Javascript Basic

written by Hong Le Nguyen 08.2024

Variables ---

A variable
is *a value* assigned to *an identifier*, so you can reference and use it later in the program.

In JavaScript, *objects and functions (name) are also variables*.

Declaring a Variable ---

creating a variable is called "declaring" a variable.
A variable must be declared before you can use it.

To declare variables using **let** or **const** or **var** keyword

The let keyword
is used to decalare *variables whose value might change* at different parts of the program.

The const keyword
is used to declare constants. Constants are *variables that cannot be reassigned* other values.

In particular, it does not mean the value cannot change - it means it cannot be reassigned. If the variable points to an object or an array the content of the object or the array can freely change.

The var keyword
was used in all JavaScript code from 1995 to 2015.

JavaScript variables can hold many types of data

A variable declared without a value will have the value *undefined*.

When to Use var, let, or const? --

1. Always declare variables
2. Always use const if the value should not be changed
3. Always use const if the type should not be changed (Arrays and Objects)
4. Only use let if you can't use const
5. Only use var if you MUST support old browsers.

Variable scope ---

Variable scope
is parts of the program where the variable is visible / accessible
The scope of a variable depends on where the variable is declared.

In JavaScript we have global-, block- and function- scope.

block scope --
A variable defined as **const** or **let** is only *visible inside the block where it is defined*.
A block is the part of the program in a pair of curly braces {}, like the ones we can find inside an if statement or a for loop.

function scope --
A variable defined as **const** or **let** *inside a function* (but outside of any block) is visible in that function
A variable defined as **var** inside a function is visible everywhere in that function

global scope ---
If a variable is defined *outside of a function or block*, it has a global scope, which means it's available in every part of a program.

```
// global scope
const url = "https://mydomain.com/api/data"

function getData() {

  // function scope
 const data = [];

    if (true) {
         // block scope
         const res = fetch(…)
    }

  return value ;
}
```

JavaScript Data Types ---

primitive types:
string, number, boolean, symbol, null, undefined.

objects are king –

*All JavaScript **values**, except primitives, are objects.*

Objects are objects
Functions are objects

Arrays are objects
Maps are objects
Sets are objects
Maths are objects
Dates are objects
…

Objects are always passed by reference.

The typeof Operator --

The typeof operator returns the type of a variable or an expression:

```
const car = {
   brand: 'Ford',
   color: 'blue',
}
```

```
function getData(param1, param2){
   console.log(param1, param2)
}

console.log(typeof "hello")       // string
console.log(typeof (3+4))         // number
console.log(typeof (car))         // object
console.log(typeof (getData))   // function
console.log(typeof (()=>{}))      // function
```

Object –-

Objects are containers for properties and methods.
Properties are named values.
Properties can be primitive values, functions, or even other objects.
Methods are functions stored as properties.

How to create an object ? --

the object literal syntax

```
const car1 = {
   brand: 'Ford',
   color: 'blue',
}
```

You can also use the *new Object* syntax:

```
const car2 = new Object()
car2.brand = 'BMW'
car2.color = 'red'
```

using the new keyword before a function with a capital letter

```
function Car(brand, color) {
   this.brand = brand
   this. color = color
}
const car3 = new Car('VW', 'white')
```

prototype property –

“Each object has a special prototype property as a pointer to the object from which it was created.”

We can get and show prototype property of objects

```
console.log(Reflect.getPrototypeOf(car1))  // [object Object]
console.log(Reflect.getPrototypeOf(car2))  // [object Object]
console.log(Reflect.getPrototypeOf(car3))  // [object Object]
```

We see that the objects car1, car2, car3 are created from *a common object of type Object*.
We can add a method to the prototype, then the method will be accessible to all objects car1, car2, car3.

```
Reflect.getPrototypeOf(car1).display = function () {
   console.log(`I am a ${this.brand}-${this.color} car`);
};

car1.display();  // I am a Ford-blue car
car2.display();  // I am a BMW-red car
car3.display();  // I am a VW-white car
```

Object Methods –

method
functions can be assigned to a property, and in this case they are called methods.

this
inside a method defined using a *regular function* (function() {} syntax) we have access to the object instance by referencing **this**. We don't have access to this if we use an *arrow function*.

This is the reason why regular functions are often used as object methods.

Functions ---

JavaScript function

is a block of code designed to perform a particular task.
A function is executed when "something" invokes it (calls it).

The object nature of functions

"Functions are also objects, they can have properties and methods, can be returned from functions, can be passed as arguments."

function declaration ---

as Regular function

```
function getData (parameter1, parameter2) {
   // code to be executed
   return some-value;
}
```

or

```
const getData = function (parameter1, parameter2) {
   //...
}
```

Function name (e.g. getData) is a variable of type "function".
Function *parameters* are listed inside the parentheses () in the function definition.
Function *arguments* are the values received by the function when it is invoked.
Inside the function, *the parameters behave as local variables (function scope).*
By default a function returns undefined, unless you add a return keyword with a value.

Arrow functions –
are anonymous functions. We must assign them to a variable.

```
const getData = (parameter1, parameter2) => {
   console.log(parameter1, parameter2)
}
```

function invocation –

getData(value1, value2)

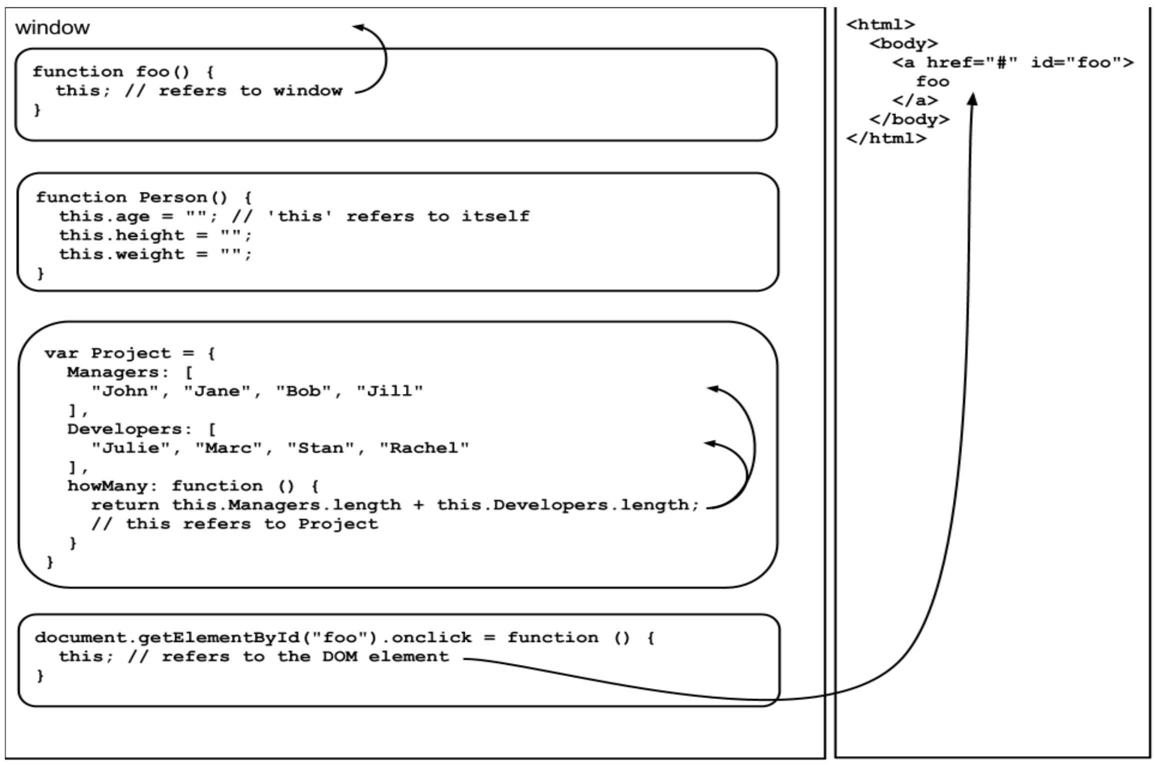
The code inside the function will execute when "something" invokes (calls) the function:
When an event occurs (when a user clicks a button)
When it is invoked (called) from JavaScript code
Automatically (self invoked)

execution context of a function and this –

*"Value of **this** variable in function is the current execution context of a function."*

Value of **this** variable in different contexts

In a function, this refers to the global object (in strict mode, this is undefined).
In an *(object constructor-) function*, this refers to the object.
In an object method, this refers to the object.
In an event handler, this refers to the element that received the event.



```
function foo() {
  this; // refers to window
}
```

```
function Person() {
  this.age = ""; // 'this' refers to itself
  this.height = "";
  this.weight = "";
}
```

```
var Project = {
  Managers: [
    "John", "Jane", "Bob", "Jill"
  ],
  Developers: [
    "Julie", "Marc", "Stan", "Rachel"
  ],
  howMany: function () {
    return this.Managers.length + this.Developers.length;
    // this refers to Project
  }
}
```

```
document.getElementById("foo").onclick = function () {
  this; // refers to the DOM element
}
```

```
<html>
  <body>
    <a href="#" id="foo">
      foo
    </a>
  </body>
</html>
```

An exsample

```
<!DOCTYPE html>
<html>
  <body>
  <button type="button" id="myBtn">Click Me!</button>
  <script>

    function getContext(){
      console.log(this);
    }
```

```
        function Person(name, age) {
            this.name = name;
            this.age = age;
        }

        getContext();  //  call function, out put :  Window object

        const jakob = new Person("Jakob", 7);
        jakob.getContext = getContext;

        jakob.getContext(); // call function, out put :  Person object

        document.getElementById("myBtn").addEventListener("click", getContext);
        // by click button=> call function, out put : <button type="button" id="myBtn">Click Me!</button>

    </script>
    </body>
</html>
```

Nested functions –

A function can contain other arrow /regular functions.

Nested functions and Closures

“Local variables within a function will remain available even after that function has finished executing.”

“A nested function is a closure that can have free variables together with an environment that binds those variables (that "closes" the expression). In other words, the inner function contains the scope of the outer function.”

```
function greet(name) {
    let hello = "Hello " + name;
    const doPrintf = function () { // a nested function is a closure
        console.log(hello);
    }
    return doPrintf;
}

// call greet(name)
const doPrintf = greet("Kitty");
// doPrintf() can access hello variable
doPrintf(); => "Hello Kitty"
```

Note: you can run JS code online
https://reqbin.com/code/javascript

JavaScript Modules ---

JavaScript modules allow you to break up your code into separate files.
This makes it easier to maintain a code-base.

Modules are imported from external files with the import statement.

Export

Modules with functions or variables can be stored in any external file.
There are two types of exports: Named Exports and Default Exports.

Named Exports

way 1 : In-line individually

```
// person.js
export const name = "Jesse";
export const age = 40;
```

way 2: all at once at the bottom.

```
// person.js
const name = "Jesse";
const age = 40;

export {name, age};
```

Default Exports

You can *only have one default export in a file*.

```
// message.js
const message = () => {
  const name = "Jesse";
  const age = 40;
  return name + ' is ' + age + 'years old.';
};

export default message;
```

Import

You can import modules into a file in two ways, based on if they are named exports or default exports.
Named exports are constructed using curly braces. Default exports are not.

Import from named exports

```
import { name, age } from "./person.js";
```

Import from default exports

```
import message from "./message.js";
```

-----some data types examples ------

Strings are written inside double or single quotes.
Numbers are written without quotes.
Example

```
const pi = 3.14;
let person = "John Doe";
```

Constant *Arrays*
You can change the elements of a constant array
Example

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];
// You can change an element:
cars[0] = "Toyota";
// You can add an element:
cars.push("Audi");
```

Constant *Objects*
You can change the properties of a constant object:
Example

```
// You can create a const object:
const car = {
  type:"Fiat",
  model:"500",
  color:"white"
};
// You can change a property:
car.color = "red";
// You can add a property:
car.owner = "Johnson";
```