**Spring**

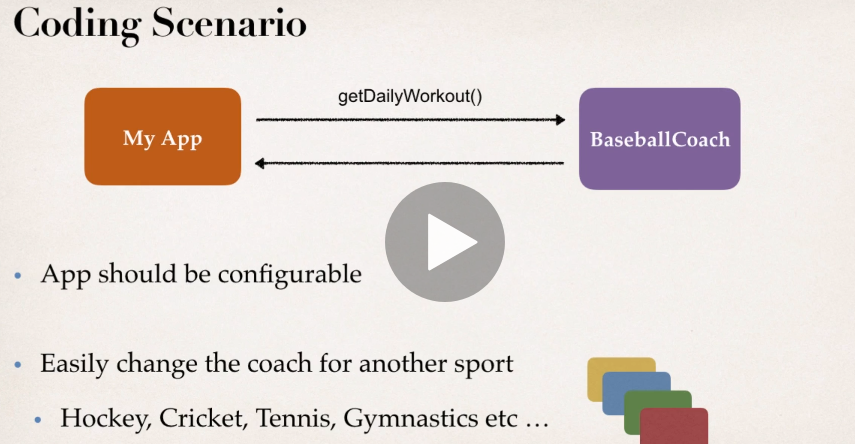
**Spring Inversion of Control**

**The approach of outsourcing the construction and management of objects.**

**The outsourcing will be handled by an Object factory.**

**Spring Inversion of Control - XML Configuration**

**Coding scenario**



**Coach.java**

****

**FootballCoach.java**

****

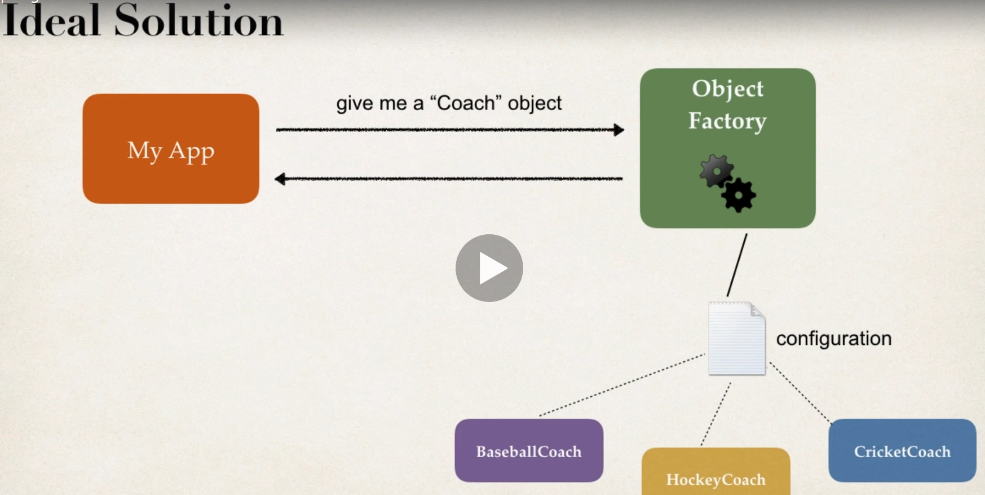
**BaseballCoach.java**

****

**MyApp.java**

****

**The above app is now able to change coach easily but the code is hardcoded. That means if we need to change the coach, we need to change the source code. Thus, it is not configurable.**

****

**Spring Container**

* **Primary functions** 
  + **Create and manage objects (Inversion of Control)**
  + **Inject object’s dependencies (Dependency Injection)**

**Configuring Spring Container**

* XML configuration file (*legacy, but most legacy apps still use this)*
* Java annotations (*modern*)
* Java source code (*modern)*

**Spring development process**

1. Configure spring beans
2. Create a spring container
3. Retrieve beans from Spring Container

NOTE  
FAQ: What is a Spring Bean?

**FAQ: What is a Spring Bean?**

A "Spring Bean" is simply a Java object.

When Java objects are created by the Spring Container, then Spring refers to them as "Spring Beans".

Spring Beans are created from normal Java classes .... just like Java objects.

---

Here's a blurb from the Spring Reference Manual



Source: <https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-introduction>

---

In the early days, there was a term called "Java Beans". Spring Beans have a similar concept but Spring Beans do not follow all of the rigorous requirements of Java Beans.

---

***In summary, whenever you see "Spring Bean", just think Java object. :-)***

**Configure spring beans**

**ApplicationContext.xml**

* Define Bean ID.
* Class, the one which we want.



**MySpringApp.java**



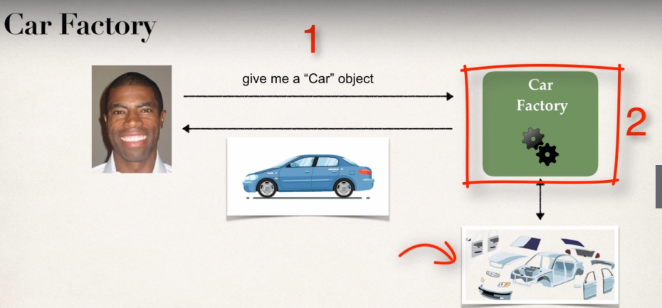
myCoach is the Bean ID we used in ApplicationContext.xml and the Coach.class specifies the interface we created earlier.

**Dependency Injection (with XML configuration)**

The client delegates to calls to another object the responsibility of providing its dependencies.

“Dependency” same thing as “helper objects”

**Explanation**

Say for example, I'm going to buy a car and this car is built at the factory on demand. So, there's nothing in the car lot. You have to actually talk to the factory and put in a request and they'll build a car for you; So, at the factory, you have all the different parts for the car. You have the car chassis, you have the engine, the tires, the seats, the electronics, the exhaust, and so on. And the mechanics or the assemblers there or the technicians, they'll actually assemble the car for you and then deliver to you the final car. So, you don't have to actually build the car. The cars already built for you at the factory. So, they actually inject all of the dependencies for the car. So, they inject the engine, they inject the tires, the seats and so on. So that's basically what you have here with dependency injection. So, you simply outsource the construction and injection of your object to an external entity. In this case, that's the car factory.

**Injection Types**

1. Constructor Injection
2. Setter Injection

Although, there are many types of injection with Spring

**Development Process – Constructor Injection**

1. Define the dependency interface and class
2. Create a constructor in your class for injections
3. Configure the dependency injection in Spring config file
4. **Define the dependency interface and class**
   1. **Dependency Interface**

****

* 1. **Class**



1. **Create a constructor in your class for injections**

****

1. **Configure the dependency injection in Spring config file**

****

**EXPLANTAION**

At first, we create the **dependency interface** and define the **method in a class.**

In the next step, we create a private instance of the interface in the class where we want to inject the dependency. After creating the instance, we call for the constructor to load the dependency into the earlier instance of the interface by passing it via an argument (here, theFortuneService).

Once the dependency is loaded, we call the method through it.

**NOTE –**

**If you want to add another dependency in the same project, make sure you call one constructor and define it for the same class. DO NOT DEFINE SEPARATE CONSTRUCTORS FOR THE SAME CLASS.**

****

**Setter Injection**

Inject dependencies by calling setter methods on our class.

Development Process

1. Create setter methods in our class
2. Configure the dependency injection in spring config file
3. Create setter methods in our class

Before creating the setter methods, we need to create the private instances of the dependencies Fortune Service and Salary Services (as a whole). Also, we need to call the no-arg constructor so just that we know we are inside the constructor to know the behind scenes (However, this part is completely optional).

After the above methods, we need to create setter methods. And then we will call the setter methods by passing a parameter in order to load the setter method. After the loading has been done, we will actually call the method through it.

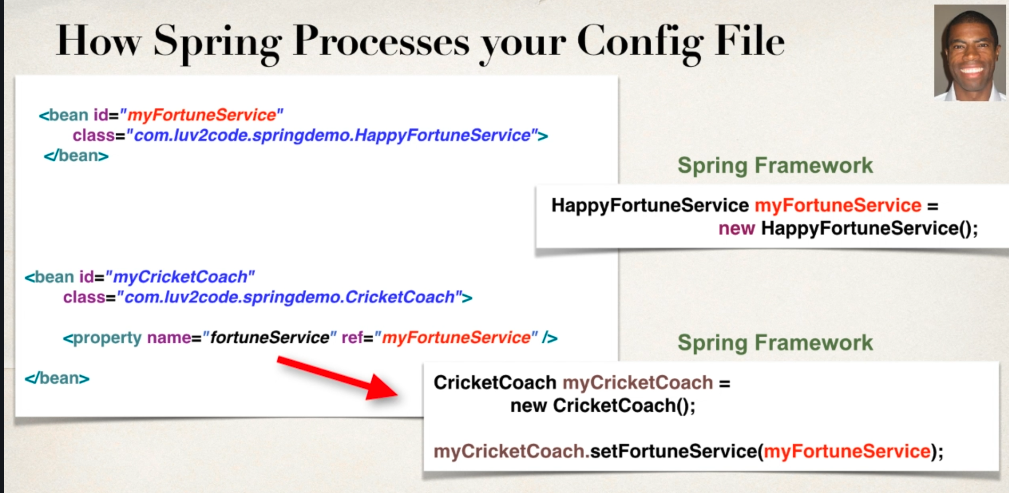


1. **Configure the dependency injection in spring config file**



The ref is used in the property is used for referencing objects or dependencies.

**BEHIND THE SCENES**



**Injecting literal values**

* **Create Setter methods in our class for injections**
* **Configure the injection in spring configure file.**

**Creating setter methods ( create private fields and then call setter methods)**



****

**Configuring the injection in config file.**

****

**Note -** The value tag is used for literal values.

**Injecting values from a properties file**

We injected literal values but those literal values were hard coded. What we want to do is to inject those values from an external properties file.

**Development process**

1. Create Properties file
2. Load the properties file in Spring config file
3. Reference values from properties file

**Creating Properties file**

**Sport.properties**

****

**2. Load the properties file in spring config file**

**3. Reference values from properties file**

****

**Bean Scope**

* Scope refers to the lifecycle of a bean.
* Duration of a bean.
* No of instances created
* How the bean is shared?

Default scope of Bean: **Singleton**

**What is Singleton?**

* Spring Container creates only one instance of the bean, by default
* Cached in memory
* All requests for the bean
  + Will return a **SHARED REFERENCE** to the **SAME BEAN**

**Additional Spring Bean Scopes**

|  |  |
| --- | --- |
| Scope | Description |
| Singleton | **Create a single shared instance of the bean. Default scope** |
| prototype | **Creates a new bean instance for each container request** |
| request | **Scoped to an HTTP web request. Only used for web apps** |
| session | **Scoped to an HTTP web session. Only used for web apps.** |
| global-session | **Scoped to a global HTTP web session. Only used for web apps.** |

**Code to check**

****

****

**BEAN LIFECYCLE**

**Container Started**  🡪 Beans Instantiated 🡪 Dependencies Injected 🡪 Internal Spring Processing 🡪 Your Custom Init Method 🡪 Bean is ready to use | Container Is Shut down 🡪 Your Custom Destroy Method 🡪 STOP

**Bean Lifecycle Methods / Hooks**

* You can add custom code during bean initialization
  + Calling custom business logic methods
  + Setting up handles to resources (db, socket, file etc.)
* You can add custom code during bean destruction
  + Calling custom business logic method
  + Clean up handles to resources (db, sockets, files, etc.)

**Development process**

* Define your methods for init and destroy
* Configure the method names in Spring config file.

**Note: Defining init and destroy methods - Method Signatures**

**Special Note about init and destroy Method Signatures**

When using XML configuration, I want to provide additional details regarding the method signatures of the init-method and destroy-method.

**Access modifier**  
The method can have any access modifier (public, protected, private)

**Return type**  
The method can have any return type. However, "void' is most commonly used. If you give a return type just note that you will not be able to capture the return value. As a result, "void" is commonly used.

**Method name**  
The method can have any method name.

**Arguments**  
The method cannot accept any arguments. The method should be no-arg.

Files used: **BaseballCoach.java**



**beanLifeCycle-applicationContext.xml**

****

**beanLifeCycleDemoApp.java**



**Java Annotations**

* Special labels/markers added to Java classes
* Provide meta-data about the class
* Processed at compile time or run time for special processing

**Why Spring configuration with Annotation?**

* XML Configuration can be verbose
* Configure your spring beans with annotations
* Annotations minimizes the XML configuration

**Scanning for Component Classes**

Once we add annotations to a class,

* Spring will scan your java classes for special annotations.
* Automatically register the beans in the Spring container

**Development Process**.

* Enable component scanning in Spring Config file
* Add the @Component Annotation to our java classes
* Retrieve bean from Spring Container.

**SPRING CONFIGURATION WITH JAVA ANNOTATIONS – INVERSION OF CONTROL (IOC)**

**Enable component scanning in Spring Config file**

**applicationContext.xml**

Base package should be the package name.



**Coach.java**

****

**Add the @Component Annotation to our java classes**

**TennisCoach.java**

Syntax: @component(“bean-id”)

****

**Retrieve bean from Spring Container.**

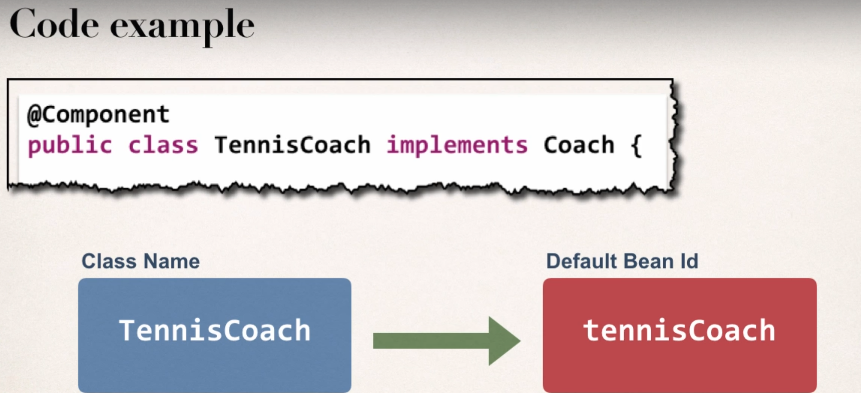
**AnnotationsDemoApp.java (Driver Application)**

****

**Spring also supports Default Bean IDs**

* **Default bean id:**  Class Name, *make first letter lower-case*

|  |  |
| --- | --- |
| Class Name | Default Bean ID |
| **TennisCoach** | **tennisCoach** |





**SPRING CONFIGURATION WITH JAVA ANNOTATIONS – DEPENDENCY INJECTION (DI)**

* For Dependency Injection, Spring can use auto-wiring
* Spring will look for a class that matches the property
  + Matches by type: class or interface
* Spring will inject it automatically. Hence it is autowired.

**Autowiring Example**

* Injecting FortuneService into a Coach implementation
* Spring will scan @Components
* Any one implements FortuneService interface??
* If so, let’s inject them. For, example: HappyFortuneService

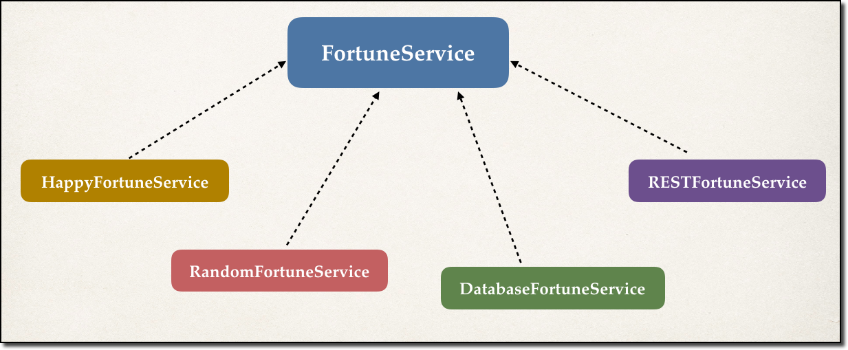
**Autowiring Injection Types**

* Constructor Injection
* Setter Injection
* Field Injection

**Development Process - Constructor Injection**

1. Define the dependency interface and class
2. Create a constructor in our class for injection
3. Configure the dependency injection with @Autowired annotation

**What if there are multiple FortuneService implementations?**

When using autowiring, what if there are multiple FortuneService implementations? Like in the image below?  
  


Spring has special support to handle this case. Use the **@Qualifier** annotation.

1. **Define the dependency interface and class**

Spring will scan for a component that implements FortuneService interface

Example: HappyFortuneService meets the requirement and the dependency is being injected into the TennisCoach automatically by using autowiring.

**FortuneService.java**

****

**HappyFortuneService.java**

****

1. **Create a constructor in our class for injection**
2. **Configure the dependency injection with @Autowired annotation**

So, the component searches for a Fortune Service class or interface that matches the type and once it founds its match, it will automatically wire the components.

****

**Autowired (OPTIONAL)**

**SETTER INJECTION WITH ANNOTATIONS**

**Autowiring Example**

* Injecting FortuneService into a Coach implementation
* Spring will scan @Components
* Any one implements FortuneService interface??

If so, let’s inject them. For, example: HappyFortuneService

**Development Process – Setter Injection**

1. Create setter method(s) in our class for injections.
2. Configure the dependency injection with @Autowired annotation.
3. **Create setter method(s) in our class for injections**

****

****

**Method Injection**

**Inject dependencies by calling ANY method on our class by simply giving: @Autowired**

Instead of using setter method, we can use any method.

**GIVE ANY CUSTOM METHOD NAME**

****

**FIELD INJECTION**

**Inject dependencies by setting field values on our class directly (even private fields) [Accomplished by using JAVA Reflection]**

**Development Process**

* Configure the dependency injection with Autowired Annotation
  + Applied directly to the field
  + No need for setter methods

**Note -**

**Reflection in Java**

Reflection is an API which is used to examine or modify the behavior of methods, classes, interfaces at runtime.

* The required classes for reflection are provided under java.lang.reflect package.
* Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object.
* Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

**Link**: <https://www.geeksforgeeks.org/reflection-in-java/>



****

**Which Injection type should we use?**

Chose a style and stay consistent in the project.

**Qualifiers for Dependency Injection**

There arose a problem, when there were multiple implementations of an interface. How would Spring understand which one to choose? And there comes an error caused by “NoUniqueBeanDefinitionFound”.

That’s when “@Qualifier” annotation comes into play.

“@Qualifier” annotation can be applied to

* Constructor Injection
* Setter Injection methods
* Field Injection.

Use the annotation @Qualifier with the desired implementation with a bean ID (preferably defaultBeanID).

This below interface has several implementations.

****



**DatabaseFortuneService.java**



**RESTFortuneService.java**



**RandomFortuneService.java**



The class file from where spring will choose which one to pick from the multiple implementations. So, spring picks up the one with “@Qualifier” annotation.



Annotations - Default Bean Names - The Special Case

**Annotations - Default Bean Names ... and the Special Case**

In general, when using Annotations, for the default bean name, Spring uses the following rule.

If the annotation's value doesn't indicate a bean name, an appropriate name will be built based on the short name of the class (with the first letter lower-cased).

For example:

HappyFortuneService --> happyFortuneService

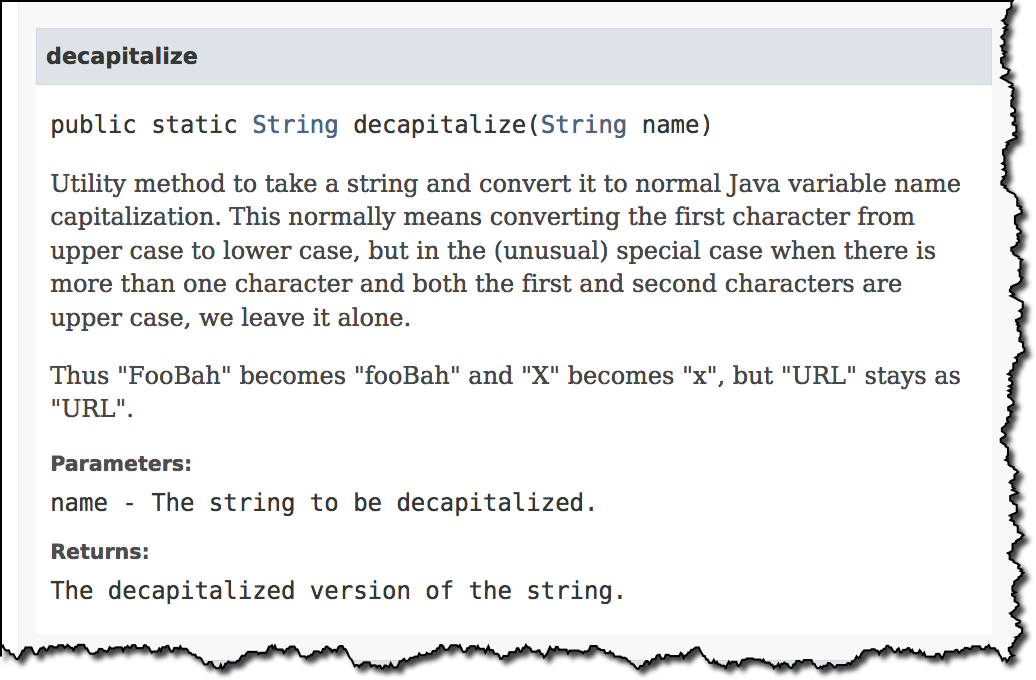
However, for the **special case of when BOTH the first and second characters of the class name are upper case**, then the **name is NOT converted**.

For the case of RESTFortuneService

RESTFortuneService --> RESTFortuneService

No conversion since the first two characters are upper case.

Behind the scenes, Spring uses the **Java Beans Introspector** to generate the default bean name. Here's a screenshot of the documentation for the key method.



Also, here's a link to the documentation.

<https://docs.oracle.com/javase/8/docs/api/java/beans/Introspector.html#decapitalize(java.lang.String)>

As always, you can give explicit names to your beans.



Using @Qualifier with Constructors

**@Qualifier** is a nice feature, but it is tricky when used with Constructors.

The syntax is much different from other examples and not exactly intuitive.  Consider this the "deep end of the pool" when it comes to Spring configuration LOL :-)

 You have to place the @Qualifier annotation inside of the constructor arguments.

Here's an example from our classroom example. I updated it to make use of constructor injection, with @Autowired and @Qualifier. Make note of the code in bold below:



For detailed documentation on using @Qualified with Constructors, see this link in the Spring Reference Manual

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-autowired-annotation-qualifiers>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**FAQ: How to inject properties file using Java annotations**

**Answer:**

This solution will show you how inject values from a properties file using annotatons. The values will no longer be hard coded in the Java code.

**1. Create a properties file to hold your properties. It will be a name value pair.**

New text file:  src/sport.properties

1. foo.email=myeasycoach@luv2code.com
2. foo.team=Silly Java Coders

Note the location of the properties file is very important. It must be stored in src/sport.properties

**2. Load the properties file in the XML config file.**

File: applicationContext.xml

Add the following lines:

    <context:property-placeholder location="classpath:sport.properties"/>

This should appear just after the <context:component-scan .../> line

1. **Inject the properties values into your Swim Coach:**
   * **Entire Code -** <http://www.luv2code.com/downloads/spring-hibernate/spring-props-annotation-demo.zip>

**BEAN SCOPES AND LIFECYCLE METHODS WITH JAVA ANNOTATIONS**

**SCOPE**

Use the “@Scope” annotation to change the scope. Place it with the component. Default scope is always singleton.

**Bean LifeCycle Methods/Hooks**

* Add custom code during bean initialization
  + Calling custom business logic methods
  + Setting up handles to resources (db, sockets, file etc.)
* Add custom code during bean destruction
  + Calling custom business logic method.
  + Clean up handles to resources (db, sockets, files etc.)

Development Process

1. Define your methods
2. Add annotations: @PostConstruct and @PreDestroy

**Init: Method Configuration**

* Give any method name you want.
* Add the Annotation “@PostConstruct”. As the annotation name suggests, the code will execute *after constructor and after injection of dependencies*.

**Destroy: Method Configuration**

* Give any method name you want.
* Add the Annotation “@PreDestroy”. As the annotation name suggests, the code will execute *before the bean is destroyed*.

Special Note about @PostConstruct and @PreDestroy Method Signatures

**Access modifier**

The method can have any access modifier (public, protected, private)

**Return type**The method can have any return type. However, "void' is most commonly used. If you give a return type just note that you will not be able to capture the return value. As a result, "void" is commonly used.

**Method name**The method can have any method name.

**Arguments**The method cannot accept any arguments. The method should be no-arg.

**HEADS UP - FOR JAVA 9, 10 and 11 USERS - @PostConstruct and @PreDestroy**

If you are using Java 9 or higher, then you will encounter an error when using @PostConstruct and @PreDestroy in your code.

These are the steps to resolve it. Come back to the lecture if you hit the error.

**Error**

Eclipse is unable to import @PostConstruct or @PreDestroy

This happens because of Java 9 and higher.

When using Java 9 and higher, javax.annotation has been removed from its default classpath. That's why we Eclipse can't find it.

**Solution**

1. Download the javax.annotation-api-1.3.2.jar from <https://search.maven.org/remotecontent?filepath=javax/annotation/javax.annotation-api/1.3.2/javax.annotation-api-1.3.2.jar>
2. Copy the JAR file to the **lib** folder of your project
3. Use the following steps to add it to your Java Build Path.
4. Right-click your project, select **Properties**
5. On left-hand side, click **Java Build Path**
6. In top-center of dialog, click **Libraries**
7. Click **Class path** and then Click **Add JARs ...**
8. Navigate to the JAR file **<your-project>/lib/javax.annotation-api-1.3.2.jar**
9. Click **OK** then click **Apply and Close**
10. Eclipse will perform a rebuild of your project and it will resolve the related build errors.

Development Process

1. Define the methods
2. Add annotations



Run the driver app class to generate the output**.**

**Special Note about Destroy Lifecycle and Prototype Scope**

**For "prototype" scoped beans, Spring does not call the @PreDestroy method.**

Here is the answer from the Spring reference manual. Section 1.5.2

<https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-scopes-prototype>

**In contrast to the other scopes, Spring does not manage the complete lifecycle of aprototype bean**: the container instantiates, configures, and otherwise assembles aprototype object, and hands it to the client, with no further record of that prototypeinstance.

Thus, although initialization lifecycle callback methods are called on all objects regardless of scope, **in the case of prototypes, configured destruction lifecycle callbacks are not called**. The client code must clean up prototype-scoped objects and release expensive resources that the prototype bean(s) are holding.

To get the Spring container to release resources held by prototype-scoped beans, try using a custom [bean post-processor](https://docs.spring.io/spring/docs/current/spring-framework-reference/core.html#beans-factory-extension-bpp), which holds a reference to beans that need to be cleaned up.

This also applies to XML configuration.

**QUESTION: How can I create code to call the destroy method on prototype scope beans**

**ANSWER:**

You can destroy prototype beans but custom coding is required. This examples shows how to destroy prototype scoped beans.

1. Create a custom bean processor. This bean processor will keep track of prototype scoped beans. During shutdown it will call the destroy() method on the prototype scoped beans.

2. The prototype scoped beans MUST implement the DisposableBean interface. This interface defines a "destroy()" method. This method should be used instead of the @PreDestroy annotation.

3. In this app, BeanProcessorDemoApp.java is the main program. TennisCoach.java is the prototype scoped bean. TennisCoach implements the DisposableBean interface and provides the destroy() method. The custom bean processing is handled in the MyCustomBeanProcessor class.

See source code here for details: [**destroy-protoscope-bean-with-custom-processor.zip**](https://drive.google.com/open?id=11YZhzKjLJVhvsMsNjKU7jMrzTHlxwvkT)

**CONFIGURE SPRING CONTAINER USING JAVA CODE**

* Instead of configuring Spring Container using XML
* Configure the Spring Container with Java code

**Development Process**

1. Create a java class and annotate as @Configuration
2. Add component scanning support: @ComponentScan **(optional)**
3. Read Spring Java Configuration class
4. Retrieve bean from Spring Container

**1,2: Creating java class and annotating it as @Configuration; Adding Component Scanning support**

**SportConfig.java**

****

**Driver Class**

****

**Defining Beans with Java Code**

1. Define method to expose bean
2. Inject bean dependencies
3. Read Spring Java Configuration
4. Retrieve bean from Spring Container

**1 & 2**



During All Java Configuration, how does the @Bean annotation work in the background?

**Answer** - This is an advanced concept. But I'll walk through the code line-by-line.

For this code:

1. @Bean
2. public Coach swimCoach() {
3. SwimCoach mySwimCoach = new SwimCoach();
4. return mySwimCoach;
5. }

At a high-level, Spring creates a bean component manually. By default the scope is singleton. So any request for a "swimCoach" bean, will get the same instance of the bean since singleton is the default scope.

However, let's break it down line-by-line

1. @Bean

The @Bean annotation tells Spring that we are creating a bean component manually. We didn't specify a scope so the default scope is singleton.

1. public Coach swimCoach(){

This specifies that the bean will bean id of "swimCoach". The method name determines the bean id. The return type is the Coach interface. This is useful for dependency injection. This can help Spring find any dependencies that implement the Coach interface.

The @Bean annotation will intercept any requests for "swimCoach" bean. Since we didn't specify a scope, the bean scope is singleton. As a result, it will give the same instance of the bean for any requests.

1. SwimCoach mySwimCoach = new SwimCoach();

This code will create a new instance of the SwimCoach.

1. return mySwimCoach;

This code returns an instance of the swimCoach.

Now let's step back and look at the method in it's entirety.

1. @Bean
2. public Coach swimCoach() {
3. SwimCoach mySwimCoach = new SwimCoach();
4. return mySwimCoach;
5. }

It is important to note that this method has the @Bean annotation. The annotation will intercept ALL calls to the method "swimCoach()". Since no scope is specified the @Bean annotation uses singleton scope. Behind the scenes, during the @Bean interception, it will check in memory of the Spring container (applicationContext) and see if this given bean has already been created.

If this is the first time the bean has been created then it will execute the method as normal. It will also register the bean in the application context. So that is knows that the bean has already been created before. Effectively setting a flag.

The next time this method is called, the @Bean annotation will check in memory of the Spring container (applicationContext) and see if this given bean has already been created. Since the bean has already been created (previous paragraph) then it will immediately return the instance from memory. It will not execute the code inside of the method. Hence this is a singleton bean.

The code for

1. SwimCoach mySwimCoach = new SwimCoach();
2. return mySwimCoach;

is not executed for subsequent requests to the method public Coach swimCoach() . This code is only executed once during the initial bean creation since it is singleton scope.

That explains how @Bean annotation works for the swimCoach example.

Now let's take it one step further.

Here's your other question

**>> Please explain in detail whats happening behind the scene for this statement.**

1. return new SwimCoach(sadFortuneService())

The code for this question is slightly different. It is injecting a dependency.

In this example, we are creating a SwimCoach and injecting the sadFortuneService().

1. // define bean for our sad fortune service
2. @Bean
3. public FortuneService sadFortuneService() {
4. return new SadFortuneService();
5. }
7. // define bean for our swim coach AND inject dependency
8. @Bean
9. public Coach swimCoach() {
10. SwimCoach mySwimCoach = new SwimCoach(sadFortuneService());
12. return mySwimCoach;
13. }

Using the same information presented earlier

The code

1. // define bean for our sad fortune service
2. @Bean
3. public FortuneService sadFortuneService() {
4. return new SadFortuneService();
5. }

In the code above, we define a bean for the sad fortune service. Since the bean scope is not specified, it defaults to singleton.

Any calls for sadFortuneService, the @Bean annotation intercepts the call and checks to see if an instance has been created. First time through, no instance is created so the code executes as desired. For subsequent calls, the singleton has been created so @Bean will immediately return with the singleton instance.

Now to the main code based on your question.

1. return new SwimCoach(sadFortuneService())

This code creates an instance of SwimCoach. Note the call to the method sadFortuneService(). We are calling the annotated method above. The @Bean will intercept and return a singleton instance of sadFortuneService. The sadFortuneService is then injected into the swim coach instance.

This is effectively dependency injection. It is accomplished using all Java configuration (no xml).

This concludes the line-by-line discussion of the source code. All of the behind the scenes work.

**Injecting values from properties file**

1. Create Properties File
2. Load Properties File in spring config
3. Reference values from Properties file.
4. **Creating Properties file (save it with .properties extension)**



1. **Loading properties file in Spring Container**



1. **Referencing values from Properties file**

****

**SPRING MVC**

* Framework for building web applications
* Based on Model-View-Controller design pattern
* Leverages features of the Core Spring Framework (IoC,DI)

**Benefits of Spring MVC**

* Spring way of building web apps UIs in java.
* Leverage a set of reusable UI components.
* Help manage application state for web requests.
* Process form data: validation, conversion, etc.
* Flexible configuration for the view layer.

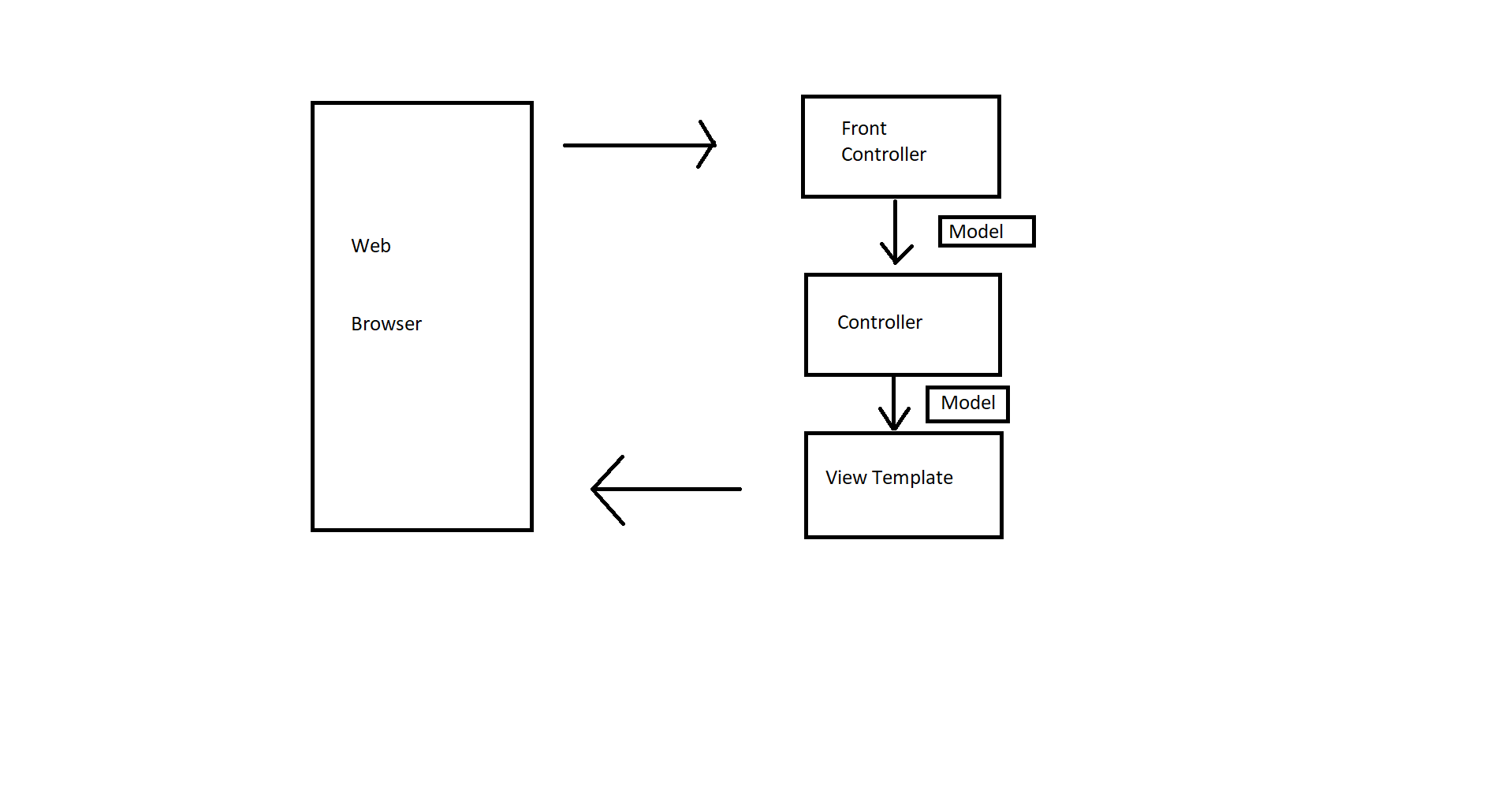
Basically, we have an incoming request coming from the browser, it'll encounter the Spring MVC front controller. This person will actually delegate the request off to a controller code. This controller code is code that you create that contains your business logic. You basically create a model, and you send the model back to the front controller, and then the front controller, will pass that model over to your view template. So, your view template is basically like a html page, or a JSP page that will take that data, and then render a response to the browser. So that's kind of the big picture of the MVC framework.

The MVC pattern results in separating the different aspects of the application (input logic, business logic, and UI logic), while providing a loose coupling between these elements.

* The **Model** encapsulates the application data and in general they will consist of POJO.
* The **View** is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.
* The **Controller** is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

**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)



***Everything starts off with that first incoming request and it encounters something called a front controller. So, the front controller is known as the dispatcher servlet. This front controller's actually part of the Spring framework. It's already developed by the Spring development team so you don't have to create this. This is part of the Spring jar files that you download; what this front controller will do is it will actually delegate the request to some other objects or items here in our system.***

***As a developer, you will create the model, the view, and the controller MVC.***

* ***The model objects contain data.***
* ***The view contains the templates i.e. the JSP Pages or the view page to actually render the data.***
* ***The controller classes contain the actual business logic or the processing logic.***

**Controller**

So, when the front controller has a request, it delegates the request to the controller. The controller is the code that you will actually create. Basically, in this controller, this contains your business logic. So, this is where you'll handle the request where you'll maybe read some form data then you'll take this data and store it or retrieve it. You may store it into a database or retrieve information from a web service. Basically, once you have your data and you're using it, then you can take that data and place it into the model. So, the model is just a container for your data and then you pass it back to the appropriate view template.

* Created by developer
* Contains the business logic
  + Handles the request
  + Stores/Retrieve data (db, web service.)
  + Place data in model
* Send to appropriate view template.

**Model**

The model contains your data. So, when your controller goes off and performs an operation to retrieve data from a backend system, like a database or web service, or a spring bean, you can take that data and place it into the model.so the model again is your container, like your suitcase or your luggage, for shipping data between various parts of your spring mvc application.so that model data will actually get passed over to the view template and they can actually handle that for displaying the data.

* Contains data
* Store/Retrieve data via backend systems
  + Database, web Service, etc.
  + Use spring bean if preferred
* Place data in model
  + Data can be any Java object/Collection

**View Template**

The most common view template that we'll use is JSP and JSTL. Spring MVC is very flexible. There are many different view template types. This model data comes over to your view template and then your JSP page can read that model data and display it. So, say for example, we have a list of students, or list of products, then JSP page can create a table to display that product list or that student list. Or, say for example, somebody is signing up for an airline flight, or is signing up for a class, then your view template, or your page can give them a confirmation, hey, you're registered for the class, here's your confirmation number. So that's the idea of the view template. It's basically a JSP page that will provide data to the user.

* Spring MVC is flexible
  + Supports many view templates
* Most common is **JSP**+**JSTL**
* Developer creates a page
  + Displays data
* Other view templates supported
  + Thymeleaf, Groovy
  + Velocity, Freemaker

Details: [www.luv2code.com/spring-mvc-views](http://www.luv2code.com/spring-mvc-views)

**Explanation: Client sends request, request goes to the web.xml file, web.xml file all the request to the dispatcher servlet, dispatcher servlet check the configuration file and it sends the request to the respective controller. now the controller will do some processing. the controller will send response to the dispatcher servlet. The dispatcher servlet will receive data and view name. Now there will be lots of pages and how will the dispatcher servlet know which page to send back. So, the view name will be written in the controller itself and the controller will send back the data and the view page to the front controller i.e. the dispatcher servlet. So, the client will only interact with the front Controller and it will have no idea how many controllers are there in the back-end side. Now when everything is good, there arises a question, “why should the dispatcher servlet send know which view to call?” – because it may happen that for time being, we are using JSP as our main technology and in future we may change some other view technology like Thymeleaf, Freemarker or velocity, we just have to change the configuration file for the dispatcher servlet.**

**Create Home Controller and View**

**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

1. **Create Controller Class**
2. **Define Controller Method**
3. **Add Request Mapping to controller method**
4. **Return View name**

* **Annotate class with @Controller**
  + **@Controller inherits from @Componenet. Supports scanning.**

Basically, this annotation tells that it is a controller and since it is inherited from Component, so when the Spring will do its component scanning, it will also pick up Controllers on its way.



1. **Develop View Name**

****

**Reading HTML Form data**

**Development Process**

1. Create Controller Class
2. Show HTML Form
   1. Create controller method to show HTML Form
   2. Create View Page for HTML Form
3. Process HTML Form
   1. Create controller method to process HTML Form
   2. Develop View Page for confirmation

**1,2,3**

****

**View page to show the Form**

****

**View Page after processing of Form**

****

**Adding Data to Spring Model**

* Model is a container for our application data
* In our controller
  + We can put anything in the model
  + Strings, objects, info from the databases, etc.
* JSP can access data from the model

**Development Process**

1. Create a method to process the data or use existing method to process the data
2. Add the data to the model.

**1,2**





**View Page**



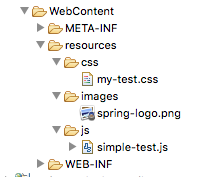
**How to use CSS, JavaScript and Images in a Spring MVC Web App?**

**Answer**

Here are the steps on how to access static resources in a Spring MVC. For example, you can use this to access images, css, JavaScript files etc.

Any static resource is processed as a URL Mapping in Spring MVC. You can configure references to static resources in the spring-mvc-demo-servlet.xml.

In my example, I'm going to have the following directory structure:



I chose to put everything in the "resources" directory. But you can use any name for "resources", such as "assets", "foobar" etc. Also, you can give any name that you want for the subdirectories under "resources".

---

**Step 1:** Add the following entry to your Spring MVC configuration file: **spring-mvc-demo-servlet.xml**

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

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

**Step 2:** Now in your view pages, you can access the static files using this syntax:

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

You need to use the JSP expression **${pageContext.request.contextPath}** to access the correct root directory for your web application.

Apply the same technique for reading CSS and JavaScript.

---

Here's a full example that reads CSS, JavaScript and images.

<!DOCTYPE html> <html>

<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>

<body>

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

<a href="showForm">Plain Hello World</a>

<br><br>

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

<br><br>

<input type="button" onclick="doSomeWork()" value="Click Me"/>

</body>

</html>

---

Source code for this example are available here:

- <https://gist.github.com/darbyluv2code/9a09543a226baeedc04b9a5037ca52ec>

Deploying To Tomcat using WAR files

**Bonus: Deploying your App to Tomcat as a Web Application Archive (WAR) file**

When you deploy your Java web apps, you can make use of a Web Application Archive (WAR) file.

The Web Application Archive (WAR) file is a compressed version of your web application. It uses the zip file format but the file has the .war extension.

If you are using Eclipse, then the best way to visualize it is think of your "WebContent" directory being compressed as a zip file with the .war extension.

This includes all of your web pages, images, css etc. It also includes the WEB-INF directory which includes your classes in WEB-INF/classes and supporting JAR files in WEB-INF/lib.

The WAR file format is part of the Java EE / Servlet specification. As a result, all Java EE servers support this format (ie jboss, weblogic, websphere, glassfish and tomcat).

Below, I provide steps on how to create a WAR file in Eclipse. I also show how to deploy the WAR file on Tomcat.

---

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

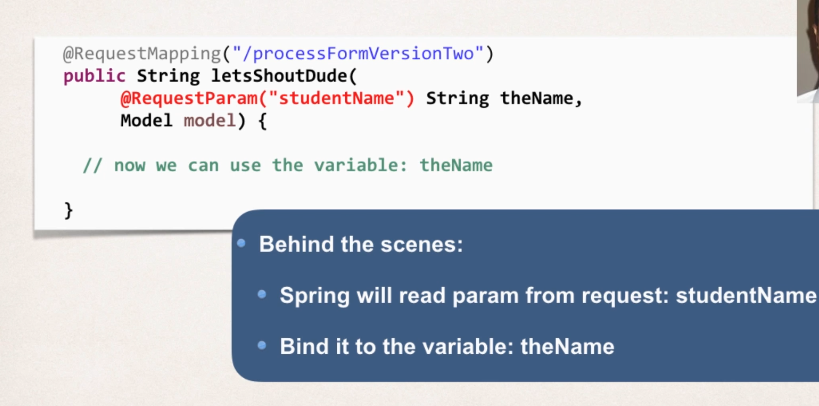
6. Deploy your new WAR file by copying it to <tomcat-install-directory>\webapps

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/>

**Binding Request Params**

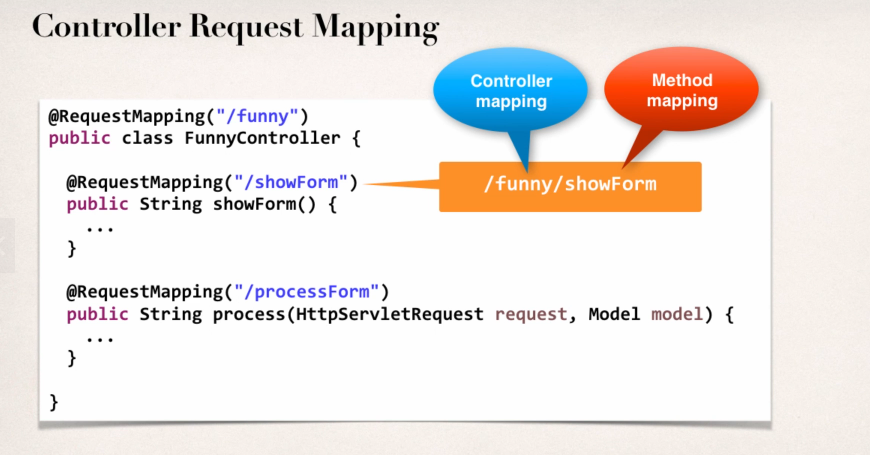
**Read HTML Form data with @RequestParam Annotation**



****

**Controller Level Request Mapping**

* Serves as parent mapping for controller
* All request mappings on methods in the controller are relative
* Similar to folder directory structures

****

All of the method mappings are relative to the Controller Level Request Mapping. Also a nice way to group the request mappings together, and a great way to resolve any conflicts that we may have with other request mappings.

**Conflicts in request Paths**

**Same request mappings with different method names.**

**NewController.java**

****

**HelloWorldController.java**

****

*So, both of the Controllers got same request mapping but different methods. So, there arises an ambiguity while running it on the server. To resolve this, we need Controller Level Request Mapping.*

**Changing the HelloWorldController.java**

****

***So showForm is now relative to hello. And we need to update the links in the view pages too.***

**How /hello is getting appended to the jsp file action for "processform"?**

**Answer**

You can use "processform" because it is a relative path to the controller "/hello" request mapping. Here is how it works.

1. When you wish to view the form, the HTML link points to "hello/showform". This calls the controller and it displays the form.

2. At this point the browser URL/path is: http://localhost:8080/spring-mvc-demo/hello

3. The HTML form uses "processform" for the form action. Notice that it does not have a forward slash, as a result, this will be relative to the current browser URL. Since the current browser URL is

Http://localhost:8080/spring-mvc-demo/hello

Then the actual form URL submission will send it to

Http://localhost:8080/spring-mvc-demo/hello/processform

The part in bold with map to the controller with top-level request mapping "/hello" and then map to request mapping in that class "/processform"

The key here is relative path of showing the form and then submitting to relative path.

**Spring MVC Form Tags**

* Spring MVC Form Tags are the building blocks for a web page.
* Form Tags are configurable and reusable for a web page.

**Data Binding**

* Spring MVC Form Tags can make use of data binding
* Automatically setting/retrieving data from a Java Object/bean.

**Web Structure**

* JSP page with special Spring MVC Form tags

<html>  
…..regular HTML

…..Spring MVC form tags

……More HTML

</html>

**Reference Spring MVC Form Tags**

* Specify the spring namespace at beginning of the JSP file.

****

|  |  |
| --- | --- |
| Form Tag | Description |
| form:form | It is a container tag that contains all other form tags. |
| form:input | This tag is used to generate the text field. |
| form:radiobutton | This tag is used to generate the radio buttons. |
| form:checkbox | This tag is used to generate the checkboxes. |
| form:password | This tag is used to generate the password input field. |
| form:select | This tag is used to generate the drop-down list. |
| form:textarea | This tag is used to generate the multi-line text field. |
| form:hidden | This tag is used to generate the hidden input field. |

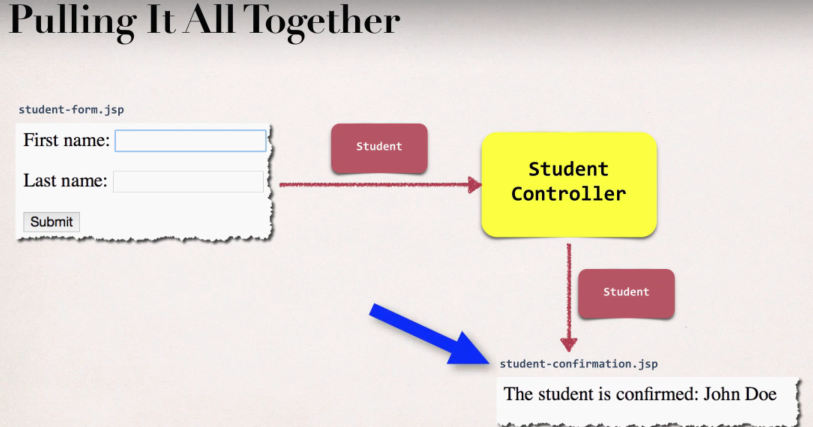
**Showing Form**

In the Spring Controller

* Before the form is shown, ***model attribute*** must be added.
* This is a bean that will hold the form data for the data binding.

**Creating a simple Student Login Page**

1. Create Student Class
2. Create Student Controller Class
3. Create HTML Form
4. Create processing code
5. Create Confirmation Code.



1. **Create Student Class**

****

**2,4 Create Student Controller Class and Processing Code**

****

**2** 🡪 Before showing the form, we added a parameter “theModel” for the Model. A student object “theStudent” (containing the first name and last name from the Student class) is being created and added to the model attribute. “student” is the name of the Model attribute and the value of the Model attribute “student” is the object i.e. “theStudent”. And at last, the student-form is being returned.

**4** 🡪 We add the method ***processForm()*** and bind the model attribute “***student***” (same as the modelAttribute in the JSP form) to the instance ***theStudent*** of the class ***Student***.

1. **Create HTML Form**

****

The highlighted taglib uri is added for the Spring form tags. We used the <form:form></form:form> tag with the action=”processForm” and modelAttribute=”student”. The model attribute is very much important for our controller. It should same as the Model Attribute in our student controller.

When the form is loaded, getter methods are being called i.e. the Spring MVC will call ***student.getFirstName()*** and ***student.getLastName()***. A similar thing happens when we submit the form. Spring MVC will call the ***student.setFirstName()*** and ***student.setLastName();***

**Refer to the previous point now for the “processing”**

**5. Create Confirmation Code**

****

**5**  🡪 We are making the confirmation page that the processForm leads to. The model attribute helps to retrieve the firstName and the lastName.

**Drop-Down List is represented by the tag <form:select>**

How to use properties file to load country options

**Answer:**

This solution will show you how to place the country options in a properties file. The values will no longer be hard coded in the Java code.

**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

1. BR=Brazil
2. FR=France
3. CO=Colombia
4. IN=India

Note the location of the properties file is very important. It must be stored in WEB-INF/countries.properties

**2. Update header section for Spring config file**

We are going to use a new set of Spring tags for <util>. As a result, you need to update the header information in the Spring config file.

File: spring-mvc-dmo-servlet.xml

Remove the previous header and add this.

1. <?xml version="1.0" encoding="UTF-8"?>
2. <beans xmlns="http://www.springframework.org/schema/beans"
3. xmlns:context="http://www.springframework.org/schema/context"
4. xmlns:mvc="http://www.springframework.org/schema/mvc"
5. xmlns:util="http://www.springframework.org/schema/util"
6. xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
7. xsi:schemaLocation="
8. http://www.springframework.org/schema/beans
9. http://www.springframework.org/schema/beans/spring-beans.xsd
10. http://www.springframework.org/schema/context
11. http://www.springframework.org/schema/context/spring-context.xsd
12. http://www.springframework.org/schema/mvc
13. http://www.springframework.org/schema/mvc/spring-mvc.xsd
14. http://www.springframework.org/schema/util
15. http://www.springframework.org/schema/util/spring-util.xsd">

**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" />

**4.1 In StudentController.java, add the following import statement:**

1. import java.util.Map;

**4.2 Inject the properties values into your Spring Controller: StudentController.java**

1. @Value("#{countryOptions}")
2. private Map<String, String> countryOptions;

**5. Add the country options to the Spring MVC model. Attribute name: theCountryOptions**

1. @RequestMapping("/showForm")
2. public String showForm(Model theModel) {
4. // create a student object Student
5. Student theStudent = new Student();
7. // add student object to the model
8. theModel.addAttribute("student", theStudent);
10. // add the country options to the model
11. theModel.addAttribute("theCountryOptions", countryOptions);
13. return "student-form";
14. }

**6. Update the JSP page, student-form.jsp, to use the new model attribute for the drop-down list: theCountryOptions**

1. <form:select path="country">
2. <form:options items="${theCountryOptions}" />
3. </form:select>

**7. Remove all references to country option from your Student.java.**

---

**DOWNLOAD FULL SOURCE CODE**

You can download entire code from here:

- <http://www.luv2code.com/downloads/spring-hibernate/spring-props-mvc-demo.zip>

**Radio Button is represented by the tag <form:radiobutton>**

**How to populate radiobuttons with items from Java class like we did with selectlist?**

You can follow a similar approach that we used for the drop-down list.

Here are the steps

1. Set up the data in your Student class

Add a new field

    private LinkedHashMap<String, String> favoriteLanguageOptions;

In your constructor, populate the data

        // populate favorite language options  
        favoriteLanguageOptions = new LinkedHashMap<>();

        // parameter order: value, display label  
        //  
        favoriteLanguageOptions.put("Java", "Java");  
        favoriteLanguageOptions.put("C#", "C#");  
        favoriteLanguageOptions.put("PHP", "PHP");  
        favoriteLanguageOptions.put("Ruby", "Ruby");

Add getter method

    public LinkedHashMap<String, String> getFavoriteLanguageOptions() {  
        return favoriteLanguageOptions;  
    }

2. Reference the data in your form

        Favorite Language:  
          
        <form:radiobuttons path="favoriteLanguage" items="${student.favoriteLanguageOptions}"  />

Source code is available here:  
- <https://gist.github.com/darbyluv2code/debb69b1bf8010d84d50e0542e809ffb>

**Checkbox is represented by the tag <form: checkbox>**

**In java code**

**Need to add support when user selects multiple options**

**Array of Strings**

**Add appropriate get/set methods**

* Use array field in class file if you want to print elements in list format in the JSP Page. The code works fine without the String.

**Spring MVC Form Validation – Applying Built-in Validation Rules**

**The need of validation.**

Check the user input form for

* Required fields
* Valid numbers in a range
* Valid format (postal code)
* Custom Business Rule.

**Java Standard Bean Validation API**

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

All defined at once on: <https://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 project.

**Bean Validation Features**

* required
* validate length
* validate numbers
* validate with regular expressions
* custom validation

**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 a given date |
| Others…. | …. |

**RoadMap**

* Setup dev environment
* Required field
* Validate number range: min, max
* Validate using regular expression
* Custom Validation

**Setup Dev Environment**

* Download Validation JARs files from this website - <https://hibernate.org/validator/>
* Add to the project

**Checking for Required Fields Overview**



The form is submitted and the data from the fields are passed to the controller. The validation is being checked there. If the validation is successful, then the confirmation page is being sent to view. And if the validation fails, the form is being resent to the user in order to fix it and resubmit it.

**Development Process**

1. Add validation rule to Class
2. Display error messages on HTML form
3. Performer validation in controller class
4. Update confirmation page.

Special Note about BindingResult Parameter Order

When performing Spring MVC validation, the location of the BindingResult parameter is very important. In the method signature, **the BindingResult parameter must appear immediately after the model attribute**.

If you place it in any other location, Spring MVC validation will not work as desired. In fact, your validation rules will be ignored.



Here is the relevant section from the Spring Reference Manual

***Defining @RequestMapping methods***

*@RequestMapping handler methods have a flexible signature and can choose from a range of supported controller method arguments and return values.  
...*

*The Errors or BindingResult parameters have to follow the model object that is being  
bound immediately ...*

*Source:* [*https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-methods*](https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html#mvc-ann-methods)

1. **Adding validation rule to the Class file**

****

1. **Displaying Error Messages on HTML form.**

****

1. **Perform Validation in Controller Class**

****

1. **Update Confirmation Page**

****

However, there’s one error when we enter whitespaces in the required field, the validation doesn’t work.

@InitBinder

* **@InitBinder** annotation works as a pre-processor
* It will pre-process each web request to our controller
* Method annotated with @InitBinder is executed.
* @InitBinder will be used to trim strings
  + Remove leading and trailing white space
* If string only has white spaces, it will be trimmed to null

Reminder: @InitBinder will basically pre-process all web requests coming to the controller.

***StringTrimmerEditor(provided in the String API):*** removes whitespaces – leading and trailing.

* Pre-process every String form data
* Remove leading and trailing white space
* If string has white space – trim it to null.



**Validate a Number Range**

* Input field where user can enter a range from 0 to 10 (as desired)

**Development Process**

1. Add validation rule to entity class
2. Display error messages on HTML form
3. Perform validation in Controller Class
4. Update Confirmation Page
5. **Adding validation to the entity class**





1. **Display error messages on HTML Form**



1. **Performing validation in Controller Class**

****

****

1. **Updating confirmation Page**

****

**Applying Regular Expressions**

**Regular Expressions**

* A sequence of characters that define a search pattern
  + This pattern is used to find or match strings
* Applied for validation of forms.

**What are Regular Expressions?**

**A *Regular Expression* is a sequence of characters that constructs a search pattern. When you search for data in a text, we can use this search pattern to describe what we are looking for.**

**^[a-z0-9\_-]{3-15}$**

**^ 🡪 Start of the line**

**[a-z0-9\_-] 🡪 letters, numbers, underscores, hyphens**

**{3-15} 🡪 3 to 15 characters long**

**$ 🡪 End of the line**

**What is Java Regex?**

The Java Regex is an API which is used to define a patten for searching or manipulating *Strings.* It is widely used to define the constraint on Strings such as password and email validation.

* **Matcher Class**

This class is used to perform the match operations on a character sequence.

Comprises of various methods.

|  |  |
| --- | --- |
| boolean matches() | Tests if the given regular expression matches or not |
| boolean find() | Used to find the next expression that matches the pattern |
| boolean find(int start) | Searches the next expression from the given start number |
| string group() | Used to return the matched sequence |
| int start() | Returns the starting index |
| int end() | Returns the ending index |
| int groupcount() | Returns the total number of the matched sequence |

* **Pattern Class**

A compiled version of regular expression which is used to define the pattern of a regex engine.

**Various methods**

|  |  |
| --- | --- |
| Static Pattern compile(String regex) | It compiles the given regex and returns the instance of a pattern |
| Matcher matcher(charSequence input) | Used to create a matcher that matches the given input with the pattern |
| Static boolean matches(String regex) | Used to split the given String around matches of a given pattern |
| String pattern() | Helps to return the regex pattern |
| Int end() | Returns the ending index |

**Character Class**

|  |  |
| --- | --- |
| [abc] | a,b or c [A simple class] |
| [^abc] | Any Class except a, b or c[negation] |
| [a-zA-Z] | **A through Z** or **a through z** |
| [a-d[m-p]] | A through d or m through p |
| [a-z&&[def]] | d,e, or f(Intersection) |
| [a-z&&[^bc]] | A through except b or c(Substraction) |
| [a-z&&[^m-p]] | A through z andnot m through p (Substraction) |

**Regex Quantifiers**

Quantifiers specify the number of occurrences of a character.

|  |  |
| --- | --- |
| X? | X occurs once or not at all |
| X+ | X occurs more than one times |
| X\* | X occurs zero or more time |
| X{n} | X occurs n times only |
| X{n,} | X occurs n or more times only |
| X{y,z} | X occurs at least y times but less than z times |

**Regex Metacharacters**

The regular expression metacharacters work as shortcuts.

|  |  |
| --- | --- |
| . | Can be any character |
| \d | Represents any digit |
| \D | Represents any non digit |
| \s | Represents any white space |
| \S | Non white space character |
| \w | Can be a word character |
| \W | Any non-word character, short for [^\w] |
| \b | Represents a word boundary |
| B | Represents a non word boundary |

Links –

1. [**https://www.javatpoint.com/java-regex**](https://www.javatpoint.com/java-regex)
2. [**https://www.youtube.com/watch?v=f0lZbeueVzU&t=523s**](https://www.youtube.com/watch?v=f0lZbeueVzU&t=523s)

**Email Validator** -[**https://mkyong.com/regular-expressions/how-to-validate-email-address-with-regular-expression/**](https://mkyong.com/regular-expressions/how-to-validate-email-address-with-regular-expression/)

**Development Process**

1. Adding validation Rules to entity class
2. Display error messages on HTML Form
3. Update confirmation page
4. Adding validation Rules to entity class





1. Displaying error messages on HTML Form



1. Updating Confirmation Page



**Make Integer Field Required**

**Problem: If we left the field “Free Passes” empty and submitted it even after adding** @NotNull(message="is required")

There will be an error showing that “failed to convert…” since the blank spaces are considered as a null string

**Solution:**

Change the primitive data type to Wrapper class data type i.e. int to Integer in the class file.

**Handle String input for integer fields**

**Development Process**

1. Create custom error message
   1. src/resources/message.properties
2. Load Custom messages resource in Spring Config file
   1. WebContent/WEB-INF/spring-mvc-demo-servlet.xml

**1.a**

messages.properties



typeMismatch 🡪 Error type

customer 🡪 Spring model attribute name

freePasses 🡪 Field name

Invalid number 🡪 Custom message

**2.a**

****

**Explanation :** [**https://www.udemy.com/course/spring-hibernate-tutorial/learn/lecture/6846302#content**](https://www.udemy.com/course/spring-hibernate-tutorial/learn/lecture/6846302#content)

**How to Make Integer field required and handle Strings: freePasses?**

***Question:***

I am getting the following error when i submit the form with an empty value for customer "freePasses". I am  using @NotNull on the field "freePasses". It is not throwing "is required" after validation after submit.

How to fix this? Please help.

Also, how do I handle validation if the user enters String input for the integer field?

-----

***Answer:***

Great question!

The root cause is the freePasses field is using a primitive type: int. In order to check for null we must use the appropriate wrapper class: **Integer**.

To resolve this, In Customer.java, replace "int" with "Integer"

@NotNull(message="is required")       
@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 **Integer** freePasses;

Then update your getter/setter methods to use "Integer"

    public **Integer** getFreePasses() {  
        return freePasses;  
    }

    public void setFreePasses(**Integer** freePasses) {  
        this.freePasses = freePasses;  
    }

Save everything and retest.

=====

Here is the full source code.

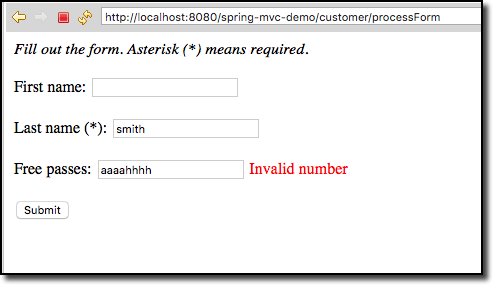
1. package com.luv2code.springdemo.mvc;
2. import javax.validation.constraints.Max;
3. import javax.validation.constraints.Min;
4. import javax.validation.constraints.NotNull;
5. import javax.validation.constraints.Pattern;
6. import javax.validation.constraints.Size;
8. public class Customer {
10. private String firstName;
12. @NotNull(message="is required")
13. @Size(min=1, message="is required")
14. private String lastName;
16. @NotNull(message="is required")
17. @Min(value=0, message="must be greater than or equal to zero")
18. @Max(value=10, message="must be less than or equal to 10")
19. private Integer freePasses;
21. @Pattern(regexp="^[a-zA-Z0-9]{5}", message="only 5 chars/digits")
22. private String postalCode;

25. public String getPostalCode() {
26. return postalCode;
27. }
29. public void setPostalCode(String postalCode) {
30. this.postalCode = postalCode;
31. }
33. public String getFirstName() {
34. return firstName;
35. }
37. public void setFirstName(String firstName) {
38. this.firstName = firstName;
39. }
41. public String getLastName() {
42. return lastName;
43. }
45. public void setLastName(String lastName) {
46. this.lastName = lastName;
47. }
49. public Integer getFreePasses() {
50. return freePasses;
51. }
53. public void setFreePasses(Integer freePasses) {
54. this.freePasses = freePasses;
55. }
57. }

====

**Handle String Input for Integer Fields**

If the user enters String input such as "abcde" for the Free Passes integer field, we'd like to give a descriptive error message.



We basically need to override the default Spring MVC validation messages.

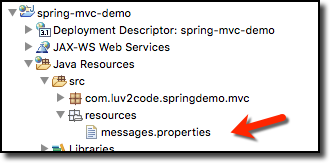
Follow these steps.

1. In your Eclipse project, expand the node: **Java Resources**

2. Right-click the **src** directory and create a new sub-directory: **resources**

3. Right-click the **resources** sub-directory and create a new file named: **messages.properties**

Your directory structure should look like this:



4. Add the following entry to the **messages.properties** file

typeMismatch.customer.freePasses=Invalid number

5. Save file

---

This file contains key/value pairs for the error message type

For a basic example:

  typeMismatch.customer.freePasses=Invalid number

The format of the error key is:   code + "." + object name + "." + field

To find out the given error code, in your Spring controller, you can log the details of the binding result

 System.out.println("Binding result: " + theBindingResult);

For details, see the docs here

- <http://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/validation/DefaultMessageCodesResolver.html>

---

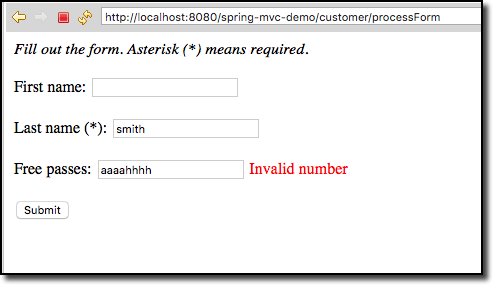
6. Edit your config file: **WEB-INF/spring-mvc-demo-servlet.xml**

Add the following:

1. <bean id="messageSource"
2. class="org.springframework.context.support.ResourceBundleMessageSource">
4. <property name="basenames" value="resources/messages" />
6. </bean>

7. Save the file. Restart the Tomcat server

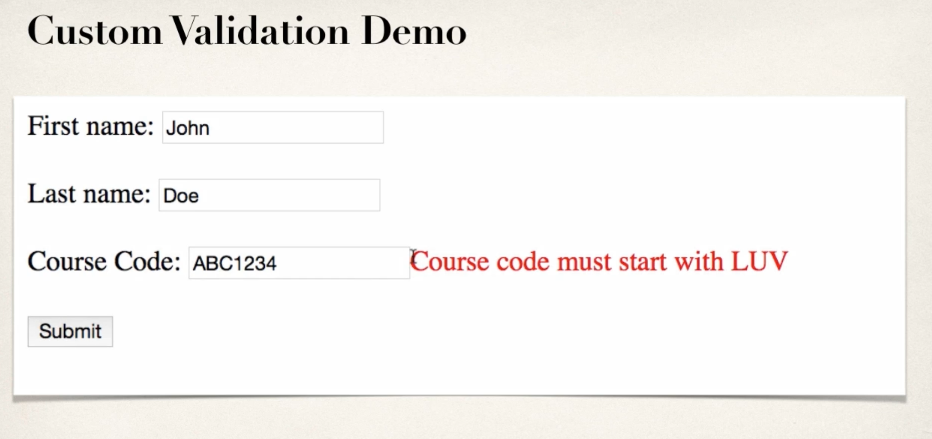
8. Run your app and add bad data for the "Free Passes" field. You will see the error message from our properties file.



Resources for this lecture

* [solution-code-spring-mvc-validation-handle-strings-for-integer-fields.zip](javascript:void(0))

**Spring MVC Form Validation – Creating CUSTOM Validation Rules**

****

Adding custom validation on “Course Code”. Should start with **LUV**.

**Custom Validation**

* Perform custom validation based on the business rules
  + Our example: Course code must start with “LUV”
* Spring MVC calls our custom validation
* Custom validation returns boolean value for pass/fail (true/false)

**Create Custom Java Annotation**

So far, we have used predefined validation rules: @Min, @Max and others. For custom validation, we will create a **CUSTOM JAVA ANNOTATION** such as @CourseCode

**Development Process**

1. Create custom validation rule
   1. Create @CourseCode annotation
   2. Create CourseCodeConstraintValidator [CourseCodeConstraintValidator – Helper Class. Contains custom business logic for validation]

Point a & b are according to the current example used here.

1. Add validation rule to the entity class
2. Display error messages on HTML form
3. Update confirmation page.
4. **Creating Custom Validation Code**
   1. **Creating @CourseCode annotation**



1. **Creating Custom Validation Code**
   1. **Creating CourseCodeConstraintValidator**

****

1. **Adding validation rules to entity class.**

****

1. **Displaying error messages on HTML**

****

1. **Updating confirmation page**

****

**Is it possible to integrate multiple validation string in one annotation?**

**Question:**

Is it possible to integrate multiple validation string in one annotation? For example, validate against both LUV and TOPS.

**Answer:**

Yes, you can do this. In your validation, you will make use of an array of strings.

Here's an overview of the steps.

1. Update CourseCode.java to use an array of strings

2. Update CourseCodeConstraintValidator.java to validate against array of strings

3. Update Customer.java to validate using array of strings

---

**Detailed Steps**

**1. Update CourseCode.java to use an array of strings**

Change the value entry to an array of Strings:

1. // define default course code
2. public String[] value() default {"LUV"};

Note the use of square brackets for the array of Strings. Also, the initialized value uses curley-braces for array data.

**2. Update CourseCodeConstraintValidator.java to validate against array of strings**

Change the field for coursePrefixes to an array

private String[] coursePrefixes;

Update the isValid(...) method to loop through the course prefixes. In the loop, check to see if the code matches any of the course prefixes.

1. @Override
2. public boolean isValid(String theCode,
3. ConstraintValidatorContext theConstraintValidatorContext) {
4. boolean result = false;
6. if (theCode != null) {
8. //
9. // loop thru course prefixes
10. //
11. // check to see if code matches any of the course prefixes
12. //
13. for (String tempPrefix : coursePrefixes) {
14. result = theCode.startsWith(tempPrefix);
16. // if we found a match then break out of the loop
17. if (result) {
18. break;
19. }
20. }
21. }
22. else {
23. result = true;
24. }
26. return result;
27. }

**3. Update Customer.java to validate using array of strings**

1. @CourseCode(value={"TOPS", "LUV"}, message="must start with TOPS or LUV")
2. private String courseCode;

Note the use of curley braces.

---

Complete Source Code:

<https://gist.github.com/darbyluv2code/0275ddb6e70e085a10fd464e36a42739>

---

That's it. This provides a solution to integrate multiple validation string in one annotation. In this example, the code validates against both LUV and TOPS.