Specification for JSON Abstract Data Notation Version 1.0

Committee Specification Draft 01

21 October 2020

This version:

https://docs.oasis-open.org/openc2/jadn/v1.0/csd01/jadn-v1.0-csd01.md (Authoritative)
https://docs.oasis-open.org/openc2/jadn/v1.0/csd01/jadn-v1.0-csd01.html
https://docs.oasis-open.org/openc2/jadn/v1.0/csd01/jadn-v1.0-csd01.pdf

Previous version:

N/A

Latest version:

https://docs.oasis-open.org/openc2/jadn/v1.0/jadn-v1.0.md (Authoritative)
https://docs.oasis-open.org/openc2/jadn/v1.0/jadn-v1.0.html
https://docs.oasis-open.org/openc2/jadn/v1.0/jadn-v1.0.pdf

Technical Committee:

OASIS Open Command and Control (OpenC2) TC

Chairs:

Joe Brule (jmbrule@radium.ncsc.mil), National Security Agency
Duncan Sparrell (duncan@sfractal.com), sFractal Consulting LLC

Editor:

David Kemp (dkemp@radium.ncsc.mil), National Security Agency

Additional artifacts:

This prose specification is one component of a Work Product that also includes:

- Schema for JADN specifications: https://docs.oasis-open.org/openc2/jadn/v1.0/csd01/schemas/

- Conformance test data: https://docs.oasis-open.org/openc2/jadn/v1.0/csd01/tests/

Abstract:

JSON Abstract Data Notation (JADN) is an information modeling language. It has several purposes including defining data structures, validating data instances, informing user interfaces working with structured data, and facilitating protocol internationalization. JADN specifications consist of two parts: abstract type definitions that are independent of data format, and serialization rules that define how to represent type instances using specific data formats. A JADN schema is itself a structured information object that can be serialized and transferred between applications, documented in multiple formats such as text-based interface definition languages, property tables or diagrams, and translated into concrete schemas used to validate specific data formats.

Status:

This document was last revised or approved by the OASIS Open Command and Control (OpenC2) TC on the above date. The level of approval is also listed above. Check the "Latest version" location noted above for possible later revisions of this document. Any other numbered Versions and other technical work produced by the Technical Committee (TC) are listed at https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=openc2#technical.

TC members should send comments on this specification to the TC's email list. Others should send comments to the TC's public comment list, after subscribing to it by following the instructions at the "Send A Comment" button on the TC's web page at https://www.oasis-open.org/committees/openc2/.

This specification is provided under the Non-Assertion Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (https://www.oasis-open.org/committees/openc2/ipr.php).

Note that any machine-readable content (Computer Language Definitions) declared Normative for this Work Product is provided in separate plain text files. In the event of a discrepancy between any such plain text file and display content in the Work Product's prose narrative document(s), the content in the separate plain text file prevails.

Citation format:

When referencing this specification the following citation format should be used:

[JADN-v1.0]

Specification for JSON Abstract Data Notation Version 1.0. Edited by David Kemp. 21 October 2020. OASIS Committee Specification Draft 01. https://docs.oasis-open.org/openc2/jadn/v1.0/csd01/jadn-v1.0-csd01.html. Latest version: https://docs.oasis-

open.org/openc2/jadn/v1.0/jadn-v1.0.html.

Notices

All capitalized terms in the following text have the meanings assigned to them in the OASIS Intellectual Property Rights Policy (the "OASIS IPR Policy"). The full Policy may be found at the OASIS website.

This document and translations of it may be copied and furnished to others, and derivative works that comment on or otherwise explain it or assist in its implementation may be prepared, copied, published, and distributed, in whole or in part, without restriction of any kind, provided that the above copyright notice and this section are included on all such copies and derivative works. However, this document itself may not be modified in any way, including by removing the copyright notice or references to OASIS, except as needed for the purpose of developing any document or deliverable produced by an OASIS Technical Committee (in which case the rules applicable to copyrights, as set forth in the OASIS IPR Policy, must be followed) or as required to translate it into languages other than English.

The limited permissions granted above are perpetual and will not be revoked by OASIS or its successors or assigns.

This document and the information contained herein is provided on an "AS IS" basis and OASIS DISCLAIMS ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY OWNERSHIP RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

OASIS requests that any OASIS Party or any other party that believes it has patent claims that would necessarily be infringed by implementations of this OASIS Committee Specification or OASIS Standard, to notify OASIS TC Administrator and provide an indication of its willingness to grant patent licenses to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification.

OASIS invites any party to contact the OASIS TC Administrator if it is aware of a claim of ownership of any patent claims that would necessarily be infringed by implementations of this specification by a patent holder that is not willing to provide a license to such patent claims in a manner consistent with the IPR Mode of the OASIS Technical Committee that produced this specification. OASIS may include such claims on its website, but disclaims any obligation to do so.

OASIS takes no position regarding the validity or scope of any intellectual property or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; neither does it represent that it has made any effort to identify any such rights. Information on OASIS' procedures with respect to rights in any document or deliverable produced by an OASIS Technical Committee can be found on the OASIS website. Copies of claims of rights made available for publication and any assurances of licenses to be made

available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this OASIS Committee Specification or OASIS Standard, can be obtained from the OASIS TC Administrator. OASIS makes no representation that any information or list of intellectual property rights will at any time be complete, or that any claims in such list are, in fact, Essential Claims.

The name "OASIS" is a trademark of OASIS, the owner and developer of this specification, and should be used only to refer to the organization and its official outputs. OASIS welcomes reference to, and implementation and use of, specifications, while reserving the right to enforce its marks against misleading uses. Please see https://www.oasis-open.org/policies-guidelines/trademark for above guidance.

Table of Contents

1 Introduction

Internet RFC 3444 describes the difference between information models and data models, noting that the purpose of an information model is to model data at a conceptual level, independent of specific implementations or protocols used to transport the data. The IETF report on Semantic Interoperability, RFC 8477 describes a lack of consistency across Standards Developing Organizations in defining application layer data, attributing it to the lack of an encoding-independent standardization of the information represented by that data.

This document defines an information modeling language intended to address that gap. JADN is a formal description technique that combines *structural abstraction* based on graph theory and *data abstraction* based on information theory to specify the syntax of structured data. As with any FDT this approach is intended to be formal, descriptive, and technically useful. Tools with no specific JADN knowledge are able to treat an information model as a generic graph, allowing reuse of existing design processes and tooling for software objects, interfaces, services and systems.

Graph theory - a JADN information model is a graph that defines pairwise relations between nodes. Each node has a name that is unique across the model. Each edge has a name that is unique within the node that defines it. Any graph with these properties can be either a view of or a structural template for a JADN information model.

Information theory - each node defines a DataType (UML Section 10.2) in terms of the characteristics it provides to applications. Information theory quantifies the novelty (news value, or "entropy") of data, and JADN DataTypes define the information conveyed by an instance separately from the data used to serialize it. Separating significant information from insignificant data allows a single information model to define data models ranging from nearly pure-information specifications such as RFC 791 to highly-verbose XML.

JADN defines three equivalence relationships between information and data:

1. Serialization of primitives such as dates and IP addresses by binary value or text representation (formats)
2. Serialization of enumerated strings by tag or value (vocabularies and field IDs)
3. Serialization of table rows by column name or position (records)

The W3C Data Workshop used the terms "Friendly" for XML and JSON encodings that associate data types directly with variables and "UnFriendly" for encodings that use repeated variable names in name-value pairs. JADN serialization rules can define multiple data formats (name-value, friendly, or machine-optimized) within one media-type, making it possible to transform data between data formats as well as media-types. This is particularly useful for defining CBOR data models that are both concise and equivalent to data models for name-value or friendly XML or JSON:

Data Format:	JSON	Compact JSON	Machine JSON

Data Format:	JSON	Compact JSON	Machine JSON
1. Primitives	Text Rep	Text Rep	Base64
2. Strings	Value	Value	Tag
3. Table Rows	Col Name	Position	Position

1.1 Requirements

The language defined in this document addresses the following requirements from RFC 8477:

> ***Formal Languages for Documentation Purposes***
>
> *To simplify reviewand publication, SDOs need formal descriptions of their data and interaction models. Several of them use a tabular representation found in the specification itself but use a formal language as an alternative way of describing objects and resources for formal purposes.*

JADN serves both purposes. It is a formal information modeling language (expressable as JSON data) that can be validated for correctness, and its definitions can be converted to/from both tabular and text representations, ensuring that the body of a specification accurately represents the formal model.

> ***Formal Languages for Code Generation***
>
> *Code-generation tools that use formal data and information modeling languages are needed by developers.*

A JADN schema, expressed as JSON data, can be read by applications and either interpreted as "byte code" to validate and serialize application data on the fly, or be used to generate static validation and serialization code.

> ***Debugging Support***
>
> *Debugging tools are needed that implement generic object browsers, which use standard data models and/or retrieve formal language descriptions from the devices themselves.*

A JADN schema is itself an information object that can be serialized to a device's data format and retrieved from the device, retrieved from a repository, or transferred along with application data. This allows tools to display schema-annotated application data independently of data format.

> ***Translation***
>
> - *The working assumption is that devices need to have a common data*

> *model with a priori knowledge of data types and actions.*
> - *Another potential approach is to have a minimal amount of information on the device to allowfor a runtime binding to a specific model,*
> - *Moreover, gateways, bridges and other similar devices need to dynamically translate (or map) one data model to another one.*

Devices and gateways can use JADN information models that are either known a-priori or bound at runtime. Once the IM is known, it is used by devices to serialize, deserialize and validate data, and by gateways to validate and translate data from one format to another. Security gateways can use the IM to filter out non-significant data and reject invalid data, whether generated maliciously or by accident.

Numerous data definition languages are in use. JADN is not intended to replace any of them; it exists as a Rosetta stone to facilitate translation among them. Starting with a common information model and deriving multiple data models from it, as shown in RFC 3444, provides more accurate translation results than attempting to translate across separately-developed data models.

1.2 IPR Policy

This specification is provided under the Non-Assertion Mode of the OASIS IPR Policy, the mode chosen when the Technical Committee was established. For information on whether any patents have been disclosed that may be essential to implementing this specification, and any offers of patent licensing terms, please refer to the Intellectual Property Rights section of the TC's web page (https://www.oasis-open.org/committees/openc2/ipr.php).

1.3 Terminology

1.3.1 Glossary

- **Schema**: An abstract schema, or information model, describes the structure and value constraints of information used by applications. A concrete schema, or data model, describes the structure and value constraints of a document used to store information or communicate it between applications.
- **Graph**: A mathematical structure used to model pairwise relations between objects. A graph is made up of nodes and edges. An information model is a graph where nodes define information types and edges define relationships between types.
- **Package**: A namespace for the set of nodes it contains. A node may reference nodes contained in other packages by namespace.
- **Document**: A series of octets described by a data format applied to an information model, or equivalently, by a data model.
- **Well-formed**: A well-formed document follows the syntactic structure of the document's media type.

- **Valid**: An instance is valid if it satisfies the constraints defined in an information model. A document is valid if it is well-formed and also corresponds to a valid instance.

- **Data Format**: A data format, defined by serialization rules, specifies the media type (XML, JSON, ...), design goals (human readability, efficiency), and style preferences for documents in that format. This specification defines a baseline set of data formats. Additional data formats may be defined for any media types that can represent instances of the JADN information model.

- **Instance**: An instance, or API value, is an item of application information to which a schema applies. An instance has one of the base types defined in Section 3 and value constraints defined in the schema by type name. The base types are:

    - **Primitive:** Null, Boolean, Binary, Integer, Number, String
    - **Enumeration:** Enumerated
    - **Structured:** Array, ArrayOf(value_type), Choice, Map, MapOf(key_type, value_type), Record.

- **Equality**: Two instances are equal if and only if they are of the same type and have the same value according to the information model. Formatting differences, including a document's data format, are insignificant. An IPv4 address serialized as a JSON dotted-quad is equal to an IPv4 address serialized as a CBOR byte string if and only if they have the same 32 bit value. A Record instance serialized as an array is equal to a Record instance serialized as a map if and only if they have the same keys and the same value for each key.

- **Serialization**: Serialization, or encoding, is the process of converting application information into a document. De-serialization, or decoding, converts a document into information instances usable by an application.

- **Description**: Description elements are reserved for comments from schema authors to readers or maintainers of the schema, and are ignored by applications using the schema.

1.3.2 Key words used to indicate requirement levels

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in [RFC2119] and [RFC8174] when, and only when, they appear in all capitals, as shown here.

1.4 Normative References

[ES9]

ECMA International, *"ECMAScript 2018 Language Specification"*, ECMA-262 9th Edition, June 2018, https://www.ecma-international.org/ecma-262.

[EUI]

"IEEE Registration Authority Guidelines for use of EUI, OUI, and CID", IEEE, August 2017, https://standards.ieee.org/content/dam/ieee-standards/standards/web/documents/tutorials/eui.pdf.

[JSONSCHEMA]

Wright, A., Andrews, H., Hutton, B., *"JSON Schema Validation"*, Internet-Draft, 16 September 2019, https://tools.ietf.org/html/draft-handrews-json-schema-validation-02, or for latest drafts: https://json-schema.org/work-in-progress.

[RFC791]

Postel, J., "Internet Protocol", RFC 791, September 1981, http://www.rfc-editor.org/info/rfc791.

[RFC2119]

Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, http://www.rfc-editor.org/info/rfc2119.

[RFC2673]

Crawford, M., *"Binary Labels in the Domain Name System"*, RFC 2673, August 1999, https://tools.ietf.org/html/rfc2673.

[RFC4291]

Hinden, R., Deering, S., "IP Version 6 Addressing Architecture", RFC 4291, February 2006, http://www.rfc-editor.org/info/rfc4291.

[RFC4632]

Fuller, V., Li, T., "Classless Inter-domain Routing (CIDR): The Internet Address Assignment and Aggregation Plan", RFC 4632, August 2006, http://www.rfc-editor.org/info/rfc4632.

[RFC4648]

Josefsson, S., "The Base16, Base32, and Base64 Data Encodings", RFC 4648, October 2006, http://www.rfc-editor.org/info/rfc4648.

[RFC5234]

Crocker, D., Overell, P., *"Augmented BNF for Syntax Specifications: ABNF"*, RFC 5234, January 2008, https://tools.ietf.org/html/rfc5234.

[RFC6901]

Bryan, P., Zyp, K., Nottingham, M., "JavaScript Object Notation (JSON) Pointer", RFC 6901, April 2013, https://tools.ietf.org/html/rfc6901

[RFC7049]

Bormann, C., Hoffman, P., *"Concise Binary Object Representation (CBOR)"*, RFC 7049, October 2013, https://tools.ietf.org/html/rfc7049.

[RFC7405]

Kyzivat, P., "Case-Sensitive String Support in ABNF", RFC 7405, December 2014, https://tools.ietf.org/html/rfc7405

[RFC8174]

Leiba, B., "Ambiguity of Uppercase vs Lowercase in RFC 2119 Key Words", BCP 14, RFC 8174, DOI 10.17487/RFC8174, May 2017, http://www.rfc-editor.org/info/rfc8174.

[RFC8200]

Deering, S., Hinden, R., "Internet Protocol, Version 6 (IPv6) Specification", RFC 8200, July 2017, http://www.rfc-editor.org/info/rfc8200.

[RFC8259]

Bray, T., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, December 2017, http://www.rfc-editor.org/info/rfc8259.

1.5 Informative References

[AVRO]

Apache Software Foundation, *"Apache Avro Documentation"*, https://avro.apache.org/docs/current/.

[BRIDGE]

Thaler, Dave, *"IoT Bridge Taxonomy"*, https://www.iab.org/wp-content/IAB-uploads/2016/03/DThaler-IOTSI.pdf

[DRY]

"Don't Repeat Yourself", https://en.wikipedia.org/wiki/Don%27t_repeat_yourself.

[FDT]

König, H., *"Protocol Engineering, Chapter 8"*, https://link.springer.com/chapter/10.1007%2F978-3-642-29145-6_8

[GRAPH]

Rennau, Hans-Juergen, *"Combining graph and tree"*, XML Prague 2018, https://archive.xmlprague.cz/2018/files/xmlprague-2018-proceedings.pdf

[PROTO]

Google Developers, *"Protocol Buffers"*, https://developers.google.com/protocol-buffers/.

[RELAXNG]

OASIS Technical Committee, *"RELAX NG"*, November 2002, https://www.oasis-open.org/committees/tc_home.php?wg_abbrev=relax-ng.

[RFC3444]

Pras, A., Schoenwaelder, J., *"On the Difference between Information Models and Data Models"*, RFC 3444, January 2003, https://tools.ietf.org/html/rfc3444.

[RFC3552]

Rescorla, E. and B. Korver, "Guidelines for Writing RFC Text on Security Considerations", BCP 72, RFC 3552, DOI 10.17487/RFC3552, July 2003, https://www.rfc-editor.org/info/rfc3552.

[RFC7493]

Bray, T., "The I-JSON Message Format", RFC 7493, March 2015, https://tools.ietf.org/html/rfc7493.

[RFC8340]

Bjorklund, M., Berger, L., *"YANG Tree Diagrams"*, RFC 8340, March 2018, https://tools.ietf.org/html/rfc8340.

[RFC8477]

Jimenez, J., Tschofenig, H., Thaler, D., *"Report from the Internet of Things (IoT) Semantic Interoperability (IOTSI) Workshop 2016"*, RFC 8477, October 2018, https://tools.ietf.org/html/rfc8477.

[RFC8610]

Birkholz, H., Vigano, C., Bormann, C., *"Concise Data Definition Language"*, RFC 8610, June 2019, https://tools.ietf.org/html/rfc8610.html.

[THRIFT]

Apache Software Foundation, *"Writing a .thrift file"*, https://thrift-tutorial.readthedocs.io/en/latest/thrift-file.html.

[TRANSFORM]

Boyer, J., et. al., *"Experiences with JSON and XML Transformations"*, October 2011, https://www.w3.org/2011/10/integration-

workshop/s/ExperienceswithJSONandXMLTransformations.v08.pdf

[UML]

"Unified Modeling Language", Version 2.5.1, December 2017, https://www.omg.org/spec/UML/2.5.1/PDF

[UNION]

"Tagged Union", Wikipedia, https://en.wikipedia.org/wiki/Tagged_union.

2 Information vs. Data

Information is *what* needs to be communicated between applications, and data is *how* that information is represented when communicating. More formally, information is the unexpected data, or "entropy", contained in a document. When information is serialized for transmission in a canonical format, the additional data used for purposes such as text conversion, delimiting, and framing contains no information because it is known a priori. If the serialization is non-canonical, any additional entropy introduced during serialization (e.g., variable whitespace, leading zeroes, field reordering, case-insensitive capitalization) is discarded on deserialization.

A variable that can take on 2^N different values conveys at most N bits of information. For example, an IPv4 address that can specify 2^32 different addresses is, by definition, a 32 bit value*. But different data may be used to represent that information:

- IPv4 dotted-quad contained in a JSON string: "192.168.141.240" (17 bytes / 136 bits).
- IPv4 dotted-quad contained in a CBOR string: 0x6F3139322E3136382E3134312E323430 (16 bytes / 128 bits)
- Hex value contained in a JSON string: "C0A88DF0" (10 bytes / 80 bits)
- CBOR byte string: 0x44c0a88df0 (5 bytes / 40 bits).
- IPv4 packet: 0xc0a88df0 (4 bytes / 32 bits).

The 13 extra bytes used to format a 4 byte IP address as a dotted quad are useful for display purposes, but provide no information to the receiving application. Directly converting display-oriented JSON data to CBOR format does not achieve the conciseness for which CBOR was designed. Instead, information modeling is key to effectively using both binary data formats such as Protobuf and CBOR and text formats such as XML and JSON.

* *Note: all references to information assume independent uniformly-distributed values. Non-uniform or correlated data has less than one byte of entropy per data byte, but source coding is outside the scope of this specification.*

2.1 Graph Modeling

A JADN information model is a list of type definitions (Section 3.1) in the form of a graph. The graph elements of a node (TypeName, FieldId, FieldName, FieldType) define relationships between nodes.

- A graph is a collection of nodes and relationships between nodes (edges).
- An undirected graph has symmetric edges; the edges in a directed graph have an orientation.
- A tree is a connected acyclic undirected graph, an undirected graph where any pair of nodes is connected by exactly one path.
- A directed (or rooted) tree is a hierarchy. A directed tree is constructed from a tree by selecting one node as root and assigning all edge directions either toward or away from the root. Every rooted tree has an underlying undirected tree.

- A directed acyclic graph (DAG) is a directed graph with no directed cycles, or equivalently a directed graph with a topological ordering, a sequence of nodes such that every edge is directed from earlier to later in the sequence.

Every JADN information model MUST be a collection of one or more DAGs. This ensures that all serialized data has finite and known nesting depth. A DAG can be refactored into another DAG having the same underlying undirected graph but with a different root, and the underlying graphs of two DAGs can be compared for equality. This graph-based equivalence expands the range of data models that can be derived from a single information model.

A DAG differs from a directed tree in that a node may have more than one parent. A DAG can be converted to a directed tree by denormalizing (copying subtrees below multi-parent nodes), and a directed tree can be converted to a DAG by normalizing (combining identical subtrees). Reuse of common types is an important goal in both design of information models and analysis of data, and JADN's flat type structure facilitates and encourages reuse. However, it is often useful to have a tree-structured representation of a document's structure. Converting an information model into a directed tree supports applications such as model queries that are otherwise difficult to implement, tree-structured content statistics, content transformations, and documentation.

2.2 Information Modeling

JADN type definitions are based on the CBOR data model but with an information-centric focus. The information elements of a node (BaseType, TypeOptions, FieldOptions) define the constraints that must be satisfied for an instance to be considered valid. Constraints define the upper bound on the information in an instance.

Data-centric	Information-centric
A data definition language defines a specific data storage and exchange format.	An information modeling language expresses application needs in terms of desired effects.
Serialization-specific details are built into applications.	Serialization is a communication function like compression and encryption, provided to applications.
JSON Schema defines integer as a value constraint on the JSON number type: "integer matches any number with a zero fractional part".	Distinct Integer and Number core types exist regardless of data representation.

Data-centric	Information-centric
CDDL says: "While arrays and maps are only two representation formats, they are used to specify four loosely-distinguishable styles of composition".	Core structured types are based on five distinct composition styles. Each type can be represented in multiple data formats.
No table composition style is defined.	Tables are a fundamental way of organizing information. The Record core type contains tabular information that can be represented as either arrays or maps in multiple data formats.
Instance equality is defined at the data level.	Instance equality is defined in ways meaningful to applications. For example "Optional" and "Nullable" are different at the data level but applications make no logical distinction between "not present" and "null value".
Data-centric design is often Anglocentric, embedding English-language identifiers in protocol data.	Information-centric design encourages definition of natural-language-agnostic protocols while supporting localized text identifiers within applications.

2.3 Example Definitions

Google Protocol Buffers (Protobuf) is a typical data definition language. A Protobuf definition looks like:

```
message Person {
  required string name = 1;
  required int32 id = 2;
  optional string email = 3;
}
```

The corresponding JADN definiton in IDL format (Section 5) is structurally similar to Protobuf, Thrift, ASN.1 and other data definition languages that use named type definitions:

```
Person = Record
   1 name          String
   2 id            Integer
   3 email         String optional
```

JADN is formally defined in Section 3 as structured data expressed in JSON format. JSON is unambiguous and enjoys broad support across programming languages and platforms.


```
["Person", "Record", [], "", [
    [1, "name", "String", [], ""],
    [2, "id", "Integer", [], ""],
    [3, "email", "String", ["[0"], ""]
]]
```

IDL is preferred for use in documentation, but conformance is based on the formal language, and specifications in other formats are validated by converting them to native format. This "data-first" approach allows documentation styles to be adjusted if necessary without affecting the JADN language.

2.4 Implementation

Two general approaches can be used to implement IM-based protocol specifications:

1. Translate the IM to a data-format-specific schema language such Relax-NG, JSON Schema, Protobuf, or CDDL, then use format-specific serialization and validation libraries to process data in the selected format. Applications use data objects specific to each serialization format.
2. Use the IM directly as a format-independent schema language, using IM serialization and validation libraries to process data without a separate schema generation step. Applications use the same IM instances regardless of serialization format, making it easy to bridge from one format to another.

Implementations based on serialization-specific code interoperate with those using an IM serialization library, allowing developers to use either approach.

3 JADN Types

JADN core types are defined in terms of the characteristics they provide to applications. A programming mechanism (variable type, object class, etc.) is conforming if it exhibits the required behavior. A data format is usable if it carries the information needed to support the required behavior.

Table 3-1. JADN Types

JADN Type	Definition
Primitive	
Binary	A sequence of octets. Length is the number of octets.
Boolean	An element with one of two values: true or false.
Integer	A positive or negative whole number.
Number	A real number.
Null	An unspecified or non-existent value, distinguishable from other values such as zero-length String or empty Array.
String	A sequence of characters, each of which has a Unicode codepoint. Length is the number of characters.
Enumeration	
Enumerated	A vocabulary of items where each item has an id and a string value
Structured	
Array	An ordered list of labeled fields with positionally-defined semantics. Each field has a position, label, and type.
ArrayOf(*vtype*)	An ordered list of fields with the same semantics. Each field has a position and type *vtype*.
Choice	A discriminated union: one type selected from a set of named or labeled types.
Map	An unordered map from a set of specified keys to values with semantics bound to each key. Each key has an id and name or label, and is mapped to a value type.

JADN Type	Definition
MapOf(*ktype*, *vtype*)	An unordered map from a set of keys of the same type to values with the same semantics. Each key has key type *ktype*, and is mapped to value type *vtype*.
Record	An ordered map from a list of keys with positions to values with positionally-defined semantics. Each key has a position and name, and is mapped to a value type. Represents a row in a spreadsheet or database table.

- An application that uses JADN types MUST exhibit the behavior specified in Table 3-1. Applications MAY use any programming language data types or mechanisms that exhibit the required behavior.
- An instance of a Map, MapOf, or Record type MUST NOT have more than one occurrence of each key.
- An instance of a Map, MapOf, or Record type MUST NOT have a key of the Null type.
- An instance of a Map, MapOf, or Record type with a key mapped to a Null value MUST compare as equal to an otherwise identical instance without that key.
- The length of an Array, ArrayOf or Record instance MUST not include Null values after the last non-Null value.
- Two Array, ArrayOf or Record instances that differ only in the number of trailing Nulls MUST compare as equal.

UML Section 7.5 "Types and Multiplicity" defines two properties, isUnique and isOrdered, that constrain the kind and number of values contained in a collection. The JADN types that hold homogeneous (single-type) and heterogeneous (defined-type) collections with these properties are:

isOrdered	isUnique	Name	JADN Values	JADN Typed Values
false	true	Set	ArrayOf + set	Map, MapOf, Choice
true	false	Sequence	ArrayOf	Array
true	true	OrderedSet	ArrayOf + unique	Record
false	false	Bag	ArrayOf + unordered	none

Note that a data instance of an unordered mapping type containing multiple elements with the same key is invalid because its values are neither unique nor ordered. In order to reference its values it must be converted into, for example, an array of maps or a map of arrays.

3.1 Type Definitions

JADN type definitions have a fixed structure designed to be easily describable, easily

processed, stable, and extensible. Every definition has five elements:

1. **TypeName:** the name of the type being defined
2. **BaseType:** the JADN type (Table 3-1) of the type being defined
3. **TypeOptions:** an array of zero or more **TypeOption** (Section 3.2.1) applicable to the type being defined
4. **TypeDescription:** a non-normative comment
5. **Fields:** an array of item or field definitions

- TypeName MUST NOT be a JADN type.
- BaseType MUST be a JADN type.
- If BaseType is a Primitive type, ArrayOf, or MapOf, the Fields element MUST be empty:

```
[TypeName, BaseType, [TypeOption, ...], TypeDescription, []]
```

- If BaseType is Enumerated, each item definition MUST have three elements:

  1. **ItemID:** the integer identifier of the item
  2. **ItemValue:** the string value of the item
  3. **ItemDescription:** a non-normative comment

```
[TypeName, BaseType, [TypeOption, ...], TypeDescription, [
    [ItemID, ItemValue, ItemDescription],
    ...
]]
```

- If BaseType is Array, Choice, Map, or Record, each field definition MUST have five elements:

  1. **FieldID:** the integer identifier of the field
  2. **FieldName:** the name or label of the field
  3. **FieldType:** the type of the field, TypeName with optional Namespace ID prefix **NSID:TypeName**
  4. **FieldOptions:** an array of zero or more **FieldOption** (Section 3.2.2) or **TypeOption** (Section 3.2.1) applicable to the field
  5. **FieldDescription:** a non-normative comment

```
[TypeName, BaseType, [TypeOption, ...], TypeDescription, [
    [FieldID, FieldName, FieldType, [FieldOption, TypeOption, ...],
FieldDescription],
    ...
]]
```

- FieldID and FieldName values MUST be unique within a type definition.
- If BaseType is Array or Record, FieldID MUST be the ordinal position of the field within the type, numbered consecutively starting at 1.

- If BaseType is Enumerated, Choice, or Map, FieldID MAY be any nonconflicting integer tag.
- FieldType MUST be a Primitive type, ArrayOf, MapOf, or a Defined type.
- If FieldType is not a JADN Type, FieldOptions MUST NOT contain any TypeOption.
- ItemValue MAY be any string or MAY be constrained to hold a valid FieldName.

Including TypeOption values within FieldOptions is an extension (Section 3.3.1). The Derived Enumerations and Pointers TypeOptions are extensions that supply field definitions and require Fields to be empty.

This structure allows each type definition to be treated as a graph node with minimal JADN-specific knowledge. Node name (TypeName), edge names (FieldID/FieldName), and edge endpoints (FieldType) are found at fixed positions. Other positions can be ignored when reading and filled with empty values when creating a node. Data such as multiplicity and link options can be used enrich a graph if recognized by the application.

3.1.1 Name Formats

JADN does not restrict the syntax of TypeName and FieldName, but naming conventions can aid readability of specifications.

- JADN specifications MAY override the default name formats by defining one or more of:
  - The permitted format for TypeName
  - The permitted format for FieldName
  - A "System" character used in tool-generated or specially-processed type names
  - The permitted format for the Namespace Identifier (NSID) used in type references
- Schema authors MUST NOT create FieldNames containing the JSON Pointer field separator "/", which is reserved for use in the Pointers extension
- Schema authors SHOULD NOT create TypeNames containing the System character, but schema processing tools MAY do so
- Specifications that do not define alternate name formats MUST use the definitions in Figure 3-1 expressed as ABNF and Regular Expression:

```
ABNF:
TypeName   = UC *31("-" / Sys / UC / LC / DIGIT)    ; e.g., Color-Values,
length = 1-32 characters
FieldName  = LC *31("_" / UC / LC / DIGIT)          ; e.g., color_values,
length = 1-32 characters
NSID       = (UC / LC) *7(UC / LC / DIGIT)          ; Namespace ID, length = 1-
8 characters

Sys        = "$"      ; 'DOLLAR SIGN', Used in tool-generated type names, e.g.,
Color$values.
UC         = %x41-5A  ; A-Z
LC         = %x61-7A  ; a-z
DIGIT      = %x30-39  ; 0-9
```



```
Regular Expression:
TypeName:  ^[A-Z][-$A-Za-z0-9]{0,31}$
FieldName: ^[a-z][_A-Za-z0-9]{0,31}$
NSID:      ^[A-Za-z][A-Za-z0-9]{0,7}$
```

Figure 3-1: JADN Default Name Syntax in ABNF and Regular Expression Formats

Specifications MAY use the same syntax for TypeName and FieldName. Using distinct formats may aid understanding but does not affect the meaning of type definitions.

3.1.2 Upper Bounds

Type definitions based on variable-length types may include maximum size limits. If an individual type does not define an explicit limit, it uses the default limit defined by the specification. If the specification does not define a default, the definition uses the limits shown here, which are deliberately conservative to encourage specification authors to define limits based on application requirements.

- JADN specifications SHOULD define size limits on the variable-length values shown in Figure 3-2.
- Specifications that do not define alternate size limits MUST use the values shown in Figure 3-2.
- An instance MUST be considered invalid if its size exceeds the limit specified in its type definition, or the default limit defined in the specification containing its type definition, or if the specification does not define a default, the limit shown in Figure 3-2.

Type	Name	Limit	Description
Binary	MaxBinary	255	Maximum number of octets
String	MaxString	255	Maximum number of characters
Array, ArrayOf, Map, MapOf, Record	MaxElements	100	Maximum number of items/properties

Figure 3-2: JADN Default Size Limits

3.1.3 Descriptions

Description elements (TypeDescription, ItemDescription and FieldDescription) are reserved for comments from schema authors to readers or maintainers of the schema.

- The description value MUST be a string, which MAY be empty.
- Implementations MUST NOT present this string to end users.
- Tools for editing schemas SHOULD support displaying and editing descriptions.
- Implementations MUST NOT take any other action based on the presence, absence, or content of description values.

Description values MAY be used in debug or error output which is intended for developers making use of schemas. Tools that translate other media types or programming languages to

and from a JADN schema MAY choose to convert that media type or programming language's native comments to or from description values. Implementations MAY strip description values at any point during processing.

3.2 Options

This section defines the mechanism used to support a varied set of information needs within the strictly regular structure of Section 3.1. New requirements can be accommodated by defining new options without modifying that structure.

Each option is a text string that may be included in TypeOptions or FieldOptions, encoded as follows:

- The first character is the option ID. Its Unicode codepoint is the numeric value (FieldID) shown in Section 3.2.1 and Section 3.2.2.
- The remaining characters are the option value. Boolean options have no additional characters; if the option ID is present it's value is True, otherwise it's value is False.

3.2.1 Type Options

Type options apply to the type definition as a whole. Structural options are intrinsic elements of the types defined in (Table 3-1). Validation options are optional; if present they constrain which data values are instances of the defined type.

```
TypeOption = Choice
   61 id        Boolean    // '=' Items and Fields are denoted by FieldID
rather than FieldName (Section 3.2.1.1)
   42 vtype     String     // '*' Value type for ArrayOf and MapOf (Section
3.2.1.2)
   43 ktype     String     // '+' Key type for MapOf (Section 3.2.1.3)
   35 enum      String     // '#' Extension: Enumerated type derived from a
specified type (Section 3.3.3)
   62 pointer   String     // '>' Extension: Enumerated type pointers derived
from a specified type (Section 3.3.5)
   47 format    String     // '/' Semantic validation keyword (Section 3.2.1.5)
   37 pattern   String     // '%' Regular expression used to validate a String
type (Section 3.2.1.6)
  121 minf      Number     // 'y' Minimum real number value (Section 3.2.1.7)
  122 maxf      Number     // 'z' Maximum real number value
  123 minv      Integer    // '{' Minimum integer value, octet or character
count, or element count (Section 3.2.1.7)
  125 maxv      Integer    // '}' Maximum integer value, octet or character
count, or element count
  113 unique    Boolean    // 'q' ArrayOf instance must not contain duplicate
values (Section 3.2.1.8)
  115 set       Boolean    // 's' ArrayOf instance is unordered and unique
(Section 3.2.1.9)
```



```
   98 unordered Boolean    // 'b' ArrayOf instance is unordered (Section
3.2.1.10)
   88 extend    Boolean    // 'X' Type has an extension point where fields may
be added (Section 3.2.1.11)
   33 default   String     // '!' Default value (Section 3.2.1.12)
```

- TypeOptions MUST contain zero or one instance of each TypeOption.
- TypeOptions MUST contain only TypeOption instances allowed for BaseType as shown in Table 3-3, plus a default value.
- If BaseType is ArrayOf, TypeOptions MUST include the *vtype* option.
- If BaseType is MapOf, TypeOptions MUST include *ktype* and *vtype* options.

Table 3-3. Allowed Options

BaseType	Allowed Options
Binary	minv, maxv, format
Boolean	
Integer	minv, maxv, format
Number	minf, maxf, format
Null	
String	minv, maxv, format, pattern
Enumerated	id, enum, pointer, extend
Choice	id, extend
Array	extend, format, minv, maxv
ArrayOf	vtype, minv, maxv, unique, set, unordered
Map	id, extend, minv, maxv
MapOf	vtype, ktype, minv, maxv
Record	extend, minv, maxv

3.2.1.1 Field Identifiers

The *id* option used with Enumerated, Choice, and Map types determines how fields are specified in API instances of these types. If the *id* option is absent, API instances use FieldName and the type is referred to as "named". If the *id* option is present, API instances use FieldID and the type is referred to as "labeled". The Record type is always named and

has no *id* option; the Array type is its labeled equivalent.

- In named types, FieldName is a defined name that is included in the semantics of the type, must be populated in the type definition, and may appear in serialized data depending on serialization format.
- In labeled types, FieldName is a suggested label that is not included in the semantics of the type, may be empty in the type definition, and never appears in serialized data regardless of data format.

For example an Enumerated list of HTTP status codes could include the field [403, "Forbidden"]. If the type definition does not include an *id* option, the API value is "Forbidden" and serialization rules determine whether FieldID or FieldName is used in serialized data. With the *id* option the API and serialized values are always the FieldID 403. The label "Forbidden" may be displayed in messages or user interfaces, as could customized labels such as "NotAllowed", "Verboten", or "Interdit".

3.2.1.2 Value Type

The *vtype* option specifies the type of each field in an ArrayOf or MapOf type. It may be any JADN type or Defined type.

- An ArrayOf or MapOf instance MUST be considered invalid if any of its elements is not an instance of *vtype*.

3.2.1.3 Key Type

The *ktype* option specifies the type of each key in a MapOf type.

- *ktype* SHOULD be a Defined type, either an enumeration or a type with constraints such as a pattern or semantic valuation keyword that specify a fixed subset of values that belong to a category.
- A MapOf instance MUST be considered invalid if any of its keys is not an instance of *ktype*.

3.2.1.4 Derived Enumeration

The *enum* option is an extension that creates an Enumerated type derived from a referenced Array, Choice, Map or Record type. (See Section 3.3.3).

3.2.1.5 Semantic Validation

The *format* option value is a semantic validation keyword. Each keyword specifies validation requirements for a fixed subset of values that are accurately described by authoritative resources. The *format* option may also affect how values are serialized, see Section 4.

Table 3-4. Semantic Validation Keywords

Keyword	Type	Requirement
JSON Schema formats	String	All semantic validation keywords defined in Section 7.3 of JSON Schema.
eui	Binary	IEEE Extended Unique Identifier (MAC Address), EUI-48 or EUI-64 as specified in EUI
ipv4-addr	Binary	IPv4 address as specified in RFC 791 Section 3.1
ipv6-addr	Binary	IPv6 address as specified in RFC 8200 Section 3
ipv4-net	Array	Binary IPv4 address and Integer prefix length as specified in RFC 4632 Section 3.1
ipv6-net	Array	Binary IPv6 address and Integer prefix length as specified in RFC 4291 Section 2.3
i8	Integer	Signed 8 bit integer, value must be between -128 and 127.
i16	Integer	Signed 16 bit integer, value must be between -32768 and 32767.
i32	Integer	Signed 32 bit integer, value must be between -2147483648 and 2147483647.
u<*n*>	Integer	Unsigned integer or bit field of <*n*> bits, value must be between 0 and 2^<*n*> - 1.

3.2.1.6 Pattern

The *pattern* option specifies a regular expression used to validate a String instance.

- The *pattern* value SHOULD conform to the Pattern grammar of ECMAScript Section 21.2.
- A String instance MUST be considered invalid if it does not match the regular expression specified by *pattern*.

3.2.1.7 Size and Value Constraints

The *minv* and *maxv* options specify size or integer value limits. The *minf* and *maxf* options specify real number value limits.

- For Binary, String, Array, ArrayOf, Map, MapOf, and Record types:
  - if *minv* is not present, it defaults to zero.
  - if *maxv* is not present or is zero, it defaults to the upper bound specified in Section 3.1.2.

    - a Binary instance MUST be considered invalid if its number of bytes is less than *minv* or greater than *maxv*.
    - a String instance MUST be considered invalid if its number of characters is less than *minv* or greater than *maxv*.
    - an Array, ArrayOf, Map, MapOf, or Record instance MUST be considered invalid if its number of elements is less than *minv* or greater than *maxv*.
- For Integer types:
    - if *minv* is present, an instance MUST be considered invalid if its value is less than *minv*.
    - if *maxv* is present, an instance MUST be considered invalid if its value is greater than *maxv*.
- For Number types:
    - if *minf* is present, an instance MUST be considered invalid if its value is less than *minf*.
    - if *maxf* is present, an instance MUST be considered invalid if its value is greater than *maxf*.

3.2.1.8 Unique Values

The *unique* option specifies that values in an array must not be repeated.

- For the ArrayOf type, if *unique* is present an instance MUST be considered invalid if it contains duplicate values.

3.2.1.9 Set

The *set* option specifies that an ArrayOf type is unordered and unique.

- For the ArrayOf type, if *set* is present an instance MUST be considered invalid if it contains duplicate values.

3.2.1.10 Unordered

The *unordered* option specifies that an ArrayOf type may contain duplicate values and that its values have no defined order. Because values cannot be selected by value or position, it has the semantics of a "bag" or "urn" from which elements are picked at random.

3.2.1.11 Extension Point

The *extend* option is an assertion that an Enumerated, Choice, Array, Map or Record type MAY be incomplete and that future versions MAY add new fields that do not change the definitions of existing fields. This option does not affect the validity of data with respect to a specific schema, it is an indicator that applications may be able to obtain a newer version of the same package for which the data is valid. Types without this option assert that the package identifier will be changed if any field is added, modified, or deleted.

3.2.2.3 Default Value

The *default* option specifies the initial or default value of a field. Applications deserializing a document MUST initialize an unspecified type with its default value. Serialization behavior is not defined; applications MAY omit or populate fields whose values equal the default.

3.2.2 Field Options

Field options may be specified for each field within a structured type definition.

```
FieldOption = Choice
   91 minc      Integer    // '[' Minimum cardinality (Section 3.2.2.1)
   93 maxc      Integer    // ']' Maximum cardinality
   38 tagid     Enumerated // '&' Field containing an explicit tag for this
Choice type (Section 3.2.2.2)
   60 dir       Boolean    // '<' Use FieldName as a path prefix for fields in
FieldType (Extension: Section 3.3.5)
   75 key       Boolean    // 'K' Field is a primary key for this type
(Extension: Section 3.3.6)
   76 link      Boolean    // 'L' Field is a relationship link to a type
instance (Extension: Section 3.3.6)
```

- FieldOptions MUST NOT include more than one of each option.
- All TypeOption values (Section 3.2.1) included in FieldOptions are extensions. Each TypeOption MUST apply to FieldType as defined in Table 3-3.

3.2.2.1 Multiplicity

Multiplicity, as used in the Unified Modeling Language (UML Section 7.5.4), is a range of allowed cardinalities. The *minc* and *maxc* options specify the minimum and maximum cardinality (number of elements) in a field of an Array, Choice, Map, or Record type:

minc	maxc	Multiplicity	Description	Keywords
0	1	0..1	No instances or one instance	optional
1	1	1	Exactly one instance	required
0	0	0..*	Zero or more instances	optional, repeated
1	0	1..*	At least one instance	required, repeated
m	n	m..n	At least m but no more than n instances	required, repeated if m > 1

- if *minc* is not present, it defaults to 1.
- if *maxc* is not present, it defaults to the greater of 1 or *minc*.
- if *maxc* is 0, it defaults to the MaxElements upper bound specified in Section 3.1.2.

- if *maxc* is less than *minc*, the field definition MUST be considered invalid.

If minc is 0, the field is optional, otherwise it is required.
If maxc is 1 the field is a single element, otherwise it is an array of elements.

Multiplicities of optional (0..1) and required (1..1) are part of the JADN core. A field definition with minc other than 0 or 1, or maxc other than 1, is an extension described in Section 3.3.2.

Within a Choice type minc values of 0 and 1 are equivalent because all fields are optional and exactly one must be present. Values greater than 1 specify an array of elements.

3.2.2.2 Discriminated Union with Explicit Tag

The Choice type represents a Discriminated Union, a data structure that could take on several different, but fixed, types. By default a Choice is a Map with exactly one key-value pair, where the key determines the value type. But if a "tag field" (*tagid*) option is present on a Choice field in an Array or Record container, it indicates that a separate Tag field within that container determines the value type.

- The Tag field MUST be an Enumerated type derived from the Choice. It MAY contain a subset of fields from the Choice.

Example:

```
Product = Choice                          // Discriminated union
   1 furniture    Furniture
   2 appliance    Appliance
   3 software     Software

Dept = Enumerated                         // Explicit Tag values derived from the
Choice
   1 furniture
   2 appliance
   3 software

Software = String /uri

Stock1 = Record                           // Discriminated union with intrinsic
tag
   1 quantity     Integer
   2 product      Product                 // Value = Map with one key/value

Stock2 = Record                           // Container with explicitly-tagged
discriminated union
   1 dept         Dept                    // Tag = one key from Choice
   2 quantity     Integer
   3 product      Product(TagId[dept])    // Choice specifying an explicit tag
```



```
field
```

Example JSON serializations of these types are:

Stock1 - Choice with intrinsic tag:

```
{
    "quantity": 395,
    "product": {"software": "http://www.example.com/B902D1P0W37"}
}
```

Stock2 - Choice with explicit tag:

```
{
    "dept": "software",
    "quantity": 395,
    "product": "http://www.example.com/B902D1P0W37"
}
```

Intrinsic tags:

When discriminated unions are grouped the distinction between intrinsic and explicit tags becomes more apparent. A collection with intrinsic tags is simply a Map, which results in what the W3C JSON and XML Transformations Workshop called "Friendly" encodings.

```
    Hashes = Map{1..*}            // Multiple discriminated unions with
intrinsic tag is a Map
       1 md5          Binary{16..16} /x optional
       2 sha1         Binary{20..20} /x optional
       3 sha256       Binary{32..32} /x optional
```

Hashes Example:

```
{
    "sha256":
"C9004978CF5ADA526622ACD4EFED005A980058B7B9972B12F9B3A5D0DA46B7D9",
    "md5": "B64CF5EAF07E86D1697D4EEE96A670B6"
}
```

Explicit tags:

A collection with explicit tags is an array of tag-value pairs. It is more complex to specify, and it results in "UnFriendly" encodings with repeated tag and value keys. Yet because some specifications are written in this style, the "TagId" option exists to designate an explicit tag field to be used to specify the value type.


```
    Hashes2 = ArrayOf(HashVal)    // Multiple discriminated unions with
explicit tags is an Array

    HashVal = Record
       1 algorithm    Enumerated(Enum[HashAlg])  // Tag - one key from Choice
       2 value        HashAlg(TagId[algorithm])  // Value selected from Choice
by 'algorithm' field

    HashAlg = Choice
       1 md5          Binary{16..16} /x
       2 sha1         Binary{20..20} /x
       3 sha256       Binary{32..32} /x
```

Hashes2 Example:

```
[
  {
    "algorithm": "md5",
    "value": "B64CF5EAF07E86D1697D4EEE96A670B6"
  },{
    "algorithm": "sha256",
    "value": "C9004978CF5ADA526622ACD4EFED005A980058B7B9972B12F9B3A5D0DA46B7D9"
  }
]
```

3.3 JADN Extensions

JADN consists of a set of core definition elements, plus several extensions that make type definitions more compact or that support the DRY software design principle. Extensions can be replaced by core definitions without changing the meaning of the definition. In other words, extensions are a form of syntactic sugar. Simplifying ("de-sugaring") reduces the code needed to serialize and validate data and may make specifications easier to understand. But it creates additional definitions that must be kept in sync, expanding the specification and increasing maintenance effort.

The following extensions can be converted to core definitions:

- Anonymous type definition within a field
- Field multiplicity other than required/optional
- Derived enumeration
- MapOf type with Enumerated key type
- Pointers
- Links

3.3.1 Type Definition Within Fields

A type without fields (Primitive types, ArrayOf, MapOf) may be defined anonymously within a field of a structure definition. Simplifying converts all anonymous type definitions to explicit named types and excludes all TypeOption values (Section 3.2.1) from FieldOptions.

Example:

```
Member = Record
   1 name          String
   2 email         String /email
```

Simplifying replaces this with:

```
Member = Record
   1 name          String
   2 email         Member$email

Member$email = String /email           // Tool-generated type definition.
```

3.3.2 Field Multiplicity

Fields may be defined to have multiple values of the same type. Simplifying converts each field that can have more than one value to a separate ArrayOf type. The minimum and maximum cardinality (minc and maxc) FieldOptions (Section 3.2.2) are moved from FieldOptions to the minimum and maximum size (minv and maxv) TypeOptions of the new ArrayOf type, except that if minc is 0 (field is optional), it remains in FieldOptions and the new ArrayOf type defaults to a minimum size of 1.

Example:

```
Roster = Record
   1 org_name      String
   2 members       Member [0..*]          // Optional and repeated: minc=0,
maxc=0
```

Simplifying replaces this with:

```
Roster = Record
   1 org_name      String
   2 members       Roster$members optional// Optional: minc=0, maxc=1

Roster$members = ArrayOf(Member){1..*} // Tool-generated array: minv=1, maxv=0
```

If a list with no elements should be represented as an empty array rather than omitted, its type definition must include an explicit ArrayOf type rather than using the field multiplicity extension:


```
Roster = Record
   1 org_name     String
   2 members      Members       // members field is required: default minc = 1,
maxc = 1

Members = ArrayOf(Member)       // Explicitly-defined array: default minv = 0,
maxv = 0
```

3.3.3 Derived Enumerations

An Enumerated type defined with the *enum* option has fields copied from the type referenced in the option rather than being listed individually in the definition. Simplifying removes *enum* from Type Options and adds fields containing FieldID, FieldName, and FieldDescription from each field of the referenced type.

In JADN-IDL (Section 5.1) the *enum* option is represented as a function string: "Enum(<referenced-type>)". Within ArrayOf and MapOf types, the *ktype* and *vtype* options may contain an enum option. As an example the IDL value "ArrayOf(Enum(Pixel))" corresponds to the JADN vtype option "*#Pixel".

Simplifying references an explicit Enumerated type if it exists, otherwise it creates an explicit Enumerated type. It then replaces the type reference with the name of the explicit Enumerated type.

Example:

```
Pixel = Map
   1 red          Integer
   2 green        Integer
   3 blue         Integer

Channel = Enumerated(Enum[Pixel])       // Derived Enumerated type

ChannelMask = ArrayOf(Enum[Pixel])      // ArrayOf(derived enumeration)
```

Simplifying replaces the Channel and ChannelMask definitions with:

```
Channel2 = Enumerated
   1 red
   2 green
   3 blue

ChannelMask2 = ArrayOf(Channel)
```

3.3.4 MapOf With Enumerated Key

A MapOf type where *ktype* is Enumerated is equivalent to a Map. Simplifying replaces the MapOf type definition with a Map type with keys from the Enumerated *ktype*. This is the complementary operation to derived enumeration. In order to use this extension, each ItemValue of the Enumerated type must be a valid FieldName.

Example:

```
Channel3 = Enumerated
   1 red
   2 green
   3 blue

Pixel3 = MapOf(Channel3, Integer)
```

Simplifying replaces the Pixel MapOf with the explicit Pixel Map shown under Derived Enumerations.

3.3.5 Pointers

Applications may need to model both individual types and collections of types, similar to the way filesystems have files and directories. The "dir" option (Section 3.2.2) marks a field as a collection of types. The dir option has no effect on the structure or serialization of information; its sole purpose is to support pathname generation using the Pointer extension.

A recursive filesystem listing contains pathnames of all files in and under the current directory. The Pointer extension (Section 3.2.1) generates a list of all type definitions in and under the specified type. Simplifying replaces the Pointer extension with an Enumerated type containing a JSON Pointer pathname for each type. If no fields in the specified type are marked with the "dir" option, the Pointer extension has the same fields as the Derived Enumeration extension except that IDs are sequential rather than copied from the referenced type.

Example:

```
Catalog = Record
   1 a             TypeA
   2 b/            TypeB

TypeA = Record
   1 x             Number
   2 y             Number

TypeB = Record
   1 foo           String
   2 bar           Integer

Paths = Enumerated(Pointer[Catalog])
```


In this example, Catalog field "a" is a single type and field "b" is designated as a collection by the "dir" option (shown as "b/"). Simplifying replaces Paths with an Enumerated type containing JSON Pointers to all leaf types in and under Catalog:

```
Paths2 = Enumerated
   1 a                                  // Item 1
   2 b/foo                              // Item 2
   3 b/bar                              // Item 3
```

This is useful when an application 1) needs a category of types, e.g., "Items", 2) defines these types in multiple locations in a hierarchy, and 3) needs identifiers for each type in the category.

It also allows referencing type definitions across specifications. If TypeB is defined in Specification B, its subtypes can be referenced from Specification A under field name "b". This facilitates distributed development of packages regardless of whether the underlying data format has native namespace support.

The structure of a "Catalog" instance is not affected by this extension. Although "a/x" is a valid JSON Pointer to a specific value (57.9), "Catalog" does not define "a" as a dir so "a/x" is not listed in Paths and its value is not considered an "Item":

```
{
  "a": {"x": 57.9, "y": 4.841},     <-- "a" is Item 1 (TypeA)
  "b": {                            <-- "b" is a dir or namespace mount point,
not an Item.
    "foo": "Elephant",              <-- "b/foo" is Item 2 (String)
    "bar": 762                      <-- "b/bar" is Item 3 (TypeC)
  }
}
```

Note that the *enum* and *pointer* extensions create shallow dependencies: the referenced types are needed in order to simplify them but types below the direct references are not.

3.3.6 Links

Types in an information model cannot reference themselves, either directly or indirectly through other types. In other words, a type graph cannot have cycles. But an information model can represent arbitrarily-connected *instance* graphs using links. Links can have any syntax including integers, UUIDs, addresses, format-specific strings such as URIs, or unrestricted strings. The only information modeling requirement is that the identifier of an instance (its primary key) must have the same syntax as any links that reference it (foreign keys).

The link extension automates that requirement: the *key* option within a structured type designates a field as a primary key, and the *link* option designates a reference to an instance of a specified type. The *key* and *link* options do not affect the serialization or validation of

data, but they MAY be used by applications to perform relationship-aware operations such as checking or enforcing referential integrity.

As an example, a Person type might be used to represent friends, family and employment relationships. This example assumes that an Organization type is defined elsewhere with a Key field called 'ein':

```
Person = Record
    1 id        Key(Integer)
    2 name      String
    3 mother    Link(Person)
    4 father    Link(Person)
    5 siblings  Link(Person) [0..*]
    6 friends   Link(Person) [0..*]
    7 employer  Link(Organization) optional
```

Simplifying creates an explicit key type and replaces links with that type, but discards the explicit indicator that a field is a primary key or relationship:

```
Person = Record
    1 id        Person$id
    2 name      String
    3 mother    Person$id
    4 father    Person$id
    5 siblings  Person$id [0..*]
    6 friends   Person$id [0..*]
    7 employer  Organization$ein optional

Person$id = Integer
Organization$ein = String{10..10}
```


4 Serialization

Applications may use any internal information representation that exhibits the characteristics defined in Table 3-1. Serialization rules define how to represent instances of each type using a specific format. Several serialization formats are defined in this section. In order to be usable with JADN, serialization formats defined elsewhere must:

- Specify an unambiguous serialized representation for each JADN type
- Specify how each option applicable to a type affects serialized values
- Specify any validation requirements defined for that format

Data formats are either "schemaless" or "schema-required". Serialization rules for schemaless formats such as JSON, CBOR, and XML should maintain independence of serialization and validation. As an example, the rules for converting XML elements and attributes into an API instance should not depend on the information model. Rules for schema-required data formats such as RFC 791-style field layouts, Protobuf, and Avro should facilitate separation of serialization and validation to the extent practical.

4.1 JSON Serialization

The following serialization rules are used to represent JADN data types in a human-friendly JSON format.

- When using JSON serialization, instances of JADN types without a format option listed in this section MUST be serialized as:

JADN Type	JSON Serialization Requirement
Binary	JSON **string** containing Base64url encoding of the binary value as defined in Section 5 of RFC 4648.
Boolean	JSON **true** or **false**
Integer	JSON **number**
Number	JSON **number**
Null	JSON **null**
String	JSON **string**
Enumerated	JSON **string** ItemValue
Enumerated with "id"	JSON **integer** ItemID

JADN Type	JSON Serialization Requirement
Choice	JSON **object** with one property. Property key is FieldName.
Choice with "id"	JSON **object** with one property. Property key is FieldID converted to string.
Array	JSON **array** of values with types specified by FieldType. Omitted optional values are **null** if before the last specified value, otherwise omitted.
ArrayOf	JSON **array** of values with type *vtype*, or JSON **null** if *vtype* is Null.
Map	JSON **object**. Property keys are FieldNames.
Map with "id"	JSON **object**. Property keys are FieldIDs converted to strings.
MapOf	JSON **object** if *ktype* is a String type, JSON **array** if *ktype* is not a String type, or JSON **null** if *vtype* is Null. Properties have key type *ktype* and value type *vtype*. MapOf types with non-string keys are serialized as in CBOR: a JSON **array** of keys and cooresponding values [key1, value1, key2, value2, ...].
Record	Same as **Map**.

Format options that affect JSON serialization

- When using JSON serialization, instances of JADN types with one of the following format options MUST be serialized as:

Option	JADN Type	JSON Serialization Requirement
x	Binary	JSON **string** containing Base16 (hex) encoding of a binary value as defined in RFC 4648 Section 8. Note that the Base16 alphabet does not include lower-case letters.
ipv4-addr	Binary	JSON **string** containing a "dotted-quad" as specified in RFC 2673 Section 3.2.
ipv6-addr	Binary	JSON **string** containing the text representation of an IPv6 address as specified in RFC 4291 Section 2.2.
ipv4-net	Array	JSON **string** containing the text representation of an IPv4 address range as specified in RFC 4632 Section 3.1.

Option	JADN Type	JSON Serialization Requirement
ipv6-net	Array	JSON **string** containing the text representation of an IPv6 address range as specified in RFC 4291 Section 2.3.

4.2 CBOR Serialization

The following serialization rules are used to represent JADN data types in Concise Binary Object Representation (CBOR) format, where CBOR type #x.y = Major type x, Additional information y.

CBOR type names from Concise Data Definition Language (CDDL) are shown for reference.

- When using CBOR serialization, instances of JADN types without a format option listed in this section MUST be serialized as:

JADN Type	CBOR Serialization Requirement
Binary	**bstr**: a byte string (#2).
Boolean	**bool**: a Boolean value (False = #7.20, True = #7.21).
Integer	**int**: an unsigned integer (#0) or negative integer (#1)
Number	**float64**: IEEE 754 Double-Precision Float (#7.27).
Null	**null**: (#7.22)
String	**tstr**: a text string (#3).
Enumerated	**int**: an unsigned integer (#0) or negative integer (#1) ItemID.
Choice	**struct**: a map (#5) containing one pair. The first item is a FieldID, the second item has the corresponding FieldType.
Array	**record**: an array of values (#4) with types specified by FieldType. Omitted optional values are **null** (#7.22) if before the last specified value, otherwise omitted.
ArrayOf	**vector**: an array of values (#4) of type *vtype*, or **null** (#7.22) if vtype is Null.
Map	**struct**: a map (#5) of pairs. In each pair the first item is a FieldID, the second item has the corresponding FieldType.

JADN Type	CBOR Serialization Requirement
MapOf	**table**: a map (#5) of pairs, or **null** if *vtype* is Null. In each pair the first item has type *ktype*, the second item has type *vtype*.
Record	Same as **Array**.

Format options that affect CBOR Serialization

- When using CBOR serialization, instances of JADN types with one of the following format options MUST be serialized as:

Option	JADN Type	CBOR Serialization Requirement
f16	Number	**float16**: IEEE 754 Half-Precision Float (#7.25).
f32	Number	**float32**: IEEE 754 Single-Precision Float (#7.26).

4.3 M-JSON Serialization:

Minimized JSON serialization rules represent JADN data types in a compact format suitable for machine-to-machine communication. They produce JSON instances equivalent to the diagnostic notation of CBOR instances.

- When using M-JSON serialization, instances of JADN types MUST be serialized as:

JADN Type	M-JSON Serialization Requirement
Binary	JSON **string** containing Base64url encoding of the binary value as defined in Section 5 of RFC 4648.
Boolean	JSON **true** or **false**
Integer	JSON **number**
Number	JSON **number**
Null	JSON **null**
String	JSON **string**
Enumerated	JSON **integer** ItemID
Choice	JSON **object** with one property. Property key is the FieldID converted to string.

JADN Type	M-JSON Serialization Requirement
Array	JSON **array** of values with types specified by FieldType. Unspecified values are **null** if before the last specified value, otherwise omitted.
ArrayOf	JSON **array** of values with type *vtype*, or JSON **null** if *vtype* is Null.
Map	JSON **object**. Property keys are FieldIDs converted to strings.
MapOf	JSON **object** if *ktype* is a String type, JSON **array** if *ktype* is not a String type, or JSON **null** if *vtype* is Null. Members have key type *ktype* and value type *vtype*. MapOf types with non-string keys are serialized as in CBOR: a JSON **array** of keys and cooresponding values [key1, value1, key2, value2, ...].
Record	Same as **Array**.

4.4 XML Serialization:

This section is informative. Normative XML serialization rules will be defined in a future version of this specification.

- When using XML serialization, instances of JADN types without a format option listed in this section MUST be serialized as:

JADN Type	XML Serialization Requirement
Binary	<xs:element name="FieldName" type="xs:base64Binary"/>
Boolean	<xs:attribute name="FieldName" type="xs:boolean"/>
Integer	<xs:element name="FieldName" type="xs:integer"/>
Number	<xs:element name="FieldName" type="xs:decimal"/>
Null	<xs:attribute name="FieldName" xsi:nil="true"/>
String	<xs:element name="FieldName" type="xs:string"/>
Enumerated	<xs:element name="FieldName" type="xs:string"/> ItemValue of the selected item
Choice	<xs:element name="FieldName"/> containing one element with name FieldName of the selected field

JADN Type	XML Serialization Requirement
Array	<xs:element name="FieldName"/> containing elements with name FieldName of each field
ArrayOf	<xs:element name="FieldName"/> containing elements with the same FieldName for all fields
Map	<xs:element name="FieldName"/> containing "MapEntry" elements with "key=" attribute
MapOf	<xs:element name="FieldName"/> containing "MapEntry" elements with "key=" attribute
Record	same as **Map**

Format options that affect XML serialization

- When using XML serialization, instances of JADN types with one of the following format options MUST be serialized as:

Option	JADN Type	XML Serialization Requirement
x	Binary	<xs:element name="FieldName" type="xs:hexBinary"/>
i8	Integer	<xs:element name="FieldName" type="xs:byte"/>
i16	Integer	<xs:element name="FieldName" type="xs:short"/>
i32	Integer	<xs:element name="FieldName" type="xs:int"/>
u1..u8	Integer	<xs:element name="FieldName" type="xs:unsignedByte"/>
u9..u16	Integer	<xs:element name="FieldName" type="xs:unsignedShort"/>
u17..u32	Integer	<xs:element name="FieldName" type="xs:unsignedInt"/>
u33..u*	Integer	<xs:element name="FieldName" type="xs:nonNegativeInteger"/>

5 Definition Formats

Section 3.1 defines the native JSON format of JADN type definitions. Although JSON data is unambiguous and supported in many programming languages, it is cumbersome to use as a documentation format. This section defines a formal text-based interface definition language and illustrates several alternative ways of documenting information models.

5.1 JADN-IDL Format

This section is normative

JADN Interface Definition Language (IDL) is a textual representation of JADN type definitions. It replicates the structure of Section 3.1 but combines each type and its options into a single string formatted for readability. The conversion between JSON and JADN-IDL formats is lossless in both directions.

The JADN-IDL definition formats are:

Primitive types:

```
    TypeName = TYPESTRING                       // TypeDescription
```

Enumerated type:

```
    TypeName = TYPESTRING                       // TypeDescription
        ItemID ItemValue                        // ItemDescription
        ...
```

Structured types without the *id* option:

```
    TypeName = TYPESTRING                       // TypeDescription
        FieldID FieldName[/] FIELDSTRING        // FieldDescription
        ...
```

If a field includes the *dir* FieldOption, the SOLIDUS character (/) as specified in RFC 6901 is appended to FieldName.

Structured types with the *id* option treat the item/field name as an informative label (see Section 3.2.1.1) and display it in the description followed by a label terminator ("::"):

```
    /* Enumerated.ID */
    TypeName = TYPESTRING                       // TypeDescription
        ItemID                                  // ItemValue:: ItemDescription

    /* Choice.ID, Map.ID */
```



```
    TypeName = TYPESTRING                       // TypeDescription
        FieldID FIELDSTRING                     // FieldName[/]::
FieldDescription
        ...
```

Type Options:

TYPESTRING is the value of BaseType or FieldType, followed by string representations of the type options, if applicable to TYPE as specified in Table 3-3.

```
TYPESTRING    = TYPE [".ID"] [S1] [VRANGE] [FORMAT]   ; TYPE is the value of
BaseType or FieldType
S1            = "(" *ktype* "," *vtype* ")"           ; if TYPE is MapOf
              | "(" *vtype* ")"                       ; if TYPE is ArrayOf
              | "(Enum[" *enum* "])"                  ; if TYPE is Enumerated
              | "(Pointer[" *enum* "])"               ; if TYPE is Enumerated
              | "(%" *pattern* "%)"                   ; if TYPE is String
VRANGE        = "{" *minv* ".." *maxv* "}"
FORMAT        = " /" *format*
```

Field Options:

FIELDSTRING is the value of TYPESTRING combined with string representations of two mutually-exclusive field options:

```
FIELDSTRING   = TYPESTRING [MULTIPLICITY | TAGID]
MULTIPLICITY  = "[" *minc* ".." *maxc* "]"
              | " optional"
TAGID        = "(TagId[" *tagid* "])"
```

An ABNF grammar for JADN-IDL is shown in Appendix F.

5.2 Table Style

This section is informative

Some specifications present type definitions in property table form, using varied style conventions. This specification does not define a normative property table format, but this section is one example of how JADN definitions may be displayed as property tables.

This style is structurally similar to JADN-IDL and uses its TYPESTRING syntax, but breaks out the MULTIPLICITY field option into a separate column:

```
+----------+-----------+----------------+
| TypeName | TYPESTRING | TypeDescription |
+----------+-----------+----------------+
```


followed by (for structured types without the *id* option):

```
+---------+--------------+------------+--------+-----------------+
| FieldID | FieldName[/]  | FIELDSTRING | [m..n] | FieldDescription |
+---------+--------------+------------+--------+-----------------+
```

or (for structured types with the *id* option):

```
+---------+------------+--------+---------------------------------+
| FieldID | FIELDSTRING | [m..n] | FieldName[/]:: FieldDescription  |
+---------+------------+--------+---------------------------------+
```

Example Markdown Table:

Type: Person (Record)

ID	Name	Type	#	Description
1	**name**	String	1	
2	**id**	Integer	1	
3	**email**	String	0..1	

Example HTML Table:

Person = Record

1	name	String	1
2	id	Integer	1
3	email	String	0..1

5.3 Entity Relationship Diagrams

This section is informative

JADN type definitions have a graphical representation similar to ERDs used in database design. This document does not address the design of information models, but processes similar to those used for databases and software may be used.

The differences between database and information ERDs are:

1. An information model is defined entirely by the entities (nodes) shown on an information ERD. Edges are derived from the node content and are shown to visually illustrate the information model. Static relationships are directed arrows from containing to contained

types and may include multiplicity information. Dynamic relationships (links) are directed from foreign key to primary key, the reverse direction from container relationships. Links may be illustrated as undirected edges, or as directed edges visually distinguishable from static edges.

2. In an information ERD attributes have both an ID and a name, each of which must be unique within the node.

3. All entities in a relational database ERD are tables, while entities in an information ERD are type definitions. A type definition contains both the name of the type being defined and its base type, for example "Person : Record" or "EmailAddr : String".

As an example, Figure 5-1 is a database ERD from a diagramming tool's template collection.

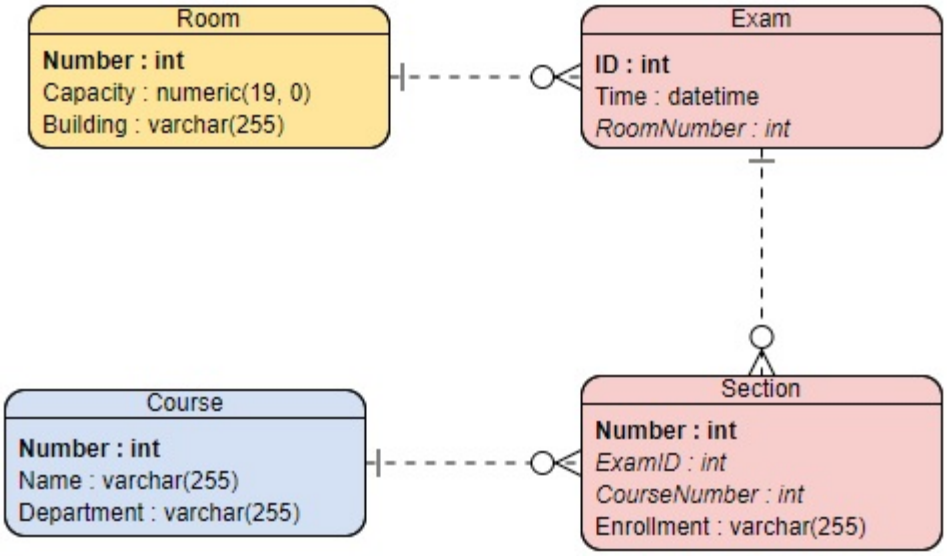

Figure 5-1. Database Entity Relationship Diagram

An Information ERD can be derived from that example by selecting an appropriate entity as root, assigning the structured type Record to each entity, and showing connectors from containing to contained types.

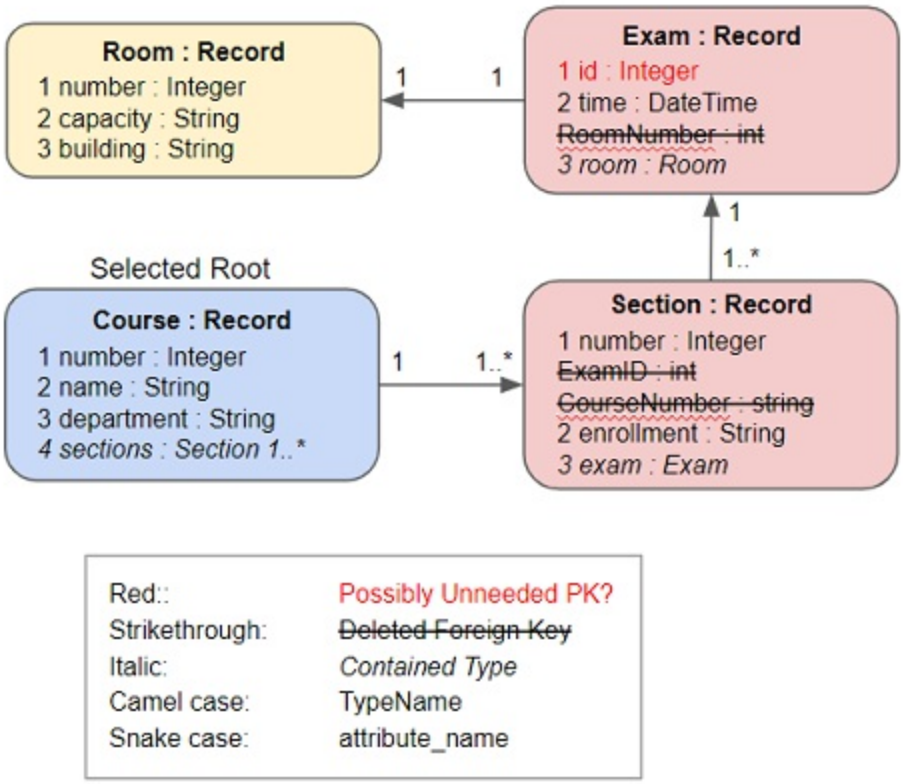

Figure 5-2. Information Model Entity Relationship Diagram

Different information models can be derived from the same database schema depending on which types are designated as roots. In this example the logical choice for root is "Course", resulting in data values like:

```
{ "course": {
    "number": 7241,
    "name": "Algebra 2",
    "department": "Math",
    "sections": [{
      "number": 2,
      "enrollment": "foo",
      "exam": {
        "id": 8231,
        "time": "3/19/20 14:00",
        "room": {
          "number": 107,
          "capacity": 42,
          "building": "Stewart"
}}}]}}
```

"Exam" may be less desirable as a root for an information model: it either duplicates course

information in each section of a course or relies on a separate course index (Map):

```
{ "exam": {
    "id": 8231,
    "time": "3/19/20 14:00",
    "room": {
      "number": 107,
      "capacity": 42,
      "building": "Stewart"
    },
    "sections": [{
      "number": 2,
      "enrollment": "foo",
      "course": {
        "number": 7241,
        "name": "Algebra 2",
        "department": "Math"
}}]}}
```

5.4 Tree Diagrams

This section is informative

Tree diagrams provide a simplified graphical overview of an information model. The structure of a JADN IM can be displayed as a YANG tree diagram using the following conventions:

TBSL

6 Schemas

JADN schemas are organized into packages. A package consists of an optional information section and a list of type definitions:

```
    Schema = Record                              // Definition of a JADN package
       1 info         Information optional       // Information about this
package
       2 types        Types                      // Types defined in this package
```

If the information section is present the *package* field is required; all others are optional.

- **package:** A namespace URI that allows type definitions in this package to be unambiguously referenced from other packages. This is an identifier but not necessarily a locator for accessible resources. The namespace may include major or major.minor versioning information, such as http://example.com/acme2 or http://example.com/acme/v1.3.
- **version:** Incremental version of this package, a string that compares lexicographically higher than previous versions. The *namespaces* field references only package namespaces. Version may be used to determine the most recent definition of a namespace.
- **title:** A short name for this package.
- **description:** A brief description of purpose or capabilities of this package
- **comment:** Any other information applicable to the package.
- **copyright:** A copyright notice.
- **license:** License for this package. Value is an SPDX licenseId, CC0-1.0 is recommended.
- **namespaces:** Local map of NSIDs (short names) to namespaces. Used within this package to reference types defined in other packages.
- **exports:** Root types. All types defined in a package can be referenced under the package's namespace. Exports may be used by schema tools to detect unused types or prune when copying definitions between files.
- **config:** Values such as name formats and size limits that are customized for this package.

7 Operational Considerations

TBSL

- Serialization (bulk vs pull)
- Validation (integrated with serialization, separate)
- Localization
- Schema embedding - self-describing data
- Bridging
- Tabular data (not too many optional columns, sort fields by required/optional. Tuples.)

8 Security Considerations

This document presents a language for expressing the information needs of communicating applications, and rules for generating data structures to satisfy those needs. As such, it does not inherently introduce security issues, although protocol specifications based on JADN naturally need security analysis when defined. Such specifications need to follow the guidelines in RFC 3552.

Additional security considerations applicable to JADN-based specifications:

- The JADN language could cause confusion in a way that results in security issues. Clarity and unambiguity of this specification could always be improved through operational experience and developer feedback.
- Where a JADN data validator is part of a system, the security of the system benefits from automatic data validation but depends on both the specificity of the JADN specification and the correctness of the validation implementation. Tightening the specification (e.g., by defining upper bounds and other value constraints) and testing the validator against unreasonable data instances can address both concerns.

Security and bandwidth efficiency are benefits of using an information model. Enumerating strings and map keys defines the information content of those values, which greatly reduces opportunities for exploitation. A firewall with a security policy of "Allow specific things I understand plus everything I don't understand" is less secure than a firewall that allows only things that are understood. The "Must-Ignore" policy of RFC 7493 compromises security by allowing everything that is not understood. Information modeling's "Must-Understand" approach enhances security and accommodates new protocol elements by adding them to the IM's enumerated lists of things that are understood. An executable IM format such as JADN provides the agility required to support evolving protocols.

Writers of JADN specifications are strongly encouraged to value simplicity and transparency of the specification. Although JADN makes it easier to both define and understand complex specifications, complexity that is not essential to satisfying operational requirements is itself a security concern.

9 Conformance

Conformance targets: This document defines two conformance levels for JADN implementations: Core and Extensions.

This document defines several data formats. Conformance claims are made with respect to a specified data format, and conforming implementations must support at least one data format.

- Core JADN
  - Validate schema packages according to Section 3.1, Section 3.2 and section 6
  - Validate API values against a schema package
  - Encode and decode documents according to serialization rules for data format <X> defined in Section Section 4
- JADN Extensions
  - Satisfy all Core requirements
  - Translate JADN packages bi-directionally between JSON format and JADN IDL format defined in Section 5.1
  - Perform all type simplification operations defined in Section 3.3

This document describes several schema support functions but defines no corresponding conformance requirements:

- JADN Extensions
  - Recognize and reverse type simplification operations, i.e., given a core schema package, generate syntactic sugar where applicable.
- JADN Schema Translator
  - Translate JADN packages to informative documentation formats (table, diagram, tree) described in Section 5
- JADN Concrete Schema Generators
  - Generate format-specific concrete schemas per serialization rules in Section 4.x.

Appendix A. Acknowledgments

The following individuals have participated in the creation of this specification and are gratefully acknowledged:

OpenC2 TC Members:

First Name	Last Name	Company
Brian	Berliner	Symantec
Joseph	Brule	National Security Agency
Toby	Considine	University of North Carolina
Jason	Romano	General Dynamics

External Reviewers:

First Name	Last Name	Company
Carsten	Bormann	Universität Bremen
Hans-Jürgen	Rennau	parsQube GmbH

Appendix B. Revision History

Revision	Date	Editor	Changes Made
jadn-v1.0-wd01	2020-10-21	David Kemp	Initial working draft

Appendix C. JADN Meta-schema

A meta-schema is a schema against which other schemas can be validated. The JADN meta-schema validates itself and other JADN schemas. In order to validate itself, the meta-schema requires a name format change from the JADN default (Section 3.1.1):

- FieldName needs to allow configuration variables beginning with '$' and capitalized JADN types

```
  "config": {
    "$FieldName": "^[$A-Za-z][_A-Za-z0-9]{0,31}$"
  }
```

C.1 Package

A package is a collection of type definitions along with information about the package.

```
       title: "JADN Metaschema"
     package: "http://oasis-open.org/jadn/v1.0/schema"
 description: "Syntax of a JSON Abstract Data Notation (JADN) package."
     license: "CC0-1.0"
     exports: ["Schema"]
      config: {"$FieldName": "^[$A-Za-z][_A-Za-z0-9]{0,31}$"}

Schema = Record                                 // Definition of a JADN package
   1 info         Information optional          // Information about this package
   2 types        Types                         // Types defined in this package

Information = Map                               // Information about this package
   1 package      Namespace                     // Unique name/version of this
package
   2 version      String{1..*} optional         // Incrementing version within
package
   3 title        String{1..*} optional         // Title
   4 description  String{1..*} optional         // Description
   5 comment      String{1..*} optional         // Comment
   6 copyright    String{1..*} optional         // Copyright notice
   7 license      String{1..*} optional         // SPDX licenseId (e.g., 'CC0-
1.0')
   8 namespaces   Namespaces optional           // Referenced packages
   9 exports      Exports optional              // Type defs exported by this
package
  10 config       Config optional               // Configuration variables

Namespaces = MapOf(NSID, Namespace){1..*}       // Packages with referenced type
```



```
defs

Exports = ArrayOf(TypeName){1..*}            // Type defs intended to be
referenced

Config = Map{1..*}                           // Config vars override JADN
defaults
   1 $MaxBinary   Integer{1..*} optional     // Schema default max octets
   2 $MaxString   Integer{1..*} optional     // Schema default max characters
   3 $MaxElements Integer{1..*} optional     // Schema default max
items/properties
   4 $Sys         String{1..1} optional      // System character for TypeName
   5 $TypeName    String{1..127} optional    // TypeName regex
   6 $FieldName   String{1..127} optional    // FieldName regex
   7 $NSID        String{1..127} optional    // Namespace Identifier regex
```

C.2 Type Definitions

The structure of JADN type definitions (Section 3.1) is intended to remain stable, with options providing extensibility.

```
Types = ArrayOf(Type)
Type = Array
   1  TypeRef                                // type_name::
   2  BaseType                               // base_type::
   3  Options                                // type_options::
   4  Description                            // type_description::
   5  JADN-Type(TagId[base_type])            // fields::

BaseType = Enumerated
   1 Binary
   2 Boolean
   3 Integer
   4 Number
   5 Null
   6 String
   7 Enumerated
   8 Choice
   9 Array
  10 ArrayOf
  11 Map
  12 MapOf
  13 Record

JADN-Type = Choice
   1 Binary       Empty
```



```
   2 Boolean      Empty
   3 Integer      Empty
   4 Number       Empty
   5 Null         Empty
   6 String       Empty
   7 Enumerated   Items
   8 Choice       Fields
   9 Array        Fields
  10 ArrayOf      Empty
  11 Map          Fields
  12 MapOf        Empty
  13 Record       Fields

Empty = Array{0..0}
Items = ArrayOf(Item)
Item = Array
   1  FieldID                                  // item_id::
   2  String                                   // item_value::
   3  Description                              // item_description::

Fields = ArrayOf(Field)
Field = Array
   1  FieldID                                  // field_id::
   2  FieldName                                // field_name::
   3  TypeRef                                  // field_type::
   4  Options                                  // field_options::
   5  Description                              // field_description::

FieldID = Integer{0..*}
Options = ArrayOf(Option){0..10}
Option = String{1..*}
Description = String
Namespace = String /uri                        // Unique name of a package
NSID = String{pattern="$NSID"}                 // Default = ^[A-Za-z][A-Za-z0-9]
{0,7}$
TypeName = String{pattern="$TypeName"}         // Default = ^[A-Z][-$A-Za-z0-9]
{0,31}$
FieldName = String{pattern="$FieldName"}       // Default = ^[a-z][_A-Za-z0-9]
{0,31}$
TypeRef = String                               // Autogenerated ($NSID ':')?
$TypeName
```


Appendix D. Definitions in JADN format

This appendix contains the JADN definitions corresponding to all JADN-IDL definitions in this document.

Section 2.3 Example Definitions:

```
["Person", "Record", [], "", [
    [1, "name", "String", [], ""],
    [2, "id", "Integer", [], ""],
    [3, "email", "String", ["[0"], ""]
]]
```

Section 3.2.2.2 Discriminated Union with Explicit Tag:

```
  ["Product", "Choice", [], "Discriminated union", [
    [1, "furniture", "Furniture", [], ""],
    [2, "appliance", "Appliance", [], ""],
    [3, "software", "Software", [], ""]
  ]],
  ["Dept", "Enumerated", [], "Explicit Tag values derived from the Choice", [
    [1, "furniture", ""],
    [2, "appliance", ""],
    [3, "software", ""]
  ]],
  ["Software", "String", ["/uri"], "", []],
  ["Stock1", "Record", [], "Discriminated union with intrinsic tag", [
    [1, "quantity", "Integer", [], ""],
    [2, "product", "Product", [], "Value = Map with one key/value"]
  ]],
  ["Stock2", "Record", [], "Container with explicitly-tagged discriminated
union", [
    [1, "dept", "Dept", [], "Tag = one key from Choice"],
    [2, "quantity", "Integer", [], ""],
    [3, "product", "Product", ["&1"], "Choice specifying an explicit tag
field"]
  ]],
  ["Hashes", "Map", ["{1"], "Multiple discriminated unions with intrinsic tags
is a Map", [
    [1, "md5", "Binary", ["/x", "{16", "}16", "[0"], ""],
    [2, "sha1", "Binary", ["/x", "{20", "}20", "[0"], ""],
    [3, "sha256", "Binary", ["/x", "{32", "}32", "[0"], ""]
  ]],
  ["Hashes2", "ArrayOf", ["*HashVal"], "Multiple discriminated unions with
explicit tags is an Array", []],
```



```
  ["HashVal", "Record", [], "", [
    [1, "algorithm", "Enumerated", ["#HashAlg"], "Tag - one key from Choice"],
    [2, "value", "HashAlg", ["&1"], "Value selected from Choice by 'algorithm'
field"]
  ]],
  ["HashAlg", "Choice", [], "", [
    [1, "md5", "Binary", ["/x", "{16", "}16"], ""],
    [2, "sha1", "Binary", ["/x", "{20", "}20"], ""],
    [3, "sha256", "Binary", ["/x", "{32", "}32"], ""]
  ]]
```

Section 3.3.1 Type Definition Within Fields:

```
["Member", "Record", [], "", [
    [1, "name", "String", [], ""],
    [2, "email", "String", ["/email"], ""]
]],
["Member2", "Record", [], "", [
    [1, "name", "String", [], ""],
    [2, "email", "Member2$email", [], ""]
]],
["Member2$email", "String", ["/email"], "Tool-generated type definition.", []]
```

Section 3.3.2 Field Multiplicity:

```
["Roster", "Record", [], "", [
    [1, "org_name", "String", [], ""],
    [2, "members", "Member", ["[0", "]0"], "Optional and repeated: minc=0,
maxc=0"]
]],
["Roster2", "Record", [], "", [
    [1, "org_name", "String", [], ""],
    [2, "members", "Roster2$members", ["[0"], "Optional: minc=0, maxc=1"]
]],
["Roster2$members", "ArrayOf", ["*Member", "{1"], "Tool-generated array:
minv=1, maxv=0", []],
["Roster3", "Record", [], "", [
    [1, "org_name", "String", [], ""],
    [2, "members", "Members", [], "members field is required: default minc = 1,
maxc = 1"]
]],
["Members", "ArrayOf", ["*Member"], "Explicitly-defined array: default minv =
0, maxv = 0", []]
```

Section 3.3.3 Derived Enumerations:


```
["Channel", "Enumerated", ["#Pixel"], "Derived Enumerated type", []],
["ChannelMask", "ArrayOf", ["*#Pixel"], "ArrayOf(derived enumeration)", []],
["Channel2", "Enumerated", [], "", [
    [1, "red", ""],
    [2, "green", ""],
    [3, "blue", ""]
]],
["ChannelMask2", "ArrayOf", ["*Channel"], "", []]
```

Section 3.3.4 MapOf with Enumerated Key:

Note that the order of elements in **TypeOptions** and **FieldOptions** is not significant.

```
["Channel3", "Enumerated", [], "", [
    [1, "red", ""],
    [2, "green", ""],
    [3, "blue", ""]
]],
["Pixel3", "MapOf", ["+Channel3", "*Integer"], "", []]
```

Section 3.3.5 Pointers:

```
["Catalog", "Record", [], "", [
    [1, "a", "TypeA", [], ""],
    [2, "b", "TypeB", ["<"], ""]
]],
["TypeA", "Record", [], "", [
    [1, "x", "Number", [], ""],
    [2, "y", "Number", [], ""]
]],
["TypeB", "Record", [], "", [
    [1, "foo", "String", [], ""],
    [2, "bar", "Integer", [], ""]
]],
["Paths", "Enumerated", [">Catalog"], "", []],
["Paths2", "Enumerated", [], "", [
    [1, "a", "Item 1"],
    [2, "b/foo", "Item 2"],
    [3, "b/bar", "Item 3"]
]]
```

Section 3.3.6 Links:

```
  ["Person", "Record", [], "", [
    [1, "id", "Integer", ["K"], ""],
```



```
    [2, "name", "String", [], ""],
    [3, "mother", "Person", ["L"], ""],
    [4, "father", "Person", ["L"], ""],
    [5, "siblings", "Person", ["[0", "]0", "L"], ""],
    [6, "friends", "Person", ["[0", "]0", "L"], ""],
    [7, "employer", "Organization", ["[0", "L"], ""]
  ]],
  ["Person", "Record", [], "", [
    [1, "id", "Person$id", [], ""],
    [2, "name", "String", [], ""],
    [3, "mother", "Person$id", [], ""],
    [4, "father", "Person$id", ["], ""],
    [5, "siblings", "Person$id", ["[0", "]0"], ""],
    [6, "friends", "Person$id", ["[0", "]0"], ""],
    [7, "employer", "Organization$ein", ["[0"], ""]
  ]],
  ["Person$id", "Integer", [], "", []],
  ["Organization$ein", "String", ["{10", "}10"], "", []]
```

Appendix C. JADN Meta-schema:

```
{
 "info": {
  "title": "JADN Metaschema",
  "package": "http://oasis-open.org/jadn/v1.0/schema",
  "description": "Syntax of a JSON Abstract Data Notation (JADN) package.",
  "license": "CC0-1.0",
  "exports": ["Schema"],
  "config": {
   "$FieldName": "^[$A-Za-z][_A-Za-z0-9]{0,31}$"
  }
 },
 "types": [
  ["Schema", "Record", [], "Definition of a JADN package", [
    [1, "info", "Information", ["[0"], "Information about this package"],
    [2, "types", "Types", [], "Types defined in this package"]
  ]],
  ["Information", "Map", [], "Information about this package", [
    [1, "package", "Namespace", [], "Unique name/version of this package"],
    [2, "version", "String", ["{1", "[0"], "Incrementing version within
package"],
    [3, "title", "String", ["{1", "[0"], "Title"],
    [4, "description", "String", ["{1", "[0"], "Description"],
    [5, "comment", "String", ["{1", "[0"], "Comment"],
    [6, "copyright", "String", ["{1", "[0"], "Copyright notice"],
    [7, "license", "String", ["{1", "[0"], "SPDX licenseId (e.g., 'CC0-1.0')"],
```



```
    [8, "namespaces", "Namespaces", ["[0"], "Referenced packages"],
    [9, "exports", "Exports", ["[0"], "Type defs exported by this package"],
    [10, "config", "Config", ["[0"], "Configuration variables"]
  ]],
  ["Namespaces", "MapOf", ["*Namespace", "+NSID", "{1"], "Packages with
referenced type defs", []],
  ["Exports", "ArrayOf", ["*TypeName", "{1"], "Type defs intended to be
referenced", []],
  ["Config", "Map", ["{1"], "Config vars override JADN defaults", [
    [1, "$MaxBinary", "Integer", ["{1", "[0"], "Schema default max octets"],
    [2, "$MaxString", "Integer", ["{1", "[0"], "Schema default max
characters"],
    [3, "$MaxElements", "Integer", ["{1", "[0"], "Schema default max
items/properties"],
    [4, "$Sys", "String", ["{1", "}1", "[0"], "System character for TypeName"],
    [5, "$TypeName", "String", ["{1", "}127", "[0"], "TypeName regex"],
    [6, "$FieldName", "String", ["{1", "}127", "[0"], "FieldName regex"],
    [7, "$NSID", "String", ["{1", "}127", "[0"], "Namespace Identifier regex"]
  ]],
  ["Types", "ArrayOf", ["*Type"], "", []],
  ["Type", "Array", [], "", [
    [1, "type_name", "TypeRef", [], ""],
    [2, "base_type", "BaseType", [], ""],
    [3, "type_options", "Options", [], ""],
    [4, "type_description", "Description", [], ""],
    [5, "fields", "JADN-Type", ["&2"], ""]
  ]],
  ["BaseType", "Enumerated", [], "", [
    [1, "Binary", ""],
    [2, "Boolean", ""],
    [3, "Integer", ""],
    [4, "Number", ""],
    [5, "Null", ""],
    [6, "String", ""],
    [7, "Enumerated", ""],
    [8, "Choice", ""],
    [9, "Array", ""],
    [10, "ArrayOf", ""],
    [11, "Map", ""],
    [12, "MapOf", ""],
    [13, "Record", ""]
  ]],
  ["JADN-Type", "Choice", [], "", [
    [1, "Binary", "Empty", [], ""],
    [2, "Boolean", "Empty", [], ""],
    [3, "Integer", "Empty", [], ""],
```



```
    [4, "Number", "Empty", [], ""],
    [5, "Null", "Empty", [], ""],
    [6, "String", "Empty", [], ""],
    [7, "Enumerated", "Items", [], ""],
    [8, "Choice", "Fields", [], ""],
    [9, "Array", "Fields", [], ""],
    [10, "ArrayOf", "Empty", [], ""],
    [11, "Map", "Fields", [], ""],
    [12, "MapOf", "Empty", [], ""],
    [13, "Record", "Fields", [], ""]
  ]],
  ["Empty", "Array", ["}0"], "", []],
  ["Items", "ArrayOf", ["*Item"], "", []],
  ["Item", "Array", [], "", [
    [1, "item_id", "FieldID", [], ""],
    [2, "item_value", "String", [], ""],
    [3, "item_description", "Description", [], ""]
  ]],
  ["Fields", "ArrayOf", ["*Field"], "", []],
  ["Field", "Array", [], "", [
    [1, "field_id", "FieldID", [], ""],
    [2, "field_name", "FieldName", [], ""],
    [3, "field_type", "TypeRef", [], ""],
    [4, "field_options", "Options", [], ""],
    [5, "field_description", "Description", [], ""]
  ]],
  ["FieldID", "Integer", ["{0"], "", []],
  ["Options", "ArrayOf", ["*Option", "}10"], "", []],
  ["Option", "String", ["{1"], "", []],
  ["Description", "String", [], "", []],
  ["Namespace", "String", ["/uri"], "Unique name of a package", []],
  ["NSID", "String", ["%$NSID"], "Default = ^[A-Za-z][A-Za-z0-9]{0,7}$", []],
  ["TypeName", "String", ["%$TypeName"], "Default = ^[A-Z][-$A-Za-z0-9]
{0,31}$", []],
  ["FieldName", "String", ["%$FieldName"], "Default = ^[a-z][_A-Za-z0-9]
{0,31}$", []],
  ["TypeRef", "String", [], "Autogenerated ($NSID ':')? $TypeName", []]
 ]
}
```


Appendix E. JSON Schema for JADN

A JADN package has the following structure:

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "$id": "https://oasis-open.org/openc2/jadn/v1.0",
  "description": "Validates structure of a JADN schema, does not check values",
  "type": "object",
  "required": ["types"],
  "additionalProperties": false,
  "properties": {
    "info": {
      "type": "object",
      "required": ["package"],
      "additionalProperties": false,
      "properties": {
        "package": {"type": "string"},
        "version": {"type": "string"},
        "title": {"type": "string"},
        "description": {"type": "string"},
        "comment": {"type":  "string"},
        "copyright": {"type": "string"},
        "license": {"type": "string"},
        "namespaces": {"$ref": "#/definitions/Namespaces"},
        "exports": {"$ref": "#/definitions/Exports"},
        "config": {"$ref": "#/definitions/Config"}
      }
    },
    "types": {
      "type": "array",
      "items": {
        "type": "array",
        "minItems": 2,
        "maxItems": 5,
        "items": [
          {"$ref": "#/definitions/TypeRef"},
          {"$ref": "#/definitions/BaseType"},
          {"$ref": "#/definitions/Options"},
          {"$ref": "#/definitions/Description"},
          {"$ref": "#/definitions/Fields"}
        ]
      }
    }
  },
```



```
  "definitions": {
    "Namespaces": {
      "type": "object",
      "propertyNames": {"$ref": "#/definitions/NSID"},
      "patternProperties": {
        "": {
          "type": "string",
          "format": "uri"
        }
      }
    },
    "Exports": {
      "type": "array",
      "items": {"type": "string"}
    },
    "Config": {
      "type": "object",
      "additionalProperties": false,
      "properties": {
        "$MaxBinary": {"type": "integer", "minValue": 1},
        "$MaxString": {"type": "integer", "minValue": 1},
        "$MaxElements": {"type": "integer", "minValue": 1},
        "$Sys": {"type": "string", "minLength": 1, "maxLength": 1},
        "$TypeName": {"type": "string", "minLength": 1, "maxLength": 127},
        "$FieldName": {"type": "string", "minLength": 1, "maxLength": 127},
        "$NSID": {"type": "string", "minLength": 1, "maxLength": 127}
      }
    },
    "Fields": {
      "type": "array",
      "items": [
        {"anyOf": [
          {"$ref": "#/definitions/Item"},
          {"$ref": "#/definitions/Field"}
        ]}
      ]
    },
    "Item": {
      "type": "array",
      "minItems": 2,
      "maxItems": 3,
      "items": [
        {"type": "integer"},
        {"type": "string"},
        {"$ref": "#/definitions/Description"}
      ]
```



```
    },
    "Field": {
      "type": "array",
      "minItems": 3,
      "maxItems": 5,
      "items": [
        {"type": "integer"},
        {"$ref": "#/definitions/FieldName"},
        {"$ref": "#/definitions/TypeRef"},
        {"$ref": "#/definitions/Options"},
        {"$ref": "#/definitions/Description"}
      ]
    },
    "NSID": {
      "type": "string"
    },
    "TypeName": {
      "type": "string"
    },
    "TypeRef": {
      "type": "string"
    },
    "FieldName": {
      "type": "string"
    },
    "BaseType": {
      "type": "string",
      "enum": ["Binary", "Boolean", "Integer", "Number", "Null", "String",
               "Enumerated", "Choice",
               "Array", "ArrayOf", "Map", "MapOf", "Record"]
    },
    "Options": {
      "type": "array",
      "items": {"type": "string"}
    },
    "Description": {
      "type": "string"
    }
  }
}
```


Appendix F. ABNF Grammar for JADN IDL

Case-sensitive ABNF grammar for JADN Interface Definition Language (Section 5.1).

TBSL

