25th August 2019

Global object

The global object provides variables and functions that are available anywhere. By default, those that are built into the language or the environment.

In a browser it is named `window`, for Node.js it is `global`, for other environments it may have another name.

Recently, `globalThis` was added to the language, as a standardized name for a global object, that should be supported across all environments. In some browsers, namely non-Chromium Edge, `globalThis` is not yet supported, but can be easily polyfilled.

We'll use `window` here, assuming that our environment is a browser. If your script may run in other environments, it's better to use `globalThis` instead.

All properties of the global object can be accessed directly:

```
alert("Hello");
// is the same as
window.alert("Hello");
```

In a browser, global functions and variables declared with `var` (not `let/const`!) become the property of the global object:

```
var gVar = 5;

alert(window.gVar); // 5 (became a property of the global object)
```

Please don't rely on that! This behavior exists for compatibility reasons. Modern scripts use JavaScript modules where such thing doesn't happen.

If we used `let` instead, such thing wouldn't happen:

```
let gLet = 5;

alert(window.gLet); // undefined (doesn't become a property of the global obj
```

If a value is so important that you'd like to make it available globally, write it directly as a property:

```
// make current user information global, to let all scripts access it
window.currentUser = {
```

```
  name: "John"
};

// somewhere else in code
alert(currentUser.name);  // John

// or, if we have a local variable with the name "currentUser"
// get it from window explicitly (safe!)
alert(window.currentUser.name); // John
```

That said, using global variables is generally discouraged. There should be as few global variables as possible. The code design where a function gets “input” variables and produces certain “outcome” is clearer, less prone to errors and easier to test than if it uses outer or global variables.

Using for polyfills

We use the global object to test for support of modern language features.

For instance, test if a built-in `Promise` object exists (it doesn’t in really old browsers):

```
if (!window.Promise) {
  alert("Your browser is really old!");
}
```

If there’s none (say, we’re in an old browser), we can create “polyfills”: add functions that are not supported by the environment, but exist in the modern standard.

```
if (!window.Promise) {
  window.Promise = ... // custom implementation of the modern language featur
}
```

Summary

- The global object holds variables that should be available everywhere.

  That includes JavaScript built-ins, such as `Array` and environment-specific values, such as `window.innerHeight` – the window height in the browser.

- The global object has a universal name `globalThis`.

  …But more often is referred by “old-school” environment-specific names, such as `window` (browser) and `global` (Node.js). As `globalThis` is a recent proposal, it’s not supported in non-Chromium Edge (but can be polyfilled).

- We should store values in the global object only if they’re truly global for our project. And keep their number at minimum.

- In-browser, unless we’re using modules, global functions and variables declared with `var` become a property of the global object.

- To make our code future-proof and easier to understand, we should access properties of the global object directly, as `window.x`.

Previous lesson Next lesson

Share

Tutorial map

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>`, for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)