Chapter 35: JSON

Parameter	Details
JSON.parse	**Parse a JSON string**
`input(string)`	JSON string to be parsed.
`reviver(function)`	Prescribes a transformation for the input JSON string.
JSON.stringify	**Serialize a serializable value**
`value(string)`	Value to be serialized according to the JSON specification.
`replacer(function` or `String[]` or `Number[])`	Selectively includes certain properties of the `value` object.
`space(String` or `Number)`	If a `number` is provided, then `space` number of whitespaces will be inserted of readability. If a `string` is provided, the string (first 10 characters) will be used as whitespaces.

JSON (JavaScript Object Notation) is a lightweight data-interchange format. It is easy for humans to read and write and easy for machines to parse and generate. It is important to realize that, in JavaScript, JSON is a string and not an object.

A basic overview can be found on the json.org website which also contains links to implementations of the standard in many different programming languages.

Section 35.1: JSON versus JavaScript literals

JSON stands for "JavaScript Object Notation", but it's not JavaScript. Think of it as just a *data serialization format* that *happens* to be directly usable as a JavaScript literal. However, it is not advisable to directly run (i.e. through `eval()`) JSON that is fetched from an external source. Functionally, JSON isn't very different from XML or YAML – some confusion can be avoided if JSON is just imagined as some serialization format that looks very much like JavaScript.

Even though the name implies just objects, and even though the majority of use cases through some kind of API always happen to be objects and arrays, JSON is not for just objects or arrays. The following primitive types are supported:

- String (e.g. `"Hello World!"`)
- Number (e.g. 42)
- Boolean (e.g. **`true`**)
- The value **`null`**

`undefined` is not supported in the sense that an undefined property will be omitted from JSON upon serialization. Therefore, there is no way to deserialize JSON and end up with a property whose value is **`undefined`**.

The string `"42"` is valid JSON. JSON doesn't always have to have an outer envelope of `"{...}"` or `"[...]"`.

While some JSON is also valid JavaScript and some JavaScript is also valid JSON, there are some subtle differences between both languages and neither language is a subset of the other.

Take the following JSON string as an example:

```
{"color": "blue"}
```

This can be directly inserted into JavaScript. It will be syntactically valid and will yield the correct value:

```
const skin = {"color": "blue"};
```

However, we know that "color" is a valid identifier name and the quotes around the property name can be omitted:

```
const skin = {color: "blue"};
```

We also know that we can use single quotes instead of double quotes:

```
const skin = {'color': 'blue'};
```

But, if we were to take both of these literals and treat them as JSON, **neither will be syntactically valid** JSON:

```
{color: "blue"}
{'color': 'blue'}
```

JSON strictly requires all property names to be double quoted and string values to be double quoted as well.

It's common for JSON-newcomers to attempt to use code excerpts with JavaScript literals as JSON, and scratch their heads about the syntax errors they are getting from the JSON parser.

More confusion starts arising when *incorrect terminology* is applied in code or in conversation.

A common anti-pattern is to name variables that hold non-JSON values as "json":

```
fetch(url).then(function (response) {
  const json = JSON.parse(response.data); // Confusion ensues!

  // We're done with the notion of "JSON" at this point,
  // but the concept stuck with the variable name.
});
```

In the above example, `response.data` is a JSON string that is returned by some API. JSON stops at the HTTP response domain. The variable with the "json" misnomer holds just a JavaScript value (could be an object, an array, or even a simple number!)

A less confusing way to write the above is:

```
fetch(url).then(function (response) {
  const value = JSON.parse(response.data);

  // We're done with the notion of "JSON" at this point.
  // You don't talk about JSON after parsing JSON.
});
```

Developers also tend to throw the phrase "JSON object" around a lot. This also leads to confusion. Because as mentioned above, a JSON string doesn't have to hold an object as a value. "JSON string" is a better term. Just like "XML string" or "YAML string". You get a string, you parse it, and you end up with a value.

Section 35.2: Parsing with a reviver function

A reviver function can be used to filter or transform the value being parsed.

Version ≥ 5.1

```
var jsonString = '[{"name":"John","score":51},{"name":"Jack","score":17}]';

var data = JSON.parse(jsonString, function reviver(key, value) {
  return key === 'name' ? value.toUpperCase() : value;
});
```

Version ≥ 6

```
const jsonString = '[{"name":"John","score":51},{"name":"Jack","score":17}]';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'name' ? value.toUpperCase() : value
);
```

This produces the following result:

```
[
  {
    'name': 'JOHN',
    'score': 51
  },
  {
    'name': 'JACK',
    'score': 17
  }
]
```

This is particularly useful when data must be sent that needs to be serialized/encoded when being transmitted with JSON, but one wants to access it deserialized/decoded. In the following example, a date was encoded to its ISO 8601 representation. We use the reviver function to parse this in a JavaScript `Date`.

Version ≥ 5.1

```
var jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

var data = JSON.parse(jsonString, function (key, value) {
  return (key === 'date') ? new Date(value) : value;
});
```

Version ≥ 6

```
const jsonString = '{"date":"2016-01-04T23:00:00.000Z"}';

const data = JSON.parse(jsonString, (key, value) =>
  key === 'date' ? new Date(value) : value
);
```

It is important to make sure the reviver function returns a useful value at the end of each iteration. If the reviver function returns **`undefined`**, no value or the execution falls off towards the end of the function, the property is deleted from the object. Otherwise, the property is redefined to be the return value.

Section 35.3: Serializing a value

A JavaScript value can be converted to a JSON string using the `JSON.stringify` function.

```
JSON.stringify(value[, replacer[, space]])
```

1. `value` The value to convert to a JSON string.

```
/* Boolean */  JSON.stringify(true)             // 'true'
/* Number  */  JSON.stringify(12)               // '12'
/* String  */  JSON.stringify('foo')            // '"foo"'
/* Object  */  JSON.stringify({})               // '{}'
               JSON.stringify({foo: 'baz'})     // '{"foo": "baz"}'
/* Array   */  JSON.stringify([1, true, 'foo']) // '[1, true, "foo"]'
/* Date    */  JSON.stringify(new Date())       // '"2016-08-06T17:25:23.588Z"'
/* Symbol  */  JSON.stringify({x:Symbol()})     // '{}'
```

2. `replacer` A function that alters the behaviour of the stringification process or an array of String and Number objects that serve as a whitelist for filtering the properties of the value object to be included in the JSON string. If this value is null or is not provided, all properties of the object are included in the resulting JSON string.

```
// replacer as a function
function replacer (key, value) {
    // Filtering out properties
    if (typeof value === "string") {
        return
    }
    return value
}

var foo = { foundation: "Mozilla", model: "box", week: 45, transport: "car", month: 7 }
JSON.stringify(foo, replacer)
// -> '{"week": 45, "month": 7}'
```

```
// replacer as an array
JSON.stringify(foo, ['foundation', 'week', 'month'])
// -> '{"foundation": "Mozilla", "week": 45, "month": 7}'
// only the `foundation`, `week`, and `month` properties are kept
```

3. `space` For readability, the number of spaces used for indentation may be specified as the third parameter.

```
JSON.stringify({x: 1, y: 1}, null, 2)  // 2 space characters will be used for indentation
/* output:
    {
      'x': 1,
      'y': 1
    }
*/
```

Alternatively, a string value can be provided to use for indentation. For example, passing `'\t'` will cause the tab character to be used for indentation.

```
JSON.stringify({x: 1, y: 1}, null, '\t')
/* output:
    {
        'x': 1,
        'y': 1
    }
*/
```

Section 35.4: Serializing and restoring class instances

You can use a custom `toJSON` method and reviver function to transmit instances of your own class in JSON. If an object has a `toJSON` method, its result will be serialized instead of the object itself.

Version < 6

```
function Car(color, speed) {
  this.color = color;
  this.speed = speed;
}

Car.prototype.toJSON = function() {
  return {
    $type: 'com.example.Car',
    color: this.color,
```

```
    speed: this.speed
  };
};

Car.fromJSON = function(data) {
  return new Car(data.color, data.speed);
};
```

Version ≥ 6

```
class Car {
  constructor(color, speed) {
    this.color = color;
    this.speed = speed;
    this.id_ = Math.random();
  }

  toJSON() {
    return {
      $type: 'com.example.Car',
      color: this.color,
      speed: this.speed
    };
  }

  static fromJSON(data) {
    return new Car(data.color, data.speed);
  }
}
```

```
var userJson = JSON.stringify({
  name: "John",
  car: new Car('red', 'fast')
});
```

This produces the a string with the following content:

```
{"name":"John","car":{"$type":"com.example.Car","color":"red","speed":"fast"}}
```

```
var userObject = JSON.parse(userJson, function reviver(key, value) {
  return (value && value.$type === 'com.example.Car') ? Car.fromJSON(value) : value;
});
```

This produces the following object:

```
{
  name: "John",
  car: Car {
    color: "red",
    speed: "fast",
    id_: 0.19349242527065402
  }
}
```

Section 35.5: Serializing with a replacer function

A `replacer` function can be used to filter or transform values being serialized.

```
const userRecords = [
  {name: "Joe", points: 14.9, level: 31.5},
  {name: "Jane", points: 35.5, level: 74.4},
  {name: "Jacob", points: 18.5, level: 41.2},
```

```
  {name: "Jessie", points: 15.1, level: 28.1},
];

// Remove names and round numbers to integers to anonymize records before sharing
const anonymousReport = JSON.stringify(userRecords, (key, value) =>
  key === 'name'
    ? undefined
    : (typeof value === 'number' ? Math.floor(value) : value)
);
```

This produces the following string:

```
'[{"points":14,"level":31},{"points":35,"level":74},{"points":18,"level":41},{"points":15,"level":28}]'
```

Section 35.6: Parsing a simple JSON string

The `JSON.parse()` method parses a string as JSON and returns a JavaScript primitive, array or object:

```
const array = JSON.parse('[1, 2, "c", "d", {"e": false}]');
console.log(array); // logs: [1, 2, "c", "d", {e: false}]
```

Section 35.7: Cyclic object values

Not all objects can be converted to a JSON string. When an object has cyclic self-references, the conversion will fail.

This is typically the case for hierarchical data structures where parent and child both reference each other:

```
const world = {
  name: 'World',
  regions: []
};

world.regions.push({
  name: 'North America',
  parent: 'America'
});
console.log(JSON.stringify(world));
// {"name":"World","regions":[{"name":"North America","parent":"America"}]}

world.regions.push({
  name: 'Asia',
  parent: world
});

console.log(JSON.stringify(world));
// Uncaught TypeError: Converting circular structure to JSON
```

As soon as the process detects a cycle, the exception is raised. If there were no cycle detection, the string would be infinitely long.