Building Web Applications with Servlets & JSPs

2nd Edition
Complete Coverage of 1.1

Programming

Jakarta Struts

O'REILLY®

Chuck Cavaness

CHAPTER 11

The Validator Framework

The Struts framework allows input validation to occur inside the `ActionForm`. To perform validation on data passed to a Struts application, developers must code special validation logic inside each `ActionForm` class. Although this approach works, it has some serious limitations. This chapter introduces David Winterfeldt's Validator framework, which was created specifically to work with the Struts components and to help overcome some of these limitations.

The Validator allows you to declaratively configure validation routines for a Struts application without programming special validation logic. The Validator has become so popular and widely used by Struts developers that it has been added to the list of Jakarta projects and to the main Struts distribution.

The Need for a Validation Framework

Chapter 7 discussed how to provide validation logic inside the `ActionForm` class. The solution presented there requires you to write a separate piece of validation logic for each property that you need to validate. If an error is detected, you have to manually create an `ActionError` object and add it to the `ActionErrors` collection. Although this solution works, there are a few problems with the approach.

The first problem is that coding validation logic within each `ActionForm` places redundant validation logic throughout your application. Within a single web application, the type of validation that needs to occur across HTML forms is very similar. The need to validate required fields, dates, times, and numbers, for example, typically occurs in many places throughout an application. Most nontrivial applications have multiple HTML forms that accept user input that must be validated. Even if you use a single `ActionForm` for your entire application, you might still end up duplicating the validation logic for the various properties.

The second major problem is one of maintenance. If you need to modify or enhance the validation that occurs for an `ActionForm`, the source code must be recompiled. This makes it very difficult to configure an application.

The Validator framework allows you to move all the validation logic completely outside of the `ActionForm` and declaratively configure it for an application through external XML files. No validation logic is necessary in the `ActionForm`, which makes your application easier to develop and maintain. The other great benefit of the Validator is that it's very extensible. It provides many standard validation routines out of the box, but if you require additional validation rules, the framework is easy to extend and provides the ability to plug in your own rules (again without needing to modify your application).

Installing and Configuring the Validator

The Validator framework is now part of the Jakarta Commons project. It's included with the Struts main distribution, but you can also get the latest version from the Commons download page at *http://jakarta.apache.org/commons/*. Unless you need the source code or the absolute latest version, you'll find all the necessary files included with the Struts 1.1 distribution.

Required Packages

The Validator depends on several other packages to function properly, and the most important of these is the Jakarta ORO package. The ORO package contains functionality for regular expressions, performing substitutions, and text splitting, among other utilities. The libraries were originally developed by ORO, Inc. and donated to the Apache Software Foundation. Earlier versions of the Validator framework depended on a different regular expression package, called Regexp, which is also a Jakarta project. However, ORO was considered the more complete of the two, and the Validator that is included with Struts 1.1 now depends on the ORO package.

Other packages required by the Validator are Commons BeansUtils, Commons Logging, Commons Collections, and Digester. All of the dependent packages for the Validator are included in the Struts 1.1 download. The *commons-validator.jar* and *jakarta-oro.jar* files need to be placed into the *WEB-INF/lib* directory for your web application. The other dependent JAR files must also be present, but they should already be there due to Struts framework requirements.

Configuring the Validation Rules

As mentioned earlier, the Validator framework allows the validation rules for an application to be declaratively configured. This means that they are specified externally to the application source. There are two important configuration files for the Validator framework: *validation-rules.xml* and *validation.xml*.

The validation-rules.xml file

The *validation-rules.xml* configuration file contains a global set of validation rules that can be used out of the box by your application. This file is application-neutral and can be used by any Struts application. You should need to modify this file only if you plan to modify or extend the default set of rules.

If you do need to extend the default rules, you might be better off putting your custom rules in a different XML file, so as to keep them separate from the default ones. This will help when it comes time to upgrade to a newer version of the Validator framework.

The *validator-rules_1_1.dtd* describes the syntax of the *validation-rules.xml* file. The root element is the `form-validation` element, which requires one or more `global` elements:

```
<!ELEMENT form-validation (global+)>
<!ELEMENT global (validator+)>
```

Each `validator` element describes one unique validation rule. The following fragment from the *validation-rules.xml* file is the definition for the `required` validation rule:

```
<validator
  name="required"
  classname="org.apache.struts.util.StrutsValidator"
  method="validateRequired"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionErrors,
    javax.servlet.http.HttpServletRequest"
  msg="errors.required">
</validator>
```

The `validator` element also allows a `javascript` subelement, but for the sake of brevity it is not shown here. The JavaScript support in the Validator framework is discussed later in the chapter.

The `validator` element supports seven attributes, as shown here:

```
<!ATTLIST validator name           CDATA #REQUIRED
                    classname      CDATA #REQUIRED
                    method         CDATA #REQUIRED
                    methodParams   CDATA #REQUIRED
                    msg            CDATA #REQUIRED
                    depends        CDATA #IMPLIED
                    jsFunctionName CDATA #IMPLIED>
```

The `name` attribute assigns a logical name to the validation rule. It is used to reference the rule from other rules within this file and from the application-specific validation file discussed in the next section. The name must be unique.

The `classname` and `method` attributes define the class and method that contain the logic for the validation rule. For example, as shown in the earlier code fragment, the `validateRequired()` method in the `StrutsValidator` class will be invoked for the `required` validation rule. The `methodParams` attribute is a comma-delimited list of parameters for the method defined in the `method` attribute.

The `msg` attribute is a key from the resource bundle. The Validator framework uses this value to look up a message from the Struts resource bundle when a validation error occurs. By default, the Validator framework uses the following values:

```
errors.required={0} is required.
errors.minlength={0} cannot be less than {1} characters.
errors.maxlength={0} cannot be greater than {1} characters.
errors.invalid={0} is invalid.
errors.byte={0} must be a byte.
errors.short={0} must be a short.
errors.integer={0} must be an integer.
errors.long={0} must be a long.
errors.float={0} must be a float.
errors.double={0} must be a double.
errors.date={0} is not a date.
errors.range={0} is not in the range {1} through {2}.
errors.creditcard={0} is not a valid credit card number.
errors.email={0} is an invalid email address
```

You should add these to your application's resource bundle, or change the key values in the *validation-rules.xml* file if you plan to use alternate messages.

The `depends` attribute is used to specify other validation rules that should be called before the rule specifying it. The `depends` attribute is illustrated in the `minLength` validation rule here:

```
<validator
  name="minLength"
  classname="org.apache.struts.util.StrutsValidator"
  method="validateMinLength"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionErrors,
    javax.servlet.http.HttpServletRequest"
  depends="required"
  msg="errors.minlength">
</validator>
```

Before the `minLength` validation rule is called, the `required` rule will be invoked. You can also set up a rule to depend on multiple rules by separating the rules in the `depends` attribute with a comma:

```
depends="required,integer"
```

If a rule that is specified in the `depends` attribute fails validation, the next rule will not be called. For example, in the `minLength` validation rule shown previously, the

validateMinLength( ) method will not be invoked if the required validation rule fails. This should stand to reason, because there's no sense in checking the length of a value if no value is present.

> The final attribute supported by the validator element is the jsFunctionName attribute. This optional attribute allows you to specify the name of the JavaScript function. By default, the Validator action name is used.

The Validator framework is fairly generic. It contains very basic, atomic rules that can be used by any application. As you'll see later in this chapter, it's this generic quality that allows it to be used with non-Struts applications as well. The org.apache.commons.Validator.GenericValidator class implements the generic rules as a set of public static methods. Table 11-1 lists the set of validation rules available in the GenericValidator class.

Table 11-1. Validation rules in the GenericValidator class

Method name	Description
isBlankOrNull	Checks if the field isn't null and the length of the field is greater than zero, not including whitespace.
isByte	Checks if the value can safely be converted to a byte primitive.
isCreditCard	Checks if the field is a valid credit card number.
isDate	Checks if the field is a valid date.
isDouble	Checks if the value can safely be converted to a double primitive.
isEmail	Checks if the field is a valid email address.
isFloat	Checks if the value can safely be converted to a float primitive.
isInRange	Checks if the value is within a minimum and maximum range.
isInt	Checks if the value can safely be converted to an int primitive.
isLong	Checks if the value can safely be converted to a long primitive.
isShort	Checks if the value can safely be converted to a short primitive.
matchRegex p	Checks if the value matches the regular expression.
maxLength	Checks if the value's length is less than or equal to the maximum.
minLength	Checks if the value's length is greater than or equal to the minimum.

Because the validation rules in the GenericValidator are so fine-grained, the Struts developers added a utility class to the Struts framework called org.apache.struts.util.StrutsValidator, which defines a set of higher-level methods that are coupled to the Struts framework but make it easier to use the Validator with Struts. They are listed here without descriptions because the names are similar enough to the ones from Table 11-1 to indicate their functionality.

- validateByte
- validateCreditCard
- validateDate

- validateDouble
- validateEmail
- validateFloat
- validateInteger
- validateLong
- validateMask
- validateMinLength
- validateMaxLength
- validateRange
- validateRequired
- validateShort

The StrutsValidator class contains the concrete validation logic used by Struts. This class and the methods listed above are declaratively configured in the *validation-rules.xml* file. When one of these methods is invoked and the validation fails, an ActionError is automatically created and added to the ActionErrors object. These errors are stored in the request and made available to the view components.

The validation.xml file

The second configuration file that is required by the Validator framework is the *validation.xml* file. This file is application-specific; it describes which validation rules from the *validation-rules.xml* file are used by a particular ActionForm. This is what is meant by declaratively configured—you don't have to put code inside of the ActionForm class. The validation logic is associated with one or more ActionForm classes through this external file.

The *validation.xml* file is governed by the *validation_1_1.dtd*. The outermost element is the form-validation element, which can contain two child elements, global and formset. The global element can be present zero or more times, while the formset element can be present one or more times:

```
<!ELEMENT form-validation (global*, formset+)>
```

The global element allows you to configure constant elements that can be used throughout the rest of the file:

```
<!ELEMENT global (constant*)>
```

This is analogous to how you might define a constant in a Java file and then use it throughout the class. The following fragment shows a global fragment that defines two constants:

```
<global>
 <constant>
  <constant-name>phone</constant-name>
```

```
      <constant-value>^\(?(\d{3})\)?[-| ]?(\d{3})[-| ]?(\d{4})$</constant-value>
     </constant>
     <constant>
      <constant-name>zip</constant-name>
      <constant-value>^\d{5}(-\d{4})?$</constant-value>
     </constant>
    </global>
```

This fragment includes two constants, `phone` and `zip`, although you can include as many as you need. These constants are available to the elements within the `formset` section. You can reuse them many times within the `formset` simply by referring to them by name.

This is best illustrated with an example. Example 11-1 shows a simple *validation.xml* file.

Example 11-1. A simple validation.xml file

```
<form-validation>
 <global>
  <constant>
   <constant-name>phone</constant-name>
   <constant-value>^\(?(\d{3})\)?[-| ]?(\d{3})[-| ]?(\d{4})$</constant-value>
  </constant>
 </global>
 <formset>
  <form name="checkoutForm">
   <field
     property="phone"
     depends="required,mask">
     <arg0 key="registrationForm.firstname.displayname"/>
     <var>
       <var-name>mask</var-name>
       <var-value>${phone}</var-value>
     </var>
    </field>
   </form>
  </formset>
</form-validation>
```

In Example 11-1, the `phone` constant that's declared in the `global` section is used in the `var` element to help validate the `phone` property.

The `formset` element can contain two child elements, `constant` and `form`. The `constant` element has the same format as the one in the `global` section. It can be present zero or more times. The `form` element can be present one or more times within the `formset` element:

```
    <!ELEMENT formset (constant*, form+)>
```

The `formset` element supports two attributes that deal with I18N, `language` and `country`:

```
    <!ATTLIST formset language CDATA #IMPLIED
                      country  CDATA #IMPLIED>
```

If you don't have any I18N requirements for your validation routines and want to use the default locale, you can leave out these attributes. The section "Internationalizing the Validation" later in this chapter discusses this topic in more detail.

The `form` element defines a set of fields to be validated. The `name` corresponds to the identifier the application assigns to the form. In the case of the Struts framework, this is the `name` attribute from the `form-beans` section.

The `form` element defines a set of fields that are to be validated. It contains a single attribute `name`, which should match one of the `name` attributes from the `form-beans` section of the Struts configuration file.

The `form` element can contain one or more `field` elements:

```
<!ELEMENT form (field+)>
```

The `field` element corresponds to a specific property of a JavaBean that needs to be validated. In a Struts application, this JavaBean is an `ActionForm`. In Example 11-1, the sole `field` element for the `checkoutForm` corresponds to the `phone` property in an `ActionForm` called `checkoutForm` in the `form-beans` section of the Struts configuration file. The `field` element supports several attributes, which are listed in Table 11-2.

Table 11-2. The attributes of the field element

Attribute	Description
`property`	The property name of the JavaBean (or `ActionForm`) to be validated.
`depends`	The comma-delimited list of validation rules to apply against this field. For the field to succeed, all the validators must succeed.
`page`	The JavaBean corresponding to this form may include a `page` property. Only fields with a `page` attribute value that is equal to or less than the value of the `page` property on the form JavaBean are processed. This is useful when using a "wizard" approach to completing a large form, to ensure that a page is not skipped.
`indexedListProperty`	The method name that will return an array or a `Collection` used to retrieve the list and then loop through the list, performing the validations for this field.

Both the `ValidatorActionForm` and `DynaValidatorActionForm` match on the action mapping rather than the form name. That is, instead of matching on the form name in the `name` attribute of the `form` element, you can use the `path` attribute of the `action` element. This allows the same form to be used for different action mappings, where each action mapping may depend on only certain form fields to be validated and the others to be left alone.

The `field` element contains several child elements:

```
<!ELEMENT field (msg?, arg0?, arg1?, arg2?, arg3?, var*)>
```

The `msg` child element allows you to specify an alternate message for a `field` element. The validation rule can use this value instead of the default message declared

with the rule. The value for the msg element must be a key from the application resource bundle. For example:

```
<field property="phone" depends="required,mask">
  <msg name="mask" key="phone.invalidformat"/>
  <arg0 key="registrationForm.firstname.displayname"/>
  <var>
    <var-name>mask</var-name>
    <var-value>${phone}</var-value>
  </var>
</field>
```

The msg element supports three attributes:

```
<!ATTLIST msg name     CDATA #IMPLIED
              key      CDATA #IMPLIED
              resource CDATA #IMPLIED >
```

The name attribute specifies the rule with which the msg should be used. The value should be one of the rules specified in the *validation-rules.xml* file or in the global section.

The key attribute specifies a key from the resource bundle that should be added to the ActionError if validation fails. If you want to specify a literal message, rather than using the resource bundle, you can set the resource attribute to false. In this case, the key attribute is taken as a literal string.

The field element allows up to four additional elements to be included. These elements, named arg0, arg1, arg2, and arg3, are used to pass additional values to the message, either from the resource bundle or from the var or constant elements. The arg0 element defines the first replacement value, arg1 defines the second replacement value, and so on. Each arg element supports three attributes, name, key, and resource, which are the same as the attributes of the msg element described earlier.

Example 11-1 included elements for arg0 and arg1 like this:

```
<field property="phone" depends="required,mask,minLength">
  <arg0 key="registrationForm.firstname.displayname"/>
  <arg1 name="minlength" key="${var:minLength}" resource="false"/>
  <var>
   <var-name>mask</var-name>
   <var-value>${phone}</var-value>
  </var>
  <var>
   <var-name>minLength</var-name>
   <var-value>5</var-value>
  </var>
</field>
```

The last of the field child elements is the var element, as seen in Example 11-1 and in the previous fragment. The var element can set parameters that a field element may need to pass to one of its validation rules, such as the minimum and maximum values in a range validation. These parameters may also be referenced by one of the arg elements using a shell syntax: ${var:var-name}.

In Example 11-1, the substituted value for the phone constant is passed into the mask validation rule so that it can be used to check whether the property value conforms to the proper phone mask. The field element can have zero or more var elements.

Once you have the two XML resource files configured for your application, you need to place them in the *WEB-INF* directory. They will be referenced within the Struts configuration file, as described in the next section.

Plugging in the Validator

Each Struts application needs to know that the Validator framework is being employed. As discussed in Chapter 9, you can use the PlugIn mechanism to hook the Validator framework into a Struts application.

Earlier versions of the Validator used an extra servlet to inform the Struts application that the Validator components were present. The ValidatorServlet has been deprecated and should not be used.

The following fragment illustrates how to set up the Validator as a plug-in:

```
<plug-in className="org.apache.struts.validator.ValidatorPlugIn">
  <set-property
    property="pathnames"
    value="/WEB-INF/validator-rules.xml,/WEB-INF/validator.xml"/>
</plug-in>
```

There was some confusion in one of the earlier beta releases for the Validator that used multiple set-property elements. That is no longer supported—you should use a single set-property element that specifies multiple Validator resource files, separated by commas. Also notice that the property value is the plural pathnames.

The Struts framework will call the init( ) method in the ValidatorPlugIn class when the application starts up. During this method, the Validator resources from the XML files are loaded into memory so that they will be available to the application. Before calling the init( ) method, however, the pathnames property value is passed to the ValidatorPlugIn instance. This is how the ValidatorPlugIn finds out which Validator resources to load. For more information on how the PlugIn mechanism works, see "Using the PlugIn Mechanism" in Chapter 9.

Using an ActionForm with the Validator

You can't use the standard Struts ActionForm class with the Validator. Instead, you need to use a subclass of the ActionForm class that is specifically designed to work

with the Validator framework. There are two root subclasses to select from, depending on whether you are planning to use dynamic ActionForms. Figure 11-1 shows the ActionForm and its descendants, to help you visualize the hierarchy.

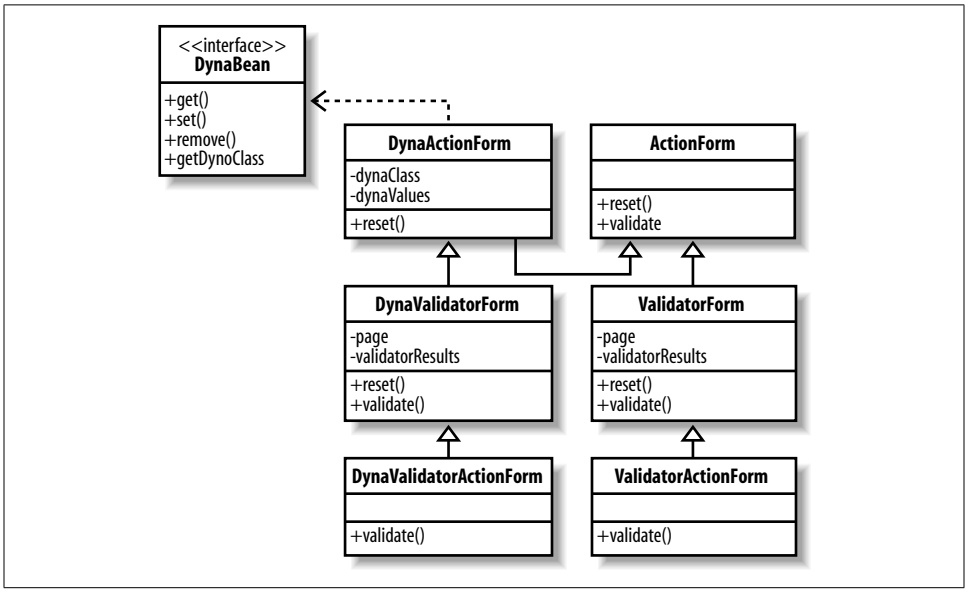

Figure 11-1. The ActionForm class hierarchy

If you are using dynamic ActionForms, you should use the DynaValidatorForm branch of the hierarchy. If you are using standard ActionForms, you can use the ValidatorForm or one of its descendants instead.

Whether you use dynamic or regular ActionForms, the manner in which you configure the Validator is the same. Just be sure that whichever ActionForm subclass you choose, you configure the form-bean section of the Struts configuration file using the fully qualified class name. See "The form-beans element" in Chapter 4 for more details.

Dynamic or standard is only the first decision that you have to make when choosing the proper ActionForm subclass. Notice that in both the dynamic and standard branch of the ActionForm hierarchy in Figure 11-1, there are two versions of ValidatorForm. The parent class is called ValidatorForm, or DynaValidatorForm for the dynamic branch.

Each of these has a subclass that contains the name Action in its title. The subclass of the ValidatorForm is called ValidatorActionForm, and the subclass of the DynaValidatorForm is called DynaValidatorActionForm. The purpose of the two different versions is to allow you to associate the validation with the form-bean definition or the action definition. The ValidatorActionForm and DynaValidatorActionForm classes

pass the path attribute from the action element into the Validator, and the Validator uses the action's name to look up the validation rules. If you use the ValidatorForm or DynaValidatorForm, the name of the ActionForm is used to look up the set of validation rules to use. The only reason for using one or the other is to have more fine-grained control over which validation rules are executed.

For example, suppose that an ActionForm contains three different validation rules, but only two of them should get executed for a particular action. You could configure the rules to perform only the subset of validation rules when that action gets invoked. Otherwise, all of the rules would be invoked. In general, using ValidatorForm or DynaValidatorForm should be sufficient for your needs.

Let's look at a more complete example of using the Validator framework. As in previous chapters, we'll employ the Storefront application to help us understand the Validator better. In particular, we'll look at the HTML form used to capture the shipping information during checkout of the Storefront application. This is shown in Figure 11-2.

For this example, we are going to use a dynamic form. Therefore, we'll use the DynaValidatorForm class to capture the shipping address details. Because the checkout process will span multiple pages and we want to capture this information across pages, we will need to configure the form bean to have session scope. We will also capture all of the checkout properties in a single ActionForm class. Instead of having a ShippingForm and a CreditCardForm, we will have a single form called CheckoutForm that captures all of the information.

In our Struts configuration file, we set up the checkoutForm as shown here:

```
<form-bean
  name="checkoutForm"
  type="org.apache.struts.validator.DynaValidatorForm">
  <form-property name="firstName" type="java.lang.String"/>
  <form-property name="lastName" type="java.lang.String"/>
  <form-property name="address" type="java.lang.String"/>
  <form-property name="city" type="java.lang.String"/>
  <form-property name="state" type="java.lang.String"/>
  <form-property name="postalCode" type="java.lang.String"/>
  <form-property name="country" type="java.lang.String"/>
  <form-property name="phone" type="java.lang.String"/>
</form-bean>
```

The type attribute specifies the exact ActionForm subclass.

In early beta releases of Struts 1.1, the form-bean section required that you set the dynamic attribute to true when using dynamic ActionForms. This is no longer necessary, as the framework will determine whether the class specified in the type attribute is a descendant of the DynaActionForm class.

Figure 11-2. Capturing the shipping address information

The next step is to edit the application-specific validation logic, which is done in the *validation.xml* file. You must declare a validation rule for each property in the form that you need to validate. In some cases, you might need to specify multiple rules. In Figure 11-2, for example, the phone field is required, and it must fit a specific format. These are two separate rules that both must evaluate to `true`, or the validation for the form will fail. The entire *validation.xml* file is not shown because it's too large and most of it is redundant. The section shown in Example 11-2 will help you understand how things are connected.

Example 11-2. A sample validation.xml file for the checkout form

```
<formset>
  <constant>
    <constant-name>phone</constant-name>
    <constant-value>^\(?(\d{3})\)?[-| ]?(\d{3})[-| ]?(\d{4})$</constant-value>
  </constant>
  <constant>
    <constant-name>zip</constant-name>
    <constant-value>^\d{5}(-\d{4})?$</constant-value>
  </constant>
  <form name="checkoutForm">
    <field
      property="firstName"
      depends="required,mask">
      <arg0 key="label.firstName"/>
      <var>
        <var-name>mask</var-name>
        <var-value>^[a-zA-Z]*$</var-value>
      </var>
    </field>
    <field
      property="postalCode"
      depends="required,mask">
      <arg0 key="registrationForm.zip"/>
      <var>
        <var-name>mask</var-name>
        <var-value>${zip}</var-value>
      </var>
    </field>
    <field
      property="phone"
      depends="required,mask">
      <arg0 key="registrationForm.phone"/>
      <var>
        <var-name>mask</var-name>
        <var-value>${phone}</var-value>
      </var>
    </field>
  </form>
 </formset>
</form-validation>
```

Now that we have everything configured for the Storefront, it's time to run the example. The nice thing about using a declarative approach versus a programmatic one is that once you have everything configured, you're ready to go. The absence of programming makes the declarative approach much simpler. This is especially true for the Validator framework. There's nothing to code, as long as the default validation rules satisfy your requirements.

When we submit the shipping address page with no information in the fields, the validation rules kick in. The result is shown in Figure 11-3.

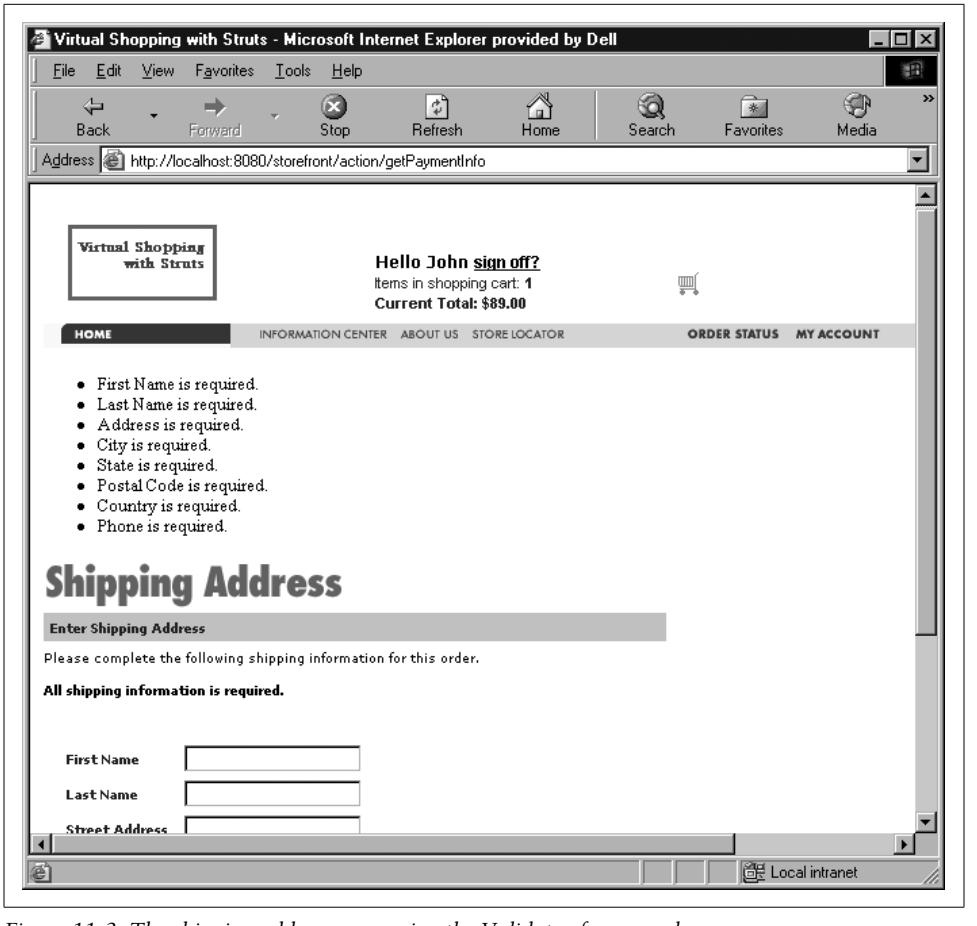

Figure 11-3. The shipping address page using the Validator framework

Creating Your Own Validation Rules

The Validator framework is preconfigured with many of the most common rules that you're likely to need for your Struts applications. If your application has validation requirements that are not met by the default rules, you have complete freedom to create your own. There are several steps that you must follow, however, to create your own customized rules:

1. Create a Java class that contains the validation methods.
2. Edit the *validation-rules.xml* file or create your own version. If you do create a new validation resource file, be sure to add it to the list of resource files in the Validator plug-in.
3. Use the new validation rules in the *validation.xml* file for your application.

Each validation method you create must have the following signature:

```
public static boolean validateXXX( java.lang.Object,
                                   org.apache.commons.validator.ValidatorAction,
                                   org.apache.commons.validator.Field,
                                   org.apache.struts.action.ActionErrors,
                                   javax.servlet.http.HttpServletRequest,
                                   javax.servlet.ServletContext );
```

where `validateXXX` can be whatever you want it to be, as long as it's not a duplicate rule name. Table 11-3 lists the arguments for the `validateXXX( )` method.

Table 11-3. The validateXXX() method arguments

Parameter	Description
`Object`	The JavaBean on which validation is being performed
`ValidatorAction`	The current `ValidatorAction` being performed
`Field`	The field object being validated
`ActionErrors`	The errors objects to add an `ActionError` to if the validation fails
`HttpServletRequest`	The current request object

In most cases, the method should be static. However, you can define instance-level methods as well. Regardless of whether your methods are static, you must ensure that they are thread-safe. Example 11-3 illustrates a new validation rule that validates whether a `String` value is a valid `boolean`.

Example 11-3. A validation rule that validates a boolean value

```
import java.io.Serializable;
import java.util.Locale;
import javax.servlet.ServletContext;
import javax.servlet.http.HttpServletRequest;
import org.apache.commons.validator.Field;
import org.apache.commons.validator.GenericTypeValidator;
import org.apache.commons.validator.GenericValidator;
import org.apache.commons.validator.ValidatorAction;
import org.apache.commons.validator.ValidatorUtil;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.util.StrutsValidatorUtil;

public class NewValidator implements Serializable {
  /**
   * A validate routine that ensures the value is either true or false.
   */
  public static boolean validateBoolean( Object bean, ValidatorAction va,
    Field field, ActionErrors errors, HttpServletRequest request ) {

    String value = null;
    // The boolean value is stored as a String
```

Example 11-3. A validation rule that validates a boolean value (continued)

```
    if (field.getProperty() != null && field.getProperty().length() > 0){
      value = ValidatorUtil.getValueAsString(bean, field.getProperty() );
    }


    Boolean result = Boolean.valueOf(value);
    if ( result == null ){
      errors.add( field.getKey(),
                  StrutsValidatorUtil.getActionError(request, va, field));
    }

    // Return true if the value was successfully converted, false otherwise
    return (errors.empty());
  }
}
```

The next step is to add this new rule to the *validation-rules.xml* file, or to a new file to keep your customized rules separate. The `validator` element for the `validateBoolean` rule should look something like:

```
<validator name="boolean"
  classname="NewValidator"
  method="validateBoolean"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionErrors,
    javax.servlet.http.HttpServletRequest"
  msg="errors.boolean">
```

The final step is to use the new validation rule in the *validation.xml* file. This involves creating a `field` element that matches a `boolean` property on an `ActionForm`:

```
<field property="sendEmailConfirmation" depends="boolean">
   <arg0 key="label.emailconfirmation"/>
</field>
```

The Validator and JSP Custom Tags

Several JSP custom tags included within the Struts tag libraries can be used with the Validator framework. One of the tags is used to generate dynamic JavaScript based on the validation rules. The other tags are part of the core Struts framework and are used both with and without the Validator.

The tags listed in Table 11-4 are generic and can be used with or without the Validator framework, but they all come in handy when using it.

Table 11-4. JSP custom tags that can be used with the Validator

Tag name	Description
`Errors`	Displays any validation errors found during processing
`ErrorsExist`	Determines if there were any validation errors
`Messages`	Displays any messages found during processing
`MessagesExist`	Determines if there were any messages during processing

The tags in Table 11-4 allow JSP pages to detect and obtain access to messages or errors that were detected in the Struts application. These tags were discussed in more detail in Chapter 8.

Using JavaScript with the Validator

The Validator framework is also capable of generating JavaScript for your Struts application using the same framework as for server-side validation. This is accomplished by using a set of JSP custom tags designed specifically for this purpose.

Configuring the validation-rules.xml file for JavaScript

The Validator custom tag called `JavascriptValidator` is used to generate client-side validation based on a `javascript` attribute being present within the `validator` element. Before the JSP custom tag can be used, there must be a `javascript` element for the validation rule. The following code fragment illustrates the `required` validation rule that includes a `javascript` element:

```
<validator
  name="required"
  classname="org.apache.struts.util.StrutsValidator"
  method="validateRequired"
  methodParams="java.lang.Object,
    org.apache.commons.validator.ValidatorAction,
    org.apache.commons.validator.Field,
    org.apache.struts.action.ActionErrors,
    javax.servlet.http.HttpServletRequest"
  msg="errors.required">
  <javascript><![CDATA[
   function validateRequired(form) {
      var bValid = true;
      var focusField = null;
     var i = 0;
     var fields = new Array( );
     oRequired = new required( );

     for (x in oRequired) {
      if ((form[oRequired[x][0]].type == 'text' ||
           form[oRequired[x][0]].type ==    'textarea' ||
           form[oRequired[x][0]].type == 'select-one' ||
           form[oRequired[x][0]].type == 'radio' ||
```

```
            form[oRequired[x][0]].type == 'password') &&
            form[oRequired[x][0]].value == '') {
             if (i == 0)
              focusField = form[oRequired[x][0]];
              fields[i++] = oRequired[x][1];
            bValid = false;
           }
      }

       if (fields.length > 0) {
         focusField.focus();
       alert(fields.join('\n'));
      }
      return bValid;
     }]]>
   </javascript>
</validator>
```

When the `JavascriptValidator` tag is included in the JSP page, the text from the `javascript` element is written to the JSP page to provide client-side validation. When the user submits the form, the client-side validation is executed, and any validation rules that fail present messages to the user.

You will need to include the `javascript` tag with the name of the `ActionForm` that it's going to validate against:

```
<html:javascript formName="checkoutForm"/>
```

The `formName` attribute is used to look up the set of validation rules to include as JavaScript in the page. You will have to add an `onsubmit` event handler for the form manually:

```
<html:form action="getPaymentInfo" onsubmit="return validateCheckoutForm(this);">
```

When the form is submitted, the `validateCheckoutForm( )` JavaScript function will be invoked. The validation rules will be executed, and if one or more rules fail, the form will not be submitted. The `javascript` tag generates a function with the name `validateXXX( )`, where `XXX` is the name of the `ActionForm`. Thus, if your `ActionForm` is called `checkoutForm`, the `javascript` tag will create a JavaScript function called `validateCheckoutForm( )` that executes the validation logic. This is why the `onsubmit( )` event handler called the `validateCheckoutForm( )` function.

By default, the `JavascriptValidator` tag generates both static and dynamic JavaScript functions. If you would like to include a separate file that contains static JavaScript functions to take advantage of browser caching or to better organize your application, you can use the `dynamicJavascript` and `staticJavascript` attributes. By default, both of these are set to `true`. You can set the `staticJavascript` attribute to `false` in your form and include a separate JavaScript page with the `dynamicJavascript` attribute set to `false` and the `staticJavascript` attribute set to `true`. See the documentation for the `JavascriptValidator` tag for more information.

Internationalizing the Validation

The Validator framework uses the application resource bundles to generate error messages for both client-side and server-side validation. Thus, from an I18N perspective, much of the work for displaying language-specific messages to the user is included within the framework.

It was mentioned earlier that the `formset` element in the *validation.xml* file supports attributes related to internationalization. Those attributes are `language`, `country`, and `variant`. As you know, these attributes correspond to the `java.util.Locale` class. If you don't specify these attributes, the default `Locale` is used.

If your application has I18N validation requirements, you will need to create separate `formset` elements, one for each `Locale` that you need to support for each form that you need to validate. For example, if your application has to support validation for a form called `registrationForm` for both the default locale and the French locale, the *validation.xml* file should contain two `formset` elements—one for the default locale and the other for the French locale. This is shown in the following fragment:

```
<formset>
  <form name="registrationForm">
    <field
      property="firstName"
      depends="required,mask,minLength">
      <arg0 key="registrationForm.firstname.displayname"/>
      <var>
        <var-name>mask</var-name>
        <var-value>^\w+$</var-value>
      </var>
      <var>
        <var-name>minLength</var-name>
        <var-value>5</var-value>
      </var>
    </field>
  </form>
</formset>
<formset language="fr">
  <form name="registrationForm">
  <field
    property="firstName"
    depends="required,mask,minLength">
    <arg0 key="registrationForm.firstname.displayname"/>
    <var>
      <var-name>mask</var-name>
      <var-value>^\w+$</var-value>
    </var>
    </field>
  </form>
</formset>
```

Using the Validator Outside of Struts

Although the Validator was originally designed to work with the Struts framework, it can be used to perform generic validation on any JavaBean. There are several steps that must be performed before the framework can be used outside of Struts.

Although the Validator is not dependent on the Struts framework, a considerable amount of work has been done inside of Struts to make it easier to use the Validator. This behavior will need to be replicated for your application if you plan to use the Validator without Struts.

The package dependencies are exactly the same for Struts and non-Struts applications. The ORO, Commons Logging, Commons BeanUtils, Commons Collections, and Digester packages are all required. You will also need an XML parser that conforms to the SAX 2.0 specification. You will not need to include the Struts framework, however.

Functions for loading and initializing the XML Validator resources are the first behaviors to replicate. These are the two XML files that are used to configure the rules for the Validator. When the Validator framework is used in conjunction with Struts, the `org.apache.struts.Validator.ValidatorPlugIn` class performs this duty. However, because the `ValidatorPlugIn` is dependent on Struts, you will need to create an alternate approach for initializing the appropriate Validator resources. To do this, you can create a simple Java class that performs the same behavior as the `ValidatorPlugIn` but doesn't have a dependency on the Struts framework. A simple example is provided in Example 11-4.

Example 11-4. Using the Validator outside of Struts

```
import java.util.*;
import java.io.*;
import org.apache.commons.validator.ValidatorResources;
import org.apache.commons.validator.ValidatorResourcesInitializer;

public class ValidatorLoader{

  private final static String RESOURCE_DELIM = ",";
  protected ValidatorResources resources = null;
  private String pathnames = null;

  public ValidatorLoader( ) throws IOException {
    loadPathnames( );
    initResources( );
  }

  public ValidatorResources getResources( ){
    return resources;
  }
```

Example 11-4. Using the Validator outside of Struts (continued)

```
  public String getPathnames( ) {
    return pathnames;
  }

  public void setPathnames(String pathnames) {
    this.pathnames = pathnames;
  }

  protected void loadPathnames( ){
    // Set a default just in case
    String paths = "validation-rules.xml,validation.xml";
    InputStream stream = null;

    try{
      // Load some properties file
      stream = this.getClass( ).getResourceAsStream( "validator.properties" );
      if ( stream != null ){
        Properties props = new Properties( );
        props.load( stream );
        // Get the pathnames string from the properties file
        paths = props.getProperty( "validator-pathnames" );
      }
    }catch( IOException ex ){
      ex.printStackTrace( );
    }
    setPathnames( paths );
  }

  protected void initResources( ) throws IOException {
    resources = new ValidatorResources( );

    if (getPathnames() != null && getPathnames( ).length( ) > 0) {
      StringTokenizer st = new StringTokenizer(getPathnames( ), RESOURCE_DELIM);
      while (st.hasMoreTokens( )) {
        String validatorRules = st.nextToken( );
        validatorRules = validatorRules.trim( );

        InputStream input = null;
        BufferedInputStream bis = null;
        input = getClass( ).getResourceAsStream(validatorRules);

        if (input != null){
          bis = new BufferedInputStream(input);

          try {
            // pass in false so resources aren't processed
            // until last file is loaded
            ValidatorResourcesInitializer.initialize(resources, bis, false);
          }catch (Exception ex){
            ex.printStackTrace( );
          }
        }
```

Example 11-4. Using the Validator outside of Struts (continued)

```
      }
      // process resources
      resources.process();
    }
  }
}
```

The work being done in the `ValidatorLoader` from Example 11-4 is very similar to what the `ValidatorPlugIn` does—it loads and initializes an instance of the `ValidatorResources` class. The object is an in-memory representation of the validation rules for an application. This example uses the `getResourceAsStream()` method to find and load a properties file that contains the list of Validator resource files.

Once you create and initialize an instance of the `ValidatorResources` class, you will need to cache it somewhere. In a Struts application, it's cached in the `ServletContext`. Your application can hang onto this object, or you can wrap the resource inside of a Singleton.

Modifying the validation-rules.xml File

In the earlier section "Creating Your Own Validation Rules," you saw how to extend the Validator framework with your own customized rules. You'll have to do this here as well, but the method signatures will be different. The method signature in Example 11-3 included parameters that are part of the Servlet and Struts APIs. You will need to use different arguments to keep from being coupled to the Servlet API or the Struts framework.

First, the `methodParams` attribute needs to be modified to support the alternate arguments to the validation method. The following is a fragment for a rule called `currency`:

```
<global>
 <validator name="currency"
  classname="com.oreilly.struts.storefront.Validator"
  methodParams="java.lang.Object,org.apache.commons.validator.Field,java.util.List"
  method="isCurrency"
  msg="Value is not a valid Currency Amount."/>
</global>
```

Once the validation rule itself is set up, you need to use it in the application-specific validation file:

```
<form-validation>
 <global>
 </global>
 <formset>
  <form name="checkoutForm">
   <field property="paymentAmount" depends="required,currency">
     <arg0 key="registrationForm.paymentamount.invalid"/>
```

```
    </field>
  </formset>
</form-validation>
```

Somewhere in the application, you must obtain access to the `ValidatorResources` object instance that was initialized in Example 11-4 and use it to validate the JavaBean:

```
ValidatorResources resources = // Get instance of the ValidatorResources
Validator validator = new Validator(resources, "checkoutForm");

validator.addResource(Validator.BEAN_KEY, bean);
validator.addResource("java.util.List", lErrors);

try {
  // Execute the validation rules
  validator.validate( );
} catch ( ValidatorException ex ) {
 // Log the validation exception
 log.warn( "A validation exception occured", ex );
}
```

Although the Validator framework is designed for use with or without Struts, some work is required before it's ready to use outside of the Struts framework. However, with a little up-front sweat, you can save yourself plenty of work downstream in the development cycle.