One more ABAP to JSON Serializer and Deserializer

Author: Alexey Arseniev
Submitted: 20.11.2013
Other code samples from me:
- Dynamic Data Accessor Helper Class for ABAP
- Number base conversion in ABAP

Why
There are a lot of other implementations of the ABAP to JSON Serializer and Deserializer in SDN, but for different reasons, all implementations I have found were not suitable for my needs. From SAP_BASIS 7.40 there is also a simple transformation available for converting ABAP to JSON and JSON to ABAP. It is the best choice if you need maximal performance and does not care about serialization format, but for proper handling of ABAP types and name pretty printing, it fits bad.

So, I have written my own ABAP JSON serializer and ABAP JSON deserializer which has some key differentiates from other implementation.

Below you can find a snippet of the ABAP JSON class I wrote, that you can use as a local class or global renamed.

An original and actual version of the source can be found in class /UI2/CL_JSON delivered with UI2 Add-on (can be applied to SAP_BASIS 700 – 76X). So, you can use this ABAP JSON parser in your standard code mostly on any system.

What it can

**ABAP to JSON**

- Serialize classes, structures, internal tables, class and data references, any kind of elementary types. Complex types, as a table of structures/classes, classes with complex attributes, etc. are also supported and recursively processed.
- **ABAP to JavaScript** adopted way of data type serializations:
  - strings, character types to JavaScript string format (no length limitation),
  - ABAP_BOOL / BOOLEAN / XFELD / BOOLE_D to JavaScript Boolean,
  - Built in TRIBOOL (TRUE/FALSE/UNDEFINED = 'X'/'-'/'') support, for better control of initial values when serializing into JavaScript
  - int/floats/numeric/packed to JavaScript Integers/floats,
  - date/time to JavaScript date/time string representation as “2015-03-24” or “15:30:48”,
  - timestamp to JavaScript integer or to ISO8601 string
  - structures to JavaScript objects (include types are also supported; aliases => AS are ignored)
- **convert ABAP internal table to JSON**, e.g JavaScript arrays or associative arrays (objects)
- Support of conversion exits on ABAP data serialization
- Pretty Printing of JavaScript property names: `MY_DATA` -> `myData`, `/SAPAPO/MY_DATA` -> `sapapoMyData`
- Condensing of default values: initial values are not rendered into the resulting JSON string
- Performance is optimized for processing big internal tables with structures

**JSON to ABAP**

- Deserialize JSON objects, arrays and any elementary types into corresponding ABAP structures. Complex objects, with embedded arrays and objects with any level of nesting, are also supported.
- **Convert JSON to an internal table**
- Generic deserialization of JSON objects into reference data types:
  - as simple data types (integer, boolean or string into generic data reference (REF TO DATA) -> ABAP type is selected based on JSON type.
  - as dynamically generated complex object (structures, tables, mixed) for initial REF TO DATA fields
  - as typed references for prefilled REF TO DATA fields (you assign a reference to typed empty data object to REF TO DATA field in execution time)
- **Deserialization of unknown JSON structures possible using method GENERATE into on the fly created data types**
- On JSON to ABAP transformation following rules are used:
  - objects parsed into corresponding ABAP structures, classes (only classes with constructors with no obligatory parameters are supported) or to internal hash/sorted tables
  - arrays converted to internal tables (complex tables are also supported).
  - Boolean converted as ABAP_BOOL ("" or 'X')
  - Date/Time/Timestamps from JSON converted based on the type of corresponding ABAP element
  - integers/floats/strings moved to corresponding fields using ABAP move semantic (strings are un-escaped). There is no limit on the size of deserialized strings, the only restriction is the constraints of receiving data type.
elementary data types are converted if do not match: JavaScript integer can come into ABAP string or JavaScript string into ABAP integer and etc.
Transformation takes into account property naming guidelines for JSON and ABAP so that camelCase names will be copied into corresponding CAMEL_CASE field if CAMELCASE field is not found in ABAP structure. Do not forget to use the same PRETTY_MODE for deserialization, as you have used for serialization.
Default field values, specified in reference ABAP variable are preserved, and not overwritten if not found in the JSON object
Transformation of JSON structures into ABAP class instances is NOT supported.
Support of conversion exits on deserialization

Parser for serialize/deserialize uses single-pass parsing and optimized to provide the best possible performance in ABAP in release independent way. But for time-critical applications, which have kernel version 7.20 and higher, it is recommended to use built-in JSON to ABAP transformation s (CALL TRANSFORMATION). If transformation for some reason does not work, please assist the following notes: 1650141 and 1648418.

Usage example

```
DATA: lt_flight TYPE STANDARD TABLE OF sflight,
lrf_descr TYPE REF TO cl_abap_typedescr,
 lv_json   TYPE string.

SELECT * FROM sflight INTO TABLE lt_flight.

* serialize table lt_flight into JSON, skipping initial fields and
  converting ABAP field names into camelCase
lv_json = /ui2/cl_json=>serialize( data = lt_flight compress = abap_true
  pretty_name = /ui2/cl_json=>pretty_mode-camel_case ).
WRITE / lv_json.
CLEAR lt_flight.

* deserialize JSON string json into internal table lt_flight doing
  camelCase to ABAP like field name mapping
/u2/cl_json=>deserialize( EXPORTING json = lv_json pretty_name =
  /ui2/cl_json=>pretty_mode-camel_case CHANGING data = lt_flight ).

* serialize ABAP object into JSON string
lrf_descr = cl_abap_typedescr=>describe_by_data( lt_flight ).
 lv_json = /ui2/cl_json=>serialize( lrf_descr ).
WRITE / lv_json.
```

Output

```
["mandt":"000","carrid":"AA","connid":0017,"fldate":20130515,"price":422.94,"currency":"USD","planetype":"
747-400","seatsmax":385,
  "seatsocc":192683.30,"seatsmaxB":31,"seatsoccB":31,"seatsmaxF":21,"seatsoccF":19],{"mandt":
  "000","carrid":"AA",
  "connid....
  {"ABSOLUTE_NAME":"\TYPE=%_T00004S00000000O0000012480","ADMIN_TAB":"\TYPE=%_T00004S00000000O0000012480",
  "ADMIN_TAB_LINE":"\TYPE=%_T00004S00000000O0000012480","DECIMALS":0,"HAS_UNIQUE_KEY":false,"INITIAL_SIZE":0
  ,"KEY":{""MANDT"},{"NAME":"CARRID"},{"NAME":"CO....
```

API description

There are two static methods that are of most interest in common cases: SERIALIZE and DESERIALIZE. The rest of the public methods are done
public only for reuse purpose if you would like to build/extend your own serialization/deserialization code.

**SERIALIZE**: Serialize ABAP object into JSON

- **DATA** (any) - any ABAP object/structure/table/element to be serialized
- **COMPRESS** (bool, default=false) - tells serializer to skip empty elements/objects during serialization. So, all for which IS INITIAL = TRUE.
- **NAME** (string, optional) - optional name of the serialized object. Will "name" : {...} instead of '{...}' if supplied.
- **PRETTY_NAME** (enum, optional) - mode, controlling how ABAP field names transformed in JSON attribute names. See description below.
- **TYPE_DESCR** (ref to CL_ABAP_TYPEDESCR, optional) - if you know object type already - pass it to improve performance.
- **ASSOC ARRAYS** (bool, default = false) - controls how to serialize hash or sorted tables with unique keys. See below for details.
- **ASSOC ARRAYS_OPT** (bool, default = false) - when set, serializer will optimize rendering of name-value associated arrays (hash maps) in JSON
- **TS_AS_ISO8601** (bool, default = false) - says serializer to output timestamps using ISO8601 format.
- **NUMC_AS_STRING** (bool, default = false) - Controls the way how NUMC fields are serialized. If set to ABAP_TRUE, NUMC fields serialized not as integers, but as strings, with all leading zeroes. Deserialization works compatibly with both ways of NUMC serialized data.
- **CONVERSION_EXITS** (bool, default = false) - use DDIC conversion exits on serialize of values (performance lost!)
- **R JSON** - output JSON string.

**DESERIALIZE**: Deserialize ABAP object from JSON string

- **JSON** (string) - input JSON object string to deserialize
- **PRETTY_NAME** (enum, optional) - mode, controlling how JSON field names mapped to ABAP component names. See description below.
- **ASSOC ARRAYS** (bool, default = false) - controls how to deserialize JSON objects into hash or sorted tables with unique keys. See below for details.
- **ASSOC ARRAYS_OPT** (bool, default = false) - when set, the deserializer will take into account optimized rendering of associated arrays (properties) in JSON.
- **TS_AS_ISO8601** (bool, default = false) - says deserializer to read timestamps from strings into timestamps fields using ISO8601 format.
- **CONVERSION_EXITS** (bool, default = false) - use DDIC conversion exits on deserialize of values (performance lost!)
- **DATA** (any) - ABAP object/structure/table/element to be filled from JSON string. If ABAP structure contains more fields than in JSON object, a content of unmatched fields is preserved.
- **R JSON** - output JSON string.

**GENERATE**: Generates ABAP object from JSON

- **JSON** (string) - input JSON object string to deserialize
- **PRETTY_NAME** (enum, optional) - mode, controlling how JSON field names mapped to ABAP component names. See description below.
- **RR_DATA** (REF TO DATA) - reference to ABAP structure/table dynamically generated from JSON string.

In addition to the explained methods, there are two options, that need a wider explanation:

**PRETTY_NAME** : enumeration of modes, defined as constant /UI2/CL_JSON=>pretty_name.

- **NONE** - ABAP component names serialized as is (UPPERCASE).
- **LOW_CASE** - ABAP component names serialized in low case
- **CAMEL_CASE** - ABAP component types serialized in CamelCase where symbol "_" treated as word separator (and removed from the resulting name).
- **EXTENDED** - works the same way as CAMEL_CASE but in addition, has extended logic for encoding special characters, as: ".", "@", "~", etc. Shall be used if you need JSON names with characters not allowed for ABAP data component names. Do not use it, if you do not have special characters in JSON names - the performance would be slower in comparison with CAMEL_CASE mode. Example: ABAP name ‘__A__SCHEMA’ translates in JSON name ‘@schema’

Encoding rules (ABAP name JSON name):

- E - ‘!’
- N - ‘’
- D - ‘$’
- P - ‘%’
- M - ‘&’
- S - ‘’
- H - ‘’
- T - ‘’
- L - ‘’
- C - ‘’
- V - ‘’
- A - ‘@’
- O - ‘’ or ‘‘’

NONE and LOW_CASE works the same way for DESERIALIZE.

**ASSOC ARRAYS**: 


This option controls the way how hashed or sorted tables with unique keys serialized/deserialized. Normally, ABAP internal tables serialized into JSON arrays, but in some case, you will like to serialize them as associated arrays (JSON object) where every row of the table shall be reflected as a separated property of JSON object. This can be achieved by setting the ASSOC_ARRAYS parameter to TRUE. If set, serializer checks for sorted[hashed] tables with a UNIQUE key(s) and serialize them as an object. The JSON property name, reflecting row, constructed from values of fields, used in key separated by constant MC_KEY_SEPARATOR = ".". If the table has only one field marked as key, the value of this single field become a property name and REMOVED from the associated object (to eliminate redundancy). If TABLE_LINE used as a unique key, all values of all fields construct key property name (separated by MC_KEY_SEPARATOR). During deserialization, logic works vice versa: if ASSOC_ARRAYS set to TRUE, and JSON object matches internal hash or sorted table with the unique key, the object is transformed into the table, where every object property reflected in a separated table row. If the ABAP table has only one key field, property name transformed into a value of this key field.

**ASSOC_ARRAYS_OPT:**

By default, when dumping hash/sorted tables with a unique key into JSON, the serializer will write key field as property name and rest of the fields will write object value of properties:

**Dumping of hash tables from ABAP to JSON**

```abap
TYPES: BEGIN OF ts_record,
  key TYPE string,
  value TYPE string,
END OF ts_record.

DATA: lt_act TYPE SORTED TABLE OF ts_record WITH UNIQUE KEY key.
lv_json = /ui2/cl_json=>serialize( data = lt_act assoc_arrays = abap_true).
```

**Output JSON**

```
{
  "KEY1": {
    "value": "VALUE1"
  },
  "KEY2": {
    "value": "VALUE2"
  }
}
```

But if you will use assoc_arrays_opt flag during serialization, the serializer will try to omit unnecessary object nesting on dumping of simple, name/value tables, containing only one key field and one value field:

**Dumping of hash tables from ABAP to JSON**

```
lv_json = /ui2/cl_json=>serialize( data = lt_act assoc_arrays = abap_true assoc_arrays_opt = abap_true).
```

**Output JSON**

```
{
  "KEY1": "VALUE1",
  "KEY2": "VALUE2"
}
```

For deserialization, the flag is used to tell the deserializer that value shall be placed in a non-key field of the structure.
Supported SAP_BASIS releases

The code was tested from SAP_BASIS 7.00 and higher, but I do not see the reasons why it cannot be downported on lower releases too. But if you plan to use it on SAP_BASIS 7.02 and higher (and do not need property name pretty printing) better consider the standard solution for ABAP, using CALL TRANSFORMATION. It shall be definitely faster, while implemented in the kernel. See the blog of Horst Keller for details. Maybe the best will be, if you need support in lower SAP_BASIS releases as well as in 7.02 and higher, to modified provided a class in a way to generate same JSON format as standard ABAP CALL TRANSFORMATION for JSON does and redirect flow to home-made code or built-in ABAP transformation depending on SAP_BASIS release.

Related notes

See full version history below.

Further optimizations

- Be aware, that usage of flag conversion_exits may significantly decrease performance - use only in cases, when you are sure that you need it.
- Escaping of property values can be expensive. To optimize performance, in this case, you can replace escapement code by some kernel implemented function (from cl_http_utility class for example), instead of explicit REPLACE ALL OCCURRENCES calls.
- It is possible to significantly increase performance for serialization/deserialization by dropping support of releases below 7.40. That can be realized by moving base parsing from ABAP to kernel implemented classes cl_sxml_string_writer and cl_sxml_string_reader.

Remarks

Due to optimization reasons, some methods were converted to macros, to reduce overhead for calling methods for data type serialization. If performance in your case is not critical, and you prefer clean/debuggable code you can replace macro calls by corresponding methods.

The /UI2/CL_JSON code

Below you can find the code itself, you can use.

If you want to use the class globally, I suggest to create a proxy class, in your own namespace, with a reduced interface (serialize/deserialize only) and call local copy (local class of it) of /UI2/CL_JSON. Then you can easily update to new version of /UI2/CL_JSON from SDN or call UI Addon implementation if it is installed.

<table>
<thead>
<tr>
<th><strong>ZCL_JSON code</strong></th>
</tr>
</thead>
</table>
| *
| CLASS zcl_json DEFINITION
| *
| CLASS zcl_json DEFINITION.
| PUBLIC SECTION.
| TYPE-POOLS abap .
| CLASS cx_sy_conversion_error DEFINITION LOAD .
| TYPES:
| BEGIN OF name_mapping,
|   abap TYPE abap_compname,
|   json TYPE string,
| END OF name_mapping .
| TYPES:
| name_mappings TYPE HASHED TABLE OF name_mapping WITH UNIQUE KEY abap . |
TYPES json TYPE string.
TYPES bool TYPE char1.
TYPES tribool TYPE char1.
TYPES pretty_name_mode TYPE char1.

CONSTANTS:
  BEGIN OF pretty_mode,
    none      TYPE char1 VALUE ``,
    low_case  TYPE char1 VALUE `L`,
    camel_case TYPE char1 VALUE `X`,
    extended  TYPE char1 VALUE `Y`,
    user      TYPE char1 VALUE `U`,
    user_low_case TYPE char1 VALUE `C`,
  END OF pretty_mode.

CONSTANTS:
  BEGIN OF c_bool,
    true  TYPE bool VALUE `X`,
    false TYPE bool VALUE ``,
  END OF c_bool.

CONSTANTS:
  BEGIN OF c_tribool,
    true      TYPE tribool VALUE c_bool-true,
    false     TYPE tribool VALUE `-`,
    undefined TYPE tribool VALUE ``,
  END OF c_tribool.

CONSTANTS version TYPE i VALUE 5 ##NO_TEXT.
CONSTANTS mc_key_separator TYPE string VALUE `-` ##NO_TEXT.

CLASS-DATA sv_white_space TYPE string READ-ONLY.
CLASS-DATA mc_bool_types TYPE string READ-ONLY VALUE `\TYPE-POOL=ABAP\TYPE=ABAP_BOOL\TYPE=BOOLEAN\TYPE=BOOLE_D\TYPE=XFELD` ##NO_TEXT.
CLASS-DATA mc_bool_3state TYPE string READ-ONLY VALUE `\TYPE=BOOLEAN` ##NO_TEXT.
CLASS-DATA mc_json_type TYPE string READ-ONLY.

CLASS-METHODS class_constructor.

METHODS constructor
  IMPORTING
    compress         TYPE bool DEFAULT c_bool-false
    pretty_name      TYPE pretty_name_mode DEFAULT pretty_mode-none
    assoc_arrays     TYPE bool DEFAULT c_bool-false
    ts_as_iso8601    TYPE bool DEFAULT c_bool-false
    expand_includes  TYPE bool DEFAULT c_bool-true
    assoc_arrays_opt TYPE bool DEFAULT c_bool-false
    strict_mode      TYPE bool DEFAULT c_bool-false
    numc_as_string   TYPE bool DEFAULT c_bool-false
    nameMappings     TYPE name_mappings OPTIONAL.
CLASS-METHODS deserialize
IMPORTING
  json           TYPE json OPTIONAL
  jsonx          TYPE xstring OPTIONAL
  pretty_name    TYPE pretty_name_mode DEFAULT pretty_mode-none
  assoc_arrays   TYPE bool DEFAULT c_bool-false
  assoc_arrays_opt TYPE bool DEFAULT c_bool-false
  name_mappings  TYPE name_mappings OPTIONAL
CHANGING
  data           TYPE data.

CLASS-METHODS serialize
IMPORTING
  data           TYPE data
  compress       TYPE bool DEFAULT c_bool-false
  name           TYPE string OPTIONAL
  pretty_name    TYPE pretty_name_mode DEFAULT pretty_mode-none
  type_descr     TYPE REF TO cl_abap_typedescr OPTIONAL
  assoc_arrays   TYPE bool DEFAULT c_bool-false
  ts_as_iso8601  TYPE bool DEFAULT c_bool-false
  expand_includes TYPE bool DEFAULT c_bool-true
  assoc_arrays_opt TYPE bool DEFAULT c_bool-false
  numc_as_string TYPE bool DEFAULT c_bool-false
  name_mappings  TYPE name_mappings OPTIONAL
RETURNING
  VALUE(r_json)  TYPE json.

CLASS-METHODS generate
IMPORTING
  json           TYPE json
  pretty_name    TYPE pretty_name_mode DEFAULT pretty_mode-none
  name_mappings  TYPE name_mappings OPTIONAL
RETURNING
  VALUE(rr_data) TYPE REF TO data.

METHODS generate_int
IMPORTING
  json           TYPE json
RETURNING
  VALUE(rr_data) TYPE REF TO data.

METHODS serialize_int
IMPORTING
  data           TYPE data
  name           TYPE string OPTIONAL
  type_descr     TYPE REF TO cl_abap_typedescr OPTIONAL
RETURNING
  VALUE(r_json)  TYPE json.

METHODS deserialize_int
IMPORTING
  json   TYPE json OPTIONAL
  jsonx  TYPE xstring OPTIONAL
CHANGING
data  TYPE data
RAISING
        cx_sy_move_cast_error .

CLASS-METHODS string_to_xstring IMPORTING in TYPE string CHANGING
        VALUE(out) TYPE any .
CLASS-METHODS xstring_to_string IMPORTING in TYPE any RETURNING
        VALUE(out) TYPE string .
CLASS-METHODS bool_to_tribool IMPORTING iv_bool TYPE bool RETURNING
        VALUE(rv_tribool) TYPE tribool .
CLASS-METHODS tribool_to_bool IMPORTING iv_tribool TYPE tribool
        RETURNING VALUE(rv_bool) TYPE bool .

CLASS-METHODS raw_to_string
        IMPORTING
        iv_xstring       TYPE xstring
        iv_encoding      TYPE abap_encoding OPTIONAL
        RETURNING
        VALUE(rv_string) TYPE string .
CLASS-METHODS string_to_raw
        IMPORTING
        iv_string         TYPE string
        iv_encoding       TYPE abap_encoding OPTIONAL
        RETURNING
        VALUE(rv_xstring) TYPE xstring .

PROTECTED SECTION.

TYPES:
        BEGIN OF t_s_symbol,
        header       TYPE string,
        name         TYPE string,
        type         TYPE REF TO cl_abap_datadescr,
        value        TYPE REF TO data,
        compressable TYPE abap_bool,
        read_only    TYPE abap_bool,
        END OF t_s_symbol .

TYPES:
        t_t_symbol TYPE STANDARD TABLE OF t_s_symbol WITH DEFAULT KEY .

TYPES:
        BEGIN OF t_s_field_cache,
        name         TYPE string,
        type         TYPE REF TO cl_abap_datadescr,
        value        TYPE REF TO data,
        compressed   TYPE abap_bool,
        read_only    TYPE abap_bool,
        END OF t_s_field_cache .

TYPES:
        t_t_field_cache TYPE HASHED TABLE OF t_s_field_cache WITH UNIQUE KEY
        name .

DATA mv_compress TYPE bool .
DATA mv_pretty_name TYPE pretty_name_mode .
DATA mv_assoc_arrays TYPE bool.
DATA mv_ts_as_iso8601 TYPE bool.
DATA: mt_name_mappings TYPE name_mappings.
DATA mv_expand_includes TYPE bool.
DATA mv_assoc_arrays_opt TYPE bool.
DATA mv_strict_mode TYPE bool.
DATA mv_numc_as_string TYPE bool.

METHODS dump_symbols FINAL
  IMPORTING
    it_symbols    TYPE t_t_symbol
  RETURNING
    VALUE(r_json) TYPE json.

METHODS get_symbols FINAL
  IMPORTING
    type_descr      TYPE REF TO cl_abap_typedescr
    data            TYPE REF TO data OPTIONAL
    object          TYPE REF TO object OPTIONAL
    include_aliases TYPE abap_bool DEFAULT abap_false
  RETURNING
    VALUE(result)   TYPE t_t_symbol.

METHODS get_fields FINAL
  IMPORTING
    type_descr       TYPE REF TO cl_abap_typedescr
    data             TYPE REF TO data OPTIONAL
    object           TYPE REF TO object OPTIONAL
  RETURNING
    VALUE(rt_fields) TYPE t_t_field_cache.

METHODS dump_int
  IMPORTING
    data          TYPE data
    type_descr    TYPE REF TO cl_abap_typedescr OPTIONAL
  RETURNING
    VALUE(r_json) TYPE json.

METHODS is_compressable
  IMPORTING
    type_descr         TYPE REF TO cl_abap_typedescr
    name               TYPE csequence
  RETURNING
    VALUE(rv_compress) TYPE abap_bool.

METHODS restore
  IMPORTING
    json              TYPE json
    length            TYPE i
    VALUE(type_descr) TYPE REF TO cl_abap_typedescr OPTIONAL
    field_cache       TYPE t_t_field_cache OPTIONAL
  CHANGING
    data              TYPE data OPTIONAL
offset          TYPE i DEFAULT 0
RAISING
   cx_sy_move_cast_error .

METHODS restore_type
IMPORTING
   json              TYPE json
   length            TYPE i
   VALUE(type_descr) TYPE REF TO cl_abap_typedescr OPTIONAL
   field_cache       TYPE t_t_field_cache OPTIONAL
CHANGING
   data              TYPE data OPTIONAL
   offset            TYPE i DEFAULT 0
RAISING
   cx_sy_move_cast_error .

METHODS pretty_name_ex
IMPORTING
   in         TYPE csequence
RETURNING
   VALUE(out) TYPE string .

METHODS generate_int_ex
   FINAL
IMPORTING
   json   TYPE json
   length TYPE i
CHANGING
   data   TYPE data
   offset TYPE i .

METHODS pretty_name
IMPORTING
   in         TYPE csequence
RETURNING
   VALUE(out) TYPE string .

PRIVATE SECTION.
   DATA mv_extended TYPE bool .
   CLASS-DATA mc_me_type TYPE string .
ENDCLASS.

*------------------------------------------------------------------------*
*       CLASS zcl_json MACROS                                           *
*------------------------------------------------------------------------*

DEFINE escape_json_inplace.
* replace all occurrences of regex `\[\"]` in &1 with `\\\$0`. <-- this is slower than 2 plain replaces
   replace all occurrences of `\` in &1 with `\\`.
   replace all occurrences of `"` in &1 with `\"`.
END-OF-DEFINITION.
DEFINE escape_json.
  move &1 to &2.
  escape_json_inplace &2.
END-OF-DEFINITION.

DEFINE is_compressable.
  IF mv_extended IS INITIAL.
    &3 = abap_true.
  ELSE.
    &3 = is_compressable( type_descr = &1 name = &2 ).
  ENDIF.
END-OF-DEFINITION.

DEFINE dump_type.
  case &2->type_kind.
    when cl_abap_typedescr=>typekind_float or cl_abap_typedescr=>typekind_int or cl_abap_typedescr=>typekind_int1 or cl_abap_typedescr=>typekind_int2 or cl_abap_typedescr=>typekind_packed or `8`. " TYPEKIND_INT8 -> '8' only from 7.40.
      if &2->type_kind eq cl_abap_typedescr=>typekind_packed and mv_ts_as_iso8601 eq c_bool-true and &2->absolute_name cp `\TYPE=TIMESTAMP*`.
        if &1 is initial.
          &3 = `""`.
        else.
          move &1 to &3.
          if &2->absolute_name eq `\TYPE=TIMESTAMP`.
            concatenate `""` &3(4) `--` &3+4(2) `--` &3+6(2) `T` &3+8(2) `:` &3+10(2) `:` &3+12(2) `.0000000Z"` into &3.
            else.
            endif.
          elseif &2->absolute_name eq `\TYPE=TIMESTAMPL`.
            concatenate `""` &3(4) `--` &3+4(2) `--` &3+6(2) `T` &3+8(2) `:` &3+10(2) `:` &3+12(2) `.` &3+15(7) `Z"` into &3.
            endif.
        endif.
    endif.
    elseif &1 is initial.
      &3 = `0`.
    else.
      move &1 to &3.
      if &1 lt 0.
        if &2->type_kind <> cl_abap_typedescr=>typekind_float. "float: sign is already at the beginning
          shift &3 right circular.
        endif.
      else.
        condense &3.
      endif.
    endif.
  when cl_abap_typedescr=>typekind_num.
    if mv_numc_as_string eq abap_true.
      if &1 is initial.
        &3 = `""`.
    ```
else.
    concatenate \" \" &1 \" \" into &3.
endif.
else.
  if &1 is initial.
    &3 = \"0\".
  else.
    move &1 to &3.
    shift &3 left deleting leading \"0\".
  endif.
endif.
when cl_abap_typedescr=>typekind_string or
cl_abap_typedescr=>typekind_csequence or cl_abap_typedescr=>typekind_clike.
  if &1 is initial.
    &3 = \"\".
  elseif &2->absolute_name eq mc_json_type.
    &3 = &1.
  else.
    escape_json &1 &3.
    concatenate \" \" &3 \" \" into &3.
  endif.
when cl_abap_typedescr=>typekind_xstring or
cl_abap_typedescr=>typekind_hex.
  if &1 is initial.
    &3 = \"\".
  else.
    &3 = xstring_to_string( &1 ).
    escape_json_inplace &3.
    concatenate \" \" &3 \" \" into &3.
  endif.
when cl_abap_typedescr=>typekind_date.
  concatenate \" \" &1(4) \"-\" &1+4(2) \"\" &1+6(2) \" \" into &3.
when cl_abap_typedescr=>typekind_time.
  concatenate \" \" &1(2) \":\" &1+2(2) \:" &1+4(2) \" \" into &3.
when others.
  if &1 is initial.
    &3 = \"null\".
  else.
    move &1 to &3.
  endif.
DEFINE format_name.
case &2.
  when pretty_mode-camel_case.
    &3 = pretty_name( &1 ).
  when pretty_mode-extended.
    &3 = pretty_name_ex( &1 ).
  when pretty_mode-user_low_case.
    read table mt_name_mappings with table key abap = &1 assigning <cache>.
    if sy-subrc is initial.
      &3 = <cache>-json.
    else.
      &3 = &1.
      translate &3 to lower case.                     "#EC SYNTCHAR
    endif.
  when pretty_mode-user.
    read table mt_name_mappings with table key abap = &1 assigning <cache>.
    if sy-subrc is initial.
      &3 = <cache>-json.
    else.
      &3 = &1.
    endif.
  when pretty_mode-low_case.
    &3 = &1.
    translate &3 to lower case.                     "#EC SYNTCHAR
  when others.
    &3 = &1.
endcase.
END-OF-DEFINITION.

DEFINE throw_error.
  raise exception type cx_sy_move_cast_error.
END-OF-DEFINITION.

DEFINE while_offset_cs.
  while offset < length.
    find first occurrence of json+offset(1) in &1.
    if sy-subrc is not initial.
      exit.
    endif.
    offset = offset + 1.
  endwhile.
END-OF-DEFINITION.

DEFINE eat_white.
  while_offset_cs sv_white_space.
END-OF-DEFINITION.
DEFINE eat_string.
    if json+offset(1) eq `"`.
        mark   = offset + 1.
        offset = mark.
        unescape = abap_false.
    do.
        find first occurrence of `"` in section offset offset of json match offset pos.
        if sy-subrc is not initial.
            throw_error.
        endif.
        offset = pos.
        pos = pos - 1.
        " if escaped search further
        while pos ge 0 and json+pos(1) eq `\`
            pos = pos - 1.
            unescape = abap_true.
        endwhile.
        match = ( offset - pos ) mod 2.
        if match ne 0.
            exit.
        endif.
        offset = offset + 1.
    enddo.
    match = offset - mark.
    &1 = json+mark(match).
    if unescape eq abap_true.
        replace all occurrences of `"` in &1 with `"`.
    endif.
    " \ shall be unescaped always, while we do not have check for that
    replace all occurrences of `\\` in &1 with `\\`.
    offset = offset + 1.
else.
    throw_error.
endif.
END-OF-DEFINITION.

DEFINE eat_number.
    mark   = offset.
    while_offset_cs `0123456789+-eE`.                        #EC NOTEXT
    match = offset - mark.
    &1 = json+mark(match).
    if unescape eq abap_true.
        replace all occurrences of `\`` in &1 with `\``.
    endif.
    " \ shall be unescaped always, while we do not have check for that
    replace all occurrences of `\\` in &1 with `\\`.
    offset = offset + 1.
else.
    throw_error.
endif.
END-OF-DEFINITION.

DEFINE eat_bool.
    mark   = offset.
    while_offset_cs `aeflnrstu`.                              #EC NOTEXT
    match = offset - mark.
    &1 = json+mark(match).
    if json+mark(match) eq `true`.                            #EC NOTEXT
        &1 = c_bool-true.
    elseif json+mark(match) eq `false`.                       #EC NOTEXT
        if type_descr is bound and mc_bool_3state cs type_descr->absolute_name.
            &1 = c_tribool-false.
        endif.
        " shall be unescaped always, while we do not have check for that
        replace all occurrences of `\\` in &1 with `\\`.
    endif.
    offset = offset + 1.
else.
    throw_error.
endif.
END-OF-DEFINITION.
else.
   &1 = c_bool-false.
endif.
elseif json+mark(match) eq `null`.
   clear &1.
endif.
END-OF-DEFINITION.

DEFINE eat_char.
   if offset < length and json+offset(1) eq &1.
      offset = offset + 1.
   else.
      throw_error.
   endif.
END-OF-DEFINITION.

*----------------------------------------------------------------------*
*       CLASS zcl_json IMPLEMENTATION
*----------------------------------------------------------------------*

CLASS zcl_json IMPLEMENTATION.

METHOD class_constructor.

   DATA: lo_bool_type_descr    TYPE REF TO cl_abap_typedescr,
       lo_tribool_type_descr TYPE REF TO cl_abap_typedescr,
       lo_json_type_descr    TYPE REF TO cl_abap_typedescr,
       lv_pos                LIKE sy-fdpos,
       lv_json_string        TYPE json.

   lo_bool_type_descr    = cl_abap_typedescr=>describe_by_data( c_bool-true ).
   lo_tribool_type_descr = cl_abap_typedescr=>describe_by_data( c_tribool-true ).
   lo_json_type_descr    = cl_abap_typedescr=>describe_by_data( lv_json_string ).

   CONCATENATE mc_bool_types lo_bool_type_descr->absolute_name INTO mc_bool_types.
   CONCATENATE mc_bool_3state lo_tribool_type_descr->absolute_name INTO mc_bool_3state.
   CONCATENATE mc_json_type lo_json_type_descr->absolute_name INTO mc_json_type.

   FIND FIRST OCCURRENCE OF `\TYPE=` IN lo_json_type_descr->absolute_name
   MATCH OFFSET lv_pos.
   IF sy-subrc IS INITIAL.
      mc_me_type = lo_json_type_descr->absolute_name(lv_pos).
   ENDIF.

   sv_white_space = cl_abap_char_utilities=>get_simple_spaces_for_cur_cp( ).
ENDMETHOD.                  

METHOD constructor.

DATA: rtti TYPE REF TO cl_abap_classdescr,
       pair LIKE LINE OF name_mappings.

mv_compress         = compress.
mv_pretty_name      = pretty_name.
mv_assoc_arrays     = assoc_arrays.
mv_ts_as_iso8601    = ts_as_iso8601.
mv_expand_includes  = expand_includes.
mv_assoc_arrays_opt = assoc_arrays_opt.
mv_strict_mode      = strict_mode.
mv_numc_as_string   = numc_as_string.

LOOP AT name_mappings INTO pair.
   TRANSLATE pair-abap TO UPPER CASE.
   INSERT pair INTO TABLE mt_name_mappings.
ENDLOOP.

IF mt_name_mappings IS NOT INITIAL.
   IF mv_pretty_name EQ pretty_mode-none.
      mv_pretty_name = pretty_mode-user.
   ELSEIF pretty_name EQ pretty_mode-low_case.
      mv_pretty_name = pretty_mode-user_low_case.
   ENDIF.
ENDIF.

rtti ?= cl_abap_classdescr=>describe_by_object_ref( me ).
IF rtti->absolute_name NE mc_me_type.
   mv_extended = c_bool-true.
ENDIF.

ENDMETHOD.

METHOD serialize.

"*
******************************************************************************
"! Usage examples and documentation can be found on SCN:
"*
http://wiki.scn.sap.com/wiki/display/Snippets/One+more+ABAP+to+JSON+Serializer+and+Deserializer
"*
****************************************************************************** *

DATA: lo_json  TYPE REF TO zcl_json.

CREATE OBJECT lo_json
   EXPORTING
      compress  = compress
      pretty_name = pretty_name
METHOD serialize_int.
  DATA: lo_descr TYPE REF TO cl_abap_typedescr.

  IF type_descr IS INITIAL.
    lo_descr = cl_abap_typedescr=>describe_by_data( data ).
  ELSE.
    lo_descr = type_descr.
  ENDIF.

  r_json = dump_int( data = data type_descr = lo_descr ).

  REPLACE ALL OCCURRENCES OF cl_abap_char_utilities=>cr_lf IN r_json WITH '
'.
  REPLACE ALL OCCURRENCES OF cl_abap_char_utilities=>newline IN r_json WITH '
'.
  REPLACE ALL OCCURRENCES OF cl_abap_char_utilities=>horizontal_tab IN r_json WITH '	'.

  IF name IS NOT INITIAL AND ( mv_compress IS INITIAL OR r_json IS NOT INITIAL ).
    CONCATENATE `"` name `"`: ` r_json INTO r_json.
  ENDIF.

ENDMETHOD.

METHOD deserialize.

  DATA: lo_json TYPE REF TO zcl_json.

  "
  **************************************************************
  "! Usage examples and documentation can be found on SCN:
  "
  http://wiki.scn.sap.com/wiki/display/Snippets/One+more+ABAP+to+JSON+Serializer+and+Deserializer
  "
  **************************************************************  "

  IF json IS NOT INITIAL OR jsonx IS NOT INITIAL.
CREATE OBJECT lo_json
EXPERTING
    pretty_name      = pretty_name
    name_mappings    = name_mappings
    assoc_arrays     = assoc_arrays
    assoc_arrays_opt = assoc_arrays_opt.

TRY .
    lo_json->deserialize_int( EXPORTING json = json jsonx = jsonx
                             CHANGING data = data ).
    CATCH cx_sy_move_cast_error.
    ENDTRY.
    ENDMETHOD.                    "deserialize

METHOD deserialize_int.

    DATA: length    TYPE i,
          unescaped LIKE json.

    IF json IS NOT INITIAL OR jsonx IS NOT INITIAL.
        IF jsonx IS NOT INITIAL.
            unescaped = raw_to_string( jsonx ).
        ELSE.
            unescaped = json.
        ENDIF.
    ENDIF.

    REPLACE ALL OCCURRENCES OF `\r\n` IN unescaped WITH
        cl_abap_char_utilities=>cr_lf.
    REPLACE ALL OCCURRENCES OF `\n` IN unescaped WITH
        cl_abap_char_utilities=>newline.
    REPLACE ALL OCCURRENCES OF `\t` IN unescaped WITH
        cl_abap_char_utilities=>horizontal_tab.

    length = numofchar( unescaped ).
    restore_type( EXPORTING json = unescaped length = length CHANGING
                  data = data ).
    ENDIF.
    ENDMETHOD.                    "deserialize

METHOD generate.

    DATA: lo_json TYPE REF TO zcl_json,
          lv_json LIKE json.

    lv_json = json.

    REPLACE ALL OCCURRENCES OF `\r\n` IN lv_json WITH
cl_abap_char_utilities=>cr_lf.
  REPLACE ALL OCCURRENCES OF `\n` IN lv_json WITH
cl_abap_char_utilities=>newline.
  REPLACE ALL OCCURRENCES OF `\t` IN lv_json WITH
cl_abap_char_utilities=>horizontal_tab.

CREATE OBJECT lo_json
  EXPORTING
    pretty_name      = pretty_name
    name_mappings    = name_mappings
    assoc_arrays     = c_bool-true
    assoc_arrays_opt = c_bool-true.

TRY .
  rr_data = lo_json->generate_int( lv_json ).
  CATCH cx_sy_move_cast_error.
ENDTRY.

ENDMETHOD.

METHOD generate_int.

  TYPES: BEGIN OF ts_field,
    name  TYPE string,
    value TYPE json,
  END OF ts_field.

  DATA: length TYPE i,
        offset TYPE i.

  DATA: lt_json      TYPE STANDARD TABLE OF json WITH DEFAULT KEY,
        lv_json      LIKE LINE OF lt_json,
        lv_comp_name TYPE abap_comppname,
        lt_fields    TYPE SORTED TABLE OF ts_field WITH UNIQUE KEY name,
        lo_type      TYPE REF TO cl_abap_datadescr,
        lt_comp      TYPE SORTED TABLE OF abap_componentdescr WITH UNIQUE KEY name,
        lt_comp_std  TYPE abap_component_tab,
        ls_comp      LIKE LINE OF lt_comp.

  FIELD-SYMBOLS: <data>   TYPE any,
                 <struct> TYPE any,
                 <field>  LIKE LINE OF lt_fields,
                 <table>  TYPE STANDARD TABLE.

  length = numofchar( json ).

  eat_white.

  CASE json+offset(1).
    WHEN `(`."result must be a structure
      restore_type( EXPORTING json = json length = length CHANGING data
                                 = lt_fields ).
  END_CASE.
IF lt_fields IS NOT INITIAL.
    ls_comp-type = cl_abap_refdescr=>get_ref_to_data( ).
    LOOP AT lt_fields ASSIGNING <field>.
        ls_comp-name = <field>-name.
        TRANSLATE ls_comp-name USING `/_:~._-_`. " remove characters 
        not allowed in component names
        IF mv_pretty_name EQ pretty_mode-camel_case OR mv_pretty_name 
        EQ pretty_mode-extended.
            REPLACE ALL OCCURRENCES OF REGEX `([a-z])\([A-Z]\)` IN
            ls_comp-name WITH `\$1_\$2`. " #EC NOTEXT
        ENDIF.
        TRANSLATE ls_comp-name TO UPPER CASE.
        ls_comp-name = lv_comp_name = ls_comp-name. " truncate by 
        allowed field name length
        INSERT ls_comp INTO TABLE lt_comp.
        CHECK sy-subrc IS NOT INITIAL.
        DELETE lt_fields.
    ENDLOOP.
    TRY.
        lt_comp_std = lt_comp.
        lo_type = cl_abap_structdescr=>create( p_components =
        lt_comp_std p_strict = c Bool-false ).
        CREATE DATA rr_data TYPE HANDLE lo_type.
        ASSIGN rr_data->* TO <struct>.
        LOOP AT lt_fields ASSIGNING <field>.
            ASSIGN COMPONENT sy-tabix OF STRUCTURE <struct> TO <data>.
            <data> = generate_int( <field>-value ).
        ENDLOOP.
        CATCH cx_sy_create_data_error cx_sy_struct_creation.
        ENDDO.
    ENDTRY.
    WHEN `\`."result must be a table of ref
    restore_type( EXPORTING json = json length = length CHANGING data
    = lt_json ).
    CREATE DATA rr_data TYPE TABLE OF REF TO data.
    ASSIGN rr_data->* TO <table>.
    LOOP AT lt_json INTO lv_json.
        APPEND INITIAL LINE TO <table> ASSIGNING <data>.
        <data> = generate_int( lv_json ).
    ENDLOOP.
    WHEN OTHERS.
    IF json+offset(1) EQ `\"`.
        CREATE DATA rr_data TYPE string.
    ELSEIF json+offset(1) CA `-0123456789.`.
        IF json+offset CS `\."`.
            CREATE DATA rr_data TYPE f.
        ELSE.
            CREATE DATA rr_data TYPE i.
        ENDIF.
    ELSEIF json+offset EQ `true` OR json+offset EQ `false`.
        CREATE DATA rr_data TYPE abap_bool.
    ENDIF.
    IF rr_data IS BOUND.
ASSIGN rr_data->* TO <data>.
restore_type( EXPORTING json = json length = length CHANGING data = <data> ).
ENDIF.
ENDCASE.
ENDMETHOD.

METHOD generate_int_ex.

DATA: lv_assoc_arrays     LIKE mv_assoc_arrays,
     lv_assoc_arrays_opt LIKE mv_assoc_arrays_opt,
     lv_mark             LIKE offset,
     lv_match            LIKE lv_mark,
     lv_json             TYPE zcl_json=>json.

lv_mark = offset.
restore_type( EXPORTING json = json length = length CHANGING offset =
offset ).
lv_match = offset - lv_mark.
lv_json = json+lv_mark(lv_match).

lv_assoc_arrays     = mv_assoc_arrays.
lv_assoc_arrays_opt = mv_assoc_arrays_opt.

mv_assoc_arrays     = abap_true.
mv_assoc_arrays_opt = abap_true.

data = generate_int( lv_json ).

mv_assoc_arrays = lv_assoc_arrays.
mv_assoc_arrays_opt = lv_assoc_arrays_opt.
ENDMETHOD.

METHOD dump_int.

DATA: lo_typedesc   TYPE REF TO cl_abap_typedescr,
     lo_elem_descr TYPE REF TO cl_abap_elemdescr,
     lo_classdesc  TYPE REF TO cl_abap_classdescr,
     lo_structdesc TYPE REF TO cl_abap_structdescr,
     lo_tabledescr TYPE REF TO cl_abap_tabledescr,
     lt_symbols    TYPE t_t_symbol,
     lt_keys       LIKE lt_symbols,
     lt_properties TYPE STANDARD TABLE OF string,
     lt_fields     TYPE STANDARD TABLE OF string,
     lo_obj_ref    TYPE REF TO object,
     lo_data_ref   TYPE REF TO data,
     ls_skip_key   TYPE LINE OF abap_keydescr_tab,
     lv_array_opt  TYPE abap_bool,
     lv_prop_name  TYPE string,
     lv_keyval     TYPE string,
     lv_itemval    TYPE string.
CASE type_descr->kind.
  WHEN cl_abap_typedescr=>kind_ref.
    IF data IS INITIAL.
      r_json = 'null'.
    ENDIF.
    ELSEIF type_descr->type_kind EQ cl_abap_typedescr=>typekind_dref.
      lo_data_ref ?= data.
      lo_typedesc = cl_abap_typedescr=>describe_by_data_ref( lo_data_ref ).
      ASSIGN lo_data_ref->* TO <data>.
      r_json = dump_int( data = <data> type_descr = lo_typedesc ).
    ELSE.
      lo_obj_ref ?= data.
      lo_classdesc ?= cl_abap_typedescr=>describe_by_object_ref( lo_obj_ref ).
      lt_symbols = get_symbols( type_descr = lo_classdesc object = lo_obj_ref ).
      r_json = dump_symbols( lt_symbols ).
    ENDIF.
  WHEN cl_abap_typedescr=>kind_elem.
    lo_elem_descr ?= type_descr.
    dump_type data lo_elem_descr r_json.
  WHEN cl_abap_typedescr=>kind_struct.
    lo_structdesc ?= type_descr.
    GET REFERENCE OF data INTO lo_data_ref.
    lt_symbols = get_symbols( type_descr = lo_structdesc data = lo_data_ref ).
    r_json = dump_symbols( lt_symbols ).
  WHEN cl_abap_typedescr=>kind_table.
    lo_tabledescr ?= type_descr.
    lo_typedesc = lo_tabledescr->get_table_line_type( ).
    ASSIGN data TO <table>.
    r_json = 'null'.
    ELSEIF lo_typedesc->kind EQ cl_abap_typedescr=>kind_struct.
      lo_structdesc ?= lo_typedesc.
      CREATE DATA lo_data_ref LIKE LINE OF <table>.
      ASSIGN lo_data_ref-* TO <line>.
      lt_symbols = get_symbols( type_descr = lo_structdesc data =
lo_data_ref).

" here we have differentiation of output of simple table to JSON array

" and sorted or hashed table with unique key into JSON associative array

IF lo_tabledescr->has_unique_key IS NOT INITIAL AND
mv_assoc_arrays IS NOT INITIAL.

IF lo_tabledescr->key_defkind EQ
lo_tabledescr->keydefkind_user.

LOOP AT lo_tabledescr->key ASSIGNING <key>.

READ TABLE lt_symbols WITH KEY name = <key>-name ASSIGNING <symbol>.

APPEND <symbol> TO lt_keys.
ENDLOOP.
ENDIF.

IF lines( lo_tabledescr->key ) EQ 1.
READ TABLE lo_tabledescr->key INDEX 1 INTO ls_skip_key.
DELETE lt_symbols WHERE name EQ ls_skip_key-name.
" remove object wrapping for simple name-value tables
IF mv_assoc_arrays_opt EQ abap_true AND lines( lt_symbols )
EQ 1.

lv_array_opt = abap_true.
ENDIF.
ENDIF.

LOOP AT <table> INTO <line>.
CLEAR: lt_fields, lv_prop_name.
LOOP AT lt_symbols ASSIGNING <symbol>.

ASSIGN <symbol>-value-* TO <value>.

IF mv_compress IS INITIAL OR <value> IS NOT INITIAL OR
<symbol>-compressable EQ abap_false.

IF <symbol>-type->kind EQ cl_abap_typedescr=>kind_elem.

lo_elem_descr ?= <symbol>-type.

dump_type <value> lo_elem_descr lv_itemval.
ELSE.

lv_itemval = dump_int( data = <value> type_descr =
<symbol>-type ).
ENDIF.

ENDIF.

IF lv_array_opt EQ abap_false.

CONCATENATE <symbol>-header lv_itemval INTO lv_itemval.
ENDIF.

APPEND lv_itemval TO lt_fields.
ENDIF.
ENDLOOP.

IF lo_tabledescr->key_defkind EQ
lo_tabledescr->keydefkind_user.

LOOP AT lt_keys ASSIGNING <symbol>.

ASSIGN <symbol>-value-* TO <value>.

MOVE <value> TO lv_keyval.
CONDENSE lv_keyval.
    IF lv_prop_name IS NOT INITIAL.
        CONCATENATE lv_prop_name mc_key_separator lv_keyval INTO lv_prop_name.
    ELSE.
        lv_prop_name = lv_keyval.
    ENDIF.
ENDLOOP.
ELSE.
    LOOP AT lt_symbols ASSIGNING <symbol>.
        ASSIGN <symbol>-value-* TO <value>.
        MOVE <value> TO lv_keyval.
        CONDENSE lv_keyval.
        IF lv_prop_name IS NOT INITIAL.
            CONCATENATE lv_prop_name mc_key_separator lv_keyval INTO lv_prop_name.
        ELSE.
            lv_prop_name = lv_keyval.
        ENDIF.
    ENDLOOP.
ENDIF.

CONCATENATE LINES OF lt_fields INTO lv_itemval SEPARATED BY ` `, `. `.
    IF lv_array_opt EQ abap_false.
        CONCATENATE `"` lv_prop_name `":{` lv_itemval `}` INTO lv_itemval.
    ELSE.
        CONCATENATE `"` lv_prop_name `":` lv_itemval `"` INTO lv_itemval.
    ENDIF.
    APPEND lv_itemval TO lt_properties.
ENDLOOP.

CONCATENATE LINES OF lt_properties INTO r_json SEPARATED BY ` `, `. `.
CONCATENATE `{ ` r_json ` `} INTO r_json.
ELSE.
    LOOP AT <table> INTO <line>.
        CLEAR lt_fields.
        LOOP AT lt_symbols ASSIGNING <symbol>.
            ASSIGN <symbol>-value-* TO <value>.
            IF mv_compress IS INITIAL OR <value> IS NOT INITIAL OR <symbol>-compressable EQ abap_false.
                IF <symbol>-type->kind EQ cl_abap_typedescr->kind_elem.
                    lo_elem_descr ?= <symbol>-type.
                    dump_type <value> lo_elem_descr lv_itemval.
                ELSE.
                    lv_itemval = dump_int( data = <value> type_descr = <symbol>-type ).
                ENDIF.
            ENDIF.
        ENDLOOP.
    ENDLOOP.
METHOD dump_symbols.

DATA: lv_properties TYPE STANDARD TABLE OF string,
    lv_itemval TYPE string.

FIELD-SYMBOLS: <value> TYPE any,
    <symbol> LIKE LINE OF it_symbols.

LOOP AT it_symbols ASSIGNING <symbol>.
    ASSIGN <symbol>-value-* TO <value>.
    IF mv_compress IS INITIAL OR <value> IS NOT INITIAL OR
    <symbol>-compressable EQ abap_false.
        lv_itemval = dump_int( data = <value> type_descr = <symbol>-type ).
        CONCATENATE <symbol>-header lv_itemval INTO lv_itemval.
        APPEND lv_itemval TO lv_properties.
    ENDIF.
ENDLOOP.

CONCATENATE LINES OF lv_properties INTO r_json SEPARATED BY `,`..
CONCATENATE `{` r_json `}` INTO r_json.
ENDMETHOD.
METHOD get_fields.

DATA: lt_symbols TYPE t_t_symbol,
  lv_name    TYPE char128,
  ls_field   LIKE LINE OF rt_fields.

FIELD-SYMBOLES: <sym> LIKE LINE OF lt_symbols,
  <cache> LIKE LINE OF mt_name_mappings.

lt_symbols = get_symbols( type_descr = type_descr data = data object =
  object include_aliases = abap_true ).

LOOP AT lt_symbols ASSIGNING <sym> WHERE read_only EQ abap_false.
  ls_field-name  = <sym>-name.
  ls_field-type  = <sym>-type.
  ls_field-value = <sym>-value.

  " insert as UPPERCASE
  INSERT ls_field INTO TABLE rt_fields.

  " insert as lower case
  TRANSLATE ls_field-name TO LOWER CASE.
  INSERT ls_field INTO TABLE rt_fields.

  " as pretty printed
  IF mv_pretty_name NE pretty_mode-none AND mv_pretty_name NE
  pretty_mode-low_case.
    format_name <sym>-name mv_pretty_name ls_field-name.
    INSERT ls_field INTO TABLE rt_fields.
  " let us check for not well formed _camelCase_ to be compatible with
  old logic
  lv_name = ls_field-name.
  TRANSLATE lv_name(1) TO UPPERCASE.
  ls_field-name = lv_name.
  INSERT ls_field INTO TABLE rt_fields.
  ENDIF.

  ENDLOOP.

ENDMETHOD.

METHOD get_symbols.

DATA: comp_tab     TYPE cl_abap_structdescr=>component_table,
  symb_tab     LIKE result,
  symb         LIKE LINE OF symb_tab,
  class_descr  TYPE REF TO cl_abap_classdescr,
  struct_descr TYPE REF TO cl_abap_structdescr.

FIELD-SYMBOLES: <comp> LIKE LINE OF comp_tab,
  <attr> LIKE LINE OF cl_abap_objectdescr=>attributes,
LIKE LINE OF mt_name_mappings,
 TYPE any.

IF type_descr->kind EQ cl_abap_typedescr->kind_struct.

struct_descr ?= type_descr.
comp_tab = struct_descr->get_components( ).

LOOP AT comp_tab ASSIGNING <comp>.
  IF <comp>-name IS NOT INITIAL AND
    ( <comp>-as_include EQ abap_false OR include_aliases EQ abap_true
       OR mv_expand_includes EQ abap_false ).
    symb-name = <comp>-name.
    symb-type = <comp>-type.
    IF data IS BOUND.
      is_compressable symb-type symb-name symb-compressable.
      ASSIGN data->(symb-name) TO <field>.
      GET REFERENCE OF <field> INTO symb-value.
      format_name symb-name mv_pretty_name symb-header.
      CONCATENATE `"` symb-header `":` INTO symb-header.
    ENDIF.
    APPEND symb TO result.
  ENDIF.
  IF <comp>-as_include EQ abap_true AND mv_expand_includes EQ abap_true.
    struct_descr ?= <comp>-type.
    symb_tab = get_symbols( type_descr = struct_descr include_aliases = include_aliases ).
    LOOP AT symb_tab INTO symb.
      CONCATENATE symb-name <comp>-suffix INTO symb-name.
      IF data IS BOUND.
        is_compressable symb-type symb-name symb-compressable.
        ASSIGN data->(symb-name) TO <field>.
        GET REFERENCE OF <field> INTO symb-value.
        format_name symb-name mv_pretty_name symb-header.
        CONCATENATE `"` symb-header `":` INTO symb-header.
      ENDIF.
      APPEND symb TO result.
    ENDLOOP.
  ENDIF.
ENDLOOP.
ELSEIF type_descr->type_kind EQ cl_abap_typedescr->typekind_class.

class_descr ?= type descr.
LOOP AT class_descr->attributes ASSIGNING <attr> WHERE is_constant IS INITIAL AND alias_for IS INITIAL AND
  ( is_interface IS INITIAL OR type_kind NE cl_abap_typedescr->typekind_oref ).
  ASSIGN object->(<attr>-name) TO <field>.
  CHECK sy-subrc IS INITIAL. " we can only assign to public attributes
  symb-name = <attr>-name.
symb-read_only = <attr>-is_read_only.
symb-type = class_descr->get_attribute_type( <attr>-name ).
is_compressable symb-type symb-name symb-compressable.
GET REFERENCE OF <field> INTO symb-value.
format_name symb-name mv_pretty_name symb-header.
CONCATENATE `"` symb-header `":` INTO symb-header.
APPEND symb TO result.
ENDLOOP.

ENDIF.
ENDMETHOD.  "GET_SYMBOLS

METHOD is_compressable.
  rv_compress = abap_true.
ENDMETHOD.

METHOD pretty_name.

  DATA: tokens TYPE TABLE OF char128,
       cache LIKE LINE OF mt_name_mappings.

  FIELD-SYMBOLS: <token> LIKE LINE OF tokens,
                  <cache> LIKE LINE OF mt_name_mappings.

  READ TABLE mt_name_mappings WITH TABLE KEY abap = in ASSIGNING <cache>.
  IF sy-subrc IS INITIAL.
    out = <cache>-json.
  ELSE.
    out = in.

    REPLACE ALL OCCURRENCES OF `__` IN out WITH `*`.

    TRANSLATE out TO LOWER CASE.
    TRANSLATE out USING `/__/_~_`.
    SPLIT out AT `_` INTO TABLE tokens.
    LOOP AT tokens ASSIGNING <token> FROM 2.
      TRANSLATE <token>(1) TO UPPER CASE.
      ENDLOOP.

    CONCATENATE LINES OF tokens INTO out.
    REPLACE ALL OCCURRENCES OF `*` IN out WITH `_`.

    cache-abap  = in.
    cache-json = out.
    INSERT cache INTO TABLE mt_name_mappings.
  ENDIF.

ENDMETHOD.  "pretty_name

METHOD pretty_name_ex.

  DATA: tokens TYPE TABLE OF char128,
cache LIKE LINE OF mt_name_mappings,
lt_match TYPE match_result_tab.

FIELD-SYMBOLS: <token> LIKE LINE OF tokens,
<cache> LIKE LINE OF mt_name_mappings,
<match> LIKE LINE OF lt_match,
<sub_match> TYPE LINE OF submatch_result_tab.

READ TABLE mt_name_mappings WITH TABLE KEY abap = in ASSIGNING <cache>.
IF sy-subrc IS INITIAL.
  out = <cache>-json.
ELSE.
  out = in.
  TRANSLATE out TO LOWER CASE.
  TRANSLATE out USING `/:_~`.
  REPLACE ALL OCCURRENCES OF `__e__` IN out WITH `!`.
  REPLACE ALL OCCURRENCES OF `__n__` IN out WITH `#`.
  REPLACE ALL OCCURRENCES OF `__d__` IN out WITH `$`.
  REPLACE ALL OCCURRENCES OF `__p__` IN out WITH `%`.
  REPLACE ALL OCCURRENCES OF `__m__` IN out WITH `&`.
  REPLACE ALL OCCURRENCES OF `__s__` IN out WITH `*`.
  REPLACE ALL OCCURRENCES OF `__h__` IN out WITH `-`.
  REPLACE ALL OCCURRENCES OF `__t__` IN out WITH `~`.
  REPLACE ALL OCCURRENCES OF `__l__` IN out WITH `/`.
  REPLACE ALL OCCURRENCES OF `__c__` IN out WITH `:`.
  REPLACE ALL OCCURRENCES OF `__v__` IN out WITH `|`.
  REPLACE ALL OCCURRENCES OF `__a__` IN out WITH `@`.
  REPLACE ALL OCCURRENCES OF `__o__` IN out WITH `.`.
  REPLACE ALL OCCURRENCES OF `__` IN out WITH `.`.
  REPLACE ALL OCCURRENCES OF `__` IN out WITH `"`.
  SPLIT out AT `_` INTO TABLE tokens.
  LOOP AT tokens ASSIGNING <token> FROM 2.
    TRANSLATE <token>(1) TO UPPER CASE.
  ENDL loop.
  CONCATENATE LINES OF tokens INTO out.
  REPLACE ALL OCCURRENCES OF `"` IN out WITH `\`.
  cache-abap = in.
  cache-json = out.
  INSERT cache INTO TABLE mt_name_mappings.
ENDIF.

ENDMETHOD. "pretty_name_ex

METHOD restore.

DATA: mark LIKE offset,
  match LIKE offset,
pos LIKE offset,
unescape TYPE abap_bool,
ref_descr TYPE REF TO cl_abap_refdescr,
data_descr TYPE REF TO cl_abap_datadescr,
data_ref TYPE REF TO data,
object_ref TYPE REF TO object,
fields LIKE field_cache,
name_json TYPE string.

FIELD-SYMBOLS: <value> TYPE any,
    <field_cache> LIKE LINE OF field_cache.

fields = field_cache.

IF type_descr IS NOT INITIAL AND type_descr->kind EQ type_descr->kind_ref.
    ref_descr ?= type_descr.
    type_descr = ref_descr->get_referenced_type( ).
    IF ref_descr->type_kind EQ ref_descr->typekind_oref.
    IF data IS INITIAL.
        "can fire an exception, if type is abstract or constructor protected
        CREATE OBJECT data TYPE (type_descr->absolute_name).
    ENDIF.
    object_ref ?= data.
    fields = get_fields( type_descr = type_descr object = object_ref ).
ELSEIF ref_descr->type_kind EQ ref_descr->typekind_dref.
    IF data IS INITIAL.
    data_descr ?= type_descr.
    CREATE DATA data TYPE HANDLE data_descr.
    ENDIF.
    data_ref ?= data.
    ASSIGN data_ref->* TO <value>.
    fields = get_fields( type_descr = type_descr data = data_ref ).
    restore( EXPORTING json = json length = length type_descr =
    type_descr field_cache = fields
    CHANGING data = <value> offset = offset ).
    RETURN.
ENDIF.
ENDIF.

IF fields IS INITIAL AND type_descr IS NOT INITIAL AND type_descr->kind EQ type_descr->kind_struct.
    GET REFERENCE OF data INTO data_ref.
    fields = get_fields( type_descr = type_descr data = data_ref ).
ENDIF.
eat_white.
eat_char `\`. 
eat_white.

WHILE offset < length AND json+offset(1) NE `}`.`.
READ TABLE fields WITH TABLE KEY name = name_json ASSIGNING <field_cache>.
   IF sy-subrc IS NOT INITIAL.
      TRANSLATE name_json TO UPPER CASE.
      READ TABLE fields WITH TABLE KEY name = name_json ASSIGNING <field_cache>.
   ENDIF.

   IF sy-subrc IS INITIALLY.
      ASSIGN <field_cache>-value-* TO <value>.
      restore_type( EXPORTING json = json length = length type_descr = <field_cache>-type CHANGING data = <value> offset = offset ).
   ELSE.
      restore_type( EXPORTING json = json length = length CHANGING offset = offset ).
   ENDIF.

   IF offset < length AND json+offset(1) NE `}`.
      eat_char ``,``.
   ELSE.
      EXIT.
   ENDIF.
ENDWHILE.

   eat_char `}`).
ENDMETHOD.                     "restore

METHOD restore_type.

DATA: mark        LIKE offset,
      match       LIKE offset,
      unescape    TYPE abap_bool,
      sdummy      TYPE string,
      lr_idummy   TYPE REF TO i,  "#EC NEEDED
      lr_bdummy   TYPE REF TO bool,  "#EC NEEDED
      lr_sdummy   TYPE REF TO string,  "#EC NEEDED
      pos         LIKE offset,
      line        TYPE REF TO data,
      key_ref     TYPE REF TO data,
      data_ref    TYPE REF TO data,
      key_name    TYPE string,
      key_value   TYPE string,
      lt_fields   LIKE field_cache,
FIELD-SYMBOLS: <line> TYPE any,
  <value> TYPE any,
  <data> TYPE data,
  <field> LIKE LINE OF lt_fields,
  <table> TYPE ANY TABLE,
  <value_sym> LIKE LINE OF lt_symbols.

IF type_descr IS INITIAL AND data IS SUPPLIED.
  type_descr = cl_abap_typedescr=>describe_by_data( data ).
ENDIF.

eat_white.

TRY .
  IF type_descr IS NOT INITIAL AND type_descr->absolute_name EQ mc_json_type.
    " skip deserialization
    mark = offset.
    restore_type( EXPORTING json = json length = length CHANGING
    offset = offset ).
    match = offset - mark.
    data = json+mark(match).
  ENDIF.
  CASE json+offset(1).
    WHEN `{`. " object
      IF type_descr IS NOT INITIAL.
        IF mv_assoc_arrays EQ c_bool-true AND type_descr->kind EQ
cl_abap_typedescr=>kind_table.
          table descr ?= type_descr.
          data descr = table descr->get_table_line_type( ).
        IF table descr->has_unique_key IS NOT INITIAL.
          eat_char `{`. 
          eat_white.
          IF json+offset(1) NE `}`
            ASSIGN data TO <table>.
            CLEAR <table>.
            CREATE DATA line LIKE LINE OF <table>.
            ASSIGN line->* TO <line>.
            lt_fields = get_fields( type descr = data descr data =
            line ).
            IF table descr->key_defkind EQ
            table descr->keydefkind_user AND lines( table descr->key ) EQ 1.
              READ TABLE table descr->key INDEX 1 INTO key_name.
              READ TABLE lt_fields WITH TABLE KEY name = key_name
            ASSIGNING <field>.
              key_ref = <field>-value.
IF mv_assoc_arrays_opt EQ c_bool-true.
lt_symbols = get_symbols( type_descr = data_descr
data = line ).
DELETE lt_symbols WHERE name EQ key_name.
IF lines( lt_symbols ) EQ 1.
READ TABLE lt_symbols INDEX 1 ASSIGNING
<value_sym>. 
ENDIF.
ENDIF.
ENDIF.
WHILE offset < length AND json+offset(1) NE `}`).
CLEAR <line>.
eat_white.
eat_string key_value.
eat_white.
eat_char `:`.
eat_white.
IF <value_sym> IS ASSIGNED.
ASSIGN <value_sym>-value->* TO <value>.
restore_type( EXPORTING json = json length = length
 type_descr = <value_sym>-type
CHANGING data = <value> offset =
offset ).
ELSE.
restore_type( EXPORTING json = json length = length
type_descr = data_descr field_cache = lt_fields
CHANGING data = <line> offset =
offset ).
ENDIF.
IF table_descr->key_defkind EQ
 table_descr->keydefkind_user.
IF key_ref IS BOUND.
ASSIGN key_ref->* TO <value>.
IF <value> IS INITIAL.
MOVE key_value TO <value>.
ENDIF.
ENDIF.
ELSEIF <line> IS INITIAL.
MOVE key_value TO <line>.
ENDIF.
INSERT <line> INTO TABLE <table>.
eat_white.
IF offset < length AND json+offset(1) NE `}`)`.
eat_char `,`.
ELSE.
EXIT.
ENDIF.
ENDWHILE.
ELSE.
CLEAR data.
ENDIF.
eat_char `}`.

ELSE.
    restore( EXPORTING json = json length = length CHANGING
    offset = offset ).
ENDIF.
ELSEIF type_descr->type_kind EQ
cl_abap_typedescr=>typekind_dref.
    IF data IS INITIAL.
        generate_int_ex( EXPORTING json = json length = length
    CHANGING offset = offset data = data ).
    ELSE.
        data_ref ?= data.
        type_descr = cl_abap_typedescr=>describe_by_data_ref( data_ref ).
        ASSIGN data_ref->* TO <data>.
        restore_type( EXPORTING json = json length = length
    type_descr = type_descr CHANGING data = <data> offset = offset ).
    ENDIF.
ELSE.
    restor
WHILE offset < length AND json+offset(1) NE '}]'.
  CLEAR <line>.
  restore_type( EXPORTING json = json length = length
type_descr = data_descr field_cache = lt_fields
                CHANGING data = <line> offset = offset ).
  INSERT <line> INTO TABLE <table>.
  eat_white.
  IF offset < length AND json+offset(1) NE '}]'.
    eat_char ', '.
  ELSE.
    EXIT.
  ENDIF.
ENDWHILE.
ELSE.
  " skip array
WHILE offset < length AND json+offset(1) NE '})'.
  eat_white.
  restore_type( EXPORTING json = json length = length
CHANGING offset = offset ).
  eat_white.
  IF offset < length AND json+offset(1) NE '}]'.
    eat_char ', '.
  ELSE.
    EXIT.
  ENDIF.
ENDWHILE.
  IF type_descr IS NOT INITIAL.
    eat_char '}]'.
    throw_error.
  ENDIF.
ENDIF.
ELSE.
  CLEAR data.
ENDIF.
  eat_char '}]'.
ENDIF.
WHEN `"` " string
  eat_string sdummy.
  IF type_descr IS NOT INITIAL.
    " unescape string
    IF sdummy IS NOT INITIAL.
      IF type_descr->kind EQ cl_abap_typedescr->kind_elem.
        elem_descr != type_descr.
        CASE elem_descr->type_kind.
          WHEN cl_abap_typedescr->typekind_char.
            IF elem_descr->output_length EQ 1 AND mc_bool_types
            CS elem_descr->absolute_name.
              IF sdummy(1) CA `XxTt1`.
                data = c_bool-true.
              ELSE.
                data = c_bool-false.
              ENDIF.
            ELSE.
              data = c_bool-true.
            ENDIF.
RETURN.
ENDIF.
WHEN cl_abap_typedescr=>typekind_xstring OR
   cl_abap_typedescr=>typekind_hex.
   string_to_xstring( EXPORTING in = sdummy CHANGING out = data ).
   RETURN.
WHEN cl_abap_typedescr=>typekind_date.
   REPLACE FIRST OCCURRENCE OF REGEX
   `\(\d\(4\)\)-\(\d\(2\)\)-\(\d\(2\)\)` IN sdummy WITH `\$1\$2\$3`
   REPLACEMENT LENGTH match REPLACEMENT OFFSET pos. "#EC
   IF sy-subrc EQ 0 AND pos EQ 0.
      sdummy = sdummy(match).
   ENDIF.
WHEN cl_abap_typedescr=>typekind_time.
   REPLACE FIRST OCCURRENCE OF REGEX
   `\(\d\(2\)\):(\d\(2\)\):(\d\(2\)\)` IN sdummy WITH `\$1\$2\$3`
   REPLACEMENT LENGTH match REPLACEMENT OFFSET pos. "#EC
   IF sy-subrc EQ 0 AND pos EQ 0.
      sdummy = sdummy(match).
   ENDIF.
WHEN cl_abap_typedescr=>typekind_packed.
   REPLACE FIRST OCCURRENCE OF REGEX
   `\(\d\(4\)\)-?\(\d\(2\)\)-?\(\d\(2\)\)T\(\d\(2\)\):?\(\d\(2\)\):?\(\d\(2\)\):?\(\d\(0,7\)\)` IN sdummy WITH `\$1\$2\$3\$4\$5\$6\$.7`
   REPLACEMENT LENGTH match REPLACEMENT OFFSET pos. "#EC
   IF sy-subrc EQ 0 AND pos EQ 0.
      sdummy = sdummy(match).
   ENDIF.
ENDCASE.
ELSEIF type_descr->type_kind EQ
   cl_abap_typedescr=>typekind_dref.
   CREATE DATA lr_sdummy TYPE string.
   MOVE sdummy TO lr_sdummy->*.
   data ?= lr_sdummy.
   RETURN.
ELSE.
   throw_error. " Other wise dumps with
   OBJECTS_MOVE_NOT_SUPPORTED
   ENDIF.
   MOVE sdummy TO data.
   ELSEIF type_descr->kind EQ cl_abap_typedescr=>kind_elem.
      CLEAR data.
   ELSE.
      throw_error. " Other wise dumps with
   OBJECTS_MOVE_NOT_SUPPORTED
   ENDIF.
   WHEN `\-`.
      " number
      IF type_descr IS NOT INITIAL.
   ELSE.
      throw_error. " Other wise dumps with
   OBJECTS_MOVE_NOT_SUPPORTED
IF type_descr->kind EQ type_descr->kind_ref AND type_descr->type_kind EQ cl_abap_typedescr=>typekind_dref.
CREATE DATA lr_idummy TYPE i.
edate lr_idummy->*.
ENDIF.
ELSEIF type_descr->kind EQ type_descr->kind_elem.
edate lr_idummy.
ELSE.
et numb sdummy.
throw erros.
ENDIF.
ELSE.
et numb sdummy.
ENDIF.
WHEN OTHERS.
FIND FIRST OCCURRENCE OF json+offset(1) IN `0123456789`.
IF sy-subrc IS INITIAL. " number
IF type_descr IS NOT INITIAL.
IF type_descr->kind EQ type_descr->kind_ref AND type_descr->type_kind EQ cl_abap_typedescr=>typekind_dref.
CREATE DATA lr_idummy TYPE i.
edate lr_idummy->*.
ENDIF.
ELSEIF type_descr->kind EQ type_descr->kind_elem.
edate lr_idummy.
ELSE.
et numb sdummy.
throw erros.
ENDIF.
ELSE.
et numb sdummy.
ENDIF.
ELSE. " true/false/null
IF type_descr IS NOT INITIAL.
IF type_descr->kind EQ type_descr->kind_ref AND type_descr->type_kind EQ cl_abap_typedescr=>typekind_dref.
CREATE DATA lr_bdummy TYPE bool.
edate lr_bdummy->*.
ENDIF.
ELSEIF type_descr->kind EQ type_descr->kind_elem.
edate lr_bdummy.
ELSE.
et bool sdummy.
throw error.
ENDIF.
ELSE.
et bool sdummy.
ENDIF.
ENDIF.
ENDCASE.
CATCH cx_sy_move_cast_error cx_sy_conversion_no_number cx_sy_conversion_overflow INTO lo_exp.
CLEAR data.
IF mv_strict_mode EQ abap_true.
   RAISE EXCEPTION TYPE cx_sy_move_cast_error EXPORTING previous = lo_exp.
ENDIF.
ENDTRY.
ENDMETHOD.                    "restore_type

METHOD bool_to_tribool.
   IF iv_bool EQ c_bool-true.
      rv_tribool = c_tribool-true.
   ELSEIF iv_bool EQ abap_undefined. " fall back for abap_bool
      rv_tribool = c_tribool-undefined.
   ELSE.
      rv_tribool = c_tribool-false.
   ENDIF.
ENDMETHOD.                    "bool_to_tribool

METHOD tribool_to_bool.
   IF iv_tribool EQ c_tribool-true.
      rv_bool = c_bool-true.
   ELSEIF iv_tribool EQ c_tribool-undefined.
      rv_bool = abap_undefined. " fall back to abap_undefined
   ENDIF.
ENDMETHOD.                    "TRIBOOL_TO_BOOL

METHOD raw_to_string.
   DATA: lv_output_length TYPE i,
       lt_binary_tab    TYPE STANDARD TABLE OF sdokcntbin.
   CALL FUNCTION 'SCMS_XSTRING_TO_BINARY'
      EXPORTING
         buffer        = iv_xstring
      IMPORTING
         output_length = lv_output_length
      TABLES
         binary_tab    = lt_binary_tab.
   CALL FUNCTION 'SCMS_BINARY_TO_STRING'
      EXPORTING
         input_length  = lv_output_length
         encoding      = iv_encoding
      IMPORTING
         text_buffer   = rv_string
         output_length = lv_output_length
      TABLES
         binary_tab    = lt_binary_tab.
ENDMETHOD.

METHOD string_to_raw.
CALL FUNCTION 'SCMS_STRING_TO_XSTRING'
  EXPORTING
    text     = iv_string
    encoding = iv_encoding
  IMPORTING
    buffer   = rv_xstring
  EXCEPTIONS
    OTHERS  = 1.

  IF sy-subrc IS NOT INITIAL.
    CLEAR rv_xstring.
  ENDIF.
ENDMETHOD.

METHOD string_to_xstring.
  DATA: lv_xstring TYPE xstring.

  CALL FUNCTION 'SSFC_BASE64_DECODE'
    EXPORTING
      b64data = in
    IMPORTING
      bindata = lv_xstring
    EXCEPTIONS
      OTHERS  = 1.

  IF sy-subrc IS INITIAL.
    MOVE lv_xstring TO out.
  ELSE.
    MOVE in TO out.
  ENDIF.
ENDMETHOD.  

METHOD xstring_to_string.
  DATA: lv_xstring TYPE xstring.

  " let us fix data conversion issues here
  lv_xstring = in.

  CALL FUNCTION 'SSFC_BASE64_ENCODE'
    EXPORTING
      bindata = lv_xstring
    IMPORTING
      b64data = out
    EXCEPTIONS
      OTHERS  = 1.

  IF sy-subrc IS NOT INITIAL.
    MOVE in TO out.
  ENDIF.
Custom ABAP to JSON, JSON to ABAP name mapping

By default, you control the way JSON names formatted/mapped to ABAP names by selecting proper pretty_mode as a parameter for SERIALIZE/DESERIALIZE/GENERATE method. But in some cases, the standard, hard-coded formatting, is not enough. For example, if you need special rules for name formatting (for using special characters) or because JSON attribute name is too long and you can not map it to ABAP name (which has 30 characters length limit).

The recommended way for custom mapping was an extension of the /UI2/CL_JSON class and redefining methods PRETTY_NAME or PRETTY_NAME_EX, but since note 2526405 there is an easier way, without the need in own class. If you have a static list of field mappings from ABAP to JSON you can pass name mapping table as a parameter for the constructor/serialize/deserialize and control the way JSON names formatted/mapped to ABAP names.

**ABAP to JSON name mapping**

```plaintext
TYPES:
  BEGIN OF tp_s_data,
    sschema           TYPE string,
    odatacontext      TYPE string,
    shortened_abap_name TYPE string,
    standard          TYPE string,
  END OF tp_s_data.

DATA: ls_exp         TYPE tp_s_data,
      lt_mapping     TYPE /ui2/cl_json=>name_mappings,
      lv_json        TYPE /ui2/cl_json=>json.

lt_mapping = VALUE #( ( abap = `SSCHEMA` json = `$schema` )
                      ( abap = `ODATACONTEXT` json = `@odata.context` )
                      ( abap = `SHORTENED_ABAP_NAME` json = `VeeeeryyyyyLooooongJSONAttrbuuuuuuuuuuuuuutteeeee` ) ).

lv_json = /ui2/cl_json=>serialize( data = ls_exp name_mappings = lt_mapping ).
```

Serialization/deserialization of hierarchical/recursive data

Handling of the recursive data structure in ABAP is not very trivial. And it is not very trivial to serialize and deserialize it either. If you would like to model your hierarchical data (tree-like) as ABAP structures, the only allowed way will be to do it like in the example below, where you use references to generic data:
Modeling of recursive data types in ABAP

TYPES:
   BEGIN OF ts_node,
      id        TYPE i,
      children  TYPE STANDARD TABLE OF REF TO data WITH DEFAULT KEY,
   END OF ts_node.

DATA: lv_exp    TYPE string,
      lv_act    TYPE string,
      ls_data   TYPE ts_node,
      lr_data   LIKE REF TO ls_data.

ls_data-id = 1.

CREATE DATA lr_data.
lr_data->id = 2.
APPEND lr_data TO ls_data-children.

Such way more or less straightforward and will work, but leads to losing type information for data persisted in children table. That will mean that you will need to cast data when you access it. In addition to that, it blocks you from being able to deserialize such data from JSON, while parser will not be able to deduce the type of the data needs to be created in children table. But serialization will work fine:

Serialization of recursive ABAP structures

lv_exp = '{"ID":1,"CHILDREN":[{"ID":2,"CHILDREN":[]}]}'
lv_act = /ui2/cl_json=>serialize( data = ls_data ).
claunit_assert=>assert_equals( act = lv_act exp = lv_exp msg = 'Serialization of recursive data structure fails' ).

The better way to model hierarchical data in ABAP is with help of objects, while objects are always processed as references and ABAP allow you to create nested data structures, referring to objects of the same type:

Modeling of recursive data in ABAP using objects

CLASS lcl_test DEFINITION FINAL.
PUBLIC SECTION.
   DATA: id TYPE i.
   DATA: children TYPE STANDARD TABLE OF REF TO lcl_test.
ENDCLASS.

In that manner, you are able to process data in the same way as with ABAP structures but using typed access and serialization/deserialization of data in JSON works fine while types can be deduced on
### Serialization/deserialization of recursive objects in ABAP

```abap
DATA: lo_act TYPE REF TO lcl_test,
     lo_exp TYPE REF TO lcl_test,
     lv_json TYPE string,
     lo_child LIKE lo_data.

CREATE OBJECT lo_exp.
lo_exp ->id = 1.

CREATE OBJECT lo_child.
lo_child->id = 2.
APPEND lo_child TO lo_exp->children.

lv_json = /ui2/cl_json=>serialize( data = lo_exp ).
ui2/cl_json=>deserialize( EXPORTING json = lv_json CHANGING data =  lo_act ).
```

Remark: There are some constraints for data design exist in regard to the deserialization of objects:

- You cannot use constructors with obligatory parameters
- References to interfaces will be not deserialized

### Partial serialization/deserialization

When it is needed:

- You deserialize JSON to ABAP but would like some known parts to be deserialized as JSON string, while you do not know nesting JSON structure.
- You deserialize a collection (array/associative array) which has objects with heterogeneous structure (for example the same field has different type depending on object type). Using partial deserialization, you can restore such type as JSON string in ABAP and apply later additional deserialization based on the object type.
- You serialize ABAP to JSON and have some ready JSON pieces (strings) which you would like to mix in.

The solution /UI2/CL_JSON has for this type /UI2/CL_JSON=>JSON (alias for built-in type string). ABAP fields using declared with this type will be serialized/deserialized as JSON pieces. Be aware that during serialization from ABAP to JSON, the content of such JSON piece is not validated for correctness, so if you pass invalid JSON block, it may destroy whole resulting JSON string at the end.

Below you can find examples for partial serialization/deserialization.

**Serialization:**
Partial serialization of ABAP to JSON

TYPES: BEGIN OF ts_record,
    id      TYPE string,
    columns TYPE /ui2/cl_json=>json,
END OF ts_record.

DATA: lv_json   TYPE /ui2/cl_json=>json,
     lt_data   TYPE SORTED TABLE OF ts_record WITH UNIQUE KEY id,
     ls_data   LIKE LINE OF lt_data.

ls_data-id = '0000001ZZ_SO_GRES_CONTACTS'.
ls_data-columns =
  '
"AGE":{"bVisible":true,"iPosition":2},"BRSCH":{"bVisible":true}}'.
INSERT ls_data INTO TABLE lt_data.

ls_data-id = '0000001ZZ_TRANSIENT_TEST_A'.
ls_data-columns =
  '
"ABTNR":{"bVisible":false},"CITY1":{"bVisible":false},"IC_COMPANY_KEY":{
"bVisible":true}}'.
INSERT ls_data INTO TABLE lt_data.

lv_json = /ui2/cl_json=>serialize( data = lt_data assoc_arrays = abap_true
pretty_name = /ui2/cl_json=>pretty_mode-camel_case )

WRITE / lv_json.
Deserialization:
Partial deserialization of JSON into ABAP

```
TYPES: BEGIN OF ts_record,
  id      TYPE string,
  columns TYPE /ui2/cl_json=>json,
END OF ts_record.

DATA: lv_json  TYPE string,
     lt_act   TYPE SORTED TABLE OF ts_record WITH UNIQUE KEY id.

CONCATENATE
  '"O000001ZZ_SO_GRES_CONTACTS":{"columns":{"AGE":{"bVisible":true,"iPosition":2},"BRSCH":{"bVisible":true}}},'
  '"O000001ZZ_TRANSIENT_TEST_A":{"columns":{"ABTNR":{"bVisible":false},"CITY1":{"bVisible":false},"IC_COMPANY_KEY":{"bVisible":true}}}}'
INTO lv_json.

" if you know first level of undelying structure ("columns" field) -> Output Var 1
/ui2/cl_json=>deserialize( EXPORTING json = lv_json assoc_arrays = abap_true CHANGING data = lt_act ).

" if you do not know underlying structure of first level (naming of second filed e.g columns in example does not matter )
" => result is a little bit different -> Output Var 2
/ui2/cl_json=>deserialize( EXPORTING json = lv_json assoc_arrays = abap_true assoc_arrays_opt = abap_true CHANGING data = lt_act ).

Results in the following ABAP data:

**ABAP Output (variant 1)**

<table>
<thead>
<tr>
<th>ID(CString)</th>
<th>COLUMNS(CString)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O000001ZZ_SO_GRES_CONTACTS</td>
<td>&quot;{&quot;columns&quot;:{&quot;AGE&quot;:{&quot;bVisible&quot;:true,&quot;iPosition&quot;:2},&quot;BRSCH&quot;:{&quot;bVisible&quot;:true}}}&quot;</td>
</tr>
<tr>
<td>O000001ZZ_TRANSIENT_TEST_A</td>
<td>&quot;{&quot;columns&quot;:{&quot;ABTNR&quot;:{&quot;bVisible&quot;:false},&quot;CITY1&quot;:{&quot;bVisible&quot;:false},&quot;IC_COMPANY_KEY&quot;:{&quot;bVisible&quot;:true}}}&quot;</td>
</tr>
</tbody>
</table>

**ABAP Output (variant 2)**

<table>
<thead>
<tr>
<th>ID(CString)</th>
<th>COLUMNS(CString)</th>
</tr>
</thead>
<tbody>
<tr>
<td>O000001ZZ_SO_GRES_CONTACTS</td>
<td>{&quot;columns&quot;:{&quot;AGE&quot;:{&quot;bVisible&quot;:true,&quot;iPosition&quot;:2},&quot;BRSCH&quot;:{&quot;bVisible&quot;:true}}}}</td>
</tr>
<tr>
<td>O000001ZZ_TRANSIENT_TEST_A</td>
<td>{&quot;columns&quot;:{&quot;ABTNR&quot;:{&quot;bVisible&quot;:false},&quot;CITY1&quot;:{&quot;bVisible&quot;:false},&quot;IC_COMPANY_KEY&quot;:{&quot;bVisible&quot;:true}}}}</td>
</tr>
</tbody>
</table>

" if you know first level of undelying structure ("columns" field) -> Output Var 1
" if you do not know underlying structure of first level (naming of second filed e.g columns in example does not matter )
" => result is a little bit different -> Output Var 2
" if you do not know underlying structure of first level (naming of second filed e.g columns in example does not matter )
" => result is a little bit different -> Output Var 2
" if you do not know underlying structure of first level (naming of second filed e.g columns in example does not matter )
" => result is a little bit different -> Output Var 2
If standard class functionality does not fit your requirements there are two ways of how you can adapt it to your needs:

- Use a local copy of the class /UI2/CL_JSON and modify logic directly, by the change of original code.
- Inherit from class /UI2/CL_JSON and override methods where another logic is required.

The advantage of the first approach that you are completely free in what you may change and have full control of class lifecycle. The disadvantage, you will probably need to merge your changes with /UI2/CL_JSON updates.

For the second approach you can use /UI2/CL_JSON directly (prerequisite is the latest version of note 2330592), do not need to care about the merge, but can override only some methods. The methods are:

**IS_COMPRESSIBLE** – called to check, if given type output may be suppressed during ABAP to JSON serialization when a value is initial.

- > **TYPE_DESCR** (ref to CL_ABAP_TYPEDESCRIPTOR) – value type
- < **RV_COMPRESS** (bool) – compress initial value

The default implementation of the method allows compressing any initial value.

**PRETTY_NAME** – called to format ABAP field name written to JSON or deserialized from JSON to ABAP field, when the pretty_name parameter of **SERIALIZE/DESERIALIZE** method equal to **PRETTY_MODE-CAMEL_CASE**.

- > **IN** (CSEQUENCE) – Field name to pretty print.
- < **OUT** (STRING) – Pretty printed field name

The default implementation applies camelCase formatting, based on usage of “_” symbol. To output “_” symbol, use double “__” symbol in the field name.

**PRETTY_NAME_EX** – called to format ABAP field name written to JSON or deserialized from JSON to ABAP field, when the pretty_name parameter of **SERIALIZE/DESERIALIZE** method equal to **PRETTY_MODE-EXTENDED**.

- > **IN** (CSEQUENCE) – Field name to pretty print.
- < **OUT** (STRING) – Pretty printed field name

The default implementation does the same as **PRETTY_NAME**, plus converting special characters "!#$%&*-~/@.".

**DUMP_INT** - called for recursive serialization of complex ABAP data objects (structure, class, table) into JSON string

- > **DATA** (DATA) – Any data to serialize.
- > **TYPE_DESCR** (ref to CL_ABAP_TYPEDESCRIPTOR, optional) – Type of data provided
- < **R_JSON** (JSON) – serialized JSON value

**DUMP_TYPE** - called for serialization of elementary ABAP data type (string, boolean, timestamp, etc) into the JSON attribute value. Overwrite it if you, for example, want to apply data output data conversion of currency rounding

- > **DATA** (DATA) – Any data to serialize
- > **TYPE_DESCR** (ref to CL_ABAP_TYPEDESCRIPTOR) – Type of data provided
- < **R_JSON** (JSON) – serialized JSON value

**RESTORE** - called for deserializing JSON objects into ABAP structures

- > **JSON** (JSON) – JSON string to deserialize
- > **LENGHT** (I) – Length of the JSON string
- > **TYPE_DESCR** (ref to CL_ABAP_TYPEDESCRIPTOR, optional) – Type of changing data provided
- > **FIELD_CACHE** (type T_T_FIELD_CACHE, optional) – Cache of ABAP data fields with type information
- <= **DATA** (type DATA, optional) – ABAP data object to fill
- <= **OFFSET** (I) – parsing start point in JSON string

**RESTORE_TYPE** - called to deserializing simple JSON attributes and JSON arrays

- > **JSON** (JSON) – JSON string to deserialize
- > **LENGHT** (I) – Length of the JSON string
- > **TYPE_DESCR** (ref to CL_ABAP_TYPEDESCRIPTOR, optional) – Type of changing data provided
- > **FIELD_CACHE** (type T_T_FIELD_CACHE, optional) – Cache of ABAP data fields with type information
- <= **DATA** (type DATA, optional) – ABAP data object to fill
- <= **OFFSET** (I) – parsing start point in JSON string

**CLASS_CONSTRUCTOR** - used to initialize static variables. You can not overwrite it, but you can implement your own class constructor that adapts default globals. For example, adds additional boolean type to be recognized during serialization/deserialization.

**SERIALIZE/DESERIALIZE** - these methods are static therefore cannot be redefined. Methods are helpers for a consumption code, hiding the construction of the class instance and further "_INT" calls. So, if you would like to use something similar, in your custom class, you need to copy
mentioned methods to new ones e.g * _EX and overwrite there /UI2/CL_JSON type to your custom class name. And use these methods instead of standard.

Extension using inheritance:

<table>
<thead>
<tr>
<th>Extension of /UI2/CL_JSON</th>
</tr>
</thead>
<tbody>
<tr>
<td>CLASS lc_json_custom DEFINITION FINAL INHERITING FROM /ui2/cl_json.</td>
</tr>
<tr>
<td>PUBLIC SECTION.</td>
</tr>
<tr>
<td>CLASS-METHODS:</td>
</tr>
<tr>
<td>class_constructor,</td>
</tr>
<tr>
<td>deserialize_ex IMPORTING json TYPE json OPTIONAL</td>
</tr>
<tr>
<td>pretty_name TYPE pretty_name_mode DEFAULT</td>
</tr>
<tr>
<td>pretty_mode-none</td>
</tr>
<tr>
<td>CHANGING data TYPE data,</td>
</tr>
<tr>
<td>serialize_ex IMPORTING data TYPE data</td>
</tr>
<tr>
<td>compress TYPE bool DEFAULT c_bool-false</td>
</tr>
<tr>
<td>pretty_name TYPE pretty_name_mode DEFAULT</td>
</tr>
<tr>
<td>pretty_mode-none</td>
</tr>
<tr>
<td>RETURNING value(r_json) TYPE json .</td>
</tr>
<tr>
<td>PROTECTED SECTION.</td>
</tr>
<tr>
<td>METHODS:</td>
</tr>
<tr>
<td>is_compressable REDEFINITION,</td>
</tr>
<tr>
<td>pretty_name REDEFINITION.</td>
</tr>
<tr>
<td>ENDCCLASS.</td>
</tr>
</tbody>
</table>

CLASS lc_json_custom IMPLEMENTATION. |
METHOD class_constructor. |
CONCATENATE mc_bool_types `\TYPE=/UI2/BOOLEAN` INTO mc_bool_types. |
ENDMETHOD. |
METHOD is_compressable. |
IF type_descr->absolute_name EQ `\TYPE=STRING` OR name EQ `INITIAL`. |
rv_compress = abap_false. |
ELSE. |
rv_compress = abap_true. |
ENDIF. |
ENDMETHOD. |
METHOD pretty_name. |
out = super->pretty_name( in ). |
CONCATENATE out 'Xxx' INTO out. |
ENDMETHOD. |
METHOD serialize_ex. |
DATA: lo_json TYPE REF TO lc_json_custom. |
CREATE OBJECT lo_json |
EXPORTING |
compress = compress |
pretty_name = pretty_name |
assoc_arrays = abap_true |
assoc_arrays_opt = abap_true |
expand_includes = abap_true |
umc_as_string = abap_true |
ts_as_iso8601 = abap_true. |
r_json = lo_json->serialize_int( data = data ). |
METHOD serialize_ex.
    DATA: lo_json TYPE REF TO lc_json_custom.
    IF json IS NOT INITIAL.
        CREATE OBJECT lo_json
            EXPORTING
                pretty_name = pretty_name
                assoc_arrays = abap_true
                assoc_arrays_opt = abap_true.
        TRY .
            lo_json->deserialize_int( EXPORTING json = json CHANGING data = data ).
        CATCH cx_sy_move_cast_error.
        ENDTRY.
    ENDIF.
ENDMETHOD.                    "deserialize_ex

METHOD deserialize_ex.
    DATA: lo_json TYPE REF TO lc_json_custom.
    IF json IS NOT INITIAL.
        CREATE OBJECT lo_json
            EXPORTING
                pretty_name      = pretty_name
                assoc_arrays     = abap_true
                assoc_arrays_opt = abap_true.
        TRY .
            lo_json->deserialize_int( EXPORTING json = json CHANGING data = data ).
        CATCH cx_sy_move_cast_error.
        ENDTRY.
    ENDIF.
ENDMETHOD.                    "deserialize_ex

ENDCLASS.                    "lc_json_custom IMPLEMENTATION

TYPES:
    BEGIN OF tp_s_data,
        tribool   TYPE lc_json_custom=>tribool,
        bool      TYPE lc_json_custom=>bool,
        str1      TYPE string,
        str2      TYPE string,
        initial   TYPE i,
    END OF tp_s_data.

DATA: ls_exp          TYPE tp_s_data,
    ls_act          LIKE ls_exp,
    lo_json_custom  TYPE REF TO lc_json_custom,
    lv_json_custom  TYPE lc_json_custom=>json.

ls_exp-tribool = lc_json_custom=>c_tribool-false.
ls_exp-bool    = lc_json_custom=>c_bool-false.
ls_exp-str1    = ''. 
ls_exp-str2    = 'ABC'.
ls_exp-initial = 0.
CREATE OBJECT lo_json_custom
    EXPORTING
        compress    = abap_true
        pretty_name = lc_json_custom=>pretty_mode-camel_case.

lv_json_custom = lo_json_custom->serialize_int( data = ls_exp ).
lo_json_custom->deserialize_int( EXPORTING json = lv_json_custom CHANGING data = ls_act ).

" alternative way
lc_json_custom=>deserialize_ex( EXPORTING json = lv_json_custom CHANGING data = ls_act ).
c1_aunit_assert=>assert_equals( act = ls_act exp = ls_exp msg = 'Custom
Deserialization of an untyped (unknown) JSON object

If you need to deserialize a JSON object with unknown structure, or you do not have passing data type on ABAP side or data type of the resulting object may vary, you can generate ABAP object on the fly, using corresponding GENERATE method. The method has some limitations comparing to standard deserialization like:

- all fields are generated as a reference (even elementary types)
- you can not control how deserialized arrays or timestamps
- you can not access components of generated structure statically (while the structure is unknown at compile time) and need to use dynamic access

The simplest example, with straightforward access:
### Generating of ABAP Data using /UI2/CL_JSON

```abap
DATA: lv_json TYPE /ui2/cl_json=>json,
     lr_data TYPE REF TO data.

FIELD-SYMBOLS:
   <data>   TYPE data,
   <struct> TYPE any,
   <field>  TYPE any.

lv_json =
  '{"name":"Key1","properties":{"field1":"Value1","field2":"Value2"}}'.
lr_data = /ui2/cl_json=>generate( json = lv_json ).

" OK, generated, now let us access somete field :( 
IF lr_data IS BOUND.
  ASSIGN lr_data->* TO <data>.
  ASSIGN COMPONENT `PROPERTIES` OF STRUCTURE <data> TO <field>.
  IF <field> IS ASSIGNED.
    lr_data = <field>.
    ASSIGN lr_data->* TO <data>.
    ASSIGN COMPONENT `FIELD1` OF STRUCTURE <data> TO <field>.
    IF <field> IS ASSIGNED.
      lr_data = <field>.
      ASSIGN lr_data->* TO <data>.
      WRITE: <data>. " We got it -> Value1 
    ENDF.
  ENDF.
ENDIF.
ENDIF.
```

A nice alternative, using dynamic data accessor helper class:

```abap
DATA: lv_json TYPE /ui2/cl_json=>json,
     lr_data TYPE REF TO data,
     lv_val  TYPE string.

lv_json =
  '{"name":"Key1","properties":{"field1":"Value1","field2":"Value2"}}'.
lr_data = /ui2/cl_json=>generate( json = lv_json ).

/ui2/cl_data_access=>create( ir_data = lr_data iv_component =
  `properties-field1`)->value( IMPORTING ev_data = lv_val ).
WRITE: lv_val.
```

### Access generated ABAP data object using dynamic data accessor helper

```abap
DATA: lv_json TYPE /ui2/cl_json=>json,
     lr_data TYPE REF TO data,
     lv_val  TYPE string.

lv_json =
  '{"name":"Key1","properties":{"field1":"Value1","field2":"Value2"}}'.
lr_data = /ui2/cl_json=>generate( json = lv_json ).

Write: /ui2/cl_data_access=>create( ir_data = lr_data iv_component =
  `properties-field1` )->value( IMPORTING ev_data = lv_val ).
WRITE: lv_val.
```

### Implicit generation of ABAP objects on deserialization

In addition to the explicit generation of the ABAP data objects from JSON string, the deserializer supports an implicit way of generation, during DESERIALIZE(INT) call. To trigger generation, your output data structure shall contain a field with type REF TO DATA and name of the field shall
match JSON attribute (pretty name rules are considered). Depending on the value of the field, the behavior may differ:

- The value is not bound (initial): deserialize will use generation rules when creating corresponding data types of the referenced value
- The value is bound (but may be empty): the deserializer will create new referenced value based on the referenced type.

**Example of implicit generation of ABAP data from JSON string**

```plaintext
TYPES:
  BEGIN OF ts_dyn_data1,
    name     TYPE string,
    value    TYPE string,
  END OF ts_dyn_data1,
  BEGIN OF ts_dyn_data2,
    key      TYPE string,
    value    TYPE string,
  END OF ts_dyn_data2,
  BEGIN OF ts_data,
    str     TYPE string,
    data    TYPE REF TO data,
  END OF ts_data.

DATA:
  ls_data  TYPE ts_data,
  lv_json  TYPE /ui2/cl_json=>json.

lv_json = `{"str":"Test","data":{"name":"name1","value":"value1"}}`.

" deserialize data and use generic generation for field "data",
" the same as with method GENERATE (using temporary data type)
/ui2/cl_json=>deserialize( EXPORTING json = lv_json CHANGING data = ls_data )
.

" deserialize data and use type TS_DYN_DATA1 for the field "data"
CREATE DATA ls_data-data TYPE ts_dyn_data1.
/ui2/cl_json=>deserialize( EXPORTING json = lv_json CHANGING data = ls_data )
.

" deserialize data and use alternative type TS_DYN_DATA2 for the field "data"
CREATE DATA ls_data-data TYPE ts_dyn_data2.
/ui2/cl_json=>deserialize( EXPORTING json = lv_json CHANGING data = ls_data )
.

**JSON/ABAP serialization/deserialization with runtime type information**

Automatic deserialization of the JSON into appropriate ABAP structure is not supported. The default implementation assumes that you need to know target data structure (or at least partial structure, it will also work) to deserialize JSON in ABAP and then work with typed data.

But if for some reason one needs the ability to deserialize JSON in source ABAP structure in a generic way, he can extend both serialize/deserialize methods and wrap outputs/inputs of /UI2/CL_JSON data by technical metadata describing source ABAP structure and use this information during deserialization (or use GENERATE method). Of course, you need to ensure that source ABAP data type is known in deserialization scope (global and local types are "visible").

See example below:
Serialization and deserialization with runtime type information

```abap
TYPES: BEGIN OF ts_json_meta,
  abap_type LIKE cl_abap_typedescr=>absolute_name,
  data      TYPE string,
END OF ts_json_meta.

DATA: lt_flight TYPE STANDARD TABLE OF sflight,
    lv_json   TYPE string,
    lo_data   TYPE REF TO data,
    ls_json   TYPE ts_json_meta.

FIELD-SYMBOLS: <data> TYPE any.

SELECT * FROM sflight INTO TABLE lt_flight.

* serialize table lt_flight into JSON, skipping initial fields and
conveting ABAP field names into camelCase
ls_json-data      = /ui2/cl_json=>serialize( data = lt_flight compress =
abap_true pretty_name = /ui2/cl_json=>pretty_mode-camel_case ).
ls_json-abap_type = cl_abap_typedescr=>describe_by_data( lt_flight
)->absolute_name.
lv_json           = /ui2/cl_json=>serialize( data = ls_json compress =
abap_true pretty_name = /ui2/cl_json=>pretty_mode-camel_case ).
WRITE / lv_json.

CLEAR: ls_json, lt_flight.

* deserialize JSON string json into internal table lt_flight doing
camelCase to ABAP like field name mapping
/ui2/cl_json=>deserialize( EXPORTING json = lv_json pretty_name =
/ui2/cl_json=>pretty_mode-camel_case CHANGING data = ls_json ).
CREATE DATA lo_data TYPE (ls_json-abap_type).
ASSIGN lo_data->* TO <data>.
/ui2/cl_json=>deserialize( EXPORTING json = ls_json-data pretty_name =
/ui2/cl_json=>pretty_mode-camel_case CHANGING data = <data> ).

IF lo_data IS NOT INITIAL.
  BREAK-POINT. " check here lo_data
ENDIF.
```

Exception Handling in /UI2/CL_JSON

By default, /UI2/CL_JSON tries to hide from consumer thrown exceptions (that may happen during deserialization) catching them at all levels. In some cases, it will result in missing attributes, in other cases, when an error was critical and the parser can not restore, you will get an empty object back. The main TRY/CATCH block, not letting exceptions out is in DESERIALIZE method.

If you want to get a reporting in case of error, you shall use instance method DESERIALIZE_INT which may fire CX_SY_MOVE_CAST_ERROR. The reporting is rather limited - all errors translated into CX_SY_MOVE_CAST_ERROR and no additional information available.

JSON to ABAP transformation with the use of CALL TRANSFORMATION
Below is a small example of CALL TRANSFORMATION usage to produce JSON from ABAP structures. Do not ask me details - I do not know them. 😊 Was just a small test of me.

CALL TRANSFORMATION for JSON

```abap
DATA: lt_flight          TYPE STANDARD TABLE OF sflight,
     lo_writer          TYPE REF TO cl_sxml_string_writer,
     lv_output_length  TYPE i,
     lt_binary_tab     TYPE STANDARD TABLE OF sdokcntbin,
     lv_jsonx          TYPE xstring,
     lv_json           TYPE string.

SELECT * FROM sflight INTO TABLE lt_flight.

* ABAP to JSON
lo_writer = cl_sxml_string_writer=>create( type = if_sxml=>co_xt_json ).
CALL TRANSFORMATION id SOURCE text = lt_flight RESULT XML lo_writer.
lv_jsonx = lo_writer->get_output( ).

CALL FUNCTION 'SCMS_XSTRING_TO_BINARY'
  EXPORTING
    buffer                = lv_jsonx
  IMPORTING
    output_length         = lv_output_length
  TABLES
    binary_tab            = lt_binary_tab.

CALL FUNCTION 'SCMS_BINARY_TO_STRING'
  EXPORTING
    input_length          = lv_output_length
  IMPORTING
    text_buffer           = lv_json
    output_length         = lv_output_length
  TABLES
    binary_tab            = lt_binary_tab.

* JSON to ABAP
CALL TRANSFORMATION id SOURCE XML lv_jsonx RESULT text = lt_flight.
```

Version History

Note 2786259 - PL11

/UI2/CL_JSON
- Optimized. Performance lost, introduced in PL10 (note 2763854) when unescaping special characters (\r\n\t\"
- Fixed. Short dump, with <STRING_OFFSET_TOO_LARGE> when running GENERATE method with empty or invalid input

/UI2/CL_DATA_ACCESS
- Fixed. Short dump, when accessing elements of null array
Note 2763854 - PL10

- **Fixed:** Deserialization and generation of the ABAP data from JSON strings with Unicode characters fail
- **Fixed:** Unescaping of `\t` and `\n` char combinations in strings handled incorrectly
- **Fixed:** GENERATE method fails on JSON attribute names containing spaces

Note 2650040

- **New:** Support for deserialization of OData Edm.Guid
- **New:** Support of Enum data types in ABAP. From SAP_BASIS 7.51, below - enums are ignored.
- **New:** Support of conversion exits for serializing and deserializing.
- **Fixed:** SERIALIZE method delivers an invalid JSON string, when NUMC type, filled with spaces is used.

Note 2629179

- **New:** JSON timestamp fields, serialized in OData Edm.DateTime format (e.g. "/Date(1467981296000)/") are supported, and properly deserialized in ABAP date, time or timestamp fields
- **New:** JSON timestamp fields, serialized in OData Edm.Time format (e.g. "PT1H34M55S") are supported, and properly deserialized in ABAP date, time or timestamp fields
- **Fixed:** content is scrambled, when using GENERATE method for JSON objects with a name containing special characters (for example "__metadata")
- **Fixed:** GENERATE method does not consider custom name mapping pairs passed as a parameter for CONSTRUCTOR or GENERATE methods
- **Fixed:** generation of very long integers (serialized numeric date) fails, due to I type overflow (you get 0 instead of an expected number)

Note 2526405

- **Fixed:** Deserialization of the inconsistent data (JSON string into ABAP table) leads to a short dump if JSON string is empty.
- **Fixed:** Serialization of data with includes with defined suffix (RENAME WITH SUFFIX) dumps
- **Fixed:** GENERATE method fails, if the JSON object contains duplicate attributes and PRETTY_MODE-CAMEL_CASE is not used.
- **Fixed:** GENERATE method fails, if JSON object contains attribute names longer than 30 characters (allowed ABAP field length). Can also occur in case name is shorter than 30 characters, but PRETTY_MODE-CAMEL_CASE is used.
- **New:** methods DUMP_INT, DUMP_TYPE, RESTORE_TYPE, and RESTORE can be overridden now. So, you can introduce your own data type conversion on serialization and deserialization.
- **New:** now it is possible to pass name mapping table as a parameter for the constructor/serialize/deserialize and control the way JSON names formatted/mapped to ABAP names. This may help if you need special rules for name formatings (for special characters or two long JSON attributes) and standard pretty printing modes cannot help. With this feature, you may eliminate the need for the class extension and redefine of PRETTY_NAME and PRETTY_NAME_EX methods.
- **New:** PRETTY_NAME_EX method extended to support the encoding of more special characters (characters needed in JSON names but that can not be used as part of ABAP name). The supported characters are: "!#$%&*-~/:|@.". Used with pretty_mode-extended.
- **New:** /UI2/CL_DATA_ACCESS class for working with dynamic ABAP data object (generated with method /UI2/CL_JSON=>GENERATE). The class can be used as a replacement for multiple ASSIGN COMPONENT language constructions.

Note 2292558

- **Fixed:** Empty JSON objects, serialized as entries of the table, are not deserialized into corresponding ABAP structures and further parsing of the JSON string after an empty object is skipped.
- **Fixed:** JSON fields containing stringified timestamp representation in ISO 8601 format are not deserialized properly in the corresponding ABAP timestamp field.

Note 2300508

- **Fixed:** Recursive (hierarchical) JSON objects cannot be deserialized.

Note 2330592

- **Fixed:** Partial serialization/deserialization of the JSON is not supported
- **New:** Extending of the class is supported
- **New:** Added support for serializing named include structures from ABAP as embedded sub-objects in JSON

Note 2368774
• Fixed: /UI2/CL_JSON creates unnecessary wrapping JSON object around value for name/value (table with 2 fields) tables with 1 unique key
• Fixed: Performance of serialization/deserialization of big tables into/from JSON associative arrays (maps) is slow
• Fixed: When trying to deserialize invalid (not matching) structure from JSON to ABAP dump OBJECTS_MOVE_NOT_SUPPORTED occurs

Note 2382783

• Fixed: Unescape of symbol '\' on JSON deserialization does not work
• Fixed: Short dump on serialization of classes with protected/private attributes
• Fixed: Short dump when serializing dynamic, not existing, types

Note 2429758

• Fixed: Short Dump on deserialization of classes with read-only attributes
• New: Serialization parameter added NUMC_AS_STRING, controlling the way how NUMC fields are serialized. The default is FALSE. If set to TRUE, NUMC fields serialized not as numbers, but as strings, with all leading zeroes. Deserialization works compatibly with both ways of NUMC serialized data.
• New: GENERATE and GENERATE_INT methods are added for on the fly creation of ABAP data objects from JSON, without the need to have predefined ABAP structure. Supports automatic creation of ABAP structures, tables, and elementary types, with respect to JSON types. Supports structure/table nesting.
• New: DESERIALIZE_INT method throws an exception CX_SY_MOVE_CAST_ERROR and stops further processing in case of malformed data found and STRICT_MODE parameter in constructor set to TRUE.
• New: Added support of XSTRING as input for deserialization.

Note 2480119

• New: GENERATE method creates local custom class for deserialization (lc_json_custom), instead of standard /ui2/cl_json
• Fixed: Internal tables are not initialized when deserializing JSON with empty arrays
• New: Deserialization into a field with REF TO data type, if the field is bound, using a referenced data type
• New: Deserialization uses automatic generation of the data if the field has "REF TO DATA" type and bound data is initial