

# JSON-B: Java™ API for JSON Binding

*Version 1.0 Early Draft*

*July 3, 2015*

Editors:

Martin Grebac

Martin Vojtek

Comments to: [users@jsonb-spec.java.net](mailto:users@jsonb-spec.java.net)

*Oracle Corporation*

*500 Oracle Parkway, Redwood Shores, CA 94065 USA.*

DRAFT

**JSR-367 Java API for JSON Binding (“Specification”)**  
**Version: 1.0**  
**Status: Early Draft**  
**Release: July 3, 2015**  
**Copyright 2014 Oracle America, Inc. (“Oracle”)**  
**500 Oracle Parkway, Redwood Shores, California 94065, U.S.A**  
**All rights reserved.**

TBD

DRAFT

# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Status . . . . .	1
1.2	Goals . . . . .	1
1.3	Non-Goals . . . . .	2
1.4	Conventions . . . . .	2
1.5	Terminology . . . . .	3
1.6	Expert Group Members . . . . .	3
1.7	Acknowledgements . . . . .	4
<b>2</b>	<b>Runtime API</b>	<b>5</b>
<b>3</b>	<b>Default Mapping</b>	<b>7</b>
3.1	General . . . . .	7
3.2	Errors . . . . .	7
3.3	Basic Java Types . . . . .	7
3.3.1	java.lang.String, Character . . . . .	8
3.3.2	java.lang.Byte, Short, Integer, Long, Float, Double . . . . .	8
3.3.3	java.lang.Boolean . . . . .	8
3.3.4	java.lang.Number . . . . .	8
3.4	Specific Standard Java SE Types . . . . .	8
3.4.1	java.math.BigInteger, BigDecimal . . . . .	9
3.4.2	java.net.URL, URI . . . . .	9
3.4.3	java.util.Optional, OptionalInt, OptionalLong, OptionalDouble . . . . .	9
3.5	Dates . . . . .	10
3.5.1	java.util.Date, Calendar, GregorianCalendar . . . . .	10
3.5.2	java.util.TimeZone, SimpleTimeZone . . . . .	10
3.5.3	java.time.* . . . . .	11

3.6	Untyped mapping . . . . .	11
3.7	Java Class . . . . .	12
3.7.1	Scope and Field access strategy . . . . .	12
3.7.2	Nested Classes . . . . .	12
3.7.3	Static Nested Classes . . . . .	12
3.7.4	Anonymous Classes . . . . .	13
3.8	Polymorphic Types . . . . .	13
3.9	Enum . . . . .	13
3.10	Interfaces . . . . .	13
3.11	Collections . . . . .	13
3.12	Arrays . . . . .	14
3.13	Attribute order . . . . .	14
3.14	Null value handling . . . . .	14
3.14.1	Null Java field . . . . .	14
3.14.2	Null Array Values . . . . .	15
3.15	Names and identifiers . . . . .	15
3.16	Generics . . . . .	15
3.17	Big numbers . . . . .	15
3.17.1	Type resolution algorithm . . . . .	15
3.18	Must-Ignore policy . . . . .	17
3.19	Uniqueness of properties . . . . .	17
3.20	JSON Processing integration . . . . .	17
<b>4</b>	<b>Customizing Mapping</b>	<b>19</b>
4.1	Customizing Property Names . . . . .	19
4.1.1	javax.json.bind.annotation.JsonbTransient . . . . .	19
4.1.2	javax.json.bind.annotation.JsonbProperty . . . . .	19
4.1.3	javax.json.bind.config.PropertyNamingStrategy . . . . .	20
4.1.4	Property names resolution . . . . .	20
4.2	Customizing Property Order . . . . .	20
4.3	Customizing Null Handling . . . . .	21
4.3.1	javax.json.bind.annotation.JsonbNillable . . . . .	21
4.3.2	Global null handling configuration . . . . .	21
4.4	I-JSON support . . . . .	21
4.4.1	Strict date serialization . . . . .	22

4.5	Simple values . . . . .	22
4.6	Custom instantiation . . . . .	22
4.7	Custom visibility . . . . .	22
4.8	Custom mapping . . . . .	23
4.9	Custom date format . . . . .	23
4.10	Custom number format . . . . .	23
4.11	Custom binary data handling . . . . .	24
<b>Bibliography</b>		<b>25</b>

DRAFT



# Chapter 1

## Introduction

This specification defines binding API between Java objects and JSON [4] documents. Readers are assumed to be familiar with JSON; for more information about JSON, see:

- Architectural Styles and the Design of Network-based Software Architectures[7]
- The REST Wiki[8]
- JSON on Wikipedia[5]

### 1.1 Status

This is an early draft; this specification is not yet complete. A list of open issues can be found at:

[http://java.net/jira/browse/JSONB\\_SPEC](http://java.net/jira/browse/JSONB_SPEC)

The corresponding Javadocs can be found online at:

<http://jsonb-spec.java.net/>

The reference implementation will be obtainable from:

<http://eclipselink.org/>

The expert group is seeking feedback from the community on any aspect of this specification. Please send comments to:

[users@jsonb-spec.java.net](mailto:users@jsonb-spec.java.net)

### 1.2 Goals

The goals of the API are as follows:

**JSON** Support binding (marshalling and unmarshalling) for all RFC 7159-compatible JSON documents.

**Relationships to JSON Related specifications** JSON-related specifications will be surveyed to determine their relationship to JSON-Binding.

**Consistency** Maintain consistency with JAXB (Java API for XML Binding) and other Java EE and SE APIs where appropriate.

**Convention** Define default mapping of Java classes and instances to JSON document counterparts.

**Customization** Allow customization of the default mapping definition.

**Ease Of Use** Default use of the APIs should not require prior knowledge of the JSON document format and specification.

**Partial Mapping** In many use cases, only a subset of JSON Document is required to be mapped to a Java object instance.

**Integration** Define or enable integration with following Java EE technology standards:

- JSR 374 - Java API for JSON Processing (JSON-P) 1.1
- JSR 349 - Bean Validation (BV) 1.1
- JSR 370 - Java™API for RESTful Web Services (JAX-RS) 2.1

## 1.3 Non-Goals

The following are non-goals:

**Preserving equivalence (Round-trip)** The specification recommends, but does not require equivalence of content for unmarshalled and marshalled JSON documents.

**JSON Schema** Generation of JSON Schema from Java classes, as well as validation based on JSON schema.

**JEP 198 Lightweight JSON API Support** Support and integration with Lightweight JSON API as defined within JEP 198 is out of scope of this specification. Will be reconsidered in future specification revisions.

## 1.4 Conventions

The keywords ‘MUST’, ‘MUST NOT’, ‘REQUIRED’, ‘SHALL’, ‘SHALL NOT’, ‘SHOULD’, ‘SHOULD NOT’, ‘RECOMMENDED’, ‘MAY’, and ‘OPTIONAL’ in this document are to be interpreted as described in RFC 2119[1].

Java code and sample data fragments are formatted as shown in figure 1.1:

URIs of the general form ‘http://example.org/...’ and ‘http://example.com/...’ represent application or context-dependent URIs.

All parts of this specification are normative, with the exception of examples, notes and sections explicitly marked as ‘Non-Normative’. Non-normative notes are formatted as shown below.

**Note:** *This is a note.*

Figure 1.1: Example Java Code

```
1 package com.example.hello;
2
3 public class Hello {
4     public static void main(String args[]) {
5         System.out.println("Hello World");
6     }
7 }
```

## 1.5 Terminology

**Data binding** Process which defines the representation of information in a JSON document as an object instance, and vice versa.

**Unmarshalling** Process of reading a JSON document and constructing a tree of content objects, where each object corresponds to part of JSON document, thus the content tree reflects the document's content.

**Marshalling** Inverse process to unmarshalling. Process of traversing content object tree and writing a JSON document that reflects the tree's content.

## 1.6 Expert Group Members

This specification is being developed as part of JSR 367 under the Java Community Process. It is the result of the collaborative work of the members of the JSR 367 Expert Group. The following are the present expert group members:

- Martin Grebac (Oracle)
- Martin Vojtek (Oracle, Datlowe)
- Hendrik Saly (Individual Member)
- Gregor Zurowski (Individual Member)
- Inderjeet Singh (Individual Member)
- Eugen Cepoi (Individual Member)
- Przemyslaw Bielicki (Individual Member)
- Kyung Koo Yoon (TmaxSoft, Inc.)
- Otavio Santana (Individual Member)
- Rick Curtis (IBM)
- Alexander Salvanos (Individual Member)
- Romain Manni-Bucau (Tomitribe)

## 1.7 Acknowledgements

During the course of this JSR we received many excellent suggestions. Special thanks to Heather VanCura and David Delabassee for feedback and help with evangelizing the specification, and John Clingan for feedback and language corrections.

During the course of this JSR we also received many excellent suggestions on the JSR's java.net project mailing lists. Thanks in particular to Olena Syrota, Oleg Tsal-Tsalko and whole JUG UA for their contributions.

The following individuals have also made invaluable technical contributions: ... .

## Chapter 2

# Runtime API

The JSON-B runtime API provides access to marshalling and unmarshalling operations for manipulating JSON documents and mapped JSON-B classes and instances. The full specification of the binding framework is available in the javadoc for the `javax.json.bind` package accompanied with this specification.

DRAFT

# Chapter 3

## Default Mapping

This section defines the default binding (representation) of Java components and classes to JSON documents. The default binding defined here can be further customized as specified in Chapter 4 - Customizing Mapping.

### 3.1 General

JSON Binding implementations ('implementations' in further text) **MUST** support binding of JSON documents as defined in RFC 7159 JSON Grammar [4]. Marshalled JSON output **MUST** conform to the RFC 7159 JSON Grammar [4] and be encoded in UTF-8 encoding as defined in Section 8.1 (Character Encoding) of RFC 7159 [4]. [JSB-3.1-1] Implementations **MUST** support unmarshalling of documents conforming to RFC 7159 JSON Grammar [4]. [JSB-3.1-2] In addition, implementations **SHOULD NOT** allow unmarshalling of RFC 7159 [4] non-conforming text (e.g. unsupported encoding, ...) and report error in such cases. [JSB-3.1-3] Detection of UTF encoding of an unmarshalled document **MUST** follow the encoding process defined in the Section 3 (Encoding) of RFC 4627 [3]. [JSB-3.1-4] Implementations **SHOULD** ignore the presence of an UTF byte order mark (BOM) and not treat it as an error.[JSB-3.1-5]

### 3.2 Errors

Implementations **SHOULD NOT** allow unmarshalling of RFC 7159 [4] non-conforming text (e.g. unsupported encoding, ...) and report an error in such case. [JSB-3.2-1] Implementations **SHOULD** also report an error during an unmarshalling operation, if it is not possible to represent a JSON document value with the expected Java type. [JSB-3.2-2]

### 3.3 Basic Java Types

Implementations **MUST** support binding of the following basic Java classes and their corresponding primitive types: [JSB-3.3-1]

- java.lang.String
- java.lang.Character
- java.lang.Byte

- `java.lang.Short`
- `java.lang.Integer`
- `java.lang.Long`
- `java.lang.Float`
- `java.lang.Double`
- `java.lang.Boolean`

### 3.3.1 `java.lang.String`, `Character`

Instances of type `java.lang.String` and `java.lang.Character` are marshalled to JSON String values as defined within RFC 7159 Section 7 (Strings) [4] in UTF-8 encoding without a byte order mark. [JSB-3.3.1-1] Implementations SHOULD support unmarshaling of JSON text in other (than UTF-8) UTF encodings into `java.lang.String` instances. [JSB-3.3.1-2]

### 3.3.2 `java.lang.Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`

Marshalling of type `java.lang.Byte`, `Short`, `Integer`, `Long`, `Float` or `Double` (and their corresponding primitive types) to a JSON Number MUST follow the conversion process defined in the javadoc specification for the corresponding type's `toString()` method [JSB-3.3.2-1]. Unmarshalling of a JSON value into `java.lang.Byte`, `Short`, `Integer`, `Long`, `Float` or `Double` instance (or their corresponding primitive types) MUST follow the conversion process defined in the javadoc specification for the corresponding `parse$Type` method, such as `java.lang.Byte.parseByte()` for `Byte`. [JSB-3.3.2-2]

### 3.3.3 `java.lang.Boolean`

Marshalling of type `java.lang.Boolean` and its corresponding boolean primitive type to a JSON value MUST follow the conversion process defined in the javadoc specification for `java.lang.Boolean.toString()` method [JSB-3.3.3-1]. Unmarshalling of a JSON value into `java.lang.Boolean` instance or boolean primitive type MUST follow the conversion process defined in the javadoc specification for `java.lang.Boolean.parseBoolean()` method. [JSB-3.3.3-2]

### 3.3.4 `java.lang.Number`

Marshalling of `java.lang.Number` instances (if their more concrete type is not defined elsewhere in this chapter) to a JSON string MUST retrieve double value from `java.lang.Number.doubleValue()` method and convert it to a JSON Number as defined in subsection 3.3.2 `java.lang.Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`. [JSB-3.3.4-1]. Unmarshalling of a JSON value into `java.lang.Number` type MUST return an instance of `java.math.BigDecimal` by using conversion process defined in the javadoc specification for constructor of `java.math.BigDecimal` with `java.lang.String` argument. [JSB-3.3.4-2]

## 3.4 Specific Standard Java SE Types

Implementations MUST support binding of the following standard Java SE classes: [JSB-3.4-1]



- `java.math.BigInteger`
- `java.math.BigDecimal`
- `java.net.URL`
- `java.net.URI`
- `java.util.Optional`
- `java.util.OptionalInt`
- `java.util.OptionalLong`
- `java.util.OptionalDouble`

### 3.4.1 `java.math.BigInteger`, `BigDecimal`

Marshalling of type `java.math.BigInteger` or `BigDecimal` to a JSON Number MUST follow the conversion process defined in the javadoc specification for the corresponding type's `toString()` method [JSB-3.4.1-1]. Unmarshalling of a JSON value into `java.math.BigInteger` or `BigDecimal` instance MUST follow the conversion process defined in the javadoc specification for the constructor of `java.math.BigInteger` or `BigDecimal` with `java.lang.String` argument. [JSB-3.4.1-2]

### 3.4.2 `java.net.URL`, `URI`

Marshalling of type `java.net.URL` or `URI` to a JSON String MUST follow the conversion process defined in the javadoc specification for the corresponding type's `toString()` method [JSB-3.4.2-1]. Unmarshalling of a JSON value into `java.net.URL` or `URI` instance MUST follow the conversion process defined in the javadoc specification for the constructor of `java.net.URL` or `URI` with `java.lang.String` argument. [JSB-3.4.2-2]

### 3.4.3 `java.util.Optional`, `OptionalInt`, `OptionalLong`, `OptionalDouble`

Non-empty instances of type `java.util.Optional`, `OptionalInt`, `OptionalLong`, `OptionalDouble` are marshalled to a JSON value by retrieving their contained instance and converting it to JSON value based on its type and corresponding mapping definitions within this chapter. [JSB-3.4.3-1] Empty optional instances marshalled as object instance properties are ignored during marshalling. [JSB-3.4.3-2] Empty optional instances marshalled as JSON array elements are marshalled as null values [JSB-3.4.3-3]. Unmarshalling into `Optional`, `OptionalInt`, `OptionalLong`, `OptionalDouble` returns empty optional value for properties which are not present in JSON document or contain a null value. [JSB-3.4.3-4] Otherwise any non-empty `Optional`, `OptionalInt`, `OptionalLong`, `OptionalDouble` value is constructed of type which unmarshalled based on mappings defined in this chapter.[JSB-3.4.3-5]

Instances of type `java.util.Optional<T>` are marshalled to a JSON value as JSON objects when T alone would be marshalled as JSON object. When T would be marshalled as a JSON value (e.g. `java.lang.String`, `java.lang.Integer`), an instance of `java.util.Optional<T>` is marshalled as a JSON value (without curly brackets). [JSB-3.4.3-6]

Unmarshalling of a JSON value into `java.util.Optional<T>` MUST be supported if unmarshalling of a JSON value into instance of T is supported. [JSB-3.4.3-7]

## 3.5 Dates

Implementations MUST support binding of the following standard Java date/time classes: [JSB-3.5-1]

- `java.util.Date`
- `java.util.Calendar`
- `java.util.GregorianCalendar`
- `java.util.TimeZone`
- `java.util.SimpleTimeZone`
- `java.time.Instant`
- `java.time.Duration`
- `java.time.Period`
- `java.time.LocalDate`
- `java.time.LocalDateTime`
- `java.time.ZonedDateTime`
- `java.time.ZoneId`
- `java.time.ZoneOffset`
- `java.time.OffsetDateTime`
- `java.time.OffsetTime`

If not specified otherwise in this section, GMT standard time zone and offset specified from UTC Greenwich is used. [JSB-3.5-2] If not specified otherwise, the date time format for marshalling and unmarshalling is ISO 8601 without offset, as specified in `java.time.format.DateTimeFormatter.ISO_DATE`. [JSB-3.5-3] Implementations MUST report an error if the datetime string in a JSON document does not correspond to the expected datetime format. [JSB-3.5-4]

### 3.5.1 `java.util.Date`, `Calendar`, `GregorianCalendar`

The marshalling format of `java.util.Date`, `Calendar`, `GregorianCalendar` instances with no time information is ISO\_DATE. [JSB-3.5.1-1]. If time information is present, the format is ISO\_DATE\_TIME [JSB-3.5.1-2].

Implementations MUST support unmarshalling of both ISO\_DATE and ISO\_DATE\_TIME into `java.util.Date`, `Calendar`, `GregorianCalendar` instances. [JSB-3.5.1-3]

### 3.5.2 `java.util.TimeZone`, `SimpleTimeZone`

Implementations MUST support unmarshalling of any time zone format specified in `java.util.TimeZone` into a field or property of type `java.util.TimeZone`, `SimpleTimeZone`. [JSB-3.5.2-1] Implementations MUST report an error for deprecated three-letter time zone IDs as specified in `java.util.Timezone`. [JSB-3.5.2-2] The marshalling format of `java.util.TimeZone` and `SimpleTimeZone` is NormalizedCustomID as specified in `java.util.TimeZone`. [JSB-3.5.2-3]

### 3.5.3 java.time.\*

The marshalling output for a `java.time.Instant` instance **MUST** be in a `ISO_INSTANT` format, as specified in `java.time.format.DateTimeFormatter`. [JSB-3.5.3-1] Implementations **MUST** support the unmarshalling of an `ISO_INSTANT` formatted JSON string to a `java.time.Instant` instance. [JSB-3.5.3-2]

For other `java.time.*` classes, the following mapping table maps Java types to their corresponding formats: [JSB-3.5.3-3]

Java Type	Format
<code>java.time.Instant</code>	<code>ISO_INSTANT</code>
<code>java.time.LocalDate</code>	<code>ISO_LOCAL_DATE</code>
<code>java.time.LocalTime</code>	<code>ISO_LOCAL_TIME</code>
<code>java.time.LocalDateTime</code>	<code>ISO_LOCAL_DATE_TIME</code>
<code>java.time.ZonedDateTime</code>	<code>ISO_ZONED_DATE_TIME</code>
<code>java.time.OffsetDateTime</code>	<code>ISO_OFFSET_DATE_TIME</code>
<code>java.time.OffsetTime</code>	<code>ISO_OFFSET_TIME</code>

Implementations **MUST** support the unmarshalling of any time zone ID format specified in `java.time.ZoneId` into a field or property of type `java.time.ZoneId`. [JSB-3.5.3-4] The marshalling format of `java.time.ZoneId` is the normalized zone ID as specified in `java.time.ZoneId`. [JSB-3.5.3-5]

Implementations **MUST** support the unmarshalling of any time zone ID format specified in `java.time.ZoneOffset` into a field or property of type `java.time.ZoneOffset`. [JSB-3.5.3-6] The marshalling format of `java.time.ZoneOffset` is the normalized zone ID as specified in `java.time.ZoneOffset`. [JSB-3.5.3-7]

Implementations **MUST** support the unmarshalling of any duration format specified in `java.time.Duration` into a field or property of type `java.time.Duration`. [JSB-3.5.3-8] This is super-set of ISO 8601 duration format. The marshalling format of `java.time.Duration` is the ISO 8601 seconds based representation, such as `PT8H6M12.345S`. [JSB-3.5.3-9]

Implementations **MUST** support the unmarshalling of any period format specified in `java.time.Period` into a field or property of type `java.time.Period`. [JSB-3.5.3-10] This is a super-set of ISO 8601 period format. The marshalling format of `java.time.Period` is ISO 8601 period representation. [JSB-3.5.3-11] A zero-length period is represented as zero days `'P0D'`. [JSB-3.5.3-12]

## 3.6 Untyped mapping

For an unspecified output type of an unmarshal operation, as well as where output type is specified as `Object.class`, implementations **MUST** unmarshal a JSON document using Java runtime types specified in table below: [JSB-3.6-1]

JSON value	Java type
object	<code>java.util.Map &lt;String, Object &gt;</code>
array	<code>java.util.List &lt;Object &gt;</code>
string	<code>java.lang.String</code>
number	<code>java.lang.Integer</code> — <code>Long</code> — <code>java.math.BigDecimal</code>
true, false	<code>java.lang.Boolean</code>
null	<code>null</code>

JSON object values are unmarshalled into an implementation of `java.util.Map <String, Object >` with a predictable iteration order. [JSB-3.6-2]

JSON number values are unmarshalled into the smallest of types Integer, Long, BigDecimal that can hold the value represented by number without loss of value or precision.[JSB-3.6-3]

## 3.7 Java Class

Any instance passed to an unmarshalling operation must have a public or protected no-argument constructor. Implementations SHOULD throw an error if this condition is not met. [JSB-3.7-1] This limitation does not apply to marshalling operations. [JSB-3.7-2]

### 3.7.1 Scope and Field access strategy

For an unmarshalling operation of a Java property, if a matching public setter method exists, the method is called to set the value of the property. If a matching setter method with private, protected, or defaulted to package-only access exists, then this field is ignored. If no matching setter method exists and the field is public, then direct field assignment is used. [JSB-3.7.1-1]

For a marshalling operation, if a matching public getter method exists, the method is called to obtain the value of the property. If a matching getter method with private, protected, or defaulted to package-only access exists, then this field is ignored. If no matching getter method exists and the field is public, then the value is obtained directly from the field. [JSB-3.7.1-2]

JSON Binding implementations MUST NOT unmarshal into transient, final or static fields and MUST ignore name/value pairs corresponding to such fields. [JSB-3.7.1-3]

Implementations MUST support marshalling of final fields. [JSB-3.7.1-4] Transient and static fields MUST be ignored during marshalling operation. [JSB-3.7.1-5]

If a JSON document contains a name/value pair not corresponding to field or setter method, then this name/value pair MUST be ignored. [JSB-3.7.1-6]

Public getter/setter methods without a corresponding field MUST be supported. When only public getter/setter method without corresponding field is present in the class, the getter method is called to obtain the value to marshall, and the setter method is called during unmarshall operation. [JSB-3.7.1-7]

### 3.7.2 Nested Classes

Implementations MUST support the binding of public and protected nested classes. [JSB-3.7.2-1] For unmarshalling operations, both nested and encapsulating classes MUST fulfill the same instantiation requirements as specified in subsection 3.7.1 Scope and Field access strategy. [JSB-3.7.2-2]

### 3.7.3 Static Nested Classes

Implementations MUST support the binding of public and protected static nested classes. [JSB-3.7.3-1] For unmarshalling operations, the nested class MUST fulfill the same instantiation requirements as specified in subsection 3.7.1 Scope and Field access strategy. [JSB-3.7.3-2]

### 3.7.4 Anonymous Classes

Unmarshalling into anonymous classes is not supported. Marshalling of anonymous classes is supported by default object mapping. [JSB-3.7.2-1]

## 3.8 Polymorphic Types

Unmarshalling into polymorphic types is not supported by default mapping. [JSB-3.8-1]

### 3.9 Enum

Marshalling of an Enum instance to a JSON String value **MUST** follow the conversion process defined in javadoc specification for their toString method [JSB-3.9-1]. Unmarshalling of a JSON value into an enum instance **MUST** be done by calling the enum's valueOf(String) method. [JSB-3.9-2]

### 3.10 Interfaces

Implementations **MUST** support the unmarshalling of specific interfaces defined in section 3.11 Collections, and subsection 3.3.4 java.lang.Number. [JSB-3.10-1] Unmarshalling to other interfaces is not supported and implementations **SHOULD** report error in such case. [JSB-3.10-2] If a class property is defined with an interface and not concrete type, then the mapping for a marshalled the property is resolved based on its runtime type.[JSB-3.10-3]

### 3.11 Collections

Implementations **MUST** support the binding of the following collection interfaces, classes and their implementations. [JSB-3.11-1] Implementations of these interfaces must provide an accessible default constructor. JSON Binding implementations **MUST** report an unmarshalling error if a default constructor is not present or is not in accessible scope. [JSB-3.11-2]

- java.util.Collection
- java.util.Map
- java.util.Set
- java.util.HashSet
- java.util.NavigableSet
- java.util.SortedSet
- java.util.TreeSet
- java.util.LinkedHashSet
- java.util.TreeHashSet
- java.util.HashMap
- java.util.NavigableMap

- `java.util.SortedMap`
- `java.util.TreeMap`
- `java.util.LinkedHashMap`
- `java.util.TreeHashMap`
- `java.util.List`
- `java.util.ArrayList`
- `java.util.LinkedList`
- `java.util.Deque`
- `java.util.ArrayDeque`
- `java.util.Queue`
- `java.util.PriorityQueue`
- `java.util.EnumSet`
- `java.util.EnumMap`

## 3.12 Arrays

JSON Binding implementations **MUST** support the binding of Java arrays of all supported Java types from this chapter into/from JSON array structures as defined in Section 5 of RFC 7159 [4]. [JSB-3.12-1] Arrays of primitive types and multi-dimensional arrays **MUST** be supported. [JSB-3.6-2]

## 3.13 Attribute order

Declared fields **MUST** be marshalled in lexicographical order into the resulting JSON document. In case of inheritance, declared fields of super class **MUST** be marshalled before declared fields of child class. [JSB-3.13-1]

When unmarshalling a JSON document, declared fields **MUST** be set in the order of attributes present in the JSON document. [JSB-3.13-2]

## 3.14 Null value handling

### 3.14.1 Null Java field

The result of marshalling a java field with a null value is the absence of the property in the resulting JSON document. [JSB-3.14.1-1] The unmarshalling operation of a property absent in JSON document **MUST** not set the value of the field, the setter (if available) **MUST** not be called, and thus original value of the field **MUST** be preserved. [JSB-3.14.1-2]

The unmarshalling operation of a property with a null value in a JSON document **MUST** set the value of the field to null value (or call setter with null value if setter is present). [JSB-3.14.1-3]

### 3.14.2 Null Array Values

The result of unmarshalling n-ary array represented in JSON document is n-ary Java array. [JSB-3.14.2-1]. Null value in JSON array is represented by null values in Java array. [JSB-3.14.2-2] Marshalling operation on Java array with null value at index i MUST output null value at index i of the array in resulting JSON document. [JSB-3.14.2-3]

## 3.15 Names and identifiers

According to RFC 7159 Section 7 [4], every Java identifier name can be transformed using identity function into a valid JSON String. Identity function MUST be used for transforming Java identifier names into name Strings in JSON document. [JSB-3.15-1] For unmarshal operations defined in section 3.6 Untyped mapping section, identity function is used to transform JSON name strings into Java String instances in the resulting map `Map<String, Object>`. [JSB-3.15-2] Identity function is used also for other unmarshalling operations. [JSB-3.15-3] If a Java identifier with corresponding name does not exist or is not accessible, the implementations MUST report error. [JSB-3.15-4] Naming and error reporting strategies can be further customized in chapter 4 Customizing Mapping.

## 3.16 Generics

JSON Binding implementations MUST support binding of generic types. [JSB-3.16-1] Due to type erasure, there are situations when it is not possible to obtain generic type information.

There are two ways for JSON Binding implementations to obtain generic type information. If there is a class file available (in the following text referred as static type information), it is possible to obtain generic type information (effectively generic type declaration) from Signature attribute (if this information is present). [JSB-3.16-2] The second option is to provide generic type information at runtime. To provide generic type information at runtime, an argument of `java.lang.reflect.Type` MUST be passed to `Jsonb::toJson` or to `Jsonb::fromJson` method. [JSB-3.16-3]

## 3.17 Big numbers

JSON Binding implementation MUST serialize numbers that express greater magnitude or precision than an IEEE 754 double precision number as strings. [JSB-3.17-1]

### 3.17.1 Type resolution algorithm

There are several levels of information JSON Binding implementations may obtain about the type of field/class/interface: [JSB-3.17.1-1]

1. runtime type provided via `java.lang.reflect.Type` parameter passed to `Jsonb::toJson` or `Jsonb::fromJson` method
2. static type provided in class file (effectively stored in Signature attribute)
3. raw type

## 4. no information about the type

If there is no information about the type, JSON Binding implementation **MUST** treat this type as `java.lang.Object`. [JSB-3.17.1-2] If only raw type of given field/class/interface is known, then the type **MUST** be treated like raw type. [JSB-3.17.1-3] For example, if the only available information is that given field/class/interface is of type `java.util.ArrayList`, then the type **MUST** be treated as `java.util.ArrayList<Object>`.

JSON Binding implementations **MUST** use the most specific type derived from the information available. [JSB-3.17.1-4]

Let's consider situation where there is only static type information of a given field/class/interface known, and there is no runtime type information available. Let `GenericClass<T1, ..., Tn>` be part of generic type declaration, where `GenericClass` is name of the generic type and `T1, ..., Tn` are type parameters. For every `Ti`, where `i` in `1, ..., n`, there are 3 possible options: [JSB-3.17.1-5]

1. `Ti` is concrete parameter type
2. `Ti` is bounded parameter type
3. `Ti` is wildcard parameter type without bounds

In case 1, the most specific parameter type **MUST** be given concrete parameter type `Ti`. [JSB-3.17.1-6]

For bounded parameter type, using bounds `B1, ..., Bm`. If `m = 1`, then the most specific parameter type **MUST** be derived from the given bound `B1`. [JSB-3.17.1-7] If `B1` is class or interface, the most specific parameter type **MUST** be the class or interface. [JSB-3.17.1-8] Otherwise, the most specific parameter type **SHOULD** be `java.lang.Object`. [JSB-3.17.1-9]

If multiple bounds are specified, the first step is to resolve every bound separately. Let's define result of such resolution as `S1, ..., Sm` specific parameter types. If `S1, ..., Sm` are `java.lang.Object`, then the bounded parameter type `Ti` **MUST** be `java.lang.Object`. [JSB-3.17.1-10] If there is exactly one `Sk`, where `1 ≤ k ≤ m` is different than `java.lang.Object`, then the most specific parameter type for this bounded parameter type `Ti` **MUST** be `Sk`. [JSB-3.17.1-11] If there exists `Sk1, Sk2`, where `1 ≤ k1 ≤ k2 ≤ m`, then the most specific parameter type is `Sk1`. [JSB-3.17.1-12]

For wildcard parameter type without bounds, the most specific parameter type **MUST** be `java.lang.Object`. [JSB-3.17.1-13]

Any unresolved type parameter **MUST** be treated as `java.lang.Object`. [JSB-3.17.1-14]

If runtime type is provided via `java.lang.reflect.Type` parameter passed to `Jsonb::toJson` or `Jsonb::fromJson` method, than that runtime type overrides static type declaration wherever applicable. [JSB-3.17.1-15]

There are situations when it is necessary to use combination of runtime and static type information.

Figure 3.1: Example Type resolution

```

1 public class MyGenericType<T,U> {
2     public T field1;
3     public U field2;
4 }

```

To resolve type of `field1`, runtime type of `MyGenericType` and static type of `field1` is required.



### 3.18 Must-Ignore policy

When JSON Binding implementation encounters key in key/value pair that it does not recognize, it should treat the rest of the JSON document as if the element simply did not appear, and in particular, the implementation **MUST NOT** treat this as an error condition. [JSB-3.18-1]

### 3.19 Uniqueness of properties

JSON Binding implementations **MUST NOT** produce JSON documents with members with duplicate names. In this context, "duplicate" means that the names, after processing any escaped characters, are identical sequences of Unicode characters. [JSB-3.19-1]

When non-unique property (after override and rename) is found, implementation **MUST** throw an exception. This doesn't apply for customized user serialization behavior implemented with the usage of `JsonbAdapter` mechanism. [JSB-3.19-2]

### 3.20 JSON Processing integration

JSON Binding implementations **MUST** support binding of the following JSON Processing types. [JSB-3.20-1]

- `javax.json.JsonObject`
- `javax.json.JsonArray`
- `javax.json.JsonStructure`
- `javax.json.JsonValue`
- `javax.json.JsonPointer`
- `javax.json.JsonString`
- `javax.json.JsonNumber`

Marshalling of supported `javax.json.*` objects/interfaces/fields **MUST** have the same result as marshalling these objects with `javax.json.JsonWriter`. [JSB-3.20-2] Unmarshalling into supported `javax.json.*` objects/interfaces/fields **MUST** have the same result as unmarshalling into such objects with `javax.json.JsonReader`. [JSB-3.20-3]

DRAFT

# Chapter 4

## Customizing Mapping

This section defines several ways how to customize the default behavior. The default behavior can be customized annotating a given field (or JavaBean property), or by providing an implementation of particular strategy, e.g. `PropertyOrderStrategy`. JSON Binding provider MUST support both these customization options. [JSB-4-1]

### 4.1 Customizing Property Names

There are two standard ways how to customize serialization of field (or JavaBean property) to JSON document. The same applies to deserialization. The first way is to annotate field (or JavaBean property) with `javax.json.bind.annotation.JsonbProperty` annotation. The second option is to set `javax.json.bind.config.PropertyNamingPolicy`.

#### 4.1.1 `javax.json.bind.annotation.JsonbTransient`

JSON Binding implementations MUST NOT serialize fields (or JavaBean properties) annotated with `javax.json.bind.annotation.JsonbTransient`. When placed on a class, indicates that the class shouldn't be mapped to JSON by itself. Properties on such class will be mapped to JSON along with its derived classes, as if the class is inlined. [JSB-4.1.1-1]

`JsonbTransient` annotation is mutually exclusive with all other JSON Binding defined annotations. If this condition is not met, JSON Binding implementation MUST throw `JsonbException`. [JSB-4.1.1-2]

#### 4.1.2 `javax.json.bind.annotation.JsonbProperty`

According to default mapping `refsec:naming`, property names are serialized unchanged to JSON document (identity transformation). To provide custom name for given field (or JavaBean property), `javax.json.bind.annotation.JsonbProperty` may be used. `JsonbProperty` annotation may be specified on field, getter or setter method. If specified on field, custom name is used both for serialization and deserialization. If `javax.json.bind.annotation.JsonbProperty` is specified on getter method, it is used only for serialization. If `javax.json.bind.annotation.JsonbProperty` is specified on setter method, it is used only for deserialization. It is possible to specify different values for getter and setter method for `javax.json.bind.annotation.JsonbProperty` annotation. In such case the different custom name will be used for serialization and deserialization. [JSB-4.1.2-1]

### 4.1.3 `javax.json.bind.config.PropertyNamingStrategy`

To customize name translation of properties, JSON Binding provides `javax.json.bind.config.PropertyNamingStrategy` interface.

Interface `javax.json.bind.config.PropertyNamingStrategy` provides the most common property naming strategies.

- `IDENTITY`
- `LOWER_CASE_WITH_DASHES`
- `LOWER_CASE_WITH_UNDERSCORES`
- `UPPER_CAMEL_CASE`
- `UPPER_CAMEL_CASE_WITH_SPACES`
- `CASE_INSENSITIVE`

The detailed description of property naming strategies can be found in javadoc.

The way to set custom property naming strategy is to use `javax.json.bind.JsonbConfig::withPropertyNamingStrategy` method. [JSB-4.1.2-1]

### 4.1.4 Property names resolution

Property name resolution consist of two phases:

1. Standard override mechanism
2. Applying property name resolution, which involves the value of `@JsonbProperty`

If duplicate name is found exception **MUST** be thrown. The definition of duplicate (non-unique) property can be found in 3.19. [JSB-4.1.4-1]

## 4.2 Customizing Property Order

To customize order of serialized properties, JSON Binding provides `javax.json.bind.config.PropertyOrderStrategy` class.

Class `javax.json.bind.config.PropertyOrderStrategy` provides the most common property order strategies.

- `LEXICOGRAPHICAL`
- `ANY`
- `REVERSE`

The detailed description of property order strategies can be found in javadoc.

The way to set custom property order strategy is to use `javax.json.bind.JsonbConfig::withPropertyOrderStrategy` method. [JSB-4.2-2]

To customize order of serialized properties only for one specific type, JSON Binding provides `javax.json.bind.annotation.JsonbPropertyOrder` annotation. Order specified by `JsonbPropertyOrder` annotation overrides order specified by `PropertyOrderStrategy`. [JSB-4.2-3]

The order is applied to already renamed properties as stated in 4.1.

## 4.3 Customizing Null Handling

There are three ways how to change default null handling. The first option is to annotate type or package with `javax.json.bind.annotation.JsonbNillable` annotation. The second option is to annotate field or JavaBean property with `javax.json.bind.annotation.JsonbProperty` and to set nillable parameter to true. The third option is to set config wide configuration via `JsonbConfig::withNullValues` method.

If annotations (`JsonbNillable` or `JsonbProperty`) on different level apply to the same field (or JavaBean property) or if there is config wide configuration and some annotation (`JsonbNillable` or `JsonbProperty`) which apply to the same field (or JavaBean property), the annotation with the smallest scope applies. For example, if there is type level `JsonbNillable` annotation applied to some class with field which is annotated with `JsonbProperty` annotation with `nillable = false`, then `JsonbProperty` annotation overrides `JsonbNillable` annotation.

### 4.3.1 `javax.json.bind.annotation.JsonbNillable`

To customize the result of serializing field (or JavaBean property) with null value, JSON Binding provides `javax.json.bind.annotation.JsonbNillable` and `javax.json.bind.annotation.JsonbProperty` annotations.

When given object (type or package) is annotated with `javax.json.bind.annotation.JsonbNillable` annotation, the result of null value will be presence of associated property in JSON document with explicit null value. [JSB-4.3.1-1]

The same behavior as `JsonbNillable`, but only at field, parameter and method (JavaBean property) level is provided by `javax.json.bind.annotation.JsonbProperty` annotation with its `nillable` parameter. [JSB-4.3.1-2]

JSON Binding implementations MUST implement override of annotations according to target of the annotation (FIELD, PARAMETER, METHOD, TYPE, PACKAGE). Type level annotation overrides behavior set at the package level. Method, parameter or field level annotation overrides behavior set at the type level. [JSB-4.3.1-3]

### 4.3.2 Global null handling configuration

Null handling behavior can be customized via `javax.json.bind.JsonbConfig::withNullValues` method.

The way to enforce serialization of null values, is to call method `javax.json.bind.JsonbConfig::withNullValues` with parameter true.

The way to skip serialization of null values is to call method `javax.json.bind.JsonbConfig::withNullValues` with parameter false. [JSB-4.3.2-1]

## 4.4 I-JSON support

I-JSON (short for "Internet JSON") is a restricted profile of JSON designed to maximize interoperability and increase confidence that software can process it successfully with predictable results. The profile is defined in [4].

JSON Binding provides full support for I-JSON standard. Without any configuration, JSON Binding produces JSON documents which are compliant with I-JSON with three exceptions.

- JSON Binding does not restrict the serialization of top-level JSON texts that are neither objects nor

arrays. The restriction should happen at application level.

- JSON Binding does not serialize binary data with base64url encoding.
- JSON Binding does not enforce additional restrictions on dates/times/duration.

These exceptions refer only to recommended areas of I-JSON.

To enforce strict compliance of serialized JSON documents, JSON Binding implementations **MUST** implement configuration option "jsonb.i-json.strict-ser-compliance". [JSB-4.4-1]

The way to enable strict compliance of serialized JSON documents, is to call method `JsonbConfig::withStrictIJSONSerializationCompliance` with parameter `true`.

#### 4.4.1 Strict date serialization

Uppercase rather than lowercase letters **MUST** be used. [JSB-4.4.1-1] The timezone **MUST** always be included and optional trailing seconds **MUST** be included even when their value is "00". [JSB-4.4.1-2]

JSON Binding implementations **MUST** serialize `java.util.Date`, `java.util.Calendar`, `java.util.GregorianCalendar`, `java.time.LocalDate`, `java.time.LocalDateTime` and `java.time.Instant` in the same format as `java.time.ZonedDateTime`. [JSB-4.4.1-3]

The result of serialization of duration must conform to the "duration" production in Appendix A of RFC 3339, with the same additional restrictions. [JSB-4.4.1-4]

#### 4.5 Simple values

Using `javax.json.bind.annotation.JsonbValue` annotation, a class can be mapped to a simple value. Class can contain at most one mapped property or field that is annotated with `javax.json.bind.annotation.JsonbValue`. [JSB-4.5-1]

Annotation `javax.json.bind.annotation.JsonbValue` indicates that result of the annotated non-void method or field or constructor parameter will be used as the single value to serialize for the instance. [JSB-4.5-2]

#### 4.6 Custom instantiation

In many scenarios instantiation with the use of default constructor is not enough. To support these scenarios, JSON Binding provides `javax.json.bind.annotation.JsonbCreator` annotation.

`JsonbCreator` annotation can be used to annotation custom constructor or static void factory method.

#### 4.7 Custom visibility

To customize scope and field access strategy as specified in section 3.7.1, it is possible to specify `javax.json.bind.annotation.JsonbVisibility` annotation or to override default behavior globally calling `JsonbConfig::withPropertyVisibilityStrategy` method with given custom property visibility strategy. [JSB-4.7-1]

## 4.8 Custom mapping

To provide custom mapping for specific java type, it is necessary to implement `javax.json.bind.adapter.JsonbAdapter` interface. [JSB-4.8-1]

There are two ways how to register custom `JsonbAdapter`. Using `JsonbConfig::withAdapters` method or annotating specific field or `JavaBean` property with `JsonbJavaTypeAdapter` annotation. [JSB-4.8-2]

`JsonbAdapter` registered via `JsonbConfig::withAdapters` is visible to all serialize/deserialize operations performed with given `JsonbConfig`. `JsonbAdapter` registered with annotation is visible to serialize/deserialize operation used only for given field/`JavaBean` property annotated. [JSB-4.8-3]

## 4.9 Custom date format

To specify custom date format, it is necessary to annotate given annotation target with `javax.json.bind.annotation.JsonbDateFormat` annotation. [JSB-4.9-1] `JsonbDateFormat` annotation can be applied to the following targets:

- field
- method
- type
- parameter
- package

Annotation applied to more specific target overrides the same annotation applied to target with wider scope. For example, annotation applied to type target will override the same annotation applied to package target. [JSB-4.9-2]

## 4.10 Custom number format

To specify custom number format, it is necessary to annotate given annotation target with `javax.json.bind.annotation.JsonbNumberFormat` annotation. [JSB-4.10-1] `JsonbNumberFormat` annotation can be applied to the following targets:

- field
- method
- type
- parameter
- package

Annotation applied to more specific target overrides the same annotation applied to target with wider scope. For example, annotation applied to type target will override the same annotation applied to package target. [JSB-4.10-2]

## 4.11 Custom binary data handling

To customize encoding of binary data, JSON Binding provides `javax.json.bind.config.BinaryDataStrategy` class.

Class `javax.json.bind.config.BinaryDataStrategy` provides the most common binary data encodings.

- `BYTE`
- `BASE_64`
- `BASE_64_URL`

The detailed description of binary encoding strategies can be found in javadoc.

The way to set custom binary data handling strategy is to use `javax.json.bind.JsonbConfig::withBinaryDataStrategy` method. [JSB-4.11-1]



# Bibliography

- [1] Scott Bradner. Key words for use in rfcs to indicate requirement levels. RFC, IETF, March 1997.
- [2] T. Berners-Lee, R. Fielding, and L. Masinter. RFC 3986: Uniform Resource Identifier (URI): Generic Syntax. RFC, IETF, January 2005. See <http://www.ietf.org/rfc/rfc3986.txt>.
- [3] Douglas Crockford. The application/json media type for javascript object notation (json). RFC, IETF, July 2006.
- [4] Ed. T. Bray. The javascript object notation (json) data interchange format. RFC 2070-1721, IETF, March 2014.
- [5] JSON. Web site, Wikipedia. See <http://en.wikipedia.org/wiki/JSON>.
- [6] Representational State Transfer. Web site, Wikipedia. See [http://en.wikipedia.org/wiki/Representational\\_State\\_Transfer](http://en.wikipedia.org/wiki/Representational_State_Transfer).
- [7] R. Fielding. Architectural Styles and the Design of Network-based Software Architectures. Ph.d dissertation, University of California, Irvine, 2000. See <http://roy.gbiv.com/pubs/dissertation/top.htm>.
- [8] REST Wiki. Web site. See <http://rest.blueoxen.net/cgi-bin/wiki.pl>.
- [9] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification - second edition. Book, Sun Microsystems, Inc, 2000. [http://java.sun.com/docs/books/jls/second\\_edition/html/j.title.doc.html](http://java.sun.com/docs/books/jls/second_edition/html/j.title.doc.html).
- [10] Kohsuke Kawaguchi. The Java Architecture for XML Binding (JAXB). JSR, JCP, December 2009. See <http://jcp.org/en/jsr/detail?id=222>.
- [11] Rajiv Mordani. Common Annotations for the Java Platform. JSR, JCP, July 2005. See <http://jcp.org/en/jsr/detail?id=250>.
- [12] Emmanuel Bernard. Bean Validation 1.1. JSR, JCP, March 2013. See <http://jcp.org/en/jsr/detail?id=349>.
- [13] Kin-Man Chung. Java API for JSON Processing. JSR, JCP, 2015. See <http://jcp.org/en/jsr/detail?id=374>.
- [14] Marek Potociar Santiago Pericas-Geertsens. Java API for RESTful Web Services. JSR, JCP, 2015. See <http://jcp.org/en/jsr/detail?id=370>.