JAKARTA EE

Jakarta JSON Binding

Jakarta JSON Binding Team, https://projects.eclipse.org/projects/ee4j.jsonb

2.0-RC2, июня 08, 2020: Milestone Draft

Table of Contents

```
Specification: Jakarta JSON Binding

Version: 2.0-RC2

Status: Milestone Draft

Release: июня 08, 2020
```


Eclipse Foundation Specification License

By using and/or copying this document, or the Eclipse Foundation document from which this statement is linked, you (the licensee) agree that you have read, understood, and will comply with the following terms and conditions:

Permission to copy, and distribute the contents of this document, or the Eclipse Foundation document from which this statement is linked, in any medium for any purpose and without fee or royalty is hereby granted, provided that you include the following on ALL copies of the document, or portions thereof, that you use:

- link or URL to the original Eclipse Foundation document.
- All existing copyright notices, or if one does not exist, a notice (hypertext is preferred, but a textual representation is permitted) of the form: "Copyright (c) [$date-of-document] Eclipse Foundation, Inc. [url to this license]"

Inclusion of the full text of this NOTICE must be provided. We request that authorship attribution be provided in any software, documents, or other items or products that you create pursuant to the implementation of the contents of this document, or any portion thereof.

No right to create modifications or derivatives of Eclipse Foundation documents is granted pursuant to this license, except anyone may prepare and distribute derivative works and portions of this document in software that implements the specification, in supporting materials accompanying such software, and in documentation of such software, PROVIDED that all such works include the notice below. HOWEVER, the publication of derivative works of this document for use as a technical specification is expressly prohibited.

The notice is:

"Copyright (c) 2018 Eclipse Foundation. This software or document includes material copied from or derived from [title and URI of the Eclipse Foundation specification document]."

Disclaimers

THIS DOCUMENT IS PROVIDED "AS IS," AND THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION MAKE NO REPRESENTATIONS OR WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE, NON-INFRINGEMENT, OR TITLE; THAT THE CONTENTS OF THE DOCUMENT ARE SUITABLE FOR ANY PURPOSE; NOR THAT THE IMPLEMENTATION OF SUCH CONTENTS WILL NOT INFRINGE ANY THIRD PARTY PATENTS, COPYRIGHTS, TRADEMARKS OR OTHER RIGHTS.

THE COPYRIGHT HOLDERS AND THE ECLIPSE FOUNDATION WILL NOT BE LIABLE FOR ANY DIRECT, INDIRECT, SPECIAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF ANY USE OF THE DOCUMENT OR THE PERFORMANCE OR IMPLEMENTATION OF THE CONTENTS THEREOF.

The name and trademarks of the copyright holders or the Eclipse Foundation may NOT be used in advertising or publicity pertaining to this document or its contents without specific, written prior permission. Title to copyright in this document will at all times remain with copyright holders.

Jakarta JSON Binding Specification, Version 1.0

Chapter 1. Introduction

This specification defines binding API between Java objects and JSON 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
- JSON on Wikipedia

1.1. Status

A list of open issues can be found at:

> https://github.com/eclipse-ee4j/jsonb-api/issues

The corresponding source code can be found online at:

> https://github.com/eclipse-ee4j/jsonb-api

The committer group is seeking feedback from the community on any aspect of this specification. Please join our discussion groups at:

> https://gitter.im/eclipse/jsonb

1.2. Goals

The goals of the API are as follows:

- **JSON**
  Support binding (serialization and deserialization) 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 Jakarta EE and Java 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 Jakarta JSON Processing (JSON-P) 1.1.

1.3. Non-Goals

The following are non-goals:

- **Preserving equivalence (Round-trip)**
  The specification recommends, but does not require equivalence of content for deserialized and serialized 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 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.

Java code and sample data fragments are formatted as shown in Figure 1:

Example Java Code

```
package com.example.hello;

public class Hello {
    public static void main(String args[]) {
        System.out.println("Hello World");
    }
}
```

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.*

1.5. Terminology

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

Deserialization
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.

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

1.6. Acknowledgements

This specification was originally developed as part of JSR 367 under the Java Community Process. It was the result of the collaborative work of the members of the JSR 367 Expert Group. The following were the expert group members:

- Dmitry Kornilov (Oracle)
- Roman Grigoriadi (Oracle)
- Lukas Jungmann (Oracle)
- Martin Vojtek (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)
- Nathan Rauh (IBM)
- Alexander Salvanos (Individual Member)
- Romain Manni-Bucau (Tomitribe)

During the course of JSR 367 we received many excellent suggestions. Special thanks to Heather VanCura, David Delabassee and Reza Rahman for feedback and help with evangelizing the specification, and John Clingan for feedback and language corrections.

During the course of JSR 367 we also received many excellent suggestions. Thanks in particular to

Mark Struberg, Olena Syrota, Oleg Tsal-Tsalko and whole JUG UA for their contributions.

Chapter 2. Runtime API

The JSON-B runtime API provides access to serialization and deserialization 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 `jakarta.json.bind` package accompanied with this specification.

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 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]. Serialized JSON output MUST conform to the RFC 7159 JSON Grammar and be encoded in UTF-8 encoding as defined in Section 8.1 (Character Encoding) of RFC 7159. Implementations MUST support deserialization of documents conforming to RFC 7159 JSON Grammar. In addition, implementations SHOULD NOT allow deserialization of RFC 7159 non-conforming text (e.g. unsupported encoding, ...) and report error in such cases. Detection of UTF encoding of a deserialized document MUST follow the encoding process defined in the Section 3 (Encoding) of RFC 4627. Implementations SHOULD ignore the presence of an UTF byte order mark (BOM) and not treat it as an error.

3.2. Errors

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

3.3. Basic Java Types

Implementations MUST support binding of the following basic Java classes and their corresponding primitive types:

- 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 serialized to JSON String values as defined within RFC 7159 Section 7 (Strings) in UTF-8 encoding without a byte order mark. [JSB-3.3.1-1] Implementations SHOULD support deserialization of JSON text in other (than UTF-8) UTF encodings into `java.lang.String` instances.

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

Serialization 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]. Deserialization 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`.

3.3.3. java.lang.Boolean

Serialization 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. Deserialization 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.

3.3.4. java.lang.Number

Serialization 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 section-3.3.2,section 3.3.2. Deserialization 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.

3.4. Specific Standard Java SE Types

Implementations MUST support binding of the following standard Java SE classes:

- 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

Serialization 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. Deserialization 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.

3.4.2. java.net.URL, URI

Serialization 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. Deserialization 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.

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

Non-empty instances of type java.util.Optional, OptionalInt, OptionalLong, OptionalDouble are serialized 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. Class fields containing empty optional instances are treated as having a null value and serialized based on section 3.14.1.

Empty optional instances in array items are serialized as null.

Deserializing into `Optional`, `OptionalInt`, `OptionalLong`, `OptionalDouble` return empty optional value for properties containing a null value. Otherwise any non-empty `Optional`, `OptionalInt`, `OptionalLong`, `OptionalDouble` value is constructed of type which deserialized based on mappings defined in this chapter.

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

Deserialization of a JSON value into `java.util.Optional<T>` MUST be supported if deserialization of a JSON value into instance of T is supported.

3.5. Dates

Implementations MUST support binding of the following standard Java date/time classes:

- 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.LocalTime
- 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. If not specified otherwise, the date time format for serialization and deserialization is ISO 8601 without offset, as specified in `java.time.format.DateTimeFormatter.ISO_DATE`.

Implementations MUST report an error if the date/time string in a JSON document does not correspond to the expected date/time format.

If in strict I-JSON compliance mode, default date format is changed as it's described in 4.4.1.

3.5.1. java.util.Date, Calendar, GregorianCalendar

The serialization format of `java.util.Date`, `Calendar`, `GregorianCalendar` instances with no time information is `ISO_DATE`.

If time information is present, the format is `ISO_DATE_TIME`.

Implementations MUST support deserialization of both `ISO_DATE` and `ISO_DATE_TIME` into `java.util.Date`, `Calendar` and `GregorianCalendar` instances.

3.5.2. java.util.TimeZone, SimpleTimeZone

Implementations MUST support deserialization of any time zone format specified in `java.util.TimeZone` into a field or property of type `java.util.TimeZone` and `SimpleTimeZone`.

Implementations MUST report an error for deprecated three-letter time zone IDs as specified in

`java.util.Timezone`.

The serialization format of `java.util.TimeZone` and `SimpleTimeZone` is `NormalizedCustomID` as specified in `java.util.TimeZone`.

3.5.3. java.time.*

The serialization output for a java.time.Instant instance MUST be in a `ISO_INSTANT` format, as specified in `java.time.format.DateTimeFormatter`. Implementations MUST support the deserialization of an `ISO_INSTANT` formatted JSON string to a `java.time.Instant` instance.

For other `java.time.*` classes, the following mapping table maps Java types to their corresponding formats:

Table 1. Date/time formats for java.time. types*

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

Implementations MUST support the deserialization of any time zone ID format specified in `java.time.ZoneId` into a field or property of type `java.time.ZoneId`. The serialization format of `java.time.ZoneId` is the normalized zone ID as specified in `java.time.ZoneId`.

Implementations MUST support the deserialization of any time zone ID format specified in `java.time.ZoneOffset` into a field or property of type `java.time.ZoneOffset`. The serialization format of `java.time.ZoneOffset` is the normalized zone ID as specified in `java.time.ZoneOffset`.

Implementations MUST support the deserialization of any duration format specified in `java.time.Duration` into a field or property of type `java.time.Duration`. This is super-set of ISO 8601 duration format. The serialization format of `java.time.Duration` is the ISO 8601 seconds based representation, such as PT8H6M12.345S.

Implementations MUST support the deserialization of any period format specified in `java.time.Period` into a field or property of type `java.time.Period`. This is a super-set of ISO 8601 period format. The serialization format of `java.time.Period` is ISO 8601 period representation. A zero-length period is represented as zero days 'P0D'.

3.6. Untyped mapping

For an unspecified output type of a deserialization operation, as well as where output type is specified as `Object.class`, implementations MUST deserialize a JSON document using Java runtime types specified in table below:

Table 2. Untyped Mapping

JSON Value	Java Type
object	java.util.Map<String, Object>
array	java.util.List<Object>
string	java.lang.String
number	java.math.BigDecimal
true, false	java.lang.Boolean
null	null

JSON object values are deserialized into an implementation of `java.util.Map<String, Object>` with a predictable iteration order.

3.7. Java Class

Any instance passed to a deserialization operation must have a public or protected no-argument constructor. Implementations SHOULD throw an error if this condition is not met. This limitation does not apply to serialization operations, as well as to classes which specify explicit instantiation methods as described in section 4.5.

3.7.1. Scope and Field access strategy

For a deserialization 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.

For a serialization 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.

JSON Binding implementations MUST NOT deserialize into transient, final or static fields and MUST ignore name/value pairs corresponding to such fields.

Implementations MUST support serialization of final fields. Transient and static fields MUST be ignored during serialization operation.

If a JSON document contains a name/value pair not corresponding to field or setter method then this name/value pair is skipped (see 3.18).

Public getter/setter methods without a corresponding field MUST be supported. When only public getter/setter methods without corresponding fields are present in the class, the getter method is called to obtain the value to serialize, and the setter method is called during deserialization operation.

3.7.2. Nested Classes

Implementations MUST support the binding of public and protected nested classes. For deserialization operations, both nested and encapsulating classes MUST fulfill the same instantiation requirements as specified in 3.7.1.

3.7.3. Static Nested Classes

Implementations MUST support the binding of public and protected static nested classes. For deserialization operations, the nested class MUST fulfill the same instantiation requirements as specified in 3.7.1.

3.7.4. Anonymous Classes

Deserialization into anonymous classes is not supported. Serialization of anonymous classes is supported by default object mapping.

3.8. Polymorphic Types

Deserialization into polymorphic types is not supported by default mapping.

3.9. Enum

Serialization of an Enum instance to a JSON String value MUST follow the conversion process defined in javadoc specification for their `name()`.

Deserialization of a JSON value into an enum instance MUST be done by calling the enum's `valueOf(String)` method.

3.10. Interfaces

Implementations MUST support the deserialization of specific interfaces defined in 3.11 and 3.3.4.

Deserialization to other interfaces is not supported and implementations SHOULD report error in such case.

If a class property is defined with an interface and not concrete type, then the mapping for a serialized property is resolved based on its runtime type.

3.11. Collections

Implementations MUST support the binding of the following collection interfaces, classes and their implementations:

- 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.HashMap
- java.util.NavigableMap
- java.util.SortedMap
- java.util.TreeMap
- java.util.LinkedHashMap
- java.util.List
- java.util.ArrayList
- java.util.LinkedList
- java.util.Deque
- java.util.ArrayDeque
- java.util.Queue
- java.util.PriorityQueue

Implementations of these interfaces must provide an accessible default constructor.

JSON Binding implementations MUST report a deserialization error if a default constructor is not present or is not in accessible scope.

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.

Arrays of primitive types and multi-dimensional arrays MUST be supported.

3.13. Attribute order

Class properties MUST be serialized in lexicographical order into the resulting JSON document. In case of inheritance, properties declared in super class MUST be serialized before properties declared in a child class.

When deserializing a JSON document, field values MUST be set in the order of attributes present in the JSON document.

3.14. Null value handling

3.14.1. Null Java field

The result of serializing a java field with a null value is the absence of the property in the resulting JSON document.

The deserialization 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.

The deserialization 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). The exception is `java.util.Optional`, `OptionalInt`, `OptionalLong`, `OptionalDouble` instances. In this case the value of the field is set to an empty optional value.

3.14.2. Null Array Values

The result of deserialization n-ary array represented in JSON document is n-ary Java array.

Null value in JSON array is represented by null value in Java array.

Serialization operation on Java array with null value at index `i` MUST output null value at index `i` of the array in resulting JSON document.

3.15. Names and identifiers

According to RFC 7159 Section 7, 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 Strings in JSON document.

For deserialization operations defined in 3.6 section, identity function is used to transform JSON name strings into Java `String` instances in the resulting map `Map<String, Object>`.

Naming strategy can be further customized in customization.

3.16. Big numbers

JSON Binding implementation MUST serialize/deserialize numbers that express greater magnitude or precision than an IEEE 754 double precision number as strings.

3.17. Generics

JSON Binding implementations MUST support binding of generic types.

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).

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.

3.17.1. Type resolution algorithm

There are several levels of information JSON Binding implementations may obtain about the type of field/class/interface:

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`.

If only raw type of given field/class/interface is known, then the type MUST be treated like raw type. 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.

Let's consider situation when there is only a static type information of a given field/class/interface known, and there is no runtime type information available.

Let `GenericClass<`$T_1 \cdots T_n$`>` be part of generic type declaration, where `GenericClass` is name of the generic

type and $T_1 \cdots T_n$ are type parameters. For every , where i in $1 \cdots n$, there are 3 possible options:

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

In case 1, the most specific parameter type MUST be given concrete parameter type.

For bounded parameter type, let's use bounds $B_1, \cdots, B_m$.

If $m = 1$, then the most specific parameter type MUST be derived from the given bound.

If is class or interface, the most specific parameter type MUST be the class or interface.

Otherwise, the most specific parameter type SHOULD be `java.lang.Object`.

If multiple bounds are specified, the first step is to resolve every bound separately. Let's define result of such resolution as $S_1, \cdots, S_m$ specific parameter types.

If $S_1, \cdots, S_m$ are `java.lang.Object`, then the bounded parameter type MUST be `java.lang.Object`.

If there is exactly one , where $1 \Leftarrow k \Leftarrow m$ is different than `java.lang.Object`, then the most specific parameter type for this bounded parameter type MUST be .

If there exists S_{k1}, S_{k2}, where $1 \Leftarrow k1 \Leftarrow k2 \Leftarrow m$, then the most specific parameter type is .

For wildcard parameter type without bounds, the most specific parameter type MUST be `java.lang.Object`.

Any unresolved type parameter MUST be treated as `java.lang.Object`.

If runtime type is provided via `java.lang.reflect.Type` parameter passed to `Jsonb::toJson` or `Jsonb::fromJson` method, then that runtime type overrides static type declaration wherever applicable.

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

Example Type resolution

```
public class MyGenericType<T,U> {
    public T field1;
    public U field2;
}
```

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 during deserialization 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.

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.

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` and `JsonbSerializer/JsonbDeserializer` mechanisms.

3.20. JSON Processing integration

JSON Binding implementations MUST support binding of the following JSON Processing types:

- jakarta.json.JsonObject
- jakarta.json.JsonArray
- jakarta.json.JsonStructure
- jakarta.json.JsonValue
- jakarta.json.JsonString
- jakarta.json.JsonNumber

Serialization of supported `jakarta.json.*` objects/interfaces/fields MUST have the same result as serialization these objects with `jakarta.json.JsonWriter`.

Deserialization into supported `jakarta.json.*` objects/interfaces/fields MUST have the same result as deserialization into such objects with `jakarta.json.JsonReader`.

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, JavaBean property, type or package, or by providing an implementation of particular strategy, e.g. `PropertyOrderStrategy`. JSON Binding provider MUST support these customization options.

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 `jakarta.json.bind.annotation.JsonbProperty` annotation. The second option is to set `jakarta.json.bind.config.PropertyNamingStrategy`.

4.1.1. jakarta.json.bind.annotation.JsonbTransient

JSON Binding implementations MUST NOT process fields, JavaBean properties or types annotated with `jakarta.json.bind.annotation.JsonbTransient`.

`JsonbTransient` annotation is mutually exclusive with all other JSON Binding defined annotations. Implementations must throw `JsonbException` in the following cases:

- Class field is annotated with `@JsonbTransient`
  - Exception must be thrown when this field, getter or setter is annotated with other JSON Binding annotations.
- Getter is annotated with `@JsonbTransient`
  - Exception is thrown if when the field or this getter are annotated with other JSON Binding annotations. Exception is not thrown if JSON Binding annotations are presented on the setter.
- Setter is annotated with `@JsonbTransient`
  - Exception is thrown if when the field or this setter are annotated with other JSON Binding annotations. Exception is not thrown if JSON Binding annotations are presented on the getter.

4.1.2. jakarta.json.bind.annotation.JsonbProperty

According to default mapping 3.15, property names are serialized unchanged to JSON document (identity transformation). To provide custom name for given field (or JavaBean property), `jakarta.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 `jakarta.json.bind.annotation.JsonbProperty` is specified on getter method, it is used only for serialization. If `jakarta.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 `jakarta.json.bind.annotation.JsonbProperty` annotation. In such case the different custom name will be used for serialization and deserialization.

4.1.3. jakarta.json.bind.config.PropertyNamingStrategy

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

Interface `jakarta.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 `jakarta.json.bind.JsonbConfig::withPropertyNamingStrategy` method.

4.1.4. Property names resolution

Property name resolution consists 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.

4.2. Customizing Property Order

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

Class `jakarta.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 `jakarta.json.bind.JsonbConfig::withPropertyOrderStrategy` method.

To customize the order of serialized properties only for one specific type, JSON Binding provides `jakarta.json.bind.annotation.JsonbPropertyOrder` annotation. Order specified by `JsonbPropertyOrder` annotation overrides order specified by `PropertyOrderStrategy`.

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 `jakarta.json.bind.annotation.JsonbNillable` annotation. The second option is to annotate field or JavaBean property with `jakarta.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. jakarta.json.bind.annotation.JsonbNillable

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

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

The same behavior as `JsonbNillable`, but only at field, parameter and method (JavaBean property) level is provided by `jakarta.json.bind.annotation.JsonbProperty` annotation with its `nillable` parameter.

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.

4.3.2. Global null handling configuration

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

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

The way to skip serialization of null values is to call method `jakarta.json.bind.JsonbConfig::withNullValues` with parameter `false`.

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 The I-JSON Message Format.

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.strict-ijson".

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

Strict I-JSON compliance changes only default mapping behavior (see Section 3).

4.4.1. Strict date serialization

Uppercase rather than lowercase letters MUST be used.

The time zone MUST always be included and optional trailing seconds MUST be included even when their value is "00".

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`.

The result of serialization of duration must conform to the "duration" production in Appendix A of RFC 3339, with the same additional restrictions.

4.5. Custom instantiation

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

At most one `JsonbCreator` annotation can be used to annotate custom constructor or static factory method in a class, otherwise `JsonbException` MUST be thrown.

Factory method annotated with `JsonbCreator` annotation should return instance of a particular class this annotation is used for, otherwise `JsonbException` MUST be thrown.

Mapping between parameters of constructor/factory method annotated with `JsonbCreator` and JSON fields is defined using `JsonbProperty` annotation on all parameters.

In case `JsonbProperty` annotation on parameters is not used, parameters should be mapped from JSON fields with the same name. In this case the proper mapping is NOT guaranteed.

In case a field required for a parameter mapping doesn't exist in JSON document, `JsonbException` MUST be thrown.

4.6. Custom visibility

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

4.7. Custom mapping

Some Java types do not map naturally to a JSON representation and annotations cannot be used to customize mapping. An example can be some third party classes or classes without no-arg constructor. To customize mapping in this case JSON Binding has two mechanisms: Adapters and Serializers.

4.7.1. Adapters

Adapter is a class implementing `jakarta.json.bind.adapter.JsonbAdapter` interface. It has a custom code to convert the "unmappable" type (Original) into another one that JSONB can handle (Adapted).

On serialization of Original type JSONB calls `JsonbAdapter::adaptToJson` method of the adapter to convert Original to Adapted and serializes Adapted the standard way.

On deserialization JSONB deserializes Adapted from JSON and converts it to Original using `JsonbAdapter::adaptFromJson` method.

There are two ways how to register `JsonbAdapter`:

1. Using `JsonbConfig::withAdapters` method;
2. Annotating a class field with `JsonbTypeAdapter` annotation.

`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 annotated field.

Implementations must provide a CDI support in adapters to allow injection of CDI managed beans into it.

4.7.2. Serializers/Deserializers

Sometimes adapters mechanism is not enough and low level access to JSONP parser/generator is needed.

Serializer is a class implementing `jakarta.json.bind.serializers.JsonbSerializer` interface. It is used to serialize the type it's registered on (Original). On serializing of Original type JSONB calls `JsonbSerializer::serialize` method. This method has to contain a custom code to serialize Original type using provided `JsonpGenerator`.

Deserializer is a class implementing `jakarta.json.bind.serializers.JsonbDeserializer` interface. It is used to deserialize the type it's registered on (Original). On deserialization of Original type JSONB calls `JsonbDeserializer::deserialize` method. This method has to contain a custom code to deserialize Original type using provided `JsonpParser`.

There are two ways how to register `JsonbSerializer/JsonbDeserializer`:

1. Using `JsonbConfig::withSerializers/ JsonbConfig::withDeserializers` method;
2. Annotating a type with `JsonbSerializer/JsonbDeserializer` annotation.

Implementations must provide a CDI support in serializers/deserializers to allow injection of CDI managed beans into it.

4.8. Custom date format

To specify custom date format, it is necessary to annotate given annotation. `JsonbDateFormat` annotation can be applied to the following targets:

- field
- getter/setter

- type
- parameter
- package

Default date format and default locale can be customized globally using `jakarta.json.bind.JsonbConfig::withDateFormat` and `jakarta.json.bind.JsonbConfig::withLocale` methods.

If `jakarta.json.bind.annotation.JsonbDateFormat` is specified on a getter method, it is used only for serialization. If `jakarta.json.bind.annotation.JsonbDateFormat` is specified on a setter method, it is used only for deserialization.

Annotation applied to more specific target overrides the same annotation applied to target with wider scope and global configuration. For example, annotation applied to type target will override the same annotation applied to package target.

4.9. Custom number format

To specify custom number format, it is necessary to annotate given annotation target with `jakarta.json.bind.annotation.JsonbNumberFormat` annotation. `JsonbNumberFormat` annotation can be applied to the following targets:

- field
- getter/setter
- type
- parameter
- package

If `jakarta.json.bind.annotation.JsonbNumberFormat` is specified on a getter method, it is used only for serialization. If `jakarta.json.bind.annotation.JsonbNumberFormat` is specified on a setter method, it is used only for deserialization.

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.

4.10. Custom binary data handling

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

Class `jakarta.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 `jakarta.json.bind.JsonbConfig::withBinaryDataStrategy` method.

Chapter 5. Appendix

5.1. Change Log

5.1.1. Changes Since 1.0 Early Draft

- Section 3.7: Clarified that default constructor is not needed in case of `JsonbCreator`.
- Chapters 3 and 4: Synchronized vocabulary to serialization and deserialization.
- Section 3.9: Conversion method changed from `toString()` to `name()`.
- Section 3.4.3: Changed serialization rules of object properties with `Optional` type and `null` value.
- Section 3.14.1: Added an exception for `Optional` fields.
- Section 3.6: Removed 'smallest possible type' rule for number types. JSON number type is always mapped to `BigDecimal` in case target type is not specified.
- Removed 'Simple Value' customization (`@JsonbValue`). Adapters shouldbe used instead.
- Adapters section (4.7.1) changed.
- Serializers/Deserializers section (4.7.2) added.

5.1.2. Changes Since 1.0 Public Draft

- Section 3.17.1: Sample fixed.
- Section 4.4: Method name is changed from `JsonbConfig::withStrictIJSONSerializationCompliance` to `JsonbConfig::withStrictIJSON`. Config property name is changed from `jsonb.i-json.strict-ser-compliance` to `jsonb.strict-ijson`.
- Sections 4.7.1 and 4.7.2: Added CDI support.
- Section 4.8: Added a paragraph explicitly explaining the use case when `JsonbDateFormat` annotation is placed on getter or setter.
- Section 4.9: Added a paragraph explicitly explaining the use case when `JsonbNumberFormat` annotation is placed on getter or setter.
- Section 4.5: `JsonbProperty` on parameters is required for proper mapping. If not present mapping is is done by matching names, but is not guaranteed. Clarified condition when exception is raised.
- Section 4.1.1: Clarified conditions when exceptions are throwed.
- Section 4.4: Clarified that strict I_JSON compliance affects only default mapping mechanism.
- Section 3.13: Declared fields changed to class properties.