JavaScript Typescript Graffiti

Blackat

Table of Contents

Chapter 1. JavaScript Graffiti

1.1. Variable scope

1.1.1. Global variable and how to create it

> A global variable is a variable that is visible in every scope.
>
> — Douglas Crockford, JavaScript: The Good Parts

> A global variable is a variable declared outside a function definition and it is property of the global scope [2: In a browser the global scope is represented by the `window` object, instead in a Node.js application by the object called `global` adn in Web Workers by `self`.].

There are three ways to create a global variable in JavaScript and remember we are talking about **global variable**, it will be useful for the next argument.

Three way to declare a global variable in JavaScript

```
window.s = 'felix';    // window global
s = 'felix';           // implied global
var s = 'felix';       // declared global
```

Explanation

- **window global:** the variable is directly set on the `window` object. Working in a browser the `window` object is root scope, there is nothing higher than that.
  - `console.log(window.s) // felix`
- **implied global:** when an identifier is used, the interpreter resolves it traversing up the scope chain [5: More on the scope chain and variable resolution in the David Shariff's post [identifier-resolution].]. If the identifier is not found in the local scope, the global one is involved. If not found a new global variable is created otherwise the old value is updated.
  - `console.log(s) // felix`
- **declared global:** use `var` reserved keyword to declare a variable. If a local variable and the global variable have the same identifier, the local variable will take the precedence (shadowing [6: In JavaScript *shadowing* is a behavior that allows a local variable to take the precedence over the outer or global variable having the same identifier, the inner variable over the outer.]).
  - `console.log(s) // felix`

What is the difference?

But how could I know which declaration method has been used? In general there is no difference, the variable becomes a property of the global object `window`. Going a bit more into details through the method `Object.getOwnPropertyDescriptor` [7: For method details please refer to MDN page.] we can see that there is a difference:

Window global

```
window.s = 'felix';
console.log(Object.getOwnPropertyDescriptor(window, 's'));
Object {value: "felix", writable: true, enumerable: true, configurable: true}
```

Implied global

```
s = 'felix';
console.log(Object.getOwnPropertyDescriptor(window, 's'));
Object {value: "felix", writable: true, enumerable: true, configurable: true}
```

Declared global

```
s = 'felix';
console.log(Object.getOwnPropertyDescriptor(window, 's'));
Object {value: "felix", writable: true, enumerable: true, configurable: false}
```

In the declared global case the `configurable` property is `false`!

configurable

true if and only if the type of this property descriptor may be changed and if the property may be deleted from the corresponding object.

So if I try to delete the property

Declared global

```
var s = 'felix';
delete window.s
false
```

and having a look at the MDN page of the `delete` operator we can read that

`delete` is only effective on an object's properties. It has no effect on variable or function names.

So better to review our examples

Declarations reviewed

```
window.s = 'felix';     // creates the property x on the global object
s = 'felix';            // creates the property x on the global object
var s = 'felix';        // creates the property y on the global object, and marks it as
non-configurable
```

Use strict?

Create global variables is in general a bad practice, even if some global variable should exists, but the implied global declaration is really a bad bad practice because we can accidentally create a global variable and have some strange effect later on in the application execution. Using the `strict` mode can protect against accidentally implied global declaration:

Use strict mode

```
// Use an IIFE to activate strict mode in the browser console
(function() {
   "use strict";
   c = 4}
   )();

VM735:1 Uncaught ReferenceError: c is not defined
```

If the variable has not been defined before, the VM will throw an exception. More on strict mode on the MDN page.

Chapter 2. TypeScript Graffiti

2.1. Interfaces

Type-checking is based on the shape that objects have.