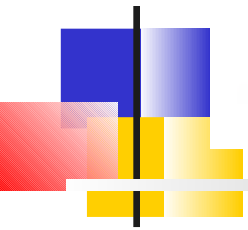
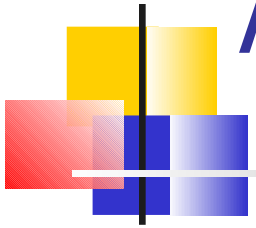


# Spring

java/j2ee Application Framework

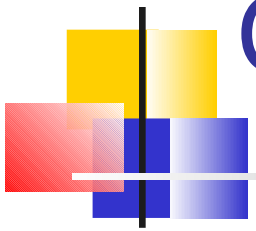
Java User Group – Pune

29th April, 2006



# Agenda of Session

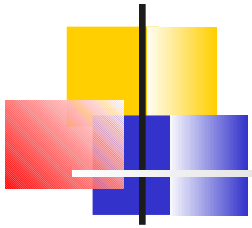
- Comparison between Spring and EJB
- Overview of Spring features
- Spring modules
- Inversion of Control / Dependency Injection in Spring
- Aspect Oriented Programming
- Spring MVC Walkthrough
- Alternatives to Spring MVC



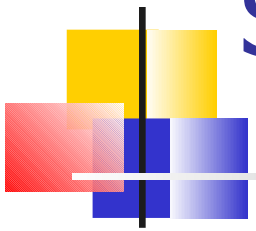
# Current State of J2EE

- EJB never fulfilled intended purpose – simplification of J2EE enterprise application development
- Requires deployment descriptors, plumbing code (home and remote/local interfaces)
- Follows code-wait-test development model due to heavyweight container
- EJB is complicated
  - Entity beans
  - Transaction management
  - Persistence services, remoting and security services

# What is Spring?

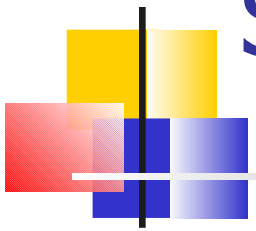


- Created to ease complexity of enterprise application development
- Makes it possible to use JavaBeans to do things previously possible only with EJBs
- Designed with following beliefs:
  - Good design is more important than underlying technology
  - JavaBeans loosely coupled through interfaces is a good model
  - Code should be easy to test
- Spring = lightweight inversion of control and aspect-oriented container framework.



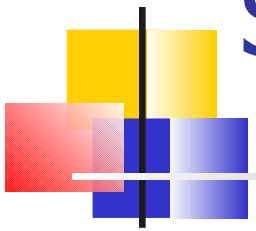
# Spring is Lightweight

- Lightweight in terms of size
  - Entire Spring framework distributable in a JAR file a little over 1MB in size
- Lightweight in terms of overhead
  - Processing overhead required by Spring is negligible
- Non-intrusive
  - Objects in Spring-enabled application have no dependencies on Spring-specific classes



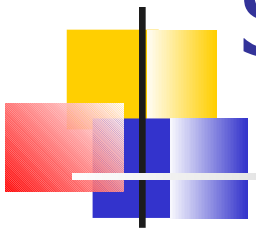
# Spring provides IoC and AOP

- Loose coupling promoted through Inversion of Control
  - Objects are given their dependencies instead of creating or looking for them
  - Container gives dependencies to the object at instantiation
- Cohesive development supported through Aspect Oriented Programming
  - Separates application business logic from system services like logging, transaction management and auditing
  - Application objects only perform business logic and nothing more



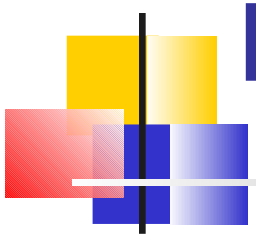
# Spring is a Container

- Contains and manages life cycle and configuration of application objects
- Enables configuration of each bean
  - Manages creation of each bean and its association with other beans
- Not a traditional heavyweight J2EE container



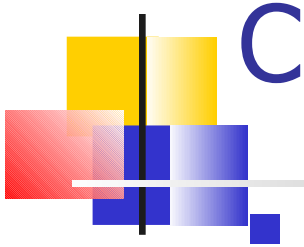
# Spring is a Framework

- Makes it possible to configure and compose complex applications from simpler components
- Application objects composed declaratively, typically in XML files
- Provides infrastructural functionality, leaving only development of application logic to programmer



# Pros and Cons of EJBs

- Pros
  - EJB is a standard
    - Wide industry support
    - Wide adoption
    - Toolability
- Cons
  - Writing an EJB is complicated
  - EJB is invasive – binds component code to EJB technology
  - Entity EJBs not as flexible or feature-rich as other ORM tools



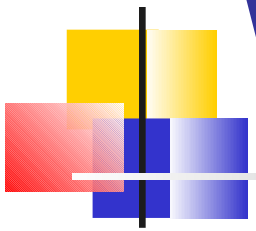
# Comparing Spring to EJB

## Feature comparison

| Feature | EJBs | Spring |
|---|---|---|
| Transaction management | ▪ JTA<br>▪ Supports transactions spanning RMIs | ▪ Multiple transaction environments: Hibernate, JTA, JDBC and JDO<br>▪ No native support to distributed transactions; only through JTA |
| Declarative Transaction support | ▪ Define transactions declaratively in deployment descriptor<br>▪ Can define transaction behavior per method per class<br>▪ Cannot declaratively define rollback behavior; only programmatically | ▪ Define transactions declaratively in Spring config file / class metadata<br>▪ Can define transaction behavior per method per class<br>▪ Can declaratively define rollback behavior per method and per exception type. |
| Persistence | ▪ Programmatic bean-managed<br>▪ Declarative container-managed | ▪ Framework for integrating with various persistence technologies |

# Comparing Spring to EJB (2)

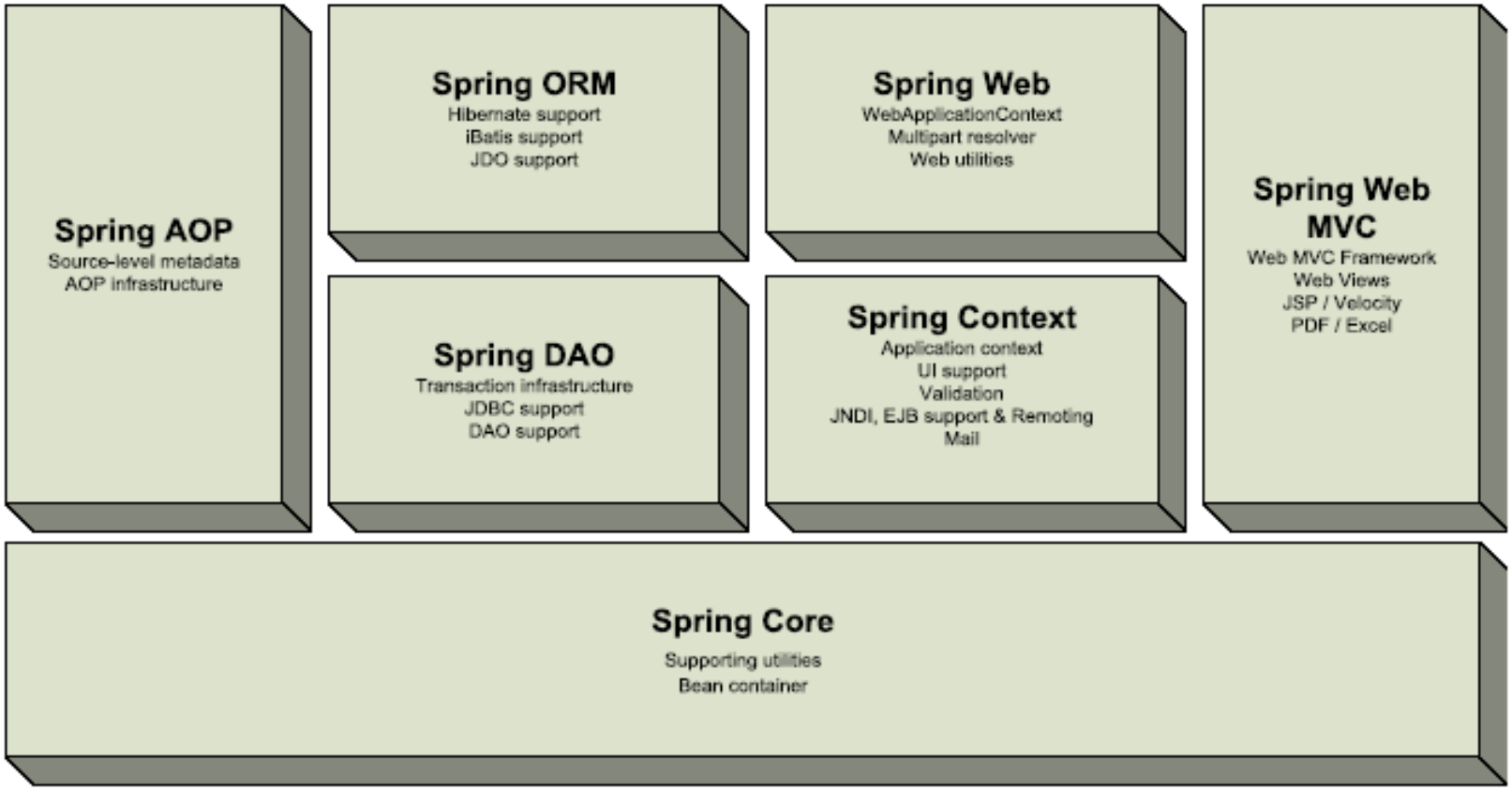
| Feature | EJBs | Spring |
|---|---|---|
| Declarative Security | ▪Supports declarative security through users and roles. Management and implementation of users and roles is container specific.<br>▪Declarative security configured in deployment descriptor. | ▪ No security implementation out-of-the box.<br>▪Acegi, an open source security framework built on top of Spring, provides declarative security through the Spring config file or class metadata. |
| Distributed computing | ▪Provides container-managed remote method calls. | ▪Provides proxying for remote calls via RMI, JAX-RPC, and web services |

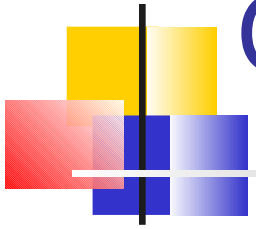


# Why Spring over EJB?

- Provides nearly all services provided by EJB container
- Allows development of code that is cleaner, more manageable and easier to test
  - Definition of implementation as POJO and wire in any additional services needed through injection or AOP.
- Lightweight container
- App components not required to implement, extend or use any Spring-specific classes or interfaces
  - Component reuse possible even in absence of Spring
- Spring + Hibernate (or another ORM framework) : Entity objects not directly coupled with persistence mechanism.
  - Makes them light enough to be passed across application tiers.

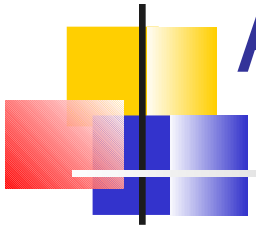
# Spring Modules





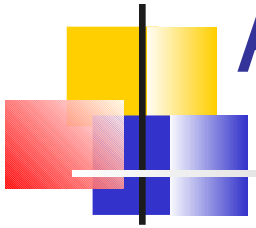
# Core Container

- Fundamental functionality
- `BeanFactory`– heart of any Spring app
  - Implementation of factory pattern
  - Applies IoC to separate the application's configuration and dependency specifications from application code.



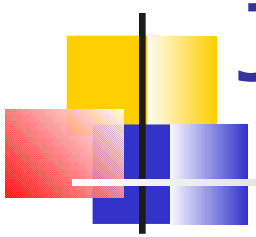
# Application Context Module

- Context module makes Spring a framework.
- Extends the concept of `BeanFactory`, adding support for:
  - Internationalization (I18N) messages
  - Application lifecycle events
  - Validation.
- Supplies enterprise services such as e-mail, JNDI access, EJB integration, remoting, and scheduling.
- Includes support for integration with templating frameworks like Velocity and FreeMarker.



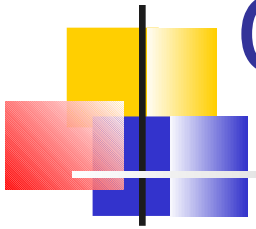
# AOP Module

- Serves as basis for developing aspects for Spring-enabled application.
- Interoperability between Spring and other AOP frameworks: AOP support based on the AOP Alliance API, defining a common set of interfaces and components.
- Spring AOP module introduces metadata programming to Spring.
  - Enables addition of annotations to source code that instruct Spring on where and how to apply aspects.



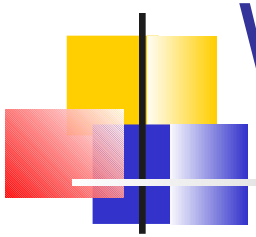
# JDBC Abstraction & DAO Module

- Abstracts away JDBC boilerplate code - getting connection, creating statement, processing result set, and closing the connection.
- Keeps database code clean and simple; prevents problems that result from a failure to close database resources.
- Builds a layer of meaningful exceptions on top of the error messages given by several database servers.
- Uses Spring's AOP module to provide transaction management services for objects in a Spring application.



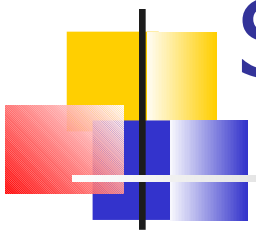
# ORM Integration Module

- Spring doesn't implement its own ORM solution
- Provides hooks into several popular ORM frameworks, including Hibernate, JDO, and iBATIS SQL Maps.
- Spring's transaction management supports ORM frameworks as well as JDBC.



# Web Module

- Web context module builds on the application context module
- Provides context that is appropriate for web-based applications.
- Contains support for several web-oriented tasks e.g.: transparently handling multipart requests for file uploads, programmatic binding of request parameters to business objects.
- Contains integration support with Jakarta Struts.

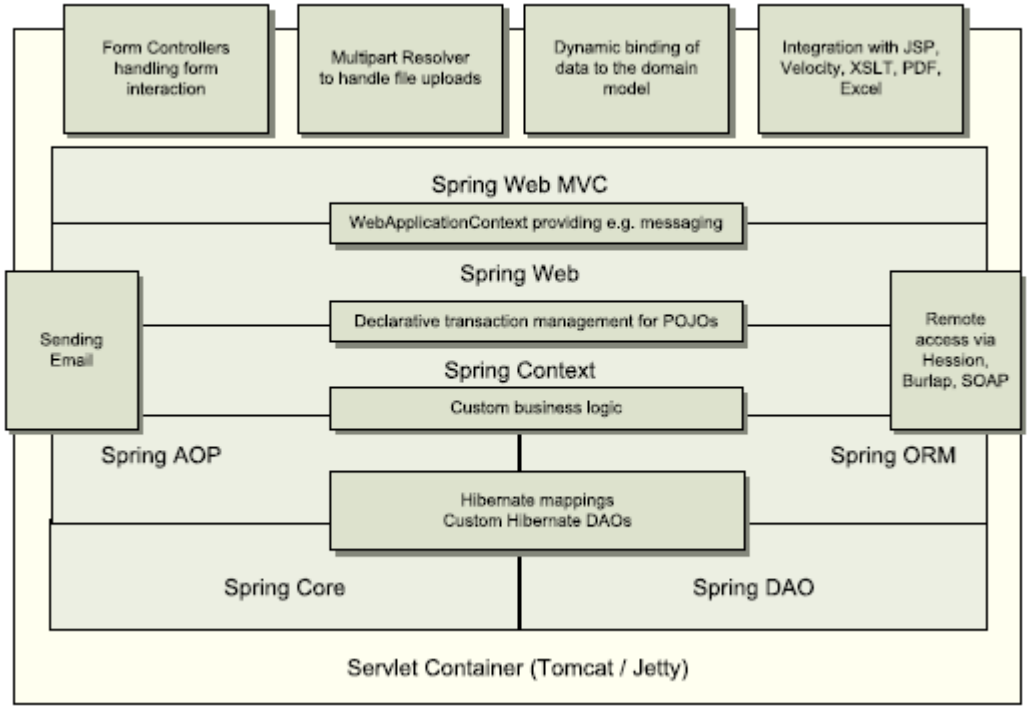


# Spring MVC Framework

- Spring can be integrated with other MVC frameworks, such as Struts
- Spring's MVC framework uses IoC to provide for clean separation of controller logic from business objects.
- Allows declarative binding of  request parameters to business objects
- Spring's MVC framework can use Spring's other services, such as I18N messaging and validation.

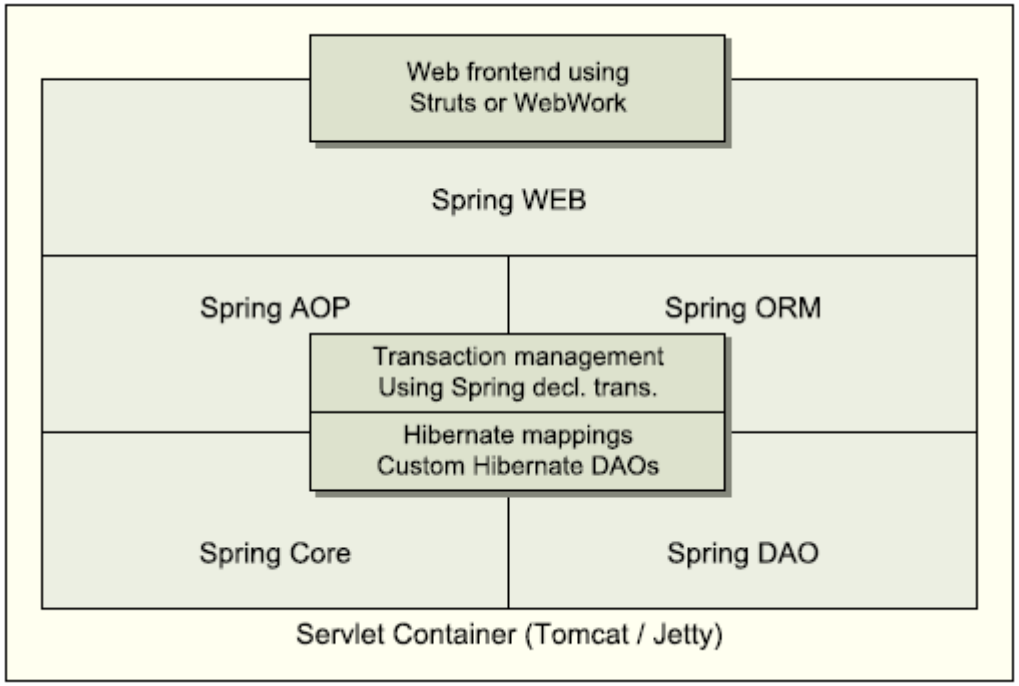
# Spring Usage Scenarios

- Spring in a web application



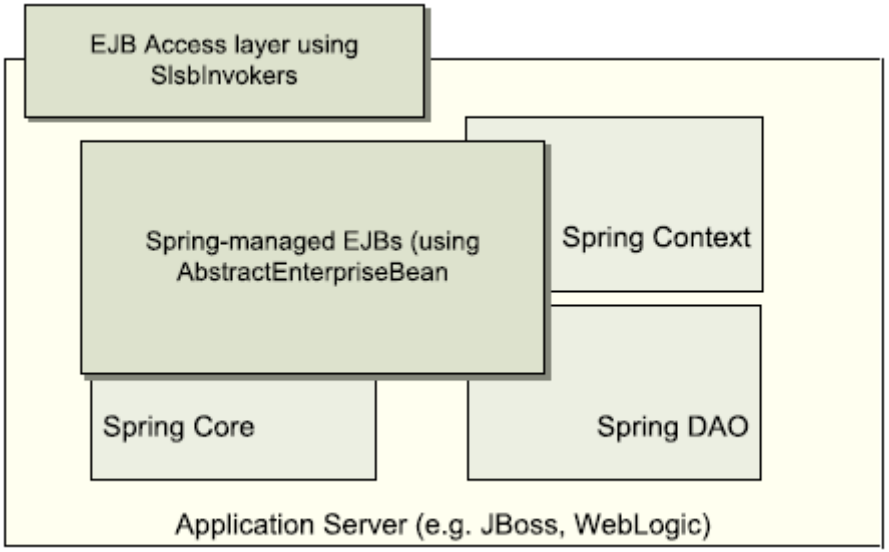
# Spring Usage Scenarios (2)

- Spring middle tier using third-party web framework



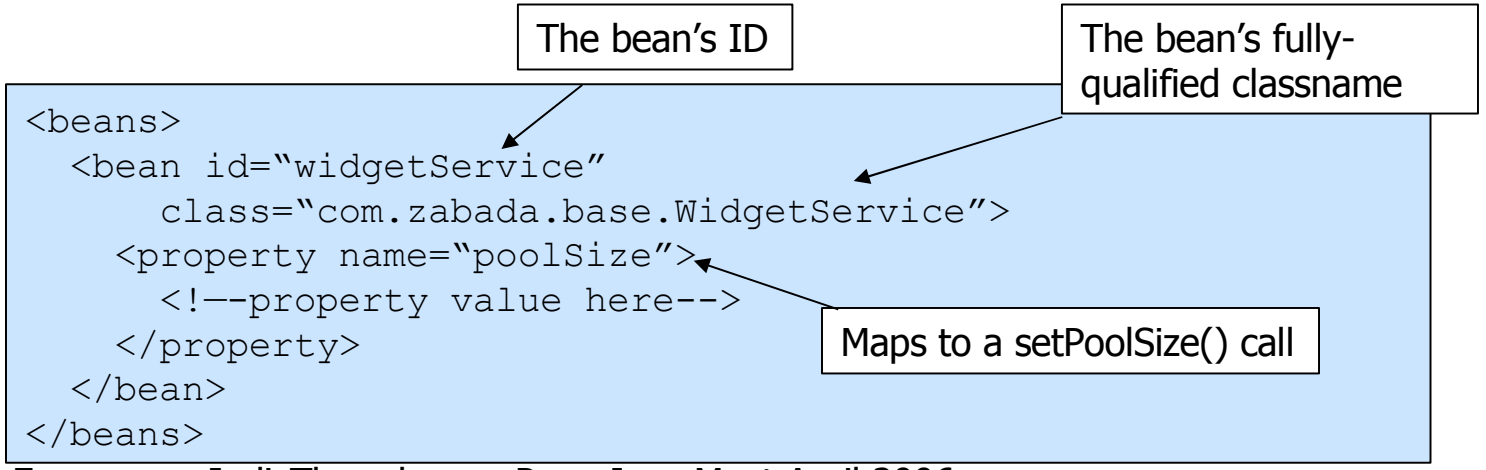
# Spring Usage Scenarios (3)

- EJBs: Wrapping existing POJOs

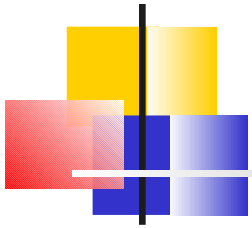


# BeanFactory & ApplicationContext

- Beans: Synonymous with application objects
- BeanFactory: Actual container which instantiates, configures, and manages a number of beans.
- Beans collaborate with one another, have dependencies between themselves.
- Dependencies reflected in configuration data used by BeanFactory
- Bean Factory instantiated explicitly by user, or loaded at startup by the Spring framework



# BeanFactory & ApplicationContext (2)



- ApplicationContext builds on top of the BeanFactory (it's a subclass)
- Adds other functionality such as easier integration with Spring's AOP features, message resource handling (for use in internationalization), event propagation, declarative mechanisms to create the ApplicationContext etc.
- BeanFactory provides configuration framework and basic functionality; ApplicationContext adds J2EE and enterprise-centric capabilities to it
- ApplicationContext is complete superset of BeanFactory: any description of BeanFactory capabilities and behavior applies to ApplicationContext also

# Bean Definitions

- Represented by BeanDefinition objects
- Contain class name, behavioral config elements, constructor arguments and property values to set, and collaborator/ dependency info

```
<bean id="dataSource" class="org.apache.commons.dbcp.BasicDataSource">
      <property name="driverClassName">
       <value>com.mysql.jdbc.Driver</value>
      </property>
      <property name="url">
       <value>jdbc:mysql://10.224.33.202:3307/mmpocdb</value>
      </property>
      <property name="username">
         <value>root</value>
      </property>
      <property name="password">
         <value>password</value>
      </property>
 </bean>
```

# Example Spring Application - HelloWorld

- `GreetingService` Interface has method `sayGreeting()`
- `GreetingServiceImpl` : implements the `GreetingService` interface
- `GreetingServiceImpl` has a single property `greeting`, which is a String containing the greeting to be printed
- `greeting` can be set by a constructor or the property's setter method

```
public class GreetingServiceImpl implements GreetingService {
  private String greeting;

  public GreetingServiceImpl() {}

  public GreetingServiceImpl(String greeting) {
    this.greeting = greeting;
  }

  public void sayGreeting() {
    System.out.println(greeting);
  }

  public void setGreeting(String greeting) {
    this.greeting = greeting;
  }
}
```

# Example Spring Application – Spring context file

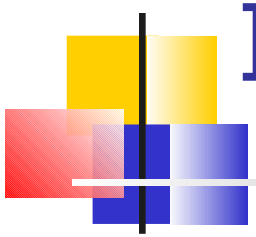
- Declare an instance of a `GreetingServiceImpl` in the Spring container
- `<beans>` element is the root element of Spring configuration file.
- `<bean>` element tells the Spring container about the class and how it should be configured.
- `<property>` element is used to set a property, in this case the greeting property.
- The value of the greeting is defined within the `<value>` element.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="greetingService"
        class="com.eg.hello.GreetingServiceImpl">
        <property name="greeting">
            <value>Buenos Dias!</value>
        </property>
    </bean>
</beans>
```

# Example Spring Application – Instantiating `GreetingService`

- The `BeanFactory` class here is the Spring container.
- After loading the hello.xml file into the container, the `main()` method calls the `getBean()` method on the `BeanFactory` to retrieve a reference to the greeting service.
- With this reference in hand, it finally calls the `sayGreeting()` method to print "Buenos Dias!"

```
public class HelloApp {
    public static void main(String[] args) throws Exception {
        BeanFactory factory =
            new XmlBeanFactory(new FileInputStream("hello.xml"));
        GreetingService greetingService =
            (GreetingService) factory.getBean("greetingService");
        greetingService.sayGreeting();
    }
}
```

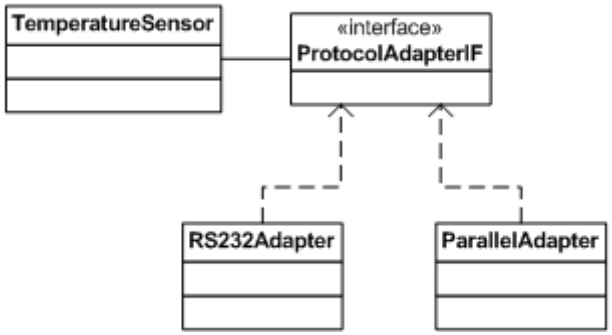


# Inversion of Control

- Aspect of control being inverted: Acquisition of dependent objects
- Also called *dependency injection*
- *Then:*
  - Each object responsible for obtaining its own references to its dependencies.
  - Leads to highly coupled and hard-to-test code.
- *Now, with IoC:*
  - Objects given their dependencies at creation time by some external entity that coordinates each object in the system.
  - Thus, dependencies are *injected* into objects.
  - IoC means an inversion of responsibility with regard to how an object obtains references to collaborating objects.

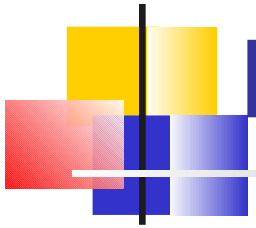
# Example of IoC

- System to measure temperature
- Sensor objects implement the `ProtocolAdapterIF` interface and any of them can be plugged into the `TemperatureSensor` object
- When a `TemperatureSensor` is needed, some entity in the system must know what implementation of `ProtocolAdapterIF` to produce and associate with the sensor object.



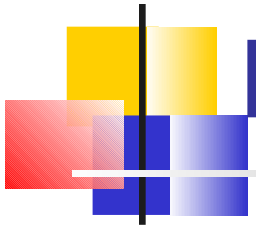

```
<bean id="tempSensor"
    class="com.project.sensor.TemperatureSensor">
    <property name="sensorDelegate">
        <ref bean="sensor"/>
    </property>
</bean>

<!-- Sensor to associate with tempSensor -->
<bean id="sensor" class="com.project.comm.RS232Adapter"/>
```

# Applying Aspect Oriented Programming



- AOP Jargon:
  - **Joinpoints**: Well-defined point during the execution of your application e.g.: call to a method
  - **Advice**: Code executed at a particular joinpoint
  - **Pointcuts**: Collection of joinpoints that define when advice should be executed. E.g.: collection of all method invocations in a particular class.
  - **Aspects**: Definition of the logic that should be included in the application and where it should execute. E.g.: logging and auditing
  - **Weaving**: Process of inserting aspects into the application code at the appropriate point.
  - **Target**: Object whose execution flow is modified by some AOP process

# Applying Aspect Oriented Programming (2)



- Before advice
  - Can modify the arguments passed to a method
  - Can prevent the method from executing by raising an exception.
- After returning advice
  - Executed after the method invocation at the joinpoint returns.
  - Arguments that are passed to it cannot be changed.
- Around advice
  - Combination of before and after advice
  - You can modify the return value and prevent the method from actually executing; entire method implementation can be replaced with your own implementation.
  - Modeled as an **interceptor** using the **MethodInterceptor** interface.
- Interceptor strategy uses 3 objects: target, proxy and interceptor

# Example of AOP

- Define interceptor and proxies for all target objects

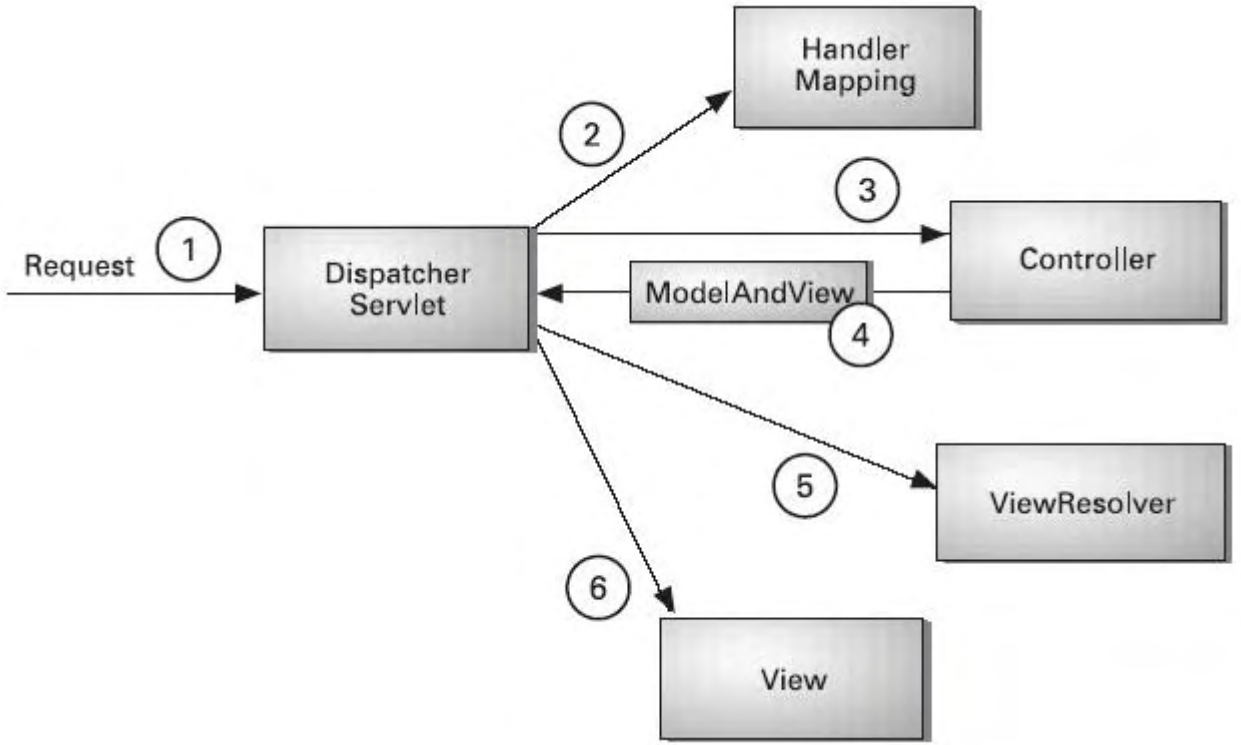
```
<bean id="LogInterceptor"
  class="com.cts.mmpoc.pp.interceptor.LogInterceptor">
</bean>

<bean id="userProxyCreator" class="org.springframework.aop.framework.autoproxy.BeanNameAutoProxyCreator">
    <property name="beanNames">
      <value>*</value>
    </property>
    <property name="interceptorNames">
      <list>
       <value>LogInterceptor</value>
       <value>AuditInterceptor</value>
      </list>
    </property>
</bean>
```

- Write interceptor code

```
public class LogInterceptor implements MethodInterceptor{
    public LoggingManager log = null;
    public Object invoke(MethodInvocation arg0) throws Throwable{
        log = LoggingManager.getInstance(arg0.getMethod().getDeclaringClass().getName());
        log.logInfo("Entered method "+arg0.getMethod().getName()+" at "+new Date());
        Object retVal = arg0.proceed();
        log.logInfo("Exited method "+arg0.getMethod().getName()+" at "+new Date());
        return retVal;
    }
}
```

# Spring MVC: Walkthrough

Life cycle of a request from start to finish



# Usage of Spring MVC

- Configure `DispatcherServlet`:

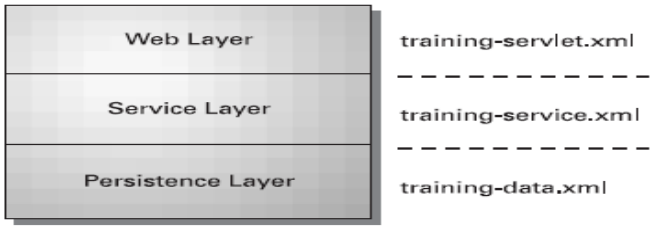
```
<servlet>
  <servlet-name>training</servlet-name>
  <servlet-class>org.springframework.web.servlet.DispatcherServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
```

Name of Spring context file

- Handle all URLs ending with *.htm

```
<servlet-mapping>
  <servlet-name>training</servlet-name>
  <url-pattern>*.htm</url-pattern>
</servlet-mapping>
```

- Split the application context XML file logically across application lavers



# Usage of Spring MVC (2)

- Configure servlet listener `ContextLoaderListener` (containers using Servlet spec 2.3 or higher) or `ContextLoaderServlet`
- Give location of Spring context file to load

```
<context-param>
  <param-name>contextConfigLocation</param-name>
  <param-value>/WEB-INF/training-service.xml,
      ➥ /WEB-INF/training-data.xml</param-value>
</context-param>
```

- Write the controller class that performs the logic behind the homepage.

```
public class HomeController implements Controller {
  public ModelAndView handleRequest(HttpServletRequest request,
      HttpServletResponse response) throws Exception {
    return new ModelAndView("home", "message", greeting);
  }

  private String greeting;
  public void setGreeting(String greeting) {
    this.greeting = greeting;
  }
}
```

# Usage of Spring MVC (3)

- Configure the controller in the `DispatcherServlet`'s context configuration file (`training-servlet.xml`)

```
<bean name="/home.htm"
    class="com.springinaction.training.mvc.HomeController">
  <property name="greeting">
    <value>Welcome to Spring Training!</value>
  </property>
</bean>
```

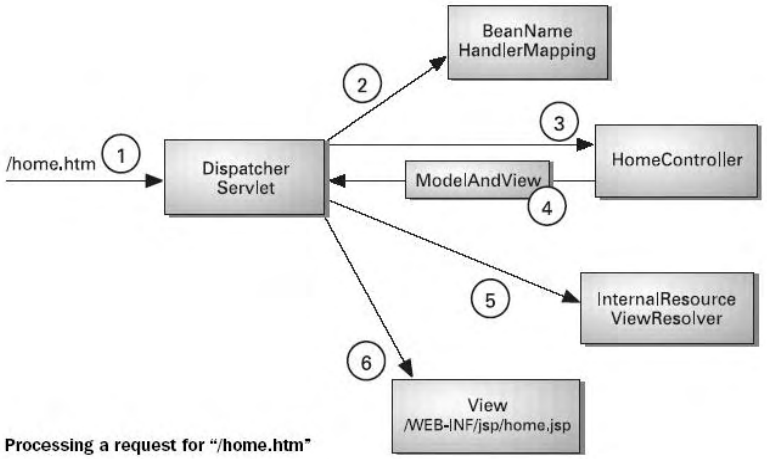
- Configure a view resolver to resolve "home" to a JSP file that renders the home page

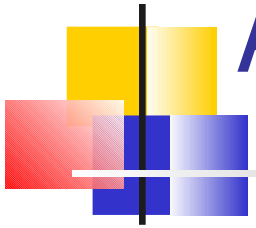
```
<bean id="viewResolver" class="org.springframework.web.
      servlet.view.InternalResourceViewResolver">
  <property name="prefix">
    <value>/WEB-INF/jsp/</value>
  </property>
  <property name="suffix">
    <value>.jsp</value>
  </property>
</bean>
```

- Write the JSP that will render the homepage to the user

# Spring MVC Summary

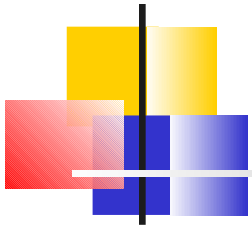
1. `DispatcherServlet` receives request with URL pattern `/home.htm`
2. `DispatcherServlet` consults `BeanNameUrlHandlerMapping` to find matching controller; finds `HomeController`
3. `DispatcherServlet` dispatches the request to `HomeController` for processing
4. `HomeController` returns a `ModelAndView` object with logical view name of `home`.
5. `DispatcherServlet` consults its `InternalResourceViewResolver` to find view whose logical name is `home`
6. Accordingly, `DispatcherServlet` forwards the request to the JSP at `/WEB-INF/jsp/home.jsp`





# Alternatives to Spring MVC

- View Layer alternatives to JSP
  - Velocity
  - Freemarker
  - Tiles
- Alternative web frameworks: Spring can be integrated with other web frameworks as well
  - Jakarta Struts
  - Tapestry
  - JavaServer Faces
  - WebWork



# Questions?