```
    public String toString() {
        <omitted>
    }
}
```

Note that whether a DAO is implemented in JDBC or Hibernate, it's used the same way in a service bean, as the service depends on the DAO interface rather than the implementation. Because of this transparency, we could have just stopped here without showing how the service layer of SOBA performs bill payment tasks with `HibernateBillPaymentDao` working behind the scene. However, this actually is a good place for us to get a little deeper on Spring validation framework by picking up where we left off with the bill payment use scenario demonstrated in Section 2.7. This is the subject of the next section.

5.6 SPRING DATA VALIDATION FRAMEWORK

To get a complete picture about how Spring validation works with the bill pay service, let's begin with the programmatic logic flow with the bill pay service implemented in SOBA.

5.6.1 Programmatic Logic Flow with the Bill Pay Service

Since we have covered so much about how Spring MVC works, let us capture the essence of the programmatic logic flow associated with the bill pay service by looking at what classes get involved at each layer. Table 5.2 lists the Java classes associated with this service. It should be clear what each Java class does based on its name and in which layer it is placed.

Table 5.2 Java classes at each layer for the bill pay service

Layer	Java Classes
Domain object	BillPayment.java
DAO	BillPaymentDao.java HibernateBillPayemntDao.java
Service	BillPayManager.java/SimpleBillPayManager.java CreateBillPayValidator.java
Web	CreateBillPayFormController.java CreateBillPaySuccessController.java
View	createBillPayForm.jsp createBillPaySuccess.jsp

Now let us take a look at how the bill pay service flows programmatically based on the Java classes associated with it as listed in Table 5.2. This service is initiated when a user clicks on the Bill Payment tab on the home page as shown in Figure 2.18. That home page was generated with the `activityList.jsp`, which has an embedded link as shown below:

```
<a href = "<c:url value = "createBillPayForm/customerId/${customerId}/
accountId/${accountId}"/>">Bill Payment</a>
```

Note the url value hard-coded in the above HTML element. It is similar to the RequestMapping we introduced in Listing 4.5 `CreateCustomerFormController.java`. Not surprisingly, it is mapped to the `CreateBillPayFormController.java` class as shown in Listing 5.11. When control is directed to this class, the `setupForm` method is executed first, which creates a `billPayment` object with some of the attributes pre-populated. Most of the pre-populated attributes here are purely for our convenience except the `fromAccount` attribute so that we don't have to type them every time when we test this service. Then, the `setupForm` method returns control to the `createBillPayForm`, namely, the `createBillPayment.jsp`. The form is then presented to the user for entering all required information for a bill payment transaction. Refer back to Figure 2.19 for an actual instance of this form.

Listing 5.11 CreateBillPayFormController.java

```
package com.perfmath.spring.soba.web;
import java.sql.Timestamp;
import java.util.List;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Controller;
import org.springframework.ui.Model;
import org.springframework.validation.BindingResult;
import org.springframework.web.bind.annotation.ModelAttribute;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RequestMethod;
import org.springframework.web.bind.annotation.RequestParam;
import org.springframework.web.bind.annotation.SessionAttributes;
import org.springframework.web.bind.support.SessionStatus;

import com.perfmath.spring.soba.model.domain.BillPayment;
import com.perfmath.spring.soba.service.BillPayManager;
import com.perfmath.spring.soba.service.CreateBillPayValidator;
import com.perfmath.spring.soba.util.RandomID;

@Controller
@RequestMapping("/createBillPayForm/customerId/{customerId}/accountId/{acco
untId}")
@SessionAttributes("billPay")
public class CreateBillPayFormController {

    private CreateBillPayValidator validator;
    private BillPayManager billPayManager;
    @Autowired
    public CreateBillPayFormController(BillPayManager billPayManager,
            CreateBillPayValidator validator) {
        this.billPayManager = billPayManager;
        this.validator = validator;
```

```
    }
    @RequestMapping(method = RequestMethod.GET)
    public String setupForm(
            @PathVariable String accountId,
            Model model) {
        BillPayment billPayment = new BillPayment();
        billPayment.setFromAccount(accountId);
        billPayment.setDescription("bill pay test");
        billPayment.setAddress("One Way");
        billPayment.setCity("Any City");
        billPayment.setState("CA");
        billPayment.setZipcode("95999");
        model.addAttribute("billPayment", billPayment);
        return "createBillPayForm";
    }
    @RequestMapping(method = RequestMethod.POST)
    public String submitForm(
            @PathVariable String customerId,
            @ModelAttribute("billPayment") BillPayment billPayment,
            BindingResult result, SessionStatus status) {
    validator.validate(billPayment, result);

     if (result.hasErrors()) {
            return "createBillPayForm";
        } else {
            billPayManager.storeBillPayment(billPayment);
            status.setComplete();
            return "redirect:/createBillPaySuccess/customerId/" + customerId;
     }
    }
}
```

After a user fills in the bill payment form and hits Submit button, control is redirected back to the class `CreateBillPayFormController.java`, and the `submitForm` is initiated. As is seen from Listing 5.11, the `submitForm` method validates the `billPayment` object using the `validate` method of its `validator`. This is where *validation* gets invoked, as is discussed next. If validation is successful without errors, control is directed to the `CreateBillPaySuccessController`, which presents the responses back to the user via the `createBillPaySuccessForm.jsp` as listed in Table 5.2. If errors occurred during validation, control would be directed back to the bill pay form to display the errors to the user so that the user can correct the errors and resubmit the bill pay again.

Next, we focus on understanding how Spring Validation Interface works in the specific context of this bill pay service example.

5.6.2 Spring Validation Interface

Refer back to Figure 2.19, which shows an actual instance of a bill pay form. Since the user's own account ID has been pre-populated, we don't have to worry about it at all. Of course, in reality, a user might have an option to decide from which account the fund should be used to pay the bill, but that is not very important for our example here. Our concern is how to validate the data entered on this form by the user.

We are particularly concerned about how the bill pay amount is validated. As is shown in Listing 5.12 `CreateBillPayValidator.java`, we use a `Validator` interface in Spring's `validation` package, which is very basic and usable. This `BillPayment` validator has two methods: `supports` (…) and `validate` (…). The `supports` method checks whether the passed-in type supports validation, whereas the `validate` method does the actual validation if this validator supports validation. The `validate` method uses a method of `rejectIfEmptyOrWhitespace (errors, "amount", "required.amount", "amount is required.")` on the Spring `ValidationUtils` class to validate the attribute `amount` of type `double`. This entry is rejected if a user enters an empty or whitespace string or an invalid item in the `amount` field on the `createBillPayForm` form defined in `createBillPayForm.jsp` file. If a type mismatch occurs, an error message of "`invalid data`" would be displayed near the `amount` field entry, according to the `typeMismatch` entry defined in the `messages.properteis` file located at the root class path of SOBA.

Of course, you can introduce additional validation based on your business context, after form data format is validated. For example, the `validate` method for this example further validates that the bill pay amount must be larger than zero, after a user enters an amount that is syntactically correct. If a less than zero bill pay amount is entered, it would pass the `rejectIfEmptyOrWhitespace` validation, but not the following validation as shown in Listing 5.12, which returns control to the bill pay form with an error message of "`bill pay amount must be > 0`" displayed along with the amount field:

```
if (billPayment.getAmount() <= 0.0) {
    errors.rejectValue("amount", "invalid.billPayAmount",
    "bill pay amount must be > 0");
}
```

Table 5.3 lists some interesting test cases about how this validation works. As you can see, this is a very simple, yet very powerful validation framework. You can consult http://static.springsource.org/spring/docs/3.0.x/javadoc-api/org/springframework/validation/ to learn more about how this framework works and what other Spring validation APIs are available to meet your specific needs.

Listing 5.12 CreateBillPayValidator.java

```
package com.perfmath.spring.soba.service;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Component;
import org.springframework.validation.ValidationUtils;
import org.springframework.validation.Validator;
import org.springframework.validation.Errors;
```

```
import org.apache.commons.logging.Log;
import org.apache.commons.logging.LogFactory;
import com.perfmath.spring.soba.model.domain.BillPayment;
import com.perfmath.spring.soba.util.RandomID;
@Component
public class CreateBillPayValidator implements Validator {
    /** Logger for this class and subclasses */
    protected final Log logger = LogFactory.getLog(getClass());
    public boolean supports(Class clazz) {
        return BillPayment.class.isAssignableFrom(clazz);
    }
    public void validate(Object target, Errors errors) {
        BillPayment billPayment = (BillPayment) target;
            ValidationUtils.rejectIfEmptyOrWhitespace(errors, "amount",
                    "required.amount", "amount is required.");
            if (billPayment.getAmount() <= 0.0) {
                errors.rejectValue ("amount", "invalid.billPayAmount",
                "bill pay amount must be > 0");
            }
            billPayment.setId(Long.parseLong(new RandomID(10).getId()));
            billPayment.setScheduleDate(new Timestamp(System.currentTimeMillis()));
            billPayment.setSendDate(new Timestamp(System.currentTimeMillis()));
            billPayment.setStatus("complete");
    }
}
```

Table 5.3 BillPayment form data validation on the *amount* attribute

Input	Comment	Result
1.a0	Letter/number mixed	Error: invalid data
abc	A string	Error: invalid data
1	An integer	Ok ($1.0 was paid)
empty	An empty string	Error: invalid data/amount is required
" "	A two-space string	Error: invalid data/amount is required
1.0 1	A space b/t "0" and "1"	Ok ($1.01 was paid with " " ignored)
1 0.1	A space b/t "1" and "0"	Ok ($10.1 was paid with " " ignored)
10.0.1	Two dots	Error: invalid data
0.0	Zero amount	Error: bill pay amount must be > 0
-10.0	negative amount	Error: bill pay amount must be > 0

5.6.3 JSR-303 Bean Validation

JSR-303 bean validation is a spec about validating domain objects using Java annotations under the package of `javax.validation.constraints`. For example, with the Payment domain object shown in Listing 5.10, we could have added the following annotations of `@NotNull` and `@Size` to add JSR-303 based validation to limit the length of a required attribute of `description`:

```
…
Import javax.validation.constraints.NotNullNull;
Import javax.validation.constraints.Size;
…
@NotNull
@Size (min=2, max=50)
Private String description;
…
```

Both Spring and Hibernate validators support JSR-303 Bean Validation. However, make a careful decision when choosing which validation mechanism to use with your application. Your application will be less performing and scalable if you have double or even triple validating implemented at all layers for the same validation. Also, keep in mind that it's hard to achieve the same finer granularity with PSR-303 as with Spring validation interface. For example, it might be challenging to use JSR-303 to specify that an entry like "10.0.1" is an invalid input for the `amount` attribute of the bill payment domain object as shown in Table 5.3. Therefore, most of the time, Spring validation interface is a cleaner, more efficient validation mechanism. In a word, try to avoid using JSR-303 bean validation unless you can't do without it.

5.7 SUMMARY

In this chapter, we explained how Spring Data Access Framework supports JDBC and Hibernate data access methods. Real SOBA code examples were used to demonstrate the key concepts and technologies associated with JDBC and Hibernate. We also covered the Spring validation interface using the bill pay service. Since SOBA is a fully functioning, integrated sample, you can explore more JDBC and Hibernate features as well as Spring data validations using SOBA as your experimental platform.

The next chapter covers how RESTful Web Services is supported by Spring and applied to SOBA. This is an interesting subject, since RESTful Web Services has become more and more popular for building enterprise applications. Since the JDBC and Hibernate parts are re-usable, they will not be repeated.