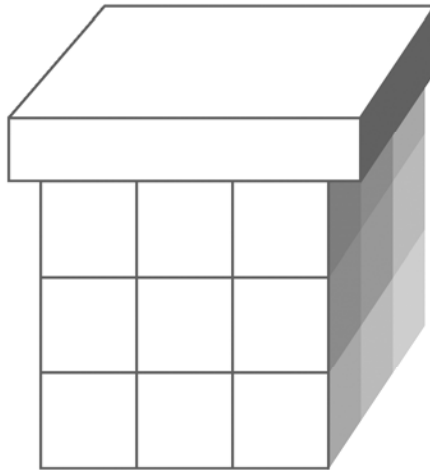


# CHAPTER 6

## VALIDATION



## OBJECTIVES

*After completing “Validation,” you will be able to:*

- **Implement validators to assert constraints on bean state.**
- **Invoke validators on target objects and graphs of objects, and organize their error reporting using centralized error compilations.**
- **Define message keys to take advantage of resource localization.**
- **Delegate from one validator to another to match the logical decoupling in the target model.**
  - Use nested paths in error-reporting objects to correctly support such “nested validators.”

## We All Seek Validation

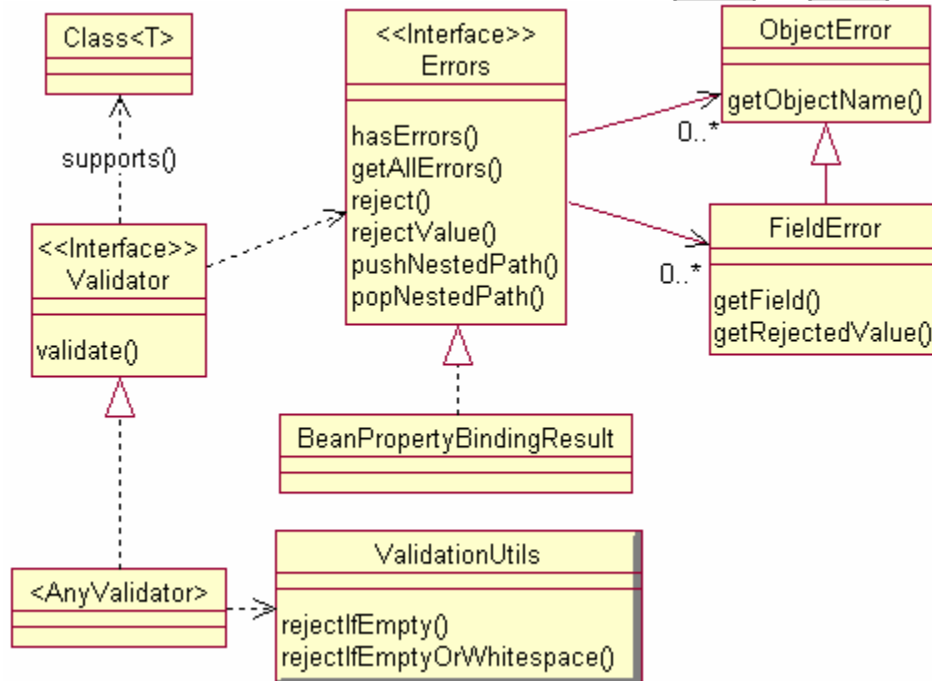
---

- Most applications – very nearly all web applications – have to consider the problem of validating user input.
- Spring therefore makes validation fundamental:
  - It is part of the core module – Spring’s authors understand that data validation, is at least potentially, business logic. (At the very least, we can say it’s not a web-specific function, and so must be decoupled from any web artifacts.)
  - It is integrated tightly with more general exception and error-handling mechanisms.
  - It is entirely flexible, as any class can be a validator for any other class – or for itself, though that’s not generally the idea.

Evaluation Only

## Spring Validation

- Spring's model for validation is expressed in the package `org.springframework.validation`:



- Any class can act as a validator; it must only implement the **Validator** interface, which has just two methods.
- Validation is an error-reporting process, and there is a common structure for gathering errors over some error-handled duration: this is the **Errors** interface.
- A class utility **ValidationUtils** serves two purposes: it can be used to launch validation on a particular object, and validators can use it to simplify reporting of the most common validation errors, which are those having to do with missing or empty fields.

## The Validator and Errors Interfaces

---

- A validator is just a class that implements a **validate** function and can announce what class or classes it supports as targets:

```
public interface Validator
{
    public boolean supports (Class);
    public void validate (Object, Errors);
}
```

- Given a target object, a validator is responsible for reporting any and all errors through the given errors object.

– Following is a partial listing:

```
public interface Errors
{
    public String getObjectName ();
    public void reject (String code);
    public void rejectValue
        (String field, String code);

    public boolean hasErrors ();
    public List getAllErrors ();
    public List getGlobalErrors ();
    public List getFieldErrors ();
    public List getFieldErrors (String field);
}
```

- So the model allows for lists of errors “globally” – meaning for the whole target object – and for each field on the object.

## The ValidationUtils Class

---

- **ValidationUtils** is a gathering point for utility methods serving different functions for different types of callers.

```
public abstract class ValidationUtils
{
    public static void invokeValidator
        (Validator, Object, Errors);

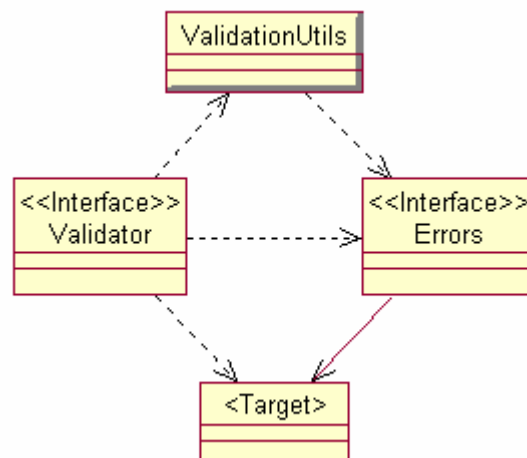
    public static void rejectIfEmpty
        (Errors, String field, String code);
    public static void rejectIfEmptyOrWhitespace
        (Errors, String field, String code);
}
```

- Any party wanting to trigger validation can do so with a call to **invokeValidator**.
  - The method won't conjure up the validator or errors object for you, but it will check that the validator supports the target object before calling **validate**, and do appropriate logging and error reporting.
- Validators themselves can use the **rejectIfEmpty** and **rejectIfEmptyOrWhitespace** to record these common error cases easily.
  - Many validators are essentially scripts of calls to these methods.
  - Not all overloads of these two methods are shown; there are others, analogous to the **Errors** methods that were omitted on the previous page, that provide default message values and/or arguments for replaceable parameters in the message.

## Context for the Errors Object

- You may have noticed the absence of a parameter identifying the object of validation in several of the methods we've just seen:
  - **Errors.rejectValue** accepts a field name, but no object reference.
  - **ValidationUtils.rejectIfEmpty** and related methods are similar: they take field names, but nowhere is a reference to the object being validated passed to this method.
- If these methods aren't given any reference to the object that they must test, how can they carry out their work?
- This wouldn't be obvious, since we've yet to see how an **Errors** object is instantiated, but the object reference is given to the **Errors** object initially.
- So the validator, the **Errors** object, and **ValidationUtils** all can see the target object in some way:

- An **Errors** object holds this context information and uses it in error reporting.
- **ValidationUtils** takes advantage of this connectivity, since it receives a reference to the **Errors** object as an argument to its methods.



## An Ellipsoid Validator

**EXAMPLE**

- In **Examples/Ellipsoid/Step5** we see **EllipsoidValidator**, which assures that all values on an **Ellipsoid** are valid:

```
public class EllipsoidValidator
    implements Validator
{
    public boolean supports (Class cls)
    {
        return cls.equals (Ellipsoid.class);
    }

    public void validate
        (Object object, Errors errors)
    {
        Ellipsoid target = (Ellipsoid) object;
        if (target.getA () <= 0)
            errors.rejectValue ("a",
                "validation.Ellipsoid.a",
                "Semi-axis A must be a postive number.");
        if (target.getB () <= 0)
            errors.rejectValue ("b",
                "validation.Ellipsoid.b",
                "Semi-axis B must be a postive number.");
        if (target.getC () <= 0)
            errors.rejectValue ("c",
                "validation.Ellipsoid.c",
                "Semi-axis C must be a postive number.");
    }
}
```

- We'll give this validator a spin in a moment.

## The MessageSource Interface

---

- Though Spring includes robust support for internationalization, most developers don't encounter it until they try to report errors.
- Validation and other error mechanisms plug into the i18n support in Spring, and this makes it advisable to get at least a basic string table in place for your application.
- Most error-reporting functions are overloaded.
  - We've seen the ones that only take an error code.
  - For each of these there are often two others: one that adds a default value in case the error code is not resolved, and another that adds a default value and an array of arguments for replaceable parameters that may be in the error string.
- To convert message codes (and arguments) into messages, call a method on some implementation of **MessageSource**, from package **org.springframework.context**:

```
public interface MessageSource
{
    public String getMessage
        (String code, Object[] args, Locale locale);
    public String getMessage (String code,
        Object[] args, String default, Locale locale);
    public String getMessage
        (MessageSourceResolvable resolvable,
        Locale locale);
}
```

## Deriving a MessageSource

---

- A **MessageSource** implementation will represent one or more **message bundles**.
  - These are declared in the usual way, according to Java i18n standards, and resolved by locale at runtime.
- Any **ApplicationContext** is automatically a **MessageSource**.
  - Standard implementations automatically look for a bean of a specific name, “messageSource”, and make that their delegate.
- Ordinary **BeanFactory** implementations don’t have it so easy.
  - An application must instantiate its own message source – preferably by way of Spring IoC – and populate it with message keys and strings.
  - The likely candidate here is **ResourceBundleMessageSource**, from **org.springframework.context.support**.
  - This can be primed with the base name for a message bundle, which it will load and represent through the **MessageSource** methods.
- For our upcoming lab, we’ll do without i18n support.
  - Notice that the error reporting in **EllipsoidValidator** supplies default values throughout; you’ll do the same thing with the **OrderValidator** you’re about to build.

## Putting Validators to Work

---

- Standalone Spring applications can instantiate the appropriate objects by hand, and call **ValidationUtils.invokeValidator**.
  - Create the validator of choice.
  - **BeanPropertyBindingResult** is a handy **Errors** implementation – find this right here in package **org.springframework.validation**.
- Then report errors in any appropriate manner.
  - From the **Errors** object, you can see lists of all errors, all global errors, all field-oriented errors, or errors for a particular field.
  - Each list will hold objects of type **ObjectErrors** – methods specific to fields will return objects downcastable to **FieldErrors**.
  - These have **toString** methods that dump all known values to a log, and also methods to provide the individual values that went into the error report, to be formatted into more user-friendly output.
- Web applications (those lucky sods) enjoy more infrastructure here as well.
  - Typically, errors will be reported based on the **binding** of request parameters to a **command object**, as well as on subsequent validation of the object.
  - Default and configurable error handlers will log the errors.
  - Custom tags in JSPs will report the errors to the web user.

## Validating Ellipsoids

**EXAMPLE**

- In **Examples/Ellipsoid/Step5**, the **Controller** class has been enhanced to perform validation prior to reporting on the requested bean:

```
public static void reportOn (Ellipsoid subject)
{
    final Validator validator =
        new EllipsoidValidator ();

    Errors errors = new BeanPropertyBindingResult
        (subject, "ellipsoid");
    ValidationUtils.invokeValidator
        (validator, subject, errors);
    if (errors.hasErrors ())
    {
        System.out.println ("Ellipsoid is invalid:");
        for (Object error : errors.getAllErrors ())
            System.out.println (error);
        return;
    }

    System.out.println
        ("Three-dimensional ellipsoid -- properties:");
    ...
}
```

- **Ellipsoids.xml** now has three new beans, none of which are valid.

## Validating Ellipsoids

**EXAMPLE**

- **Build and test – against one of the good beans first:**

**ant**

**run cool**

Three-dimensional ellipsoid -- properties:

Semi-axis A: 4.0

Semi-axis B: 4.0

Semi-axis C: 4.0

Volume: 268.082573106329

Type: Sphere

Definition:

Where  $A = B = C$ , offering perfect rotational symmetry in all three dimensions.

- **Now try the “unacceptable” bean, whose values are (0, 1, 1):**

**run unacceptable**

Ellipsoid is invalid:

Field error in object 'ellipsoid' on field 'a':

rejected value [0.0]; codes

[validation.Ellipsoid.a.ellipsoid.a,

validation.Ellipsoid.a.a,

validation.Ellipsoid.a.double,

validation.Ellipsoid.a]; arguments []; default

message [Semi-axis A must be a postive number.]

- **Try the other two as well, which are:**

– **notCool** (-4, -4, -4)

– **clearlyInferior** (2, 3, -4)

## Single-Order Validation

LAB 6A

**Suggested time: 15 minutes**

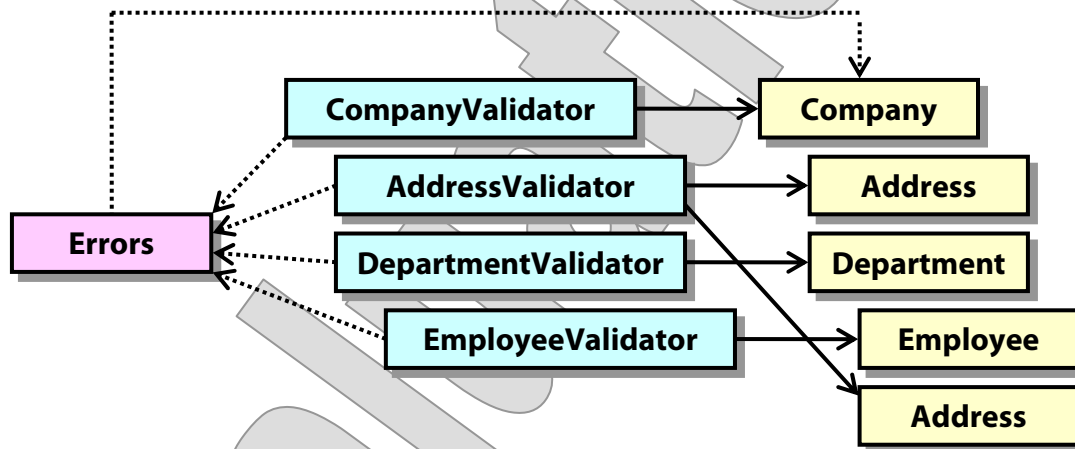
In this lab you will implement a validator for the **Order** bean and apply it to sales data prior to fulfilling orders.

Detailed instructions are found at the end of the chapter.

Evaluation  
Only

## Nesting Validators

- It's natural to define a validator for each domain class.
- Where one domain class relies on another as a collaborator, its validator will typically delegate to the validator for that collaborator – and so on from the head of an object graph to all the nodes in the graph.



- There is one complication, concerning the **Errors** object:
  - On one hand, we want a single compilation of all errors encountered anywhere in the process; we wouldn't want a distinct **Errors** object for each node in the object graph.
  - On the other hand, we know that both the validators and **Errors** are contextual: each “knows” what object it was meant to work with – and they must have the same idea which object that is!
  - This threatens to frustrate good decoupling of validation logic, because lower-level validators will ask the **Errors** object to, for instance, reject a named field – and the **Errors** object won't be able to find that field!

## Nested Property Paths

---

- We've discussed Spring's support for **compound property** names, of forms such as **a.b.c** and **a[0].b**.
- As we get deeper into working with command objects, and especially with validation, we see more of Spring's support for what are correctly called **nested properties**.
- Nested properties are essential to managing complex data models; the tension between the "global" context of the **Errors** object and the "local" context necessary for a given validator is just one example of this need.
- The solution here is to allow validators to **push** and **pop** fragments of compound property paths into the **Errors** object.

```
public void pushNestedPath (String partialPath);  
public void popNestedPath ();
```

- This allows the object to maintain **two contexts**: the top-level object it was given originally, and a reference to the object currently under scrutiny.
- For our validators so far, these have always been the same object.
- When drilling down through a more complex object graph, often they will not be the same object.
- The rest of the field-sensitive methods on **Errors**, including **rejectValue**, will treat field names passed as arguments as being **relative** to the current nested path.

## Wholesale Validation

LAB 6B

**Suggested time: 30 minutes**

In this lab you will complete the validation layer for the Wholesale application. You already have an **OrderValidator** ready to roll. You will build a higher-level validator for **ListOfBatches**, that delegates to the **OrderValidator** for each of its sub-parts. You'll put this new validator in play from the **Controller** class, simplifying the logic there.

Detailed instructions are found at the end of the chapter.

Evaluation Only

## SUMMARY

- **Data validation is a general concern for most applications, and Spring makes it straightforward to support validation logic in any tier, including in the business tier with no ties to web or other presentation decisions.**
- **Internationalization, including for error messages, is practicable for standalone applications, but much easier for web applications, thanks to some helpful code in the Web module.**
- **The errors and utilities objects use nested paths to resolve the tension between their global and local responsibilities, when nested validators are used to check a complex graph of objects.**

# Single-Order Validation

**LAB 6A**

In this lab you will implement a validator for the **Order** bean and apply it to sales data prior to fulfilling orders.

<b>Lab workspace:</b>	<b>Labs/Lab6A</b>
<b>Backup of starter code:</b>	<b>Examples/Wholesale/Step6</b>
<b>Answer folder(s):</b>	<b>Examples/Wholesale/Step7</b>
<b>Files:</b>	<b>src/Controller.java</b> <b>src/cc/sales/OrderValidator.java</b> (to be created)

**Instructions:**

1. Create a new class **cc.sales.OrderValidator** that implements the **Validator** interface.
2. Implement **supports** in the usual way; see the **EllipsoidValidator** example.
3. Define your implementation of **validate**, with parameters **object** and **errors** as shown in the example.
4. Start your implementation of **validate** by downcasting the first argument to type **Order**, and assigning that to a local variable **target**.
5. The product name is required, so call **ValidationUtils.rejectIfEmptyOrWhitespace**, passing four arguments: **errors**, the field name “product”, a message key of your own choosing (we won’t get to use it in this lab anyway!), and a default error message.
6. If a call to **target.getPrice** returns a number less than or equal to zero, call **errors.rejectValue**, passing the field name “price”, another dummy message key, and the “real” (default) error message.
7. Do the same thing with **target.getQuantity**, which must be positive.
8. Build at this point to check your coding so far.
9. Open **Controller.java** and add code before the call to **engine.fulfill**. First, initialize an **OrderValidator** called **orderValidator**.
10. Create an outer loop over the **List<Order>** objects in the list returned by **lob.getBatches**.
11. Create an inner loop over the **Order** objects themselves.

**Single-Order Validation****LAB 6A**

12. In this inner loop, add code similar to what you saw in the **EllipsoidValidator** example: initialize an **Errors** object; call **invokeValidator**, and check if **errors.hasErrors**. If so, loop through them and produce them to the console before aborting the process.

Thus only if there are no validity errors will **engine.fulfill** be called.

13. Build and test, and, lo and behold, there is a validity error!

**ant**

**run**

Validation failed:

Field error in object 'order' on field 'quantity':

rejected value [0]; codes

[validation.Order.quantity.order.quantity,validation.Order.quantity.qua

ntity,validation.Order.quantity.int,validation.Order.quantity];

arguments []; default message [Quantity must be a positive integer.]

14. But, which **Order** object was invalid? Since your **Errors** object is created with the individual **Order** instance as its scope, the error messages don't carry that contextual information. You could find this out with some good, old-fashioned debugging or diagnostics, but in the upcoming lab you're going to improve and broaden your validation logic anyway, and be able to identify the culprit with minimal fuss.

So, let's leave it be for now ...

# Wholesale Validation

**LAB 6B**

In this lab you will complete the validation layer for the Wholesale application. You already have an **OrderValidator** ready to roll. You will build a higher-level validator for **ListOfBatches**, that delegates to the **OrderValidator** for each of its sub-parts. You'll put this new validator in play from the **Controller** class, simplifying the logic there.

<b>Lab workspace:</b>	<b>Labs/Lab6B</b>
<b>Backup of starter code:</b>	<b>Examples/Wholesale/Step7</b>
<b>Answer folder(s):</b>	<b>Examples/Wholesale/Step8</b>
<b>Files:</b>	<b>src/cc/sales/ListOfBatchesValidator.java</b> (to be created) <b>src/Controller.java</b>

**Instructions:**

1. Create a new class **cc.sales.ListOfBatchesValidator** that implements the **Validator** interface. You can use **OrderValidator** as a template; the **supports** method is entirely boilerplate.
2. To implement **validate**, first downcast the **Object** parameter to a **ListOfBatches** – call this **target**.
3. Call **ValidationUtils.rejectIfEmpty**, passing your errors object, the field name “batches”, the error code “validation.ListOfBatches.batches”, and a default message saying there must be at least one batch of orders.
4. Instantiate an **OrderValidator** – call it **orderValidator** – which you'll use repeatedly in the loop you're about to write.
5. Now iterate through the batches on the target object, and all the orders on each batch. For each order, make a call to **orderValidator.validate**, passing along the errors object to gather all object and field errors.
6. In the **main** method of **Controller.java**, you can now simplify the code for validating the list of batches, by invoking the **ListOfBatchesValidator** once on the whole **lob** target. The validator now does all the looping and delegating to **OrderValidator**, so you don't have to duplicate that logic here. Just create your **Errors** object, invoke the validator, and report any errors.

**Wholesale Validation****LAB 6B**

7. Build and test your code.

**ant**

**run**

```
Exception in thread "main"
org.springframework.beans.NotReadablePropertyException: Invalid
property 'product' of bean class [cc.sales.ListOfBatches]: Bean
property 'product' is not readable or has an invalid getter method:
Does the return type of the getter match the parameter type of the
setter?
etc.
```

8. This message, and then also the exception stack trace that isn't shown above, make it clear that the error originates from the **OrderValidator**. Though you haven't yet seen this class work, assume for the moment that it is correctly implemented. What might be going on that would make this class function incorrectly?
9. Well, the preconditions for the call to **OrderValidator.validate** are basically the target object – which is what it is, good or bad – and the errors object. And a closer look at the error message shows that we're trying to check a property “product” on an object of type **ListOfBatches**. Something's out of synch.

Here we're seeing the need for nested properties. The **Errors** object keeps a reference to the target object, which is the **ListOfBatches**. But the **OrderValidator** doesn't even know about that class; it just wants to validate a single **Order**, and it assumes that the **Errors** object given to it knows what it's doing. Right now it doesn't – but it could, if we'd just tell it!

10. Before calling **OrderValidator.validate**, call **pushNestedPath** on the errors object, passing a string of the form “batches[*bat*][*ord*]” where *bat* and *ord* are the index values for the position in the nested loop over batches and orders. This way, when the **OrderValidator** calls **rejectIfEmptyOrWhitespace**, the errors object will know to translate “product” to “batches[*bat*][*ord*].product” – which is the appropriate property name, since the embedded target object is a **ListOfBatches**.

After the call to **OrderValidator.validate**, call **popNestedPath** on the errors object.

**Wholesale Validation****LAB 6B**

11. If you test now, you should get a clear error message that tells you exactly where that bad datum is that we've been living with since the previous lab:

```
ant
```

```
run
```

```
Validation failed:
```

```
Field error in object 'listOfBatches' on field  
'batches[2][2].quantity': rejected value [0]; codes  
[validation.Order.quantity.listOfBatches.batches[2][2].quantity,validat  
ion.Order.quantity.listOfBatches.batches[2].quantity,validation.Order.  
quantity.listOfBatches.batches.quantity,validation.Order.quantity.batch  
es[2][2].quantity,validation.Order.quantity.batches[2].quantity,validat  
ion.Order.quantity.batches.quantity,validation.Order.quantity.quantity,  
validation.Order.quantity.int,validation.Order.quantity]; arguments [];  
default message [Quantity must be a positive integer.]
```

12. Batch 2, order 2 translates to the third line of **MixedBag.txt**, and sure enough:

```
Winesap           56.50   2  
Vanity            1200.00  1  
Candles         2.25   0
```

13. Fix the error in the file – the correct quantity of candles is 40 – and test again, and everything should run smoothly.