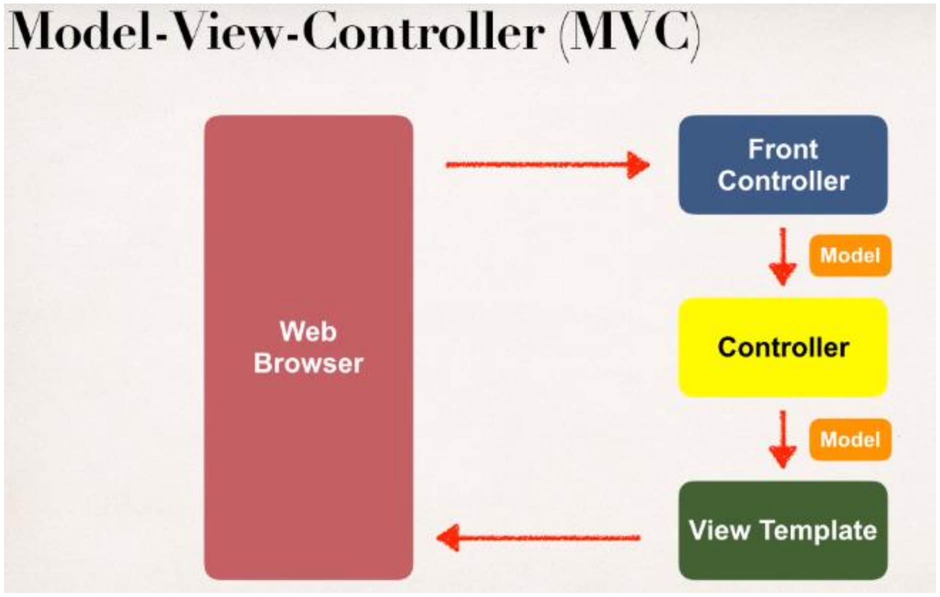
Spring MVC

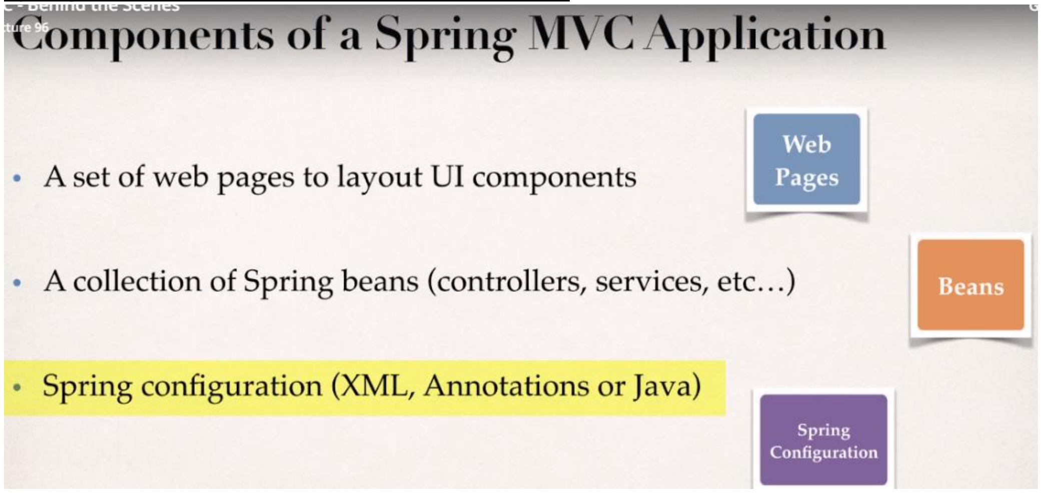
It is a Framework based on MVC pattern for Web applications. It uses the features of core Spring framework like IOC and DI



Benefits of Spring MVC:

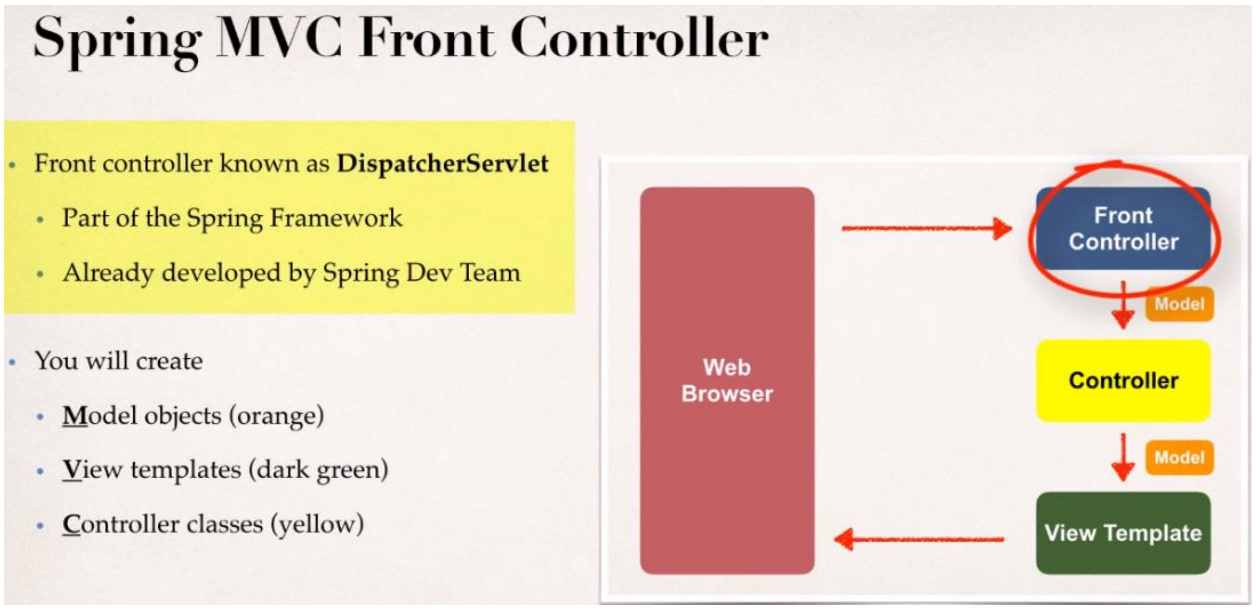
- Web apps can be built using spring concepts
- It provides a set of reusable UI components
- IT helps managing application state for web requests
- Form validation
- Flexible View configuration i.e not restricted to JSP

Spring MVC behind the scene and Architecture:



Components of a Spring MVC Application

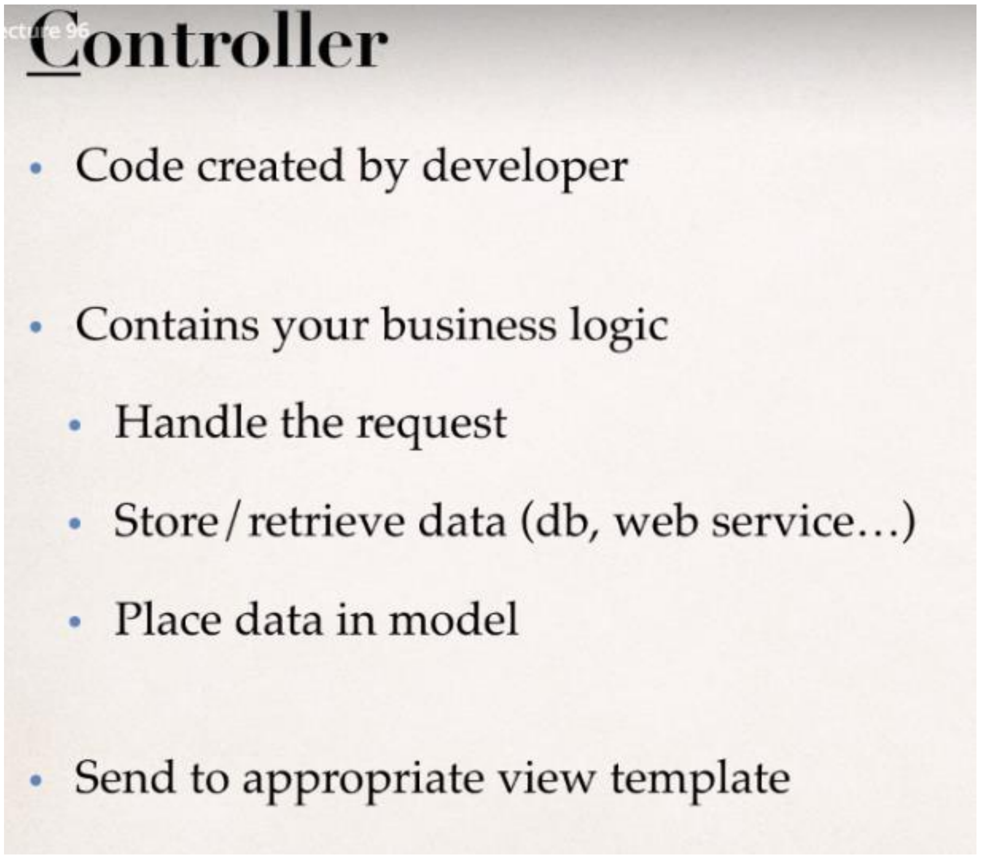
- A set of web pages to layout UI components
- A collection of Spring beans (controllers, services, etc…)
- Spring configuration (XML, Annotations or Java)



Spring MVC Front Controller

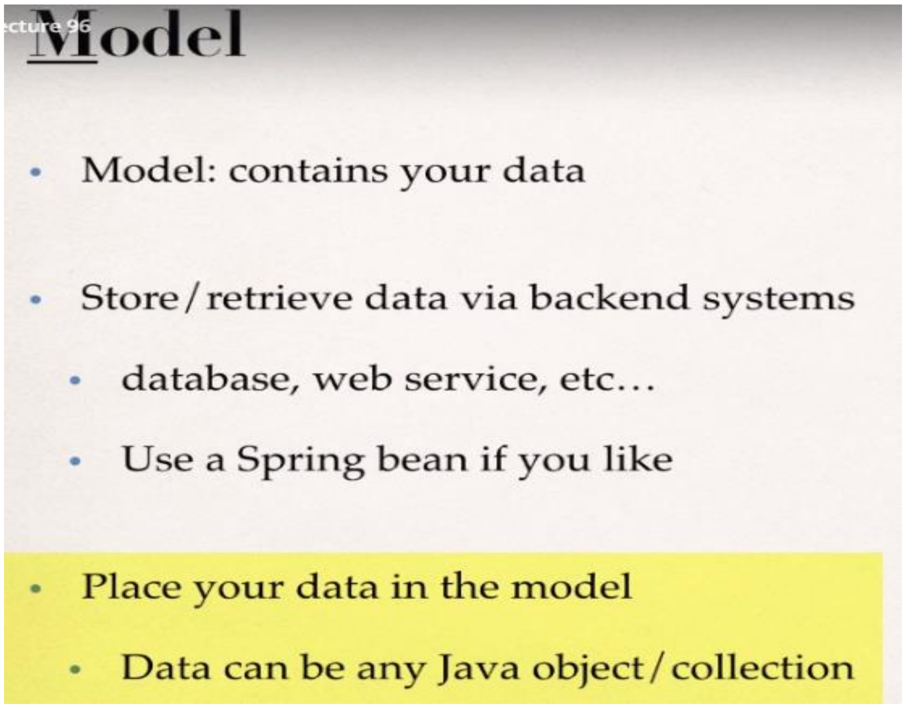
- Front controller known as **DispatcherServlet**
  - Part of the Spring Framework
  - Already developed by Spring Dev Team
- You will create
  - **M**odel objects (orange)
  - **V**iew templates (dark green)
  - **C**ontroller classes (yellow)

No need of writing Front Controller



Controller

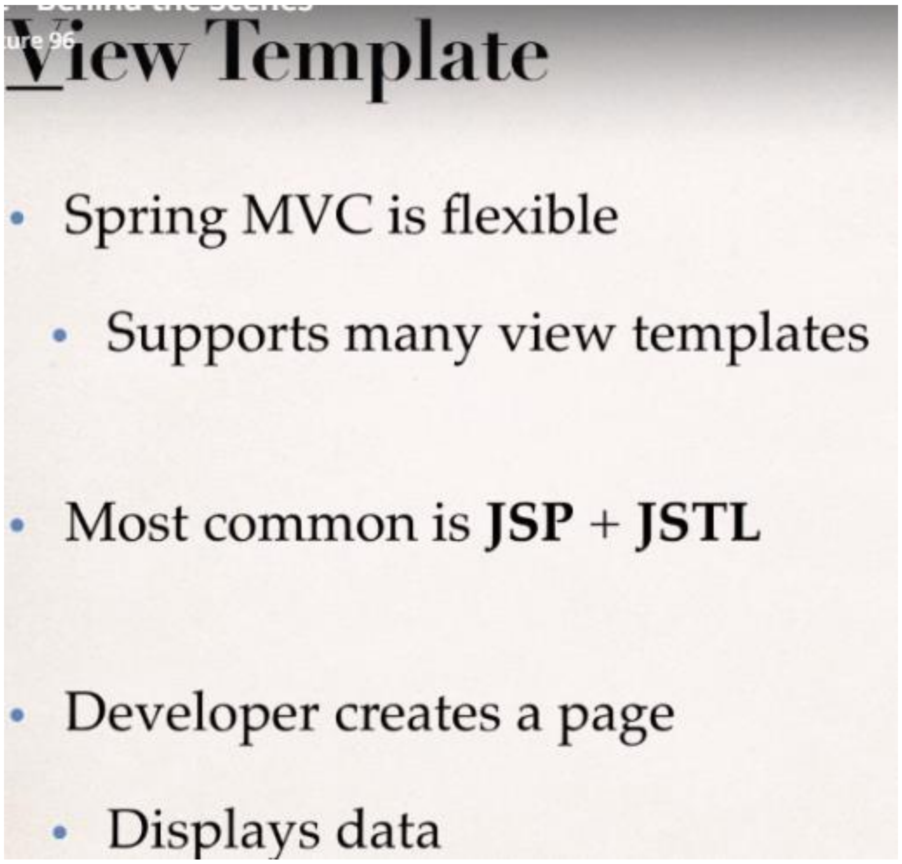
- Code created by developer
- Contains your business logic
  - Handle the request
  - Store / retrieve data (db, web service…)
  - Place data in model
- Send to appropriate view template



ecture 96

Model

- Model: contains your data
- Store / retrieve data via backend systems
  - database, web service, etc…
  - Use a Spring bean if you like
- Place your data in the model
  - Data can be any Java object / collection



ure 96

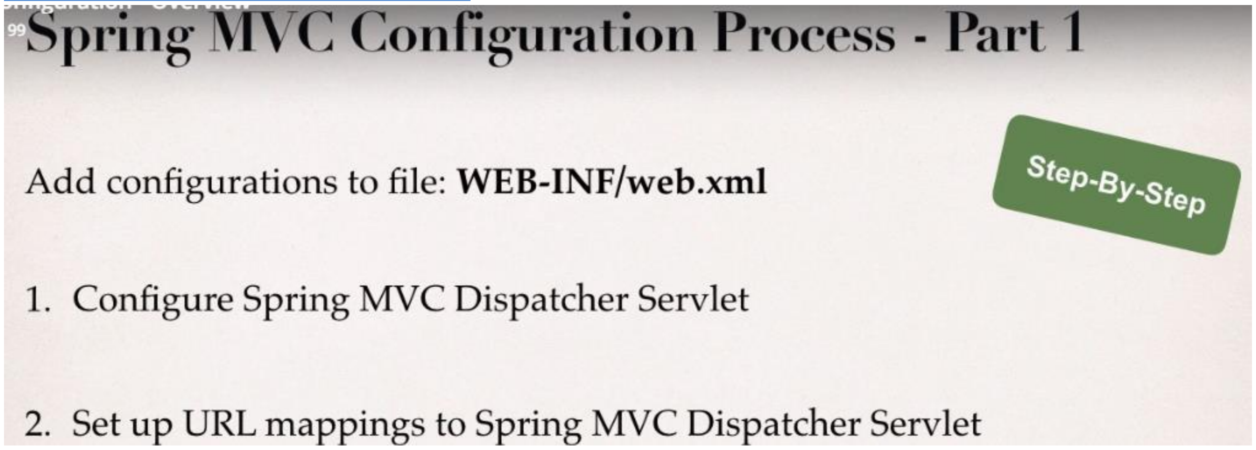
View Template

- Spring MVC is flexible
  - Supports many view templates
- Most common is **JSP** + **JSTL**
- Developer creates a page
  - Displays data

Environment Setup:

- Tomcat server
- Eclipse
- Tomcat connected to Eclipse

Spring MVC configuration:

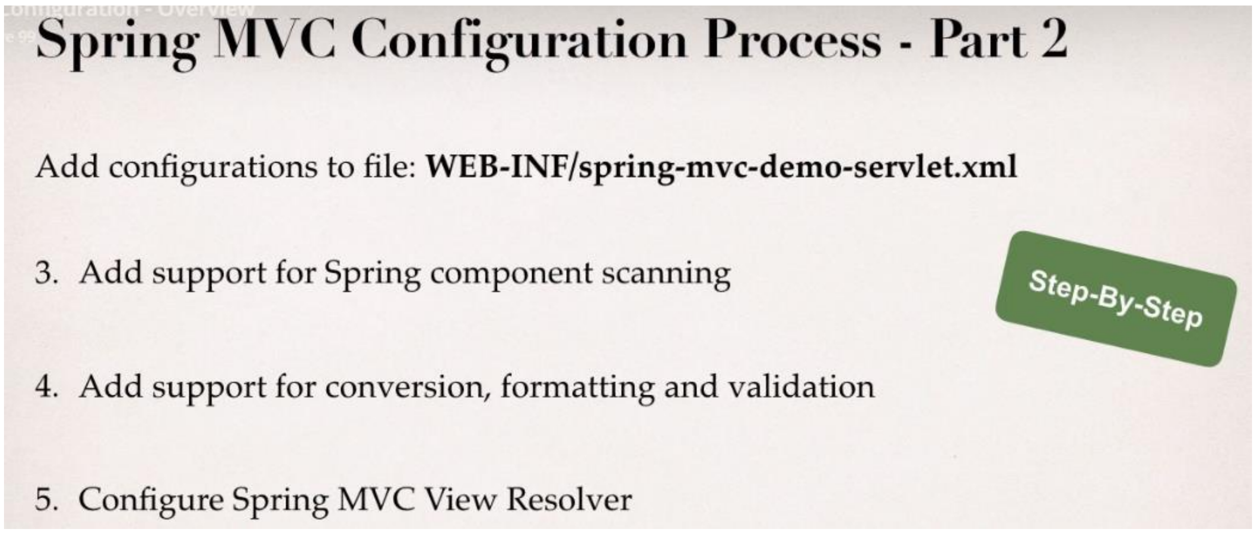


Spring MVC Configuration Process - Part 1

Add configurations to file: **WEB-INF/web.xml**

1. Configure Spring MVC Dispatcher Servlet
2. Set up URL mappings to Spring MVC Dispatcher Servlet

Step-By-Step

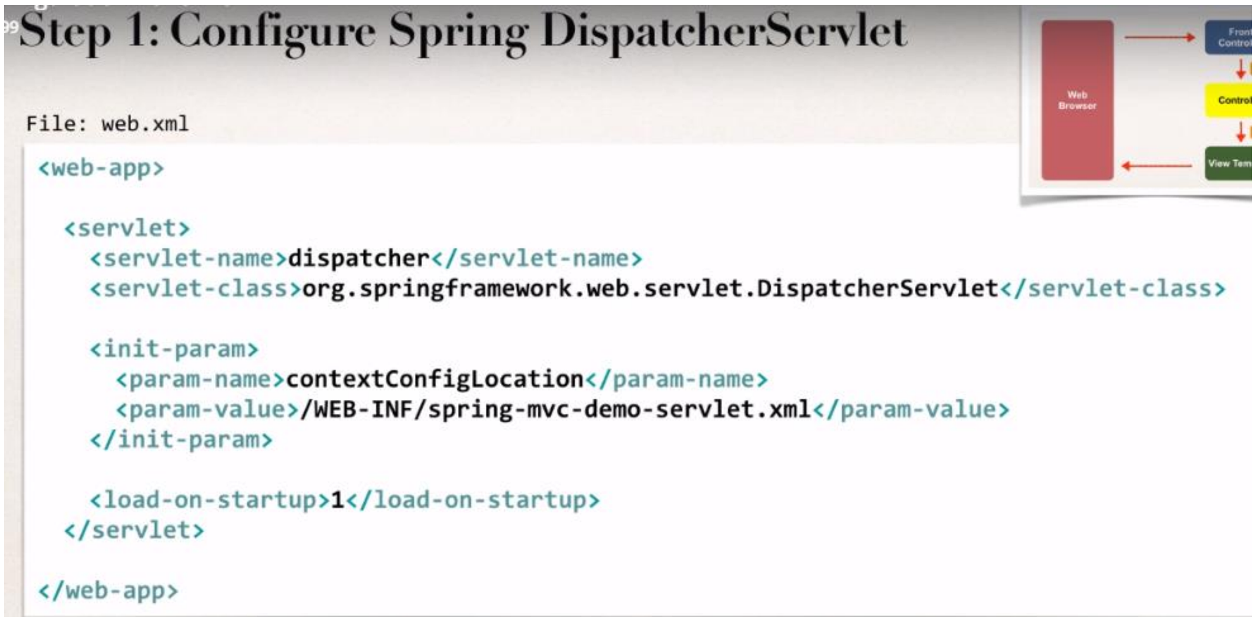


Spring MVC Configuration Process - Part 2

Add configurations to file: **WEB-INF/spring-mvc-demo-servlet.xml**

3. Add support for Spring component scanning
4. Add support for conversion, formatting and validation
5. Configure Spring MVC View Resolver

Step-By-Step

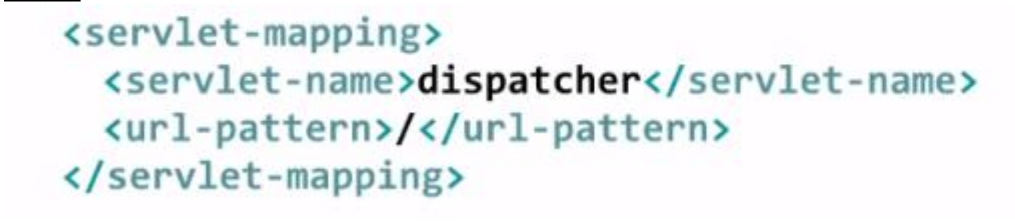


Step 1: Configure Spring DispatcherServlet

File: web.xml

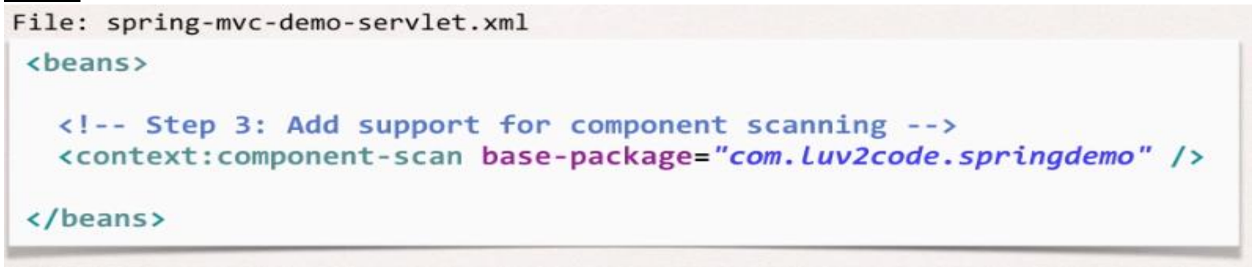
```
<web-app>

  <servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>

    <init-param>
      <param-name>contextConfigLocation</param-name>
      <param-value>/WEB-INF/spring-mvc-demo-servlet.xml</param-value>
    </init-param>

    <load-on-startup>1</load-on-startup>
  </servlet>

</web-app>
```

Step2:



```
  <servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
  </servlet-mapping>
```

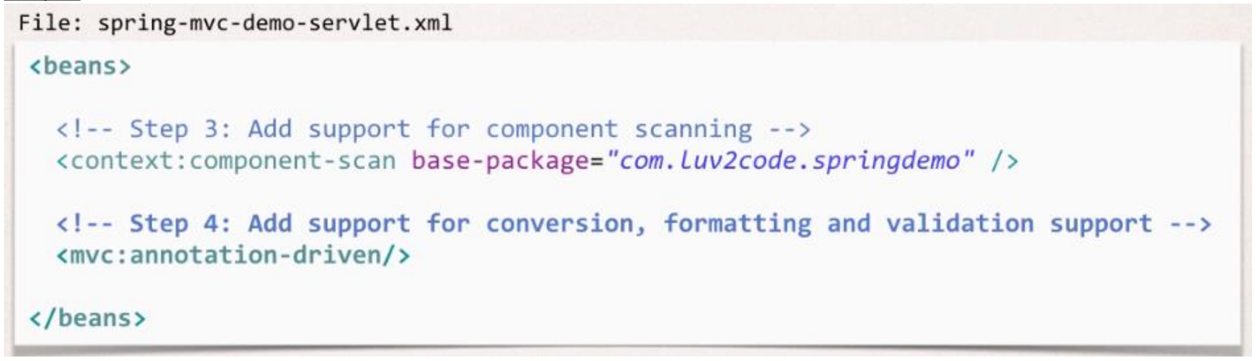
Step3:



File: spring-mvc-demo-servlet.xml

```
<beans>

  <!-- Step 3: Add support for component scanning -->
  <context:component-scan base-package="com.luv2code.springdemo" />

</beans>
```

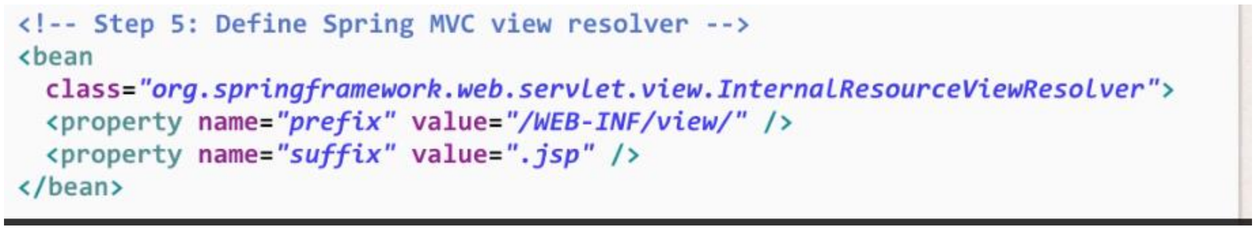
Step4:



File: spring-mvc-demo-servlet.xml

```
<beans>

  <!-- Step 3: Add support for component scanning -->
  <context:component-scan base-package="com.luv2code.springdemo" />

  <!-- Step 4: Add support for conversion, formatting and validation support -->
  <mvc:annotation-driven/>

</beans>
```

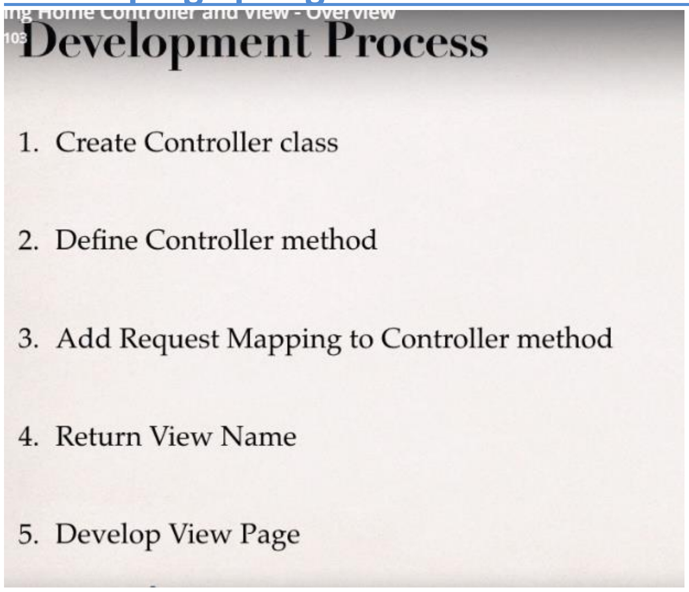
Step5:



```xml
<!-- Step 5: Define Spring MVC view resolver -->
<bean
  class="org.springframework.web.servlet.view.InternalResourceViewResolver">
  <property name="prefix" value="/WEB-INF/view/" />
  <property name="suffix" value=".jsp" />
</bean>
```

Web.xml and springConfig.xml should be placed under WEB-INF folder and spring jars under WEB_INF/lib

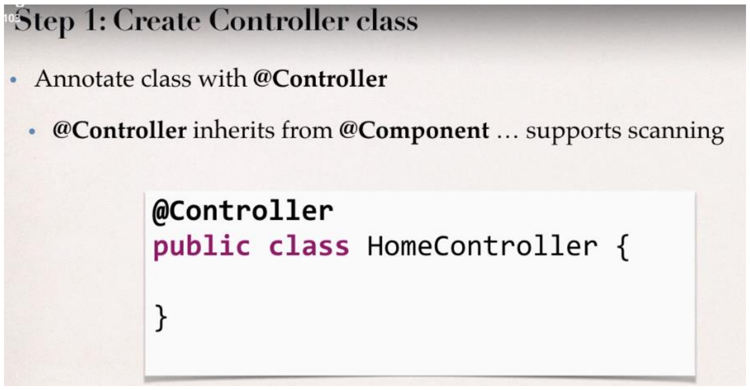
Developing Spring Controller and views:



ing Home Controller and View - Overview

Development Process

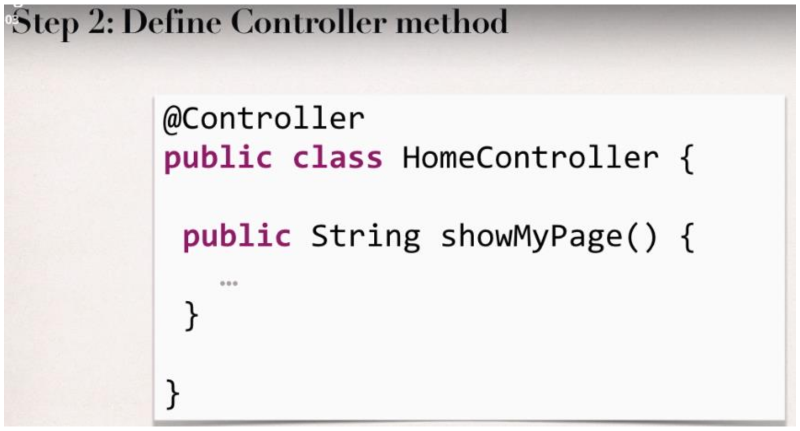
1. Create Controller class
2. Define Controller method
3. Add Request Mapping to Controller method
4. Return View Name
5. Develop View Page



Step 1: Create Controller class

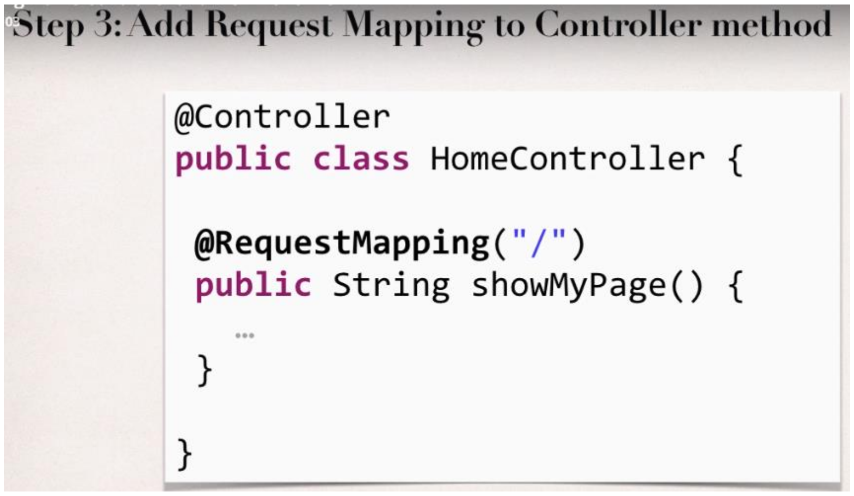
- Annotate class with **@Controller**
  - **@Controller** inherits from **@Component** … supports scanning

```java
@Controller
public class HomeController {

}
```



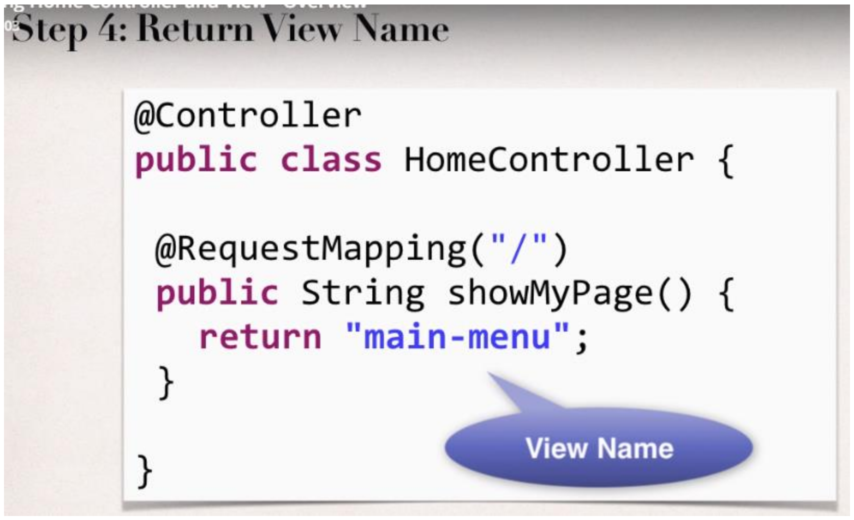
Step 2: Define Controller method

```java
@Controller
public class HomeController {

 public String showMyPage() {
    ...
 }

}
```



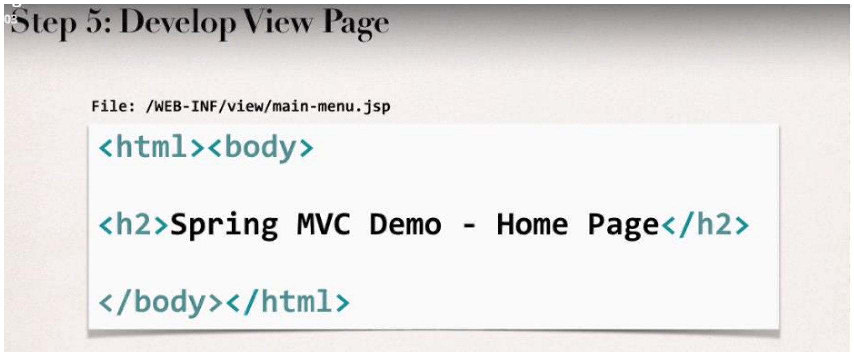
Step 3: Add Request Mapping to Controller method

```java
@Controller
public class HomeController {

 @RequestMapping("/")
 public String showMyPage() {
    ...
 }

}
```



Step 4: Return View Name

```java
@Controller
public class HomeController {

 @RequestMapping("/")
 public String showMyPage() {
   return "main-menu";
 }

}
```



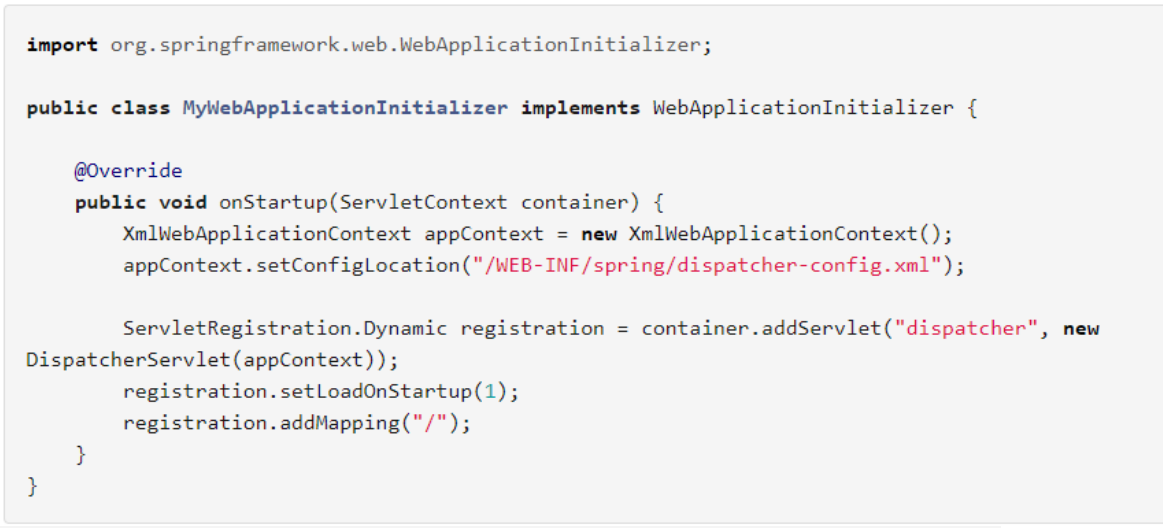
Step 5: Develop View Page

File: /WEB-INF/view/main-menu.jsp

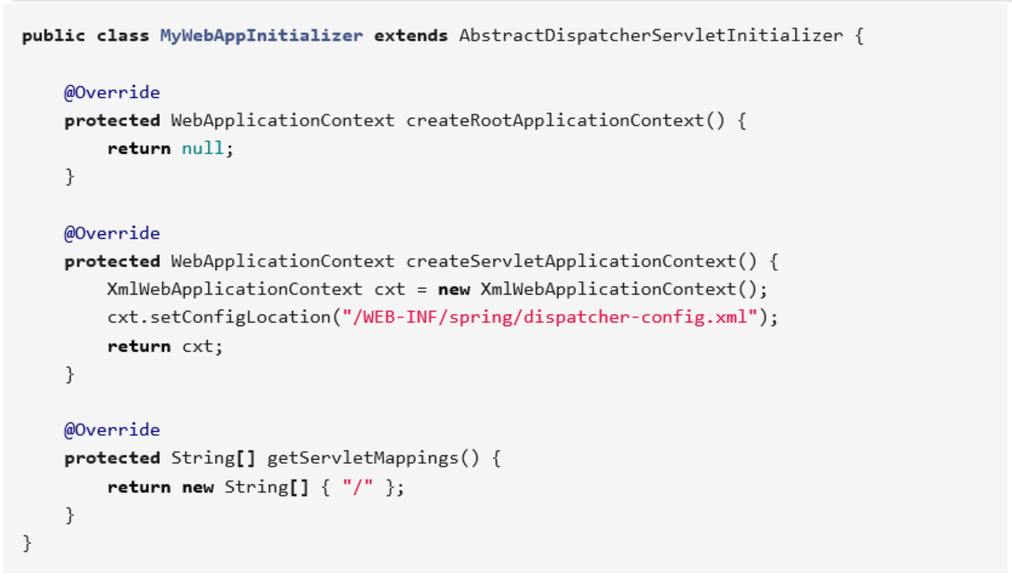
```
<html><body>

<h2>Spring MVC Demo - Home Page</h2>

</body></html>
```

We can configure without xml also

Link: https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-container-config

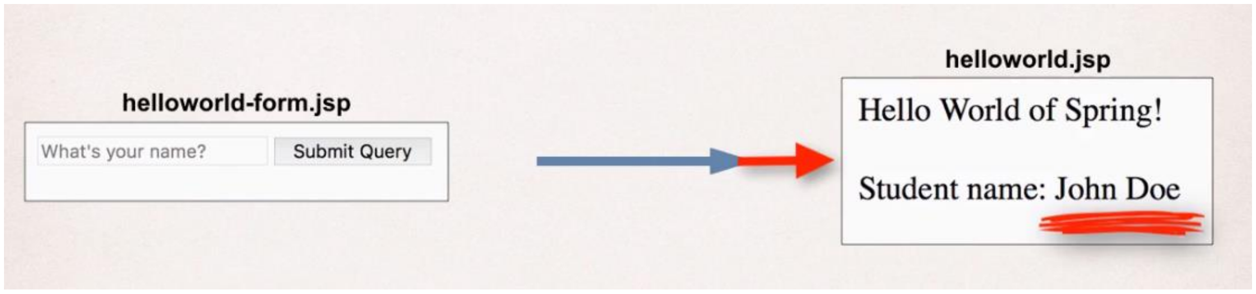


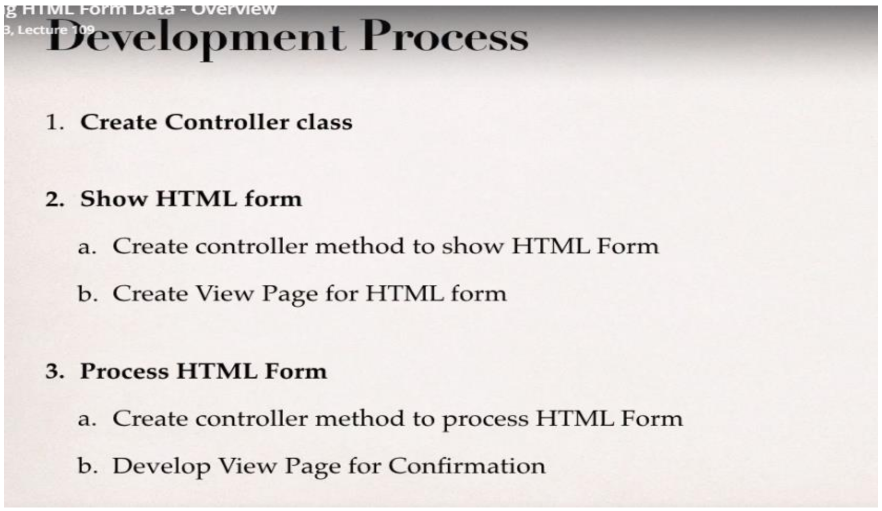
```
import org.springframework.web.WebApplicationInitializer;

public class MyWebApplicationInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext container) {
        XmlWebApplicationContext appContext = new XmlWebApplicationContext();
        appContext.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");

        ServletRegistration.Dynamic registration = container.addServlet("dispatcher", new
DispatcherServlet(appContext));
        registration.setLoadOnStartup(1);
        registration.addMapping("/");
    }
}
```



```
public class MyWebAppInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        XmlWebApplicationContext cxt = new XmlWebApplicationContext();
        cxt.setConfigLocation("/WEB-INF/spring/dispatcher-config.xml");
        return cxt;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

Scenario 2: Reading Data from Form





g HTML Form Data - Overview

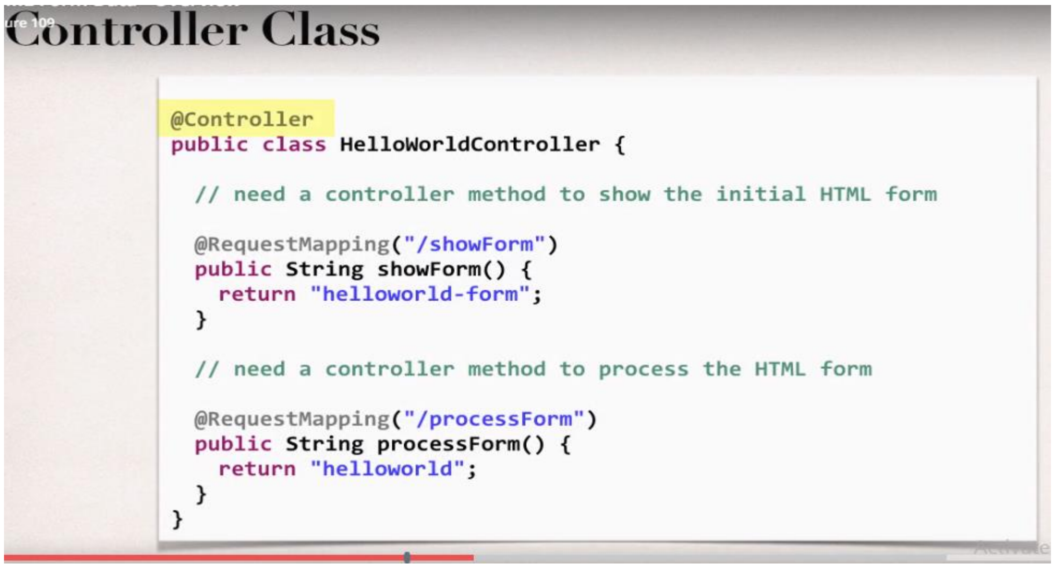
3, Lecture 109

Development Process

1. **Create Controller class**

2. **Show HTML form**
   a. Create controller method to show HTML Form
   b. Create View Page for HTML form

3. **Process HTML Form**
   a. Create controller method to process HTML Form
   b. Develop View Page for Confirmation

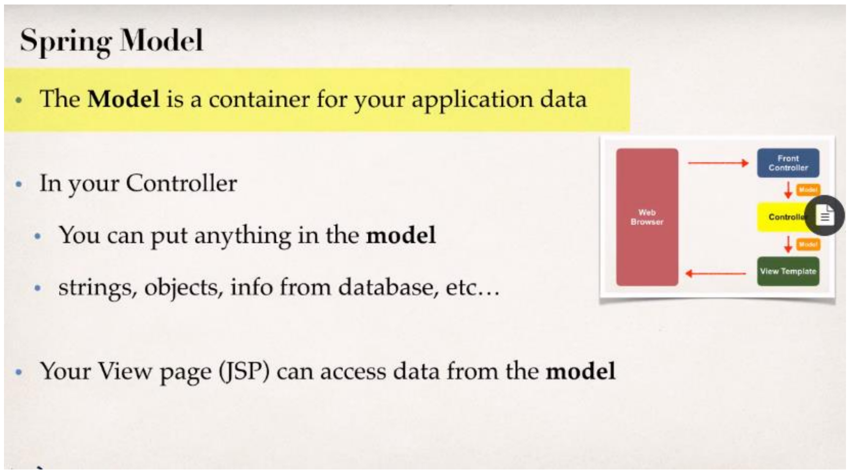


ure 109

Controller Class

```
@Controller
public class HelloWorldController {

  // need a controller method to show the initial HTML form

  @RequestMapping("/showForm")
  public String showForm() {
    return "helloworld-form";
  }

  // need a controller method to process the HTML form

  @RequestMapping("/processForm")
  public String processForm() {
    return "helloworld";
  }
}
```

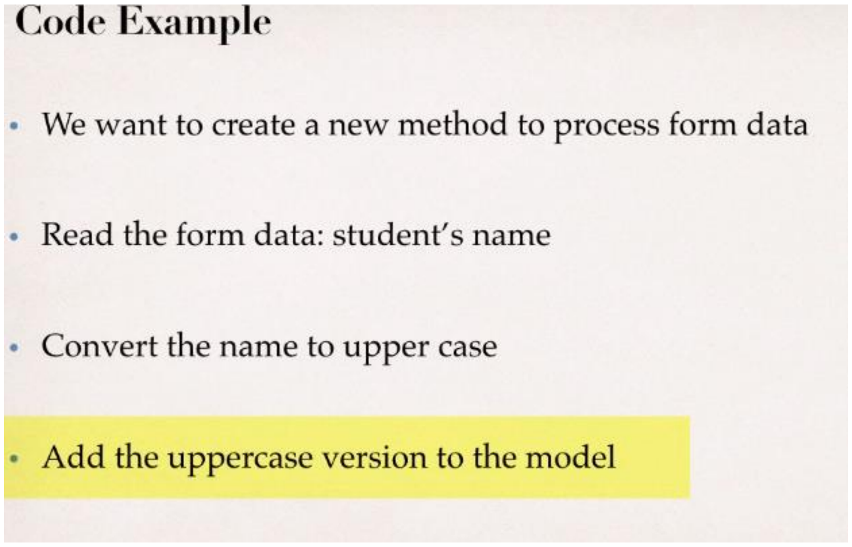
Adding Data to model Layer



Spring Model

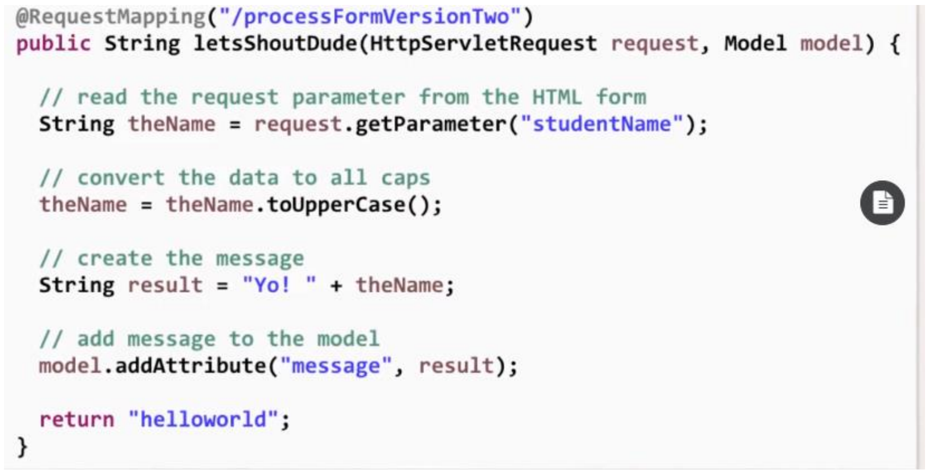
- The **Model** is a container for your application data
- In your Controller
  - You can put anything in the **model**
  - strings, objects, info from database, etc…
- Your View page (JSP) can access data from the **model**

Web Browser
Front Controller
Model
Controller
Model
View Template

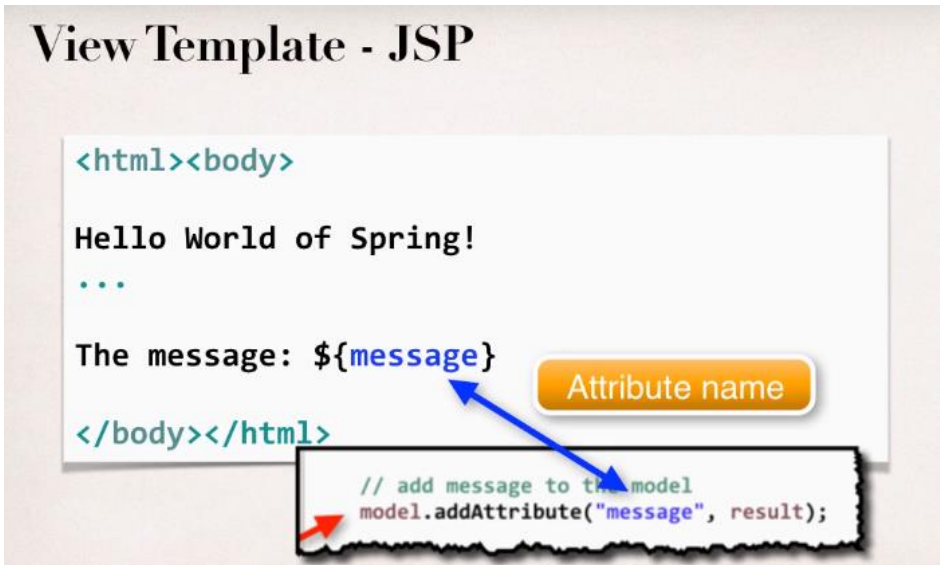


Code Example

- We want to create a new method to process form data
- Read the form data: student's name
- Convert the name to upper case
- Add the uppercase version to the model



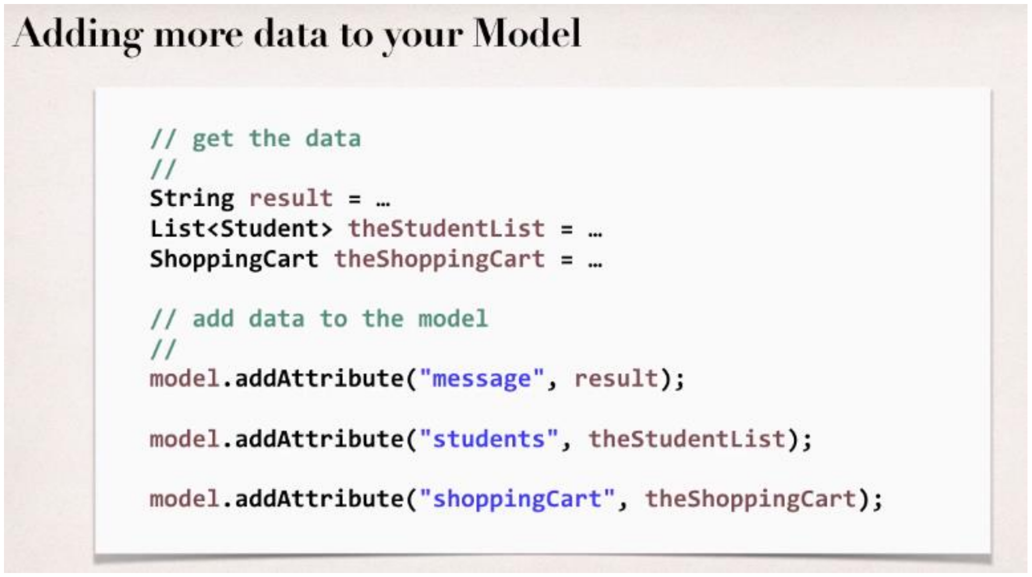
```java
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(HttpServletRequest request, Model model) {

  // read the request parameter from the HTML form
  String theName = request.getParameter("studentName");

  // convert the data to all caps
  theName = theName.toUpperCase();

  // create the message
  String result = "Yo! " + theName;

  // add message to the model
  model.addAttribute("message", result);

  return "helloworld";
}
```



View Template - JSP

```
<html><body>

Hello World of Spring!
...

The message: ${message}

</body></html>
```

```
// add message to the model
model.addAttribute("message", result);
```

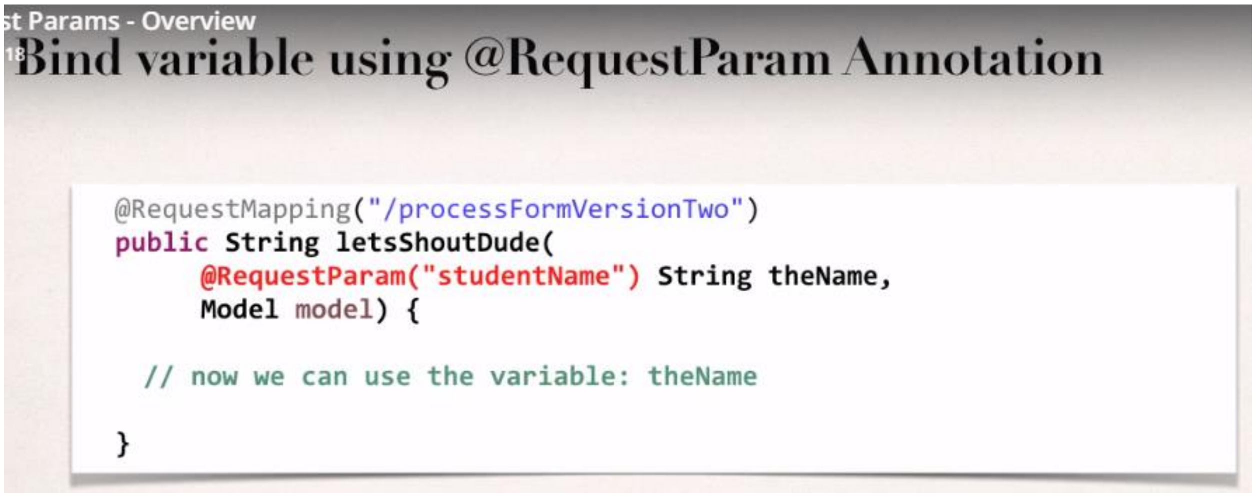


Adding more data to your Model

```
// get the data
//
String result = …
List<Student> theStudentList = …
ShoppingCart theShoppingCart = …

// add data to the model
//
model.addAttribute("message", result);

model.addAttribute("students", theStudentList);

model.addAttribute("shoppingCart", theShoppingCart);
```

Now Instead of using HTTPServletRequest we can use a special annotation **@RequestParam**

The above code can be re done as


st Params - Overview

18 Bind variable using @RequestParam Annotation

```
@RequestMapping("/processFormVersionTwo")
public String letsShoutDude(
     @RequestParam("studentName") String theName,
     Model model) {

  // now we can use the variable: theName

}
```

How to Use CSS, images, javascript

Any static resource is processed as URL mapping in Spring MVC, so we should configure the reference to our static resources in Spring.xml file.

Step1:
Create your resources folder structure and files.

Step2:
You can place this entry anywhere in your Spring MVC config file.

```
<mvc:resources mapping="/resources/**"  location="/resources/">
</mvc:resources>
```

Step3:
Now in your view pages, you can access the static files using this syntax:

For image

```
<img src="${pageContext.request.contextPath}/resources/images/spring-logo.png">
```

For CSS and javascript

```
<head>
        <link rel="stylesheet" type="text/css"
          href="${pageContext.request.contextPath}/resources/css/my-test.css">
        <script src="${pageContext.request.contextPath}/resources/js/simple-test.js"></script>
</head>
```

Special Note:-

How to deploy webApplication as WAR

1. In Eclipse, stop Tomcat
2. Right-click your project and select Export > WAR File
3. In the Destination field, enter: <any-directory>/mycoolapp.war
4. Outside of Eclipse, start Tomcat-If you are using MS Windows, then you should find it on the Start menu
5. Make sure Tomcat is up and running by visiting: http://localhost:8080
6. Deploy your new WAR file by copying it to <tomcat-install-directory>\webapps. Give it about 10-15 seconds to make the deployment. You'll know the deployment is over because you'll see a new folder created in webapps ... with your WAR file name.
7. Visit your new app. If your war file was: mycoolapp.war then you can access it with: http://localhost:8080/mycoolapp/

Adding Request Mapping to Controller

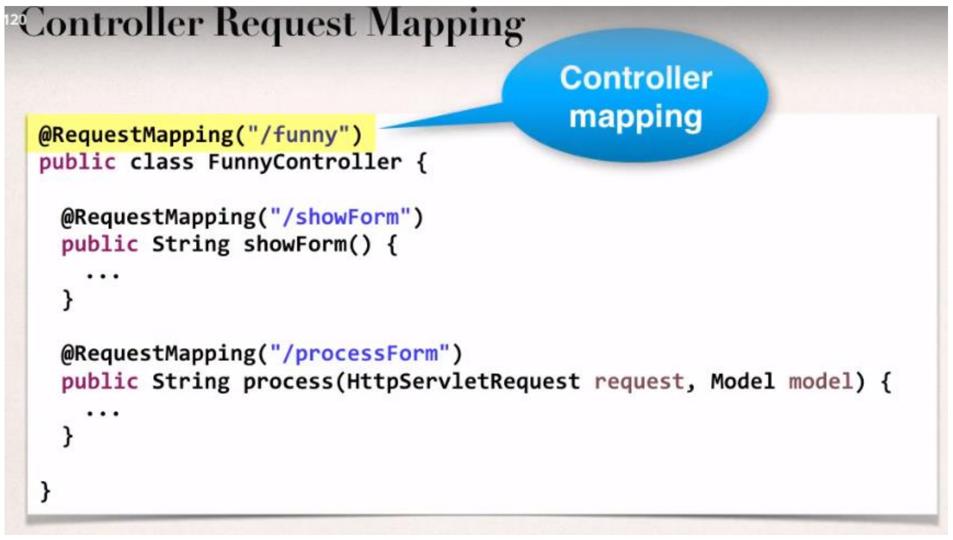
Sometime there can be same request url in different controllers, eg /showForm in HelloWorldController and /showForm in SillyController, this may result in error condition that bean cannot be created since already exists.

Error:

```
java.lang.IllegalStateException: Ambiguous mapping. Cannot map
'sillyController' method
public java.lang.String
com.luv2code.springdemo.mvc.SillyController.displayForm()
```

In such cases we can do request mapping to controller.
It serves as ParentMapping similar to directories and sub-directories

It can be done as

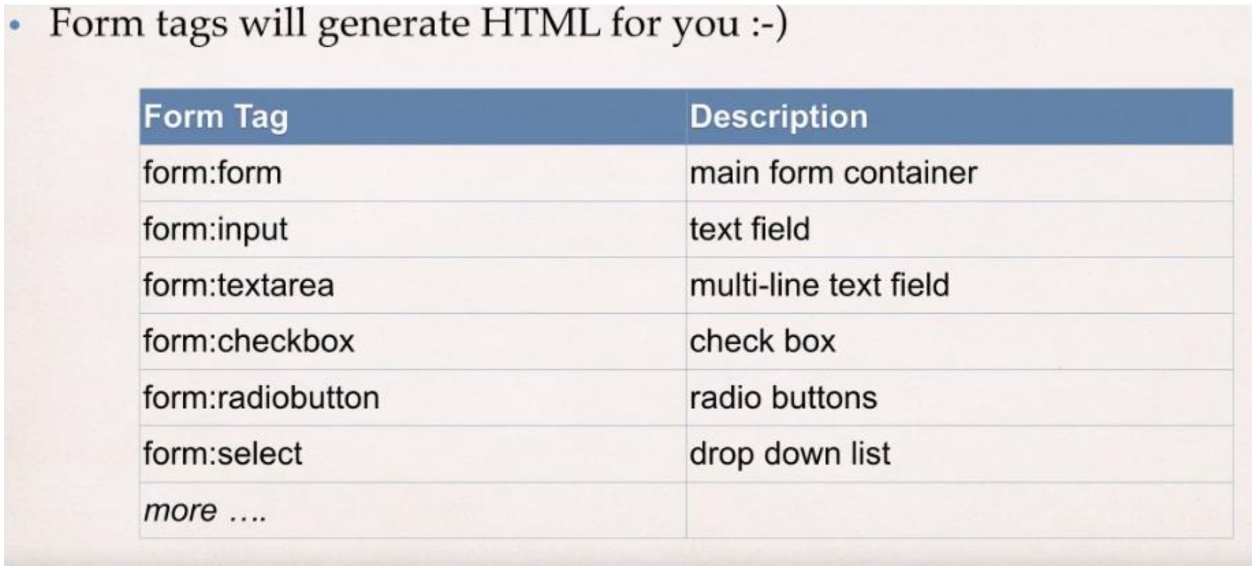


Controller Request Mapping

```
@RequestMapping("/funny")
public class FunnyController {

  @RequestMapping("/showForm")
  public String showForm() {
    ...
  }

  @RequestMapping("/processForm")
  public String process(HttpServletRequest request, Model model) {
    ...
  }

}
```

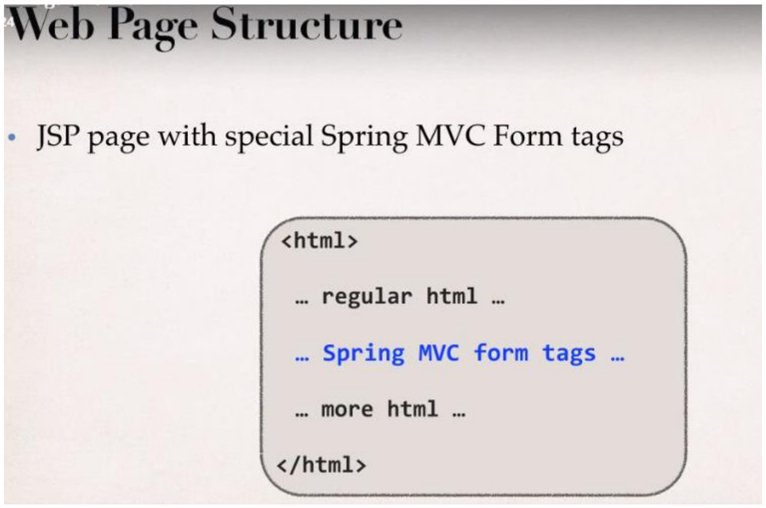
Spring MVC Tags overview

- Spring MVC forms support data binding
- Automatically setting/retrieving data in java objects



- Form tags will generate HTML for you :-)

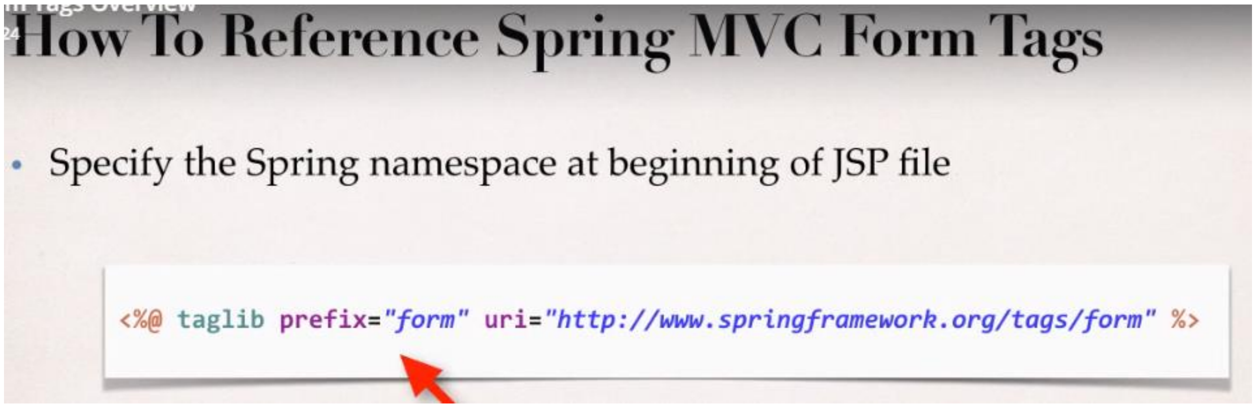
Form Tag	Description
form:form	main form container
form:input	text field
form:textarea	multi-line text field
form:checkbox	check box
form:radiobutton	radio buttons
form:select	drop down list
more ….	



Web Page Structure

- JSP page with special Spring MVC Form tags

```
<html>
  … regular html …
  … Spring MVC form tags …
  … more html …
</html>
```

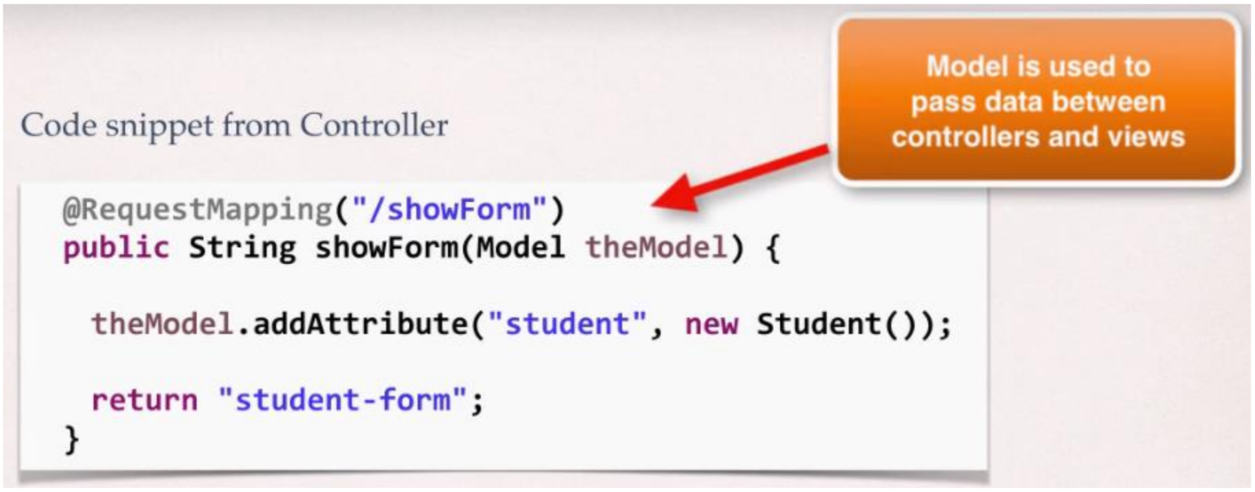


How To Reference Spring MVC Form Tags

- Specify the Spring namespace at beginning of JSP file

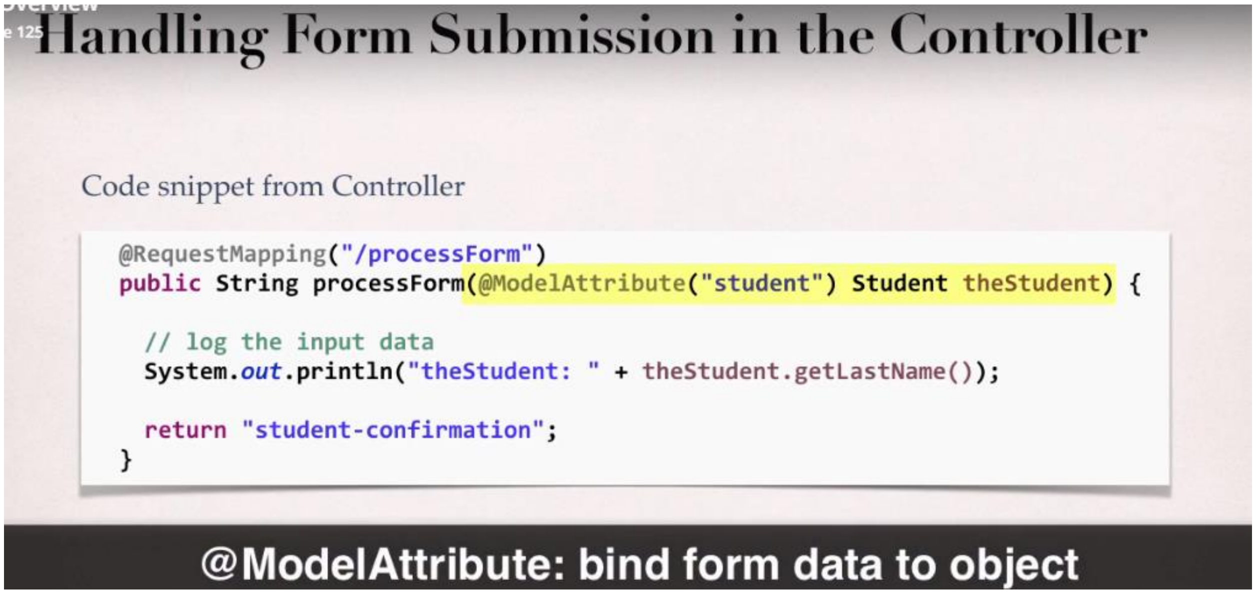
```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form" %>
```

Model attribute is must for using form tags


Code snippet from Controller

```java
@RequestMapping("/showForm")
public String showForm(Model theModel) {

  theModel.addAttribute("student", new Student());

  return "student-form";
}
```

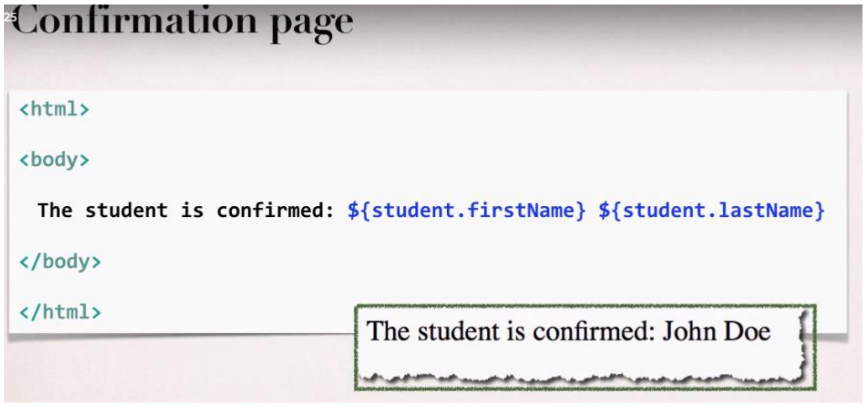
Attribute name should same used by form


Handling Form Submission in the Controller

Code snippet from Controller

```java
@RequestMapping("/processForm")
public String processForm(@ModelAttribute("student") Student theStudent) {

  // log the input data
  System.out.println("theStudent: " + theStudent.getLastName());

  return "student-confirmation";
}
```

@ModelAttribute: bind form data to object

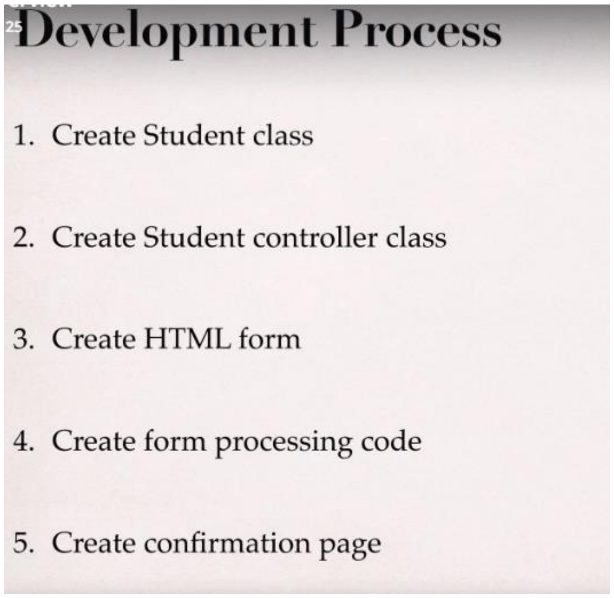

Confirmation page

```html
<html>

<body>

 The student is confirmed: ${student.firstName} ${student.lastName}

</body>

</html>
```

The student is confirmed: John Doe

Important annotations

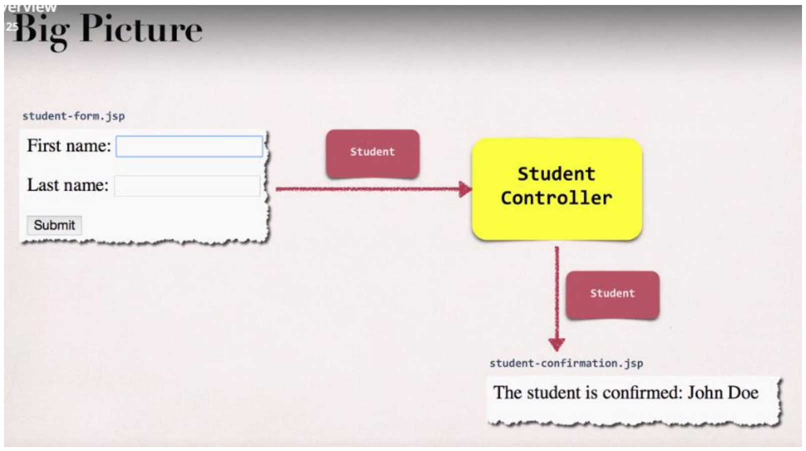
@ModelAttribute
modelAttribute in form
getter and setter methods



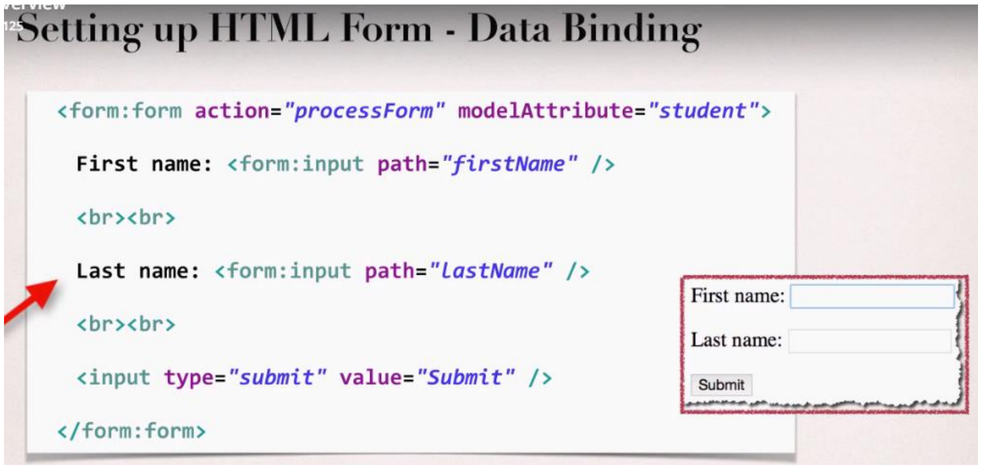
Development Process

1. Create Student class
2. Create Student controller class
3. Create HTML form
4. Create form processing code
5. Create confirmation page

<u>Spring MVC form tag for Text fields:</u>



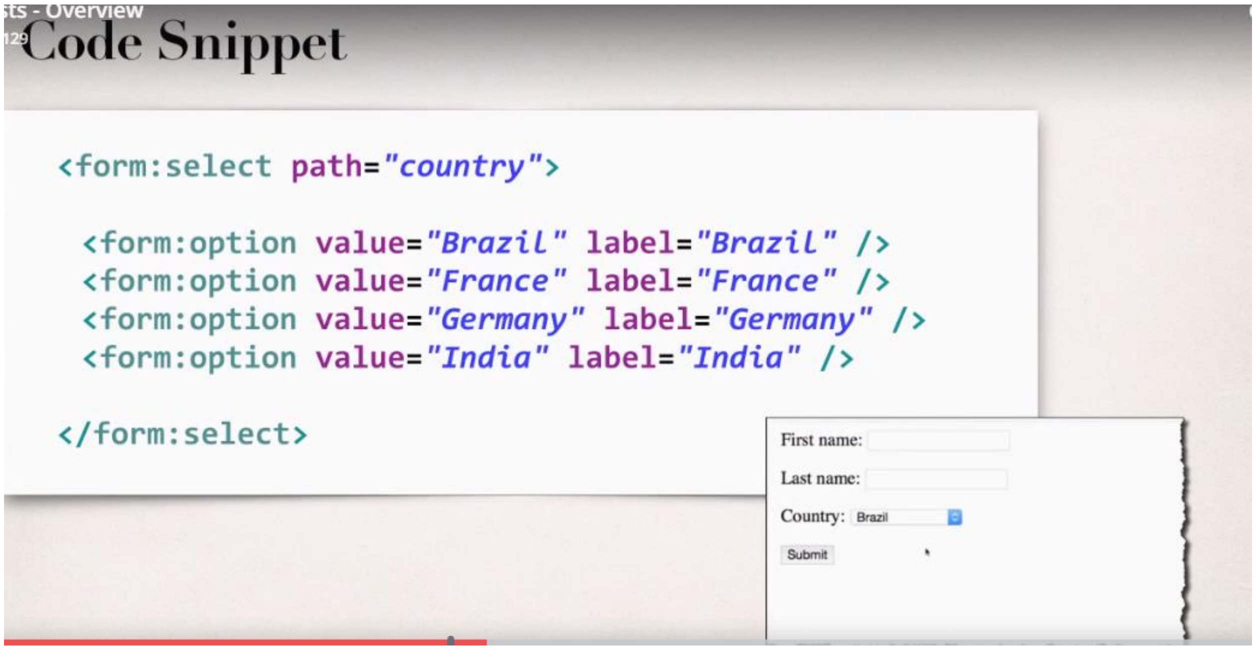
Big Picture


Setting up HTML Form - Data Binding

```
<form:form action="processForm" modelAttribute="student">

  First name: <form:input path="firstName" />

  <br><br>

  Last name: <form:input path="lastName" />

  <br><br>

  <input type="submit" value="Submit" />

</form:form>
```

Path="firstName" and path="lastName" Spring MVC uses this for getter and setter, so it should be same as bean's member variable

Spring MVC form tag for Dropdown list:


Code Snippet

```
<form:select path="country">

  <form:option value="Brazil" label="Brazil" />
  <form:option value="France" label="France" />
  <form:option value="Germany" label="Germany" />
  <form:option value="India" label="India" />

</form:select>
```

<form:select path="">

Instead of hard coding values in jsp, we can read it from java class

- Use a Map to create contry options
- Intialize it in constructor
- <form: **options item =${student.countryOptions}/>**

To read from properties file

1. Create a properties file to hold the countries. It will be a name value pair. Country code is name. Country name is the value.
New text file: WEB-INF/countries.properties

2. Update header section for Spring config file

3. Load the country options properties file in the Spring config file. Bean id: countryOptions
File: spring-mvc-dmo-servlet.xml
Add the following lines:
<util:properties id="countryOptions" location="classpath:../countries.properties" />

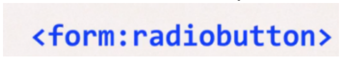
4. Inject the properties values into your Spring Controller: StudentController.java
@Value("${countryOptions}")
private Map<String, String> countryOptions;

5. Add the country options to the Spring MVC model. Attribute name: theCountryOptions
theModel.addAttribute("theCountryOptions", countryOptions);

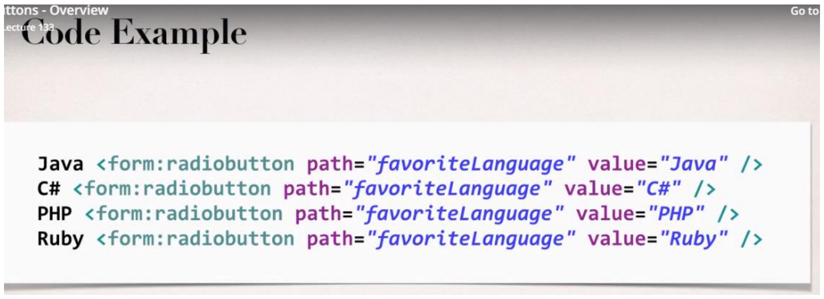
6. Update the JSP page, student-form.jsp, to use the new model attribute for the drop-down list: theCountryOptions
<form:select path="country">
 <form:options items="${theCountryOptions}" />
</form:select>

Spring MVC form tags for Radio buttons

Radio Button is represented as



```
<form:radiobutton>
```



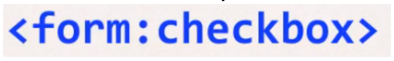
Code Example

```
Java <form:radiobutton path="favoriteLanguage" value="Java" />
C# <form:radiobutton path="favoriteLanguage" value="C#" />
PHP <form:radiobutton path="favoriteLanguage" value="PHP" />
Ruby <form:radiobutton path="favoriteLanguage" value="Ruby" />
```

We can also populate the radiobuttons from java class, create a map similar to dropdown list and use <form:**radiobuttons** path="favoriteLanguage" **items="${student.favoriteLanguageOptions**}" />

Spring MVC form tags for Check box

A check box is represented as

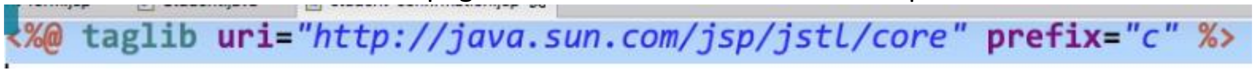


```
<form:checkbox>
```

To store the check box input we should use a String[] instead of private string field.
Example:

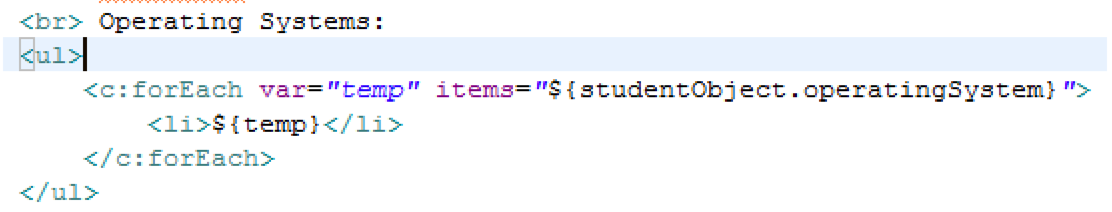
```
public String[] getOperatingSystem() {
	return operatingSystem;
}
public void setOperatingSystem(String[] operatingSystem) {
	this.operatingSystem = operatingSystem;
}
```

Also to show the values on view page we need to use the for each loop as below



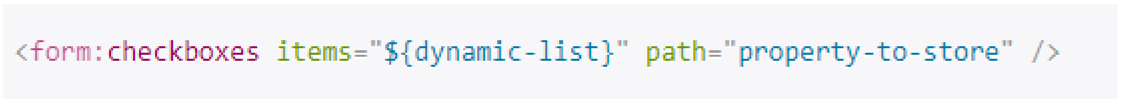
```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
```

In the view page it should be written as



```
<br> Operating Systems:
<ul>
    <c:forEach var="temp" items="${studentObject.operatingSystem}">
        <li>${temp}</li>
    </c:forEach>
</ul>
```

For getting check boxes from java class we can use



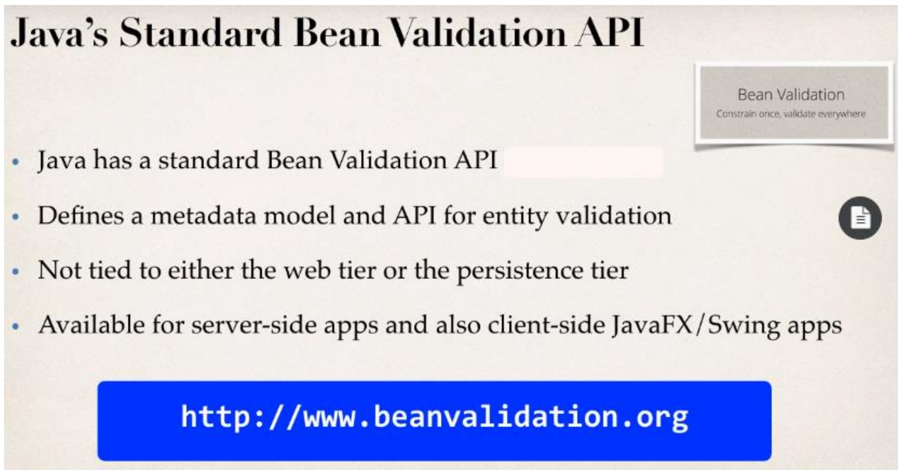
```
<form:checkboxes items="${dynamic-list}" path="property-to-store" />
```

Spring MVC Form Validation

Need for Form Validation:

Check user inputs for form

- Required fileds
- Valid number ranges
- Valid format
- Custom business rules

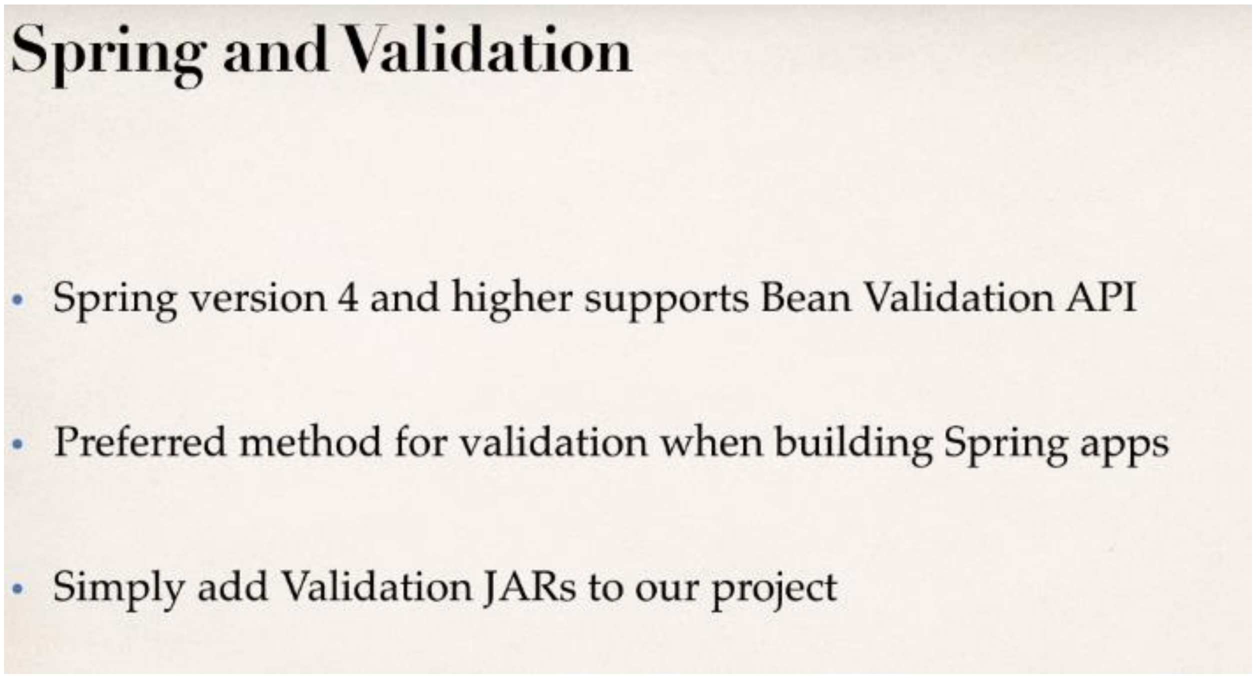


Java's Standard Bean Validation API

Bean Validation
Constrain once, validate everywhere

- Java has a standard Bean Validation API
- Defines a metadata model and API for entity validation
- Not tied to either the web tier or the persistence tier
- Available for server-side apps and also client-side JavaFX/Swing apps

http://www.beanvalidation.org

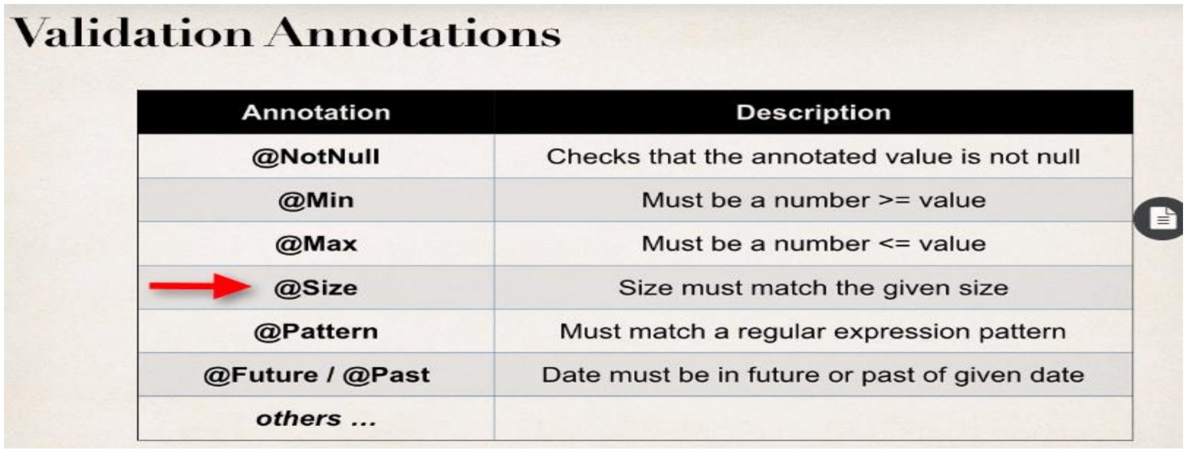


Spring and Validation

- Spring version 4 and higher supports Bean Validation API
- Preferred method for validation when building Spring apps
- Simply add Validation JARs to our project

Validation feature

- Required
- Validate length
- Validate numbers
- Validate regular expressions
- Custom validations



Validation Annotations

Annotation	Description
@NotNull	Checks that the annotated value is not null
@Min	Must be a number >= value
@Max	Must be a number <= value
@Size	Size must match the given size
@Pattern	Must match a regular expression pattern
@Future / @Past	Date must be in future or past of given date
others ...	

Important topics

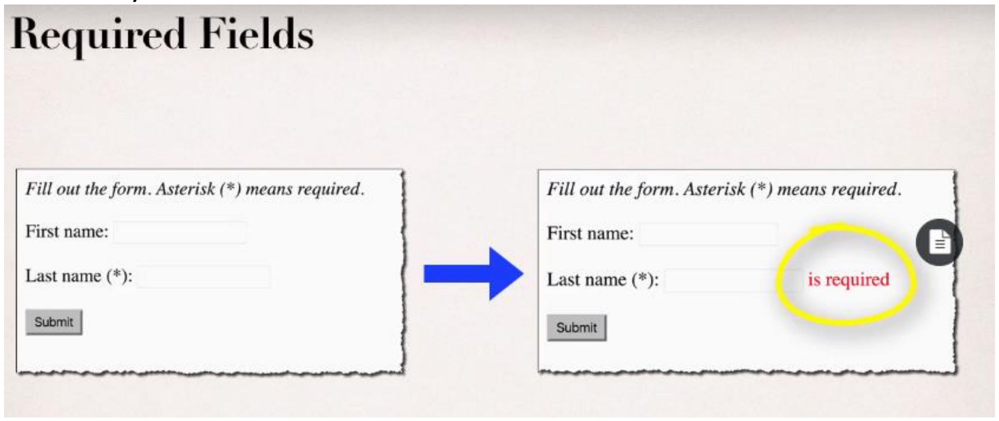
1. Setup development environment
2. Validate required field
3. Validate number ranges min, max
4. Validate regular expressions
5. Custom validations

1. **Development Environment**
   Download hibernate validation jars
   Go to www.hibernate.org > Hibernate validator > download
   Copy hibernate validator under lib(from downloads) and jars under lib/ required to WEB_INF/lib in project

   Validations are performed in Controller classes

2. **Required Field validation**

   Case study



Required Fields

Fill out the form. Asterisk (*) means required.

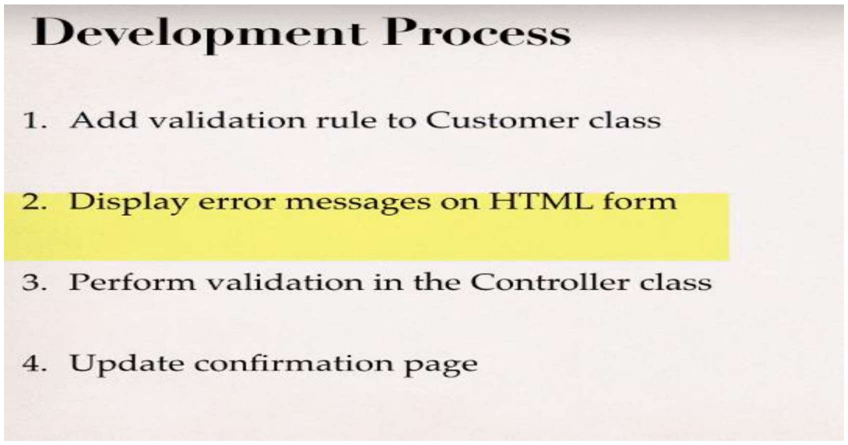
First name:

Last name (*):

Fill out the form. Asterisk (*) means required.

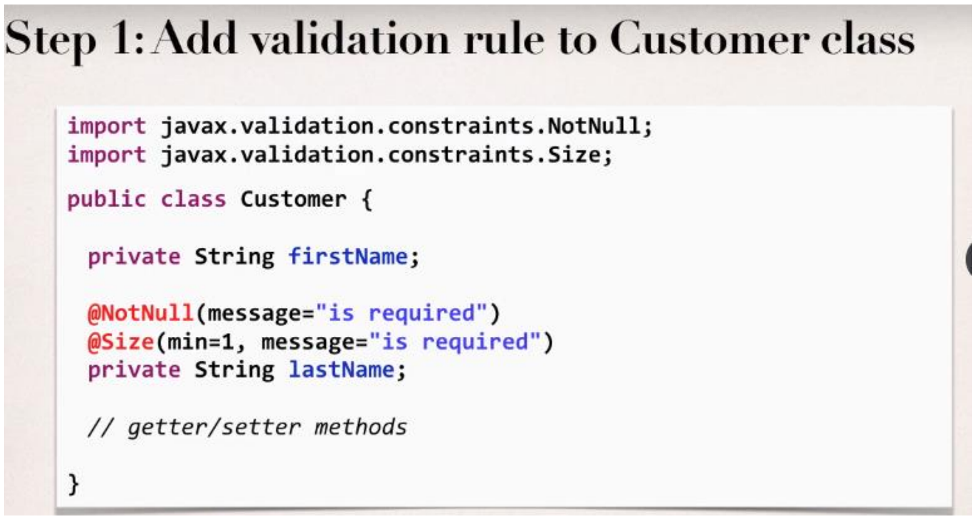
First name:

Last name (*): is required



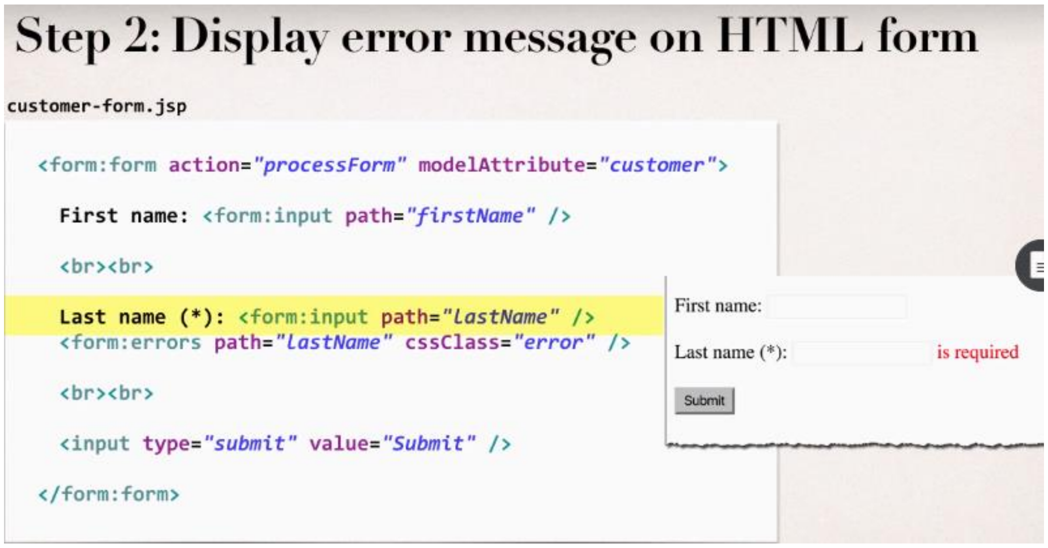
Development Process

1. Add validation rule to Customer class
2. Display error messages on HTML form
3. Perform validation in the Controller class
4. Update confirmation page



Step 1: Add validation rule to Customer class

```
import javax.validation.constraints.NotNull;
import javax.validation.constraints.Size;

public class Customer {

  private String firstName;

  @NotNull(message="is required")
  @Size(min=1, message="is required")
  private String lastName;

  // getter/setter methods

}
```



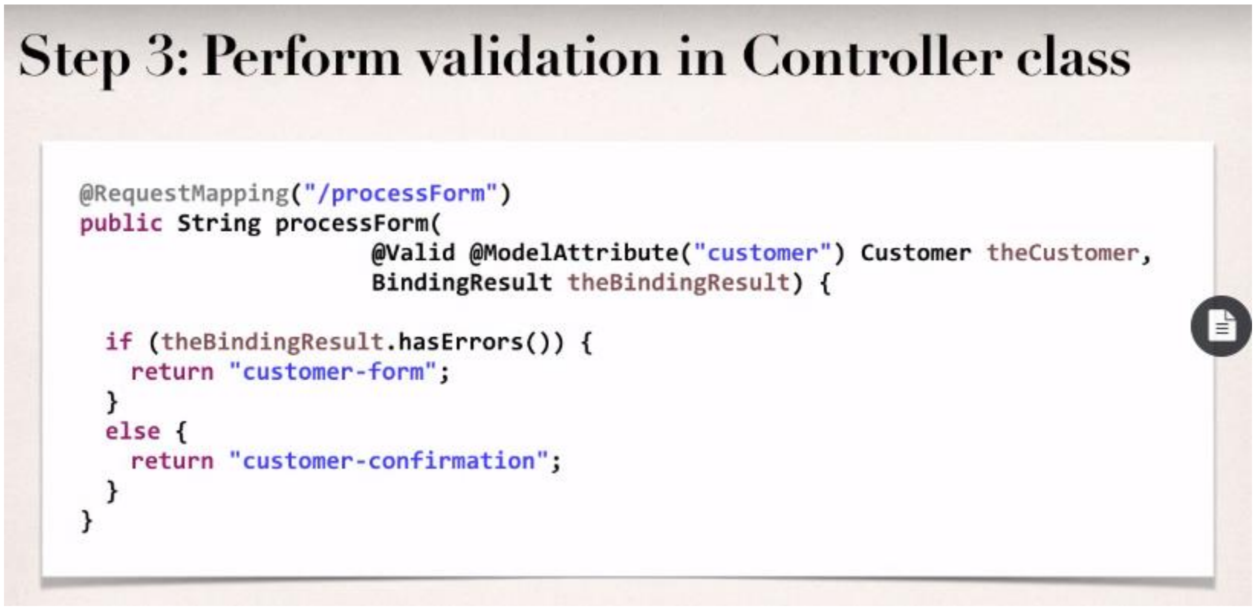
Step 2: Display error message on HTML form

customer-form.jsp

```
<form:form action="processForm" modelAttribute="customer">

  First name: <form:input path="firstName" />

  <br><br>

  Last name (*): <form:input path="lastName" />
  <form:errors path="lastName" cssClass="error" />

  <br><br>

  <input type="submit" value="Submit" />

</form:form>
```

First name:

Last name (*): is required



Step 3: Perform validation in Controller class

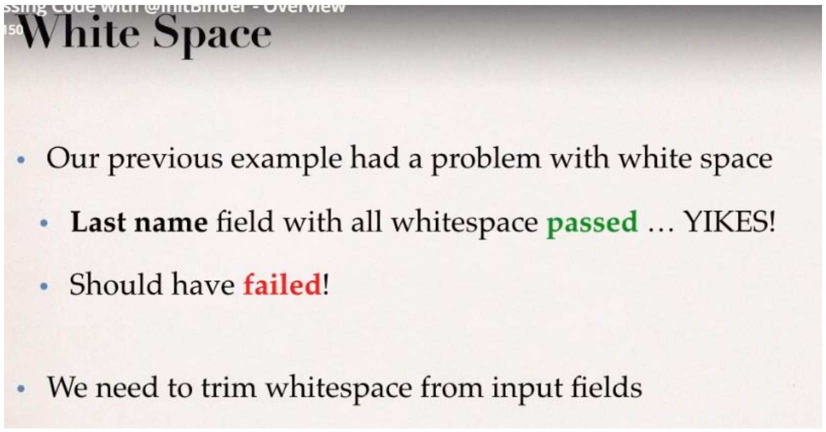
```
@RequestMapping("/processForm")
public String processForm(
                        @Valid @ModelAttribute("customer") Customer theCustomer,
                        BindingResult theBindingResult) {

  if (theBindingResult.hasErrors()) {
    return "customer-form";
  }
  else {
    return "customer-confirmation";
  }
}
```

Step4: view page

Note:- When performing Validations below things should be taken care of,

- Validation tag should be in bean class(customer.java)
- Input form jsp should use <form: error > tag
- Controller class will use two new attributes @Valid →it states that validation should be performed and BindingResult to store the validation result
- the BindingResult parameter must immediately after the model attribute.

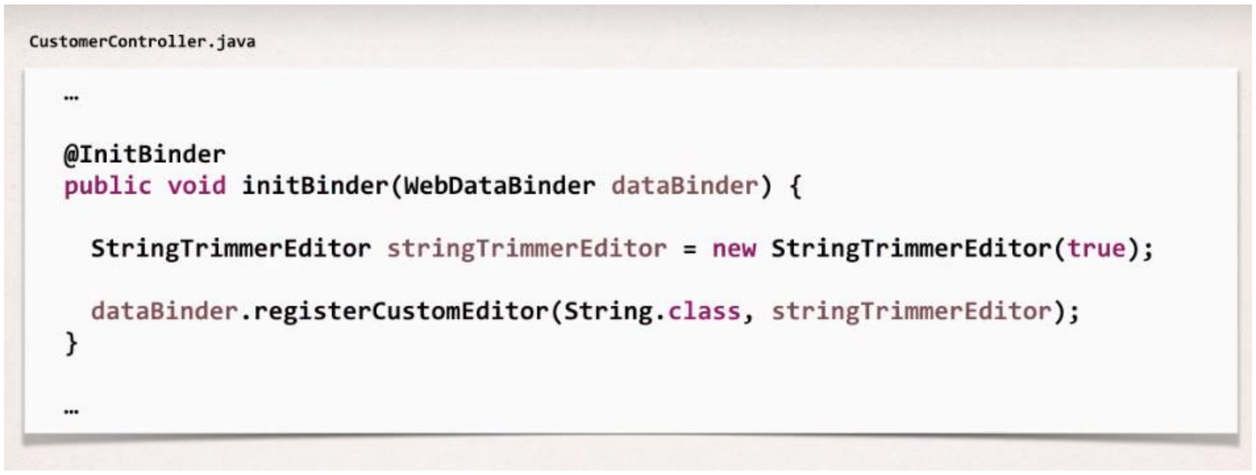
@InitBinder Annotation



White Space

- Our previous example had a problem with white space
  - **Last name** field with all whitespace **passed** … YIKES!
  - Should have **failed**!
- We need to trim whitespace from input fields

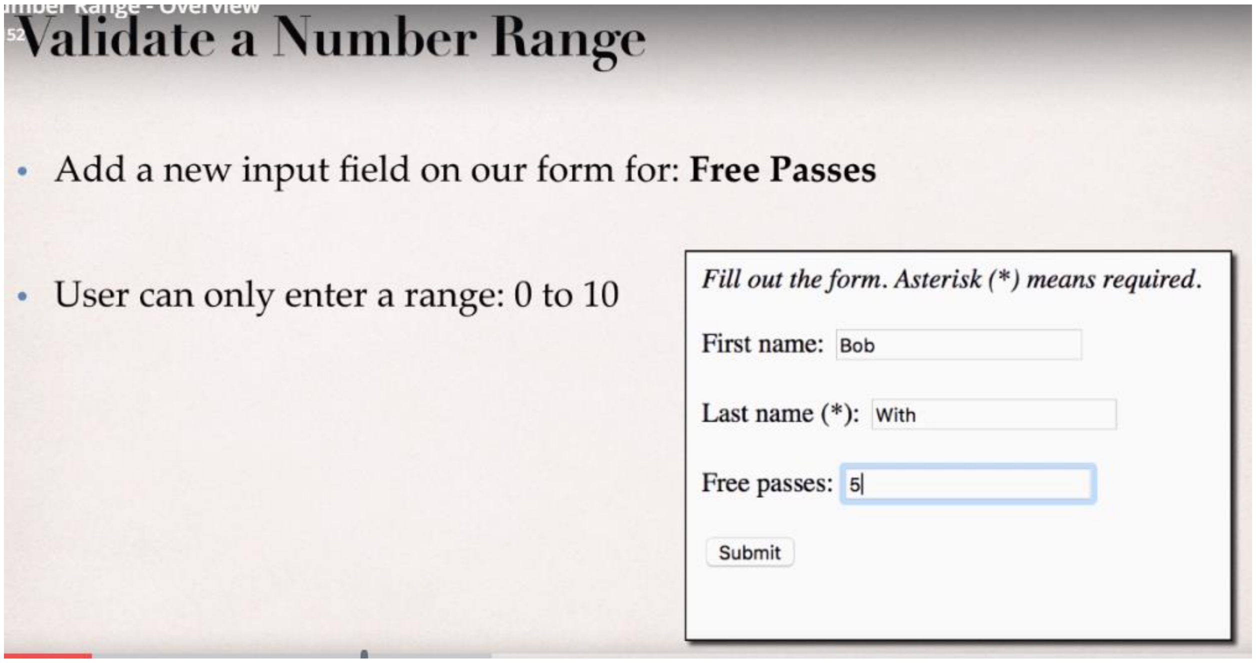
This annotation works as preprocessor



CustomerController.java

```
...

@InitBinder
public void initBinder(WebDataBinder dataBinder) {

  StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);

  dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);
}

...
```

Validating Number Ranges

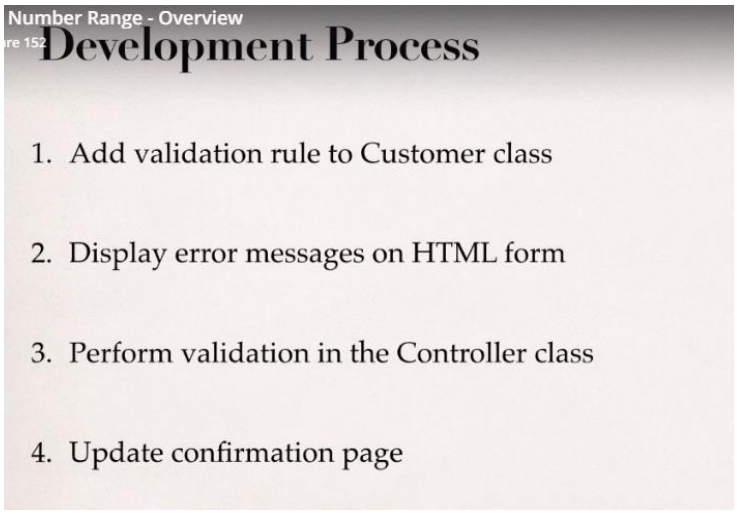


Validate a Number Range

- Add a new input field on our form for: **Free Passes**
- User can only enter a range: 0 to 10

Fill out the form. Asterisk (*) means required.

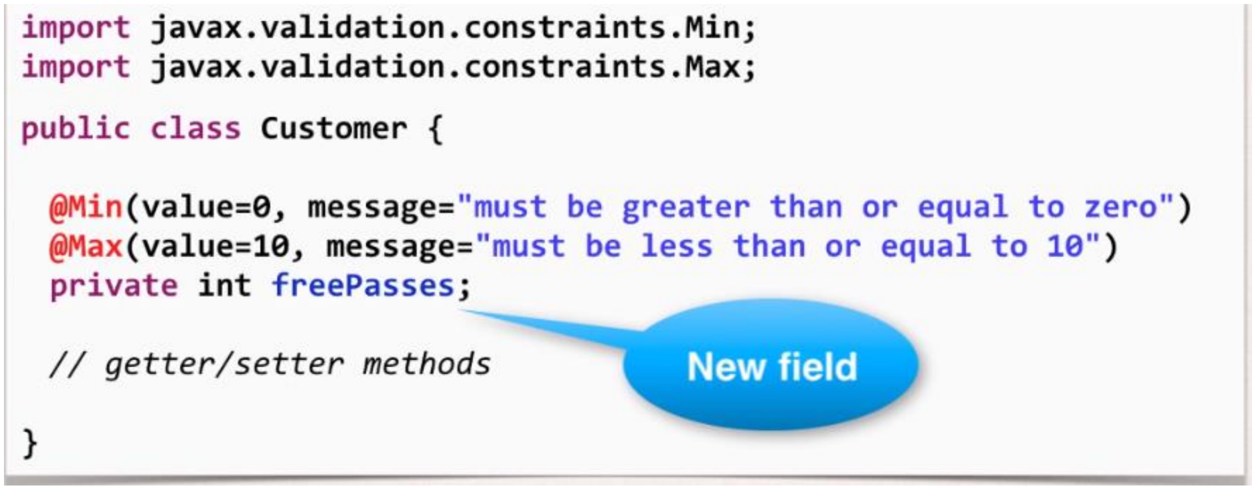
First name:	Bob
Last name (*):	With
Free passes:	5


Number Range - Overview

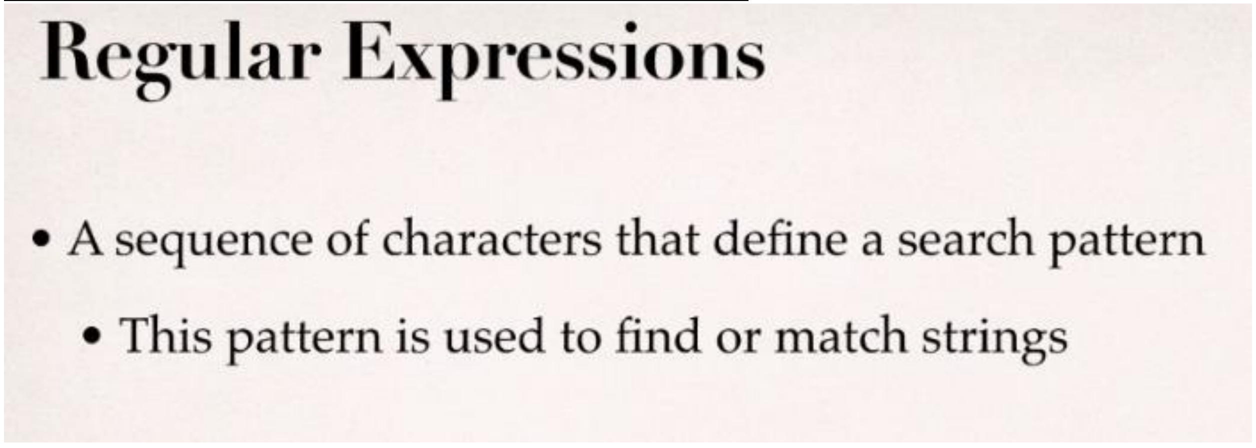
re 152

Development Process

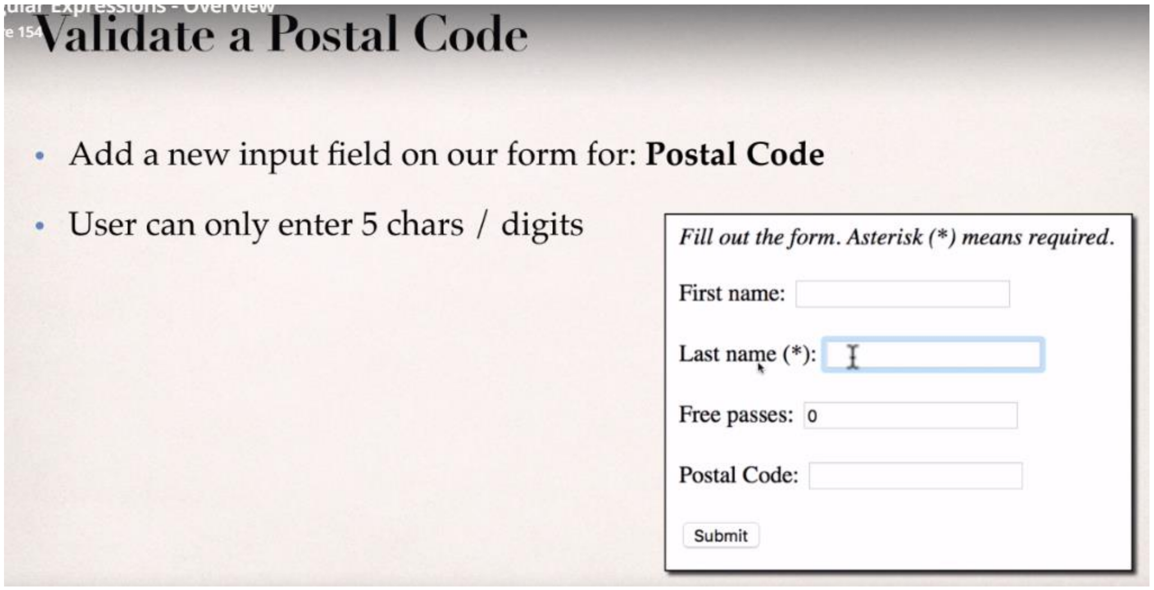
1. Add validation rule to Customer class
2. Display error messages on HTML form
3. Perform validation in the Controller class
4. Update confirmation page


```java
import javax.validation.constraints.Min;
import javax.validation.constraints.Max;

public class Customer {

  @Min(value=0, message="must be greater than or equal to zero")
  @Max(value=10, message="must be less than or equal to 10")
  private int freePasses;

  // getter/setter methods

}
```

Spring MVC Validations with Regular Expression


Regular Expressions

- A sequence of characters that define a search pattern
  - This pattern is used to find or match strings



Validate a Postal Code

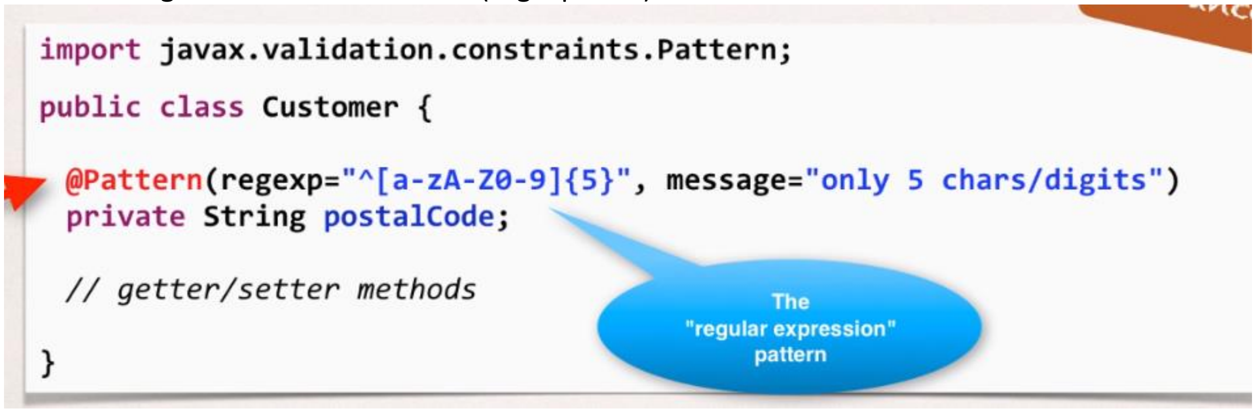
- Add a new input field on our form for: **Postal Code**
- User can only enter 5 chars / digits

Fill out the form. Asterisk (*) means required.

First name:	
Last name (*):	
Free passes:	0
Postal Code:	

Submit

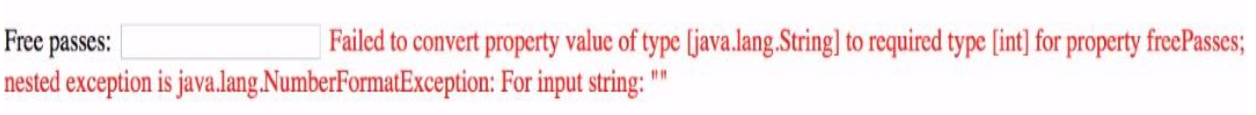
The Main tag to be used is @Pattern(regexp="…")



```
import javax.validation.constraints.Pattern;

public class Customer {

  @Pattern(regexp="^[a-zA-Z0-9]{5}", message="only 5 chars/digits")
  private String postalCode;

  // getter/setter methods

}
```

How to Make an Integer field required

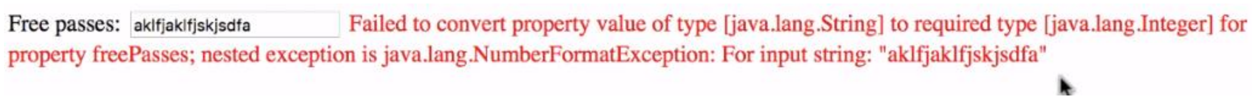
If we use @NotNull directly on the primitive types we will below error message



Free passes: Failed to convert property value of type [java.lang.String] to required type [int] for property freePasses; nested exception is java.lang.NumberFormatException: For input string: ""

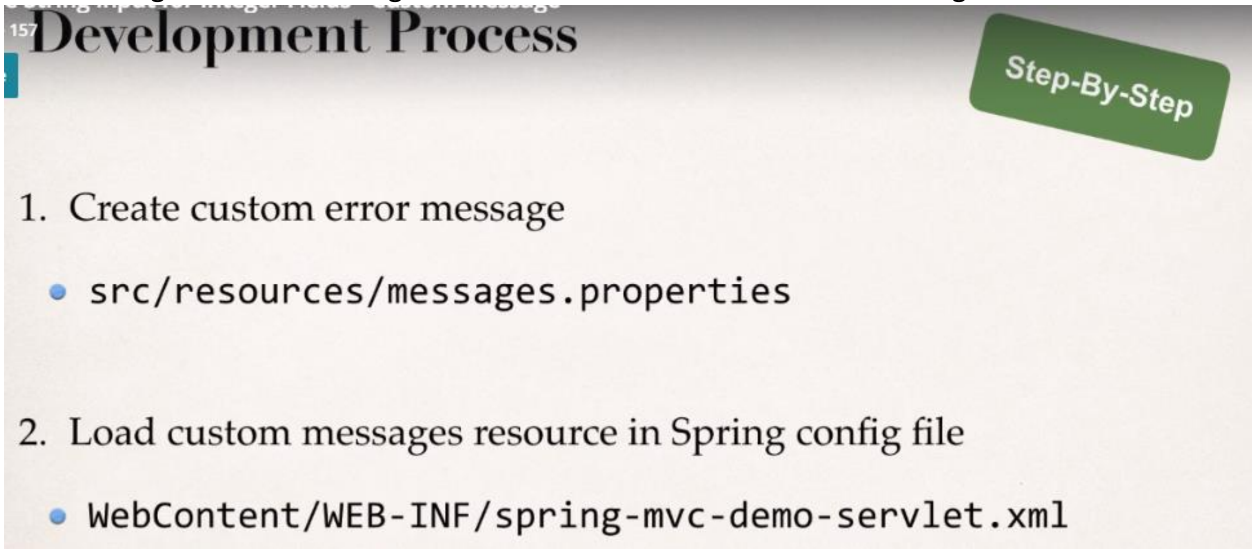
In order to resolve this we should use the wrapper class like Integer.

With the above approach if we add String to free passes it will again fail with NumberFormatException



Free passes: aklfjaklfjskjsdfa Failed to convert property value of type [java.lang.String] to required type [java.lang.Integer] for property freePasses; nested exception is java.lang.NumberFormatException: For input string: "aklfjaklfjskjsdfa"

In order to get the above thing done we need to add custom error message

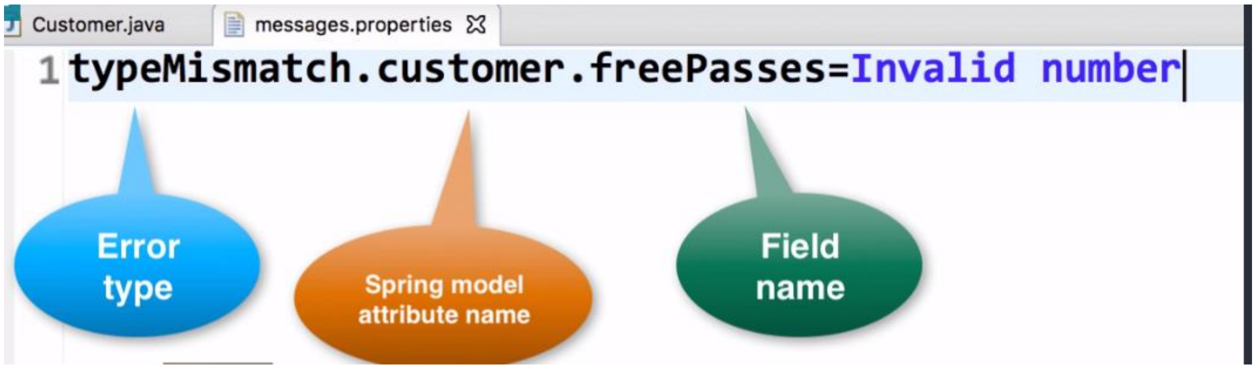


Development Process

Step-By-Step

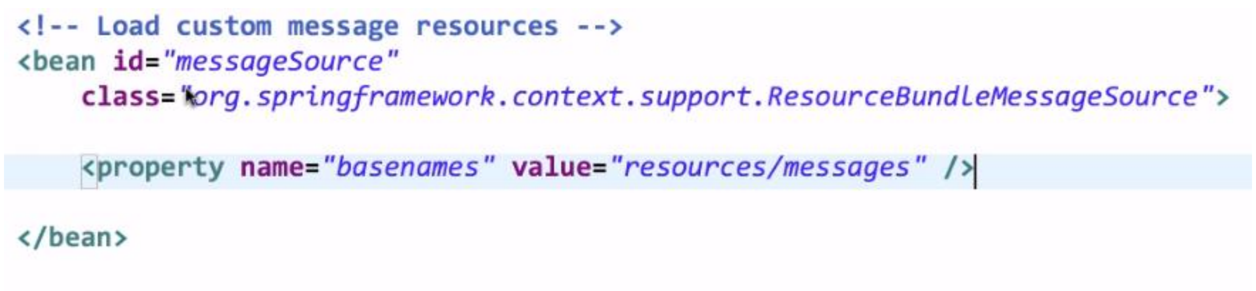
1. Create custom error message
   - src/resources/messages.properties

2. Load custom messages resource in Spring config file
   - WebContent/WEB-INF/spring-mvc-demo-servlet.xml



```
typeMismatch.customer.freePasses=Invalid number
```

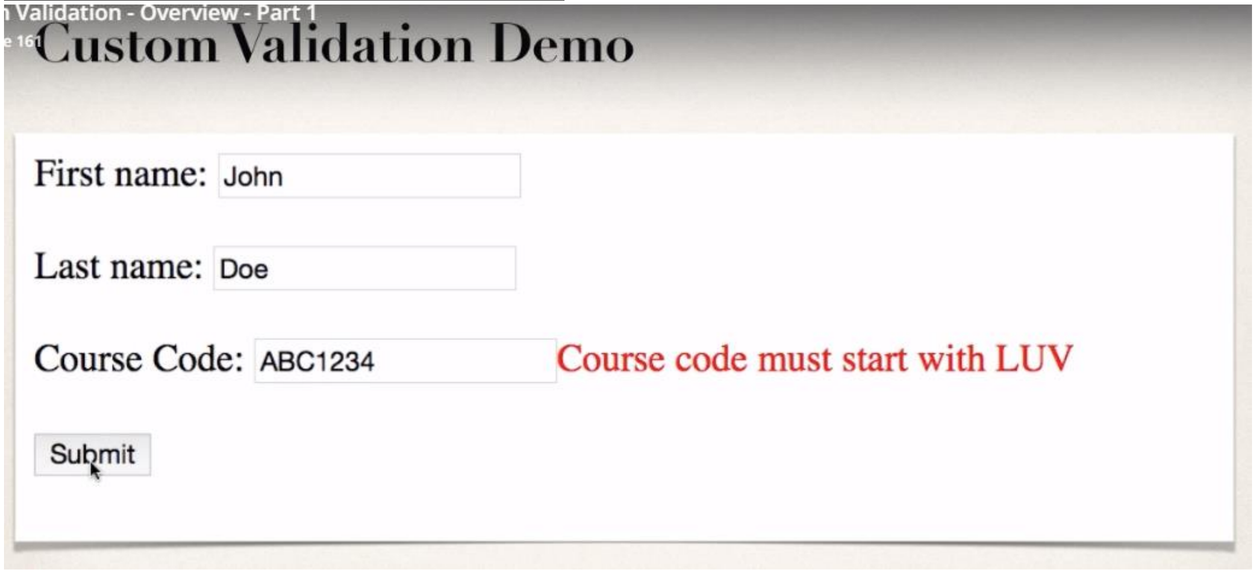
Exact same code should be used



```
<!-- Load custom message resources -->
<bean id="messageSource"
    class="org.springframework.context.support.ResourceBundleMessageSource">

    <property name="basenames" value="resources/messages" />

</bean>
```

The Data for messages.properties can be obtained by printing BindingResult

Spring MVC Custom Validations


Validation - Overview - Part 1

Custom Validation Demo

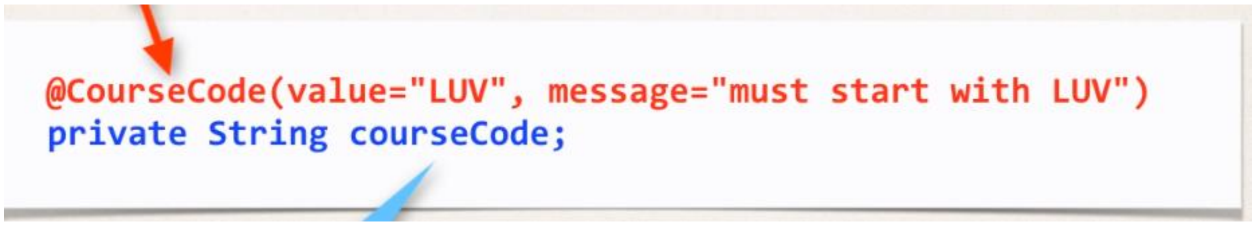
First name: John

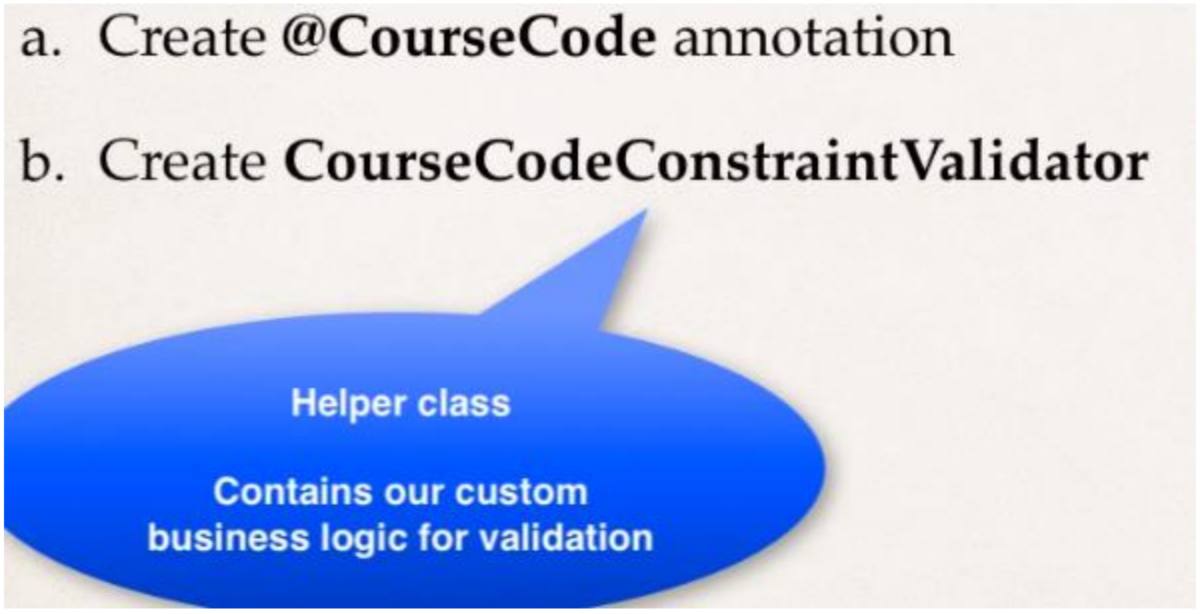
Last name: Doe

Course Code: ABC1234 Course code must start with LUV

Custom Validations should return boolean value

Creating our own Java Annotation eg:@CourseCode

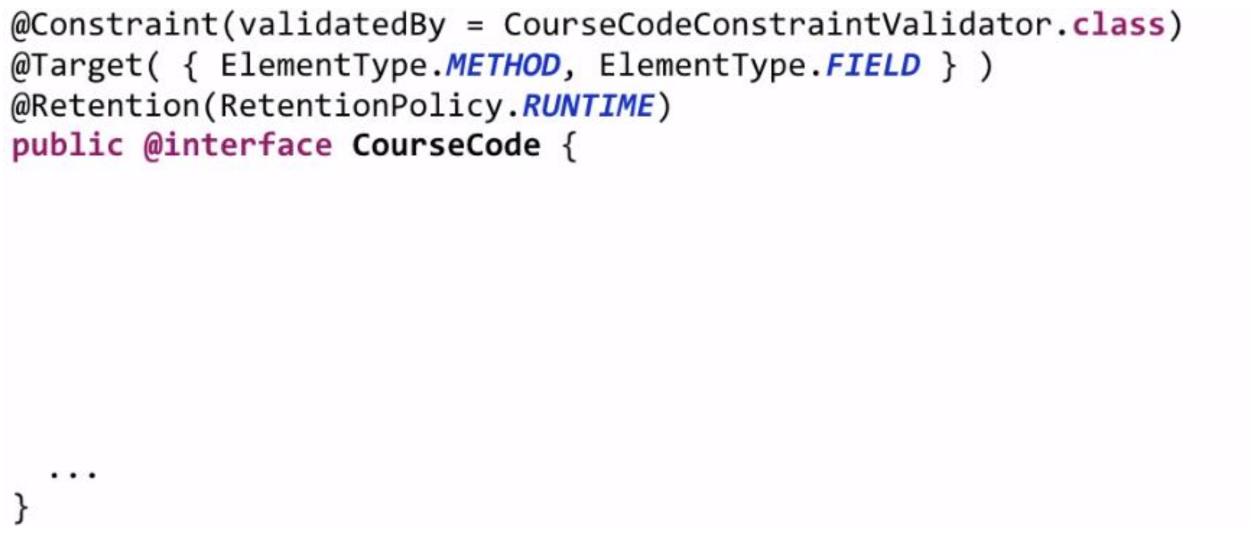

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```


a. Create **@CourseCode** annotation

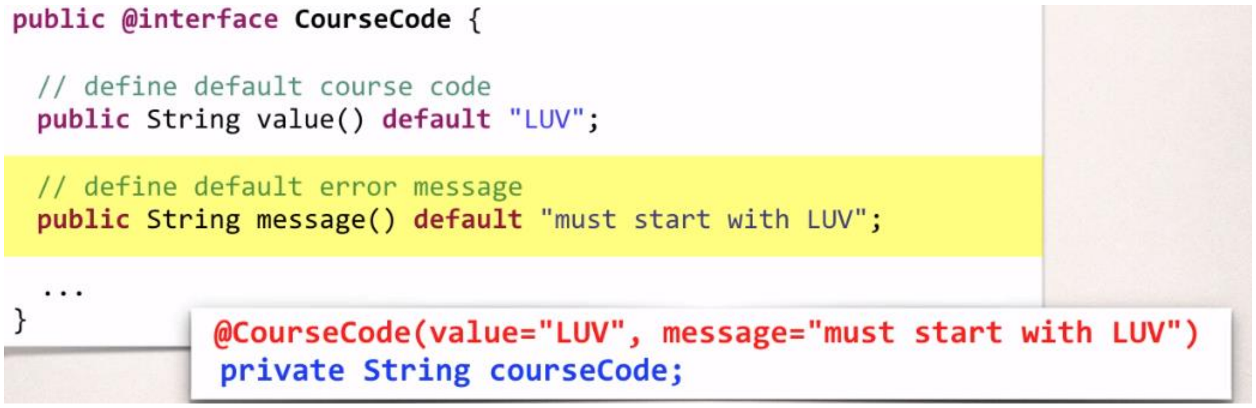
b. Create **CourseCodeConstraintValidator**

Helper class
Contains our custom business logic for validation

Creating annotation

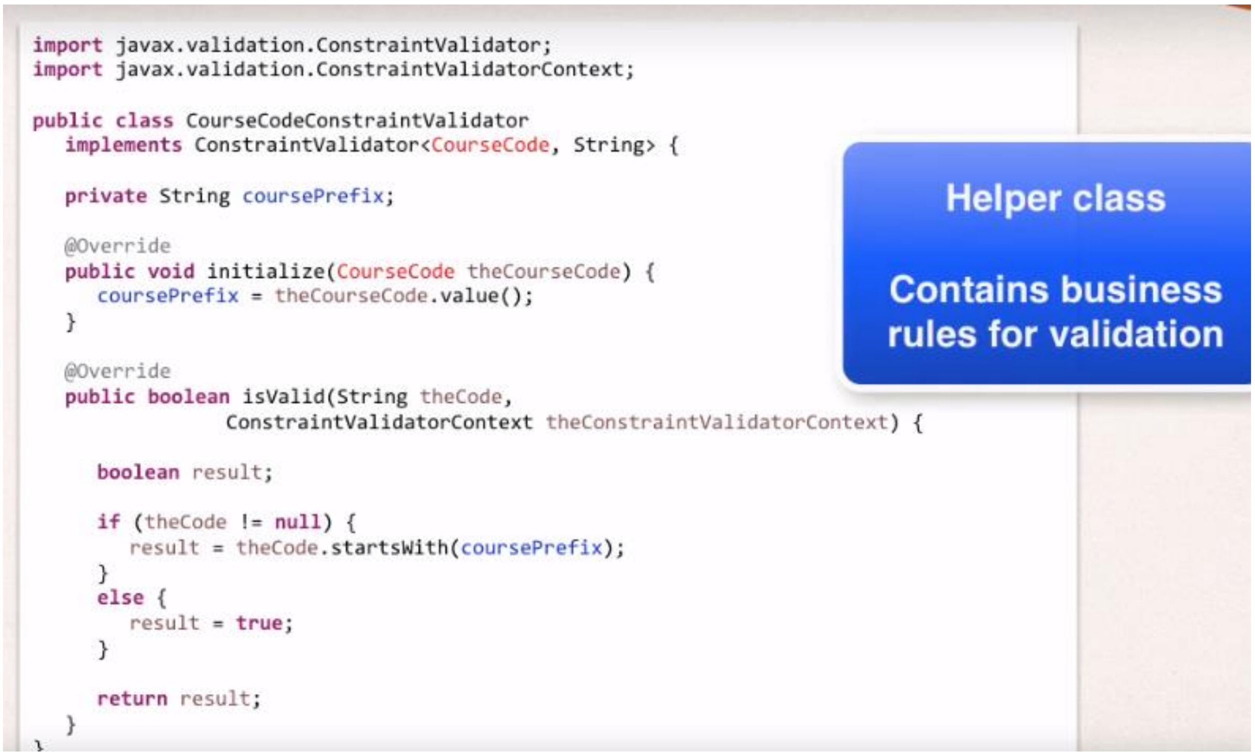


```
@Constraint(validatedBy = CourseCodeConstraintValidator.class)
@Target( { ElementType.METHOD, ElementType.FIELD } )
@Retention(RetentionPolicy.RUNTIME)
public @interface CourseCode {

  ...
}
```



```
public @interface CourseCode {

  // define default course code
  public String value() default "LUV";

  // define default error message
  public String message() default "must start with LUV";

  ...
}
```

```
@CourseCode(value="LUV", message="must start with LUV")
private String courseCode;
```



```
import javax.validation.ConstraintValidator;
import javax.validation.ConstraintValidatorContext;

public class CourseCodeConstraintValidator
   implements ConstraintValidator<CourseCode, String> {

   private String coursePrefix;

   @Override
   public void initialize(CourseCode theCourseCode) {
      coursePrefix = theCourseCode.value();
   }

   @Override
   public boolean isValid(String theCode,
                  ConstraintValidatorContext theConstraintValidatorContext) {

      boolean result;

      if (theCode != null) {
         result = theCode.startsWith(coursePrefix);
      }
      else {
         result = true;
      }

      return result;
   }
```