Lezione02:03-Objects and Functions

Thursday, March 4, 2021 9:06 AM

<WA1/>
<AW1/>
2021

JavaScript: Objects and Functions

"The" language of the Web

Fulvio Corno
Luigi De Russis
Enrico Masala

Outline

- Objects
- Functions
  - Closures
- Dates

JavaScript: The Definitive Guide, 7th Edition
Chapter 5. Objects

Mozilla Developer Network
- Learn web development JavaScript » Dynamic client-side scripting » Introducing JavaScript objects
- Web technology for developers » JavaScript » JavaScript reference » Standard built-in objects » Object
- Web technology for developers » JavaScript » JavaScript reference » Expressions and operators » in operator

JavaScript – The language of the Web

OBJECTS

Big Warnings *(a.k.a., forget Java objects)*

- In JavaScript, Objects may exist without Classes
  - Usually, Objects are created directly, without deriving them from a Class definition
- In JavaScript, Objects are dynamic
  - You may add, delete, redefine a *property* at any time
  - You may add, delete, redefine a *method* at any time
- In JavaScript, there are no access control methods
  - Every property and every method is always public (private/protected don't exist)
- There is no real difference between properties and methods (because of how JS functions work)

Java->Everything is a class-> Objects are only instances of a Class
JavaScript->Everything is an objects (Arrays and Strings are just 2 types of objects! JS Classes are just sintactic sugar on functions!) -> You can have an Object without having defined any Classes
Object can be defined everywhere!

-> Methods are just properties that happens to be functions (they're not special)
(we could also modify a single object of type string changing its method! Very dangerous!)
No access control (private, public..) -> behaving well -> don't touch my objects, I won't touch yours!
Language doesn't help.

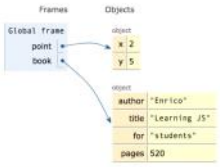
Object

- An object is an unordered collection of properties
  - Each property has a **name** (key), and a **value**
- You store and retrieve *property values*, through the *property names*
- Object creation and initialization:

```
let point = { x: 2, y: 5 };

let book = {
  author : "Enrico",
  title : "Learning JS",
  for: "students",
  pages: 520,
};
```

Object literals syntax:
{"name": value,
"name": value, }
or:
{name: value,
name: value, }



Creation and initialization of properties together! (No empty properties, I create them when I need them)

(Using Prototypes, objects derives properties from them)

Final "," is ignored, useful for inserting line easily

JSON -> JS sintax notation, always "", no trailing ,

Object Properties

Property names are ...

- Identified as a string
- Must be unique in each object
- Created at object initialization
- Added after object creation
  - With assignment
- Deleted after object creation
  - With `delete` operator

Property values are ...

- Reference to any JS value
- Stored inside the object
- May be primitive types
- May be arrays, other objects, ...
  - Beware: the object stores the reference, the value is *outside*
- May also be functions (*methods*)

Everything is a pointer!

Operator-> no ()
Delete point.x

Accessing properties

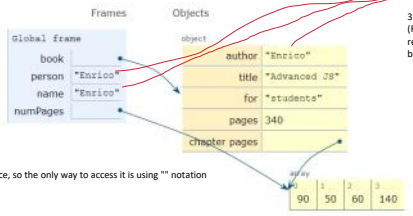
- Dot (`.`) or square brackets `[ ]` notation

The . dot notation and omitting the quotes are allowed **when the property name is a valid identifier, only**.
book.title or book['title']
book['my title'] and not ~~book.my title~~

```
let book = {
  author : "Enrico",
  title : "Learning JS",
  for: "students",
  pages: 340,
  "chapter pages": [90,50,60,140]
};

let person = book.author;
let name = book["author"];
let numPages =
    book["chapter pages"];
book.title = "Advanced JS";
book["pages"] = 340;
```

It contains a space, so the only way to access it is using "" notation



Enrico"

3 different references to the same (immutable) object
(For strings it is not important to distinguish between reference and real value because in both cases they can't be modified!)

Object.property_name OR
Object["property_name"] (p="chapter pages"; Object[p])

If I modify the array (book["chapter pages"][0]=85 also numPages will be affected (it's the same array))

= -> Copy by reference!

We well use object (more general) but if we want we can also explore the map type in the standard library

Objects as associative arrays

Map 1 string (key) into a value

- The `[ ]` syntax looks like array access, but the index is *a string*
  - Generally known as *associative arrays*
- Setting a non-existing property creates it:
  - `person["telephone"] = "0110901234";`
  - `person.telephone = "0110901234";`
- Deleting properties
  - `delete person.telephone;`
  - `delete person["telephone"];`

Computed property names

Create object where the name of a property is inside a variable

C="ita" -> we want to create an object a with property "ita" and value of that property=3
a={c:3} -> create an object a with property "c" and value of that property=3!! Not what we want!
a={[c]:3} OR a[c]=3 -> Exactly what we want! Interprets c as a variable instead of as a string

- Flexibility in creating object properties
  - {[prop]:value} -> creates an object with property name equal to *the value of the variable prop*
  - [] can contain more complex expressions: e.g., i-th line of an object with multiple "address" properties (address1, address2, ...): person["address"+i]
    - **Using expressions is not recommended...**
- Beware of quotes:
  - book["title"] -> property called title
    - Equivalent to book.title
  - book[title] -> property called with the value of variable title (if exists)
    - If title=="author", then equivalent to book["author"]
    - No equivalent in dot-notation

!

Not very common, but useful!

Property access errors

a.z=3
K=(a.w+3) -> tries to access a property that doesn't exist! No runtime error! K=undefined (valid value!) -> always check if the result is defined or not!

- If a property is not defined, the (attempted) access returns undefined
- If unsure, must check before accessing

```
let surname = undefined;
if (book) {
    if (book.author) {
        surname = book.author.surname;
    }
}
```

book and book.author are automatically converted to Booleans when inside an if or in a logical operation (&&, ||, !) because they can only operate on a Boolean. Then, I want to assign the surname variable only in the case when both of them are defined. Finally, surname will really contain the book.author.surname value, which could be defined or not, BUT THE IMPORTANT PART IS THAT SURNAME WILL THEN ACTUALLY REPRESENT THE VALUE OF BOOK.AUTHOR.SURNAME AND IT WILL NOT BE THE RESULT OF AN IMPOSSIBLE OPERATION (ACCESS TO AN OBJECT/PROPERTY WHICH IS FALSY)!

Equivalent!

```
surname = book && book.author && book.author.surname;
```


Iterating over properties

Objects!

Useful for accessing object with unknown properties!

- for .. in iterates over the properties

```
for( let a in {x: 0, y:3}) {
    console.log(a) ;
}
```

```
x
y
```

For..of -> Arrays

```
let book = {
  author : "Enrico",
  pages: 340,
  chapterPages: [90,50,60,140],
};

for (const prop in book)
  console.log(`${prop} = ${book[prop]}`);
```

N.B. IT IS CONSTANT BECAUSE PROP LIVES IN THE SCOPE OF THE FOR!
EACH ITERATION A NEW VARIABLE IS DEFINED! IF IT IS NOT CHANGED INSIDE THE SINGLE ITERATION IT CAN BE DEFINED AS CONST!!!

Template literal -> Inject values of expressions inside string
Without it: (prop + " = " + book[prop])

```
author = Enrico
pages = 340
chapterPages = 90,50,60,140
```


Iterating over properties

- All the (enumerable) properties names (keys) of an object can be accessed as an array, with:
  - let keys = Object.keys(my_object) ;
- All pairs [key, value] are returned as an array with:
  - let keys_values = Object.entries(my_object)

```
[ 'author', 'pages' ]
```

```
[ [ 'author', 'Enrico' ], [ 'pages', 340 ] ]
```

To iterate over properties -> for..in OR for + extract all keys into an array
To iterate over entries -> for + extract all entries into an array
To iterate over values -> for + extract all values into an array

- let values = Object.values(my_object)

Copying objects

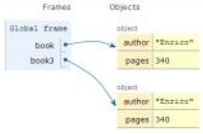
```
let book = {
  author : "Enrico",
  pages: 340,
};

let book2 = book;
```

Reference copy

```
let book = {
  author : "Enrico",
  pages: 340,
};

let book3 =
   Object.assign({}, book);
```



Real copy

Assign properties of the second(source) object to the first(target) objects
(Merge and overwrite properties)
Target has properties a,b
Source has properties b,c

Target will have a,b(FROM SOURCE),c

If target has a subset of source properties (null subset or even same properties) we effectively create a copy of the source

Object.assign

Modifies the target object AND returns the updated target object

- `let new_object = Object.assign(target, source);`
- Assigns all the properties from the source object to the target one
- The target may be an existing object
- The target may be a new object: {}
- Returns the target object (after modification)

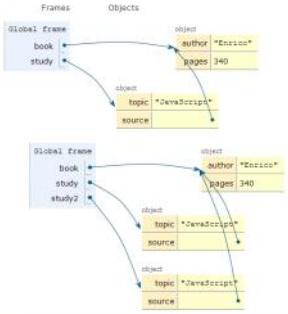
Beware! Shallow copy, only

Real copy of an object containing references to arrays or references to objects will only copy the references, not the original array/object!

```
let book = {
  author : "Enrico",
  pages: 340,
};

let study = {
  topic: "JavaScript",
  source: book,
};
```

```
let study2 = Object.assign({},
study);
```

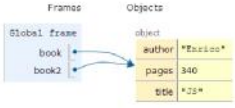


Merge properties (on existing object)

- `Object.assign(target, source, default values, ..);`

```
let book = {
  author : "Enrico",
  pages: 340,
};

let book2 = Object.assign(
 book, {title: "JS"}
);
```

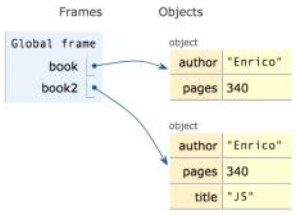


Merge properties (on new object)

- `Object.assign(target, source, default values, ..);`

```
let book = {
  author : "Enrico",
  pages: 340,
};

let book2 = Object.assign(
 {}, book, {title: "JS"}
);
```



Copying with spread operator (ES9 – ES2018)

Very convenient!!!

Spreads properties of an object

```
let book = {
  author : "Enrico",
  pages: 340,
};

let book2 = {...book, title: "JS"};

console.log(book2);
```

```
{ author: 'Enrico', pages: 340, title: 'JS' }
```

```
const {a,b,...others} =
   {a:1, b:2, c:3, d:4};

console.log(a);
console.log(b);
console.log(others);
```

```
1
2
{ c: 3, d: 4 }
```

Let copyofobj={...obj}

Checking if properties exist

Different than for..in!!

- Operator `in`
  - Returns true if property is in the object. ~~Do not use with Array~~

```
let book = {
  author : "Enrico",
  pages: 340,
};

console.log('author' in book);
delete book.author;
console.log('author' in book);
```

```
true
false
```

```
const v=['a','b','c'];

console.log('b' in v);


console.log('PI' in Math);
```

```
false
true
```


Object creation (equivalent methods)

- By object literal: `const point = {x:2, y:5} ;`
- By object literal (empty object): `const point = {} ;`

Preferred

- By constructor: `const point = new Object() ;`
- By object static method create:
  `const point = Object.create({x:2,y:5}) ;`

Equivalent, no reason to type more

- Using a *constructor function*

Java-like

JavaScript – The language of the Web

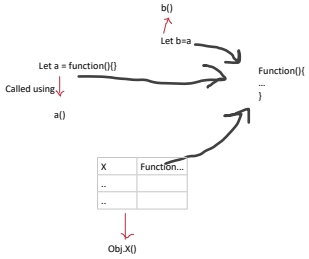
FUNCTIONS

Functions

- One of the most important elements in JavaScript
- Delimits a block of code with a private scope
- Can accept parameters and returns one value
  - Can also be an object

"one"-> array/ object

- Functions themselves **are objects** in JavaScript
  - They can be assigned to a variable
  - Can be passed as an argument
  - Used as a return value



Declaring functions: 3 ways

1) Classic

Rarely used

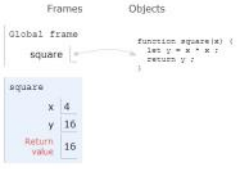
```
function do(params) {
  /* do something */
}
```


Classic functions

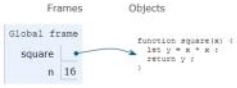
```
function square(x) {
  let y = x * x ;
  return y ;
}

let n = square(4) ;
```

During execution



After execution



Parameters

Params always passed by reference

- Comma-separated list of parameter names
  - May assign a default value, e.g., `function(a, b=1) {}`
- Parameters are passed by-value
  - Copies of the reference to the object
- Parameters that are not passed in the function call get the value 'undefined'
- Check missing/optional parameters with:
  - `if(p===undefined) p = default_value ;`
  - `p = p || default_value ;`

Function(a,b,c){}
Function(3) -> a=3, b=c=undefined

P=default_value only if p is falsy

Variable number of parameters

- Syntax for functions with variable number of parameters, using the . . . operator (called "rest")

```
function fun (par1, par2, ...arr) { }
```

- The "rest" parameter must be the last, and will deposit all extra arguments into an array

```
function sumAll(initVal, ...arr) {
  let sum = initVal;
  for (let a of arr) sum += a;
  return sum;
}
sumAll(0, 2, 4, 5); // 11
```


Declaring functions: 3 ways

1) Classic

```
function do(params) {
  /* do something */
}
```

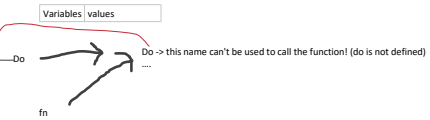
"Do" variable is created implicitly!

const fn=do
OR

2a) Function expression

```
const fn = function(params) {
  /* do something */
}
```

Can be written anywhere! Not global scope always!
Function expression returns an object of type function and we can use fn to reference TO THE FUNCTION
We use the variable that contains the value to call the function, the value does not have its name or even if it has its own name, that can't be used to call the function



2b) Named function expression

```
const fn = function do(params) {
  /* do something */
}
```

Useful only for debugging!
I can't call var=do(params) -> not defined!!

Array of functions:
Const operations = [function(x) {return x+1}, function(x){return x-1}]
a=operations[0](5) //a=6
Const operations2=[increase, decrease] // defined before

```
Function increase(x){x=x+1;}

Let a=5;
Increase(a);
//a=5!!! -> when I enter the function I have a COPY of the stack, with A COPY of the variable that POINTS TO THE SAME VALUE as the other variable.
Inside the function a=6, but as soon as the function returns, a=5 again.
-----
IF WE WANT TO MODIFY THE PARAMETERS PASSED BY REFERENCE WE CAN USE C-LIKE SYNTAX (USE RETURN VALUE TO RETURN UPDATED VERSION OF THE PARAMETERS) OR WE CAN DIRECTLY MODIFY THE PARAMETER PASSED BY REFERENCE USING IN-PLACE METHODS!
function actOnObject(x){x.n=5;}
let a= {n:0};
actOnObject(a); //a is modified!
-----------------------------
function cLikeReturn(x){
  let y={n:x.n+5};
  return y;
}
let a= {n:0};
let b=a;
b=cLikeReturn(a); //b is modified.

WE CAN'T ASSIGN (=) OR CREATE NEW ARRAY INSIDE THE FUNCTION WITHOUT RETURN THE NEW COPY OTHERWAYS THE PARAMETER REAL VALUE WILL NOT BE MODIFIED!
function newObjectWrong(x){
  x={t:3};
}
let a= {n:0};
let b=a;
newObjectWrong(a); This modifies x while it is in the function, but at the end a is unaltered!
---
Function appendWrong(a){
  a=[ ...a, 1 ]; //NEW ARRAY!!!
}
Let c=[1,7,5];
AppendWrong(c); //c is not modified!!! = operator CHANGES the reference, not the real copy!

Function append(a){
  a.push(1); //IN-PLACE!
}
Let c=[1,7,5];
Append(c); //c is modified!
```

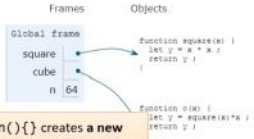
Function expression: indistinguishable

90% of JS are callbacks -> call function with functions as parameter
(lambda functions/expression derives immediately from the fact that functions are objects)

```
function square(x) {
  let y = x * x ;
  return y ;
}

let cube = function c(x) {
  let y = square(x)*x ;
  return y ;
}

let n = cube(4) ;
```



The *expression* `function(){}` creates **a new object of type 'function'** and returns the result.

Any variable may "refer" to the function and call it. You can also store that reference into an array, an object property, pass it as a parameter to a function, redefine it, ...

method

callback

Declaring functions: 3 ways

1) Classic

```
function do(params) {
  /* do something */
}
```

2a) Function expression

```
const fn = function(params) {
  /* do something */
}
```

3) Arrow function

```
const fn = (params) => {
  /* do something */
```

2b) Named function expression

```
const fn = function do(params) {
  /* do something */
```

This keyword has special behaviour in different functions

3) Arrow function

```
const fn = (params) => {
  /* do something */
}
```

No function keyword

2b) Named function expression

```
const fn = function do(params) {
  /* do something */
}
```

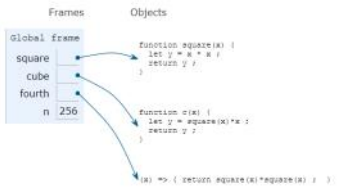

Arrow Function: just a shortcut

```
function square(x) {
  let y = x * x ;
  return y ;
}

let cube = function c(x) {
  let y = square(x)*x ;
  return y ;
}

let fourth = (x) => { return
square(x)*square(x) ;  }

let n = fourth(4) ;
```



Parameters in arrow functions

```
const fun = () => {  /* do something */ }          // no params

const fun = param => {  /* do something */ }       // 1 param

const fun = (param) => {  /* do something */ }     // 1 param

const fun = (par1, par2) => {  /* smtg */ } // 2 params

const fun = (par1 = 1, par2 = 'abc') => {  /* smtg */ }   // default values
```

2 params +

Return value

- Default: `undefined`
- Use `return` to return a value
- Only one value can be returned
- However, objects (or arrays) can be returned

```
const fun = () => {  return ['hello', 5] ; }
const [ str, num ] = fun() ;
console.log(str) ;
```

- Arrow functions have implicit return if there is only one value

```
let fourth = (x) => { return square(x)*square(x) ;  }
let fourth = x => square(x)*square(x) ;
```

(looks like a lamba->1 param, 1 result)

Nested functions

- Function can be nested, i.e., defined within another function

```
function hypotenuse(a, b) {
    const square = x => x*x ;
    return Math.sqrt(square(a) + square(b));
}
```

(lighter)

=> Preferred in nested functions

x*(x+a) -> Inner scopes can access variable a!!! (CLOSURE PROPERTIES)

```
function hypotenuse(a, b) {
    function square(x) { return x*x; }
    return Math.sqrt(square(a) + square(b));
}
```

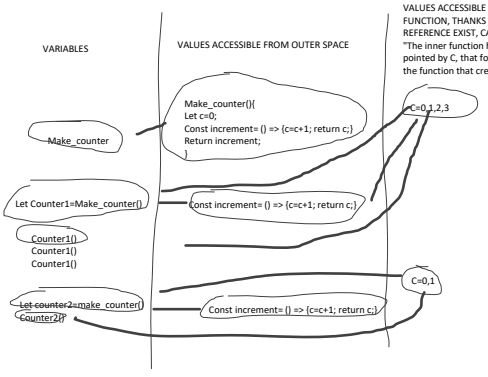
Outer scopes can't access x!

Outer scopes can't access square!

- The inner function is *scoped within* the external function and cannot be called outside
- The inner function might *access variables declared* in the *outside* function

Strange!!

VALUES ACCESSIBLE ONLY USING THE RETURNED NESTED FUNCTION, THANKS TO CLOSURE FEATURE (NO DIRECT REFERENCE EXIST, CAN'T BE ACCESSED DIRECTLY)
"The inner function has a closure over the value(object) pointed by C, that for this reason will survive even after the function that created it has ended"



Closure: definition (somewhat cryptic)

Key concept to create objects and classes in JS!

```
Function count(){
Let c = 0;
c = c +1 ;
Return c}
Count() //1
Count() //1
No!
```

Closure: definition (somewhat cryptic)

Key concept to create objects and classes in JS!

A closure is a name given to a feature in the language by which a nested function executed after the execution of the outer function can still access outer function's scope.

(increment)

(c)

Really: one of the most important concepts in JS

https://vykchandra.medium.com/learn-javascript-closures-through-the-laws-of-karma-49d32d35b3f7

https://medium.com/@vykchandra/learn-javascript-closures-through-the-laws-of-karma-49d32d35b3f7

Function count(){
Let c = 0;
c = c +1 ;
Return c}
Count() //1
Count() //1
No!	

Function make_counter(){
Let c=0;
Let Increment = () => { c= c+1; return c;};
Return increment; //This let me call the function from outside AND let me access c even after make_counter have ended because c is not deleted until all of its references are deleted
}
 	
Let counter1 = make_counter1();
 	
Console.log(counter1());
Console.log(counter1());
Console.log(counter1());
//1 2 3
Let counter2 = make_counter2();
 	
Console.log(counter2());
Console.log(counter2());
//1 2
	Let somebody outside the function count call a function inside it to increment a var inside it and return it outside
	Counter is () => {c=c+1; return c;}
	C is in the scope of make_counter, is recreated every time we call make_counter()
	Outside make_counter we can access the function increment, using the variable "counter", AND INCREMENT CAN ACCESS THE VARIABLE INSIDE make_counter()!
	THE VARIABLE C IS DESTROYED AT THE END OF THE FUNCTION, BUT
	THE VALUE(OBJECT) POINTED BY C IS NOT DESTROYED AT THE END OF THE FUNCTION! IT IS REALLY DESTROYED (by the garbage collector) ONLY WHEN NO MORE REFERENCE TO THAT OBJECT EXISTS.
	
	I can't access C from outside without using the increment function!!
	Even if I want to delete C I need to delete increment (counter=undefined) -> removes references to value C -> garbage collector will delete both the increment function value and the C value
	
	Counter2 will create ANOTHER DIFFERENT copy of the increment function value even if code is the same, because internally they are different (they refers to different C)
	Different from parameters because they are destroyed at the end!
	INCREMENT HAS CLOSURE OVER C VARIABLE

Closures

- JS uses *lexical scoping*
  - Each new functions defines a *scope* for the variables declared inside
  - Nested functions may access the scope of *all enclosing* functions
- Every function object remembers the scope where it is defined, even after the external function is no longer active → Closure

```
"use strict" ;

function greeter(name) {
    const myname = name ;

    const hello = function () {
        return "Hello " + myname ;
    }

    return hello ;
}

const helloTom = greeter("Tom") ;
const helloJerry = greeter("Jerry") ;

console.log(helloTom()) ;
console.log(helloJerry()) ;
```

Warning: not `return hello() ;`

Closures

- `hello` accesses the variable `myname`, defined in the outer scope
- The function is returned (as `helloTom` or `helloJerry`)
- Each of the functions "remembers" the reference to `myname`, when it was defined
- The variable `myname` goes out of scope, but is not destroyed
  - Still accessible (referred) by the `hello` functions.

```
"use strict" ;

function greeter(name) {
    const myname = name ;

    const hello = function () {
        return "Hello " + myname ;
    }

    return hello ;
}

const helloTom = greeter("Tom") ;
const helloJerry = greeter("Jerry") ;

console.log(helloTom()) ;
console.log(helloJerry()) ;
```

greeter scope

hello scope

Using closures to emulate objects

```
"use strict" ;

function counter() {
    let value = 0 ;

    const getNext = () => {
        value++;
        return value;
    }

    return getNext ;
}
```

```
const count1 = counter() ;
console.log(count1()) ;
console.log(count1()) ;
console.log(count1()) ;

const count2 = counter() ;
console.log(count2()) ;
console.log(count2()) ;
console.log(count2()) ;
```

```
1
2
3
1
2
3
```


Using closures to emulate objects (with methods)

```
"use strict";

function counter() {
    let n = 0;

    // return an object,
    // containing two function valued
    // properties
    return {
        count: function() {
            return n++; },
        reset: function() { n = 0; }
    };
}
```

Return function in a variable

```
let c = counter(), d = counter();
        // Create two counters

c.count()
        // => 0

d.count()
        // => 0: they count independently

c.reset()
        // reset() and count() methods

c.count()
        // => 0: because we reset c

d.count()
        // => 1: d was not reset
```

Create object (Counter) with the functions needed to modify its state variables (n) (=class)

Basic mechanism, a little convoluted, but you could do oop using just this (without classes)!

This pattern (function that creates an object and returns an object containing the functional properties to modify it) is so

```
function
outer() {
        var a = 10;
        var sum;

        // Nested function definition
        function inner() {
        var b = 20;

        // Nested function can access outer function's variables and scope
        sum = a + b;
        }

        inner();
        alert(sum); //sum=30
        }
```

```
        return n++; },
    reset: function() { n = 0; }
  };
}
```

```
c.count()
        // => 0: because we reset c

d.count()
        // => 1: d was not reset
```

This pattern (function that creates an object and returns an object containing the functional properties to modify it) is so frequent that it has its own syntax:
Counter is a constructor function, which can be called using the keyword new
The returned object is called this

Immediately Invoked Function Expressions (IIFE)

- Functions may protect the *scope* of variables and inner functions
- May declare a function
  - With internal variables
  - With inner functions
  - Call it only once, and discard everything

```
( function() {
    let a = 3 ;
    console.log(a) ;
} ) () ;
```

```
let num = ( function() {
    let a = 3 ;
    return a ;
} ) () ;
```

Variation of this pattern:
If we only need to call make_counter once (Singleton)(can be useful for init functions!), we could define a function, execute it immediately and then be remove it immediately after! (Less memory footprint, convenient!). (In this way we avoid to do function=null and wait for the garbage collector to remove it, because the system will know that the function must be removed immediately!

This is the basic of information hiding in js! Modules will have functions that contains variables, data structures, other functions, but then they will return only a subset of those!
This is also the mechanism used to have different files contributing to the same program!

If we don't need a return value we can also use

```
!function() {
        alert("Hello from IIFE!");
        }();
```

! Can be substituted by "+", "-", "~" , "void" or basically any unary operator can be used. But they won't work with arrow functions.

;(function shorten4() {...})(); can be useful to avoid problems if previous line didn't end with ;

https://flaviocopes.com/javascript-iife/
https://vvkchandra.medium.com/essential-javascript-mastering-immediately-invoked-function-expressions-67791338ddc6

https://flaviocopes.com/javascript-iife/
https://medium.com/@vvkchandra/essential-javascript-mastering-immediately-invoked-function-expressions-67791338ddc6

```
// There is also this variation with smal variations
        (function() {
        alert("I am an IIFE!");
        }());
```


Using IIFE to emulate objects (with methods)

```
"use strict";

const c = (
    function () {
        let n = 0;

        return {
            count: function () {
                return n++; },
            reset: function () {
                n = 0; }
        };
    })();
```

```
console.log(c.count());
console.log(c.count());
c.reset();
console.log(c.count());
console.log(c.count());
```

I can't have 2 counters! Singleton pattern!

```
0
1
0
1
```

```
(function IIFE_initGame() {
        // Private variables that no one has access to outside this IIFE
        var lives;
        var weapons;

        init();

        // Private function that no one has access to outside this IIFE
        function init() {
        lives = 5;
        weapons = 10;
        }
        }());
```

Next time whenever you are creating a bunch of variables and functions in global scope that no one uses outside your code, just wrap all of that in an IIFE and get a lot of good JavaScript karma for doing that. Your code will continue to work, but now you are not polluting global scope. Also you are shielding your code from someone who may change your globals accidentally, or sometimes intentionally!

Construction functions

- Define the object type (Convention)
  - Use a capital initial letter
  - Set the properties with the keyword this
- Create an instance of the object with new

Function designed (in our mind) for creating and returning a new object.

```
function Car(model, year){
 this.model= model;
 this.year= year;
}

let mycar = new Car("fiat", 2005);
```

Each time we call new we create a new object!

```
function newcar(model, year){
 return {model: model, year: year};
}

let mycar = newcar("fiat", 2005);
```

This is semantically equivalent, BUT it can't be called using the New keyword. In this case I create my object, while using new the new object is created for my by the language. (small difference)

Here the constructor
Creates an empty Object called"this"

Then it is populated by Properties and Functional Properties ("methods")

Then it is returned to the caller

```
function Car(make, model, year) {
    this.make = make;
    this.model = model;
    this.year = year;
    this.isNew = ()=>(year>2000);
}
```

N.B. return (this) is implicit!

Call function using new keyword

```
let mycar = new Car('Eagle',
'Talon TSi', 1993);
```

"this" (in this case) is just a convenient way to refer to the object that we would have created and populated anyway usign a normal function

(We can divide the program across different modules, organize it importing different files, but the important part is that the interpreter will see them all as a single stream)
(Property of the runtime environment more than of the language itself)

Courses_and_scores.js

JavaScript: The Definitive Guide, 7th Edition
Chapter 9.4 Dates and Times

Mozilla Developer Network
Web technology for developers » JavaScript »
JavaScript reference »
Standard built-in objects » Date

Day.js
https://day.js.org/en/

SQLITE:
You can represent dates as https://sqlite.org/datatype3.html
- STRINGS: "YYYY-MM-DD HH:MM:SS.SSS"
- INTEGER UNIX TIME Number of seconds since Epoch
- (REAL number of days since 4717 BC?!?)

You can use functions on dates as https://sqlite.org/lang_datefunc.html
To format, extract, convert, computation between days,... (non standard SQL syntax!)

JavaScript – The language of the Web

DATES

Date object

- Store a time instant with *millisecond* precision, counted from Jan 1, 1970 UTC (Unix Epoch)
- Careful with time zones
  - Most methods work in local time (not UTC) the computer is set to

```
let now = new Date();
```

```
let newYearMorning = new Date(
2021, // Year 2021
0, // January (from 0)
1, // 1st
18, 15, 10, 743);
// 18:15:10.743, local time
```

ISO format correct

```
UTC vs Local time zone are confusing.
> new Date('2020-03-18')
2020-03-18T00:00:00.000Z
> new Date('18 March 2020')
2020-03-17T23:00:00.000Z
```

Long syntax CONVERTS the date to local time because no hour was specified

Formatting is locale-dependent

Comparisons are difficult (no way to specify which fields you want, must set them to zero explicitly)

If I want to compare a Date to Now I need to put to zero all the fields of the date that I don't want to compare (hours, minutes, seconds, milliseconds)! Very tedious and error prone!

JS standard library has Date Object Type!
Works using local time of the server running the code! Could be a problem for Web App! (You don't know where the client's browser will be!)

Serious JS date/time handling libraries

https://day.js.org/

https://moment.github.io/luxon/

https://momentjs.com/

https://date-fns.org/

https://js-joda.github.io/js-joda/

Use native data types but with functions without the original problems

WAY BETTER!

They have very similar functionalities and are all used. Momentjs was the standard, very complete, until google introduced the possibility of seeing the size of the library, then it was discovered that it was HUGE. (They started new project: Luxon -> Java.Time library has similar philosophy

Day.js reimplemented moment library using smaller modules that you can load only if needed.

Day.js Library

DAY.JS https://day.js.org/

API

- Goals
  - Compatible with moment.js
    - But very small (2kB)
  - Works in nodejs and in the browser
  - All objects are *immutable*
    - All API functions that modify a date, will always return a new object instance
  - Localization
  - Plugin system for extending functionality

Safe programming model compatible with functional programming

- Install

```
npm init # if not already done
npm install dayjs
```

- Import

```
const dayjs = require('dayjs')
```

- Use

```
let now = dayjs()
console.log(now.format())
```

DayJs object data type!
Dayjs() is the constructor.

Basic operations with Day.js

Creating date objects – dayjs() constructor

```
let now = dayjs()  // today
let date1 = dayjs('2019-12-27T16:00');
        // from ISO 8601 format
let date2 = dayjs('20191227');
        // from 8-digit format
let date3 = dayjs(new Date(2019, 11, 27));
        // from JS Date object
let date5 = dayjs.unix(1530471537);
    // from Unix timestamp

By default, Day.js parses in local time
```

https://day.js.org/docs/en/parse/parse

Displaying date objects – format()

```
console.log(now.format());
        2021-03-02T16:38:38+01:00

console.log(now.format('YYYY-MM [on the] DD'));
        2021-03 on the 02

console.log(now.toString());
        Tue, 02 Mar 2021 15:43:46 GMT

By default, Day.js displays in local time
```

Compatible with sqlite

Human readable custom format

Old Date Object

Internally the object saves it as ISO format with universal time (if we try to save today's date, it knows our local time, so it convert the date to one our before! (yesterday at 23:00) and saves that! When we use it, with format, it will return today's date! Internally universal time, but what we see is local time! When we call the format function, it will return the ISO string of the LOCAL time.

We'll put inside the Database Strings object (iso time), but in the code we will use DayJS Objects, because they have a lot of functionalities!

Call function without params

Extract/modify dates

Get/Set date/time components

Call function with params

```
# obj.unit() -> get
# obj.unit(new_val) -> set

let now2 = now.date(15);
let now2 = now.set('date', 15);
        2021-03-15T16:50:26+01:00

let now3 = now.minute(45);
let now3 = now.set('minute',45);
        2021-03-02T16:45:26+01:00

let today_day = now.day();
let today_day = now.get('day');
        2
```

Returns new date with this different field

Unit	Shorthand	Description
date	D	Date of Month
day	d	Day of Week (Sunday as 0, Saturday as 6)
month	M	Month (January as 0, December as 11)
year	y	Year
hour	h	Hour
minute	m	Minute
second	s	Second
millisecond	ms	Millisecond

https://day.js.org/docs/en/get-set/get-set

Computing dates

Date Manipulation and Comparison

Value, unit, or measure

```
let wow = dayjs('2019-01-25').add(1, 'day').subtract(1, 'year').year(2009).toString() ;
// "Sun, 25 Jan 2009 23:00:00 GMT"
```

- Methods to "modify" a date (and return a modified one)
- `.add` / `.subtract`
- `.startOf` / `.endOf`
- `d1.diff(d2, 'unit')`
- Specify the unit to be added/subtracted/rounded
- Can be easily *chained*

- Day.js objects can be compared
- `.isBefore` / `.isSame` / `.isAfter`
- `.isBetween`
- `.isLeapYear` / `.daysInMonth`

Only use this functions!!! Why bother with all possible edge cases when library does exactly this for us?

Additional functionalities that we can load if we need them. (They are installed with the module, but they are loaded only when needed to keep memory footprint as low as possible!)

Day.js Plugins

- To keep install size minimal, several functions are only available in *plugins*
- Plugins must be
  - Loaded
  - Registered into the libraries
- Then, functions may be freely used

```
const isLeapYear =
   require('dayjs/plugin/isLeapYear') ;
        // load plugin

dayjs.extend(isLeapYear) ;
        // register plugin

console.log(now.isLeapYear()) ;
        // use function
```


Advanced Day.js Topics

- Localization / Internationalization
  - Language-aware and locale-aware parsing and formatting
  - Various formatting patterns for different locales/languages
- Durations
  - Measuring time intervals (the difference between two time instants)
  - Interval arithmetic
- Time Zones
  - Conversion between time zones

License

- These slides are distributed under a Creative Commons license "**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**"
- **You are free to:**
  - **Share** — copy and redistribute the material in any medium or format
  - **Adapt** — remix, transform, and build upon the material
  - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
  - **Attribution** — You must give appropriate credit, provide a link to the license, and indicate if changes were made. You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
  - **NonCommercial** — You may not use the material for commercial purposes.
  - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the same license as the original.
  - **No additional restrictions** — You may not apply legal terms or technological measures that legally restrict others from doing anything the license permits.
- https://creativecommons.org/licenses/by-nc-sa/4.0/