Spring MVC Annotation Types

- Can handle multiple actions.
- Do not need to be stored in a configuration file.
  - Using the **RequestMapping** annotation type, a method can be annotated to make it a request-handling method.
- Controller Annotation Type

```
@Controller
public class CustomerController
```

172

Notes:

Spring MVC Annotation Types ...

- If more than one attributes appear in @RequestMapping, the value attribute name must be written.
- The method attribute takes a set of HTTP methods that will be handled by the corresponding method.

```
method={RequestMethod.POST, RequestMethod.PUT}
```

- If the method attribute is not present, the request-handling method will handle any HTTP method.

```
@RequestMapping(value="/delete",
        method={RequestMethod.POST, RequestMethod.PUT})
```

174

Notes:

Using An Annotation-Based Controller

- All requests, including those for static resources, are directed to the dispatcher servlet.
  - In order for static resources to be handled properly, some <resources/> need to be added

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

- Without <annotation-driven/>, the <resources/> elements will prevent any controller from being invoked.

176

Notes:

Using An Annotation-Based Controller ...

- The View

```
@import url(css/main.css)

<input type="text" id="name" name="name"
        tabindex="1">


@import url(css/main.css)

${product.name}<br/>
    Description: ${product.description}<br/>
    Price: $${product.price}
```

178

Notes:

Dependency Injection ...

@Autowired

```
@Controller
public class ProductController {

   private static final Log logger = LogFactory
         .getLog(ProductController.class);

   @Autowired
   private ProductService productService;

   @RequestMapping(value = "/product_input")
   public String inputProduct() {
      logger.info("inputProduct called");
      return "ProductForm";
   }

@RequestMapping(value = "/product_view/{id}")
   public String viewProduct(@PathVariable Long id, Model model) {
      Product product = productService.get(id);
      model.addAttribute("product", product);
      return "ProductView";
   }
}
```

180

Notes:

Redirect and Flash Attributes

- A forward is faster than a redirect because a redirect requires a round-trip to the server and a forward does not.
  - Redirect to an external site, like a different web site.
  - Cannot use a forward to target an external site so a redirect is the only choice.
- Avoid the same action from being invoked again when the user reloads the page.
  - If the page is reloaded after the form is submitted, saveProduct will be called again and the same product would potentially be added the second time.
  - No side-effect when called repeatedly.
  - For instance, the user could be redirected to a ViewProduct page

182

Notes:

Redirect and Flash Attributes ...

```
@RequestMapping(value = "product_save", method = RequestMethod.POST)
public String saveProduct(ProductForm productForm,
      RedirectAttributes redirectAttributes) {
   logger.info("saveProduct called");
   // no need to create and instantiate a ProductForm
   // create Product
   Product product = new Product();
   product.setName(productForm.getName());
   product.setDescription(productForm.getDescription());
   try {
      product.setPrice(Float.parseFloat(productForm.getPrice()));
   } catch (NumberFormatException e) {
   }

   // add product
   Product savedProduct = productService.add(product);

   redirectAttributes.addFlashAttribute("message",
         "The product was successfully added.");

   return "redirect:/product_view/" + savedProduct.getId();
}
```

184

Notes:

Request Parameters and Path Variables ...

- An argument that captures request parameter productId.

```
public void sendProduct(@RequestParam int productId)
```

- A path variable is similar to a request parameter, except that there is no key part, just a value.

```
/product_view/productId
```

- Using path variables

```
@RequestMapping(value = "/product_view/{id}")
public String viewProduct(@PathVariable Long id, Model model) {
   Product product = productService.get(id);
   model.addAttribute("product", product);
   return "ProductView";
}
```

- Can use multiple path variables in request mapping

186

Notes:

@ModelAttribute ...

- If no key name is defined, then the name will be derived from the name of the type to be added to the Model.
  - Every time the following method is invoked, an instance of **Order** will be retrieved or created and added to the **Model** using attribute key **order**.
- A method annotated with @ModelAttribute will be invoked right before a request-handling method.

```
@ModelAttribute
public Product addProduct(@RequestParam String productId) {
   return productService.get(productId);
}
```

188

Notes:

Data Binding Overview

- Data binding is a feature that binds user input to the domain model.
  - HTTP request parameters, which are always of type **String**, can be used to populate object properties of various types.
  - Makes form beans (e.g. instances of **ProductForm**) redundant.
- Need the Spring form tag library.

190

Notes:

Data Binding Overview ...

- Parse the **price** property in the ProductForm because it was a String and a float was needed to populate the Product 's **price** property.
- Because of data binding, the ProductForm class is no longer needed and no parsing is necessary for the price property of the Product object.
- Another benefit of data binding is for repopulating an HTML form when input validation fails.

192

Notes:

Controller Code

- RequestMapping code in the **Controller**:

```
@RequestMapping(value = "/book_input")
public String inputBook(Model model) {
   ...
   model.addAttribute("book", new Book());
   return "BookAddForm";
}
 @RequestMapping(value = "/book_save")
   public String saveBook(@ModelAttribute Book book) {
      Category category = bookService.getCategory(book.getCategory().getId());
      book.setCategory(category);
      bookService.save(book);
      return "redirect:/book_list";
   }
```

194

Notes:

Data Binding Example ...

The **BookList.jsp** page:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE HTML>
<html>
<head>
<title>Book List</title>
<style type="text/css">@import url("<c:url value="/css/main.css"/>");</style>
</head>
<body>

<div id="global">
<h1>Book List</h1>
<a href="<c:url value="/book_input"/>">Add Book</a>
<table>
<tr>
   <th>Category</th>
   <th>Title</th>
   <th>ISBN</th>
   <th>Author</th>
   <th> </th>
</tr>
<c:forEach items="${books}" var="book">
   <tr>
      <td>${book.category.name}</td>
      <td>${book.title}</td>
      <td>${book.isbn}</td>
      <td>${book.author}</td>
      <td><a href="book_edit/${book.id}">Edit</a></td>
   </tr>
</c:forEach>
</table>
</div>
</body>
</html>
```

196

Notes:

Data Binding Example ...

The **BookEditForm.jsp** page:

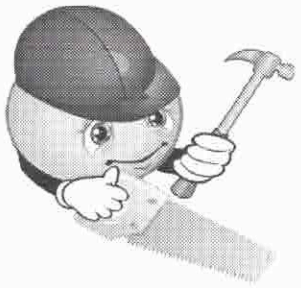
```
<%@ taglib prefix="form" uri="http://www.springframework.org/tags/form"
%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<!DOCTYPE HTML>
<html>
<head>
<title>Edit Book Form</title>
<style type="text/css">@import url("<c:url value="/css/main.css"/>");</style>
</head>
<body>

<div id="global">
<form:form commandName="book" action="/book_update" method="post">
  <fieldset>
    <legend>Edit a book</legend>
    <form:hidden path="id"/>
    <p>
      <label for="category">Category: </label>
       <form:select id="category" path="category.id" items="${categories}"
         itemLabel="name" itemValue="id"/>
    </p>
    <p>
      <label for="title">Title: </label>
      <form:input id="title" path="title"/>
    </p>
    <p>
      <label for="author">Author: </label>
      <form:input id="author" path="author"/>
    </p>
    <p>
      <label for="isbn">ISBN: </label>
      <form:input id="isbn" path="isbn"/>
    </p>

    <p id="buttons">
      <input id="reset" type="reset" tabindex="4">
      <input id="submit" type="submit"
tabindex="5"
        value="Update Book">
    </p>
  </fieldset>
</form:form>
</div>
</body>
</html>
```

198

Notes:

Overview

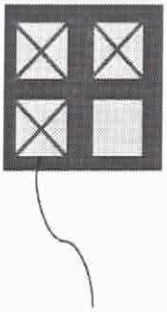
- Why AOP?
- Lots of New Terms!
- What tools are needed?
- How Does It Work?



200

Notes:

Why Aspect Oriented Programming? ...



Unneeded capabilities

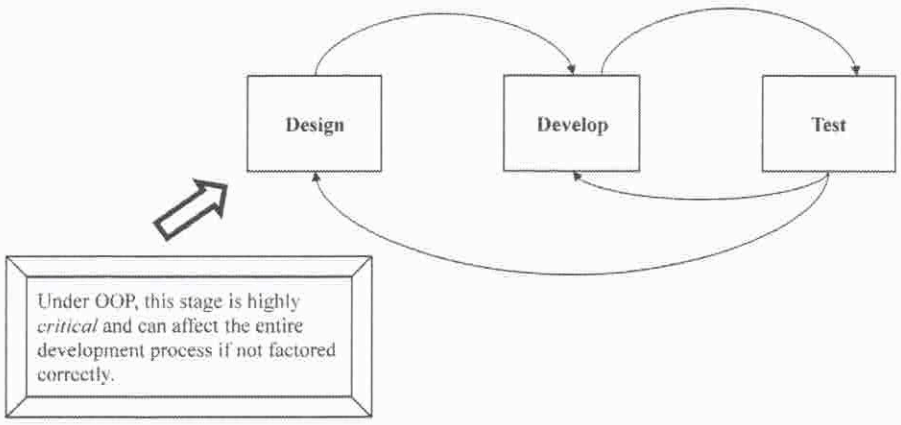
- Problem:
  - You are about to use an **architectural model** (e.g. *EJB*) that incorporates features such as persistence, transactions, and security—yet, you do not have the need for all of these features
- What do you do?
  1) Continue with an inappropriate model
  2) Design an application that modularizes the secondary support features for easy addition or removal of those features

202

Notes:

Why Aspect Oriented Programming? ...

- Impossible to <u>exactly</u> design an application
  - Possibilities of over design, under design occur



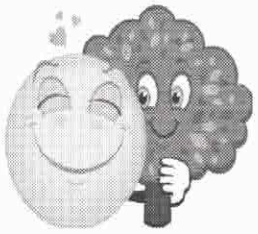
204

Notes:

When a proper model is not developed initially, lots of time can be spent refactoring applications. Classes must be added and removed, methods must be re-written, and code must be rebuilt then retested. This makes the initial design stages very important.

Core vs. Crosscutting Concerns

- ***Concerns:***
  - A *concern* is a requirement (subsystem) implemented within a larger application
  - Examples:
    - Transactional systems
    - Logging
    - Persistence
    - Error handling systems
    - Thread Synchronization
    - Caching Operations



206

Notes:

There are typically two types of concerns: **core concerns** and **crosscutting concerns**. Core concerns tend to be your primary requirements and your main business services within the application.

Core vs. Crosscutting Concerns ...

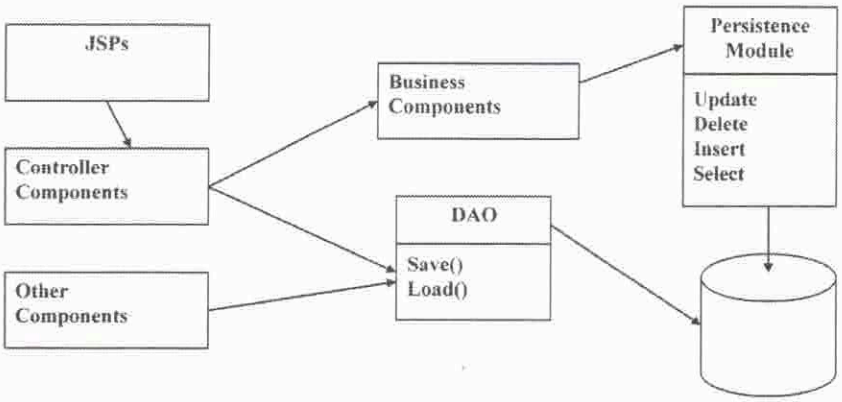
- ***Crosscutting Concerns** (cont'd):*
  - *Example: An Enterprise Application* – Core elements include the server-components, business objects, delegates.
  - A logging system would be "tied" into these objects at various points throughout the application.

208

Notes:

Recognizing Concerns

- Code Example illustrating Crosscutting Concerns (***scattering***)



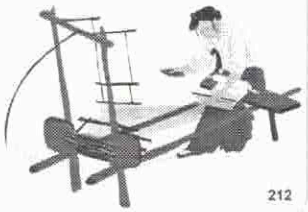
210

Notes:

Another form of implementing crosscutting concerns in conventional systems is via the technique known as code **scattering**. Scattering involves the "spreading" of referenced code throughout the various components within a system. Scattering may lead to repeated code or similarly written code written in various components throughout the system.

The Weaver

- ***Weaving*** → The process the AOP compiler undertakes by combining core and crosscutting concerns to yield a final component
  - Aspects contain crosscutting concerns
    - They are *weaved* into the core concerns
  - Weaving can be done at compile-time (like AspectJ), or
  - At runtime (like Spring AOP)



212

Notes:

Consider an XSL Transformation as an example. In XSLT, the transformation engine receives a source document that must be converted into a different format on the output. The conversion process is largely dictated by the "rules" that are applied to the transformation. The "rules" in XSLT are found in the XSL files. These "rules" can be easily changed by supplying different XSL transformation files.

AOP works much the same way, in which the final system will be a result of the original class files (source files) and the "rules" applied during the weaving process. The "rules" in this case are known as **Aspects**.

Aspects

- *Aspects* are the "rules for weaving"
- They contain the instructions for "weaving" *crosscutting concerns* into the core system
- ***Spring Aspects*** can use either
  - SpringAOP *(managed within XML Definition files)*
  - AspectJ *(managed with annotations, e.g. @something)*

Notes:

Aspect Structure

```
public aspect SomeAspect
{
        // fields

        // methods

        // pointcuts

        // advice
}
```

- Aspects are composed of:
  - Declarations
  - Introductions
  - Pointcuts
  - Advice

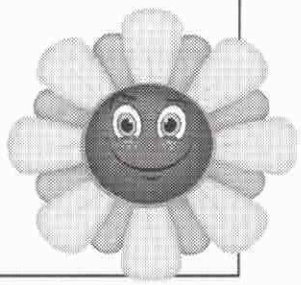
Aspects may also contain data members and methods much like Java POJOs.

216

Notes:

A Simple Example

```
public aspect DayOfWeekAspect
{
      pointcut call_GetDayOfWeekPointcut( ) :
                        call(* DayOfWeekService.getDayOfWeek(..));

      before() : call_GetDayOfWeekPointcut( )
      {
            System.out.println("In the before-advice...");
      }

      after() : call_GetDayOfWeekPointcut()
      {
            System.out.println("In the after-advice...");
      }
}
```

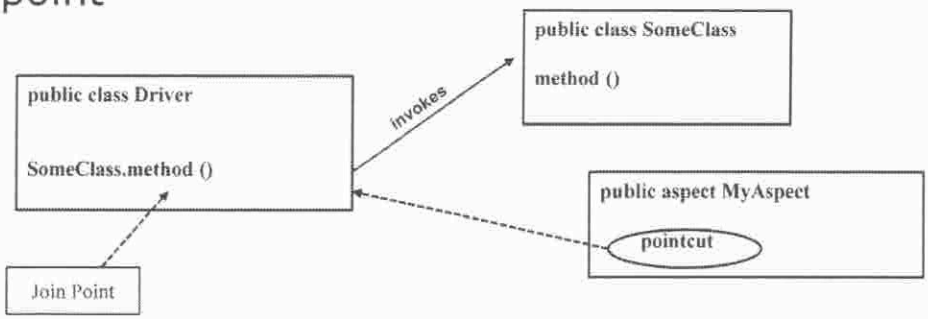


218

Notes:

Pointcuts

- Structures within an Aspect that select a join point



Pointcut Syntax:

pointcut pointCutName(args) : pointcut_designators;

Example:

pointcut captureMethod() : call(SomeClass.method(..));*

220

Notes:

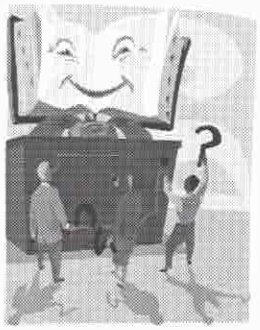
Pointcuts are found within aspects and capture join points. A pointcut can be thought of as the "rule" to be applied and the join point is the condition where that rule would be invoked.

Advice

- ***Advice*** is the specific instructions to be applied when the AOP compiler selects a pointcut
  - It can be applied *before*, *after*, or *around* (during) the execution of a join point

Example:

```
after() : captureMethod()
{
  System.out.println("Class2.method() completed!");
}
```



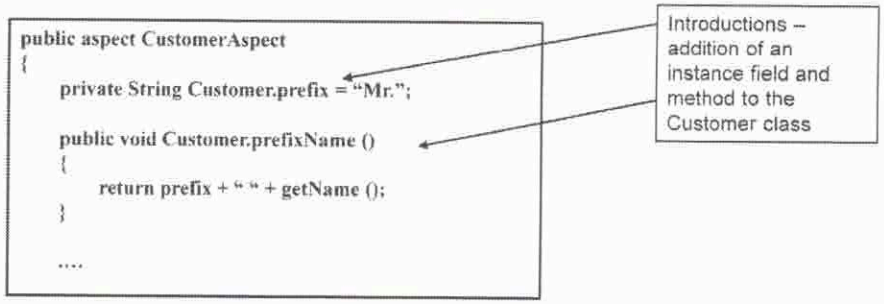
222

Notes:

Advice can be called **before** the execution of a join point's code, **after** the execution of a join point, or **around** it. Around advice can skip, replace, or modify the parameters of the code representing the join point.

Introductions

- ***Introductions*** are instructions that can make static additions to a class



224

Notes:

Notice that the aspect (CustomerAspect) is adding a new instance variable and method to the customer class. This is known as static crosscutting.

Additional forms of static crosscutting include:

- ✓modifying the type-hierarchy
- ✓softening exceptions
- ✓declaring compile-time error/warnings

Lab 9: Exploring Aspects

- In this exercise you will create and implement an *Aspect* within an *AspectJ project*
  - Follow the steps below to install **AspectJ** into Eclipse.
  - Create the *DayOfWeekAspect* aspect (*File → New → Other → AspectJ → Aspect*)
  - Use *before* and *after* advice on the DayOfWeekService class methods
  - Follow the additional instructions in **SpringLab9a**

226

Notes:

Installing AspectJ. AspectJ can be installed directly from within Eclipse by selecting:

Help → Software Updates... → Available Software (tab) → Add Site... (button)

Then type in the following URL:

http://download.eclipse.org/tools/ajdt/34/update

Next, in the list of available software items, select the AJDT Update Site item and then choose Install...