# Json-C Lib

Json-C is the third-party library which is used to read/write JSON files. It’s written in C. It's **free and reliable**.

* Git: <https://github.com/json-c/json-c>
* Doxygen: <https://json-c.github.io/json-c/json-c-current-release/doc/html/files.html>

Note: Using Json-C requires both of its header files and static libs. So, you should **compile Json-C’s source code and install its libs** before writing your own client code.

There are many APIs provided by Json-C. But in most cases, we only need to use some. They're:

|  |  |  |
| --- | --- | --- |
| **Group** | **API** | **Usage** |
| Read file | json\_object\_from\_file | Read the full contents of the given file, then convert it to a root json\_object. |
| Write file | json\_object\_to\_file\_ext | Open and truncate the given file, creating it if necessary, then convert the json\_object to a string and write it to the file. |
| Close file | json\_object\_put | Decrement the reference count of json\_object and free if it reaches zero. |
| Get property value | json\_object\_object\_get\_ex | Get the json\_object associate with a given object field. |
| json\_object\_get\_type | Get the type of the json\_object.  typedef enum json\_type {  json\_type\_null,  json\_type\_boolean,  json\_type\_double,  json\_type\_int,  json\_type\_object,  json\_type\_array,  json\_type\_string  } json\_type; |
| json\_object\_get\_string | Get the string value of a json\_object. |
| json\_object\_get\_double | Get the double value of a json\_object. |
| json\_object\_get\_int | Get the int value of a json\_object. |
| json\_object\_get\_boolean | Get the boolean value of a json\_object. |
| Set property value | json\_object\_set\_string | Set the string value of a json\_object. |
| json\_object\_set\_double | Set the double value of a json\_object. |
| json\_object\_set\_int | Set the int32 value of a json\_object. |
| json\_object\_set\_int64 | Set the int64 value of a json\_object. |
| json\_object\_set\_boolean | Set the boolean value of a json\_object. |
| Create property | json\_object\_new\_string | Create a new empty json\_object of type json\_type\_string. |
| json\_object\_new\_double | Create a new empty json\_object of type json\_type\_double. |
| json\_object\_new\_int | Create a new empty json\_object of type json\_type\_int 32. |
| json\_object\_new\_int64 | Create a new empty json\_object of type json\_type\_int 64. |
| json\_object\_new\_boolean | Create a new empty json\_object of type json\_type\_boolean. |
| json\_object\_new\_object | Create a new empty object with a reference count of 1.  Note:   * object property = property with child properties. * string (or boolean, or double, or int) property = property without child properties.   For example:  {  info: { 🡸 This is an object property  name: "abc", 🡸 This is a string property  id: 123 🡸 This is an integer property  },  } |
| Add property | json\_object\_object\_add | Add an object field to a json\_object of type json\_type\_object.  Note: This method is mostly used after creating property. |
| Delete property | json\_object\_object\_del | Delete the given json\_object field. |
| Setting | json\_c\_set\_serialization\_double\_format | Set format for double numbers.  This function is required to prevent garbage digits. E.g.: Value to be set is 511.2, but value in JSON file after written is 511.19999999999999.  With this function, value in JSON file after written will be 511.2 (exactly what we set to). |

**Notes**:

* After reading a file, all of its properties are converted to objects and saved in memory. That means the **value of each property has a fixed data type** which cannot be changed. So, **when you get/set property value, make sure you use the correct method for each data type**. For example, it’s invalid to set a string value using json\_object\_set\_double; only a double value is valid.
* **Setting propery value** only affects the memory, so **make sure to write file to reflect changes to disk**. That means after one or several json\_object\_set\_, a json\_object\_to\_file\_ext is required to make these changes to the file.

But **getting property doesn’t require to re-read the file**. For example, you 1) get property A, then 2) set A with a new value, then 3) get A again. Between step 2 and 3, you missed re-reading the file. So, after step 3, you might think that A still has the old value at step 1. But actually, it’s updated with the lastest value. That’s because the whole content of the file is loaded into the memory; and getting/setting value is about modifying the memory.

# JsonParser Class

This is our self-defined class for handling parsing JSON files. It's like **a wrapper of the Json-C lib**.

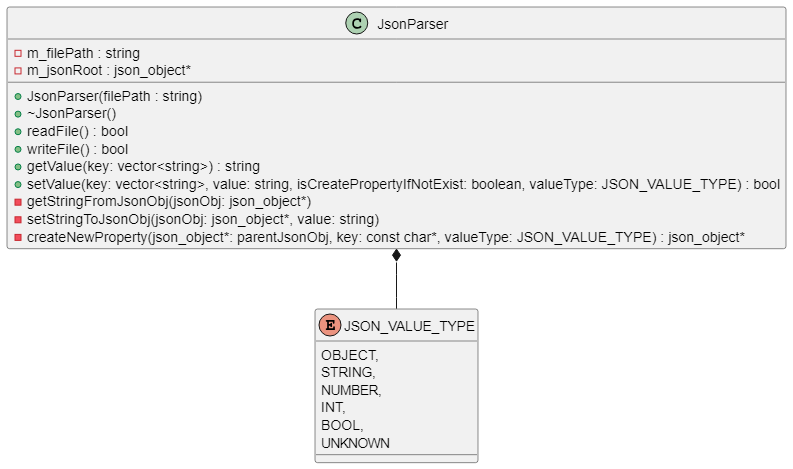
We'll design it like that:

## Class Diagrams

### Methods

|  |  |  |  |
| --- | --- | --- | --- |
| **Method** | **Access Modifier** | **Usage** | **Note** |
| readFile | public | Open and read JSON file. | The content is converted to objects and stored in a root json\_object. |
| writeFile | public | Write JSON file with the given root json\_object. |  |
| getValue | public | Get property value based on the given property key. | Despite the real data type (int, double, bool, string ...) of value, the returned valuet is always a string. |
| setValue | public | Set property value based on the given property key. | Despite the real data type (int, double, bool, string ...) of value, the inputed value is always a string.  The func will convert string to real data type when writing to JSON file. |
| createNewProperty | private | Create new property. | There is an abnormal case that some properties don't exist in JSON files. |
| getStringFromJsonObj | private | Get property value with its real data type from JSON object, then convert the value to string. |  |
| setStringToJsonObj | private | Set property value to JSON object. |  |

### Class Diagram



PlantUML script:

@startuml

enum JSON\_VALUE\_TYPE {

OBJECT,

STRING,

NUMBER,

INT,

BOOL,

UNKNOWN

}

class JsonParser {

- m\_filePath : string

- m\_jsonRoot : json\_object\*

+ JsonParser(filePath : string)

+ ~JsonParser()

+ readFile() : bool

+ writeFile() : bool

+ getValue(key: vector<string>) : string

+ setValue(key: vector<string>, value: string, isCreatePropertyIfNotExist: boolean, valueType: JSON\_VALUE\_TYPE) : bool

- getStringFromJsonObj(jsonObj: json\_object\*)

- setStringToJsonObj(jsonObj: json\_object\*, value: string)

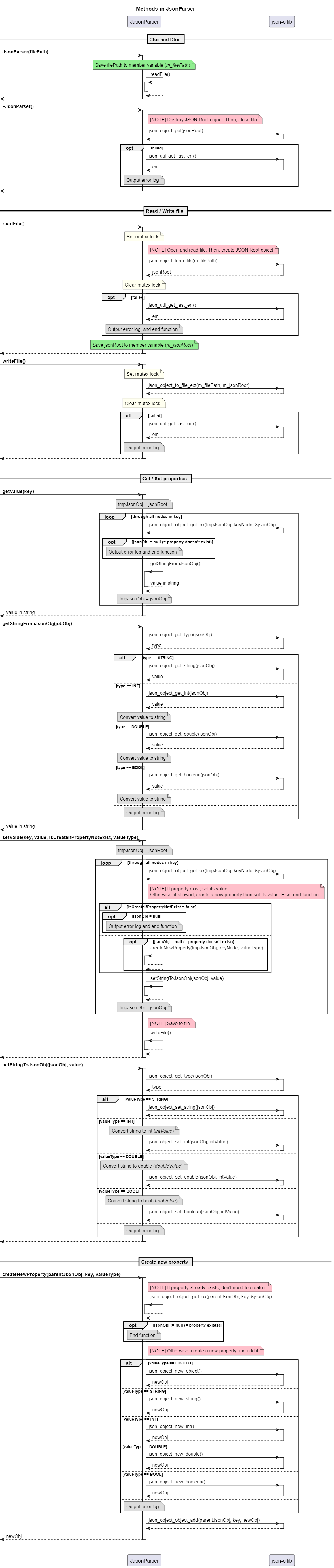
- createNewProperty(json\_object\*: parentJsonObj, key: const char\*, valueType: JSON\_VALUE\_TYPE) : json\_object\*

}

JsonParser \*-- JSON\_VALUE\_TYPE

@enduml

### Sequence Diagram



PlantUML script:

@startuml

skinparam NoteBackgroundColor #DDDDDD

skinparam NoteBorderColor #666666

skinparam BoxPadding 20

TITLE Methods in JsonParser

PARTICIPANT JasonParser AS jp

PARTICIPANT "json-c lib" AS lib

== Ctor and Dtor ==

-> jp++: \*\*JsonParser(filePath)\*\*

NOTE OVER jp #lightgreen

Save filePath to member variable (//m\_filePath//)

END NOTE

jp -> jp++: readFile()

RETURN

RETURN

-> jp++: \*\*~JsonParser()\*\*

NOTE RIGHT jp #pink: [NOTE] Destroy JSON Root object. Then, close file

jp -> lib++: json\_object\_put(jsonRoot)

RETURN

OPT failed

jp -> lib++: json\_util\_get\_last\_err()

RETURN err

NOTE OVER jp: Output error log

END

RETURN

|||

== Read / Write file ==

-> jp++: \*\*readFile()\*\*

NOTE OVER jp #ivory: Set mutex lock

NOTE RIGHT jp #pink: [NOTE] Open and read file. Then, create JSON Root object

jp -> lib++: json\_object\_from\_file(m\_filePath)

RETURN jsonRoot

NOTE OVER jp #ivory: Clear mutex lock

OPT failed

jp -> lib++: json\_util\_get\_last\_err()

RETURN err

NOTE OVER jp: Output error log, and end function

END

NOTE OVER jp #lightgreen

Save jsonRoot to member variable (//m\_jsonRoot//)

END NOTE

RETURN

-> jp++: \*\*writeFile()\*\*

NOTE OVER jp #ivory: Set mutex lock

jp -> lib++: json\_object\_to\_file\_ext(m\_filePath, m\_jsonRoot)

RETURN

NOTE OVER jp #ivory: Clear mutex lock

ALT failed

jp -> lib++: json\_util\_get\_last\_err()

RETURN err

NOTE OVER jp: Output error log

END

RETURN

|||

== Get / Set properties ==

-> jp++: \*\*getValue(key)\*\*

NOTE OVER jp: tmpJsonObj = jsonRoot

LOOP through all nodes in key

jp -> lib++: json\_object\_object\_get\_ex(tmpJsonObj, keyNode, &jsonObj)

RETURN

OPT jsonObj = null (= property doesn't exist)

NOTE OVER jp: Output error log and end function

END

jp -> jp++: getStringFromJsonObj()

RETURN value in string

NOTE OVER jp: tmpJsonObj = jsonObj

END

RETURN value in string

-> jp++: \*\*getStringFromJsonObj(jobObj)\*\*

jp -> lib++: json\_object\_get\_type(jsonObj)

RETURN type

ALT type == STRING

jp -> lib++: json\_object\_get\_string(jsonObj)

RETURN value

ELSE type == INT

jp -> lib++: json\_object\_get\_int(jsonObj)

RETURN value

NOTE OVER jp: Convert value to string

ELSE type == DOUBLE

jp -> lib++: json\_object\_get\_double(jsonObj)

RETURN value

NOTE OVER jp: Convert value to string

ELSE type == BOOL

jp -> lib++: json\_object\_get\_boolean(jsonObj)

RETURN value

NOTE OVER jp: Convert value to string

ELSE

NOTE OVER jp: Output error log

END

RETURN value in string

-> jp++: \*\*setValue(key, value, isCreateIfPropertyNotExist, valueType)\*\*

NOTE OVER jp: tmpJsonObj = jsonRoot

LOOP through all nodes in key

jp -> lib++: json\_object\_object\_get\_ex(tmpJsonObj, keyNode, &jsonObj)

RETURN

NOTE RIGHT jp #pink

[NOTE] If property exist, set its value.

Otherwise, if allowed, create a new property then set its value. Else, end function

END NOTE

ALT isCreateIfPropertyNotExist = false

OPT jsonObj = null

NOTE OVER jp: Output error log and end function

END

ELSE

OPT jsonObj = null (= property doesn't exist)

jp -> jp++: createNewProperty(tmpJsonObj, keyNode, valueType)

RETURN

END

END

jp -> jp++: setStringToJsonObj(jsonObj, value)

RETURN

NOTE OVER jp: tmpJsonObj = jsonObj

END

NOTE RIGHT jp #pink: [NOTE] Save to file

jp -> jp++: writeFile()

RETURN

RETURN

-> jp++: \*\*setStringToJsonObj(jsonObj, value)\*\*

jp -> lib++: json\_object\_get\_type(jsonObj)

RETURN type

ALT valueType == STRING

jp -> lib++: json\_object\_set\_string(jsonObj)

RETURN

ELSE valueType == INT

NOTE OVER jp: Convert string to int (//intValue//)

jp -> lib++: json\_object\_set\_int(jsonObj, intValue)

RETURN

ELSE valueType == DOUBLE

NOTE OVER jp: Convert string to double (//doubleValue//)

jp -> lib++: json\_object\_set\_double(jsonObj, intValue)

RETURN

ELSE valueType == BOOL

NOTE OVER jp: Convert string to bool (//boolValue//)

jp -> lib++: json\_object\_set\_boolean(jsonObj, intValue)

RETURN

ELSE

NOTE OVER jp: Output error log

END

RETURN

|||

== Create new property ==

-> jp++: \*\*createNewProperty(parentJsonObj, key, valueType)\*\*

NOTE RIGHT jp #pink: [NOTE] If property already exists, don't need to create it

jp -> jp++: json\_object\_object\_get\_ex(parentJsonObj, key, &jsonObj)

RETURN

OPT jsonObj != null (= property exists)

NOTE OVER jp: End function

END

NOTE RIGHT jp #pink: [NOTE] Otherwise, create a new property and add it

ALT valueType == OBJECT

jp -> lib++: json\_object\_new\_object()

RETURN newObj

ELSE valueType == STRING

jp -> lib++: json\_object\_new\_string()

RETURN newObj

ELSE valueType == INT

jp -> lib++: json\_object\_new\_int()

RETURN newObj

ELSE valueType == DOUBLE

jp -> lib++: json\_object\_new\_double()

RETURN newObj

ELSE valueType == BOOL

jp -> lib++: json\_object\_new\_boolean()

RETURN newObj

ELSE

NOTE OVER jp: Output error log

END

jp -> lib++: json\_object\_object\_add(parentJsonObj, key, newObj)

RETURN

RETURN newObj

@enduml

## Notes

### Abnormal Cases

When parsing JSON file, following abnormal cases can happen:

|  |  |
| --- | --- |
| **Abnormal case** | **Solution** |
| A JSON file is opening by **multiple threads** | Use mutex lock while reading/writing file. |
| **File doesn't exist**, or locked **permission**, or been **removed**, or **broken** | Output error log and do nothing. |
| **Key doesn't exist** in JSON file | Create new property/field for that key based on the given value type. |

# Example

Tutorials\C-C++\Code\_C++\JsonParser