Object Data Notation
File Format Specification
Version 1.0 Revision 1

By khhs

Table of Contents

Preface

Just a little something before this specification starts.

The specification you are currently reading was thrown together in the span of an hour or two. It is by no means anything in-depth or complex, and is simply there to give at least some specification to the format.

This means that the thing might not be well written, it might lack major points or more. But it is a version 1.0, and something like a second edition may be written at a later point in time.

The specification contains a “license” at the very bottom of the document.

Keywords

In the text, certain words may appear that could cause confusion for the reader. In order to prevent that, most of those words will be listed here alongside a definition.

Keyword	**Definition**
ODF	Open Documentation Format
Object	A collection of fields or properties
Type-safe	The type of everything is defined beforehand, and cannot change. If something is considered a string, it will remain a string
Integer	A 32-bit integer number
Float	Binary representation of a decimal number, short for floating point number
Double	A double precision number, with the same concept and structure as a floating point number
Short	A 16-bit integer number
Long	A 64-bit integer number
Byte	An 8-bit integer number
String	A series of characters that make up a piece of text
Boolean/Bool	A boolean is a representation of either true or false
Comment	A piece of text in the file that should be ignored by the parser
Serialize	The act of taking a piece of program data and converting it into text or binary that can be stored on disk
Deserialize	The act of taking something serialized and converting it back into program data.

On top of these keywords, there are also the “demand words”. These specify how an implementation should handle something.

MUST - All implementations need to do this to conform to the standard

SHOULD - This isn't required to be standard conforming, but is heavily recommended
COULD - This isn't required at all, but support is still according to standard

Please note that some of these can be inverted, into **MUST NOT** and **SHOULD NOT**.

There will also be a couple of fonts and styles used in the document. They are the following:

- Regular text, anything that contains written text
- `Code, anything that is more arbitrary, such as ODN examples`

Adding onto this, I would like to state that the word processing program I am using, doesn't use regular quotation marks, but instead uses some odd unicode ones, and because of this, you shouldn't copy-paste anything from this specification, but should instead type everything in manually.

Design

ODN was designed to make a format that was like the JSON format, but with the more granular types of NBT.

The format is centered around fields. A field can either be a tag or a value. It also follows an object hierarchy, in such a way that a value can have children, either unnamed(for lists) or named(for objects).

These fields are then split up into 2 types, one of which has 2 child-types(making 4 in total):

- Value fields - ints, strings, longs etc.
    - Object fields - A type of value fields that contain child-tags.
    - List fields - A type of value fields that contain child value fields
- Tag fields - A field that can be used to specify a named field

These 4 types can then be used to construct any document.

The format has a root field, which then is the root for any possible hierarchy. The document may not contain multiple roots.

Let us use an example to describe this.

Example

Let us say you have a car data you want to store using ODN. Let's start by specifying the car data structure:

```
CarData:
      int topSpeed = 10
      string name = “Ford Focus”
      double acceleration = 9.8
```

For those who aren't too experienced in reading pseudocode, this is a simple definition of the structure named CarData. It contains an int named topSpeed with the value of 10, a string called name with the value of *Ford Focus*, and a double called acceleration with the value of 9.8.
When serializing this data, we should get the following ODN:

```
(
      {topSpeed:int} 10
      {name:string} “Ford Focus”
      {acceleration:double} 9.8
)
```

And that looks just fine, but thanks to this thing in the language known as an auto-type, we can simplify the result into the following:

```
(
      {topSpeed} 10
      {name} “Ford Focus”
      {acceleration:double} 9.8
)
```

Now, you may wonder where the “int” and the “string” went, and why “double” is still there? Well, it is because of how auto-types work in ODN, and it is something that will be elaborated upon further later in the specification. For now, you’ve gotten a taste of the syntax and workings of the format, and that was my top priority.

Syntax

ODN has a simplistic syntax, and instead of spending ages describing it in text form, I'll make a table with the syntax in it. If something isn't explicitly written in the specification, don't assume it is valid. This includes single quotation marks. They are invalid.

Syntax	Description
`// ...`	A comment, replace the "..." with the contents of the comment
`{ name } value`	An auto-typed tag, replace "name" with the name of the tag and "value" with a value field
`{ name : type } value`	A typed tag, same as the auto-typed tag, but replace "type" with the value type.
`( ... )`	An object, replace the "..." with the tags of the object
`[ ... ]`	A list, replace the "..." with the values of the list
`< type > [ ... ]`	A valued list, same as the list, but replace "type" with the type of the values of the list
`“ ... “`	A string, replace the "..." with the contents of the string. These may contain escape codes.

These simple parts make up the entirety of ODN, not much, right?

A couple things have been left out of the table, most notably numerics and booleans, but that is because they're easier to describe via text.

A numeric is simply a series of numbers, e.g 5940. They cannot contain spaces, but they can contain a decimal. There must not be more than one decimal per numeric, but this decimal can be placed anywhere in the number, including at the start and at the end. At the very start of a numeric, a "-" may be put, to symbolize a negative number. Yes, this means that -.5 is correct.

Anywhere where I specify "name", "type" or "value", a valid identifier is expected. A valid identifier must start with a letter, and can only contain letters, numbers and underscore. Anything else should terminate the identifier. Identifiers can have values such as "float" or "bool".

A boolean is effectively an identifier with a special value. It must either be "true" or "false". If one of these is detected, the identifier should instantly be considered a boolean.

Escape Codes

Strings may contain escape codes, which are representations of special characters. They are recognized by a backslash followed by some character. They are as following:

Escape Code	**Description**
`\n`	Newline
`\t`	Horizontal tab
`\b`	Backspace
`\f`	Form feed
`\r`	Carriage return(commonly used on windows, `CRLF`, or `\r\n`)
`\”`	Double quotation marks
`\\`	Backslash

You can also skip line breaks in strings using backslashes, like so:

```
“still on the \
same line”
```

The Test

This section is meant for programmers, and might not be the most readable section. Sorry about that.

Below is attached a piece of ODN that should successfully parse in any ODN parser. It is a simple test, and the expected result will be attached as well.

```
// This is a testing file for any ODN parser.
// It uses every official feature of ODN, and should parse
successfully.
// This. Is. The. Standard.

(
    {auto_object_list} [
      (
            {test_idx_property} "property_value"
      )
      (
            {test_idx_property} "property_value"
      )
    ]

    {typed_list} <string> [
      "string_idx_0"
      "string_idx_1"
    ]

    {test_object:object} ()
    {test_object_auto} ()

    {test_object_children} (
      {test_child_property} "property_value"
    )

    {test_string_auto} "string"
    {test_string:string} "string"

    {test_string_escape_codes} "\n\t\b\f\r\"\\"
    {test_string_newline_break} "still on the \
same line" // Note that we go back to the start.

    {test_float_auto} 1.0
    {test_float:float} 1.0
```

```
    {test_float_number:float} 1
    {test_float_auto_short} .0

    {test_float_negative_auto} -1.0
      {test_float_negative:float} -1.0
      {test_float_negative_number:float} -1
      {test_float_negative_auto_short} -.0

    {test_int_auto} 1
    {test_int:int} 1

    {test_int_negative_auto} -1
      {test_int_negative:int} -1

    {test_double:double} 2.0
    {test_double_short:double} .0
    {test_double_number:double} 2

    {test_short:short} 1
    {test_long:long} 1
    {test_byte:byte} 1

    {test_bool:bool} true
    {test_bool_auto} false

)
```

The expected result of the test is a tree with the structure specified. Please note that the output is gathered from the initial parser, and thus might look a little odd. The escape code test has also been de-escaped here for readability.

```
> OdnObjectField
  > auto_object_list:list
      > OdnListField:object
      > OdnObjectField
      > test_idx_property:string
            > property_value
      > OdnObjectField
      > test_idx_property:string
            > property_value
  > typed_list:list
      > OdnListField:string
      > string_idx_0
      > string_idx_1
```

```
> test_object:object
  > OdnObjectField
> test_object_auto:object
  > OdnObjectField
> test_object_children:object
  > OdnObjectField
  > test_child_property:string
  > property_value
> test_string_auto:string
  > string
> test_string:string
  > string
> test_string_escape_codes:string
  > [\n\t\b\f\r\"\\]
> test_string_newline_break:string
  > still on the same line
> test_float_auto:float
  > 1
> test_float:float
  > 1
> test_float_number:float
  > 1
> test_float_auto_short:float
  > 0
> test_float_negative_auto:float
  > -1
> test_float_negative:float
  > -1
> test_float_negative_number:float
  > -1
> test_float_negative_auto_short:float
  > -0
> test_int_auto:int
  > 1
> test_int:int
  > 1
> test_int_negative_auto:int
  > -1
> test_int_negative:int
  > -1
> test_double:double
  > 2
> test_double_short:double
  > 0
```

```
> test_double_number:double
    > 2
> test_short:short
    > 1
> test_long:long
    > 1
> test_byte:byte
    > 1
> test_bool:bool
    > True
> test_bool_auto:bool
    > False
```

All of these tests are available online for copy-pasting as well, at the following links:

- https://openability.tech/odn_files/1.0/ODN_TEST.odn - The test ODN provided in here
- https://openability.tech/odn_files/1.0/ODN_TEST_RESULT.txt - The expected test results

License-like Thing

Definitions:

- Whenever “the work” is mentioned, the Object Data Notation File Format Specification is meant.
- Whenever “the author” is mentioned, OpenAbility is meant.
- Whenever “you” is mentioned, the consumer is meant

The work may be used in any way shape or form seen fit by you, including redistribution and modification, as long as the following criteria is met:

- The work and the authors are mentioned.
- This license alongside copyright notices is carried over
- Any modifications done are specified
- If the full work, or part of the full work is redistributed commercially, the parts that stem from the work is less than 50% of the end product.

In more understandable language it means:

- You are free to do whatever you want with the work, as long as you are crediting the original work and it’s authors, using the same license(with copyright notice), and you’re not trying to sell it as part of something where the work makes up 50% or more of the thing being sold.

If this “license” ever ends up meaning something legally, please make sure common sense is applied. It was thrown together in ten minutes and isn’t very well written.