JSON Abstract Data Notation (JADN) Version 2.0

Committee Specification Draft 01

19 February 2025

This version:

https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/jadn-v2.0-csd01.md (Authoritative)
https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/jadn-v2.0-csd01.html
https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/jadn-v2.0-csd01.pdf

Previous 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

Latest version:

https://docs.oasis-open.org/openc2/jadn/v2.0/jadn-v2.0.md (Authoritative)
https://docs.oasis-open.org/openc2/jadn/v2.0/jadn-v2.0.html
https://docs.oasis-open.org/openc2/jadn/v2.0/jadn-v2.0.pdf

Technical Committee:

OASIS Open Command and Control (OpenC2) TC

Chair:

Duncan Sparrell (duncan@sfractal.com), sFractal Consulting LLC
Michael Rosa (mjrosa@nsa.gov), National Security Agency

Editor:

David Kemp (d.kemp@cyber.nsa.gov), National Security Agency

Additional artifacts:

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

- JADN metaschema for JADN documents:
  - https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/artifacts/jadn-v2.0-schema.jadn
  - https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/artifacts/jadn-v2.0-schema.jidl
- JSON schema for JADN documents:
  - https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/artifacts/jadn-v2.0-schema.json
- JADN schema for Examples:
  - https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/artifacts/v2.0-examples.jadn
  - https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/artifacts/v2.0-examples.jidl

Abstract:

An Information Model (IM) defines the meaning and essential content of data used in computing independently of how it is represented for processing, communication or storage. JSON Abstract Data Notation (JADN) is an information modeling language based on Unified Modeling Language (UML) logical DataTypes, used to both express the meaning of data items at a conceptual level and formally define and validate instances of those types. JADN uses information theory to define logical equivalence, which

enables representation of essential content in a wide range of formats and ensures translation among representations without loss. This document defines the normative DataTypes and data formats used to construct a JADN IM, and describes several equivalent non-normative model representations including a textual information definition language, a table format, and a diagram format. Because a JADN IM is a logical value, it can also be serialized in the same formats as the data it describes, allowing the model to accompany the data if desired and facilitating dynamic model updates.

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 stage" 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.

Key words:

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "NOT RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in BCP 14 [RFC 2119] and [RFC 8174] when, and only when, they appear in all capitals, as shown here.

Citation format:

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

[JADN-v2.0]
JSON Abstract Data Notation Version 2.0. Edited by David Kemp. 19 February 2025. OASIS Committee Specification Draft 01. https://docs.oasis-open.org/openc2/jadn/v2.0/csd01/jadn-v2.0-csd01.html. Latest version: https://docs.oasis-open.org/openc2/jadn/v2.0/jadn-v2.0.html.

Notices

Table of Contents

1 Introduction

> *An information model is a representation of concepts, relationships, constraints, rules, and operations to specify data semantics for a chosen domain of discourse. An information modeling language is a formal syntax that allows users to capture data semantics and constraints.*

-- [Information Modeling], Y. Tina Lee, NIST

This is the reference specification for the JADN information modeling language. See [JADN-CN] for additional detail on the information modeling process and how to construct and use JADN information models. While the term information modeling is used broadly and covers a range of applications, a JADN information model defines the essential content of discrete data items used in computing independently of how that content is represented for processing, communication or storage.

- **Essential content** (information, meaning) is defined by information theory, where the amount of information conveyed in a message is not directly related to the size or format of the message.
- **Data items** (documents, messages, protocol data units, data structures, object state, etc.) are JADN's scope within a system's domain of discourse.

An information model defines the question "What does the recipient know after receiving a data item" separately from "what does a data item look like".

> *The objective of UML is to provide system architects, software engineers, and software developers with tools for analysis, design, and implementation of software-based systems as well as for modeling business and similar processes.*

-- [Unified Modeling Language (UML)]

JADN is a UML profile for documents and messages. UML's organizing principle is classification, where a classifier represents a classification of instances according to their features. The values that are classified by a classifier are called instances of the classifier. UML defines several kinds of classifier including DataType and Class. Instances of a DataType are identified by their value, and all instances of a DataType with the same value are considered to be equal instances. DataType instances are immutable because by definition a different value is a different instance. A value may be classified as an instance of multiple DataTypes, but value comparison is meaningful only among instances of the same type.

Instances of a Class are objects that model operations and behavior. An object does not have an immutable value: its state can change over time and two objects instantiated from the same Class, even with identical property values, are different instances. Although objects are not values, DataTypes model object features that are values, such as documents and messages in business processes and public fields and API (getter/setter) views of private state in software-based systems. Additional differences between DataType and Class include:

- Collection DataTypes specify if value order is significant. Class public fields and API values do not have an order.
- DataType distinguishes between values and references, Class does not. For example, software functions cannot persistently modify arguments passed by value but can modify those passed by reference. Validating a document for correctness or integrity validates the values it contains but not the values it references. A document DataType can distinguish between local and external references and validate that local references identify values contained within that instance.
- Misusing Class to model data is a common practice, but often results in contradictions such as modeling a one-dimensional Coordinate (e.g., latitude) as a DataType but a two-dimensional Coordinate (latitude, longitude) as a Class.

The **Resource Description Framework** [RDF] includes DataTypes:

> *RDF defines an abstract syntax (a data model) which serves to link all RDF-based languages and specifications. RDF graphs are sets of subject-predicate-object triples, where the elements may be IRIs, blank nodes, or **datatyped literals**. They are used to express descriptions of resources.*

RDF defines DataType as having a "lexical-to-value (L2V) mapping", and while an RDF graph defines relationships among physical and data resources, DataType is the only RDF element that defines a data resource in terms of both a literal representation and its representation-independent logical value.

Defining equivalence across representations is the primary distinction between information modeling and other data modeling approaches. An information model is constructed from DataTypes, not Classes, because its purpose is to compare literal values for equivalence based on their logical information content, and only DataTypes have instances that can be validated for content integrity and compared for equality.

1.1 Glossary

1.1.1 Definitions of terms

- **Information (essential content)**: Informally, essential means that if data can be removed from a message without affecting its meaning, then it is not essential. Formally, information theory quantifies the entropy (novelty, or news value) of a message in bits, excluding data that is insignificant or is redundant with what is known *a priori*. The information content of a message can be no greater than the smallest data value that accurately represents it.
- **Information Model**: An abstract schema that defines the meaning, structure and value constraints of information used in computing systems independently of representation, plus a set of application-independent mappings between external data values and internal logical values.
- **Equivalence**: The relation between the meaning represented by two data values such that each logically implies the other. Two data values are equivalent if and only if they are classified as instances of the same DataType and have the same logical value.
- **DataType (logical type, type)**: An abstract type that defines the meaning and essential content of a discrete data item used in computing independently of how it is represented for processing, communication or storage. DataTypes are defined by and composed using an information modeling language. Every DataType has a value space as defined in XSD Part 2 Section 2.1 and a lexical space defined by a specified data format.
- **Logical Value (information value)**: An immutable instance of a DataType used for processing and comparison, specified by behavioral effect independently of programming languages and techniques. Every logical value is a member of the value space of its DataType.
- **Data Value (document, message, artifact, lexical value, literal value)**: An immutable instance of a DataType used for transmission or storage, consisting of a sequence of octets or characters in an external data format. Every lexical value is a member of a lexical space of its DataType.
- **Data Format**: Serialization rules that specify the media type (e.g., XML, JSON, CBOR, Protobuf), design goals (human readability, efficiency), and style preferences for data values in that format. A data format defines a lexical space and a lexical mapping for each DataType.
- **Data Model**: A concrete schema that defines the structure and value constraints of serialized data. A single information model corresponds to multiple equivalent data models; data models are equivalent if they define data values representing the same information.
- **Presentation Format**: A view of logical values that does not necessarily preserve all essential content, used for display or documentation purposes.
- **Well-formed**: A data value that is valid according to a structured syntax [RFC 7303] (e.g., "+json", "+der"), if one is specified by the data format.
- **Valid**: A logical value is valid if it satisfies the constraints of its DataType. A data value is valid if it is well-formed and is classified as an instance of a DataType.
- **Serialization**: Serialization, or encoding, converts a logical value into a data value. De-serialization, or decoding, classifies a data value and converts it into an instance of a DataType.
- **Description (annotation)**: Description fields of an information model are reserved for comments from authors to readers or maintainers of the model and are ignored by information processing applications.

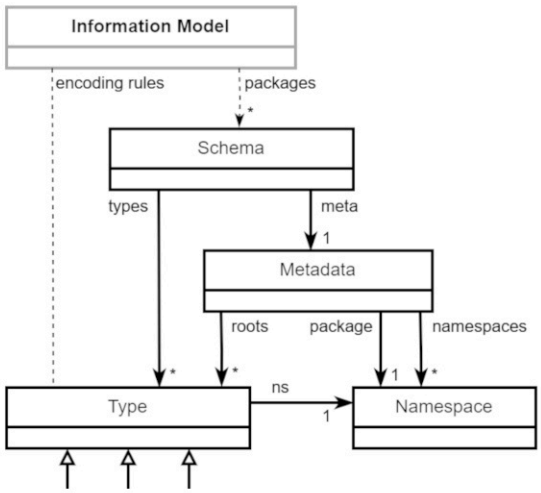
1.1.2 Acronyms and abbreviations

- **DAG**: Directed Acyclic Graph
- **DM**: Data Model
- **IM**: Information Model

2 Information Models

A JADN information model defines the essential content of discrete data items used in computing independently of how that content is represented for processing, communication or storage. Information values are instances of abstract UML DataTypes, and as shown in Figure 2-1 DataType definitions are organized into abstract schema packages which are included in an application's information model.

Figure 2-1 -- Information Model Organization



- A JADN IM consists of a set of abstract schemas that define information content, and a set of encoding rules that define the lexical-to-value mapping in a specific data format for each JADN core type.
- Schema is the top level JADN type. It has two fields:
  - "Metadata" containing descriptive and functional information about the schema package as a whole.
  - List of "Type" containing JADN type definitions. Every type definition is a UML DataType.
- An instance of the Schema type is identified by a globally-unique package namespace. Types defined in a package have names qualified by its namespace, and reference types defined in other packages by their qualified names. An individual Schema instance is called a "package" because it is an instance, not a Type, and to distinguish it from an "application schema" that is the set of packages in an information model.
- There is no "information model" type containing or assigning a name to a set of schema packages. Applications load relevant package(s) plus any additional packages needed to resolve type references.

Section 3 defines schema packages and metadata.
Section 4 defines the JADN core types.
Section 5 defines shortcuts that make type definitions more convenient without affecting meaning.
Section 6 discusses using encoding rules to define concrete data formats.
Section 7 describes non-normative alternate JADN schema formats:

- a text-based information definition language (IDL) defined and validated by a language grammar
- property tables used in protocol or document format specifications
- entity-relationship diagrams (ERDs) used for data modeling

The normative format of a Schema package, as defined in Sections 3 and 4, is JSON data that can be validated by a schema, but a package can also be represented unambiguously in other formats more suited to human understanding. This specification uses JSON to precisely define the structure of a JADN schema, but uses the IDL format described in Section 7 where understanding purpose and meaning is the primary goal. These representations are equivalent, but if there is a conflict the JSON definition has precedence.

3 Schema Packages

Packages provide the main structuring and organizing capability of UML. A UML package is a namespace for its members, and a JADN abstract schema is composed using packages. All packages, including the one defining JADN itself, are instances of JADN's `Schema` type. Schema has two fields: package metadata defined in Fig. 3-1, and a list of type definitions defined in Section 4.

Fig. 3-1. JADN Schema: Metadata

```
        title: "JADN Metaschema"
      package: "http://oasis-open.org/openc2/jadn/v2.0/schema"
  description: "Syntax of a JSON Abstract Data Notation (JADN) package."
      license: "CC-BY-4.0"
        roots: ["Schema"]
       config: {"$FieldName": "^[$A-Za-z][_A-Za-z0-9]{0,63}$"}

Schema = Record                             // Definition of a JADN package
   1 meta           Metadata optional       // Information about this package
   2 types          Type unique [1..*]      // Types defined in this package

Metadata = 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 of this package
   8 namespaces     PrefixNs unique [0..*]  // Referenced packages
   9 roots          TypeName unique [0..*]  // Roots of the type tree(s) in this package
  10 config         Config optional         // Configuration variables
  11 jadn_version   Namespace optional      // JADN Metaschema package

PrefixNs = Array                            // Prefix corresponding to a namespace IRI
   1  NSID                                  // prefix:: Namespace prefix string
   2  Namespace                             // namespace:: Namespace IRI

Config = Map{1..*}                          // Config vars override JADN defaults
   1 $MaxBinary     Integer{1..*} optional  // Package max octets, default = 255
   2 $MaxString     Integer{1..*} optional  // Package max characters, default = 255
   3 $MaxElements   Integer{1..*} optional  // Package max items/properties, default = 255
   4 $Sys           String{1..1} optional   // System character for TypeName, default = '.'
   5 $TypeName      String /regex optional  // Default = ^[A-Z][-.A-Za-z0-9]{0,63}$
   6 $FieldName     String /regex optional  // Default = ^[a-z][_A-Za-z0-9]{0,63}$
   7 $NSID          String /regex optional  // Default = ^([A-Za-z][A-Za-z0-9]{0,7})?$

Namespace = String /uri                     // Unique name of a package
NSID = String{pattern="$NSID"}              // Namespace prefix matching $NSID
TypeName = String{pattern="$TypeName"}      // Name of a logical type
FieldName = String{pattern="$FieldName"}    // Name of a field in a structured type
TypeRef = String                            // Reference to a type, matching ($NSID ':')? $TypeName
```

3.1.1 Descriptive Metadata

These Metadata fields provide information about a package but have no effect on schema processing:

- **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:** SPDX licenseId of the contents of this package.

3.1.2 Functional Metadata

These Metadata fields affect schema processing:

- **package:** A namespace [IRI] that unambiguously identifies this Schema instance and allows type definitions in this package to be unambiguously referenced from other packages. This is a unique identifier but not necessarily a resource locator.

- **version:** Incremental revision of this package, a string that compares lexicographically higher than previous revisions. A package namespace uniquely identifies both the topic and published version of a referenced package. This field identifies the latest revision of a package when more than one revision is available.
- **jadn_version:** Package namespace of the JADN version used to validate this package.
- **namespaces:** A set of associations between Namespace IDs (prefixes) and namespace IRIs. Types defined in this package may reference types from other packages using `PrefixedName` as defined in [XML Namespaces]. Associating a blank prefix with a package namespace indicates that its types are treated as if they were defined in this package. This requires the referenced package to have non-conflicting type names and compatible metadata including name formats and namespaces.
- **roots:** List of top-level types defined in this package. This designates a single starting point or a catalog of library types defined in this package, and allows schema processing tools to flag unreferenced type definitions.
- **config:** Configuration variables used to tailor schema processing within a package. Variables not configured in a package have an implementation-defined default value, with recommended defaults shown below.
    - **Name Formats:** JADN syntax does not restrict the allowed name formats, but establishing naming conventions using distinct formats for TypeName and FieldName (Section 4.1) can aid schema readability. These variables define a package's naming conventions:
        - **$Sys:** A "system" character used in software-generated TypeNames. Default = `"."`
        - **$TypeName:** The regex used to validate TypeName. Default begins with an upper-case character: `^[A-Z][-.A-Za-z0-9]{0,63}$`
        - **$FieldName:** The regex used to validate FieldName. Default begins with a lower-case character: `^[a-z][_A-Za-z0-9]{0,63}$`
          The JADN Metaschema overrides the default $FieldName pattern to allow config variables beginning with '$' and core type names beginning with a capital letter.
        - **$NSID:** The regex used to validate an external type reference's prefix string. Default: `^([A-Za-z][A-Za-z0-9]{0,7})?$`
          External type references (TypeRef in Figure 4-2) are prefixed names that include an NSID.
    - **Size Limits:** These variables define default maximum sizes for variable-sized Primitive and Compound types (Section 4). Individual type definitions override implementation or package defaults using type options.
        - **$MaxBinary:** Maximum number of octets in a Binary instance. Default `maxLength` = 255
        - **$MaxString:** Maximum number of characters in a String instance. Default `maxLength` = 255
        - **$MaxElements:** Maximum number of items in an ArrayOf or MapOf instance. Default `maxOccurs` = 255

3.1.3 Package Conformance Requirements

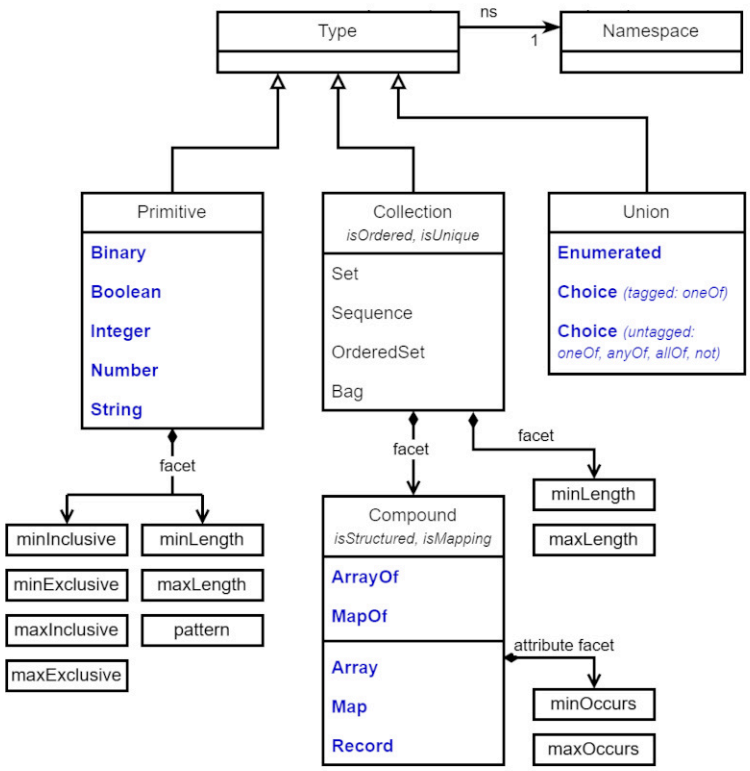
- The $TypeName format MUST permit TypeNames containing the $Sys character, which is used in type names generated by schema processing and translation.
- The $FieldName format MUST NOT permit FieldNames containing the $Sys character, to enable its use as a path component separator.
- A TypeRef instance MUST be a qualified name (`QName`) as defined in [XML Namespaces] using $NSID and $TypeName instances as `Prefix` and `LocalPart` respectively.
- If Metadata is present in a Schema instance it MUST include the package field; all other fields are optional.

4 JADN Types

An information modeling language's abstract DataTypes define their meaning and application behavior. As shown in Figure 4-1, JADN defines twelve core types in three categories:

- **Primitive** (Section 4.2.1): Types whose instances are atomic (non-decomposable) values.
- **Compound** (Section 4.2.2: Types whose instances are collections of values.
- **Union** (Section 4.2.3): Types whose instances are selected from a set of possible values.

Fig. 4-1. JADN Core DataTypes



4.1 Type Definition Structure

All JADN type definitions have the identical structure, shown in Figure 4-2, designed to be easily describable, easily processed, stable, and extensible.

Fig. 4-2. JADN Schema: Types

```
Type = Array
   1  TypeName                                  // type_name::
   2  Enumerated(Enum[JADN-Type])               // core_type::
   3  Options optional                          // type_options::
   4  Description optional                      // type_description::
   5  JADN-Type(TagId[core_type]) optional      // fields::

JADN-Type = Choice
   1 Binary           Empty
   2 Boolean          Empty
   3 Integer          Empty
   4 Number           Empty
   5 String           Empty
   6 Enumerated       Items
   7 Choice           Fields
   8 Array            Fields
   9 ArrayOf          Empty
  10 Map              Fields
  11 MapOf            Empty
  12 Record           Fields

Empty = Array{0}
```

```
Items = ArrayOf(Item)
Fields = ArrayOf(Field)

Item = Array
   1  FieldID                                  // item_id::
   2  String                                   // item_value::
   3  Description optional                     // item_description::

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

FieldID = Integer
Options = ArrayOf(Option) unique
Option = String{1..*}
Description = String
```

Each type definition has five elements:

1. **TypeName:** the name of the type being defined
2. **CoreType:** the JADN built-in type of the type being defined
3. **TypeOptions:** an array of zero or more **TypeOption** values applicable to **CoreType**
4. **TypeDescription:** a non-normative comment
5. **Fields:** an array of **Item** or **Field** definitions

Defaults:

- If TypeOptions is not present in a type definition, its default is the empty array.
- If TypeDescription is not present, its default is the empty string.
- If Fields is not present, its default is the empty array.

4.1.1 Primitive

If CoreType is a Primitive or unstructured Compound type, the **Fields** array is empty.

JSON Format and Example:

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

["Username", "String", ["%^[a-z][a-z0-9]{3,11}$"]]
["Users", "ArrayOf", ["*Username"]]
```

IDL Example:

```
Username = String{pattern="^[a-z][a-z0-9]{3,11}$"}
Users = ArrayOf(Username)
```

4.1.2 Enumerated

If CoreType is the Enumerated Type, each item definition in the **Fields** array has three elements:

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

JSON Format and Example:

```
[TypeName, CoreType, [TypeOption, ...], TypeDescription, [
    [ItemId, ItemValue, ItemDescription],
    ...
]]

["Color", "Enumerated", [], "", [
    [1, "red"],
    [2, "green"],
    [3, "blue"]
]]
```

IDL Example:

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

4.1.3 Compound

If CoreType is a structured Compound or Choice type, each field definition in the **Fields** array has 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, a **TypeReference**
4. **FieldOptions:** an array of zero or more **FieldOption** or **TypeOption** values applicable to **FieldType**
5. **FieldDescription:** a non-normative comment

JSON Format and Example:

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

["Coordinate", "Record", [], "A GPS coordinate", [
    [1, "latitude", "Latitude", [], "A Number between -90 and 90 degrees"],
    [2, "longitude", "Longitude", [], "A Number between -180 and 180 degrees"]
]]
```

IDL Example:

```
Coordinate = Record                     // A GPS coordinate
   1 latitude       Latitude            // A Number between -90 and 90 degrees
   2 longitude      Longitude           // A Number between -180 and 180 degrees
```

4.1.4 Type and Field Options

Type and field options are the mechanisms to support a varied set of information needs within the strictly regular type definition structure. New requirements can be accommodated by defining new options without modifying that structure. Each TypeOption and FieldOption provides a limited piece of information about some aspect of the DataType to which it applies, similar in purpose to an [XSD] *facet*. Each option has an ID and value listed in Section 4.2, and is represented in JSON format as a string where the first character's Unicode codepoint is the option's ID and the remaining characters are its value. Boolean options have no additional characters; if the option ID is present its value is True, otherwise False.

As an example the TypeOption "minLength = 1" is represented as:

```
+----+-----------+     Option ID = 0x7b (Left Curley Bracket) = "minLength"
| ID | Value     |     Value = 1 (Integer)
+----+-----------+     TypeOption = "{1" (String)
```

In some cases the character represented by an option ID has a mnemonic relationship to its purpose but this is not true in general; option IDs are non-semantic integer identifiers.

4.1.5 Type Conformance Requirements

- TypeName MUST NOT be a JADN core type
- CoreType MUST be a JADN core type
- FieldID and FieldName values MUST be unique within a type definition.
- If CoreType is Array or Record, FieldID MUST be the ordinal position of the field within the type, numbered consecutively starting at 1.
- If CoreType is Enumerated, Choice, or Map, ItemID/FieldID MAY be any integer.
- FieldType MUST be a Primitive type, ArrayOf, MapOf, or a model-defined (non-core) type.
- If FieldType is not a core type, FieldOptions MUST NOT contain any TypeOption.
- If FieldOptions includes a TypeOption, that option MUST apply to FieldType.

- If the Derived Enumerations or Pointers shortcuts are present in TypeOptions, the Fields array MUST be empty.
- The default value of TypeOptions, Fields and FieldOptions is the empty Array.
- The default value of TypeDescription, ItemDescription and FieldDescription is the empty String.
- Description values MUST have no effect on validation or serialization.

4.2 Core Types

4.2.1 Primitive Types

A primitive type has no substructure, and specifies an unrestricted space of atomic values without regard to processing mechanisms or data format. As shown in Figure 4-1 the primitive core types are Binary, Boolean, Integer, Number and String.

Type options specify value restrictions such as size, range, and regular expression patterns. Semantic validation keywords (formats) listed in Section 4.2.5 also define value restrictions on primitive types.

Primitive TypeOptions are listed in Table 4-1:

Table 4-1: TypeOptions Specific to Primitive Types

ID	Chr	Type	Name	Description
0x25	%	String	pattern	Instance matches the specified regular expression
0x7b	{	Integer	minLength	Minimum octet or character count
0x7d	}	Integer	maxLength	Maximum octet or character count
0x75	u	*	default	Instance equals default if no value is given
0x76	v	*	const	Instance is equal to option value
0x77	w	*	minInclusive	Instance is greater than or equal to option value
0x78	x	*	maxInclusive	Instance is less than or equal to option value
0x79	y	*	minExclusive	Instance is greater than option value
0x7a	z	*	maxExclusive	Instance is less than option value

* indicates that the option value must evaluate to an instance of CoreType.

- The `default` option specifies a pre-set value to be used for an optional/nullable variable when no other value is supplied.
  - When parsing a literal value of `null` or when no literal value is present, the logical value is set to the default.
  - When classifying a logical value of `null` or when no logical value is present, the classifier uses the default.
  - When serializing a logical value equal to the default, the literal value is either omitted or `null` as specified by the data format.
- The `const` option specifies a pre-set value used as a classifier, equivalent to setting both `minInclusive` and `maxInclusive` to that value.

*Note: This specification does not define an expression language but does not preclude their use. For options with type = * the result of using a value other than a single terminal element (literal instance of a Primitive type) is not defined here. In principle the `default` and `const` options apply to Compound types but cannot be used until a Compound literal format is defined.*

4.2.1.1 Boolean

A Boolean instance is one of the predefined values *true* and *false*.

Options: const, default

4.2.1.2 Integer

An Integer instance is a value in the ordered infinite set of integers (…, -2, -1, 0, 1, 2, …).

Options: const, default
Range Options: minInclusive, maxInclusive, minExclusive, maxExclusive

4.2.1.3 Number

A Number instance is a value in the ordered infinite set of real numbers.

Options: const, default
Range Options: minInclusive, maxInclusive, minExclusive, maxExclusive

4.2.1.4 String

A String instance is a sequence of characters in a character set. Value range options are meaningful if the character set defines a collation order. The pattern, length, and range options are not normally used together, but if more than one kind is present in a type definition an instance must satisfy all conditions.

Options: pattern, const, default
Length Options: minLength, maxLength
Range Options: minInclusive, maxInclusive, minExclusive, maxExclusive

4.2.1.5 Binary

A Binary instance is sequence of octets. Binary values are not ordered so range options do not apply.

Options: const, default
Length Options: minLength, maxLength

4.2.1.6 Primitive Type Conformance Requirements

- A value MUST satisfy the conditions defined for each type option listed in Table 4-1 to be classified as an instance of a type containing that option.
- The *pattern* option value SHOULD conform to the Pattern grammar of [ECMAScript] Section 22.2.

4.2.2 Compound Types

Compound types define a collection of items. As shown in Figure 4-1 a compound type defines how the items in a collection are specified, while the collection itself is a UML "MultiplicityElement" with cardinality bounds and collection properties. The Compound types are listed in Table 4-2:

Table 4-2: Compound Types

Compound Type	Structured	Mapping	Collection Properties
ArrayOf(vtype)	No	No	Ordered, non-Unique (sequence)
Array	Yes	No	Ordered, non-Unique (sequence)
MapOf(ktype, vtype)	No	Yes	non-Ordered, Unique (set)
Map	Yes	Yes	non-Ordered, Unique (set)
Record	Yes	Both	non-Ordered, Unique (set)

By default, ArrayOf and Array specify a sequence of items and MapOf, Map, and Record specify a set of items, but these collection properties can be modified using TypeOptions.

- The `vtype` option specifies the type of each instance in an ArrayOf or MapOf type.
- The `ktype` option specifies the type of each key in a MapOf type.
- If a collection is Ordered, item order is significant when comparing instances, otherwise it is not.
- If a collection is Unique, no item is duplicated within a collection instance, otherwise duplicates are allowed.
- A Structured type includes individual field definitions. Each field defines an association between an identifier (position and/or key) and a type and may include field-specific options (Section 4.2.2.1). A non-structured compound type defines a collection where each item is an instance of the same type.
- Each item in a Mapping type is a key:value pair with a unique key, otherwise each item is a value.
- The Record type defines the key order, which allows Record instances to be represented as either arrays where items are identified by position within the array, or associative arrays (maps) where items are identified by key.

Compound TypeOptions are listed in Table 4-3:

Table 4-3: TypeOptions Specific to Compound Types

ID	Chr	Type	Name	Description
0x2a	*	TypeRef	vtype	Value type for ArrayOf and MapOf
0x2b	+	TypeRef	ktype	Key type for MapOf
0x7b	{	Integer	minLength	Minimum number of items in a collection, default is 0
0x7d	}	Integer	maxLength	Maximum number of items in a collection, default is unlimited
0x3d	=	Boolean	id	Fields are identified by FieldID not FieldName
0x71	q	Boolean	unique/ordered	isOrdered = true, isUnique = true (ordered set)
0x73	s	Boolean	set	isOrdered = false, isUnique = true (set)
0x62	b	Boolean	unordered	isOrdered = false, isUnique = false (bag)

- Map and Record types have Fields identified by both a numeric FieldID and a text FieldName, both of which are unique within a type.
- FieldIDs for Array and Record types denote position within the collection and must be numbered consecutively starting at 1.
- For Map, Enumerated and Choice types the `id` option indicates that fields are always identified by FieldID.
  - Without `id`, 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.
  - With `id`, FieldName is a suggested label that is not included in the semantics of the type, may be empty in the type definition, has no effect on validation, and never appears in serialized data regardless of data format.
  - The `id` option cannot be used with Record; the Array type is equivalent to Record with id.
- TypeOption `0x71` (collection is an ordered set) is referred to as `unique` when used with the ArrayOf type and `ordered` when used with MapOf, Map or Record types.

Example: the `id` option indicates that values use FieldId instead of FieldName

```
["Colors", "Enumerated", [], "", [
  [1, "red", "The color of roses"],
  [2, "green"],
  [3, "blue", "Violets"]
]],

["ColorIds", "Enumerated", ["="], "", [
  [1, "red", "The color of roses"],
  [2, "green"],
  [3, "blue", "Violets"]
]]
```

```
Colors = Enumerated
   1 red                        // The color of roses
   2 green
   3 blue                       // Violets

ColorIds = Enumerated#
   1                            // red:: The color of roses
   2                            // green::
   3                            // blue:: Violets
```

Multiplicity TypeOptions specify the ordering and uniqueness semantics of compound types. This allows collection instances with uniqueness constraints to be validated and instances with the same ordering significance to be compared, independently of their compound type. The ArrayOf compound type can specify the four UML collection types (sequence, set, ordered set, bag). Structured and MapOf compound types are always unique, so they can specify only set or ordered set collections. The collection type specified by a Compound type and multiplicity option are listed in Table 4-4:

Table 4-4: Collection Types

Compound Type	Multiplicity Option	Collection Properties
ArrayOf		Ordered, non-Unique (sequence)
Array		Ordered, non-Unique (sequence)
MapOf		Non-Ordered, Unique (set)
Map		Non-Ordered, Unique (set)
Record		Non-Ordered, Unique (set)
---	---	---
ArrayOf	set	Non-Ordered, Unique (set)
ArrayOf	unique	Ordered, Unique (ordered set)
ArrayOf	unordered	Non-Ordered, Non-Unique (bag)
Array	set	Non-Ordered, Unique (set)
MapOf	ordered	Ordered, Unique (ordered set)
Map	ordered	Ordered, Unique (ordered set)
Record	ordered	Ordered, Unique (ordered set)

The TypeOptions applicable to each compound type are listed in Table 4-5:

Table 4-5: Applicable Compound Type Options

Compound Type	Allowed TypeOptions
ArrayOf(vtype)	minLength, maxLength, set, unique, unordered, vtype
Array	minLength, maxLength, set
MapOf(ktype, vtype)	minLength, maxLength, ordered, ktype, vtype
Map	minLength, maxLength, ordered, id
Record	minLength, maxLength, ordered

4.2.2.1 Field Options

Structured compound types (Array, Map and Record) and the Choice type have Fields that define each item in a collection individually. Each Field has a numeric ID, Name, TypeReference, and FieldOptions shown in Table 4-6:

Table 4-6: Field Options

ID	Chr	Type	Name	Description
0x5b	[	Integer	minOccurs	min cardinality, default = 1, 0 = field is optional
0x5d	]	Integer	maxOccurs	max cardinality, default = 1, <0 = inherited or none
0x4b	K	Boolean	key	field is the primary key for this type
0x4c	L	Boolean	link	field is a link (foreign key) to an instance of FieldType

4.2.2.2 Multiplicity

The **minOccurs** and **maxOccurs** options specify the minimum and maximum number of instances (the multiplicity) of a field within a collection:

minOccurs	maxOccurs	Multiplicity	Description
1	1	1	One instance (required) - default
0	1	0..1	Zero or one instances (optional)
0	0	0	Zero instances (prohibited)
0	< 0	0..*	Zero or more instances
1	< 0	1..*	One or more instance
m	n	m..n	At least m but no more than n instances

- The default value of minOccurs and maxOccurs is 1.
- maxOccurs includes non-negative integers (0..n), plus two reserved sentinel values less than 0:
  - UNSPECIFIED (-1) indicates that the upper bound is the $MaxElements package default (Figure 3-1), or if not specified, an implementation-defined default.
  - UNLIMITED (-2) indicates that no upper bound is defined. Implementations are still limited by available storage capacity and the results of resource exhaustion are undefined.
- If a field has more than one instance, the data format specifies whether its representation differs from that of a single instance. The Field Multiplicity Shortcut generates an ArrayOf() type definition for data formats (e.g., JSON) with different representations for single and multiple instances of a type.

4.2.2.3 Links

An information model defines type relationships in two ways: as collections containing values or as references to values. Collection relationships are normally hierarchical: a root compound type such as book contains chapters, which contain sentences, which contain leaf types such as words. A hierarchy is a directed acyclic graph (DAG), meaning that its types have no circular dependencies and its values have no indefinitely-deep recursive nesting. When collection types have cyclic relationships either directly or indirectly through other types, the cycles should be broken by replacing a contained value with a reference to eliminate recursive nesting.

The `key` and `link` TypeOptions support type references:

- The `key` option designates one field of a structured compound type as its primary key. Although the key field is normally a primitive type, it may be defined as a compound type to support composite keys.
- The `link` option designates a field as a foreign key that references an instance of the specified type, flattening collection values and supporting relationship-aware application operations such as checking referential integrity.

As an example, an instance of a Person type with cyclic relationships would contain denormalized (duplicated) nested values. Using `link` references to eliminate contained value cycles results in a flat set of independent, normalized values:

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

Organization = Record
    1 name      String
    2 ein       Key(String{10..10})
    3 ceo       Link(Person)
```

Example composite key:

```
LineItem = Record
   1 item_id    Key(ItemId)     // Composite unique identifier for a line item
   2 quantity   Integer         // Other information about the ordered item

ItemId = Array
   1 Integer                   // order_id:: Order unique identifier
   2 Integer                   // product_id:: Product unique identifier
```

4.2.2.4 Compound Type Conformance Requirements

- A compound type MUST NOT include more than one multiplicity option (set, unique, ordered, or unordered).
- If CoreType is ArrayOf, TypeOptions MUST include the `vtype` option.
- If CoreType is MapOf, TypeOptions MUST include `ktype` and `vtype` options.
- The `ktype` option SHOULD be a constrained type such as an enumeration, pattern or semantic valuation keyword that specifies a fixed subset of values.
- All values in an ArrayOf or MapOf instance must be an instance of `vtype`.
- All keys in a MapOf instance MUST be an instance of `ktype`.
- The number of items in a collection instance MUST NOT be less than minLength.
- The number of items in a collection instance MUST NOT be greater than maxLength.
- FieldIDs for Array and Record types denote position within the collection and MUST be numbered consecutively starting at 1.
- 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.
- An Array, Map or Record type MUST have no more than one `key` field. The key field MAY be a compound type.
- Values referenced by the `link` option MUST be instances of the referenced type.
- The value of a field with the `link` option MUST equal the value of the `key` field of the referenced type.

4.2.3 Union Types

A union type specifies a set of alternatives used to classify a value. Like Compound types, some Union types have fields individually identified by tag, where the tag consists of an integer FieldID and a string FieldName, each of which is local to and unique within the type definition. Union types define a set of tags, types or both as shown in Table 4-7:

Table 4-7: Union Types

Type	Tag	Type	Definition
Enumerated	Yes	-	Vocabulary, a set of tags.
Choice	Yes	Yes	Tagged union, a set of tag:type pairs.
Choice(Cx)	-	Yes	Untagged union, a specified logical combination of types.

The TypeOptions applicable to Union types are shown in Table 4-8:

Table 4-8: Union Type Options

ID	Chr	Type	Name	Description
0x3d	=	Boolean	id	If present Tag is an integer FieldID, otherwise a string FieldName
0x43	C	String	combine	Option value is a character specifying the untagged union combining function
0x23	#	TypeRef	enum	Enumerated type derived from a structured type
0x3e	>	TypeRef	pointer	Enumerated type containing pointers derived from a structured type

4.2.3.1 Enumerated

An Enumerated type defines a vocabulary, an explicitly listed set of `item_id`:`item_value` pairs. Enumerated is described as "a degenerate tagged union of unit type" [ENUM] because it defines the tags of a tagged union without any associated type, and an instance equals one of the defined tags. The `id` option specifies that an instance is an integer matching an `item_id`, otherwise it is a string matching the corresponding `item_value`.

The `enum` (Section 5.3) and `pointer` (Section 5.5) options are shortcuts that expand to an Enumerated type containing the tags from a referenced structured type.

4.2.3.2 Choice (Tagged)

The Choice type without a `combine` TypeOption is a tagged union, a structure that defines a set of tag:type pairs. Values include a tag specifying a single FieldType from the set, and an instance is a value that matches the FieldType specified by the tag.

4.2.3.3 Choice (Untagged)

The Choice type containing a `combine` TypeOption is an untagged union, a structure that defines a set of types used collectively to classify a value.

The `combine` option value is a single character that specifies the required combination of FieldTypes:

- A: value must be an instance of `allOf` the types
- O: value must be an instance of `anyOf` the types, tried in field order until a match is found
- X: value must be an instance of `oneOf` the types and no others

Field order does not matter for the `allOf` and `oneOf` options because values must always be evaluated against all FieldTypes.

Field order is significant when using the `anyOf` option and the FieldTypes are not disjoint because this performs both classification and validation. A value may be an instance of more than one classifier, and classification may be used to answer two questions:

- given a classifier A, is value X an instance of A? (validation)
- given a value X, which classifier among {A, B, C, ...} is it to be considered an instance of? (classification)

In this example the value "Home" is an instance of both the predefined and custom types and could be classified as either one. If any processing operations depend on the classification decision, the predefined type must appear first in the Choice otherwise it will never match and all values will be tagged, serialized, and processed as instances of the custom type:

```
PhoneType = Choice(anyOf)
  1 predefined  PhoneNumberTypes   // Pre-defined names
  2 custom      String{3..10}      // Any name 3-10 characters in length

PhoneNumberTypes = Enumerated
  1 Home
  2 Cell
  3 Office
```

An untagged Choice with a single field can be used to define an alias for FieldType. The `combine` option has no effect when there is only one field.

4.2.3.4 Field Options

The FieldOptions applicable to Union types are shown in Table 4-9:

Table 4-9: Union Field Options

ID	Chr	Type	Name	Description
0x26	&	Integer	tagId	field holding the tag used for a Tagged Union
0x4E	N	Boolean	not	value is not an instance of FieldType in an untagged Union

4.2.3.4.1 TagId

A tagged union within a structured type may use the `tagId` option to specify a separate field within that type to be used as its tag. The value of the designated field must be a valid field identifier for the Choice, and is normally an Enumerated type generated from the Choice using the Derived Enumeration shortcut:

```
Connection = Record
  1 version      Enumerated(Enum[IP-Addr])    // src and dst versions must agree
  2 source       IP-Addr(TagId[version])
  3 destination  IP-Addr(TagId[version])

IP-Addr = Choice
```

```
   1 v4           IPv4-Addr
   2 v6           IPv6-Addr
```

4.2.3.4.2 Not

A field within an untagged union may use the `not` (logical negation) option to complement its match result. This option is valid only in an `allOf` Choice where one or more fields restrict the set of instances, because a complement without a restriction matches instances of arbitrary size, type and complexity.

```
UserName = Choice(allOf)              // lower, upper and digits, but not all digits.
   1  String{pattern="^[a-zA-Z0-9]$"}  // a::
   2  String{4..*} [1..16]            // b::
   3  !String{pattern="^[0-9]$"}      // c::
```

4.2.3.5 Union Type Conformance Requirements

- The FieldIDs of a Choice(anyOf) type MUST be numbered sequentially starting at 1.
- A value MUST be classified against the fields of a Choice(anyOf) type in field order and as an instance of the first matching field.
- The `not` FieldOption MUST appear only in a Choice(allOf) type containing at least one field without a `not` option.

4.2.4 General Type Options

The TypeOptions applicable to all core types are shown in Table 4-10:

Table 4-10: General Type Options

ID	Chr	Type	Name	Description
0x65	e	TypeRef	extends	Inheritance extension: superset of referenced type
0x72	r	TypeRef	restricts	Inheritance restriction: subset of referenced type
0x61	a	Boolean	abstract	Inheritance abstract: non-instantiatable type
0x66	f	Boolean	final	Inheritance final: cannot be subtyped

4.2.4.1 Type Inheritance

UML defines inherited classifiers, and JADN defines a mechanism for constructing DataType inheritance hierarchies using the `extends` and `restricts` TypeOptions. Type inheritance is static; it can be implemented as a shortcut that transforms inherited type definitions into expanded form prior to use, or as a runtime classifier operation.

Unlike class inheritance, type inheritance mechanisms are defined using a simple subset rule:

- If type B `extends` type A, then every instance of A is also an instance of B
- If type B `restricts` type A, then every instance of B is also an instance of A
- The `abstract` TypeOption indicates that the type cannot be used as a classifier; values may be classified against its subtypes.
- The `final` TypeOption indicates that this type can be used as a classifier but cannot have subtypes.

Although the subset rule is definitive and inheritance TypeOptions are valid for all core types, in practice inheritance is useful with only some types:

- **Primitive:** Inheritance is not useful with primitive types because:
  - It is not possible to extend a Primitive type because every value that could be an instance of that type already is.
  - It is not useful to restrict a Primitive type because the options defined in Section 4.2.1 perform restrictions directly without referencing a parent type.
  - Determining subsets analytically is not always practical. But an untagged Choice (`anyOf` or `allOf`) of types based on the same primitive type is equivalent to extend or restrict respectively.

Examples:

```
Name1 = Choice(anyOf)               // Extend: 2915, a34c, D72F are valid.  g16H is not.
   1  String{pattern="^[a-z0-9]$"}   // a::
```

```
   2  String{pattern="^[A-Z0-9]$"}   // b::

Name2 = Choice(allOf)               // Restrict: 2915 is valid.  a34c, D72F, g16H are not.
   1  String{pattern="^[a-z0-9]$"}   // a::
   2  String{pattern="^[A-Z0-9]$"}   // b::
```

- **Compound:**
  - Inheritance may not be useful with unstructured compound types (ArrayOf and MapOf) because the minLength and maxLength options defined in Section 4.2.2 are used directly to define collections with different cardinality limits without referencing a parent type.
  - Inheritance is used to add, remove, or modify the cardinality of fields in structured compound types.

Examples:

```
Entity = Record abstract               // Base type, cannot be instantiated
  1 id      Integer
  2 name    String optional

Person = Record extends(Entity)        // Add email address
  3 email   String /email optional

AnonymousPerson = Record restricts(Person) final  // Prohibit "name", no subtypes
  2 name    String [0]
```

- **Enumerated:**
  - Items can be added to an Enumerated type using `extends`.
  - No mechanism is currently defined to remove items from an Enumerated type.

Examples:

```
Colors1 = Enumerated                       // Primary colors
  5 red
  3 green
 16 blue

Colors2 = Enumerated extends(Colors1)     // Primary and secondary colors
  2 yellow
  7 magenta
  6 cyan
```

4.2.4.2 General Type Conformance Requirements

- A type MUST NOT have more than one `extends` or `restricts` TypeOption.
- A type MUST NOT have both `extends` and `restricts` TypeOptions.
- A type with an `extends` or `restricts` TypeOption MUST have the same CoreType as the type referenced by that option.

4.2.5 Semantic Validation

Semantic validation supplements type validation, ensuring that data values are within boundaries that applications will understand. Each format type option is a semantic validation keyword that references requirements defined by authoritative resources outside this specification.

The TypeOptions field of a type definition (Section 4.1) is an id:value mapping whose keys must be unique. But format options have no value; the keyword is part of the key so a type may include multiple format options.

ID	Chr	Type	Name	Description
0x2f	/	Enumerated	format	Semantic validation keyword

4.2.5.1 JADN Semantic Validation Keywords

JADN types define both logical values and literals, and format options affect both validation and translation between values and text representations. See Section 6. The JADN format keywords are shown in Table 4-11:

Table 4-11: JADN Formats

Keyword	Type	Requirement
i<n>	Integer	Signed n-bit integer, value must be between -2^(n-1) and 2^(n-1) - 1.
u<n>	Integer	Unsigned integer or bit field of n bits, value must be between 0 and 2^n - 1.
d<n>	Integer	Decimal integer scale factor of 10^n: for n>0 value has n fractional digits.
f16	Number	IEEE 754 Half-Precision Float
f32	Number	IEEE 754 Single-Precision Float
f64	Number	IEEE 754 Double-Precision Float
f128	Number	IEEE 754 Quadruple-Precision Float
f256	Number	IEEE 754 Octuple-Precision Float
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
eui	Binary	IEEE Extended Unique Identifier (MAC Address), EUI-48 or EUI-64 as specified in EUI
uuid	Binary	Universally Unique ID (UUID) as defined in RFC 9562
tag-uuid	Array	UUID with string prefix
date-time	Integer	POSIX time: the number of seconds since the Epoch
date	Integer	POSIX time
time	Integer	POSIX time
duration	Integer	A number of seconds

Integer and Number Formats

The signed and unsigned integer keywords `/i<n>` and `/u<n>` indicate a range constraint on a logical value, equivalent to the `minInclusive` and `maxInclusive` options using two's-complement bounds for signed integers. They also indicate the size of the bit field used to hold a literal value in direct binary data format.

The decimal scale factor keyword `/d<n>` indicates that an Integer holds an application value multiplied by the specified power of 10, using an integer to hold a fixed-precision rational number, or changing the unit scaling of a physical value:

```
Amount = Integer /d2    // Integer 152 represents an application value of 1.52,
                        // changing currency unit from US dollars to cents
```

The IEEE 754 floating point number keywords `/f#` indicate the significand and exponent ranges of logical Number instances, and the size and structure of lexical Number instances when using binary data formats.

Address and Identifier Formats

The `/uuid` keyword indicates a Universally Unique IDentifier (UUID), a 128 bit Binary label used to uniquely identify items, structured and serialized as defined in RFC 9562.

The `tag-uuid` keyword indicates an Array consisting of a String prefix and a Binary UUID, similar in purpose to a [STIX] Section 2.9 `Identifier`. Although STIX defines the prefix to be the *type* property of the object identified by the UUID, this specification is not specific to any message protocol and does not constrain

prefix content:

```
ObjectId = Array /tag-uuid
   1  String                          // prefix:: Type Prefix
   2  UUID                            // uuid:: Unique Identifier
```

When serialized in a text data format the `prefix` and `uuid` fields are separated by two dashes:

```
"ipv4-addr--ff26c055-6336-5bc5-b98d-13d6226742dd"
```

Time Formats

The meaning of an Integer with a time-related option is defined by the Portable Operating System Interface (POSIX) specification as "the number of seconds since the Epoch". An epoch is a fixed date and time used as a reference from which time is measured. The Unix epoch is 00:00:00 UTC on January 1, 1970, but POSIX permits other epochs such as 00:00:00 UTC on January 1, 1900. Interoperability between systems using Integer time representations requires them to have a common epoch; in practice this means the Unix epoch is used unless specifically documented otherwise. POSIX also defines the relationship between integer time and the `tm` calendar time structure, which includes tm_year, tm_mon, tm_mday, tm_hour, tm_min, tm_sec.

The logical value of an Integer with the `date` keyword is any Integer corresponding to the specified year, month and day of month, ignoring the time fields.
The logical value of an Integer with the `time` keyword is any Integer corresponding to the specified hour, minute and second, ignoring the date fields.

The Integer type with these keywords is a logical value independent of representation, which can include strings in RFC 3339 format, other date and time formats, decimal string, hex string, base64 string, or an integer value in binary serializations.

The decimal scale factor format `/d<n>` can be used with Integer times to specify time resolution:

```
Timestamp = Integer /date-time          // 1727877600 sec:     2024-10-02T15:00:00Z
Timestamp-ms = Integer /date-time /d3   // 1727877600000 msec: 2024-10-02T15:00:00.000Z
```

A String type with a time-related keyword is a logical string equal to its text representation, where different strings are non-equal values that sort alphabetically even if they represent the same logical time:

```
Timestamp2 = String /date-time

"2024-10-02T10:00:00-05:00"
"2024-10-02T15:00:00Z"
"2024-10-02T15:00:00.000Z"
"10:00:00 AM, October 2, 2024 EST"
"Wednesday, October 2, 2024 11:00:00 AM GMT-04:00 DST"
```

4.2.5.2 XSD Semantic Validation Keywords

XML Schema Definition Language ([XSD]) Section 3 defines a set of built-in DataTypes using a text-centric approach:

> The *value space* of *anyAtomicType* is the union of the value spaces of all the *primitive* datatypes defined here or supplied as implementation-defined primitives.

Information models are value-centric: the JADN *value space* consists of the five Primitive types defined in Section 4.2.1, and the *lexical space* is constructed using semantic keywords defined here or supplied from elsewhere. This difference has several effects:

- Enumerated is a first-class JADN DataType, not a facet of string or integer representations.
- Integer and Number are distinct first-class JADN DataTypes, not subsets of a *decimal* DataType. Open and closed intervals apply to both Integers and Numbers.
- System time (Epoch + Integer offset and the Seven-property subset of POSIX `tm`) is the value space of time-related Integers. The lexical space is broad, and lexical mappings beyond ISO 8601 (DMY/YMD/MDY, 12/24 hour, locale specifics) are out of scope but can be expressed in JADN as externally-defined format options.

Table 4-12 shows XSD-derived format options. Many are aliases for JADN options applicable to all serialized data formats; some are specific to XML but may be generalized to all serializations.

Table 4-12: XSD Formats

XSD DataType	JADN DataType	JADN Opts	XSD-compatible
string	String		
- normalizedString	String		/normalizedString
- token	String		/token
- language	String		/language
- name	String		/name
boolean	Boolean		
decimal (integer)	-	-	-
- integer	Integer		
- long	Integer	/i64	/long
- int	Integer	/i32	/int
- short	Integer	/i16	/short
- byte	Integer	/i8	/byte
- nonNegativeInteger	Integer	[0, *]	/nonNegativeInteger
- positiveInteger	Integer	(0, *]	/positiveInteger
- unsignedLong	Integer	/u64	/unsignedLong
- unsignedInt	Integer	/u32	/unsignedInt
- unsignedShort	Integer	/u16	/unsignedShort
- unsignedByte	Integer	/u8	/unsignedByte
- nonPositiveInteger	Integer	[*, 0]	/nonPositiveInteger
- negativeInteger	Integer	[*, 0)	/negativeInteger
decimal (float)	Number	-	-
float	Number	/f32	/float
double	Number	/f64	/double
duration	Integer	/duration	
- dayTimeDuration	Integer		/dayTimeDuration
- yearMonthDuration	Integer		/yearMonthDuration
dateTime	Integer	/date-time	/dateTime
time	Integer	/time	
date	Integer	/date	
gYearMonth	Integer		/gYearMonth
gYear	Integer		/gYear

XSD DataType	JADN DataType	JADN Opts	XSD-compatible
gMonthDay	Integer		/gMonthDay
hexBinary	Binary	/x, /X	/hexBinary
base64Binary	Binary	/b64	/base64Binary
anyUri	String	/uri, /iri	/anyUri
QName	String		/QName
Notation	String		/Notation

4.2.5.3 JSON Schema Semantic Validation Keywords

Table 4-13 shows semantic validation keywords defined in [JSON Schema] Section 7.3. Because JSON Schema defines only text representations, these keywords have the meanings listed here when used with the JADN String type. Table 4-11 defines the meaning of some of these keywords when used with types other than String.

For example, a String with `date-time` format has literal values such as:

- "2024-10-02T10:00:00-05:00"
- "2024-10-02T15:00:00Z"
- "2024-10-02T15:00:00.000Z"

These are unequal strings even though they represent the same timestamp.

Table 4-13: JSON Schema Formats

Keyword	Type	Requirement
date-time	String	String literal RFC 9557 Section 4.1 "date-time-ext"
date	String	String literal RFC 3339 Section 5.6 "full-date"
time	String	String literal RFC 3339 Section 5.6 "full-time"
duration	String	String literal RFC 3339 Appendix A "duration"
email	String	"Mailbox" as defined in RFC 5321 Section 4.1.2
idn-email	String	"Mailbox" as defined in RFC 6531 Section 3.3
hostname	String	RFC 1123 Section 2.1
idn-hostname	String	RFC 1123 or RFC 5890 Section 2.3.2.3
ipv4	String	"dotted quad" representation as defined in RFC 2673 Section 3.2
ipv6	String	Text representation of an IPv6 address as defined in RFC 4291 Section 2.2
uri	String	RFC 3986
uri-reference	String	RFC 3986
iri	String	RFC 3987
iri-reference	String	RFC 3987
uuid	String	"hex-and-dash" representation of a UUID as defined in RFC 9562
uri-template	String	RFC 6570
json-pointer	String	RFC 6901 Section 5

Keyword	Type	Requirement
relative-json-pointer	String	No current specification, last I-D expired Dec 2023
regex	String	Regular Expression according to ECMA-262 Section 22.2.1 "Pattern"

5 Shortcuts

JADN consists of a set of core definition elements, plus several shortcuts that make type definitions more compact or support the DRY software design principle. Shortcuts are syntactic sugar that can be replaced by core definitions without changing their meaning. Expanding shortcuts into core definitions simplifies serialization and validation code and may aid understanding, but creates additional definitions that must be kept in sync.

The following shortcuts can be converted to core definitions:

- 5.1: Anonymous type definition within a field
- 5.2: Multi-value field multiplicity
- 5.3: Derived enumeration
- 5.4: MapOf type with Enumerated key type
- 5.5: Derived path enumeration

5.1 Anonymous Type Definition

This shortcut allows fields within a structured type to be defined anonymously. Expanding the definition generates a named type for each anonymous field, moves all TypeOptions included in the field to the generated type, and replaces the field type with a reference to the generated type. This requires the anonymous field to be a non-structured core type and any TypeOption values included in FieldOptions to apply to FieldType.

Example: a structured type with anonymous fields:

```
Coordinate = Record                         // A GPS coordinate
   1 latitude     Number [-90.0, 90.0]     // A Number between -90 and 90 degrees
   2 longitude    Number [-180.0, 180.0]   // A Number between -180 and 180 degrees
```

Expanded type with references to generated types:

```
Coordinate = Record                         // A GPS coordinate
   1 latitude     Coordinate.latitude      // A Number between -90 and 90 degrees
   2 longitude    Coordinate.longitude     // A Number between -180 and 180 degrees

Coordinate.latitude = Number [-90.0, 90.0]
Coordinate.longitude = Number [-180.0, 180.0]
```

5.2 Field Multiplicity

Fields may be defined to have multiple values of the same type. Expanding converts each field that can have more than one value to a separate ArrayOf type. The multiplicity (`minOccurs` and `maxOccurs`) FieldOptions (Section 4.2.2.2) are moved from FieldOptions to the minimum and maximum length (`minLength` and `maxLength`) TypeOptions (Section 4.2.3)) of the new ArrayOf type, except that if `minOccurs` is 0 (field is optional), it remains in FieldOptions and the new ArrayOf type has a minimum length of 1.

Example:

```
Roster = Record
   1 org_name   String
   2 members    Member [0..*]   // Optional repeated: minOccurs=0, maxOccurs=MAX_DEFAULT
```

Expanding replaces this with:

```
Roster = Record
   1 org_name   String
   2 members    Roster.members optional // Optional: minOccurs=0, default maxOccurs (1)

Roster.members = ArrayOf(Member){1..*}  // Tool-generated array: minLength=1, no maxLength
```

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 shortcut:

```
Roster = Record
   1 org_name   String
   2 members    Members     // members field is required: default minOccurs (1), maxOccurs (1)

Members = ArrayOf(Member)   // Explicitly-defined array: no minLength, no maxLength
```

5.3 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. Expanding removes `enum` from Type Options and adds fields containing FieldID, FieldName, and FieldDescription from each field of the referenced type.

In JADN-IDL (Section 7.1) 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".

Expanding 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)
```

Expanding replaces the Channel and ChannelMask definitions with:

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

ChannelMask2 = ArrayOf(Channel)
```

5.4 MapOf With Enumerated Key

A MapOf type where `ktype` is Enumerated is equivalent to a Map. Expanding 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 shortcut, each ItemValue of the Enumerated type must be a valid FieldName.

Example:

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

Pixel3 = MapOf(Channel3, Integer)
```

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

5.5 Pointers

The Pointer shortcut generates a depth-first list of paths, similar to a recursive filesystem listing. Expanding replaces the Pointer shortcut with an Enumerated type containing a JSON Pointer pathname for each leaf type under the specified TypeRef. Link fields are listed but not followed.

Example:

```
BOM = Record
   1 bomFormat        BomFormat
   2 version          String
   3 metadata         Metadata

BomFormat = Enumerated
   1 cyclonedx
   2 spdx

Metadata = Record
   1 timestamp        String /date-time
   2 tools            Tool [1..*]

Tool = Record{1..*}
   1 vendor           String optional
   2 name             String optional

BomList = Enumerated(Pointer[BOM])
```

Expanding replaces BomList with:

```
BomList = Enumerated
   1 bomFormat
   2 version
   3 metadata/timestamp
   4 metadata/tools/#/vendor
   5 metadata/tools/#/name
```

6 Serialization and Data Formats

Applications may use any internal information representation that exhibits the characteristics defined in Section 4. 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

6.1 Verbose JSON Serialization

The following serialization rules represent JADN data types in a human-readable JSON format using name-value encoding for tabular data.

- When using JSON serialization, instances of JADN types without a format option listed in this section MUST be serialized as shown in Table 6-1:

Table 6-1: Verbose JSON

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**
String	JSON **string**
Enumerated	JSON **string** ItemValue
Enumerated with "id"	JSON **integer** ItemID
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	JSON **object**. Property keys are FieldNames.

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 shown in Table 6-2:

Table 6-2: Verbose JSON Formats

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

Specifications MAY define additional format options for textual representation of Binary, Integer, Number or Array data.

6.2 Compact JSON Serialization:

The following serialization rules represent JADN types in a human-readable JSON format using positional encoding for tabular data.

- When using Compact JSON serialization, instances of JADN types MUST be serialized as in Table 6.1 except as shown in Table 6-3.

Table 6-3: Compact JSON

JADN Type	Concise JSON Serialization Requirement
Record	JSON **array** of values with types specified by FieldType. Omitted optional values are **null** if before the last specified value, otherwise omitted.

6.3 Concise JSON Serialization:

Concise JSON serialization rules represent JADN data types in a format optimized for minimum size. JSON data in this format may be used directly for communication or to visualize the content of CBOR-serialized data.

- When using Concise JSON serialization, instances of JADN types MUST be serialized as in Table 6.1 except as shown in Table 6-3.

Table 6-4: Concise JSON

JADN Type	Concise JSON Serialization Requirement
Enumerated	JSON **integer** ItemID
Choice	JSON **object** with one property. Property key is the FieldID converted to string.
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. 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	JSON **array** of values with types specified by FieldType. Omitted optional values are **null** if before the last specified value, otherwise omitted.

All formats specifying a textual representation for Binary, Integer, Number, or Array types are ignored when using Concise serialization.

6.4 CBOR Serialization

The following serialization rules are used to represent JADN data types in Concise Binary Object Representation (CBOR) format. The initial byte of each encoded data item contains both information about the major type (the high-order 3 bits) and additional information (the low-order 5 bits). In this section 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 shown in Table 6-5.

Table 6-5: CBOR Serialization

JADN Type	CDDL	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).
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.
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	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 shown in Table 6-6.

Table 6-6: CBOR Serialization Formats

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).
f64	Number	**float64**: IEEE 754 Double-Precision Float (#7.27).

6.5 XML Serialization:

- When using XML serialization, instances of JADN types without a format option listed in this section MUST be serialized as shown in Table 6-7.

Table 6-7:

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"/>
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
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 shown in Table 6-8.

Table 6-8: XML Serialization Formats

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"/>

7 Alternate Schema Representations

Section 4 defines the normative JSON format of JADN type definitions. Although JSON data is unambiguous, it is not ideal as a documentation format. This section suggests several more readable ways of describing and documenting information models.

Note: *This section is informative*

7.1 Information Definition Language

JADN Interface Definition Language (IDL) is a textual representation of JADN type definitions. It replicates the structure of Section 4.1 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, meaning that the IDL described here is unambiguous and complete. But it is not intended to be immutable; syntactic details may be updated to accommodate new use cases or improve usability without affecting the JADN standard.

The JADN-IDL definition formats are:

Primitive types:

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

Enumerated type without the `id` option:

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

Compound types without the `id` option:

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

Structured types with the `id` TypeOption treat the item/field name as an informative label and display it in the description followed by a label terminator ("::"):

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

/* Choice#, Map# */
TypeName = TYPESTRING                    // TypeDescription
    FieldID FIELDSTRING                  // FieldName:: FieldDescription
    ...
```

Type Options:

TYPESTRING is the value of CoreType or FieldType, followed by string representations of the type options, if applicable to TYPE as specified in Section 4.2.

- TYPEREF is a type name with optional namespace prefix as specified in Section 3.1.3.
- FMTNAME is the name of a semantic validation function as specified in Section 4.1.5.

```
    TYPESTRING  = TYPE [ID] [FUNC] [RANGEPAT] [FORMAT] [KW]  ; TYPE is CoreType or FieldType
    ID          = ".ID"
    FUNC        = "(" TYPEREF ["," TYPEREF] ")"         ; if TYPE is MapOf, ArrayOf
                | "(" FUNCNAME "[" TYPEREF "])"         ; if TYPE is Enumerated
    RANGEPAT    = "{" NUM [".." NUM] "}"
                | "{pattern=" DQUOTE 1*STR DQUOTE "}"   ; if TYPE is String. *STR should be a valid
regular expression
    FORMAT      = " /" FMTNAME
    FUNCNAME    = "Enum" | "Pointer"
    KW          = "unique" | "set" | "unordered"        ; if TYPE is ArrayOf
```

```
DQUOTE      = %x22                              ; Double-quote character (")
STR         = %x20-%x7e                         ; Visible characters plus space
```

Field Options:

Type and Field options affect the entire line of a field's IDL text:

```
FIELDLINE   = INT FIELDSTRING
FIELDSTRING = [FIELDNAME] [DIR] TYPE [MULT | TAGID] [FIELDDESC]
INT         = 1*DIGIT
DIR         = "/"
TYPE        = TYPESTRING
            | "Key(" TYPESTRING ")"
            | "Link(" TYPESTRING ")"
MULT        = "[" INT [".." INT] "]"
TAGID       = "(TagId[" (INT | FIELDNAME) "])"
FIELDDESC   = "//" [FIELDNAME "::"] STR
```

7.2 Property Tables

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 shows 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 options into a separate column:

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

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

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

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

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

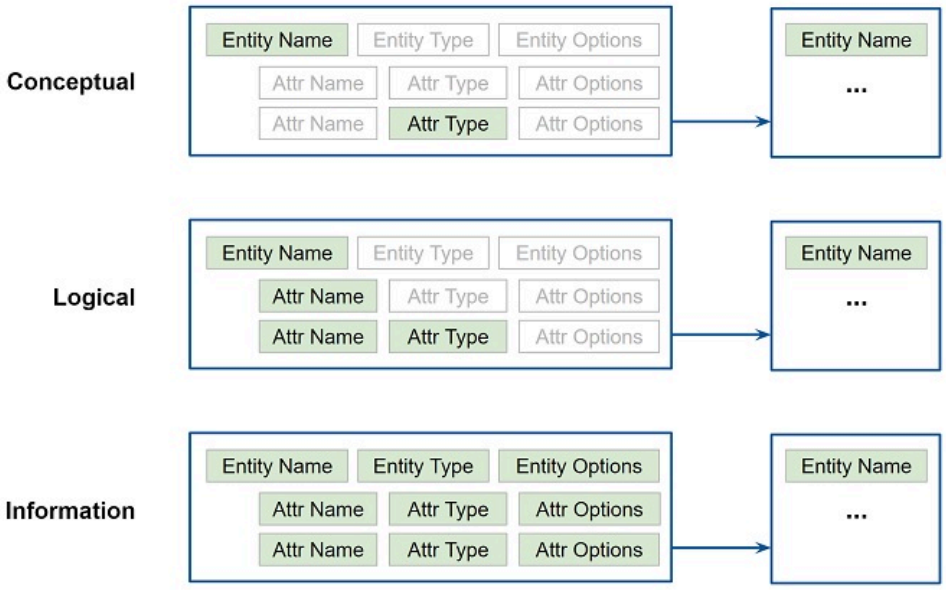
Example Markdown Table:

Type: Person (Record)

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

7.3 Entity Relationship Diagrams

The same type definition structure can be populated with various levels of detail. At the conceptual level, only TypeName is present, along with FieldType for attributes that reference other model-defined types. At the logical level FieldName is populated for both core and reference attribute types. In a full information model, all Type and Options elements are defined:



Information models extend the Conceptual/Logica/Physical design process. While UML defines a class diagram format that has been adopted for use in that process, it does not define a datatype diagram format suitable for representing information models. As noted in the introduction, logical/class models are undirected graphs with semantic relationships while information/datatype models are directed graphs with two relationship types: contain and reference. Information models may be represented as entity relationship diagrams using the following conventions:

1. Solid edges represent container relationships, dashed edges represent references.
2. All edges are directed, from container to contained type or from referencing to referenced type.

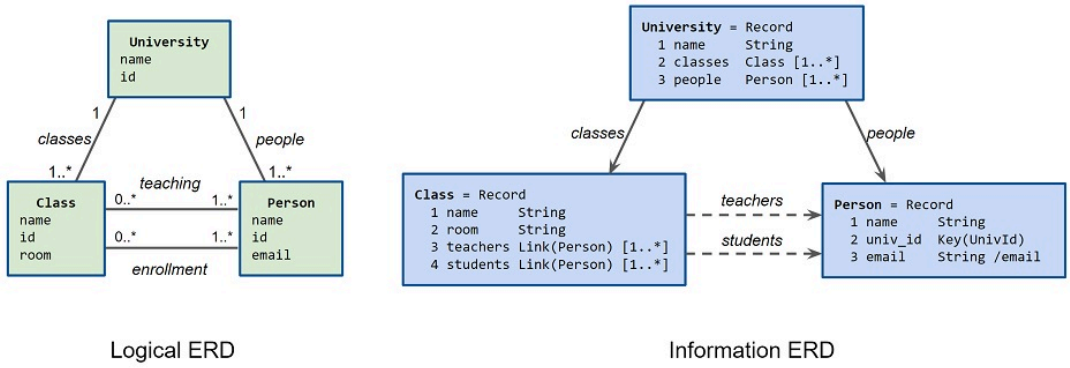


Figure 7-1: Logical and Information Entity Relationship Diagrams

The edge type and direction show how instances are serialized, in this case using references from Class to Person. An alternate information model derived from the same logical model might use references "teaches" and "enrolled_in" from Person to Class.

Figure 5-2 is a GraphViz "dot" file generated from the University information model showing a conceptual level of detail. Dot diagrams may be viewed at, for example, https://sketchviz.com.

```
# package: http://example.com/uni
# exports: ['University']

digraph G {
  graph [fontname=Times, fontsize=12];
  node [fontname=Arial, fontsize=8, shape=box, style=filled, fillcolor=lightskyblue1];
  edge [fontname=Arial, fontsize=7, arrowsize=0.5, labelangle=45.0, labeldistance=0.9];
  bgcolor="transparent";

  n0 [label="University"]
    n0 -> n1 [label="classes", headlabel="1..*", taillabel="1"]
    n0 -> n2 [label="people", headlabel="1..*", taillabel="1"]
  n1 [label="Class"]
    n1 -> n2 [style="dashed", label="teachers", headlabel="1..*", taillabel="1"]
```

```
    n1 -> n2 [style="dashed", label="students", headlabel="1..*", taillabel="1"]
  n2 [label="Person"]
}
```

Figure 7-2: GraphViz Source for University Conceptual ERD

Figure 7-3 is an example instance of the University type serialized in verbose and compact JSON data formats:

```
{
  "name": "Faber College",
  "classes": [
    {
      "name": "ECE1010",
      "room": "DRGN 105",
      "teachers": ["U-004932"],
      "students": ["U-194325", "U-029437"]
    },
    {
      "name": "ECE1750",
      "room": "FLRS 102",
      "teachers": ["U-004932"],
      "students": ["U-127439", "U-194325", "U-029437"]
    }
  ],
  "people": [
    {
      "name": "Damien Braun",
      "univ_id": "U-004932",
      "email": "d.braun@faber.edu"
    },
    {
      "name": "Ellie Osborne",
      "univ_id": "U-194325",
      "email": "ellie.osborne@faber.edu"
    },
    {
      "name": "Pierre Cox",
      "univ_id": "U-029437",
      "email": "pc9000@outlook.com"
    },
    {
      "name": "Alden Cantrel",
      "univ_id": "U-127439",
      "email": "alden.cantrel@faber.edu"
    }
  ]
}
```

```
[
  "Faber College",
  [
    ["ECE1010", "DRGN 105", ["U-004932"], ["U-194325", "U-029437"]],
    ["ECE1750", "FLRS 102", ["U-004932"], ["U-127439", "U-194325", "U-029437"]]
  ],
  [
    ["Damien Braun", "U-004932", "d.braun@faber.edu"],
    ["Ellie Osborne", "U-194325", "ellie.osborne@faber.edu"],
    ["Pierre Cox", "U-029437", "pc9000@outlook.com"],
    ["Alden Cantrel", "U-127439", "alden.cantrel@faber.edu"]
  ]
]
```

Figure 7-3: JSON instance of University

8 Conformance

Information Modeling is applied within a system design process that may include:

- IM Design
  - Abstract Schema Design and Validation
  - Alternate Schema Format Translation
- Message Processing
  - Single Format Message Validation
  - Multiple Format Lossless Roundtrip Message Translation
- Concrete Schema Conversion

As noted in the introduction, an information modeling language is a formal syntax that allows users to capture data semantics and constraints. This specification defines the JADN IM language, and its conformance requirements address schema design and validation. Although Sections 6 and 7 present example message encoding rules and alternate schema presentation formats, this specification has no conformance requirements related to those activities.

Conforming implementations SHALL satisfy all conformance requirements listed in Sections 1-5 of this document, including the following sections:

- 3.1.3 Package
- 4.1.5 Types
- 4.2.1.6 Primitive Types
- 4.2.2.4 Compound Types
- 4.2.3.5 Union Types
- 4.2.4.2 Inherited Types

Appendix A. References

This appendix contains the normative and informative references that are used in this document. Normative references are specific (identified by date of publication and/or edition number or version number) and Informative references are either specific or non-specific.

While any hyperlinks included in this appendix were valid at the time of publication, OASIS cannot guarantee their long-term validity.

A.1 Normative References

The following documents are referenced in such a way that some or all of their content constitutes requirements of this document.

[ECMASCRIPT]

ECMA International, *"ECMAScript 2024 Language Specification"*, ECMA-262 15th Edition, June 2024, https://www.ecma-international.org/ecma-262 (*or corresponding section(s) in current edition*).

[EUI]

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

[IEEE754]

"Floating Point Arithmetic", IEEE Std 754-2019, https://ieeexplore.ieee.org/document/8766229, ISO/IEC 60559:2020, https://www.iso.org/obp/ui/en/#iso:std:80985

[IRI]

Duerst, M., Suignard, M., *"Internationalized Resource Identifiers (IRIs)"*, January 2005, https://datatracker.ietf.org/doc/html/rfc3987

[JSONSCHEMA]

Wright, A., Andrews, H., Hutton, B., *"JSON Schema Validation"*, Internet-Draft, 16 June 2022, https://json-schema.org/draft/2020-12/draft-bhutton-json-schema-validation-01.

[RFC791]

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

[RFC2119]

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

[RFC2673]

Crawford, M., *"Binary Labels in the Domain Name System"*, RFC 2673, August 1999, https://www.rfc-editor.org/rfc/rfc2673.

[RFC3339]

Klyne, G., Newman, C., *"Date and Time on the Internet: Timestamps"*, RFC 3339, July 2002, https://www.rfc-editor.org/rfc/rfc3339.html

[RFC3986]

Berners-Lee, T., Fielding, R., Masinter, L., *"Uniform Resource Identifier (URI): Generic Syntax:*, RFC 3986, https://www.rfc-editor.org/rfc/rfc3986.html.

[RFC4291]

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

[RFC4632]

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

[RFC4648]

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

[RFC5234]

Crocker, D., Overell, P., *"Augmented BNF for Syntax Specifications: ABNF"*, RFC 5234, January 2008, https://www.rfc-editor.org/rfc/rfc5234.

[RFC6570]

Gregorio, J., Fielding, R., Hadley, M., Nottingham, M., Orchard, D., *"URI Template"*, RFC 6570, March 2012, https://www.rfc-editor.org/rfc/rfc6570.html.

[RFC6901]

Bryan, P., Zyp, K., Nottingham, M., *"JavaScript Object Notation (JSON) Pointer"*, RFC 6901, April 2013, https://www.rfc-editor.org/rfc/rfc6901.

[RFC8949]

Bormann, C., Hoffman, P., *"Concise Binary Object Representation (CBOR)"*, RFC 8949, October 2013, https://www.rfc-editor.org/rfc/rfc8949.

[RFC7405]

Kyzivat, P., *"Case-Sensitive String Support in ABNF"*, RFC 7405, December 2014, https://www.rfc-editor.org/rfc/rfc7405.

[RFC8174]

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

[RFC8200]

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

[RFC8259]

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

[RFC9557]

Sharma, U., Bormann, C., *"Date and Time on the Internet: Timestamps with Additional Information"*, IETF RFC 9557, April 2024, https://www.rfc-editor.org/rfc/rfc9557.

[RFC9562]

Davis, K., Peabody, B., Leach P., *"Universally Unique IDentifiers (UUIDs)"*, IETF RFC 9562, May 2024, https://www.rfc-editor.org/rfc/rfc9562.

[POSIX Time]

IEEE and The Open Group, *"POSIX.1-2024 - standard operating system and environment: time()"*, "https://pubs.opengroup.org/onlinepubs/9799919799/functions/time.html"

[XML Namespaces]

W3C, *"Namespaces in XML 1.0"*, December 2009, https://www.w3.org/TR/xml-names/

[XSD]

W3C, *"XML Schema Definition Language (XSD) 1.1 Part 1: Structures"*, 5 April 2012, https://www.w3.org/TR/xmlschema11-1.
W3C, *"XML Schema Definition Language (XSD) 1.1 Part 2: Datatypes"*, 5 April 2012, https://www.w3.org/TR/xmlschema11-2.

A.2 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.

[DATAMOD]

InfoAdvisors, *"What are Conceptual, Logical, and Physical Data Models?"*, https://www.datamodel.com/index.php/articles/what-are-conceptual-logical-and-physical-data-models.

[DIEK]

Dammann, Olaf, *"Data, Information, Evidence, and Knowledge"*, https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6435353/pdf/ojphi-10-e224.pdf.

[DRY]

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

[ENUM]

"Enumerated Type", https://en.wikipedia.org/wiki/Enumerated_type.

[FDT]

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

[FIX]

FIX Trading Community Technical Standards, https://www.fixtrading.org/standards/.

[GRAPH]

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

[GRAPHVIZ]

"Graph Visualization Software", https://graphviz.gitlab.io/.

[INFORMATION MODELING]

Lee, Y. Tina, *"Information Modeling: From Design to Implementation"*, IEEE Transactions on Robotics and Automation, 1999, https://tsapps.nist.gov/publication/get_pdf.cfm?pub_id=821265.

[JADN-CN]

OASIS, *"Information Modeling with JADN"*, https://docs.oasis-open.org/openc2/imjadn/v2.0/imjadn-v2.0.md.

[ORDER]

LaFontaine, Robin, *"Element order is always important in XML, except when it isn't"*, Balisage: The Markup Conference, 2021, https://www.balisage.net/Proceedings/vol26/html/LaFontaine01/BalisageVol26-LaFontaine01.html.

[PROTO]

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

[RDF]

W3C, *"RDF 1.2 Concepts and Abstract Syntax"*, https://www.w3.org/TR/rdf12-concepts/.

[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://www.rfc-editor.org/rfc/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/rfc/rfc3552.

[RFC5321]

Klensin, J., *"Simple Mail Transfer Protocol"*, RFC 5321, October 2008, https://www.rfc-editor.org/rfc/rfc5321.html.

[RFC6531]

Yao, J., Mao, W., *"SMTP Extension for Internationalized Email"*, RFC 6531, February 2012, https://www.rfc-editor.org/rfc/rfc6531.html.

[RFC7303]

Hansen, T., Melnikov, A., *"Additional Media Type Structured Syntax Suffixes"*, RFC 7303, January 2013, https://www.rfc-editor.org/rfc/rfc7303.

[RFC7493]

Bray, T., "The I-JSON Message Format", RFC 7493, March 2015, https://www.rfc-editor.org/rfc/rfc7493.

[RFC8340]

Bjorklund, M., Berger, L., *"YANG Tree Diagrams"*, RFC 8340, March 2018, https://www.rfc-editor.org/rfc/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://www.rfc-editor.org/rfc/rfc8477.

[RFC8610]

Birkholz, H., Vigano, C., Bormann, C., *"Concise Data Definition Language"*, RFC 8610, June 2019, https://www.rfc-editor.org/rfc/rfc8610.html.

[STIX]

Bret Jordan, Rich Piazza, Trey Darley, *"Structured Threat Information Expression (STIX) Version 2.1"*, OASIS Cyber Threat Intelligence (CTI) TC, 10 June 2021, https://docs.oasis-open.org/cti/stix/v2.1/stix-v2.1.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]

"Union Type", Wikipedia, https://en.wikipedia.org/wiki/Union_type.

[TAGGEDUNION]

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

Appendix B. Safety, Security and Privacy 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.

Appendix C. Acknowledgments

C.1 Special Thanks

The following individuals shared their expertise during creation of this specification and are gratefully acknowledged:

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

C.2 Participants

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

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

Appendix D. Revision History

Changes from v1.0 to v2.0

- Add type inheritance options.
- Add untagged union options to Choice type.
- Allow multiple namespace prefixes to designate the same namespace.
- Define two special values for maxOccurs upper bound: "unspecified" and "unlimited".
- Split single range option into value range and length.
- Add format options:
  - /d - decimal scale factor for fixed-point Integer type
  - /tag-uuid for labeling uuid references to specific types
- Define separate format option behavior when applied to logical vs. text values.
- Add XML serialization rules.
- Define XSD-compatible format options.
- In package header:
  - rename "Information" to "Metadata" to avoid conflation with information modeling.
  - rename "exports" to "roots" to better describe purpose and effect.

Changes from v1.0 CSD 01 to v1.0

- Added serialization style description.
- Removed the Null core type.
- Added default values for type definition elements.
- Raised the default maximum length for type and field names from 32 to 64 characters.

Appendix E. 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.

As stated in the OASIS IPR Policy, the following three paragraphs in brackets apply to OASIS Standards Final Deliverable documents (Committee Specification, Candidate OASIS Standard, OASIS Standard, or Approved Errata).

[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 Standards Final Deliverable, 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 deliverable.]

[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 OASIS Standards Final Deliverable 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 OASIS Standards Final Deliverable. 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 OASIS Standards Final Deliverable 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 Standards Final Deliverable, 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.