JSON

Federico Ulliana
UM, LIRMM, INRIA GraphIK

Slides collected from
D. Crockford, J. Robie, and R. Costello

Readings

- JSONiq : The History of a Query Language
  Florescu, Fourny (2013)

- JSONiq, the SQL of NoSQL
  Fourny (available at http://www.28.io/jsoniq-the-sql-of-nosql/)

XML and JSON

- XML was born out of SGML as markup language and became the way of exchanging any data in the Web

- JSON was born out of Javascript to represent simple objects and exchanging them between client-server.
  - Later, it became also the "reference" data-model for NoSQL systems

XML and JSON

« XML provides a way to label information from diverse data sources including structured and semi-structured documents, relational databases, and object repositories.

The Extensible Markup Language, XML, is having a profoundly unifying effect on diverse forms of information. For the first time, XML provides an information interchange format that is editable, easily parsed, and capable of representing nearly any kind of structured or semi-structured information »

Don Chamberlin, Jonathan Robie, Daniela Florescu, 2000

XML and JSON

« Unfortunately, XML is not well suited to data-interchange, much as a wrench is not well-suited to driving nails. It carries a lot of baggage, and it doesn't match the data model of most programming languages. When most programmers saw XML for the first time, they were shocked at how ugly and inefficient it was. It turns out that that first reaction was the correct one.

There is another text notation that has all of the advantages of XML, but is much better suited to data-interchange. That notation is JavaScript Object Notation (JSON).

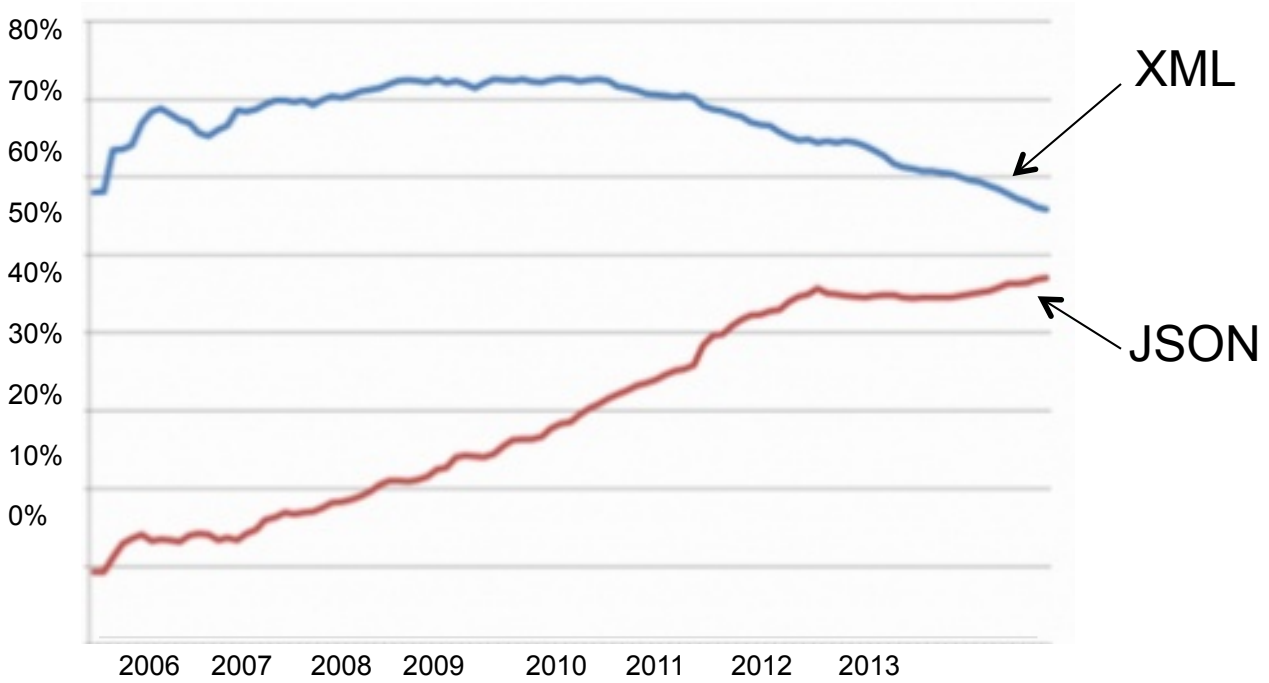
JSON is a better data exchange format.

XML is a better document exchange format.

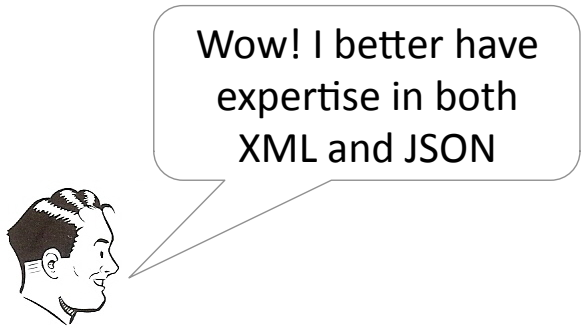
Use the right tool for the right job. »

Douglas Crockford, 2002

Trends for XML and JSON usage



Based on directory of 11,000 web APIs listed at Programmable Web, December 2013



JSON

- Named the "fat free alternative to XML”

- Its popularity is partly inherited from the widespread of XML in web applications
  - I need to use XML → JSON is simpler than XML
    → I can/need to use JSON

XML and JSON side by side

XML

```
<books>
    <book>
    <title>My First Book</title>

    <Author>Fake Author</Author>
   </book>

    <book>
    <title>Test book</title>

    <Author>Real Author</Author>
   </book>
</books>
```

JSON

```
{
   “books” : [
          {
          ”title”:“My First Book”,
          “Author” :“Fake Author”
          }
     ,
      {
          “title”:“Test book” ,
          “Author” :“Real Author”
          }
   ]
}
```

JSON

- Language Independent.
- Text-based.
- Light-weight.
- Easy to parse.

Versionless

- JSON has no version number.
- No revisions to the JSON grammar are anticipated.
- JSON is very stable.

JSON Is Not...

- JSON is not a document format.
  - document = something that contains text & structure and which is destined for publication
- JSON is not a markup language.
- JSON is not an universal serialization format.
  - No cyclical/recurring structures.
  - No invisible structures.
  - No functions.
  - YAML is an universal serialization format

Why JSON is better that XML

- XML syntax is heavier than JSON
  - XML uses tags to describe user data and tags increase the size of data

- To navigate XML, we have to use XPath which is an overhead removed in JSON because JSON is native to JavaScript

JSONIQ

JSONiq : XQuery for JSON

- XQuery is well suited for hierarchical semi-structured data. Many implementations exist, and can easily be adapted to add JSON support.

- Goals
  - powerful query language for JSON, without the complexity of XML.
  - data integration query language that can query and create JSON, XML, or HTML.

- The fatter XQuery that can also do JSON !

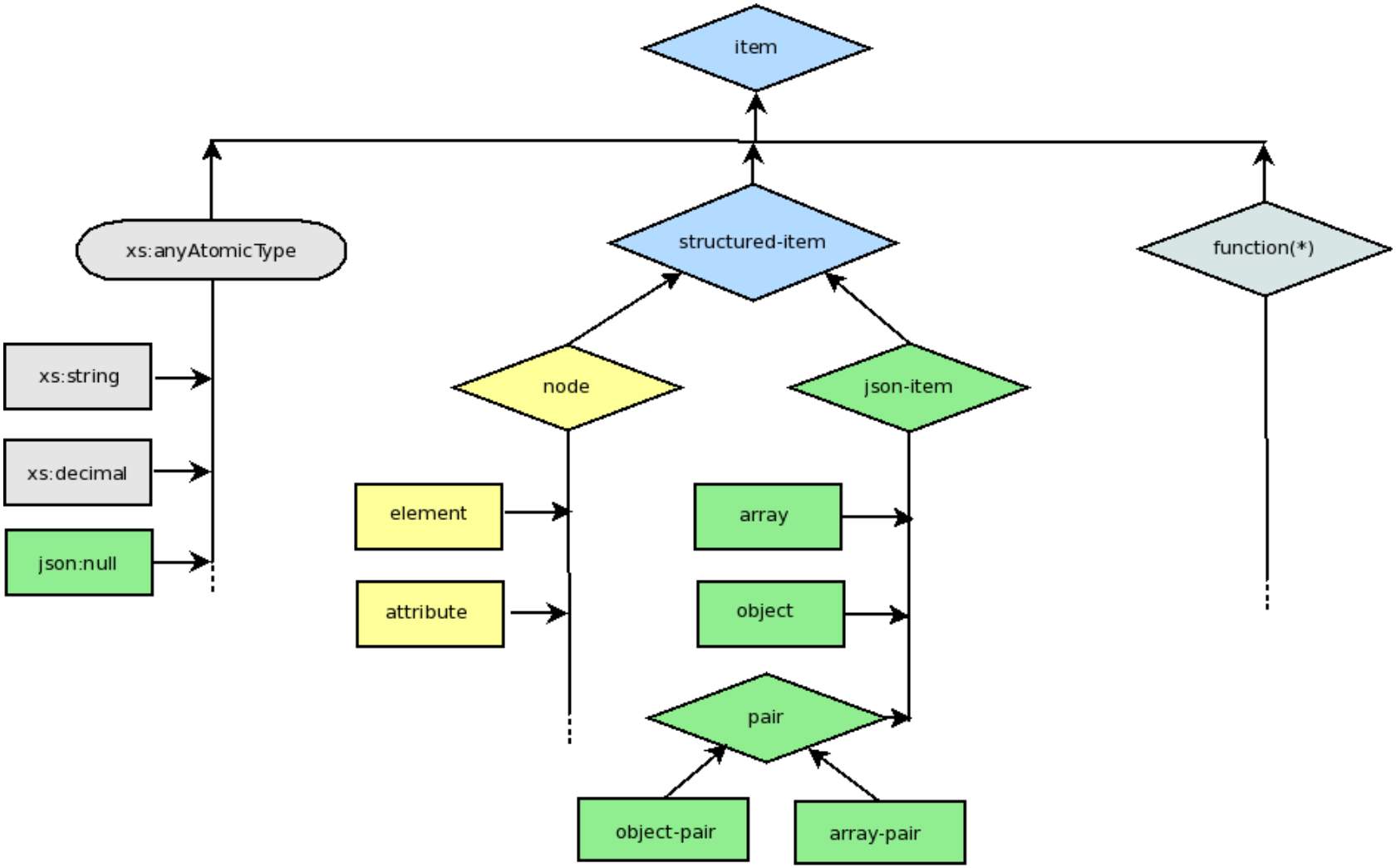
Composability

```
let $sarah := collection("users")[.("name") = "Sarah"]
return {
  "name" : "Amanda",
  "age" : $sarah("age") + 1,
  "gender" : "female",
  "friends" : $sarah("friends")
}
```

Every subexpression
is an expression

```
{
  "name" : "Sarah",
  "age" : 13,
  "gender" : "female",
  "friends" : [ "Jim", "Mary", "Jennifer"]
}
```

Adding JSON to XQuery Data Model



Aggregates

```
{
  for $sales in collection("sales")
  let $pname := $sales("product")
  group by $pname
  return $pname :
    sum(for $s in $sales return $s("quantity"))
}
```

```
{
  "blender" : 250,
  "broiler" : 20,
  "shirt" : 10,
  "socks" : 510,
  "toaster" : 200
}
```

Aggregates

```
{ "product" : "broiler", "store number" : 1, "quantity" : 20  },
{ "product" : "toaster", "store number" : 2, "quantity" : 100 },
{ "product" : "toaster", "store number" : 2, "quantity" : 50 },
{ "product" : "toaster", "store number" : 3, "quantity" : 50 },
{ "product" : "blender", "store number" : 3, "quantity" : 100 },
{ "product" : "blender", "store number" : 3, "quantity" : 150 },
{ "product" : "socks", "store number" : 1, "quantity" : 500 },
{ "product" : "socks", "store number" : 2, "quantity" : 10 },
{ "product" : "shirt", "store number" : 3, "quantity" : 10 }
```

JSONiq

- http://www.jsoniq.org/

- Zorba
  - implementation http://www.zorba.io/home
  - sandbox http://try.zorba.io/queries/xquery (needs link to public available JSON data)

JSON IN ORACLE DATABASES

JSON in oracle databases

- Why ? We have NoSQL systems
  - MongoDB, CouchDB, etc..

- But these may not give you a rigorous consistency model (updates&transactions)
  - that's what NoSQL systems tend to give up for the sake of performances

Oracle Database JSON Support

Store and manage JSON documents in database

- JSON documents can be stored as text, indexed and queried

- No new JSON datatype (as for XMLType)
  - JSON stored in CLOB, VARCHAR2, RAW, BLOB

- Querying capabilities
  - Extends .dotted notation to navigate structure with wildcards and array ranges

Creating a table

```
CREATE TABLE po                                          (purchase order)
   (
    id              NUMBER NOT NULL,
    document        CLOB

    CONSTRAINT ensure_json CHECK (document IS JSON)

);
```

Oracle recommends to add an `IS_JSON` check to validate data

Inserting a document

```
INSERT INTO po
  VALUES (1,
          '{"PONumber"               : 1600,
            "Reference"              : "ABULL-20140421",
            "Requestor"              : "Alexis Bull",
            "User"                   : "ABULL",
            "CostCenter"             : "A50",
            "ShippingInstructions"   : {...},
            "Special Instructions"   : null,
            "AllowPartialShipment"   : true,
            "LineItems"              : [...]}');
```

Resulting table

1	JSON record 1
2	JSON record 2
…	

Querying

```
SELECT document.PONumber
FROM   po
```

(notice case sensitive part)

Returns the list of values corresponding to `PONumber`

Querying

```
SELECT document.lineitems[0]
 FROM   po
```

Returns the first line-item of each record

Querying

```
SELECT document.ShippingInstruction.*
FROM   po
```

Returns all objects composing the shipping instruction of each record

Querying

```
SELECT count(*)
FROM   po
WHERE
  JSON_EXISTS(document,
  $.ShippingInstruction.address.state)
```

Counts the number of records with a state address shipping instruction

Indexing

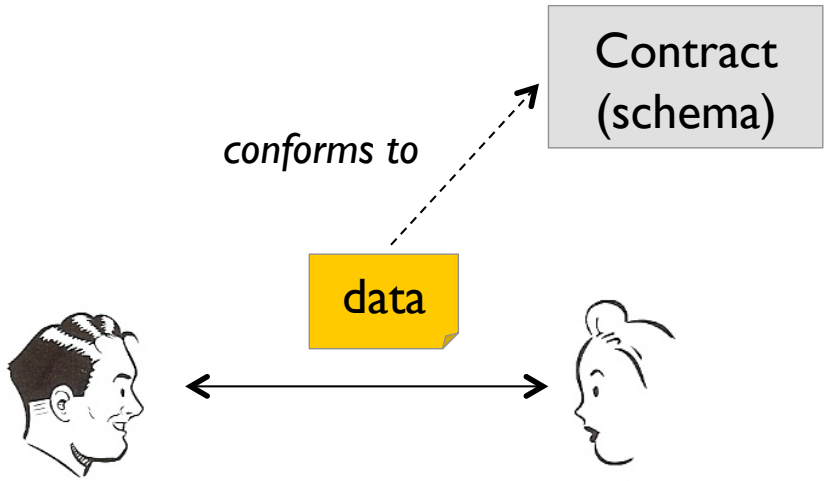
```
CREATE INDEX cost_center
OP
  (document,$.OPNumber)
```

Indexes all purchase order numbers to allow for a quicker search (eg. `OPNumber=42`)

JSON SCHEMA

A contract for data exchanges

Both XML Schema and JSON Schema may be used as a contract for data exchanges:



Fundamental difference between XML Schema and JSON Schema

- XML Schema: specifies *closed content* unless deliberate measures are taken to make it open (e.g., sprinkle the <any> element liberally throughout the schema).
- JSON Schema: specifies *open content* unless deliberate measures are taken to make it closed (e.g., sprinkle "additionalProperties": false liberally throughout the schema).

Definition of Open Content: instance documents can contain items above and beyond those specified by the schema.

Definition of Closed Content: instance documents can contain only those items specified by the schema.

The power of JSON Schema comes from:

- Recursion
- Simplicity: No regular expression types

A JSON Schema is a JSON object

```
{
  "keyword": value,
  "keyword": value,
  "keyword": value,
   …
}
```

A JSON Schema is a JSON object

```
{
  "keyword": value,
  "keyword": value,
  "keyword": value,
   …
}
```

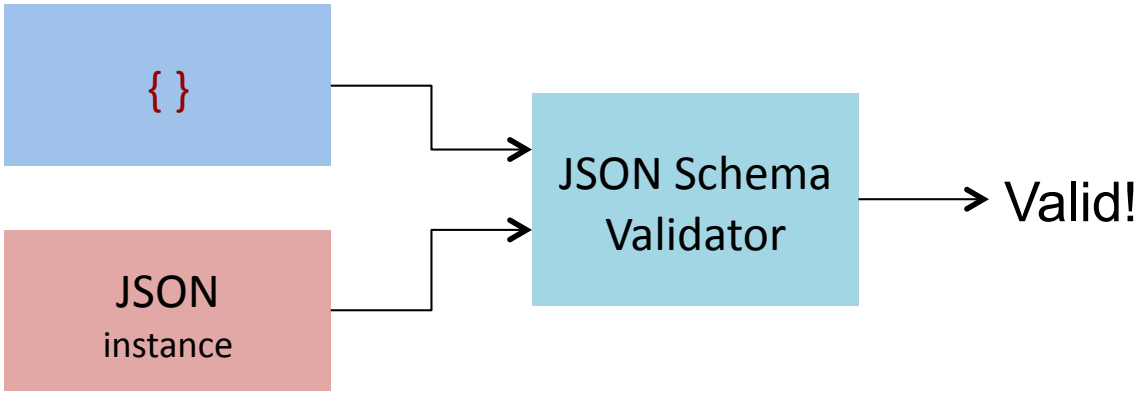
The JSON Schema specification defines the set of keywords and values that can be used to construct a schema.

Empty schema

This is a valid JSON Schema!

It places no restrictions on JSON instances.

Every JSON instance is valid!



JSON Schema Example

```
{
  "$schema": "http://json-schema.org/draft-04/schema
  "type": "object",
  "properties": {
    "Book": {
      "type": "object",
      "properties": {
        "Title": {"type": "string"},
        "Authors": {"type": "array", "minItems": 1, "maxItems": 5, "items": { "type": "string" }},
        "Date": {"type": "string", "pattern": "^[0-9]{4}$"},
        "Publisher": {"type": "string", "enum": ["Springer", "MIT Press", "Harvard Press"]}
      },
      "required": ["Title", "Authors", "Date"],
      "additionalProperties": false
    }
  },
  "required": ["Book"],
  "additionalProperties": false
}
```

Missing full power of Regular Expressions

```
<ELEMENT book  (author , title)>
```

- cannot be expressed in JSON Schema as there is no order among keys

JSON IN WEB APPLICATIONS

JSON can be the **X** in Ajax

JSON in Ajax

- JSON data is built into the page.

```
<html>...
  <script>
   var data = { ... JSONdata ... };
  </script>...
</html>
```

JSON in Ajax

- XMLHttpRequest
  - Obtain **`responseText`**
  - Parse the **`responseText`**

```
responseData = eval(
    '(' + responseText + ')');
```

JSON in Ajax

- Is it safe to use **`eval`** with XMLHttpRequest?
- The JSON data comes from the same server that vended the page. **`eval`** of the data is no less secure than the original html.
- If in doubt, use ***`string`*`.parseJSON`** instead of **`eval`**. Which is available in http://www.json.org/json.js
  - but eval() is faster than parseJSON()

JSONRequest

- A new facility to replace XMLHttpRequest when exchanging JSON

- Current campaign to make a standard feature of all browsers.

ECMAScript Fourth Ed.

- New Methods:
  - `Object.prototype.toJSONString`
  - `Array.prototype.toJSONString`
  - `Date.prototype.toJSONString`
  - `Boolean.prototype.toJSONString`
  - `Number.prototype.toJSONString`
  - `String.prototype.toJSONString`
  - `String.prototype.parseJSON`
- Available now: `JSON.org/json.js`

supplant

```
var template = '<table border="{border}">' +
    '<tr><th>Last</th><td>{last}</td></tr>' +
    '<tr><th>First</th><td>{first}</td></tr>' +
    '</table>';

var data = {
    "first":  "Carl",
    "last":   "Hollywood",
    "border": 2
};

mydiv.innerHTML = template.supplant(data);
```

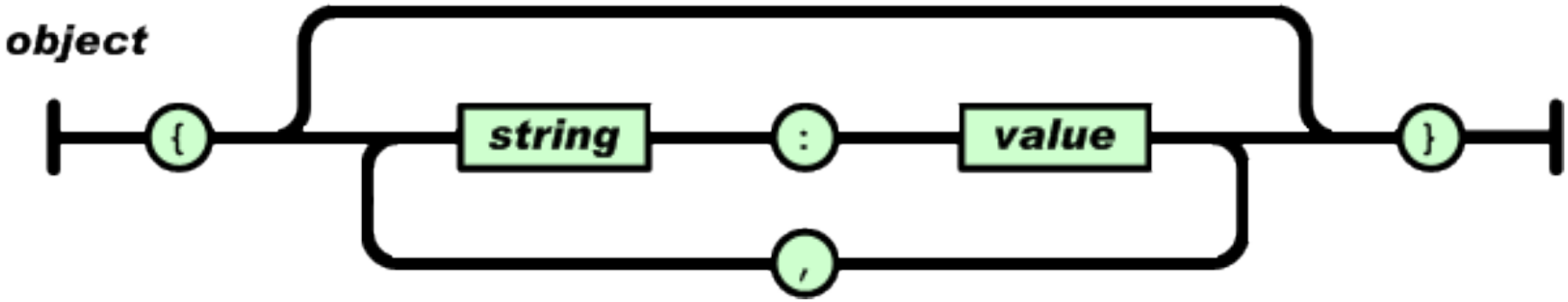
JSON DATA MODEL

JSON Structures

- JSON is built on two structures:
  - A collection of name/value pairs.
    - In various languages, this is realized as an *object*, record, dictionary, hash table, keyed list, or associative array.
  - An ordered list of values.
    - In most languages, this is realized as an *array*, vector, list, or sequence.

Syntax of JSON

- **Object**
  - An *object* is an unordered set of name/value pairs.
  - An object begins with { (left brace)
    and ends with } (right brace)
  - Each name is followed by : (colon)
    and the name/value pairs are separated by , (comma)



Syntax of JSON

- Objects can be **nested** (=they can be values)

$$\texttt{Object [Object}_1\texttt{ […] , Object}_2\texttt{ […]]}$$

Object

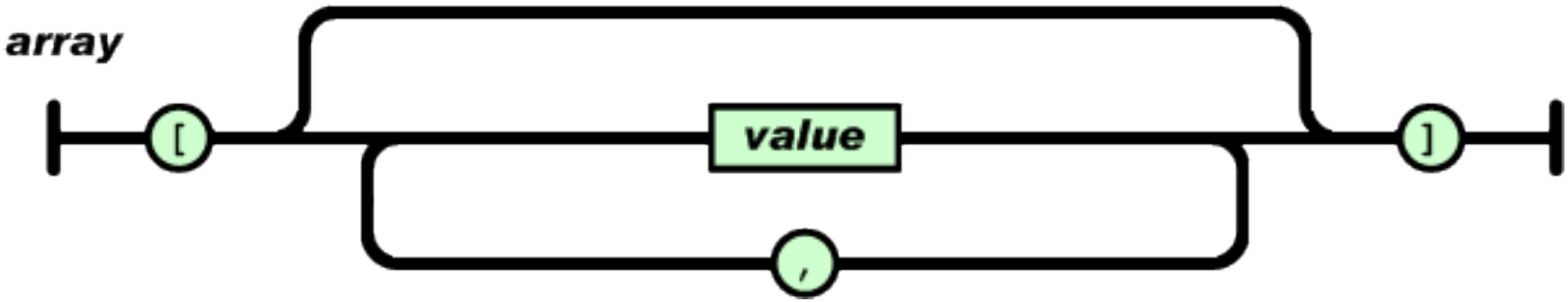
```
{"name":"Jack B. Nimble","at large":
true,"grade":"A","level":3, "format":
{"type":"rect","width":1920, "height":
1080,"interlace":false, "framerate":24}}
```

Object

```
{
    "name":      "Jack B. Nimble",
    "at large": true,
    "grade":     "A",
    "format": {
        "type":       "rect",
        "width":      1920,
        "height":     1080,
        "interlace": false,
        "framerate": 24
    }
}
```

Syntax of JSON

- **Array**
  - An *array* is an ordered collection of values.
  - An array begins with [(left bracket)
    and ends with] (right bracket).
    Values are separated by , (comma).

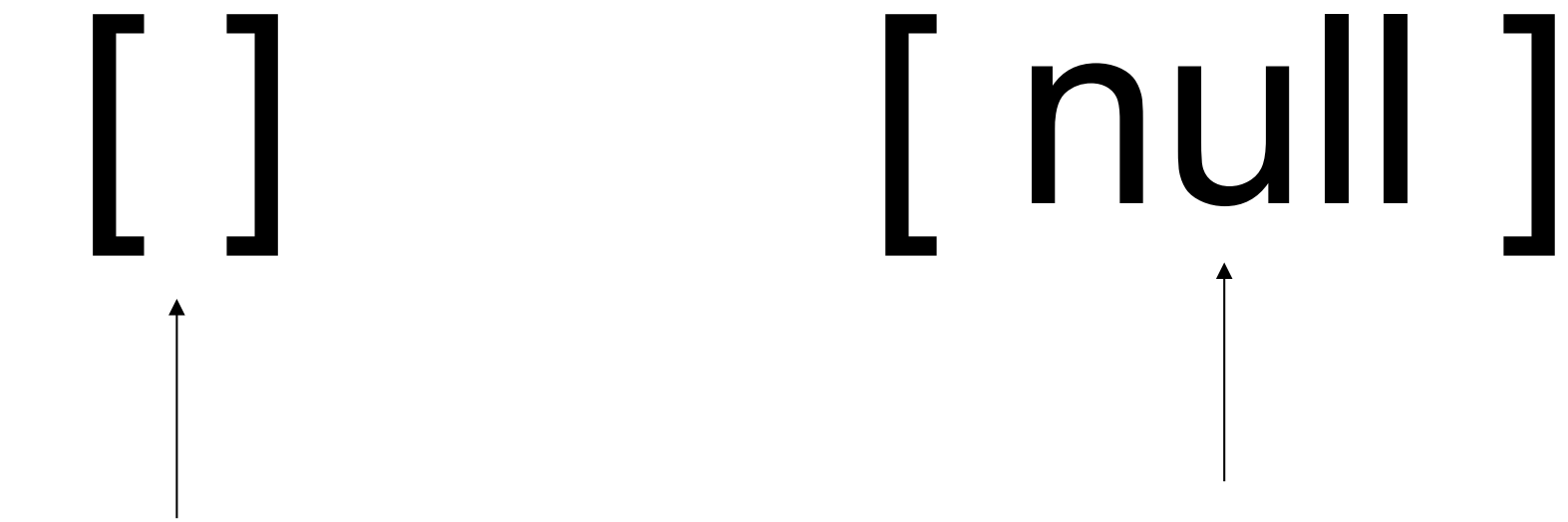


Array

```
["Sunday", "Monday", "Tuesday",
  "Wednesday", "Thursday",
  "Friday", "Saturday"]


[
   [0,-1, 0],
   [1, 0, 0],
   [0, 0, 1]
]
```

Empty array vs. array with a null value



Arrays vs Objects

- Use objects when the key names are arbitrary strings.
- Use arrays when the key names are sequential integers.
- Don't get confused by the terms Associative-Array (=list of key-value pairs) and Array (=list of values)

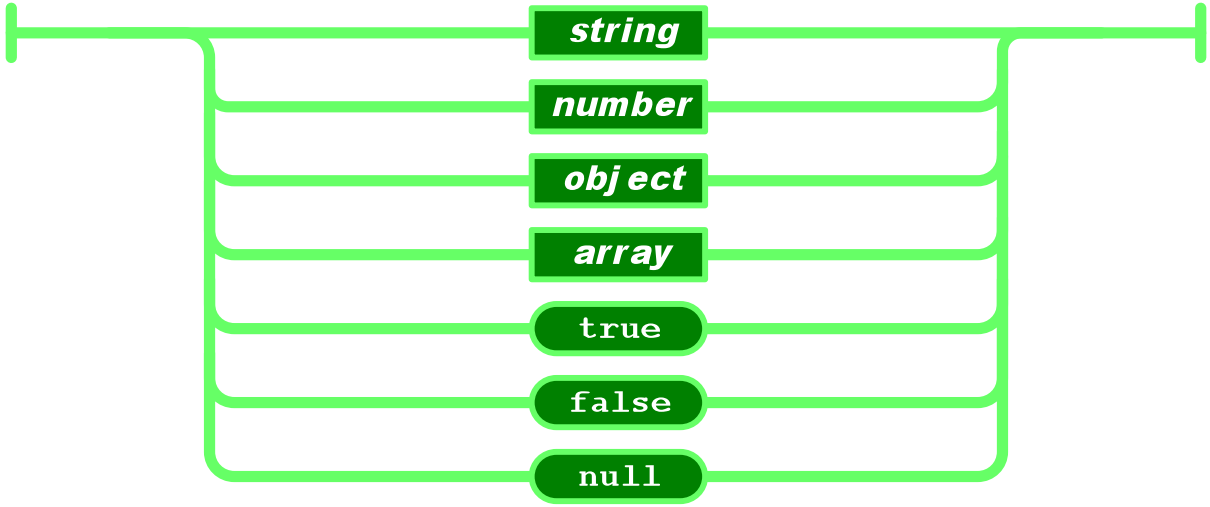
Values

- Strings
- Numbers
- Booleans

- Objects
- Arrays

- `null`

Value

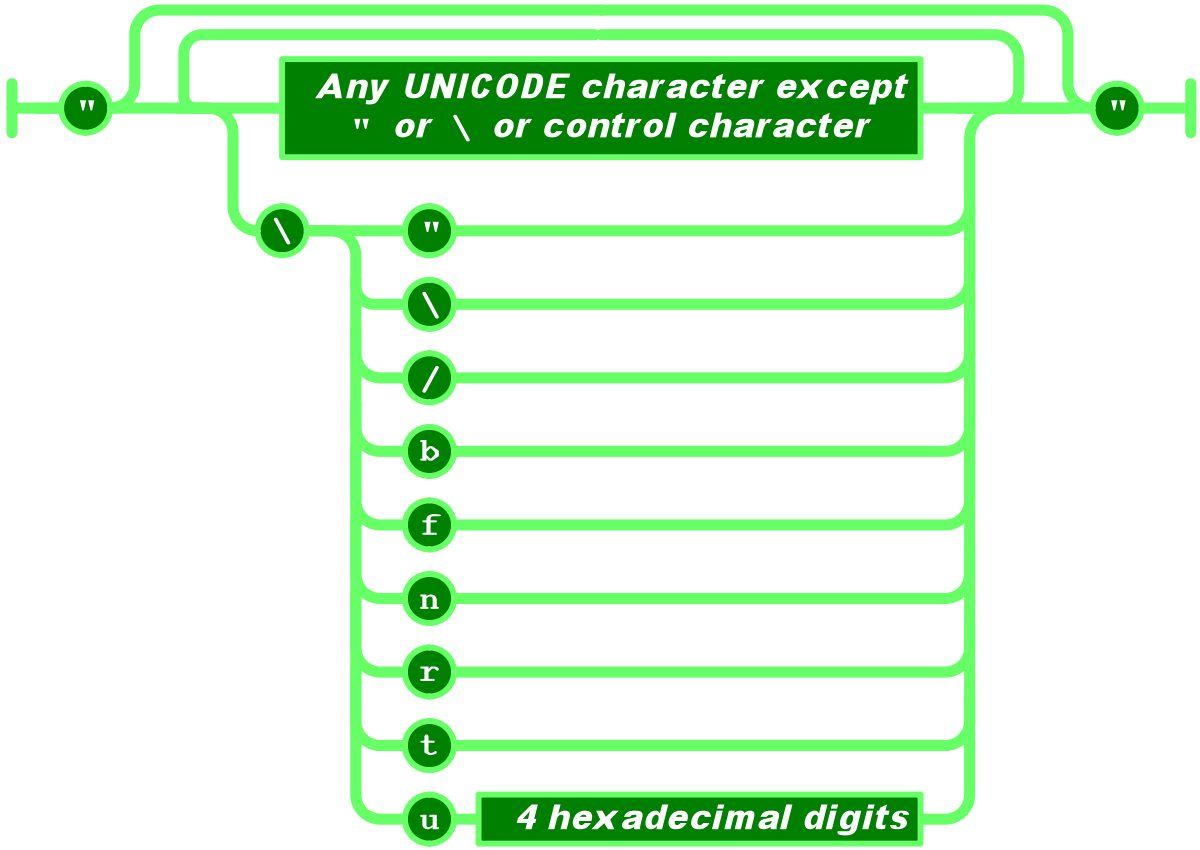


Strings

- Sequence of 0 or more Unicode characters
- No separate character type

  A character is represented as a string with a length of 1
- Wrapped in "double quotes"
- Backslash escapement

String

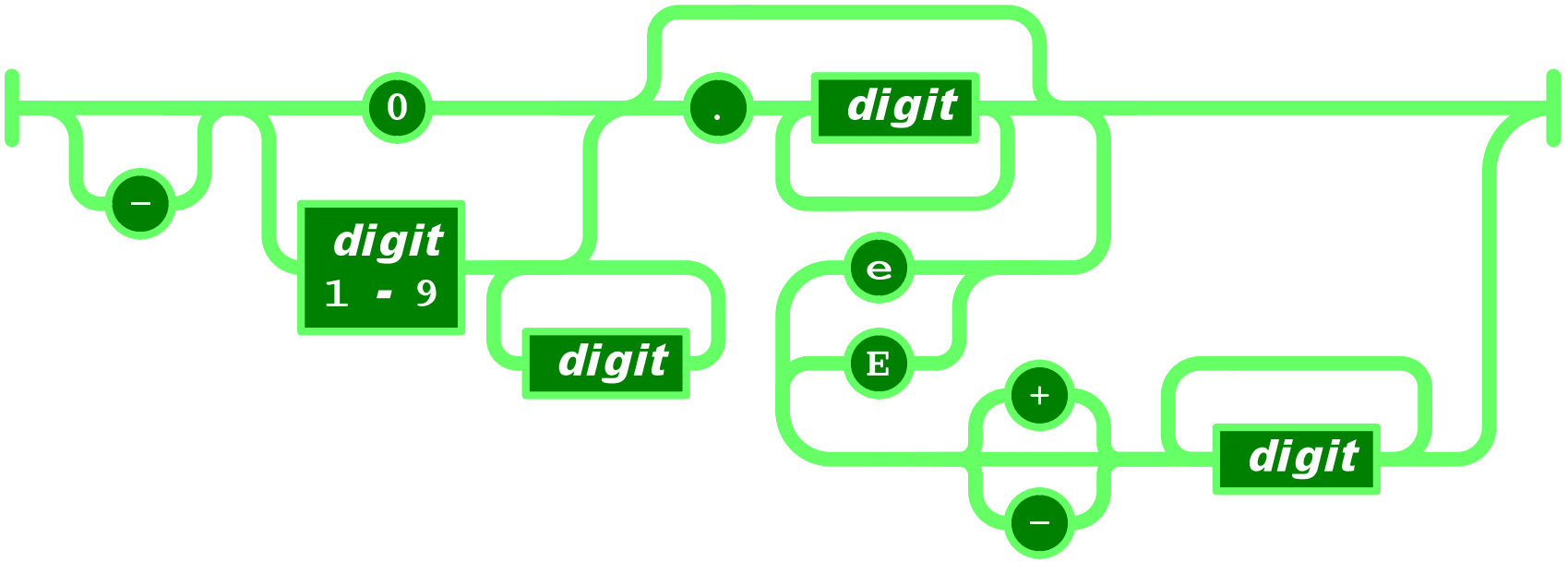

"
Any UNICODE character except
" or \ or control character
"
\
"
\
/
b
f
n
r
t
u
4 hexadecimal digits

Numbers

- Integer
- Real
- Scientific

- No octal or hex
- No **`NaN`** or **`Infinity`**
  - Use **`null`** instead

Number



Booleans

- `true`
- `false`

`null`

- A value that isn't anything

Booleans and `null`

```
{
    "name":       "Jack B. Nimble",
    "at large":   true,
    "grade":      null

}
```

Is this a legal JSON instance ?

42

so is this

"Hello World"

and so is this

true

and this

[true, null, 12, "ABC"]

JSON Is Not XML

- objects
- arrays
- strings
- numbers
- Booleans
- `null`

- order

ACCESSING JSON DATA

JSON Paths

```
json.address[1].streetAddress
```

- A JSON path expression can address
  - entire object, a scalar value, an array, a nested-object
- Similar to XPath expressions
  - the entire document is referenced by $
  - Key-names are separated by a .dot
  - Positions 0,1,..,n only matters for arrays (not for keys)
  - They are case sensitive

How to access JSON object in JavaScript

```
{
    "name": "mkyong",
    "age": 30,
    "address": {
       "streetAddress": "8nd St",
       "city": "New York"
    },
    "phoneNumber": [
        {
            "type": "home",
            "number": "111 111-1"
        },
        {
            "type": "fax",
            "number": "222 222-2"
        }
    ]
}
```

```
<script>
    var data = '{"name": "mkyong","age": 30,"address":
    {"streetAddress": "88 8nd Street","city": "New
    York"},"phoneNumber": [{"type": "home","number":
    "111 111-1111"},{"type": "fax","number": "222
    222-2222"}]}';

    var json = JSON.parse(data);

    alert(json.name);                       // yields mkyong

    alert(json.address.streetAddress);      //88nd Street

    alert(json["address"].city);            //New York

    alert(json.phoneNumber[0].number);      //111 111-1

    alert(json.phoneNumber[1].type);        //fax

    alert(json.phoneNumber.number);         //undefined

</script>
```

JPath : XPath for JSON

jpath.js disponible https://github.com/stsvilik/jPath

- JPath is a simple lightweight Javascript Class which provides an XPath-like querying ability to JSON objects.

JPath

```
var library = {
    'name' :'My Library',
    '@open' : '2007-17-7',
    'address' : {
      'city' : 'Springfield',
      'zip' : '12345',
      'state' : 'MI',
       'street' :
          'Mockingbird Lane'
    }
}
```

```
var jp = new JPath( library);

jp.$('address/city').json;

//will result in "Springfield"
```

JPath

```
var library = {
 'books':[
   {'title' : 'Harry Potter',
   'isbn'  : '1234-1234',
   'category' : 'Childrens',
   'available' : '3',
   'chapters' : [ 'Chapter 1',
   'Chapter 2' ] },

   {'title' : 'Brief History',
   'isbn'  : '1234-ABCD',
   'category' : 'Science',
   'chapters' : [ '1', '2' ] },

   {'title' : 'Lord of the Rings',
   'isbn'  : '1234-PPPP',
   'category' : 'Fiction',
   'chapters' : [ 'Section 1',
   'Section 2' ]  }
  ] }
```

```
var jp = new JPath( library);

//the Harry Potter book object

jp.$('books').$(1).json;


//all the books/category properties

jp.$('category').json;
```

JPath

```
var library = {
 'books':[
   {'title' : 'Harry Potter',
   'isbn'  : '1234-1234',
   'category' : 'Childrens',
   'available' : '3',
   'chapters' : [ 'Chapter 1',
   'Chapter 2' ] },

   {'title' : 'Brief History',
   'isbn'  : '1234-ABCD',
   'category' : 'Science',
   'chapters' : [ '1', '2' ] },

   {'title' : 'Lord of the Rings',
   'isbn'  : '1234-PPPP',
   'category' : 'Fiction',
   'chapters' : [ 'Section 1',
   'Section 2' ]  }
  ] }
```

```
var jp = new JPath( library);

//the Harry Potter book object

jp.query('books[1]');


//all the books/category properties

jp.query('//category');
```

JPath

```
var library = {
 'books':[
   {'title' : 'Harry Potter',
   'isbn'  : '1234-1234',
   'category' : 'Childrens',
   'available' : '3',
   'chapters' : [ 'Chapter 1',
   'Chapter 2' ] },

   {'title' : 'Brief History',
   'isbn'  : '1234-ABCD',
   'category' : 'Science',
   'chapters' : [ '1', '2' ] },

   {'title' : 'Lord of the Rings',
   'isbn'  : '1234-PPPP',
   'category' : 'Fiction',
   'chapters' : [ 'Section 1',
   'Section 2' ]  }
   ] }
```

```
var jp = new Jpath(library);

//all book objects that have a
    property of available > 0

jp.query('/books[available]');

//a more advanced query showing
    how predicates can be nested.
    this will return 'Fiction Books'

jp.query("  categories[name ==
              /books[last()]
              /category ]
                      /description ");
```

JPath's List of Operators

/tagname
//tagname
tagname
* wildcard
[] predicates
operators (>=, ==, <=)
array selection
..
*
and, or
nodename[0]
nodename[last()]
nodename[position()]
nodename[last()-1]
nodename[somenode > 3]/node
nodename[count() > 3]/node