Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Day 14,15,16. Servlet JSP

Smita B Kumar

Google Cloud Platform

Windows Azure

amazon web services

APACHE kafka. A distributed streaming platform

Apache SOFTWARE FOUNDATION

APACHE Spark

Java

hadoop

Table of Contents

Day	Module	Title
Day 1	1	Introduction to Spring framework
	2	Getting started
	3	Spring IOC/DI
	4	Bean life cycle and callback methods
Day 2	5	Introduction to Aspect Oriented Programming
	6	The AOP Advice types and point cuts
	7	Spring JDBC
	8	Web Module
Day 3	5	**Customizing Spring MVC**
	6	**Form Controllers**

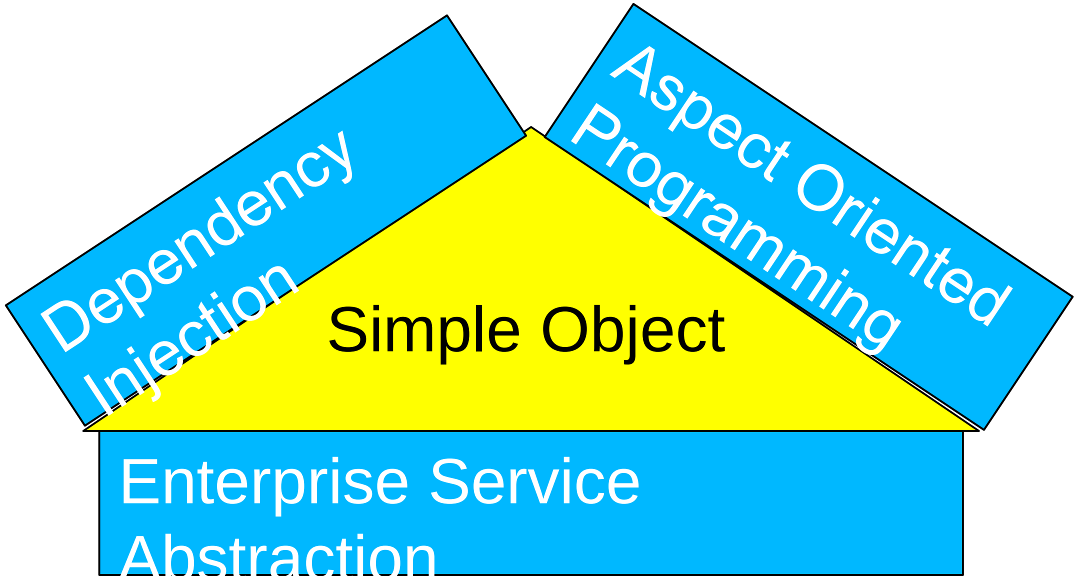
Module 1. Introduction to Spring

- **Overview**

  - ➢ Spring as a solution for business logic implementation
  - ➢ Spring architecture
  - ➢ Spring in different tiers
  - ➢ Spring’s light weight container

What is Spring framework?

- Light weight, Open source framework for developing Enterprise Grade Application.
- Developed in 2004 by Rode Johnson
- Lightweight, Simple solution to compete EJB
- Provides light weight container to create objects lazily, singleton
- Declarative plumbing, transaction, security and logging.



Dependency Injection

Aspect Oriented Programming

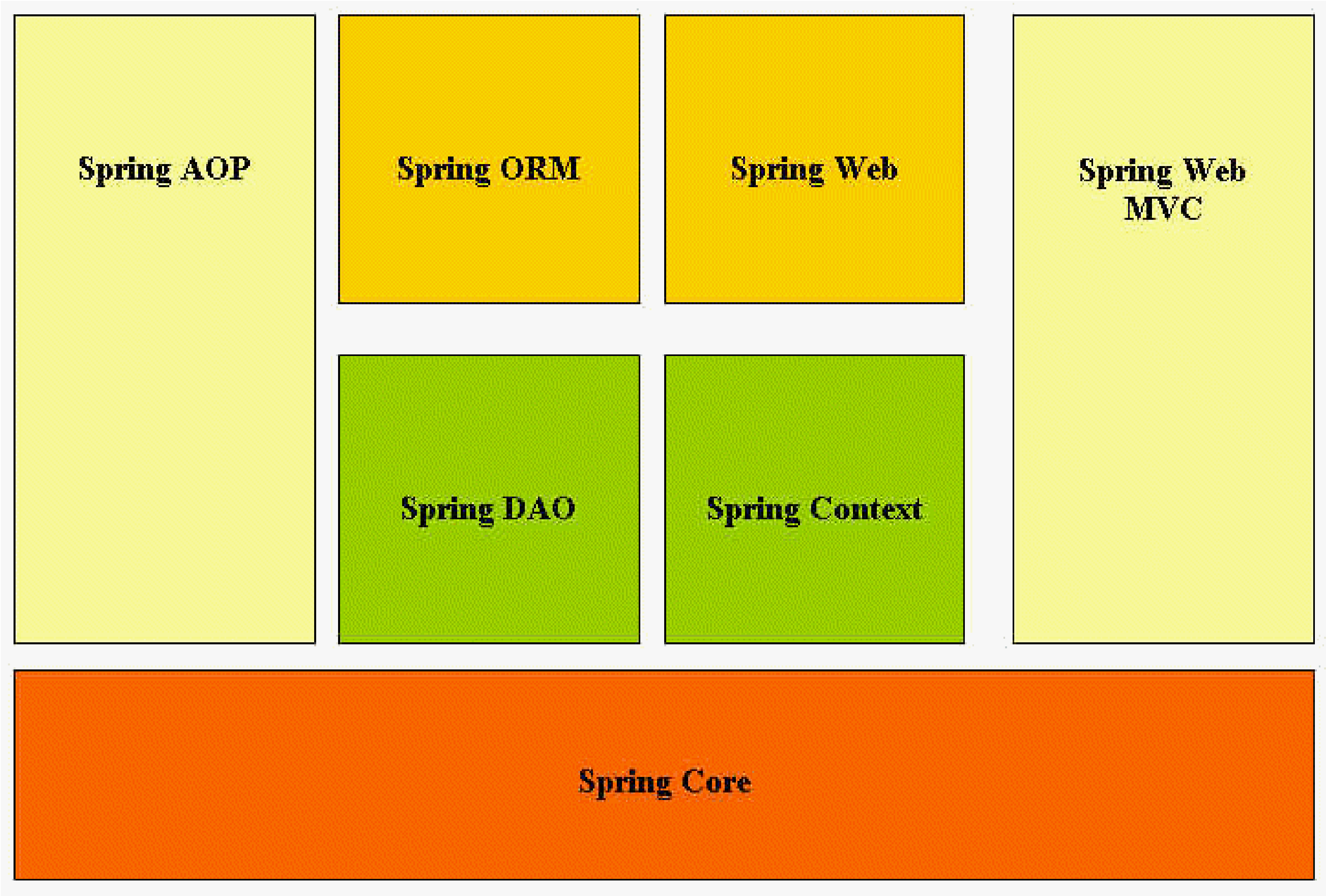
Simple Object

Enterprise Service Abstraction

Why Spring framework?

- Provides loose coupling with services, objects, tools and technology as they are configurable and declarative.
- Encourage non-invasive approach to code
- Encourages unit testing
- Light weight as may run without server
- Provides abstraction to integrate with best breed of products like Hibernate, Aceji, Struts etc.
- Flexible to run in both managed and un-managed environment.
- Rich of solutions for different tiers

Spring architecture



Spring AOP

Spring ORM

Spring Web

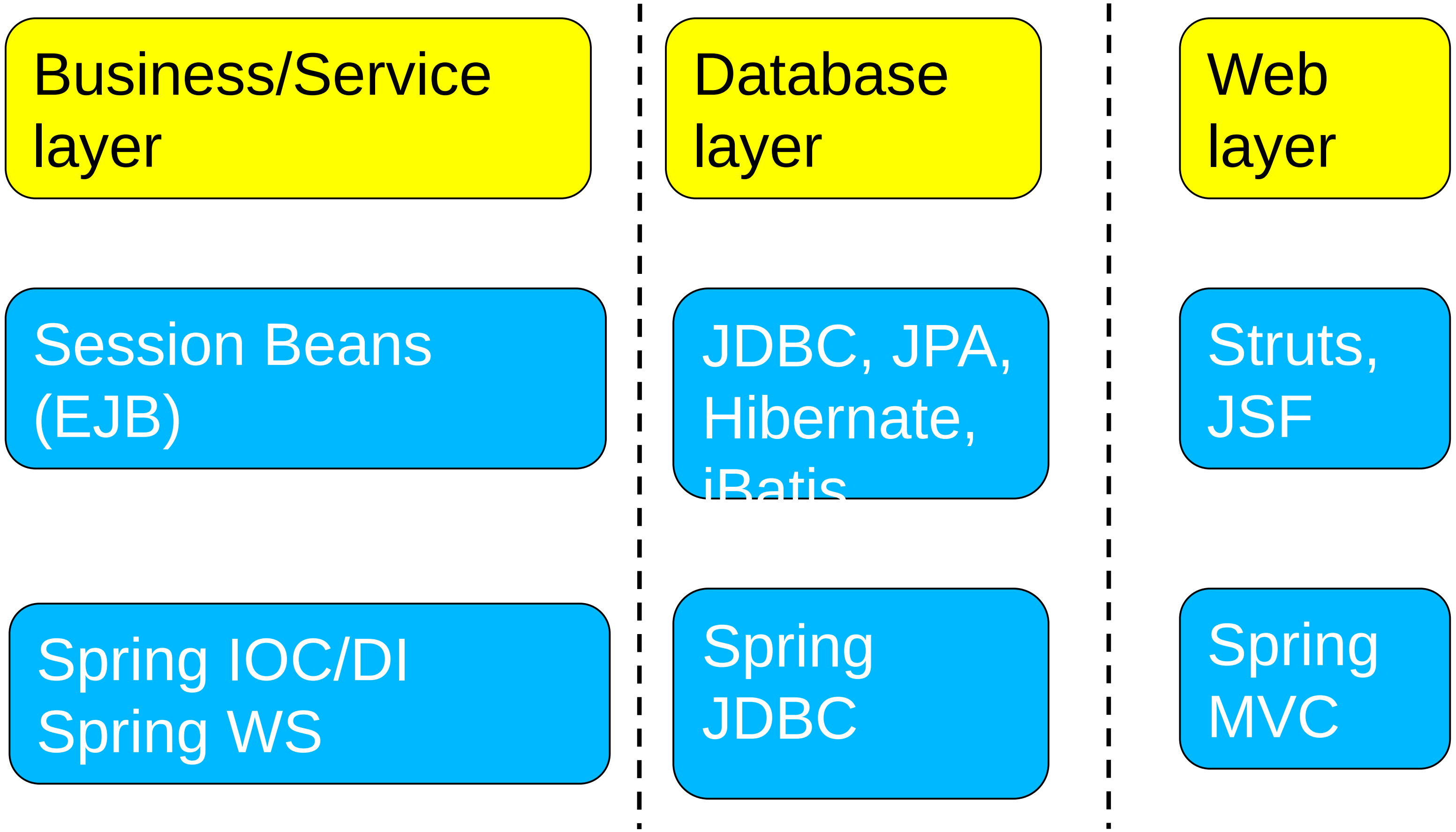
Spring Web MVC

Spring DAO

Spring Context

Spring Core

Spring in different tiers



Business/Service layer	Database layer	Web layer
Session Beans (EJB)	JDBC, JPA, Hibernate, iBatis	Struts, JSF
Spring IOC/DI Spring WS	Spring JDBC	Spring MVC

IoC in Spring

The IoC is the broader concept with two main types:

1. **The dependency lookup**: The container provides call-backs to components and lookup context. The component uses JNDI to do lookup of other components and resources.

```
@Resource
private DataSource dataSource;
```

2. **The dependency injection**: IoC container configures the object externalizing resource lookup from application code i.e. Dependencies are injected into object calling setter/constructor. Object does not do lookup of resources.

```
public void setDataSource(DataSource ds){
        dataSource = ds;
}
 <bean id="emp_list“  class="pack_10_ioc.EmpDao">
                <property    name="dataSource“    ref="dsRefOrcl“/>
</bean>
```

Spring container

Container reads configuration meta data to...

- Instantiate object with the given scope.
- To initialize object with configured data
- To inject dependencies in object etc

Container reads meta information from...

- XML configuration file.
- Java Annotations
- Java code.

Spring container

Its a light weight implementation of BeanFactory interface

BeanFactory: Sophisticated implementation of factory pattern. Gives objects with scope- Singleton or prototype. Decouples business logic from configuration and dependency specifications.

ApplicationContext: Augments BeanFactory with enterprise centric functionalities like integration with AOP features, message resource, event propagation etc.

WebApplicationContext: Augments ApplicationContext with web application specific features like scopes- globalSession, sesssion and request etc.

Spring container

The ApplicationContext:

- MessageSource
- Access to resources such as URLs and files.
- Event propagation
- Multiple hierarchical context.

Module 2. Getting started

- **Overview**
  - Spring as a bean factory
  - Bean initialization
  - Bean scopes
  - Constructor injection
  - Setter injection
  - Bean configuration using Annotations

Spring as a bean factory

Configuring bean...

```
 <bean id="resourceBean1" class="pack_10_ioc..GlobalInvestment">

</bean>
```

Instantiating bean...

```
 BeanFactory beanFactory =  new XmlBeanFactory(
                                     new ClassPathResource(
“                                      path-configuration\\context.xml"));

GlobalInvestment bean =
(GlobalInvestment)beanFactory..getBean(“resourceBean1”);
```

- Class name goes in configuration file.
- Client code refers bean with identification name and not by class name.
- The beanFactory refers to the container.
- The getBean method instantiate object and return reference.

Bean initialization

Bean initialization through constructor

```
 <constructor-arg type="String">
          <value>Gobal Investment Pvt. Ltd.</value>
</constructor-arg>
<constructor-arg type="String" >
          <value>Ladder of success</value>
</constructor-arg>
```

- Type “String” is optional. But mandatory for any other type.
- Order and type of <constructor-arg> decides the signature of constructor.
- Spring container invokes appropriate constructor on the object while instantiating the object.

Bean initialization through setters

```
 <property name="globalRank">
          <value>104</value>
</property>
```

- Property name is the name of the setter method without “set” word.
- Spring container calls appropriate setter method and passes value on the

Bean initialization (Contd…)

Initializing ‘Set’ field...

```
<property name="directorsPanel">
        <set>
                <value>Mr. Malhotra</value>
                <value>Mr. Gihrotra</value>
                <value>Mr. Sanmitra</value>
        </set>
</property>
```

Similarly, other tags for initializing array, list fields are…

<array>

and

<list>

Bean initialization (Contd…)

Initializing ‘map’ field

```
 <property name="branches">
        <map>
               <entry key="Mumbai">
                      <value>Shop 123, Lane 6, Subhash Road</value>
               </entry>
               <entry key="Delhi">
                      <value>Shop 234, Shoper's Stop, Link Road</value>
               </entry>
        </map>
</property>
```

Initializing ‘java.util.properties’ field

```
 <property name="branchManagers">
        <props>
               <prop key="Mumbai">Mr. Jagtap</prop>
               <prop key="Delhi">Mr. Bhupendra Singh</prop>
        </props>
</property>
```

Bean Scopes

Bean with scope- “singleton”

```
<bean id="resourceBean1" class="pack_10_ioc..GlobalInvestment“
                                                   scope=“singleton”>

</bean>
```

•The scope singleton is optional. A bean without ‘scope’ clause is singleton.

Bean with scope- “prototype”

```
<bean id="resourceBean1" class="pack_10_ioc.EmpBean“
                                             scope=“prototype”>

</bean>
```

- The scope of Spring can not overrule the scope if declared programmatically in the object.

Resource injections

Bean with constructor and setter method

```
 public class EmpDao {
        private DataSource dataSource;
        private GlobalInvestment resource;

        public EmpDao(GlobalInvestment resource){
                this.resource = resource;
        }

        public void setDataSource(DataSource dataSource) {
                this.dataSource = dataSource;
        }
        ……………
}
```

- The resource “GlobalInvestment” will be injected through Constructor injection while resource DataSource will be injected through setter method.

Setter injection

Bean of type DriverManager

```
                <bean id="ds"

         class="org.springframework.jdbc.datasource.DriverManagerDataSource“>

                          <property name="driverClassName"
value="oracle.jdbc.OracleDriver"/>
                          <property name="url"
value="jdbc:oracle:thin:@172.16.0.51:1521:orcl"/>
                          <property name="username" value="scott"/>
                          <property name="password" value="tiger"/>
                <bean>
```

- The DriverManagerDataSource is a class of Spring API.
- It has already written setter methods like setUrl(), setUsername() etc.

Setter injection

```
      <bean id="emp_list“    class="pack_01_xml.EmpDao">
                <property    name="dataSource“    ref="ds“ />
      </bean>
```

Constructor injection

Bean of type GlobalResource

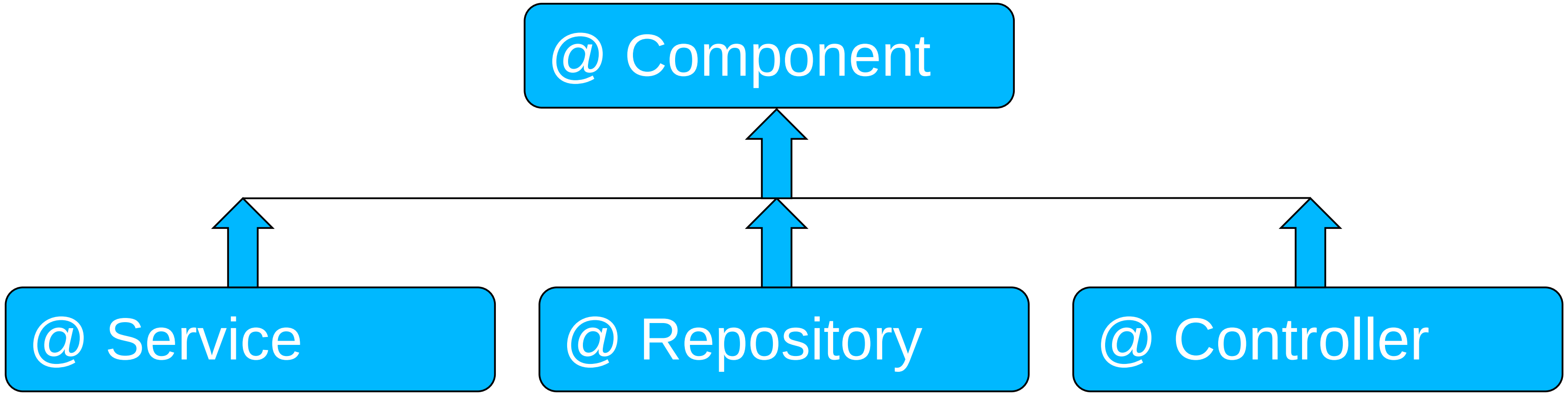
```
        <bean id=“globalResource"
                  class=“pack.GlobalInvestment“>

                  ......................
                  .....................
        <bean>
```

Constructor injection

```
 <bean id="emp_list“    class="pack_01_xml.EmpDao">
        <constructor-arg   ref="globalResource“ />
</bean>
```

- **The ‘ref’ clause referes to bean of which dependency is to be injected in EmpDao object. The EmpDao class must have constructor to accept GlobalResource type of parameter.**

Annotations to declare bean



Different types of annotations document a bean for a specific functionality.

- **@ Service**: Bean having set of services.
- **@ Repository**: The DAO types of beans.
- **@ Controller**: The execution flow controlling functionalilties.

Annotation to declare a bean

Bean Declaration...

```
        @Service ("empEntity")
        @Scope ("prototype")
        public class EmpBean {


}
```

The XML description

```
<context:component-scan base-package="pack_02_annot" />
```

- Bean is not declared in xml. The full name of a package container containing bean is declared.
- Allows more than one component-scan declarations to declare more than one package containers for multiple beans.

Bean configuration using Java code

Configuration using Java Code

```
@Configuration
public class AppConfig {

	@Bean
	public CustomerService customerService() {
		return new CustomerServiceImpl();

	}
	@Bean
	public BankService bankService() {
		BankServiceImpl bankService = new BankServiceImpl();
		bankService.setCustomerService(customerService());
		return bankService;
	}
}
```

Client Code…

```
ApplicationContext ctx = new AnnotationConfigApplicationContext(AppConfig.class);

BankService bankService = ctx.getBean("bankService", BankService.class);
```

Module 3. Spring IoC and DI

- **Overview**
  - Ways of auto-wiring
  - Spring configuration for Inheritance
  - Spring using factory
  - The Dependency Relationship using ApplicationContextAware

Autowiring

Autowiring: Framework automatically sets the dependency.

There are two ways-

1. **By Type:** Bean searched for wiring is searched on basis of type.

2. **By Name:** Bean searched for wiring is searched on basis of name.

Autowiring in XML

Autowiring by name

```
public class EmpDao {
        private DataSource dataSource;
        private GlobalInvestment resource;

        public void setDataSourceRef(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        public void setMainResource(GlobalInvestment resource){
                this.resource = resource;
        }
}
```

Declaration in XML

```
        <bean id="emp_list"
                class="pack_20_wireXml.EmpDao" autowire="byName" >

        <bean id=“dataSourceRef"
                class="org.springframework.jdbc.datasource.DriverManagerDataSource"
```

Autowiring in XML

Autowiring by type

```
public class EmpDao {
        private DataSource dataSource;
        private GlobalInvestment resource;

        public void setDataSourceRef(DataSource dataSource) {
                this.dataSource = dataSource;
        }

        public void setMainResource(GlobalInvestment resource){
                this.resource = resource;
        }
}
```

Declaration in XML

```
        <bean id="emp_list"
                class="pack_20_wireXml.EmpDao" autowire="byType" >

        <bean id=“dataSourceRef"
```

Autowiring using annotation

Autowiring using annotation by name and type

```
        @Repository("emp_list")
public class EmpDao {
        private DataSource dataSource;
        private GlobalInvestment resource;

         @Autowired @Qualifier("dataSourceRef")   // Autowiring by name
        public void setDataSourceRef(DataSource dataSource) {
                        this.dataSource = dataSource;
        }
        @ Autowired // Autowiring by type
        public void setMainResource(GlobalInvestment resource){
                        this.resource = resource;
        }
}
```

Declaration in XML

```
        <bean id=“dataSourceRef"
                        class="org.springframework.jdbc.datasource.DriverManagerDataSource"
```

Autowiring using @Resource

Autowiring using annotation by name and type

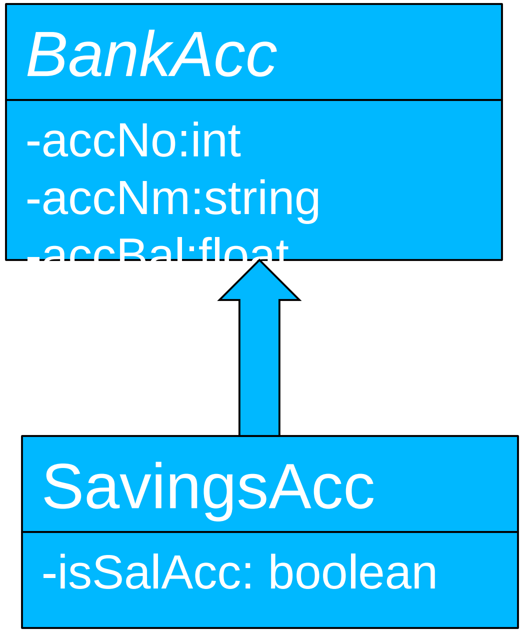
```
    @Repository("emp_list")
public class EmpDao {
    private DataSource dataSource;
    private GlobalInvestment resource;

     @Resource(name="dataSourceRef")  // Autowiring by name
    public void setDataSourceRef(DataSource dataSource) {
            this.dataSource = dataSource;
    }
    @ Resource // Autowiring by type
    public void setMainResource(GlobalInvestment resource){
            this.resource = resource;
    }
}
```

Declaration in XML

```
    <bean id="dataSourceRef"
            class="org.springframework.jdbc.datasource.DriverManagerDataSource"
```

Spring configuration for inheritance

```
public abstract class BankAcc {
                private int accNo;
                private String accNm;
                private float accBal;


                .............
}

Public class SavingsAcc extends BankAcc {
                private boolean isSalAcc;


                ............
}
```



Configuration in XML…

```
<bean id="bankAcc" class="pack_10_ioc.BankAcc"   abstract="true“  />

<bean id="savingsAcc" class="pack_10_ioc.SavingsAcc"  parent="bankacc"  />
```

Spring using factory

Why to use factory pattern instead of using Spring’s factory feature?

- Need to create objects of a singleton legacy class/from a non-spring application for a Spring application.

- A spring application needs to get objects from legacy factory class/factory class of a non-spring application.

- Need to create objects of different types on the basis of outcome of multiple statements/condition.

Spring using factory-Singleton (Contd…)

Spring creating object using factory method of a programmatic singleton class…

```
public class CommonResources {
        private static CommonResources resources;

        private GlobalInvestment companyDetails;
        …………..

        private CommonResources(){
                // Constructing and
                // assigning resources.
                ………………
        }

        public static CommonResources getResources(){
                System.out.println("Factory method.");
                if (resources == null){
                        resources = new CommonResources();
                }

                return resources;
        }
}
```

The XML configuration

```
<bean id="resources" class="pack_10.CommonResources"
    factory-method="getResources" >
 </bean>
```

- Here, the factory method clause takes property name of static method.
- The 'prototype' scope of Spring can not overrule programmatic singleton of a scope.
- The above code is trivial implementation of a programmatic singleton-ness. The factory-method clause works for thread-safe implementation also.

Spring using factory-Factory class (Contd…)

Spring creating object using factory class…

```
public class EntityFactory {

public SavingsAcc getNewSavingsAcc(){
        // Make decisions
        // Create other depending objects
        SavingsAcc saveAcc = new SavingsAcc();
        // Set dependencies.
        return saveAcc ;
}
```

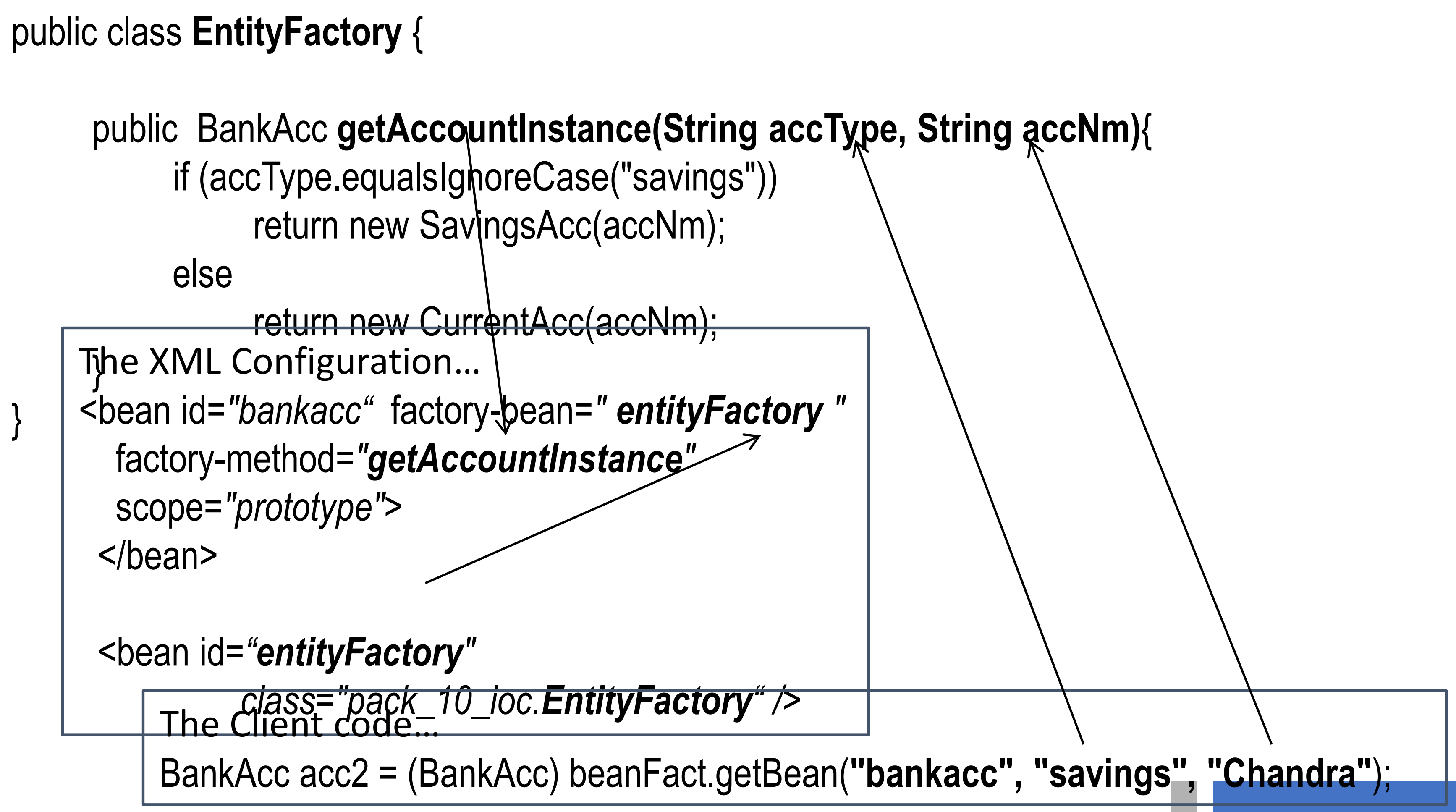
```
public CurrentAcc getNewCurrentAcc(){
        // Make decisions
        // Create other depending objects
        CurrentAcc currentAcc = new CurrentAcc();
        // Set dependencies.
        return currentAcc;
}
```

Configuration in XML…

```
<bean id="savingsAcc" class="pack_10_ioc.SavingsAcc"
   factory-bean=" entitiesFactory"
   factory-method="getNewSavingsAcc" scope="prototype">
</bean>


…………
 <bean id="entitiesFactory " class="pack_10_ioc.EntityFactory" />
```

Spring using factory- Parameterized (Contd…)

Spring creating object using parameterized factory class…



```
public class EntityFactory {

        public  BankAcc getAccountInstance(String accType, String accNm){
                if (accType.equalsIgnoreCase("savings"))
                        return new SavingsAcc(accNm);
                else
                        return new CurrentAcc(accNm);
        }
}
```

The XML Configuration…

```
<bean id="bankacc“  factory-bean=" entityFactory "
   factory-method="getAccountInstance"
   scope="prototype">
 </bean>

 <bean id=“entityFactory"
       class="pack_10_ioc.EntityFactory“ />
```

The Client code…

```
BankAcc acc2 = (BankAcc) beanFact.getBean("bankacc", "savings", "Chandra");
```

Spring using factory- FactoryBean (Contd…)

Spring creating object using FactoryBean implementation…

```
public class EntityFactory  implements FactoryBean<BankAcc> {

public BankAcc getObject() throws Exception {
            // Make decisions
            // Create other depending objects
            BankAcc bcc = new BankAcc();
            // Set dependencies.
            return bcc;
      }

      public boolean isSingleton() {

            return false;
      }
}
```

Configuration in XML…

```
<bean id="bankAcc" class="pack_10_ioc.Entity Factory"  />
```

The Client code…

```
BankAcc acc2 = (BankAcc) beanFact.getBean("bankAcc");
```

Bean referring to Context

Bean can receive a reference of Spring Context in two ways...

I. Injection...

```
public class NewsPrintManager implements ApplicationContextAware {

        private ApplicationContext ctx;

        public void setApplicationContext(ApplicationContext ctx)
        throws BeansException {

                this.ctx = ctx;
                }
}  // The setter is called automatically as a part of bean creation by Spring container.
```

II. Resource lookup...

```
public class NewsPrintManager  {
                @ Resource/@Autowire
        private ApplicationContext ctx;

} // Field reference is automatically set after object is created.
```

Dependency Relationship

A method of an object does lookup for another object for fetching service or delegating responsibilities.

```
public class NewsPrintManager  {

        @Resource
        private ApplicationContext ctx;


        public void printNews(){

                TodaysNews dtNews = ctx.getBean("todays_news", TodaysNews.class);
                ......  // Fetch service from TodaysNews
        }
}
```

Configuration in XML…

```
   <bean id="news_manager" class="pack_10_ioc.NewsPrintManager" />

  <bean id="todays_news"  class="pack_10_ioc.TodaysNews" />
```

Module 4. Bean life cycle and callback methods

- **Overview**
  - Beans with configurable initial state;
  - Container call-back methods through interfaces
  - Annotations for container call-back methods

Bean Construction and destruction

There are different ways to declare container call back methods for a bean. These call back methods form a life cycle.

- The init-method/destroy-method clauses.
- The InitializingBean and DesposableBean.
- The @ PostConstruct and @ PreDestroy annotations.

The init-method, destroy-method clauses

Object with multiple sets of initial data or initialization resources

```
public class CouponsTracking {
        private boolean breakFastCoupon;
        private boolean lunchCoupon;

        public void initForTraining(){
                        …… // Allocate initial state or resources
                }

                public void initForSeminar(){
                        …… // Allocate initial state or resources
                }

        public void releaseResources(){
                …..
        }
}
```

The XML Configuration…

```
        <bean id="trackCoupon“ class="pack_20_lifecycle.CouponsTracking"
```

The InitializingBean, DisposableBean interfaces

Spring’s way to validate configuration inputs…

```
public class LeavesConfiguration implements InitializingBean, DisposableBean{
        private int maxPLAnnually;

        // Implementation of InitializingBean
        public void afterPropertiesSet() throws Exception {
                // Validating configured inputs and/or allotting mandatory resources to bean
                if (maxPLAnnually>22)
                                throw new Exception("Wrong configuraton of max PL");
                }

        // Implementation of DisposableBean
        public void destroy() throws Exception {
                // De-allotting resources if any
        }
```

This method is automatically executed after all configured properties are called on the created object.

The configuration in XML

```
        <bean id="leavesLimits“   class="pack_20_lifecycle.LeavesConfiguration"  >
                <property name="maxPLAnnually">
                                <value>25</value>  <!– The afterPropertySet() validates this value. --
                >
```

The @PostConstruct and @PreDestroy

Spring’s way to assign mandatory resources on condition…

```
@ Component
public class LeavesConfiguration  {
        private int maxPLAnnually;

        @PostConstruct
        public void assignDefaultResources() throws Exception {
                // Assign logger.
        }

        @PostConstruct
        public void assignSpecificResources() throws Exception {
                // Assign persistent storage.
        }

        @PreDestroy
        public void releaseResources() throws Exception {
                // Close all resources
        }
}
```

The @PostConstruct and @PreDestroy (Contd...)

The configuration in XML...

```
<context:component-scan base-package="pack_20_lifecycle.pack30_annotate" />
```

Points to note...

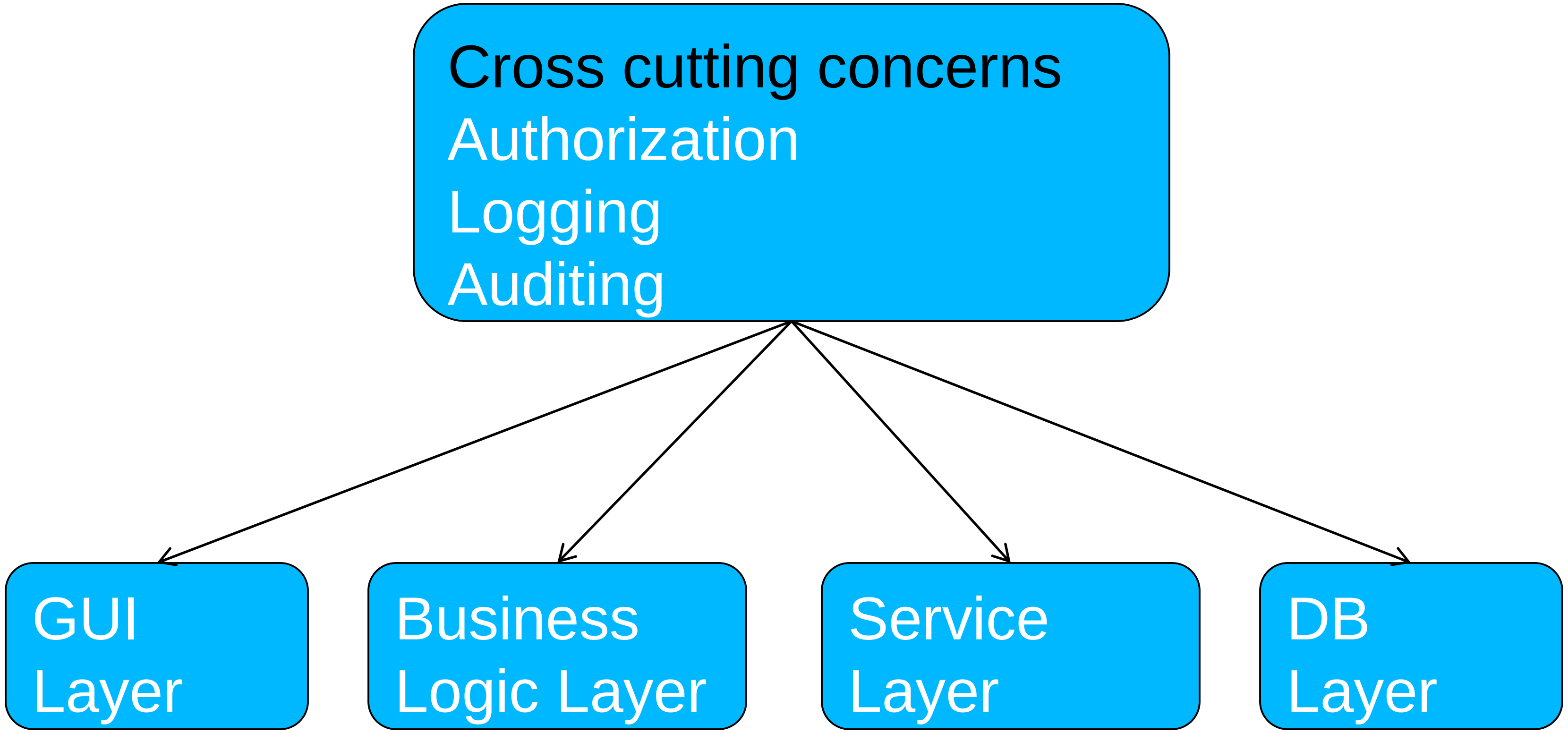
- Sets of initialization statements can be segregated across multiple methods.
- Such multiple methods are executed in the physical order from top to bottom.
- Annotation gives liberty to give context specific meaningful name to Methods.
- This approach is preferred for being Java standard.

Module 5. Introduction to Aspect Oriented Programming

- **Overview**

  - AOP for cross cutting concerns;
  - Why AOP?
  - Understanding aspects, advices, interceptors, join-points, point-cuts etc;
  - Weaving techniques;
  - Configuring proxies for executing aspects;

Cross cutting concerns



Cross cutting concerns
Authorization
Logging
Auditing

GUI Layer

Business Logic Layer

Service Layer

DB Layer

Few AOP terminologies

- **Aspect**: A modularization of concerns that cuts across multiple classes.

- **Join point**: A point during execution which is intercepted by aspects.

  In AspectJ, methods and fields can be join points. Any call to method or change in value of a field triggers interception of advices.

  ```
  private int accBal;
  private void withdraw(float amt);
  ```

- **Advices**: Action taken by aspect at a particular join point.

  The authentication, logging logics can be advices.

- **Point cut**: A predicate that matches join points.

  AspectJ notation to decide targets for advicing. These targets are normally in the form of join points.

  execution(pack_40_aop.pack_10_aspect.pack_joinpoints.pack_dao.*.*(..))*

Why AOP?

- **Manual weaving...**

```
public class EmpDao {
        public Collection getAllEmps(){
              // Call to interceptor1- Logger
              // Call to interceptor2- Authentication check

              // Logic to fetch and populate data in collection

              // Call to interceptor3- Transaction management
              // Call to interceptor4- Auditor
        }
}
```

Why AOP? (Contd...)

- The code for cross cutting concerns appear like a boiler plate code most of the time.
- The business logic clutters up with such a code. Thus hampers readability.
- Using AOP, business logic becomes clean.
- A separate layer of advices can be designed. Managed and administrated separately.
- The advices become completely transparent to join points.

Why AOP? (Contd...)

Business logic layer

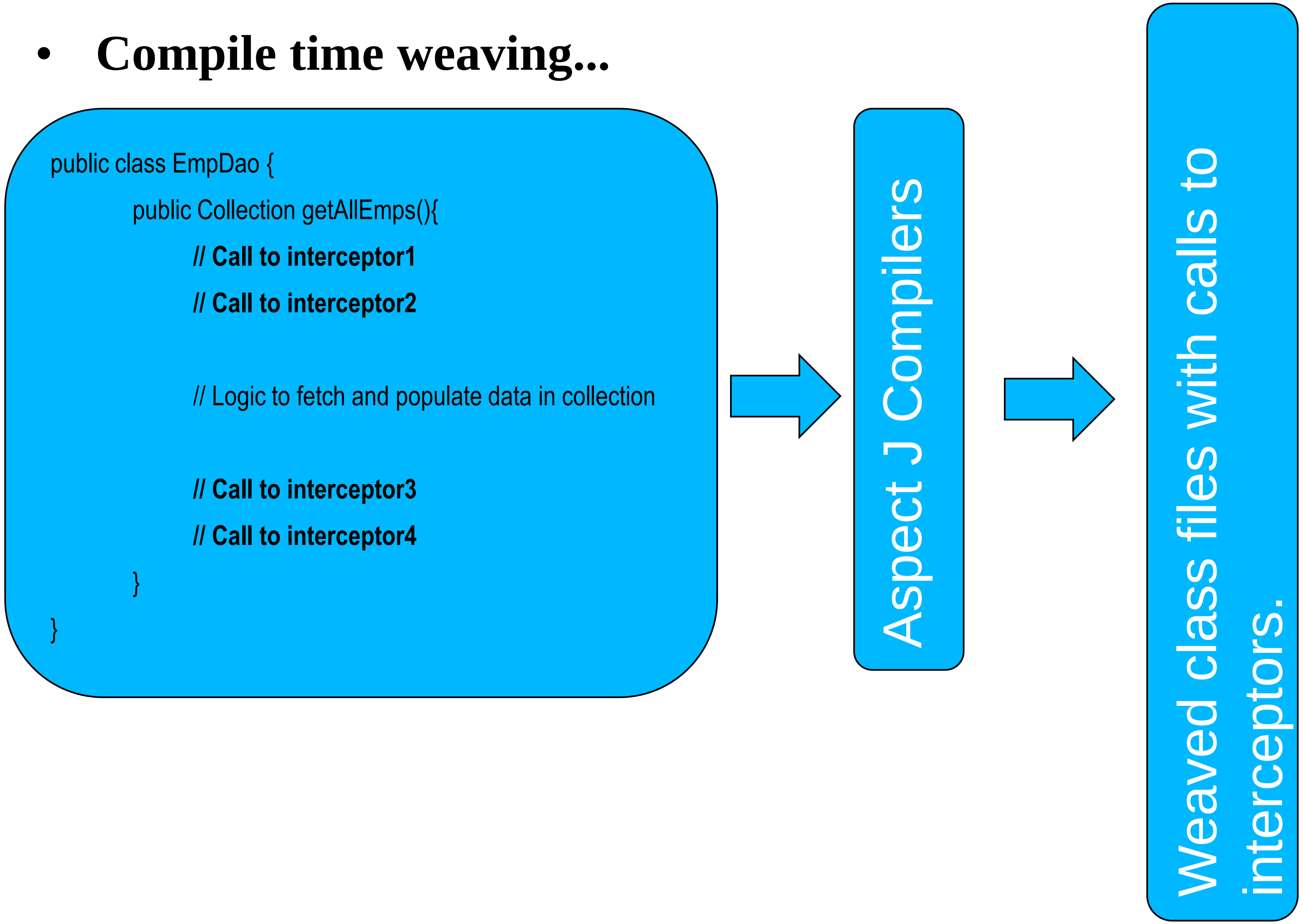
```
public class EmpDao {
        public Collection getAllEmps(){
                // Logic to fetch and populate
                data in collection
                }
}
```

Layer of cross cutting concerns

```
public class Loggers {
        public void log(String msg){
                // Code doing logging
                }
}
```

Weaving techniques

- **Compile time weaving...**



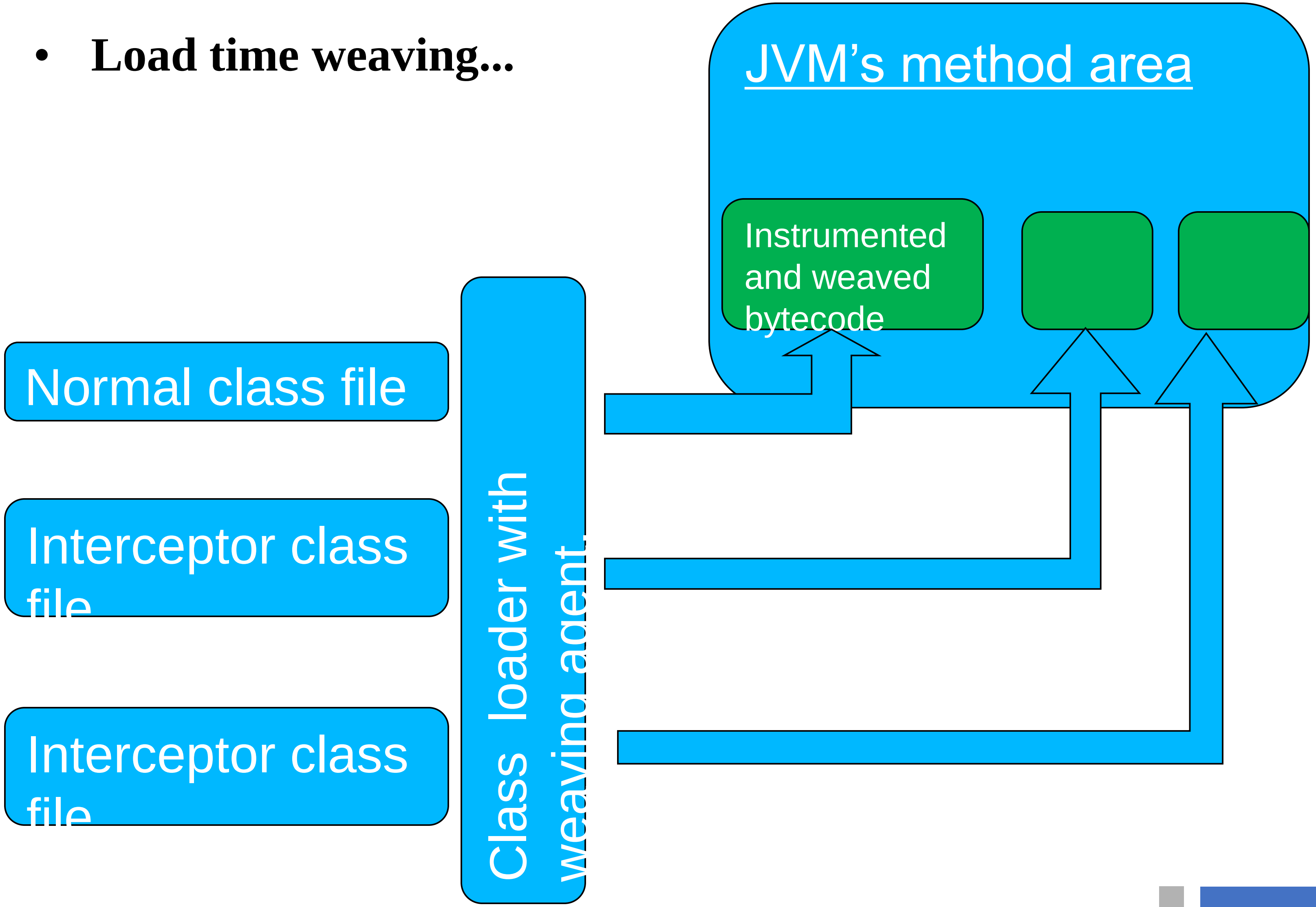
```
public class EmpDao {
        public Collection getAllEmps(){
              // Call to interceptor1
              // Call to interceptor2

              // Logic to fetch and populate data in collection

              // Call to interceptor3
              // Call to interceptor4
        }
}
```

Aspect J Compilers

Weaved class files with calls to interceptors.

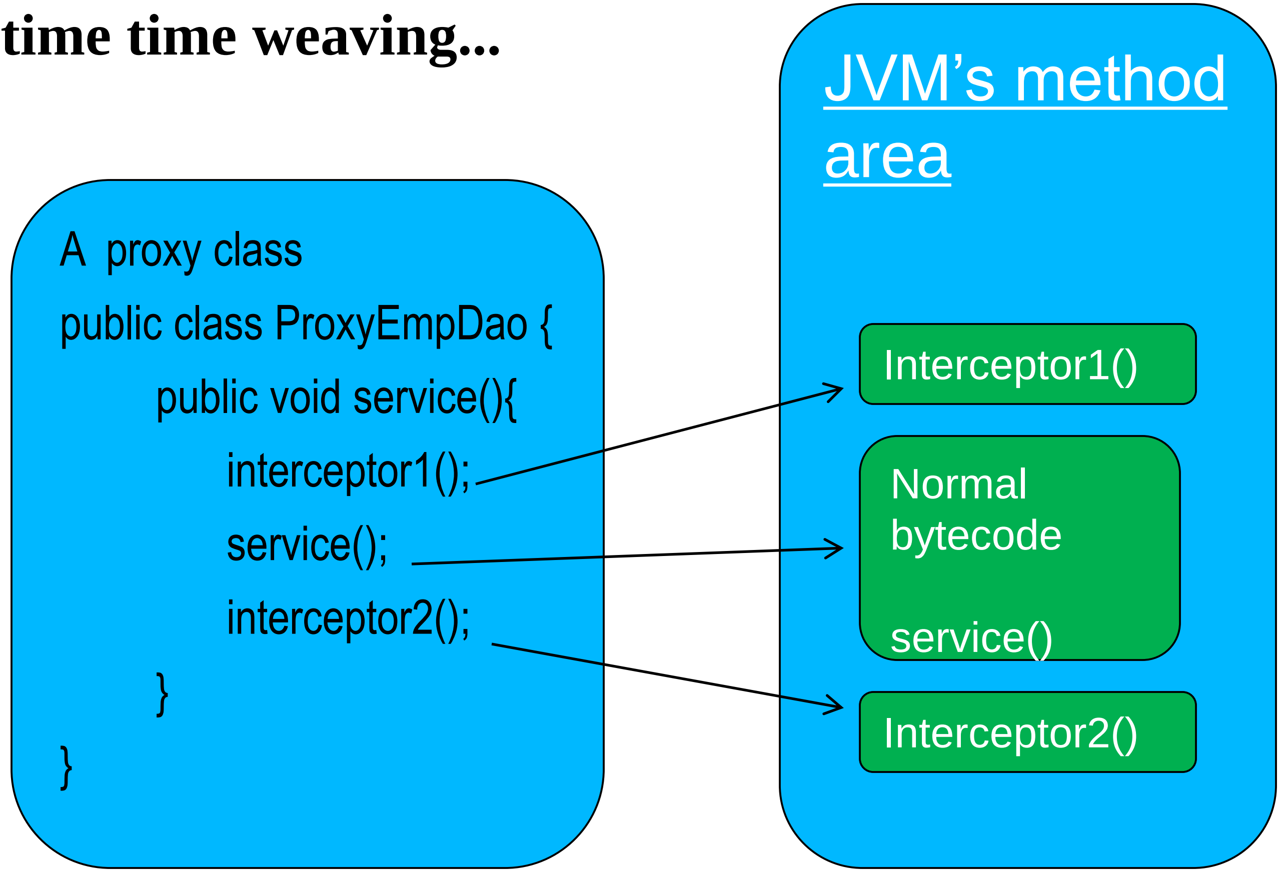
Weaving techniques

- **Load time weaving...**



Weaving techniques

- **Runtime time weaving...**



A proxy class

```
public class ProxyEmpDao {
        public void service(){
                interceptor1();
                service();
                interceptor2();
        }
}
```

Join points…

Methods as join points…

```
public class CustomerServiceImpl implements CustomerService {

        public void applyChqBk(long acno) {
                // Logic to register requisition for Cheque book
                .............
        }

        public void applyForCreditCard(String name, String address) {
                // Logic to register requisition for Credit card
                ...............
        }

        public void stopChequePayment(long acno, int chequeNo) {
                // Logic to stop cheque payment
                .....................
        }
}
```

Advice...

Around type of advice...

```
public class LogAdvice {

        public Object log(ProceedingJoinPoint call) throws Throwable {

                System.out.println("\nfrom logging aspect: before method  call");

                Object point = call.proceed(); // Actual method call

                System.out.println("from logging aspect: after method call");
                return point;
        }
}
```

Pointcut: Predicate deciding joinpoint targets

Configuration in XML...

```
<aop:config>
        <aop:aspect ref="loggingInterceptor">
                <aop:pointcut id="pointcut1" expression="execution(* pack_40_aop.*.a*(..))" />
                <aop:around pointcut-ref="pointcut1" method="log" />
        </aop:aspect>
</aop:config>
```

Advices and Pointcuts using annotations

Around type of advice…

```
@Aspect
public class LogAdvice{

        @Around (“execution(* pack_40_aop.*.a*(..))")
    public Object logMessage(ProceedingJoinPoint call) throws Throwable {

        System.out.println("\nfrom logging aspect: before method  call");

        Object point = call.proceed(); // Actual method call

        System.out.println("from logging aspect: after method call");
        return point;
    }
}
```

Configuration in XML…

```
<aop:aspectj-autoproxy />

<bean id="customerService" class="pack_40_aop.CustomerServiceImpl" />
```

Module 6. The AOP advice types and point cuts

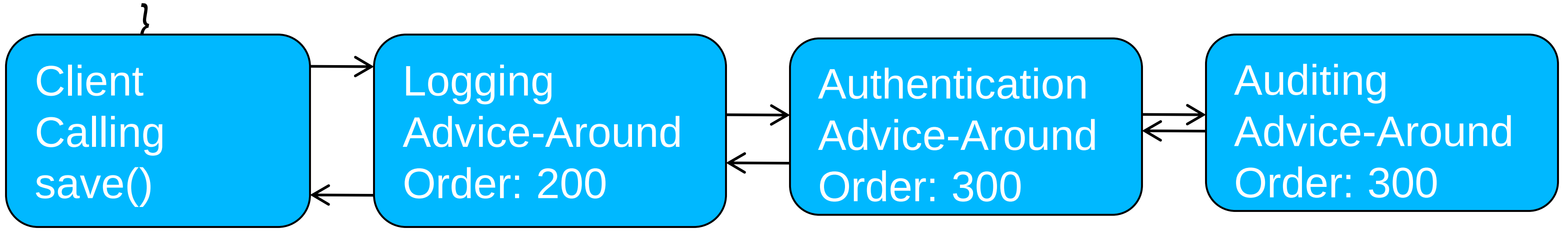
- **Overview**

  - ➢ Ordering aspects
  - ➢ Different advice types
  - ➢ Different point-cuts
  - ➢ Named point-cuts

Ordering aspects

- Aspects should be segregated advice wise. It may be needed to have different sets of aspects at different join points in an application. For a join point, if there are multiple aspects, their order may be relevant.

```
public class AuthorizationAspect implements Ordered {
        private int order;

        // Interface imposes implementation of this method.
public int getOrder() {
        return order;
}
```



Client Calling save()

Logging Advice-Around Order: 200

Authentication Advice-Around Order: 300

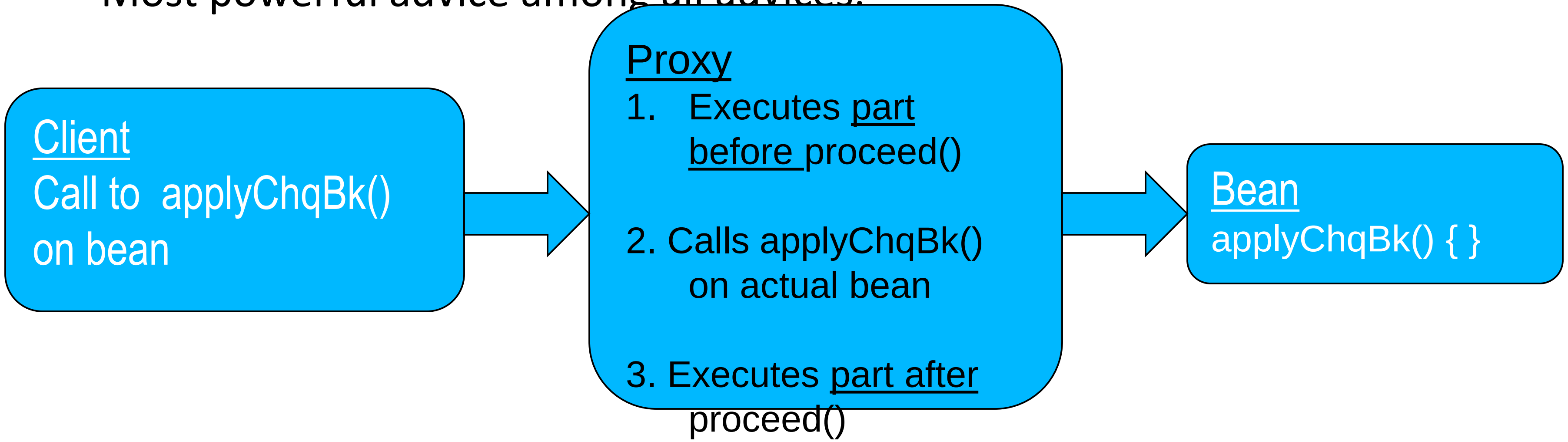
Auditing Advice-Around Order: 300

Order value can be set through XML configuration. Larger order nests the exception.

Advice Types: Around

@Around ("execution(* pack_40_aop.*.a*(..))")

- Most powerful advice among all advices.



Client
Call to applyChqBk()
on bean

Proxy
1. Executes part before proceed()
2. Calls applyChqBk() on actual bean
3. Executes part after proceed()

Bean
applyChqBk() { }

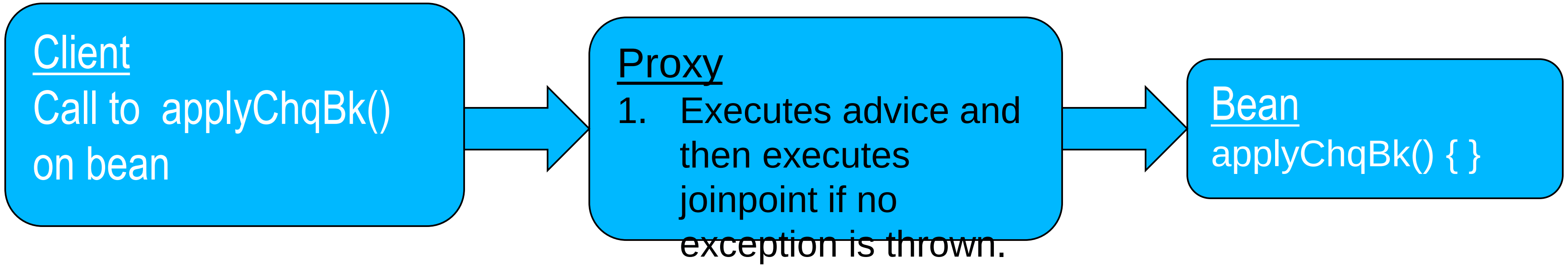
- Can bypass call to actual method by ignoring call to proceed();

- Can call actual method more than once by calling proceed() more than once.

Advice Types: Before

@Before (“execution(* pack_40_aop.*.a*(..))")

- Works like pre-call part of try block.



Client
Call to applyChqBk()
on bean

Proxy
1. Executes advice and then executes joinpoint if no exception is thrown.

Bean
applyChqBk() { }

- Can not call proceed(). No choice of at what point of time to execute a call to actual method.

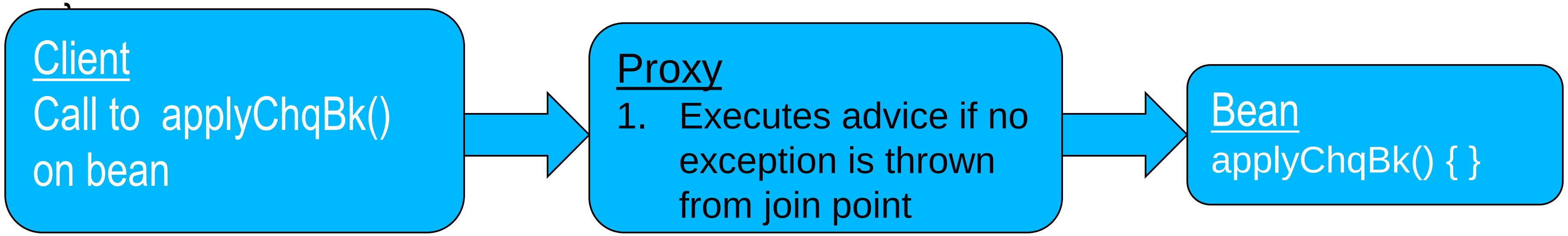
- After this advice actual method is bound to execute except no exception is thrown.

Advice Types: AfterReturning

- Works like post-call part of try block.

@AfterReturning (“execution(* pack_40_aop.*.a*(..))“, returning=“retValue”)

```
public void businessLogic(JoinPoint jp, Object retVal) {
                .......................
}
```



Client
Call to applyChqBk() on bean

Proxy
1. Executes advice if no exception is thrown from join point

Bean
applyChqBk() { }

- Can not call proceed(). No choice of at what point of time to execute a call to actual method.

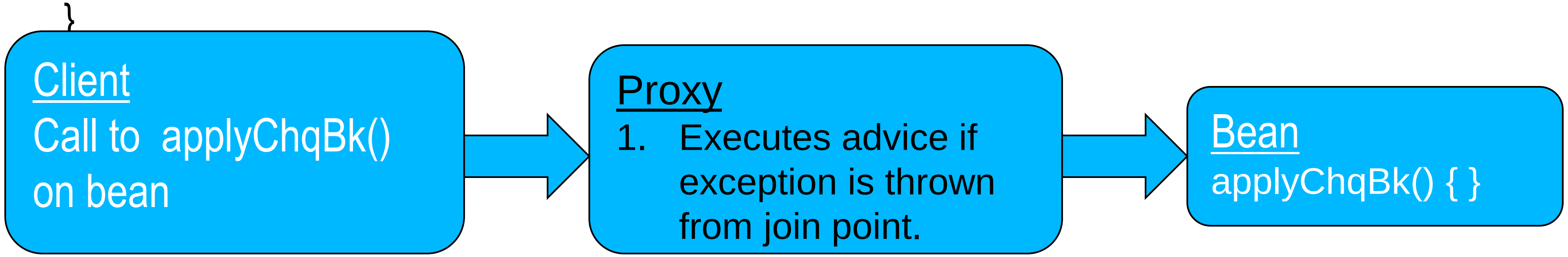
- After successful execution of actual method, this advice is executed.

- Can be used to analyze the return value of a method. If return value breaches validity, free to update and return default value.

Advice Types: AfterThrowing

- Works like catch block.

```
@AfterThrowing(“execution(* pack_40_aop.*.a*(..))“, throwing=“ex”)
public void handleException(JoinPoint jp, RuntimeException ex) throws Throwable {
            // Handling exception and raising custom exception
}
```



Client
Call to applyChqBk() on bean

Proxy
1. Executes advice if exception is thrown from join point.

Bean
applyChqBk() { }

- Can not call proceed(). No choice of at what point of time to execute a call to actual method.

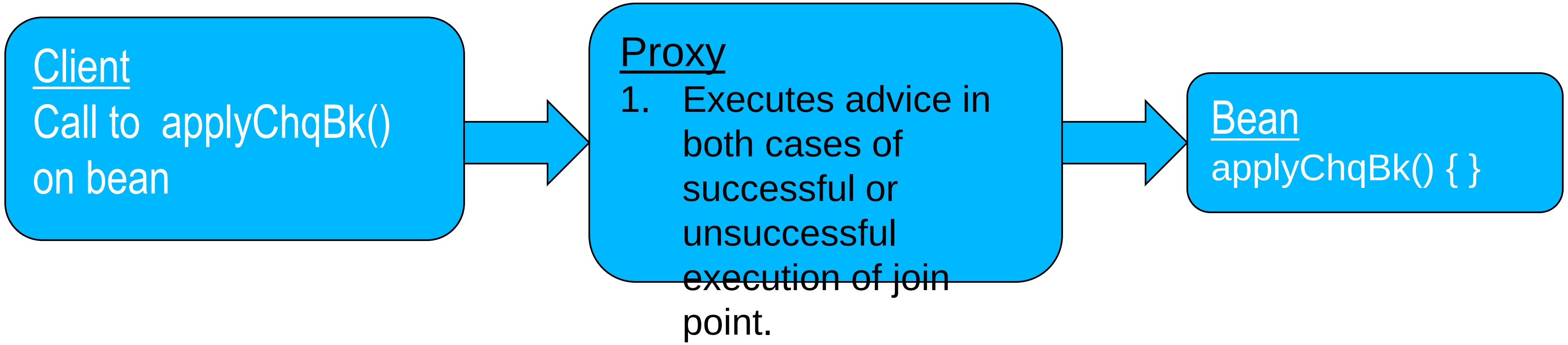
- After un-successful execution of actual method, this advice is executed.

- Can be used to overcome the exception thrown and can throw outer layer specific exception. Thereby, it makes a business logic clutter free of try-catch block.

Advice Types: After

@AfterThrowing("execution(* pack_40_aop.*.a*(..))")

- Works like finally block.



Client
Call to applyChqBk() on bean

Proxy
1. Executes advice in both cases of successful or unsuccessful execution of join point.

Bean
applyChqBk() { }

- Can not call proceed(). No choice of at what point of time to execute a call to actual method.

- What ever be the fate of execution of actual method, this advice is executed.

- Can be used to replace finally block.

Pointcuts

- If join points are all methods within all classes of ‘service’ package

  @Pointcut(“within(com.xyz.service.*))

- If join points are all methods within all classes of package ‘web’ and its all subpackages.

  @Pointcut(“within(com.xyz.someapp.web..*)”)

- If join points are all methods taking any number of parameters (..) of all classes of service package excluding classes of sub-package of service package.

  execution(* com.xyz.service.*.*(..))

- If join points are all methods of implementation of interface AccountService of package ‘service’.

  execution(* com.xyz.service.AccountService.*(..))

- Joints points as any method annotated with given annotation.

  execution(@Log * *(..))

Named Point-cuts

Named point cuts designs a tier of point cuts separate from advices. It makes actual point-cuts transparent from advices as well as from join points.

Naming Point-cuts in Java code

```
@Aspect
public class PointcutConfig {

        @Pointcut("execution(public * pack_40_aop.pack_20_pointcut.*.*(..))“)
        public void serviceComponents() {} //This is a name given to the pointcut
expression
}
```

Configuration in XML

```
        <aop:aspectj-autoproxy />
        <bean id="pointcutConfig" class="pack_40_aop.pack_20_pointcut.PointcutConfig"
/>
```

Declaring Advices

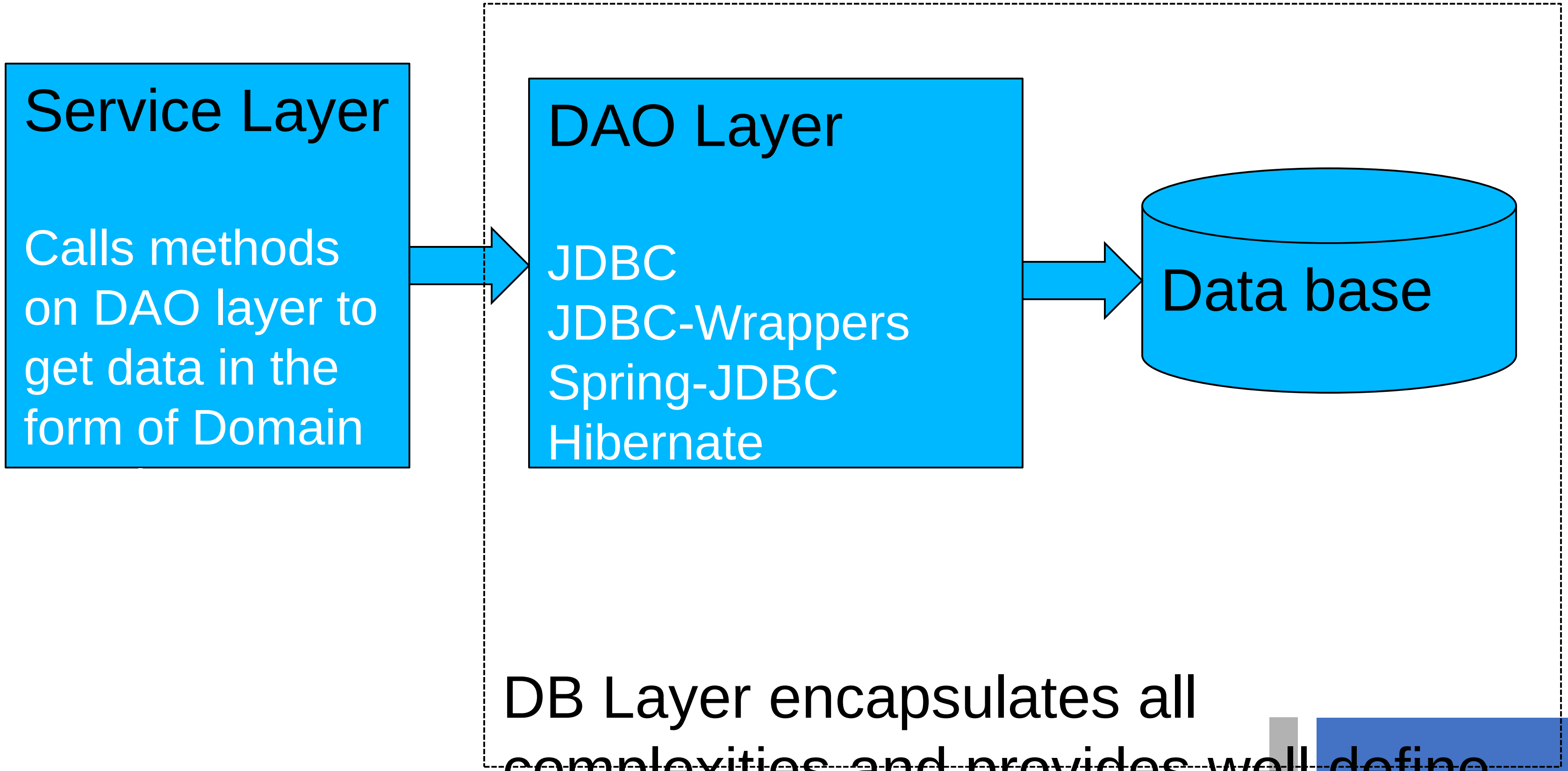
```
@Aspect
public class LoggingAspect {

    @Before("pack_40_aop.pack_20_pointcut.PointcutConfig.serviceComponents()")
```

Module 7. Spring JDBC

- **Overview**
  - Role of Spring-JDBC
  - Issues with low level JDBC and solution by Spring
  - Spring's abstraction for DAO classes
  - Spring's JDBC support
    - Using JdbcDaoSupport and JdbcTemplate class
    - Using SimpleJdbcTemplate
    - Calling stored procedures

Role of Spring JDBC

Its a solution for designing database access layer(DAO). Its a kind of JDBC Wrapper to provide abstraction for DB access.



Service Layer

Calls methods on DAO layer to get data in the form of Domain

DAO Layer

JDBC
JDBC-Wrappers
Spring-JDBC
Hibernate

Data base

DB Layer encapsulates all complexities and provides well define interfaces for Service layer

Problems with JDBC

```
public List<Emp> getAllEmps() throws SQLException{
		String qry = "Select EMPNO, ENAME from EMP";
		Statement stmt = null;
		ResultSet rs = null;
		Connection conn = null;

		try {
				conn = cf.getConnection();
				stmt = conn.createStatement();

				rs = stmt.executeQuery(qry);
				List<Emp> lst = new ArrayList<Emp>();

				while(rs.next()){
						Em
						e.
						e.
						ls
				}
		} finally{
				rs.close();
				stmt.close();
				conn.close();
		}
```

- Code redundancy: Most of this code is written for another method like getEmpsWithSalRange(). Most of the code is boiler plate code.

Spring JDBC abstraction…

```
String query="Select EMPNO, ENAME, SAL from EMP";
List<Map<String, Object>> list =
simpleJdbcTemplate.queryForList(query, new Object[] {});
```

Problems with JDBC

```
public void joinNewEmployee(Emp e) throws SQLException{
        String qry = "INSERT INTO EMP (EMPNO, ENAME) VALUES(?, ?)";
        Connection conn = cf.getConnection();
        PreparedStatement pstmt = null;

        try {
                conn.setAutoCommit(false);
                pstmt = conn.prepareStatement(qry);

                pstmt.setInt(1, e.getEmpNo());
                pstmt.setString(2, e.getEmpNm());

                pstmt.execute();
                conn.commit();
        } catch(SQLException ex){
                conn.rollback();
                throw new
        }       finally{
                pstmt.clos
                conn.setA
        }
}
```

Spring JDBC abstraction…

```
query = "Insert into EMP(EMPNO, ENAME, SAL) values (?, ?, ?)";
jdbcTemplate.update(query, new Object[]{9000, "Suchita", 25000});
```

Features- Spring JDBC

Its a solution for designing database access layer(DAO).

- The DAO layer isolates application logic from Database complexities i.e. Encapsulates DB complexities.
- Manages database resources effectively. Provides configurable, automatic connection management.
- Provides support for data access technologies like JDBC, Hibernate, JPA.
- Eliminates boiler plate code thus reduces likelihood of bugs.

Features- Spring JDBC (Contd…)

- Provides declarative transaction management. Methods form transaction boundary and takes part in existing transaction or starts own new transaction.

- Better exception handling. Wide hierarchy of exceptions, proper release of resources etc.

- Converts result-set into domain specific graph.

- Simplifies use of JDBC by providing kind of JDBC Wrappers.

who does what?

Action	Spring	You
Define connection parameters.		X
Open the connection.	X	
Specify the SQL statement.		X
Declare parameters and provide parameter values		X
Prepare and execute the statement.	X	
Set up the loop to iterate through the results (if any).	X	
Do the work for each iteration.		X
Process any exception.	X	
Handle transactions.	X	
Close the connection, statement and resultset.	X	

Creating DriverManager and JdbcTemplate

Configuration in XML…

```
<bean id="ds“    class="org.springframework.jdbc.datasource.DriverManagerDataSource“>
            <property name="driverClassName" value="oracle.jdbc.OracleDriver"/>
            <property name="url" value="jdbc:oracle:thin:@ipOfDBServer:1521:orcl"/>
            <property name="username" value="scott"/>
            <property name="password" value="tiger"/>
 </bean>

 <bean id=“templateDS“  class="org.springframework.jdbc.core.JdbcTemplate">
        <property name="dataSource“    ref="ds“ />
</bean>
```

Getting bean reference from Spring Container…

```
JdbcTemplate jdbcTemplate = (JdbcTemplate)beanFactory.getBean("templateDS");
```

Using JdbcTemplate

- Retrieving single primitive…

```
int rowctr = jdbcTemplate.queryForInt("select COUNT(0) from EMP");
```

- Retrieving single object by binding parameters for interactive query…

```
String query = "Select ENAME from EMP where EMPNO=?";
String empName =  jdbcTemplate.queryForObject(query, new Object[] {7499}, String.class);
```

- Retrieving list of records…

```
String query = "Select EMPNO, ENAME, SAL, COMM from EMP";
List list = jdbcTemplate.queryForList(query, new Object[] {});
```

- DML statements by binding parameters for interactive query…

```
String query = "Insert into EMP(EMPNO, ENAME, SAL) values (?, ?, ?)";
// query = "Update EMP set ENAME=? where EMPNO=?";
//  query = "Delete from EMP where EMPNO=?";
jdbcTemplate.update(query, new Object[]{9000, "Suchita", 25000});
```

Creating and using NamedParaJdbcTemplate

Configuration in XML…

```
<bean id=“namedTemplateDS“
        class="org.springframework.jdbc.core.namedparam.NamedParameterJdbcTemplate">
        <!-- It does not have property to set DataSource.  It has a constructor -->
        <constructor-arg>
                <ref bean="ds"/> <!– Reference to DriverManager Bean -->
        </constructor-arg>
</bean>
```

Mapping parameters with interactive query…

```
        NamedParameterJdbcTemplate namedTemplate =

        (NamedParameterJdbcTemplate)beanFact.getBean(“namedTemplateDS ");


        String query = "Select SAL from EMP where EMPNO=:givenImpId";

        // Mapping the named parameter with actual value
        SqlParameterSource namedPara = new MapSqlParameterSource("givenImpId"
```

Creating SimpleJdbcTemplate and Mapper

Configuration in XML…

```
<bean id=“simpleTemplateDS“
            class="org.springframework.jdbc.core.simple.SimpleJdbcTemplate">
            <!-- It does not have property to set DataSource.  It has a constructor -->
            <constructor-arg>
                        <ref bean="ds"/> <!– Reference to DriverManager Bean -->
            </constructor-arg>
</bean>
```

Creating mapper for domain object…

```
ParameterizedRowMapper<EmpPojo> mapper = new ParameterizedRowMapper<EmpPojo>(){

      public EmpPojo mapRow(ResultSet rs, int rowNum) throws SQLException{
            EmpPojo emp = new EmpPojo();
                  emp.setEmpNo(rs.getInt("EMPNO"));
                  emp.setEmpNm(rs.getString("ENAME"));
                  emp.setEmpSal(rs.getFloat("SAL"));
            return emp;
      }
};
```

Using SimpleJdbcTemplate

Get reference to SimpleJdbcTemplate…

```
SimpleJdbcTemplate simpleDS = (SimpleJdbcTemplate)beanFact.getBean("simpleTemplateDS ");
```

Get a single customized object…

```
        String query = "Select EMPNO, ENAME, SAL from EMP where EMPNO=?";
        EmpPojo emp = simpleDS.queryForObject(query, mapper, 7521);
```

Get a list of customized objects…

```
        String query="Select EMPNO, ENAME, SAL from EMP";
        List<Map<String, Object>> list = simpleDS.queryForList(query, mapper, new Object[]
{});
```

Using SimpleJdbcCall

CreateSimpleJdbcCall…

```
        SimpleJdbcCall procCall = new
SimpleJdbcCall(ds).withProcedureName("procMaxEmpNo");

        Map<String, Object> out = procReadEmployee.execute();

        BigDecimal bd = (BigDecimal) out.get("MAXENO");
```

Build COMPETENCY
across your TEAM

SYNERGETICS GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

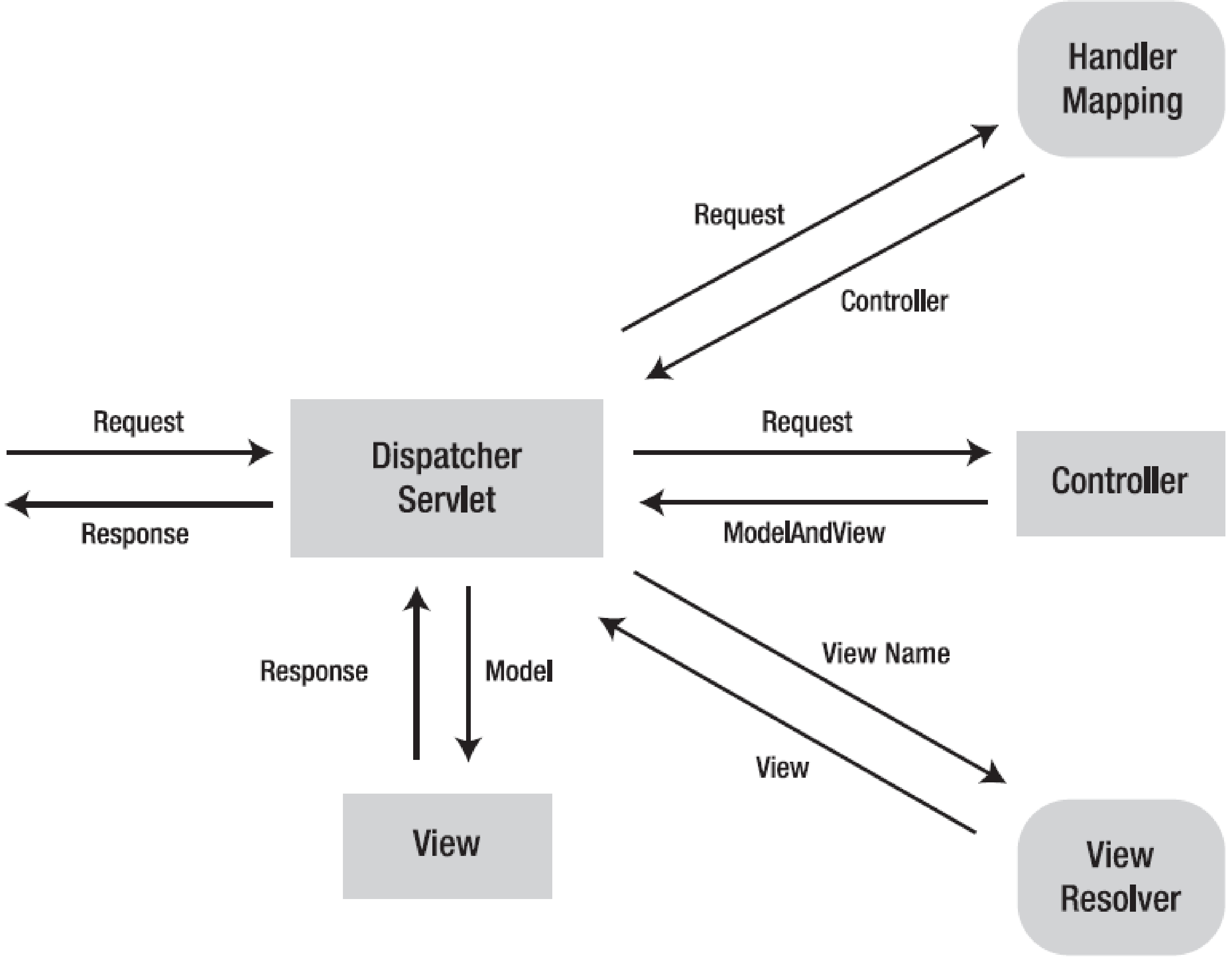
Spring MVC

Dispatcher Servlet

- The central component of Spring MVC is DispatcherServlet.

- It acts as the front controller of the Spring MVC framework, and every web request must go through it so that it can manage the entire request-handling process.

Flow of Request



Handler Mapping

- A handler mapping is a bean configured in the web application context that implements the **HandlerMapping** interface. It is responsible for returning an appropriate handler for a request.

- Handler mappings usually map a request to a handler according to the request’s URL.

- A **handler** is an arbitrary Java object that can handle web requests.
- The most typical handler used in Spring MVC for handling web requests is a ***controller***.

ModelAndView

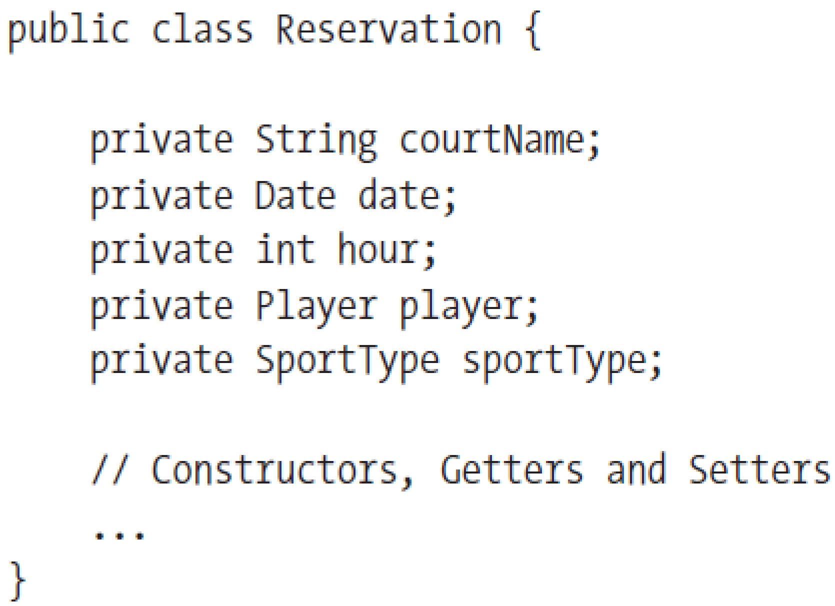
- After a controller has finished handling the request, it returns a model and a view name, or sometimes a view object, to DispatcherServlet.
- The model contains the attributes that the controller wants to pass to the view for display.
- If a view name is returned, it will be resolved into a view object for rendering. The basic class that binds a model and a view is ModelAndView.

ViewResolver

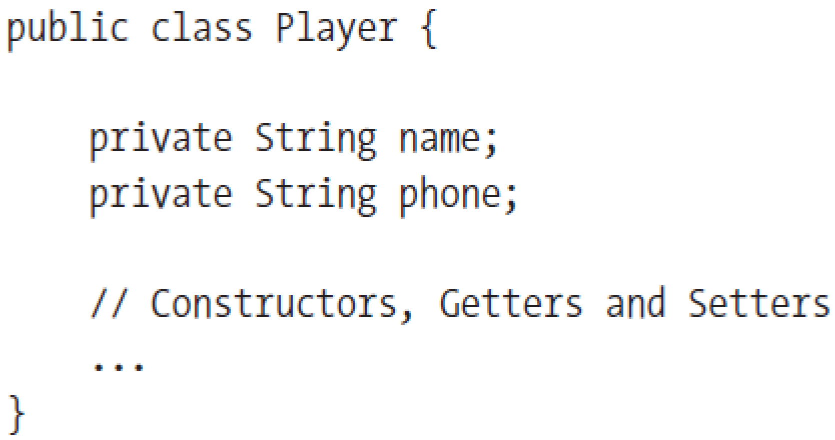
- When DispatcherServlet receives a model and a view name, it will resolve the logical view name into a view object for rendering.
- DispatcherServlet resolves views from one or more view resolvers.
- **A view resolver is a bean configured in the web application context that implements the ViewResolver interface.** Its responsibility is to return a view object for a logical view name.

Example

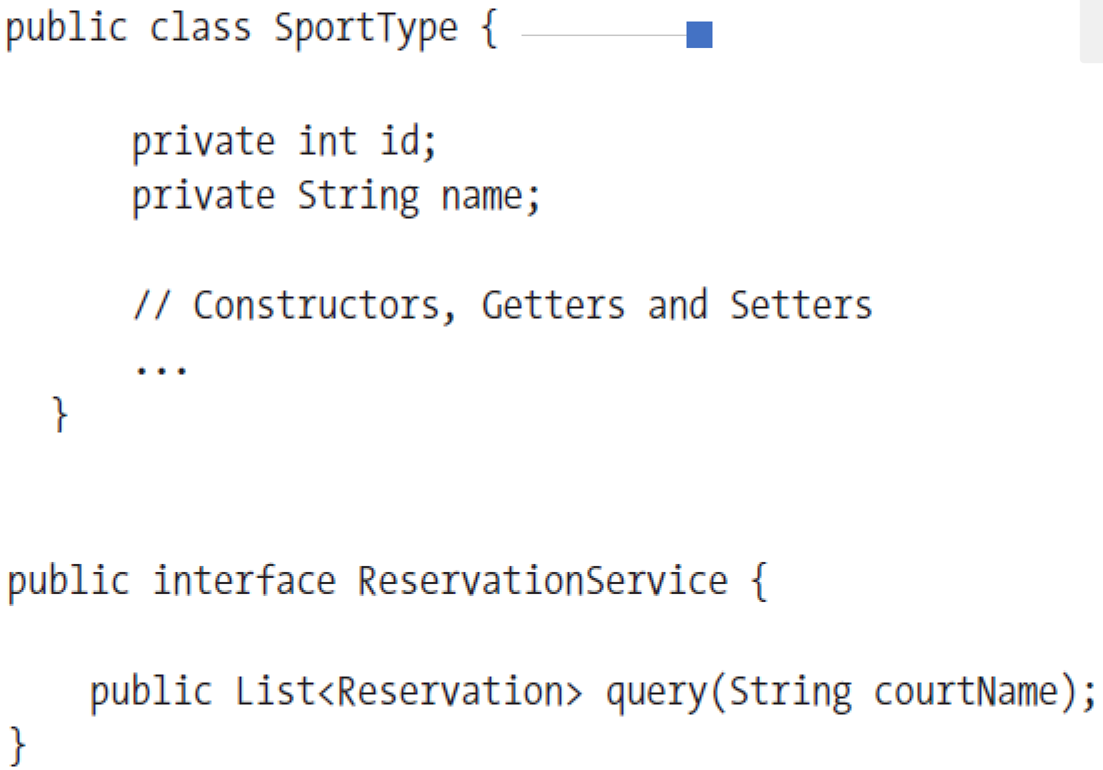
- We are going to develop a court reservation system for a sports center. Following are the classes in our application :



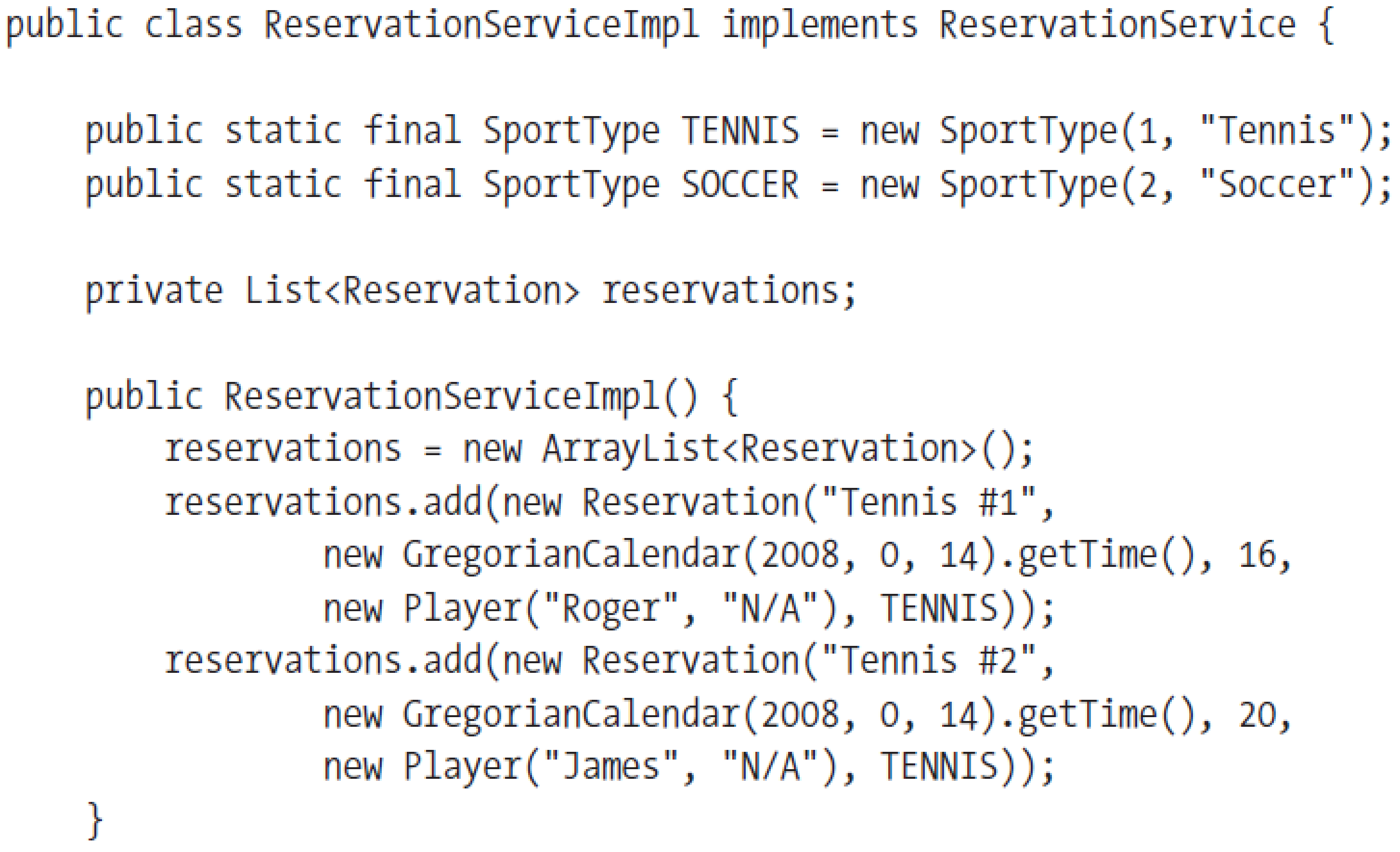
```
public class Reservation {

    private String courtName;
    private Date date;
    private int hour;
    private Player player;
    private SportType sportType;

    // Constructors, Getters and Setters
    ...
}
```



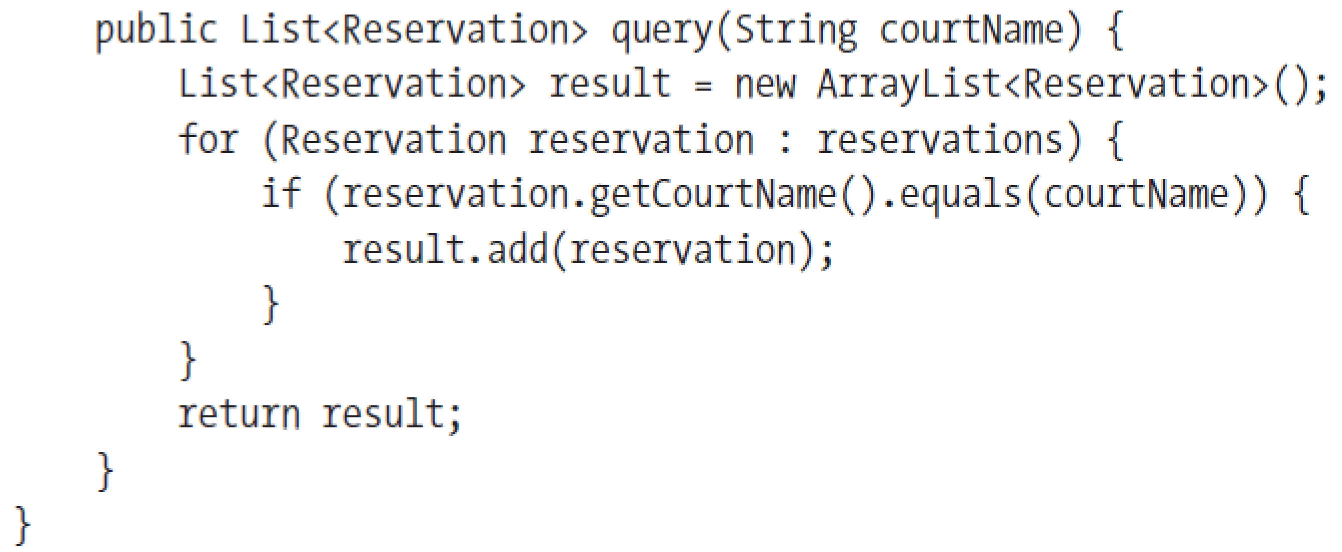
```
public class Player {

    private String name;
    private String phone;

    // Constructors, Getters and Setters
    ...
}
```



```
public class SportType {

        private int id;
        private String name;

        // Constructors, Getters and Setters
        ...
   }


public interface ReservationService {

     public List<Reservation> query(String courtName);
}
```



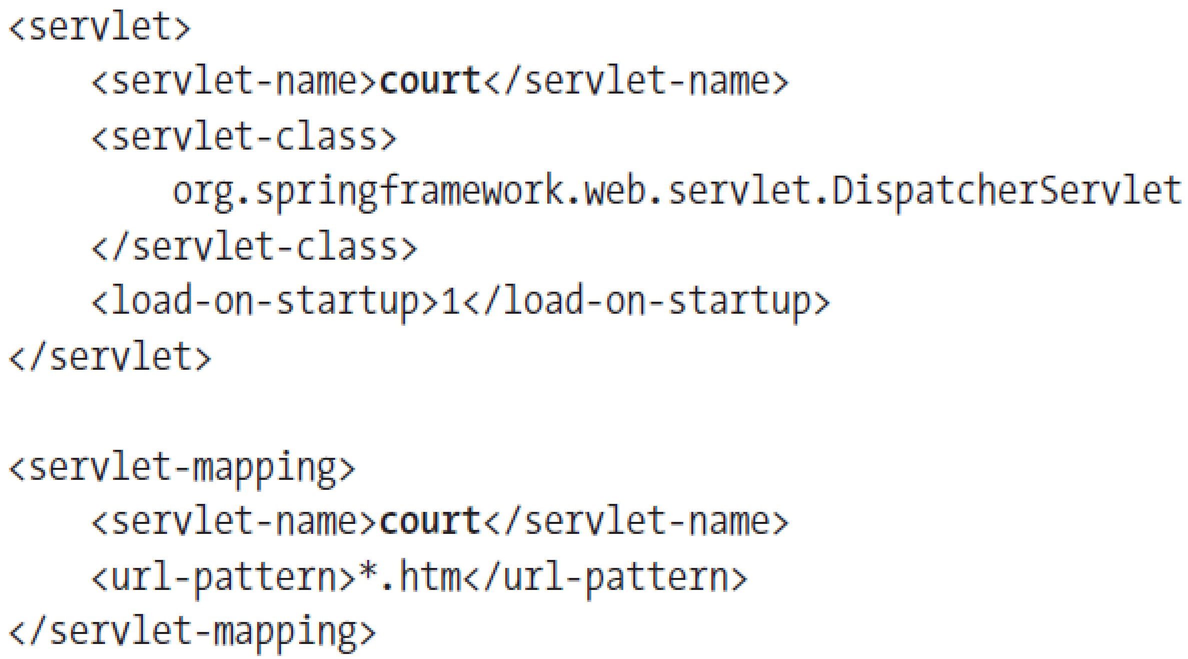
```
public class ReservationServiceImpl implements ReservationService {

    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Soccer");

    private List<Reservation> reservations;

    public ReservationServiceImpl() {
        reservations = new ArrayList<Reservation>();
        reservations.add(new Reservation("Tennis #1",
                new GregorianCalendar(2008, 0, 14).getTime(), 16,
                new Player("Roger", "N/A"), TENNIS));
        reservations.add(new Reservation("Tennis #2",
                new GregorianCalendar(2008, 0, 14).getTime(), 20,
                new Player("James", "N/A"), TENNIS));
    }
```



```
    public List<Reservation> query(String courtName) {
        List<Reservation> result = new ArrayList<Reservation>();
        for (Reservation reservation : reservations) {
            if (reservation.getCourtName().equals(courtName)) {
                result.add(reservation);
            }
        }
        return result;
    }
}
```

Setting Up a Spring MVC Application

Configure the following Servlet in web.xml



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

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

The <servlet-name> given to the servlet is significant. By default, when DispatcherServlet is loaded, it will load the Spring application context from an XML file whose name is based on the name of the servlet. In this case, because the servlet is named court, DispatcherServlet will try to load the ApplicationContext from a file named **court-servlet.xml.**

Breaking up the application context

- It is recommend that you split your application context across application layers in to different xml configuration files.
- To ensure that all of these configuration files are loaded, you'll need to configure a context loader in your web.xml file
- A context loader loads context configuration files in addition to the one that DispatcherServlet loads. The most commonly used context loader is a servlet listener called **ContextLoaderListener** that is configured in web.xml as follows

```
<listener>
    <listener-class> org.springframework.web.context.ContextLoaderListener
  </listener-class>
</listener>
```

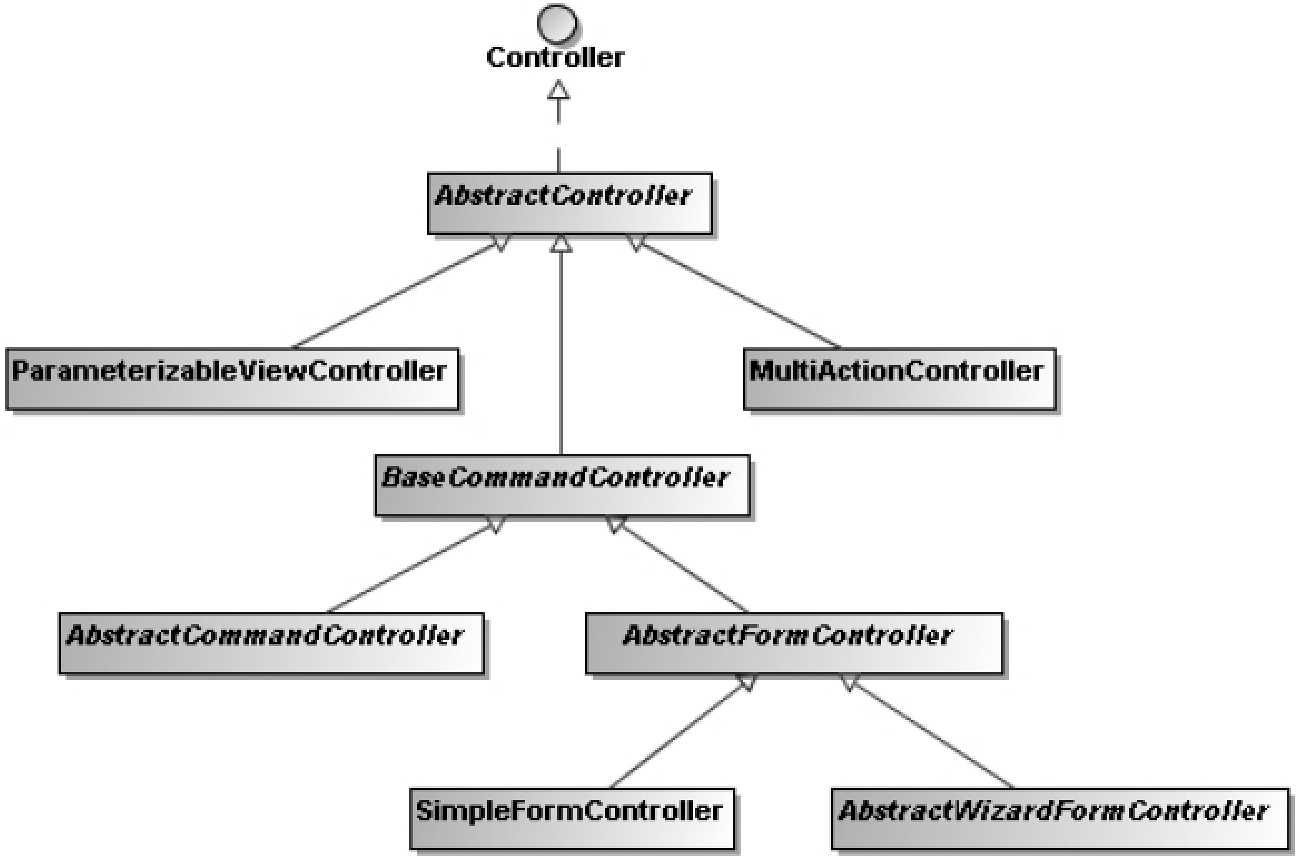
Breaking up the application context

- With ContextLoaderListener configured, you’ll need to tell it the location of the Spring configuration file(s) to load. If not specified otherwise, the context loader will look for a Spring configuration file at /WEB-INF/applicationContext.xml.

- You can specify one or more Spring configuration files for the context loader to load by setting the **contextConfigLocation** parameter in the servlet context:

```
< context- param >
    < param-name>contextConfigLocation</ param-name>
        < param-value>
        /WEB-INF/court-service.xml
        /WEB-INF/court-data.xml
        /WEB-INF/court-security.xml
        </ param-value>
</ context-param>
```

Spring MVC Controllers



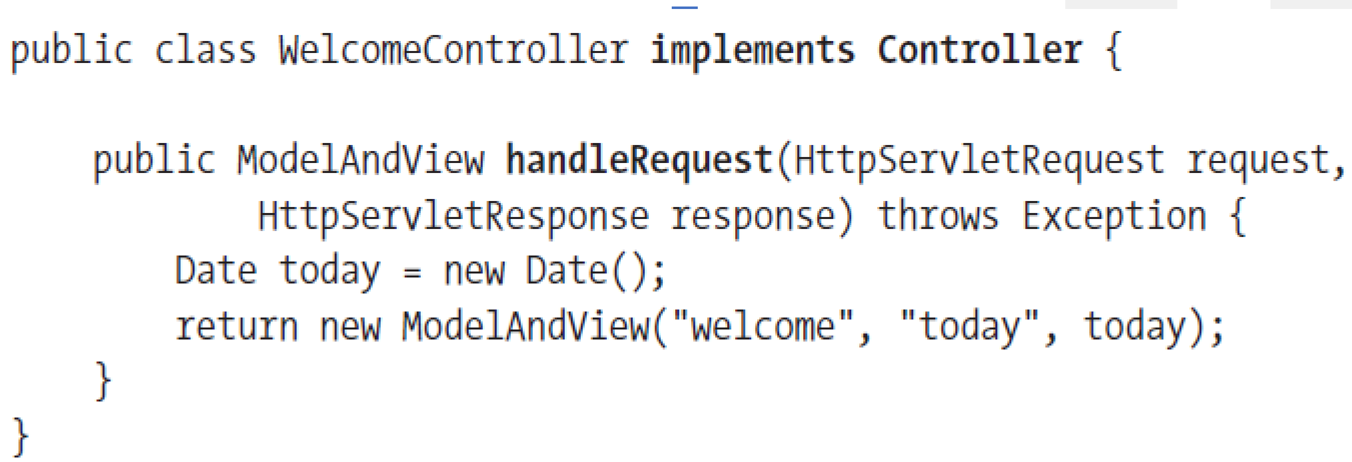
- The Controller interface is the base interface for all the controller classes in Spring MVC.

- You can create your own controller by implementing this interface. In the **handleRequest**() method, you are free to handle a web request just as you do in a servlet.

- Finally, you have to return a ModelAndView object that includes a view name or a view object, and some model attributes.

- For example, you can create a welcome controller for your court reservation system as follows.

Implementing Controller



```
public class WelcomeController implements Controller {

    public ModelAndView handleRequest(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        Date today = new Date();
        return new ModelAndView("welcome", "today", today);
    }
}
```

After you create a controller class, you have to declare its instance in the web application context

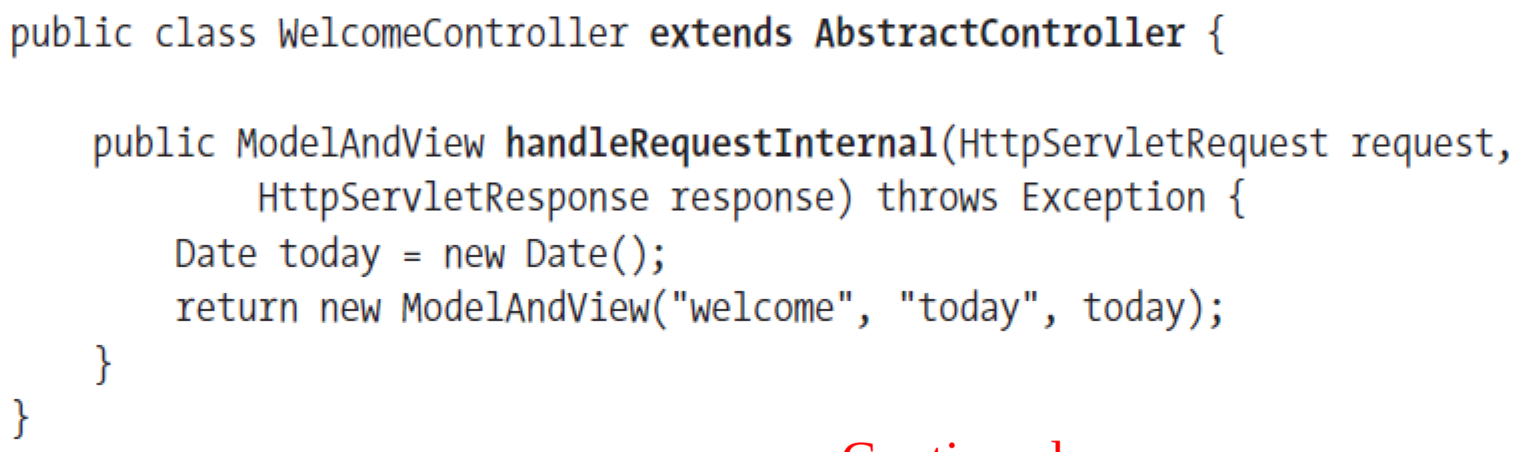
<bean name="/welcome.htm"
class="com.training.court.web.WelcomeController" />

By default, DispatcherServlet uses **BeanNameUrlHandlerMapping** as its default handler mapping

This handler mapping maps requests to handlers according to the URL patterns specified in the bean names of the handlers.

Implementing AbstractController

- If you would like your controller to have some basic controller features like filtering the supported HTTP methods (GET, POST, and HEAD) and generating the cache-control header in HTTP responses, you can have it extend the **AbstractController** class, which implements the
- Controller interface.
- Note that the method you have to override in this controller class is **handleRequestInternal**().



```
public class WelcomeController extends AbstractController {

    public ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        Date today = new Date();
        return new ModelAndView("welcome", "today", today);
    }
}
```

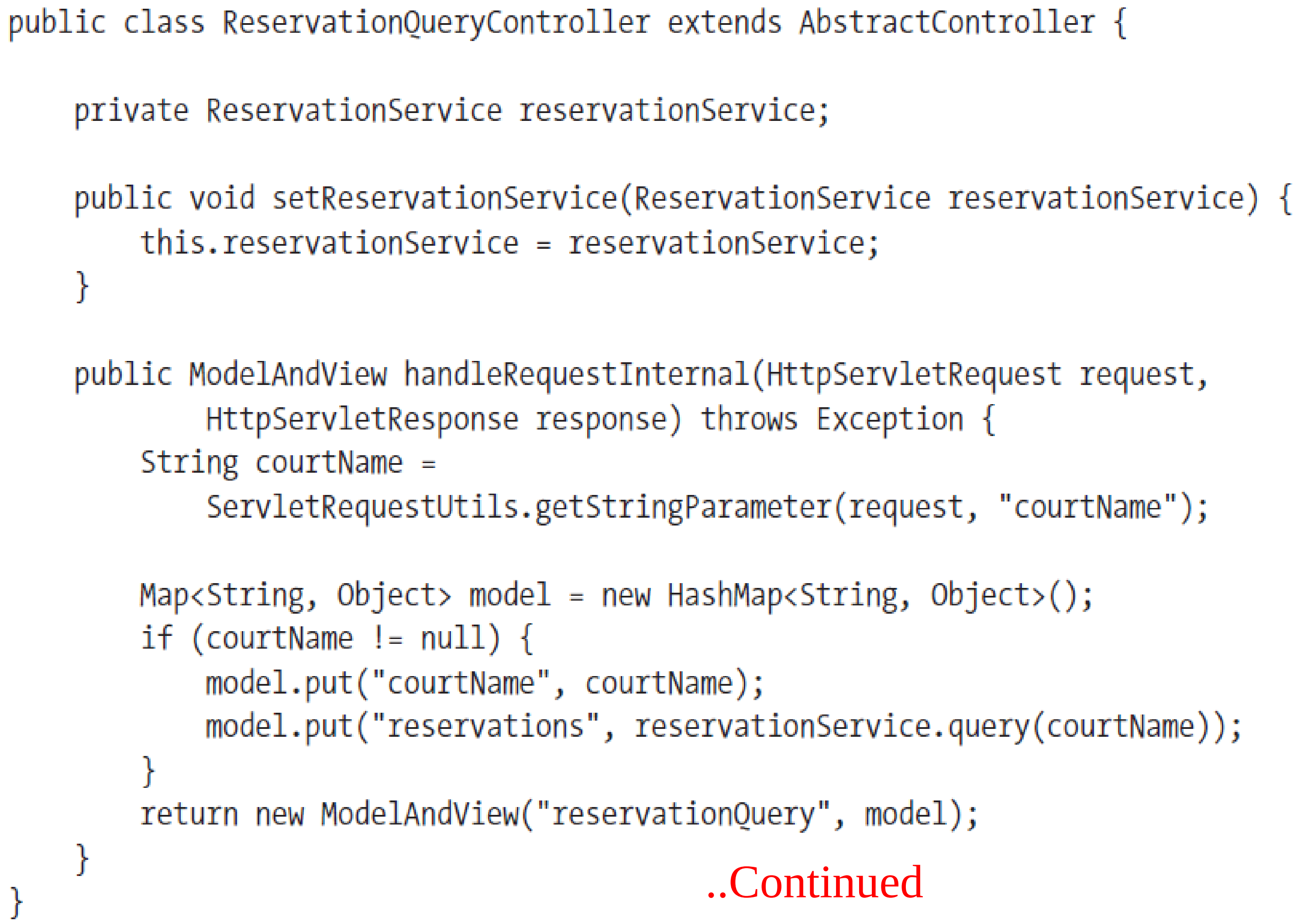
..Continued

- <bean name="/welcome.htm“ class="com.training.court.web.WelcomeController">
- <property name="supportedMethods" value="GET" />
- <property name="cacheSeconds" value="60" />
- </bean>

- If an HTTP request’s method is not in this list, a ServletException will be thrown.

- You can also define the cache seconds for a controller so that it will be set in an HTTP response’s header.

Another implementation of AbstractController



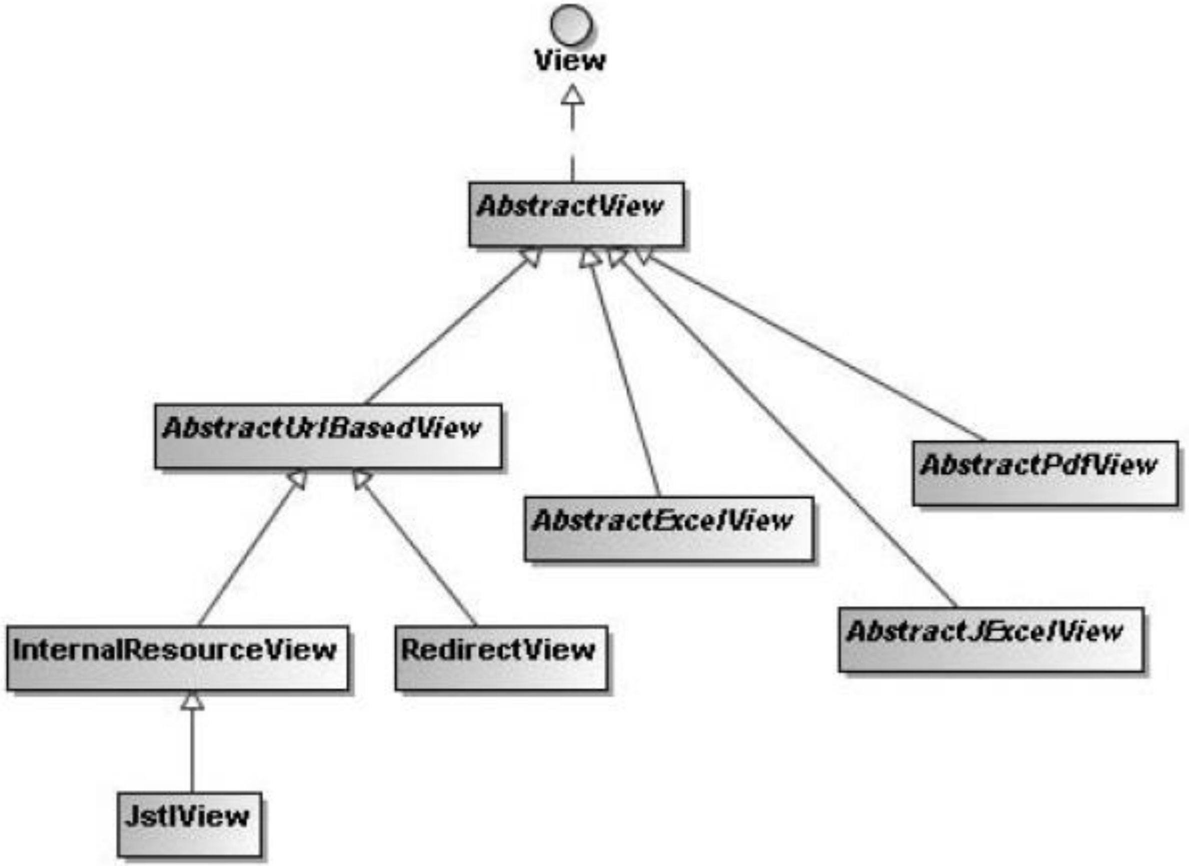
```
public class ReservationQueryController extends AbstractController {

    private ReservationService reservationService;

    public void setReservationService(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    public ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        String courtName =
            ServletRequestUtils.getStringParameter(request, "courtName");

        Map<String, Object> model = new HashMap<String, Object>();
        if (courtName != null) {
            model.put("courtName", courtName);
            model.put("reservations", reservationService.query(courtName));
        }
        return new ModelAndView("reservationQuery", model);
    }
}
```

..Continued

- When you have more than one model attribute to pass to the view, you can store them in a map and pass this map to the constructor of ModelAndView.

- <bean name="/reservationQuery.htm“ class="com.training.court.web.ReservationQueryController">
- <property name="reservationService" ref="reservationService" />
- </bean>

View


View
AbstractView
AbstractUrlBasedView
AbstractExcelView
AbstractPdfView
AbstractJExcelView
InternalResourceView
RedirectView
JstlView

Configuring ViewResolver

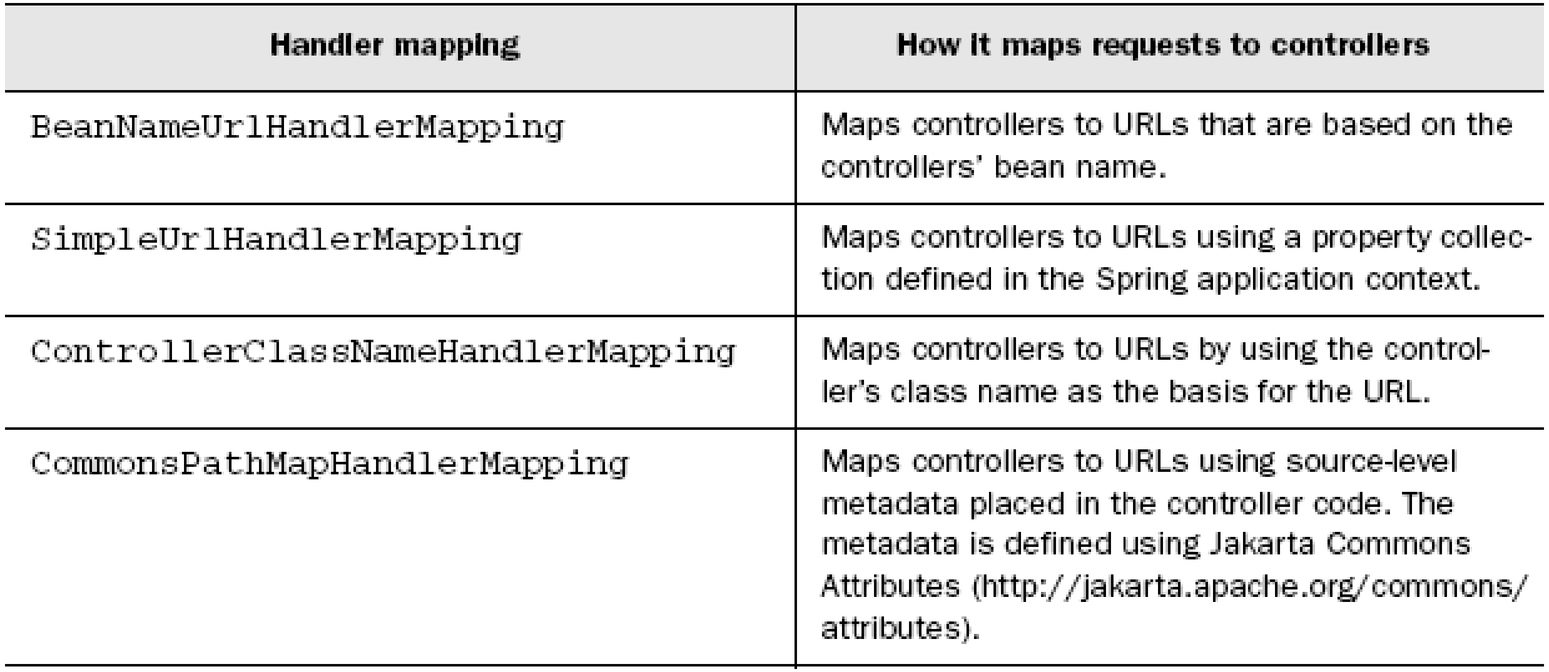
- When DispatcherServlet receives a view name returned from a handler, it will resolve the logical view name into a view object for rendering.

- For example, you can configure an **InternalResourceViewResolver** bean in the web application context to resolve view names
- into JSP files in the /WEB-INF/jsp/ directory:

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

Mapping Requests to Controllers

- Handler mappings are used to map requests to controllers .
- All of Spring MVC's handler mappings implement the **org.springframework.web.servlet.HandlerMapping** interface.
- Spring comes prepackaged with four useful implementations of HandlerMapping, as listed below :



Handler mapping	How it maps requests to controllers
`BeanNameUrlHandlerMapping`	Maps controllers to URLs that are based on the controllers' bean name.
`SimpleUrlHandlerMapping`	Maps controllers to URLs using a property collection defined in the Spring application context.
`ControllerClassNameHandlerMapping`	Maps controllers to URLs by using the controller's class name as the basis for the URL.
`CommonsPathMapHandlerMapping`	Maps controllers to URLs using source-level metadata placed in the controller code. The metadata is defined using Jakarta Commons Attributes (http://jakarta.apache.org/commons/attributes).

Using SimpleUrlHandlerMapping

- *SimpleUrlHandlerMapping* lets you map URL patterns directly to controllers without having to name your beans in a special way.
- Consider the following example :

```
<bean id="simpleUrlMapping" class=
"org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
        <property name="mappings">
        <props>
        <prop key="/home.htm">homePageController</prop>
        <prop key="/displayEmployess.htm“> displayEmployeeController</prop>
        <prop key="/login.htm">loginController</prop>
        <prop key="/register.htm">registerController</prop>
        <prop key="/addEmployee.htm">addEmployeeController</prop>
        </props>
        </property>
</bean>
```

Using ControllerClassNameHandlerMapping

- By configuring **ControllerClassNameHandlerMapping**, you are telling Spring’s DispatcherServlet to map URL patterns to controllers following a simple convention.
- Instead of explicitly mapping each controller to a URL pattern, Spring will automatically map controllers to URL patterns that are based on the controller’s class name.
- Put simply, to produce the URL pattern, the Controller portion of the controller’s class name is removed (if it exists), the remaining text is lowercased, a slash (/) is added to the beginning, and ".htm" is added to the end to produce the URL pattern.
- For example **, com.springmvc.controller.DisplayEmployeeController is mapped to URL displayemployee.htm** .
- Configure a bean as follows :

```
  <bean id="urlMapping"
  class="org.springframework.web.servlet.mvc.ControllerClassNameHandler
     Mapping"/>
```

- Some Examples :
- **WelcomeController → /welcome***
- **ReservationQueryController→ /reservationquery***

- if you want your URL patterns to follow the variable-naming convention in Java (e.g., generating /reservationQuery* instead of reservationquery*), you have to set the caseSensitive property to true

- you can specify a prefix for the generated URL patterns
- in the pathPrefix property. You can also specify the base package in the basePackage property, and then the subpackage relative to it will be included in the mapping.
-
-
-

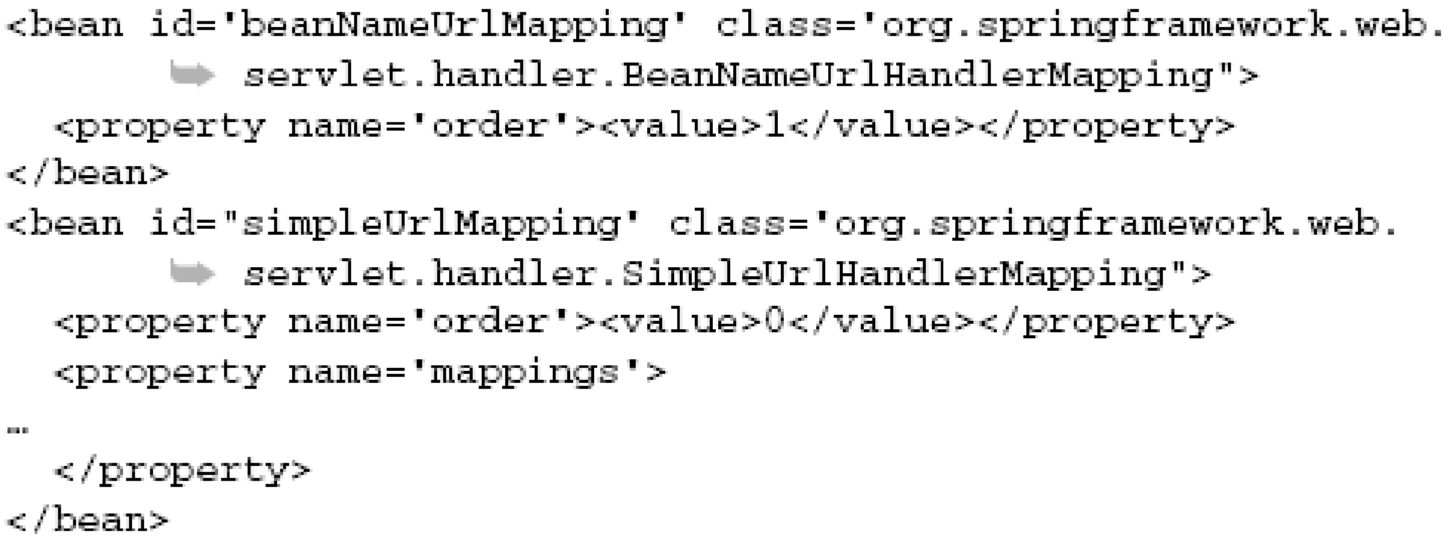
…Continued

- <bean class="org.springframework.web.servlet.mvc.support.
- ControllerClassNameHandlerMapping">
- <property name="caseSensitive" value="true" />
- <property name="pathPrefix" value="/reservation" />
- <property name="basePackage"
  value="com.trainingspringrecipes.court" />
- </bean>

- This ControllerClassNameHandlerMapping definition will generate the following handler
- mappings:
- **WelcomeController →/reservation/web/welcome***
- **ReservationQueryController → /reservation/web/reservationQuery***

Working with multiple handler mappings

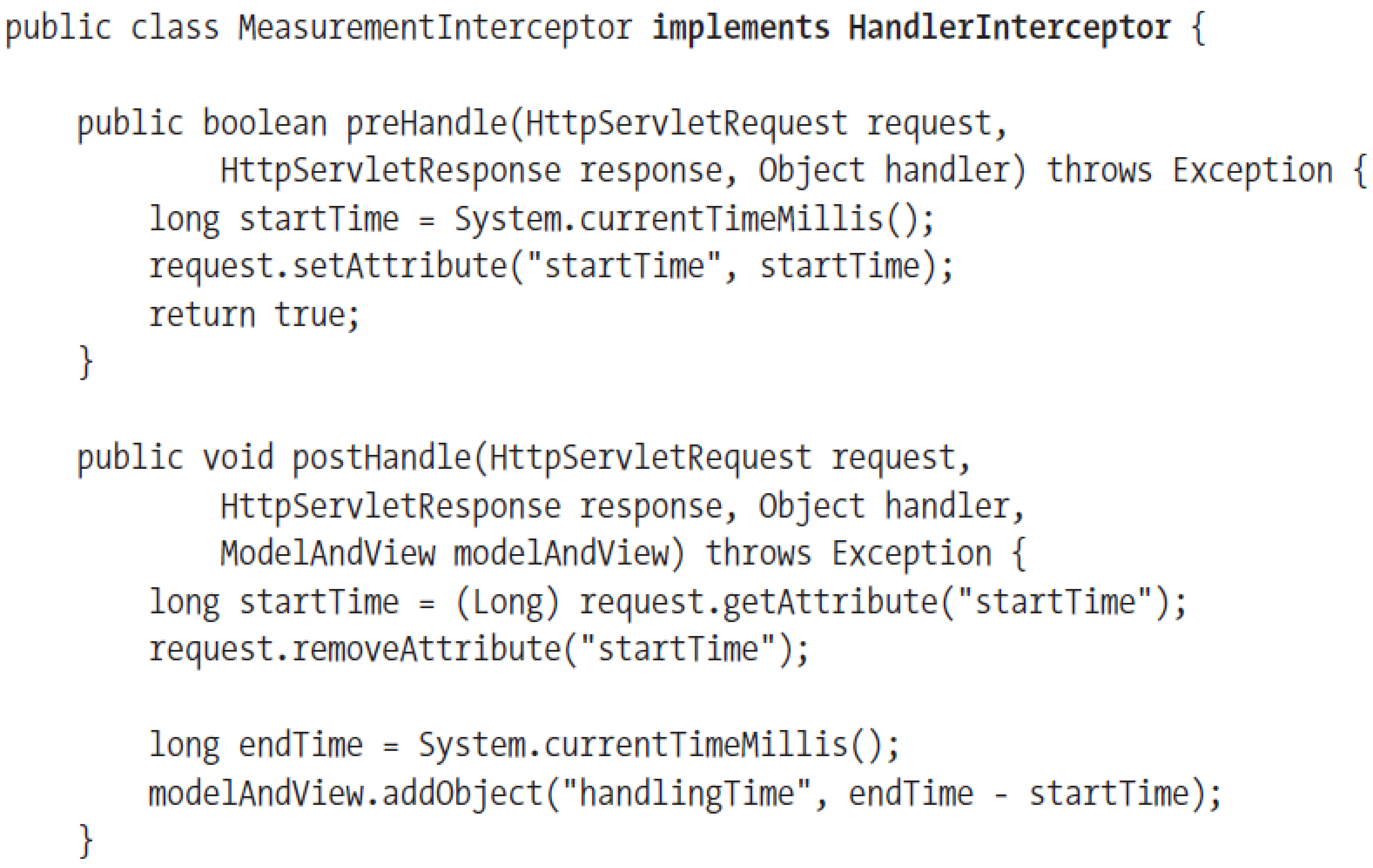
- All of the handler mapping classes implement Spring's **Ordered interface**. This means that you can declare multiple handler mappings in your application and **set their order properties** to indicate which has precedence with relation to the others.
- For example, suppose you want to use both BeanNameUrlHandlerMapping and SimpleUrlHandlerMapping alongside each other in the same application. We need to declare the handler mapping beans as follows:



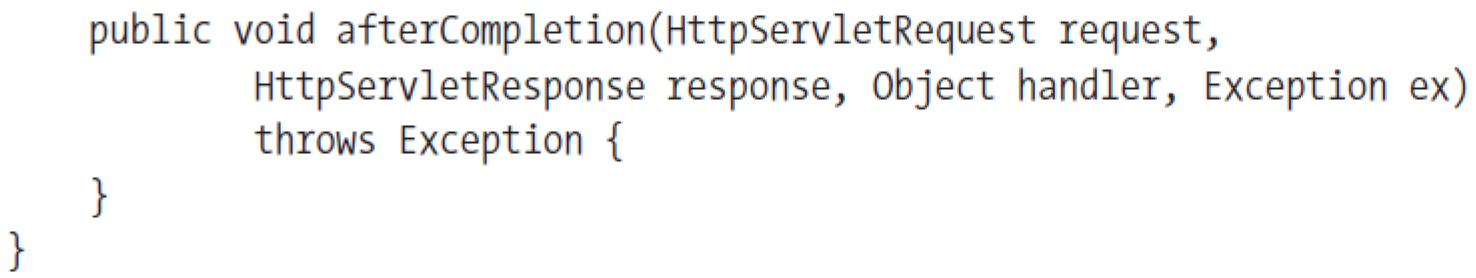
```
<bean id='beanNameUrlMapping' class='org.springframework.web.
        ➥ servlet.handler.BeanNameUrlHandlerMapping">
  <property name='order'><value>1</value></property>
</bean>
<bean id="simpleUrlMapping' class='org.springframework.web.
        ➥ servlet.handler.SimpleUrlHandlerMapping">
  <property name='order'><value>0</value></property>
  <property name='mappings'>
…
  </property>
</bean>
```

Intercepting Requests with Handler Interceptors

- Just like Servlet filters .
- Spring MVC allows you to intercept web requests for pre-handling and post-handling through *handler interceptors*
- A handler interceptor is registered for particular handler mappings, so it will only intercept requests mapped by these handler mappings.
- Each handler interceptor must implement the HandlerInterceptor interface, which contains three callback methods for you to implement: preHandle(), postHandle(), and afterCompletion().
- afterCompletion() method will be called after the completion of all request processing (i.e., after the view has been rendered).



```
public class MeasurementInterceptor implements HandlerInterceptor {

    public boolean preHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler) throws Exception {
        long startTime = System.currentTimeMillis();
        request.setAttribute("startTime", startTime);
        return true;
    }

    public void postHandle(HttpServletRequest request,
            HttpServletResponse response, Object handler,
            ModelAndView modelAndView) throws Exception {
        long startTime = (Long) request.getAttribute("startTime");
        request.removeAttribute("startTime");

        long endTime = System.currentTimeMillis();
        modelAndView.addObject("handlingTime", endTime - startTime);
    }
```



```
        public void afterCompletion(HttpServletRequest request,
                HttpServletResponse response, Object handler, Exception ex)
                throws Exception {
        }
    }
```

preHandle ()method should return true, allowing DispatcherServlet to proceed. with request handling. Otherwise, DispatcherServlet assumes that this method has already handled the request, so DispatcherServlet will return the response to the user directly.

A handler interceptor is registered to a handler mapping bean to intercept web requests mapped by this bean. You can specify multiple interceptors for a handler mapping in the interceptors property, whose type is an array.

Configuring Handler Interceptors

- <bean id="measurementInterceptor"
- class="com.training.court.web.MeasurementInterceptor" />
- <bean   class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
- **<property name="interceptors">**
- **<list>**
- **<ref bean="measurementInterceptor" />**
- **</list>**
- **</property>**
- </bean>
- <bean class="org.springframework.web.servlet.mvc.support.
- ControllerClassNameHandlerMapping">
- **<property name="interceptors">**
- **<list>**
- **<ref bean="measurementInterceptor" />**
- **</list>**
- **</property>**
- </bean>

Resolving User Locales

- In order for your web application to support internationalization, you have to identify each user's preferred locale and display contents according to this locale.

- A user's locale is identified by a locale resolver, which has to
- implement the LocaleResolver interface.

- You can define a locale resolver simply by registering a bean of type LocaleResolver in the web application context. You must set the **bean name of the locale resolver to localeResolver** for DispatcherServlet to auto-detect.

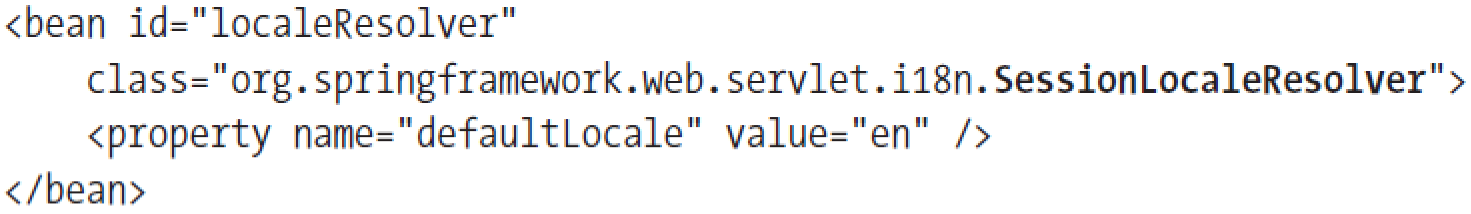
Resolving Locales by an HTTP Request Header

- The default locale resolver used by Spring is **AcceptHeaderLocaleResolver**. It resolves locales by inspecting the accept-language header of an HTTP request.

Resolving Locales by a Session Attribute

- **SessionLocaleResolver** resolves locales by inspecting a predefined attribute in a user’s session. Attribute name is **SessionLocaleResolver. LOCALE_SESSION_ATTRIBUTE_NAME**

- If the session attribute doesn’t exist, this locale resolver will determine the default locale from the accept-language HTTP header.



```
<bean id="localeResolver"
    class="org.springframework.web.servlet.i18n.SessionLocaleResolver">
    <property name="defaultLocale" value="en" />
</bean>
```

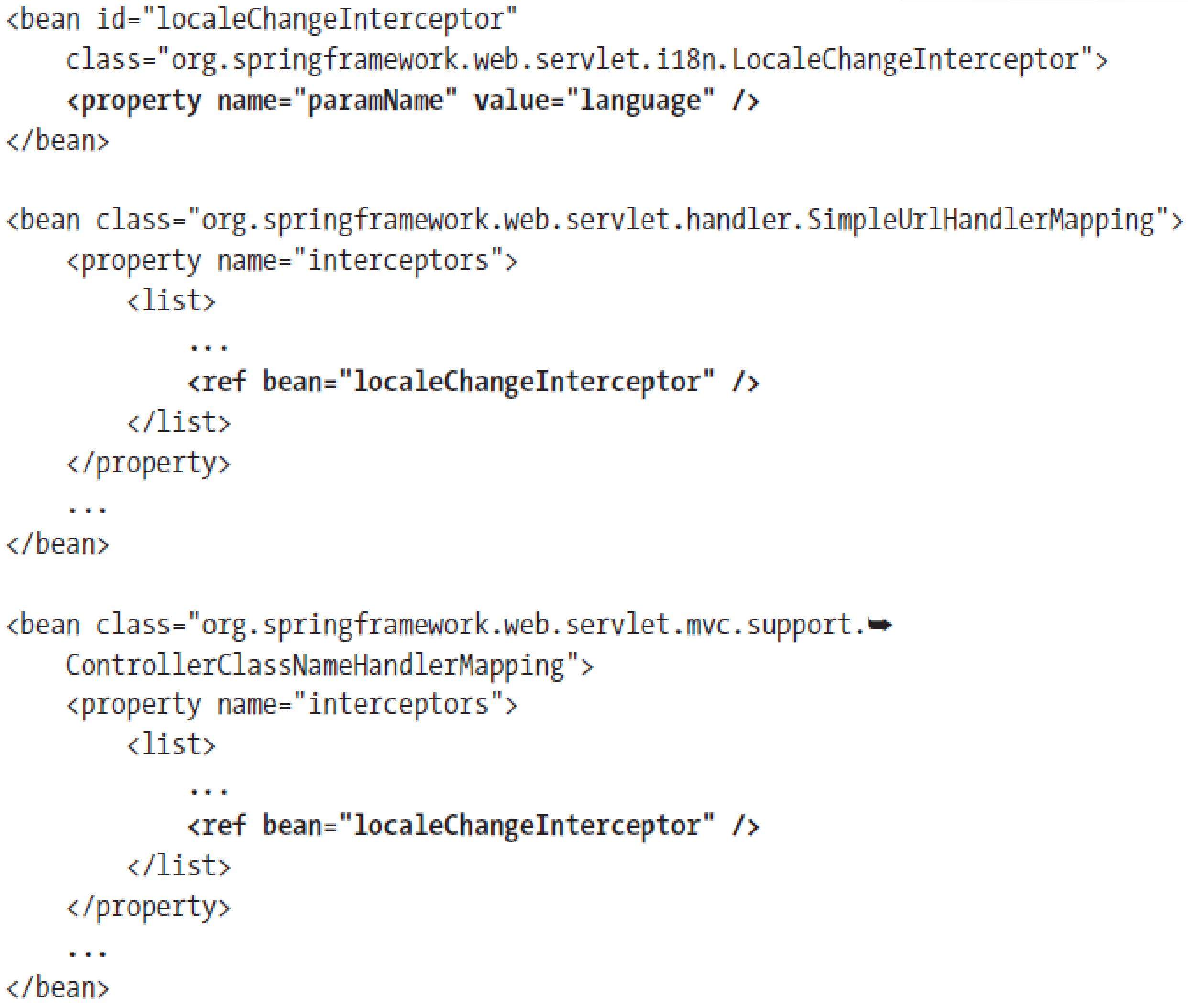
Resolving Locales by a Cookie

- You can also use **CookieLocaleResolver** to resolve locales by inspecting a cookie in a user's browser. If the cookie doesn't exist, this locale resolver will determine the default locale from the accept-language HTTP header.

- <bean id="localeResolver"
- class="org.springframework.web.servlet.i18n.**CookieLocaleResolver" />**

Changing a User’s Locale

- In addition to changing a user’s locale by calling LocaleResolver.setLocale() explicitly, you can also apply **LocaleChangeInterceptor** to your handler mappings.
- This interceptor will detect if a special parameter is present in the current HTTP request. The parameter name can be customized with the paramName property of this interceptor.
- If such a parameter is present in the current request, this interceptor will change the user’s locale according to the parameter value.
- See next slide for configuration ……



```
<bean id="localeChangeInterceptor"
    class="org.springframework.web.servlet.i18n.LocaleChangeInterceptor">
    <property name="paramName" value="language" />
</bean>

<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
    <property name="interceptors">
        <list>
            ...
            <ref bean="localeChangeInterceptor" />
        </list>
    </property>
    ...
</bean>

<bean class="org.springframework.web.servlet.mvc.support.➥
    ControllerClassNameHandlerMapping">
    <property name="interceptors">
        <list>
            ...
            <ref bean="localeChangeInterceptor" />
        </list>
    </property>
    ...
</bean>
```

Internationalization

- You can define a message source simply by registering a bean of type MessageSource in the web application context. You must set the bean name of the message source to messageSource for DispatcherServlet to auto-detect.
- You can register only one message source per DispatcherServlet.

- <bean id="messageSource"
- class="org.springframework.context.support.ResourceBundleMessageSource">
- <property name="basename" value="messages" />
- </bean>

- Then you create two resource bundles, **messages.properties** and **messages_de.**
- **properties**, to store messages for the default and German locales. These resource bundles should be put in the root of the classpath.
- welcome.title=Welcome
- welcome.message=Welcome to Court Reservation System
- welcome.title=Willkommen
- welcome.message=Willkommen zum Spielplatz-Reservierungssystem

- Now, in a JSP file , you can use the <spring:message> tag to resolve a message given the code. This tag will automatically resolve the message according to a user’s current locale.

- **<spring:message code="welcome.title“ text="Welcome" />**

- **<spring:message code="welcome.message“ text="Welcome to Court Reservation System" />**

Resolving Views by Names

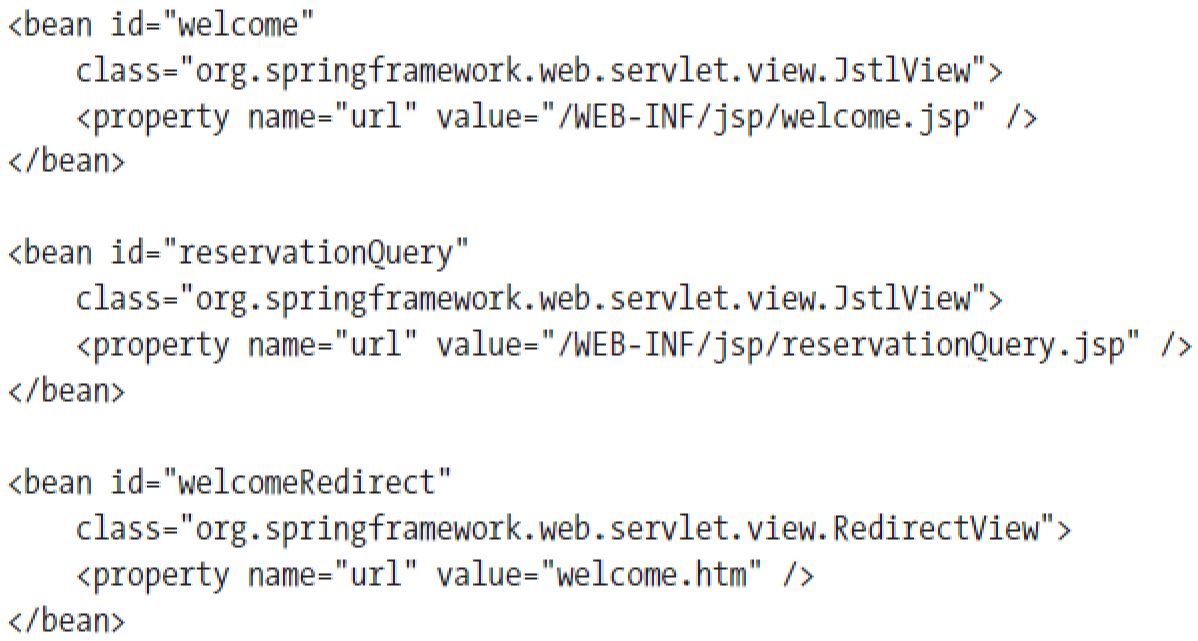
- Views are resolved by one or more view resolver beans declared
- in the web application context. These beans have to implement the **ViewResolver** interface for DispatcherServlet to auto-detect them .

Resolving Views from an XML Configuration File

- You can declare the view beans in the same configuration file as the web application context, but it would be better to isolate them in a separate configuration file. By default, **XmlViewResolver** loads view beans from /WEB-INF/views.xml, but this location can be overridden through the location property.

```
•  <bean class="org.springframework.web.servlet.view.XmlViewResolver">
•          <property name="location">
•                  <value>/WEB-INF/court-views.xml</value>
•          </property>
•  </bean>
```

In the court-views.xml configuration file, you can declare each view as a normal Spring bean by setting the class name and properties. In this way, you can declare any types of views (e.g., RedirectView and even custom view types).



```
<bean id="welcome"
    class="org.springframework.web.servlet.view.JstlView">
    <property name="url" value="/WEB-INF/jsp/welcome.jsp" />
</bean>

<bean id="reservationQuery"
    class="org.springframework.web.servlet.view.JstlView">
    <property name="url" value="/WEB-INF/jsp/reservationQuery.jsp" />
</bean>

<bean id="welcomeRedirect"
    class="org.springframework.web.servlet.view.RedirectView">
    <property name="url" value="welcome.htm" />
</bean>
```

Resolving Views from a Resource Bundle

- <bean class="org.springframework.web.servlet.view.**ResourceBundleViewResolver**">
- <property name="basename" value="views" />
- </bean>

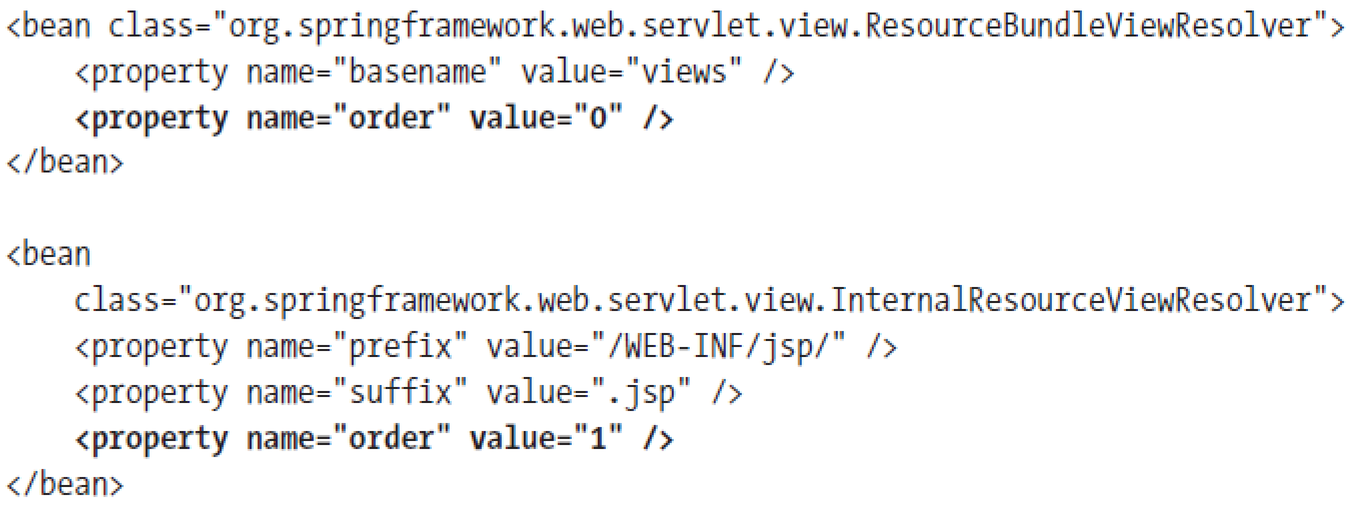
- ResourceBundleViewResolver can also take advantage of the resource bundle capability to load view beans from different resource bundles for different locales.

- welcome.(class)=org.springframework.web.servlet.view.JstlView
- welcome.url=/WEB-INF/jsp/welcome.jsp

- reservationQuery.(class)=org.springframework.web.servlet.view.JstlView
- reservationQuery.url=/WEB-INF/jsp/reservationQuery.jsp

- welcomeRedirect.(class)=org.springframework.web.servlet.view.RedirectView
- welcomeRedirect.url=welcome.htm

Resolving Views with Multiple Resolvers



```
<bean class="org.springframework.web.servlet.view.ResourceBundleViewResolver">
    <property name="basename" value="views" />
    <property name="order" value="0" />
</bean>

<bean
    class="org.springframework.web.servlet.view.InternalResourceViewResolver">
    <property name="prefix" value="/WEB-INF/jsp/" />
    <property name="suffix" value=".jsp" />
    <property name="order" value="1" />
</bean>
```

The Redirect Prefix

- If you have InternalResourceViewResolver configured in your web application context, it can resolve redirect views by using the redirect prefix in the view name. Then the rest of the view name will be treated as the redirect URL.

- For example, the view name redirect:welcome.htm will trigger a redirect to the relative URL welcome.htm.

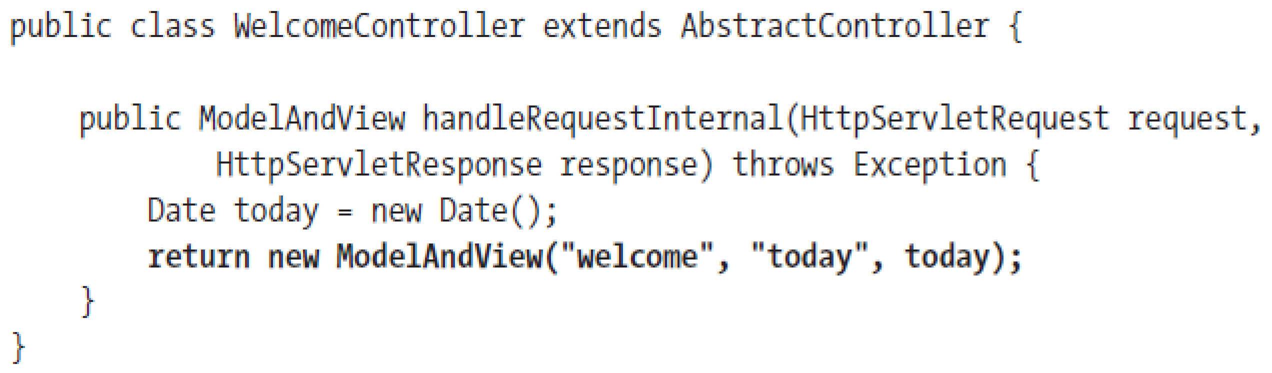
Mapping Exceptions to Views

- In a Spring MVC application, you can register one or more exception resolver beans in the web application context to resolve uncaught exceptions.
- These beans have to implement the **HandlerExceptionResolver** interface for DispatcherServlet to auto-detect them

```
<bean class="org.springframework.web.servlet.handler.SimpleMappingExceptionResolver">
      <property name="exceptionMappings">
           <props>
                <prop key="com.training.court.service.
                                                         ReservationNotAvailableException">
                  reservationNotAvailable
                 </prop>
                  <prop key="java.lang.Exception">error</prop>
           </props>
        </property>
</bean>
```

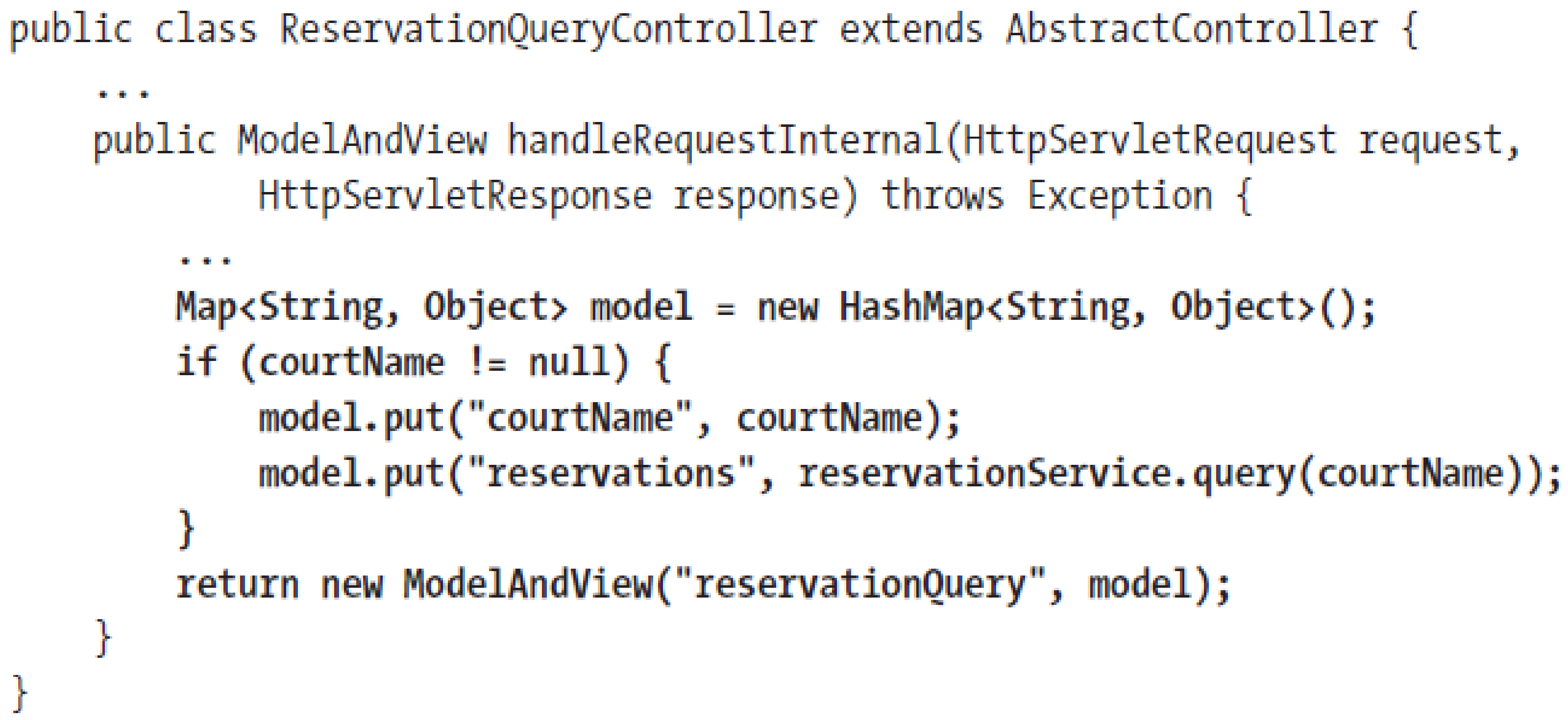
Constructing ModelAndView Objects

- When you have a single model attribute to return, you can simply construct a ModelAndView object by specifying the attribute in the constructor



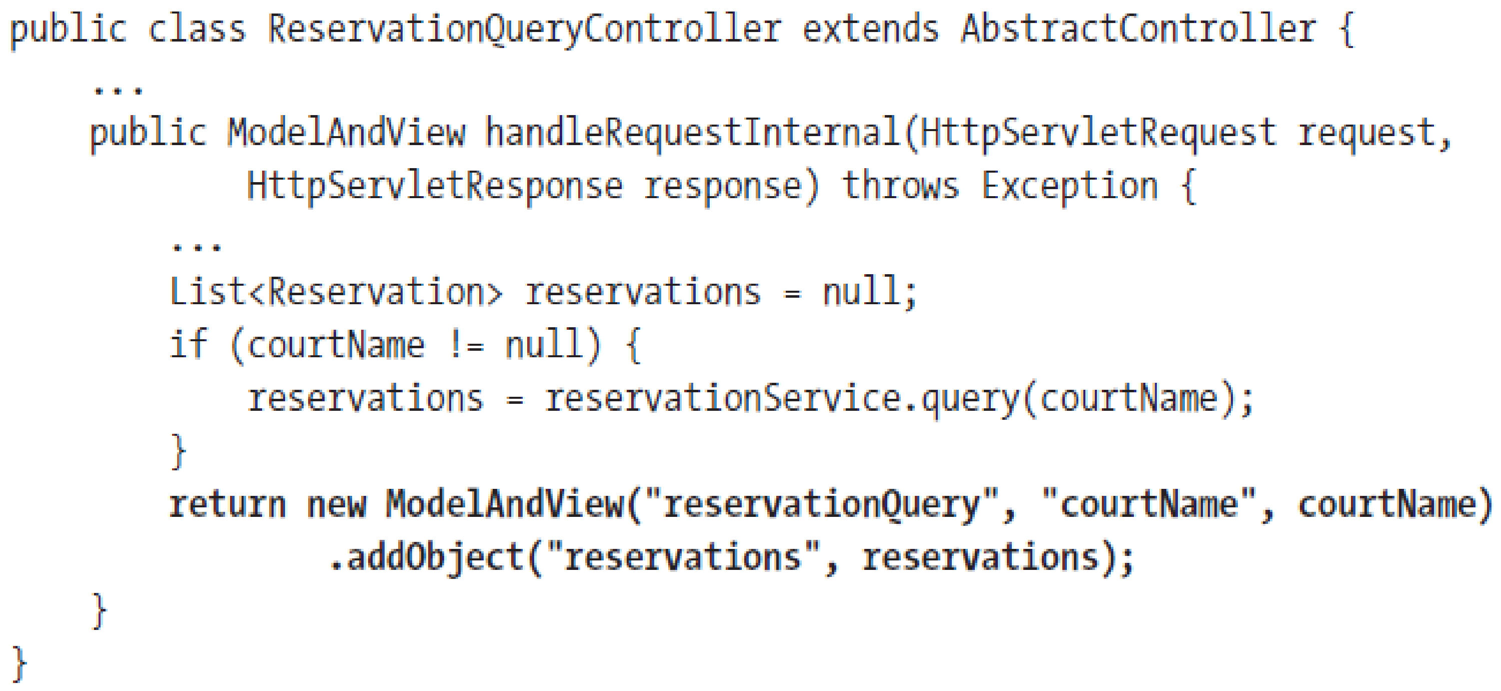
```
public class WelcomeController extends AbstractController {

    public ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        Date today = new Date();
        return new ModelAndView("welcome", "today", today);
    }
}
```

- If there’s more than one attribute to return, you can construct a ModelAndView object by passing in a map:



```
public class ReservationQueryController extends AbstractController {
    ...
    public ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        ...
        Map<String, Object> model = new HashMap<String, Object>();
        if (courtName != null) {
            model.put("courtName", courtName);
            model.put("reservations", reservationService.query(courtName));
        }
        return new ModelAndView("reservationQuery", model);
    }
}
```

- After you construct a ModelAndView object, you can still add model attributes to it using the addObject() method. This method returns the ModelAndView object itself so that you can construct a ModelAndView object in one statement.



```
public class ReservationQueryController extends AbstractController {
    ...
    public ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        ...
        List<Reservation> reservations = null;
        if (courtName != null) {
            reservations = reservationService.query(courtName);
        }
        return new ModelAndView("reservationQuery", "courtName", courtName)
                .addObject("reservations", reservations);
    }
}
```

Creating a Controller with a Parameterized View

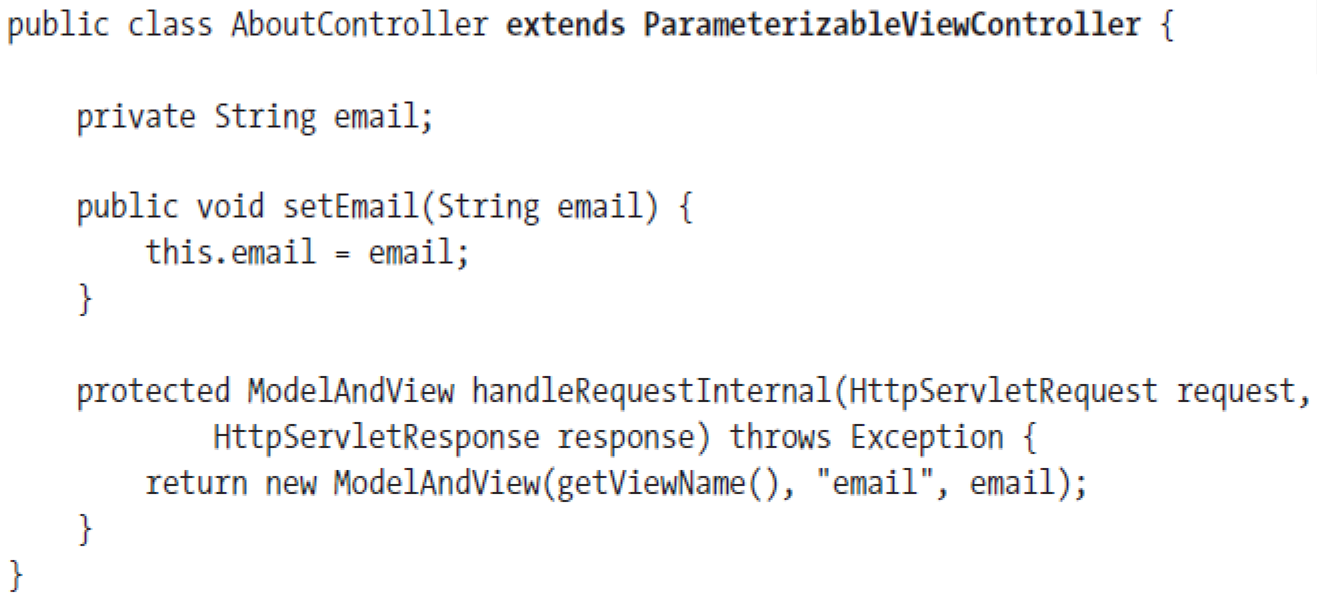
- When creating a controller, you don't want to hard-code the view name in the controller class. Instead, you would like the view name to be parameterized so that you can specify it in the bean configuration file.
- You can use this controller class directly for a controller that only renders a view to users without any processing logic, or you can extend this controller class to inherit the viewName property.

- **ParameterizableViewController** is a subclass of AbstractController that defines a **viewName** property .
- This is same as **ForwardAction** in Struts.

- For example, suppose you have a very simple controller whose purpose is only to render the about view. You can declare a controller of type ParameterizableViewController and specify the viewName property as about

```
• <bean id="aboutController"
• class="org.springframework.web.servlet.mvc.ParameterizableViewController">
•           <property name="viewName" value="about" />
• </bean>
```

- If you would like to add some processing logic to your controller while preserving the parameterized view, your controller class can extend ParameterizableViewController. The following AboutController accepts an email property and includes it in the model:

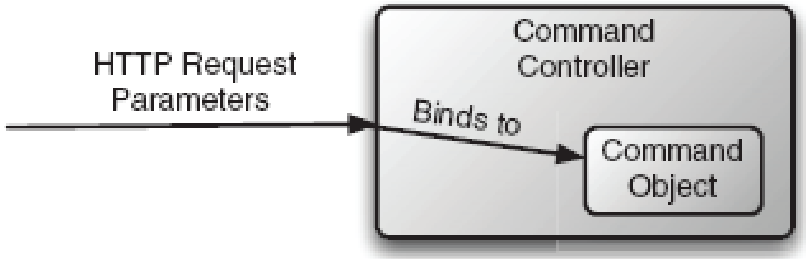


```
public class AboutController extends ParameterizableViewController {

    private String email;

    public void setEmail(String email) {
        this.email = email;
    }

    protected ModelAndView handleRequestInternal(HttpServletRequest request,
            HttpServletResponse response) throws Exception {
        return new ModelAndView(getViewName(), "email", email);
    }
}
```

```
<bean id="aboutController“ class="com.training.court.web.AboutController">
        <property name="viewName" value="about" />
       <property name="email" value="reservation@court.com" />
</bean>
```

Command Controllers

- We can use **AbstractController** when we don't need to bind request parameters to business objects
- Even when we need to bind request parameters to business objects, we can use **AbstractController** . But we have to write all the logic .
- In the event that your controller will need to perform work based on parameters, your controller class should extend a command controller class such as **AbstractCommandController**.



Command controllers relieve you from the hassle of dealing with request parameters directly. They bind the request parameters to a command object that you'll work with instead.

Example

```
public class AddEmployeeController extends AbstractCommandController{
        EmployeeDao employeeDao;
        public EmployeeDao getEmployeeDao() {
                return employeeDao;
        }
        public void setEmployeeDao(EmployeeDao employeeDao) {
                this.employeeDao = employeeDao;
        }
        protected ModelAndView handle(HttpServletRequest request, HttpServletResponse response, Object command, BindException errors) throws Exception {
                Employee employee=(Employee)command;
                employeeDao.addEmployee(employee);
                return new ModelAndView("displaySuccess","employee",employee);
        }


}
```

```
<bean name="/addEmployee.htm"  class="com.mvcapp.AddEmployeeController">
            <property name="commandClass" value="com.mvcapp.Employee" />
            <property name="commandName" value="employee" />
            <property name="employeeDao" ref="employeeDao" />
</bean>
```

SimpleFormController

- The SimpleFormController class provided by Spring MVC defines the basic form-handling flow.

- It supports the concept of a command object and can bind form field values to a command object’s properties of the same name.

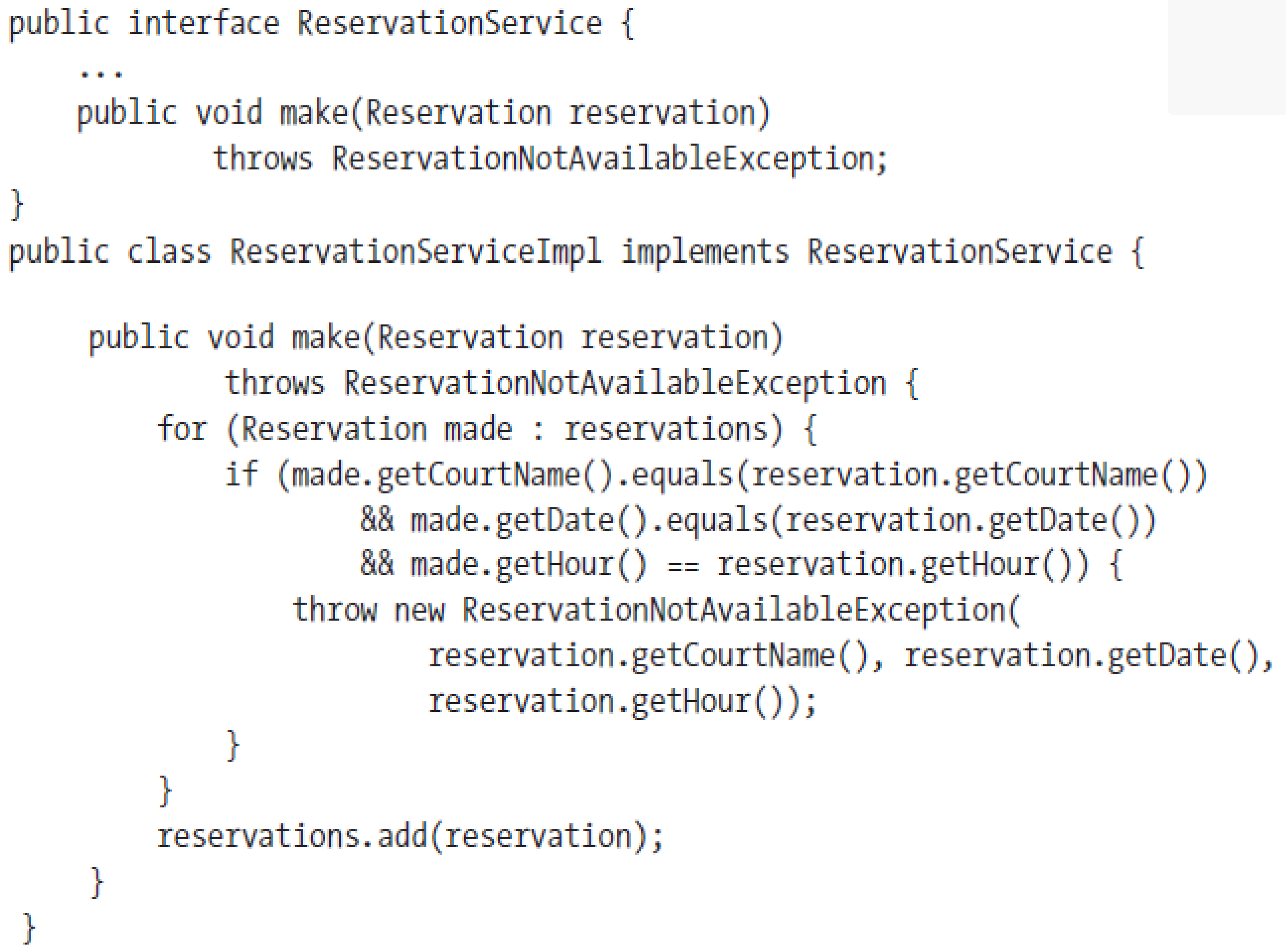
- When SimpleFormController is asked to show a form by an HTTP GET request, it will render the form view to the user.

- When the form is submitted by an HTTP POST request,
- SimpleFormController will handle the form submission by binding the form field values to a command object and invoking the **onSubmit**() method

- If the form is handled successfully, it will render the success view to the user. Otherwise, it will render the form view again with the
- errors.

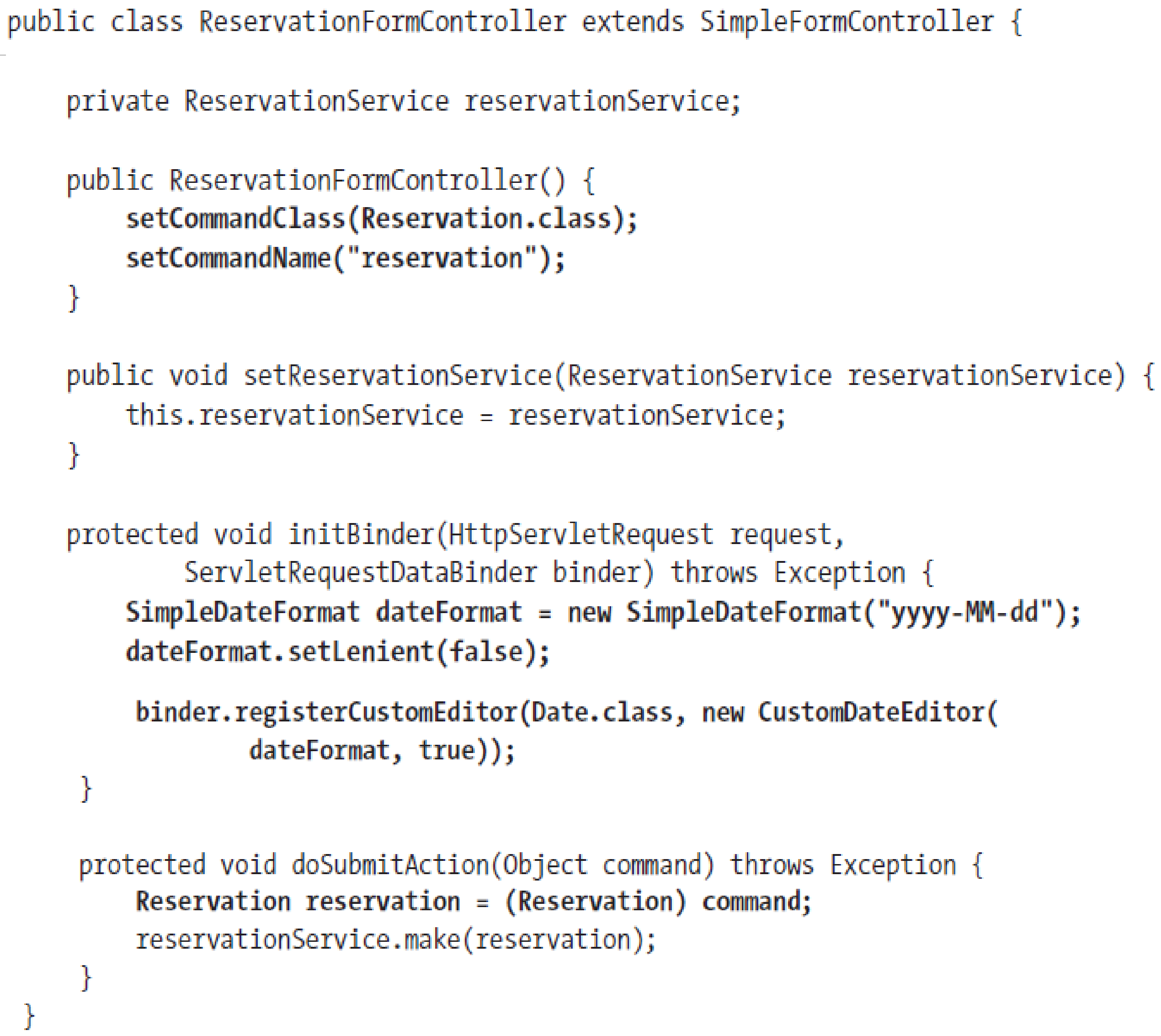
Example

- Suppose you would like to allow a user to make a court reservation by filling out a form.



```
public interface ReservationService {
    ...
    public void make(Reservation reservation)
            throws ReservationNotAvailableException;
}
public class ReservationServiceImpl implements ReservationService {

     public void make(Reservation reservation)
             throws ReservationNotAvailableException {
         for (Reservation made : reservations) {
             if (made.getCourtName().equals(reservation.getCourtName())
                     && made.getDate().equals(reservation.getDate())
                     && made.getHour() == reservation.getHour()) {
                 throw new ReservationNotAvailableException(
                         reservation.getCourtName(), reservation.getDate(),
                         reservation.getHour());
             }
         }
         reservations.add(reservation);
     }
 }
```

Creating a Form Controller



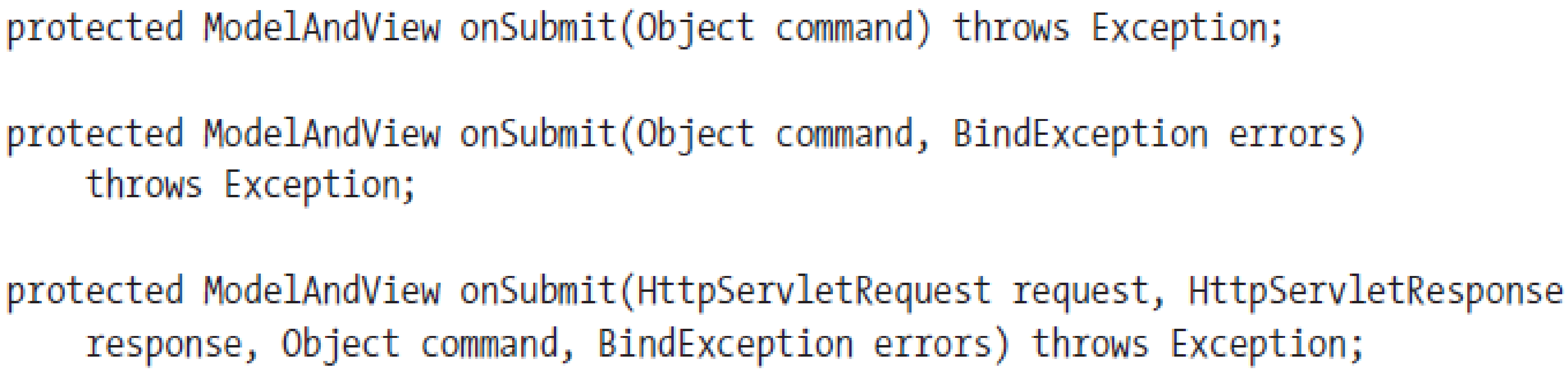
```
public class ReservationFormController extends SimpleFormController {

    private ReservationService reservationService;

    public ReservationFormController() {
        setCommandClass(Reservation.class);
        setCommandName("reservation");
    }

    public void setReservationService(ReservationService reservationService) {
        this.reservationService = reservationService;
    }

    protected void initBinder(HttpServletRequest request,
            ServletRequestDataBinder binder) throws Exception {
        SimpleDateFormat dateFormat = new SimpleDateFormat("yyyy-MM-dd");
        dateFormat.setLenient(false);

         binder.registerCustomEditor(Date.class, new CustomDateEditor(
                dateFormat, true));
     }

     protected void doSubmitAction(Object command) throws Exception {
         Reservation reservation = (Reservation) command;
         reservationService.make(reservation);
     }
 }
```

initBinder

- To bind form field values to a command object, a form controller may have to perform type conversions for the field values, as they are all submitted as strings.
- The type conversions are actually performed by property editors registered in this controller. Spring has preregistered several property editors to convert well-known data types such as numbers and
- Booleans. You have to register your custom editors for other data types such as java.util.Date.

- Custom property editors are registered to the **ServletRequestDataBinder** argument in the **initBinder**() method.

- If there’s anything wrong when binding the form field values, SimpleFormController will automatically render the form view again with the errors.
- Otherwise, it will call the onSubmit() method to handle the form submission
- There are three variants of the onSubmit() method that you can override. You should override the simplest of them to get access to the method arguments that meet your needs.



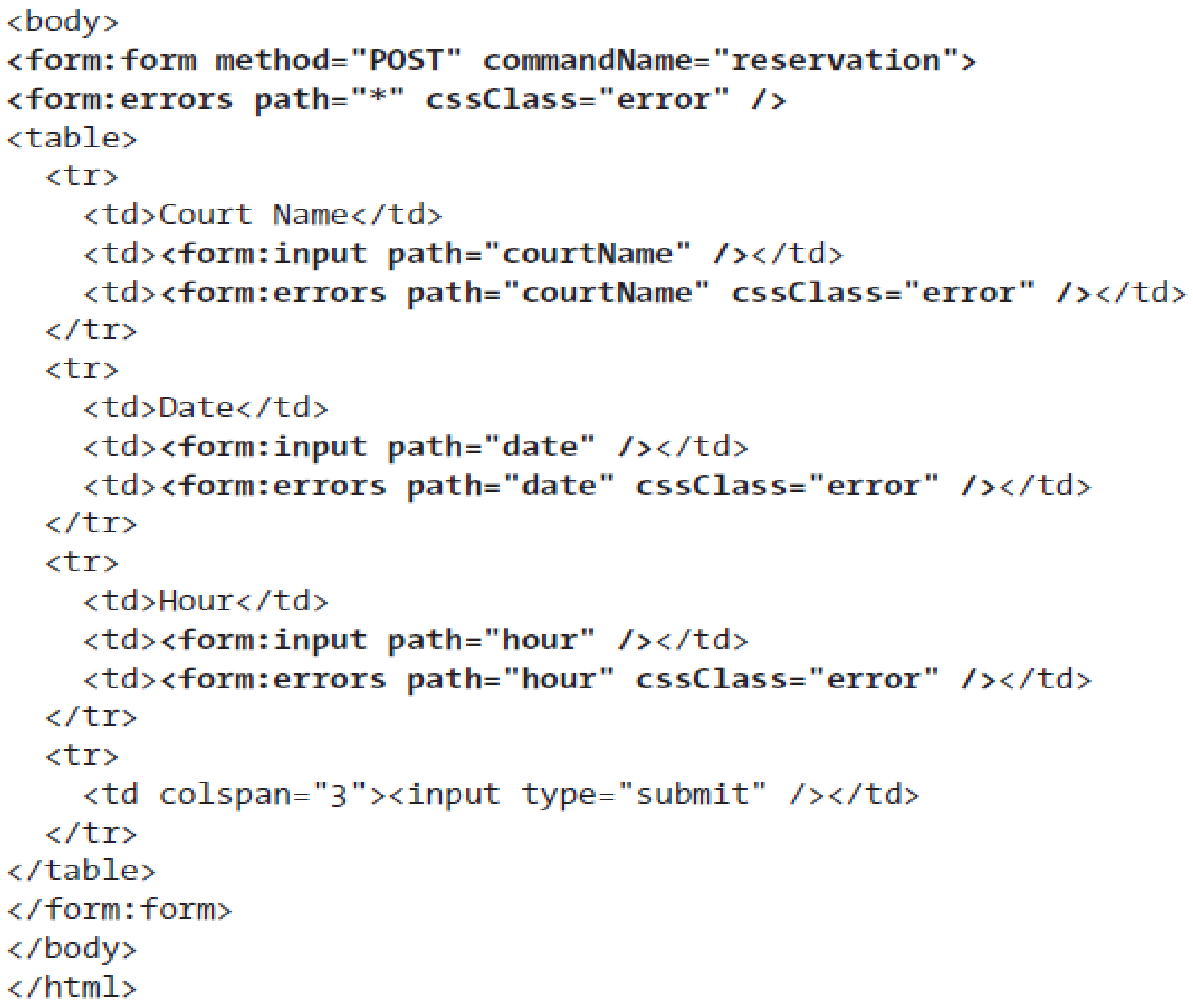
```
protected ModelAndView onSubmit(Object command) throws Exception;

protected ModelAndView onSubmit(Object command, BindException errors)
    throws Exception;

protected ModelAndView onSubmit(HttpServletRequest request, HttpServletResponse
    response, Object command, BindException errors) throws Exception;
```

- When overriding an onSubmit() method, you have to return a ModelAndView object.

- If you only need to perform an action on the command object and return the success view when this action completes, you can override the **doSubmitAction**() method instead, whose return type is void, and which will render the success view by default.

- <bean id="reservationFormController“ class="com.training.court.web.ReservationFormController">
- <property name="reservationService" ref="reservationService" />
- <property name="formView" value="reservationForm" />
- <property name="successView" value="reservationSuccess" />
- </bean>

reservationForm.jsp



```
<body>
<form:form method="POST" commandName="reservation">
<form:errors path="*" cssClass="error" />
<table>
  <tr>
    <td>Court Name</td>
    <td><form:input path="courtName" /></td>
    <td><form:errors path="courtName" cssClass="error" /></td>
  </tr>
  <tr>
    <td>Date</td>
    <td><form:input path="date" /></td>
    <td><form:errors path="date" cssClass="error" /></td>
  </tr>
  <tr>
    <td>Hour</td>
    <td><form:input path="hour" /></td>
    <td><form:errors path="hour" cssClass="error" /></td>
  </tr>
  <tr>
    <td colspan="3"><input type="submit" /></td>
  </tr>
</table>
</form:form>
</body>
</html>
```

- The <form:input> and <form:select> tags can bind to a property path of the command object by specifying the path attribute.

- **They will show the user the original value of the field, which is either the bound property value or the value rejected due to binding error.**

- They must be used inside the <form:form> tag, which defines a form and binds to the command object by its name.

- If you don’t specify the command object’s name, then the **default name command** will be used.

- In the form field–binding process, errors may occur due to invalid values.
- For example, if the date is not in a valid format as specified in CustomDateEditor, the preceding form controller won't be able to convert these fields.

- This controller will generate a list of selective error codes for each error. For an invalid value input in the date field, the following error codes will be generated:

- typeMismatch.command.date
- typeMismatch.date
- typeMismatch.java.util.Date
- typeMismatch

- If you have ResourceBundleMessageSource defined, you can include the following error messages in your resource bundle for the appropriate locale

- **typeMismatch.date=Invalid date format**
- **typeMismatch.hour=Invalid hour format**

Applying the Post/Redirect/Get Design Pattern

- When you refresh the web page in the form success view, the form you just submitted will be resubmitted again. This problem is known as *duplicate form submission.*

- *To avoid* this problem, you can apply the *post/redirect/get design pattern, which recommends redirecting* to another URL after a form submission is handled successfully, instead of returning an HTML page directly.

- Consider following controller declaration :

```
<bean id="reservationFormController"
class="com.training.court.web.ReservationFormController">
          <property name="reservationService" ref="reservationService" />
          <property name="formView" value="reservationForm" />
         <property name="successView" value="reservationSuccessRedirect" />
</bean>
```

- As you have ResourceBundleViewResolver configured in your web application context, you can define the following redirect view in views.properties in the classpath root:

- **reservationSuccessRedirect.(class)=org.springframework.web.servlet.view.RedirectView**
- **reservationSuccessRedirect.url=reservationSuccess.htm**

- **U can configure as follows :**

```
<bean class="org.springframework.web.servlet.handler.SimpleUrlHandlerMapping">
..........
      <property name="mappings">
           <props>
                     ...
                <prop key="/reservationSuccess.htm">reservationSuccessController</prop>
           </props>
      </property>
</bean>
```

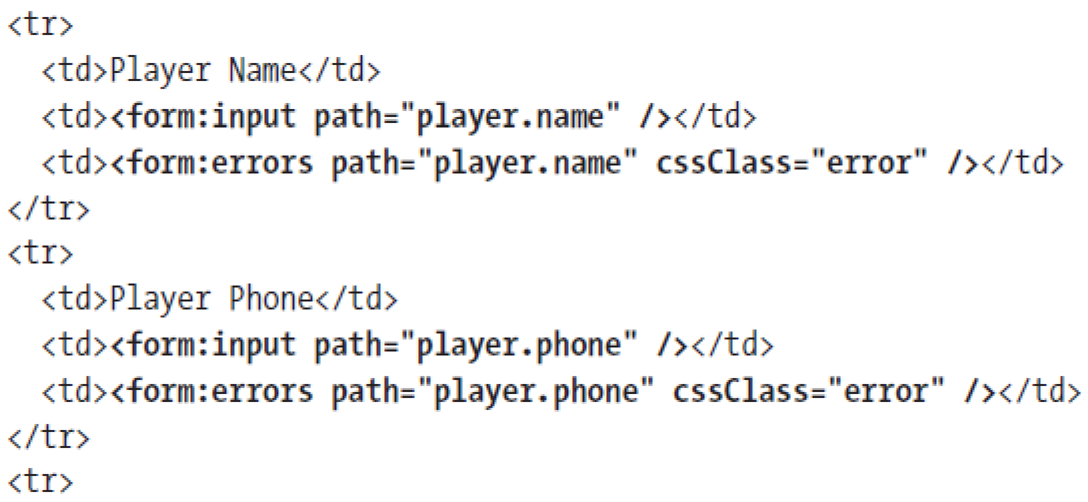
- <bean id="reservationSuccessController"
- class="org.springframework.web.servlet.mvc.ParameterizableViewController">
- <property name="viewName" value="reservationSuccess" />
- </bean>

- U can simply configure the same redirect view as follows :

- <bean id="reservationFormController"
- class="com.training.court.web.ReservationFormController">
- <property name="reservationService" ref="reservationService" />
- <property name="formView" value="reservationForm" />
- <property name="successView" value=“**redirect:reservationSuccess" />**
- </bean>

Initializing the Command Object

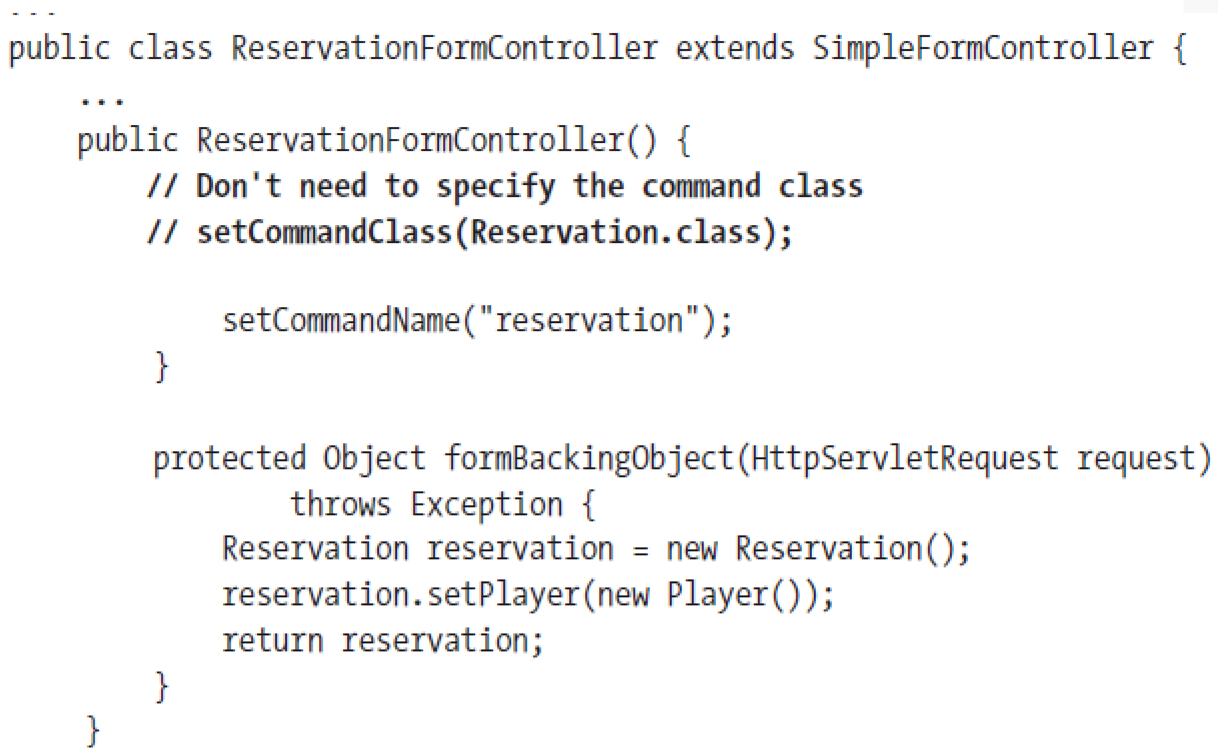
- in some cases you may have to initialize the command
- object by yourself. For example, let's consider a situation where you are going to bind a player's name and phone to a Reservation object's player property:



```
<tr>
  <td>Player Name</td>
  <td><form:input path="player.name" /></td>
  <td><form:errors path="player.name" cssClass="error" /></td>
</tr>
<tr>
  <td>Player Phone</td>
  <td><form:input path="player.phone" /></td>
  <td><form:errors path="player.phone" cssClass="error" /></td>
</tr>
<tr>
```

However, when the command class Reservation has just been instantiated, the player property is null, so it will cause an exception when rendering this form.

- To solve this problem, you have to initialize the command object by yourself. You can override the **formBackingObject**() method of SimpleFormController for this purpose.



```
...
public class ReservationFormController extends SimpleFormController {
    ...
    public ReservationFormController() {
        // Don't need to specify the command class
        // setCommandClass(Reservation.class);

            setCommandName("reservation");
        }

        protected Object formBackingObject(HttpServletRequest request)
                throws Exception {
            Reservation reservation = new Reservation();
            reservation.setPlayer(new Player());
            return reservation;
        }
    }
```

bindOnNewForm

- The bindOnNewForm property sets whether the request parameters should be bound to the command object when creating a new form.
- If bindOnNewForm is set to true, the request parameters will be bound to the properties of the same name.
- U can test with following URL
- http://localhost:8080/court/reservationForm.htm?date=2008-01-14

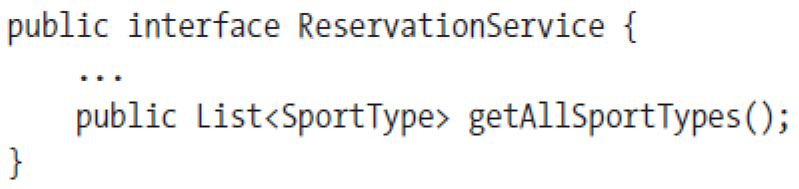
sessionForm

- **sessionForm**, sets whether the command object should be stored in
- the session.
- By default, this is false so that a new command object will be created on each request, even when rendering the form again due to binding errors.

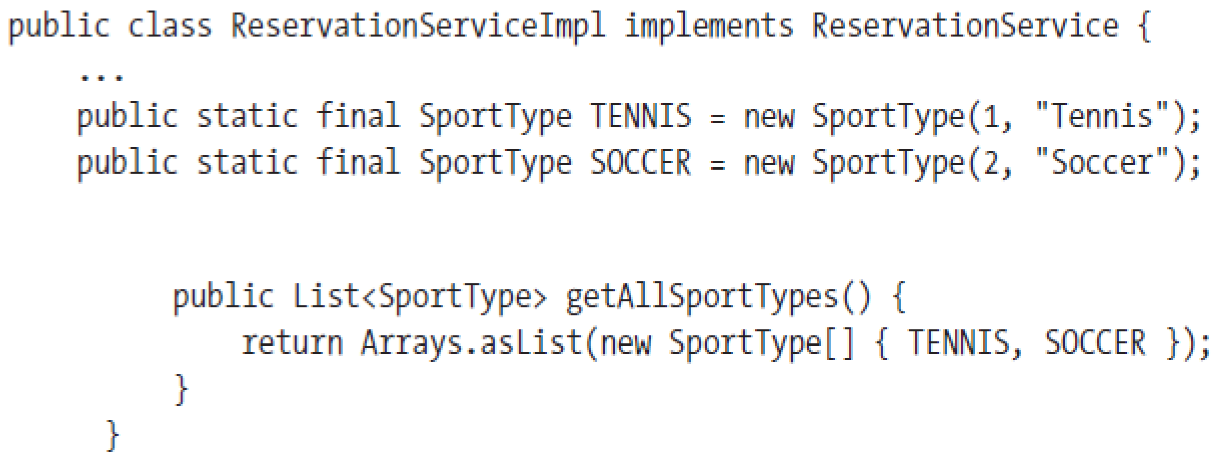
- If this property is set to true, the command object will be stored in the session for subsequent uses, until the form completes successfully. Then this command object will be cleared from the session.

Providing Form Reference Data

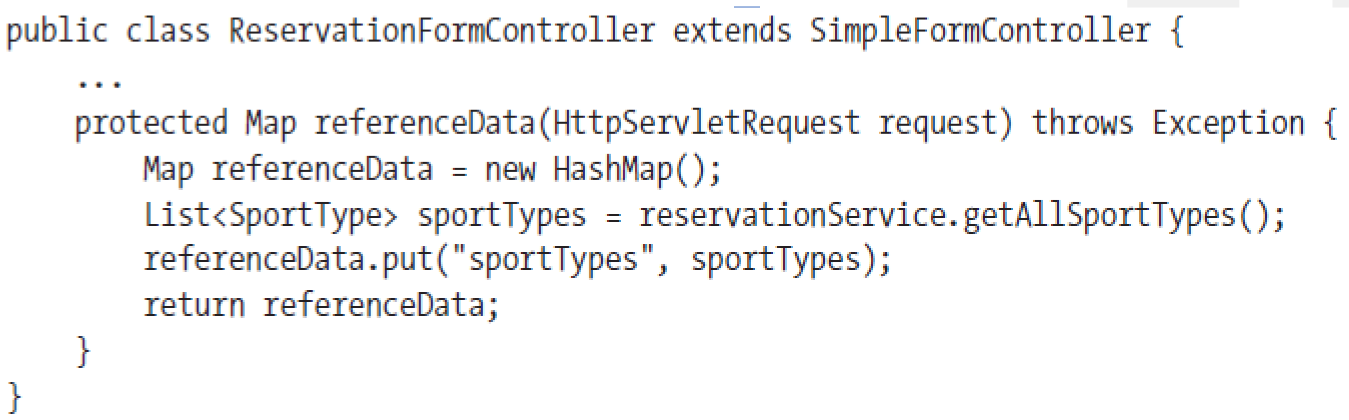
- When a form controller is requested to render the form view, it may have some types of reference data to provide to the form (e.g., the items to display in an HTML selection). Now suppose you would like to allow a user to select the sport type when reserving a court.



```
public interface ReservationService {
    ...
    public List<SportType> getAllSportTypes();
}
```



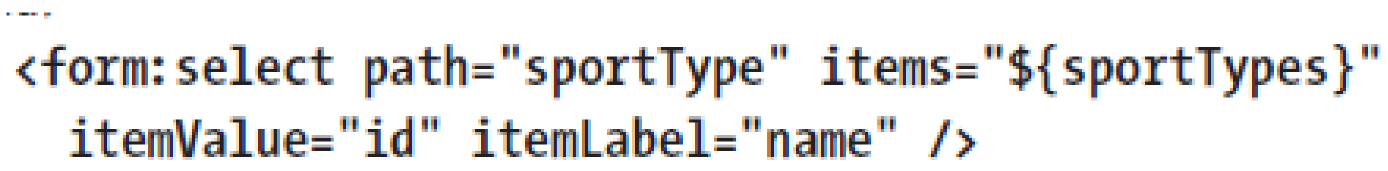
```
public class ReservationServiceImpl implements ReservationService {
    ...
    public static final SportType TENNIS = new SportType(1, "Tennis");
    public static final SportType SOCCER = new SportType(2, "Soccer");


        public List<SportType> getAllSportTypes() {
            return Arrays.asList(new SportType[] { TENNIS, SOCCER });
        }
  }
```



```
public class ReservationFormController extends SimpleFormController {
    ...
    protected Map referenceData(HttpServletRequest request) throws Exception {
        Map referenceData = new HashMap();
        List<SportType> sportTypes = reservationService.getAllSportTypes();
        referenceData.put("sportTypes", sportTypes);
        return referenceData;
    }
}
```

You should put the reference data in a map and return it for this method. This map will be added to the model and passed to the form view automatically.

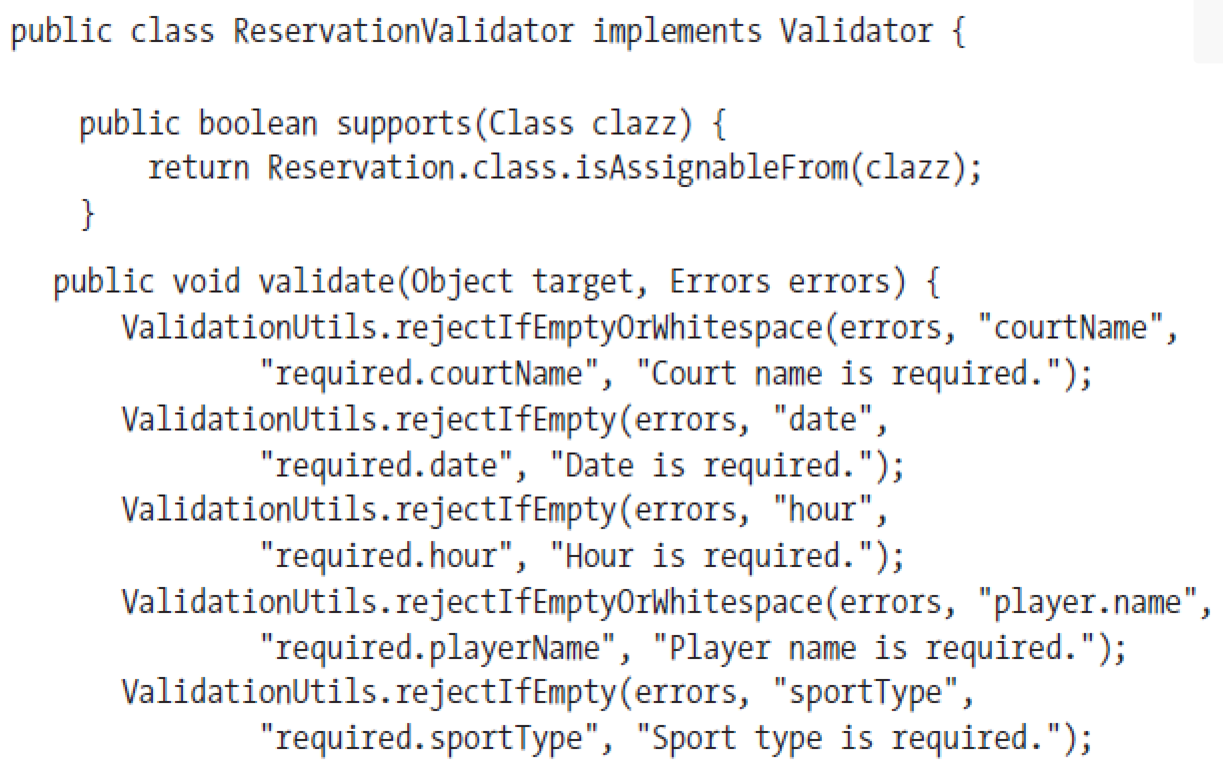
In JSP, U can use the following code to display the reference Data



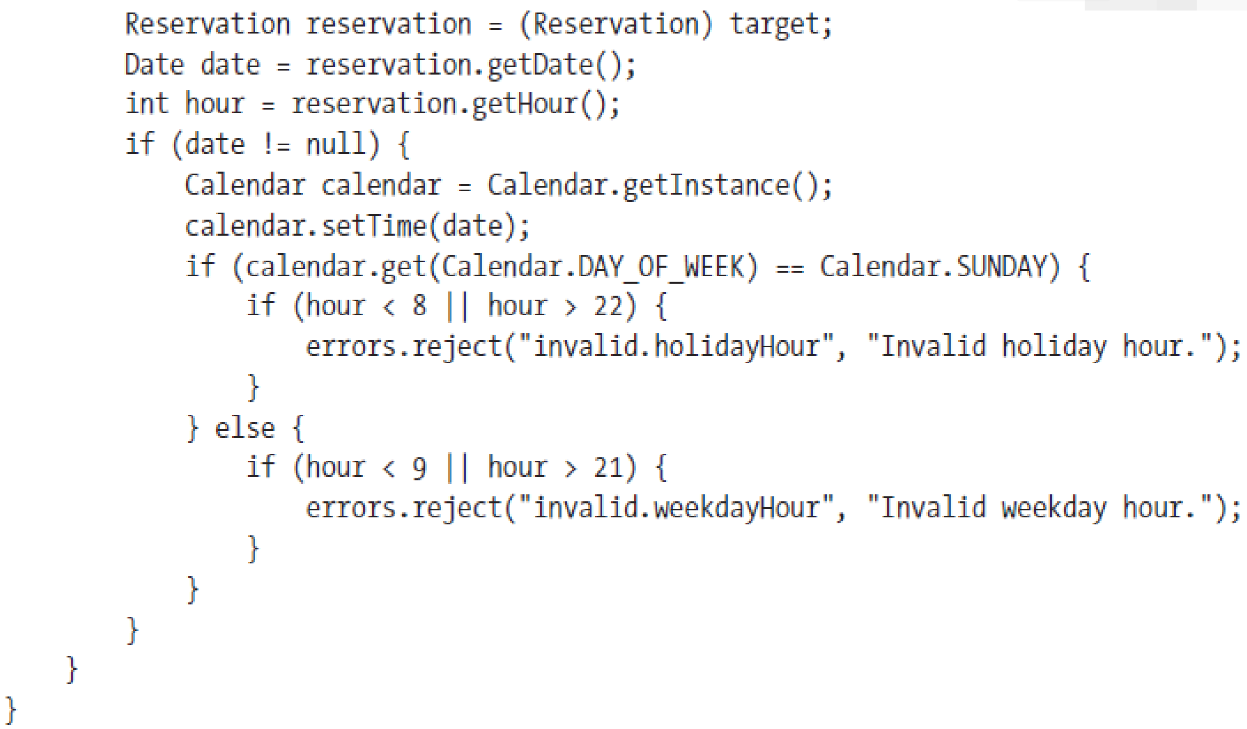
```
<form:select path="sportType" items="${sportTypes}"
  itemValue="id" itemLabel="name" />
```

Validating Form Data

- SimpleFormController can help you to validate the command object after it binds the form field values. The validation is done by a validator object that implements the **Validator** interface.



```
public class ReservationValidator implements Validator {

    public boolean supports(Class clazz) {
        return Reservation.class.isAssignableFrom(clazz);
    }

   public void validate(Object target, Errors errors) {
       ValidationUtils.rejectIfEmptyOrWhitespace(errors, "courtName",
               "required.courtName", "Court name is required.");
       ValidationUtils.rejectIfEmpty(errors, "date",
               "required.date", "Date is required.");
       ValidationUtils.rejectIfEmpty(errors, "hour",
               "required.hour", "Hour is required.");
       ValidationUtils.rejectIfEmptyOrWhitespace(errors, "player.name",
               "required.playerName", "Player name is required.");
       ValidationUtils.rejectIfEmpty(errors, "sportType",
               "required.sportType", "Sport type is required.");
```



```
        Reservation reservation = (Reservation) target;
        Date date = reservation.getDate();
        int hour = reservation.getHour();
        if (date != null) {
            Calendar calendar = Calendar.getInstance();
            calendar.setTime(date);
            if (calendar.get(Calendar.DAY_OF_WEEK) == Calendar.SUNDAY) {
                if (hour < 8 || hour > 22) {
                    errors.reject("invalid.holidayHour", "Invalid holiday hour.");
                }
            } else {
                if (hour < 9 || hour > 21) {
                    errors.reject("invalid.weekdayHour", "Invalid weekday hour.");
                }
            }
        }
    }
}
```

In this validator, you use utility methods such as rejectIfEmptyOrWhitespace() and rejectIfEmpty() in the ValidationUtils class to validate the required form fields. If any of these formfields is empty, these methods will create a ***field error that will be bound to the*** **field.** The second argument of these methods is the property name, while the third and fourth are the error code and default error message.

- <bean id="reservationFormController"
- class="com.training.court.web.ReservationFormController">
- ...
- **<property name="validator">**
- **<bean class="com.training.court.domain.ReservationValidator" />**
- **</property>**
- </bean>

- As your validators may create errors during validation, you should define messages for
- the error codes for displaying to the user. If you have ResourceBundleMessageSource defined, you can include the following error messages in your resource bundle for the appropriate locale (e.g., messages.properties for the default locale):
- required.courtName=Court name is required
- required.date=Date is required
- required.hour=Hour is required
- required.playerName=Player name is required
- required.sportType=Sport type is required
- invalid.holidayHour=Invalid holiday hour

Validating with Commons Validator

- Spring Modules provides an implementation of Validator called DefaultBeanValidator. DefaultBeanValidator can be configured as follows :

```
<bean id="beanValidator" class=
"org.springmodules.commons.validator.DefaultBeanValidator">
<property name="validatorFactory" ref="validatorFactory" />
</bean>
```

- DefaultBeanValidator doesn’t do any actual validation work. Instead, it delegates to Commons Validator to validate field values.

- As you can see, DefaultBeanValidator has a validatorFactory property that is wired with a reference to a validatorFactory bean. The validatorFactory bean is declared using the following XML:

```
<bean id="validatorFactory" class=
"org.springmodules.commons.validator.DefaultValidatorFactory">
        <property name="validationConfigLocations">
                <list>
                <value>WEB-INF/validator-rules.xml</value>
                <value>WEB-INF/validation.xml</value>
                </list>
        </property>
</bean>
```

- DefaultValidatorFactory is a class that loads the Commons Validator configuration on behalf of DefaultBeanValidator. The validationConfigLocations property takes a list of one or more validation configurations. Here we’ve asked it to load two configurations: validator-rules.xml and validation.xml.

.Grouping Multiple Actions into a Controller

- The **MultiActionController** class provided by Spring MVC allows you to group multiple related actions into a single controller. Your controller can extend MultiActionController and include multiple handler methods to handle multiple actions.

- In a multi-action controller, you can define one or more handler methods of the following form:

- **public (ModelAndView | Map | String | void) actionName(**
- **HttpServletRequest, HttpServletResponse [,HttpSession] [,CommandObject]);**

Mapping methods

- When a request is mapped to a multi-action controller by handler mappings, it has to be narrowed down to a particular handler method within the controller. MultiActionController allows you to configure method mappings with a **MethodNameResolver** object.

- See the example 52SpringMvcUsingMultiActionController .

- By default, MultiActionController maps URLs to handler methods by the method names.
- For the controller in example, the URLs will be mapped to the following methods:
- /member/add.htm → **add()**
- /member/remove.htm →**remove()**
- /member/list.htm → **list()**

Mapping URLs to Handler Methods

- By default, MultiActionController uses **InternalPathMethodNameResolver** to map URLs to
- handler methods by the method names. However, if you want to add a prefix or a suffix to the mapped method names, you have to configure this resolver explicitly:

- <bean id="memberController"
- class="com.training.web.MemberController">
- ...
- <property name="methodNameResolver">
- <bean class="org.springframework.web.servlet.mvc.multiaction.
- **InternalPathMethodNameResolver">**
- **<property name="suffix" value="Member" />**
- </bean>
- </property>
- </bean>

- Then the last path of a URL before the extension will be mapped to a handler method by adding Member as the suffix:

- /member/**add.htm → addMember()**
- /member/**remove.htm → removeMember()**
- /member/**list.htm →listMember()**

- Alternatively, you can configure **PropertiesMethodNameResolver** to map URLs to handler methods by specifying the mapping definition explicitly:
- <bean id="memberController"
- class="com.training.web.MemberController">
- ...
- <property name="methodNameResolver">
- <bean class="org.springframework.web.servlet.mvc.multiaction.
- **PropertiesMethodNameResolver">**
- <property name="mappings">
- <props>
- **<prop key="/member/add.htm">addMember</prop>**
- **<prop key="/member/remove.htm">removeMember</prop>**
- **<prop key="/member/list.htm">listMember</prop>**
- </props>
- </property>
- </bean>
- </property>
- </bean>

Q & A

Contact: smitakumar@synergetics-india.com

Build COMPETENCY
across your TEAM

SYNERGETICS
GET IT RIGHT

Microsoft Partner
Gold Cloud Platform
Silver Learning

Thank You