JSON – JavaScript Object Notation

Some of the slides taken from Douglas Crockford / Virgule-Solidus

What are We Talking About Today?

- What is JSON?
  - Lightweight, text-based data interchange format
  - Used for representing data structures in web apps
- Syntax and Data Types
  - Supports strings, numbers, booleans, null, arrays, and objects
  - Key-value pairs within {} for objects and [] for arrays
- Usage
  - Transmits data between server and client
  - Integrates with various programming languages and frameworks

What is JSON

- **JSON**, short for **JavaScript Object Notation**, is a lightweight data interchange format.
  - It is a text-based, human-readable format for representing simple data structures and associative arrays (called objects).
- The JSON format was originally specified in **RFC 4627** (**July 2006**) authored by Douglas Crockford.
  - https://tools.ietf.org/rfc/rfc4627.txt
  - The official MIME type for JSON is **application/json**. The JSON file extension is .json.
- The JSON format is often used for transmitting structured data over a network connection in a process called serialization.
  - Its main application is in Ajax web application programming, where it serves as an alternative to the use of the XML format.
- Code for parsing and generating JSON data is readily available for a large variety of programming languages. The **www.json.org** website provides a comprehensive listing of existing JSON bindings, organized by language.

JSON Text

- JSON text is a serialized object or array
- ***JSON-text*** = Object / Array

JSON Basic Data Types

- ***String*** (double-quoted unicode with backslash escaping)
- ***Numbers*** (integer, real, or floating point)
- ***Booleans*** (true and false)
- ***Object*** (collection of key:value pairs, comma-separated and enclosed in curly brackets)
- ***Array*** (an ordered sequence of values, comma-separated and enclosed in square brackets)
- ***Null*** (a value that isn't anything)

String

- 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
- Empty strings are allowed

Object

- Objects are unordered containers of key/value pairs
- Objects are wrapped in **{ }**
- **,** separates key/value pairs
- **:** separates keys and values
- Keys are strings (unlike in JavaScript)
- Values are JSON values
- Can be used to represent struct, record, hashtable, object

Example Object

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

Example Object Formatted

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

Array

- Arrays are ordered sequences of values
- Arrays are wrapped in [ ] (square brackets)
- Commas (,) separates values

Two Examples of JSON Arrays

- **One dimensional**

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

- **Two dimensional**

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

Arrays vs Objects

- Use objects when the key names are arbitrary strings
- Use arrays when the key names are sequential integers

Brief Early History

- JSON was based on a subset of the JavaScript programming language (specifically, Standard ECMA-262 - now in its 15th Edition)
  - however, it is a language-independent data format.
  - For the complete specification see

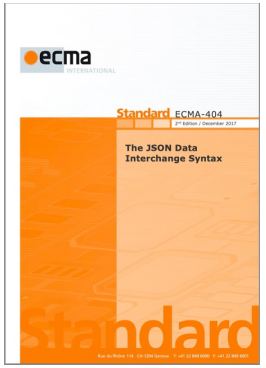
  https://www.ecma-international.org/publications/standards/Ecma-262.htm

- Douglas Crockford was the original developer of JSON while he was at State Software, Inc. He is now Senior JavaScript Architect at Paypal
- http://www.json.org/, is a website devoted to JSON discussions and includes many JSON parsers
- See Crockford's mini-talk on JSON vs. YAML: https://www.youtube.com/watch?v=126pVnFs2e4

Recent Developments History

- In December 2017, ECMA released **ECMA-404**, "The JSON data interchange syntax," 2nd Edition.
  - It includes several modification to the JSON syntax
  - For the complete specification see

  https://ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf



- Since 2006, several RFCs were issued by the Internet Engineering Task Force (IETF), titled "The JavaScript Object Notation (JSON) Data Interchange Format".
- The current Internet Standard RFC, authored by **Tim Bray** of Textuality (formerly at Google and Amazon) is **RFC 8259**, released in December 2017.
- This RFC recommends avoiding some practices included in ECMA-404 in the "interest of maximal interoperability."
- See the complete specification at: https://datatracker.ietf.org/doc/html/rfc8259

JSON Grammar (2017)

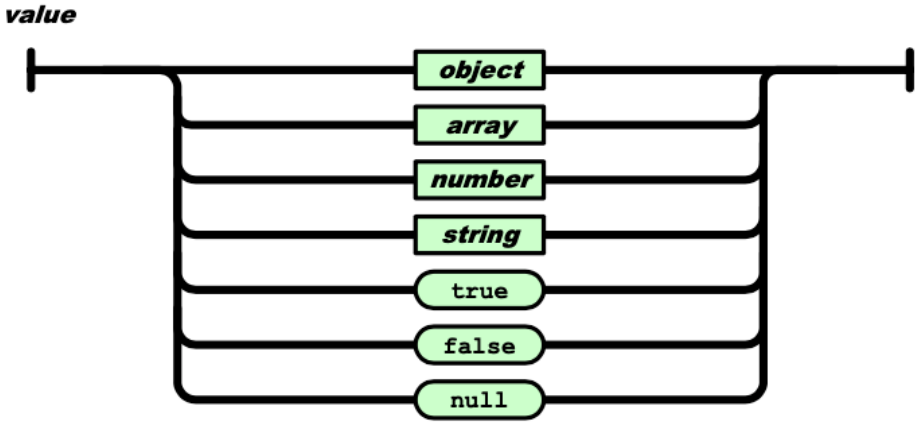
- A JSON text is a sequence of tokens. The set of tokens includes six structural characters, strings, numbers, and three literal names.

```
JSON-text = ws value ws

  These are the six structural characters:

     begin-array     = ws %x5B ws  ; [ left square bracket

     begin-object    = ws %x7B ws  ; { left curly bracket

     end-array       = ws %x5D ws  ; ] right square bracket

     end-object      = ws %x7D ws  ; } right curly bracket

     name-separator  = ws %x3A ws  ; : colon

     value-separator = ws %x2C ws  ; , comma

ws = *(
              %x20 /              ; Space
              %x09 /              ; Horizontal tab
              %x0A /              ; Line feed or New line
              %x0D )              ; Carriage return
```

Values

- A JSON value MUST be an object, array, number, or string, or one of the following three literal names:

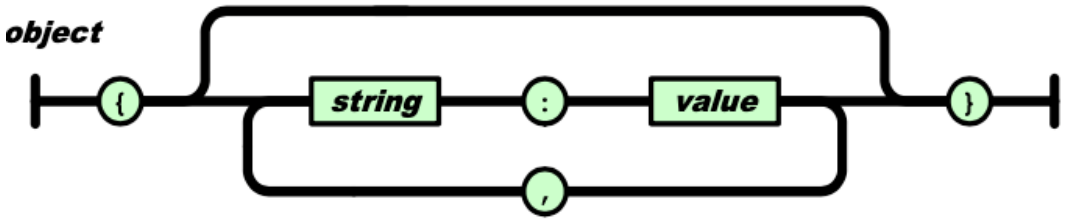
```
   false
   null
   true

The literal names MUST be lowercase.  No other literal names are
allowed.

   value = false / null / true / object / array / number / string

   false = %x66.61.6c.73.65   ; false

   null  = %x6e.75.6c.6c      ; null

   true  = %x74.72.75.65      ; true
```



Objects

- An object structure is represented as a pair of curly brackets surrounding zero or more name/value pairs (or members).
- A name is a string. A single colon comes after each name, separating the name from the value.
- A single comma separates a value from a following name. The names within an object SHOULD be unique.

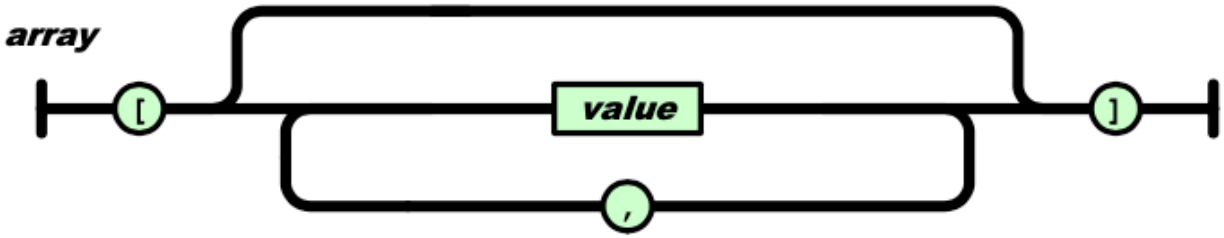
```
object = begin-object [ member *( value-separator member ) ]
              end-object

     member = string name-separator value
```



Arrays

- An array structure is represented as square brackets surrounding zero or more values (or elements).
- Elements are separated by commas.
- There is no requirement that the values in an array be of the same type.

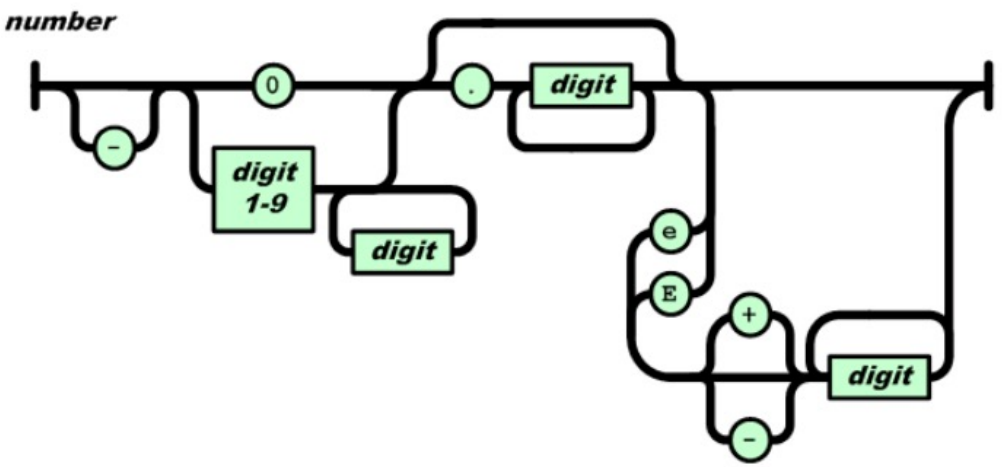
```
array = begin-array [ value *( value-separator value ) ] end-array
  type.
```



Numbers

- A number is represented in base 10 using decimal digits. It contains an integer component that may be prefixed with an optional minus sign, which may be followed by a fraction part and/or an exponent part. Leading zeros are not allowed.

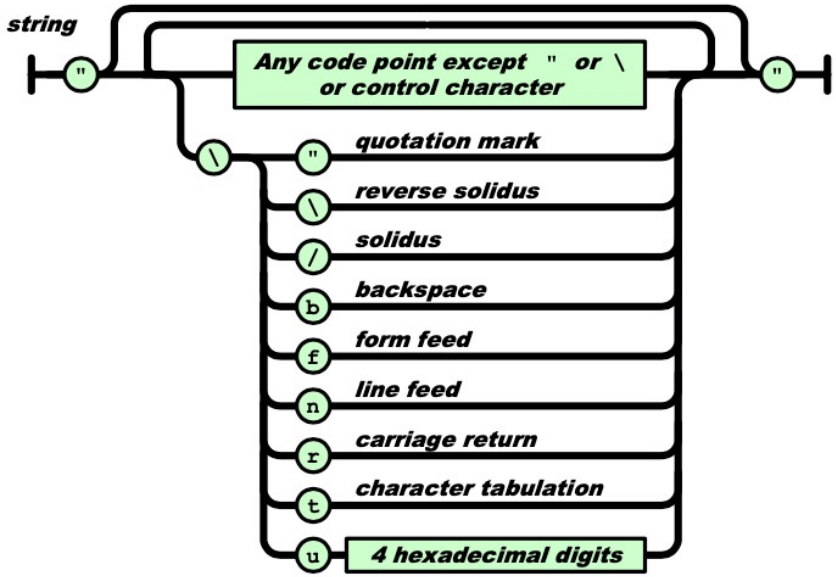
```
number = [ minus ] int [ frac ] [ exp ]

      decimal-point = %x2E        ; .

      digit1-9 = %x31-39          ; 1-9

      e = %x65 / %x45             ; e E

      exp = e [ minus / plus ] 1*DIGIT

      frac = decimal-point 1*DIGIT

      int = zero / ( digit1-9 *DIGIT )

      minus = %x2D                ; -

      plus = %x2B                 ; +

      zero = %x30                 ; 0
```



Strings

- The representation of strings is similar to conventions used in the C family of programming languages. A string begins and ends with quotation marks. All Unicode characters may be placed within the quotation marks, except for the characters that MUST be escaped.

```
string = quotation-mark *char quotation-mark

     char = unescaped /
         escape (
             %x22 /          ; "    quotation mark  U+00
             %x5C /          ; \    reverse solidus U+00
             %x2F /          ; /    solidus         U+00
             %x62 /          ; b    backspace       U+00
             %x66 /          ; f    form feed       U+00
             %x6E /          ; n    line feed       U+00
             %x72 /          ; r    carriage return U+00
             %x74 /          ; t    tab             U+00
             %x75 4HEXDIG )  ; uXXXX                U+XX

     escape = %x5C              ; \

     quotation-mark = %x22      ; "

     unescaped = %x20-21 / %x23-5B / %x5D-10FFFF
```



Interoperability Issues

- Character Encoding
  - JSON text exchanged between systems that are not part of a closed ecosystem MUST be encoded using UTF-8 [RFC3629].
  - Previous specifications of JSON have not required the use of UTF-8 when transmitting JSON text.
- Unicode Characters
  - When all the strings represented in a JSON text are composed entirely of Unicode characters [UNICODE] (however escaped), then that JSON text is interoperable.
- Parsers
  - A JSON parser transforms a JSON text into another representation. A JSON parser MUST accept all texts that conform to the JSON grammar. A JSON parser MAY accept non-JSON forms or extensions.
- Generators
  - A JSON generator produces JSON text. The resulting text MUST strictly conform to the JSON grammar.

How to use the JSON format

- A JSON file (or data stream) allows one to load data from the server or to send data to it.
- Working with JSON involves three steps: (i) the client (browser) processing, (ii) the server processing, and (iii) the data exchange between them.

1. **Client side (browser)**
   - The content of a JSON file (or stream), or the definition of JSON data is assigned to a variable, and this variable becomes an object of the program.
2. **Server side**
   - A JSON file (or data stream) on the server can be operated upon by various programming languages, including PHP and Java thanks to parsers that process the file and may even convert it into classes and attributes of the language.
3. **Data exchange**
   - Loading a JSON file from the server may be accomplished in JavaScript in several ways:
   - directly including the file into the HTML page, as a JavaScript **.json external file**.
   - loading by a JavaScript command
   - using **fetch()** or **XMLHttpRequest()**
   - To convert JSON into an object, it can be parsed with **JSON.parse()** or it can be passed to the JavaScript **eval()** function – insecure way!
   - Sending the file to the server may be accomplished by **fetch() or XMLHttpRequest().**

   The file is sent as a text file and processed by the parser of the programming language that uses it.

JSON and fetch Example

- **Fetch** will be covered in more detail in the AJAX lecture
- **The fetch code:**

```
// Mocking the fetch().then(response => response.json())
behavior:
const response = '{"menu": {"name": "menuName"}, "commands":
[{"title": "MyTitle", "action": "MyAction"}]}'
const doc = JSON.parse(response) // Now doc is a json-parsed
object with the following structure:
console.log(doc)
// outputs {commands: [{title: "MyTitle", action: "MyAction"}],
menu: {name: "menuName"}}
```

- **Using the data:**

```
var menuName = doc.menu.name;// finding a field menu
doc.menu.name = "new name";  // assigning a value to the field
```

- **If you want to update the DOM:**

```
const menuElement = document.getElementById('menu')
menuElement.innerText = doc.menu.name // sets 'new name' as a
   text content of the DOM element with id='menu'
```

JSON and fetch Example

- **Fetch** will be covered in more detail in the AJAX lecture

- **The fetch code:**

```
const doc = fetch('https://mywebsite/file.json')
            .then(response => response.json()) // parses response
as json
```

- **Using the data:**

```
var menuName = doc.getElementById('menu'); // finding a field
  menu
doc.menu.value = "my name is"; // assigning a value to the
  field
```

- **How to access data:**

```
doc.commands[0].title // read value of the "title" field in
  the array
doc.commands[0].action // read value of the "action" field in
  the array
```

JSON and ESM imports

- **As JSON is a valid JS object, you can import them as a regular JS module (CORS still applies)**
- **Using the import function:**

```
const data = await import("https://example.com/1.json", { with: { type:
"json" } });
console.log(data.default.title); // default import
```

```
> const data = await import("https://jsonplaceholder.typicode.com/todos/1", {with: {type: "json"}})
< undefined
> data.default
< ▼ {userId: 1, id: 1, title: 'delectus aut autem', completed: false} i
    completed: false
    id: 1
    title: "delectus aut autem"
    userId: 1
  ▶ [[Prototype]]: Object
```

- **Using the import statement:**

```
<script type="module">
  import data from "https://example.com/1.json" with { type: "json" };
  const p = document.createElement("p");
  p.textContent = `name: ${data.title}`;
  document.body.appendChild(p);
</script>
```

JSON and XMLHttpRequest Example

- **XMLHttpRequest** will be covered in more detail in the AJAX lecture
- **The XMLHttpRequest code:**

```
var req = new XMLHttpRequest();
req.open("GET", "file.json", true); // "asynchronous" operation
req.onreadystatechange = myCode; // the callback
req.send(null);
```

- **The JavaScript callback: JSON.parse() or eval() parses JSON, creates an object and assigns it to variable doc**

```
function myCode() {
  if (req.readyState == 4 && req.status == 200) {
    var doc = JSON.parse(req.responseText);
    // or
    var doc = eval('(' + req.responseText + ')');
 }
}
```

- **Using the data:**

```
var menuName = doc.getElementById('menu'); // finding a field menu
doc.menu.value = "my name is"; // assigning a value to the field
```

- **How to access data:**

```
doc.commands[0].title // read value of the "title" field in the array
doc.commands[0].action // read value of the "action" field in the array
```

JavaScript eval()

- The JavaScript **`eval()`** is a function property of the Global Object, and evaluates a string and executes it as if it was JavaScript code, e.g.

```
<script type="text/javascript">
eval("x=10;y=20;document.write(x*y)");
document.write("<br />");
document.write(eval("2+2"));
document.write("<br />");
var x=10;
document.write(eval(x+17));
document.write("<br />"); </script>
```

- produces the output

```
200
4
27
```

- Because JSON-formatted text is also syntactically legal JavaScript code, an easy way for a JavaScript program to parse JSON-formatted data is to use the built-in JavaScript eval() function
- the JavaScript interpreter itself is used to *execute* the JSON data to produce native JavaScript objects.
- The eval() technique is subject to security vulnerabilities if the data and the entire JavaScript environment is not within the control of a single trusted source; See:
- https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/eval

JSON is Not XML

JSON

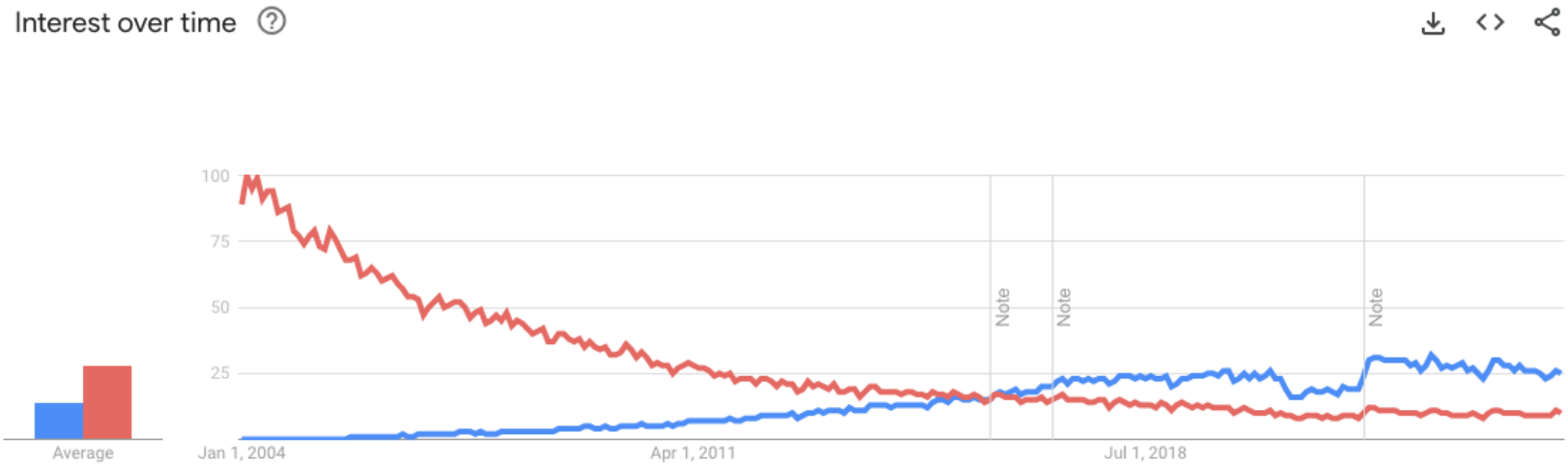
- Objects
- Arrays
- Strings
- Numbers
- Booleans
- null

XML

- element
- attribute
- Attribute string
- content
- <![CDATA[ ]]>
- Entities
- Declarations
- Schema
- Stylesheets
- Comments
- Version
- namespace

JSON vs XML

See Google Trends: https://trends.google.com/



JSON vs. XML Example

JSON

```
{
  "menu": "File",
  "commands": [
    {
      "title": "New",
      "action": "CreateDoc"
    },
    {
      "title": "Open",
      "action": "OpenDoc"
    },
    {
      "title": "Close",
      "action": "CloseDoc"
    }
  ]
}
```

XML Equivalent

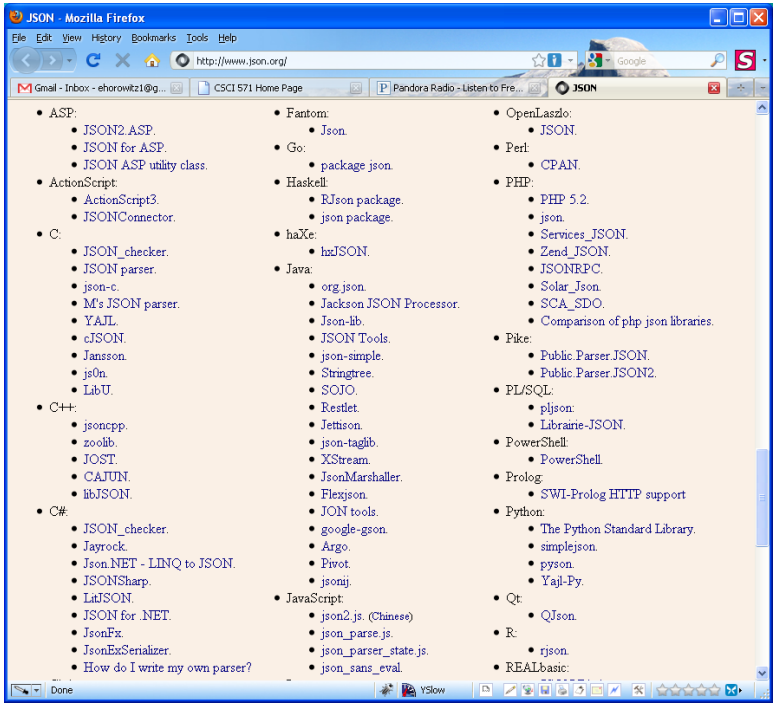
```
<?xml version="1.0" ?>
<root>
  <menu>File</menu>
  <commands>
    <item>
      <title>New</title>
      <action>CreateDoc</action>
    </item>
    <item>
      <title>Open</title>
      <action>OpenDoc</action>
    </item>
    <item>
      <title>Close</title>
      <action>CloseDoc</action>
    </item>
  </commands>
</root>
```

Rules for JSON Parsers / Generators

- A JSON decoder (parser) must accept all *well-formed* JSON text
- A JSON decoder may also accept non-JSON text
- A JSON encoder (generator) must only produce well-formed JSON text
- A list of decoders for JSON can be found at

http://www.json.org/

JSON parsers / generators for programming languages include C, C++, C#, Java, JavaScript, Perl, PHP, etc. →



Same Origin Policy

- Same origin policy is a security feature that browsers apply to client-side scripts
- It prevents a document or script loaded from one "origin" from getting or setting properties of a document from a different "origin"
  - Rationale: the browser should not trust content loaded from arbitrary websites
- Given the URL: http://www.example.com/dir/page.html

URL	Outcome	Reason
http://www.example.com/dir2/other.json	Success	Same protocol and host
http://www.example.com/dir/inner/other.json	Success	Same protocol and host
http://www.example.com:81/dir2/other.json	Failure	Same protocol and host but different port
https://www.example.com/dir2/other.json	Failure	Different protocol
http://en.example.com/dir2/other.json	Failure	Different host
http://example.com/dir2/other.json	Failure	Different host

JSON: The Cross-Domain Hack

- JSON and the **<script> tag** provide a way to get around the Same Origin Policy

<script src=http://otherdomain.com/data.js>

</script>

- The **src** attribute of a script tag can be set to a **URL from any server**, and every browser will go and retrieve it, and read it into your page
- So, a script tag can be set to point at a URL on another server with JSON data in it, and that JSON will become a global variable in the webpage
- So JSON can be used to grab data from other servers, without the use of a server-side proxy
- Available in **HTML** since 1994

JSON and Dynamic Script Tag "Hack"

- Using JSON, it is possible to get around the limitation that data can only come from a single domain (bypasses cross-domoan)
- To do this one needs to
  - to find a website that returns JSON data, and
  - A JavaScript program that contains a JSONScriptRequest class that creates a **dynamic <script> tag** and its contents
- The implementation of this class can be found at the class website:

http://csci571.com/examples/js/jsr_class.js

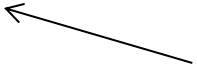
- The most important line in the script (the "hack") is the following one:

```
this.scriptObj.setAttribute("src", this.fullUrl + this.noCacheIE);
```

which sets the src attribute of the <script> tag to a new URL

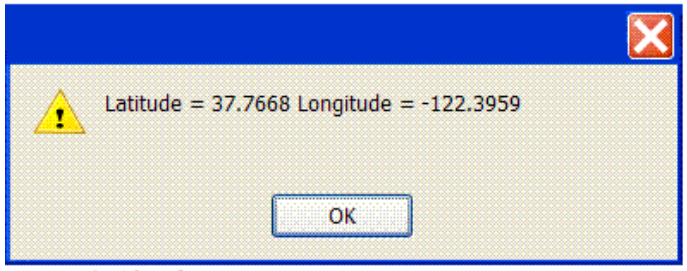
Source Code for jsr_class.js

```
// Constructor -- pass a REST request URL to the constructor
function JSONscriptRequest(fullUrl) {
            // REST request path
            this.fullUrl = fullUrl;
            // Keep IE from caching requests
            this.noCacheIE = '&noCacheIE=' + (new Date()).getTime();
            // Get the DOM location to put the script tag
            this.headLoc = document.getElementsByTagName("head").item(0);
            // Generate a unique script tag id
            this.scriptId = 'JscriptId' + JSONscriptRequest.scriptCounter++;  }
// Static script ID counter
JSONscriptRequest.scriptCounter = 1;
// buildScriptTag method
JSONscriptRequest.prototype.buildScriptTag = function () {
            // Create the script tag
            this.scriptObj = document.createElement("script");
            // Add script object attributes
            this.scriptObj.setAttribute("type", "text/javascript");
            this.scriptObj.setAttribute("charset", "utf-8");
            this.scriptObj.setAttribute("src", this.fullUrl + this.noCacheIE);
            this.scriptObj.setAttribute("id", this.scriptId); }
// removeScriptTag method
JSONscriptRequest.prototype.removeScriptTag = function () {
            // Destroy the script tag
            this.headLoc.removeChild(this.scriptObj); }
// addScriptTag method
JSONscriptRequest.prototype.addScriptTag = function () {
            // Create the script tag
            this.headLoc.appendChild(this.scriptObj); }
```



Example Using JSON and JSONscriptRequest class

```
<html><body>
// Include the JSONscriptRequest class
<script type="text/javascript" src="jsr_class.js"> </script>
<script type="text/javascript">
 // Define the callback function
 function getGeo(jsonData) {
   alert('Latitude = ' + jsonData.ResultSet.Result[0].Latitude + ' Longitude = ' +
     jsonData.ResultSet.Result[0].Longitude);
   bObj.removeScriptTag();
 }
// The web service call
var req  = 'http://api.local.yahoo.com/MapsService/V1/geocode?appid=YahooDemo&output=json&callback=getGeo&location=94107';
// Create a new request object
bObj = new JSONscriptRequest(req);
// Build the dynamic script tag
bObj.buildScriptTag();
// Add the script tag to the page
bObj.addScriptTag();
</script></body></html>
```



output

buildScriptTag creates
<script src="getGeo({"ResultSet":{"Result":[{"precision":"zip",...>

Adding the <script> tag to the page causes getGeo to be called And the JSON-encoded data to be passed to the getGeo function; The JavaScript interpreter automatically turns JSON into a JavaScript object, and the returned data can be referenced Immediately. See:

https://www.xml.com/pub/a/2005/12/21/json-dynamic-script-tag.html

Tiingo.com Example

- Tiingo provides Stock Lookup and Stock Quote APIs. Results are in JSON format.
- To access the service, you do not need an Application ID.
- Stock End-of-Day Quote JSON REST call looks like this:

  `https://api.tiingo.com/tiingo/daily/aapl/prices?startDate=2019-01-02&token=<token>`

- Response:


```
[ { "date":"2019-01-02T00:00:00.000Z", "close":157.92, "high":158.85,
"low":154.23, "open":154.89, "volume":37039737, "adjClose":157.92,
"adjHigh":158.85, "adjLow":154.23, "adjOpen":154.89, "adjVolume":37039737,
"divCash":0.0, "splitFactor":1.0 }, … ]
```


- Crypto Top-of-book JSON REST call looks like this:

`https://api.tiingo.com/tiingo/crypto/top?tickers=curebtc&token=<token>`

- Response:


```
[ { "ticker":"curebtc", "baseCurrency":"cure", "quoteCurrency":"btc",
"topOfBookData":[ { "askSize":21.55601545, "bidSize":726.29848588,
"lastSaleTimestamp":"2019-01-30T00:19:34.777000+00:00", "lastPrice":1.894e-
05, "askPrice":1.9e-05, "quoteTimestamp":"2019-01-
30T00:44:34.209957+00:00", "bidExchange":"BITTREX",
"lastSizeNotional":0.0010885247347072, "lastExchange":"BITTREX",
"askExchange":"BITTREX", "bidPrice":1.894e-05, "lastSize":57.47226688 } ] }
]
```

XMLHttpRequest Compared to the Dynamic Script Tag

	XmlHttpRequest	Dynamic script Tag
Cross-browser compatible?	No	Yes
Cross-domain browser security enforced?	Yes (*)	No
Can receive HTTP status codes?	Yes	No (fails on any HTTP status other than 200)
Supports HTTP GET and POST?	Yes	No (GET only)
Can send/receive HTTP headers?	Yes	No
Can receive XML?	Yes	Yes (but only embedded in a JavaScript statement)
Can receive JSON?	Yes	Yes (but only embedded in a JavaScript statement)
Offers synchronous and asynchronous calls?	Yes	No (asynchronous only)

Dynamic script tag is used by internet advertisers who use it to pull their ads into a web page

The script tag's main advantages are that it is not bound by the web browser's cross-domain security restrictions and that it runs identically on more web browsers than XMLHttpRequest.

If your web service happens to offer JSON output and a callback function, you can easily access web services from within your JavaScript applications without having to parse the returned data

* CORS-compatible browsers allow cross-domain XMLHttpRequest

Arguments against JSON

- JSON doesn't have namespaces
- JSON has no validator
  - Every application is responsible for validating its inputs
  - However, there exist standards like JSON Schema
- JSON is not extensible
  - But it does not need to be
- JSON is not XML
  - But a JavaScript compiler is a JSON decoder

Features that make JSON well-suited for data transfer

- It is both a human and machine-readable format;
- It has support for unicode, allowing almost any information in any human language to be communicated
- The format is self-documenting in that it describes structure and field names as well as specific values
- The strict syntax and parsing requirements allow the parsing algorithms to remain simple, efficient, and consistent
- JSON has the ability to represent the most general of computer science data structures: records, lists and trees

eval() and JSON.parse() Security

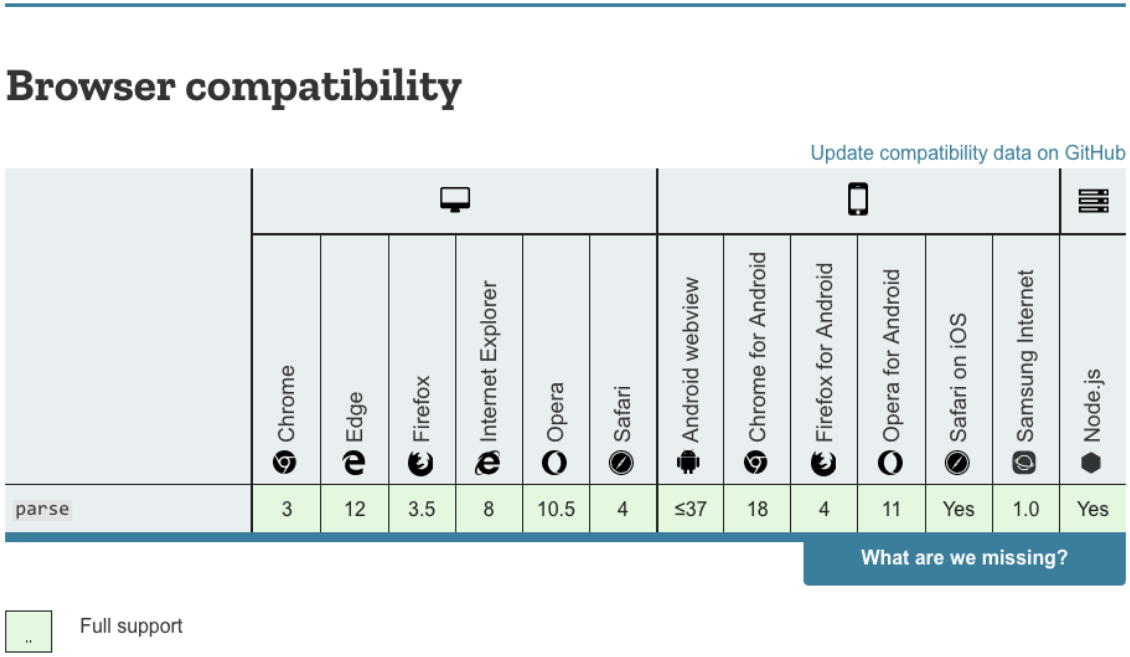
- The eval() function is very fast. However, it can compile and execute any JavaScript program, so there can be security issues
- In general
  - your browser should not trust machines not under your absolute control
  - Your server must validate everything the client tells it
- To help guard the browser from insecure JSON input, use **JSON.parse()** instead of eval ; e.g., JSON.parse() is used this way

  var myObject = **JSON.parse**(JSONtext [, reviver]);
- The optional reviver parameter is a function that will be called for every key and value at every level of the result. Each value will be replaced by the result of the reviver function. This can be used to reform generic objects into instances of pseudoclasses, or to transform date strings into Date objects.

More on JSON.parse()

- JSON.parse() is included in ECMAScript since 5th Ed. and all recent desktop browsers (Chrome, Firefox 3.5+, IE 8+, Opera 10.5+, Safari 4+) and all mobile browsers. JSON.parse() compiles faster than eval(). See:

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/JSON/parse



Browser compatibility

Update compatibility data on GitHub

	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet	Node.js
parse	3	12	3.5	8	10.5	4	≤37	18	4	11	Yes	1.0	Yes

What are we missing?

Full support

More on JSON.parse (cont'd)

- In JavaScript there is a function called JSON.parse. It uses a single call to eval to do the conversion, guarded by a single regexp test to assure that the input is safe.
- The input object is traversed recursively, and various functions are called for each member of the object in post-order (i.e. every object is reviewed after all its members have been reviewed).
- For each member, the following occurs:
  - If reviewer returns a valid value, the member value is replaced with the value returned by reviewer.
  - If reviewer returns what it received, the structure is not modified.
  - If reviewer returns null or undefined, the object member is deleted.
- The reviewer argument is often used to transform JSON representation of ISO date strings into UTC format Date objects.
- Here is the original source code for JSON.parse

```
JSON.parse = function (text) {
return
(/^(\s|[,:{}\[\]]|"(\\["\\bfnrtu]|[^\x00-\x1f"\\])*"|-
?\d+(\.\d*)?([eE][+-]?\d+)?|true|false|null)+$/.test(text))
&& eval('(' + text + ')'); };
```

- According to Doug Crockford, "It is ugly, but it is really efficient. "
- A newer implementation from IETF works like this:

```
var my_JSON_object = !(/[^,:{}\[\]0-9.\-+Eaeflnr-u
\n\r\t]/.test(text.replace(/"(\\.|[^"\\])*"/g, ''))) && eval('(' + text
+ ')');
```

- JQuery provides implicit ParseJSON() method and responseJSON property. See
  - http://api.jquery.com/jquery.parsejson/ and http://api.jquery.com/jQuery.ajax/

Douglas Crockford Discusses AJAX

- Go to
- https://www.youtube.com/watch?v=-C-JoyNuQJs
- Start video at -27 minutes and run it as long as it is interesting
- He gets to JSON at -13 minutes

Python Includes JSON functionality

- Python includes an encoder and a decoder. JSON functions include
  - json.dump – serialize object as JSON formatted string
  - *class* json.JSONDecoder - simple JSON decoder
  - *class* json.JSONEncoder – extensible JSON decoder for Python data structures
- See: https://docs.python.org/3/library/json.html

JSON	Python
object	dict
array	list
string	str
number (int)	int
number (real)	float
true	True
false	False
null	None

Compact Encoding JSON in Python

```
>>> import json
>>> json.dumps([1, 2, 3, {'4': 5, '6': 7}], separators=(',', ':’))
 '[1,2,3,{"4":5,"6":7}]'
```

Decoding JSON in Python

```
>>> import json
>>> json.loads('["foo", {"bar":["baz", null, 1.0, 2]}]’)
['foo', {'bar': ['baz', None, 1.0, 2]}]
>>> json.loads('"\\"foo\\bar”’)
'"foo\x08ar’
>>> from io import StringIO
>>> io = StringIO('["streaming API"]’)
>>> json.load(io) ['streaming API’]
['streaming API']
```

JSON in Java

- Use GSON library from Google:

https://github.com/google/gson

- Gson – How to convert Java object to / from JSON

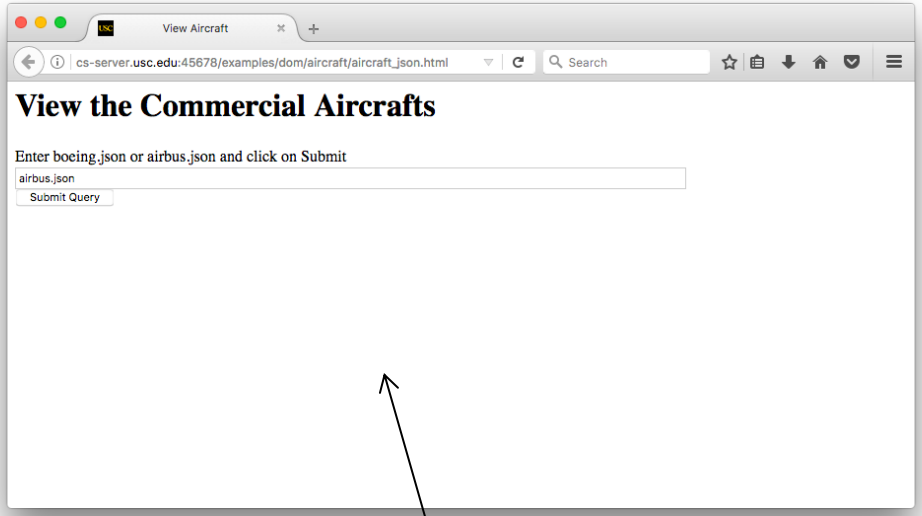
https://mkyong.com/java/how-do-convert-java-object-to-from-json-format-gson-api/

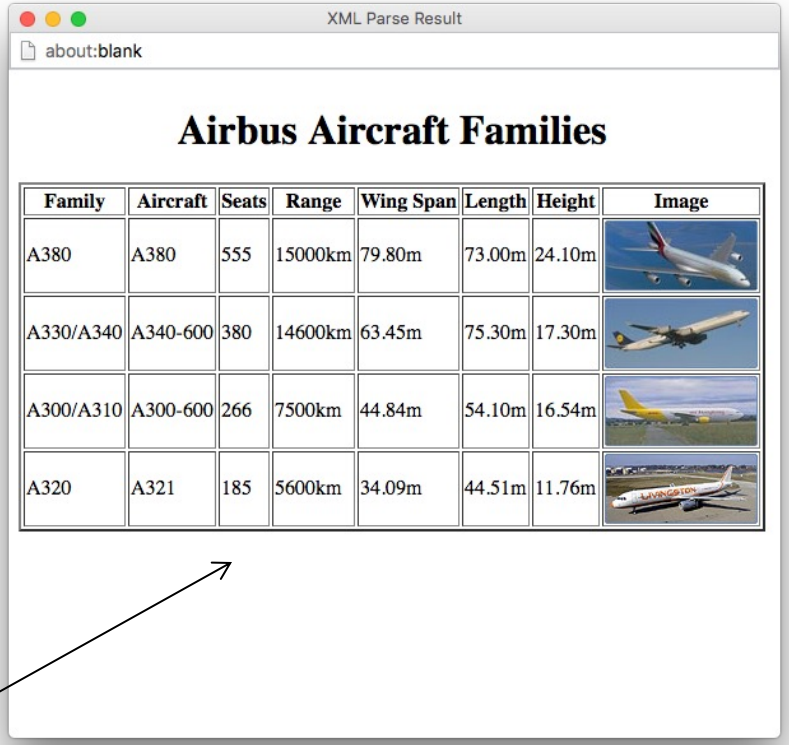
- Generate Plain Old Java Objects from JSON.

https://www.jsonschema2pojo.org/

Schema type		Java type
`string`		`java.lang.String`
`number`		`java.lang.Double`
`integer`		`java.lang.Integer`
`boolean`		`java.lang.Boolean`
`object`		*generated Java type*
`array`		`java.util.List`
`array`	(with `"uniqueItems":true`)	`java.util.Set`
`null`		`java.lang.Object`
`any`		`java.lang.Object`

Example 14: A Longer DOM Example in JSON

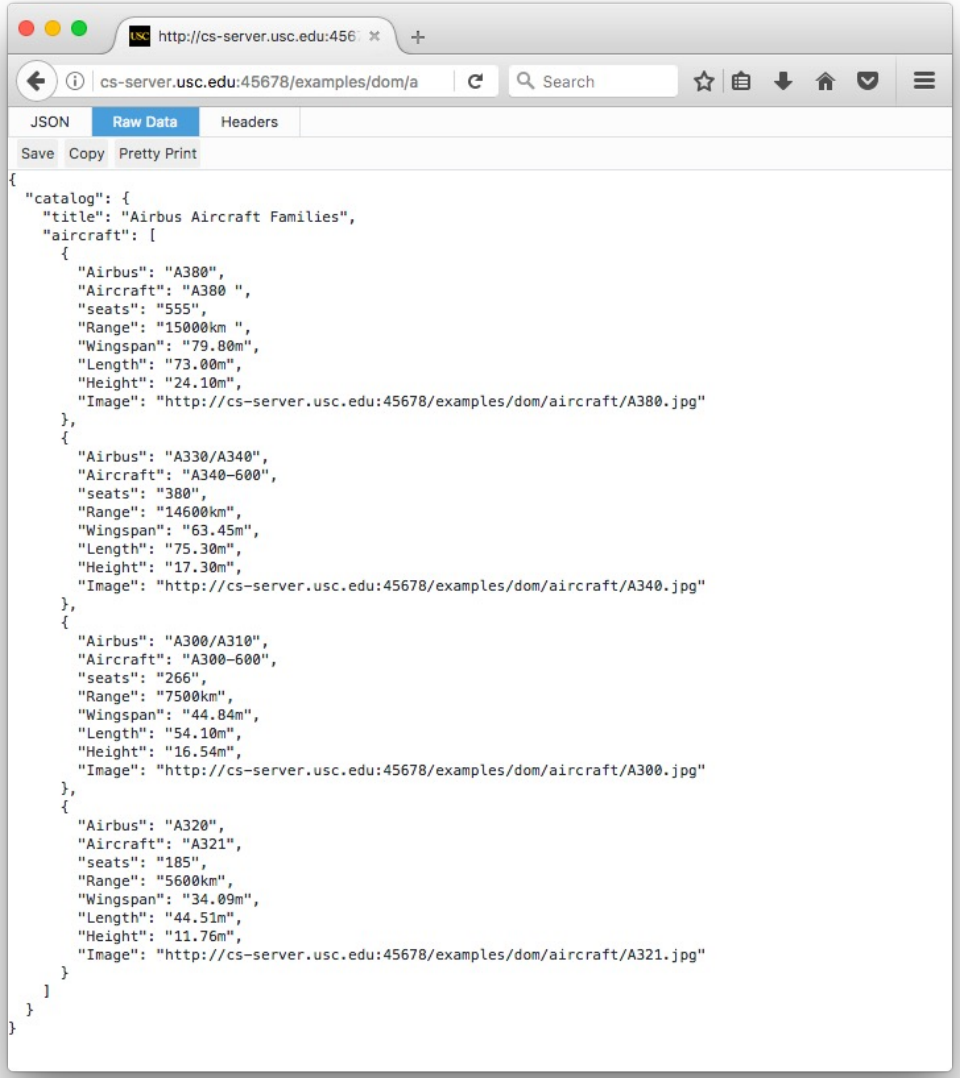




Given a URL of a JSON file that describes a set of aircraft, re-format the data into an HTML page.

See: https://csci571.com/examples/dom/aircraft/aircraft_json.html

airbus.json



HTML Code for the Initial Input

```
<h1>View the Commercial Aircrafts </h1>
Enter JSON file
<form name="myform" method="POST" id="location">
<input type="text" name="URL" maxlength="255" size="100"
    value="airbus.json" />
<br />
<input type="button" name="submit" value="Submit Query"
    onClick="viewJSON(this.form)" />
</form>
```

viewJSON Routine

```
function viewJSON(what) {
var URL = what.URL.value;

 function loadJSON(url) {
 xmlhttp=new XMLHttpRequest();
 xmlhttp.open("GET",url,false); // "synchronous” (deprecated because it freezes the page while waiting for a response) *
 xmlhttp.send();
 jsonObj= JSON.parse(xmlhttp.responseText);
 return jsonObj;
 }

 jsonObj = loadJSON(URL);
 jsonObj.onload=generateHTML(jsonObj);
 hWin = window.open("", "Assignment4", "height=800,width=600");
 hWin.document.write(html_text);
 hWin.document.close();
}
```

- See:

https://developer.mozilla.org/en-US/docs/Web/API/XMLHttpRequest/Synchronous_and_Asynchronous_Requests

generateHTML Routine

```
function generateHTML(jsonObj)   {
        root=jsonObj.DocumentElement;
        html_text="<html><head><title>JSON Parse Result</title></head><body>";
         html_text+="<table border='2'>";
        caption=jsonObj.catalog.title;
        html_text+="<caption align='left'><h1>"+caption+"</h1></caption>";
         planes=jsonObj.catalog.aircraft; // an array of planes
         planeNodeList=planes[0];
         html_text+="<tbody>";
        html_text+="<tr>";
        x=0;  y=0;
         // output the headers
         var header_keys = Object.keys(planeNodeList);
         for(i=0;i<header_keys.length;i++)          {
                    header=header_keys[i];
                    if(header=="Airbus") {  header="Family";  x=120;  y=55;  }
                    if(header=="Boeing") {  header="Family";  x=100;  y=67;  }
                    if(header=="seats")    header="Seats";
                    if(header=="Wingspan")   header="Wing Span";
                    if(header=="height")  header="Height";
                    html_text+="<th>"+header+"</th>";
        }
}
```

generateHTML Routine (cont'd)

```
html_text+="</tr>";
// output out the values
        for(i=0;i<planes.length;i++) //do for all planes (one per row)
       {
                planeNodeList=planes[i]; //get properties of a plane (an object)
                html_text+="<tr>";     //start a new row of the output table
                var aircraft_keys = Object.keys(planeNodeList);
                for(j=0;j<aircraft_keys.length;j++)
               {
                      prop = aircraft_keys[j];
                      if(aircraft_keys[j]=="Image")
                      {//handle images separately
                                  html_text+="<td><img src='"+ planeNodeList[prop] +"' width='"+x+"' height='"+y+"'></td>";
                      } else {
                                  html_text+="<td>"+ planeNodeList[prop] +"</td>";
                     }
        }
       html_text+="</tr>";
}
html_text+="</tbody>";
 html_text+="</table>";
html_text+= "</bo" + "<dy> </html>";  }
```

JSON variations and supersets

- JSON lacks some convenient and popular features, that's why there are a couple of variations worth noting:
  - **JSON5** (Superset of JSON – valid JSON is a valid JSON5, subset of JS ES5 – valid JSON5 is a valid JS)
    See: https://json5.org/, https://spec.json5.org/
  - **YAML** (Superset of JSON – valid JSON is a valid YAML)
    See https://yaml.org/

```
{
  // comments
  unquoted: 'and you can quote me on that',
  singleQuotes: 'I can use "double quotes" here',
  lineBreaks: "Look, Mom! \
No \\n's!",
  hexadecimal: 0xdecaf,
  leadingDecimalPoint: .8675309,
  andTrailing: 8675309.,
  positiveSign: +1,
  trailingComma: 'in objects',
  andIn: ['arrays',],
  "backwardsCompatible": "with JSON",
}
```

JSON5

```
#Comment: Student record
---
name: Martin D'vloper #key-value
age: 26
isStudent: true
hobbies:
- painting #first list item
- playing_music #second list item
- cooking #third list item
programming_languages:
java: Intermediate
python: Advanced
javascript: Beginner
favorite_food:
- vegetables: tomatoes
- fruits:
citrics: oranges
```

YAML