COESIONE ITALIA 21-27 PIEMONTE
Cofinanziato dall'Unione europea
REGIONE PIEMONTE

Web Developer

Programmazione - Javascript e Typescript

Docente: Shadi Lahham

Intervento realizzato da ITS TECNOLOGIE DELL'INFORMAZIONE E DELLA COMUNICAZIONE

JS in depth

Objects, arrays, functions and more

Shadi Lahham - Web development

Let, var & const

Let vs var

differenza tra block scope (let) e function scope (var)

```
function worker() {
   let x = 88;
   for (let i = 0; i < 4; i++) {
       console.log('i block =', i);
   }
   console.log('x func =', x);
   console.log('i !block =', i); // undefined
}

worker();
console.log('x !func =', x); // undefined
```

let: Block-scoped

Access restricted to nearest enclosing block

```
function worker() {
   var x = 88;
   for (var i = 0; i < 4; i++) {
       console.log('i block =', i);
   }
   console.log('x func =', x);
   console.log('i !block =', i); // output?
}
worker();
console.log('x !func =', x); // undefine
```

attenzione alla l

var: Function-scoped

Access restricted to nearest enclosing function
Common in older Javascript code

Const

```
let x = 88;
const y = 77;
x = 9;
console.log('x = ', x);
y = 17;  // TypeError: Assignment to constant variable.
console.log('y = ', y);
const y = 55; // SyntaxError: Identifier 'y' has already been declared
```

const:
Block-scoped, like **let**
Values of const variables cannot be reassignment
Const variables cannot be redeclared

Let bug in IE11

```
for (let i = 0; i < 3; ++i) {
   setTimeout(function() {
       console.log(i);
    }, i * 100);
}

// output on chrome 0,1,2
// output on IE11 3,3,3
```

bug in internet explorer 11 che esplodeva

Browser support

Let

Flow

Conditional (Ternary) Operator

```
// example 1
let age = 8;
let voteable = age < 18 ? 'Too young' : 'Old enough';

// example 2
let now = new Date();
let greeting = 'Good' + (now.getHours() > 17 ? ' evening.' : ' day.');
```

alternativa all'if else

SI CHIAMA OPERATORE TERNARIO!!

Conditional (Ternary) Operator

```
// example 3 - if else vs ternary operator
let x = 71;
let y = 8;
let result;
if (x > y) {
  result = 'good job';
} else {
  result = 20;
}
// Below is the above code written using the ternary operator
let result2 = x > y ? 'good job' : 20;
```

confronto tra i 2 modi

Conditional (Ternary) Operator

```
// chaining
let bar;
let foo = bar === 'a' || bar === 'b' ? (bar === 'a' ? 1 : 2) : 3;

// is this too much??
let i = 5;
let result = i % 2 == 0 ? 'a' : i % 3 == 0 ? 'b' : i % 5 == 0 ? 'c' : i % 7 == 0 ? 'd' : 'e';
```

Arrays

Access and assignment

```
let teachers = ['Gina', 'Amanda', 'Brenda', 'Amy'];
let classes = [];

classes[0] = 'HTML';
classes[1] = 'CSS';
classes.push('JS');
classes.pop();

let i = 0;
classes[i];
classes[1];
classes.pop();
classes.length;
```

metodi degli arrays

Iteration

```
let classes = [];
classes[0] = 'HTML';
classes[1] = 'CSS';
classes.push('JS');
for (let i = 0; i < classes.length; i++) {
  console.log(i);
}
```

ciclare E ITERARE gli elementi di un array

For .. of

```
let countries = ['Italy', 'France', 'Germany'];

for (let country of countries) {
  console.log(country);
}
```

Pro: Flessibile, permette di usare break e continue, può essere utilizzato su qualsiasi iterabile.
Contro: Non specifico per gli array e non restituisce automaticamente un nuovo array.

Browser support

for...of

ForEach

esegue una funzione su ogni elemento dell'array, ma NON RESTITUISCE un nuovo array.

```
let numbers = [1, 2, 3, 4];

// using for
for (let i = 0; i < numbers.length; i++) {
  console.log(numbers[i]);
}

// using forEach
numbers.forEach(function (number) {
  console.log(number);
});
```

USARE IL FOR EACH

Io uso quando non mi interessa e NON ho bisogno di un NUOVO ARRAY.

come ad esempio loggare valori nella console o modificare un array esistente

Non può essere concatenato (non puoi usarlo in una catena di metodi perché non restituisce un array)..

Map

esegue una funzione su ogni elemento dell'array e restituisce un nuovo array con i risultati.
È progettato per trasformazioni, ossia per creare un nuovo array modificato a partire da un array originale.

```
let numbers = [1, 2, 3, 4];
```
dichiarazione e creazione di un array contenente dei numeri

```
// using for
let newNumbers = [];
for (let i = 0; i < numbers.length; i++) {
  newNumbers[i] = numbers[i] * 2;
}

// using map
let newNumbers2 = numbers.map(function (number) {
  return number * 2;
});
```

Pro: Restituisce un nuovo array, ideale per trasformazioni, può essere concatenato con altri metodi di array.

Contro: Non dovrebbe essere usato solo per effetti collaterali (come forEach), dato che restituisce un array che potrebbe non essere utilizzato.

Method chaining

```
let numbers = [1, 2, 3, 4];
let newNumbers = numbers
  .map(function (number) {
    return number * 2;
  })
  .map(function (number) {
    return number + 1;
  });
```

COME SI CONCATENANO + METODI TRA DI LORO, con for each ad esempio non si potrebbe.

ecco perchè .MAP è molto utile

con funzioni anonime è molto + corto:

```
// shorter version
let numbers = [1, 2, 3, 4];
let newNumbers = numbers.map(number => number * 2).map(number => number + 1);
```

Filter

viene utilizzato per creare un nuovo array CONTENENTE SOLO gli elementi che soddisfano una determinata CONDIZIONE.

In altre parole, filter() filtra gli elementi di un array in base a un criterio specificato e restituisce un nuovo array con gli elementi che passano il test.

```
let numbers = [1, 2, 3, 4];
```

dichiarazione e creazione di un array contenente dei numeri

```
// using for
let newNumbers = [];
for (let i = 0; i < numbers.length; i++) {
  if (numbers[i] % 2 !== 0) {
    newNumbers.push(numbers[i] * 2);
  }
}
```

if --> controllo della condizione che sia true

```
// using filter
let newNumbers2 = numbers
  .filter(function (number) {
    return number % 2 !== 0;
  })
  .map(function (number) {
    return number * 2;
  });
```

concatenazione (chaining dei metodi) FILTER + MAP

Reduce

consente di ridurre (o accumulare) tutti gli elementi di un array in un unico valore.

Questo valore può essere un numero, una stringa, un array, un oggetto, o qualsiasi altra struttura che desideri.

```
let numbers = [1, 2, 3, 4];

// using for
let totalNumber = 0;
for (let i = 0; i < numbers.length; i++) {
  totalNumber += numbers[i] * 2;
}

// using reduce
let totalNumber2 = numbers
  .map(function (number) {
    return number * 2;
  })
  .reduce(function (total, number) {
    return total + number;
  }, 0);
```

Reduce

```
let some = [1, 2, 3, 4, 5, 6, 7].reduce(function (accu, curr) {
  if (Math.random() > 0.5) {
    accu.push(curr);
  }
  return accu;
}, []);
console.log(some);
```

Reduce

```
let kebab = ['I', 'hAve', 'A', 'drEam'].reduce(function (accu, curr, index, arr) {
  const word = index === arr.length - 1 ? curr : curr + '-';
  return accu + word.toLowerCase();
}, '');
console.log(kebab);
```

Objects

Creation and assignment

RIPASSO GENERICO DEGLI OGGETTI

```
// create object and assign property
let cat = {};
cat.furColor = 'orange';

// create and assign
let cat2 = { furColor: 'orange' };

// Object literal
let cat = {
  age: 5,
  furColor: 'orange',
  isHappy: true,
  likes: ['sleep', 'milk'],
  birthday: { month: 7, day: 17, year: 2020 }
};
```

Nested objects

come si nestano tra di loro gli oggetti, in questo caso vaso a creare all'interno di innerdoll (oggetto), un altro innerdoll (oggetto) con un nuovo size tutto suo

```
let doll = {
  size: 'large',
  innerDoll: { size: 'medium' }
};
doll.innerDoll.innerDoll = { size: 'small' };

console.log(doll);
```

Array of objects

ripasso sugli array contenente oggetti, e come accedere ai valori di essi tramite DOT NOTATION E BRAKET NOTATION

```
// array of objects
let cats = [
  { name: 'Angel', age: 18, furColor: 'grey' },
  { name: 'Evil', age: 14, furColor: 'red' },
  { name: 'Meh', age: 12, 'Fur Color': 'white' }
];
console.log(cats);
console.log(cats[1].furColor);
console.log(cats[2]['Fur Color']);
```

Dot notation

```
// Dot notation

// reading properties
let furVariable = cat.furColor;
console.log(furVariable);

// modifying properties
cat.furColor = 'grey';
console.log(furVariable);
```

assegniamo in questo esempio a una variabile il valore di una chiave di un oggetto

modifichiamo il valore di una chiave di un oggetto

Bracket notation

```
// cannot be done with dot notation
cat['fur color'] = 'orange';
let facebookFriends = {};
facebookFriends[12323] = cat;

// bracket notation with variables
let cat = {};
let prop = 'furColor';
cat[prop] = 'orange';
let color = cat[prop];
```

utile quando hai bisogno di accedere o modificare le proprietà di un oggetto in MODO DINAMICO cioè quando il nome della proprietà non è fisso ma può variare durante l'esecuzione del codice

A differenza della notazione con il punto, che richiede un nome di proprietà letterale, la notazione a parentesi ti permette di usare una variabile per determinare il nome della proprietà.

Bracket notation with variables

```
let socials = ['instagram', 'tiktok', 'twitter', 'pinterest'];

const handshake = {
  pinterestShare: function () {
    // code to share on pinterest
  },
  twitterShare: function () {
    // code to share on twitter
  }
};

for (const social of socials) {
  const callback = handshake[social + 'Share'];
  if ('function' === typeof callback) {
    callback(); // equivalent to handshake[social + 'Share']();
  }
}
```

Iterating using for .. in

```
let zoo = {
  birds: 3,
  bears: 5,
  cats: 12
};
for (let key in zoo) {
  if (zoo.hasOwnProperty(key)) {
    console.log('zoo.' + key + ' = ' + zoo[key]);
  }
}

// remember: for .. in is for objects, for .. of is for arrays
// don't use for..of on objects
const sam = { name: 'sam', age: 42 };
for (const property of sam) {
  // TypeError: sam is not iterable
  console.log(property);
}
```

Object keys(), values() & entries()

ripasso dei metodi degli oggetti

```
const zooAnimals = {
  animal1: 'Lion',
  animal2: 'Elephant'
};

Object.keys(zooAnimals).forEach(key => { // keys() returns an array of object's properties
  console.log(key, zooAnimals[key]); // animal1 Lion, animal2 Elephant
});

Object.values(zooAnimals).forEach(val => { // values() returns an array of object's values
  console.log(val); // Lion, Elephant
});

Object.entries(zooAnimals).forEach(entry => { // entries() returns an array of key-value pairs
  const [key, value] = entry;
  console.log(key, value); // animal1 Lion, animal2 Elephant
});
```

RESTITUISCE:
KEY

RESTITUISCE:
VALUE

array !!!!

RESTITUISCE:
KEY
VALUE

Operators

Logical OR ||

se lo mettiamo nell if NON RESTITUISCE TRUE,
ma lo legge come true e esegue all interno delle {}

quindi realmente restituisce il valore che risulta vero tra i 2 (è un operatore binario quindi solo 2)

The OR || operator using non-boolean values

1. Evaluates operands from left to right
2. For each operand, if it is truthy, stops and returns the original value of the operand
3. If all operands are falsy, returns the last operand

```
let name = '';
let userName = name || 'default'; // default
```

se il 1 è falsey, restituisce il 2

```
let name2 = 'james';
let userName2 = name2 || 'default'; // james
```

se il 1 è true, allora restituisce quello

Logical OR ||

person.name se l'oggetto non è stato dedinito, javascript si romperà

Logical AND &&

The AND && operator using non-boolean values

1. Evaluates operands from left to right
2. For each operand, if it is falsy, stops and returns the original value of that operand
3. If all operands are truthy, returns the last operand

```
let userName = person && person.name; // undefined
```

il 1 valore è falsey, quindi restituisce quello e non va avanti (undefind)

```
let person = {};
userName = person && person.name; // undefined
```

il 1 valore è true, allora controlla come è il 2 valore, in questo caso è false e restituisce quel valore (undefind)

```
person = { name: 'james' };
userName = person && person.name; // james
```

qui sono entrambi true, allora andrà a restituire l'ultimo valore true (james)

Logical AND &&

Functions

Another way to look at functions

```
let add = function(a, b) {
  return a + b;
};

let mad = add;

let resultA = add(5, 4); // 9

let resultB = mad(21, 7); // 28

console.log(typeof add); // function
```

```
// note: functions are regular objects with the additional capability of being invokable
```

Another way to look at functions

```
function add(a, b) {
  return a + b;
}

let mult = function(a, b) {
  return a * b;
};

let calculate = function(fn, a, b) {
  console.log('This is your result:', fn(a, b));
};

calculate(add, 2, 4);
calculate(mult, 2, 4);


// note: functions can be passed as parameters
```

Return and side effects

```
let greeter = function(name, place) {
  return 'Mister ' + name + ' of' + place;
};

// function with a side effect
let nameLogger = function(name, place) {
  let newName = 'Mister ' + name + ' of' + place;
  console.log(newName);
  return newName;
};
```

Function arguments

```
let add = function(a, b) {
  console.log(arguments); // logs [3,10]
  return a + b;
};

let sum = add(3, 10); // 13
```

Function arguments

```
let addMany = function() {
  let sum = 0;
  for (let i = 0; i < arguments.length; i++) {
    sum += arguments[i];
  }
  return sum;
};

let sumA = addMany(3, 10, 57, 24); // 94
let sumB = addMany(3, 10, 57, 24, 200, 300); // 594
```

```
// implement a function that returns the max of n arguments; your own version of Math.max()
```

Default arguments

```
let nameLogger = function (name, adj) {
  if (adj === undefined) {
    adj = 'wonderful';
  }
  let newName = 'The ' + adj + ' Mr.' + name;
  console.log(newName);
};

nameLogger('adam', 'lazy');
nameLogger('james');
```

Global and local precedence

```
let g = 'global';

function go() {
  let l = 'local';
  let g = 'in here!';
  console.log(g + ' inside go');
}

go();
console.log(g + ' outside go');
```

Your turn

1.Soundwave

Given the following array

```
let noisesArray = ['quack', 'sneeze', 'boom'];
```

Produce the following array, then print it to the console

```
['Quack!','qUack!!','quAck!!!','quaCk!!!!','quacK!!!!!','Sneeze!','sNeeze!!','snEeze!!!','sneEze!!!!
','sneeZe!!!!!','sneezE!!!!!!','Boom!','bOom!!','boOm!!!','booM!!!!']
```

2.Babies

- Create an empty array of babies
- Each baby should have the following properties
  - "name" (a string)
  - "months" (age in months as number)
  - "noises" (an array of strings)
  - "favoriteFoods" (an array of strings)
- Add 4 different babies to the array using as many different ways as possible
- Iterate through the array printing key and value pairs e.g [name:”Lyla”]
- Now add an "outfit" property to each baby in the array
  - Outfit should describes at least 3 parts of their clothing using different properties, for example, "shirt": "blue"
  - Print each baby again with their outfit in a nicely formatted object

3.Baby processing

Using the babies array from the previous exercise:

- Write a getBabyOutfit() function that returns a description a baby's outfit
  - e.g "Lyla is wearing a blue shirt and red pants and a green hat"
- Write a feedBaby() function that prints what a baby is eating.
  - e.g. "Lyla is eating food3, food1, food4 and food2"
  - All foods in favoriteFoods should appear but randomly each time the function is called
- Run both function on all the babies

4.Clone

Write a function clone() that clones any object

- Test it on the object in the next slide
- Change the name of the cloned object and make sure that the original did not change

Important:
Write the function yourself, do not use built-in functions such as Object.assign(), jQuery.extend() or JSON.parse(JSON.stringify())

Continues on next page >>>

4.Clone

```
The object to clone and test:
{
  name: 'Green Mueller',
  email: 'Rigoberto_Muller47@yahoo.com',
  address: '575 Aiden Forks',
  bio: 'Tenetur voluptatem odit labore et voluptatem vel qui placeat sit.',
  active: false,
  salary: 37993,
  birth: Sun Apr 18 1965 13:38:00 GMT+0200 (W. Europe Daylight Time),
  bankInformation:
   { amount: '802.04',
     date: Thu Feb 02 2012 00:00:00 GMT+0100 (W. Europe Standard Time),
     business: 'Bernhard, Kuhn and Stehr',
     name: 'Investment Account 8624',
     type: 'payment',
     account: '34889694' }
}
```

Bonus

5.Clone strings

Write a function cloneStrings() that only clones string properties of an object
Starting with the example object of the previous exercise this should be the result

```
{ name: 'Green Mueller',
  email: 'Rigoberto_Muller47@yahoo.com',
  address: '575 Aiden Forks',
  bio: 'Tenetur voluptatem odit labore et voluptatem vel qui placeat sit.',
  bankInformation:
   { amount: '802.04',
     business: 'Bernhard, Kuhn and Stehr',
     name: 'Investment Account 8624',
     type: 'payment',
     account: '34889694' }
}
```

6.Reduce All

Make sure that you fully understand the Array reduce method

Write functions that use the reduce method to implement your version of the following Array methods:
forEach() , map(), filter() , indexOf() , slice()

For each method, implement parameters and return values as in the documentation

- do not use Array.prototype
- your functions receive as a first parameter the array on which to operate
- all other parameters should be identical to the documentation
- except for the thisArg parameter, you don't have to implement it

For example your implementation of forEach could be something like this:

```
function myForEach(arr, ... ) {
}
```

Note: *This exercise is harder than the ones you have done so far. Dedicate enough time for it*

Continues on next page >>>

6.Reduce All

Testing:

- write tests that compare the output of your functions to those of the Array methods
- write several and comprehensive tests for each method
- make sure that your methods give the same output as the originals

Note: *See the following slide with an example of how to test myMap()*

Continues on next page >>>

6.Reduce All

```
// Example of testing myMap
// group of arrays used for testing
let testGroup = [
    [ 1, 2, 3, 4, 5 ],
    [ 0, 0, 3, 4, 5 ],
    [ 7, 0, 9, 74, 85, 1, 42, 3, 88 ]
];

// test function for testing map - can be any function as long as the parameters are what map
expects
let testFunc = function(num) {
  return num * 2;
};

// replace this with your implementation of map using reduce
function myMap(arr, ... ) {
}
```

Continues on next page >>>

6.Reduce All

```
console.log('==== Testing Array.map() method ====');
testGroup.forEach(function(arr) {
  console.log(arr.map(testFunc));
});

console.log('\n==== Testing the function myMap() ====');
testGroup.forEach(function(arr) {
  console.log(myMap(arr, testFunc));
});

// note that tests for forEach, indexOf, filter and slice will be different because the methods
behave differently
```

References

Ternary operator

ternary operator vs. if statement

An alternative to if/else and switch in JavaScript

How to Read Nested Ternary Operators

Ternary operator discussion

References

Array methods

JavaScript Array Reference

Array - JavaScript

Functional programming in Javascript: map, filter and reduce

References

for .. in and for .. of

for...in - JavaScript

for...of - JavaScript

for..in versus for..of Loops

Logical operators

The && and || Operators in JavaScript — Marius Schulz

References

Functions have more complexity than you think

Explore these references to learn more

JavaScript Function Definitions

Every Possible Way to Define a Javascript Function