

<WA1/>
<AW1/>
2026

JavaScript: Objects and Functions

The language of the Web

Fulvio Corno

Luigi De Russis

JavaScript Cheat Sheet page 2
Programming Language of Web JS JS

Math PROPERTIES <ul style="list-style-type: none"><code>E</code> Euler's constant<code>LN2</code> natural logarithm of 2<code>LN10</code> natural logarithm of 10<code>LOG2E</code> base 2 logarithm of E<code>LOG10E</code> base 10 logarithm of E<code>PI</code> ratio circumference/diameter<code>SQRT1_2</code> square root of 1/2<code>SQRT2</code> square root of 2 METHODS <ul style="list-style-type: none"><code>abs(x)</code> absolute value<code>cbrt(x)</code> cube root<code>clz32(x)</code> return leading zero bits (32)<code>exp(x)</code> return e^x<code>expm1(x)</code> return e^x-1<code>hypot(x1, x2...)</code> length of hypotenuse<code>imul(a, b)</code> signed multiply<code>log(x)</code> natural logarithm (base e)<code>log1p(x)</code> natural logarithm (1+x)<code>log10(x)</code> base 10 logarithm<code>log2(x)</code> base 2 logarithm<code>max(x1, x2...)</code> return max number<code>min(x1, x2...)</code> return min number<code>pow(base, exp)</code> return base^{exp}<code>random()</code> float random number [0,1)<code>sign(x)</code> return sign of number<code>sqrt(x)</code> square root of number ROUND METHODS <ul style="list-style-type: none"><code>ceil(x)</code> superior round (smallest)<code>floor(x)</code> inferior round (largest)<code>fround(x)</code> nearest single precision<code>round(x)</code> round (nearest integer)<code>trunc(x)</code> remove fractional digits TRIGONOMETRIC METHODS <ul style="list-style-type: none"><code>acos(x)</code> arccosine<code>acosh(x)</code> hyperbolic arccosine<code>asin(x)</code> arcsine<code>asinh(x)</code> hyperbolic arcsine<code>atan(x)</code> arctangent<code>atan2(x, y)</code> arctangent of quotient x/y<code>atanh(x)</code> hyperbolic arctangent<code>cos(x)</code> cosine<code>cosh(x)</code> hyperbolic cosine<code>sin(x)</code> sine<code>sinh(x)</code> hyperbolic sine<code>tan(x)</code> tangent<code>tanh(x)</code> hyperbolic tangent JSON METHODS <ul style="list-style-type: none"><code>parse(str, tf(k,v))</code> parse string to object<code>stringify(obj, repl[w], sp)</code> convert to str Error() PROPERTIES <ul style="list-style-type: none"><code>name</code> return name of error<code>message</code> return description of error	Object() PROPERTIES <ul style="list-style-type: none"><code>constructor</code> return ref. to object func. METHODS <ul style="list-style-type: none"><code>assign(dst, src1, src2...)</code> copy values<code>create(proto, prop)</code> create obj w/prop<code>defineProperties(obj, prop)</code><code>defineProperty(obj, prop, desc)</code><code>freeze(obj)</code> avoid properties changes<code>getOwnPropertyDescriptor(obj, prop)</code><code>getOwnPropertyNames(obj)</code><code>getOwnPropertySymbols(obj)</code><code>getPrototypeOf(obj)</code> return prototype<code>is(val1, val2)</code> check if are same value<code>isExtensible(obj)</code> check if can add prop<code>isFrozen(obj)</code> check if obj is frozen<code>isSealed(obj)</code> check if obj is sealed<code>keys(obj)</code> return only keys of object<code>preventExtensions(obj)</code> avoid extend<code>seal(obj)</code> prop are non-configurable<code>setPrototypeOf(obj, prot)</code> change prot INSTANCE METHODS <ul style="list-style-type: none"><code>hasOwnProperty(prop)</code> check if exist<code>isPrototypeOf(obj)</code> test in another obj<code>propertyIsEnumerable(prop)</code><code>toString()</code> return equivalent string<code>toLocaleString()</code> return locale version<code>valueOf()</code> return primitive value Promise() METHODS <ul style="list-style-type: none"><code>all(obj)</code> return promise<code>catch(onRejected(s)) = .then(undef,s)</code><code>then(onFulfilled(v), onRejected(s))</code><code>race(obj)</code> return greedy promise (res/rej)<code>resolve(obj)</code> return resolved promise<code>reject(reason)</code> return rejected promise Proxy() METHODS <ul style="list-style-type: none"><code>apply(obj, arg, arglist)</code> trap function call<code>construct(obj, arglist)</code> trap new oper<code>defineProperty(obj, prop, desc)</code><code>deleteProperty(obj, prop)</code> trap delete<code>enumerate(obj)</code> trap for...in<code>get(obj, prop, rec)</code> trap get property<code>getOwnPropertyDescriptor(obj, prop)</code><code>getPrototypeOf(obj)</code><code>has(obj, prop)</code> trap in operator<code>ownKeys(obj)</code><code>preventExtensions(obj)</code><code>set(obj, prop, value)</code> trap set property<code>setPrototypeOf(obj, proto)</code> globals METHODS <ul style="list-style-type: none"><code>eval(str)</code> evaluate javascript code<code>isFinite(obj)</code> check if is a finite number<code>isNaN(obj)</code> check if is not a number<code>parseInt(s, radix)</code> string to integer<code>parseFloat(s, radix)</code> string to float<code>encodeURIComponent(URI)</code> = to %3D<code>decodeURIComponent(URI)</code> %3D to =	Set() PROPERTIES <ul style="list-style-type: none"><code>size</code> return number of items METHODS <ul style="list-style-type: none"><code>add(item)</code> add item to set ws<code>has(item)</code> check if item exists ws<code>delete(item)</code> del item & return if del ws<code>clear()</code> remove all items from set ITERATION METHODS <ul style="list-style-type: none"><code>entries()</code> iterate items<code>values()</code> iterate only value of items CALLBACK FOR EACH METHODS <ul style="list-style-type: none"><code>forEach(cb(e,i,a), arg)</code> exec for each Map() PROPERTIES <ul style="list-style-type: none"><code>size</code> return number of elements METHODS <ul style="list-style-type: none"><code>set(key, value)</code> add pair key=value wm<code>get(key)</code> return value of key wm<code>has(key)</code> check if key exist wm<code>delete(key)</code> del elem. & return if ok wm<code>clear()</code> remove all elements from map ITERATION METHODS <ul style="list-style-type: none"><code>entries()</code> iterate elements<code>keys()</code> iterate only keys<code>values()</code> iterate only values CALLBACK FOR EACH METHODS <ul style="list-style-type: none"><code>forEach(cb(e,i,a), arg)</code> exec for each Symbol() PROPERTIES <ul style="list-style-type: none"><code>iterator</code> specifies default iterator<code>match</code> specifies match of regexp<code>species</code> specifies constructor function METHODS <ul style="list-style-type: none"><code>for(key)</code> search existing symbols<code>keyFor(sym)</code> return key from global reg Generator() METHODS <ul style="list-style-type: none"><code>next(value)</code> return obj w/(value,done)<code>return(value)</code> return value & true done<code>throw(throw)</code> throw an error Others FAST TIPS <ul style="list-style-type: none"><code>var</code> declare variablelet declare block scope local variable<code>const</code> declare constant (read-only)<code>func(a=1)</code> default parameter value<code>func(...a)</code> rest argument (spread operator)<code>(a) => { ... }</code> function equivalent (fat arrow)<code>'string \${a}'</code> template with variables<code>0bn</code> binary (2) number <code>n</code> to decimal<code>0on</code> octal (8) number <code>n</code> to decimal<code>0xn</code> hexadecimal (16) number <code>n</code> to decimal<code>for (i in array) { ... }</code> iterate array, <code>i</code> = index<code>for (e of array) { ... }</code> iterate array, <code>e</code> = value<code>class B extends A { }</code> class sugar syntax
---	--	--

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*)

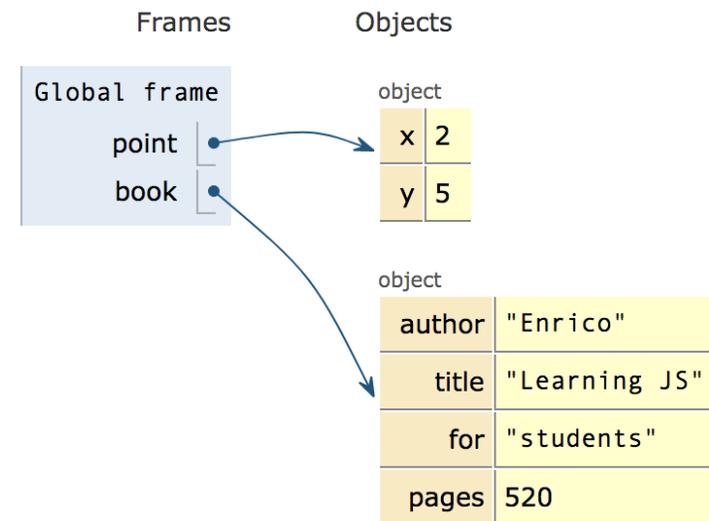
- In JavaScript, Objects may exist without Classes
 - Usually, Objects are **created directly**, without deriving them from a Class definition
- In JavaScript, Objects are dynamic
 - You may **add, delete, redefine** a *property* at any time
 - You may add, delete, redefine a *method* at any time
- In 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)

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, }



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*)

Accessing Properties

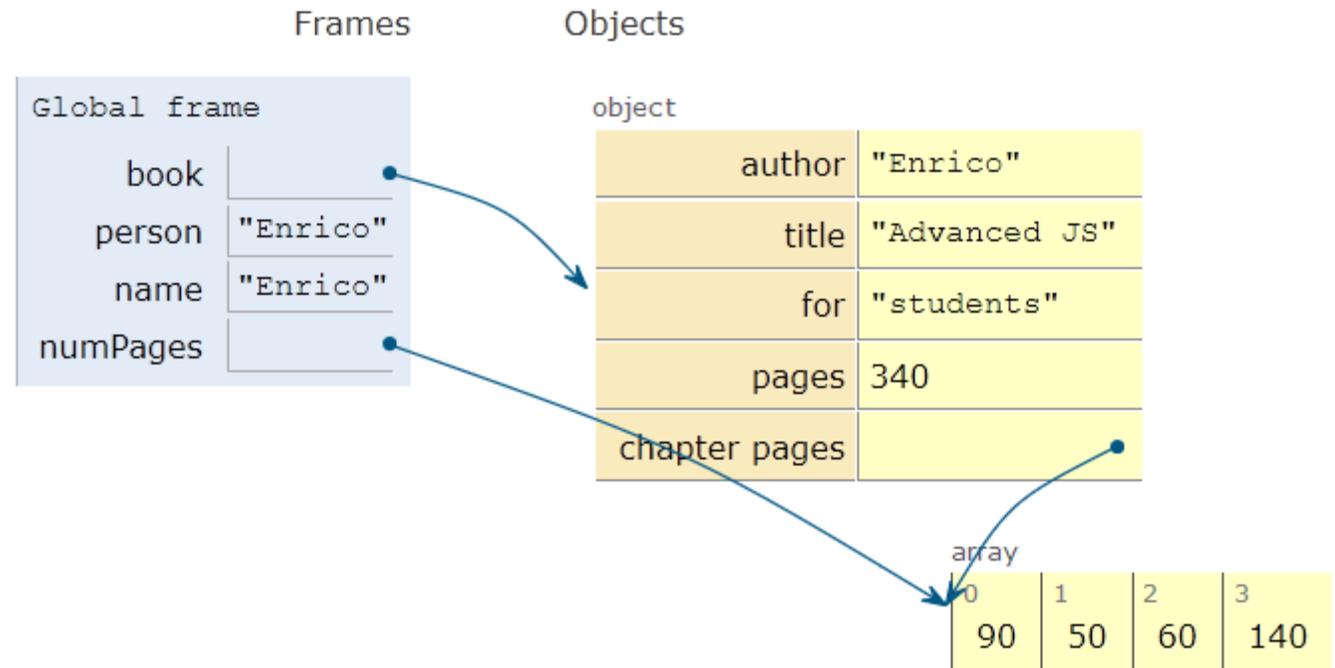
The . dot notation and omitting the quotes are allowed **when the property name is a valid identifier, only.**

`book.title` or `book['title']`
`book['my title']` and not `book.my title`

- Dot (.) or square brackets [] notation

```
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;
```



Objects as Associative Arrays

- 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

- 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

Property Access Errors

- If a property is not defined, the (attempted) access returns `undefined`
- If unsure, must check before accessing
 - Remember: `undefined` is *falsy*, you may use it in Boolean expressions

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

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

Iterating over properties

- **for .. in** iterates over the properties

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

```
x  
y
```

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

```
for (const prop in book)  
    console.log(`${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) ;`

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

- All pairs [key, value] are returned as an array with:

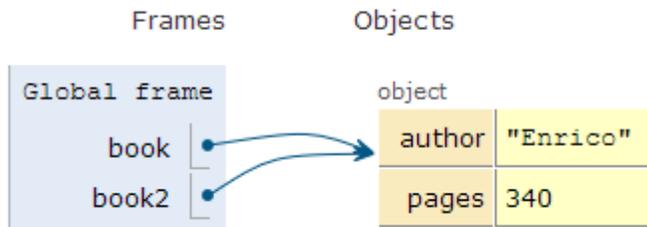
- `let keys_values = Object.entries(my_object)`

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

Copying Objects

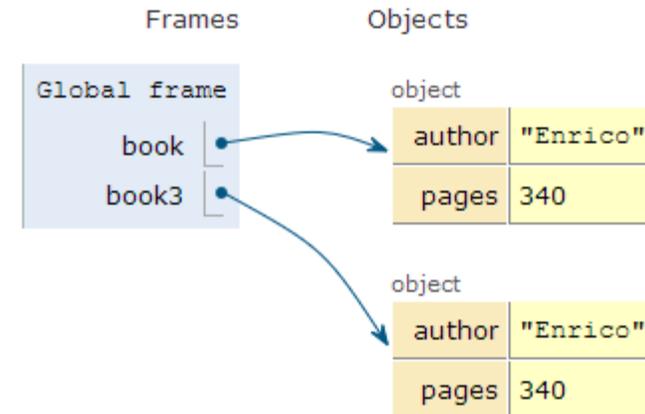
```
let book = {  
  author : "Enrico",  
  pages: 340,  
};
```

```
let book2 = book; // ALIAS
```



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

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



Object.assign

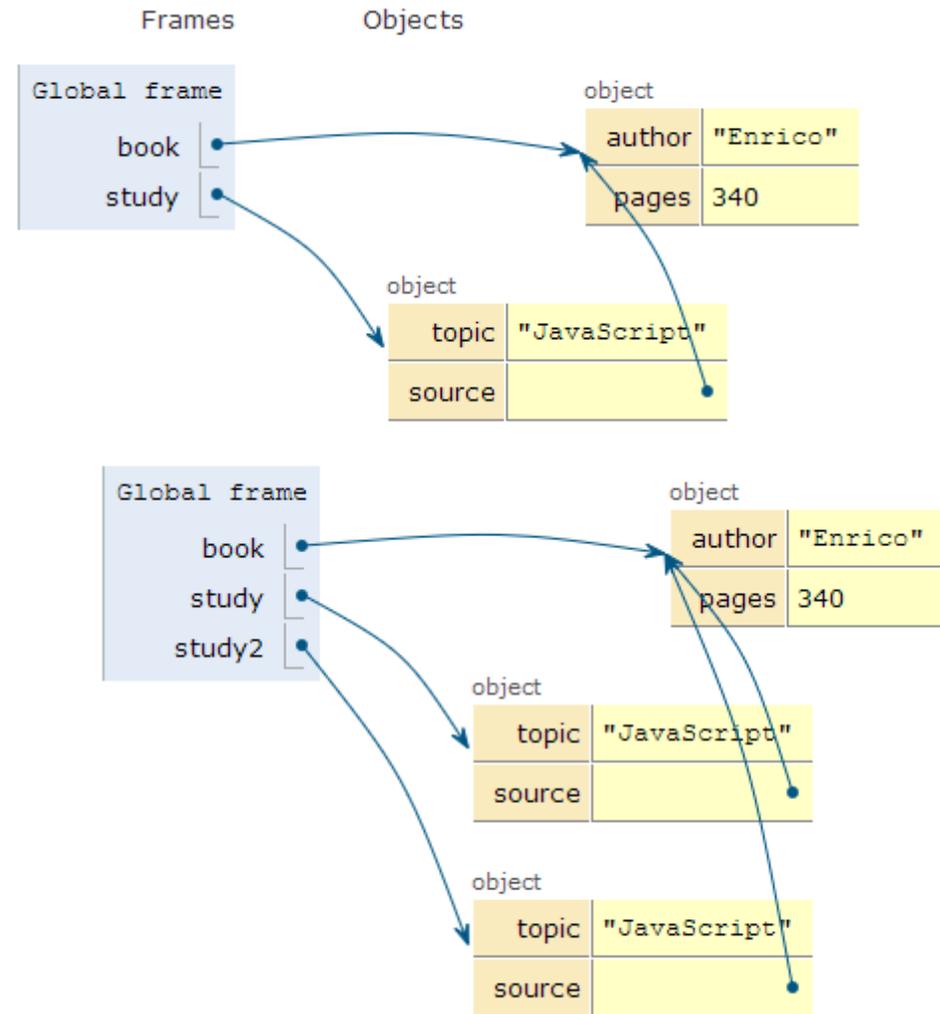
- `let new_object = Object.assign(target, source);`
- Assigns all the properties from the `source` object to the `target` one
- The target may be an existing object
- The target may be a new object: `{}`
- Returns the target object (after modification)

Beware! Shallow copy, only

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

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

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

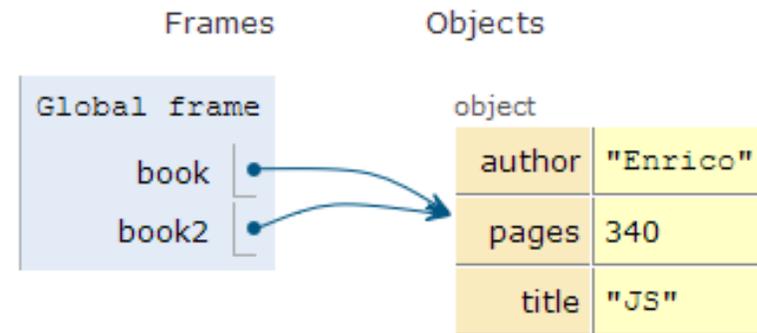


Merge properties (on existing object)

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

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

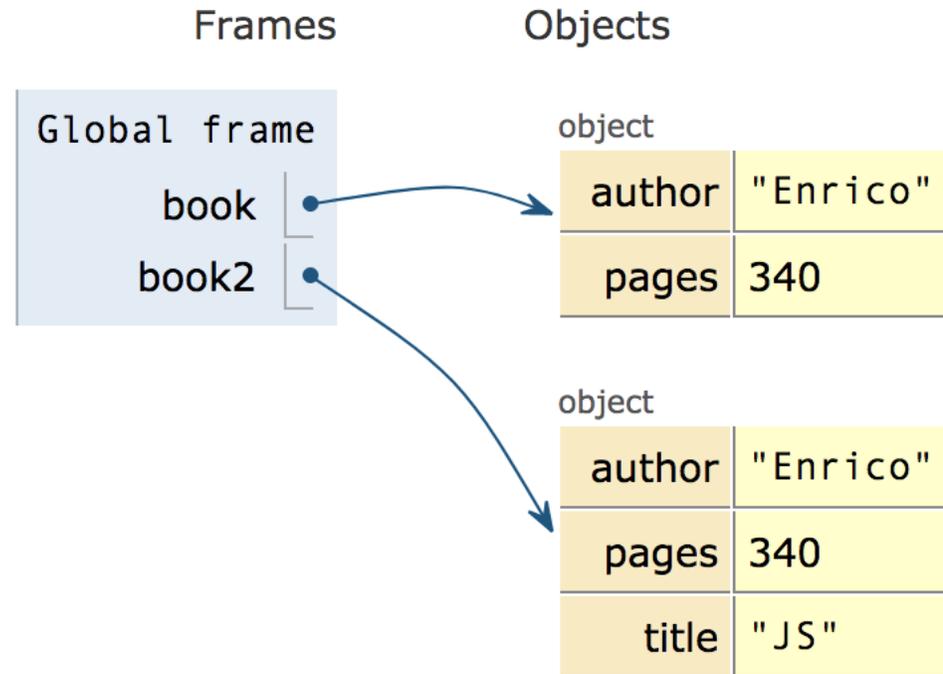
let book2 = Object.assign(
  book, {title: "JS"}
);
```



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)

```
let book = {  
  author : "Enrico",  
  pages: 340,  
};  
  
let book2 = {...book, title: "JS"};  
let book3 = { ...book2 } ;  
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 }
```

Checking if properties exist

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

Preferred



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
- 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

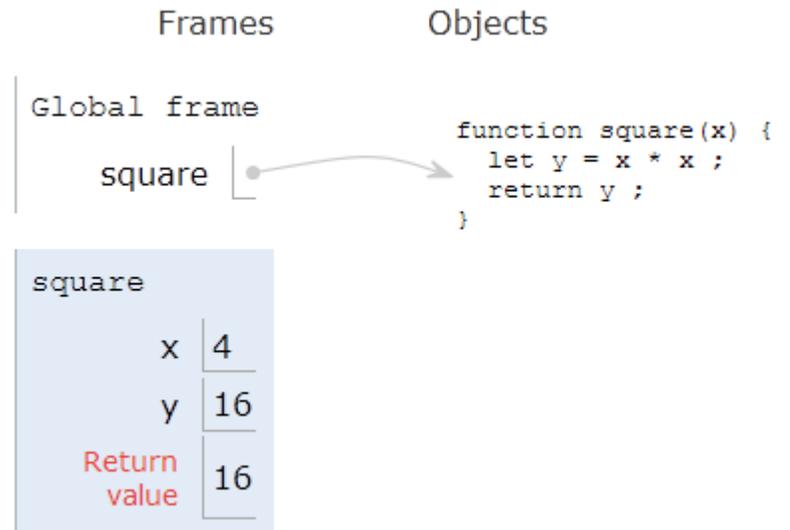
```
function do(params) {  
  /* do something */  
}
```

Classic functions

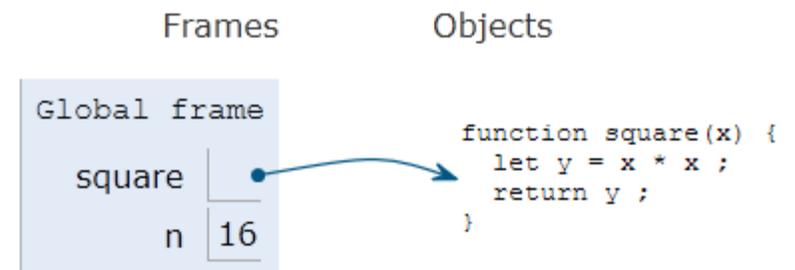
```
function square(x) {  
  let y = x * x ;  
  return y ;  
}
```

```
let n = square(4) ;
```

During
execution



After
execution



Parameters

- 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 ;`

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 */  
}
```

2a) Function expression

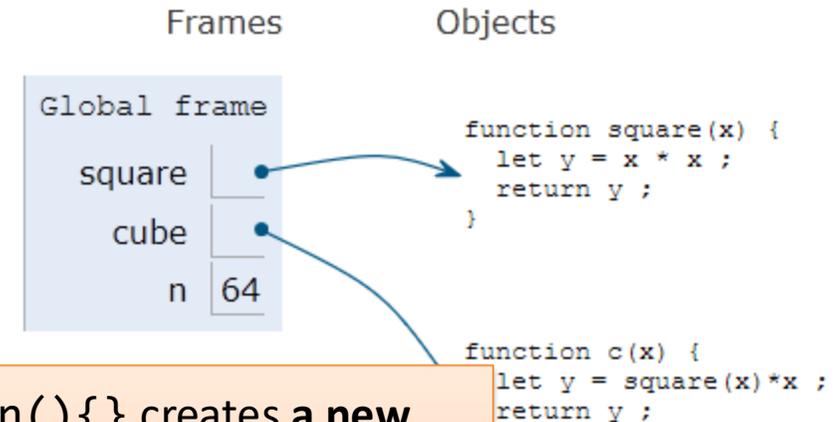
```
const fn = function(params) {  
  /* do something */  
}
```

2b) Named function expression

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

Function expression: indistinguishable

```
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

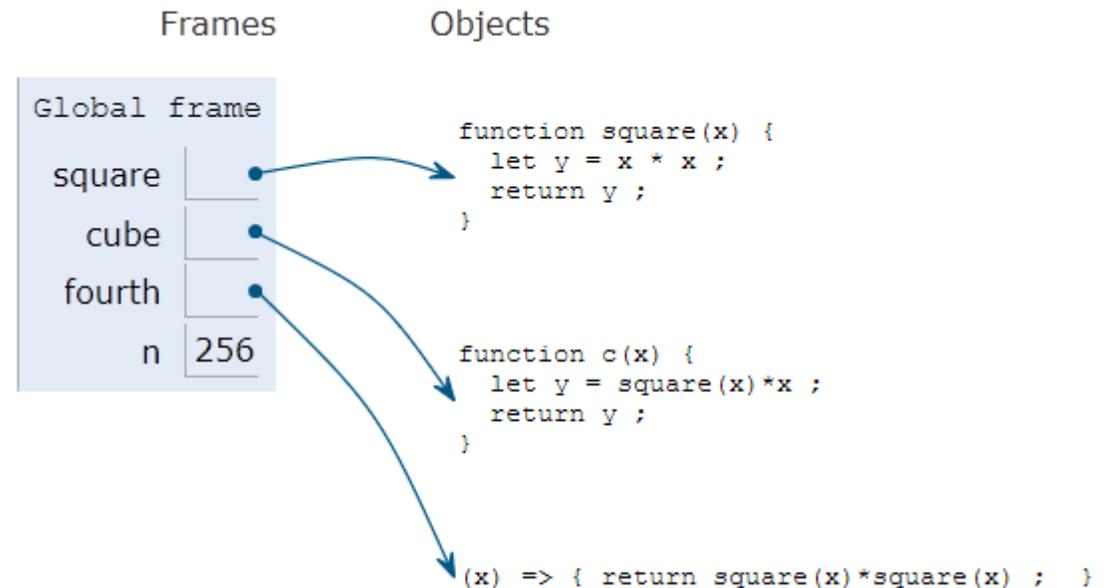
```
const fn = (params) => {  
  /* do something */  
}
```

2b) Named function expression

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

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
```

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) ;
```

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));  
}
```

=> Preferred in nested functions

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

- 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

Closure: definition (somewhat cryptic)

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**.

Really: an important
concepts in JS

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

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

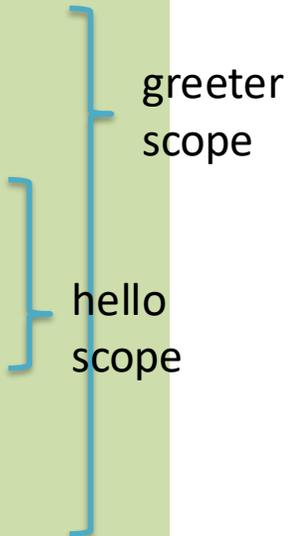
- `hello` accesses the variable `myname`, defined in the outer scope
- The function is returned (as `helloTom` or `helloJerry`)
- Each of the functions “remembers” the reference to `myname`, when it was defined
- The variable `myname` goes out of scope, but is not destroyed
 - Still accessible (referred)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



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; }
  };
}
```

```
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
```

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 ;  
} ) () ;
```

<https://flaviocopes.com/javascript-iife/>

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

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());
```

```
0
1
0
1
```

Construction functions

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

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

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



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/>

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

Once the [Temporal API](#) will be stable, the Date object will be a *legacy* feature.

```
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
```

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
```



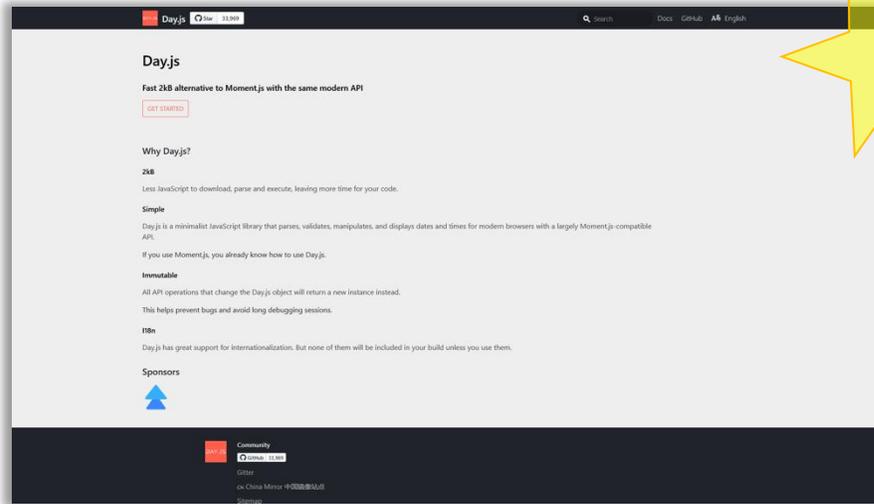
Formatting is locale and implementation dependent



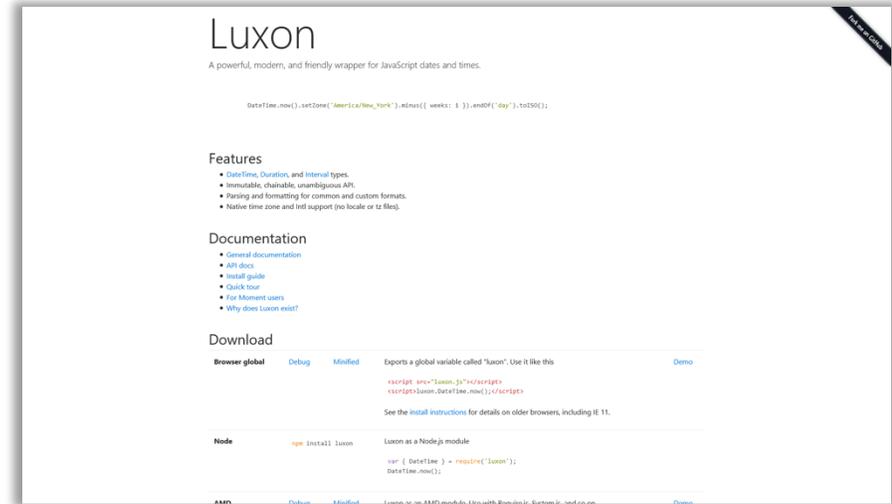
Comparisons are difficult (no way to specify which fields you want, must set them to zero explicitly)



Serious JS date/time handling libraries



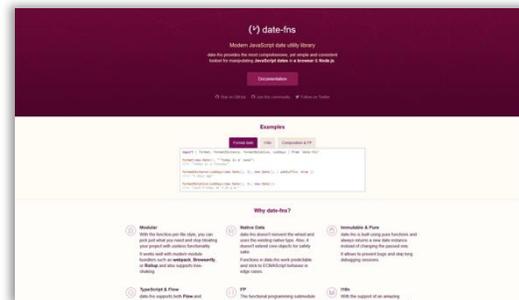
<https://day.js.org/>



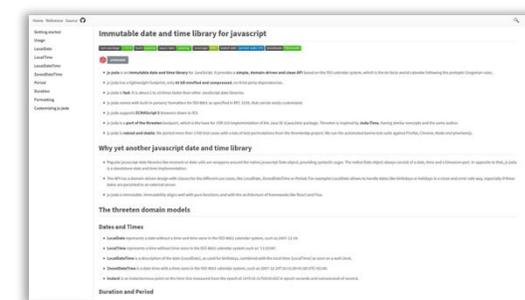
<https://moment.github.io/luxon/>



<https://momentjs.com/>



<https://date-fns.org/>



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

Day.js Library

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

- 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
- Install

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

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

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

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

Get/Set date/time components

```
# 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
```

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>

Date Manipulation and Comparison

```
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`

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/>

