Object Data Notation
File Format Specification
Version 1.1

By khhs

Written and edited by khhs, also known as Jimmy.

The Object Data Notation file format is part of the public domain, however this specification is not.

This is the 1st revision of the 1.1 specification, and the second version of ODN. The previous specifications are: *Object Data Notation File Format Specification Version 1.0 Revision 2, Object Data Notation File Format Specification Version 1.0 Revision 1* and *Object Data Notation File Format Specification Version 1.0*.

The ODN file format was created by khhs, inspired by JSON which was created by Douglas Crockford, and NBT which was created by Markus Persson.

Contact us at contact@openability.tech

The name *Ford Focus* is property of the *Ford Motor Company*, and is only used as an example name. We are not attempting to promote or defame the *Ford Focus* nor the *Ford Motor Company*.

The ODN test file and test file results are under Public Domain. The online links are not guaranteed to work nor safe, as we do not know when you're reading this. Because of this, anything malicious gained by accessing said links are your own responsibility.

Written and published in 2023

Table of Contents

A “Humorous” Heads Up

This specification is somewhat bad. Actually, somewhat bad might be an understatement. This specification is bad. It wasn’t written with any sort of coherency and was mainly just a project to get some sort of documentation written for the format. By doing this we just end up with a patchwork mess that in reality is just bodge on top of bodge, but that’s how most software works.

There is not any documentation in the world that is both cohesive and well written, at least not for any major projects. There comes a point at which the sheer amount of stuff that needs to be packed into the documentation makes it close to impossible to plan stuff out ahead of time. Especially for iteratively developed stuff. If you create documentation for your small library, you might end up with a simple docs page that describes what it needs to describe, and contains links and references to other things. Then you start working on that library more and more, and at the end you get this big conglomerate of functions and classes and interfaces, and only 35% of it is documented. So you go in and start documenting. The documentation starts off smoothly but then as time passes, it gets messier and messier. This is when you start making rules, and use said rules to document the entire project.

After documenting the project using these rules, you’ll probably end up with a cohesive piece of documentation, but it won’t be well written. Well written documentation needs to have freedom and adapt around what it is trying to describe, or you’ll end up with something stale that, whilst it covers most bases, is flawed in a lot of areas.

You could also write the entire thing without any rules, but then you’d have to plan everything out ahead of time for it to be cohesive, and we all know that’s not happening.

This has been my issue with any documentation I’ve ever read. And this specification is not an exception. It doesn’t follow any rules, and it isn’t cohesive. You’re not getting anything from any of the worlds.

This doesn’t matter.

The entire purpose of this specification is to provide a basis and description that can help you make a compatible parser or any other kind of processor.

It doesn’t even have to be compatible, just supporting 90% will probably work just as well.

That’s all I had to say.

Jimmy

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**
ODN	Open Documentation Notation
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.

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 or similar, 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 detailed 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 or named.

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
)
```

This ODN is very similar to the CarData we already had, it only has a couple differences, most notably, the lack of the “CarData” specifier. Now, let’s do this.

```
Object: root
 ↳ Tag: topSpeed
    ↳ Value: 10
 ↳ Tag: name
    ↳ Value: Ford Focus
 ↳ Tag: acceleration
    ↳ Value: 9.8
```

I’ve now converted this ODN into a field hierarchy of sorts. The root of the hierarchy is, that’s right, “root”. The root object then contains 3 child tags, one called “topSpeed”, one called “name” and one called “acceleration”. These each contain a child value each. That’s the general system. These values could also be an object, like so:

```
Object: root
 ↳ Tag: car
    ↳ Object
       ↳ Tag: topSpeed
          ↳ Value: 10
       ↳ Tag: name
          ↳ Value: Ford Focus
       ↳ Tag: acceleration
          ↳ Value: 9.8
 ↳ Tag: name
    ↳ Value: John Doe
```

In this reworked tree, we now have a root object, with a car tag and a name tag. The car tag holds the car object from earlier, and the name tag holds the value “John Doe”.

Hopefully this should give you a grasp of the ideology, and if it doesn’t, you can always study more about it in formats such as JSON or NBT.

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.
`# ...`	An extension declaration, replace the "..." with the extension specifier. Any leading/trailing whitespace should be ignored.

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”
```

Comments

Comments are **single-line only**. This means that you need to start each line of the comment with new double-slashes. A comment can start anywhere in code, except for inside strings, and they’ll effectively remove anything from the double-slashes to the newline character. They should however affect error messages by incrementing any line counters/column counters.

Types

ODN has a very basic type system, but it is still worth explaining, as it has some quirks to it.

The language features a couple types, most of them were explained in the keywords portion, but I'll still list them here:

- `object`
- `list`
- `string`
- `long`
- `int`
- `short`
- `byte`
- `double`
- `float`

These all are different in their own way. There is however one final type, known as `auto`. `auto` is the default type, this means that if no other type is specified, it'll default to it, and in most cases, auto typing should work for you.

Auto Resolution

Auto typing is resolved in a very basic manner. For lists it simply checks the type of the first value in the list, and for tags it checks the type of the value. Checking the type of a value is done as such:

- If it uses string syntax(“ … “) it is sure to be a string.
- If it uses object syntax(`(` … `)` it is sure to be an object.
- If it uses list syntax(`<` … `>` `[` … `]` or `[` … `]`) it is sure to be a list.
- If it is a numeric without a decimal point, it resolves to an integer
- If it is a numeric with a decimal point, it resolves to a float
- if it is one of the boolean constants, it resolves to a bool.

This means that types like double and short have to be manually typed.

Extensions

ODN supports extensions. An extension is effectively some extra thing added onto the language, that aren't part of the standard, but are useful to add extra functionality to a serializer/deserializer.

Extensions are specified in the document prefixed with a "#", and if an extension is not fully supported, the parser must error.

Extension names are always written in SCREAMING_SNAKE_CASE(that's how it's stylized)

Any extensions officially sanctioned by OpenAbility can be found in the OpenAbility extension listing(linked to at the bottom of the document), and are individually specified.

In case any extensions were to contain incompatibilities, the topmost extension in the document has priority. And in all cases, the official ODN spec should be treated with the least priority(it's at the bottom of the document), so that any extension may override ODN behavior.

Please note that there's one extension that is officially considered part of the specification, called "OA_EXT_NULL". OA_EXT_NULL does specify change anything, and is simply there in order to provide a basis for "The Test"(see p.12), so that it can require extension support.

A document may include the same extension multiple times, in which case only the first case of the inclusion should count, and the rest of them ignored.

The Test

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

Below is attached a piece of ODN that should successfully parse in any standard-compliant ODN parser(whether that'd be the original implementation or something but together for some hobby project, doesn't matter)

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

// New to 1.1, extensions!
#OA_EXT_NULL
# OA_EXT_NULL

(
    {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

)
```

Online Access to Files

A number of things mentioned in this specification are also available online, for example the ODN test, the results to said test, as well as the extension listing

- https://openability.tech/odn_files/1.1/ODN_TEST.odn - The test ODN provided in here
- https://openability.tech/odn_files/1.1/ODN_TEST_RESULT.txt - The expected test results
- https://openability.tech/odn_files/extensions - The ODN extension listing