STRUTS2 IN ACTION

Donald Brown
Chad Michael Davis
Scott Stanlick

MANNING

Table of Contents

5 Data transfer: OGNL and type conversion

This chapter covers

- Transferring data
- Working with OGNL
- Using the built-in type converters
- Customizing type conversion

Now that we've covered the action and interceptor components, you should have a good idea of how the major chunks of server-side logic execute in the Struts 2 framework. We've avoided poking our noses into the details of two of the more important tasks that the framework helps us achieve: data transfer and type conversion. We've been able to avoid thinking about these important tasks because the framework automates them so well. This'll continue to be true for large portions of your development practice. However, if we give a small portion of our energy to learning how the data transfer and type conversion actually works, we can squeeze a whole lot more power out of the framework's automation of these crucial tasks.

We've already learned how to take advantage of the automatic data transfer for simple cases. In this chapter, we'll learn how to take advantage of more complex

forms of automatic data transfer. Most of the increased complexity comes when transferring data onto more complex Java-side types, such as `Maps` and `Lists`. We haven't mentioned it much so far, but when the framework transfers data from string-based request parameters to strictly typed Java properties, it must also convert from string to Java type. The framework comes with a strong set of built-in type converters that support all common conversions, including complex types such as `Maps` and `Lists`. The central focus of this chapter will be explaining how to take advantage of the framework's ability to automatically transfer and convert all manner of data. At the end of the chapter, we'll also show you how to extend the type conversion mechanism by developing custom converters that can handle any types, including user-defined types.

This chapter also starts our two-part formal coverage of OGNL. OGNL is currently the default expression language used to reference data from the various regions of the framework in a consistent manner. We've already seen how to use OGNL expressions to point incoming form fields at the Java properties they should target when the framework transfers the request data. Accordingly, this chapter will cover OGNL from the point of view of incoming data transfer and type conversion. But OGNL is also going to be critical when we introduce the Struts 2 tag API in the next chapter. The tag is used to pull data out of the framework into the rendering response pages. We'll thus divide the coverage of OGNL between this chapter and the next. To be specific, this chapter will focus on how OGNL fits into the framework, and the role it plays in binding data throughout the various regions of the framework. The next chapter, which introduces tags, will cover the OGNL expression language from a more syntactic perspective, which you will need when using the tags.

Let's start by examining the data transfer and type conversion mechanisms at close range.

5.1 Data transfer and type conversion: common tasks of the web application domain

In chapter 1 of this book, we said that one of the common tasks of the web application domain was moving and converting data from string-based HTTP to the various data types of the Java language. If you've worked with web applications for many years, you'll be familiar with the tedious task of moving data from form beans to data beans. This boring task is complicated by the accompanying task of converting from strings to Java types. Parsing strings into doubles and floats, catching the exceptions that arise from bad data, and so on is no fun at all. Worse yet, these tasks amount to pure infrastructure. All you're doing is preparing for the real work.

Data transfer and type conversion actually happen on both ends of the request-processing cycle. We've already seen that the framework moves the data from the string-based HTTP requests to our JavaBeans properties, which are clearly Java types. Moreover, the same thing happens on the other end. When the result is rendered, we typically funnel some of the data from those JavaBeans properties back out into the resulting HTML page. Again, while we haven't given it much thought, this means that the data has been reconverted from the Java type back out to a string format.

This process occurs with nearly every request in a web application. It's an inherent part of the domain. No one will moan about handing this responsibility over to the framework. Nonetheless, there'll be times when you want to extend or configure this automated support. The Struts 2 type conversion mechanisms are powerful and quite easily extended. We think you'll be excited when you see the possibilities for writing your own custom converters. First, though, we need see who's responsible for all of this automated wizardry.

5.2 OGNL and Struts 2

We call it wizardry, but, as we all know, computers are rational machines. Perhaps unsolved mystery is a more accurate phrase. What exactly are these unsolved mysteries? To be specific, we haven't yet explained how all of that data makes it from the HTML request to the Java language and back out to HTML through the JSP tags. The next section will clarify this mysterious process.

5.2.1 What OGNL does

What OGNL does isn't mysterious at all. In fact, OGNL is quite ordinary. *OGNL* stands for the *Object-Graph Navigation Language*. Sounds perfectly harmless, right? No? Actually, I agree. It sounds horrifying, as if I should have studied harder in school. In an attempt to make it sound less academic, the makers of OGNL suggest pronouncing it like the last few syllables of "orthogonal."

OGNL is a powerful technology that's been integrated into the Struts 2 framework to help with data transfer and type conversion. OGNL is the glue between the framework's string-based HTTP input and output and the Java-based internal processing. It's quite powerful and, while it seems that you can use the framework without really knowing about OGNL, your development efforts will be made many times more efficient by spending a few moments with this oddly named power utility.

From the point of view of a developer building applications on the Struts 2 framework, OGNL consists of two things: an expression language and type converters.

EXPRESSION LANGUAGE

First, let's look at the expression language. We've been using OGNL's expression language in our form input field names and JSP tags. In both places, we've been using OGNL expressions to bind Java-side data properties to strings in the text-based view layers, commonly found in the name attributes of form input fields, or in various attributes of the Struts 2 tags. The simplicity of the expression language, in its common usage, makes for a ridiculously low learning curve. This has allowed us to get deep into Struts 2 without specifically covering it. Let's review what we've already been doing.

The following code snippet, from our Struts 2 Portfolio application's RegistrationSuccess.jsp, shows a Struts 2 tag using the OGNL expression language:

```
<h5>Congratulations! You have created </h5>
<h3>The <s:property value="portfolioName" /> Portfolio</h3>
```

The OGNL expression language is the bit inside the double quotes of the `value` attribute. This Struts 2 property tag takes a value from a property on one of our Java

objects and writes it into the HTML in place of the tag. This is the point of expression languages. They allow us to use a simplified syntax to reference objects that reside in the Java environment. The OGNL expression language can be much more complex than this single element expression; it even supports such advanced features as invoking method calls on the Java objects that it can access, but the whole idea of an expression language is to simplify access to data.

HEADS-UP The integration of OGNL into the Struts 2 framework is tight. Pains have been taken to make the simplest use cases just that: simple. With this in mind, many instances of OGNL expressions require no special escaping. While there's an OGNL escape sequence, `%{expression}`, that signals to the framework when to process the expression as an expression rather than interpreting it as a string literal, this isn't often used. Using intelligent defaults, Struts 2 will automatically evaluate the string as an OGNL expression in all contexts that warrant such a default behavior. In contexts where strings are most likely going to be strings, the framework will require the OGNL escape sequence. As we move along, we'll specifically indicate which context is which.

Here's the other side of the coin. While the `property` tag, which resides in a result page, reaches back into the Java environment to pull a value from the `portfolioName` property, we've also seen that OGNL expressions are used in HTML forms to target properties in the Java environment as destinations for the data transfer. In both cases, the role of the OGNL expression is to provide a simple syntax for binding things like Struts 2 tags to specific Java-side properties, for moving data both into and out of the framework. OGNL creates the pathways for data to flow through the framework. It helps move data from the request parameters onto our action's JavaBeans properties, and it helps move data from those properties out into rendering HTML pages.

But we must investigate how the type conversion occurs when moving data between the string-based worlds of HTML and the native Java types of the framework.

TYPE CONVERTERS

In addition to the expression language, we've also been using OGNL type converters. Even in this simple case of the Struts 2 `property` tag, a conversion must be made from the Java type of the property referenced by the OGNL expression language to the string format of the HTML output. Of course, in the case of the `portfolioName`, the Java type is also a string. But this just means that the conversion is easy. Every time data moves to or from the Java environment, a translation must occur between the string version of that data that resides in the HTML and the appropriate Java data type. Thus far, we've been using simple data types for which the Struts 2 framework provides adequate built-in OGNL type converters. In fact, the framework provides built-in converters to handle much more than we've been asking of it. Shortly, we'll cover the built-in type converters and show you how to map your incoming form fields to a wide variety of Java data types, including all the primitives as well as a variety of collections. But, first, let's look at where OGNL fits into the framework, just to be clear about things.

5.2.2 How OGNL fits into the framework

Understanding OGNL's role in the framework, from an architectural perspective, will make working with it much easier. Figure 5.1 shows how OGNL has been incorporated into the Struts 2 framework.

Figure 5.1 shows the path of data into and out of the framework. Everything starts with the HTML form in the InputForm.html page, from which the user will submit a request. Everything ends with the response that comes back to the user, represented in figure 5.1 as ResultPage.html. Now, let's follow the data into and out of the framework and see how OGNL helps bind and convert the data as it moves from region to region.

DATA IN

Our data's journey starts at the InputForm.html page shown in figure 5.1. In this case, the form contains two text input fields. Note that, in the interest of space, we've created pseudo-HTML markup for these fields; this won't validate. The strings in the pseudo-text input tags are the `name` attributes of the fields. Again, it's important to realize that these names are valid OGNL expressions. All that we need now is a user to enter two values for the fields and submit the form to the framework.

When the request enters the framework, as we can see in figure 5.1, it's exposed to the Java language as an `HttpServletRequest` object. As we learned earlier, Struts 2 is built on the Servlet API. The request parameters are stored as name/value pairs, and

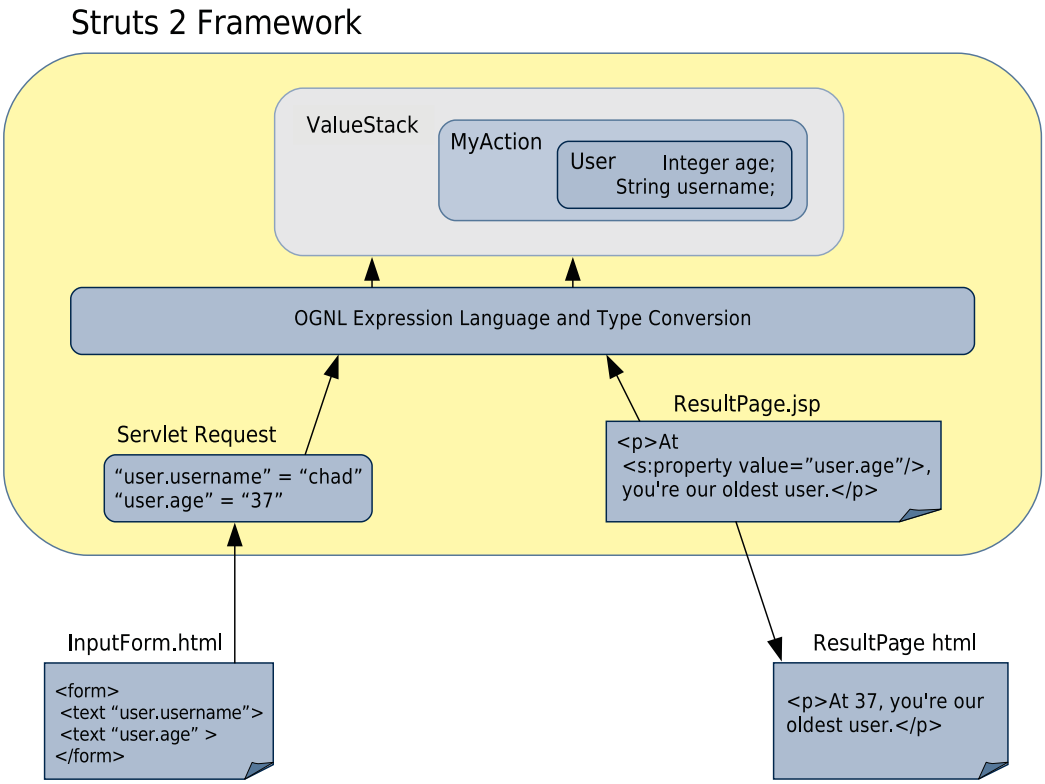

Figure 5.1 OGNL provides the framework's mechanism for transferring and type-converting data.

both name and value are `String`s. As you can see in figure 5.1, the request object just has a couple of name/value pairs, where the names are the names of our form's text fields and the values are the values entered by the user when the form was submitted. Everything is still a string. This is where the framework and OGNL pick up the ball.

We know the framework is going to handle the transfer and type conversion of the data from these request parameters. The first question is where should the data go? OGNL will point the way. But, first, OGNL needs a context in which to search for its targets. In chapter 3, we saw that, when the framework automatically transfers parameters to our action object, the action is sitting on something called the `ValueStack`. In figure 5.1, we can see that our action object has been placed on the `ValueStack`. In the case represented by this figure, we're exposing our `User` object as a JavaBeans property on our action object. With our action object on the `ValueStack`, we're ready for OGNL to do its navigational work.

From our study of interceptors, we know that the `params` interceptor will move the data from the request object to the `ValueStack`. The tricky part of the job is mapping the name of the parameter to an actual property on the `ValueStack`. This is where OGNL comes in. The `params` interceptor interprets the request parameter name as an OGNL expression to locate the correct destination property on the `ValueStack`. If you look at this in figure 5.1, you might expect that the expression would need to be something more like `myAction.user.username`. To the contrary, only the `user.username` is necessary. This is because the `ValueStack` is a sort of virtual object that exposes the properties of its contained objects as its own.

DEFINITION The `ValueStack` is a Struts 2 construct that presents an aggregation of the properties of a stack of objects as properties of a single virtual object. If duplicate properties exist—two objects in the stack both have a `name` property—then the property of the highest object in the stack will be the one exposed on the virtual object represented by the `ValueStack`. The `ValueStack` represents the data model exposed to the current request and is the default object against which all OGNL expressions are resolved.

The `ValueStack` is a virtual object? It sounds complicated, but it's not. The `ValueStack` holds a stack of objects. These objects all have properties. The magic of the `ValueStack` is that all the properties of these objects appear as properties of the `ValueStack` itself. In our case, since the action object is on the `ValueStack`, all of its properties appear as properties of the `ValueStack`. The tricky part comes when more than one object is placed on the `ValueStack`. When this happens, we can have a contention of sorts between properties of those two objects. Let's say that two objects on the stack both have a `username` property. How does this get resolved? Simply: the username exposed by the `ValueStack` will always be that of the highest object in the stack. The properties of the higher objects in the stack cover up similarly named properties of objects lower in the stack. We'll cover this in more detail when we discuss the OGNL expression language in chapter 6.

For now, this should be enough to see how the request parameters find the way to their correct homes. In figure 5.1, one of the request parameters is named `user.age`.

If we resolve this as an OGNL expression against the `ValueStack`, we first ask, "Does the `ValueStack` have a `user` property?" As we've just learned, the `ValueStack` exposes the properties of the objects it contains, so we know that the `ValueStack` does have a `user` property. Next, does this `user` property have an `age` property? Of course it does. Obviously, we've found the right property. Now what?

Once the OGNL expression has been used to locate the destination property, the data can be moved onto that property by calling the property's setter with the correct value. But, at this point, the original value is still the string "37". Here's where the type converters come into play. We need to convert the string to the Java type of the `age` property targeted by the OGNL expression, which is an `int`. OGNL will consult its set of available type converters to see if any of them can handle this particular conversion. Luckily, the Struts 2 framework provides a set of type converters to handle all the normal conversions of the web application domain. Conversion between strings and integers is provided for by the built-in type converters. The value is converted and set on the `user` object, just where we'll find it when we start our action logic after the rest of the interceptors have fired.

Now that we've seen how data makes it into the framework, let's work our way through the other half of figure 5.1 to see how it makes it back out.

DATA OUT

Now for the other half of the story. Actually, it's the same story, but in reverse. After the action has done its business, calling business logic, doing data operations, and so forth, we know that eventually a result will fire that'll render a new view of the application to the user. Importantly, during the processing of the request, our data objects will remain on the `ValueStack`. The `ValueStack` acts as a kind of place holder for viewing the data model throughout the various regions of the framework.

When the result starts its rendering process, it'll also have access to the `ValueStack`, via the OGNL expression language in its tags. These tags will retrieve data from the `ValueStack` by referencing specific values with OGNL expressions. In figure 5.1, the result is rendered by ResultPage.jsp. In this page, the age of the user is retrieved with the Struts 2 `property` tag, a tag that takes an OGNL expression to guide it to the value it should render. But, once again, we must convert the value; this time we convert from the Java type of the property on the `ValueStack` to a string that can be written into the HTML page. In this case, the `Integer` object is converted back into a string for the depressing message that "At 37, you're our oldest user." Must be a social networking site.

Now you know what OGNL does. This chapter is going to focus on the framework's data transfer and type conversion. We won't say much more on the details of OGNL, such as its expression language syntax. We'll save that for the next chapter, which covers the Struts 2 tags. For now, you just need to understand the big picture of what OGNL does for the framework.

If you need a break before we go on to cover the built-in type converters, you could practice saying "OGNL."

5.3 Built-in type converters

Now that we've seen how OGNL has been integrated into the framework to provide automatic data transfer and type conversion, it's time to see the nuts and bolts of working with the built-in type converters. Here's where we'll learn how to safely guide a variety of data types into and out of the framework. As we saw in the previous section, type conversion plays an important role in that process. Out of the box, the Struts 2 framework can handle almost any type conversions that you might require. These conversions are done by the built-in type converters that come with the framework.

In this section, we'll show you what the framework will handle, give you plenty of examples, and show you how to write OGNL expressions for form field names that need to locate more complexly typed properties such as arrays, `Maps`, and `Lists`. In the coming pages, we'll provide a clear-cut how-to and reference for using OGNL to map incoming form fields to Java properties of all the types supported by the framework's built-in type converters. When you start developing the forms that'll submit data to your actions, this section will be a great reference.

5.3.1 Out-of-the-box conversions

The Struts 2 framework comes with built-in support for converting between the HTTP native strings and the following list of Java types:

- `String`—Sometimes a string is just a string.
- `boolean/Boolean`—`true` and `false` strings can be converted to both primitive and object versions of Boolean.
- `char/Character`—Primitive or object.
- `int/Integer`, `float/Float`, `long/Long`, `double/Double`—Primitives or objects.
- `Date`—String version will be in `SHORT` format of current `Locale` (for example, 12/10/97).
- `array`—Each string element must be convertible to the array's type.
- `List`—Populated with `Strings` by default.
- `Map`—Populated with `Strings` by default.

When the framework locates the Java property targeted by a given OGNL expression, it'll look for a converter for that type. If that type is in the preceding list, you don't need to do anything but sit back and receive the data.

In order to utilize the built-in type conversion, you just need to build an OGNL expression that targets a property on the `ValueStack`. The OGNL expression will either be the name of your form field, under which the parameter will be submitted in the HTTP request, or it'll be somewhere in your view-layer tags, such as one of the Struts 2 JSP tags. Again, our current discussion will focus on the data's entry into the framework as request parameters. Chapter 6 will focus on the tag point of view. However, this is mostly a matter of convenience; OGNL serves the same functional roles at both ends of the request processing—its expression language navigates our object graph to locate the specified property, and the type converters manage the data type

translations between the string-based HTTP world and the strictly typed Java world. Data in, data out? It doesn't matter. The type conversion and OGNL will be the same.

5.3.2 Mapping form field names to properties with OGNL expressions

Hooking up your Java properties to your form field names to facilitate the automatic transfer and conversion of request parameters is a two-step process. First, you need to write the OGNL expressions for the name attributes of your form fields. Second, you need to create the properties that'll receive the data on the Java side. You could do these in reverse order; it doesn't matter. We'll go through each of the built-in conversions in the order from the preceding bulleted list, showing how to set up both sides of the equation.

PRIMITIVES AND WRAPPER CLASSES

Because the built-in conversions to Java primitives and wrapper classes, such as `Boolean` and `Double`, are simple, we provide a single example to demonstrate them. We won't show every primitive or wrapper type; they all work the same way. Let's start by examining the JSP-side OGNL. Listing 5.1 shows the chapter 5 version of our Struts 2 Portfolio's registration form, from Registration.jsp.

Listing 5.1 OGNL expressions that target specific properties on the `ValueStack`

```
<h4>Complete and submit the form to create your own portfolio.</h4>
<s:form action="Register">
  <s:textfield name="user.username" label="Username"/>
  <s:password name="user.password" label="Password"/>
  <s:textfield name="user.portfolioName" label="Enter a name "/>
  <s:textfield name="user.age" label="Enter your age as a double "/>
  <s:textfield name="user.birthday" label="Enter birthday. (mm/dd/yy)"/>
  <s:submit/>
</s:form>]
```

This is nothing new. But now that you know that each input field name is actually an OGNL expression, you'll see a lot deeper into this seemingly simple form markup. Recall that our OGNL expressions resolve against the `ValueStack`, and that our action object will be automatically placed there when request processing starts. In this case, our `Register` action uses a JavaBeans property, `user`, backed directly with our `User` domain object. The following snippet, from our chapter 5 version of Register.java, shows the JavaBeans property that exposes our `User` object:

```
private User user;

public User getUser() {
  return user;
}

public void setUser(User user) {
  this.user = user;
}
```

If you want, you can see the full source for Register.java by looking at the sample application, but it's nothing new. The only thing important to our current discussion is the

exposure of the user object as a JavaBeans property. Since the type of this property is our `User` class, let's look at that class to see what properties it exposes. Listing 5.2 shows the full listing of the `User` bean.

Listing 5.2 The JavaBeans properties targeted by the OGNL expressions in Listing 5.1

```
public class User {

  private String username;
  private String password;
  private String portfolioName;
  private Double age;
  private Date birthday;

  public String getPassword() {
    return password;
  }
  public void setPassword(String password) {
    this.password = password;
  }
  public String getPortfolioName() {
    return portfolioName;
  }
  public void setPortfolioName(String portfolioName) {
    this.portfolioName = portfolioName;
  }
  public String getUsername() {
    return username;
  }
  public void setUsername(String username) {
    this.username = username;
  }
  public Double getAge() {
    return age;
  }
  public void setAge(Double age) {
    this.age = age;
  }
  public Date getBirthday() {
    return birthday;
  }
  public void setBirthday(Date birthday) {
    this.birthday = birthday;
  }
}
```

As you can see, the action just exposes a bunch of JavaBeans properties to carry data. Let's put them to use. Go ahead and test the registration out. Click the Create an Account link from the chapter 5 version of the Struts 2 Portfolio application. The request comes into the framework with a map of name/value pairs that associate the name from the form input field with the string value entered. The name is an OGNL expression. This expression is used to locate the target property on the `ValueStack`. In the case of the `Register` action, the action itself is on top of the stack and an OGNL

expression such as `user.birthday` finds the `user` property on the action, then finds the `birthday` property on that user object. In Java, this becomes the following snippet:

```
getUser().getBirthday();
```

OGNL sees that the birthday property is of Java type `Date`. It then locates the string-to-`Date` converter, converts the value, and sets it on the property. All of the simple object types and primitives are just this easy. If the incoming string value doesn't represent a valid instance of the primitive or type, then a conversion exception is thrown. Note the difference between type conversion and validation, and thus the difference between a type conversion error and a validation error. Validation code is about validating the data as valid instances of the data types from the perspective of the business logic of the action; this occurs via the `validation` interceptor or the `workflow` interceptor's invocation of the `validate()` method. Conversion problems occur when trying to bind the HTTP string values to their Java types; this occurs, for instance, when the `params` interceptor transfers the request data.

Conversion errors result in the user being returned to the input page, similar to validation errors. Normally, a default error message will inform the user that the string value he submitted cannot be converted to the Java type targeted by the OGNL. You can customize the error reporting done in the face of type conversion problems, but we'll save that for later in the book. In chapter 11, we'll learn how to customize the conversion exception handling and the error messages shown to the user when such conversion errors arise.

Now that we've seen how to map your incoming data to Java primitives and their wrapper classes, let's see what Struts 2 can do to automatically handle various multivalued request parameters.

HANDLING MULTIVALUED REQUEST PARAMETERS

As you probably know, multiple parameter values can be mapped to a single parameter name in the incoming request. There are a variety of ways for a form to submit multiple values under a single parameter name. There are also many ways to map these to Java-side types, implying a variety of ways to wield our OGNL. In the coming sections, we'll cover the ways you can handle this.

Struts 2 provides rich support for transferring multivalued request parameters to a variety of collection-oriented data types on the Java side, ranging from arrays to actual `Collections`. In the interest of being semi-exhaustive, we won't attempt to integrate the following examples into the functional soul of the Struts 2 Portfolio. Rather, you'll find that a portion of the chapter 5 version of the sample application has been specifically dedicated to demonstrating the various techniques shown in the coming pages.

For each of these examples, we'll reuse a single action object, the `DataTransferTest`. From the perspective of data transfer and type conversion, action objects need only expose the properties that'll receive the data. The `DataTransferTest` exposes all of the properties for the examples in this chapter. We did this to consolidate the various permutations of data transfer into a convenient point of reference. But each example is mapped in the chapterFive.xml file as a distinct Struts 2 action component, which is

perfectly valid. Note the semantic difference between a Struts 2 action component and a Java class that provides an action implementation. We can, and do, reuse a single class for multiple actions.

These examples demonstrate how to set up the data transfer and type conversion. Our action will do little more than serve as a data holder for these examples. Forms will submit request data, the framework will transfer that data to the properties exposed on the action, the action will do nothing but forward to the success result, and that result will display the data by pulling it off of the action with Struts 2 tags. This should serve as a clean reference that'll make all the variations on data transfer and type conversion crystal clear. We encourage you to use this portion of the sample application as a reference for proper OGNL-to-Java property mapping.

Now, let's see how to have Struts 2 automatically transfer multiple values to array-backed properties.

ARRAYS

Struts 2 provides support for converting data to Java arrays. If you've worked with array-backed properties, also known as *indexed JavaBeans properties*, you'll appreciate the ease with which Struts 2 handles these properties. Most of these improvements come from the OGNL expression language's support for navigating to such properties. You can see the array data transfer in action by clicking the array data transfer link on the chapter 5 home page. Listing 5.3 shows the form from ArraysDataTransferTest.jsp that submits data targeted at array properties.

Listing 5.3 Targeting array properties for data transfer

```
<s:form action="ArraysDataTransferTest">
  <s:textfield name="ages" label="Ages"/>
  <s:textfield name="ages" label="Ages"/>
  <s:textfield name="ages" label="Ages"/>

  <s:textfield name="names[0]" label="names"/>
  <s:textfield name="names[1]" label="names"/>
  <s:textfield name="names[2]" label="names"/>

  <s:submit/>
</s:form>
```

❶ These target the ages property

❷ These target the names property

On the OGNL expression side, you just have to know how to write an expression that can navigate to an array property on a Java object. The form shown in listing 5.3 submits data to two different array properties. The first array property, named `ages`, will receive the data from the first three fields ❶. The second array property, `names`, will receive the data from the second three fields ❷. These properties, if the transfer is to work, must exist on the `ValueStack`. For this example, we'll expose the array properties on our action object, which we'll see momentarily.

This form demonstrates two syntaxes for targeting arrays with OGNL expressions. To understand what'll happen, we need to refresh our memories of the HTTP and Servlet API details that will occur as a result of this form being submitted. The first thing to remember is that the name of each input field, as far as HTTP and the Servlet

API are concerned, is just a string name. These layers know nothing about OGNL. With that in mind, it's time for a pop quiz. How many request parameters will be submitted by this form? The correct answer is four. The first three fields all have the same name; this'll result in a single request parameter with three values, perfectly valid in HTTP. On the other hand, the second set of fields will each come in as a distinct parameter with a single value mapped to it. When this request hits the framework, four request parameters will exist, as follows:

Parameter name	Parameter value(s)
ages	12, 33, 102
names[0]	Chad
names[1]	Don
names[2]	Beth

Now let's look at the implementation of the properties that'll receive this data. These properties will be exposed on our action object. Listing 5.4 shows the target properties, each of type `Array`, from DataTransferTest.java.

Listing 5.4 Array properties targeted by OGNL input field names

```
private Double[] ages ;

public Double[] getAges() {
  return ages;
}

public void setAges(Double[] ages) {
  this.ages = ages;
}

private String[] names = new String[10];

public String[] getNames() {
  return names;
}

public void setNames(String[] names) {
  this.names = names;
}
```

First, note that we don't need indexed getters and setters for these properties. OGNL handles all the indexing details. We just need to expose the array itself via a getter and setter pair. Now, consider what happens when the framework transfers the `ages` parameter. First, it resolves the property and finds the `ages` property on the action, as seen in listing 5.4. The value of the `ages` parameter in the request is an array of three strings. Since the `ages` property on the action is also an array, this makes the data transfer simple. OGNL creates a new array and sets it on the property. But OGNL does

even more for us. In this case, the `ages` property is an array of element type `Double`. OGNL sees this and automatically runs its type conversion for each element of the array. Very nice! Also note that, since the framework is creating the array for us in these cases, we don't need to initialize the array ourselves. For this example, we used multiple text input fields with the same name to submit multiple values under the `ages` parameter. In real applications, parameters with multiple values mapped to them are frequently the result of input fields that allow selection of multiple values, such as a select box.

Now let's look at how the framework handles the three individual parameters with names that look like array indexing. As far as the Servlet API is concerned, these are just three unique names. We can see that they seem to refer to a single array, and provide indexing into that single array, but the Servlet API sees only unique strings. However, when the framework hands these names to OGNL, they're accurately interpreted as references to specific elements in a specific array. These parameters are set, one at a time, into the elements of the names array. We should make a couple of comments before moving on. First, using this method requires initializing the array. This is necessary because the OGNL expressions are targeting individual elements of an existing array; the previous method was setting the entire array, so it didn't require an existing array. Second, we still don't need indexed getters and setters!

With all of this in place, the framework will automatically transfer and convert the request parameters onto our action's properties. The action, in these examples, does nothing but forward to the result page, ArraysDataTransferSuccess.jsp. The following snippet shows the code from this page:

```
<h5>Congratulations! You have transferred and converted data to and from
  Arrays.</h5>
<h3>Age number 3 = <s:property value="ages[2]" /> </h3>
<h3>Name number 3 = <s:property value="names[2]" /> </h3>
```

We don't want to say too much about the Struts 2 tags now; that's the topic of the next chapter. But it should be easy enough to understand that this result page pulls some data off the action's array properties just to prove that everything's working. The rest of the examples in this chapter will follow a similar pattern of using a result page to pull the data off the action just to verify that the transfer and conversion is working. We may not show this code from the result pages every time though.

Many developers prefer to work with some of the more feature-rich classes from the Java Collections API. Next, we'll look at working with `Lists`.

LISTS

In a fashion similar to array properties, Struts 2 supports automatically converting sets of request parameters into properties of various `Collection` types, such as `Lists`. Using `Lists` is almost like using arrays. The only difference is that `Lists`, prior to Java 5, don't support type specification. This typeless nature of `Lists` has an important consequence for the type conversion mechanisms of Struts 2. When the framework works with arrays, the type conversion can easily find the element type by inspecting the property itself, as arrays are always typed in Java. With `Lists`, there's no way to automatically discover this.

We have two choices when working with `Lists`: either specify the type for our elements or accept the default type. By default, the framework will convert request parameters into `Strings` and populate the `List` property with those `Strings`. Our first example will accept this default behavior. The mechanics of using `Lists` are almost identical to using arrays. The following snippet shows the form field markup from ListsDataTransferTest.jsp that'll target some `List` properties.

```
<s:textfield name="middleNames[0]" label="middleNames"/>
<s:textfield name="middleNames[1]" label="middleNames"/>
<s:textfield name="middleNames[2]" label="middleNames"/>
```

These target the middleNames property

```
<s:textfield name="lastNames" label="lastNames"/>
<s:textfield name="lastNames" label="lastNames"/>
<s:textfield name="lastNames" label="lastNames"/>
```

These target the lastNames property

As you can see, we again show two different notations for referencing target properties with OGNL expressions. These are the same notations as used with arrays. The only difference is in the Java side. In Java, the `List` properties are much like the array properties except the type is different. Listing 5.5 shows the target `List` properties from DataTransferTest.java.

Listing 5.5 Using `List` properties to receive the data

```
private List lastNames ;

public List getLastNames()

{
  return lastNames;
}
public void setLastNames ( List lastNames ) {
  this.lastNames=lastNames;
}

private List middleNames ;

public List getMiddleNames()
{
  return middleNames;
}
public void setMiddleNames ( List middleNames ) {
  this.middleNames=middleNames;
}
```

These look much like the array properties, except the type is `List`. There are a couple of things to note. First, you don't have to preinitialize any of the `Lists`, even the ones that'll receive data from the indexed OGNL notation. Second, without type specification, the elements of these `Lists` will all be `String` objects. If that works for your requirements, great. In our case, our data is first and last names, so this is fine. But if you name your field `birthdays`, don't expect the framework to convert to `Dates`. It'll just make a `List` of `Strings` out of your incoming `birthday` strings. If you want to see this example in action, check out the List Data Transfer Test link on the chapter 5 home page. Again, the result page will pull some values out of the `List` properties on

the action just to prove everything is working as advertised. If you want to look at the JSP to see the tags, check out ListsDataTransferSuccess.jsp.

Sometimes, you'll want to specify a type for your `List` elements rather than just working with `String`s. No problem. We just need to inform OGNL of the element type we want for a given property. This is done with a simple properties file. The OGNL type conversion uses properties files for several things. Later in the chapter, when we write our own type converters, we'll see another use for these files. For now, we're just going to make a properties file that tells OGNL what type of element we want in our `List` property. In order to specify element types for properties on our action object, we create a file according to the naming convention shown in figure 5.2.

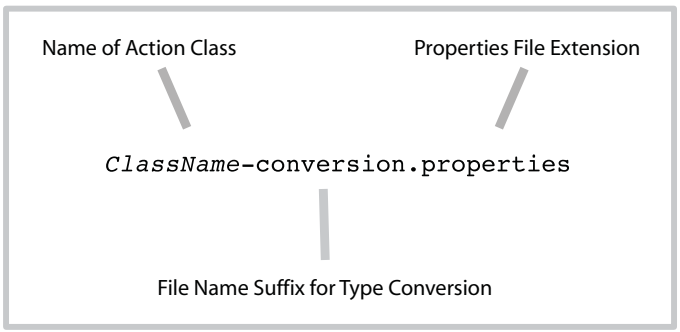

Figure 5.2 Naming convention for type conversion properties files

We then place this file next to the class in your Java package. We're going to create a file called DataTransferTest-conversion.properties and place it next to our `DataTransferTest.java` class in the `manning.chapterFive` package. If you check out the sample application, you'll see that this is the case. Figure 5.3 provides an anatomical dissection of the single property from that file.

This brief line, `Element-weights=java.lang.Double`, is all the type conversion process needs to add typed elements to our `List`. Now, our `List` property will work just like the array property; each individual element will be converted to a `Double`. Here's the markup from ListsDataTransferTest.jsp for our new `weights` property:

```
<s:textfield name="weights[0]" label="weights"/>
<s:textfield name="weights[1]" label="weights"/>
<s:textfield name="weights[2]" label="weights"/>
```

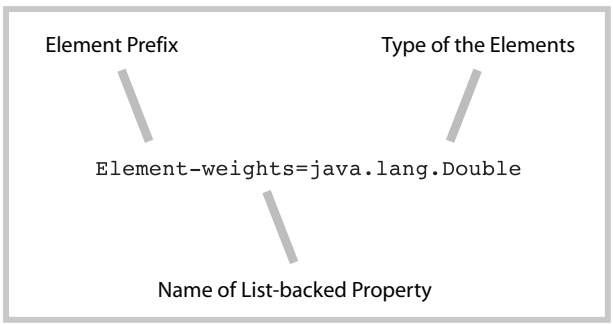

Figure 5.3 Specifying the type for a `List`-backed property

Note that we could've used the nonindexed OGNL notation. You may have a preference, but it doesn't matter to the framework. On the Java side, the property on the action object doesn't change. Here's the implementation of the `List` property from DataTransferTest.java.

```
private List weights;

public List getWeights() {
  return weights;
}

public void setWeights(List weight) {
  this.weights = weight;
}
```

You might've noticed that it's no different from the previous untyped version. This shouldn't be too surprising. Since the `List` is typeless from a Java point of view, we can't see that the elements are `Doubles` unless we try to cast them at runtime. Unless you want to modify the source code and test that the elements are actually `Doubles`, you'll have to take our word for it. Of course, when you use this technique in a real application, whose business logic will depend on those elements being `Doubles`, you'll know soon enough that they are, in fact, `Doubles`. Note that, if you want to see this in action, it's right there on the same page as the previous example.

NOTE *Java 5 generics and Type conversion*—If you have the pleasure of using Java 5 or higher, we highly recommend using generics to type your collections and maps. Besides being a recommended best practice, the Struts 2 type conversion mechanism can use generics-based typing to learn the correct target type for the conversions. If you do this, you don't have to use the properties file configuration. This is a big bonus for Java 5 users. Go Tiger!

Before moving on, we need to make one warning regarding this type specification process. When specifying the type for `Lists` and other `Collections`, take care not to preinitialize your `List`. If you do, you'll get an error. While untyped `Lists` will work whether you initialize your `List` or not, with type-specific `List` conversion, you can't preinitialize your `List`.

WARNING Don't preinitialize your `List` when using the typed element conversion mechanism supported by the ClassName-conversion.properties file.

Now we want to show a full-powered example. This example will use a `List` property that specifies the Struts 2 Portfolio's `User` class as its element type. This allows us to take advantage of the convenience of using `Lists` with the convenience of using our domain objects at the same time. Also, we want to prove to you that the type specification is actually working. Here's the markup, again from ListsDataTransferTest.jsp, that accesses our `List` of `Users`:

```
<s:textfield name="users[0].username" label="Usernames"/>
<s:textfield name="users[1].username" label="Usernames"/>
<s:textfield name="users[2].username" label="Usernames"/>
```

As you can see, these field names reference the `username` property on a `User` element in the `users` list, which is a property on our action. As before, from the property itself, which comes from DataTransferTest.java, we can discern nothing about the type of the elements it'll contain. Here's the property from our `DataTransferTest` action class:

```
private List users ;

public List getUsers()
{
  return users;
}
public void setUsers ( List users ) {
  this.users=users;
}
```

In order to make the framework populate our `List` with actual `User` objects, we once again employ the aid of the conversion properties file. The following line in DataTransferTest-conversion.properties specifies that our list-backed property will contain objects of type `User`.

```
Element_users=manning.utils.User
```

This is cool stuff. In case you want to verify the type-specific conversion, just check the tags in the ListsDataTransferSuccess.jsp page. They're reaching back into the `List` to retrieve usernames that simply wouldn't exist if the elements weren't of type `User`. If you've done much work with moving data around in older frameworks, we know you'll be able to appreciate the amount of work that something like this will save you.

Next, we'll point out a use case that you might not immediately infer from the previous examples. Let's assume that you have a `List` for which you specify a `User` element type. Now let's say that you expect an unpredictable amount of element data from the request. This could be due to something like a multiple select box. In this case, you simply combine the indexless naming convention with the deeper property reference. Note the following set of imaginary (they're not in the sample code) text fields:

```
<s:textfield name="users.username" label="Usernames"/>
<s:textfield name="users.username" label="Usernames"/>
<s:textfield name="users.username" label="Usernames"/>
```

This will submit a set of three `username` strings under the single parameter name of `users.username`. When OGNL resolves this expression, it'll first locate the `users` property. Let's assume this is the same `users` property as in the previous example. This means that `users` is a `List` for which the element type has been specified as `User`. This information allows OGNL to use this single parameter name, with its multiple values, to create a `List` and create `User` elements for it, setting the `username` property on each of those `User` objects.

That should be enough to keep you working with `Lists` for some time to come. Now we'll look at `Maps`, in case you need to use something other than a numeric index to reference your data.

MAPS

The last out-of-the-box conversion we'll cover is conversion to `Map`s. Similar to its support for `List`s, Struts 2 also supports automatic conversion of a set of values from the HTTP request into a `Map` property. `Map`s associate their values with keys rather than indexes. This keyed nature has a couple of implications for the Struts 2 type conversion process. First, the OGNL expression syntax for referencing them is different than for `List`s because it has to provide a key rather than a numeric index. The second implication involves specifying types for the `Map` properties. We've already seen that we can specify a type for our `List` elements. You can do this with `Map`s also. With `Map`s, however, you can also specify a type for the key object. As you might expect, both the `Map` element and key will default to a `String` if you don't specify a type. We'll explore all of this in the examples of this section. Again, the examples can be seen in action on the chapter 5 home page.

We'll start with a simple version of using a `Map` property to receive your data from the request. Here's the form markup from MapsDataTransferTest.jsp:

```
<s:textfield name="maidenNames.mary" label="Maiden Name"/>
<s:textfield name="maidenNames.jane" label="Maiden Name"/>
<s:textfield name="maidenNames.hellen" label="Maiden Name"/>

<s:textfield name="maidenNames['beth']" label="Maiden Name"/>
<s:textfield name="maidenNames['sharon']" label="Maiden Name"/>
<s:textfield name="maidenNames['martha']" label="Maiden Name"/>
```

These target the maidenNames property

Again, the main difference between this and the `List` property version is that we now need to specify a key value, a string in this case. We're using first names as keys to the incoming maiden names. As you can see, OGNL provides a couple of syntax options for specifying the key. First, you can use a simple, if somewhat misleading, property notation. Second, you can use a bracketed syntax that makes the fact that the property is a `Map` more evident. It doesn't matter which you use, though you'll probably find one or the other more flexible in certain situations. Just to prove that the syntax doesn't matter, all of our fields in this example will submit to the same property, a `Map` going by the name of `maidenNames`. Since we haven't specified a type for this `Map` with a type conversion properties file, all of the values will be converted into elements of type `String`. Similarly, our keys will also be treated as `String`s.

With the OGNL expressions in place in the form elements, we just need a property on the Java side to receive the data. In this case, we have a `Map`-backed property implemented on the `DataTransferTest.java` action. Here's the property that receives the data from the form:

```
private Map maidenNames ;

public Map getMaidenNames()
{
  return maidenNames;
}
```

```
public void setMaidenNames ( Map maidenNames ) {
  this.maidenNames=maidenNames;
}
```

Nothing special. If you want to see it in action, click Maps Data Transfer Test on the chapter 5 home page. You'll see that the MapsDataTransferTest.jsp page is successfully pulling some data out of the `maidenNames` property.

Now let's see an example where we specify the type for our `Map` elements. First, we just need to add a line to our DataTransferTest-conversion.properties file. Here's the line:

```
Element_myUsers=manning.utils.User
```

Again, this property simply specifies that the `myUsers` property, found on the `DataTransferTest` action, should be populated with elements of type `User`. How easy is that? Next, we need some form markup to submit our data to the `myUsers` property:

```
<s:textfield name="myUsers['chad'].username" label="Usernames"/>
<s:textfield name="myUsers['jimmy'].username" label="Usernames"/>
<s:textfield name="myUsers['elephant'].username" label="Usernames"/>

<s:textfield name="myUsers.chad.birthday" label="birthday"/>
<s:textfield name="myUsers.jimmy.birthday" label="birthday"/>
<s:textfield name="myUsers.elephant.birthday" label="birthday"/>
```

This form submits the data to a `Map` property named `myUsers`. Since we've specified a `User` element type for that map, we can use OGNL syntax to navigate down to properties, such as `birthday`, that we know will be present on the `Map` elements. Moreover, the conversion of our birthday string to a Java `Date` type will occur automatically even at this depth.

Just to make sure, we'll check the `myUsers` property on DataTransferTest.java to make sure it's still simple:

```
private Map myUsers ;

public Map getMyUsers()
{
  return myUsers;
}
public void setMyUsers ( Map myUsers ) {
  this.myUsers=myUsers;
}
```

That's pretty simple. Again, the Java code is still type unaware. You'll still have to cast those elements to `Users` if you access them in your action logic, but, as far as the OGNL references are concerned, both from the input-side form fields and from the result-side tags, you can take their type for granted. We should point out another cool feature while we're here. We've already noted that you don't have to initialize your `Maps` and `Lists`. The framework will create them for you. You might have also noticed, in this example, that the framework is creating the `User` objects for you as well. Basically, the framework will create all objects it needs as it tunnels down to the level of the `birthday` property on the `User`.

TIP The framework will automatically instantiate any intermediate properties in deep OGNL expressions if it finds them to be null when attempting to navigate to the target property. This ability to resolve null property access depends on the existence of a no-argument constructor for each property. So make sure that your classes have no-argument constructors.

In addition to specifying a type for the elements, you can specify a type for the key objects when using `Map` properties. Java `Maps` support all objects as keys. Just as with your values, OGNL will treat the name of your parameter as a string that it should attempt to convert to the type you specify. Let's say we want to use `Integers` as the keys for the entries in our `Map` property, perhaps so we can order the values. Let's make a version of `myUsers` that'll use `Integers` as keys. We'll call it `myOrderedUsers`. First, we add the following two lines to our DataTransferTest-conversion.properties file:

```
Key_myOrderedUsers=java.lang.Integer
Element_myOrderedUsers=manning.utils.User
```

These lines specify the key and element types for our `myOrderedUsers Map` property. As they say, we've been through most of this before, so we'll go fast. Here's the form markup that submits the data:

```
<s:textfield name="myOrderedUsers['1'].birthday" label="birthday"/>
<s:textfield name="myOrderedUsers['2'].birthday" label="birthday"/>
<s:textfield name="myOrderedUsers['3'].birthday" label="birthday"/>
```

Here we use a key value that is a valid `Integer`. If we didn't, we'd get a conversion error on the key when the framework tries to turn the string into the `Integer` type. This is no different from the `myUsers` example, except the keys are now `Integer` objects rather than `Strings`. Now let's look at the property that receives this data.

```
private Map myOrderedUsers ;

public Map getMyOrderedUsers()
{
  return myOrderedUsers;
}
public void setMyOrderedUsers ( Map myOrderedUsers ) {
  this.myOrderedUsers=myOrderedUsers;
}
```

As you can see, the Java property looks no different. It's still just a `Map`. The only important thing is that the name matches the name used in the OGNL expression. As we noted in the section on lists, if you're using Java 5 or higher, you can use generics to type your collections and maps, and Struts 2 will pick up on this during type conversion, making the properties file configuration unnecessary.

That does it for the built-in type conversions. We've seen a lot of ways to automatically transfer and convert your data. These methods provide a lot of flexibility. The variety of options can seem overwhelming at first. In the end, it's a simple process. You make a property to receive the data, then you write OGNL expressions that point to that property. Remember, you can consult the sample code for this chapter as a reference of the various OGNL-to-Java type mapping techniques.

The next section takes on an advanced topic. In case you want the framework to convert to some type that it doesn't support out of the box, you can write your own custom converters. It's a simple process, as you'll see.

5.4 Customizing type conversion

While the built-in type conversions are powerful and full featured, sometimes you might want to write your own type converter. You can, if you desire, specify a conversion logic for translating any string to any Java type. The only thing you need to do is create the string syntax and the Java class, then link them together with a converter. The possibilities are limitless. This is an advanced topic, but the implementation is simple and will provide insight that might help debugging even if you never need to write your own type converter.

In this section, we'll implement a trivial type converter that converts between strings and a simple `Circle` class. This means that we'll be able to specify a string syntax that represents a `Circle` object. The string syntax will represent the circle objects in the text-based HTTP world, and the `Circle` class will represent the same objects in the Java world. Our converter will automatically convert between the two just as the built-in converters handle changing the string `"123.4"` into a Java `Double`. The syntax you choose for your strings is entirely arbitrary. For our demonstration we'll specify a string syntax as shown on the right.

Syntax	Example
`C:rinteger`	`C:r10`

If a request parameter comes in with this syntax, the framework will automatically convert it to a `Circle` object. In this section, we'll see how to implement the converter code and how to tell the framework to use our converter.

5.4.1 Implementing a type converter

As we've explained, type conversion is a part of OGNL. Due to this, all type converters must implement the `ognl.TypeConverter` interface. Generally, OGNL type converters can convert between any two data types. In the web application domain, we have a narrower set of requirements. All conversions are made between Java types and HTTP strings. For instance, we convert from `Strings` to `Doubles`, and from `Doubles` to `Strings`. Or, in our custom case, we'll convert from `Strings` to `Circles`, and from `Circles` to `Strings`.

Taking advantage of this narrowing of the conversion use case, Struts 2 provides a convenience base class for developers to use when writing their own type converters. The `org.apache.struts2.util.StrutsTypeConverter` class is provided by the framework as a convenient extension point for custom type conversion. The following snippet lists the abstract methods of this class that you must implement:

```
public abstract Object convertFromString(Map context, String[] values,
  Class toClass);

public abstract String convertToString(Map context, Object o);
```

When you write a custom converter, as we'll do shortly, you merely extend this base class and fill in these two methods with your own logic. This is a straightforward process, as we've noted. The only thing that might not be intuitive in the preceding signatures is the fact that the string that comes into your conversion is actually an array of strings. This is because all request parameter values are actually arrays of string values. It's possible to write converters that can work with multiple values, but, for the purposes of this book, we'll stick to a simple case of a single parameter value.

5.4.2 Converting between Strings and Circles

The logic that we put in the conversion methods will largely consist of string parsing and object creation. It's not rocket science. As with many of the Struts 2 advanced features, the stroke of genius will be when you decide that a given use case can be handled elegantly by something like a custom type converter. The implementation itself will take much less brainpower. Listing 5.6 shows our manning.utils.CircleTypeConverter.java file.

Listing 5.6 The `CircleTypeConverter` provides custom type conversion.

```
public class CircleTypeConverter extends StrutsTypeConverter {

  public Object convertFromString(Map context, String[] values,
                          Class toClass) {

    String userString = values[0];
    Circle newCircle = parseCircle ( userString );
    return newCircle;

  }

  public String convertToString(Map context, Object o) {

    Circle circle = (Circle) o;
    String userString = "C:r" + circle.getRadius();
    return userString;

  }

  private Circle parseCircle( String userString )
                          throws TypeConversionException
  {
    Circle circle = null;
    int radiusIndex = userString.indexOf('r') + 1;

    if (!userString.startsWith( "C:r")  )
      throw new TypeConversionException ( "Invalid Syntax");
    int radius;
    try {
      radius = Integer.parseInt( userString.substring( radiusIndex ) );
    }catch ( NumberFormatException e ) {
      throw new TypeConversionException ( "Invalid Value for Radius"); }

    circle = new Circle();
    circle.setRadius( radius );
    return circle;

  }
}
```

Extends StrutsTypeConverter

Convert String to Circle

Convert Circle to String

You should focus on the first two methods. We include the `parseCircle()` method here to make sure you realize nothing sneaky is going on. The first method, `convertFromString()`, will be used to convert the request parameter into a `Circle` object. This is the same thing that has been happening behind the scenes when we've been taking advantage of the built-in type conversions that come with Struts 2 by default. All this method does is parse the string representation of a `Circle` and create an actual `Circle` object from it. So much for the mystique of data binding. Going back from a `Circle` object to a string is equally straightforward. We just take the bits of data from the `Circle` object and build the string according to the syntax we specified earlier.

5.4.3 Configuring the framework to use our converter

Now that we have our converter built, we have to let the framework know when and where it should be used. We have two choices here. We can configure our converter to be used local to a given action or globally. If we configure our converter local to an action, we just tell the framework to use the converter when handling a specific `Circle` property of a specific action. If we configure the converter to be used globally, it'll be used every time a `Circle` property is set or retrieved through OGNL anywhere in the application. Let's look at how each of these configurations is handled.

PROPERTY-SPECIFIC

The first choice is to specify that this converter should be used for converting a given property on a given action class. We've already worked with the configuration file used for configuring aspects of type conversion for a specific action. We used the ActionName-conversion.properties file when we specified types for our `Collection` property's elements in the earlier `Map` and `List` examples. Now we'll use the same file to specify our custom converter for a specific `Circle` property.

We've created a specific action to demonstrate the custom type conversion. This action is `manning.chapterFive.CustomConverterTest`. The action does little. Its most important characteristic for our purposes is that it exposes a JavaBeans property with type `Circle`, as seen in the following snippet:

```
private Circle circle;

public Circle getCircle() {
  return circle;
}

public void setCircle(Circle circle) {
  this.circle = circle;
}
```

This action has almost no execution logic. It functions almost exclusively as a carrier of our untransformed data. For our purposes, we're only interested in the type conversion that'll occur when a request parameter comes into the framework targeting this property. Here's the line from CustomConverterTest-conversion.properties that specifies our custom converter as the converter to use for this property:

```
circle=manning.utils.CircleTypeConverter
```

This simple line associates a property name, `Circle`, with a type converter. In this case, the type converter is our custom type converter. Now, when OGNL wants to set a value on our `Circle` property, it'll use our custom converter. Here's the form, from CustomConverterTest.jsp, that submits a request parameter targeting that property:

```
<s:form action="CustomConverterTest">
  <s:textfield name="circle" label='Circle'/>
  <s:submit/>
</s:form>
```

The name is an OGNL expression that'll target our property. Since the action object is on top of the `ValueStack` and the `Circle` property is a top-level property on that action, this OGNL expression is simple. Now let's try it out. Go to the Custom Converter Test link in the chapter 5 samples. Enter a valid `Circle` string in the form, as shown in figure 5.4.

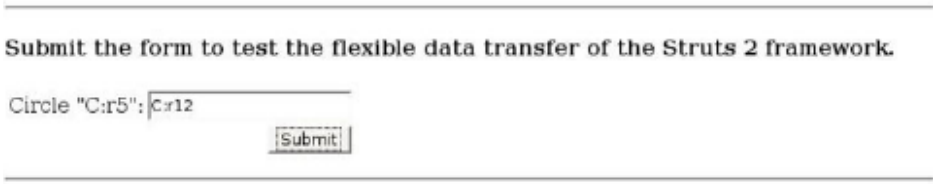

Figure 5.4 Submitting a string that our custom `CircleTypeConverter` will convert into a Java `Circle` object

When this form is submitted, this string will go in as a request parameter targeted at our `Circle` property. Since this property has our custom converter specified as its converter, this string will automatically be converted into a `Circle` object and set on the `Circle` property targeted by the parameter name. Figure 5.5 shows the result page, CustomConverterSuccess.jsp, that confirms that everything has worked according to plan.

Our result page verifies that the conversion has occurred by retrieving the first and last name from the `Circle` property, which holds the `Circle` object created by the converter. And, just for fun, the result page also tests the reverse conversion by printing the `Circle` property as a string again.

While we're at it, try to enter a string that doesn't meet our syntax requirements. If you do, you'll be automatically returned to the form input page with an error message informing you that the string you entered was invalid. This useful and powerful mechanism is just a part of the framework's conversion facilities. Tapping into it for your

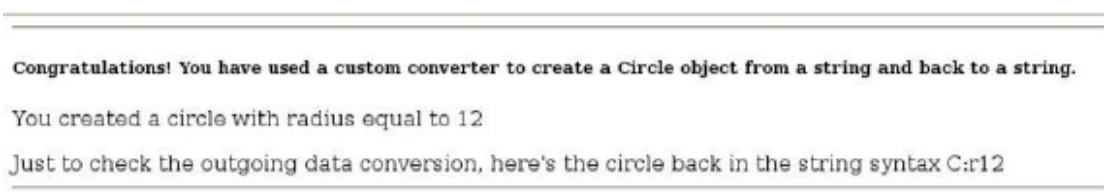

Figure 5.5 The result page from our custom conversion test pulls data from the `Circle` property to verify that the conversion worked.

custom type converters is easy. To access this functionality, we throw a `com.opensymphony.xwork2.util.TypeConversionException` when there's a problem with the conversion. In our case, this exception is thrown by our `parseCircle()` method. When we receive the input string value, we do some testing to make sure the string meets our syntax requirements for a valid representation of a `Circle`. If it doesn't, we throw the exception.

GLOBAL TYPE CONVERTERS

We just saw how to set up a type converter for use with a specific property of a specific action. We can also specify that our converter be used for all properties of type `Circle`. This can be quite useful. The process differs little from our previous example, and we'll go through it quickly without an example. The differences are so minute, you can alter the sample code yourself if you want some first-hand proof. Instead of using the ActionClassName-conversion.properties file to configure your type converter, use xwork-conversion.properties. As you can probably tell, this is just a global version of the conversion properties. To this file, add a line such as follows:

```
manning.utils.Circle=manning.utils.CircleTypeConverter
```

Now, our custom type converter will run every time OGNL sets or gets a value from a property of the `Circle` type. By the way, the xwork-conversion.properties file goes on the classpath, such as in WEB-INF/classes/ .

That wraps up custom type converters. As we promised, the implementation is easy. Now the challenge, as with custom interceptors, is finding something cool to do with them. One of the problems is that most of the converters you'll ever need have already been provided with the framework. Even if you never make a custom one, we're sure that knowing how they work will help in your daily development chores. If you do come up with a rad type converter, though, don't keep it a secret. Go to the Struts 2 community and let them know. We're all anxious to benefit from your labor!

5.5 Summary

That does it for our treatment of the Struts 2 data transfer and type conversion mechanisms. While you can get away with minimal awareness of much of the information in this chapter, trying to do so will only leave you frustrated and less productive in the long run, and even in the short run. Let's review some of the things we learned about how the framework moves data from one end of the request processing to the other, all while transparently managing a wide range of type conversions.

The Object-Graph Navigation Language (OGNL) is tightly integrated into Struts 2 to provide support for data transfer and type conversion. OGNL provides an expression language that allows developers to map form fields to Java-side properties, and it also provides type converters that automatically convert from the strings of the request parameters to the Java types of your properties. The expression language and type converters also work on the other end of the framework when Struts 2 tags in the view-layer pages, such as JSPs, pull data out of these Java properties to dynamically render the view.

We conducted an extensive review of the built-in type converters that come with the framework. We saw that they pretty much support all primitives and common wrapper types of the Java language. We also saw that they support a flexible and rich set of conversions to and from `Array`s and `Collection`s. And if that's not enough, you can always build your own custom type converters. Thanks to a convenient base class provided by the framework, implementing a custom converter is easy.

As we promised at the onset, this chapter started a two-part introduction to OGNL. This chapter's efforts focused more on the role that OGNL plays in the framework, providing a binding between string-based data of the HTTP realm and the strict type of the Java realm. We showed enough OGNL expression language details to make full use of the built-in type conversions to such complex property types as `Map`s and `List`s.

Now, we're ready to head to the result side of the framework and see how data is pulled from the model, via tag libraries, and rendered in the view. The tags tend to take more advantage of the full expression language. Accordingly, in the next few chapters, which deal specifically with the tags, we'll spend a lot more time on the OGNL expression language. On to chapter 6!