# JO - JSON Object Language Reference Manual

## Team

[ { "Name" : "Abhinav Bajaj", "UNI" : "ab3900", "Role" : "System Architect" },

{ "Name" : "Arpit Gupta", "UNI" : "ag3418", "Role" : "Language Guru" },

{ "Name" : "Chase Larson", "UNI" : "col2107", "Role" : "Manager" },

{ "Name" : "Sriharsha Gundappa", "UNI" : "sg3163", "Role" : "Verification & Validation" } ]

# 1. Introduction

JSON or JavaScript Object Notation is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. It is used as lightweight data interchange format to transmit data between a server and web application. JSON is also emerging as a preferred format in "NoSQL" databases. While languages like Python and Java have libraries to handle JSON data, they are not a native aspect of the language. JSON is presently a data format, rather than something fundamental to the language, like the object of an object oriented language, or the function of a functional language. With rise of trends in Big Data, Internet of Things, No-SQL databases, we believe that our language can be provide a platform for building applications for these technologies with ease.

JO is simple yet powerful language to handle and manipulate JSON data. The language will treat JSON object as first class citizens and provide built-in functions that operate on these objects. These basic functions can be used to define complex libraries and applications like merging JSON, finding diff in JSON, SQL like queries on JSON objects. Our language attempts to facilitate any data operations by handling a lot of the business logic of handling JSON and their manipulations under the hood, and allowing the programmer to use JSON in a more native and intuitive way.

# 2. Lexical Convention

## 2.1 Comments

The characters /* introduce a multi-line comment, which terminates with the characters */. Multi-line comments cannot be nested within multi-line comments. Single line comments are also written in the same way as multi-line comments, with /* and */ appearing on the same line.

```
/* single line comments look like this */

/* this is

    how a multiple

    line comment looks like */


/* this however

  /* does not */

  works */
```

## 2.2 Tokens

A token is a string of ASCII characters that is always at least 1 character long. There are different types of tokens in JO. These are described below.

### 2.2.1 Identifiers

An identifier consists of a letter followed by other letters, digits and underscores. The letters are the ASCII characters a-z and A-Z. Digits are ASCII characters 0-9. The language is case sensitive.

*letter* → [‘a’-‘z’ ‘A’-‘Z’]

*digit* → [‘0’-‘9’]

*underscore* → ‘_’

*identifier* → letter (letter | digit | underscore)*

### 2.2.2 Keywords

Keywords are identifiers reserved by the language. Thus, they are not available for re-definition or overloading by users.

| Keyword | Description |
|---|---|
| Number, String, Bool, Json, List | Data types |
| True, False, Null | Literals |
| for, if , else, elsif , end | statement constructs |
| func, return | function declaration constructs |
| print type typestruct join read makeString | in-built functions |

### 2.2.3 Literals

Literals are expressions with fixed value. In the language there is capability for String, Number, Bool literals

*digit* → [‘0’-‘9’]

decimal → ‘.’

*String Literal* → (.)+

*Number Literal* → (digit)+ (decimal)? (digit)+

*Bool Literal* → True | False

### 2.2.4 Newlines

"EOL" is taken as a newline character.

### 2.2.5 Whitespace

Whitespace consists of any sequence of blank and tab characters. Whitespace is used to separate tokens and format programs. The compiler ignores all whitespace. As a result, indentations are insignificant.

# 3. Types and Type Inference

## 3.1 Primitive Types

There are four types of primitives in JO language. These are Bool, Number, String and Null.

### 3.1.1 Booleans

Bool type can either carry a value of true or false. Booleans are considered their own type, meaning that an expression that uses a boolean operator and a non-boolean variable will cause an error. For example -

```
a = true

If (a && 10) will cause error.
```

### 3.1.2 Numbers

A Number in JO is a double- precision floating-point format storing numbers in 64 bits, where the number (the fraction) is stored in bits 0 to 51, the exponent in bits 52 to 62, and the sign in bit 63:1.7E +/- 308 (15 digits). All numeric types will be stored as Number in JO. Hence, Number contains decimal part by default.

### 3.1.3 String

A string is a sequence of characters surrounded by double quotes ‘‘ ’’.

### 3.1.4 Null

null is an empty data type. For example -

```
a = null
```

## 3.2 Non-Primitive Types

There are two types of non-primitive types or complex data types in JO Language. They are explained below -

### 3.2.1 List

List is an ordered data type of primitive or complex data types. So a list can contain another list as one of its element along with JSON type as element and other Primitive types as elements in the list. Lists are enclosed in []. For example

```
["apple", 45, {"name":"harris"} ]
```

### 3.2.2 JSON

JSON or JavaScript Object Notation is an open standard format that uses human-readable text to transmit data objects consisting of attribute–value pairs. JSON is declared within curly braces {}.

For example -

```
{
"Name": { "First":"Arpit", "Last":"Gupta" },
"School": "Columbia",
"Age": 22,
"Courses": [ "PLT", "ML" ]
}
```

## 3.3 Type Inference

Data types in JO are expressed using a finite and well-defined set of data types. However when writing a JO program, the data types are not explicitly declared. In this sense JO is type-inferred language as opposed to dynamically typed language.

# 4.Operators

## 4.1 Type of Operators

### 4.1.1 Object Operators

| Operator | Description |
|---|---|
| + | Concatenation, works on any data type arguments, returns list<br>Example<br>1. 5 + 2 = [ 5 , 2 ]<br>2. [5, 2] + 3 = [5, 2, 3]<br>3. JsonA + JsonB = [JsonA, JsonB] |
| - | Usage : *A* - *B*, works when *A* is a Json or List<br>Removes attributes from A which matches with B<br>Valid Data types -<br>• Json - Json<br>• Json - String<br>• List - List<br>• List - String<br>• List - Json<br>• List - Number<br>Example<br>1. { {“name”: {first:chase, last:larson}}, {subject : “plt”}, marks : [2,3,4] }- { “name”: {first:chase, last:larson}, marks: [2,3,4]} = { subject : “plt”}<br>2. { “name”: {first:chase, last:larson}, subject : “plt”} - { “name”: {first:abhinav, last:larson}} = { “name”: {first:chase, last:larson}, subject : “plt”}<br>3. { {“name”: {first:chase, last:larson}}, marks: [2,3]} - "name" = {marks: [2,3]}<br>4. ["able", "barista", "carrie"] - ["barista", "carrie"] = ["able"]<br>5. ["able", "barista", "carrie"] - "barista" = ["able", "carrie"] |
| [] | 1. [ ] access values for attributes. only work on JSON objects<br>• a = json1[‘Name’] , returns the value at the attribute.<br>• json1[‘Name’] = ‘Arpit’ , stores value ‘Arpit’ for attribute ‘Name’<br>2. [ ] - constructs a new list |
| == | Compare two same data types. Returns true if their values match else false |
| != | Compare two same data types. Returns false if their values match else true |
| = | Assignment Operator |
| . | Calls function on an object |
| {} | Constructs a Json object |

### 4.1.2 Mathematical Operators

All Mathematical Operators are only valid for Type Number.

| Operator | Description | Example |
|---|---|---|
| ++ | Addition | 2 ++ 2 results in 4 |
| -- | Subtraction | 2 -- 2 results in 0 |
| ** | Multiplication | 2 ** 2 results in 4 |
| // | Division | 2 // 2 results in 1 |
| > | Greater Than | 2 > 1 results in true |
| < | Less Than | 2 < 1 results in false |
| %% | Modulo | 7 % 3 results in 1 |

### 4.1.3 String Operators

| Operator | Description | Example |
|---|---|---|
| ++ | String concatenation | "JS" ++ "ON" results in "JSON" |

### 4.1.4 Logical Operators

All Logical Operators only valid for Data type Bool.

| Operator | Description | Example |
|---|---|---|
| && | Logical And | if A and B are true, (A && B) is true |
| \|\| | Logical Or | if A or B are true, (A \|\| B) is true |
| ! | Logical Negation | if A is false, !A is true |

### 4.1.5 Membership Operators

| Operator | Description | Example |
|---|---|---|
| in | Results in true if variable is in given list | A in B: results in true if variable A is found in list B. |
| not in | Results in true if variable is not in given list | A not in B: results in true if variable A is not found in list B |

## 4.2 Operator Precedence

Operators are evaluated left to right.

The below sequence of operator is in decreasing order of precedence. The operators on the same line have same precedence.

. ( ) [] {}

%%, **, //

++, --

<, > , <=, >=

==, !=

&& , || , !

+, -

in, not in

=

# 5. Expressions

An expression contains at least one operand and zero or more operators that return a value. Operands are objects such as constants, variables, and functions.

Expressions are evaluated left to right. Parentheses can be used to group sub-expressions, with the innermost sub-expression being evaluated first. For example -

(2 ++ 2) // ( (3 -- 1) ** 2)

3 -- 1 is evaluated to 2 first. Then 2 ++ 2 and 2 ** 2 are both evaluated to 4. Finally 4 // 4 is evaluated resulting in 1.

## 5.1 Function Declaration

The declaration specifies the name of the function, list of parameters

The *func* keyword is used signify the declaration of a function. The general form is:

```
func <function_name> (<parameters>)
    <function_body>
   return <arg>
end
```

The keyword *func* must be followed by a space, with the *function name* following. Then the list of *parameters* must be on the same line and enclosed by parentheses. The parameters are separated by commas. Following the closing parentheses, a new line must start before the *function body*.

The *function body* is a series of statements that specifies what the function actually does. It must be on a new line following the function declaration. The function body must contain a *return* statement.

The *return statement* is the keyword *return* followed by the expression to be returned from the function.

For example:

```
func sum(x, y)
   return ( x ++ y)
end
```

## 5.2 Function Call

You call a function by using its name and supplying any required parameters. If no parameters are required by the function, the parenthesis is still required.

General Form:

<function_name> (<parameters>)

For example:

```
foo (5, A)
```

Here the function ‘foo’ is called with the parameters ‘5’ and ‘A’ (here ‘A’ is a variable that has been declared previously).

# 6. Statements

You write statements to cause action and to control flow within your program.

## 6.1 Assignment Statement

An assignment statement consists of a modifiable variable name followed by the assignment operator "=" followed by a valid expression.

For example -

```
myJson = { "name" : "john" }
```

## 6.2 Expression Statement

An expression statement executes the specified expression.

For example -

2 ++ 3

foo (2, 3)

Expression statements are useful when they have some sort of side effect, such as calling a function or storing a variable.

## 6.3 If...else… statement

The *if ... else* statement is used to execute part of your program only under certain conditions. The general form is:

if <condition>

  <then-statement>

else

  <else-statement>

end

or

if <condition>

  <then-statement>

end

The *then-statement* is executed if the condition evaluates to true. The else statement is optional, and it executes only if the condition evaluates to false.

The *condition* must be on the same line and immediately following the *if* keyword. The *condition* must be a boolean expression.

The *then statement* must start on a new line following the condition. If an *else* clause is included, the else statement must be on its own line and the *else statement* must be on a new line following the *else* keyword.

The entire statement must end with the *end* keyword.

## 6.4 For statement

The *for* statement is used to iterate over list, executing a block of code. The general form is:

for <iterative expression>

  <statement>

end

The iterative expression must be of the form:

<variable> in <List>

The *statement* is executed once for each element in the list. The iterative expression must be on the same line as the keyword *if*. The *statement* must be on a new line following the iterative expression. The statement must end with the keyword *end*.

For example -

```
for x in [1, 2, 3]
      y = x ++ 1
end
```

# 7. Scope

If an object or variable is declared within a function, if-else or a for statement block then it is only visible within that block. Otherwise, the object or variable is visible across the program.

For example, the following program results in an error:

```
for x in [1,2,3]
   if x > 1
      y = x
   end
   z = y
end
```

The variable *y* only has scope within the *if* block, because that is the smallest block within which it is declared. The NUMBER z cannot be assigned to the NUMBER y, because the program can't "see" the y outside of the *if* statement.

# 8. Built-in Functions

JO provides utility functions which assists in JSON manipulation. Below are the functions built-in the language.

## 8.1 Read

By using the read function one can read file from specified path. This file will be parsed and returned as a List of Literals. Individual elements in the file have to be comma separated. Each file is parsed into one List.

```
input = read(“path/to/file.txt”) /* returns a List */
/* It parses to list of
[5,”apple”,{“name”:”arpit”, “courses”:[“PLT”,”OS”]}, [3,4,”mike”]] */
```

Below is the file contents of file “path/to/file.txt”

```
5,”apple”,{“name”:”arpit”, “courses”:[“PLT”,”OS”]}, [3,4,”mike”]
```

## 8.2 Print

Print function either print to standard output or can write to a file. Each function call prints on a new line. Print can take any type as input. If the input is a JSON object then the print function outputs a pretty print of JSON object.

To print to a file, a second parameter for the file path needs to be specified.

```
/* Printing */
print(“Hello World”)
print(aJsonObject)
print(aJsonObject, “path/to/file.txt”)
```

## 8.3 Type

Type returns a String of the data type of a variable.

```
/* find datatype */
type(5) /*returns "Number"*/
type({}) /*returns Json */
```

## 8.4 TypeStruct

This is a utility function built inside JSON. It returns a String containing the data type of the attribute-values stored in JSON object.

```
/* Find Json attribute-value data types */

JSON myJSON = { "name": { "first": "chase", "last": "larson" },
"age" : 23 }

myJSON.typeStruct() /* returns String of  { String : { String:
String, String: String}, String: Number } */
```

## 8.5 Join

This is also a utility function built inside JSON. It can be used to combine JSONs having same attribute (key), say “name” and which have values as JSON containing same attribute (key), say “first”. Function returns a JSON with the values of JSON combined.

```
/* JSON Join example */

JSON a = { "name" : { "first" : "chase" } }

JSON b = { "name" : { "first" : "arpit" } }

a.join(b) /* returns { "name" : {"first": ["chase", "arpit"] }}  */
```

## 8.6 makeString

makeString takes any data type supported in JO as input and returns a String data type.

```
/* Convert to String */

makeString(5) /* returns “5” */

makeString(aJson)
```