JSON-ND Format Specification V1.0

JSON-ND or ***JSON with Named Datatypes*** is a very simple way to include data-types in JSON data.

All that is needed to encode JSON data into JSON-ND is to **append the data-type to the member name**. So the example below:

```
{
  "name": "Alice",
  "isActive": 0,
  "amountPaid": 20
}
```

becomes:

```
{
  "name:string": "Alice",
  "isActive:boolean": 0,
  "amountPaid:currency": 20
}
```

Both forms of this JSON object are valid JSON in most, if not all, modern browser Javascript implementations.

Arrays can be defined with a type in the same way:

```
{
  "counts:integer[0,3]": [226,33,206]
}
```

For un-name and mixed type arrays, the data type can be encoded in the ***value***:

```
[
  "Bob:string",
  "To be\u003A or not to be:string",
  "22:currency"
]
```

The use of the colon character and the data-type in the member name *is* legal JSON. The original JSON post made the term ***name*** slightly ambigous by using it in the text of the post, but not defining it in the accompanying McKeeman Form or Workflow diagrams where it is simply referred to as "string". *The JSON standard* ECMA-ST-ECMA-404 clears this up by stating that the name of any JSON member is a *string*, with no restrictions: it may contain any unicode character.

This fact allows for a very simple and efficient means to fully qualify the data-type in JSON syntax. It also provides, as a side-effect, a way to define complex types and remote methods (aka remote procedure calls).

The term **"data-type"** is completely open and it should be interpreted to mean:

Text that describes how the associated value should be interpreted by the intended consumer.

JSON-ND defines only three data-types ***Enum***, ***MixedType[]*** and ***Interface***. All other data-types and syntax for definitions of data-type depend on the ***Language Style***.

For example, in C style langages, an integer is typically written as *int*, but in Pascal, an integer is written as *Integer*.

Any language style may be used in JSON-ND. The style being used can be conveyed within the message or in a protocol header.

What is the JSON-ND Specification?

The JSON-ND specification is currently a draft specification that describes the semantics required to efficiently ***qualify data types for JSON elements***, to ***define complex-data types*** and provides a means to ***define Remote Method Calls*** using JSON.

Its primary purpose is to **reduce or eleminate ambiguity** of JSON values by **including type information** in the JSON string (name) or value.

This is approach is most useful when using JSON syntax to communicate between strongly typed and untyped language implementations of a service; or with strongly typed languages using a dynamic data structure where the exact data type may not be known at compile time. This is also helpful for data visualisation tools when displaying never-before-seen data.

The data-type is **always** optional and may be applied to none, some, or all of the *member names* or *array elements* and follows a perferred language style (eg C++, C#, GO, Pascal, Typescript). The language style can be conveyed in a number of ways including pre-agreement, in the protocol header or in the data itself.

Using JSON-ND syntax

JSON syntax defines:

- how to qualify the intended data-type for any element;
- how define complex types; and
- the approach for defining methods that may be evoked on a remote server (typically Web Service methods)

All JSON is valid JSON-ND, and all JSON-ND* MUST BE *valid JSON.

JSON-ND defines five reserved words:

- **Interface** - the keyword to indicate that the named element is a data-type definition
- **MixedType[]** - a keyword indicating a JSON Array that elements of mixed data-type.
- **Enum** - to define enumerated types
- **required** - to indicate that for a valid message, a particular element is required.
- **property** - to indicate that an element property rather than a field

Qualifying Object Members

Other methods for including typing in JSON often use multiple JSON elements for each piece of data, eg:

```
{"name":{ "type": "string", "data": "Alice"}}
```

This is moderately inefficient, typically more than doubling the message length. **JSON-ND does not use this approach**.

An alternative method is to use a pre-processing section in the message header containing the data types:

```
{"types": {
   "string" : [
      "name"
  ],
   "integer" : [
     "count"
   ]
},
 "data": {
    "count": 1,
    "rows" : [
      {"name": "Alice"}
    ]
  }
}
```

which is generally more efficient than the first form when there are repeating elements. **JSON-ND does not use this approach either**.

JSON-ND Syntax represents the same element much more simply as:

```
{ "name:string" : "Alice" }
```

which is less verbose and, depending on the implementation, has lower memory overhead and parsing advantages over the previous two forms.

JSON-ND syntax **does not *require* elements to be qualified**, so the form:

```
{ "name":"Alice", "isActive": false, "amount:currency": 32 }
```

is also valid JSON-ND. Only the ambiguous *amount* member here needs qualification. The others are simply unmodified JSON.

Qualifying Array Elements

With JSON arrays, the data-type and array range of contained elements may be encoded in the form:

```
{ "<label>:<data-type>[<lower bound>,<length>]" : [ <elements>  ] }
```

For Example:

```
{ "transport:string[0,4]" : [ "car", "bus", "plane", "train" ] }
```

where the array is defined with:

- a data-type of *string*,

- an optional lower bound of *0*,
- and an optional length of *4*.

There are parsing advantages in this form over standard JSON as the length of array is known in advance and can be pre-allocated.

In order to handle JSON Arrays of mixed type, JSON-ND Syntax uses the ***MixedType[]*** reserved data-type, and encodes mixed type JSon Array values by appending the data type to ***JSON value*** as in the example:

```
{
  "stuff:MixedType[]" : [
    "Alice:string",
    true,
    "1:currency",
    "To be\u003A Or not to be:string"
  ]
}
```

Note that the colon in the text has been escaped to help with parsing.

The data-type qualifier is ***always*** optional, even for mixed type Json Arrays.

In order to prevent a situation where a parser cannot decide if an Array element value has a data type or not, it is **NOT** permitted to specify the data-type in the JSON Array name AND the JSON value ie:

`"ids":["1:integer"]` is valid

`"ids:integer[]":[1]` is valid

`"ids:integer[]":["1:integer"]` is **invalid**.

Encoding Text Strings Containing Colons.

To remove abiguity with strings and the presense or absence of colon in the text, either escape the text colons and include a data type OR simply include the data-type and the final colon will delimit the data-type:

When encoding the string "To be: Or not to be", ALL the following forms are valid:

`"To be\u003A Or not to be"`

`"text":"To be\u003A Or not to be"`

`"text:string":"To be: Or not to be"`

`["To be\u003A Or not to be"]`

`["To be\u003A Or not to be:string"]`

`["To be: Or not to be:string"]`

`"text":["To be\u003A Or not to be"]`

`"text:string[]":["To be: Or not to be"]`

```
"text:MixedType[]":["To be\u003A Or not to be"]
```

```
"text:MixedType[]":["To be\u003A Or not to be:string"]
```

```
"text:MixedType[]":["To be: Or not to be:string"]
```

But `"To be: Or not to be"` and `"ids:MixedType[]":["To be: Or not to be"]` represent a " *Or not to be*" data-type with a value of "To be".

Also, `"text:string":"To be\u003A Or not to be:string"` and `"values:string[]": ["To be\u003A Or not to be:string"]` represent a *string* of value "To be: Or not to be:string"

Rules for URLs are identical to strings, but represent a specific use case. When encoding the URL, "http://myserver.com/api/user" the following forms are valid (assumes *url* is a defined type):

```
"http://myserver.com/api/user:url"
```

```
"http\u003A//myserver.com/api/user:string"
```

```
"endpoint":"http://myserver.com/api/user"
```

```
"endpoint:url":"http://myserver.com/api/user"
```

```
"endpoints:string[]":["http://myserver.com/api/user"]
```

```
"values:MixedType[]":["http://myserver.com/api/user:url"]
"values:MixedType[]":["http\u003A//myserver.com/api/user:url"]
```

But:

`"http://myserver.com/api/user"`; and

`"values":["http://myserver.com/api/user"]`; and

`"values:MixedType[]":["http://myserver.com/api/user"]`

represents a data-type of "//myserver.com/api/user" with a value of "http".

Defining Complex Data-Types

With the simple convention of including the data-type in the name of a JSON member or an element, there is sufficient syntax to define **custom data types** and define **remote methods**.

This approach is far simpler, (but also less complete) than the JSON Schema approach because

- the data and definition are not (necessarily) separated
- and no prior knowledge is required by a service to use the definition.

Defining Enumerable (Ordinal) Types

JSON-ND defines Enumerable (or *Ordinal*) types as mixed type array with the reserved data-type ***Enum***. The initial index of the enumerated type can be qualified as a constant data-type. For example:

```
{
  "RoleType:Enum" :[
    "admin:1",
    "accounts",
    "sales",
    "service"
  ]
}
```

There does not seem to be a less ambigous way to convey an enumerated type in valid JSON and be certain of not confusing it with an object type: so this is the recommended approach.

In the case where a specific language has alternative convention for defining an enumerated type that can be conveyed using legal JSON syntax, then this is not precluded.

Defining Custom Data-types

JSON-ND syntax defines data-types using the reserved keyword ***Interface***.

Complex custom types can be created by combining primitive and other custom defined types. The actual data-type name and syntax will vary depending on language style.

Using the "RoleType" defined above:

```
{
  "User:Interface": [
        "id:int",
        "name:string",
        "roles:RoleType[0,]"
    ]
}
```

This new User type can be then be used in a more complex custom type:

```
{
  "Order:Interface [
    "id:int",
    "orderDate:Date",
    "user:User",
    "customer:Customer"
  ]
}
```

where primitive types (*id*, *orderDate*), the previously defined *User* object and an other (presumuably defined) data-type *Customer* are used in the *Order* data-type.

There is no restriction of the use of complex types so nested objects are permitted.

Optional and Required Data

In JSON-ND syntax, for simple or complex types, the data-type qualifier is **always optional**. This does not speak to whether an element is required for a particular service or method.

The reserved keyword ***required*** can be prepended to any data type using a space as delimiter, in order to inform the consumer that element should never be omitted or null.

For example it may be that a service needs the *id* element to work correctly. So the definition for the service or type can indicate that a request will be rejected if the element is missing:

```
{
  "User:Interface": [
        "id:required int",
        "name:string",
        "roles:RoleTypes[0,]"
    ]
}
```

Many languages styles support the option of non-nullable data-type: this may or may not be explict. So how this is implemented in JSON-ND will often dependend on the language style.

For example the *int* type in C# is explicitly not nullable, so in this case where the language style is specified, the *required* keyword is redundant and can be omitted.

Defining Methods and Remote Methods (Remote Procedure Calls)

Because the JSON String type has no character format restrcitions, JSON-ND allows for a method call to be written exactly as it is in the specific language.

When defining a **method** JSON-ND allows two forms: as part of a ***Custom Type*** form and a ***method-delegate*** form. The delegate form MAY be expressed *in-line* in an object instance.

Custom Type Form

Using Pascal Style, a simple method for adding two integers could be defined as part of a custom data-type of type *Interface*:

```
 {
   "mathService:Interface :[
     "function AddTwoIntegers(int1:integer; int2:integer)   :integer"
   ]
 }
```

Pascal, Kotlin and Typescript styles (to name a few) have the added advantage that the syntax complies with the convention that the data-type (return type in this case) is appended after a last colon in the JSON String.

For C style languages, the method would be defined as:

```
 {
   "mathService:Interface [
     "int AddTwoIntegers(int int1, int int2)"
   ]
 }
```

Unfortunately, this style does not append the data type so, implementers of JSON-ND in these languages will have to accept a minor exception to the rule.

In-line Delegate Form

The *in-line* form is intended to convey both the method syntax and additional information about the method, for example, the endpoint URI where the method can be evoked. The Keyword Interface is not used.

This form requires special language specific handling and is only optionally implemented.

Using Pascal languages, the delegate (or method pointer) definition for the AddTwoIntegers function might be:

```
Type
  AddTwoIntegers = function(int1:integer; int2:integer): integer;
```

So the *in-line* delegate form of the AddTwoIntegers function in Pascal style would be:

```
{
  "AddTwoIntegers:function(int1:integer; int2:integer):integer"
    : "https://myserver.com/api/mathService"
}
```

Using C, as all methods may be referenced by a pointer, the language definition is simply its ordinary definition:

```
int AddTwoIntegers(int int1, int int2);
```

So, the C form would be:

```
{
  "AddTwoIntegers:int AddTwoIntegers(int int1, int int2)" : "https://myserver.com/api/mathS
}
```

This form may also be used in custom data-types.

```
  "mathService:Interface": [
    "function(int1:integer; int2:integer):integer"
  ]
```

This method is especially useful when defining custom-data types to be implemented as classes.

The implementation of in-line delegate definitions needs different approaches depending on the language. Because JSON-ND states that the data-type is *the text following the last colon* the Pascal parser would need to identify the return type of Integer and then use keywords to validate the definition as part considered to be the *value*. The C parser would intepret the data-type as the method and then simply assign the name to the static method. A C++ parser may also encounter colons in the case of a data-type namespace (eg std::) and would need a careful approach to correctly interpret definition.

Method-Delegate Form

The same *AddTwoIntegers* method may also be defined using the method-delgate form. This form is most universal than the in-line method requiring less specific handling. If the consumer application chooses to support method definitions, this approach MUST be implemented (the *in-line* form is optionally implemented)

The method is defined as a type of *Interface* and the delegate (or method pointer) form of the function is defined as the *value*.

Once defined, the method data-type, it can be included in a custom data-type:

```
 {
   "AddTwoIntegers:Interface": "function(int1:integer; int2:integer):integer",
   "mathService:Interface": [
     "addTwoIntegers:AddTwoIntegers"
   ]
 }
```

An instance of the mathService can then contain the endpoint for an implementation:

```
{
  mathService : {
    "addTwoIntegers": "./api/addtwointegers"
  }
}
```

This approach also supports multiple alternate implementations of the same type on different services:

```
 {
   "AddTwoIntegers:Interface": "int AddTwoIntegers(int int1, int int2)",

   "mathService:Interface": [
     ...,
     "addTwoIntegers:AddTwoIntegers",
     "addTwoIntegers_REST:AddTwoIntegers",
     "addTwoIntegers_AU:AddTwoIntegers",
     "addTwoIntegers_NZ:AddTwoIntegers",
   ]
 }
```

with the instance:

```
{
  mathService : {
    "addTwoIntegers": "./api/addtwointegers",
    "addTwoIntegers_REST": "./api/addtwointegers/:int1/:int2",
     "addTwoIntegers_AU: "https://apis.company.com.au/api/addtwointegers",
     "addTwoIntegers_NZ": "https://apis.company.co.nz/api/addtwointegers"
  }
}
```

Defining Class Interfaces

It is possible to define ***Interfaces*** that can be implemented as *Class* depending on the consumer language.

Many langauges support Classes including Javascript as of ES6.

Take the Typescript definition below:

```
class Vendor {
  name: string;
  constructor(name: string) {
    this.name = name;
  }

  greet() {
    return "Hello, welcome to " + this.name;
  }
}
```

This can easily be represented as an interface in *in-line delegate* form as JSON-ND as:

```
{
  "Vendor:Interface" : [
    "constructor(name: string):Vendor",
    "greet():void"
  ]
}
```

Otherwise, in method-delegate form:

```
{
  "Greet:Interface" : "greet():void",
  "Constructor:Interface" : "constructor(name: string):Vendor",
  "Vendor:Interface" : [
    "constructor:Constructor",
    "greet:Greet"
  ]
}
```

Defining interface properties

When defining a property the ***property*** keyword may be used. The definition of the *getters* and *setters* for the property will depend upon the language.

For example, the C# interface below:

```
public interface ISampleInterface
{
    string Name
    {
        get;
        set;
    }
}
```

can be described in JSON-ND as

```
{
   "ISampleInterface:Interface" : [
     "name:property string"
   ]
}
```

The default behaviour is to assume that both *getter* and *setter* are present. To indicate whether a property is read or write only will be dependent on the language style;

The JSON-ND specification does not specify the exact syntax for defining getters and setters (these may be defined in future standards for language specific implementations), but suggests the following approach.

For C#, the following method-delegate form may be sufficient:

```
{
   "Name:Interface":"string name{get;set;}",

   "ISampleInterface:Interface" : [
     "name:property Name"
   ]
}
```

For Pascal, where and interface definition must contain the method name for the getter and/or setter:

```
 Type
   ISampleInterface = Interface
     function GetName: string;
     procedure SetName(const Value: string);
     property name: string read GetName write SetName;
   end;
```

the following form is suggested:

```
{
   "getName:Interface":"function GetName: string",
   "setName:Interface":"procedure SetName(const Value: string)",
   "Name:Interface":"string read getName write setName}",

   "ISampleInterface:Interface" : [
     "name:property Name"
   ]
}
```

Other languages could follow the same approach.

In JSON-ND, *Class* definitions are technical possibility: it would require the implementation of methods to be included in the definition and this is an extreme security risk.

However, implementations SHOULD NOT be included in definitions because it is extremely difficult to prevent arbitrary code from being executed in the consumer application. Scripting languages are particularly vulnerable (eg dynamic SQL and the *exec* command in javascript). The inclusion of implementations for compiled languages might encourage developers to include dynamic execution methods in their applications making them vulnerable.

Conveying Language Style and error handling

Data consumers need to be certain of the message syntax (either JSON-ND or JSON) to ensure the data is correctly interpretted.

The specification defines 3 qualifiers for indicating the message type.

1. "version" indicating the version of JSON-ND specifciation ("1.0")

2. "style" conveys the context of data-types and method syntax (eg "xs", "c++", "pascal", "c#")
3. "strict" indicator for error handling purposes.

These indicators can be conveyed to the data consumer in any way. For example:

- out of bounds : A pre-agreement where JSON-ND version, style and strictness have been coded into the application by design
- by using a protocol specific metadata exchange (eg HTTP Content-type)
- by including a JSON-ND object within the message.

The official MIME-Type `application/json-nd` with the parameters "version", "style" and "strict". This mime-type registered with IANA.

Scoping JSON-ND using the JSON Object

The Scope of a language style may be restricted using then JSON-ND object.

The JSON-ND object is defined as:

```
 "Json-ND" :{
  "version": 1.0,
  "style": "<language>",
  [,"strict": true|false]
  [, "data":{}]
}
```

This allows a single message to include multiple versions of the defintion or data to be included in a single message.

```
{
  "Json-ND" :{
    "version": 1.0,
    "style": "pascal",
    "data":{
      "Vendor:Interface" : [
        "constructor Create(name: string)",
        "procedure greet"
      ]
    }
  },
  "Json-ND" :{
    "version": 1.0,
    "style": "C#",
    "strict": true,
    data:{
      "Vendor:Interface" : [
        "Vendor(string name)",
        "void greet()"
      ]
    }
  }
}
```

The only allowed value of ***version*** is currently "1.0".

The ***style*** property is a string value that indicates a programming language or specification where data-types and/or method syntax are defined. Any string may be used here to sufficiently disambiguate all data-types and method syntax used within the message.

The ***strict*** property indicates that non-conformant messages should be rejected. Where the strict propery is not used (or set to false) then no error message should be generated by the parsing process, however it may be that a method may still fail during processing.

Error handling (*strict* indicator)

The introduction of typing in JSON-ND introduces the need for error handling because there is a potential for data to be incompatible with the definition.

The default action when parsing JSON-ND syntax for unknown properties, non-conformant values, or misalignment of range bounds in Arrays is to **ignore data and adhere the specified type**. For example: the following non-conformant message:

```
{"name:string": true, "items:integer[0,2]" : "[3,2.5,7]" }
```

should be interpretted as:

```
{"name:string": null, "items:integer[0,2]" : "[3,<default||null>]" }
```

and the consumer should attempt to use the interpretted data as supplied.

When the JSON-ND "strict" qualifier is used (either unqualified or set to "true") then the entire message transaction should be rejected as an error.

For Example using a Scoped JSON-ND Object:

```
 "Json-ND" :{
  "version": 1.0,
  "style": "pascal",
  "strict": true,
  "data":{
      "id:required integer": null,
      "age:integer": "old",
   }
}
```

should result in an exception being raised and error handling to be evoked. This applies equally an entire message using the JSON-ND mime-type, for example in the http request:

```
GET /api/user HTTP/1.1
Content-Type: application/json-nd; version=1.0; style="pascal"; strict

{
  "id:required integer": null,
  "age:integer": "old"
}
```

The receiving server should reject the message as a `400 Bad Request` as the required Identifier "id" is not supplied and the integer element "age" is not an integer.

Guidelines for using JSON-ND Syntax

1. All JSON is valid JSON-ND, and all JSON-ND **MUST BE** valid JSON.
2. To translate JSON to JSON-ND, for JSON Object, JSON Object Elements and named JSon Arrays, (optionally) **append the data-type to the end of the element name or value prefixed by a colon**.
3. For named arrays, **the data-type and array range may be appended to the name prefixed by a colon.** The array range is optional.
4. For JSON name/value pairs, the Value MUST NEVER include a data-type.
5. The term **"data-type"** is completely open and it should be interpreted to mean: `Text that describes how the associated value should be interpreted by the intended consumer`.
6. The data-type is **always** optional and may be applied to none, some, or all of the elements or array element values.
7. For *all* Json Arrays, the data-type may be encoded in the JSON value by **converting the value to a string, and then appending the data type prefixed by a colon**. It is recommended that this approach be used *ONLY* when:

    1. the array contains data elements of mixed type; or
    2. when the array is un-named.
8. Encoding the data-type in the value is allowed in named arrays, but NOT allowed if the data-type is applied to array name (except for MixedType[]) ie `{"a:string[]":["one:string"]}` is **invalid** and will likely be interpretted incorrectly by the parser.
9. In the case where the JSON Array element is a string containing one or more colons, the standard JSON Unicode escape *\u003A* should be applied, otherwise the data-type is **assumed to be the text following the final colon.**
10. For specific languages, the COLON forms part of the method definition. An exception to the rule above may be required when defining methods in-line.
11. In order to qualify how to correctly interpret the data-type, an optional ***language style*** can be specified in the form of a JSON-ND object:
12. JSON-ND can be passed without the style element where entities within a JSON transaction assume a pre-arranged language style or the data-type has been included in a transaction via meta-data (eg HTTP Content-type).
13. In cases where there is no pre-arranged style,

    1. the style qualifier may be added as an additional element.
    2. A fully qualified JSON-ND Object may also be used to restrict scope or if no metadata mechanism is available: eg

```
{
   "Json-ND": {"version": 1.0,"style": "pascal"}
   "id:UInt32": 345,
   "name:string": "Bob",
   "cost:currency": 1400
}
```

```
{
   "Json-ND": {
   "version": 1.0,
   "style": "pascal",
   "data": {
      "id:UInt32": 345,
      "name:string": "Bob",
      "cost:currency": 1400
    }
  }
}
```

12. In the case where the intended consumers of a message are not known (eg. public APIs), multiple comsumer language versions of the definition may be generated for each of the likely implementations of the definition (eg C++, C#, Pascal, Go, Haskell). Additionally a "default" language such as XSD Built-in Types is recommended for public apis

```
"Json-ND" :{
  "version": 1.0,
  "style": "http://www.w3.org/2001/XMLSchema-datatypes"
}
```

13.

    Remote method calls can be defined using the syntax of the Language Style. There are two forms

    1. In-line; or
    2. using Method-Delegate approach.

14.

    A complex data-type can be used to define an Class Interface in JSON-ND.

15.

    Technically a Class defintion could be encoded in JSON-ND, however the inclusion of **method implementations** in JSON-ND messages ***is strongly discouraged***.

Example Javascript JSON-ND Validation

Below is a very simple JSON-ND parser/validator. raw source

```
<html>
<head>
 <title>JSON-ND Example Parse - Javascript</title>
<script>
  const JSONND =
  {
      Parse : function(ndObj, strict)
□ {
        var failed = false;
var failedMessage= "";
var obj = JSON.parse(ndObj);
```

```
for (var propertyName in obj) {
var p = propertyName.indexOf(":");
var v = obj[propertyName];
if (p > 0) {
//specific check for <Big>int (not number) ECMAScript 2020
if (propertyName.substr(p+1,3)==='int')
{
var nv=parseInt(v); // it will truncate floats,
if (strict)
{
  if (!nv || (parseFloat(v)-nv!=0) )
  {
    failedMessage += '"'+propertyName+'" value <'+v+'> not a valid Integer;';
    failed=true;
v=null;
  }
} else {
  v=nv;
}
}; // add more validation.
obj[propertyName.substr(0, p)] = v;
delete(obj[propertyName]);
}
};
if (failed) throw failedMessage;
return obj;
  }
  };
</script>
</head>
<body>
  <h1>Example demonstration of JSON-ND parser.</h1>
  <script>
     var nd = '{"abc:string":"abc contains a string", "def:integer" : "not a number"}';
     document.write("<h3>Original 'nd' object:<pre>",nd,"<pre></h3>");
     document.write("<p>See that there is no 'nd.abc' property: abc=<b>" ,nd.abc,"</b></p>");
 document.write("<p>Attempting accessing 'nd.abc:string' property is not valid Javascript</p>
     document.write("<h3>Then validate with JSON_ND_Parse:<pre>",JSON.stringify(JSONND.Parse(
     document.write("<p style='color: red'>Note that 'def' failed validation, but not <i>stri

 document.write("<h3>With <b>strict</b> operator in place:");
 try
 {
   var strictNd = '{"abc:string":"abc contains a string", "def:integer" : "20.5"}';
       document.write("<h3>'StrictNd' object:<pre>",strictNd,"<pre></h3>");
   document.write("<pre>",JSON.stringify(JSONND.Parse(strictNd,true)),"</pre>");
 } catch (e)
 {
   document.write(" Caught exception: <pre>",e,"</pre>");
 }
  </script>
</body>
</html>
```

demo