

# JSON Tools Core 1.6

Bruno Ranschaert

June 29, 2021

## Contents

<b>1</b>	<b>Introduction</b>	<b>2</b>
1.1	Introduction . . . . .	2
1.2	Acknowledgements . . . . .	3
1.3	About S.D.I-Consulting . . . . .	3
1.4	Dependencies . . . . .	3
1.5	License . . . . .	3
1.6	JSON Extensions . . . . .	3
<b>2</b>	<b>The Core Tools</b>	<b>3</b>
2.1	Parsing - Reading JSON . . . . .	3
2.2	Rendering - Writing JSON . . . . .	4
2.3	Mapping . . . . .	4
2.3.1	When to choose mapping . . . . .	4
2.3.2	The mapping process . . . . .	5
2.4	Serialization . . . . .	7
2.4.1	Primitive Types . . . . .	8
2.4.2	Reference Types . . . . .	8
2.4.3	The serialization process . . . . .	10
2.5	Validation . . . . .	11
2.5.1	Basic Rules . . . . .	12
2.5.2	Type Rules . . . . .	13
2.5.3	Attribute Rules . . . . .	13
2.5.4	Structural Rules . . . . .	15
<b>A</b>	<b>License Header</b>	<b>18</b>
<b>B</b>	<b>Validator for Validators</b>	<b>19</b>
<b>C</b>	<b>Changes since 1.5</b>	<b>22</b>

# 1 Introduction

## 1.1 Introduction

JSON (JavaScript Object Notation) is a file format to represent data. It is similar to XML but has different characteristics. It is suited to represent configuration information, implement communication protocols and so on. XML is more suited to represent annotated documents. JSON parsing is very fast, the parser can be kept lean and mean. It is easy for humans to read and write. It is based on a subset of the JavaScript programming language<sup>1</sup>. JSON is a text format that is completely language independent but uses conventions that are familiar to programmers of the C-family of languages<sup>2</sup>. These properties make JSON an ideal data-interchange language. The format is specified on the JSON web site<sup>3</sup>, for the details please visit this site.

JSON is a very simple format. As a result, the parsing and rendering is fast and easy, you can concentrate on the content of the file instead of the format. In XML it is often difficult to fully understand all features (e.g. name spaces, validation, ...). As a result, XML tends to become part of the problem i.s.o. the solution. In JSON everything is well defined, all aspects of the representation are clear, you can concentrate on how you are going to represent your application concepts. The following example comes from the JSON example page<sup>4</sup>.

```
1 { "widget" : {
2     "debug" : "on",
3     "text" : {
4         "onMouseUp" : "sun1.opacity_=(sun1.opacity_/100)*90;",
5         "hOffset" : 250,
6         "data" : "Click_Here",
7         "alignment" : "center",
8         "style" : "bold",
9         "size" : 36,
10        "name" : "text1",
11        "vOffset" : 100 },
12    "image" : {
13        "hOffset" : 250,
14        "alignment" : "center",
15        "src" : "Images/Sun.png",
16        "name" : "sun1",
17        "vOffset" : 250 },
18    "window" : {
19        "width" : 500,
20        "height" : 500,
21        "title" : "Sample_Konfabulator_Widget",
22        "name" : "main_window" } } }
```

This project wants to provide the tools to manipulate and use the format in a Java application.

<sup>1</sup>Standard ECMA-262 3rd Edition - December 1999.

<sup>2</sup>Including C, C++, "C#", Java, JavaScript, Perl, Python, and many others.

<sup>3</sup>The JSON web site: <http://www.json.org/>

<sup>4</sup>JSON Example page: <http://www.json.org/example.html>

## 1.2 Acknowledgements

The JSON Tools library is the result of many suggestions, contributions and reviews from the users. Without the feedback the library would not be as versatile and stable as it is today. Thank you for all the feedback that makes the library better.

## 1.3 About S.D.I-Consulting

Visit the web site <http://www.sdi-consulting.com> for more information. Visit the JSON Tools project summary<sup>5</sup> page, the project itself is hosted on the Berlios service<sup>6</sup>.

## 1.4 Dependencies

The parser uses ANTLR 2.7.7<sup>7</sup>, so the ANTLR runtime is needed for this. It might work with other versions, I simply did not test it. I am aware that ANTLR v3 is available, but it is still in beta. We will stick to the stable release. The project is based on the maven2 build system.

The JSON Tools libraries are written using the new language features from JDK 1.5<sup>8</sup>. Enumerations and generics are used because these make the code nicer to read. There are no dependencies to the new libraries. On the other hand, there is no guarantee we will keep it this way in future releases. If you want to use the libraries for an earlier version of the JDK, the retrotranslator tool<sup>9</sup> might be an option.

## 1.5 License

The library is released under the LGPL<sup>10</sup>. You are free to use it for commercial or non-commercial applications as long as you leave the copyright intact and add a reference to the project. You can find a copy of the license header in appendix A. Let me know what you like and what you don't like about the library so that I can improve it.

## 1.6 JSON Extensions

Comments. I added line comments which start with "#". It is easier for the examples to be able to put comments in the file. The comments are not retained, they are skipped and ignored.

# 2 The Core Tools

## 2.1 Parsing - Reading JSON

The most important tool in the tool set is the parser, it enables you to convert a JSON file or stream into a Java model. All JSON objects remember the position in the file (line, column), so if you are doing post processing of the data you can always refer to the position in the original file. Invoking the parser is very simple as you can see in this example:

---

<sup>5</sup>JSON Tools summary: <http://sdi-consulting.com/menu/software/frameworks/jsontools.html>

<sup>6</sup>JSON Tools project page: <http://jsontools.berlios.de>

<sup>7</sup>ANTLR: <http://www.antlr.org/>

<sup>8</sup>Sun Java 5: <http://java.sun.com/j2se/1.5>

<sup>9</sup>Retrotranslator: <http://retrotranslator.sourceforge.net>

<sup>10</sup>Gnu LGPL: <http://www.gnu.org/licenses/lgpl.html>

```
1 JSONParser lParser = new JSONParser(JSONTest.class.getResourceAsStream("/config.json"));
2 JSONValue lValue = lParser.nextValue();
```

The JSON model is a hierarchy of types, the hierarchy looks like this:

```
1 JSONValue
2     JSONComplex
3         JSONObject
4         JSONArray
5     JSONSimple
6         JSONNull
7         JSONBoolean
8         JSONString
9         JSONNumber
10             JSONInteger
11             JJsonDecimal
```

## 2.2 Rendering - Writing JSON

The classes in the JSON model can render themselves to a String. You can choose to render to a pretty form, nicely indented and easily readable by humans, or you can render to a compact form, no spaces or indentations are provided. This is suited to use on a communications channel when you are implementing a communication protocol.

In the introduction we already saw a pretty rendering of some widget data. The same structure can be rendered without pretty printing in order to reduce whitespace. This can be an interesting feature when space optimization is very important, e.g. communication protocols.

## 2.3 Mapping

### 2.3.1 When to choose mapping

Both mapping tool (this section) and serialization tool (section 2.4 on page 7) can be used to convert Java into JSON and vice versa. These tools have different goals. The goals of the mapper are:

- The JSON text should be clean and straightforward. So no meta information can be stored.
- The data contained in the JSON text should not be dependent on the Java programming language. You do not need to know that the data was produced in Java or that it will be parsed in Java.
- Not all Java data structures have to be mapped: recursive structures, differentiation between primitives and reference types are less important. If a trade-off has to be made, it will be in favor of the JSON format.

The JSON from the mapper can be easily interpreted in another language. An example could be JavaScript in the context of an AJAX<sup>11</sup> communication with the server. The service could

---

<sup>11</sup>AJAX = Asynchronous JavaScript and XML, see <http://en.wikipedia.org/wiki/AJAX> for more information.

be talking some JSON protocol. It is not difficult to map Java data to JSON. It can be done like this:

```
1 import com.sdicons.json.mapper.*;
2 ...
3 JSONValue lObj = JSONMapper.toJSON(myPojo);
```

Converting back to Java is done like this:

```
1 import com.sdicons.json.mapper.*;
2 ...
3 MyBean bean = (MyBean) JSONMapper.toJava(lObj, MyBean.class);
```

Note that the mapper needs some help to convert JSON into Java. As we stated in the goals of the mapper, we cannot store meta information in the JSON text. As a result the mapper cannot know how the JSON text should be mapped. Therefore we pass a class to the mapper (line 3) so that the mapper can exploit this information. In fact, there are two kinds of information the mapper can work with (1) classes as in the example and (2) types e.g. `List<Integer>`. The rationale for this might be illustrated by the following example. Consider a JSON text

```
1 [ "01/12/2006", "03/12/2007", ... ]
```

This list could be interpreted as a list of Strings, but also as a list of Dates. The mapper has no idea what to do with it. When we pass the type `LinkedList<Date>` or the type `LinkedList<String>`, the mapper can exploit this type information and do the right thing. Also note that the mapper automatically exploits this information when the outer layer is a bean, and the list is one of the beans properties.

### 2.3.2 The mapping process

The mapper uses a repository of helpers. Each helper is specialized in mapping instances of a certain class or interface. The mappers are organized in the repository in a hierarchical way, ordered according to the class hierarchy. When mapping an object, the mapper will try to find the most specific helper available. The default hierarchy looks like this:

```
1 // Calling this method:
2 System.out.println(JSONMapper.getRepository().prettyPrint());
3
4 // Results in this output:
5 java.lang.Object
6     java.lang.String
7     java.lang.Boolean
8     java.lang.Byte
9     java.lang.Short
10    java.lang.Integer
11    java.lang.Long
12    java.lang.Float
13    java.lang.Double
14    java.math.BigInteger
15    java.math.BigDecimal
16    java.lang.Character
```

```

17 java.util.Date
18 java.util.Collection
19 java.util.Map

```

The basic Java types (byte, int, char, ..., arrays) are handled internally by the mapper, no helpers are used for this. For all reference types, the repository is used to find an appropriate handler. If there is no specific helper available, the mapper will eventually use the root mapper. Currently there are two flavors available of the root mapper that handles `java.lang.Object`.

- `ObjectMapper` is the default helper for objects that have no specific helper. It tries to access the object as a JavaBean. The object has to have an empty constructor, and the helper will only look at the getters/setters to retrieve the contents of the bean. This helper is the default root helper for compatibility reasons with earlier versions of the JSON Tools. The JavaBean helper can be explicitly activated by calling the method `JSONMapper.useJavaBeanAccess()`.
- `ObjectMapperDirect` is optional, this helper will access the fields directly, no getters or setters are needed. The fields can even be private. This POJO helper can be activated by calling the method `JSONMapper.usePojoAccess();`.

It is also possible to add your own mapper helpers to the repository. As you can see, the default repository is only two levels deep, but it can be much more specialized according to the business needs. There are two different ways to create a helper or to influence the mapping process.

- `@JSONMap`, `@JSONConstruct`. If the `ObjectHelperDirect` is activated as described above, then the class that you want to map can simply annotate two methods with these annotations. The `@JSONMap` annotation has to be used to mark a method that returns an object array. This method will be called by the mapping process when an instance of the class is mapped from the Java model to JSON. These values will be used by the mapper when the instance is mapped from JSON to Java by invoking the constructor that is annotated with the `@JSONConstruct` annotation.
- Another way to create a helper is to create a new class, derived from `SimpleMapperHelper`, and add it to the mapper repository by calling the method `JSONMapper.addHelper(myHelper)`.

Here is an example of an annotated class. It is the first solution, in combination with `ObjectMapperDirect`. Do not forget to activate the POJO mapper.

```

1 public class MyDate
2 {
3     // The fields will be mapped as well, independent of the
4     // constructor values.
5     private Date theDate;
6     private String theTimeZone;
7
8     // Because of this annotation, the ObjectMapperDirect will call this
9     // function and serialize the values in the object array. These values
10    // will be used later on to call the annotated constructor.
11    @JSONMap
12    public Object[] getTime()
13    {

```

```

14         return new Object[]{theDate.getTime(), theTimeZone};
15     }
16
17     // This constructor will be called with the same values that were
18     // provided by the other annotated method.
19     @JSONConstruct
20     public MyDate(long aTime, String aTimeZone)
21     {
22         theDate = new Date(aTime);
23         theTimeZone = aTimeZone;
24     }
25 }

```

## 2.4 Serialization

Both mapping tool (section 2.3 on page 4) and serialization tool (this section) can be used to convert Java into JSON and vice versa. These tools have different goals. The goals of the serializer are:

- The serialization tool could be an alternative for native serialization<sup>12</sup> (regarding functionality). This does not mean that all kinds of classes are supported out of the box, but it means that the general mechanism should be there and there should be an easy way to extend the mechanism so that we can deal with all classes.
- The serialization tool should preserve the difference between reference types and primitive types.
- Recursive types should be supported without putting the (de)serializer into an infinite loop.
- Instance identity should be preserved. If the same instance is referenced from other instances, the same structure should be reconstructed during de-serialization. There should only be one instance representing the original referenced instance.
- The content of the JSON text can contain meta information which can help de-serialization. We are allowed to add extra information in the JSON text in order to accomplish the other goals.

This tool enables you to render POJO's<sup>13</sup> to a JSON file. It is similar to the XML serialization in Java or the XML Stream library, but it uses the JSON format. The result is a very fast text serialization, you can customize it if you want. The code is based on the SISE project, it was adjusted to make use of and benefit from the JSON format. Marshaling (converting from Java to JSON) as well as un-marshaling is very straightforward:

```

1 import com.sdicons.json.serializer.marshall.*;
2 ...
3 myTestObject = ...
4 Marshall marshall = new JSONMarshall();
5 JSONObject result = marshall.marshall(myTestObject);

```

<sup>12</sup>Java serialization: <http://java.sun.com/j2se/1.5.0/docs/guide/serialization/>

<sup>13</sup>POJO = Plain Old Java Object. See [http://en.wikipedia.org/wiki/Plain\\_Old\\_Java\\_Object](http://en.wikipedia.org/wiki/Plain_Old_Java_Object) for more information.

And the other way around:

```
1 import com.sdicons.json.serializer.marshall.*;
2 ...
3 JSONObject myJSONObject = ...
4 MarshallValue lResult = marshall.unmarshall(myJSONObject);
5 ... = lResult.getReference()
```

You might wonder what the `MarshallValue` (on line 4) is all about, why is un-marshaling giving an extra object back? The answer is that we went to great lengths to provide marshaling or un-marshaling for both Java reference types as Java basic types. A basic type needs to be fetched using specific methods (there is no other way). In order to provide these specific methods we need an extra class.

### 2.4.1 Primitive Types

Primitive types are represented like this.

```
1 { ">" : "P",
2   "=" : "1",
3   "t" : "int" }
```

The ">" attribute with value "P" indicates a primitive type. The "=" attribute contains the representation of the value and the "t" attribute contains the original Java type.

### 2.4.2 Reference Types

An array is defined recursively like this. We can see the ">" attribute this time with the "A" value, indicating that the object represents an array. The "C" attribute contains the type representation for arrays as it is defined in Java. The "=" attribute contains a list of the values.

```
1 { ">" : "A",
2   "c" : "I",
3   "=" :
4     [
5       {
6         ">" : "P",
7         "=" : "0",
8         "t" : "int" },
9       {
10        ">" : "P",
11        "=" : "1",
12        "t" : "int" },
13      {
14        ">" : "P",
15        "=" : "2",
16        "t" : "int" },
17      {
18        ">" : "P",
19        "=" : "3",
```



```

20         "t" : "int" },
21     {
22         ">" : "P",
23         "=" : "4",
24         "t" : "int" },
25     {
26         ">" : "P",
27         "=" : "5",
28         "t" : "int" } ] }

```

An object is represented like this.

```

1  {
2      ">" : "O",
3      "c" : "com.sdicons.json.serializer.MyBean",
4      "&" : "id0",
5      "=" : {
6          "int2" :
7              { ">" : "null" },
8          "ptr" :
9              { ">" : "R",
10                "*" : "id0" },
11          "name" :
12              { ">" : "O",
13                "c" : "java.lang.String",
14                "&" : "id2",
15                "=" : "This_is_a_test..." },
16          "int1" :
17              { ">" : "null" },
18          "id" :
19              { ">" : "O",
20                "c" : "java.lang.Integer",
21                "&" : "id1",
22                "=" : "1003" } } }

```

The ">" marker contains "O" for object this time. The "C" attribute contains a fully qualified class name. The "&" contains a unique id, it can be used to refer to the object so that we are able to represent recursive data structures. The "=" attribute contains a JSON object having a property for each JavaBean property. The property value is recursively a representation of a Java object. Note that there is a special notation to represent Java `null` values.

```

1  { ">" : "null" }

```

Also note that you can refer to other objects with the reference object which looks like this:

```

1  { ">" : "R",
2      "*" : "id0" }

```

### 2.4.3 The serialization process

The serialization process uses the same mechanism as the mapping process (section 2.3.2 on page 5), but the repository contains serialization helpers in stead of mapping helpers. There are also two different flavors of root serializers available:

- `ObjectHelper` Serializes an instance as a JavaBean. This is the default for compatibility reasons. You can explicitly activate it by calling `((JsonMarshall) marshall).useJavaBeanAccess()`.
- `ObjectHelperDirect` Serializes an instance as a POJO. You can activate this by calling the method `((JsonMarshall) marshall).usePojoAccess()`.

You can customize the serializer for your own business model in two ways.

- `@JSONSerialize`, `@JSONConstruct` in combination with the `ObjectHelperDirect`.
- Deriving your own helper class from `MarshallHelper` and adding it with the method call `((JsonMarshall) marshall).addHelper(myHelper)`.

Here is an example of an annotated class.

```
1 public class MyDate
2 {
3     // These private fields will be serialized in addition to the
4     // constructor values.
5     private Date theDate;
6     private String theTimeZone;
7
8     // This method will be called during serialization to obtain the
9     // values that can later be used to call the constructor.
10    @JSONSerialize
11    public Object[] getTime()
12    {
13        return new Object[]{theDate.getTime(), theTimeZone};
14    }
15
16    // This constructor will be called with the values that were provided
17    // by the other annotated method.
18    @JSONConstruct
19    public MyDate(long aTime, String aTimeZone)
20    {
21        theDate = new Date(aTime);
22        theTimeZone = aTimeZone;
23    }
24 }
```

The result of the serialization looks like the following listing. As you can see, there are two extra artificial fields `cons-0` on line 5 and `cons-1` on line 10 which are generated automatically by the serializer, these properties contain the values which were provided by the method which was annotated with `@JSONSerialize`. These same properties will be used for calling the `@JSONConstruct` annotated constructor.

```

1 { ">" : "0",
2   "&" : "id0",
3   "c" : "MyDate",
4   "=" : {
5     "cons-0" : {
6       ">" : "0",
7       "&" : "id1",
8       "c" : "java.lang.Long",
9       "=" : "1212717107857" },
10    "cons-1" : {
11      ">" : "0",
12      "&" : "id2",
13      "c" : "java.lang.String",
14      "=" : "CEST" },
15    "theDate" : {
16      ">" : "0",
17      "&" : "id3",
18      "c" : "java.util.Date",
19      "=" : "2008-06-06_03:51:47,857_CEST" },
20    "theTimeZone" : {
21      ">" : "R",
22      "*" : "id2" } } }

```

## 2.5 Validation

This tool enables you to validate your JSON files. You can specify which content you expect, the validator can check these constraints for you. The system is straightforward to use and extend. You can add your own rules if you have specific needs. The validation definition is in JSON - as you would expect. Built-in rules:

```

1 { "name" : "Some_rule_name",
2   "type" : "<built-in-type>" }

```

A validation document consists of a validation rule. This rule will be applied to the `JSONValue` that has to be validated. The validation rules can be nested, so it is possible to create complex rules out of simpler ones. The "type" attribute is obligatory. The "name" is optional, it will be used during error reporting and for re-use. The predefined rules are listed below. The name can come in handy while debugging. The name of the failing validation will be available in the exception. If you give each rule its own name or number, you can quickly find out on which predicate the validation fails. Here is an example of how you can create a validator.

```

1 // First we create a parser to read the validator specification which is
2 // defined using the (what did you think) JSON format.
3 // The validator definition is located in the "my-validator.json" resource in the
4 // class path.
5 JSONParser lParser =
6     new JSONParser(
7         MyClass.class.getResourceAsStream("my-validator.json"));
8
9 // We parse the validator spec and convert it into a Java representation.
10 JSONObject lValidatorObject = (JSONObject) lParser.nextValue();
11
12 // Finally we can convert our validator using the Java model.

```

```

13 Validator lValidator = new JSONValidator(lValidatorObject);
14
15 And now that you have the validator, you can start validating your data.
16
17 // First we create a parser to read the data.
18 JSONParser lParser = new JSONParser(MyClass.class.getResourceAsStream("data.json"));
19
20 // We parse the datafile and convert it into a Java representation.
21 JSONValue lMyData = lParser.nextValue();
22
23 // Now we can use the validator to check on our data. We can test if the data has the
24 // correct format or not.
25 lValidator.validate(lMyData);

```

## 2.5.1 Basic Rules

These rules are the basic rules in boolean logic.

### “type” : “true”

- Parameters: -
- Description: This rule always succeeds.

A validator that will succeed on all JSON data structures.

```

1 { "name" : "This validator validates everything",
2   "type" : "true" }

```

### “type” : “false”

- Parameters: -
- Description: This rule always fails.

A validator that rejects all data structures.

```

1 { "name" : "This validator rejects all",
2   "type" : "false" }

```

### “type” : “and”

- Parameters:
  - *rules*: Array of nested rules.
- Description: All nested rules have to hold for the and rule to succeed.

A validator that succeeds if the object under scrutiny is both a list and has content consisting of integers.

```

1 { "name" : "List of integers",
2   "type" : "and",
3   "rules" : [ {"type": "array"}, {"type": "content", "rule": {"type": "int"} } ] }

```

### “type” : “or”

- Parameters:
  - *rules*: Array of nested rules.

- Description: One of the nested rules has to succeed for this rule to succeed.

A validator that validates booleans or integers.

```
1 { "name" : "Null_or_int",
2   "type" : "or",
3   "rules" : [ {"type": "int"}, {"type": "bool"} ] }
```

#### **“type” : “not”**

- Parameters:
  - *rule*: A single nested rule.
- Description: The rule succeeds if the nested rule fails and vice versa.

### **2.5.2 Type Rules**

These rules are predefined rules which allows you to specify type restrictions on the JSON data elements. The meaning of these predicates is obvious, they will not be discussed. See the examples for more information. Following type clauses can be used:

- **“type” : “complex”**
- **“type” : “array”**
- **“type” : “object”**
- **“type” : “simple”**
- **“type” : “null”**
- **“type” : “bool”**
- **“type” : “string”**
- **“type” : “number”**
- **“type” : “int”**
- **“type” : “decimal”**

### **2.5.3 Attribute Rules**

These rules check for attributes of certain types.

#### **“type” : “length”**

- Parameters:
  - *min*: (optional) The minimal length of the array.
  - *max*: (optional) The maximal length of the array.
- Description: Applicable to complex objects and string objects. The rule will fail if the object under investigation has another type. For array objects the number of elements is counted, for objects the number of properties and for strings, the length of its value in Java (not the JSON representation; “\n” in the file counts as a single character).

A validator that only wants arrays of length 5.

```
1 { "name" : "Array_of_length_5",
2   "type" : "and",
3   "rules" : [{"type": "array"}, {"type": "length", "min": 5, "max": 5}] }
```

### **“type” : “range”**

- Parameters:
  - *min*: (optional) The minimal value.
  - *max*: (optional) The maximal value.
- Description: Applicable to [JSONNumbers](#), i.e. [JSONInteger](#) and [JSONDecimal](#).

Allow numbers between 50 and 100.

```
1 { "name" : "Range_validator",  
2   "type" : "range",  
3   "min" : 50,  
4   "max" : 100 }
```

### **“type” : “enum”**

- Parameters:
  - *values*: An array of JSON values.
- Description: The value has to occur in the provided list. The list can contain simple types as well as complex nested types.

An enum validator.

```
1 { "name" : "Enum_validator",  
2   "type" : "enum",  
3   "values" : [13, 17, "JSON", 123.12, [1, 2, 3], {"key": "value"}] }
```

### **“type” : “regexp”**

- Parameters:
  - *pattern*: A regular expression pattern.
- Description: For strings, requires a predefined format according to the regular expression.

A validator that validates strings containing a sequence of a's , b's and c's.

```
1 { "name" : "A-B-C_validator",  
2   "type" : "regexp",  
3   "pattern" : "a*b*c*" }
```

### **“type” : “content”**

- Parameters:
  - *rule*: The rule that specifies how the content of a complex structure - an array or the property values of an object - should behave.
- Description: Note that in contrast with the “properties” rule (for objects), you can specify in a single rule what all property values of an object should look like.

See **“type” : “and”** on page 12.

### **“type” : “properties”**

- Parameters:
  - *pairs*: A list of “key/value” pair descriptions. Note that in contrast with the content rule above you can specify a rule per attribute. Each description contains three properties:
    - \* *key*: The key string.
    - \* *optional*: A boolean indicating whether this property is optional or not.
    - \* *rule*: A validation rule that should be applied to the properties value.
- Description: This predicate is only applicable (and only has meaning) on object data structures. It will fail on any other type.

It will validate objects looking like this:

```

1 Example data structure that will be validated:
2 {{"name":"Bruno Ranschaert", "country":"Belgium", "salary":13.0 }}
3
4 The validator looks like this:
5 { "name" : "Contact spec.",
6   "type" : "properties",
7   "pairs" : [{"key":"name", "optional":false, "rule":{"type":"string"}},
8               {"key":"country", "optional":false, "rule":{"type":"string"}},
9               {"key":"salary", "optional":true, "rule":{"type":"decimal" } } ] }
```

## 2.5.4 Structural Rules

### “type” : “ref”

- Parameters:
  - \*: The name of the rule to invoke.
- Description: This rule lets you specify recursive rules. Be careful not to create infinite validations which is quite possible using this rule. The containing rule will be fetched just before validation, there will be no error message during construction when the containing rule is not found. The rule will fail in this case. If there are several rules with the same name, only the last one with that name is remembered and the last one will be used.

A validator that validates nested lists of integers. A ref is needed to enable recursion in the validator.

```

1 { "name" : "Nested list of integers",
2   "type" : "and",
3   "rules" : [
4     {"type":"array"},
5     {"type":"content",
6      "rule": {
7        "type" : "or",
8        "rules": [
9          {"type":"int"},
10         {"type":"ref", "*" : "Nested list of integers" } ] } } ] }
```

### “type” : “let”

- Parameters:
  - `rules`: A list of rules.
  - `*`: The name of the rule that should be used.
- Description: Lets you specify a number of named rules in advance. It is a convenience rule that lets you specify a list of global shared validation rules in advance before using these later on. It becomes possible to first define a number of recurring types and then give the starting point. It is a utility rule that lets you tackle more complex validations. Note that it makes no sense to define anonymous rules inside the list, it is impossible to refer to these later on.

```

1 { "name" : "Let test a's or b's",
2   "type" : "let",
3   "*"    : "start",
4   "rules" :
5     [{"name": "start", "type": "or", "rules": [{"type": "ref", "*": "a"},
6                                               {"type": "ref", "*": "b"}]},
7     {"name": "a", "type": "regex", "pattern": "a*"},
8     {"name": "b", "type": "regex", "pattern": "b*"} ] }

```

The validator class looks like this:

```

1 public class MyValidator
2 extends CustomValidator
3 {
4     public MyValidator(
5         String aName, JSONObject aRule,
6         HashMap<String, Validator> aRuleset)
7     {
8         super(aName, aRule, aRuleset);
9     }
10
11     public void validate(JSONValue aValue)
12     throws ValidationException
13     {
14         // Do whatever you need to do on aValue ...
15         // If validation is ok, simply return.
16         // If validation fails, you can use:
17         // fail(JSONValue aValue) or
18         // fail(String aReason, JSONValue aValue)
19         // to throw the Validation exception for you.
20     }
21 }

```

### “type”: “custom”

- Parameters:
  - `class`: The fully qualified class name of the validator.
- Description: An instance of this validator will be created and will be given a hash map of validations. A custom validator should be derived from `CustomValidator`.



```
1 { "name" : "Custom_test",  
2   "type" : "custom",  
3   "class" : "com.sdicons.json.validator.MyValidator" }
```

### **“type” : “switch”**

- Parameters:
  - *key*: The key name of the object that will act as the discriminator.
  - *case*: A list of objects containing the parameters “values” and “rule”. The first one is a list of values the second one a validator rule.
- Description: The switch validator is a convenience one. It is a subset of the or validator, but the problem with the or validator is that it does a bad job for error reporting when things go wrong. The reason is that all rules fail and it is not always clear why, because the reason a rule fails might be some levels deeper. The switch validator selects a validator based on the value of a property encountered in the value being validated. The error produced will be the one of the selected validator. The first applicable validator is used, the following ones are ignored. Example: The top level rule in the validator for validators contains a switch that could have been described by an or, but the switch gives better error messages.

## A License Header

JSOINTOOLS - Java JSON Tools  
Copyright (C) 2006-2008 S.D.I.-Consulting BVBA  
<http://www.sdi-consulting.com>  
<mailto://nospam@sdi-consulting.com>

This library is free software; you can redistribute it and/or modify it under the terms of the GNU Lesser General Public License as published by the Free Software Foundation; either version 2.1 of the License, or (at your option) any later version.

This library is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU Lesser General Public License for more details.

You should have received a copy of the GNU Lesser General Public License along with this library; if not, write to the Free Software Foundation, Inc., 51 Franklin St, Fifth Floor, Boston, MA 02110-1301 USA

## B Validator for Validators

This example validator is able to validate validators. The example is a bit contrived because the validators really don't need validation because it is built-in in the construction. It is interesting because it can serve as a definition of how to construct a validator.

```
1 {
2   "name": "Validator_validator",
3   "type": "let",
4   "key": "rule",
5   "rules":
6   [
7     ##### START #####
8     {
9       "name": "rule",
10      "type": "switch",
11      "key": "type",
12      "case":
13      [
14        {"values": ["true", "false", "null"], "rule": {"type": "ref", "*": "atom-rule"}},
15        {"values": ["int", "complex", "array", "object", "simple",
16          "null", "bool", "string", "number", "decimal"],
17          "rule": {"type": "ref", "*": "type-rule"}},
18        {"values": ["not", "content"], "rule": {"type": "ref", "*": "rules-rule"}},
19        {"values": ["and", "or"], "rule": {"type": "ref", "*": "ruleset-rule"}},
20        {"values": ["length", "range"], "rule": {"type": "ref", "*": "minmax-rule"}},
21        {"values": ["ref"], "rule": {"type": "ref", "*": "ref-rule"}},
22        {"values": ["custom"], "rule": {"type": "ref", "*": "custom-rule"}},
23        {"values": ["enum"], "rule": {"type": "ref", "*": "enum-rule"}},
24        {"values": ["let"], "rule": {"type": "ref", "*": "let-rule"}},
25        {"values": ["regexp"], "rule": {"type": "ref", "*": "regexp-rule"}},
26        {"values": ["properties"], "rule": {"type": "ref", "*": "properties-rule"}},
27        {"values": ["switch"], "rule": {"type": "ref", "*": "switch-rule"}}
28      ]
29    },
30    ##### RULESET #####
31    {
32      "name": "ruleset",
33      "type": "and",
34      "rules": [{"type": "array"}, {"type": "content", "rule": {"type": "ref", "*": "rule"}}]
35    },
36    ##### PAIRS #####
37    {
38      "name": "pairs",
39      "type": "and",
40      "rules": [{"type": "array"}, {"type": "content", "rule": {"type": "ref", "*": "pair"}}]
41    },
42    ##### PAIR #####
43    {
44      "name": "pair",
45      "type": "properties",
46      "pairs" :
47      [{"key": "key", "optional": false, "rule": {"type": "string"}},
48       {"key": "optional", "optional": false, "rule": {"type": "bool"}},
49       {"key": "rule", "optional": false, "rule": {"type": "ref", "*": "rule"}}]
50    },
51  ],
52  ##### CASES #####
53  {
54    "name": "cases",
55    "type": "and",
56    "rules": [{"type": "array"}, {"type": "content", "rule": {"type": "ref", "*": "case"}}]
57  },
58  ##### CASE #####
59  {
60    "name": "case",
61    "type": "properties",
62    "pairs" :
```

```

63         [{"key": "values", "optional": false, "rule": {"type": "array"}},
64         {"key": "rule", "optional": false, "rule": {"type": "ref", "*": "rule"}}
65     ]
66 },
67 ##### ATOM #####
68 {
69     "name": "atom-rule",
70     "type": "properties",
71     "pairs" :
72     [{"key": "name", "optional": true, "rule": {"type": "string"}},
73      {"key": "type", "optional": false, "rule":
74          {"type": "enum", "values": ["true", "false", "null"]}}
75     ]
76 },
77 ##### RULESET-RULE #####
78 {
79     "name": "ruleset-rule",
80     "type": "properties",
81     "pairs" :
82     [{"key": "name", "optional": true, "rule": {"type": "string"}},
83      {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["and", "or"]}},
84      {"key": "rules", "optional": false, "rule": {"type": "ref", "*": "ruleset"}}
85     ]
86 },
87 ##### RULES-RULE #####
88 {
89     "name": "rules-rule",
90     "type": "properties",
91     "pairs" :
92     [{"key": "name", "optional": true, "rule": {"type": "string"}},
93      {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["not", "content"]}},
94      {"key": "rule", "optional": false, "rule": {"type": "ref", "*": "rule"}}
95     ]
96 },
97 ##### TYPE #####
98 {
99     "name": "type-rule",
100    "type": "properties",
101    "pairs" :
102    [{"key": "name", "optional": true, "rule": {"type": "string"}},
103     {"key": "type", "optional": false, "rule": {"type": "enum",
104         "values": ["int", "complex", "array", "object",
105         "simple", "null", "bool", "string", "number",
106         "decimal"]}}
107    ]
108 },
109 ##### MINMAX #####
110 {
111     "name": "minmax-rule",
112     "type": "properties",
113     "pairs" :
114     [{"key": "name", "optional": true, "rule": {"type": "string"}},
115      {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["length", "range"]}},
116      {"key": "min", "optional": true, "rule": {"type": "number"}},
117      {"key": "max", "optional": true, "rule": {"type": "number"}}
118     ]
119 },
120 ##### REF #####
121 {
122     "name": "ref-rule",
123     "type": "properties",
124     "pairs" :
125     [{"key": "name", "optional": true, "rule": {"type": "string"}},
126      {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["ref"]}},
127      {"key": "*", "optional": false, "rule": {"type": "string"}}
128     ]
129 },
130 ##### CUSTOM #####
131 {

```

```

132     "name": "custom-rule",
133     "type": "properties",
134     "pairs" :
135     [{"key": "name", "optional": true, "rule": {"type": "string"}},
136      {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["custom"]}},
137      {"key": "class", "optional": true, "rule": {"type": "string"}}
138    ]
139  },
140  ##### ENUM #####
141  {
142    "name": "enum-rule",
143    "type": "properties",
144    "pairs" :
145    [{"key": "name", "optional": true, "rule": {"type": "string"}},
146     {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["enum"]}},
147     {"key": "values", "optional": true, "rule": {"type": "array"}}
148    ]
149  },
150  ##### LET #####
151  {
152    "name": "let-rule",
153    "type": "properties",
154    "pairs" :
155    [{"key": "name", "optional": true, "rule": {"type": "string"}},
156     {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["let"]}},
157     {"key": "rules", "optional": false, "rule": {"type": "ref", "*": "ruleset"}},
158     {"key": "*", "optional": false, "rule": {"type": "string"}}
159    ]
160  },
161  ##### REGEXP #####
162  {
163    "name": "regexp-rule",
164    "type": "properties",
165    "pairs" :
166    [{"key": "name", "optional": true, "rule": {"type": "string"}},
167     {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["regexp"]}},
168     {"key": "pattern", "optional": false, "rule": {"type": "string"}}
169    ]
170  },
171  ##### PROPERTIES #####
172  {
173    "name": "properties-rule",
174    "type": "properties",
175    "pairs" :
176    [{"key": "name", "optional": true, "rule": {"type": "string"}},
177     {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["properties"]}},
178     {"key": "pairs", "optional": false, "rule": {"type": "ref", "*": "pairs"}}
179    ]
180  },
181  ##### SWITCH #####
182  {
183    "name": "switch-rule",
184    "type": "properties",
185    "pairs" :
186    [{"key": "name", "optional": true, "rule": {"type": "string"}},
187     {"key": "type", "optional": false, "rule": {"type": "enum", "values": ["switch"]}},
188     {"key": "key", "optional": false, "rule": {"type": "string"}},
189     {"key": "case", "optional": false, "rule": {"type": "ref", "*": "cases"}}
190    ]
191  }
192 ]
193 }

```

## C Changes since 1.5

Changes to the mapper:

- \* Added ObjectMapperDirect, a helper that can map plain POJO's.  
In previous version only a helper for JavaBean properties was provided.
- \* Added @JSONMap, @JSONConstruct annotations that let you quickly create mapper helpers for POJO's that do not have an empty constructor.
- \* Added a method to the mapper that lets you add mappers quickly.
- \* Added an Enum mapper.

Changes to the serialzer (Marshall) same system as for the mapper was added:

- \* Added ObjectHelperDirect, a helper to serialize POJO's.
- \* Added @JSONSerialize and @JSONConstruct.
- \* Added method to the marshall that lets you add helpers.

Miscelaneous:

- \* Update to newer version of ANTLR (no not the version 3 which is still in beta).
- \* Code review, small code improvements.
- \* Converted documentation from lout to latex. Lout is a fine system, but latex is more mainstream so more tools, packages and help are available.
- \* Documentation update.