



UNIVERSITÀ DEGLI STUDI DI CAGLIARI

CORSO DI LAUREA IN INFORMATICA

Silvia Maria Massa – silviam.massa@unica.it

Fondamenti di Programmazione Web

Linguaggio JavaScript



Caratteristiche di JavaScript

- JavaScript è un linguaggio:
 - Alto livello
 - Interpretato
 - Multi-paradigma

Pagina statica che mostra informazioni in un layout fisso

JavaScript

Aggiungere HTML alla pagina, cambiare il contenuto esistente, modificare lo stile.

Reagire alle azioni dell'utente, click del mouse, movimenti del cursore, input da tastiera.

Inviare richieste al server tramite la rete.

- Nasce come tecnologia lato client per rendere più dinamiche le applicazioni Web
- Oggi è possibile utilizzarlo sia per lo sviluppo lato client che lato server (Node.js)

Tag `<script>` nella pagina HTML

- Il codice JavaScript in una pagina HTML può essere aggiunto in due modi:
 1. Includendo il sorgente direttamente nel codice della pagina HTML, tramite il tag ***script***

```
<script>
```

```
...
```

```
</script>
```

- possiamo inserire il codice sia nell' head che nel body
 - l'esecuzione delle istruzioni non contenute in funzioni avviene man mano che vengono incontrate al caricamento della pagina
2. Collegando la pagina HTML a file esterni, sempre con il tag **script** ma aggiungendo l'attributo **src**

```
<script src="myScript.js"></script>
```

```
<script src="https://...../myScript.js"></script>
```

Commenti

- Come in altri linguaggi, la sintassi è simile a quella del C

// Commento su linea singola

```
/*  
    Commento su più righe
```

...

```
*/
```

Separare le istruzioni

- Nel caso si abbiano più istruzione nella stessa riga si deve utilizzare il punto e virgola per separarle
- Un punto e virgola può essere omesso nella maggior parte dei casi quando si interrompe una riga, ma non in tutti i casi
- La soluzione migliore è quindi inserire il punto e virgola sempre
- Questo è corretto:
 - `x = 10 + 1`
 - `x = 10 + 1;`
 - `x = 10 + 1; y = 7 * 4`
- Questo non è corretto, si verifica un errore:
 - `x = 10 + 1 y = 7 * 4;`
 - `alert("Hello")`
`[1, 2].forEach(alert);`

Variabili

- Per creare una variabile in JavaScript, dobbiamo utilizzare la parola chiave **let** (**const** nel caso in cui la variabile sia una costante)
- **var** non è più utilizzato
- Il primo carattere del nome delle variabili deve essere per forza una lettera, oppure i simboli **\$** o **_** (niente numeri)
- Una variabile può includere solo lettere **a-z A-Z**, numeri **0-9** o i simboli **\$** o **_**
- I nomi sono case-sensitive, scriverli in lettere minuscole o maiuscole fa differenza
 - Count count COUNT sono nomi di variabili diverse
- C'è una lista di parole riservate, che non possono essere utilizzate come nomi di variabili, perché vengono utilizzate dal linguaggio stesso
 - Per esempio, le parole **let**, **class**, **return**, **function** sono riservate

Dinamicamente tipato

- In JavaScript ci sono diversi tipi di dato, ma le variabili non sono legate ad un tipo
- Questo vuol dire che una variabile può prima contenere un valore ad esempio di tipo string e subito dopo un valore di tipo number senza che si generi un errore

```
// nessun errore
```

```
let message = "hello";
```

```
message = 123456;
```

Tipi

- **number** per numeri di qualsiasi tipo: interi o in virgola mobile
- **bigint** viene utilizzato per definire interi di lunghezza arbitraria
`const bigInt = 1234567890123456789012345678901234567890n;`
// ricordiamo n alla fine
- **string** per stringhe. Una stringa può contenere uno o più caratteri
non esiste nessun tipo **character**.
- **boolean** per true/false
- **null** è un valore speciale utilizzato per indicare il valore “nullo”, “vuoto” o “valore sconosciuto”
- **undefined** per valori non assegnati
- **object** per strutture dati più complesse
- L'operatore **typeof** ci consente di vedere quale tipo è memorizzato nella variabile

```
let age;  
typeof age; // " undefined "
```


Tipo Number

- I number possono rappresentare i valori compresi tra -2^{53} e 2^{53} oltre a **Infinity** e **NaN** (rappresenta un errore di calcolo)

```
alert( "not a number" / 2 ); // NaN
```

```
alert("6"/2); //3
```

- Esistono dei modi per rendere i numeri più semplici da leggere per chi legge il codice

- let billion = 1000000000;

- let billion = **1_000_000_000**; // gli _ vengono ignorati

- let billion = **1e9**; // equivale a fare $1 * 1000000000$

- let ms = 0.000001; // il . separa i numeri interi da quelli decimali

- let ms = **1e-6**; // equivale a fare $1 / 1000000$

- Per arrotondare i numeri:

- Math.**floor()**, arrotonda per difetto

- Math.**ceil()**, arrotonda per eccesso:

- Math.**trunc()**, rimuove tutto dopo la virgola decimale senza arrotondare

- Math.**round()**, arrotonda all'intero più vicino

Math è un oggetto contenente diversi metodi per manipolare i numeri



Tipo String

- Le stringhe possono essere racchiuse tra:
 - Apici doppi: "Ciao"
 - Apici singoli: 'Ciao'
 - Backtick: `Ciao , \${name}, sei il visitatore numero \${count+1}!` //stringhe con "funzionalità estese"
- All'interno della stringa possiamo inserire dei caratteri speciali che iniziano sempre con \

\n	Nuova linea
\', \"	Apici
\\	Backslash
\t	Tab
\uXXXX	Simbolo Unicode rappresentato da codice esadecimale XXXX in codifica UTF-16. Esempio \u00A9 – equivale a ©.
\u{X...XXXXXX} (da 1 a 6 caratteri esadecimali)	Un simbolo Unicode in codifica UTF-32. Esempio \u{1F60D} – equivale a 🥰

- La proprietà **length** contiene la lunghezza della stringa
 - `alert('FPW\n'.length);` // 4 gli spazi e i caratteri speciali vengono considerati nella lunghezza della stringa
 - `alert('Adoro FPW \u{1F60D}'.length);` // 12, i simboli Unicode in codifica UTF-32 hanno lunghezza 2
- Per ottenere il carattere in una determinata posizione si utilizzano le parentesi quadre [**pos**]

```
let str = `Fpw`;  
alert( str[0] ); // F, Il primo carattere si trova in posizione 0  
alert( str[6] ); // undefined, quando il carattere non viene trovato viene restituito undefined
```
- Non possiamo modificare i singoli caratteri di una stringa
 - `str[0] = 'f';` // errore
 - `str = 'fpw';` // non ci da errore
- Possiamo iterare sui caratteri utilizzando **for..of**

```
for (let char of "FPW") {  
    alert(char);  
}
```

- `alert('Ciao'.toUpperCase()); // CIAO` `alert('Ciao'[1].toUpperCase()); // I`
- `alert('Ciao'.toLowerCase()); // ciao` `alert('Ciao'[0].toLowerCase()); // c`
- Per cercare una sotto-stringa o verificare se è presente:
 - `stringa.indexOf(sotto-stringa, pos)`
 - `pos` rappresenta la posizione da cui si inizia a cercare la sotto-stringa
 - restituisce `-1` se non trova la sottostringa, altrimenti la posizione iniziale della sotto-stringa
 - se ci sono più occorrenze restituisce la posizione iniziale della prima
 - `stringa.includes(sotto-stringa, pos)`
 - funziona come `indexOf` ma restituire `true` se è presente la sottostringa e `false` se non è presente
 - `alert("Oggi è una bella giornata".startsWith("Oggi")); // true`
 - `alert("Oggi è una bella giornata".endsWith("bella")); // false`

- Per ottenere una sotto-stringa:

➤ `stringa.slice(start, end)`

- ritorna la parte di stringa che va da start fino a end (escluso)
- valori negativi di start/end indicano che la posizione verrà contata a partire dalla fine della stringa

```
alert( 'Ciao sono Silvia'.slice(-6, ) ); // Silvia
```

- Quando confrontiamo due stringhe ricordiamoci che **una lettera minuscola è sempre maggiore di una maiuscola**
 - `Alert('a' > 'A'); // true`

Cambio di tipo

- Nella maggior parte dei casi, operatori e funzioni convertono automaticamente il valore nel tipo corretto
 - `let age = 25; alert(age);` -> converte il valore di `age` in una stringa
 - `x = '6' / '2'; // 3` -> converte `'6'` e `'2'` in numeri
- Ci sono casi in cui è necessario convertire esplicitamente i valori
 - `String(value)`
 - `Number(value)`

undefined	NaN
null	0
true / false	1 / 0
string	Gli spazi bianchi agli estremi vengono ignorati. Una stringa vuota diventa 0 . Se la stringa non è un numero abbiamo NaN .

- `Boolean(value)`

0, null, undefined, NaN, ""	false
qualsiasi altro valore	true



Operatori

- Addizione o concatenazione di stringhe +,
 - Se uno dei due operandi è una stringa si effettua la concatenazione di stringhe
`let a = '2'+3 ;`
`// '23'`
 - Le operazioni vengono comunque eseguite da sinistra verso destra a meno che non si usino le parentesi tonde
`let a = 2 + 2 + '1' ;`
`// "41" non "221"`
`let b = '1' + 2 + 2 ;`
`// "122" non "14"`
- Sottrazione -,
- Moltiplicazione *,
- Divisione /,
- Resto %,
- Potenza **,
- Assegnazione =,

- Modifica e assegna `+=`, `-=`, `/=`, ecc.
 - $n = n + 5 \leftrightarrow n += 5$
- Incremento/decremento `++/--`
 - `let counter = 1;`

```
let a = 2 * ++counter ; // 4  
alert(counter); //2
```

```
let a = 2 * counter++ ; // 2  
alert(counter); //2
```

- Se un'espressione ha più di un operatore, l'ordine d'esecuzione viene definito dalla loro precedenza. Possiamo modificare l'ordine usando le parente tonde.

- Gli operatori di confronto `<`, `>`, `<=`, `>=`, `==`, `!=`, `===`, `!==`, restituiscono sempre un valore booleano
- Le stringhe vengono confrontate lettera per lettera seguendo l'ordine "lessicografico" (se non ci sono più lettere da confrontare la stringa più lunga è quella più grande)
 - `let confronto = 'Bee' > 'Be' ; // true`
- Quando valori di tipo differente vengono confrontati, questi vengono convertiti in numeri
- Questo non succede quando si utilizza l'**operatore di uguaglianza stretta** `===` che verifica che siano uguali sia i valori che i tipi
 - `a = 3.1415927`
`b = "3.1415927"`
`if (a == b)... // condizione vera`
`if (a === b)... // condizione falsa`

- I valori **null** e **undefined** sono **==** solo tra di loro, e a nessun altro valore
- Va prestata attenzione quando si utilizzano gli operatori di confronto come **>** o **<** con variabili che potrebbero contenere **null/undefined** -> null diventa 0 e undefined diventa NaN

Esempio

```
let confronto = 5 > null ; // true
```

```
let confronto = 5 > undefined ; //false
```

- **||** (OR), **&&** (AND), e **!** (NOT)
- La precedenza dell'operatore AND **&&** è maggiore di quella dell'OR **||**
- La precedenza del NOT **!** è la più alta fra tutti gli operatori logici

Istruzioni condizionali - if

- **If (condizione) { se vera } else { se falsa }**
- Quando vogliamo assegnare un valore ad una variabile in base ad una certa condizione possiamo utilizzare in alternativa l'**operatore ternario**
let risultato = (condizione) ? valore_se_vera : valore_se_falsa;

```
let risultato;  
if (a > 100) {  
    risultato = "a è maggiore di 100";  
} else if (a < 100) {  
    risultato = "a è minore di 100";  
} else {  
    risultato = "a è uguale a 100";  
}
```

```
let risultato = (a > 100) ? "a è maggiore di 100" :  
    (a < 100) ? "a è minore di 100" :  
    "a è uguale a 100";
```



Istruzioni condizionali - switch

- ```
switch(x) {
 case 'value1': // if (x === 'value1')
 ...
 [break]
 case 'value2': // if (x === 'value2')
 ...
 [break]
 default:
 ...
 [break]
}
```
- Il valore di x viene controllato utilizzando l'**uguaglianza stretta** con i valori dei blocchi case
- Se l'uguaglianza viene trovata, switch inizia ad eseguire il codice partendo dal corrispondente blocco case, fino al break più vicino, oppure fino alla fine dello switch
- Se non viene trovata nessuna uguaglianza allora viene eseguito il codice del blocco default (se presente)

- Sia switch che case accettano espressioni arbitrarie

## Esempio

```
let a = "1";
```

```
let b = 0;
```

```
switch (Number(a)) {
```

```
 case b + 1:
```

```
 alert("Number(a) è identico a b+1");
```

```
 break;
```

```
 default:
```

```
 alert(" Number(a) non è identico a b+1 ");
```

```
}
```






# Cicli

- **while**- la condizione viene controllata prima di ogni iterazione  
**while (condizione) {**  
    *// corpo del ciclo*  
**}**
- **do..while** - la condizione viene controllata dopo una prima iterazione  
**do {**  
    *// corpo del ciclo*  
**} while (condizione);**
- **for (;;)** - la condizione viene controllata prima di ogni iterazione; sono possibili altre condizioni all'interno del ciclo  
**for (inizio; condizione; passo) {**  
    *// corpo del ciclo*  
**}**
- La direttiva **break** permette di far terminare un ciclo in anticipo
- La direttiva **continue** permette di saltare all'iterazione successiva

- **break/continue** supportano le **etichette prima del ciclo**  
labelName: for (...) {  
 ...  
}
- Un etichetta è l'unico modo per break/continue di uscire da cicli annidati e arrivare al ciclo più esterno

Esempio

```
outer: for (let i = 0; i < 3; i++) {
 for (let j = 0; j < 3; j++) {
 let input = prompt(`Inserisci un numero`, 0);
 if (input > 100) break outer; 
 // fa qualcosa con il numero.....
 }
}
```



# Funzioni

# Dichiarazione e parametri

- Consentono a un codice di essere utilizzato più volte, evitando ripetizioni
- Dichiarazione di una funzione

```
function nome_funzione (parametri, separati, dalla, virgola) {
 // corpo della funzione
}
```

Quando una dichiarazione di funzione viene fatta all'interno di un blocco di codice sarà visibile ovunque all'interno del blocco, ma non al suo esterno.

- Invocazione di una funzione

```
nome_funzione (value1,...,valueN)
```

- I valori passati ad una funzione come **parametri vengono copiati in variabili locali (tranne per gli oggetti)**. Se la funzione cambia il valore di un parametro che gli è stato passato, il cambiamento non è visibile all'esterno

Esempio

```
function showMessage(from, text) {
 from = '*' + from + '*';
 alert(from + ': ' + text);
}
```

```
let from = "Carlo";
showMessage(from, "Hello"); // *Carlo*: Hello
alert(from); // Carlo
```

# Parametri

- Se non viene fornito alcun parametro, questo assume il valore **undefined**

## Esempio

```
function showMessage(from, text) {
 from = '*' + from + '*';
 alert(from + ': ' + text);
}
```

```
let from = "Carlo";
showMessage(from); /* questo non è un errore, viene
visualizzato il messaggio *Carlo*: undefined */
```

- Possiamo gestire questo problema impostando un **valore di default** (**metodo 1** e **metodo 2**)

```
function showMessage (from, text = 'nessun testo fornito') {
 alert(from + ": " + text);
}
```

```
function showMessage (from, text) {
 text = text || 'nessun testo fornito';
 alert(from + ": " + text);
}
```

showMessage("Carlo");  
// Carlo: nessun testo fornito

# return

- La direttiva **return** ferma l'esecuzione e restituisce un valore al codice che ha chiamato la funzione
- Ci possono essere più direttive return nella stessa funzione

Esempio

```
function checkAge(age) {
 if (age >= 18) {
 return true;
 } else {
 alert ('Non puoi accedere a questo servizio');
 return false;
 }
}
```

```
let age = prompt('How old are you?', 18);
let accesso = checkAge (age);
```

- E' anche possibile utilizzare return senza alcun valore per uscire immediatamente dalla funzione. In questo caso è come se la funzione ritornasse **undefined**.

# Variabili locali e globali

- Una **variabile locale** viene dichiarata all'interno di una funzione (o più precisamente dentro un blocco { }) ed è visibile solamente all'interno di quella funzione
- Una **variabile globale** viene dichiarata esternamente alla funzione. La funzione può accedervi e può anche modificarla.
- La variabile globale viene utilizzata solo se non ce n'è una locale che la oscura
- Utilizzo della variabile globale

Esempio:

```
let userName = 'John'; //dichiarazione variabile globale
```

```
function showMessage() {
 userName = "Bob"; //modifica della variabile globale
 let message = 'Hello, ' + userName; // Bob
 alert(message);
}
```

Meglio evitare questo approccio

```
alert(userName); // viene visualizzato il messaggio 'John'
showMessage(); // viene visualizzato il messaggio 'Hello, Bob'
alert(userName); // viene visualizzato 'Bob'
```



# Variabili locali e globali

- Oscuramento della variabile globale

Esempio:

```
let userName = 'John'; //dichiarazione variabile globale
```

```
function showMessage() {
 let userName = "Bob"; /* dichiarazione variabile locale che oscura quella
 globale */
 let message = 'Hello, ' + userName; // Bob
 alert(message);
}
```

```
alert(userName); // viene visualizzato il messaggio 'John'
showMessage(); // viene visualizzato il messaggio 'Hello, Bob'
alert(userName); // viene visualizzato 'John'
```

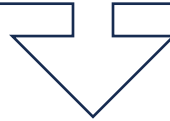
# Regole da seguire

- Per rendere il codice pulito e più facile da leggere, è consigliabile utilizzare variabili locali e parametri di funzione, non variabili esterne
- E' sempre più facile capire una funzione che accetta parametri, li utilizza per delle operazioni e ritorna un valore piuttosto di una funzione che non richiede parametri ma, come effetto collaterale, modifica variabili esterne
- Il nome della funzione dovrebbe descrivere chiaramente ciò che fa
- Una funzione è un'azione, quindi i nomi delle funzioni iniziano spesso con dei verbi . Ad esempio funzioni che iniziano con:
  - "show..." – mostrano qualcosa,
  - "get..." – ritornano un valore,
  - "calc..." – effettuano un calcolo,
  - "create..." – creano qualcosa,
  - "check..." – effettuano un controllo e ritornano un booleano, etc.
- Le funzioni dovrebbero essere brevi ed eseguire un solo compito. Se invece risultano lunghe, forse varrebbe la pena spezzarle in funzioni più piccole.

# Function expression

- Esistono dei modi alternativi per dichiarare una funzione
  - **Le function expression**

```
function func (arg1, arg2, ..., argN) {
 // corpo della funzione
}
```



```
let func = function (arg1, arg2, ..., argN) {
 //corpo della funzione
};
```

- Al contrario delle dichiarazioni di funzione, le function expression vengono create quando l'esecuzione le raggiunge e sono utilizzabili solo da quel momento in poi.
- Possono essere anonime e sono molto comode quando servono funzioni usa-e-getta.

# Arrow functions

- Le **arrow function** sono utili per azioni su una riga sola

```
function func (arg1, arg2, ..., argN) {
 // corpo della funzione
}
```

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

```
let func = (arg1, arg2, ..., argN) => {
 //corpo della funzione
};
```

## Senza parentesi graffe.

La parte destra è un'espressione: la funzione la valuta e restituisce il risultato.

## Con le parentesi graffe.

Le parentesi ci permettono di scrivere comandi multipli all'interno della funzione, ma abbiamo bisogno di dichiarare esplicitamente **return** affinché sia ritornato qualcosa.

# Callback

- Le funzioni JavaScript possono essere passate come parametri di un'altra funzione.
- La funzione passata come parametro è detta **funzione di callback**.
- L'idea è di passare una funzione e di "richiamarla" più tardi se necessario.

**Esempio** function **ask**(question, yes, no) {

    if (confirm(question)) yes()  
    else no();

}

function **showOk**() {

    alert( "You agreed." );

}

function **showCancel**() {

    alert( "You canceled the execution." );

}

**question**

Il testo della domanda

**yes**

Funzione da eseguire se la risposta è "Yes"

**no**

Funzione da eseguire se la risposta è "No"

**ask**("Do you agree?", **showOk**, **showCancel**); /\* passaggio delle  
funzioni showOk, showCancel come argomenti ad ask \*/

# Callback

- Possiamo riscrivere l'esempio di prima in modo più conciso utilizzando le **function expression** o le **arrow function**.

**Esempio**

```
function ask(question, yes, no) {
 if (confirm(question)) yes()
 else no();
}

ask(
 "Do you agree?",
 function() { alert("You agreed."); },
 function() { alert("You canceled the execution."); }
);
```

- Qui le funzioni vengono dichiarate dentro alla chiamata di ask(...).
- Queste non hanno nome, e perciò sono denominate **funzioni anonime**.
- Queste funzioni non sono accessibili dall'esterno di ask (perché non sono assegnate a nessuna variabile), ma è proprio quel che vogliamo in questo caso.

# setTimeout e setInterval

- Potremmo decidere di non eseguire subito una funzione, ma dopo un certo lasso di tempo. Ad esempio per far apparire un pop-up per iscriversi alla newsletter. Questo è detto **pianificare una chiamata**. Esistono due metodi per farlo:

## 1. `let timerId = setTimeout(func, [ritardo], [arg1], [arg2], ...)`

**func** è la funzione da eseguire

**ritardo** è Il ritardo in millisecondi prima dell'esecuzione (default 0)

**arg1, arg2...** sono gli argomenti della funzione

- Permette di eseguire una volta la funzione dopo l'intervallo prescelto

**Esempio** `function iscriviti (chi) {`

`alert( 'Ciao' + chi + ', iscriviti alla nostra newsletter per ricevere sconti' );`

`}`

`setTimeout(iscriviti, 1000, nome);`

- Per disattivare l'esecuzione utilizziamo `clearTimeout( timerId );`

## 2. `let timerId = setInterval(func, [ritardo], [arg1], [arg2], ...)`

- Gli argomenti sono uguali a prima ma la funzione viene eseguita regolarmente lasciando scorrere l'intervallo di tempo prescelto tra una chiamata e l'altra.
- Anche in questo caso si usa `clearTimeout`.



# Oggetti



# Creazione

- Gli oggetti vengono utilizzati per catalogare vari tipi di dati ed altri elementi più complessi.

- Creazione oggetto vuoto

```
let user = new Object(); // sintassi "costruttore oggetto"
```

```
let user = { }; // sintassi "oggetto letterale"
```

- Dentro le parentesi graffe possiamo inserire una lista di proprietà. Una **proprietà** è una coppia “chiave: valore”, dove “**chiave**” è una **stringa** mentre “**valore**” può essere di qualsiasi tipo.

```
let oggetto = {
 chiave_proprietà1: valore_proprietà1,
 chiave_proprietà2: valore_proprietà2,
};
```

- I valori delle proprietà sono accessibili utilizzando la notazione puntata.

```
oggetto.chiave_proprietà1
```

- Una proprietà può essere rimossa

```
delete oggetto.chiave_proprietà1
```

- Una proprietà può essere aggiunta

```
oggetto.chiave_proprietà3 = valore_proprietà3;
```

# Proprietà

- Per controllare se una proprietà con un certo nome esiste si utilizza l'operatore **in** che restituisce un booleano

**"chiave\_proprietà" in oggetto** // true o false

- Per attraversare tutte le chiavi di un oggetto si utilizza **for.... in**.
  - Se le chiavi sono interi, ad esempio "1", vengono ordinate, altrimenti si accede a seconda dell'ordine di creazione delle proprietà.

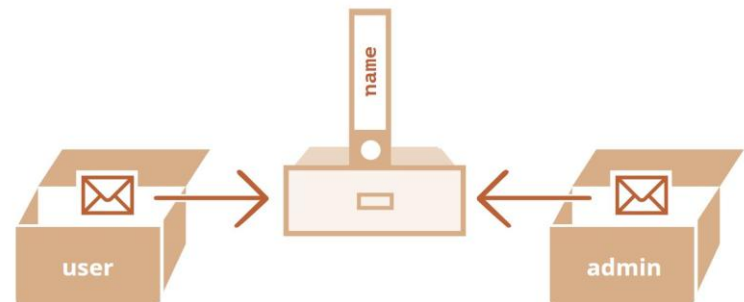
**for( let key in oggetto ) { }**

- Gli oggetti vengono assegnati e copiati **per riferimento**. In altre parole, la variabile non memorizza il "valore dell'oggetto", ma piuttosto un "riferimento" (indirizzo di memoria)

```
let user = { name: "John" };
```

```
let admin = user; // copia il riferimento
```

```
alert (admin.name); // John
```



- Tutte le operazioni effettuate su un oggetto copiato per riferimento (come aggiungere/rimuovere proprietà) vengono effettuate sullo stesso oggetto

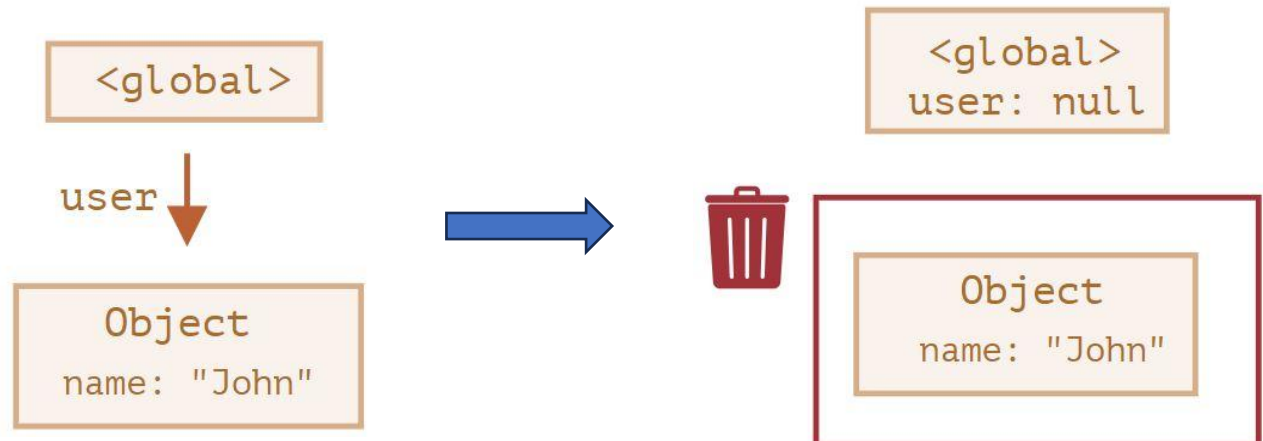
# Garbage collection

- Il processo di **garbage collection** (gestione della memoria) viene eseguito automaticamente.
- Non possiamo forzarlo o bloccarlo.
- Gli oggetti vengono mantenuti in memoria solo finché risultano raggiungibili.
- Esempio singolo riferimento

```
/* user ha un riferimento all'oggetto */
let user = {
 name: "John"
};
```



user = null;



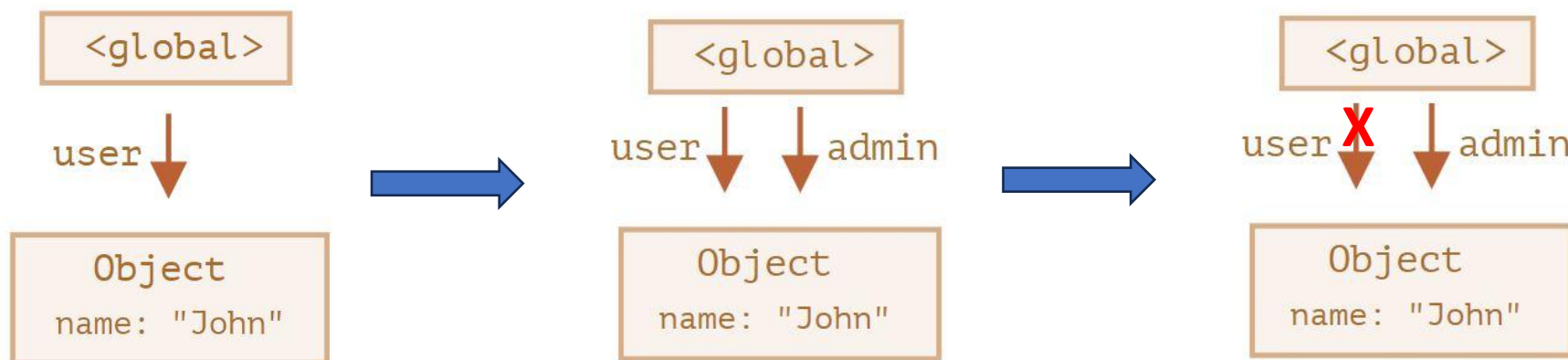
# Garbage collection

- Esempio doppio riferimento

```
/* user ha un riferimento all'oggetto */
let user = {
 name: "John"
};
```



```
let admin = user;
user = null;
alert (admin.name); // John
```



# Metodi

- Le funzioni che vengono memorizzate come proprietà di un oggetto vengono dette **metodi**. I metodi consentono agli oggetti di “agire”.
- Ci sono diversi modi per inserire un metodo in un oggetto

```
user = {
 name: "John",
 presentation () {
 alert('Ciao');
 }
};
```

← Durante la creazione dell'oggetto

```
function presentation () {
 alert('Ciao');
};
user.presentation = presentation;
```

← Successivamente alla creazione dell'oggetto  
con una dichiarazione di funzione

```
user.presentation = function() {
 alert('Ciao');
};
```

← Successivamente alla creazione dell'oggetto  
con una function expression

```
user.presentation(); // "Ciao"
```

# this

- E' molto comune che, per eseguire determinate azioni, un metodo abbia necessità di **accedere alle informazioni memorizzate nell'oggetto**
- **Per riferirsi all'oggetto**, un metodo può utilizzare la parola chiave **this** (questo non funziona però con i metodi dichiarati con le arrow function)

```
user = {
 name: "John",
 presentation () { alert(`Ciao, sono ${this.name}`); }
};
```

- In fase di esecuzione, quando viene chiamato il metodo *presentation*, il valore di **this** sarà sostituito con user
- Perché non usiamo direttamente user al posto di this? Perché ci sarebbero problemi se copiassimo il nostro oggetto.

```
user = {
 name: "John",
 presentation () { alert(`Ciao, sono ${user.name}`); }
};
```

```
let admin = user;
```

```
user = null;
```

```
admin.presentation(); //Errore: non possiamo leggere la proprietà 'name' di null
```

# Costruttore e operatore new

- La sintassi {...} ci consente di creare un oggetto. Ma spesso abbiamo bisogno di creare multipli oggetti simili, come ad esempio più utenti.
- Questo può essere fatto utilizzando un **costruttore** (una normale funzione) e l'operatore "new" (il nome inizia sempre con una lettera maiuscola)

```
function User(name, age) {
 this.name = name;
 this.age = age;
 this.presentation = function () { alert(`Ciao, sono ${this.name}`); };
}
```

```
let user1 = new User("Giacomo", 28);
let user2 = new User("Anna", 20);
```

```
alert(user1.name); // Giacomo
alert(user2.name); // Anna
```

- JavaScript fornisce costruttori per la maggior parte degli oggetti integrati nel linguaggio

# Classi

- Una classe è un costrutto utilizzato come modello per creare oggetti

```
class MyClass {
 // proprietà della classe
 prop = value;

 // metodi della classe
 constructor() { ... }
 method() { ... }

}
```

- **new MyClass()** crea un'istanza della classe MyClass (un oggetto)
- Il metodo **constructor()** viene chiamato automaticamente da new, dunque possiamo usarlo per inizializzare l'oggetto
- La notazione delle classi non va confusa con la notazione letterale per gli oggetti. In una classe non sono richieste virgole.



# Classi

- **Esempio**

```
class User {
 role = 'Admin';

 constructor(name) {
 this.name = name;
 }

 sayHi() {
 alert(`Hi ${this.name}, your role is ${this.role}`);
 }
}
```

```
let user = new User("John"); // creazione istanza della classe User
```

```
alert(user.name); // John
alert(user.role); // Admin
```

```
user.sayHi(); // utilizzo del metodo sayHi(), 'Hi John, your role is Admin'
```

# Da oggetti a primitivi

- La conversione di un oggetto a primitivo viene automaticamente effettuata da molte funzioni integrate e da operatori che si aspettano un primitivo come valore.
- Quando eseguiamo una sottrazione tra oggetti, oppure applichiamo funzioni matematiche, avviene una conversione numerica.  
Ad esempio, gli oggetti **Date** possono essere sottratti, ed il risultato di **date1-date2** è la differenza di tempo tra le due date.
- Quando mostriamo un oggetto, come in **alert( obj )** , avviene una conversione a stringa.
- Possiamo personalizzare la conversione implementando questi metodi

```
let user = {
 name: "John",
 age: 39,

 toString () { return `${this.name}`; },
 valueOf () { return this.age; }
};
```

```
alert(user); // toString -> {name: "John"}, (altrimenti [object Object])
alert(user - 13); // valueOf -> 26, (altrimenti NaN)
```



# Array

- Gli array sono uno speciale tipo di oggetto studiato per immagazzinare e gestire collezioni ordinate di dati
- Dichiarazione di un array  
**let arr = [item1, item2...];**  
alert (arr); // item1, item2, .....
- Possiamo ottenere un elemento tramite il suo **numero di indice** e le [ ]. Gli elementi di un array sono numerati a partire da **0**.
- Le [ ] ci permettono anche di rimpiazzare un elemento esistente o di aggiungerne uno
- Gli array possono contenere oggetti che sono a loro volta array. Possiamo quindi utilizzare questa proprietà per creare delle matrici

```
let matrix = [
 [1, 2, 3],
 [4, 5, 6],
 [7, 8, 9]
];
```

```
alert(matrix[1][2]); /* viene restituito il valore che si trova nella riga con indice 1
e nella colonna con indice 2, cioè il valore 6 */
```

- Possiamo eseguire sugli arrays le seguenti operazioni:
  - **push( item )** aggiunge item in coda all'array
  - **pop()** rimuove l'ultimo elemento dell'array e lo ritorna
  - **unshift( item )** aggiunge item in testa all'array
  - **shift()** rimuove un elemento dalla testa dell'array e lo ritorna
- **arr.length** restituisce la lunghezza dell' array (più precisamente l'ultimo indice numerico più uno)
- Per iterare sugli elementi di un array:
  - **for (let i = 0; i < arr.length; i++)** – restituisce l'indice dell'elemento corrente
  - **for (let item of arr)** – restituisce il valore dell'elemento corrente
- Per confrontare gli array, non possiamo utilizzare l'operatore **==** (lo stesso vale per **>**, **<**, **ecc**). Gli array vengono trattati come degli oggetti. Quindi, se confrontiamo array con **==**, non saranno mai equivalenti, a meno che non confrontiamo due variabili che fanno riferimento allo stesso array.
- Piuttosto possiamo confrontare i singoli elementi dei due array uno alla volta utilizzando **for..of**

# metodi

- Altri metodi interessanti per gestire gli array:
  - **arr.splice( start[, deleteCount, elem1, ..., elemN] )** - per rimuovere e inserire elementi
    - `let arr = [ 1, 2, 3, 4];`
    - `arr.splice(1, 1);` // a partire dall'indice 1 rimuove 1 elemento  
`alert( arr );` // 1, 3, 4
    - `arr.splice(0, 3, 5, 6);` // rimuove i primi 3 elementi e li rimpiazza con 5 e 6  
`alert( arr );` // 5, 6, 4
    - `arr.splice(2, 0, 5, 6);` // da indice 2 ne rimuove 0 e poi inserisce 5 e 6  
`alert( arr );` // 1, 2, 5, 6, 3, 4
  - **arr.slice( [start], [end] )** – crea un nuovo array e copia al suo interno gli elementi da start fino ad end (escluso). Se i valori sono negativi si inizierà a contare dalla coda dell'array.
  - **arr.concat( items... )** – ritorna un nuovo array: copia tutti gli elementi dell'array corrente e ci aggiunge items. Se uno degli items è un array, allora vengono presi anche i suoi elementi.

# metodi

- **arr.forEach(function(item, index, array) {  
    // ... fa qualcosa con l'elemento  
});** - consente di eseguire una funzione su ogni elemento dell'array
  - **// per ogni elemento chiama alert**  
["Bilbo", "Gandalf", "Nazgul"].forEach(alert);
  - **// per ogni elemento ne mostra la posizione**  
["Bilbo", "Gandalf", "Nazgul"].forEach((item, index, array) => {  
    alert(`\${item} is at index \${index} in \${array}`);  
});  
// Bilbo is at index 0 in Bilbo,Gandalf,Nazgul  
// Gandalf is at index 1 in Bilbo,Gandalf,Nazgul  
// Nazgul is at index 2 in Bilbo,Gandalf,Nazgul
- **arr.indexOf(item, from)** - cerca un item a partire dall'indirizzo from, e restituisce l'indirizzo in cui è stato trovato, altrimenti restituisce -1.  
let arr = [1, 0, false];  
alert( arr.indexOf(false) ); // 2
- **arr.includes(item, from)** - cerca un item a partire dall'indice from, e restituisce true se lo trova

# metodi

- **let result = arr.find( function(item, index, array) {**  
    /\* se viene restituito true, viene restituito l'elemento e l'iterazione si  
    ferma altrimenti ritorna undefined \*/  
});
  - per trovare un oggetto nell'array che soddisfi una specifica condizione
  - **let result = arr.findIndex( function(item, index, array) {**  
    /\* se viene restituito true, viene restituito l'indice dell'elemento e  
    l'iterazione si ferma altrimenti ritorna -1 \*/  
});
  - è molto simile a find ma restituisce l'indice
  - **let results = arr.filter(function(item, index, array) {**  
    /\* se un item è true viene messo dentro results e l'iterazione continua.  
    Qualora nessun elemento restituisse true, viene restituito un array  
    vuoto\*/  
});
  - restituisce tutti gli oggetti che soddisfano una determinata condizione
- Esempio:** let users = [  
    {id: 1, name: "John"},  
    {id: 2, name: "Pete"},  
];  
let someUsers = users.filter(item => item.id < 3);  
alert(someUsers.length); // 2



# metodi

- **let results = arr.map(function(item, index, array) {  
    // restituisce il nuovo valore piuttosto di item  
});** - la funzione viene chiamata per ogni elemento dell'array e viene restituito un array con i risultati.

## Esempio:

```
let lengths = ["Bilbo", "Gandalf", "Nazgul"].map(item => item.length);
alert(lengths); // 5,7,6
```

- **arr.sort();** - riordina il contenuto di arr e lo restituisce

# Date e Time

- Le date e gli orari in JavaScript sono rappresentate dall'oggetto **Date**. Non possiamo creare “solo una data” o “solo un orario”: l'oggetto Date li gestisce entrambi.
  - **date = new Date();** - crea un oggetto Date con la data e l'ora corrente
  - **date = new Date(year, month, date, hours, minutes, seconds, ms);** - crea un oggetto Date
    - Il campo **year** deve essere composto da 4 cifre (obbligatorio)
    - Il numero **month** inizia da 0 (Gennaio), fino a 11 (Dicembre) (obbligatorio)
    - Il parametro **date** rappresenta il giorno del mese, se non viene fornito il valore di **default** è **1**
    - Se non vengono forniti **hours/minutes/seconds/ms**, il valore di **default** è **0**
- Esempio** `date = new Date(2011, 0, 1);` // 1 Gennaio 2011, 00:00:00
- Per accedere ai componenti dell' oggetto Date:
    - **date.getFullYear()** – restituisce l'anno
    - **date.getMonth()** - fornisce il valore del mese da 0 a 11
    - **date.getDate()** - fornisce il giorno del mese, da 1 a 31
    - **date.getHours()**, **date.getMinutes()**, **date.getSeconds()**, **date.getMilliseconds()**
    - **date.getDay()** - restituisce il giorno della settimana, da 0 (Domenica) a 6 (Sabato)

# Date e Time

- Per impostare i componenti di data e tempo:
  - `date.setFullYear(year, [month], [date])`
  - `date.setMonth(month, [date])`
  - `date.setDate(date)`
  - `date.setHours(hour, [min], [sec], [ms])`
  - `date.setMinutes(min, [sec], [ms])`
  - `date.setSeconds(sec, [ms])`
  - `date.setMilliseconds(ms)`
- L' **autocorrezione** è una caratteristica molto utile degli oggetti Date. Potremmo inserire valori fuori dagli intervalli, e questi verranno automaticamente aggiustati.

**Esempio**      `let date = new Date(2013, 0, 32); // 32 Gen 2013 !?`  
`alert(date); // ...è il primo Feb 2013!`

- Se si effettua la sottrazione di due date si ottiene la loro differenza in millisecondi
- Si utilizza **Date.now()** per ottenere più rapidamente il corrente timestamp senza creare un oggetto Date (il timestamp è rappresentato come il numero di millisecondi passati dal 1° gennaio 1970 00:00:00 UTC)

# JSON

- JSON (JavaScript Object Notation) è un formato per rappresentare valori e oggetti
- JSON è comunemente usato per lo scambio di dati tra client e server
- JavaScript fornisce i metodi:
  - **JSON.stringify** per convertire oggetti in JSON
    - prende l'oggetto e lo converte in stringa
    - un oggetto codificato in JSON possiede delle fondamentali differenze da un oggetto letterale (le chiavi delle proprietà e il valore delle proprietà di tipo stringa vengono racchiuse tra doppi apici)
    - I metodi e le proprietà che contengono undefined vengono ignorate

```
student = {
 name: 'John',
 age: 30,
 isAdmin: false,
 courses: ['html', 'css', 'js'],
 married: null,
 worker: undefined
};
```

```
let json = JSON.stringify(student);
```

```
alert(json);
/* JSON-encoded object:
{
 "name": "John",
 "age": 30,
 "isAdmin": false,
 "courses": ["html", "css", "js"],
 "married": null
} */
```

# JSON

- **JSON.parse** per convertire JSON in oggetto

```
let userData =
```

```
{
 "name": "John",
 "age": 35,
 "isAdmin": false,
 "friends": [0,1,2,3]
};
```

```
let user = JSON.parse(userData);
```

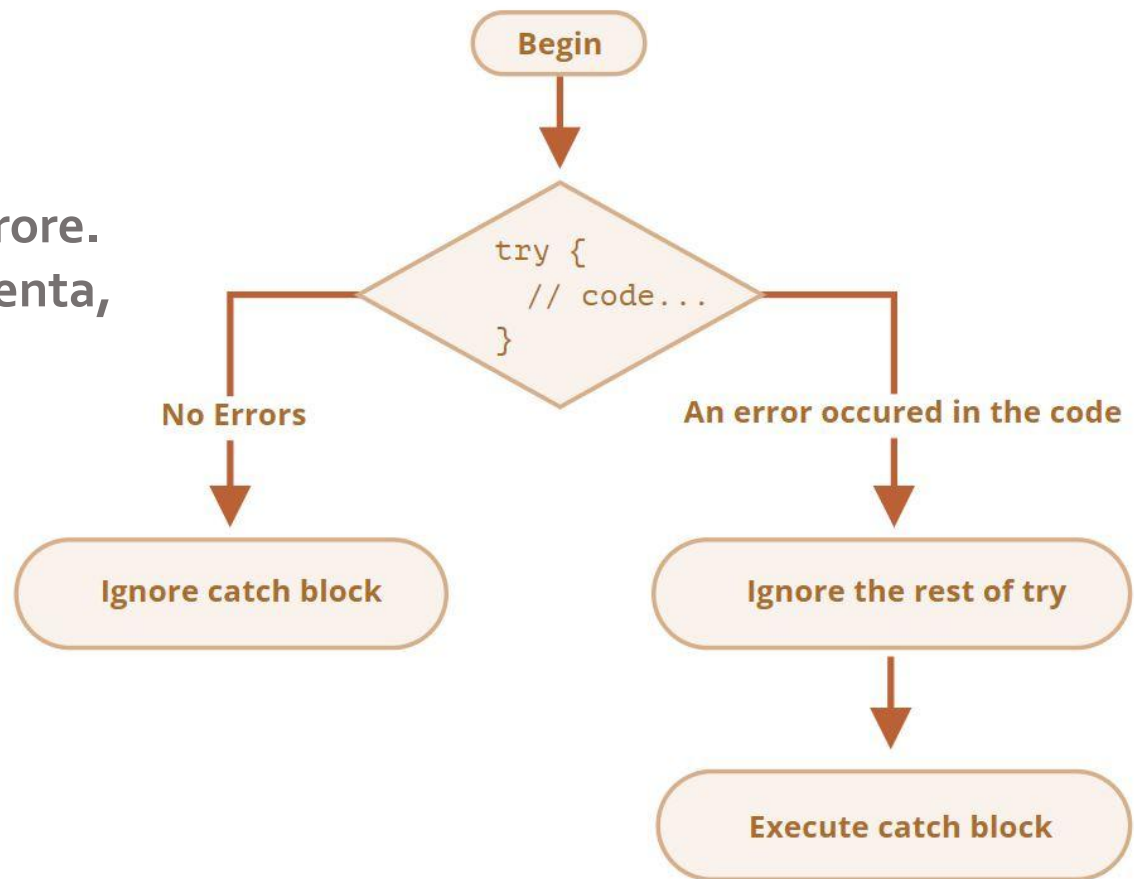
```
alert(user.isAdmin); // false
```

```
alert(user.friends[1]); // 1
```

# Gestione degli errori – try....catch

- **try...catch** permette la gestione degli errori al momento dell'esecuzione (runtime errors) evitando che lo script si fermi.

```
try {
 // esegui il codice
} catch (err) {
 /* err è l'oggetto errore.
 Se un errore si presenta,
 viene gestito */
}
```



# Gestione degli errori – try....catch

- Quando un errore si verifica, JavaScript genera un oggetto di tipo errore contenente i dettagli al riguardo.
- Due proprietà dell'oggetto errore sono:
  - **name** - il nome dell'errore
  - **message** - il messaggio testuale con i dettagli dell'errore

## Esempio

```
try {
 variabile; // errore, variabile non definita!
} catch (err) {
 alert(err.name); // ReferenceError
 alert(err.message); // la variabile non è definito
 /* Può essere anche visualizzato nel suo complesso
 L'errore è convertito in una stringa del tipo "name: message" */
 alert(err); // ReferenceError: la variabile non è definito
}
```

# Gestione degli errori – throw

- Possiamo anche generare un errore personalizzato usando l'operatore **throw**.

**throw <error object>**

- JavaScript ha già molti costruttori integrati per errori generici:

**Error, SyntaxError, ReferenceError, TypeError** e altri

**let error = new nome\_errore (messaggio);**

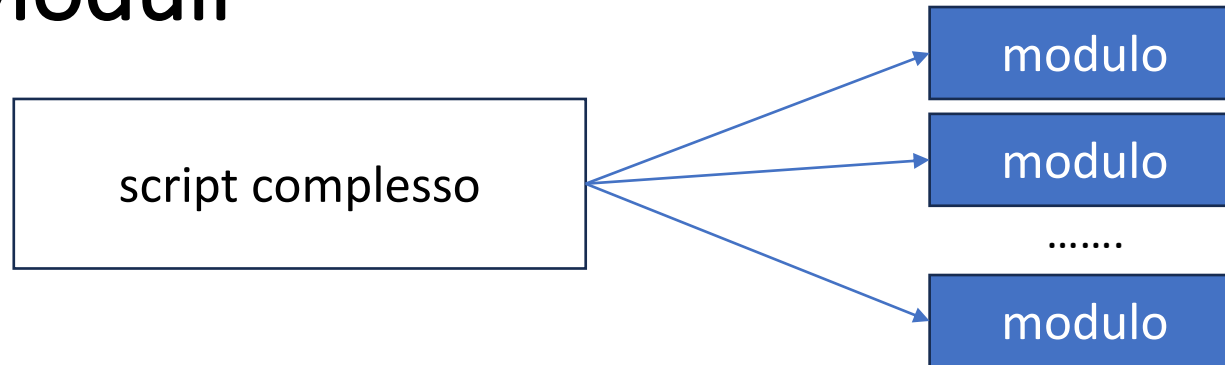
## Esempio

```
let json = '{ "age": 30 }'; // dati incompleti
```

```
try {
 let user = JSON.parse(json);
 if (!user.name) {
 throw new SyntaxError("Dati incompleti: manca name");
 }
 alert(user.name);
} catch(err) {
 alert ("JSON Error: " + err.message);
 // JSON Error : Dati incompleti: manca name
}
```



# Moduli



- La direttiva **export** all'interno di un modulo permette di rendere variabili e funzioni accessibili dall'esterno
- 1. Possiamo inserire **export** prima della dichiarazione di una variabile, una funzione o una classe.

**export class/function/variable ...**

2. Possiamo inserire **export** separatamente alla dichiarazione (la keyword **as** permette di utilizzare nomi differenti)

**export {x [as y], ...}**

# Moduli

- La direttiva **import** all'interno di un modulo permette di importare funzionalità da altri moduli
  1. Solitamente, inseriamo import prima di una lista di ciò che vogliamo importare tra le parentesi graffe

```
import {x [as y], ...} from "module"
```
  2. Possiamo importare tutto come un oggetto utilizzando

```
import * as obj from "module"
```
- I moduli dovrebbero eseguire l'export di ciò che vogliono che sia accessibile dall'esterno e l'import ciò di cui hanno bisogno
- Le istruzioni import solitamente sono inserite in cima allo script mentre quelle export alla fine