

<WA1/>

2020

# JavaScript (Part 2)

“The” language of the Web

Enrico Masala

Fulvio Corno

JS

JavaScript

Cheat Sheet

page 2

JS

Math

PROPERTIES

- `E` Euler's constant
- `LN2` natural logarithm of 2
- `LN10` natural logarithm of 10
- `LOG2E` base 2 logarithm of E
- `LOG10E` base 10 logarithm of E
- `PI` ratio circumference/diameter
- `SQRT1_2` square root of 1/2
- `SQRT2` square root of 2

METHODS

- `abs(x)` absolute value
- `cbrt(x)` cube root
- `clz32(x)` return leading zero bits (32)
- `exp(x)` return e<sup>x</sup>
- `expm1(x)` return e<sup>x</sup>-1
- `hypot(x1, x2...)` length of hypotenuse
- `imul(a, b)` signed multiply
- `log(x)` natural logarithm (base e)
- `log1p(x)` natural logarithm (1+x)
- `log10(x)` base 10 logarithm
- `log2(x)` base 2 logarithm
- `max(x1, x2...)` return max number
- `min(x1, x2...)` return min number
- `pow(base, exp)` return base<sup>exp</sup>
- `sign(x)` return sign of number
- `sqrt(x)` square root of number

ROUND METHODS

- `ceil(x)` superior round (smallest)
- `floor(x)` inferior round (largest)
- `fround(x)` nearest single precision
- `round(x)` round (nearest integer)
- `trunc(x)` remove fractional digits

TRIGONOMETRIC METHODS

- `acos(x)` arccosine
- `acosh(x)` hyperbolic arccosine
- `asin(x)` arcsine
- `asinh(x)` hyperbolic arcsine
- `atan(x)` arctangent
- `atan2(x, y)` arctangent of quotient x/y
- `atanh(x)` hyperbolic arctangent
- `cos(x)` cosine
- `cosh(x)` hyperbolic cosine
- `sin(x)` sine
- `sinh(x)` hyperbolic sine
- `tan(x)` tangent
- `tanh(x)` hyperbolic tangent

JSON

METHODS

- `parse(str, tf(k,v))` parse string to object
- `stringify(obj, replf(wl, sp))` convert to str

Error()

PROPERTIES

- `name` return name of error
- `message` return description of error

Object()

PROPERTIES

- `constructor` return ref. to object func.

METHODS

- `assign(dst, src1, src2...)` copy values
- `create(proto, prop)` create obj w/prop
- `defineProperties(obj, prop)`
- `defineProperty(obj, prop, desc)`
- `freeze(obj)` avoid properties changes
- `getOwnPropertyDescriptor(obj, prop)`
- `getOwnPropertyNames(obj)`
- `getOwnPropertySymbols(obj)`
- `getPrototypeOf(obj)` return prototype
- `is(val1, val2)` check if are same value
- `isExtensible(obj)` check if can add prop
- `isFrozen(obj)` check if obj is frozen
- `isSealed(obj)` check if obj is sealed
- `keys(obj)` return only keys of object
- `preventExtensions(obj)` avoid extend
- `seal(obj)` prop are non-configurable
- `setPrototypeOf(obj, prot)` change prot

INSTANCE METHODS

- `hasOwnProperty(prop)` check if exist
- `isPrototypeOf(obj)` test in another obj
- `propertyIsEnumerable(prop)`
- `toString()` return equivalent string
- `toLocaleString()` return locale version
- `valueOf()` return primitive value

Promise()

METHODS

- `all(obj)` return promise
- `catch(onRejected(s)) = .then(undef,s)`
- `then(onFulfilled(v), onRejected(s))`
- `race(obj)` return greedy promise (res/req)
- `resolve(obj)` return resolved promise
- `reject(reason)` return rejected promise

Proxy()

METHODS

- `apply(obj, arg, arglist)` trap function call
- `construct(obj, arglist)` trap new oper
- `defineProperty(obj, prop, desc)`
- `deleteProperty(obj, prop)` trap delete
- `enumerate(obj)` trap for...in
- `get(obj, prop, rec)` trap get property
- `getOwnPropertyDescriptor(obj, prop)`
- `getPrototypeOf(obj)`
- `has(obj, prop)` trap in operator
- `ownKeys(obj)`
- `preventExtensions(obj)`
- `set(obj, prop, value)` trap set property
- `setPrototypeOf(obj, proto)`

globals

METHODS

- `eval(str)` evaluate javascript code
- `isFinite(obj)` check if is a finite number
- `isNaN(obj)` check if is not a number
- `parseInt(s, radix)` string to integer
- `parseFloat(s, radix)` string to float
- `encodeURIComponent(URI)` = to %3D
- `decodeURIComponent(URI)` %3D to =

Set()

PROPERTIES

- `size` return number of items

METHODS

- `add(item)` add item to set
- `has(item)` check if item exists
- `delete(item)` del item & return if del
- `clear()` remove all items from set

ITERATION METHODS

- `entries()` iterate items
- `values()` iterate only value of items

CALLBACK FOR EACH METHODS

- `forEach(cb(e,i,a), arg)` exec for each

Map()

PROPERTIES

- `size` return number of elements

METHODS

- `set(key, value)` add pair key=value
- `get(key)` return value of key
- `has(key)` check if key exist
- `delete(key)` del elem. & return if ok
- `clear()` remove all elements from map

ITERATION METHODS

- `entries()` iterate elements
- `keys()` iterate only keys
- `values()` iterate only values

CALLBACK FOR EACH METHODS

- `forEach(cb(e,i,a), arg)` exec for each

Symbol()

PROPERTIES

- `iterator` specifies default iterator
- `match` specifies match of regexp
- `species` specifies constructor function

METHODS

- `for(key)` search existing symbols
- `keyFor(sym)` return key from global reg

Generator()

METHODS

- `next(value)` return obj w/(value,done)
- `return(value)` return value & true done
- `throw(exception)` throw an error

Others

FAST TIPS

- `var` declare variable
- `let` declare block scope local variable
- `const` declare constant (read-only)
- `func(a=1)` default parameter value
- `func(...a)` rest argument (spread operator)
- `(a) => { ... }` function equivalent (fat arrow)
- ``string ${a}`` template with variables
- `0n` octal (8) number `n` to decimal
- `0bn` binary (2) number `n` to decimal
- `0on` octal (8) number `n` to decimal
- `0xn` hexadecimal (16) number `n` to decimal
- `for (i in array) { ... }` iterate array, i = index
- `for (e of array) { ... }` iterate array, e = value
- `class B extends A { }` class sugar syntax

CodeMio

PROFESSIONAL & INSPIRED



POLITECNICO  
DI TORINO



# Outline

- Objects
- Functions
  - Closures
- Callbacks
  - Timers
- 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)

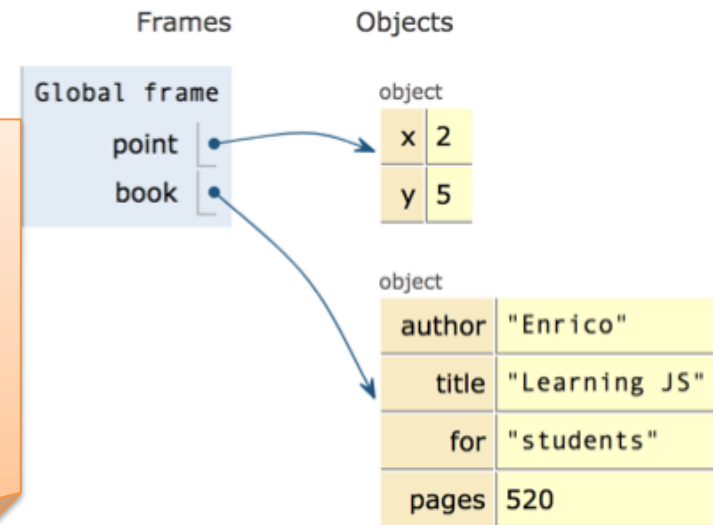
# Objects

- An object is an unordered collection of properties
  - Each property has a **name** (key), and a **value**
- 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 ...

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

# Accessing properties

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

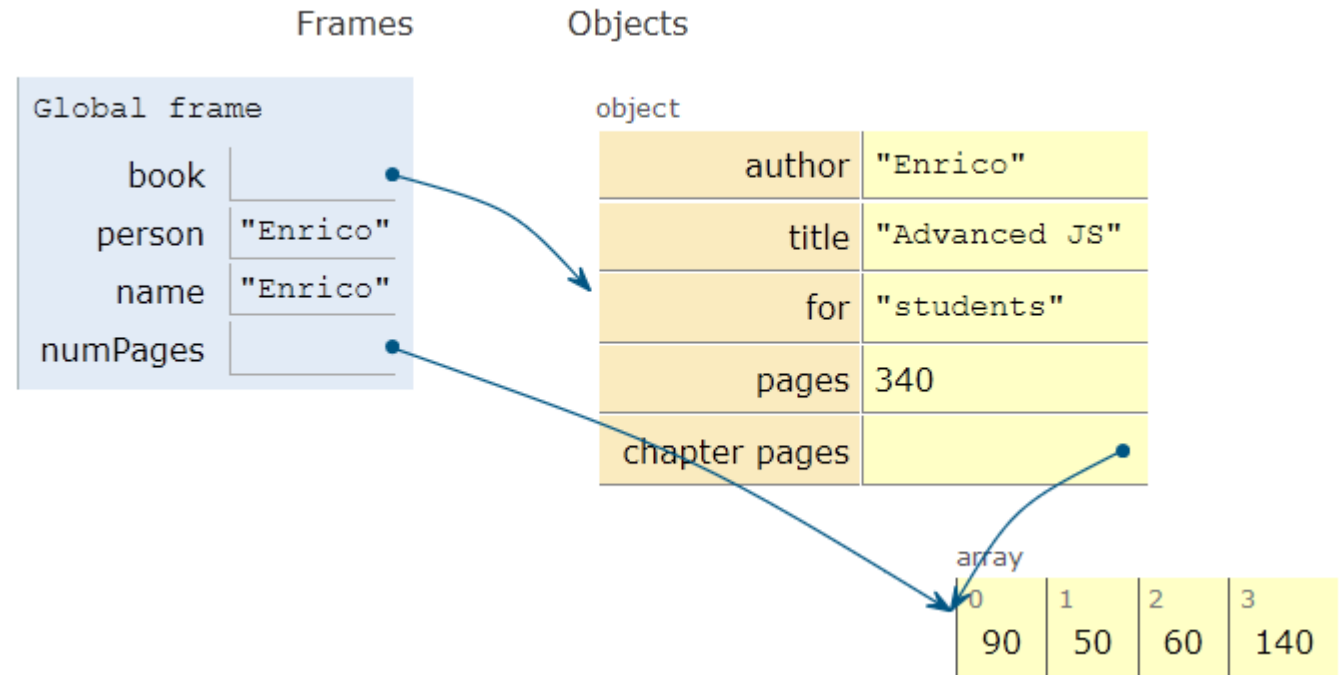
`book.title` or `book['title']`

`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 accessing object properties
  - e.g., access *i*-th line of an object with multiple "address" properties (address1, address2, ...): `person["address"+i]`
  - **Not really 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

```
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:

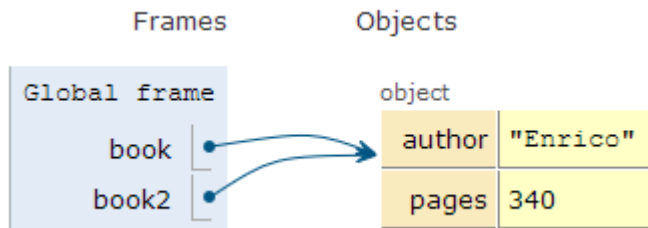
- `let keys_values = Object.entries(my_object)`

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

# Copying objects

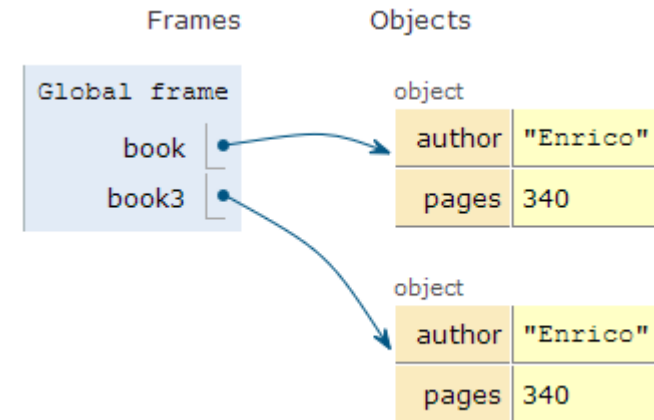
```
let book = {  
  author : "Enrico",  
  pages: 340,  
};
```

```
let book2 = book;
```



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

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



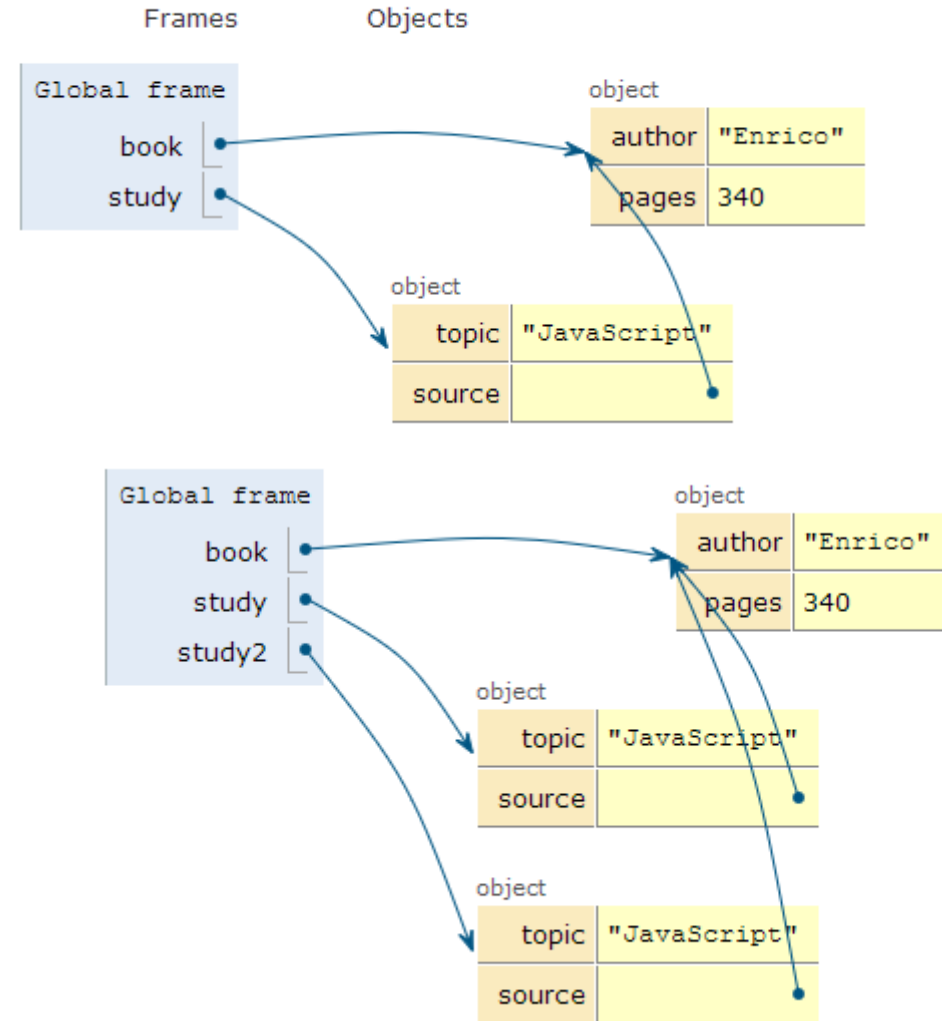
# Object.assign

- `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,  
};
```

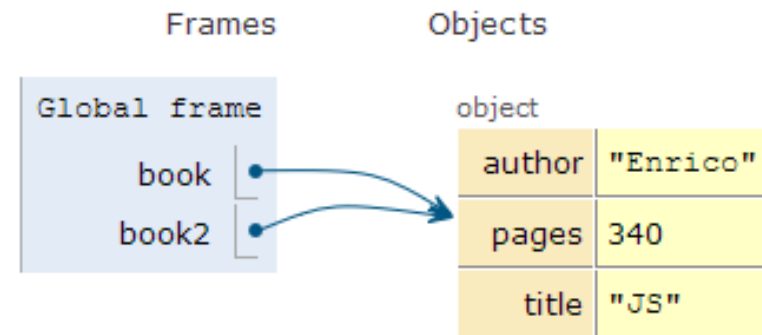
```
let study2 = Object.assign({},  
  study);
```



# Merge properties (on existing object)

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

```
let book = {  
  author : "Enrico",  
  pages: 340,  
};  
  
let book2 = Object.assign(  
  book, {title: "JS"}  
);
```

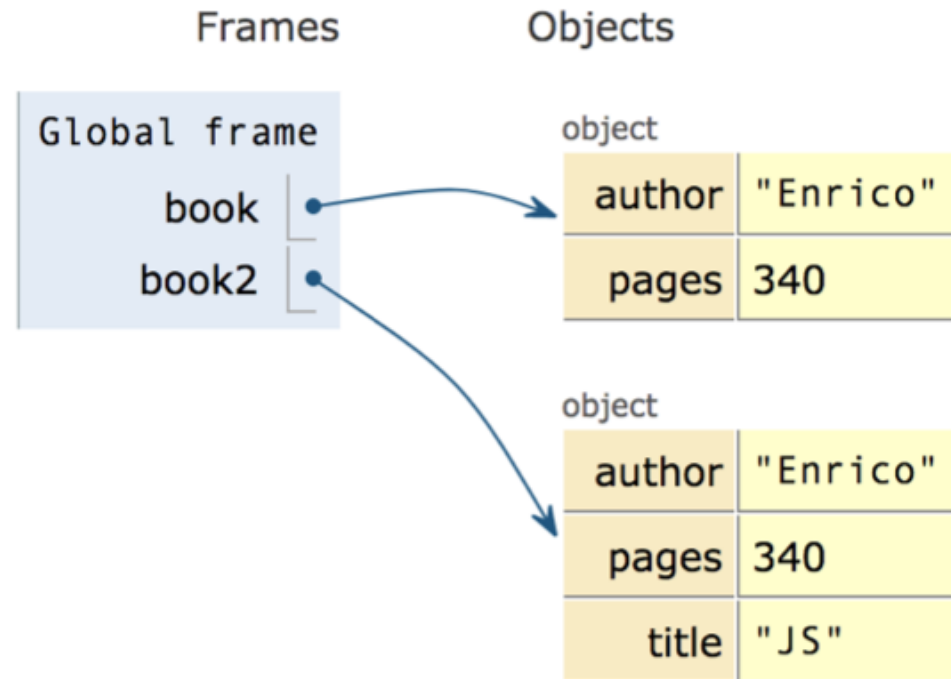




# Merge properties (on new object)

- `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"};  
  
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

# Construction functions

- Define the object type by writing a constructor function.
  - Use a capital initial letter
- Create an instance of the object with new.

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

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



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 */  
}
```

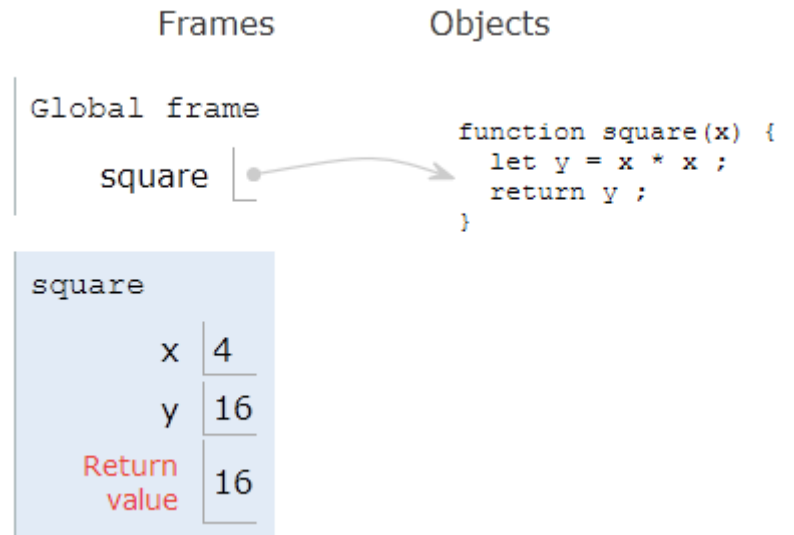


# Classic functions

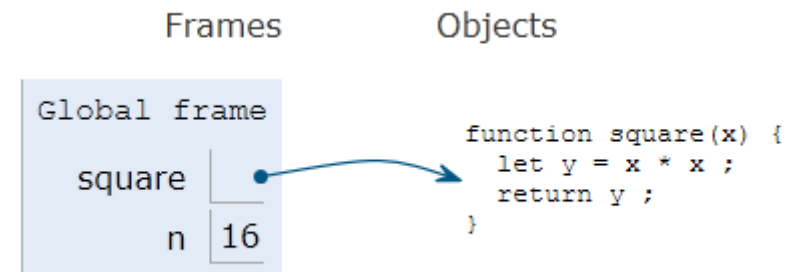
```
function square(x) {  
  let y = x * x ;  
  return y ;  
}
```

```
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

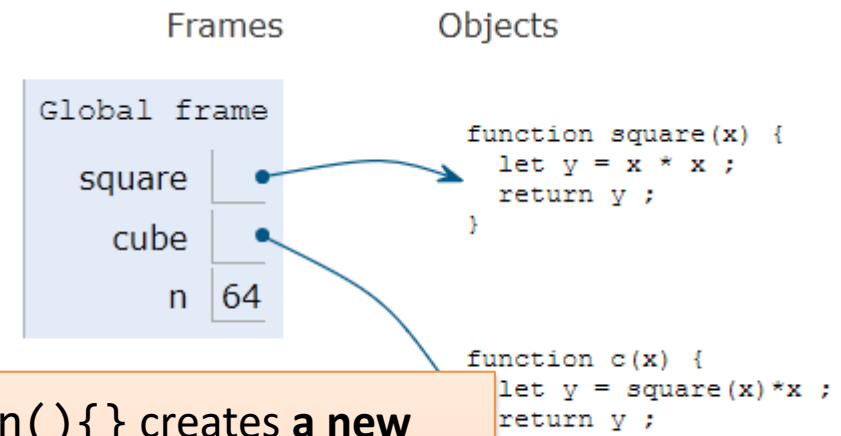
```
const fn = function(params) {  
  /* do something */  
}
```

## 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

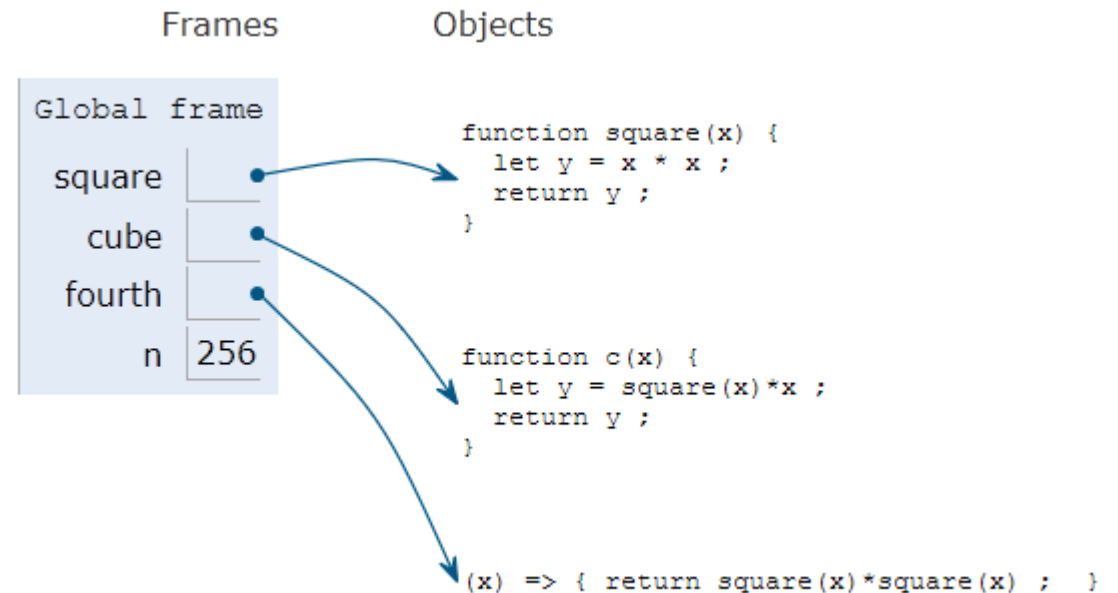
```
const fn = (params) => {  
  /* do something */  
}
```

## 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: one of the most 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

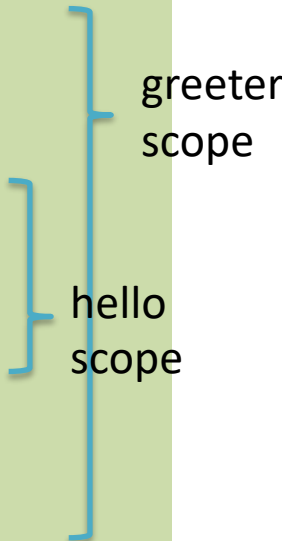
- hello accesses the variable myname, defined in the outer scope
- The function is returned (as helloTom or helloJerry)
- Each of the functions “remembers” the reference to myname, when it was defined
- The variable myname goes out of scope, but is not destroyed
  - Still accessible (referred) by the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



JavaScript – The language of the Web

# **CALLBACKS**

# Callbacks

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.
  - Synchronous
  - Asynchronous

```
function logQuote(quote) {  
    console.log(quote);  
}  
  
function createQuote(quote,  
    callback) {  
    const myQuote = `Like I always  
say, '${quote}'`;   
    callback(myQuote);  
}  
  
createQuote("WebApp I rocks!",  
    logQuote);
```

# Synchronous callbacks

- Used in functional programming
  - E.g., providing the sort criteria for array sorting

```
var numbers = [4, 2, 5, 1, 3];  
  
numbers.sort(function(a, b) {  
    return a - b;  
});  
  
console.log(numbers);
```

```
let numbers = [4, 2, 5, 1, 3];  
  
numbers.sort((a, b) => a - b);  
  
console.log(numbers);
```

# Synchronous callbacks

- Example: filter according to a criteria
  - `filter()` creates a **new** array with all elements for which the callback returns true

```
const market = [  
  { name: 'GOOG', var: -3.2 },  
  { name: 'AMZN', var: 2.2 },  
  { name: 'MSFT', var: -1.8 }  
];  
  
const bad = market.filter(stock => stock.var < 0);  
// [ { name: 'GOOG', var: -3.2 }, { name: 'MSFT', var: -1.8 } ]  
  
const good = market.filter(stock => stock.var > 0);  
// [ { name: 'AMZN', var: 2.2 } ]
```

# Asynchronous callbacks

- Handling user actions
  - E.g., button click
- Handling I/O operations
  - E.g., fetch a document
- Handling time intervals
  - E.g., timers

# Timers

- Useful to delay the execution of a function. Two possibilities from the runtime environment
  - `setTimeout()` runs the callback function after a given period of time
  - `setInterval()` runs the callback function periodically

```
const onesec = setTimeout(()=> {  
    console.log('hey') ; // after 1s  
}, 1000) ;  
  
console.log('hi') ;
```

Note: timeout value in ms,  $< 2^{31}-1$  (about 24 days)

```
const myFunction = (firstParam,  
secondParam) => {  
    // do something  
}  
  
// runs after 2 seconds  
setTimeout(myFunction, 2000,  
firstParam, secondParam)
```

# Timers

- `clearInterval()`: for stopping the periodical invocation of `setInterval`

```
const id = setInterval(() => {}, 2000) ;  
  
// «id» is a handle that refers to the timer  
  
clearInterval(id) ;
```





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

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 = Date();
```

```
let newYearMorning = new Date(  
  2020, // Year 2020  
  0, // January (from 0)  
  1, // 1st  
  18, 15, 10, 743);  
// 18:15:10.743, local time
```

# Creating dates with new Date()

1. No parameters: UTC vs Local time zone are confusing.  
that represents

```
> new Date('2020-03-18')  
2020-03-18T00:00:00.000Z
```

2. A number parameter: represents the time in milliseconds since Jan 1970 00:00

```
> new Date('18 March 2020')  
2020-03-17T23:00:00.000Z
```

3. A string, which represents a formatted date

4. A series of numbers, which represent parts of a date

Formatting is locale-dependent 🤖

– At least 3 values: y, m, d

Remember the **new** keyword

```
new Date();  
new Date(1530826365*1000);  
let date = new Date('Mar 16, 2020');  
let date = new Date('3/16/2020');  
// careful with day/month order!
```

```
let newYearAfternoon = new Date(  
  2020, // Year 2020  
  0, // January (from 0)  
  1, // 1st  
  18);  
// 18:00:00.000, local time
```

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

# Date transformation

- `Date.parse()`
  - Static method, returns a timestamp in ms, not a Date object
  - A lot of string formats supported, as for the constructor parameter
- Edit fields in the date
  - get and set methods
- `to...String()`
  - to obtain human-readable dates
- `getTime()`
  - to get timestamp in ms

```
let ts1 = Date.parse('Mon 16 2020');  
let ts2 = Date.parse('2020-03-16 09:35:22');  
let ts3 = Date.parse('3/16/2020');  
let ts4 = Date.parse('2020 MARCH');
```

```
let now = Date();  
let day = now.getDate() // 1-31  
let dow = now.getDay() // 0=Sunday 6=Saturday  
let month = now.getMonth() // 0=January  
let time = now.getTime() // ms since Jan 1, 1970  
  
now.setDate(1);  
now.setMonth(0); // First day/month of year  
  
now.toDateString(); // 'Tue Mon 16 2020'  
  
let ts = now.getTime(); // 1584367882000
```

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

# Date handling

- Comparing dates
  - Compare timestamp in ms
  - Potentially resetting some date fields (time, in case comparison is about date only)
- Date difference
  - Convert to timestamp, then handle accordingly to get the desired number of days, hours, minutes etc. needed

```
const diff = date2.getTime() - date1.getTime()  
// in ms
```

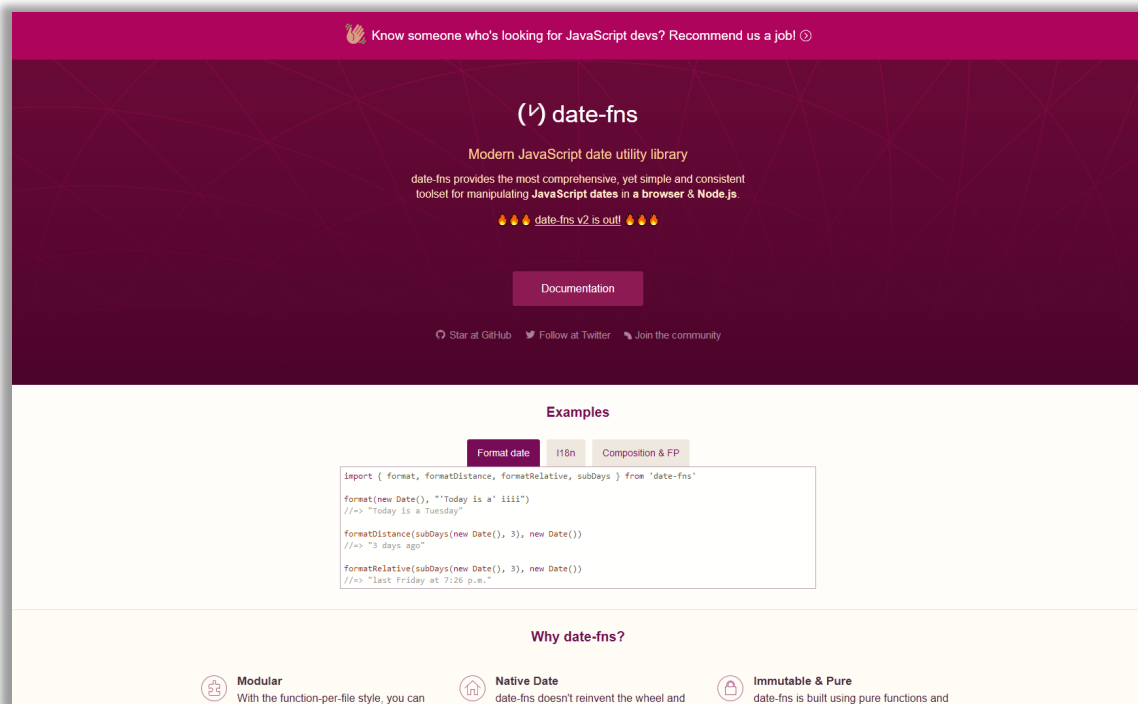
```
if (date2.getTime() === date1.getTime()) {  
    //dates (including times) are equal  
}
```

Mar 16, 2020 12:45:23 is **not** equal to new Mar 16, 2020.  
Use `setHours(0, 0, 0, 0)` to reset the time.

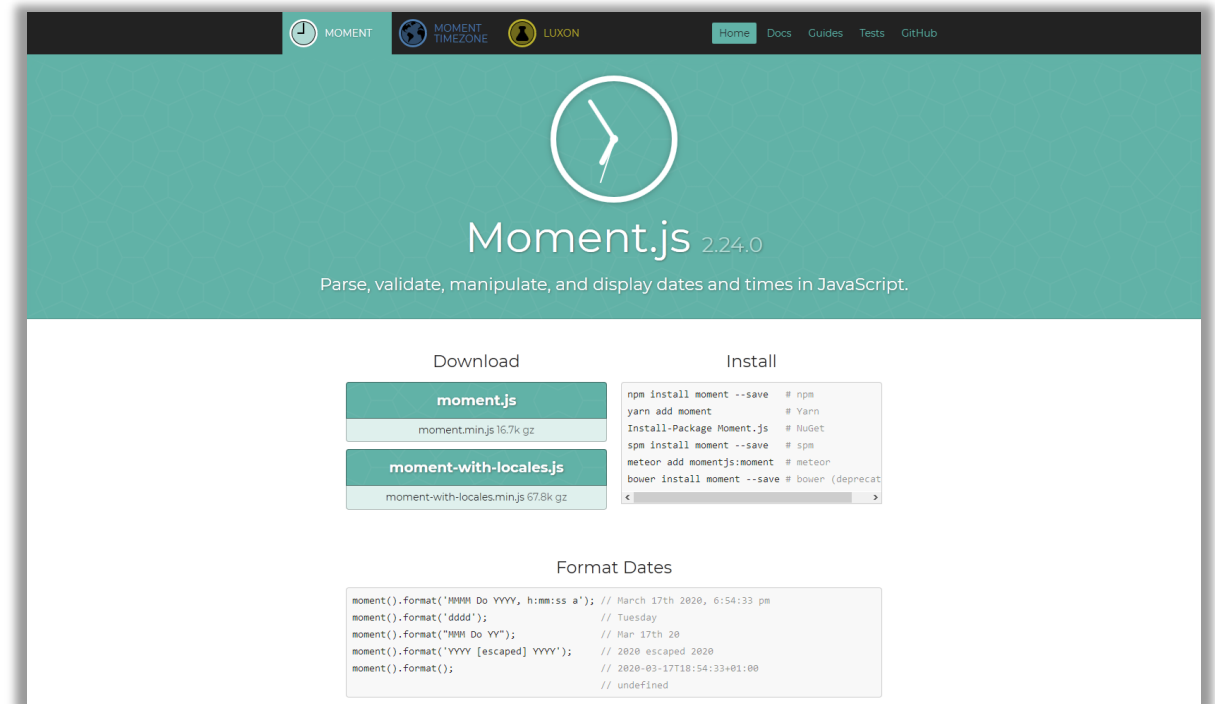
```
let d1 = new Date(); // assume Mar 16, 2020  
let d2 = new Date("Jan 1, 2020");  
let diff = d1 - d2;  
const MS_DAY = 1000*60*60*24;  
const MS_H = 1000*60*60;  
let days = Math.floor(diff/MS_DAY); // 75  
let mins = Math.floor((diff-days*MS_DAY)/MS_H);
```

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

# Serious JS date/time handling libraries



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



<https://momentjs.com/>

# 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/>

