Valid JSON

[

{

"name": "Molecule Man",

"age": 29,

"secretIdentity": "Dan Jukes",

"powers": ["Radiation resistance", "Turning tiny", "Radiation blast"]

},

{

"name": "Madame Uppercut",

"age": 39,

"secretIdentity": "Jane Wilson",

"powers": [

"Million tonne punch",

"Damage resistance",

"Superhuman reflexes"

]

}

]

You'd just have to access array items (in its parsed version) by starting with an array index, for example [0]["powers"][0].

**JSON.Stringify** → turn a JS object into JSON text and stores that JSON text in a string:

Var my\_object = {key1:”one”};

Var object\_as\_string = JSON.stringify(my\_object);

typeof(object\_as string) //”string”

**JSON.parse**→ turn a string of JSON text into a JavaScript object:

var object\_as\_string\_as\_object = JSON.parse(object\_as\_string);

typeof(object\_as\_string\_as\_object) // “object”

**[Other notes](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Objects/JSON" \l "other_notes)**

* JSON is purely a string with a specified data format — it contains only properties, no methods.
* JSON requires double quotes to be used around strings and property names. Single quotes are not valid other than surrounding the entire JSON string.
* Even a single misplaced comma or colon can cause a JSON file to go wrong, and not work. You should be careful to validate any data you are attempting to use (although computer-generated JSON is less likely to include errors, as long as the generator program is working correctly). You can validate JSON using an application like [JSONLint](https://jsonlint.com/).
* JSON can actually take the form of any data type that is valid for inclusion inside JSON, not just arrays or objects. So for example, a single string or number would be valid JSON.
* Unlike in JavaScript code in which object properties may be unquoted, in JSON only quoted strings may be used as properties.

Asynchronous programming is a technique that enables your program to start a potentially long-running task and still be able to be responsive to other events while that task runs, rather than having to wait until that task has finished. Once that task has finished, your program is presented with the result.

Synchronous programming – issue – long-running

The reason for this is that this JavaScript program is *single-threaded*. A thread is a sequence of instructions that a program follows. Because the program consists of a single thread, it can only do one thing at a time: so if it is waiting for our long-running synchronous call to return, it can't do anything else.

**[Event handlers](https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Asynchronous/Introducing" \l "event_handlers)**

The description we just saw of asynchronous functions might remind you of event handlers, and if it does, you'd be right. Event handlers are really a form of asynchronous programming: you provide a function (the event handler) that will be called, not right away, but whenever the event happens. If "the event" is "the asynchronous operation has completed", then that event could be used to notify the caller about the result of an asynchronous function call.

**Asynchronous Behavior of XMLHttpRequest**

1. **Non-blocking Execution**: When you make a request using XMLHttpRequest, the JavaScript engine does not pause your script while waiting for the server's response. Instead, it sends the request to the server and continues to execute the rest of the script.
2. **Event-Driven Responses**: The response to an XMLHttpRequest is handled using events. In your code, the **loadend** event is used. This event fires when the request completes (regardless of the success or failure of the operation), allowing a callback function to execute at that point. This callback function (**xhr.addEventListener("loadend", callback)**) is triggered asynchronously when the response is ready.
3. **Independent UI and Data Processing**: The user interface remains responsive because the data processing happens in the background. Users can continue interacting with the web page, click other buttons, or perform other tasks without interruption.

**Let vs Var**

Terminal :

-> nano xxx.js (create j file)

-> node xxx.js (run js file)

The difference between **var** and **let** in JavaScript mainly revolves around scoping and hoisting. Here’s a breakdown of these differences with examples:

**1. Scoping**

**var**: Variables declared with **var** are scoped to the nearest function block or globally if they are not inside any function. This means if **var** is used inside a function, it can be accessed anywhere within that function, even before the line where it is declared (due to hoisting, explained below).

**let**: Variables declared with **let** have block scope, meaning they can only be accessed within the block (**{}**) they are defined in. This includes control structures like loops and conditionals, as well as blocks defined by curly braces.

**Example of Scoping**

javascript

Copy code

function testVar() { if (true) { var x = 2; } console.log(x); // Outputs 2 because 'x' is function-scoped } function testLet() { if (true) { let y = 3; } console.log(y); // ReferenceError: y is not defined because 'y' is block-scoped } testVar(); testLet();

**2. Hoisting**

**var**: Variables declared with **var** are hoisted to the top of their scope. This means the declaration is moved to the top, but not the initialization. If you try to access a **var**-declared variable before its actual declaration line, it will return **undefined**.

**let**: Variables declared with **let** are also hoisted to the top of their block, but they are not initialized. Accessing a **let**-declared variable before its declaration results in a ReferenceError. This is often referred to as being in a "temporal dead zone" from the start of the block until the declaration is initialized.

**Example of Hoisting**

javascript

Copy code

console.log(a); // Outputs 'undefined' because of hoisting var a = 10; console.log(b); // ReferenceError: Cannot access 'b' before initialization let b = 20;

**Summary**

* **Scope**: **var** is function-scoped or globally-scoped, while **let** is block-scoped.
* **Hoisting**: Both **var** and **let** are hoisted, but **var** is initialized with **undefined**, making it accessible before its declaration line. In contrast, **let** remains uninitialized, causing a ReferenceError if accessed before its declaration.

These differences make **let** generally more preferable as it minimizes unexpected behaviors due to hoisting and makes code easier to understand due to block-level scoping.

The result of **console.log(obj1 === obj2);** being **false** in JavaScript stems from the way JavaScript compares objects. When you use the **===** operator with objects, JavaScript compares their references, not their contents. This type of comparison is known as "reference equality."

**Reference Equality**

* **Reference Equality** means that two object variables are considered equal if and only if they refer to the exact same object in memory.

Arrow function -> <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Functions/Arrow_functions>