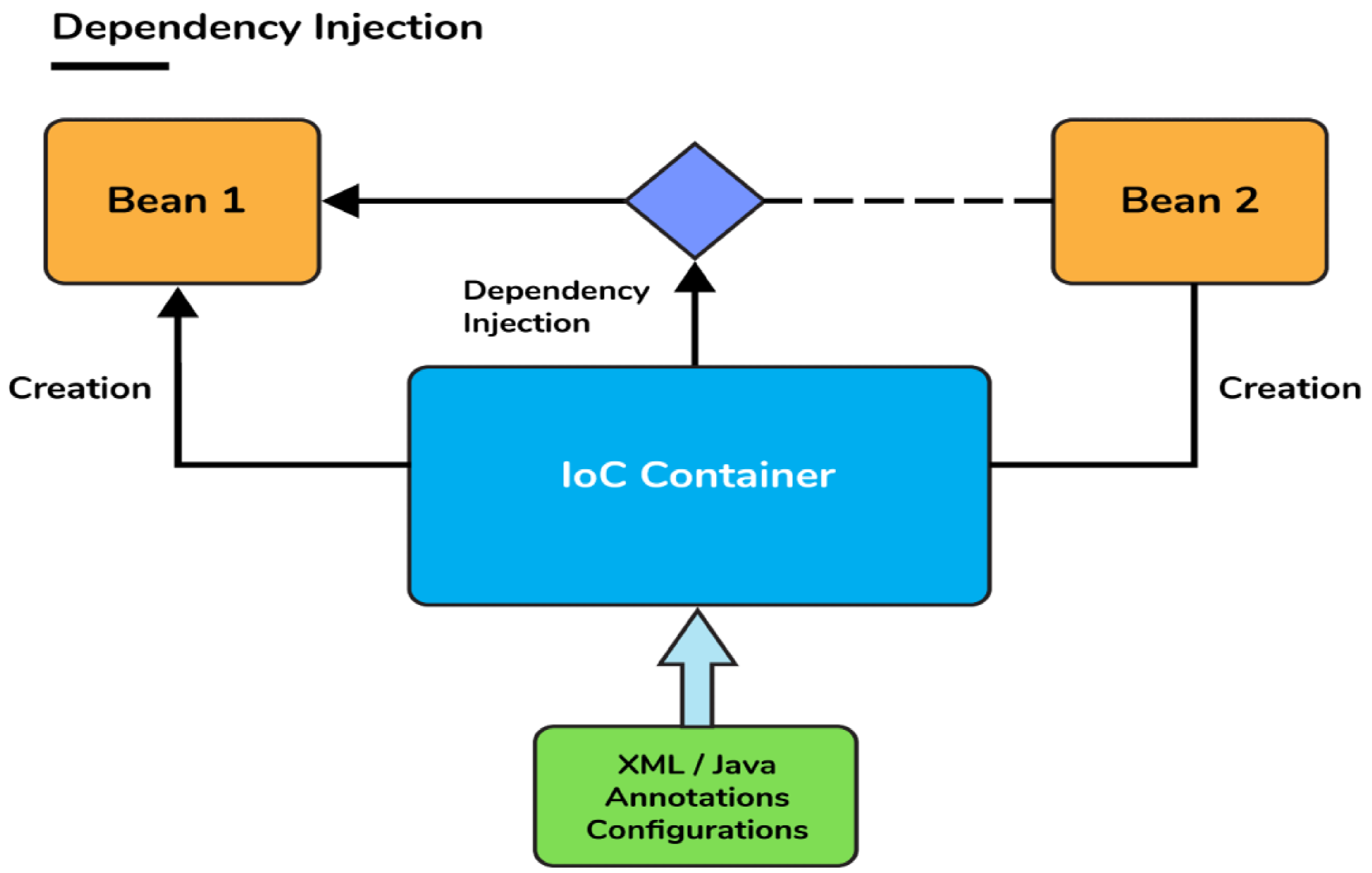
Spring Container

- ❖ The Spring container is the core of the Spring Framework and 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). However usage of Spring-containers leads to losing some of the advantages of ***static type checking***.
- ❖ The IoC container makes use of Java POJO classes and configuration metadata to produce a fully configured and executable system or application.
- ❖ Spring IoC container is totally decoupled from the format in which this configuration metadata is actually written.

contd..


Java POJO classes
Meta Data
Spring Container
Ready to use application

contd..


Dependency Injection
Bean 1
Bean 2
Dependency Injection
Creation
Creation
IoC Container
XML / Java Annotations Configurations

contd..

Sl. No.	Name	ver	Description
1.	BeanFactory		The container provides for maintaining a registry of different beans and their dependencies. Defined by the *org.springframework.beans.factory.BeanFactory* interface. It is simplest container providing basic support for DI
2.	Application Context		The 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. Defined by the ***org.springframework.context.Application Context*** interface. It includes all functionality of the *BeanFactory* container, so it is generally recommended over the *BeanFactory*.

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.	FileSystemXml Application Context		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.	ClassPathXml Application Context		this container loads the definitions of the beans from an XML file. The container will look bean configuration XML file in CLASSPATH.
4.	WebXml Application Context		this container loads the XML file with definitions of all beans from within a web application.

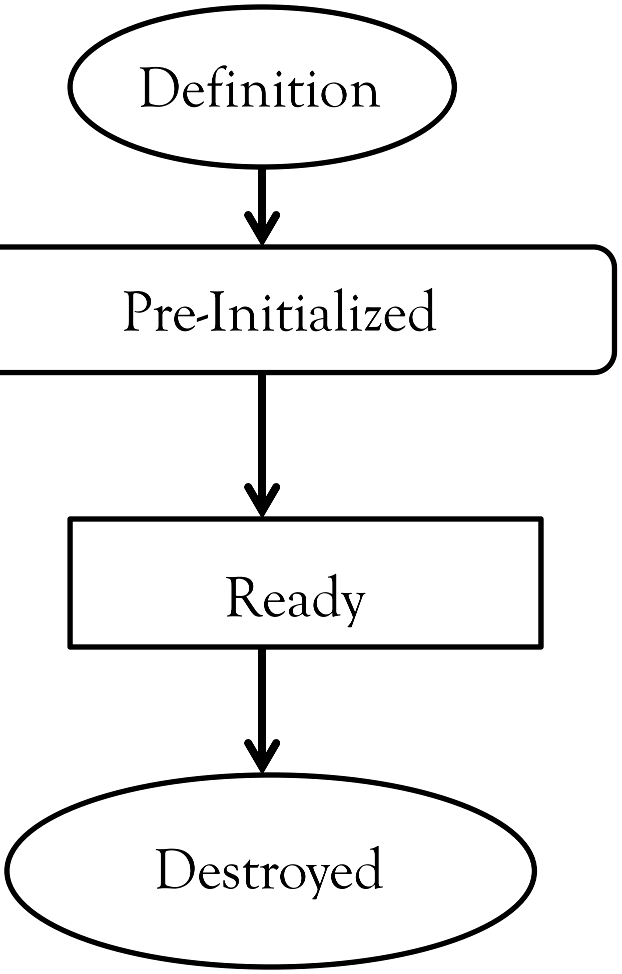
contd..

- **AnnotationConfigApplicationContext** – the container used for annotaion based configurations. It is a standalone application context which accepts annotated classes as input.
  E.g. @Configuration or @Component.
- It was introduced since Spring ver 3.0.
- Beans can be looked up with scan() or registered with register().
- Allows for registering classes one by one using register(Class..) as well as for classpath scanning using scan(String..).
- In case of multiple @Configuration classes, @Bean methods defined in later classes will override those defined in earlier classes. This can be leveraged to deliberately override certain bean definitions via an extra @Configuration class.

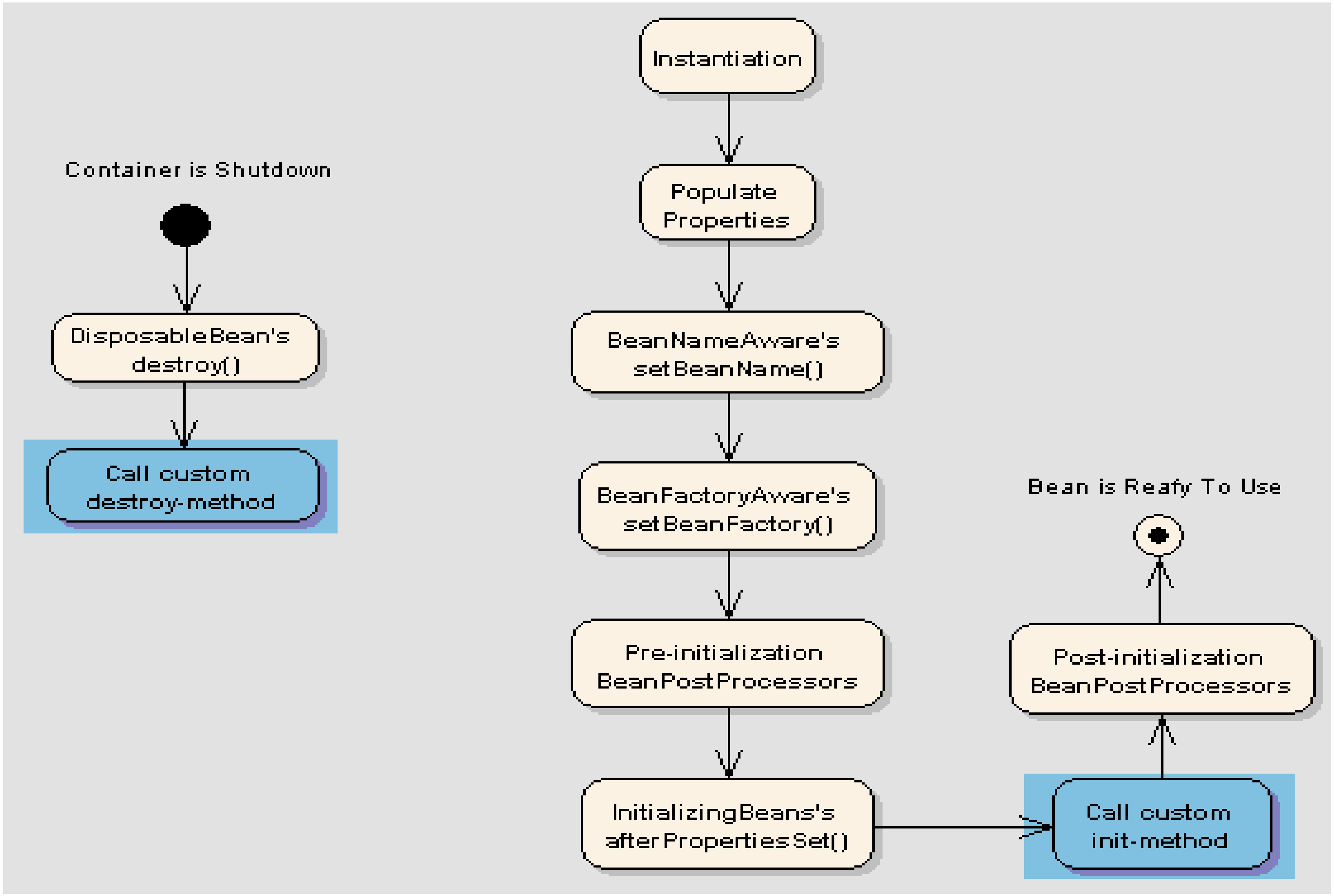
Spring Bean

- Spring bean is an object that is **instantiated, assembled,** and **managed** by a **Spring IoC container**.
- The bean definition has info called **configuration metadata** thats needed for container to know – **how to create a bean, bean's lifecycle details and bean's dependencies.** The configuration metadata is in the form of xml or annotation.
- When bean is instantiated, it may be required to perform some initialization to get it into a usable state and no longer required, it 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 **init-method** and/or **destroy-method** parameters. (xml config or annotations config). Both of these dependencies should be provided via injection.

Bean Life Cycle



contd..



Container is Shutdown
DisposableBean's destroy()
Call custom destroy-method
Instantiation
Populate Properties
BeanNameAware's setBeanName()
BeanFactoryAware's setBeanFactory()
Pre-initialization BeanPostProcessors
InitializingBeans's afterPropertiesSet()
Call custom init-method
Post-initialization BeanPostProcessors
Bean is Reafy To Use

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.

Inheritance in Spring Code

- In case of xml configuration, the <bean> element attributes **'parent'** or **'abstract'** are used to indicate **super** or **abstract class.**
  - ✓ <bean id = "..." class = "..." parent = "...">
    <!-- collaborators and configuration for this bean go here -->
    </bean>
  - ✓ <bean id = "..." class = "..." abstract = "...">
    <!-- collaborators and configuration for this bean go here -->
    </bean>
- Spring **does not provide any annotation** corresponding to the parent or abstract attributes of <bean> element xml config. The bean definition inheritance is hard coded. The child bean definition also can add new values.
- Child bean definition will inherit constructor argument values, property values, and method from the parent. Child bean init-method, destroy-method, static factory method settings, override parent corresponding settings.'
- The settings that will always be taken from the child definition are **depends on, autowire mode, dependency check, singleton, scope, lazy init.**

AutoWiring

- The **auto–wiring** feature is such with which there is no need for ref attribute and/or provide bean references explicitly. The auto–wire mode automatically wires the beans.
- The autowire mode can be configured using "autowire" attribute in case of xml configuration

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

  In case of annotation configuration use **@Autowired** annotation.
- 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	The property name is used for searching a matching bean definition in configuration file. If found, it is injected in property else a error is raised.
2.	by Type	The property's class type is used for searching a matching bean definition in configuration file. If found, it is injected in property else a error is raised.
3.	constr uctor	Similar to byType, but applies to constructor args. It will look for class type of constructor arguments, and then do a 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 values.
- 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.

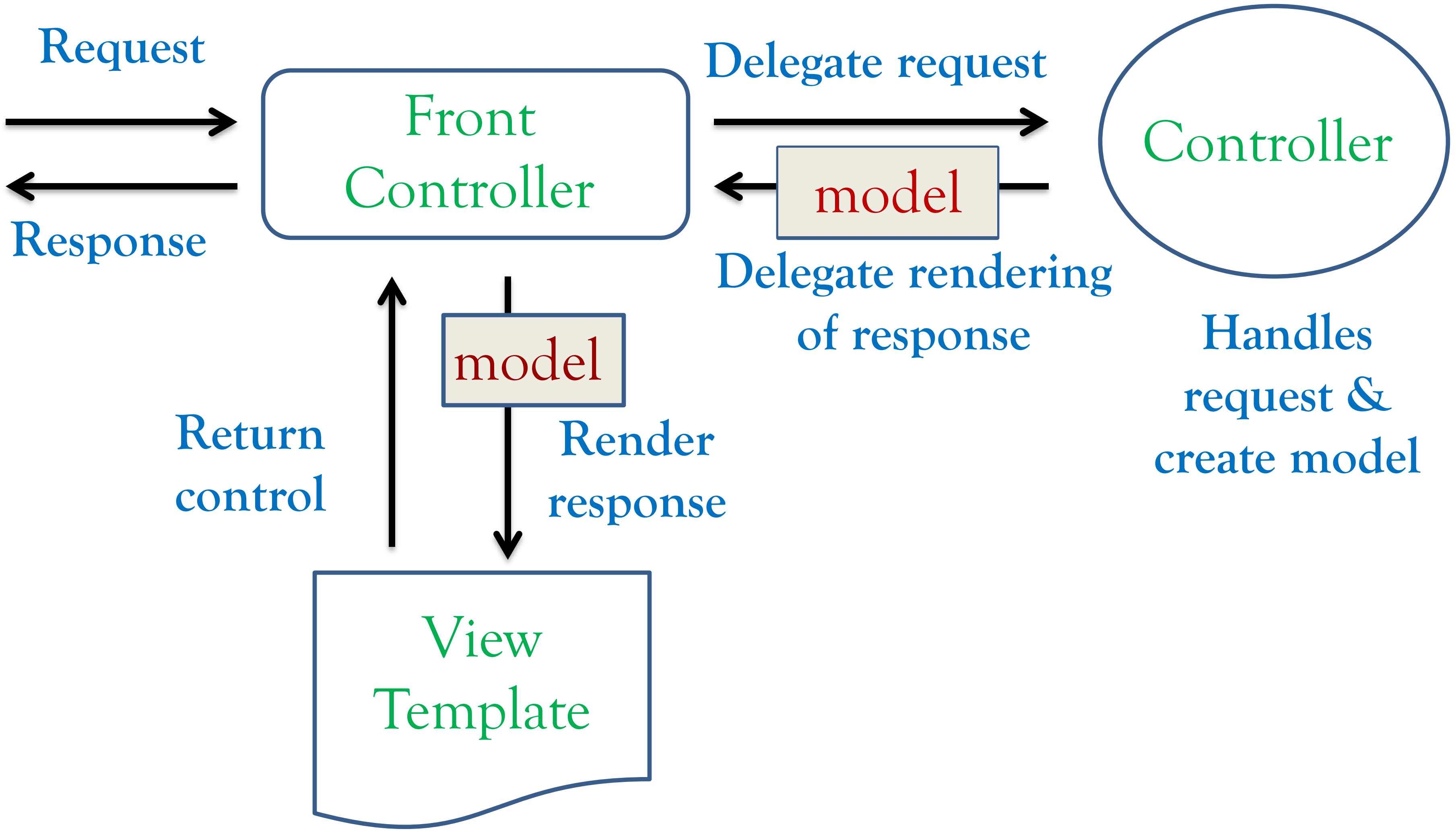
Spring Web-MVC

Front Controller Pattern

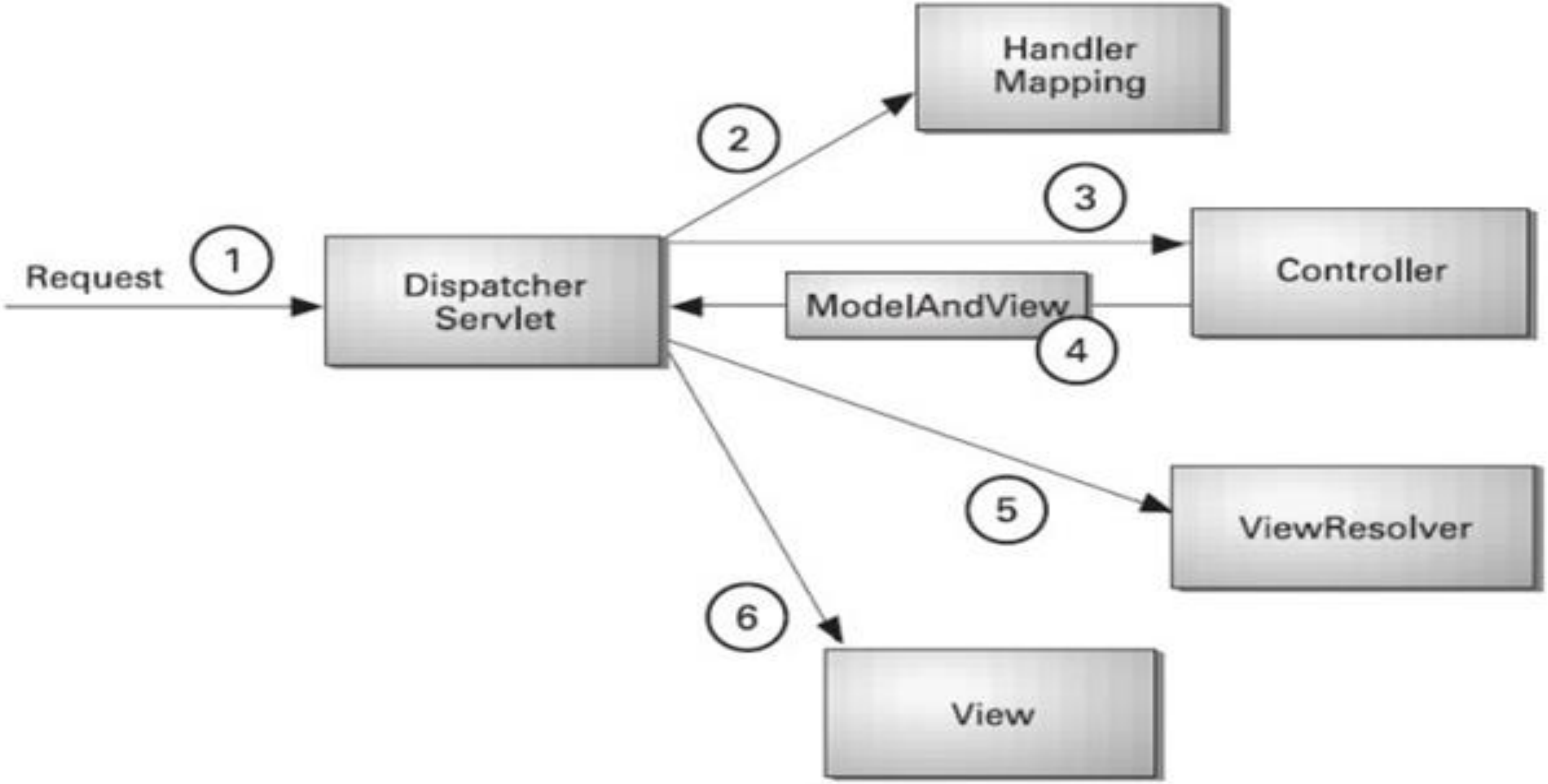
- ✓ **Controller :** is the initial contact point for handling all requests in the system; the controller may then delegate to a helper to complete authentication and authorization of a user or to initiate contact retrieval.
- ✓ **View:** represents and displays information to the client; the view retrieves 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

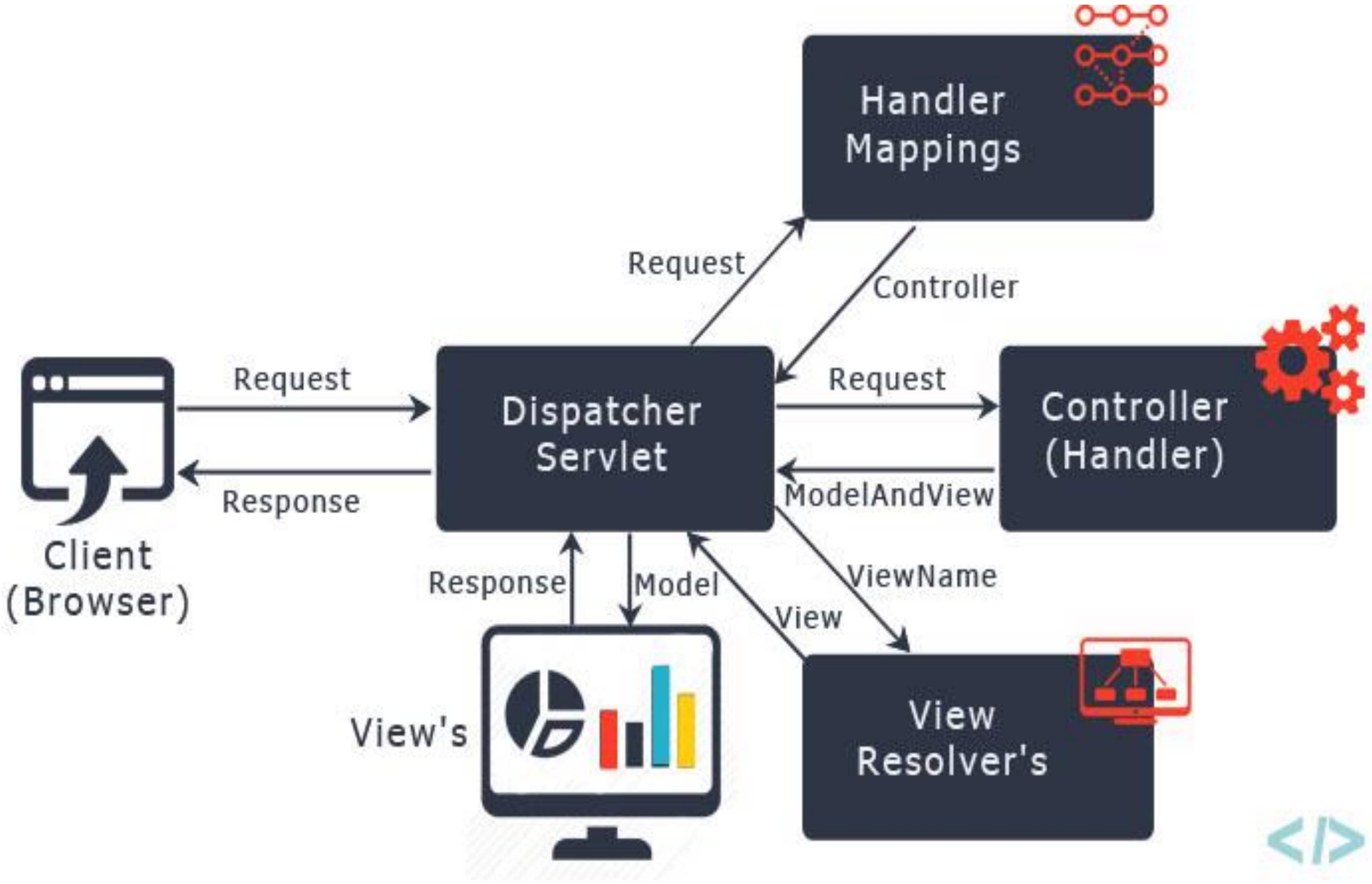
contd..

- 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..

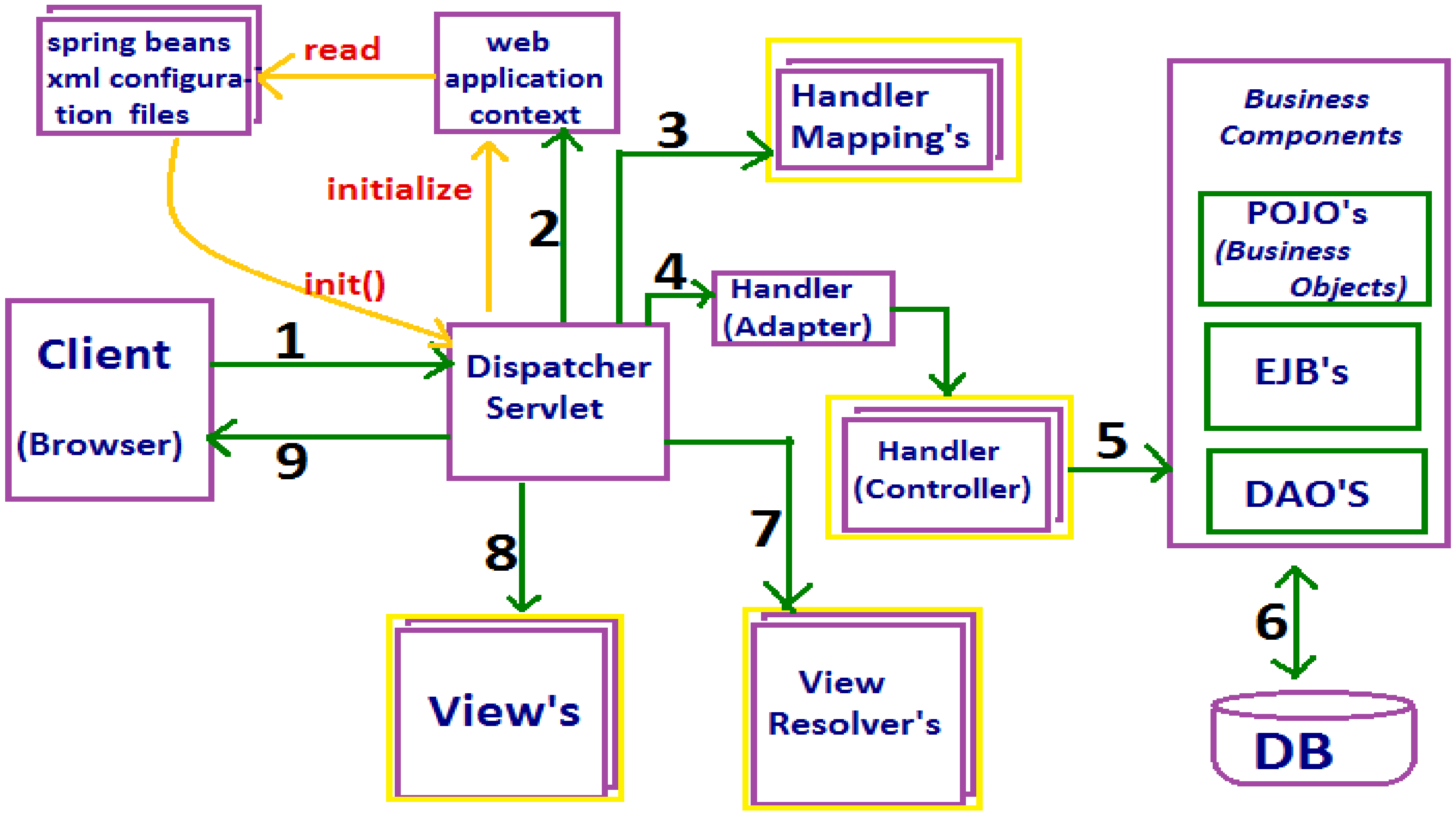
- View
  - ✓ Responsible for rendering output
  - ✓ View name resolution is highly configurable through file extension or
  - ✓ Accept header content type negotiation, through bean names, a properties file, or even a custom ViewResolver implementation.
- Model
  - ✓ is a Map interface, which allows for 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
Request
Dispatcher Servlet
Request
Controller (Handler)
Client (Browser)
Response
ModelAndView
ViewName
Response
Model
View
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** is the annotation used to map web requests onto specific handler classes and/or handler methods is **org.springframework.web.bind.annotation.RequestMapping**. Applied to the **controller class** and also to **methods**.

Sl.No.	Syntax	Description
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.
2.	**@RequestMapping with method –** @RequestMapping(value="/method0") @ResponseBody public String method0() { return "method0"; }	used with method to provide the URI pattern for which handler method will be used. The annotation can be written as @RequestMapping("/method0"). @ResponseBody is to send the String response for this web request.

contd..

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

contd..

5.	**@RequestMapping with headers –** @RequestMapping(value="/method5", headers={"name=Rohit", "id=1001"}) @ResponseBody public String method5() { return "method5"; }	the headers that should be present to invoke the handler method can be specified.
6.	**@RequestMapping** with **Produces and Consumes –** @RequestMapping (value="/method6", produces={"application/json","application/xml"}, consumes="text/html") @ResponseBody public String method6() { return "method6"; }	@RequestMapping provides **produces** and **consumes** vars 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 wanted in response.

contd..

7.	**@RequestMapping with @PathVariable –** @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; }	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.

contd..

8.	**@RequestMapping with @RequestParam for URL parameters –** @RequestMapping (value="/method9") @ResponseBody public String method9(@RequestParam("id") int id) { return "method9 with id= "+id; }	@RequestParam is used to retrieve the URL parameter and map it to the method argument. For the method to work, the parameter name should be "id" and it should be of type int.
9.	**@RequestMapping default method –** @RequestMapping() @ResponseBody public String defaultMethod() { return "default method"; }	When value is empty for a method, it works as default method for the controller class. Here the "/home" is mapped to HomeController and the method will be used for the default URI requests.

contd..

10.	**@RequestMapping fallback method –** @RequestMapping("*") @ResponseBody public String fallbackMethod() { return "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.

Thank You