12 Validating user input

Co-authored by David Winterfeldt and Ted Husted

This chapter covers

- Understanding the need to validate data
- Configuring and using the Commons Validator
- Using multipage and localized validations
- Writing your own validators

Us: *Programming today is a race between software engineers striving to build bigger and better idiot-proof programs, and the Universe trying to produce bigger and better idiots. So far, the Universe is winning.*

—Rich Cook

Them: *I never know how much of what I say is true.*

—Bette Midler

12.1 I know it when I see it

Most web applications need to collect data from their users. The input may come via freeform text fields or GUI elements such as menu lists, radio buttons, and checkboxes. In practice, what the user enters does not always make sense. Some menu options may be mutually exclusive. A phone number may be lacking a digit. Letters might be entered into a numeric field. Numbers may be entered where letters are expected. This may be because the data-entry form isn't clear or because the users are not paying attention. But in any event it happens—and with great regularity.

Nothing is more frustrating for a user than getting garbage out of an application, even if it was the user who put the garbage in. A prudent application scrutinizes all input, guards against all foreseeable errors, and protects users from themselves. After all, if they mess up, we still have to fix it.

12.1.1 Input we can't refuse

In a conventional application, data-entry controls can simply refuse to accept bad values, but they have luxury of being modal.

DEFINITION A user interface element is *modal* when it claims all the user input for an application. Other elements of the application cannot be accessed until the element is dismissed. To proceed, the user must either complete the modal dialog box or close the application. Most user interface elements are nonmodal.

Web applications, being inherently nonmodal, have fewer options. By default, the HTML elements displayed by the browser will accept anything typed into them. A given element has no clue what was entered elsewhere on the form. We can play tricks with JavaScript, but there is no guarantee that the user has enabled JavaScript.

Of course, we can validate the data when it reaches the business tier. (For more about application tiers and business objects, see chapter 2.) Many business logic objects do have some validation built in, but most business objects do not vet data before accepting it. Business-tier methods tend to be trusting creatures. They expect friendly objects will be offering reasonable data and just do as they are told. Even when business objects are more pessimistic, usually all they can do is throw an exception. It is not the responsibility of business objects to enter into a patient dialogue with the user to correct the erroneous input.

Of course, it *is* the responsibility of a business object to validate data *in context*—to see, for example, if a username and password correspond. But there are many objective validation rules that can be applied before data is ever commuted to the business tier. In a distributed application, the business object may reside on a remote machine. Creating roundtrips to resolve simple data-entry errors may be costly.

12.1.2 Web-tier validations

In real life, it often falls to the web application framework to provide objective validation routines and narrow the gap between model and view. In a nonmodal, distributed environment, we need validation routines that can do the following:

- Require that certain fields have values
- Confirm that a given value is in an expected pattern or range
- Check the entire form at once and return a list of messages
- Compare values between fields
- Return the original input for correction
- Display localized messages when required
- Perform server-side validations if JavaScript is disabled

Two other important hallmarks of a validation system are loose coupling and optional client-side validations.

Loose coupling

As a practical matter, input needs to be validated by the controller, but business validations are tied to the business tier. This implies that the validation rules should be stored separately from markup or Java code so that they can be reviewed and modified without changing any other source code. Keeping the validation rules loosely coupled makes it much easier to keep validations synchronized with business requirements.

DEFINITION The degree of *coupling* refers to the strength of a connection between two components. Coupling is a complement to cohesion. Cohesion describes how strongly the internal contents of a component are related to each other. The goal is to create components with internal integrity (strong cohesion) and small, direct, visible, and flexible relations to other components (loose coupling). [McConnell]

Some validation rules may also need to be localized. When support for a new locale is added, we should be able to update the validation rules as easily we update the resource bundle.

While the validation rules may be provided as a convenience to the presentation layer, it is important to recognize they actually "belong" to the business tier. Validation rules should not be commingled with presentation source code.

Client-side validations

Client-side validations are inherently insecure. It is easy to spoof submitting a web page and bypass any scripting on the original page. While we cannot rely on client-side JavaScript validations, they are still useful. Immediate user feedback avoids another trip to the server, saving time and bandwidth for everyone. So, another ideal feature would be to generate JavaScript and server-side validations from the same set of rules. When JavaScript is enabled, the input can be validated client-side before it is submitted. If not, the input is still validated server-side to ensure nothing is amiss.

12.1.3 Validator consequences

Using the Jakarta Commons Validator [ASF, Validator] brings several consequences:

- The Validator is a framework component that meets these requirements—and more.
- The Validator is configured from an XML file that generates validation rules for the fields in your form.
- Rules are defined by a Validator that is also configured through XML.
- Validators for basic types, like dates and integers, are provided. If needed, you can create your own.
- Regular expressions can be used for pattern-based validations such as postal codes and phone numbers.

- Multipage and localized validations are supported, so you can write wizards in any language.

DEFINITION A *regular expression* is a formula for matching strings that follow some pattern. Regular expressions are used by many Unix command-line and programming utilities. For more about regular expressions, see the "Using Regular Expressions" web page by Stephen Ramsay. [Ramsay]

Using the Jakarta Commons Validator in your application yields several benefits:

- Optimal use of resources: JavaScript validations are provided when enabled, and server-side validations are guaranteed.
- A single point of maintenance: Both client-side and server-side validations are generated from the same configuration.
- Extendibility: Custom validations can be defined as regular expressions or in Java code.
- Maintainability: It is loosely coupled to the application and can be maintained without changing markup or code.
- Localization: Localized validations can be defined only when and where they are needed.
- Integration with Struts: By default, validations share the Struts message bundle. Localized text can be centralized and reused.
- Easy deployment of server-side validation: To make use of the server-side validations, your Struts ActionForm can simply extend the ValidatorForm or ValidatorActionForm class. The rest is automatic.
- Easy deployment of client-side validation: To make use of the client-side validations, you just add a single JSP tag to generate the validation script and use that script to submit the form.
- Easy configuration: The Validator uses an XML file for configuration, just like the web application deployment descriptor and the Struts configuration.

But, of course, there are also drawbacks:

- Nonmodal client-side validations: The generated JavaScript is nonmodal; it does not engage until the form is submitted.

- Dependencies: The validations are detached from the fields and from the ActionForm properties. The page markup, the ActionForm, and the Validator and Struts configuration files must all be synchronized.
- Lack of data conversions and transformations: The package does not offer data conversions or transformations. When needed, conversions and transformations must be programmed separately.

Keep in mind that using the Jakarta Commons Validator in your application is not a panacea. Some validations may only be performed server-side. If these fail, the error messages are displayed differently than the JavaScript messages. Interface discontinuities confuse users.

DEFINITION *Data conversion* is moving data from one type to another, as from a `String` to an `Integer`. *Data transformation* is changing the internal format of data, such as adding punctuation to a `String` before it is displayed or removing unwanted punctuation from a `String` before it is stored. Localization can require transforming data into a display format.

In this chapter, we show you how to make the best use of the Commons Validator framework in your application. We cover the overall design of the Validator, and present a simple example. We then look at each component of the Validator in depth along with often-needed techniques, such as overriding default messages, canceling validations, using multipage workflows, validating collections, and more.

It is important to emphasize that objective, data-entry validations are not an omnibus solution. There are many types of errors that cannot be found without accessing the model. We can look to see if a username and password meet the business requirements for length and composition. But to see if the username and password combination is valid, we need to go up to the business tier and talk to a data service. However, by checking to see whether data could possibly be valid before we even ask, we can eliminate expensive data-access transactions, which benefits everyone.

NOTE You might be wondering, "So would I be using the Struts framework to build my application or the Struts Validator framework?" Both, actually. Most applications are built using several sets of framework components, including some that development teams create in-house. Struts builds on

Sun's Java J2SE framework. Likewise, the Struts Validator builds on the Struts framework. So, just as your application may use several classes in several packages, it may also use several frameworks. For more about working with framework architectures, see chapter 2.

Chapter 4 covers setting up the Validator with Struts 1.1. This chapter is a developer's guide to putting the Validator to work in your application.

12.2 Overview of the Struts Validator

Let's look at how the Struts Validator interacts with other components to provide both server-side and client-side validations from the same set of validation rules. You may be surprised at how easy it can be to validate your data once the Validator puts all the pieces together. Table 12.1 lists the various pieces that make up the Struts Validator.

Table 12.1 Major Struts Validator components

Component	Description
Validators	Handle native and other common types. The basic validators include `required`, `mask` (matches regular expression), `minLength`, `maxLength`, `range`, native types, `date`, `email`, and `creditCard`. Custom (or plug-in) validators may also be defined.
Resource bundle	Provides (localized) labels and messages. Shares Struts messages by default.
XML configuration file	Defines form set and validations for fields as needed. The validators can be defined in a separate file.
JSP tag	Generates JavaScript validations for a given form name or action path.
ValidatorForm	Automatically validates properties based on the form bean's name (passed to the `validate` method through the ActionMapping parameter at runtime). Must be extended to provide the properties expected on the form.
ValidatorActionForm	Automatically validates properties based on the action path (passed to the `validate` method through the ActionMapping parameter at runtime). Must be extended to provide the properties expected on the form.

Originally, the Commons Validator was created as an extension to the Struts framework. But since it could also be used outside the framework, the developers contributed it to another Jakarta subproject, the Commons.

The Struts distribution includes a Validator package with several classes that integrate the Commons Validator with Struts. This package, along with the Commons Validator package it extends, constitutes the Struts Validator. In the balance of this chapter, we refer to the Struts Validator as a superset of the Commons Validator. The Validator package is actually a collection of several Validator objects, written in Java. Each Validator object enforces a rule regarding a property on another object. In the Struts Validator, these objects are ActionForms. The validators have standard entry methods, like a Struts Action, that are used to call the Validator when needed. The Validator configuration file lets you associate one or more validators with each property in a form.

In practice, most applications need to perform a number of common validations. Some fields may require data to be entered. A postal code abbreviation may always be of a known length. Other common field types include numbers, dates, and credit card numbers.

The Validator comes equipped with several basic validators to handle these common needs, among others. If your Validator needs can't be met by one of the basic validators or a regular expression, you can roll your own Validator and plug it into the package. The basic validators are really just bundled plug-ins themselves. Your custom validators can do anything the basic validators do, and more.

The validators your application needs to use, basic or custom, can be specified in an XML configuration file, usually named validation.xml. To make maintenance easier, you can specify the rules that associate a Validator with your ActionForm properties in a separate file, usually named validator-rules.xml.

1.0 vs 1.1 The version of the Validator for Struts 1.0 uses a single validation.xml file that contains both the Validator definitions and the form validations. Struts 1.1 lets you split these components into separate files. In this chapter, we will refer to the separate files used by Struts 1.1. If you are using Struts 1.0, all configuration elements are kept in the single validation.xml file.

The validation-xml file has a `<form>` element that usually corresponds to the `<form-bean>` element in your Struts application. The `<form>` element in turn has `<field>` subelements. Each `<field>` can specify that it must pass one or more validators to succeed. If a validator fails, it can pass back a key to a message template in the application resources, along with any replacement parameters. Struts uses the key and parameters to generate a localized error message. If client-side

validations are being used, the same message can be displayed in a JavaScript window, as shown in figure 12.1.

The validation file is where you plug in whatever basic or custom validations are needed by the validator rules. Here you specify which Validator classes to use, along with the optional client-side JavaScript validations. When used, the JavaScript validations must pass before the form is submitted back to the application.

Besides the JavaScript element in the validator configuration file, the other piece of client-side validations is the `<validator:javascript>` tag (Struts 1.0) or `<html:javascript>` tag (Struts 1.1). If you place this anywhere on your JSP, it will combine the JavaScript for all the validators into a single script that can be called from a standard entry method. You can then call the entry method using the `onsubmit` attribute to the `<html:form>` tag. If the JavaScript validations pass, the form is submitted. If not, a window pops up with the localized error messages.

To enable the server-side validations, you can simply extend your ActionForms from one of the base classes in the Struts Validator package. The ValidatorForm (`org.apache.struts.validator.ValidatorForm`) class corresponds to the standard ActionForm. The DynaValidatorForm class (`org.apache.struts.validator.DynaValidatorForm`) corresponds to the DynaActionForm.

By default, the Validator `<form>` elements are matched to the ActionForms using the attribute or form bean name. Alternatively, you can use ValidatorActionForm

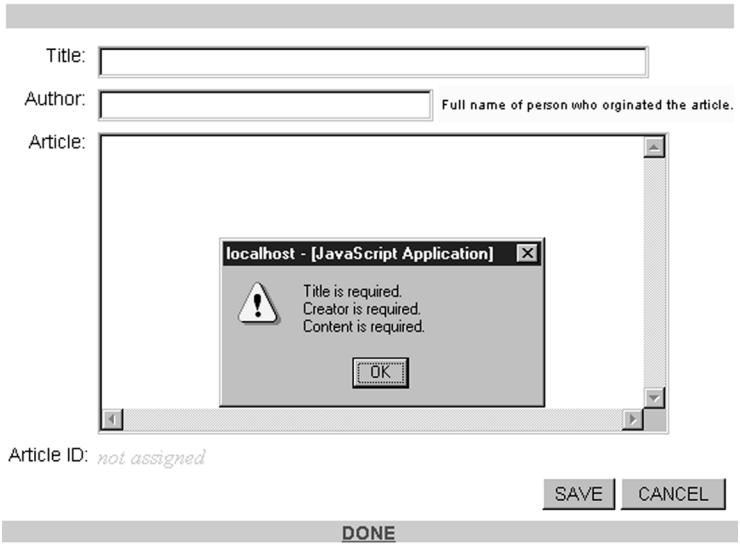

Figure 12.1 A JavaScript message window generated by the Struts Validator

(`org.apache.struts.validator.ValidatorActionForm`) or DynaValidatorActionForm (`org.apache.struts.validator.DynaValidatorActionForm`) to match up the `<form>` elements using the ActionMapping path.

In the next section, we show you how to put it all together by using the logon application as the backdrop for both client-side and server-side validation.

1.0 vs 1.1 In the 1.1 release, the Struts Validator was bundled into the Struts JAR and made an optional component of the formal distribution. The imports and some minor implementation details changed, but the package is essentially unchanged. Where there are implementation differences, we show separate listings for each release.

12.2.1 Logon example

Adapting the logon application from chapter 3 for the Struts Validator will help illustrate how these components fit together. Then, in section 12.3, we explore each component in depth.

After you set up the package (see chapter 4), your next step is to be sure the validators you need are on hand. These can be kept in a separate file, named validator-rules.xml by default. Our example only uses the `required` validator, though most applications will use several others as well.

In the following sections, we will look at what we need in order to validate a username and password using the Struts Validator.

validator-rules.xml

The Struts Validator distribution includes validators for native types and other common needs, like e-mail and credit card validations. In practice, it's likely that the basic validators will cover all your needs. If not, you can write your own and define them in the validator-rules.xml file along with the basic validators. (As noted, in Struts 1.0, both the validators and the form validations are defined in the single validation.xml file.)

Listings 12.1 and 12.2 show the Java and XML source code for the `required` validator under Struts 1.1. For more about writing your own pluggable validators, see section 12.9.

Listing 12.1 The XML source for the required validator (Struts 1.1)

```
<validator name="required"
<!-- ❶ -->
    classname="org.apache.struts.util.StrutsValidator"
    method="validateRequired"
      <!-- ❷ -->
    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">
    <!-- ❸ -->
    <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>
```

❶ This section contains the reference to the server-side validator (see listing 12.2).

❷ `methodParams` are used in Struts 1.1 only.

❸ The client-side JavaScript is included with the XML element.

As shown, the Validator elements are defined in two parts:

- A Java class and method for the server-side validation
- JavaScript for the client-side validation

The Java method the `required` validator invokes is shown in listing 12.2.

Listing 12.2 The Java source for the validateRequired method (Struts 1.1)

```
public static boolean validateRequired(Object bean,
        ValidatorAction va, Field field,
        ActionErrors errors,
        HttpServletRequest request) {
   String value = null;
    if (isString(bean)) {
        value = (String) bean;
    } else {
      value = ValidatorUtil.getValueAsString(bean, field.getProperty());
    }
    if (GenericValidator.isBlankOrNull(value)) {
        errors.add(field.getKey(),
                StrutsValidatorUtil.getActionError(request, va, field));
        return false;
    } else {
        return true;
    }
}
```

application.properties

If a validator fails, it passes back a message key and replacement parameters, which can be used with a standard resource bundle. By default, the Struts Validator shares the resource bundle used by the rest of your application. This is usually named ApplicationResources.properties, or just application.properties. When another message is not specified, the Struts Validator will automatically look for a message in the default resource bundle. Concatenating errors with a dot and the validator name usually creates the key for the message. Here is what the entry for our `required` validator looks like:

```
errors.required={0} is required.
```

When we configure a field to use the `required` validator, we also pass the field's label as a replaceable parameter. The validator can then reuse the same message for all the required fields. Alternatively, you can define your own messages to selectively override the defaults. (See section 12.4.3.)

The resource bundle will contain whatever other messages are needed by your application, along with specific labels and messages needed by the Struts Validator. Here is the block we will need for our `username` and `login` validations:

```
# -- logon --
logon.username.maskmsg=Username must be letters and numbers, no spaces.
logon.password.maskmsg=Password must be five characters long and contain a
  special character or numeral.
```

We also need labels for the `username` and `password` fields. However, these would already be provided if the application were localized:

```
logon.username.displayname=Username
logon.password.displayname=Password
```

Note that we prefix the labels and messages with `logon`. By giving each form its own namespace, we can avoid collisions as the application grows.

validator.xml

The validators and message keys are used when defining our `formset` element in the validator.xml file, as shown in listing 12.3.

Listing 12.3 A formset element

```
<!-- ❶ -->
<formset>
<!-- ❷ -->
<form name="logonForm">
<!-- ❸,❹ -->
<field
    property="username"
    depends="required,mask">
<!-- ❺ -->
    <msg
        name="mask"
        key="logon.username.maskmsg"/>
<!-- ❻ -->
    <arg0
        key="logon.username.displayname"/>
<!-- ❼,❽ -->
    <var>
        <var-name>mask</var-name>
        <var-value>^[a-zA-Z0-9]*$</var-value>
    </var>
</field>
<!-- ❾ -->
<field
    property="password"
    depends="required,minlength">
```

```
    <arg0
        key="logon.password.displayname"/>
    <var>
        <var-name>minlength</var-name>
        <var-value>5</var-value>
    </var>
</field>
</form>
<!-- ... -->
</formset>
```

❶ A `formset` is a wrapper for one or more forms.

❷ Each form element is given its own `name`. This should correspond to either the form bean name or the action path from your Struts configuration.

❸ Each `form` element is composed of a number of `field` elements.

❹ The `field` elements designate which validator(s) to use with the `depends` attribute.

❺ The optional `msg` element lets you specify a custom message key for a validator and the message key to use for any replacement parameters.

❻ The `arg0` element specifies the first replacement parameter to use with any messages that need them.

❼ The `var` element is used to pass variable properties to the validator.

❽ Here we pass a regular expression to the `mask` validator. The expression says usernames can contain only the alphabet characters and numerals.

❾ Here we say that a password is required and must be at least five characters long. The password length is a business requirement to help make the accounts more secure. The password validation message uses the default `minlength` or `required` messages, defined in validation-rules.xml (`errors.minlength` and `errors.required`).

JSP tag / logon.jsp

JavaScript validations are optional but easy to implement if you would like to use them in your application. Listing 12.4 shows the logon.jsp modified to display JavaScript validations.

Listing 12.4 logon.jsp prepared for JavaScript validations

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<!-- ❶ -->
<%@ taglib uri="/tags/struts-validator" prefix="validator" %>
<HTML><HEAD><TITLE>Sign in, Please!</TITLE></HEAD>
<BODY>
```

```
<!-- ❷ -->
<html:form action="/logonSubmit" focus="username"
    onsubmit="return validateLogonForm(this)">
<TABLE border="0" width="100%">
<TR><TH align="right">Username:</TH>
<TD align="left"><html:text property="username"/></TD>
</TR>
<TR><th align="right">Password:</TH>
<TD align="left"><html:password property="password"/></TD>
</TR>
<TR>
<TD align="right"><html:submit property="submit" value="Submit"/></TD>
<TD align="left"><html:reset/></TD>
</TR>
</TABLE>
</html:form>
<!-- ❸ -->
<validator:javascript formName="logonForm"/>
</BODY>
</HTML>
```

❶ Here we import the validator taglib.

❷ This section calls the validation script. Then, the form is submitted.

❸ Here, we add the tag to output the JavaScript anywhere on the page.

The validate method

To enable the server-side validations, all that needs to be done is to have the form bean extend ValidatorForm instead of ActionForm

```
public final class LogonForm extends
    org.apache.struts.validator.action.ValidatorForm {
```

and remove the old `validate` method. When the controller calls the `validate` method, the ValidatorForm method will kick in and follow the rules we defined in the validation.xml file.

12.3 Basic validators

As shown in table 12.2, the Struts Validator ships with 14 basic validators. These should cover the needs of most applications. If the need arises, as we will see in section 12.9, you can add custom, or plug-in, validators.

Table 12.2 Basic validators

Validator	Purpose
`required`	Succeeds if the field contains any characters other than whitespace.
`mask`	Succeeds if the value matches the regular expression given by the `mask` attribute.
`range`	Succeeds if the value is within the values given by the `min` and `max` attributes (`(value >= min) & (value <= max)`).
`maxLength`	Succeeds if the field's length is less than or equal to the `max` attribute.
`minLength`	Succeeds if the field's length is greater than or equal to the `min` attribute.
`byte`, `short`, `integer`, `long`, `float`, `double`	Succeeds if the value can be converted to the corresponding primitive.
`date`	Succeeds if the value represents a valid date. A date pattern may be provided.
`creditCard`	Succeeds if the value could be a valid credit card number.
`email`	Succeeds if the value could be a valid e-mail address.

12.3.1 The required validator

The `required` validator is both the simplest and the most commonly used of the validators:

```
<field
    property="customerId"
    depends="required"/>
```

If nothing or only whitespace is entered into a field, then the validation fails, and an error is passed back to the framework. Otherwise, the validation succeeds. To determine whether the field contains only whitespace, the standard `String.trim()` method is called (`value.trim().length() == 0`).

Since browsers do not submit empty fields, any field that isn't required will skip all validations if the field is null or has a length of zero.

12.3.2 The mask validator

The `mask` validator checks the value against a regular expression and succeeds if the pattern matches:

```
<field property="postalCode" depends="mask">
    <arg0 key="registrationForm.postalCode.displayname"/>
    <var>
        <var-name>mask</var-name
```

```
        <var-value>^\d{5}\d*$</var-value>
    </var>
</field>
```

The Jakarta RegExp package [ASF, Regexp] is used to parse the expression. If an expression needs to be used by more than one field, it can also be defined as a constant in the validation.xml file—for example:

```
    <constant>
      <constant-name>zip</constant-name>
      <constant-value>^\d{5}\d*$</constant-value>
    </constant>
```

Like most of the other standard validators, the `mask` validator is declared to be dependent on the `required` validator. Therefore, if a field depends on both `required` and `mask`, then the `required` validator must complete successfully before the `mask` validator is applied.

12.3.3 The range validator

The `range` validator checks that the value falls within a specified minimum and maximum:

```
<field property="priority"
    depends="required,integer,range">
    <arg0 key="responseForm.priority.displayname"/>
    <var>
        <var-name>min</var-name>
        <var-value>1</var-value>
    </var>
    <var>
        <var-name>max</var-name>
        <var-value>4</var-value>
    </var>
</field
```

This validator would succeed if the digit 1, 2, 3, or 4 were entered into the field. In practice, the error message should display the minimum and maximum of the range, to help the user get it right. You can use the `arg` elements to include the `min` and `max` variables in the message by reference:

```
<field property="priority"
    depends="required,integer,range">
    <arg0 key="responseForm.priority.displayname"/>
    <arg1 name="range" key="${var:min}" resource="false"/>
    <arg2 name="range" key="${var:max}" resource="false"/>
    <var>
        <var-name>min</var-name>
        <var-value>1</var-value>
```

```
    </var>
    <var>
        <var-name>max</var-name>
        <var-value>4</var-value>
    </var>
</field
```

This implies that the template for the range messages looks something like this:

```
errors.range=Please enter a value between {1} and {2}.
```

If the `range` validator fails for the priority field, the validation message would read:

```
Please enter a value between 1 and 4.
```

By default, the validator assumes that the key of an `arg` element matches a key in the resource bundle and will substitute the value of the resource entry for the value of the `key` attribute. The `resource=false` switch tells the validator to use the value as is. If the values are being rendered by a select element, you may wish to map the messages to the first and last items on the select list:

```
    <arg1 name="range" key="priority.range.first"/>
    <arg2 name="range" key="priority.range.last"/>
```

This implies that there are also entries like these

```
priority.range.first=do-it-now
priority.range.last=forget-about-it
```

in the resource bundle. If validation failed, the message for priority would read:

```
Please enter a value between do-it-now and forget-about it.
```

12.3.4 The maxLength validator

The `maxLength` validator checks the high end of the range; it succeeds if the field's length is less than or equal to the `max` attribute:

```
<field property="remarks"
    depends="maxlength">
    <arg0 key="responseForm.remarks.displayname"/>
    <arg1 name="maxlength" key="${var:maxlength}" resource="false"/>
    <var>
        <var-name>maxlength</var-name>
        <var-value>1000</var-value>
    </var>
</field
```

This field element makes sure that the length of the remarks (probably a text area field) does not exceed 1000 characters. Note that we pass the length as an argument to the validation message, as we did with the `range` validator.

12.3.5 The minLength validator

The `minLength` validator checks the low end of the range; it succeeds if the field's length is greater than or equal to the `min` attribute:

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

The `field` element stipulates that the password must be entered and must have a length of at least five characters.

12.3.6 The byte, short, integer, long, float, and double validators

These validators all apply the standard `type.parseType` methods to the value. If an Exception is caught, the validator returns false. Otherwise, it succeeds:

```
<field property="amount"
    depends="required,double">
<arg0 key="typeForm.amount.displayname"/>
</field
```

12.3.7 The date validator

The `date` validator checks to see if the value represents a valid date:

```
<field property="date"
    depends="required,date">
    <arg0 key="typeForm.date.displayname"/>
</field>
```

The validator passes the standard Locale object (`java.util.Locale`) maintained by the framework to the date utilities, so the result is automatically localized. The `datePattern` attribute will pass a standard date pattern to a `java.text.SimpleDateFormat` object:

```
    <var>
        <var-name>datePattern</var-name>
        <var-value>MM/dd/yyyy</var-value>
    </var>
```

Internally, the `datePattern` attribute is used in the SimpleDateFormat constructor and then used to parse the value:

```
SimpleDateFormat formatter = new SimpleDateFormat(datePattern);
Date date = formatter.parse(value);
```

If the parse succeeds, the validator succeeds.

If the `datePatternStrict` attribute is set instead, the length is also checked to ensure a leading zero is included when appropriate:

```
<var>
    <var-name>datePatternStrict</var-name>
    <var-value>MM/dd/yyyy</var-value>
</var>
```

When no pattern is specified, the `DateFormat.SHORT` format for the user's Locale is used. If the Struts Locale object is not available, the server's default Locale is used. The `setLenient` method is set to false for all date transformations.

12.3.8 The creditCard validator

The `creditCard` validator analyzes the value to see if it could be a credit card number:

```
<field property="creditCard"
    depends="required,creditCard">
    <arg0 key="typeForm.creditCard.displayname"/>
</field>
```

Credit card numbers include a parity-check digit. The validation checks for this digit and other business rules, such as whether the card's prefix matches one of the credit card vendors (American Express, Discover, MasterCard, VISA) and whether the length of the number is correct for the indicated vendor.

12.3.9 The email validator

The `email` validator employs an *extensive* check of the format of a prospective e-mail address to be sure it is in accordance with the published specification:

```
<field property="email"
    depends="required,email">
    <arg0 key="typeForm.email.displayname"/>
</field>
```

12.4 Resource bundles

The underlying purpose of validation is to get the user to fix the input and try again. Since that process involves displaying field labels and messages, the Struts Validator makes good use of the Java localization features and the framework's support for those features. (The Java localization features are covered in chapter 13.)

Of course, the framework also needs to provide localized labels for the fields. Since the Struts resource bundle is offered as an application-level resource, the Validator is able to share the same bundle with the framework, so you can keep all your labels and messages together. You simply need to add default messages for the validators you are using and any custom messages needed while validating a particular field.

12.4.1 The default bundle

The Struts bundle is configured through the deployment descriptor (see chapter 4). It is usually named ApplicationResources.properties or just application.properties. In our example applications, the bundles are stored in a package under /WEB-INF/src/java/resources. Our default Properties files (`java.util.Properties`) are named application.properties. Files for supported locales are then named application_*en*.properties, application_*es*.properties, application_*fr*.properties, and so forth. (Again, see chapter 13 for more about localization.)

You do not need to do anything special to start using the Struts resource bundle with the Validator. Just add whatever messages or labels you may need and then refer to the resource key in the Validator's configuration file. If the application is being localized, keys for the field labels should already be present, and you can share them with the rest of the framework, as shown here:

```
<field property="lastName"
    depends="required">
    <arg0 key="registrationForm.lastName.displayname"/>
</field>
```

12.4.2 Default validator messages

If a custom message is not provided by a field element, the default message for the validator is used when the validation fails. The key for a validator's default message is specified when it is defined. The convention is to add an `errors.` prefix to the validator's name. The default message for the `required` validator then becomes `errors.required`.

This convention dovetails with the Struts `<html:error>` tag, which looks for *errors.header* and *errors.footer* entries. Listing 12.5 shows the keys and templates you could add to an English resource bundle for the basic validators.

Listing 12.5 Default messages for the basic validators

```
# Struts Validator Basic Error Messages
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 not a valid e-mail address.
```

You may note that there is no entry for `errors.mask`. For historical reasons, the definition for the `mask` validator specifies `errors.invalid` instead of `errors.mask`.

Each of the validator messages can take up to four replacement parameters, which are specified as `arg` elements within the field definition. The first parameter, `arg0` or `{0}`, is usually the key for the field label. The validator will then look up the display text for the label from the resource bundle and merge the localized text into the message template.

12.4.3 Custom validator messages

A field can also specify a custom validation message to use instead of the default. This often happens when the `mask` validator is being used, so you can explain what pattern the regular expression expects. The key for the message is given in a `msg` element. Since more than one validator may be used by a field, the name of the validator is included as the `msg` element's `name` attribute:

```
<field
    property="username"
    depends="required,mask">
    <msg
        name="mask"
        key="logon.username.maskmsg"/>
    <arg0
        key="logon.username.displayname"/>
    <var>
        <var-name>mask</var-name>
        <var-value>^[a-zA-Z0-9]*$</var-value>
    </var>
</field>
```

In the Properties file, we could then place a key/template entry like this:

```
logon.username.maskmsg={0} must be letters and numbers, no spaces.
```

12.5 Configuration files

The strength of the Struts Validator is that the validations are declared outside the application source code using an XML configuration file. The configuration specifies which fields on a form need validation, the validators a field uses, and any special settings to be used with a field. Alternate formsets can be configured for different locales and override any locale-sensitive validations.

All of the validators used by the framework are configured through XML, including the basic validators that ship with the package. You can omit validators that your application doesn't need and plug in your own custom validators to use alongside those that ship with the framework.

This makes for a very flexible package, but combining all these configurations into a single file can result in a verbose document. As shown in table 12.3, the Struts Validator can actually use two XML files: one to set up the validators and another with the settings for your applications. (As noted, Struts 1.0 uses a single configuration file.)

Table 12.3 Struts Validator configuration files

Filename	Description
validator-rules.xml	Configuration files for the validators
validation.xml	The validations for your application

This makes for a very convenient arrangement. This way, it's easy to copy a standard set of validation rules between applications and then customize the validation.xml file. Or, if you prefer, you can still combine all the elements in a single validation.xml.

In Struts 1.1, the paths to the Validator configuration files are declared in the struts-config.xml file, as shown here:

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

See chapter 4 for more about installing the Struts Validator 1.1 components into your application.

12.6 Validator JSP tags

The Struts Validator framework combines the (optional) JavaScripts needed to validate a form into a single script. You insert the script into your JavaServer Page via a custom tag. The script can then be called when the form is submitted. If the validation fails, the script displays an error message window, like the one back in figure 12.1, and the submit fails.

Otherwise, the submit succeeds and control passes to the `validate` method on the Struts ActionForm. This ensures that the validations are triggered even when JavaScript is not available.

In listing 12.4, we introduced the `<javascript>` tag. For simplicity, this listing retained the original `<html:errors>` tag. The original `<errors>` tag is quite easy to use on the page but requires that you mix markup in with the message. Otherwise, if the messages are presented as a block, they all run together into a single paragraph.

Under the Struts Validator framework, error messages are shared with the JavaScript validations. In practice, using the same markup in both cases is problematic. A much better way to go would be to use plain messages all around.

The Struts Validator taglib provides additional tags that help with this very problem. The `<errorsPresent>` or `<messagesPresent>` tag reports whether any messages are pending. The `<messages>` tag works like an iterator, so your page can loop through the queue, providing any markup entries needed along the way.

Listing 12.6 shows the code from listing 12.4 again, this time outfitted with the messages tags. Listing 12.7 shows the Struts 1.1 version.

Listing 12.6 Logon page with validations and additional JSP tags (1.0)

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<!-- ❶ -->
<%@ taglib uri="/tags/struts-validator" prefix="validator" %>
<HTML><HEAD><TITLE>Sign in, Please!</TITLE></HEAD>
<BODY>
<!-- ❷ -->
<validator:errorsPresent>
<UL>
<!-- ❸ -->
<validator:errors id="error">
<LI><bean:write name="error"/></LI>
</validator:errors>
```

```
</UL>
</validator:errorsPresent>
<!-- ❹ -->
<html:form action="/logonSubmit" focus="username"
    onsubmit="return validateLogonForm(this)">
<TABLE border="0" width="100%">
<TR><TH align="right">Username:</th>
<TD align="left"><html:text property="username"/></TD>
</TR>
<TR><th align="right">Password:</TH>
<TD align="left"><html:password property="password"/></TD>
</TR>
<TR>
<TD align="right"><html:submit property="submit" value="Submit"/></TD>
<TD align="left"><html:reset/></TD>
<!-- ❺ -->
<TD align="left"><html:cancel onclick="bCancel=true"/></TD>
</TR>
</TABLE>
</html:form>
<!-- ❻ -->
<validator:javascript formName="logonForm"/>
</BODY>
</HTML>
```

Listing 12.7 Logon page with validations and additional JSP tags (1.1)

```
<%@ taglib uri="/tags/struts-html" prefix="html" %>
<!-- ❶ -->
<%@ taglib uri="/tags/struts-validator" prefix="validator" %>
<%@ taglib uri="/tags/struts-logic" prefix="logic" %>
<HTML><HEAD><TITLE>Sign in, Please!</TITLE></HEAD>
<BODY>
<!-- ❷ -->
<logic:messagesPresent>
<UL>
<!-- ❸ -->
<logic:messages id="error"
<LI><bean:write name="error"/></LI>
</logic:messages>
</UL>
</logic:messagesPresent>
<!-- ❹ -->
<html:form action="/logonSubmit" focus="username"
    onsubmit="return validateLogonForm(this)">
<TABLE border="0" width="100%">
<TR><TH align="right">Username:</th>
<TD align="left"><html:text property="username"/></TD>
</TR>
<TR><th align="right">Password:</TH>
<TD align="left"><html:password property="password"/></TD>
```

```
</TR>
<TR>
<TD align="right"><html:submit property="submit" value="Submit"/></TD>
<TD align="left"><html:reset/></TD>
<!-- ❺ -->
<TD align="left"><html:cancel onclick="bCancel=true"/></TD>
</TR>
</TABLE
</html:form>
<!-- ❻-->
<validator:javascript formName="logonForm"/>
</BODY>
</HTML>
```

Here are some remarks that apply to both listings 12.6 and 12.7:

❶ Since we will be using the Struts Validator tags, we need to import the taglib. In Struts 1.1, we should also import the logic taglib, since it now includes tags we can use here.

❷ The Struts 1.0 error tags, or Struts 1.1 message tags, let us keep the HTML in the page and out of the error messages. If there are no messages, this entire block is skipped.

❸ The `<errors>` or `<messages>` tag works like an iterator. The tag will expose each message under the `id` `errors` so that the bean tag can write them out in turn.

❹ A JavaScript `onsubmit` event has been added to the `<html:form>` tag. This will call our validator JavaScript when any Submit button is pushed or a JavaScript `submit` event is triggered.

❺ To allow the user to cancel the submit and bypass the validations, a JavaScript flag can be set. If the Submit button sets `bCancel` to true, then the Struts Validator will pass control through to the Action.

❻ Last, but not least, is the actual `<javascript>` tag. At runtime, this will be replaced with a script combining the JavaScript validators for this form. The script, like the ActionForm, is named after the `action-mapping` attribute name, which in this case is `logonForm`.

The basic validators provided with the framework all include JavaScript validations. If a pluggable validator (section 12.9) provides a JavaScript validation, it will be included here as well. For the script to succeed, all the validations must succeed.

The generated JavaScript observes the `page` property of the ValidatorForm and will only generate validators for fields with a page number equal to or less than the form's page number. The default page number for both is 0. This is useful for multipage wizard forms. (See section 12.10.1.)

12.7 ValidatorForm and ValidatorActionForm

To enable the Struts Validator for Struts 1.1, just follow the initial setup instructions in chapter 4 and extend your ActionForm from ValidatorForm and ValidatorActionForm. The ValidatorForm will match the `formset` name with the `form-bean` name. The ValidatorActionForm will match the `formset` name with the `action-mapping` path.

In most cases, the Struts Validator can completely replace the need to write a custom `validate` method for an ActionForm. However, should you still need a `validate` method, it can easily work alongside the validation framework. In most cases, you would want to call the validator framework first, and then run your own validations if these pass. Listing 12.8 shows how this is done.

Listing 12.8 Logon page with validations

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

    // ❶
    ActionErrors errors = super.validate(mapping, request);

    // ❷
    if (errors==null) errors = new ActionErrors();
    if (errors.empty()) {
         // ❸
       if (notGood(mapping,request)) errors.add(ActionErrors.GLOBAL_ERROR,new
            ActionError("errors.notGood","goodProperty"));
    }

    if (errors.empty()) return null;
    return errors;

    return (errors);

}
```

❶ First, we call the ancestor `validate` method. In this case, we're calling the Validator framework.

❷ Since the ancestor method could return null, we need to check for null first. Since we want to run our own validations, which may need to post errors, we create a new `ActionErrors` object if one wasn't returned. This implementation does not run our validations if the ancestor returns errors, though you could just as easily run your own if that was appropriate.

❸ Our sample validation calls a helper method called `notGood`, which returns the result of our custom validation. We pass the parameters from the `validate` method to be sure `notGood` knows everything `validate` knows.

12.8 Localized validations

The Validator configuration files include a `formset` element. The formset is a wrapper for a collection of forms that share a common locale setting. While some fields in some forms do need to be localized, usually the majority do not. The Struts Validator allows you to localize selected fields and use the default validations for the rest. The default formset omits the language, country, and variant properties. The localization is properly scoped; you can define a format to override just the language, or just the country, or both if need be. For more about internationalizing your applications, see chapter 13.

12.9 Pluggable validators

Each field element can specify a list of validators on which it depends. Some applications will use very few validators; others will use several. To allow developers to load only the validators their application needs and to make it easy to load custom objects, the validators are declared from a configuration file, essentially making them all pluggable.

DEFINITION *Pluggable* refers to an object-oriented design strategy that allows objects to be developed independently of an application and then incorporated without changing the base code. Pluggable components are often created by a third party.

12.9.1 Creating pluggable validators

Creating your own validator and plugging it in is a two-step process:

1. Create a method in a Java class to handle the server-side validations.
2. Update the validator-rules.xml file with an element for your validator. If your validator will have a client-side JavaScript component, you can make this part of the `validator` element.

Creating a validation method

Any class can be used to store your `validate` method. For Struts 1.0, the method must follow a specific signature:

```
public static boolean validateMyValidator(
        Object bean,
        ValidatorAction va,
        Field field,
        ActionErrors errors,
        HttpServletRequest request);
```

Table 12.4 provides a key to the parameters passed to your `validate` method.

Table 12.4 Validator method properties

Property	Description
`Object bean`	The bean on which validation is being performed.
`ValidatorAction va`	This is a JavaBean that represents the `validator` element for this validator from the validator-rules.xml. The bean is declared in the `org.apache.commons.validator` package.
`Field field`	This JavaBean represents the element for the field we are to validate. The bean is declared in the `org.apache.commons.validator` package.
`ActionErrors errors`	An ActionErrors object that will receive any validation errors that may occur.
`HttpServletRequest request`	The current request object underlying this operation.

In Struts 1.1, you may specify the signature for your method as part of the configuration.

For coding hints on how to get your pluggable validator to do what you need, see the `org.apache.struts.util.StrutsValidator` class. This class contains the methods for the basic validators that ship with the distribution. Your validations should work in exactly the same way as the basic ones, which themselves are essentially all plug-ins.

STRUTS TIP When developing pluggable validators, keep an eye on the log file. Many programming errors will only be logged and will not be exposed in the application. A good example is getting the package name wrong in the validator-rules.xml. A `ClassNotFound` exception will be logged, but the application will act as if the validation succeeded!

Declaring a validator element

Listings 12.1 and 12.2 show the configuration element, which is the same one you would use for your own plug-in. The `required` portion is simply:

```
<validator
    name="required"
    classname="org.apache.struts.validator.util.StrutsValidator"
    method="validateRequired"
    msg="errors.required"/>
```

This element tells the Struts Validator which method to call on what class, and the message key to use when the validation fails. (Hey, it has to fail some time; otherwise, what's the point?)

In Struts 1.1, you can also specify the parameters your `validate` method will use:

```
<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"/>
```

In addition to declaring the server-side validation, you can specify a client-side JavaScript to use with your validator. The various JavaScript elements in the stock validator-rules.xml provide several working examples of how to write a compatible script.

12.10 Techniques

In addition to the everyday uses we've covered so far, there are several special techniques you can use with the Struts Validator. These include:

- Multipage validations
- Cancel buttons
- Custom messages
- Interrelated fields
- The combining of validators with the `validate` method

12.10.1 Multipage validations

Many developers like to use wizard forms. A wizard gathers the information needed for an operation through several different forms. Then, it combines the results at the end. Since not so much information is provided at once, it can be easier for a user to complete a large form. Some developers use a different form for each page. Others like to use one big form and expose only part of it at a time.

If you are using the one-big-form wizard approach, the Struts Validator includes a `page` property on the field element and provides a corresponding `page` property on the ValidatorForm. Before performing the validation on a field, the framework checks to see if the field page is less than or equal to the ValidatorForm page. This means that as part of validating page 3, we also double-check the validations for pages 1 and 2. This is to help to keep people from skipping pages.

If your one-big-form wizard has a `reset` method, you can also use the `page` property to reset values only for the current page:

```
if (page==1) {
    // reset page 1 properties
}
if (page==2) {
    // reset page 2 properties
}
```

12.10.2 Cancel buttons

Most forms give the user the opportunity to cancel the operation altogether, usually with a Cancel button. The problem here is that the Cancel button submits the form—so the JavaScript tries to validate it. The server-side pieces can be made to acknowledge the Cancel button, but the JavaScript is client-side and unaware of the framework. So, to cancel a form, a user must first appease the JavaScript validations. Not good.

To resolve this dilemma, the Struts Validator provides a `bCancel` JavaScript variable. If `bCancel` is set to true, the JavaScript validations will also return true, allowing the cancel request to pass through to the container. The server-side validations know about the Struts `<cancel>` button and will not fire if they see it in the request. The rest is up to the Action, which should check its own `isCancelled` method before committing any operation that a user might relay:

```
<html:cancel onclick="bCancel=true;">
<bean:message key="button.cancel"/>
</html:cancel>
```

12.10.3 Custom messages

Each validator can define a default error message to use when the validation fails. For example,

```
errors.integer={0} must be an integer.
```

would automatically display

```
Quantity must be an integer.
```

when someone tried to enter a letter into a field labeled Quantity.

For most validators, this works just fine, and the default message is all you need. The exception is the `mask` validator, which is used with regular expressions. Here, the default message is

```
errors.invalid={0} is invalid.
```

If used with a `mask` on a form for entering a new password that needed to be at least five characters long, the default message would be

```
Password is invalid.
```

which doesn't tell users what they need to do to fix it. A better message would be

```
Password must be five or more characters long.
```

Of course, that message would not work as well for some other field that used the `mask` validator with a different regular expression.

In most cases, whenever you use a `mask` validator, you should also specify a custom error message that explains what the regular expression expects:

```
<field property="password" depends="required,mask">
    <msg name="mask" key="accountForm.password.mask"/>
    <arg0 key="nameForm.password.displayname"/>
    <var>
        <var-name>mask</var-name>
        <var-value>^{5}*$</var-value>
    </var>
</field>
```

And in the application resources:

```
accountForm.password.mask ={0} must be five or more characters long.
```

While the message attribute is most often used with the `mask` validator, you can override the message for any validator on a field-by-field basis.

12.10.4 Interrelated fields

If users change their password, it's commonplace for a program to have users input the new password twice, to help ensure they have typed it correctly. Many other fields in a form may also be interrelated in some way. You can compare fields in any way you like by defining your own plug-in validator.

Used together in the same application, the components shown in listings 12.9, 12.10, and 12.11 create a plug-in validator that compares two fields to be sure they are identical.

Listing 12.9 validator-rules.xml

```
<validator name="identical"
    classname="com.mysite.StrutsValidator"
    method="validateIdentical"
    depends="required"
    msg="errors.identical"/>
```

Listing 12.10 validate.xml

```
<field property="password"
    depends="required,identical">
    <arg0 key="accountForm.password.displayname"/>
    <var>
        <var-name>secondProperty</var-name>
        <var-value>password2</var-value>
    </var>
</field
```

Listing 12.11 apps.ValidateUtils

```
public static boolean validateIdentical(
        Object bean,
        ValidatorAction va,
        Field field,
        ActionErrors errors,
        HttpServletRequest request) {

    String value = ValidatorUtil.getValueAsString(bean,
        field.getProperty());
    String sProperty2 = field.getVarValue("secondProperty");
    String value2 = ValidatorUtil.getValueAsString(bean, sProperty2);

    if (!GenericValidator.isBlankOrNull(value)) {
        try {
            if (!value.equals(value2)) {
                errors.add(field.getKey(),
             ValidatorUtil.getActionError(application, request, va, field));
                return false;
```

```
                }
            }
                catch (Exception e) {
                errors.add(field.getKey(), ValidatorUtil.getActionError(
                    application, request, va, field));
                return false;
            }
        }
        return true;
    }
```

12.10.5 Combining validators with the validate method

Plug-in validators, like the one shown in section 12.10.4, can be used to ensure more complex relationships between fields are maintained. For example, if something may be picked up or shipped, and users choose Ship, you could plug in a validator to be sure a shipping option had been chosen.

The best candidates for plug-in validators are ones that you can reuse on several forms. If the validation would not be reusable, you can also override the `validate` method and use it in the normal way. Just be sure to also call the ancestor `validate` method, to ensure that any framework validations are triggered. Listing 12.12 shows how to combine a custom `validate` method with the Struts Validator.

Listing 12.12 Combining validate with validators

```
public ActionErrors validate(ActionMapping mapping,
        HttpServletRequest request) {
    ActionErrors errors = super.validate(mapping, request);

    if (errors==null) errors = new ActionErrors();
    // If selects shipping
    if ("S".equals(deliveryType)) {
        // Vendor required
        if ("".equals(getShipVendor().trim())) {
            errors.add(ActionErrors.GLOBAL_ERROR,
            new ActionError("item.shipVendor.maskmsg"));
    }

    if (errors.empty()) return null;
    return errors;
}
```

12.11 Migrating an application to the Struts Validator

While many Struts applications are written to use the Struts Validator from the get-go, most start out using their own routines before moving to the Validator. In this section, we walk through migrating a simple ActionForm `validate` method to its Struts Validator counterpart. The point of the exercise is not the example `validate` method, which is trivial, but the process of moving the method to the Struts Validator.

In chapter 11, we followed a similar process by migrating some example pages to Tiles.

12.11.1 Setting up the Validator framework

Setting up the validator varies slightly between Struts 1.0 and 1.1, but works just as well with either version.

NOTE Before you begin, make an extra backup of everything, regardless of how many backups are made in the normal course. Migrating to the Validator can be tricky at first, and, realistically, you may need to make more than one pass before everything clicks into place. So, be ready to roll back and try again. 'Nuff said.

Struts 1.0

If you haven't done so, the first step is to install the Validator package and load the Validator servlet through your application's deployment descriptor. (The Validator servlet is just a resource loader and so does not conflict with the Tiles servlet.) Then, test your application to be sure all is well by clicking through a few pages.

The Blank application for Struts 1.0 on the book's website [Husted] includes an empty Validator configuration file and sample setup.

Struts 1.1

The Validator is integrated with Struts 1.1. The steps for enabling the Validator in Struts 1.1 are covered in section 4.9 of this book.

12.11.2 Testing the default configuration

Set the `debug` and `detail` parameters in the deployment descriptor (web.xml) to level 2, and restart the application. Check the log entries carefully for any new error messages. Run any unit tests and click through the application to confirm that operation is still nominal.

12.11.3 Reviewing your validations

With the Validator up and running, the next step is to take a good hard look at your `validate` methods. Identify which will correspond to a standard Struts Validator validation and which will have to be handled on a custom basis. The Validator is not an either/or proposition. You can continue to use the ActionForm `validate` method to handle some things, and the Validator to handle the rest.

Listing 12.13 shows a code skeleton that calls the Struts Validator and then tries any custom validations.

Listing 12.13 Calling the Validator and your own custom routines

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

           // ❶
        ActionErrors errors = super.validate(mapping, request);

           // ❷
        if (null==errors) errors = new ActionErrors();

        /* ❸
              if (!(dateCheck()))
            errors.add(ActionErrors.GLOBAL_ERROR,
                 new ActionError("errors.invalid","Expiration Date"));
        */

          // ❹

        // ❺
        if (errors.empty()) return null;
        return errors;

} // end validate
```

❶ This method is meant to be used with a ValidatorForm subclass. By calling and capturing the super class `validate` method, we run any of the Struts Validator validations but leave room for calling our own.

❷ If the super class validations all pass, `validate` will return null. We might still find some errors of our own, so we create an ActionErrors collection to hold them. If the ValidatorForm super class returns some errors, we continue to use the same collection. This will provide a unified set of error messages to the user, regardless of which `validate` method ran the validation.

❸ This is an example of a commented-out validation that has been replaced by the Struts Validator. Once the method is fully tested, this would be removed.

❹ Other validation routines that have not been moved to the Struts Validator can run here.

❺ If there were no errors, this code returns null; otherwise, it returns the combined list of errors—the Struts Validator's and any of our own.

The moral of this method? If a standard validation won't be able to handle some of your validation routines, plan to leave them in for now. You might want to replace these routines with a custom validator later, but pick the low-hanging fruit first, and get the standard versions working.

12.11.4 Extending ValidatorForm or the Scaffold BaseForm

Before making any code changes to your ActionForm, ensure that it extends either the ValidatorForm or the Scaffold BaseForm:

```
import com.wintecinc.struts.action.ValidatorForm; //  Struts 1.0.x
// import org.apache.struts.validator.ValidatorForm; //  Struts 1.1

public class ActionForm extends ValidatorForm {
      // . . .
}
```

or

```
import org.apache.scaffold.struts.BaseForm;

public class ActionForm extends BaseForm {
      // . . .
}
```

In either case, you should be able to rebuild and test the application without error. Neither class changes the default ActionForm functionality. If you do expose any errors at this point, be sure to resolve them before continuing.

12.11.5 Selecting a validation to migrate

Let's look at a simple example validation from an early version of the Artimus example application [ASF, Artimus]. It tests to be sure that some required fields were present before storing an article. Listing 12.14 shows the complete class.

Listing 12.14 A simple validate method (org.apache.artimus.articles.struts.ActionForm)

```
package org.apache.artimus.article.struts;
import javax.servlet.http.HttpServletRequest;
import org.apache.struts.action.ActionError;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.action.ActionMapping;
import org.apache.artimus.struts.Form;

public class ArticleForm extends Form {

            // ❶
    private boolean isNotPresent(String field) {
        return ((null==field) || ("".equals(field)));
    }

     public ActionErrors validate(
            ActionMapping mapping,
            HttpServletRequest request) {

            // ❷
               ActionErrors errors = new ActionErrors();

            // ❸
            String title = getTitle();
        if (isNotPresent(title)) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                new ActionError("errors.required","Title"));
        }
        String content = getContent();
        if (isNotPresent(content)) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                new ActionError("errors.required","Article text"));
        }
        String creator = getCreator();
        if (isNotPresent(creator)) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                new ActionError("errors.required","Author"));
        }

            if (errors.empty()) return null;
            return errors;

    } // end validate
//  end ArticleForm
```

❶ `isNotPresent` is a simple utility method to test whether a field is null or empty.

❷ This code creates an ActionErrors collection, so that it's there if we need it.

❸ This code runs our three simple validations. If any fail, it creates an ActionError. The `errors.required` template is used to create the ActionError, and the field is merged into the message.

As validation routines go, this is no biggie, but it's better to start with something simple and work your way up. Let's see how the title validation routine would be migrated to the Struts Validator.

12.11.6 Adding the formset, form, and field elements

First, add a default `<formset>` to the Validator configuration file (validation.xml), and add to that an element for the form and a `field` element for the property. Listing 12.15 shows the initial `<formset>` for our `articleForm` example.

Listing 12.15 An initial <formset> for the Struts Validator

```
<formset>
   <form name="articleForm">
      <field
         property="title"
         depends="required">
            <arg0 key="Title" resource="false"/>
      </field>
   </form>
</formset>
```

The title validation routine in listing 12.11 is just a test that it is required. Of course, the Struts Validator has a standard validator for that. In listing 12.15, we specify that in the form `articleForm`, the field `title` depends on the `required` validator. Our error message includes the field's name as the `{0}` replacement parameter, so we can pass that to the field as the `<arg0>` element.

12.11.7 Adding new entries to the ApplicationResources

The `<field>` element in listing 12.15 uses the `resource="false"` attribute to pass a literal `String`. At this point, it's better to get on the right track and extract the language elements into the ApplicationResources bundle. When you are writing your own validation methods, it's easy to embed these language elements into the Java code. But when you are using the Struts Validator, it's just as easy to put them in the ApplicationResources bundle (where they belong). Listing 12.16 shows our initial `<formset>` with a reference to the ApplicationResources bundle.

Listing 12.16 An initial <formset>for the Struts Validator

```
<formset>
   <form name="articleForm">
      <field
         property="title"
         depends="required">
            <arg0 key="article.title.displayname"/>
      </field>
   </form>
</formset>
```

We then need to add the appropriate entry to our ApplicationResources file (/WEB-INF/src/java/resources/application.properties):

```
# /**
#  * Messages for Artimus application
#  */

# ...

# -- article Form fields --
article.title.displayname=Title
article.creator.displayname=Creator
article.content.displayname=Content

# ...
```

12.11.8 Calling the Struts Validator

In the Java source for the ArticleForm class, we can now call the ValidatorForm super class to validate our `title` field. Initially, we can just comment out the original validation and leave everything else intact. Listing 12.17 shows the revised `validate` method for our `articleForm` example. This is the code skeleton from listing 12.13 applied to listing 12.16.

Listing 12.17 A revised validate method

```
public ActionErrors validate(
            ActionMapping mapping,
            HttpServletRequest request) {
       ActionErrors errors = super.validate(mapping, request);
       if (null==errors) errors = new ActionErrors();
/*
        String title = getTitle();
        if (isNotPresent(title)) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                new ActionError("errors.required","Title"));
        }
*/
        String content = getContent();
        if (isNotPresent(content)) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                new ActionError("errors.required","Article text"));
        }
        String creator = getCreator();
        if ((isNotPresent(creator)) {
            errors.add(ActionErrors.GLOBAL_ERROR,
                new ActionError("errors.required","Author"));
        }
        if (errors.empty()) return null;
        return (errors);
    } // end validate
//  end ArticleForm
```

12.11.9 Test and repeat

Rebuild and reload the application to be sure the latest versions of *everything* are in place. Try to defeat the validation. When operation is deemed nominal, continue to the next validation routine. In our example, we ended up with this:

```
<formset>
      <form name="articleForm">
         <field
            property="title"
            depends="required">
               <arg0 key="article.title.displayname"/>
         </field>
         <field
            property="creator"
            depends="required">
               <arg0 key="article.creator.displayname"/>
         </field>
         <field
```

```
            property="contentDisplayHtml"
            depends="required">
               <arg0 key="article.content.displayname"/>
         </field>
      </form>
</formset>
```

12.11.10 Removing the ActionForm subclass

Our sample ActionForm subclassed a coarse-grained ActionForm (see chapter 5). The ArticleForm class itself provided nothing but a `validate` method that acted on the inherited properties. All of the bean's properties are defined in a base class.

In this case, once all the validations are transferred to the Struts Validator, we can just remove the class and update the `<form-bean>` element to use the base class instead.

What was:

```
  <form-beans>
<form-bean
    name="baseForm"
    type="org.apache.artimus.struts.Form"/>
 <form-bean
    name="articleForm"
    type="org.apache.artimus.article.struts.ActicleForm"/>
        <!-- ... -->
 </form-beans>
```

can now be:

```
  <form-beans>
<form-bean
    name="baseForm"
    type="org.apache.artimus.struts.Form"/>
 <form-bean
    name="articleForm"
    type="org.apache.artimus.struts.Form"/>
        <!-- ... -->
 </form-beans>
```

The same ActionForm class can be used by any number of form beans, just as an Action class can be used by any number of action mappings. Each instance is just given a different attribute name, and the Struts Validator matches the `<form>` elements by the attribute name. With a coarse-grained ActionForm in place to define our properties, the Struts Validator `<form>` element does the work of a subclass.

In Struts 1.1, you can use the DynaValidatorForm class to avoid declaring any new ActionForm class whatsoever. You can declare whatever simple properties your form needs as part of the `<form-bean>` element.

Here's the <form-bean> configuration for the Artimus 1.1 ArticleForm:

```
<form-bean
  name="articleForm"
     type="org.apache.struts.validator.DynaValidatorForm">
    <form-property
      name="keyName"
      type="java.lang.String"/>
    <form-property
      name="keyValue"
      type="java.lang.String"/>
    <form-property
      name="marked"
      type="java.lang.String"
      initialValue="0"/>
    <form-property
      name="hours"
      type="java.lang.String"/>
    <form-property
      name="articles"
      type="java.lang.String"/>
    <form-property
      name="article"
      type="java.lang.String"/>
    <form-property
      name="contributor"
      type="java.lang.String"/>
    <form-property
      name="contributedDisplay"
      type="java.lang.String"/>
    <form-property
      name="creator"
      type="java.lang.String"/>
    <form-property
      name="title"
      type="java.lang.String"/>
    <form-property
      name="contentDisplayHtml"
      type="java.lang.String"/>
  </form-bean>
```

The validator <form> for this DynaBean is shown in section 12.11.9.

These two XML constructs replace writing a conventional ActionForm class and its Validate method.

12.12 Summary

The Struts Validator is a powerful addition to the framework. It allows validations to be managed in a separate configuration file, where they can be reviewed and modified without changing Java or JavaServer Page code. This is important since, like localization, validations are tied to the business tier and should not be mixed with presentation code.

The framework ships with several basic validators that will meet most of your routine needs. You can easily add custom validators for special requirements. If required, the original Struts `validate` method can be used in tandem with the Struts Validator to be sure all your needs are met.

Like the main Struts framework, the Struts Validator is built for localization from the ground up. It can even share the standard message resource file with the main framework, providing a seamless solution for your translators.

Validating input is an essential service that a web application framework must provide, and the Struts Validator is a flexible solution that can scale to meet the needs of even the most complex applications.