

December 8, 2000

# Inxar **affirm** Developer Tutorial

Paul Johnston

December 8, 2000

## Contents

### Abstract

`org.inxar.affirm` is a set of Java interfaces and XML-based reference implementation designed to make server-side input validation straightforward. While input validation is most relevant to web-based applications, it is useful under many circumstances. The fundamental scenario is that you have a collection of data that needs to be systematically checked to ensure that it meets some predetermined criteria. The act of validating the input results either in a validated datum *or* some indication of failure and the optional association to some feedback error message.

## 1 Introduction

**affirm** uses a simple yet versatile design model. We will use the following scenario to illustrate the concepts:

You are designing a relatively simple web-based application which is designed to be used by automotive shop mechanics such that they can re-order spark plugs and the like at a moments' notice over the web. The application will need a few static HTML pages, some dynamically-generated HTML pages, and a database. Today you are writing the section of the application that allows your mechanics to enter spark plug ordering data in an HTML form, evaluates the data on the server, and store that information in the database.

### 1.1 Define your input

The first step is to ask "What data will I want to be validating?". **affirm** abstracts all input in the form of key value pairs, identical in essence to a `Hashtable`. Thus, the first practical question is "what are the keys to my data?". In this example, we will assume the `<FORM>` has three inputs:

```
<input type="text" name="partno"/>
<input type="text" name="qty"/>
<input type="text" name="comment"/>
```

Therefore, we will be validating the information associated with the keys **partno**, **qty**, **comments**. Now we need to ask “What is the valid domain for each value?”. After some analysis it is decided that **partno** is a string with length at least 6 characters and not more than 15 characters. **qty** is a number that is at least 1 but not more than 10,000. **comment** is a string of length between 0 and 4096 characters.

## 1.2 Define your error messages

We want to come up with reasonable error messages in case any of the aforementioned criteria are invalid such that this message can be reported back to the mechanic. These are:

**partno** Sorry, the part number you entered is invalid. Check that the partno you entered has at least 6 characters and not more than 15 characters.

**qty** Sorry, the quantity you requested is invalid. Please choose between 1 and 10,000.

**comment** Sorry, you entered a string that was too long. Please keep your comments within 4096 characters.

## 2 API Fundamentals

Now that we have a good idea of the problem we are trying to solve we can take a closer look at how the API is structured and how it fits together. Note that the API is essentially very simple, but since this is a tutorial we will try not to make any assumptions. It is useful to define a few terms that have specific meaning within the API:

### 2.1 Terminology

**Affirmation** We use the verb *affirm* to describe the process of validating input (affirm that it is correct) and the noun *Affirmation* in reference to the API interface **Affirmation** which encapsulates this behavior.

**Proclamation** As a noun, *Proclamation* refers to a set of *Affirmations* under a common context. In the example, our **Proclamation** is the set of (at least) three affirmations that ensure that the form input is valid. We use the verb *proclaim* to describe the process of validating the set of values. Thus, to *proclaim* your input to validate each member in the input as defined by the affirmations.

**Input** The noun *Input* is used to refer to a set of key-value pairs that represent the information we intend to affirm.

**Datum** When an **affirmation** is applied to a value in the input, that unit of information is deemed *valid* or *invalid*. In the valid case, this information becomes a *datum*. Thus, *data* (plural) refers to the set of (key,value) pairs that have been positively affirmed.

**Erratum** In the case where an input value is rejected as invalid by affirmation, an *Erratum* is created to hold the bad input and an error message describing why or how the input value is invalid. *Errata* (plural) refers to the set of (key, value, error message) trios that have been negatively affirmed.

It is also important to note that *affirmation* is not necessarily a passive process. Therefore, when an **Affirmation** implementation inspects a value, it may create a new object or transform the input object such that the value is in a more usable state. For example, if a certain **Affirmation** is charged with the task of determining whether a given **String** represents a valid **integer**, it may also instantiate a new **Integer** object on this string and return it as the return value from the **Affirmation.affirm(Object)** method.

In this way multiple **Affirmation** instances may be applied sequentially over some input value, iteratively transforming the value into some desired state. Due to the fact that sequential processing of the same key using different **Affirmation** instances is allowed, the concept of *fatality* is introduced. An **Affirmation** is *fatal* if upon negative affirmation all subsequent processing of the same key should be terminated. For example, say you have two **Affirmation** instances both defined to operate on the key **qty**. The first one's task is to determine if the given key exists while the second has the task of determining if the input represents a valid integer. In this case it makes no sense to evaluate the second one if it is found that the key does not even exist. In this way *fatal* acts like the boolean logical shortcut and operator **&&**.

## 2.2 Process

The overall sequence of events that occur during proclamation are as follows:

1. an **Input** is created from some arbitrary source which holds the (key, value) pairs of interest.
2. a **Proclamation** object is obtained either de novo or more likely retrieved from a cache and then **Proclamation.proclaim(input)** is invoked, initiating the validation.
3. each **Affirmation** is applied in arbitrary order to the **Input**. For those **Affirmation** objects that operate on the same (key, value), each one is applied sequentially in the order that it was defined. Thus, if two **Affirmations** **affirmString** and **affirmLength** are defined to operate on the same key **comment**, they are applied in sequence in that order. This is important since the input of one affirmation may depend upon the output of some previous affirmation.
4. All affirmations are applied. This process has the net effect of sorting the input values into 'good' and 'bad' input which are held by the **Data** and **Errata** containers, respectively.
5. At the end of the **proclaim(Input)** method, the **Proclamation** tests whether the **Errata** is empty. If it is not empty, at least one invalid

data member has been found and a `ProclamationException` is thrown. The `Data` and `Errata` objects may be retrieved from this exception object.

The fundamentally versatile aspect of the design is that arbitrarily complex `Affirmation` implementations can be used. At the simplest level, an `Affirmation` could simply test for the existence.

### 3 `com.inxar.affirm` Implementation

The `org.inxar.affirm` package is solely interfaces and exceptions – no implementation is defined. The `com.inxar.affirm` package is an implementation which uses XML elements to declare affirmation types and associated error messages. The XML document implementation used is `com.sun.xml.tree.XmlDocument`. Each different element in the DTD maps to a different `Affirm` implementation. Coming back to our example, a sample XML document for our proclamation is as follows:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE proclamation SYSTEM "file:/proclamation.dtd">

<proclamation name="orderform-input">
  <affirm-length key="partno" min="6" max="15">
    Sorry, the part number you entered is invalid.
    Check that the partno you entered has at least 6 characters and
    not more than 15 characters.
  </affirm-length>
  <affirm-int-range key="qty" lo="1" hi="10000">
    Sorry, the quantity you requested is invalid. Please
    choose between 1 and 10,000.
  </affirm-int-range>
  <affirm-length key="comment" min="0" max="4096">
    Sorry, you entered a string that was too long.
    Please keep your comments within 4096 characters.
  </affirm-length>
</proclamation>
```

The DTD is always the best reference for what values are allowed, here are the basic types abstracted below. An element has the general (default) form:

```
<affirm-{name} key="" key-type="String" isFinal="true" [other-attributes]>
  error message included here...
</affirm-{name}>
```

`key-type` defaults to `String` but may be one of: `boolean`, `byte`, `short`, `int`, `long`, `float`, `double`, `char`, `String`. `isFinal` defaults to `true` but may be one of `true`, `false`. The implementations defined in `com.inxar.affirm` are described briefly below:

**<affirm-exists>** Returns affirmatively with a the value unchanged if the given value is non-null.

**<affirm-boolean>** Returns affirmatively with a `Boolean` if the given value is already a `Boolean` or if the given value is a `String` and is equal to either 'true' or 'false'.

**<affirm-byte>** Returns affirmatively with a `Byte` if the given value is already a `Byte` or if the given value is a `String` and is parseable to a valid `Byte`.

**<affirm-short>** Returns affirmatively with a **Short** if the given value is already a **Short** or if the given value is a **String** and is parseable to a valid **Short**.

**<affirm-int>** Returns affirmatively with a **Integer** if the given value is already a **Integer** or if the given value is a **String** and is parseable to a valid **Integer**.

**<affirm-long>** Returns affirmatively with a **Long** if the given value is already a **Long** or if the given value is a **String** and is parseable to a valid **Long**.

**<affirm-float>** Returns affirmatively with a **Float** if the given value is already a **Float** or if the given value is a **String** and is parseable to a valid **Float**.

**<affirm-double>** Returns affirmatively with a **Double** if the given value is already a **Double** or if the given value is a **String** and is parseable to a valid **Double**.

**<affirm-char>** Returns affirmatively with a **Character** if the given value is already a **Character** or if the given value is a **String** of length 1.

**<affirm-match pattern="">** Returns affirmatively with the unchanged **String** if the given **String** matches the regular expression given by the attribute **pattern**. Requires **gnu.regex.RE** in the classpath.

**<affirm-int-range lo="" hi="">** Returns affirmatively with an **Integer** if the given value is in the (inclusive) range given by the attributes **lo** and **hi**.

**<affirm-double-range lo="" hi="">** Returns affirmatively with an **Double** if the given value is in the (inclusive) range given by the attributes **lo** and **hi**.

**<affirm-length min="" max="">** Returns affirmatively with the unchanged **String** if the given **String** has length in the (inclusive) range given by the attributes **min** and **max**.

**<affirm-type class="">** Returns affirmatively with the unchanged **Object** if the given **Object** returns true by the **Class.isInstance()** method (**instanceof** operator) for the **Class** name given by the **class** attribute.

**<affirm-cc>** Returns affirmatively with an **int[]** if the given **String** is a valid credit card as determined by the Luhn Check Digit Algorithm.

The element **<affirm>** is provided to allow use of arbitrary **Affirmation** implementations. For example, assume your application needs to know if a certain input value is a prime number. In order to check this, you write a class **org.example.PrimeAffirmation** that implements **org.inxar.affirm.Affirmation**. This class contains code which can correctly check if a value is prime or not. In order to use this class in a “plug-and-play” fashion in the XML file, you need to write a special constructor and add an entry in the file XML like so:

```
<affirm class="org.example.PrimeAffirmation">
  <detail>Please enter a number which is (probably) prime and greater than 7.</detail>
</affirm>
```

The Java class name should be given in the **class** attribute. Each child element **<arg type="" value="">** represents one constructor argument to the class (not shown in this example). The **<detail>** element is used to supply the error message.

Implementations must have a constructor which has 3 + number of **<arg>** elements. Since the **PrimeAffirmation** class does not require any additional constructor arguments, the constructor will only have three arguments, as shown below:

```
package org.example;

public class PrimeAffirmation
implements org.inxar.affirm.Affirmation
{
    public PrimeAffirmation(
        // The object to which the value is associated in the Input.
        Object key,
        // The error message that is printed when the affirmation fails.
    ) {
    }
}
```

```

        String msg,
        // The flag to determine if failure is fatal (described earlier).
        boolean isFatal
    )
    {
        ...
    }
}

```

## 4 Example Code

In this section we go over the steps that one would typically code. We assume that the xml file shown earlier already exists.

```

// the uri of our xml file
String uri = "file:/my_proc_1.xml";

// use a hashtable for the example
final Hashtable hash = new Hashtable();

// populate this data
hash.put("partno", "ABDF-122-K-001");
hash.put("qty", "200");
hash.put("comment", "Rush Order");

// make an anonymous class to wrap the hash
Input input = new Input() {
    public Object get(Object key) { return hash.get(key); }
    public String toString() { return hash.toString();}
};

// create the prolamation object on the uri. This
// may throw an exception which we don't show here...
Proclamation proc = new XMLProclamation(uri);

// now proclaim the input
try {

    // fetch the good data
    Data data = proc.proclaim(input);
    // print the valid data
    System.out.println(data.toString());

} catch (ProclamationException pex) {

    // print the valid data
    System.out.println(pex.data.toString());

    // print the invalid data
    java.util.Enumeration e = pex.errata.enumerator();
    while (e.hasMoreElements()) {
        // get the erratum
        Erratum err = (Erratum)e.nextElement();
        System.out.println("error for key "+err.getKey()+": ");
        Detail detail = err.details();
        do {
            System.out.print(detail.getMessage());

```

```
        } while (detail.hasNext());  
        System.out.println();  
    }  
}
```

## 5 Conclusion

Validation of input data is a tedious and unsavory practice that, while extremely boring to code, is quite necessary to many applications. **affirm** was designed to automate the validation phase in a simple, modular, and straightforward manner. Most standard validation can be done with the affirmation implementations defined in the `com.inxar.affirm` package. The simplicity of the **Affirmation** interface makes it easy to define new objects with complexity bounded only by your needs and imagination.