

**JSON Schema Language**  
**draft-ucarion-json-schema-language-00**

Abstract

JSON Schema Language (JSL) is a portable method for describing the format of JavaScript Object Notation (JSON) data and the errors associated with ill-formed data. JSL is designed to enable code generation from schemas.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of [BCP 78](#) and [BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at <https://datatracker.ietf.org/drafts/current/>.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on February 6, 2020.

Copyright Notice

Copyright (c) 2019 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to [BCP 78](#) and the IETF Trust's Legal Provisions Relating to IETF Documents (<https://trustee.ietf.org/license-info>) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.

## Table of Contents

<a href="#">1.</a>	<a href="#">Introduction</a>	<a href="#">2</a>
<a href="#">1.1.</a>	<a href="#">Terminology</a>	<a href="#">3</a>
<a href="#">2.</a>	<a href="#">Syntax</a>	<a href="#">3</a>
<a href="#">3.</a>	<a href="#">Semantics</a>	<a href="#">9</a>
<a href="#">3.1.</a>	<a href="#">Strict instance semantics</a>	<a href="#">9</a>
<a href="#">3.2.</a>	<a href="#">Errors</a>	<a href="#">10</a>
<a href="#">3.3.</a>	<a href="#">Forms</a>	<a href="#">10</a>
<a href="#">3.3.1.</a>	<a href="#">Empty</a>	<a href="#">11</a>
<a href="#">3.3.2.</a>	<a href="#">Ref</a>	<a href="#">11</a>
<a href="#">3.3.3.</a>	<a href="#">Type</a>	<a href="#">12</a>
<a href="#">3.3.4.</a>	<a href="#">Enum</a>	<a href="#">15</a>
<a href="#">3.3.5.</a>	<a href="#">Elements</a>	<a href="#">15</a>
<a href="#">3.3.6.</a>	<a href="#">Properties</a>	<a href="#">16</a>
<a href="#">3.3.7.</a>	<a href="#">Values</a>	<a href="#">18</a>
<a href="#">3.3.8.</a>	<a href="#">Discriminator</a>	<a href="#">19</a>
<a href="#">4.</a>	<a href="#">IANA Considerations</a>	<a href="#">22</a>
<a href="#">5.</a>	<a href="#">Security Considerations</a>	<a href="#">23</a>
<a href="#">6.</a>	<a href="#">Normative References</a>	<a href="#">23</a>
<a href="#">Appendix A.</a>	<a href="#">Comparison with CDDL</a>	<a href="#">23</a>
	<a href="#">Acknowledgments</a>	<a href="#">26</a>
	<a href="#">Author's Address</a>	<a href="#">26</a>

## [1.](#) Introduction

This document describes a schema language for JSON [[RFC8259](#)] called JSON Schema Language (JSL).

The goals of JSL are to:

- o Provide an unambiguous description of the overall structure of a JSON document.
- o Be able to describe common JSON datatypes and structures.
- o Provide a single format that is readable and editable by both humans and machines, and which can be embedded within other JSON documents.
- o Enable code generation from schemas.
- o Provide a standardized format for errors when data does not conform with a schema.

JSL is intentionally designed as a rather minimal schema language. For example, JSL is homoiconic (it both describes, and is written in, JSON) yet is incapable of describing in detail its own structure. By

Carion

Expires February 6, 2020

[Page 2]

keeping the expressiveness of the schema language minimal, JSL makes code generation and standardized errors easier to implement.

It is expected that for many use-cases, a schema language of JSL's expressiveness is sufficient. Where a more expressive language is required, alternatives exist in CDDL ([[RFC8610](#)], Concise Data Definition Language) and others.

This document has the following structure:

The syntax of JSL is defined in [Section 2](#). [Section 3](#) describes the semantics of JSL; this includes determining whether some data satisfies a schema and what errors should be produced when the data is unsatisfactory. [Appendix A](#) presents various JSL schemas and their CDDL equivalents.

### **[1.1](#). Terminology**

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [BCP 14](#) [[RFC2119](#)] [[RFC8174](#)] when, and only when, they appear in all capitals, as shown here. These words may also appear in this document in lower case as plain English words, absent their normative meanings.

The term "JSON Pointer", when it appears in this document, is to be understood as it is defined in [[RFC6901](#)].

The terms "object", "member", "array", "number", "name", and "string" in this document are to be interpreted as described in [[RFC8259](#)].

The term "instance", when it appears in this document, refers to a JSON value being validated against a JSL schema.

## **[2](#). Syntax**

This section describes when a JSON document is a correct JSL schema.

JSL schemas may recursively contain other schemas. In this document, a "root schema" is one which is not contained within another schema, i.e. it is "top level".

A correct JSL schema MUST match the "schema" CDDL rule described in this section. A JSL schema is a JSON object taking on an appropriate form. It may optionally contain definitions (a mapping from names to schemas) and additional data.

Carion

Expires February 6, 2020

[Page 3]

```
schema = {  
  form,  
  ? definitions: { * tstr => schema }  
}
```

Figure 1: CDDL Definition of a Schema

This is not a correct JSL schema, as its "definitions" object contains a number, which is not a schema:

```
{ "definitions": { "foo": 3 }}
```

JSL schemas can take on one of eight forms. These forms are defined so as to be mutually exclusive; a schema cannot satisfy multiple forms at once.

```
form = empty /  
  ref /  
  type /  
  enum /  
  elements /  
  properties /  
  values /  
  discriminator
```

Figure 2: CDDL Definition of the Schema Forms

The first form, "empty", is trivial. It is meant for matching any instance:

```
empty = {}
```

Figure 3: CDDL Definition of the Empty Form

Thus, this is a correct schema:

```
{}
```

The second form, "ref", is for when a schema is meant to be defined in terms of something in "definitions":

```
ref = { ref: tstr }
```

Figure 4: CDDL Definition of the Ref Form

For a schema to be correct, the "ref" value must refer to one of the definitions found at the root level of the schema it appears in. More formally, for a schema `_S_` of the "ref" form:

Carion

Expires February 6, 2020

[Page 4]

- o Let `_B_` be the root schema containing the schema, or the schema itself if it is a root schema.
- o Let `_R_` be the value of the member of `_S_` with the name "ref".

If the schema is correct, then `_B_` must have a member `_D_` with the name "definitions", and `_D_` must contain a member whose name equals `_R_`.

Here is a correct example of "ref" being used to avoid re-defining the same thing twice:

```
{
  "definitions": {
    "coordinates": {
      "properties": {
        "lat": { "type": "number" },
        "lng": { "type": "number" }
      }
    }
  },
  "properties": {
    "user_location": { "ref": "coordinates" },
    "server_location": { "ref": "coordinates" }
  }
}
```

However, this schema is incorrect, as it refers to a definition that doesn't exist:

```
{
  "definitions": { "foo": { "type": "number" } },
  "ref": "bar"
}
```

This schema is incorrect as well, as it refers to a definition that doesn't exist at the root level. The non-root definition is immaterial:

```
{
  "definitions": { "foo": { "type": "number" } },
  "elements": {
    "definitions": { "bar": { "type": "number" } },
    "ref": "bar"
  }
}
```





The third form, "type", constrains instances to have a particular primitive type. The precise meaning of each of the primitive types is described in [Section 3](#).

```
type = { type: "boolean" / num-type / "string" / "timestamp" }
num-type = "number" / "float32" / "float64" /
  "int8" / "uint8" / "int16" / "uint16" / "int32" / "uint32"
```

Figure 5: CDDL Definition of the Type Form

For example, this schema constrains instances to be strings that are correct [\[RFC3339\]](#) timestamps:

```
{ "type": "timestamp" }
```

The fourth form, "enum", describes instances whose value must be one of a finite, predetermined set of values:

```
enum = { enum: [+ tstr] }
```

Figure 6: CDDL Definition of the Enum Form

The values within "[+ tstr]" MUST NOT contain duplicates. Thus, the following is a correct schema:

```
{ "enum": ["IN_PROGRESS", "DONE", "CANCELED"] }
```

But this is not a correct schema, as "B" is duplicated:

```
{ "enum": ["A", "B", "B"] }
```

The fifth form, "elements", describes instances that must be arrays. A further sub-schema describes the elements of the array.

```
elements = { elements: schema }
```

Figure 7: CDDL Definition of the Elements Form

Here is a schema describing an array of [\[RFC3339\]](#) timestamps:

```
{ "elements": { "type": "timestamp" }}
```

The sixth form, "properties", describes JSON objects being used as a "struct". A schema of this form specifies the names of required and optional properties, as well as the schemas each of those properties must satisfy:



```

; One of properties or optionalProperties may be omitted,
; but not both.
properties = with-properties / with-optional-properties

with-properties = {
  properties: * tstr => schema,
  ? optionalProperties * tstr => schema
}

with-optional-properties = {
  ? properties: * tstr => schema,
  optionalProperties: * tstr => schema
}

```

Figure 8: CDDL Definition of the Properties Form

If a schema has both a member named "properties" (with value `_P_`) and another member named "optionalProperties" (with value `_O_`), then `_O_` and `_P_` MUST NOT have any member names in common. This is to prevent ambiguity as to whether a property is optional or required.

Thus, this is not a correct schema, as "confusing" appears in both "properties" and "optionalProperties":

```

{
  "properties": { "confusing": {} },
  "optionalProperties": { "confusing": {} }
}

```

Here is a correct schema, describing a paginated list of users:

```

{
  "properties": {
    "users": {
      "elements": {
        "properties": {
          "id": { "type": "string" },
          "name": { "type": "string" },
          "create_time": { "type": "timestamp" }
        },
        "optionalProperties": {
          "delete_time": { "type": "timestamp" }
        }
      }
    },
    "next_page_token": { "type": "string" }
  }
}

```



The seventh form, "values", describes JSON objects being used as an associative array. A schema of this form specifies the form all member values must satisfy, but places no constraints on the member names:

```
values = { values: * tstr => schema }
```

Figure 9: CDDL Definition of the Values Form

Thus, this is a correct schema, describing a mapping from strings to numbers:

```
{ "values": { "type": "number" }}
```

Finally, the eighth form, "discriminator", describes JSON objects being used as a discriminated union. A schema of this form specifies the "tag" (or "discriminator") of the union, as well as a mapping from tag values to the appropriate schema to use.

```
; Note well: the values of mapping are of the properties form.  
discriminator = { tag: tstr, mapping: * tstr => properties }
```

Figure 10: CDDL Definition of the Discriminator Form

To prevent ambiguous or unsatisfiable constraints on the "tag" of a discriminator, an additional constraint on schemas of the discriminator form exists. For schemas of the discriminator form:

- o Let `_D_` be the schema member with the name "discriminator".
- o Let `_T_` be the member of `_D_` with the name "tag".
- o Let `_M_` be the member of `_D_` with the name "mapping".

If the schema is correct, then all member values `_S_` of `_M_` will be schemas of the "properties" form. For each member `_P_` of `_S_` whose name equals "properties" or "optionalProperties", `_P_'s` value, which must be an object, MUST NOT contain any members whose name equals `_T_'s` value.

Thus, this is an incorrect schema, as "event\_type" is both the value of "tag" and a member name in one of the "mapping" member "properties":



```
{
  "tag": "event_type",
  "mapping": {
    "is_event_type_a_string_or_a_number?": {
      "properties": { "event_type": { "type": "number" } }
    }
  }
}
```

However, this is a correct schema, describing a pattern of data common in JSON-based messaging systems:

```
{
  "tag": "event_type",
  "mapping": {
    "account_deleted": {
      "properties": {
        "account_id": { "type": "string" }
      }
    },
    "account_payment_plan_changed": {
      "properties": {
        "account_id": { "type": "string" },
        "payment_plan": { "enum": ["FREE", "PAID"] }
      },
      "optionalProperties": {
        "upgraded_by": { "type": "string" }
      }
    }
  }
}
```

### 3. Semantics

This section describes when an instance is valid against a correct JSL schema, and the standardized errors to produce when an instance is invalid.

#### 3.1. Strict instance semantics

Users will have different desired behavior with respect to "unspecified" members in an instance. For example:

```
{ "properties": { "a": { "type": "string" } } }
```

Some users may expect that { "a": "foo", "b": "bar" } satisfies the above schema. Others may disagree. JSL addresses this point of





contention by leaving it to implementations whether to accept such "unspecified" members, or whether to reject them.

Rejecting "unspecified" members is called "strict instance semantics". Whether to use strict instance semantics is not specified within a schema - it is considered out-of-band information.

Implementations MAY allow users to choose whether to use strict instance semantics. Implementations SHOULD document whether they use strict instance semantics by default.

See [Section 3.3.6](#) for how strict instance semantics affects schema evaluation.

### **3.2. Errors**

To facilitate consistent validation error handling, this document specifies a standard error format. Implementations SHOULD support producing errors in this standard form.

The standard error format is a JSON array. The order of the elements of this array is not specified. The elements of this array are JSON objects with two members:

- o A member with the name "instancePath", whose value is a JSON string encoding a JSON Pointer. This JSON Pointer will point to the part of the instance that was rejected.
- o A member with the name "schemaPath", whose value is a JSON string encoding a JSON Pointer. This JSON Pointer will point to the part of the schema that rejected the instance.

The values for "instancePath" and "schemaPath" depend on the form of the schema, and are described in detail in [Section 3.3](#).

### **3.3. Forms**

This section describes, for each of the eight JSL schema forms, the rules dictating whether an instance is accepted, as well as the standardized errors to produce when an instance is invalid.

The forms a correct schema may take on are formally described in [Section 2](#).



### **3.3.1. Empty**

The empty form is meant to describe instances whose values are unknown, unpredictable, or otherwise unconstrained by the schema.

If a schema is of the empty form, then it accepts all instances. A schema of the empty form will never produce any errors.

### **3.3.2. Ref**

The ref form is for when a schema is meant to be defined in terms of something in the "definitions" of the root schema. The ref form enables schemas to be less repetitive, and also enables describing recursive structures.

If a schema is of the ref form, then:

- o Let `_B_` be the root schema containing the schema, or the schema itself if it is a root schema.
- o Let `_D_` be the member of `_B_` with the name "definitions". By [Section 2](#), `_D_` exists.
- o Let `_R_` be the value of the schema member with the name "ref".
- o Let `_S_` be the value of the member of `_D_` whose name equals `_R_`. By [Section 2](#), `_S_` exists, and is a schema.

The schema accepts the instance if and only if `_S_` accepts the instance. Otherwise, the standard errors to return in this case are the union of the errors from evaluating `_S_` against the instance.

For example, the schema:

```
{
  "definitions": { "a": { "type": "number" } },
  "ref": "a"
}
```

Accepts 123 but not false. The standard errors to produce when evaluating false against this schema are:

```
[{ "instancePath": "", "schemaPath": "/definitions/a/type" }]
```

Note that the ref form is defined to only look up definitions at the root level. Thus, with the schema:



```
{
  "definitions": { "a": { "type": "number" } },
  "elements": {
    "definitions": { "a": { "type": "boolean" } },
    "ref": "foo"
  }
}
```

The instance 123 is accepted, and false is rejected. The standard errors to produce when evaluating false against this schema are:

```
[{ "instancePath": "", "schemaPath": "/definitions/a/type" }]
```

Though non-root definitions are not syntactically disallowed in correct schemas, they are entirely immaterial to evaluating references.

### **3.3.3. Type**

The type form is meant to describe instances whose value is a boolean, number, string, or timestamp ([RFC3339](#)).

If a schema is of the type form, then let `_T_` be the value of the member with the name "type". The following table describes whether the instance is accepted, as a function of `_T_'s` value:



+-----+-----+	
If <code>_T_</code> equals ...   then the instance is accepted if it is ...	
+-----+-----+	
<code>boolean</code>	equal to "true" or "false"
<code>number</code>	a JSON number
<code>float32</code>	a JSON number
<code>float64</code>	a JSON number
<code>int8</code>	See Table 2
<code>uint8</code>	See Table 2
<code>int16</code>	See Table 2
<code>uint16</code>	See Table 2
<code>int32</code>	See Table 2
<code>uint32</code>	See Table 2
<code>string</code>	a JSON string
<code>timestamp</code>	a JSON string encoding a <a href="#">[RFC3339]</a> timestamp
+-----+-----+	

Table 1: Accepted Values for Type

"float32" and "float64" are distinguished from "number" in their intent. "float32" indicates data intended to be processed as an IEEE 754 single-precision float, whereas "float64" indicates data intended to be processed as an IEEE 754 double-precision float. "number" indicates no specific intent. Tools which generate code from JSL schemas will likely produce different code for "float32", "float64", and "number".

If `_T_` starts with "int" or "uint", then the instance is accepted if and only if it is a JSON number encoding a value with zero fractional part. Depending on the value of `_T_`, this encoded number must additionally fall within a particular range:





<code>_T_</code>	Minimum Value (Inclusive)	Maximum Value (Inclusive)
<code>int8</code>	-128	127
<code>uint8</code>	0	255
<code>int16</code>	-32,768	32,767
<code>uint16</code>	0	65,535
<code>int32</code>	-2,147,483,648	2,147,483,647
<code>uint32</code>	0	4,294,967,295

Table 2: Ranges for Integer Types

Note that 10, 10.0, and 1.0e1 encode values with zero fractional part. 10.5 encodes a number with a non-zero fractional part. Thus `{"type": "int8"}` accepts 10, 10.0, and 1.0e1, but not 10.5.

If the instance is not accepted, then the standard error for this case shall have an `"instancePath"` pointing to the instance, and a `"schemaPath"` pointing to the schema member with the name `"type"`.

For example:

- o The schema `{"type": "boolean"}` accepts `false`, but rejects 127.
- o The schema `{"type": "number"}` accepts 10.5, 127 and 128, but rejects `false`.
- o The schema `{"type": "int8"}` accepts 127, but rejects 10.5, 128 and `false`.
- o The schema `{"type": "string"}` accepts `"1985-04-12T23:20:50.52Z"` and `"foo"`, but rejects 127.
- o The schema `{"type": "timestamp"}` accepts `"1985-04-12T23:20:50.52Z"`, but rejects `"foo"` and 127.

In all of the rejected examples just given, the standard error to produce is:

```
[{ "instancePath": "", "schemaPath": "/type" }]
```



#### 3.3.4. Enum

The enum form is meant to describe instances whose value must be one of a finite, predetermined set of string values.

If a schema is of the enum form, then let `_E_` be the value of the schema member with the name "enum". The instance is accepted if and only if it is equal to one of the elements of `_E_`.

If the instance is not accepted, then the standard error for this case shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the schema member with the name "enum".

For example, the schema:

```
{ "enum": ["PENDING", "DONE", "CANCELED"] }
```

Accepts "PENDING", "DONE", and "CANCELED", but it rejects both 123 and "UNKNOWN" with the standard errors:

```
[{ "instancePath": "", "schemaPath": "/enum" }]
```

#### 3.3.5. Elements

The elements form is meant to describe instances that must be arrays. A further sub-schema describes the elements of the array.

If a schema is of the elements form, then let `_S_` be the value of the schema member with the name "elements". The instance is accepted if and only if all of the following are true:

- o The instance is an array. Otherwise, the standard error for this case shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the schema member with the name "elements".
- o If the instance is an array, then every element of the instance must be accepted by `_S_`. Otherwise, the standard errors for this case are the union of all the errors arising from evaluating `_S_` against elements of the instance.

For example, if we have the schema:

```
{  
  "elements": {  
    "type": "number"  
  }  
}
```



Then the instances [] and [1, 2, 3] are accepted. If instead we evaluate false against that schema, the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/elements" }]
```

Finally, if we evaluate the instance:

```
[1, 2, "foo", 3, "bar"]
```

The standard errors are:

```
[
  { "instancePath": "/2", "schemaPath": "/elements/type" },
  { "instancePath": "/4", "schemaPath": "/elements/type" }
]
```

### **3.3.6. Properties**

The properties form is meant to describe JSON objects being used as a "struct".

If a schema is of the properties form, then the instance is accepted if and only if all of the following are true:

- o The instance is an object.

Otherwise, the standard error for this case shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the schema member with the name "properties" if such a schema member exists; if such a member doesn't exist, "schemaPath" shall point to the schema member with the name "optionalProperties".

- o If the instance is an object and the schema has a member named "properties", then let `_P_` be the value of the schema member named "properties". `_P_`, by [Section 2](#), must be an object. For every member name in `_P_`, a member of the same name in the instance must exist.

Otherwise, the standard error for this case shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the member of `_P_` failing the requirement just described.

- o If the instance is an object, then let `_P_` be the value of the schema member named "properties" (if it exists), and `_O_` be the value of the schema member named "optionalProperties" (if it exists).



For every member `_I_` of the instance, find a member with the same name as `_I_'s` in `_P_` or `_O_`. By [Section 2](#), it is not possible for both `_P_` and `_O_` to have such a member. If the "discriminator tag exemption" is in effect on `_I_` (see [Section 3.3.8](#)), then ignore `_I_`. Otherwise:

- \* If no such member in `_P_` or `_O_` exists and validation is using strict instance semantics, then the instance is rejected.

The standard error for this case has an "instancePath" pointing `_I_`, and a "schemaPath" pointing to the schema.

- \* If such a member in `_P_` or `_O_` does exist, then call this member `_S_`. If `_S_` rejects `_I_'s` value, then the instance is rejected.

The standard error for this case is the union of the errors from evaluating `_S_` against `_I_'s` value.

An instance may have multiple errors arising from the second and third bullet in the above. In this case, the standard errors are the union of the errors.

For example, if we have the schema:

```
{
  "properties": {
    "a": { "type": "string" },
    "b": { "type": "string" }
  },
  "optionalProperties": {
    "c": { "type": "string" },
    "d": { "type": "string" }
  }
}
```

Then each of the following instances (one on each line) are accepted:

```
{ "a": "foo", "b": "bar" }
{ "a": "foo", "b": "bar", "c": "baz" }
{ "a": "foo", "b": "bar", "c": "baz", "d": "quux" }
{ "a": "foo", "b": "bar", "d": "quux" }
```

If we evaluate the instance 123 against this schema, then the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/properties" }]
```





If instead we evaluate the instance:

```
{ "b": 3, "c": 3, "e": 3 }
```

The standard errors, using strict instance semantics, are:

```
[
  { "instancePath": "",
    "schemaPath": "/properties/a" },
  { "instancePath": "/b",
    "schemaPath": "/properties/b/type" },
  { "instancePath": "/c",
    "schemaPath": "/optionalProperties/c/type" },
  { "instancePath": "/e",
    "schemaPath": "" }
]
```

If we the same instance were evaluated, but without strict instance semantics, the final element of the above array of errors would not be present.

### [3.3.7.](#) Values

The elements form is meant to describe instances that are JSON objects being used as an associative array.

If a schema is of the values form, then let `_S_` be the value of the schema member with the name "values". The instance is accepted if and only if all of the following are true:

- o The instance is an object. Otherwise, the standard error for this case shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to the schema member with the name "values".
- o If the instance is an object, then every member value of the instance must be accepted by `_S_`. Otherwise, the standard errors for this case are the union of all the errors arising from evaluating `_S_` against member values of the instance.

For example, if we have the schema:

```
{
  "values": {
    "type": "number"
  }
}
```



Then the instances {} and {"a": 1, "b": 2} are accepted. If instead we evaluate false against that schema, the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/values" }]
```

Finally, if we evaluate the instance:

```
{ "a": 1, "b": 2, "c": "foo", "d": 3, "e": "bar" }
```

The standard errors are:

```
[
  { "instancePath": "/c", "schemaPath": "/values/type" },
  { "instancePath": "/e", "schemaPath": "/values/type" }
]
```

### **3.3.8. Discriminator**

The discriminator form is meant to describe JSON objects being used in a fashion similar to a discriminated union construct in C-like languages. When a schema is of the "discriminator" form, it validates:

- o That the instance is an object,
- o That the instance has a particular "tag" property,
- o That this "tag" property's value is a string within a set of valid values, and
- o That the instance satisfies another schema, where this other schema is chosen based on the value of the "tag" property.

The behavior of the discriminator form is more complex than the other keywords. Readers familiar with CDDL may find the final example in [Appendix A](#) helpful in understanding its behavior. What follows in this section is a description of the discriminator form's behavior, as well as some examples.

If a schema is of the "discriminator" form, then:

- o Let `_D_` be the schema member with the name "discriminator".
- o Let `_T_` be the member of `_D_` with the name "tag".
- o Let `_M_` be the member of `_D_` with the name "mapping".



- o Let `_I_` be the instance member whose name equals `_T_'s` value. `_I_` may, for some rejected instances, not exist.
- o Let `_S_` be the member of `_M_` whose name equals `_I_'s` value. `_S_` may, for some rejected instances, not exist.

The instance is accepted if and only if:

- o The instance is an object.

Otherwise, the standard error for this case shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to `_D_`.

- o If the instance is a JSON object, then `_I_` must exist.

Otherwise, the standard error for this case shall have an "instancePath" pointing to the instance, and a "schemaPath" pointing to `_T_`.

- o If the instance is a JSON object and `_I_` exists, `_I_'s` value must be a string.

Otherwise, the standard error for this case shall have an "instancePath" pointing to `_I_`, and a "schemaPath" pointing to `_T_`.

- o If the instance is a JSON object and `_I_` exists and has a string value, then `_S_` must exist.

Otherwise, the standard error for this case shall have an "instancePath" pointing to `_I_`, and a "schemaPath" pointing to `_M_`.

- o If the instance is a JSON object, `_I_` exists, and `_S_` exists, then the instance must satisfy `_S_'s` value. By [Section 2](#), `_S_'s` value must have the properties form. Apply the "discriminator tag exemption" afforded in [Section 3.3.6](#) to `_I_` when evaluating whether the instance satisfies `_S_'s` value.

Otherwise, the standard errors for this case shall be standard errors from evaluating `_S_'s` value against the instance, with the "discriminator tag exemption" applied to `_I_`.

Each of the list items above are defined to be mutually exclusive. For the same instance and schema, only one of the list items above will apply.



To illustrate the discriminator form, if we have the schema:

```
{
  "discriminator": {
    "tag": "version",
    "mapping": {
      "v1": {
        "properties": {
          "a": { "type": "number" }
        }
      },
      "v2": {
        "properties": {
          "a": { "type": "string" }
        }
      }
    }
  }
}
```

Then if we evaluate the instance:

```
"example"
```

Against this schema, the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/discriminator" }]
```

(This is the case of the instance not being an object.)

If we instead evaluate the instance:

```
{}
```

Then the standard errors are:

```
[{ "instancePath": "", "schemaPath": "/discriminator/tag" }]
```

(This is the case of `_I_` not existing.)

If we instead evaluate the instance:

```
{ "version": 1 }
```

Then the standard errors are:

```
[{ "instancePath": "/version", "schemaPath": "/discriminator/tag" }]
```





(This is the case of `_I_` existing, but having a string value.)

If we instead evaluate the instance:

```
{ "version": "v3" }
```

Then the standard errors are:

```
[  
  { "instancePath": "/version",  
    "schemaPath": "/discriminator/mapping" }  
]
```

(This is the case of `_I_` existing and having a string value, but `_S_` not existing.)

If the instance evaluated were:

```
{ "version": "v2", "a": 3 }
```

Then the standard errors are:

```
[  
  {  
    "instancePath": "/a",  
    "schemaPath": "/discriminator/mapping/v2/properties/a/type"  
  }  
]
```

(This is the case of `_I_` and `_S_` existing, but the instance not satisfying `_S_'s` value.)

Finally, if instead the instance were:

```
{ "version": "v2", "a": "foo" }
```

Then the instance satisfies the schema. No standard errors are returned. This would be the case even if evaluation were using strict instance semantics, as the "discriminator tag exemption" would ensure that "version" is not treated as an unexpected property when evaluating the instance against `_S_'s` value.

#### **4. IANA Considerations**

No IANA considerations.



## 5. Security Considerations

Implementations of JSON Schema Language will necessarily be manipulating JSON data. Therefore, the security considerations of [RFC8259] are all relevant here.

Implementations which evaluate user-inputted schemas SHOULD implement mechanisms to detect, and abort, circular references which might cause a naive implementation to go into an infinite loop. Without such mechanisms, implementations may be vulnerable to denial-of-service attacks.

## 6. Normative References

- [RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", [BCP 14](#), [RFC 2119](#), DOI 10.17487/RFC2119, March 1997, <<https://www.rfc-editor.org/info/rfc2119>>.
- [RFC3339] Klyne, G. and C. Newman, "Date and Time on the Internet: Timestamps", [RFC 3339](#), DOI 10.17487/RFC3339, July 2002, <<https://www.rfc-editor.org/info/rfc3339>>.
- [RFC6901] Bryan, P., Ed., Zyp, K., and M. Nottingham, Ed., "JavaScript Object Notation (JSON) Pointer", [RFC 6901](#), DOI 10.17487/RFC6901, April 2013, <<https://www.rfc-editor.org/info/rfc6901>>.
- [RFC8174] Leiba, B., "Ambiguity of Uppercase vs Lowercase in [RFC 2119](#) Key Words", [BCP 14](#), [RFC 8174](#), DOI 10.17487/RFC8174, May 2017, <<https://www.rfc-editor.org/info/rfc8174>>.
- [RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, [RFC 8259](#), DOI 10.17487/RFC8259, December 2017, <<https://www.rfc-editor.org/info/rfc8259>>.
- [RFC8610] Birkholz, H., Vigano, C., and C. Bormann, "Concise Data Definition Language (CDDL): A Notational Convention to Express Concise Binary Object Representation (CBOR) and JSON Data Structures", [RFC 8610](#), DOI 10.17487/RFC8610, June 2019, <<https://www.rfc-editor.org/info/rfc8610>>.

## Appendix A. Comparison with CDDL

This appendix is informative.



To aid the reader familiar with CDDL, this section illustrates how JSL works by presenting JSL schemas and CDDL schemas which accept and reject the same instances.

The JSL schema {} accepts the same instances as the CDDL rule:

root = any

The JSL schema:

```
{
  "definitions": {
    "a": { "elements": { "ref": "b" } },
    "b": { "type": "number" }
  },
  "elements": {
    "ref": "a"
  }
}
```

Corresponds to the CDDL schema:

root = [\* a]

a = [\* b]

b = number

The JSL schema:

```
{ "enum": ["PENDING", "DONE", "CANCELED"] }
```

Accepts the same instances as the CDDL rule:

root = "PENDING" / "DONE" / "CANCELED"

The JSL schema {"type": "boolean"} corresponds to the CDDL rule:

root = bool

The JSL schema {"type": "number"} corresponds to the CDDL rule:

root = number

The JSL schema {"type": "string"} corresponds to the CDDL rule:

root = tstr

The JSL schema {"type": "timestamp"} corresponds to the CDDL rule:



root = tdate

The JSL schema:

```
{ "elements": { "type": "number" }}
```

Corresponds to the CDDL rule:

root = [\* number]

The JSL schema:

```
{
  "properties": {
    "a": { "type": "boolean" },
    "b": { "type": "number" }
  },
  "optionalProperties": {
    "c": { "type": "string" },
    "d": { "type": "timestamp" }
  }
}
```

Corresponds to the CDDL rule:

root = { a: bool, b: number, ? c: tstr, ? d: tdate }

The JSL schema:

```
{ "values": { "type": "number" }}
```

Corresponds to the CDDL rule:

root = { \* tstr => number }

Finally, the JSL schema:





```
{
  "discriminator": {
    "tag": "a",
    "mapping": {
      "foo": {
        "properties": {
          "b": { "type": "number" }
        }
      },
      "bar": {
        "properties": {
          "b": { "type": "string" }
        }
      }
    }
  }
}
```

Corresponds to the CDDL rule:

```
root = { a: "foo", b: number } / { a: "bar", b: tstr }
```

#### Acknowledgments

Thanks to Gary Court, Francis Galieue, Kris Zyp, Geraint Luff, Jason Desrosiers, Daniel Perrett, Erik Wilde, Ben Hutton, Evgeny Poberezkin, Brad Bowman, Gowry Sankar, Donald Pipowitch, Dave Finlay, Denis Laxalde, Henry Andrews, and Austin Wright for their work on the initial drafts of JSON Schema, which inspired JSON Schema Language.

Thanks to Tim Bray, Carsten Bormann, and James Manger for their help on JSON Schema Language.

#### Author's Address

Ulysse Carion

Email: [ulysssecarion@gmail.com](mailto:ulysssecarion@gmail.com)

