



CHAPTER 3
FORM BEANS



OBJECTIVES

After completing “Form Beans,” you will be able to:

- **Describe the importance of action forms in handling user inputs and HTML form data in the Struts architecture:**
 - How form beans relate to user input and to form output
 - How Struts actions can query and modify action forms
 - How action forms can provide a natural adapter to the model
- **Implement action form classes and configure them to be used with action mappings.**
- **Declare and use dynamic action forms.**

Working with HTML Forms

- Another of the common problems tackled by Struts is that of managing HTML forms from JSPs and servlets.
 - Servlets that process HTML form data must repetitively encode the “wiring” from form controls to JavaBean properties – and then often implement the reverse process to populate a form for the response.
 - JSPs get the advantage of `<jsp:setProperty>` and the wildcard feature, but form encoding in the response is still problematic.
 - The raw HTML is a bit troublesome, and requires a careful eye towards maintainability as various fields must be synchronized with bean property names in scriptlet expressions, JSP standard actions such as `<jsp:getProperty>`, or native JSP expressions.
 - There are higher-level tasks, both in receiving and transmitting forms, that could be automated as well.

Managing Information

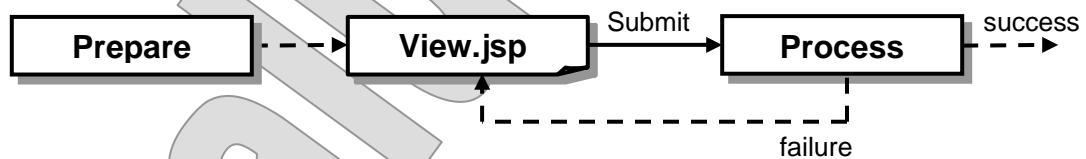
- **More generally, Web applications face the challenge of managing complex sets of information at various scopes and over various durations.**
 - Much of this information originates from HTML forms.
 - Some of it is injected into the system from persistent stores or other components, and some of it is derived by the Web application itself.
- **How can various controllers and views “see” the information they need, to read or to write, and how can they share a common record of the application’s transient state?**

What Not to Do

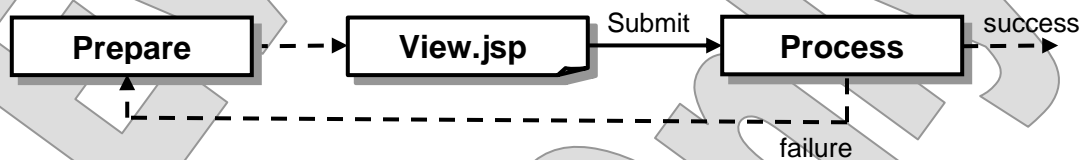
- **The temptation to store everything in session attributes should be avoided.**
 - Some information is really only relevant during a request, and yet many components must share it.
 - The session will provide a good common value space, certainly, but information will then live on after the request.
 - Eventually the session space gets cluttered with colliding names or with values that wind up overwriting each other.

What Not to Do

- Another practice whose wiles are a bit subtler is that of preparing data from a controller for a view in request-scope attributes.
 - This is viable, but one can run into difficulty.
 - It's important to be sure that all possible paths to the view that uses request-scope information will pass through the controller in question.
 - If, for instance, an action invoked by the view rejects the view's input and feeds control back to the view (directly), the request-scope values will be lost, and the presentation will be lacking.



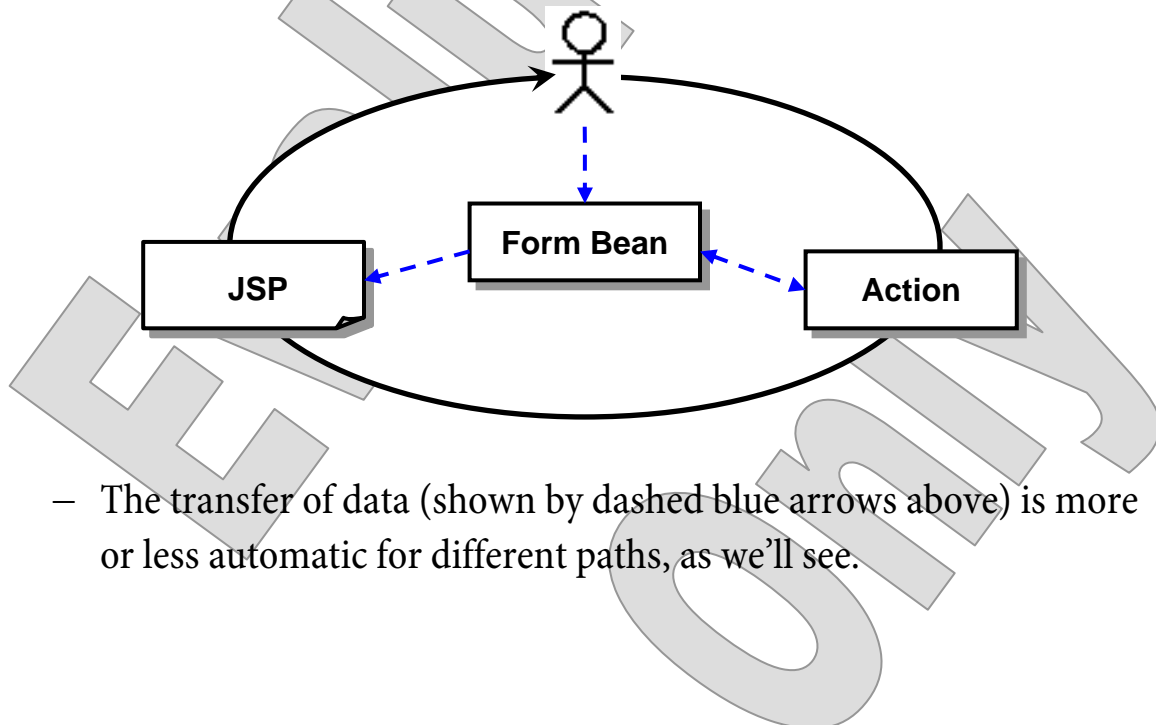
- One can guard against this by never mapping to the view, only to the controller that feeds it.



- This in turn requires that the controller only be a mediator/preparer for that view, and that it not perform any mutations on the model – we wouldn't want to repeat those mutations in the event of a later error!

Form Beans

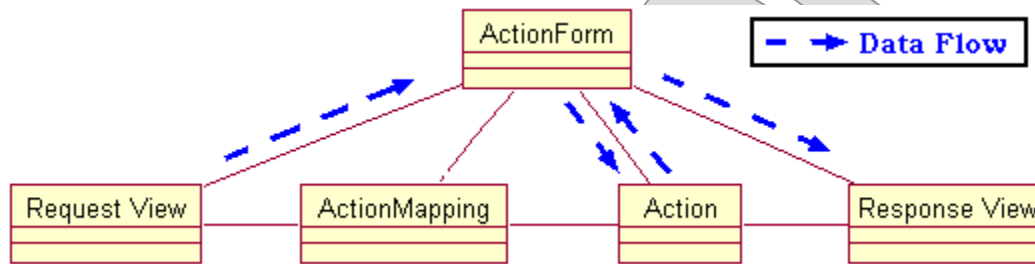
- Struts provides a more elegant solution to these problems using **form beans** – JavaBeans which extend the **ActionForm** class.
- Form beans “glue” the controller to the view in two ways:
 - Manages processing of form input, including optional validation of input values.
 - Manages production of new views populated with values from earlier submissions.
- Thus the form bean becomes a sort of universal bus for data pertaining to a given request/response cycle:



- The transfer of data (shown by dashed blue arrows above) is more or less automatic for different paths, as we’ll see.

The Action Form Class

- At its simplest, an **ActionForm** encapsulates an HTML form and the behaviors surrounding it, and makes it a true class in the application design.



- The developer now treats forms as full-fledged, stateful objects, rather than as flat streams of characters that must be painstakingly managed coming in and going out.
- There are capabilities beyond this, and experienced Struts developers often break the one-to-one relationship of HTML form to action form to take best advantage of the framework.
 - We'll work through progressively more sophisticated uses of action forms and their associated **form beans** throughout the course.

Declaring an Action Form

- Action forms are declared in the Struts configuration in a special section `<form-beans>`.
 - Child elements are of type `<form-bean>` and declare a supporting **ActionForm** class.
 - They also give the bean a name for reference elsewhere in the configuration.

```
<form-beans>
  <form-bean name="myData" type="com.my.MyData" />
</form-beans>
```

- An action mapping puts an action form into play by associating it with a particular request URI, controller, and view:
 - The **name** attribute of the `<action>` element refers to a declared form bean by its name.
 - The **scope** attribute then defines either “request” or “session” scope for the bean that will be instantiated.

```
<action
  path="/DoSomething"
  type="org.apache.struts.actions.ForwardAction"
  parameter="/Done.jsp"
  name="myData"
  scope="request"
/>
```

Relationship to Input

- At request time, Struts will assure that an instance of the declared form bean is available to gather request parameters.
 - If the scope is “request”, a new bean will be used, or perhaps an old one will be recycled and reset to initial values.
 - If the scope is “session”, a single instance of the bean will be shared between all action mappings that make the same name and scope declaration.
- The Struts framework will transfer information from the HTTP request (CGI string for HTTP GET, message body for HTTP POST) to the bean.
 - For each parameter, it will look for a bean property of the same name.
 - It will then call the mutator method on the bean, converting the argument type to fit the method signature. (That is, if the property is of type `int`, the framework is smart enough to do the `Integer.parseInt` itself, before calling the mutator.)
 - Only certain property types are supported, but the set includes all primitives, their corresponding “box” object types (such as `Integer`), strings, dates, and, most importantly, other JavaBeans.
 - The parameter and property sets needn’t match exactly: parameters for which there is no mutable property will be ignored, and properties for which there is no parameter will be left as they were.

Relationship to Actions

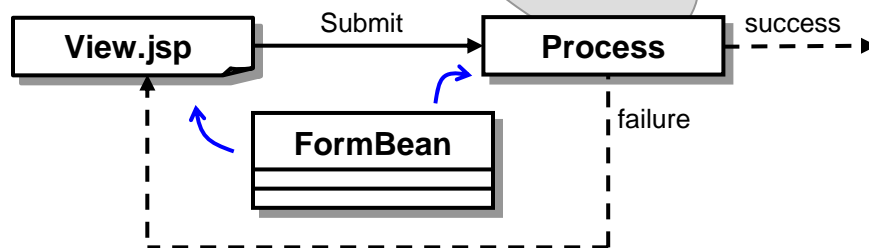
- Recall that the action form is passed as a parameter to **Action.execute**.
- Thus the action can query the action form instead of the request itself.
 - This is usually easier and less error-prone.
 - The action can mix and match reading request parameters directly and seeing them through the lens of the action form.
- Action code can also modify the form's properties.
 - This can be an excellent way to provide information to the next view in a scenario.
 - For instance, in a multi-page wizard interface, a controller might “fix up” the value of a phone number to include or remove parentheses and dashes as desired.
 - It might fill in default values for shipping address based on a user's persistent account information.

Relationship to the Model

- Although they are technically controller components – and should be scrupulously held in that area of the system – form beans provide an excellent point of contact with the model.
 - Very often, the same decomposition into classes that informs the model’s design will make sense in the presentation.
 - Thus form beans will naturally evolve that hold similar information, at a similar granularity, to the business beans that make up part of the model.
- Because an action form needs to offer certain generic behaviors to the framework code, it cannot be just any old JavaBean.
- There are many techniques for sewing the form beans to the business beans, each of which offers a different blend of robustness and coding convenience.
 - Business beans can be exposed as **nested properties** on form beans, so that bean values are found through expressions such as **formBean.value.interestingProperty**.
 - Business beans can be held privately as fields by the form beans, and accessors and mutators can delegate to the business beans.
 - A middle approach is to offer the **value** property on the form bean and to take a snapshot (or to produce one) of the business bean’s values, but not to hold on to the bean itself.

Relationship to Output

- When a view is finally served as the response, it has access to the form bean and its properties.
- It can produce dynamic values to HTML by a number of techniques:
 - The Struts HTML tags automatically find whatever form bean is in play and read from it properties whose names are declared as attributes to the tags themselves.
 - This neatly combines input and output in the same tag when building HTML forms: values from the form bean in force when the page is served are pre-populated into the form, and its values are automatically available to the form bean that applies to the next request.
 - JSP2 expressions and JSTL tags can read the form bean's properties explicitly, which can be useful at other points in the page where one doesn't want to build a form but just to produce dynamic values.
- We'll consider these in more detail in later chapters.
- One significant consequence of using form beans, then, is that values from a view can be fed back to that view without relying on a preconditioning controller:



Ellipsoid Form Bean

EXAMPLE

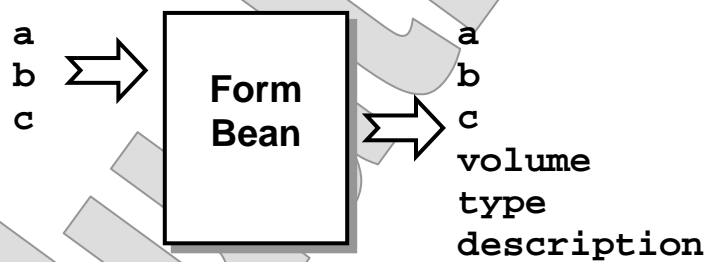
- **Examples\Ellipsoid\Step2**, which we've already reviewed for its use of a forwarding action, also uses a form bean to communicate between input form and results page.
- Rather than providing a link to the model, this form bean includes the business logic itself.
 - This is not good practice!
 - It's a simple solution for the moment, and later in the course we'll look at a more rigorous implementation that restores the use of the business bean **Ellipsoid** while continuing to use a form bean for fluid communication over a request.
- **Relevant sections of the Struts configuration:**

```
<form-beans>
  <form-bean name="Ellipsoid"
             type="cc.math.EllipsoidForm" />
</form-beans>
...
<action
  path="/Compute"
  type="org.apache.struts.actions.ForwardAction"
  parameter="/Results.jsp"
  name="Ellipsoid"
  scope="request "
/>
```

Ellipsoid Form Bean

EXAMPLE

- At runtime, the parameters **a**, **b**, and **c** as encoded in the HTTP request will be transferred to the corresponding properties on the form bean.
- Then **Results.jsp** simply requests these properties back, plus three new ones, which are the values the user was hoping to calculate.



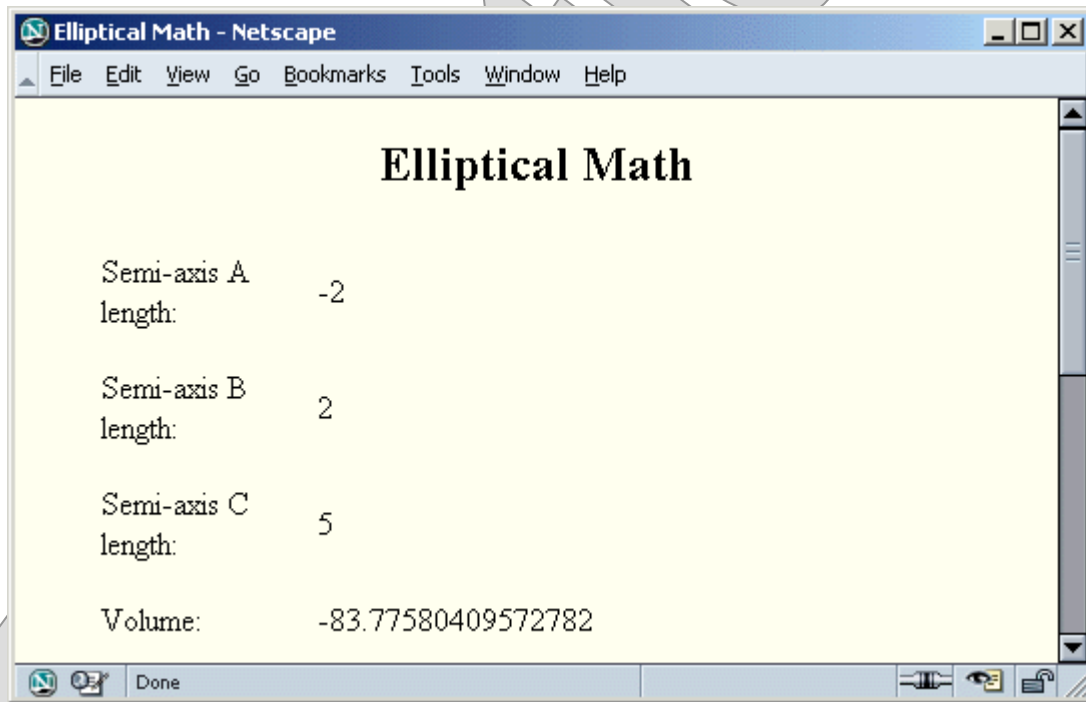
```
public double getVolume ()
{
    double a = Double.parseDouble (this.a);
    double b = Double.parseDouble (this.b);
    double c = Double.parseDouble (this.c);

    return 4 * Math.PI * a * b * c / 3;
}
```

Dropping Exceptions

EXAMPLE

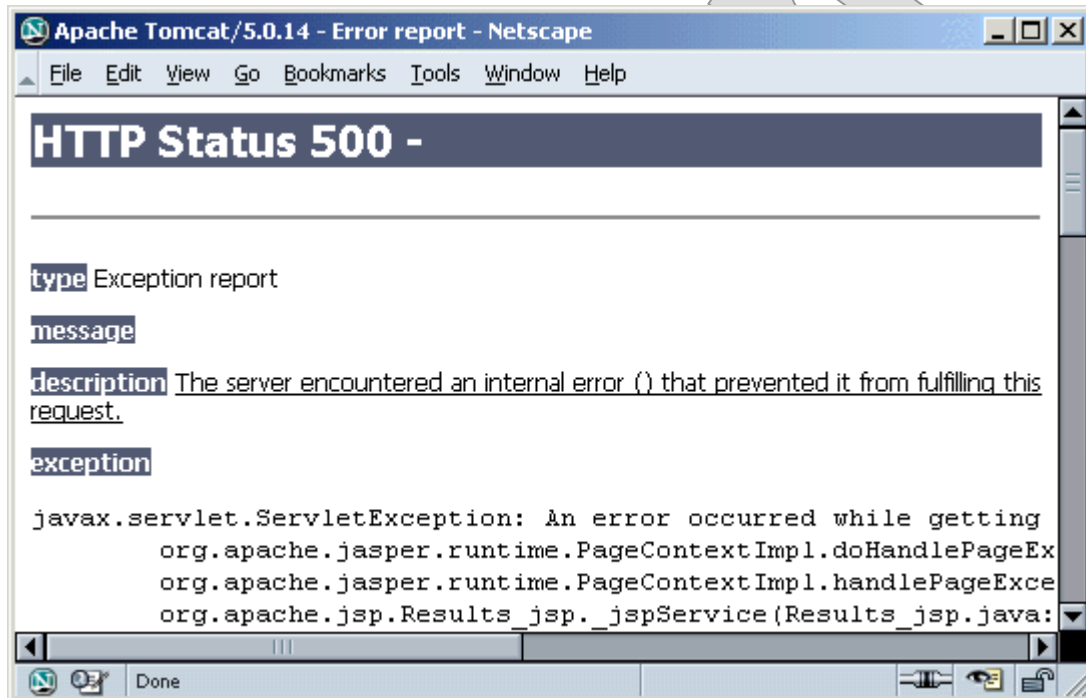
- Recall our exception-handling exercise with Ellipsoid Step 1.
- Try the same experiments with this version ...
 - Negative numbers go right through. This is because the new action form class is more lenient about input values, and will consider the possibility of a negative length and volume.



Dropping Exceptions

EXAMPLE

- Provide no value at all, however, and you'll see a blowup. Why is this, if the global exception handler is still in place?



Dropping Exceptions

EXAMPLE

- Remember that declarative exception handling is only put in place to cover the call to **Action.execute**.
- Form beans are populated by the framework, based on request parameters, **before** the action is invoked.
 - Any exceptions encountered there would not be caught by declared handlers.
- In this case, the failure to convert to a number is deferred, and occurs only when the **volume** or **type** properties are requested
 - see the code on the previous page.
 - This occurs **after** the action has been invoked, and so again we're flying without a net.
- Thus global exception handling is no panacea.
 - Actions enjoy good protection.
 - Form beans and other parts of the request/response cycle are still on their own!
 - This means decisions must be made carefully about how strict or lenient to be

Up, Periscope!

LAB 3A

In this lab you will create a Struts action class that does the same data capture that the **periscope** custom tag does at page-production time. This will enable you to snoop on request and session attributes earlier in the request-processing cycle, and this in turn will illustrate a bit more about the Struts framework and how it uses form beans. You will apply your new **Periscope** action to a copy of the Ellipsoid application, and watch Struts work.

It is also possible to skip the coding of the **Periscope** action, and simply to test the answer project, in order to focus on inspecting Struts' use of form beans.

Detailed instructions are contained in the Lab 3A write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluate Only

LoveNoteBean

LAB 3B

In this lab you will revise Love Is Blind to use a form bean to manage the input fields that make up a love note. The resulting application will behave the same way – one slight enhancement – but the form bean will provide the basis for later enhancements.

Detailed instructions are contained in the Lab 3B write-up at the end of the chapter.

Suggested time: 45 minutes.

Evaluation Only

Declaring Form Properties

- Use of a JavaBean can be highly automated.
- Building bean classes in Java can be a repetitive and time-consuming task, however.
 - Especially where the purpose of the bean is not to implement interesting logic around these properties, as it would be for a business bean, the implementations of accessor and mutator methods are usually about identical, except for the property name and perhaps the type.
- Another way to look at this – and it’s a happier way by far – is that bean implementation can also be automated.
- Struts facilitates exactly this, by reading `<form-property>` child elements of a `<form-bean>`.

```
<form-bean
  name="myBean"
  type="org.apache.struts.action.DynaActionForm"
>
  <form-property name="height"
    type="java.lang.String" />
  <form-property name="width"
    type="java.lang.String" />
</form-bean>
```

- Declaring properties then is as simple as defining **name** and **type** attributes.
- Now, one doesn’t have to write the **ActionForm** subclass at all. Instead, declare and use the Struts-implemented **dynamic action form** class.

The DynaActionForm Class

- Declare a dynamic form bean by setting its type to **org.apache.struts.action.DynaActionForm**.
- This class implements a dynamic pair of methods **get** and **set**, which are **weakly typed** compared to ordinary action forms and their methods.
 - Each method takes an additional string parameter compared to the **strongly typed** form-bean methods:

```
public Object get (String propertyName);
public void set (String propertyName, Object val);
```

- The method implementations consult the form-bean configuration and manage properties as called out by the **<form-property>** elements.
- Although the Struts request processor knows how to use this weakly-typed interface automatically, using **DynaActionForm** does require different invocation syntax from action classes and JSPs.
 - Java code in action classes must be rewritten to use **get** instead of **getXXX**, **set** instead of **setXXX**.

```
OLD: getAddress ()           NEW: get ("address")
```

- JSPs using Struts custom tags work either way, with no changes, but JSP expressions and JSTL tags need to address a special property **map** to read and write dynamic properties.

```
OLD: ${A.B}                 NEW: ${A.map.B}
```

Map-Backed Forms

- Really **DynaActionForm** is not dynamic!
 - It is only transferring property definition from Java to XML.
 - The properties are still static – that is, defined at build time.
- A truly dynamic approach is supported, allowing properties to be defined at runtime in a Java **Map**.
 - One or more bean properties can have type **java.util.Map**.
 - Property names and values are then built into this map.
 - Maps are treated specially in the expression syntax of various JSP expression types: Struts HTML, JSTL, and JSP EL.
- There is no special property name that is treated differently to allow dynamic properties to work like static ones as defined in normal **ActionForms**.
 - That is, one must choose an intermediate property such as **myMap**, and action code and JSPs must read this property explicitly, as in **myBean.myMap.myProperty**.
 - This is exactly what **DynaActionForm** is doing with its **map** property.
- This approach is overkill when the property names can be known at build time.
- It is meant for form beans that support dynamically-generated HTML pages, on which the form inputs themselves are not known in advance.

A Dynamic LoveNoteBean

LAB 3C

Optional

In this lab you will re-implement **LoveNoteBean** to use **DynaActionForm**. You will do this in two steps: first you will switch to **DynaActionForm** directly, leaving **LoveNoteBean** out of the process; then you will make **LoveNoteBean** subclass **DynaActionForm**, so as to get the declarative properties but still provide the helper method **getLoveNote** that's already in use in the **SendLoveNote** action.

Detailed instructions are contained in the Lab 3C write-up at the end of the chapter.

Suggested time: 30 minutes.

Evaluated Only

Validation

- **Form beans are the basis of input validation in Struts.**
 - **ActionForm** defines a method **validate** which can be implemented by subtypes.
 - Form beans are the natural place for data validation, as they encapsulate validation behavior with the target state.
- **We'll take a close look at validation later on, but a synopsis of the process follows:**
 - Struts processes a request via the **ActionServlet**.
 - If a form bean is involved, it is instantiated (if necessary), automatically populated.
 - Then the action servlet calls the form bean's **validate** method.

```
public ActionErrors validate  
(ActionMapping mapping,  
 HttpServletRequest request);
```

- If the returned errors reference is null or an empty collection, processing proceeds to the action; otherwise it is automatically routed to a path defined in the action mapping by the **input** parameter.
- **The Struts Validator extends this framework by hooking into validation rules declared in XML.**

Strong Typing?

- JavaBeans can expose properties of any Java type.
- Struts officially supports a subset of possible types, including primitives, strings, dates, and other beans.
- Practically speaking, though, only strings and booleans are reliable for read/write properties on form beans.
- Why?
 - The Struts framework passes raw request data to mutator methods on the form bean.
 - It makes its best effort at converting the strings from the HTTP request into the appropriate type for the mutator method – parsing as a number or a date, for instance.
 - If the parsing fails, the servlet container will catch a Java exception – this will fall through all Struts' handlers.
 - Note, too, from the discussion on the previous page, that input validation (which could catch these sorts of problems, and handle them better) is invoked after the transfer. By then, it's too late!

Form Bean Design

- As a result, form beans declare properties that are meant to interact with HTML forms as simple strings, booleans, or nested bean types.
 - Thus the relationship with input and output data is a bit dumbed-down – very weakly typed.
- Form beans can still expose properties of other types for use by controller components, and that interact with model components.
 - The classic strategy uses **helper properties** that are for use only in action classes or by other beans.
 - Often these helpers are backed by the same physical state: a property **size** might coexist with **sizeAsInt**, each functioning as a different window on the same Java field. (A mini-MVC system!)
 - Care should be taken with names so that the Struts framework is never put in a position to mistake a helper function as the target for HTTP data transfer.

Coarse-Grained Action Forms

- Although at first a one-to-one relationship between HTML forms and form beans seems most natural, after a while one looks up and notices that there's no rule against assigning multiple actions to one bean.
 - Especially where multiple HTML pages and forms are presented to the user to assemble what the application sees as one logically-related data record, one form bean per form is architecturally limiting.
 - On the flip side, an important limitation does exist: there can be no more than one form bean per action mapping.
 - When the form in question is submitting some values that belong on a certain form bean, but also provides other values, practically speaking these must be captured on that form bean, or be relegated to raw HTTP request parameters.
- These forces drive Struts development towards a strategy that uses **coarse-grained** beans.
 - Multiple actions and pages share the semantics of a single bean class, whether static or dynamic.
 - They may also share the state of that bean over multiple requests, at session scope, or they may just use the same type in individual requests.
 - In fact, it's been suggested that all a Struts application really needs is a single form bean, with as many properties as are called for by the union of all HTML forms in the system!

The reset Method

- The method **ActionForm.reset** is called by the request processor just before populating the bean's properties.
 - It will also be called when the bean is first instantiated.
- By default, the method does nothing.
- It can be implemented to clear certain properties, or to provide initial values for them.
- It is usually not necessary to implement **reset**.
- For a request-scope bean it may be useful to set a default value here or there.
- **Beware of implementing this method for a bean that will be used at session scope.**
 - The request processor will call **reset** regardless of bean scope.
 - If various pages are expecting to share information over the session by way of a form bean, and that form bean's **reset** method alters the information, things will get very tricky.

SUMMARY

- **Action forms provide a sort of data bus to which HTML forms and Struts actions can attach for both input and output.**
 - HTML forms, Struts and JSTL tags and JSP expressions interact with string-type JavaBeans properties.
 - Actions can use more sophisticated property types, as well as ordinary (non-JavaBeans) methods.
- **Form beans are controller components, closely aligned with actions themselves.**
- **They are also a natural point of implementation for adapting business beans in the Struts model.**
 - We've considered a few of the more straightforward strategies for establishing fluid transfer of information between form beans and business beans.
 - Later in the course we'll look at a few more, and especially consider a reflection-based approach facilitated by one of the Struts utility classes.
- **Coarse-grained form beans are a preferred practice, as they facilitate greater and easier information sharing and impose no performance or design penalty.**

Up, Periscope!

LAB 3A

Introduction

In this lab you will create a Struts action class that does the same data capture that the **periscope** custom tag does at page-production time. This will enable you to snoop on request and session attributes earlier in the request-processing cycle, and this in turn will illustrate a bit more about the Struts framework and how it uses form beans. You will apply your new **Periscope** action to a copy of the Ellipsoid application, and watch Struts work.

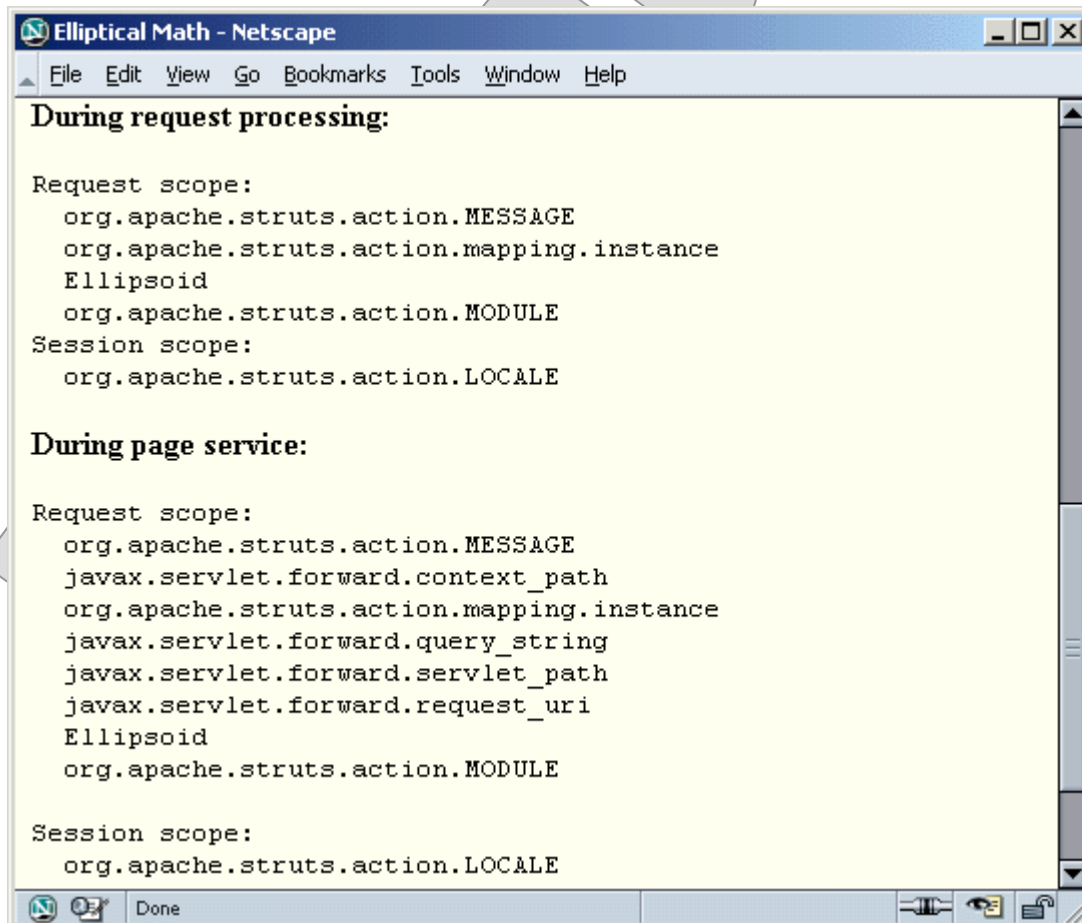
Suggested Time: 30 minutes.

Root Directory:	Capstone\Struts	
Directories:	Labs\Lab03A	(do your work here)
	Examples\Periscope\Step1	(backup copy of starter files)
	Examples\Periscope\Step2	(contains lab solution)
Files:	cc\tools\web\Periscope.java	
	Config\struts-config.xml	

Instructions

1. The starter code is just a copy of the Ellipsoid application, plus a partial source file for the new diagnostic action class. Open **Periscope.java** and implement **execute**: for each scope, get the list of attribute names (both request and session offer the same method, **getAttributeNames**, which returns an **Enumeration**) and build a string that lists these names in some clear formatting.
2. Set the string into a new request attribute “periscope”.
3. Return the “success” forward.
4. Open **struts-config.xml** and change the action mapping to invoke your new class instead of a **ForwardAction**. Remember that you’ll have to switch from a **parameter** pathname to a **<forward>** child element.

5. Note that both JSPs invoke the `<cc:periscope>` custom tag at the bottom of the page. **Results.jsp** also reads out the value of the “periscope” attribute. Thus when you test the application you will see the progress of shared objects at three stages:
 - a. Prior to the first Struts request
 - b. During request processing, when the action is invoked
 - c. Post-processing, as the second JSP is being served
6. Build and deploy the application with **ant all**.
7. Test at the following URL, and enter valid values for the three semi-axes. Note that prior to the computation request, there is nothing at all at request or session scope.
8. Submit the request and at the bottom of the page you’ll see the objects that Struts creates, several of which have to do with form processing and action forms. Note especially that the form bean is available by its declared name at the declared scope:



The screenshot shows a Netscape browser window titled "Elliptical Math - Netscape". The browser's menu bar includes File, Edit, View, Go, Bookmarks, Tools, Window, and Help. The main content area displays the output of the `<cc:periscope>` custom tag, which is organized into two sections: "During request processing:" and "During page service:". Each section lists objects at the Request scope and Session scope.

```
During request processing:

Request scope:
  org.apache.struts.action.MESSAGE
  org.apache.struts.action.mapping.instance
  Ellipsoid
  org.apache.struts.action.MODULE
Session scope:
  org.apache.struts.action.LOCALE

During page service:

Request scope:
  org.apache.struts.action.MESSAGE
  javax.servlet.forward.context_path
  org.apache.struts.action.mapping.instance
  javax.servlet.forward.query_string
  javax.servlet.forward.servlet_path
  javax.servlet.forward.request_uri
  Ellipsoid
  org.apache.struts.action.MODULE

Session scope:
  org.apache.struts.action.LOCALE
```

LoveNoteBean

LAB 3B

Introduction

In this lab you will revise Love Is Blind to use a form bean to manage the input fields that make up a love note. The resulting application will behave the same way – one slight enhancement – but the form bean will provide the basis for later enhancements.

Suggested Time: 45 minutes.

Root Directory: Capstone\Struts

Directories: Labs\Lab03B (do your work here)
Examples\LoveIsBlind_XML\Step3 (backup of starter)
Examples\LoveIsBlind_XML\Step4 (lab solution)

Files: cc\LIB\form\LoveNoteBean.java
cc\LIB\action\SendLoveNote.java
Config\struts-config.xml
View\WalkIn\Confirm.jsp

Instructions

1. Create a new source file **cc\LIB\form\LoveNoteBean.java**. (This is a new package for the application, so you'll have to create the **form** subdirectory.)
2. Create the **LoveNoteBean** class as a subclass of **ActionForm**. You'll need to import the **org.apache.struts.action** package.
3. Give the class a private field for each of the three properties **suitor**, **email** and **message**. All three are strings.
4. For each property, implement an accessor method and a mutator method, according to JavaBeans conventions.
5. Provide another, read-only property by implementing the method **getLoveNote**. This will return an object of type **cc.LIB.LoveNote**. (This is a variant of one of the strategies discussed earlier for communicating between form beans and business beans.) The method can create a new object, passing the three fields in the order listed above to the constructor, and return it directly to the caller.
6. Open **SendLoveNote.java** and re-implement **execute** to take advantage of the new form class. Since the form bean can provide an instance of the appropriate domain object, the action no longer needs to construct one, nor to deal with the request parameters. Simply downcast the passed **form** to **LoveNoteBean** and then call **getLoveNote**; pass that value to **member.addLoveNote**.

7. Configure the new form bean – what does this entail? There are two steps that need to be performed in **struts-config.xml** – can you remember what they are?
8. You need to declare the form bean, and then you need to assign it to the action mapping. First, create a **<form-beans>** section in the configuration file. According to the configuration-file DTD, this must appear before the **<global-exceptions>** element already in the file.
9. Create a child element **<form-bean>** and give it attributes **name** (“loveNote”) and **type** (your new class’ name).
10. Find the action mapping for “/WalkIn/SendLoveNote” and add attributes **name** and **scope** (which can be “request”).

11. Open **build.properties** and add the path **cc/LIB/form** to the **src** variable:

```
src=cc/LIB:cc/LIB/action:cc/LIB/form:cc/tools/XML
```

12. Build and deploy the application, and test that it still behaves as it should. Thus far, you’ve simply replaced old logic with new – better, stronger logic that does exactly the same thing.
13. Now, let’s take a little advantage. Open **Confirm.jsp** and add a reproduction of the message that was submitted, to give the confirmation a little credibility. After the paragraph saying the message was delivered, add the following code:

```
<blockquote>  
  ${loveNote.message}  
</blockquote>
```

14. This JSP2 expression simply reads a property out of the form bean ... but how would we have done this before? (There is a fairly simple way, actually – can you think what it would be?)

15. Build and retest, and you should see your enhanced confirmation:



(with apologies to Wilco)

A Dynamic LoveNoteBean

LAB 3C

Introduction

In this lab you will re-implement **LoveNoteBean** to use **DynaActionForm**. You will do this in two steps: first you will switch to **DynaActionForm** directly, leaving **LoveNoteBean** out of the process; then you will make **LoveNoteBean** subclass **DynaActionForm**, so as to get the declarative properties but still provide the helper method **getLoveNote** that's already in use in the **SendLoveNote** action.

Suggested Time: 30 minutes.

Root Directory: Capstone\Struts

Directories: Labs\Lab03C (do your work here)
Examples\LoveIsBlind_XML\Step4 (backup of starter)
Examples\LoveIsBlind_XML\Step5 (intermediate answer)
Examples\LoveIsBlind_XML\Step6 (lab solution)

Files: cc\LIB\form\LoveNoteBean.java
cc\LIB\action\SendLoveNote.java
Config\struts-config.xml
View\WalkIn\Confirm.jsp

Instructions

1. In **struts-config.xml**, substitute **DynaActionForm** for **LoveNoteBean** as the type of the **loveNote** form bean, and declare the three bean properties, as shown below.

```
<form-beans>
  <form-bean name="loveNote"
    type="org.apache.struts.action.DynaActionForm" >
    <form-property name="suitor" type="java.lang.String" />
    <form-property name="email" type="java.lang.String" />
    <form-property name="message" type="java.lang.String" />
  </form-bean>
</form-beans>
```

2. Ask yourself: what would happen if you built and tested the application right now?
3. Answer: The application would compile, but at runtime the typecast in **SendLoveNote** would fail, because now the real type of the form bean is **DynaActionForm**.
4. Open **SendLoveNote.java** and fix the code that adds the love note. Start by commenting out the existing line of code that calls **addLoveNote**.

5. Just after the old code, add a local reference **dyna** of type **DynaActionForm**, and initializing it to **form**, downcast appropriately.
6. Now, instead of calling **getLoveNote** as before, create your own **cc.LIB.LoveNote**, deriving each of the three parameters by calling **dyna.get** and passing the name of the property in question. The return from **dyna.get** will have to be downcast to type **String**. Pass the new **LoveNote** to **member.addLoveNote**, as before.
7. Build using **ant all** and test. You should find that you can add a love note without difficulty, but that when **Confirm.jsp** is served, it produces a runtime error. Can you decipher the error report?
8. Recall that a JSP expression must be adjusted to use properties in a **DynaActionForm**. Find the expression that produces the love-note message, and add the intermediate property **map** to it, as shown below:

```
#{loveNote.map.message}
```

9. Build again and you should get a successful test. (This is the intermediate answer in **Step5**.)
10. It would be nice to get that helper function **getLoveNote** back, so that action classes wouldn't have to interrogate the form bean for its properties and build the love note themselves. Open **LoveNoteBean.java** and change the base class from **ActionForm** to **DynaActionForm**.
11. Delete all six of the **get/set** methods for the properties **suitor**, **email**, and **message**.
12. Delete the three fields at the bottom of the source file.
13. Re-implement **getLoveNote** using code very much like what you wrote in **SendLoveNote**: use the **get** method and downcast to **String** for each property. You won't need the **dyna** part, however, since you are operating in the form bean class now.
14. In **SendLoveNote.java**, remove your new code and un-comment the old implementation, which should work just as it did before. Right?
15. Rebuild and test. Still something missing here, and the class-cast exception that you get should remind you what to do ...
16. Change the declared form-bean class back to **LoveNoteBean** in the config file. (Leave the **<form-property>** elements!) Build and test, and it should work cleanly now.