

Validator Overview

Before getting into the details of using the Validator framework, it's necessary to give an overview of how Validator works. Recall that without Validator, you have to code all of your form data validations into the **validate()** methods of your Form Bean objects. Each Form Bean field that you want to perform a validation on requires you to code logic to do so. Additionally, you have to write code that will store error messages for validations that fail. With Validator, you don't have to write any code in your Form Beans for validations or storing error messages. Instead, your Form Beans extend one of Validator's **ActionForm** subclasses that provide this functionality for you.

The Validator framework is set up as a pluggable system of validation routines that can be applied to Form Beans. Each validation routine is simply a Java method that is responsible for performing a specific type of validation and can either pass or fail. By default, Validator comes packaged with several useful validation routines that will satisfy most validation scenarios. However, if you need a validation that is not provided by the framework, you can create your own custom validation routine and plug it into the framework.

Validator uses two XML configuration files to tell it which validation routines should be "installed" and how they should be applied for a given application, respectively. The first configuration file, **validator-rules.xml**, declares the validation routines that are plugged into the framework and assigns logical names to each of the validations. Additionally, the **validator-rules.xml** file is used to define client-side JavaScript code for each validation routine. If configured to do so, Validator will emit this JavaScript code to the browser so that validations are performed on the client side as well as the server side. The second configuration file, **validation.xml**, defines which validation routines are applied to which Form Beans. The definitions in this file use the logical names of Form Beans from the [struts-config.xml](#) file along with the logical names of validation routines from the **validator-rules.xml** file to tie the two together.

Using Validator

Using the Validator framework involves enabling the Validator plugin, configuring Validator's two configuration files, and creating Form Beans that extend Validator's **ActionForm** subclasses. The following sections explain how to configure and use the Validator in detail.

Enabling the Validator Plugin

Although the Validator framework comes packaged with Struts, by default Validator is not enabled. In order to enable and use Validator, you have to add to your application's [struts-config.xml](#) file the following definition for the **plug-in** tag:

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

This definition causes Struts to load and initialize the Validator plugin for your application. Upon initialization, the plugin loads the comma-delimited list of Validator configuration files specified by the **pathnames** property. Each configuration file's path should be specified using a Web application-relative path, as shown in the preceding example.

Note that your application's [struts-config.xml](#) file must conform to the Struts Configuration DTD, which specifies the order in which elements are to appear in the file. Because of this, you must place the Validator **plug-in** tag definition in the proper place in the file. The easiest way to ensure that you are properly ordering

elements in the file is to use a tool like Struts Console that automatically formats your configuration file so that it conforms to the DTD.

Configuring validator-rules.xml

The Validator framework is set up as a pluggable system whereby each of its validation routines is simply a Java method that is plugged into the system to perform a specific validation. The **validator-rules.xml** file is used to declaratively plug in the validation routines that Validator will use for performing validations. Struts' example applications come packaged with preconfigured copies of this file. Under most circumstances, you will use these preconfigured copies and will not need to modify them unless you are adding your own custom validations to the framework.

Following is a sample **validator-rules.xml** file that illustrates how validation routines are plugged into Validator:

```
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons
    Validator Rules Configuration 1.0//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <global>
    <validator name="required"
      classname="org.apache.struts.validator.FieldChecks"
      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 isValid = true;
  var focusField = null;
  var i = 0;
  var fields = new Array();
  oRequired = new required();
  for (x in oRequired) {
    var field = form[oRequired[x][0]];

    if (field.type == 'text' ||
        field.type == 'textarea' ||
        field.type == 'file' ||
        field.type == 'select-one' ||
        field.type == 'radio' ||
        field.type == 'password') {

      var value = '';

      // get field's value
      if (field.type == "select-one") {
        var si = field.selectedIndex;
        if (si >= 0) {
          value = field.options[si].value;
        }
      } else {
        value = field.value;
      }

      if (trim(value).length == 0) {
        if (i == 0) {
          focusField = field;

```

```

        }
        fields[i++] = oRequired[x][1];
        isValid = false;
    }
}

if (fields.length > 0) {
    focusField.focus();
    alert(fields.join('\n'));
}

return isValid;
}

// Trim whitespace from left and right sides of s.
function trim(s) {
    return s.replace( /^\s*/, "" ).replace( /\s*$/, "" );
}

]]>
</javascript>
</validator>
</global>
</form-validation>

```

Each validation routine in the **validator-rules.xml** file has its own definition that is declared with a **validator** tag. The **validator** tag is used to assign a logical name to the routine, with the **name** attribute, and to specify the class and method for the routine. The logical name given to the routine will be used to refer to the routine by other routines in this file as well as by validation definitions in the **validation.xml** file.

Notice that the **validator** tag encapsulates a **javascript** tag. The **javascript** tag is used to define client-side JavaScript code for performing the same validation on the client side as is performed on the server side.

Included Validations

By default, Validator comes packaged with several basic validation routines that you can use to solve most validation scenarios. As mentioned, the sample applications provided by Struts come packaged with preconfigured **validator-rules.xml** files that define these routines. [Table 6-1](#) lists each of the preconfigured validations by logical name and states its purpose.

Table 6-1: Validator's Preconfigured Validations

| Validation | Description |
|------------|---|
| byte | Validates that the specified field contains a valid byte. |
| creditCard | Validates that the specified field contains a valid credit card number. |
| date | Validates that the specified field contains a valid date. |
| double | Validates that the specified field contains a valid double. |
| email | Validates that the specified field contains a valid e-mail address. |
| float | Validates that the specified field contains a valid float. |
| floatRange | Validates that the specified field contains a valid float and falls within the specified range. |
| integer | Validates that the specified field contains a valid integer. |
| intRange | Validates that the specified field contains a valid integer and falls within the specified range. |
| long | Validates that the specified field contains a valid long. |

Table 6-1: Validator's Preconfigured Validations

| Validation | Description |
|------------|---|
| mask | Validates that the specified field conforms to a given regular expression (or 'mask'). |
| maxlength | Validates that the string in the specified field's length is less than or equal to the specified maximum length. |
| minlength | Validates that the string in the specified field's length is greater than or equal to the specified minimum length. |
| range | Deprecated Use the intRange validation instead. |
| required | Validates that the specified field contains characters other than white space (i.e., space, tab, and newline characters). |
| requiredif | Performs the required validation on the specified field if a specified rule is met. |
| short | Validates that the specified field contains a valid short. |

Note Detailed information on configuring the **validator-rules.xml** file is found in [Chapter 18](#).

Creating Form Beans

In order to use Validator, your application's Form Beans have to subclass one of Validator's **ActionForm** subclasses instead of **ActionForm** itself. Validator's **ActionForm** subclasses provide an implementation for **ActionForm**'s **validate()** method that hooks into the Validator framework. Instead of hard-coding validations into the **validate()** method, as you would normally do, you simply omit the method altogether because Validator provides the validation code for you.

Parallel to the core functionality provided by Struts, Validator gives you two paths to choose from when creating Form Beans. The first path you can choose is to create a concrete Form Bean object like the one shown here:

```
package com.jamesholmes.minihr;

import org.apache.struts.validator.ValidatorForm;

public class LogonForm extends ValidatorForm {
    private String username;
    private String password;

    public String getUsername() {
        return username;
    }

    public void setUsername(String username) {
        this.username = username;
    }

    public String getPassword() {
        return password;
    }

    public void setPassword(String password) {
        this.password = password;
    }
}
```

This class is similar to one that you would create if you were not using Validator; however, this class extends **ValidatorForm** instead of **ActionForm**. This class also does not provide an implementation for **ActionForm**'s empty **reset()** and **validate()** methods, because **ValidatorForm** does.

You configure this Form Bean in the [struts-config.xml](#) file the same way you would a regular Form Bean, as shown here:

```
<form-beans>
  <form-bean name="logonForm"
            type="com.jamesholmes.minihr.LogonForm"/>
</form-beans>
```

The logical name given to the Form Bean with the **form-bean** tag's **name** attribute is the name that you will use when defining validations in the **validation.xml** file, as shown here:

```
<!DOCTYPE form-validation PUBLIC
"-//Apache Software Foundation//DTD Commons
Validator Rules Configuration 1.0//EN"
"http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <formset>
    <form name="logonForm">
      <field property="username" depends="required">
        <arg0 key="prompt.username"/>
      </field>
    </form>
  </formset>
</form-validation>
```

Validator uses the value of the **form** tag's **name** attribute to match validation definitions to the name of the Form Bean to which they are applied.

The second path you can choose when creating your Form Bean is to define a Dynamic Form Bean in the [struts-config.xml](#) file, as shown here:

```
<form-beans>
  <form-bean name="logonForm"
            type="org.apache.struts.validator.DynaValidatorForm">
    <form-property name="username" type="java.lang.String"/>
    <form-property name="password" type="java.lang.String"/>
  </form-bean>
</form-beans>
```

Dynamic Form Beans do not require you to create concrete Form Bean objects; instead, you define the properties that your Form Bean should have and their types, and Struts will dynamically create the Form Bean for you. Validator allows you to use this concept just as you would with core Struts. The only difference for Validator is that you specify that your Form Bean is of type **org.apache.struts.validator.DynaValidatorForm** instead of **org.apache.struts.action.DynaActionForm**.

Identical to the way concrete Form Beans work with Validator, the logical name given to Dynamic Form Beans is the name that you will use when defining validations in the **validation.xml** file. Validator uses the matching names to tie the validations to the Form Bean.

In addition to the two standard options for creating Form Beans, Validator provides an advanced feature for tying multiple validation definitions to one Form Bean definition. When using **ValidatorForm**- or **DynaValidatorForm**-based Form Beans, Validator uses the logical name for the Form Bean from the [struts-config.xml](#) file to map the Form Bean to validation definitions in the **validation.xml** file. This mechanism is ideal for most cases; however, there are scenarios where Form Beans are shared among multiple actions. One action may use all the Form Bean's fields and another action may use only a subset of

the fields. Because validation definitions are tied to the Form Bean, the action that uses only a subset of the fields has no way of bypassing validations for the unused fields. When the Form Bean is validated, it will generate error messages for the unused fields because Validator has no way of knowing not to validate the unused fields; it simply sees them as missing or invalid.

To solve this problem, Validator provides two additional **ActionForm** subclasses that allow you to tie validations to actions instead of Form Beans. That way you can specify which validations to apply to the Form Bean based on which action is using the Form Bean. For concrete Form Beans, you subclass **org.apache.struts.validator.ValidatorActionForm**, as shown here:

```
public class AddressForm extends ValidatorActionForm {
    ...
}
```

For Dynamic Form Beans, you specify a type of **org.apache.struts.validator.DynaValidatorActionForm** for your Form Bean definition in the [struts-config.xml](#) file:

```
<form-bean name="addressForm"
           type="org.apache.struts.validator.DynaValidatorActionForm">
    ...
</form-bean>
```

Inside your **validation.xml** file, you map a set of validations to an action path instead of a Form Bean name, because if you have two actions defined, Create Address and Edit Address, which use the same Form Bean, as shown here, each has a unique action path:

```
<action-mappings>
  <action path="/createAddress"
          type="com.jamesholmes.minihr.CreateAddressAction"
          name="addressForm"/>
  <action path="/editAddress"
          type="com.jamesholmes.minihr.EditAddressAction"
          name="addressForm"/>
</action-mappings>
```

The following **validation.xml** file snippet shows two sets of validations that are intended for the same Form Bean, but are distinguished by different action paths:

```
<formset>
  <form name="/createAddress">
    <field property="city" depends="required">
      <arg0 key="prompt.city"/>
    </field>
  </form>
  <form name="/editAddress">
    <field property="state" depends="required">
      <arg0 key="prompt.state"/>
    </field>
  </form>
</formset>
```

Because your Form Bean subclasses either **ValidatorActionForm** or **DynaValidatorActionForm**, Validator knows to use an action path to find validations instead of the Form Bean's logical name.

Configuring validation.xml

The **validation.xml** file is used to declare sets of validations that should be applied to Form Beans. Each Form Bean that you want to validate has its own definition in this file. Inside that definition you specify the validations that you want to apply to the Form Bean's fields. Following is a sample **validation.xml** file that illustrates how validations are defined:

```

<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons
    Validator Rules Configuration 1.0//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <formset>
    <form name="logonForm">
      <field property="username" depends="required">
        <arg0 key="prompt.username"/>
      </field>
      <field property="password" depends="required">
        <arg0 key="prompt.password"/>
      </field>
    </form>
  </formset>
</form-validation>

```

The first element in the **validation.xml** file is the **<form-validation>** element. This element is the master element for the file and is defined only once. Inside the **<form-validation>** element you define **<form-set>** elements that encapsulate multiple **<form>** elements. Generally, you will define only one **<form-set>** element in your file; however, you would use a separate one for each locale if you were internationalizing validations.

Each **<form>** element uses the **name** attribute to associate a name to the set of field validations it encompasses. Validator uses this logical name to map the validations to a Form Bean defined in the [struts-config.xml](#) file. Based on the type of Form Bean being validated, Validator will attempt to match the name either against a Form Bean's logical name or against an action's path. Inside the **<form>** element, **<field>** elements are used to define the validations that will be applied to specified Form Bean fields. The **<field>** element's **property** attribute corresponds to the name of a field in the specified Form Bean. The **depends** attribute specifies the logical names of validation routines from the **validator-rules.xml** file that should be applied to the field. The validations specified with the **depends** attribute will be performed in the order specified and they all must pass.

Note Detailed information on configuring the **validation.xml** file is found in [Chapter 18](#).

Configuring ApplicationResources.properties

Validator uses Struts' Resource Bundle mechanism for externalizing error messages. Instead of having hard-coded error messages in the framework, Validator allows you to specify a key to a message in the [ApplicationResources.properties](#) file that is returned if a validation fails. Each validation routine in the **validator-rules.xml** file specifies an error message key with the **validator** tag's **msg** attribute, as shown here:

```

<validator name="required"
  classname="org.apache.struts.validator.FieldChecks"
  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">

```

If the validation fails when it is run, the message corresponding to the key specified by the **msg** attribute will be returned.

The following snippet shows the default set of validation error messages from the [ApplicationResources.properties](#) file that comes prepackaged with Struts' example applications. Each

message key corresponds to those specified by the validation routines in the **validator-rules.xml** file that also comes prepackaged with Struts' example applications:

```
# Error messages for Validator framework validations
errors.required={0} is required.
errors.minlength={0} cannot be less than {1} characters.
errors.maxlength={0} cannot be greater than {2} 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 e-mail address.
```

Notice that each message has placeholders in the form of {0}, {1}, or {2}. At run time, the placeholders will be substituted for another value such as the name of the field being validated. This feature is especially useful in allowing you to create generic validation error messages that can be reused for several different fields of the same type.

Take for example the **required** validation's error message, **errors.required**:

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

When you use the **required** validation in the **validation.xml** file, you have to define the value that should be used to substitute {0} in the error message:

```
<form name="auctionForm">
  <field property="bid" depends="required">
    <arg0 key="prompt.bid"/>
  </field>
</form>
```

Error messages can have up to four placeholders: {0} - {3}. These placeholders are known as *arg0* - *arg3*, respectively, and can be specified using the **arg0** - **arg3** tags. In the preceding example, the **arg0** tag specifies the value that should be used to replace the {0} placeholder. This tag's **key** attribute specifies a message key from the [ApplicationResources.properties](#) file, such as the one shown next, whose value will be used as the replacement for the placeholder:

```
prompt.bid=Auction Bid
```

Using a message key for the placeholder value frees you from having to hard-code the replacement value over and over in the **validation.xml** file. However, if you don't want to use the Resource Bundle key/value mechanism to specify placeholder values, you can explicitly specify the placeholder value by using the following syntax for the **arg0** tag:

```
<arg0 key="Auction Bid" resource="false"/>
```

In this example, the **resource** attribute is set to 'false', telling Validator that the value specified with the **key** attribute should be taken as the literal placeholder value and not as a key for a message in the [ApplicationResources.properties](#) file.

Note Detailed information on the [ApplicationResources.properties](#) file is found in [Chapter 10](#).

Enabling Client-Side Validations

In addition to providing a framework for simplifying server-side form data validations, Validator provides an easy-to-use mechanism for performing client-side validations. Each validation routine defined in the **validator-rules.xml** file optionally specifies JavaScript code that can be run in the browser (client side) to perform the same validations that take place on the server side. When run on the client side, the validations will not allow the form to be submitted until they have all passed.

To enable client-side validation, you have to place the HTML Tag Library's **javascript** tag in each JSP for which you want validation performed, as shown here:

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

The **javascript** tag requires that you use the **formName** attribute to specify the name of a **<form>** definition from the **validation.xml** file, as shown here, for which you want validations performed:

```
<form name="logonForm">
  <field property="username" depends="required">
    <arg0 key="prompt.username"/>
  </field>
  <field property="password" depends="required">
    <arg0 key="prompt.password"/>
  </field>
</form>
```

All the validations that you have specified for the **<form>** definition to run on the server side will be run on the client side.

Creating Custom Validations

Validator comes packaged with several useful validations that will suit most validation scenarios; however, your application may require a specific validation that is not provided by the framework. For this situation, Validator provides a simple interface for creating your own custom validations that can easily be plugged into the framework. Following is a list of the steps you need to take to create your own custom validation:

1. Create a new validation method.
2. Add a new validation rule to the **validator-rules.xml** file.
3. Add new validation definitions to the **validation.xml** file.
4. Add messages to the [ApplicationResources.properties](#) file.

The following sections walk through each step of the process in detail, enabling you to create a custom validation based on the Social Security Number validation used in the example Mini HR application in [Chapter 2](#). Remember that the Social Security Number validation in [Chapter 2](#) was defined inside of the **SearchForm** Form Bean class. Creating a custom validation for social security numbers enables the validation code to be reused and to be used declaratively instead of being hard-coded in each Form Bean that wants to use it.

Creating a Validation Method

The first step in creating a custom validation is to create a validation method that can be called by the Validator framework. Typically, all of an application's methods for custom validations are grouped into a class of their own, as is the case for the example in this section. However, you can place the method in any class. Your validation method needs to have the following signature:

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

Of course, the name of your method will vary, but its arguments should match the types shown in the preceding example. [Table 6-2](#) explains each of the validation method’s arguments.

Table 6-2: The Validation Method Arguments

| Argument | Description |
|--|---|
| java.lang.Object | The Form Bean object (down casted to Object) contains the field to be validated. |
| org.apache.commons.validator.ValidatorAction | The ValidatorAction object encapsulates the <validator> definition from the validator-rules.xml file for this validation routine. |
| org.apache.commons.validator.Field | The Field object encapsulates the <field> definition from the validation.xml file for the field that is currently being validated. |
| org.apache.struts.action.ActionErrors | The ActionErrors object stores validation error messages for the field that is currently being validated. |
| javax.servlet.http.HttpServletRequest | The HttpServletRequest object encapsulates the current HTTP request. |

Following is the custom validation code for validating social security numbers. Notice that the **validateSsNum()** method conforms to the proper method signature for custom validations.

```
package com.jamesholmes.minihr;

import javax.servlet.http.HttpServletRequest;
import org.apache.commons.validator.Field;
import org.apache.commons.validator.ValidatorAction;
import org.apache.commons.validator.ValidatorUtil;
import org.apache.struts.action.ActionErrors;
import org.apache.struts.validator.Resources;

public class MiniHrValidator
{
    public static boolean validateSsNum(Object bean,
        ValidatorAction action,
        Field field,
        ActionErrors errors,
        HttpServletRequest request)
    {
        String value =
            ValidatorUtil.getValueAsString(bean, field.getProperty());

        if (value == null || value.length() < 11) {
            errors.add(field.getKey(),
                Resources.getActionError(request, action, field));
            return false;
        }

        for (int i = 0; i < 11; i++) {
            if (i == 3 || i == 6) {
                if (value.charAt(i) != '-') {
                    errors.add(field.getKey(),
                        Resources.getActionError(request, action, field));
                    return false;
                }
            }
        }
    }
}
```

```

    }
    } else if ("0123456789".indexOf(value.charAt(i)) == -1) {
        errors.add(field.getKey(),
            Resources.getActionError(request, action, field));
        return false;
    }
}
}

return true;
}
}

```

The **validateSsNum()** method begins by retrieving the value for the field being validated. The value is retrieved by determining the field's name with a call to **field.getProperty()** and then looking up that field in the Form Bean with a call to **ValidatorUtil.getValueAsString()**. The **getValueAsString()** method matches the name of the field with the name of one of the Form Bean fields and then gets that field's value. The rest of the **validateSsNum()** method performs the actual validation logic. If the validation fails for any reason, an error message will be stored in the **errors ActionErrors** object.

Adding a New Validation Rule

After the custom validation code has been created, a new validation rule needs to be added to the **validator-rules.xml** file. As discussed earlier in this chapter, validation rules “plug” validation methods into the Validator. Once defined in **validator-rules.xml**, as shown here, the validation rule can be referenced in the **validation.xml** file:

```

<validator name="ssNum"
    classname="com.jamesholmes.minihr.MiniHrValidator"
    method="validateSsNum"
    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.ssNum">
<javascript>
<![CDATA[
function validateSsNum(form) {
    var bValid = true;
    var focusField = null;
    var i = 0;
    var fields = new Array();
    oSsNum = new ssNum();

    for (x in oSsNum) {
        if ((form[oSsNum[x][0]].type == 'text' ||
            form[oSsNum[x][0]].type == 'textarea') &&
            (form[oSsNum[x][0]].value.length > 0))
        {
            var value = form[oSsNum[x][0]].value;
            var bRightFormat = true;

            if (value.length != 11) {
                bRightFormat = false;
            }

            for (var n = 0; n < 11; n++) {
                if (n == 3 || n == 6) {
                    if (value.substring(n, n+1) != '-') {
                        bRightFormat = false;
                    }
                }
                } else if ("0123456789".indexOf(
                    value.substring(n, n+1)) == -1) {
                    bRightFormat = false;
                }
            }
        }
    }
}

```

```

    }
  }

  if (!bRightFormat) {
    if (i == 0) {
      focusField = form[oSsNum[x][0]];
    }
    fields[i++] = oSsNum[x][1];
    bValid = false;
  }
}
}
if (fields.length > 0) {
  focusField.focus();
  alert(fields.join('\n'));
}
return bValid;
}
]]>
</javascript>
</validator>

```

As you can see, the validation rule applies a name to the validation with the **name** attribute; specifies the class that the validation method is housed in with the **classname** attribute; and specifies the validation method's arguments with the **methodParams** attribute. The **msg** attribute specifies a key that will be used to look up an error message in the [ApplicationResources.properties](#) file if the validation fails. Note that the name applied with the **name** attribute is the logical name for the rule and will be used to apply the rule to definitions in the **validation.xml** file.

The preceding custom validation rule also defines JavaScript code, inside the opening and closing **<javascript>** elements, which will be used if client-side validation is enabled when using the rule. The JavaScript code simply performs the same validation on the client side as is performed on the server side. If the JavaScript validation fails, it will alert the user and prevent the HTML form from being submitted.

Note that validation rules must be placed underneath the **<global>** element in the **validator-rules.xml** file, as shown here:

```

<form-validation>
  <global>
    <validator name="ssnum" .../>
    ...
  </global>
</form-validation>

```

Of course, the order of the **<validator>** elements underneath the **<global>** element is arbitrary, so you can place the new "ssnum" validation rule anywhere.

Adding New Validation Definitions

Once you have defined your custom validation rule in the **validator-rules.xml** file, you can make use of it by referencing its logical name in the **validation.xml** file:

```

<!DOCTYPE form-validation PUBLIC
  "-//Apache Software Foundation//DTD Commons
  Validator Rules Configuration 1.0//EN"
  "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <formset>
    <form name="searchForm">
      <field property="ssNum" depends="required, ssNum">

```

```
<arg0 key="prompt.ssNum"/>
</field>
</form>
</formset>
</form-validation>
```

In the preceding validation definition, each of the comma-delimited values of the **field** tag's **depends** attribute corresponds to the logical name of a validation rule defined in the **validation-rules.xml** file. The use of "ssNum" instructs Validator to use the custom Social Security Number validation. Each time you want to use the new Social Security Number validation, you simply have to add its logical name to the **depends** attribute of a **field** tag.

[Adding Messages to the ApplicationResources.properties File](#)

The final step in creating a custom validation is to add messages to the [ApplicationResources.properties](#) file:

```
prompt.ssNum=Social Security Number
errors.ssNum={0} is not a valid Social Security Number
```

Remember that the **errors.ssNum** message key was specified by the **msg** attribute of the **validator** tag for the custom validation rule in the **validator-rules.xml** file. The key's corresponding message will be used if the validation fails. The **prompt.ssNum** message key was specified by the **arg0** tag of the validation definition in the **validation.xml** file. Its corresponding message will be used as the parametric replacement for the **errors.ssNum** message's {0} parameter. Thus, if the Social Security Number custom validation fails, the following error message will be generated by substituting the **prompt.ssNum** message for {0} in the **errors.ssNum** message:

```
Social Security Number is not a valid Social Security Number
```

[Internationalizing Validations](#)

Similar to other areas of Struts, Validator fully supports internationalization. Remember that internationalization is the process of tailoring content to a specific locale or region. In Validator's case, internationalization means tailoring validation error messages to a specific locale and/or tailoring actual validation routines to a specific locale. This way, the U.S. and French versions of a Web site can each have their own language-specific validation error messages. Similarly, internationalization enables the U.S. and French versions of a Web site to validate entries in monetary fields differently. The U.S. version requires commas to separate dollar values and a period to demarcate cents (i.e., 123,456.78), whereas the French (Euro monetary system) version requires periods to separate dollar amounts and a comma to demarcate cents (i.e., 123.456,78).

Tailoring validation error messages to a specific locale is built into Struts by way of its Resource Bundle mechanism for externalizing application strings, messages, and labels. You simply create an [ApplicationResources.properties](#) file for each locale you want to support. Each locale-specific properties file will have a locale identifier at the end of the filename that denotes which locale it is for, such as **ApplicationResources_ja.properties** for Japan. (For detailed information on internationalizing a Struts application, see [Chapter 10](#).)

Thus, when Validator goes to load an error message, it will use the locale for the current request to determine which properties file to load the message from. Remember that each validation rule in the **validator-rules.xml** file specifies a key for a validation error message stored in the [ApplicationResources.properties](#) files. This key is the same across each locale's properties file, thus allowing Validator to load the appropriate message based on only a locale and a key.

Tailoring validation routines to specific locales is similar to tailoring error messages; you have to define validation definitions in the **validation.xml** file for each locale. The **validation.xml** file contains a **form-validation** tag that contains one or more **formset** tags, which in turn contain one or more **form** tags, and so on:

```
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons
    Validator Rules Configuration 1.0//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <formset>
    <form name="auctionForm">
      <field property="bid" depends="mask">
        <var>
          <var-name>mask</var-name>
          <var-value>^\d{1,3}(,?\d{3})*\.\?(\d{1,2})?</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

The **form-set** tag takes optional attributes, **country** and **language**, to tailor its nested forms to a specific locale. In the preceding example, all locales use the same currency validation to validate the “bid” property, because neither the **country** attribute nor the **language** attribute was specified for the **form-set** tag.

In order to have a generic currency validation for all users of the Web site and a Euro-specific currency validation for users from France, you’d create an additional **form-set** definition specifically for the French users, as shown here:

```
<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons
    Validator Rules Configuration 1.0//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <formset>
    <form name="auctionForm">
      <field property="bid" depends="required,mask">
        <var>
          <var-name>mask</var-name>
          <var-value>^\d{1,3}(,?\d{3})*\.\?(\d{1,2})?</var-value>
        </var>
      </field>
    </form>
  </formset>
  <formset country="fr">
    <form name="auctionForm">
      <field property="bid" depends="required,mask">
        <var>
          <var-name>mask</var-name>
          <var-value>^\d{1,3}(\.\?|\,?\d{3})*\.\?(\d{1,2})?</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>
```

In this listing, the second **<formset>** definition specifies a **country** attribute set to “fr”. That instructs Validator to use the enclosed validation definitions for users with a French locale. Notice that the bid validation in the second **<formset>** definition is different from the first definition, as the period and comma

are transposed in the mask value. Thus, the first **<formset>** definition validates that bids are in U.S. currency format and the second definition validates that bids are in Euro currency format.

A powerful feature of using internationalized **<formset>** definitions is that you can define only the validations that are locale-specific, and all other validations are taken from the default **<formset>** definition.

[Adding Validator to the Mini HR Application](#)

Now that you've seen the benefits of using the Validator framework and how it works, you are ready to revisit the Mini HR application and replace the hard-coded validation logic with Validator. Following is the list of steps involved in adding the Validator to the Mini HR application:

1. Change **SearchForm** to extend **ValidatorForm**.
2. Add a **validator-rules.xml** file.
3. Create a **validation.xml** file.
4. Add the Validator plugin to the [struts-config.xml](#) file.
5. Add Validation error messages to the [ApplicationResources.properties](#) file.
6. Recompile, repackage, and run the updated application.

The following sections walk through each step of the process in detail.

[Change SearchForm to Extend ValidatorForm](#)

The first step in converting the Mini HR application to use Validator is to change **SearchForm** to extend Validator's **ValidatorForm** class instead of extending Struts' basic **ActionForm** class. Recall that **ValidatorForm** extends **ActionForm** and provides an implementation for its **reset()** and **validate()** methods that hook into the Validator framework; thus, those methods should be removed from the **SearchForm** class. Additionally, the **isValidSsNum()** method should be removed from the **SearchForm** class because its functionality is being replaced by Validator as well.

Following is the updated [SearchForm.java](#) file:

```
package com.jamesholmes.minihr;

import java.util.List;
import org.apache.struts.validator.ValidatorForm;

public class SearchForm extends ValidatorForm
{
    private String name = null;
    private String ssNum = null;
    private List results = null;

    public void setName(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void setSsNum(String ssNum) {
        this.ssNum = ssNum;
    }

    public String getSsNum() {
        return ssNum;
    }

    public void setResults(List results) {
```

```

    this.results = results;
}

public List getResults() {
    return results;
}
}

```

Notice that this file no longer has the `reset()`, `validate()`, and `validateSsNum()` methods and that the class been updated to extend **ValidatorForm**.

Add a validator-rules.xml File

As mentioned earlier in this chapter, the sample applications that come packaged in Struts distributions contain a **validator-rules.xml** file underneath their **WEB-INF** directories that are preconfigured for all of Validator's basic validations. You will need to copy one of the preconfigured copies of the file to Mini HR's **WEB-INF** directory (e.g., `c:\java\MiniHR\WEB-INF`). Because this file is so large, its contents will not be shown here.

Create a validation.xml File

After removing the hard-coded validation logic from **SearchForm** and adding a **validator-rules.xml** file, you must create a **validation.xml** file. This file will inform Validator which validations from the **validator-rules.xml** file should be applied to **SearchForm**. Following is a basic **validation.xml** file that validates that social security numbers have the proper format if entered:

```

<!DOCTYPE form-validation PUBLIC
    "-//Apache Software Foundation//DTD Commons
    Validator Rules Configuration 1.0//EN"
    "http://jakarta.apache.org/commons/dtds/validator_1_0.dtd">

<form-validation>
  <formset>
    <form name="searchForm">
      <field property="ssNum" depends="mask">
        <arg0 key="label.search.ssNum"/>
        <var>
          <var-name>mask</var-name>
          <var-value>^\d{3}-\d{2}-\d{4}$</var-value>
        </var>
      </field>
    </form>
  </formset>
</form-validation>

```

Notice that this file does not contain any validation definitions to ensure that either a name or a social security number was entered, the way the original hard-coded logic did. This is because such logic is complicated to implement with Validator and thus should be implemented using Struts' basic validation mechanism.

Add the Validator Plugin to the struts-config.xml File

After setting up Validator's configuration files, the following snippet must be added to the Struts configuration file to cause Struts to load the Validator plugin:

```

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

```

```
</plug-in>
```

Notice that each of the configuration files is specified with the **set-property** tag. The following snippet lists the updated Struts configuration file for Mini HR in its entirety.

```
<!DOCTYPE struts-config PUBLIC
"-//Apache Software Foundation//DTD Struts Configuration 1.1//EN"
"http://jakarta.apache.org/struts/dtds/struts-config_1_1.dtd">

<struts-config>

  <!-- Form Beans Configuration -->
  <form-beans>
    <form-bean name="searchForm"
      type="com.jamesholmes.minihr.SearchForm"/>
  </form-beans>

  <!-- Global Forwards Configuration -->
  <global-forwards>
    <forward name="search" path="/search.jsp"/>
  </global-forwards>

  <!-- Action Mappings Configuration -->
  <action-mappings>
    <action path="/search"
      type="com.jamesholmes.minihr.SearchAction"
      name="searchForm"
      scope="request"
      validate="true"
      input="/search.jsp">
    </action>
  </action-mappings>

  <!-- Message Resources Configuration -->
  <message-resources
    parameter="com.jamesholmes.minihr.ApplicationResources"/>

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

</struts-config>
```

Add Validation Error Messages to the ApplicationResources.properties File

Recall from earlier in this chapter that each validation routine defined in the **validator-rules.xml** file declares a key for an error message in Struts' resource bundle file: [ApplicationResources.properties](#). At run time, Validator uses the keys to look up error messages to return when validations fail. Because you are using the "mask" validation defined in the **validator-rules.xml** file, you must add the following error message for its declared key to the [ApplicationResources.properties](#) file:

```
errors.invalid=<li>{0} is not valid</li>
```

The following code shows the updated [ApplicationResources.properties](#) file in its entirety:

```
# Label Resources
label.search.name=Name
label.search.ssNum=Social Security Number

# Error Resources
error.search.criteria.missing=<li>Search Criteria Missing</li>
```

```
error.search.ssNum.invalid=<li>Invalid Social Security Number</li>
errors.header=<font color="red"><b>Validation Error(s)</b></font><ul>
errors.footer=</ul><hr width="100%" size="1" noshade="true">
```

```
errors.invalid=<li>{0} is not valid</li>
```

Compile, Package, and Run the Updated Application

Because you removed the `reset()`, `validate()`, and `validateSsNum()` methods from **SearchForm** and changed it to extend **ValidatorForm** instead of **ActionForm**, you need to recompile and repackage the Mini HR application before you run it. Assuming that you've made modifications to the original Mini HR application and it was set up in the `c:\java\MiniHR` directory (as described in [Chapter 2](#)), the following command line will recompile the application:

```
javac -classpath WEB-INF\lib\commons-beanutils.jar;
        WEB-INF\lib\commons-collections.jar;
        WEB-INF\lib\commons-lang.jar;
        WEB-INF\lib\commons-logging.jar;
        WEB-INF\lib\commons-validator.jar;
        WEB-INF\lib\digester.jar;
        WEB-INF\lib\fileupload.jar;
        WEB-INF\lib\jakarta-oro.jar;
        WEB-INF\lib\struts.jar;
        C:\java\jakarta-tomcat-4.1.27\common\lib\servlet.jar
        WEB-INF\src\com\jamesholmes\minihr\*.java
        -d WEB-INF\classes
```

After recompiling Mini HR, you need to repackage it using the following command line:

```
jar cf MiniHR.war *
```

This command should also be run from the directory where you have set up the Mini HR application (e.g., `c:\java\MiniHR`).

Similar to the way you ran Mini HR the first time, you now need to place the new **MiniHR.war** file that you just created into Tomcat's **webapps** directory and start Tomcat. As before, to access the Mini HR application, point your browser to <http://localhost:8080/MiniHR/>. Once you have the updated Mini HR running, try entering valid and invalid social security numbers. As you will see, they are now verified using the new Validator code.