ChatScript JSON Manual

Real World JSON

JSON (JavaScript Object Notation) is an open standard format using human-readable text to transmit data objects over the web. It is a common standard largely replacing XML which is too wordy and hard to read. JSON has two datatypes that represents collections of values, the array and the object. A JSON array is a list of JSON entities separated by commas and placed within `[]`, e.g.,

```
[ A 2 [ help life] [] ]
```

Indices of an array start at 0, so the above has as values: [0] = A [1] = 2 [2] = an array of 2 values [3] = an empty array

Note that arrays can hold values of different types (since really everything internally is a text type). The JSON types are array, object, number, string (enclosed in doublequotes), and primitives (text without doublequotes that cannot contain any whitespace). Array values are ordered and always retain that order. A JSON object is a list of key-value pairs separated by commas and placed within {}, e.g.,

```
{
    "key1": 1,
    "bob": "help",
    "1": 7,
    "array": [1 2 3],
    "object12": {}
}
```

Each key must be encased in quotes and joined to an ending colon. Whitespace separates the colon from the value. Again types can be mixed in the values. `{}` is the empty object. Key-value pairs have no guaranteed order and may shuffle around if you manipulate the structure. You can nest arrays and objects inside each other.

ChatScript & JSON

JSON is an excellent language to represent more complex ChatScript facts as well as interact with the web. ChatScript can convert back and forth between JSON the text string passed over the web and ChatScript facts that represent the structure internally. If you tried to create facts using CreateFact, you would

find making the data shown below extremely difficult and non-obvious. But as JSON, it is easy to create facts to represent the structure and to access pieces of it.

```
{
    "firstName": "John",
    "lastName": "Smith",
    "age": 25,
    "address": {
        "streetAddress": "21 2nd Street",
        "city": "New York",
        "state": "NY",
        "postalCode": "10021-3100"
    },
    "phoneNumbers": [
        {
        "type": "home",
        "number": "212 555-1234"
        },
        {
        "type": "office",
        "number": "646 555-4567"
        }
    ],
    "children": [],
    "spouse": null
}
```

Note that JSON has no mechanism for sharing JSON subtrees. Hence anytime you create a JSON fact structure in CS, the facts will all be unique.

JSONFLAGS - Optional 1st arg to some JSON routines

Routines that will create facts for JSON will by default create them as transients (they die at end of volley unless you work to save them). You can override this default by saying `permanent` or `transient`. This applies to `^jsonopen`, `^jsonparse`, `^jsoncreate`, `^jsonobjectinsert`, `^jsonarrayinsert`, `^jsoncopy`.

You can also add a flag `safe` to `^jsonparse`, `^jsonobjectinsert`, `^jsonarraydelete`. You can also add a flag `unique` to `^jsonarrayinsert`. You can also add `duplicate` to ^jsonobjectinsert.

When multiple flags are desired, put them into a simple string, `"DUPLICATE PERMANENT"`. Case doesn't matter.

When you want to add a reference to a normal factid (as is returned by ^createfact)

you can add the flag `AUTODELETE`

You can also assign user flags by listing `USER_FLAG1`, through `USER_FLAG4` as a flag as well. The JSON fact will have that flag on it, which you can use in conjunction with `^query` to limit what matches can be found.

`^jsonparse( {JSONFLAGS} string )`

`string` is a JSON text string (as might be returned from a website) and this parses into facts. It returns the name of the root node JSON composite. This name will look like this:

- `ja-0` – a json array numbered `0`.
- `jo-0` – a json object numbered `0`.

Note that the above names are true only for PERMANENT JSON objects. Transient ones will be named like `ja-t0` and `jo-t0`.

As new JSON composites are created during a volley, the numbers increase to keep them all distinct. JSON composites are all created as transient facts and will die at the end of the volley unless you do something to explictly keep them (typically `^jsongather` into some factset and then saving that OR using that to remove all their transient flags OR using `^delete()` to destroy the facts in the set).

If you are keeping JSON across volleys, you should use the optional Json flags argument to make sure numbering never collides (normally the numbers start back at 0 for each new volley).

JSON has stricter requirements on its format than CS does.

While CS will output strict JSON, you can input slack JSON. You do not need to put commas between elements of an array or object. And you do not need to put quotes around a key name. So the following is legal text form:

```
[a b {bob: 1 sue: 2}]
```

Formally JSON has specific primitives named true, false, and null, that systems may care about, but to CS they are nothing special. Numbers in JSON can be integer or have a decimal point and an exponent. CS does not currently support exponent notation, so you can only do: `1325` or `566.23`.

JSON originally required a JSON string be a composite of some kind. Now it allows any JSON type to be a JSON document. CS sticks to the original requirement, because passing around a single value as JSON is pretty useless.

Once you have a JSON fact structure, you can go the opposite direction and convert it back into a string or display it visually.

CS accepts extended JSON syntax for parsing into a json fact structure. Any place you have an object value, you can refer to a ChatScript user or match variable and that value will be substituted in. E.g.,

```
^jsonparse("{ a: $var, b: _0 }")
```

Note you should use a regular quoted string and not a function string like `^"{ a: $var, b: _0 }"`. If you use a function string, substitution will happen before calling ^jsonparse. Which might be a problem if you had something like this:

```
^jsonparse("{ a: $var, b: _0aba }").
```

where you wanted the value of `b` to be `"_0aba"`. Had you used an active string, the _0 would have been replaced with its contents.

Also, you can use json dereference operators to take apart an existing json structure and use values of it in the current one. If $_y points to a json structure, then

```
^jsonparse("{ a: $var, b: $_y.e[2] }")
```

would find a json object reference on `$_y`, get the e field, and get the 3rd array value found there. An initial argument of safe will locate the correct end of the data for json parsing, allowing you to pass excess data. This is important for passing in json data in OOB information. OOB information comes in `[]` and Json itself uses `[]` at times, so it would be impossible to find the correct end of pattern with normal pattern matching. Instead just do:

```
u: ( \[ _* ) and call ^jsonparse(safe _0)
```

This will figure out where the OOB marker actually is and only parse within it. You can add NOFAIL before the string argument, to tell it to return null but not fail if a dereference path fails cannot be found.

```
^jsonparse(transient NOFAIL "{ a: $var, b: _0.e[2] }")
```

Note: You cannot send JSON text as normal input from a user because CS tokenization is set to handle human input. So doing something like this as input will not work:

```
doLogin {"token": "myid", "accounts": ["whatever"]}
```

Instead you should call a website using `^jsonopen` which will automatically convert returned JSON data into internal CS data. Or you can pass JSON data on input via OOB notation:

```
[ {"token": "myid", "accounts": ["whatever"]}]  User message
```

and then have a pattern to grab the OOB data and call `jsonparse` with it (using SAFE as a parameter). OOB input is not subject to human tokenization behavior, spellchecking, etc.

Note: There is a limit to how much JSON you can pass as OOB data nominally, because it is considered a single token. You can bypass this limit by asking

the tokenizer to directly process OOB data, returning the JSON structure name instead of all the content. Just enable `#JSON_DIRECT_FROM_OOB` on the `$cs_token` value and if it finds OOB data that is entirely JSON, it will parse it and return something like `jo-t1` or `ja-t1` in its place. Eg. `[ { "key": "value} ]` will return tokenized as `[jo-t1]`.

Note: `^jsonparse` autoconverts backslash-unnnn into corresponding the utf8 characters.

`jsonformat`(string)

Because technically JSON requires you put quotes around field names (though various places ignore that requirement) and because CS doesn't, the function takes in a slack json text string and outputs a strict one.

Accessing JSON structures

`^jsonpath( string id )`

- `string` is a description of how to walk JSON.
- `id` is the name of the node you want to start at (typically returned from `^jsonopen` or `^jsonparse`).

Array values are accessed using typical array notation like `[3]` and object fields using dotted notation. A simple path access might look like this: `[1].id` which means take the root object passed as id, e.g., `ja-1`, get the 2nd index value (arrays are 0-based in JSON).

That value is expected to be an object, so return the value corresponding to the id field of that object. In more complex situations, the value of id might itself be an object or an array, which you could continue indexing like `[1].id.firstname`.

You can walk an array by using `[$_index]` and varying the value of `$_index`. When you access an array element, you have to quote the text because it consists of multiple tokens to CS which breaks off `[` and `]`. If you are just accesing an object field of something, you can quote the string or just type it direct

```
^jsonpath(.id $object2)
^jsonpath(".id" $object2)
```

Of course you don't always have to start at the root. If you know what you want is going to become object 7 eventually, you could directly say .id given `jo-7` and the system would locate that and get the id field. Likewise if you know that key names are somehow unique, you could query for them directly using

```
^query(direct_v ? verbkey ?)
```

Or even if the key is not unique you can restrict matches to facts having the `JSON_OBJECT_FACT` flag.

```
^query(directflag_v ? verbkey ? 1 ? ? ? JSON_OBJECT_FACT)
```

Be aware that when `^jsonpath` returns the value of an object key, when the value is a simple word, it just returns the word without doublequotes (since CS just stores information as a single word). But if the value contains whitespace, or JSON special characters, that may mess up if you pass it to `^JSONFormat`. You can get `^jsonpath` to return dangerous data as a string with double quotes around it if you add a 3rd argument "safe" to the call.

```
^jsonpath(".name" $_jsonobject safe)
```

`^jsonpath`

Can also return the actual factid of the match, instead of the object of the fact. This would allow you to see the index of a found array element, or the json object/array name involved. Or you could use ^revisefact to change the specific value of that fact (not creating a new fact). Just add * after your final path, eg

```
^jsonpath(.name* $$obj)
^jsonpath(.name[4]* $$obj)
```

Correspondingly, if you are trying to dump all keys and values of a JSON object, you could do a query like this:

```
@0 = ^query(direct_s $_jsonobject ? ?)
^loop()
{
    _0 = ^first(@0all)
    and then you have _1 and _2 as key and value
}
```

If you need to handle the full range of legal keys in json, you can use text string notation like this `^jsonpath(."st. helen".data $tmp)`.

You may omit the leading . of a path and CS will by default assume it

```
^jsonpath("st. helen".data $tmp)
```

Direct access via JSON variables `$myvar.field` and '$myvar[]

If a variable holds a JSON object value, you can directly set and get from fields of that object using dotted notation. This can be a fixed static fieldname you give or a variable value: `$myvar.$myfield` is legal.

Dotted notation is cleaner and faster than `^jsonpath` and `jsonobjectinsert` and for get, has the advantage that it never fails, it only returns null if it can't find the field. If the path does not contain a json object at a level below the top, one will automatically be created on assignment, and have the same transient/permanent property as the immediately containing object. If the top level variable is not currently an object, assignment will fail. CS will not create an object for you because it doesn't know if it should be transient or permanent.

```
$x = $$obj.name.value.data.side
$$obj.name.value.data.side = 7
```

Similarly you can access JSON arrays using array notation:

```
$x = $$array[5]
$x = $$array[$_tmp]
$$obj.name[4] += 3
```

```
$x.foo[] = Bruce
```

If foo is currently undefined, the system will create a JSON array for you, with permanency that matches the JSON object of $x. You cannot do $x[] and have this happen because at the top level the system does not know what permanency to use. Once there is a JSON array in $x.foo, assignments with `foo[]` will add elements to the array. You cannot designate the index, it will be the next index in succession.

The only restriction on arrays is that you cannot add a new array index value without using ^jsonarrayinsert as you are not allowed to create discontiguous indices.

NOTE JSON is normally a non-recursive structure with no shared pointers. But ChatScript allows you to store references to JSON structures in multiple places of other JSON structures. This has its hazards. It presents no problem when transcribing to text for a website using `^jsonwrite` . And when you have something like this:

```
$x = ^jsoncreate(object)
$y = ^jsoncreate(object)
$x.field = $y
$x.field1 = $y
$x.field = null
```

Assuming that a JSON structure is not available in multiple places, the assignment of null (or any other value) to a field that already has a JSON structure will normally cause the old value structure to be fully deleted, since it's only reference is removed. And the system does check and delete the structure if it is not referred to by some other JSON field. But there are limits. The system has no idea if you have a pointer to it in a variable. Or if it is part of a pathological indirection sequence like this:

```
$x = ^jsoncreate(object)
```

```
$y = ^jsoncreate(object)
$x.field = $y
$y.field = $x
$x.field = null
```

The two structures point to each other, each only once. So assigning null will kill off both structures.

Assigning `null` will remove a JSON key entirely. Assigning `"" ^""` will set the field to the JSON literal `null`.

`^length( jsonid )`

Returns the number of top-level members in a json array or object.

Printing JSON structures

`^jsonwrite( name )`

name is the name from a json fact set (either by `^jsonpart`, `^jsonopen`, orsome query into such structures). Result is the corresponding JSON string (as a website might emit), without any linefeeds.

`^jsontree( name {depth} )`

name is the value returned by `^jsonparse` or `^jsonopen` or some query into such structures. It displays a tree of elements, one per line, where depth is represented as more deeply indented. Objects are marked with `{}` as they are in JSON. Arrays are marked with `[]`.

The internal name of the composite is shown immediately next to its opening punctuation. Optional depth number restricts how deep it displays. 0 (default) means all. 1 is just top level.

JSON structure manipulation

You can build up a JSON structure without using `^jsonparse` if you want to build it piece by piece. And you can edit existing structures.

`^jsoncreate( {JSONFLAGS} type )`

Type is either array or object and a json composite with no content is created and its name returned. See `^jsonarrayinsert`, `^jsonobjectinsert`, and

`^jsondelete` for how to manipulate it. See writeup earlier about optional json flags.

`^jsonarrayinsert( {JSONFLAGS} arrayname value )`

Given the name of a json array and a value, it adds the value to the end of the array. See writeup earlier about optional json flags. If you use the flag unique then if value already exists in the array, no duplicate will be added.

`^jsonarraydelete( [INDEX, VALUE] arrayname value )`

This deletes a single entry from a JSON array. It does not damage the thing deleted, just its member in the array. * If the first argument is `INDEX`, then value is a number which is the array index (0 ... n-1). * If the first argument is `VALUE`, then value is the value to find and remove as the object of the json fact.

You can delete every matching `VALUE` entry by adding the optional argument `ALL`. Like: `^jsonarraydelete("INDEX ALL" $array 4)`

If there are numbered elements after this one, then those elements immediately renumber downwards so that the array indexing range is contiguous.

If the key has an existing value then if the value is a json object it will be recursively deleted provided its data is not referenced by some other fact (not by any variables). You can suppress this with the `SAFE` flag. `^jsonarraydelete(SAFE $obj $key)`.

`^jsonarraysize( name )`

deprecated in favor of ^length

`^jsoncopy( name )`

Given the name of a json structure, makes a duplicate of it. If it is not the name of a json structure, it merely returns what you pass it.

`^jsonobjectinsert( {JSONFLAGS} objectname key value )`

inserts the key value pair into the object named. The key does not require quoting. Inserting a json string as value requires a quoted string. Duplicate keys are ignored unless the optional 1st argument `DUPLICATE` is given. See writeup earlier about optional json flags.

If the key has an existing value and `DUPLICATE` is not a factor, then if the value is a json object it will be recursively deleted provided its data is not referenced by some other fact (not by any variables). You can suppress this with the `SAFE` flag. `jsonobjectinsert(SAFE $obj $key null)`.

`^jsondelete( factid )`

deprecated in favor of ^delete

`^jsongather( {fact-set} jsonid {level})`

takes the facts involved in the json data (as returned by `^jsonparse` or `^jsonopen`) and stores them in the named factset.
This allows you to remove their transient flags or save them in the users permanent data file.

You can omit fact-set as an argument if you are using an assignment statement: `@1 = ^jsongather(jsonid)`

`^Jsongather` normally gathers all levels of the data recursively. You can limit how far down it goes by supplying `level`. Level 0 is all. Level 1 is the top level of data. Etc.

`^jsonlabel( label )`

assigns a text sequence to add to jo- and ja- items created thereafter. See System functions manual.

`^jsonreadcsv( TAB filepath {'^fn})`

reads a tsv (tab delimited spreadsheet file) and returns a JSON array representing it. The lines are all objects in an array. The line is an object where non-empty fields are given as field indexes. The first field is 0. Empty fields are skipped over and their number omitted.

If an optional 3rd parameter of a function name is given, the code does not create a JSON structure to return. Instead it calls the function with each field of a line being an argument. This is sort of analogous to `:document` mode in that you can potentially read large amounts of data in a single volley and may need to use `^memorymark` and `^memoryfree` to manage the issue.

^jsonundecodestring(string)

removes all json escape markers back to normal for possible printout to a user. This translates \n to newline, \r to carriage return, \t to tab, and \" to a simple quote.

WEB JSON

^jsonopen({JSONFLAGS} kind url postdata header {timeout})

this function queries a website and returns a JSON datastructure as facts. It uses the standard CURL library, so it's arguments and how to use them are generally defined by CURL documentation and the website you intend to access. See writeup earlier about optional json flag.

parameter	description
`kind`	is `POST`, `GET`, `POSTU`, `GETU`, `PUT`, `DELETE` corresponding to the usual meanings of Get and Post and url-encoded forms.
`url`	is the url to query
`postdata`	is either "" if this is not a post or is the data to send as post or put
`header`	is any needed extra request headers or "". Multiple header entries must be separated by a tilde
`timeout`	optional seconds limitation for connection and then for transfer. else $cs_jsontimeout rules

Note: 'postdata' can be a simple JSON structure name, in which case the system will automatically perform a ^jsonwrite on it and send that text data as the data. Currently limited to 500K in size of the internal buffer.

A sample call might be:

```
$$url = "https://api.github.com/users/test/repos"
$$user_agent = ^"User-Agent: Mozilla/5.0 (compatible; MSIE 10.0; Windows NT 6.2; WOW64; Tric

^jsonopen(GET $$url "" $$user_agent)
```

where GitHub requires user-agent data. As an example of a complex header value you might create neatly,

```
$header = ^"Authorization: 8daWs-dwQPpXkuzJOOo
~Accept: application/json
~Accept-Encoding: identity,*;q=0
```

```
~Accept-Language: en-US,en;q=0.5
~Cache-Control: no-cache
~Connection: close
~Host: Chatscript
~User_Agent: Mozilla/5.0 (Windows NT 10.0; WOW64; rv:42.0) "
```

ChatScript will make each line have a single space separator between line continuations. And JsonOpen will correctly get the header elements that do not include that spacing.

The results are a collection of facts representing the JSON parse tree and the value of the function is the root JSON value. The JSON elements that can be represented are arrays, objects, JSON strings, and primitives (numbers, true, false, null). JSON arrays are named `ja-n` where `n` is a unique index. JSON objects are similarly named `jo-n`.

Unlike JSON, which makes a distinction between primitives and strings, in ChatScript those things are all strings and are not quoted.So a JSON string like this:

```
[ {"id": 1 "value": "hello"} {"id": 2 "value": "bye"} ]
```

returns this value: `ja-1` and these facts. The facts have flags on them which you can use in queries. You may not have any need to use these flags, so maybe you will just ignore their existence.

fact	associated flags
(ja-1 0 jo-1)	#JSON_ARRAY_FACT #JSON_OBJECT_VALUE
(jo-1 id 1)	#JSON_OBJECT_FACT #JSON_PRIMITIVE_VALUE
(jo-1 value hello)	#JSON_OBJECT_FACT #JSON_STRING_VALUE
(ja-1 1 jo-2)	#JSON_ARRAY_FACT #JSON_OBJECT_VALUE
(jo-2 id 2)	#JSON_OBJECT_FACT #JSON_PRIMITIVE_VALUE
(jo-2 value bye)	#JSON_OBJECT_FACT #JSON_STRING_VALUE

Using queries, you could get all values of an array. Or all fields of an object. Or all JSON facts where the field is the id. You could manually write script to walk the entire tree. But more likely you will use `^jsonpath` to retrieve specific pieces of data you want. For example, you could do a query on the value returned by `^jsonopen` as subject, to find out how many array elements there are. Then use `^jsonpath` to walk each array element to retrieve a specific field buried within.

Note - for things like primitive null, null arrays, null strings, null objects, these are represented as “null” and the corresponding fact flag tells you the kind of value it is.

You can also ask CS to show those out visually using `^jsontree`.

Note that the facts created are all transient and disappear at the end of the

volley unless you have forced them to stay via `permanent`. Forcing them to stay is generally a bad idea because it will congest your user topic data file, slowing it down or exceeding its capacity, and because those facts may then collide with new facts created by a new `^jsonopen` on a new volley. The array and object ids are cleared at each volley, so you will be reusing the same names on new unrelated facts.

Using the flag values, it is entirely possible to reconstruct the original JSON from the facts (if the root is an array or object because otherwise there are no facts involved), but I can't think of use cases at present where you might want to. You cannot compile CS on LINUX unless you have installed the CURL library. For Amazon machines that means doing this:

```
sudo yum -y install libcurl libcurl-devel
```

On some other machines that doesn't install library stuff and maybe you need

```
sudo apt-get install libcurl3 libcurl3-gnutls libcurl4-openssl-dev
```

System variables `%httpresponse` will hold the most recent http return code from calling `^jsonopen`.

If you call `^jsonopen(direct ...` then the result will not be facts, but the text will be directly shipped back as the answer. Be wary of doing this if the result will be large (>30K?) since you will overflow your buffer without being checked.

^JSONopen automatically url-encodes headers and urls

JSONOpen and proxy servers

If you need JSONOpen to run thru a proxy server, these are the CS variables you need to set up: `$cs_proxycredentials` should be your login data, e.g. `myname:thesecret`. `$cs_proxyserver' is the server address, e.g.,`http://local.example.com:1080‘. ‘$cs_proxymethod’ are bits listing the authorization method to use. They come from the LIBCURL so you should OR together the bits you want. Bit 1 is the most basic choice of name and password. Read- https://curl.haxx.se/libcurl/c/CURLOPT_HTTPAUTH.html

JSON & Out-of-band output data

Out-of-band data in ChatScript is signaled by the output beginning with data enclosed in `[]`. Which might be confusing, since JSON uses `[]` to denote an array. Standalone ChatScript contains a built-in handler for OOB data and if you pass it JSON at the start of output, it will swallow it and not display it (unless you turn on OOB display).

Similarly, std webpage interfaces connecting to ChatScript do likewise. So if you want to see this information, you should put something in the output at the start which is NOT the JSON data. Anything will do. The only time you might

actually need the JSON clean at the beginning is from some special purpose application, and in that case you will write your own OOB handler anyway (or not have one).

JSON & Out-of-band input data

OOB data into ChatScript is similarly signaled by being at the start of input, with data enclosed in [], followed typically by the user's actual input. The ChatScript engine reacts specially to OOB incoming data in that it will be careful to not treat it like ordinary user chat. Tokenization is done uniquely, spell-checking, pos-tagging, parsing, named entity merging etc are all turned off and the data becomes its own sentence (the user's actual input generates more sentences to CS as input). OOB data is then processed by your script in any way you want. So one clever thing you can do is pass in JSON data within the OOB to get temporary facts into your app during a volley. Input might look like this:

```
[ [ a b { "bob": 1, "suzy": 2 } ] ] What is your name?
```

You can pattern match the oob section of the input as follows:

```
u: ( \[ _* ) $_tmp = ^jsonparse('_0)
```

`_0` will contain an excess right bracket (the end of the oob message), but that won't bother `^jsonparse`.

Representing JSON in CS facts is more than just a bunch of subject-verb-object facts linked together.

The facts have typing bits on them that describe the structure and arrays have index values that must remain consistent. Therefore you should not create and alter JSON fact structures using ordinary CS fact routines like `^createfact` and `^delete`. Instead use the JSON routines provided.

Practical Examples

Objects

The write jsonwrite and json tree print out different views of the same data..

```
u: (-testcase1) $_jsonObject = ^jsoncreate(object)
    ^jsonobjectinsert( $_jsonObject name "some name" )
    ^jsonobjectinsert( $_jsonObject phone "some number" )
    ^jsonwrite ( $_jsonObject ) \n
    ^jsontree ( $_jsonObject )\n
```

Note in this next example how to escpe a json string with ^". This makes creating json objects from static data very intuitive and clear.

```
u: (-testcase2) $_tmp = ^jsonparse( ^'{name: "Todd Kuebler", phone: "555.1212"}' )
    ^jsonwrite( $_tmp ) \n
    ^jsontree( $_tmp ) \n
    name: $_tmp.name, phone: $_tmp.phone
```

This example shows the . notation access of data inside an json object in chatscript. This is probably the most intuitive way of interacting with the data.

```
u: (-testcase3) $_tmp = ^jsoncreate(object)
    $_tmp.name = "Todd Kuebler"
    $_tmp.phone = "555-1212"
    ^jsonwrite( $_tmp ) \n
    ^jsontree( $_tmp ) \n
    name: $_tmp.name, phone: $_tmp.phone
```

Arrays of objects

In the example below, we add two items into an array of objects and we display the formatted array:

```
u: ( testcase4 )
    # create a phoneBook as an array of structured items (objects)
    $_phoneBook = ^jsoncreate(array)

    #
    # add first object in the array
    #
    $_item = ^jsoncreate(object)

    # assign values
    $_item.name = "Todd Kuebler"
    $_item.phone = "555-1212"

    ^jsonarrayinsert($_phoneBook $_item)

    #
    # add a second object in the array
    #
    $_item = ^jsoncreate(object)

    # assign values
    $_item.name = "Giorgio Robino"
    $_item.phone = "111-123456789"
```

```
^jsonarrayinsert($_phoneBook $_item)

# display JSON tree
^jsontree( $_phoneBook ) \n

#
# print formatted items in the phone book
#
phone book:\n
$_i = 0
$_size = ^length($_phoneBook)
loop($_size)
{
  # print out formatted item
  name: $_phoneBook[$_i].name, phone: $_phoneBook[$_i].phone\n
  $_i += 1
}
```