STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

MANNING

Table of Contents

10 Exploring the validation framework

This chapter covers

- Introducing the validation framework
- Wiring your actions for validation
- Building a custom validator
- Adapting the validation framework to your needs

Building on the refinements we saw in the previous chapter, where we added Hibernate-based persistence and Spring resource management to our sample application, this chapter introduces another advanced mechanism of the Struts 2 Framework, the validation framework. We've had robust data validation in our Struts 2 Portfolio since chapter 3, where we learned how to implement an action-local form of validation with the `Validateable` interface's `validate()` method. While this method works fine, it has some limitations that eventually become burdensome. We revisit the details of this basic form of validation in the first section of this chapter, and then quickly move on to explore the higher-level validation framework that comes with Struts 2.

The validation framework provides a more versatile and maintainable solution to validation than the `Validateable` interface. Throughout this chapter, we explore the components of the validation framework and learn how easy it is to work with this robust validation mechanism. We demonstrate this in code by migrating the Struts 2 Portfolio to use the validation framework instead of the `Validateable` interface mechanism. This example shows you how to wire up your actions for validation as well as demonstrates the use cases of the more common built-in validators. During this process, you'll learn all you need to leverage this advanced validation tool in your own projects.

One of the stronger points of the validation framework is the `Validator`, a reusable component in which the logic of specific types of validations are implemented. Some of the built-in validators handle such validations as checking whether a given string represents a valid email address or whether a date falls within a given range. We also demonstrate the extensible nature of the `Validators` by implementing a custom `Validator` for use in the Struts 2 Portfolio. We even point out some advanced nuances and techniques before we wind things up.

But let's get started by exploring the basic architecture of the validation framework. As you'll soon see, the learning curve is gentle.

10.1 Getting familiar with the validation framework

As with most aspects of Struts 2, the validation framework is well engineered. As we've indicated, Struts 2 is a second-generation web application framework. As with most of its components, validation has been a part of web application frameworks for a while, but Struts 2 takes it to a new level of refinement, modularity, and clean integration. Due to this, we can benefit greatly from a high-level study before kicking off into our code examples. We'll take the first section of this chapter to examine the architecture of the validation framework as well as how it fits into the workflow of Struts 2 itself. First, we'll look at the architecture.

10.1.1 The validation framework architecture

While it may make some people groan to learn that even validation has its own framework and architecture in Struts 2, we think a clean architecture just means it's easier to learn. Figure 10.1 shows the main components of the validation framework.

As you can see in figure 10.1, there are three main components at play in the validation framework: the *domain data*, *validation metadata*, and the *validators*. Each plays a vital role in the work of validation, which we explain in the next sections.

DOMAIN DATA

First, we must have some data to validate. We can see that the domain data depicted in figure 10.1 resides as properties on a Struts 2 action: `username`, `password`, and `age`. These properties are assumed to hold the data our action will work with when it begins execution. This is a common scenario we've become familiar with throughout the course of the book. However, we also know that such domain data could also be implemented in a couple of other ways, via a `ModelDriven` action for instance. All such

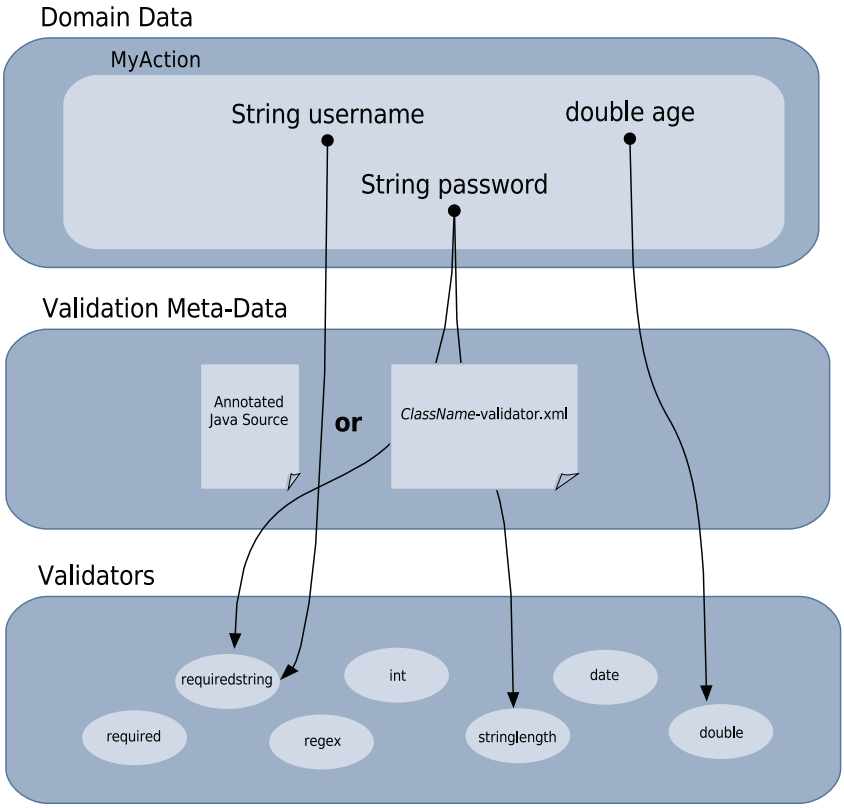

Figure 10.1 The validation framework uses metadata to associate validators with data properties.

variations on domain data handling can use this method of validation, but we'll start with this simplest case of simple JavaBeans properties on the action while we first explore the validation framework behavior. Later in the chapter, we'll demonstrate using `ModelDriven` actions.

VALIDATION METADATA

In figure 10.1, we see that a middle component lies between the validators and the data properties themselves. This middle component is the metadata that associates individual data properties with the validators that should be used to verify the correctness of the values in those properties at runtime. You can associate as many validators with each property as you like, including zero if that makes sense for your requirements.

When it comes to the details of implementation, the metadata layer offers a choice. The developer can map data properties to validators with XML files or with Java annotations. During this chapter, we'll focus our energies on the XML versions. We'll show how the annotations work at the end of the chapter. Ultimately it doesn't matter which one you use, as they're both just interfaces to the underlying validation mechanisms.

VALIDATORS

The actual work in all of this is done by the validators themselves. A validator is a reusable component that contains the logic for performing some fine-grained act of validation.

For instance, in figure 10.1 our `username` property is mapped to the `requiredstring` validator. This validator verifies that the value of the `username` property, or whatever property it validates, is a nonempty `String` value. The `password` property is mapped to `requiredstring` and `stringlength`. The `stringlength` validator checks that the string is of a desired length. The framework comes with a rich set of built-in validators, and you can even write your own. To have your data validated by these validators, you simply wire up your properties to the desired validators via some XML or Java annotations. When the validation executes, each property is validated by the set of validators with which it's been associated by the metadata layer.

But how does the validation framework actually get executed? Good question. In the next section, we examine how it fits into the Struts 2 workflow.

10.1.2 The validation framework in the Struts 2 workflow

Now let's look at how all of this validation actually gets done. As you might guess, there's an interceptor involved. Before we get into the details of that interceptor, let's take a moment to review how the basic version of validation, which we've already been using, works.

REVIEWING BASIC VALIDATION

Up until now, we've been putting our validation in the `validate()` method on our actions. We'll now provide a quick summary of how that validation works. If you want all of the details, you can go back to chapter 3 and review them, but a quick refresher should work fine.

The actions of our Struts 2 Portfolio all extend `ActionSupport`, which implements a couple of interfaces that play an important role in validation. These interfaces are `com.opensymphony.xwork2.Validateable` and `com.opensymphony.xwork2.ValidationAware`. `Validateable` exposes the `validate()` method, in which we've been stuffing our validation code, and `ValidationAware` exposes methods for storing error messages generated when validation finds invalid data. As we learned before, these interfaces work in tandem with an important interceptor known as the `workflow` interceptor.

When the `workflow` interceptor fires, it first checks to see whether the action implements `Validateable`. If it does, the `workflow` interceptor invokes the `validate()` method. If our validation code finds that some piece of data isn't valid, an error message is created and added to one of the `ValidationAware` methods that store error messages. When the `validate()` method returns, the `workflow` interceptor still has another task. It calls `ValidationAware`'s `hasErrors()` method to see if there were any problems with validation. If errors exist, the `workflow` interceptor intervenes by stopping further execution of the action by returning the input result, which returns the user back to the form that was submitted.

With that quick recap out of the way, let's see how the validation framework works.

INTRODUCING THE VALIDATION FRAMEWORK WORKFLOW

As you'll see, the validation framework actually shares quite a bit of the same functionality we've previously outlined for basic validation; it uses the `ValidationAware` interface to store errors and the `workflow` interceptor to route back to the `input` page if

necessary. In fact, the only thing that changes is the validation itself. But this is a significant change.

We start by noting that both the basic validation examples and the validation framework examples all work in the context of the `defaultStack` of interceptors that come with Struts 2. The `workflow`, as dictated by this stack of interceptors, remains constant regardless of which type of validation you choose to use. In particular, note the following sequence of interceptors from the `defaultStack`, as defined in struts-default.xml:

```
<interceptor-ref name="params"/>
<interceptor-ref name="conversionError"/>
<interceptor-ref name="validation"/>
<interceptor-ref name="workflow"/>
```

In this snippet, we've excerpted only the portion of the `defaultStack` that pertains to our current discussion. As you can see, the `params` interceptor and the `conversionError` interceptor both fire before we get to the validation-related interceptors. These two interceptors finish up the work of transferring the data from the request and converting it to the correct Java types of the target properties. If you recall from our discussion of basic validation in chapter 3, the `validation` interceptor has nothing to do with that form of validation. Recall that the `workflow` interceptor invokes the `validate()` method to conduct basic validation. Now we need to take note of the `validation` interceptor because it's the entry into the validation framework. When this interceptor fires, it conducts all the validation that's been defined via the validation metadata we mentioned in the previous section.

Figure 10.2 illustrates the workflow of the validation framework.

As we've said, all Struts 2 workflow, such as shown in figure 10.2, is ultimately determined by interceptors. This figure assumes the `defaultStack`. As you can see, the first functional unit in the pipeline is the data transfer and type conversion process. This process, conducted by the `params` and `conversionError` interceptors, moves the data from the request parameters onto the properties exposed on the `ValueStack`. In this

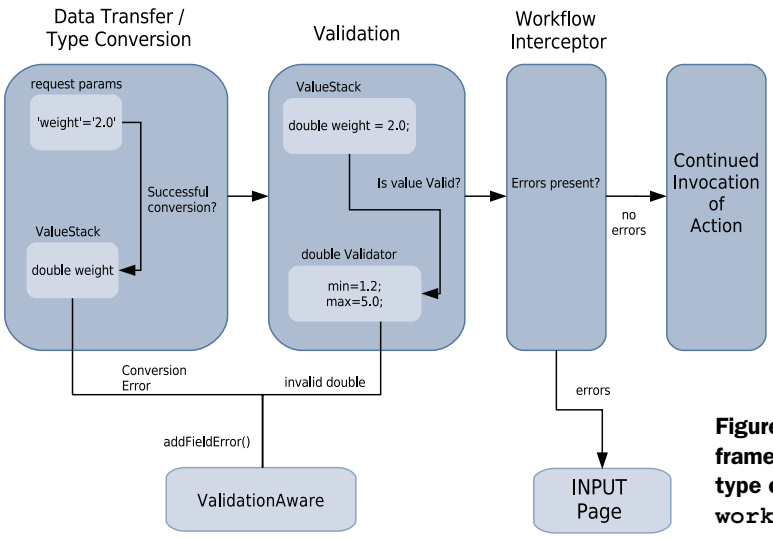

Figure 10.2 The validation framework runs after data transfer/type conversion and before the `workflow` interceptor.

case, we're moving the string value 2.0 onto the type-`double` property `weight`. In figure 10.2, this conversion is successful. If it weren't, note that an error would be added to the `ValidationAware` methods exposed on our action.

After type conversion, we proceed to the validation phase. In the figure, we're talking about the validation framework. The `validation` interceptor, which follows the `conversionError` interceptor in the `defaultStack`, provides the entry point into this validation process. Looking at figure 10.2 shows that the `weight` property, which was just populated by the `params` interceptor, is validated against a `double` validator that's been configured to verify that the `double` value falls in the range of `1.2` and `5.0`. If this weren't true, then an error would be added to the `ValidationAware` methods. Note that both conversion errors and validation errors are collected by `ValidationAware`. In our case, `2.0` passes this validation and no errors are added. Whether errors are added or not, we still proceed to the next interceptor, the `workflow` interceptor.

We've discussed the `workflow` interceptor several times in this book. We know that it has two phases. The first phase is to invoke the `validate()` method, if exposed by the current action. This is the entry point into basic validation. Let's assume, since we're using the validation framework, that we didn't implement this method. Fine. We quickly proceed to phase two of the `workflow` interceptor: checking for errors. At this point, the workflow interceptor checks the `ValidationAware` method `hasErrors()`. If there are none, it passes control on to the rest of the action invocation process. If errors are found, workflow is diverted and we return to the input page and present the user with error messages so she can correct the form data.

Before wrapping this overview up, we should make one fine but important point about the relationship between the basic validation methods and the validation framework. You can use them both at the same time. As you've seen, due to the clean lines of Struts 2, everything except the actual validation logic itself is shared between the two methods. When you use the `defaultStack`, both the `validation` and `workflow` interceptors fire every time. Ultimately, this means that you could use both forms of validation at the same time, if you like. First, the `validation` interceptor runs all validations that you define with the validation framework metadata. Then, when the `workflow` interceptor runs, it still checks to see if your action implements the `validate()` method. Even if you've already run some validators via the `validation` interceptor, you can still provide some additional validation code in a `validate()` method. When the second phase of the `workflow` interceptor checks for errors, it doesn't care or know who created them. The effect is the same.

But why use both validation mechanisms at once? One reason is common. Perhaps the strongest point of the validation framework is that the validation code is contained in reusable validators. As long as your validation needs are satisfied by the built-in validators, why write code to do that stuff? Just wire the validators to your properties and let them go.

Eventually, you'll have some validation logic that isn't handled by the built-in validators. At that point, you'll have two choices. If your validation logic is something that you can foresee reusing in the future, it probably makes sense to implement a custom

validator. However, if your validation logic truly appears to be a quirky requirement that will most likely only be applied in this one case, it makes more sense to put it in the `validate()` method. In a one-off case like this, it's much more efficient to take the quick and local fix.

That's about it. Now that you understand the architecture of the validation framework and how it fits into the interceptor-controlled Struts 2 workflow, we're ready to get back to our Struts 2 Portfolio sample application and convert it to use the validation framework.

10.2 Wiring your actions for validation

Staying true to our *in Action* name, we'll dive right in with a live example. To demonstrate the details of using the validation framework, we'll migrate our existing Struts 2 Portfolio application from the basic validation it currently uses. We'll migrate the entire thing in the sample code, which you can peruse at your convenience, but here we'll focus on the `Register` action as our case study.

The `Register` action, if you've forgotten, registers a new user of the Struts 2 Portfolio. As such, this action receives data from a registration form that collects a few pieces of information such as `username` and `password`. If you consult the older versions of this action, such as the chapter 8 version shown in listing 10.1, you can see how we'd previously implemented our validation in the `validate()` method.

Listing 10.1 An earlier version of the `Register` action that uses basic validation

```
public class Register extends ActionSupport implements SessionAware {      (1)

  public String execute(){

    //Make user and persist it.                                              (2)

    return SUCCESS;
  }

  private String username;
  private String password;
  private String portfolioName;                                              (3)
  private boolean receiveJunkMail;

  // Getters and setters omitted

  public void validate(){

    PortfolioService ps = getPortfolioService();

    if ( getPassword().length() == 0 ){                                      (4)
      addFieldError( "password", getText("password.required") );
    }
    if ( getUsername().length() == 0 ){
      addFieldError( "username", getText("username.required") );
    }

   . . .

  }
}
```

First, we note that the basic validation relies upon the `ValidationAware` implementation provided by `ActionSupport` ❶. We aren't required to extend `ActionSupport`, but, if we didn't, we'd need to implement `ValidationAware` ourselves to provide a place to store errors. Usually, we just extend `ActionSupport` to use the built-in implementation it provides. We do this regardless of which validation method we use. The `execute()` method of the `Register` action just creates the user object with the submitted data and persists that object ❷; we don't need to rehash that at this point, but you can look at the source code of the sample application if you like. Our main interest, at this time, is the use of the `validate()` method ❹ to programmatically validate the data contained in the JavaBeans properties ❸ this action exposes to hold the request data. Glancing at this code, you'll see that it programmatically tests the values in the JavaBeans properties. If it finds some validation problems, it programmatically sets a field error using the `ValidationAware` method implemented by `ActionSupport`.

10.2.1 Declaring your validation metadata with ActionClass-validations.xml

Now let's rewrite the `Register` action to use the validation framework instead of the `validate()` method. As we saw in the overview sections of this chapter, the `defaultStack` has everything in place to handle both kinds of validation. To make the switch, we just need to change the way the validation logic is invoked. We replace the programmatic validations of the `validate()` method with some metadata that creates associations between our data properties and the validators that contain the desired logic. For now, we do this with an XML file. Listing 10.2 shows the complete listing of our `Register` action's validation metadata file, Register-validation.xml.

Listing 10.2 Declares the validators that validate each exposed property

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator 1.0.2//
      EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">      ❶
<validators>                                                                  ❷
  <field name="password">                                                     ❸
     <field-validator type="requiredstring">
       <message>You must enter a value for password.</message>                ❺
     </field-validator>                                                       ❹
  </field>
  <field name="username">
    <field-validator type="stringlength">
      <param name="maxLength">8</param>
      <param name="minLength">5</param>
      <message>While ${username} is a nice name, a valid username must
            be between ${minLength} and ${maxLength} characters long.
      </message>
    </field-validator>
  </field>
  <field name="portfolioName">
    <field-validator type="requiredstring">
      <message key="portfolioName.required"/>
    </field-validator>
  </field>
```

```
    <field name="email">
      <field-validator type="requiredstring">
         <message>You must enter a value for email.</message>
      </field-validator>
      <field-validator type="email">
        <message key="email.invalid"/>
      </field-validator>
    </field>
    <validator type="expression">
      <param name="expression">username != password</param>
      <message>Username and password can't be the same.</message>
    </validator>
  </validators>
```

Listing 10.2 shows the Register-validation.xml file from our chapter 10 version of the Struts 2 Portfolio. The name of this file is derived from the name of the class that implements the action for which the validation rules apply. In this case, we're validating our `Register` action. The naming convention of the XML validation metadata file is *ActionClass*-validations.xml. This file is then placed in the package directory structure next to the action class itself. If you look back to figure 10.1, which shows the architecture of the validation framework, this XML file is the metadata component. This metadata associates sets of validators with each of the pieces of data you'd like to validate.

At the top of our Register-validations.xml file, we have a `doctype` element ❶ that you must include in all of your validation xml files. Next, we have a `validators` element ❷ that contains all of the declarations of individual validators that should be run when this action is invoked. There are two types of validators you can declare: field and nonfield.

FIELD VALIDATORS

Field validators are validators that operate on an individual field. By *field* we mean the same thing as when we say *data property*. The validators use the word field in the sense that they're coming from fields on the HTML form that submitted the request. This makes sense because, until the validator approves, the data hasn't been formally accepted into the Java side of things.

The first field declared in listing 10.2 is the `password` field ❸. Once we declare a `field` element for our data, we just need to put `field-validator` elements inside that `field` element to declare which validators should validate this piece of data. In the case of the `password`, we declare only one validator, the `requiredstring` validator ❹. This validator verifies that the string has been submitted and isn't an empty string. If the `password` string doesn't pass this verification, then a message is displayed to the user when the `workflow` interceptor sends him back to the input form. The `message` element ❺ contains the text of this message. To see how this works, go to the chapter 10 version of the sample application, navigate to the user homepage, and register for an account. If you omit the password, you'll receive this message.

A `field` element isn't limited to declaring just one the validator. It can declare as many as it likes. As an example, our `email` field ❻ in listing 10.2 declares both the `requiredstring` validator and the `email` validator.

NONFIELD VALIDATORS

You can also declare validators that don't apply logic targeted at a specific field. These validators apply to the whole action and often contain checks that involve more than one of the field values. The built-in validators only offer one instance of a nonfield validator: the `expression` validator. This useful validator allows you to embed an OGNL expression that contains the logic of the validation you wish to perform. As you've seen earlier, OGNL provides a rich expression language. You can easily write sophisticated validation logic into the expression validator.

Listing 10.2 declares a single nonfield validator ❼. This validator uses OGNL to compare two of the other fields. It's important to understand that this OGNL, like all OGNL, resolves against the `ValueStack`. Since we're in the middle of processing our action, that action and the properties it exposes are on the `ValueStack`. Thus, we can easily write a concise OGNL expression that says that the `username` and the `password` shouldn't be equal. If they're not equal, then our expression returns `true` and validation passes. If they're the same, validation will fail and the user will be returned to the input page, where he'll see the message specified in this validator's `message` element.

MESSAGE ELEMENT OPTIONS

The `message` element is used to specify the message that the user should see in the event of a validation error. In the simplest form, as seen in listing 10.2, we simply embed the message text itself in the `message` element ❺. However, several more options present themselves. First, we can use OGNL to make the message dynamic. An example of this can be seen in Register-validation.xml's declaration of the `username` field. The following snippet shows the code:

```
<field name="username">
   <field-validator type="stringlength">
     <param name="maxLength">8</param>
     <param name="minLength">5</param>
     <message>While ${username} is a nice name, a valid username
           must be between ${minLength} and ${maxLength}
           characters long.
      </message>
  </field-validator>
 </field>
```

First of all, those `param` elements are as simple as they seem. Many of the validators, such as the `stringlength`, take parameters that configure their behavior. In this case, the `maxLength` and `minLength` parameters specify the length requirements that are imposed on the `username` string when this `stringlength` validator runs. When we list all the built-in validators shortly, we'll show all the parameters that each supports.

In the `message` element in the snippet, we see three embedded OGNL expressions. These resolve at runtime against the `ValueStack`. In this case, the `username` is pulled from the stack to customize the message that the user sees when he returns to the input page. Next, we pull the `minLength` and `maxLength` values themselves from the `ValueStack`. As it turns out, the `Validator` itself has been placed on the `ValueStack`, thus its properties are also exposed. So, you can access pretty much any data you'd

want to inject into a message. Before we move on, we should point out that the OGNL in these XML files uses the `$` rather than the `%` sign that's normally used in OGNL.

The next thing you can do with `message` elements is externalize the message itself in a resource bundle. By default, Struts 2 works with properties file–backed resource bundles. As you might recall from chapter 3, `ActionSupport` implements the `TextProvider` interface to provide access to localized messages. As you can see if you refer back to listing 10.1, the basic validation in the `validate()` method calls the `TextProvider`'s `getText()` method to retrieve localized messages. This method takes a key and retrieves a locale-sensitive message from the properties file resources. The validation framework provides even easier access to our localized messages. The following snippet from Register-validation.xml shows how this works:

```
<field name="portfolioName">
  <field-validator type="requiredstring">
    <message key="portfolioName.required"/>
  </field-validator>
</field>
```

The `message` element in this snippet doesn't have a text body. Rather, it sets the `key` attribute to a value to be used to look up the message via the `TextProvider` implementation provided by ActionSupport. In other words, this key is used to find a locale-sensitive message in your properties files. The contents of the Register.properties file follow:

```
user.exists=This user ${username} already exists.
portfolioName.required=You must enter a name for your initial portfolio.
email.invalid=Your email address was not a valid email address.
```

If the previous `requiredstring` validator finds that the `portfolioName` doesn't hold a value, then it pulls the error message from this properties file when it adds the error. Of course, since the `ResourceBundle` is locale-sensitive, the message might come from Register_es.properties if that locale is specified in the information submitted by the browser and returned by the `LocaleProvider` interface, which `ActionSupport` also implements. Change the locale in your browser and run through the registration process a few times to see this in action. If you need a refresher course on that process, please refer back to chapter 3.

Now that we've seen how validators are declared, we'll take the time to cover all the validators that come bundled with the validation framework.

10.2.2 Surveying the built-in validators

We've referred to the built-in validators more than once. The framework comes with a rich set of validators to handle most validation needs. They mostly perform such straightforward tasks that little has to be said about them. You've seen some of them already in the previous section, when we migrated the `Register` action from basic validation to the validation framework. In this section, we give a full summary of the built-in validators. Table 10.1 lists them all.

Table 10.1 Built-in validators that come with Struts 2

Validator name	Params	Function	Type
`required`	None	Verifies that value is non-null.	field
`requiredstring`	`trim` (default = true, trims white space)	Verifies that value is non-null, and not an empty string.	field
`stringlength`	`trim` (default=true, trims prior to length check), `minLength`, `maxLength`	Verifies that the string length falls within the specified parameters. No checks are made for unspecified length params—if you give no minimum, then an empty string would pass validation.	field
`int`	`Min`, `max`	Verifies that the integer value falls between the specified minimum and maximum.	field
`double`	`minInclusive`, `maxInclusive`, `minExclusive`, `maxExclusive`	Verifies that the double value falls between the inclusively or exclusively specified parameters.	field
`date`	`Min`, `max`	Verifies that the date value falls between the specified minimum and maximum. Date should be specified as `MM/DD/YYYY`.	field
`email`	None	Verifies email address format.	field
`url`	None	Verifies URL format.	field
`fieldexpression`	`expression` (required)	Evaluates an OGNL expression against current `ValueStack`. Expression must return either `true` or `false` to determine whether validation is successful.	field
`expression`	`expression` (required)	Same as `fieldexpression`, but used at action level.	action
`visitor`	`Context`, `appendPrefix`	Defers validation of a domain object property, such as `User`, to validation declarations made local to that domain object.	field
`regex`	`expression` (required), `caseSensitive`, `trim`	Verifies that a `String` conforms to the given regular expression.	field

As you can see, the table provides a brief description of the functionality of each validator, a summary of the parameters supported by the validator, and the type: field or nonfield. The functionality of most of these is simple. We've already shown how to use parameters and OGNL in the messages. The only validator that requires further discussion is the `visitor`, which allows you to define validation metadata for each domain

model class, such as our Struts 2 Portfolio `User` object. We'll show how to do this in the Advanced Topics section at the end of this chapter.

The only other thing you need to know about the built-in validators is the location of their declarations. The validation framework is actually a part of a low-level framework, upon which Struts 2 has been built, called *Xwork*. You don't need to know much about Xwork to use Struts 2, but if you like you can visit the project home page at http://www.opensymphony.com/xwork. The only reason we mention it now is to show you where the validators of the validation framework are defined. If you look in the XWork JAR file, something like xwork-2.0.4.jar, you can find an XML file that declares all these built-in validators, located at /com/opensymphony/xwork2/validator/validators/default.xml. When we build a custom validator in the next section, we see how to properly add new validators to this declaration.

10.3 Writing a custom validator

Writing your own custom validator is little different than writing any of the other custom Struts 2 components. We've already seen how to write custom interceptors, results, and type converters. In this section, we follow a familiar path of extending a convenience class, declaring our new component with XML, then wiring it in to a working code example. For our example, we write a custom validator that checks for a certain level of password integrity. After we implement it, we add it to the Struts 2 Portfolio `Register` action to make sure that people are using strong passwords for their accounts.

10.3.1 A custom validator to check password strength

As with other custom components, a custom validator must implement a certain interface. In this case, all validators are obligated to implement the `Validator` or `FieldValidator` interface. The two interfaces, found in the `com.opensymphony.xwork2.validator` package, represent the two types of `Validators` as described earlier, field and nonfield. As you might expect, the framework also provides some convenience classes to make the task of writing custom validators all the more agreeable. Typically, you'll extend either `ValidatorSupport` or `FieldValidatorSupport`, both from the `com.opensymphony.xwork2.validator.validators` package.

In our case, we extend the `FieldValidatorSupport` class because our validator, like most, operates on a given field. We design our password validator to make three checks:

1. The password must contain a letter, uppercase or lower.
2. The password must contain a digit, 0–9.
3. The password must contain at least one of a set of "special characters."

The special characters have a default value but can be configured with a parameter, similar to the `stringlength` parameters that we used earlier. Note that we won't ask our password validator to check for password length, because that would be duplicating functionality already provided by the `stringlength` validator. If we want to also enforce a length requirement on the password, we can use both validators.

Without further delay, listing 10.3 shows the full code of our `manning.utils.PasswordIntegrityValidator`.

Listing 10.3 Verifying that a password contains the required characters

```
public class PasswordIntegrityValidator extends FieldValidatorSupport {      (1)

  static Pattern digitPattern = Pattern.compile( "[0-9]");                    (2)
  static Pattern letterPattern = Pattern.compile( "[a-zA-Z]");
  static Pattern specialCharsDefaultPattern = Pattern.compile( "!@#$");

  public void validate(Object object) throws ValidationException {            (3)

    String fieldName = getFieldName();                                        (4)
    String fieldValue = (String) getFieldValue(fieldName, object );

    fieldValue = fieldValue.trim();
    Matcher digitMatcher = digitPattern.matcher(fieldValue);
    Matcher letterMatcher = letterPattern.matcher(fieldValue);
    Matcher specialCharacterMatcher;

    if ( getSpecialCharacters() != null ){                                    (5)
      Pattern specialPattern =
              Pattern.compile("[" + getSpecialCharacters() + "]" );
      specialCharacterMatcher = specialPattern.matcher( fieldValue );
    } else{
      specialCharacterMatcher =
              specialCharsDefaultPattern.matcher( fieldValue );
    }

    if ( !digitMatcher.find() ) {                                             (6)
      addFieldError( fieldName, object );
    }else if ( !letterMatcher.find() ) {
      addFieldError( fieldName, object );
    }else if ( !specialCharacterMatcher.find() ) {
      addFieldError( fieldName, object );
    }
  }

  private String specialCharacters;                                           (7)

  //Getter and setter omitted

}
```

The first thing we need to do is extend the convenience class `FieldValidatorSupport` (1). Most custom validators do this. If we were writing a nonfield validator, such as one that performed a validation check involving more than one field, we'd extend `ValidatorSupport`. Extending these convenience classes provides implementations of several helper methods that we use in this example. As a developer, you're left to focus on the details of your own logic. This logic is placed in the `validate()` method (3), the entry method defined by the `Validator` interface and left unimplemented by the abstract support classes that you extend. Another preliminary duty of the developer is creating JavaBeans properties (7) to match all parameters that should be exposed to the user. In this case, we want to allow the user to set the

list of `specialCharacters`; a valid password must have at least one of these characters. The following snippet shows how a parameter is passed in from the XML file to this property:

```
<field-validator type="passwordintegrity">
    <param name="specialCharacters">$!@#?</param>
    <message>Your password must contain one letter, one number, and one
          of the following "${specialCharacters}".
    </message>
</field-validator>
```

Now, let's look at how this validator works. It's mostly a matter of Java regular expressions and string handling, but we want to make sure you see where the code is calling on the helper methods provided by the convenience classes. Most of these helper methods are actually defined in `ValidatorSupport`, which is extended by `FieldValidatorSupport`. First, note that we define some regular expression patterns as static members of our class ❷. There's even a default set of special characters defined here in case the user doesn't specify a set herself.

Next we retrieve the value of the field via a couple of calls to our helper methods ❹. Note that the `validate()` method receives the object that's being validated. Since we've defined our validations at the action level, via Register-validation.xml, the object passed into the `validate()` method is our action. Later we'll learn how to have validation run on a domain object itself. In the case of field validators, we usually want to get our hands on the actual field value as soon as possible.

Obtaining the password value is a two-step process. First, we call the `getFieldName()` method, then the `getFieldValue()` method. Again, these helper methods are defined by the support classes.

With the password in hand, we quickly move on to build our `Matchers` ❺ and then search the password for the required characters. If a password doesn't have a required character, we generate an error and add it to the set of stored errors ❻, again with helper methods inherited from the support classes. That's it. If errors are stored, the workflow interceptor will find them and divert the user back to the input page with the appropriate error messages.

Next, we wire this validator into our Struts 2 Portfolio application to help our users start making better passwords.

10.3.2 Using our custom validator

First things first. As we mentioned at the start of this chapter, all validators must first be declared so that they can be referenced when we write our validation metadata that maps fields to validators. As we said, the built-in validators are already declared in the XWork JAR file at /com/opensymphony/xwork2/validator/validators/default.xml. We declare our own custom validators in an application-local validators.xml file, which we put at the root of our `classpath`—directly under our src folder, which will be moved to WEB-INF/classes/ during the build. Listing 10.4 shows the complete source of this important file.

Listing 10.4 Using validators.xml to declare our custom validator

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE validators PUBLIC
      "-//OpenSymphony Group//XWork Validator Config 1.0//EN"
      "http://www.opensymphony.com/xwork/xwork-validator-config-1.0.dtd">

<validators>
  <validator name="passwordintegrity"
          class="manning.utils.PasswordIntegrityValidator"/>
</validators>
```

The brief validators.xml file, shown in listing 10.4, simply declares our `PasswordIntegrityValidator` as an available validator that can be referenced under the logical name `passwordintegrity`. To declare your own custom validators, simply copy this page and insert your own validator declarations. With this in place, we can use our new validator just as easily as the built-in validators.

As promised, we'll now use the password integrity validator to ensure that our aspiring artists create good passwords. I don't want to point any fingers, but I suspect this group of users sports an alarmingly poor average strength of password. To do this, we just need to add this validator to the validators mapped to the `password` field in our Register-validation.xml file. Listing 10.5 shows the new `password field` element from that file.

Listing 10.5 The `password` element that uses our `PasswordIntegrityValidator`

```
<field name="password">
  <field-validator type="stringlength">
    <param name="maxLength">10</param>
    <param name="minLength">6</param>
    <message>Your password should be 6-10 characters.</message>
  </field-validator>
  <field-validator type="passwordintegrity">      (1)
     <param name="specialCharacters">$!@#?</param>      (2)
     <message>Your password must contain one letter, one number, and one
           of the following "${specialCharacters}".
     </message>      (3)
  </field-validator>
</field>
```

This `field` element maps two validators to our `password` field. The first one is the `stringlength` validator, which we've already seen in action. Next, we map the `passwordintegrity` validator to the `password` field. This works just like using the built-in validators; we simply use a `field-validator` element to declare the type (1), using our logical name as specified in the validators.xml file. Next, we pass in the `specialCharacters` parameter (2) to indicate the set of required characters, one of which all good passwords must include. Remember, this is received by the JavaBeans property, of matching name, that we exposed on the `PasswordIntegrityValidator`. Finally, we specify the message that the user will see if his password doesn't pass the integrity check (3). This message pulls the special characters off of the validator, which sits on the `ValueStack` at runtime, to inform the user exactly what characters are needed.

We already have this implemented in our chapter 10 version of the Struts 2 Portfolio, so feel free to check out this action by playing around with the Create an Account page found on the User Home Page section of the application.

We've now covered the fundamentals of using the validation framework. We hope you're convinced that it's a powerful, yet user-friendly way of validating your data. The final section of this chapter addresses some advanced topics and nuances.

10.4 Validation framework advanced topics

In this section, we address some advanced topics of using the validation framework. Some of these advanced topics merely explore the nuances of the validation mechanism, but others demonstrate adapting the validation framework to some of the more specialized development patterns of Struts 2, such as `ModelDriven` actions and alias-mapped actions. (Using alias mappings to wire action mappings to alternative methods exposed on the action implementation class is demonstrated in full in chapter 15.) The validation topics covered in this section include inheritance of validators, mapping validation to domain model objects instead of actions, and short-circuiting out of validation when one validator fails.

10.4.1 Validating at the domain object level

In our earlier demonstrations of the validation framework, we defined our validation metadata in an XML file on a per-action basis. This is a convenient method for wiring into the validation framework. On the other hand, if you're exposing domain objects directly to the Struts 2 data transfer, then you might find it more convenient to declare your validation metadata on a per-domain object basis. We'll do just this when we continue our migration of the Struts 2 Portfolio application to the validation framework.

The `UpdateAccount` action allows a user to modify her account information. The form that submits to this action contains exactly the same set of data as the registration form with which we've been working. For the registration action, we received our data on a JavaBeans properties–exposed action, and we defined the validation metadata in an action-local XML file that mapped validators to each field of the incoming form. This time, we define our metadata in an XML file local to our domain object, `manning.utils.User`. The corresponding change in the `UpdateAccount` action itself, as opposed to the `Register` action we just worked with, is that the entire `User` object is exposed directly on a JavaBeans property. If you need to know more about exposing domain objects, refer back to chapter 3.

The first thing we need to do is define our metadata. Listing 10.6 shows the full source of our User-validation.xml.

Listing 10.6 Declaring the validators that validate each property of the user object

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator 1.0.2//
   EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
```

```
    <field name="password">
      <field-validator type="stringlength">
        <param name="maxLength">10</param>
        <param name="minLength">6</param>
        <message>Your password should be 6-10 characters.</message>
      </field-validator>
      <field-validator type="passwordintegrity">
        <param name="specialCharacters">$!@#?</param>
        <message>Your password must contain one letter, one number, and
              one of the following "${specialCharacters}".
        </message>
      </field-validator>
    </field>
    <field name="username">
      <field-validator type="stringlength">
        <param name="maxLength">8</param>
        <param name="minLength">5</param>
        <message>While ${username} is a nice name, a valid username must
              be between ${minLength} and ${maxLength} characters long.
        </message>
     </field-validator>
    </field>
    <field name="portfolioName">
      <field-validator type="requiredstring">
        <message key="portfolioName.required"/>
     </field-validator>
    </field>
    <field name="email">
      <field-validator type="requiredstring">
         <message>You must enter a value for email.</message>
      </field-validator>
      <field-validator type="email">
        <message key="email.invalid"/>
      </field-validator>
    </field>
    <validator type="expression">
      <param name="expression">username != password</param>
      <message>Username and password can't be the same.</message>
    </validator>
  </validators>
```

As you can see, this file is exactly the same as the XML file we used to define our validators for the `Register` action. This is because we're validating the same `User` data. The only difference is that, in the case of the `Register` action, the data was being exposed directly on the action itself as individual properties. Now we expose our entire `User` object on an action-local property. We can now raise the validation metadata to the level of the `User` class itself, which allows us to reuse it across all actions that work with `User`. Once we've created our User-validation.xml file, we place it next to the `User` class itself at /manning/utils/.

With our `User` validations in place, we now need to make a connection between the action that uses a `User` and these validations. This is done by the `visitor` validator. But this validator doesn't go in the User-validation.xml file. It goes in an action-local validation

file. The following snippet shows the brief contents of the UpdateAccount-validation.xml file:

```
<!DOCTYPE validators PUBLIC "-//OpenSymphony Group//XWork Validator 1.0.2//
     EN" "http://www.opensymphony.com/xwork/xwork-validator-1.0.2.dtd">
<validators>
  <field name="user">
    <field-validator type="visitor">
       <message>User:  </message>
    </field-validator>
  </field>
</validators>
```

While much shorter, the UpdateAccount-validation.xml file still serves the same purpose as all ClassName-validation.xml files. But where those files usually define many fine-grained mappings of validators to fields, this one simply uses the `visitor` validator to make a wholesale deferral of validation details to the validation metadata made on the class of the specified field name. In this case, we've specified the `user` field. We know that our `UpdateAccount` action's `user` property is of type `manning.utils.User`. The `visitor` validator uses this information to locate User-validation.xml and uses the validation logic described in that file to validate all the properties on the `user`. At this point, it all works exactly the same as our earlier examples. The only item worth noting is the body of the `message` element. The content of the message body is used as a prefix that's attached to the error messages generated by the validations defined at the `User` level. Again, if you want to see this in action, check out the update account page on the chapter 10 version of the Struts 2 Portfolio.

USING THE VALIDATION FRAMEWORK WITH MODELDRIVEN ACTIONS

You can also use the preceding technique when your actions implement the `ModelDriven` interface. As we learned in chapter 3, `ModelDriven` actions expose domain objects via a `getModel()` method rather than exposing them directly on a JavaBeans property. The `ModelDriven` magic is such that you can now access the properties of your domain model object with top-level OGNL syntax. Instead of having fields such as `user.username` and `user.password`, we would now have, simply, `username` and `password`. Since this is almost exactly like the previous `visitor` example, we'll gallop through at a quick pace.

First of all, everything is the same unless we note a difference. For instance, the User-validation.xml file doesn't change. The main changes are to the actions and to the JSP pages. In the actions, we must implement the `ModelDriven` interface. If we were working with the update account page, we'd need to convert two actions, `UpdateAccountForm` and `UpdateAccount`. The first action prepopulates the form and the second one processes the update. Changing these to `ModelDriven` amounts to little more than adding the `ModelDriven` interface and implementing its one method, `getModel()`. If you need to see how to do this, check chapter 3 for the details.

The next step is to change your JSP page form elements to contain the simpler OGNL references. The following snippet shows what the UpdateAccountForm.jsp page would look like:

```
<s:form action="UpdateAccount">
  <s:label key="username" />
  <s:hidden name="username" />
  <s:password name="password" label="Password" showPassword="true"/>
  <s:textfield name="portfolioName" label="Initial Portfolio"/>
  <s:textfield name="email" label="Email Address"/>
  <s:checkbox name="receiveJunkMail"  label="Do you want to receive junk
                                          mail?" />
  <s:submit/>
</s:form>
```

This differs only slightly from the current page. Here we've omitted the user level of the OGNL and are left with top-level references such as `username` and `password`. Based on this change of reference namespace, we now need to make a single, slight change to the `visitor` validator mapping in UpdateAccount-validation.xml. Here's the new content of that file:

```
<validators>
  <field name="model">
    <field-validator type="visitor">
       <param name="appendPrefix">false</param>
       <message>User:  </message>
    </field-validator>
  </field>
</validators>
```

Two changes have been made. First of all, since our `ModelDriven` domain object is exposed with the `getModel()` getter method, we now need to change the field name to `model`. Second, we need to use the `appendPrefix` parameter to tell the `visitor` validator that we no longer need the user prepended to the field names. Setting this parameter to `false` allows the validator to find the top-level field names. That's it. Combining the validation framework with `ModelDriven` actions is approaching a pretty efficient level of development.

In the next section, we investigate another high-level development technique and explore how it fits in with the validation framework.

10.4.2 Using validation context to refine your validations

We've already seen how to define your validation metadata on a per-action and per-domain class level. As it turns out, you might find you need a more fine-grained level of control over what validations run when. In order to control this, the validation framework introduces the notion of *context*. Context is pretty straightforward. Validation context provides a simple means of identifying the specific location in the application that's using the data that we want to validate.

The first use case for validation context arises when you use the framework's ability to define more than one entry point method for action execution. Thus far, we've stuck to the default `execute()` method as our single point of entry into an action. In chapter 15, we'll show you how to use multiple entry point methods on a single action class. These multiple entry point methods can then be mapped to multiple Struts 2 actions

that have different names, or aliases. Note the difference between a Struts 2 action component, one of the action mappings from your declarative architecture, and an action class, the Java class that can be used to back a Struts 2 action component.

Imagine we have a Java action class that provides a sorting capability. Now, suppose that we want to have several sort algorithms exposed by this action class. We could have put each one in its own class, but it makes sense to gather them together into a single class, both from a logical and design point of view. So, we have something like `manning.sort.SortAction` and it exposes two entry point methods named `bubbleSort()` and `heapSort()`. And let's assume we don't even use the `execute()` method, since Struts 2 doesn't require us to.

Here's a snippet of XML that declares two Struts 2 action components that both use this one action class:

```
<action name="BubbleSort" class="manning.sort.SortAction"
                          method="bubbleSort">
  <result>/sort/SortResults.jsp</result>
</action>
<action name="HeapSort" class="manning.sort.SortAction" method="heapSort">
  <result>/sort/SortResults.jsp</result>
</action>
```

Now let's assume that we have our validations defined in a single file for the `SortAction` named, appropriately, SortAction-validation.xml. This is what we did earlier in this chapter. This works great, but what happens if we decide that one of the sort entry points, or contexts, requires a different set of validations? Obviously, we need a finer granularity for our validation definitions. The notion of validation context solves this problem ASAP. If we need a different set of validations for each entry point method, we just replace the single SortAction-validation.xml with two new files, SortAction-BubbleSort-validation.xml and SortAction-HeapSort-validation.xml. Just to be clear, the naming convention is ActionClassName-aliasName-validation.xml. When one of the aliases from the preceding snippet is invoked, the validation framework automatically picks up the corresponding alias-named validation file. Note that if you still have a general SortAction-validation.xml defined, the validations defined in it will also be loaded. The aliased validations do not prevent the general ones from being used. This allows you to define common validations separate from the ones specific to the various contexts. Now, let's look at another use case where validation context can be helpful.

USING VALIDATION CONTEXT WITH THE VISITOR VALIDATOR AND DOMAIN OBJECTS

Earlier in this chapter, we learned how to use the `visitor` validator to point to validations declared at the level of the domain object itself. Validation context can be used in these situations as well. There are a couple of ways you can use validation context with the `visitor` validator. The first method follows the alias patterns described in our earlier `SortAction` example. In the `SortAction` example, we defined our validations at the action class level. Now imagine that we've defined the validations at the domain object level instead. If our domain object is the `User`, we end up with validations in a file next to the `User` class with a name of User-validation.xml.

Again, assume we need to have different validation rules for the `User` class depending upon the context in which it's being used. One option follows the previous pattern to define separate files for the different contexts that arise from different action aliases. The solution here is to create, again, a file for each alias such as ClassName-aliasName-validation.xml. If our `User` was being used from the two contexts of the `SortAction` defined previously, we could break the validations into the two files User-BubbleSort-validation.xml and User-HeapSort-validation.xml. Again, the validation framework is aware of the alias/context under which the action is being invoked and automatically use the correct validations.

The final variation on context usage aims to check the proliferation of validation definitions in the face of numerous alias-based contexts. A domain object such as `User` will most likely be used from dozens of locations throughout the application. If we define our validations at the `User` level, and we require context-based variations in those validations, then we could easily end up with a mess of User-thisAlias-validation.xml and User-thatAlias-validation.xml. To solve this problem, the `visitor` validator provides the `context` attribute to allow the developer to introduce a user-defined context. Let's say that the `User` object is used from 15 different action aliases. If we can identify special validation needs shared by 7 of these and another set of needs shared by the other 8, then we really only have 2 contexts, not 15. But obviously we can use the alias names of these 15 with the alias-naming scheme for our XML.

The solution is to invent two logical names for the two sets of validation needs. Let's say the two needs can be described as `admin` and `public`. With this determination made, we simply divide our validations into two files named for these contexts, User-admin-validation.xml and User-public-validation.xml. Since these context names don't match the alias names of our various action mappings, we need to somehow link the various action mappings to these contextual validations. To do this, we use the `context` attribute of the `visitor` validator. Recall that this validator is placed in the action-local validations file. Let's say we have an action, `UpdateUserAction`, that allows us to update the user, and that it exposes two aliased entry points, one for users to update themselves—`UpdateUser`—and one for administrators to update user data—`UpdateUserAdmin`. The second one obviously requires that the `User` be validated under the `admin` context. Here's how it works.

We've already defined our two contextual validations files at the `User` level. Now we just need to configure the `visitor` validator for the alias. The validations file for the `UpdateUserAdmin` alias would be UpdateUserAction-UpdateUserAdmin-validation.xml and it would contain the following visitor definition:

```
<field name="user">
  <field-validator type="visitor">
    <param name="context">admin</param>
    <message>User: </message>
  </field-validator>
</field>
```

This visitor now seeks out the `admin` validations defined at the domain object level in the file User-admin-validation.xml. Again, note that any validations defined generally for the `User` in User-validation.xml will also be used with the contextual validations.

10.4.3 Validation inheritance

Now that we've seen that validations can be declared at various levels and under various contexts, we need to briefly describe inheritance of validation declarations. This is simple, and we've already mentioned it indirectly. Recall that we said that general validations still run when contextual validations are also defined. This is part of the inheritance chain, but it's actually more complex. The following list shows the locations from which validations are collected when the framework begins its processing:

- *SuperClass*-validation.xml
- *SuperClass-aliasName*-validation.xml
- *Interface*-validation.xml
- *Interface-aliasName*-validation.xml
- *ActionClass*-validation.xml
- *ActionClass-aliasName*-validation.xml

When defining your validations, you should take advantage of this structure to define common validations at higher levels in the search list, thus allowing you to reuse definitions.

In the next section, we see how to forgo unnecessary validations by short-circuiting when one validation fails.

10.4.4 Short-circuiting validations

A useful feature of the validation framework is the ability to short-circuit further validation when a given validation fails. Let's say you have a series of validations defined for a given field. Take our password field for example. Listing 10.7 shows the declaration of our `password field` validators from User-validation.xml.

Listing 10.7 Using the `short-circuit` attribute to cancel unnecessary validations

```
<field name="password">
   <field-validator type="stringlength" short-circuit="true">
     <param name="maxLength">10</param>
     <param name="minLength">6</param>
     <message>Your password should be 6-10 characters.</message>
   </field-validator>
   <field-validator type="passwordintegrity">
      <param name="specialCharacters">$!@#?</param>
      <message>Your password must contain one letter, one number, and
            one of the following "${specialCharacters}".
      </message>
   </field-validator>
</field>
```

The only thing we've added here is the `short-circuit` attribute, which we've set to `true`. We've done this because we don't want the `passwordintegrity` check to run if the `stringlength` check fails. There's no point in wasting processing resources and there's no point in adding another error message to the user interface. Note that since this `short-circuit` is defined on a `field` validator, the rest of the validations for that field are short-circuited. If you define a short circuit at the action level, all validations are short-circuited.

With this, you've seen all the features offered by the validation framework. Before closing the chapter, we'll take the next section to tell you how you could set up your validations with annotations rather than XML-based metadata. All of the same features are available; it's just a different interface to the metadata.

10.4.5 Using annotations to declare your validations

At all levels of Struts 2, you generally have a choice between using XML or Java annotations for your various configuration and metadata needs. The validation framework is no different. In this section, we show you how to use annotations to declare your validations. Warning: this section makes no attempt at thoroughness. The idea here is to merely show you how it's done, roughly, and refer you to the proper resources if you feel like you want to use annotations.

First of all, the validation framework is the validation framework no matter how you declare your metadata. With that in mind, we just write a version of the `Register` action that uses annotations to declare its validations. Our new version of the `Register` action is `manning.chapterTen.RegisterValidationAnnotated`. Listing 10.8 shows the source.

Listing 10.8 Using annotations to describe the required metadata

```
@Validation      ❶

public class RegisterValidationAnnotated extends ActionSupport
                                implements SessionAware {
  @ExpressionValidator(expression = "username != password",
              message = "Username and password can't be the same.")      ❷
  public String execute(){

    User user = new User();
    user.setPassword( getPassword() );
    Portfolio newPort = new Portfolio();
    newPort.setName( getPortfolioName() );
    user.getPortfolios().put ( newPort.getName(), newPort );
    user.setUsername( getUsername() );
    user.setReceiveJunkMail( isReceiveJunkMail() );
    user.setEmail( getEmail() );

    getPortfolioService().createAccount( user );
    session.put( Struts2PortfolioConstants.USER, user );

    return SUCCESS;
  }
```

```
    private String username;
    private String password;
    private String portfolioName;
    private boolean receiveJunkMail;
    private String email;

    @RequiredStringValidator(type = ValidatorType.FIELD,
                     message="Email is required.")
    @EmailValidator(type = ValidatorType.FIELD, key="email.invalid",
                                message="Email no good.")
    public void setEmail(String email) {
      this.email = email;
    }
    public String getEmail() {
      return email;
    }

    @RequiredStringValidator(type = ValidatorType.FIELD,
                  message = "Portfolio name is required.")
    public String getPortfolioName() {
      return portfolioName;
    }
    public void setPortfolioName(String portfolioName) {
      this.portfolioName = portfolioName;
    }

    @StringLengthFieldValidator(type = ValidatorType.FIELD, minLength="5" ,
            maxLength = "8",  message = "Password must be between
            ${minLength}  and ${maxLength} characters.")
     @RequiredStringValidator(type = ValidatorType.FIELD,
                    message = "Password is required.")
    public String getPassword() {
      return password;
    }
    public void setPassword(String password) {
      this.password = password;
    }

    @RequiredStringValidator(type = ValidatorType.FIELD,
                           message = "Username is required.")
    @StringLengthFieldValidator(type = ValidatorType.FIELD, minLength="5" ,
                  maxLength = "8",  message = "Username must be between
                  ${minLength} and ${maxLength} characters.")
    public String getUsername() {
      return username;
    }
    public void setUsername(String username) {
      this.username = username;
    }

}
```

These annotations work with the same built-in validators that we've been working with throughout this chapter. We still map field validators to JavaBeans properties and map nonfield validators to the action as a whole. Let's see how it works. First, we must include a class-level annotation ❶ to mark the entire class as something that should be

scanned by the validation framework. Next, we define a nonfield annotation, the same expression validator ❷ that checks to make sure that the `username` and `password` aren't identical.

Next, we define all of our `field` validators. These must come right before the getter and setter of the corresponding property. We can see many of these throughout this class. For instance, we associate both the `requiredstring` and `email` validators with the `email` property by placing annotations just before the getter and setter for the `email` property ❸. Note that the annotations set the same parameters that we set in our XML-based validation metadata.

One slight difference is in the way the message is handled. The `message` attribute is a required attribute of the annotations. If you want to use a message from your properties file resource bundle, you don't write the key in as the `message` value. Rather, you add the `key` attribute to the annotation, as we've done for the `email` field's `email` validator ❹. Contrary to what you might expect, you still have to put a message value in the `message` attribute. It's considered the default message, in case the key lookup fails. While it may never reach the light of day, it's required.

If you want to see this annotated version in action, simply switch it out for the regular one in the chapterTen.xml file. Here's what the `Register action` element from that file would look like if you use the annotated version:

```
<action name="Register"
         class="manning.chapterTen.RegisterValidationAnnotated">
  <result>/chapterTen/RegistrationSuccess.jsp</result>
  <result name="input">/chapterTen/Registration.jsp</result>
</action>
```

As we've mentioned in respect to some of the other Struts 2 annotations, the annotations mechanisms are being improved daily. They're all part of a movement toward a zero-configuration development pattern that seeks to eliminate all of the XML artifacts. If you're keen on annotations, it would pay to keep an eye on the Struts 2 website and mailing lists. Completing the annotations is one of the priority milestones of the Struts 2 project. You may be able to say goodbye to those XML files sooner than you think.

10.5 Summary

We've now covered the validation framework in substantial detail. Before we move on, we should recap what we've learned in this chapter. First of all, the validation framework extends the work of the previous chapter to show yet another way to refine your Struts 2 development practices. With the validation framework, we can add a more sophisticated level of validation to our applications. While earlier chapters introduced a strong form of basic validation that uses the `Validateable` interface and its `validate()` method, the validation framework provides many enterprise-strength benefits not found with the `validate()` method strategy.

The most obvious benefit of using the validation framework arises from the reusable nature of the `Validator` component. This component contains the logic for a

given validation type and makes that validation available to any data that wants to be verified against that logic. We saw that Struts 2 comes with a set of 13 built-in validators that developers can immediately take advantage of to start validating their own data. Associating these with your data, both at the action level and the domain object level, can be done quickly with convenient XML or annotations-based metadata.

We also learned how to write custom validators that allow developers to add their own validation logic to the framework, thus allowing them to reuse their own efforts as easily as they use the built-in components. We hope that you'll be able to create many powerful validators, and we hope that you'll contribute them back to the Struts 2 community. But you don't have to, of course ;)

In the next chapter, we'll cover another topic of application refinement, internationalization.