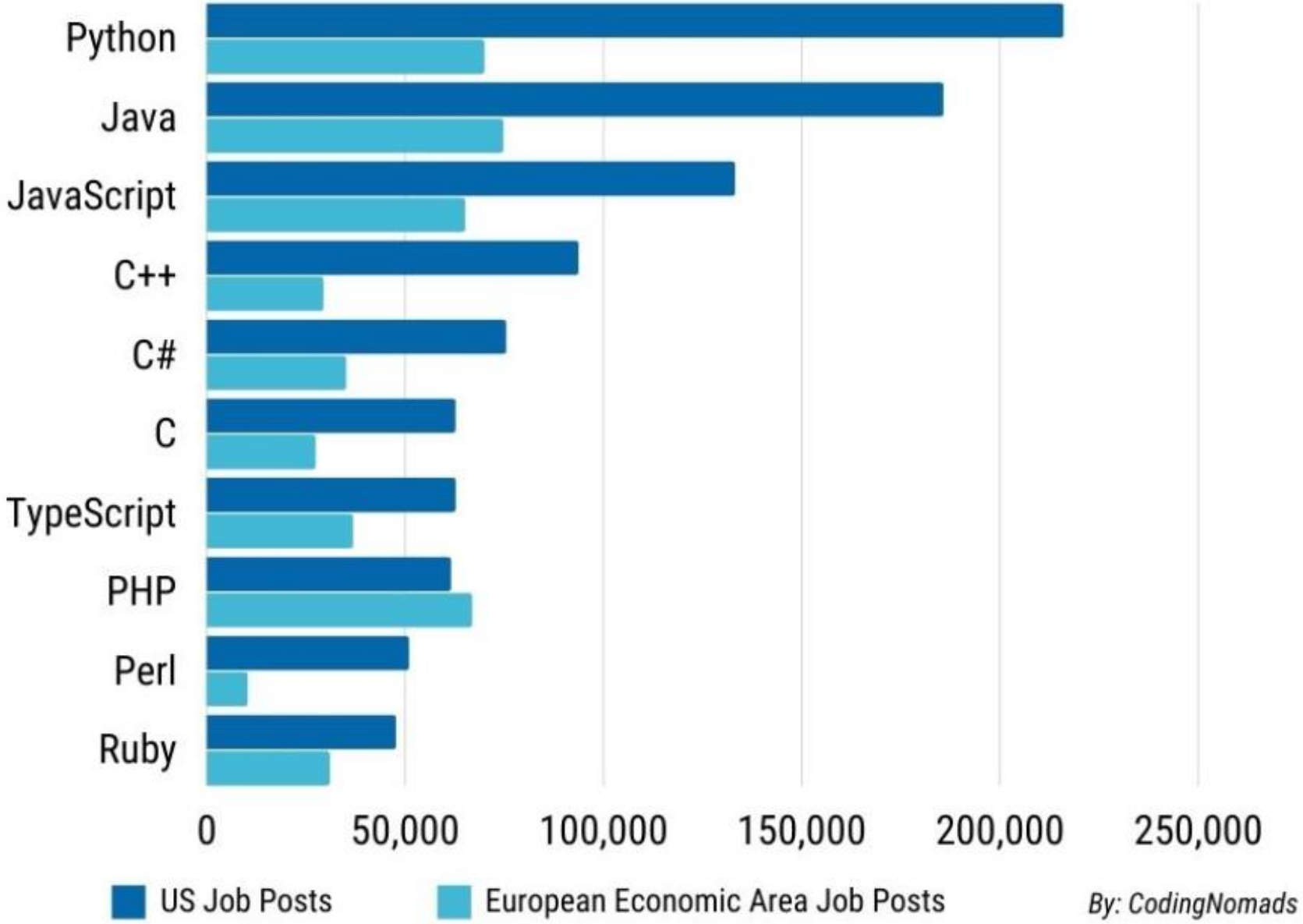
# JavaScript

di Roberta Molinari

# Most in-demand programming languages of 2022

*Based on LinkedIn job postings in the USA & Europe*



# JavaScript

- È stato creato nel 1995 da Brendan Eich in soli 10 giorni, come parte del browser *Netscape Navigator 2.0*.
- Il linguaggio di programmazione inizialmente è stato prima chiamato *Mocha*, poi *LiveScript* e infine *JavaScript* in onore dell'azienda con cui collabora per la sua realizzazione (la Sun Microsistems, dal 2010 acquistata dalla Oracle, che stava creando Java).
- La sua prima versione standardizzata è l'ECMA Script 1 (ES1) del 1997.


13

# JavaScript

JavaScript è il primo linguaggio di scripting del web. Le sue caratteristiche:

- **interpretato,**
- **object based basato su prototipi**,
- single-thread, non bloccante
- multi-paradigma, (orientato agli oggetti, imperativo e dichiarativo-funzionale)
- **guidato dagli eventi**.

**Linguaggio di scripting:** linguaggio interpretato destinato in genere a compiti di automazione del sistema operativo (*batch*) o delle applicazioni (*macro*), o a essere usato nelle pagine web


14

# JavaScript utilizzi

1. **Sviluppo web**. È utilizzato principalmente per la creazione di interattività e dinamicità all'interno delle pagine web: animazioni, effetti grafici, menu a discesa, form di validazione …
2. **Applicazioni web**. Con l'aiuto di framework JavaScript come Angular, React, Vue.js e molti altri, si possono creare applicazioni web altamente dinamiche e interattive.
3. **Applicazioni desktop**. Grazie a tecnologie come Electron, può essere utilizzato anche per la creazione di applicazioni desktop.
4. **Applicazioni mobile**. Con tecnologie come React Native e Ionic, gli sviluppatori possono creare applicazioni mobile cross-platform utilizzando JavaScript ibride e native.
5. **Server-side**: utilizzando Node.js si possono creare applicazioni server side altamente scalabili e performanti utilizzando un unico linguaggio di programmazione.

# JavaScript storia

- 1995-2000: Inizialmente molto semplice e limitato, nasce nel 1995 nel browser Navigator 2.0 della Netscape come "LiveScript". Ribattezzato JavaScript in onore dell'azienda con cui si collabora per la sua realizzazione (la Sun Microsistems, dal 2010 acquistata dalla Oracle, che stava creando Java). Microsoft realizza un linguaggio simile nel 1996 chiamandolo Jscript: oggi abbandonato. Netscape e Sun standardizzarono il linguaggio nel (1997 ES1)
- 2000-2005: JavaScript è diventato sempre più popolare e ha iniziato a essere utilizzato per creare applicazioni web più complesse. Sono state introdotte nuove librerie e framework, come jQuery e Prototype, che hanno semplificato la creazione di interattività.


16

# JavaScript storia

- 2006-2010: Nel 2006, è stato introdotto il framework JavaScript moderno AngularJS, che ha reso più facile creare applicazioni web complesse. Inoltre, sono stati introdotti anche nuovi motori JavaScript come V8, che ha migliorato notevolmente le prestazioni di JavaScript. Nel 2009 nasce Node.js ambiente di runtime JavaScript open-source basato su Chrome V8 engine, che consente di eseguire codice JavaScript lato server.
- 2010-2014: è stato introdotto il framework ReactJS, che ha semplificato la creazione di interfacce utente complesse.

# JavaScript storia

- 2015-2020: **Nel 2015, è stato introdotto il nuovo standard ECMAScript 6 (ES6), che ha introdotto nuove funzionalità avanzate come le classi, le arrow function, const e let.** Ha anche introdotto il concetto di "**JavaScript asincrono**" con l'introduzione delle **Promises**, rendendo più facile la gestione di richieste asincrone (che dalla versione del 2017 si potranno gestire anche come **async/await**). Questa versione ha reso JavaScript più facile da leggere e scrivere.

Sono stati introdotti anche framework come Vue.js e React Native, che hanno semplificato la creazione di applicazioni mobili.

# JavaScript storia

- 2020-oggi: Negli ultimi anni, sono stati introdotti nuovi concetti come TypeScript, un linguaggio di programmazione basato su JavaScript che fornisce un sistema di tipi statici. Sono stati anche introdotti nuovi framework come AngularJS 2 e ReactJS Hooks, che semplificano la creazione di applicazioni web avanzate.

# JavaScript compatibilità

## Browser web [ modifica | modifica wikitesto ]

| Engine | Browser | Conformità | | |
|---|---|---|---|---|
| | | ES5[38] | ES6[39] | 2016+[40][41] |
| V8 | Microsoft Edge 80 | 100% | 98% | 100% |
| SpiderMonkey | Firefox 73 | 100% | 98% | 82% |
| V8 | Google Chrome 80, Opera 67 | 100% | 98% | 100% |
| JavaScriptCore (Nitro) | Safari 13 | 99% | 99% | 80% |

## Implementazioni lato server [ modifica | modifica wikitesto ]

| Engine | Server | Conformità | | |
|---|---|---|---|---|
| | | ES5[38] | ES6[39] | 2016+[40][41] |
| V8 | Node.js 13.2 | 100% | 98% | 92% |

# JavaScript
## Caratteristiche

- Il browser visualizza il documento HTML e **interpreta** (esegue) le eventuali istruzioni scritte in Javascript.
- È **guidato dagli eventi**: se l'utente genera un evento e esiste del codice JavaScript associato a quell'evento (event handler), questo viene eseguito
- Fornisce i costrutti di un linguaggio di programmazione: variabili, espressioni, istruzioni, …
- Ricorda C e Java, oggi è un linguaggio completo dotato di numerose librerie con nulla da invidiare ad altri linguaggi ad alto livello


27

# JavaScript
## Cosa serve

- Un editor di testo + motore che interpreta JavaScript lato client (es. V8 della Google, in Chrome, Chakra di Microsoft, SpiderMonkey di Mozilla su Firefox)
  - Per debbuggare si può usare *"Analizza elemento"→Debugger* in Firefox o F12 in Chrome


28

# JavaScript
## Come inserire uno script

1. Inserendo direttamente nel documento HTML in un qualunque punto dell'head (non potrà fare riferimento ad oggetti del <body> perché non sono ancora stati definiti) o del body

```
<script [type="text/javascript"]>
    codice JavaScript
</script>
```

Il browser interpreta il codice quando lo incontra e lo esegue. Se c'è un errore può:

- visualizzare il documento, ma non eseguire il codice errato
- visualizzare il documento parzialmente o in bianco perché l'esecuzione dall'alto al basso è interrotta



29

# JavaScript
## Come inserire uno script

2. Caricandolo da un file di testo esterno con estensione .js

```
<script [type="text/javascript"]
   src="myfile.js"></script>
```

- Lo script viene eseguito dopo aver trasferito il codice esattamente nel punto in cui è inserito il tag.
  Se non deve essere eseguito prima del caricamento del documento, si mette in fondo prima del </body> per non rallentarlo
- Il file .js non deve contenere tag HTML o elementi di altri linguaggi
- È il metodo consigliato se si vuole:
  - proteggere il codice sorgente
  - riutilizzare il codice in più documenti


30

# JavaScript
## Come inserire uno script

3. Incorporandolo all'interno dei tag HTML come valore dei nuovi attributi che sono stati introdotti per gestire gli eventi generati dall'utente

```
<div onMouseOver ="JavaScript: codice;"
onMouseOut ="JavaScript: codice;"
onClick ="JavaScript: codice;">
Testo</div>
```

o cliccando su un link

```
<a href="JavaScript: codice
   JavaScript;"> Clicca qui
</a>
```


31

# JavaScript
## Regole sintattiche

- Ogni istruzione inizia in una nuova riga o dopo il ; Pertanto l'uso del ; è obbligatorio solo se si scrivono più istruzioni sulla stessa riga. È fortemente consigliato per evitare errori di interpretazione.
- Commenti

```
// su una riga sola ci vuole lo spazio prima
/* commento su più righe*/
```

- Gli identificatori possono iniziare con _ $ o con una lettera e non possono iniziare con un numero, non possono contenere spazi o caratteri di punteggiatura
- Non si possono usare le parole riservate
- È case sensitive


34

# JavaScript
## Tipizzazione

- Tipizzazione: assegnazione del tipo alla variabile
  - Static: durante la compilazione
  - Dynamic: durante l'esecuzione
    - dinamica forte, i valori assegnati alle variabili hanno dei tipi ben definiti
    - dinamica debole, le variabili possono riferirsi a valori di qualsiasi tipo, che possono cambiare dinamicamente in seguito a manipolazioni esterne.
- Javascript è debolmente tipizzato: le variabili non sono tipate, possono valere qualunque cosa, anche essere una funzione


35

# JavaScript
## Tipizzazione

- JavaScript è un linguaggio "debolmente tipato" quindi una variabile può cambiare tipo nel corso del suo ciclo di vita. Il tipo delle variabili è stabilito in fase di esecuzione (dinamic typing), dipende dall'ultima operazione di assegnamento eseguita

```
x = 10;
x = "s";        //prima x è un numero poi stringa
```

- I tipi attualmente (ECMAScript 2022) sono
  - 7 tipi primitivi
  - 1 tipo `Object`


36

# JavaScript
## typeof() o typeof

La definizione stabilisce il nome della variabile, mentre il tipo dipende dall'ultima assegnazione.

Per verificare il tipo della variabile in un dato momento si usa `typeof()o typeof`, che può restituire la stringa del tipo corrispondente.

È *object* qualunque tipo non primitivo.

```
x = 10;                    //typeof(x)→ "number"
x = "a";                   //typeof(x) → "string"
x = new String();          //typeof(x )→ "object"
typeof false;              // → "boolean"
typeof ({name: 'John', età: 34}); //→ "object"
typeof (null) //→ "object"
```

# JavaScript
## Tipizzazione: 7 tipi primitivi

I tipi "primitivi" hanno degli oggetti wrapper in cui vengono convertiti automaticamente quando si utilizza un suo metodo o una sua proprietà.

```
s = "ciao".toUpperCase();
```

| **Type** | **typeof return value** | **Object wrapper** |
|---|---|---|
| Null | "object" | N/A |
| Undefined | "undefined" | N/A |
| Boolean | "boolean" | Boolean |
| Number solo double a 64 bit | "number" | Number |
| BigInt ECMAScript 2020 | "bigint" | BigInt |
| String | "string" | String |
| Symbol | "symbol" | Symbol |

# JavaScript
## Variabili: tipi

- Se una variabile è dichiarata con `new`, allora è un oggetto wrapper corrispondente al tipo primitivo

```
x = new Number(1.2);
y = new Boolean(true);
s = new String("ciao");
```

`typeof()` restituisce sempre `object`

# JavaScript
## Null, Undefined

- **undefined**: è il valore di una variabile dichiarata, ma non inizializzata. Il tipo non è definito, per cui si considera che sia NaN

```
let x;              //Il valore è undefined
typeof(x)           //undefined
isNaN (x)           //true
```

- Le variabili definite possono essere svuotate del loro valore impostando il loro valore a **null** (indica che il valore è sconosciuto). Sono trasformate in oggetto. Significa vuoto, sconosciuto e NON puntatore nullo.

```
let y = 10;
y = null;           // ora svuoto il valore è null
typeof(y)           // object
```


40

# JavaScript
## Null, Undefined

- ***null***: indica che alla variabile non è stato assegnato deliberatamente un valore.
- ***undefined***: indica una variabile non inizializzata, cioè a cui non è stato ancora assegnato un valore e quindi non ha un tipo definito.

```
null == undefined    //true uguali come VALORE
null === undefined    //false diversi come TIPO
```

# JavaScript NaN

- **NaN**: è un numero (typeof() restituisce number) con il significato "non è un numero", che viene assegnato quando si fa un'operazione numerica che non restituiscono un numero

```
let x = 100 / "A"; // x = NaN (Not a Number)
```

Non si verifica con x == NaN ma con **`isNaN`**`(x)`

- Attenzione!
  - NaN == undefined //false
  - isNaN(undefined) //true
  - isNaN(null) //false
  - NaN == null //false
  - NaN === undefined === null //false

# JavaScript
## Infinity

- **Infinity**: è un numero (typeof() restituisce number) che viene assegnato quando si deve assegnare un valore superiore al massimo memorizzabile o inferiore al minimo (-Infinity)

```
let x =  2 / 0;          // Infinity
let y = -2 / 0;          // -Infinity
```

Si verifica con x==Infinity o con `isFinite(x)`

Infinity > 1000 // true

# JavaScript Boolean

- Un valore **falsy** (falseggiante) è qualcosa che restituisce FALSE. È considerato falsy:
  - false
  - 0 and -0 "" and '' (empty strings)
  - null (vuoto, sconosciuto e NON puntatore nullo)
  - undefined(valore e quindi tipo non assegnato)
  - NaN (Not a Number)

Attenzione! null, undefined e NaN son considerati falsy, ma non sono == false

```
if (!NaN) console.log("è un numero")
```

# JavaScript
## Tipi stringa

- Per le stringhe si può usare:
  - indifferentemente ' o "
  - ` Backticks (ALT+96) ci permettono di incorporare variabili ed espressioni in una stringa inserendole in $ {...}

```
let name = "John";
alert(`Hello, ${name}!`); //Hello, John!
alert(`the result is ${1 + 2}`);
  // the result is 3
```

- NON esiste il tipo char
- Per interpretare in modo corretto i caratteri "," e \ in una stringa, usare \" o \' o \\


45

# JavaScript
## Conversione tra tipi: implicita

- Operatore +: *se almeno uno dei due operandi è una stringa, allora viene effettuata una concatenazione di stringhe, altrimenti viene eseguita una addizione.*

```
x = 10 + "3"        //x="103"
x = true + null     //x=1
```

i valori *true* e *null* vengono convertiti in numeri

- Se l'operatore è un numerico si converte in numero

```
x = 10 * "3"        //x=30
```

- Per gli operatori relazionali >, >=, < e <=: *se nessuno dei due operandi è un numero, allora viene eseguito un confronto tra stringhe, altrimenti viene eseguito un confronto tra numeri.*

# JavaScript
## Casting implicito

- Conversione di tipo implicita

`x = 2 + "3"` x è stringa = "23", basta una stringa

`x ="2" * "3"` x è un numero = 6, l'operatore converte

`x = 2 * "3A"` non è possibile convertire in numero la stringa quindi x = NaN


47

# JavaScript
## Conversione tra tipi: implicita

- Da tipo →NUMBER

| Tipo | Valore numerico |
|---|---|
| **undefined** | NaN se x è undefined e faccio x+4 ottengo NaN |
| **null** | 0 se x=null e faccio x+4 ottengo 4 |
| **booleano** | 1 se true, 0 se false |
| **stringa** | intero, decimale, zero o NaN in base alla specifica stringa |

- Da tipo →BOOLEAN

| Tipo | Valore booleano |
|---|---|
| **undefined** | false |
| **null** | false |
| **numero** | false se 0 o NaN,<br>true in tutti gli altri casi |
| **stringa** | false se stringa vuota,<br>true in tutti gli altri casi |

# JavaScript
## Conversione tra tipi: implicita

- Da tipo → STRING

| Tipo | Valore stringa |
|---|---|
| **undefined** | "undefined" |
| **null** | "null" |
| **booleano** | "true" se true<br>"false" se false |
| **numero** | "NaN" se NaN,<br>"Infinity" se Infinity<br>la stringa che rappresenta il numero negli altri casi |


49

# Qual è il risultato delle espressioni seguenti?

```
"" + 1 + 0
"" - 1 + 0
true + false
6 / "3"
"2" * "3"
4 + 5 + "px"
"$" + 4 + 5
"4" - 2
"4px" - 2
7 / 0
" -9 " + 5
" -9 " - 5
null + 1
undefined + 1
```



50

# JavaScript
## == e ===

- Per == e != vale: *se entrambi gli operatori sono stringhe allora viene effettuato un confronto tra stringhe, altrimenti si esegue un confronto tra numeri; unica eccezione è*

```
null == undefined
```

  *che è vera per definizione*

- operatori di uguaglianza e disuguaglianza stretta (=== e!==). Questi operatori confrontano gli operandi senza effettuare alcuna conversione. Quindi *due espressioni vengono considerate uguali soltanto se sono dello stesso tipo ed rappresentano effettivamente lo stesso valore*. Nel caso di due oggetti restituisce sempre false a meno che puntino alla stessa area


51

# JavaScript
## == e ===

- Attenzione al confronto tra tipi elementari e classi wrapper corrispondenti

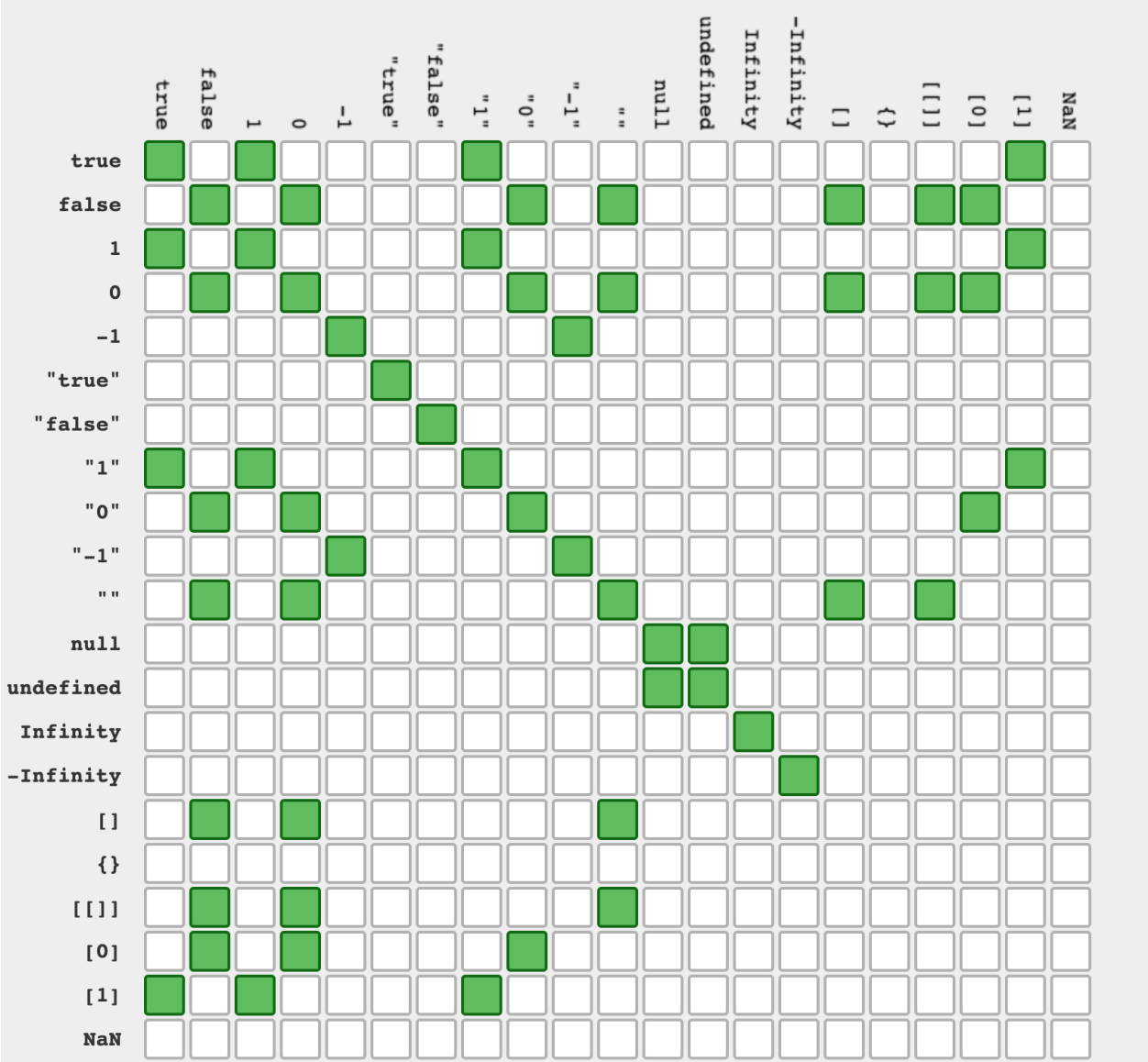
```
"ciao" == new String("ciao")  // true
"ciao" === new String("ciao") // false

3 == new Number(3) // true
3 === new Number(3) // false
```


52

# JavaScript
## == e ===

```
1==true     //T
1===true    //F
12=='12'    //T
12==='12'   //F
""==false   //T
""===false  //F
0==""       //T
0===""      //F
null ==
undefined   //T
null ===
undefined   //F
```



# JavaScript
## Conversione tra tipi: esplicita

```
s= String(10);                    //s="10"
n= parseInt("10 gradi");          //n=10
n= parseInt("11 gradi",2);        //n=11₂ ovvero 3
```

//il secondo parametro rappresenta la base numerica

La funzione `eval()` prende come argomento una stringa e la valuta o la esegue come se fosse codice JavaScript

```
x='a';y='ab';z=(eval('x<y'));          //z=true
s1= "2+2"; z=eval(s1);                 //z=4
s2= new String("2+2"); z=eval(s2);     //z="2+2"
x=eval(s2.valueOf());                  //x=4
x=10; y=20; z=eval("x+y+40")           //z=70
```


54

# JavaScript
## Casting esplicito

- Conversione di tipo esplicita

```
VarStr=String(VarNum);
```
da numero a stringa

```
VarNum=parseInt(VarStr[,BaseNumerica]);
```
da stringa a numero senza decimale (basta che la stringa inizi con un numero),

```
VarNum=parseFloat(VarStr);
```
da stringa a float

```
VarNum=eval(VarStr);
```
cerca di convertire in numerico una qualunque espressione (tipo "2+2") se non ci riesce restituisce `NaN`

```
VarNum=Number(VarStr);//restituisce un numero
                      NON un oggetto


parseInt("39 gradi") //39
Number ("39 gradi") //NaN
```


55

# JavaScript
## Variabili

- Prima di essere adoperata una variabile può essere dichiarata (non necessario, il controllo è lasco)

  `[var|let] nome[=valore];` //dichiarazione con eventuale inizializzazione. Se non si assegna un valore la variabile è "undefined"

  `let foo, boo=true;` //inizializzata a true solo boo

  `let x=y=z=3;` //inizializzate tutte a 3, ma solo x ha let

  `let a="il",b='lo';` //sono due stringhe

- Una variabile non dichiarata viene automaticamente creata al primo utilizzo ed è accessibile da qualsiasi punto di uno script (è globale). L'utilizzo di `var|let` dà la possibilità di stabilire lo scope


56

# JavaScript
## Variabili: scope

Le variabili possono essere:

- **globali**: sono accessibili da qualsiasi punto dello script. Sono:
  - dichiarate senza `var|let` in un qualunque punto, anche dentro ad una funzione
  - dichiarate con `var|let` fuori da qualsiasi funzione.
- **locali**: dichiarate con `var|let` all'interno di una funzione e sono accessibili soltanto all'interno del suo corpo; occupano spazio solo per il tempo di esecuzione della funzione. Con `let` si posso dichiarare anche a livello di blocco.


57

# JavaScript
## Variabili var: hoisting(sollevamento)

Le dichiarazioni delle variabili vengono elaborate prima dell'esecuzione di qualsiasi codice. Quindi una variabile può essere usata prima che sia dichiarata. Viene fatto l'**hoisting** la dichiarazione della variabile "viene spostata" all'inizio del suo ambito, ma il valore verrà effettivamente assegnato al raggiungimento dell'istruzione. Pertanto si raccomanda di dichiarare sempre le variabili all'inizio del loro ambito.

```
console.log (greeter);
var greeter = "say hello"
```

è interpretato come

```
var greeter;
console.log(greeter); // greeter is undefined
greeter = "say hello"
```

# JavaScript
## Variabili var: scope

Dichiarare una variabile all'interno di un blocco di codice {} con **`var`** NON crea un nuovo scope per la variabile dichiarata (come in C), resta visibile anche al di fuori del blocco.

È possibile dichiarare due volte la stessa variabile con var all'interno dello stesso blocco

Con `var` la variabile avrà scope di funzione e sarà utilizzabile (nel corpo della funzione) anche nelle righe che precedono la dichiarazione stessa (hoisting)


59

# JavaScript
## Variabili var: scope

```
var x = 0;   // x globale
alert(typeof z); // undefined, z non esiste ancora
function a() {
  var y = 2;    // y ha scope di funzione
  alert(x +" "+ y);    // 0 2
  b();      // chiamando b() si crea z globale
  function b() {//genera ReferenceError con strict mode)
    x = 3;  //assegna 3 alla var globale x
    y = 4;  //assegna alla y di a() non crea una var globale
    z = 5;  //crea una var globale
  }
  alert(x +" "+ y +" "+ z);  // 3 4 5
}
a();                    //chiama a() e quindi anche b()
alert(x +" "+ z);       // 3 5
alert(typeof y); // undefined perchè y ha scope di funzione
```

# JavaScript
## Variabili let

A differenza di `var` che è inizializzata come undefined, la variabile `let` non è inizializzata.

Quindi se provi a usare una variabile `let` prima della sua dichiarazione, otterrai un errore `ReferenceError` (non viene fatto l'hoisting)

```
console.log (saluto);
let saluto = "say hello"
//ReferenceError: can't access lexical declaration 'saluto' before initialization
```

Dal 2015 JavaScript si usa **`let + strict mode`** (stile moderno)

# JavaScript
## Variabili let: scope

**`let`**, definita dalle specifiche di ECMAScript 6 (2015), permette di dichiarare variabili limitandone la visibilità ad un blocco di codice, ad un assegnazione, ad un'espressione in cui è usata

Inoltre una variabile dichiarata con let non sarà soggetta all'hoisting e NON è possibile dichiarare due volte la stessa variabile con let/var all'interno dello stesso blocco

```
var x = 10;//anche let
var y;
{
    let  x= 20  //nasconde quella esterna
     y = x + 1;  //21
}
y = x + y;              //10+21
```

# JavaScript
## Variabili let: scope

```
let x = 10;
if (true) {
  let y = 20;
  var z = 30;
  console.log(x + y + z);  // → 60
}
// y is not visible here
console.log(x + z);// → 40
```

# JavaScript
## Costanti: primitivi

- Dallo standard ES6 è possibile dichiarare "costanti"

```
const PI=3.14;
```

- NON definisce un valore costante. Definisce un riferimento costante a un valore. Per questo motivo non possiamo cambiare i valori costanti primitivi

```
PI = PI + 10; // ERROR
```

- Ha scope a livello di blocco come `let`

```
var x = 10; // Here x is 10
  {
    const x = 2;  // Here x is 2
  }
// Here x is 10
```

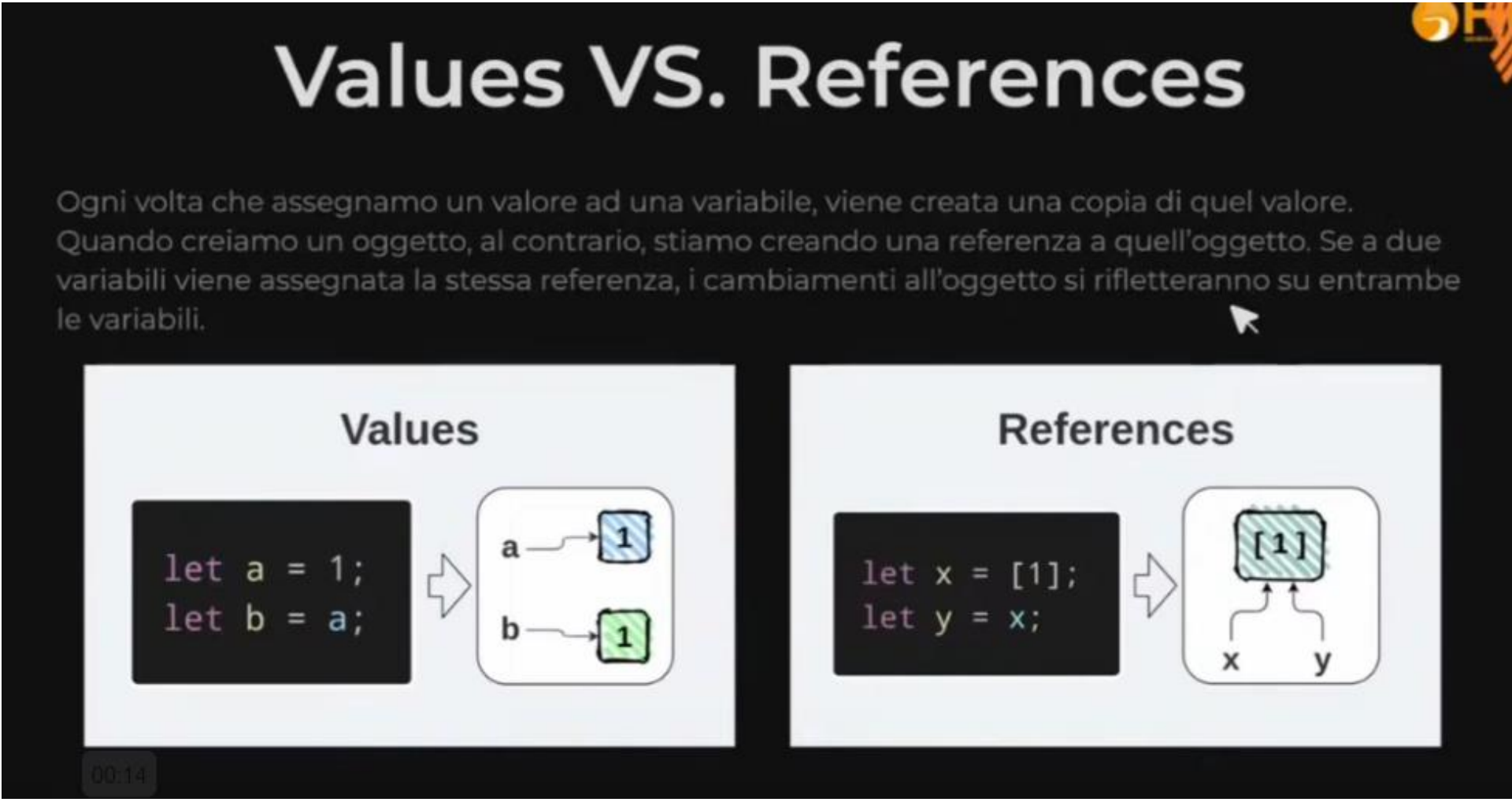
# JavaScript
## Costanti: ridichiarazione

- Non è possibile la ridichiarazione

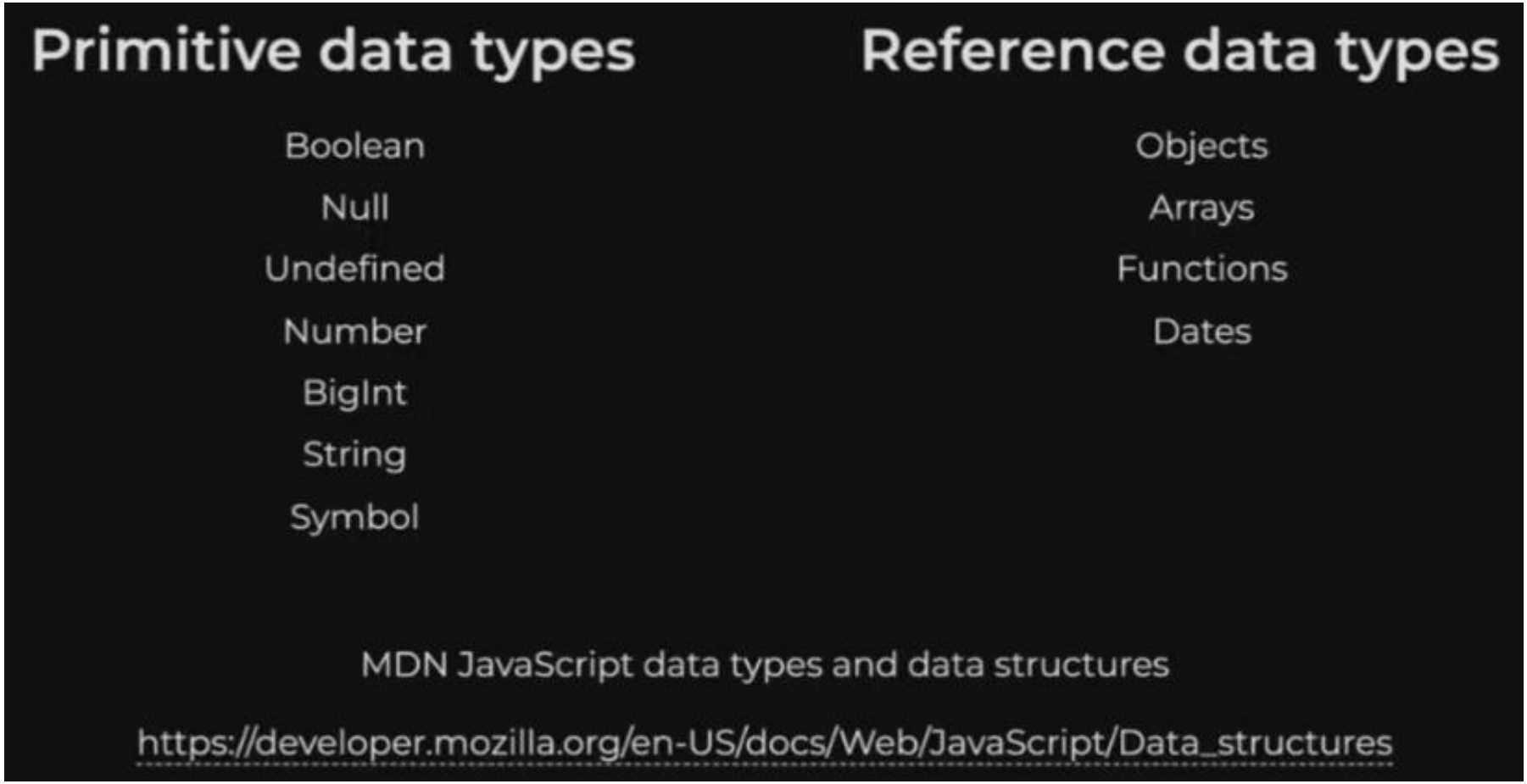
```
var x = 2;          // Allowed
const x = 2;        // Not allowed
  {
    let x = 2;      // Allowed
    const x = 2;    // Not allowed
  }
```


66

# JavaScript
## tipi primitivi e tipi referenza



# JavaScript
## tipi primitivi e tipi referenza



# JavaScript
## tipi primitivi e tipi referenza

```
// Primitives
let a = 2;  let b = a;
console.log(a, b)  //2,2
b = 3;
console.log(a, b)  //2,3

// Objects
const oggettoA = { valore: 2 };
const oggettoB = oggettoA;
console.log(oggettoA.valore, oggettoB.valore) //2,2
oggettoB.valore = 3;
console.log(oggettoA.valore, oggettoB.valore) //3,3
```

# JavaScript
## Costanti: oggetto

- NON definisce un valore costante. Per questo motivo, possiamo cambiare/aggiungere proprietà agli oggetti costanti, ma non riassegnarli.

```
const car = {model:"500",color:"white"};

// You can change/add a property:
car.color="red";
car.owner="John";

car = {model:"600",color:"red"};// ERROR
```


71

# JavaScript
## Costanti: array

- NON definisce un valore costante. Per questo motivo, possiamo cambiare/aggiungere elementi ad un array costante, ma non riassegnarli.

```
const cars = ["Saab", "Volvo", "BMW"];

// You can change/add an element:
cars[0] = "Toyota";
cars.push("Audi");


cars =["Fiat", "Volvo", "Audi"]; //ERROR
```


72

# JavaScript
## Costanti: funzioni

```
const halve = function(n) {
  return n / 2;
};

let n = 10;
console.log(halve(100));   // → 50
console.log(n);    // → 10
```


73

# JavaScript
## strict mode

Per rendere obbligatorio la dichiarazione delle variabili (consigliato, per supporto al controllo della correttezza del programma, tipo non modificare variabili omonime dichiarate altrove) si può inserire all'inizio del codice la stringa:

```
"use strict";
```

- Da un punto di vista sintattico, non si tratta di una istruzione vera e propria, ma di una stringa. Questo garantisce la compatibilità con le versioni precedenti (< standard 5) del linguaggio che semplicemente la ignoreranno senza generare errori.
- Una volta abilitato lo strict mode, ad ogni assenza di dichiarazione di variabili verrà segnalato un errore.


74

# JavaScript
## Funzioni

- Funzioni si possono dichiarare così (non si dichiara il tipo restituito, avrà uno scope)

```
function nome ([par1,…,parN]){
        [var local1,…]
  …
  [return (espressione);]    //anche oggetti
}
```

  - Si richiamera così: `nome()`

– Le funzioni possono essere richiamate prima della loro definizione, per il meccanismo dell'hoisting
– Se definite in una funzione sono visibili solo dentro essa


76

# JavaScript Funzioni

- Sono tutte funzioni, le procedure non restituiscono risultato `return` è facoltativo. Se non c'è restituisce undefined
- `typeof` su una funzione restituisce "function"
- La chiamata può avvenire anche prima della dichiarazione (non per le anonime) e con un numero anche diverso di parametri (quelli mancanti saranno undefined e quelli in eccesso non saranno considerati)
- Normalmente si dichiarano nell'HEAD
- Il passaggio dei parametri relativi a tipi di dato primitivi avviene sempre per valore per gli altri tipi di oggetti avviene sempre per riferimento.


77

# JavaScript
## Funzioni: esempio di scope

```
foo()         //per l'hoisting
function foo(){
  function bar(){ //locale anche con var bar =
      console.log("Hello")}
  bar()
}
bar()         //non visibile

//foo1()     //non viene fatto l'hoisting
foo1 = function(){
  bar1 = function(){     //globale
      console.log("Hello 1")}
  bar1()
}
foo1()
bar1()              //visibile
```


78

# JavaScript
## Funzioni: scope chain

- *Gerarchia di scope* o scope chain: una funzione può accedere allo scope locale, allo scope globale ed allo scope accessibile dalla funzione in cui è stata definita (funzione esterna), il quale può essere a sua volta il risultato della combinazione del proprio scope locale con lo scope della sua funzione esterna e così via.
- in JavaScript *l'accesso allo scope della sua funzione esterna è consentito anche dopo che questa ha terminato la sua esecuzione*.


79

# JavaScript
## Funzioni: scope chain

```
var saluto = "Buongiorno";
var visualizzaSaluto;
function saluta(persona) {
    var nc = persona.nome + " " + persona.cognome;
    return function() {console.log(saluto + " " + nc); };
}
visualizzaSaluto =
  saluta({nome: "Mario", cognome: "Rossi"});
visualizzaSaluto();
```

- In questo caso la funzione `saluta()` non visualizza direttamente la stringa ma restituisce una funzione che assolve questo compito. Pertanto, quando la funzione restituita viene invocata, la funzione `saluta()` (la sua funzione esterna) ha terminato la sua esecuzione e quindi il suo contesto di esecuzione non esiste più. Nonostante ciò è ancora possibile accedere alla variabile `nc` presente nel suo scope locale.


80

# JavaScript
## Funzioni: con valori di default

- Dal ECMAScript 6 viene introdotta la possibilità di specificare dei valori di default:

```
function somma(x = 0, y = 0) {
    let z = x + y;    return z;
}
```

- Così, se al momento della chiamata non viene passato un argomento, ad esso viene assegnato il valore di default specificato, invece del valore undefined. Quindi, ad esempio, la chiamata somma() senza argomenti restituirà il valore 0 anzichè NaN.

```
somma(3) //x=3 e y=0
somma (undefined, 5) //x=0, y=5
```


82

# JavaScript
## Funzioni: con valori di default

- Senza i valori di default quando NON vengono passati dei parametri questi sono undefined

```
function myFunction(x, y) {
      if (y === undefined) {
            y = 0;
      }
      return x + y;
  }

myFuntcion(3)//3 perchè y è undefined
```


83

# JavaScript
## Funzioni: array arguments

- Si può non definire alcun argomento nella definizione di una funzione ed accedere ai valori passati in fase di chiamata tramite un array predefinito: `arguments`
- Ad esempio, possiamo sommare un numero indefinito di valori con chiamate somma(1,2) o somma(1,2,3)…

```
function somma() {
    var z = 0;     var i;
        for (i in arguments) {
        z = z + arguments[i];
    }
     return z;
}
```

# JavaScript
## Funzioni: rest parameter

```
function eseguiOperazione(x, ...y) {
      var z = 0;
      switch (x) {
          case "somma":
            for (i in y) {
                z = z + y[i];
            } break;
          case "moltiplica":
            for (i in y) {
                z = z * y[i];
            } break;
          default:  z = NaN;  break;
    }
    return z;
}
```


85

# JavaScript
## Funzioni: rest parameter

- L'argomento x che rappresenta il nome dell'operazione da eseguire e y preceduto da tre punti che rappresenta il resto dei valori da passare alla funzione, che saranno disponibili sotto forma di vettore
- L'approccio è simile all'array predefinito arguments, ma mentre questo cattura tutti gli argomenti della funzione, il rest parameter cattura soltanto gli argomenti in più rispetto a quelli specificati singolarmente.


86

# JavaScript
## Funzioni: spread operator

- La stessa notazione del rest parameter può essere utilizzata nelle chiamate a funzioni che prevedono diversi argomenti. In questo caso si parla di **spread operator**, cioè di un operatore che sparge i valori contenuti in un array sugli argomenti di una funzione, come nel seguente esempio:

```
const addendi = [8, 23, 19, 72, 3, 39];
somma(...addendi);
```

- La chiamata con lo spread operator è equivalente alla seguente chiamata:

```
somma(8, 23, 19, 72, 3, 39);
```


87

# JavaScript
## Funzioni di prima classe

- Si dice che un linguaggio di programmazione ha **funzioni di prima classe** quando le funzioni in quel linguaggio sono trattate come qualsiasi altra variabile. Ad esempio, in un tale linguaggio, una funzione può essere passata come argomento ad altre funzioni, può essere restituita da un'altra funzione e può essere assegnata come valore a una variabile.
- JavaScript ha funzioni di prima classe.


88

# JavaScript
# Espressione Funzione (Function expression)

Ad una variabile si può assegnare un'espressione funzione (con `let` avrà uno scope se no è globale).

```
[const|let] a = function ([par1,…,parN]){
      [let local1,…]
   …
    [return (espressione);]
};  //è consigliato mettere il ;
```

Si richiamerà così: `a()`

# JavaScript
## Espressione Funzione

- Ad a viene passato tutto "il codice della funzione".

```
let a = sayHi;
function sayHi() {
    console.log( "Hello" ); }

console.log( a ); //shows the function code
```

- Con var, const e let non possono essere richiamate prima della loro dichiarazione

```
sayHi("John"); // error!
let sayHi = function(name) {
    console.log( `Hello, ${name}` ); };
```

# JavaScript
## Funzioni di callback

```
function ask(question, yes, no) {
 if (confirm(question)) yes()
 else no(); }
function showOk() {
 alert( "You agreed." ); }
function showCancel() {
 alert( "You canceled." ); }

ask("Do you agree?", showOk, showCancel);
```

I parametri `showOk`, `showCancel` sono dette **funzioni di callback**, in quanto passiamo una funzione come parametro e ci aspettiamo che sarà eseguita in seguito (non al momento del passaggio dei parametri)


92

# JavaScript
## Funzione anonime

```
function ask(question, yes, no) {
      if (confirm(question)) yes()
      else no();
}
ask( "Do you agree?",
  function(){alert("You agreed.");},
  function(){alert("You canceled.");}
    );
```

Non sono richiamabili da nessuna altra parte perchè non hanno nome

# JavaScript
## Funzione freccia

```
let func =
  (arg1, arg2, ...argN) => expression;
```

- Si crea una funzione anonima che ha gli argomenti dichiarati nelle tonde e restituisce il valore dell'espressione a destra della freccia=>. Corrisponde a:

```
let func =
  function(arg1, arg2, ...argN){
              return expression; };
```

- Si usano soprattutto come funzioni di callback.
- Non possono essere usate per definire metodi di un oggetto


94

# JavaScript
## Funzione freccia

- Se ha un solo argomento le () si possono omettere

```
const double = n => n * 2;
```

- Se non ne ha si mette ()

```
const sayHi = () => console.log ("Hello!");
```

- Se contiene più di un'istruzione si usano {} e l'istruzione return

```
const sum = (a, b) => {
  let result = a + b;
  return result;
  };
console.log( sum(1, 2) ); // 3
```


95

# JavaScript
## Funzione freccia

Esempio di uso come funzioni di callback anonime

```
let age = prompt("What's your age?",18);
let welcome = (age < 18) ?
  () => alert('Hello') :
  () => alert("Greetings!");
welcome();
```


96

# JavaScript
## Funzione freccia: limiti

- Non hanno un loro this, per cui non si usano nei metodi di un oggetto

```
let skills = {
  stamina: 9,
  decreaseStamina: () => {
    this.stamina--;
  }
}
```

Darà un errore perché non si può accedere al valore "this.stamina"

Inoltre le arrow functions non possono essere utilizzate in combinazione con l'operatore "new" e quindi come costruttori.

# JavaScript

## Espressione Funzione vs freccia

### Function literals

(a.k.a. *anonymous functions, function expressions, lambda expressions*)

```
const multiply = function(a, b) {
  return  a * b;
};
```

They make certain coding patterns easier...

### Arrow functions ES2015

```
const logValue = x => {
    console.log(x);
    return x;
};

const sum = (a, b) => { return a + b; };
const mult = (a, b) => a * b; // implicit return
```

A less verbose literal function syntax...

# JavaScript
## Funzionis self-invocking

- Una funzione può dichiararsi e contemporaneamente chiamarsi se termina con ()
- Nel caso di espressioni funzione. Non si può più chiamare con foo()

```
let foo=function () {
      let x = "Hello!!";}();
```

- Nel caso di una funzione anonima

```
(function () {
      let x = "Hello!!";     // I will invoke
  myself
  })();
```

- Non si può fare con la dichiarazione classica


104

# JavaScript
## try..catch

Serve per gestire un errore. La clausola finally viene eseguita in ogni caso, anche se c'è return

```
try {
  foo()
} catch(err) {
    alert(err.message) ;
    return 10 ;
} finally {
    alert("tutto ok") ;
}
```

Per generare un errore si utilizza l'istruzione

```
throw "Errore"      //sarà assegnato a err
```

...

```
catch(err) {
    alert(err) ;   //visualizza "Errore"
```


107

# JavaScript
## Istruzioni

- Blocchi di istruzioni

```
{
  x=2;
  y=6;
}
```

- Assegnazione x=2

```
x=2;
y=(z=0);        //sia y che z varranno 0
```

- Assegnazione condizionale

```
x =((Cond) ? valVero : valFalso);
```

Esempio

```
x=((y>0)?2:3);    //se y>0,x=2 se no x=3
```


108

# JavaScript
## Istruzioni

- Operatori su stringhe (per string, null, caratteri speciali, oggetti)
  - `s=s+"ciao";` concatenazione anche in forma compatta s+="ciao"
- Caratteri speciali
- `\f` avanzamento pagina
- `\n` inizio riga
- `\t` tabulazione
- `\\` back slash
- \u00E8 `\u` seguito dal codice Unicode (nell'esempio è)
- Operatori polimorfi
  - `+ += == != =` se operano tra tipi diversi si ha prima una conversione e poi si esegue l'operazione. Si da precedenza ai tipi string


109

# JavaScript
## Istruzioni

- Operatori numerici (per int, float e bool)
  - - + * / %(MOD) ++ --
    - `x/y;` restituisce un float
    - `x = parseInt(x/y);` equivale x=x DIV y
    - `Math.floor(0.7);` arrotonda per difetto →0
    - `Math.ceil(0.2);` arrotonda per eccesso→1
    - `Math.round(0.7);` arrotonda
    - `Math.random();` restituisce un numero tra [0..1)
    - `Math.sqr();`
    - `Math.min(x,y);` restituisce il minore tra i due
    - `Math.max(x,y);` restituisce il maggiore tra i due
    - `+"2"` + unario converte in numero →2
    - `2**3` 2^3
    - Le operazioni errate sui numeri non bloccano mai l'esecuzione (assegnano NaN al risultato)


110

# JavaScript
## Istruzioni

Come in C:

```
-= *=, /= …
```

- L'assegnamento restituisce il valore assegnato

```
let a = 1;
let b = 2;
let c = 3 - (a = b + 1);
alert( a ); // 3
alert( c ); // 0
```

# JavaScript
## Istruzioni

- Operatori logici
  - `||  &&  ^  !`  //OR AND XOR NOT
- Operatori relazionali
  - `<,><= ==, !=, <=  (anche per stringhe)`
  - `===, !==` (ug. e dis stretta senza conversione di tipi: due espressioni vengono considerate uguali soltanto se sono dello stesso tipo e rappresentano effettivamente lo stesso valore.)

```
X = 5;
X == '5';   //true
X === '5' //false
```


112

# JavaScript
## Istruzioni Condizionali

- IF

```
if (cond)
    bloccoVero;
[else if
    bloccoFalso;]
[else     bloccoAltro;]
```

- Case

```
switch (espressione)
{case cost1:blocco1;
    [break;]
[case cost2:blocco2;]
    [break;]
[default: blocco;]
}
```


113

# JavaScript
## Istruzioni Condizionali

- While

```
while (cond)
  Blocco;
```

- Do while (repeat che cicla per vero)

```
do
  Blocco;
while (cond);
```

- `break;` //esce dal blocco
- `continue ;` //ricomincia il blocco di un ciclo


114

# JavaScript For

- For

```
for(esprIniz;cond;passo;)
  blocco;
```

si esegue `esprIniz`, si verifica `cond`, se è vera si esegue `blocco`. si esegue `passo` e si ricomincia verificando `cond` ...

Per lavorare più comodamente con gli array JavaScript prevede due varianti del for:

- `for...in` per gli oggetti e i vettori
- `for...of` per le collezioni come: Array, String,...


115

# JavaScript
## For ... in

- `For in` scorre le proprietà di un oggetto (in un vettore sono gli indici)

```
let quantita = [12, 34, 45, 7, 19];
let totale = 0;
for (let indice in quantita) {
  totale = totale +  quantita[indice];
}
```

Non bisogna specificare la lunghezza dell'array nè l'istruzione di modifica della condizione. JavaScript rileva che la variabile `quantita` è un array ed assegna ad ogni iterazione alla variabile `indice` il valore dell'indice corrente (è una stringa però).


116

# JavaScript
## For ... of

- `For of` scorre i valori presenti in oggetti iterabili ovvero collezioni come: Array, Map, Set, String,...

```
let quantita = [12, 34, 45, 7, 19];
let totale = 0;
for (let valore of quantita) {
    totale = totale +  valore;
}
```

- Ad ogni iterazione JavaScript assegna alla variabile valore il contenuto di ciascun elemento dell'array.
- Fa parte delle specifiche di ECMAScript 6 e potrebbe non essere disponibile in engine JavaScript meno recenti.


117

# JavaScript With

- With

```
with (Math){
    floor(random()*quanti)+aPartireDa;
}
```

Attenzione non usare dentro il `with (Math)` variabili con lo stesso nome delle funzioni di `Math`(es min, max,…) se no non funziona

# JavaScript
## Istruzioni di Input

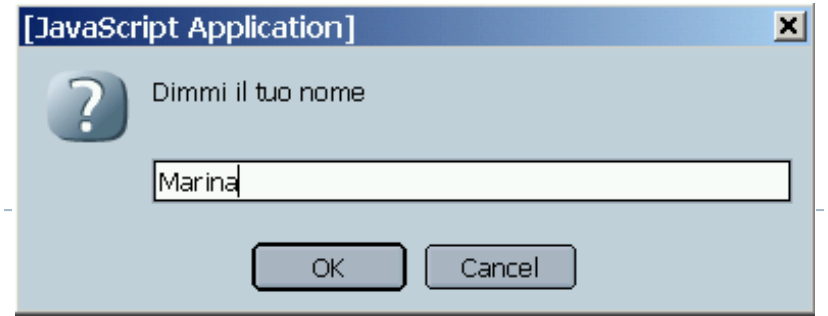
Finestre di dialogo: **possono essere** modali **(bloccanti) se non permettono la prosecuzione dell'esecuzione del resto del codice fino alla loro chiusura, oppure** non modali **(non bloccanti)**

```
x=window.prompt("Dimmi il tuo nome","");
```

Anche solo `prompt`(), il secondo parametro è il valore di default. Apre una finestra modale. Restituisce:

- se si introduce qualcosa e si preme OK, restituisce sempre una stringa anche se si introducono numeri
- `null` se preme ANNULLA
- `striga vuota` se non si inserisce nulla e non c'è un valore di default

Si può quindi valutare var==undefined oppure ==null



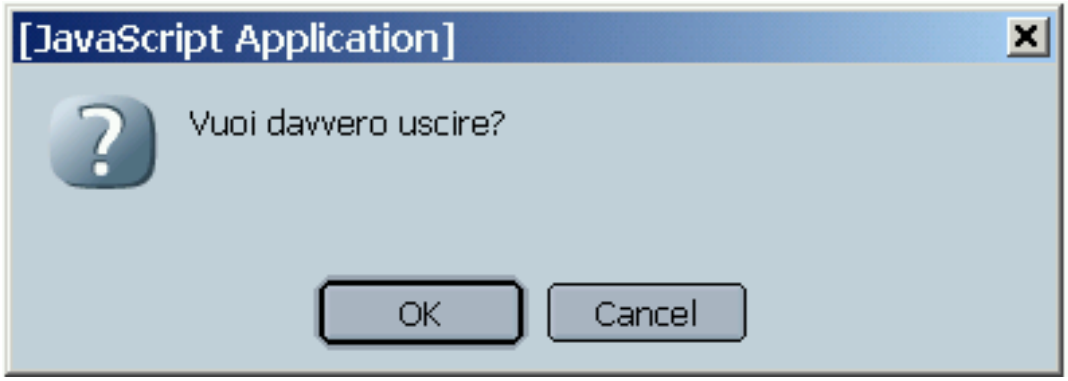


119

# JavaScript
## Istruzioni di Input

```
window.confirm("Vuoi davvero uscire?")
```

apre una finestra modale che restituisce true (OK) o false (CANCEL)





120

# JavaScript
## Istruzioni di Output

```
window.alert("Hello world");
```

apre una finestra modale che restituisce undefined

Anche solo alert()

```
window.status ("Hello world");
```

viene scritto nella barra di stato

```
console.log("Hello world");
```

viene scritto a console durante il debug

# JavaScript
## Istruzioni di Output

```
document.write("Hello world");
```

viene scritto nel flusso del documento HTML al momento dell'esecuzione

`.writeln` mette un a capo nel documento (non mette un <br>! Usa <pre>)

Con `.open()` e `.close()`

```
document.open();
document.write("Hello World");
document.close();
```