Scuola universitaria professionale della Svizzera italiana
Dipartimento tecnologie innovative
Istituto sistemi informativi e networking

SUPSI

JavaScript: Approfondimenti

Roberto Guidi

JavaScript - Variabili

- JavaScript è un linguaggio dinamico, le variabili possono contenere un qualsiasi tipo ed essere modificate nel tempo.

- Una variabile dichiarata ma non inizializzata è per default `undefined`

- La dichiarazione di variabili in JavaScript può essere fatta in modi differenti:
  - `var`: variabile con scope di funzione
  - `let`: variabile con scope di blocco
  - `const`: costante, con scope di blocco

JavaScript - `var` e `let`

- Le variabili dichiarate con `var` hanno uno scope di funzione
- Questo significa che sono visibili **all'interno di tutta la funzione nella quale sono definite**, comprese eventuali funzioni o blocchi interni
- Se dichiarate fuori da una qualsiasi funzione assumono uno scope globale e vengono aggiunte all'oggetto window.

```
// Function declaration
function testFunction(){
    var a = 'WebApp'
    testFunction2();

    // New Block
    {
        console.log('Inside block', a);
    }

    // New function
    function testFunction2(){
        console.log('Inner function', a);
    }
}
// Function call
testFunction();
```

JavaScript - `var` e `let`

- Le variabili dichiarate con `let` hanno invece uno scope di blocco.
- Sono quindi visibili **unicamente all'interno del blocco nella quale sono definite**, compresi in blocchi o funzioni interne.
- Se definite fuori da una qualsiasi funzione, assumono uno scope globale, ma non vengono aggiunte all'oggetto window.
- Se si prova a richiamare una variabile dall'esterno del blocco nel quale è definita si otterrà un **reference error**
- Le variabili dichiarate con `let` non sottostanno al meccanismo di hoisting. Se si prova ad accedere ad una variabile non ancora dichiarata si ottiene un errore.
- Prediligere l'utilizzo di `let` invece che `var`

```
// New Block
{
    let a = "Hello";
    console.log('Inside block', a);
    testFunction();

    function testFunction(){
        console.log('Inner function', a);
    }
}

//Outside block: reference error
console.log(a);
```

JavaScript - `const`

- Le variabili dichiarate con `const` hanno uno **scope di blocco** come quelle dichiarate con `let` e si comportano allo stesso modo riguardo alla visibilità.
- Si tratta di constanti, il quale valore non può più essere modificato. Non significa immutable!
- Attenzione: nel caso la variabile contenga un oggetto (tipo di riferimento), non è possibile riassegnare un altro oggetto, ma è possibile modificare i campi dell'oggetto.
- Questo poiché il contenuto della variabile di fatto non cambia (è un indirizzo di memoria).
- **Prediligere l'utilizzo di variabili di tipo `const`**

Esempio 1

```
const a = "Hello";
a = "World"
// errore read-only
```

Esempio 2

```
const b = {
   nome: 'Anakin',
   cognome: 'Skywalker'
}
b.nome = 'Darth'
b.cognome = 'Vader'
console.log(b);
// OK, il riferimento non cambia!
```

JavaScript - Hoisting variabili

- Una variabile può essere dichiarata anche dopo il suo primo utilizzo. Questo aspetto è reso possibile da un meccanismo chiamato **hoisting**.

- L'interpreta si occupa in pratica di portare la definizione delle variabili all'inizio del proprio scope.

```
console.log(a);
var a = 'Hello World'
console.log(a);
```

- La prima istruzione non darà errore ma stamperà `undefined`, in pratica è come se `var a;` venisse spostato all'inizio dello script

- Il meccanismo di hoisting viene applicato anche alla dichiarazione di funzioni. È quindi ad esempio possibile richiamare nel codice una funzione prima di averla dichiarata.

JavaScript - Truthy and Falsy values

- A differenza di altri linguaggi come Java, in JavaScript è possibile valutare un qualsiasi oggetto a `true` o `false` (boolean). Si parla di Truthiness.
- Si dice che un oggetto:
  - `Truthy` se viene valutato come `true`
  - `Falsy` se viene valutato come `false`

Truthy	Falsy
Letterale `true`	Letterale `false`
Oggetti e array (compresi `{}` e `[]` vuoti)	`null`, `undefined` e `NaN` (not a number)
Stringhe, tutte a parte la stringa vuota	Stringa vuota `''`
Tutti i numeri positivi e negativi (anche float), eccetto lo `0`	Numero `0`

JavaScript - Ternary Operator

- L'operatore ternario è in pratica un `inline if`
- La sintassi di questo operatore è formata da 3 parti:

  `condizione ? espressione se vero : espressione se falso`

```
const age = 22;
const license = age >= 18 ? 'May have driver license' : 'No driver license';
console.log(license);
```

- È possibile concatenare multipli operatori ternari uno dentro l'altro
- Evitare in ogni caso di abusare di questo costrutto

JavaScript - Template literals

- Un template literal permette di inserire in modo semplice dei valori, provenienti da variabili o espressioni all' interno di una stringa.
- Un template literal è in pratica una stringa dichiarata utilizzando al posto dei single quotes `' '` il carattere back tick `` ` ``
- All interno dei back ticks è possibile inserire del testo e concatenare con delle espressioni utilizzando i caratteri `${}`

```
const name = 'Obi-Wan';
const job = 'Jedi Master';
const helloMessage = `Hello ${name}, ${job}!`;
console.log(helloMessage);
```

- È possibile concatenare template literals con string literals (definiti con `' '` o `" "`)
- I template literals permettono anche di definire facilmente stringhe multi linea

JavaScript - Objects shorthand properties

- Gli oggetti sono delle strutture dati che possono contenere campi e metodi.
- I campi all'interno di un oggetto sono espressi sotto forma di `chiave: valore`
- Utilizzando le shorthand properties, è possibile popolare i campi di un oggetto a partire da variabili, omettendo il nome della chiave.
  La chiave assumerà il nome della variabile utilizzata.

Esempio 1

```
const name = 'Darth Vader';
const job = 'Sith Lord';
const age = 41;

// classico chiave valore
let character = {
    name: name,
    job: job,
    age: age
}

// Shorthand property, omettiamo il nome della chiave
character = {
    name,
    job,
    age
}
```

JavaScript - Objects computed property names

- In modo simile a quanto visto in precedenza, esiste anche un sistema per popolare l'oggetto con delle proprietà il quale nome può essere assegnato dinamicamente, ad esempio basandosi sul valore di una variabile

- In questo caso è possibile usare il costrutto `[keyName]: value` in modo da passare dall'esterno il nome della chiave desiderato

```
const name = 'Darth Vader';
const job = 'Sith Lord';
const age = 41;

//key names
const characterName = 'characterName';
const role = 'role';

//alcune dynamic properties, passiamo il
nome da variabile
const character = {
    [characterName]: name,
    [role]: job,
    age
}
console.log(character);
```

JavaScript - Objects destructuring

- La sintassi di destructuring permette di estrarre delle proprietà da un oggetto, indicandone il nome all'interno delle parentesi graffe { }
- È possibile destrutturare anche delle chiavi relative ad un oggetto innestato

```
const player = {
    name: 'Darth Vader',
    type: 'Sith Lord',
    stats: {
        health: 15,
        power: 30
    }
}

//Accesso con destructuring
const { name, type, stats } = player;
console.log(name, type, stats);

//Destructuring proprietà di un oggetto
nested
const {stats: {health}} = player;
console.log(health);
```

JavaScript - Objects destructuring

- In caso di conflitto tra nomi di variabili è possibile specificare un nuovo nome per la proprietà che si vuole estrarre. Questo è possibile usando la sintassi `:<name>` dopo il nome della proprietà interessata

- È possibile destrutturare direttamente un oggetto interno

```
const player = {
    name: 'Darth Vader',
    type: 'Sith Lord',
    stats: {
        health: 15,
        power: 30
    }
}

const name = "Obi Wan";
console.log(name);

//name esiste già, rinominiamo a myName
const {name: myName, type, stats} = player;
console.log(myName, type, stats);

//Destructuring partendo da un oggetto
nested
const {power} = player.stats;
console.log(power);
```

JavaScript - Objects destructuring

- Utilizzando in combinazione con la destrutturazione di un oggetto anche lo spread operator

  `...<nomeVariabile>`

  si può ottenere la parte rimanente dell'oggetto (quella non destrutturata)

- Il nome della variabile scelto in questo particolare caso è completamente arbitrario

```
const player = {
    name: 'Darth Vader',
    type: 'Sith Lord',
    stats: {
        health: 15,
        power: 30
    }
}

//destrutturiamo solo il nome e mettiamo
il resto in una variabile
const {name, ...resto} = player;
console.log(name, resto);
```

JavaScript - Objects shallow copy

- La shallow copy di un oggetto è un clone del nostro oggetto di partenza. Attenzione, in questo tipo di copia non vengono clonati gli oggetti interni, ma solo le proprietà del primo livello.

- Questo significa che se viene fatta una modifica ad un oggetto interno partendo dal riferimento originale, la copia vede la stessa modifica!

- In JavaScript è possibile ottenere delle shallow copy in più modi, ad esempio utilizzando la funzione:

```
Object.assign({}, oggettoDaClonare)
```

oppure sfruttando lo spread operator:

```
...oggettoDaClonare
```

JavaScript - Objects shallow copy, esempio

```
const gameCharacter = {
    name: 'Darth Vader',
    type: 'Sith Lord',
    stats: {
        health: 15,
        power: 30
    }
}

const shallow = Object.assign({}, gameCharacter);
const shallow2 = {...gameCharacter};

//le proprietà di primo livello sono clonate
gameCharacter.name = "Emperor Palpatine";
console.log(gameCharacter, shallow, shallow2);

//le proprietà nested non sono clonate!
gameCharacter.stats.health = 20;
console.log(gameCharacter, shallow, shallow2);
```

JavaScript - Objects deep copy

- La deep copy di un oggetto è un clone a tutti gli effetti del nostro oggetto di partenza. In questo caso anche le proprietà di oggetti interni sono copiate.

- Questo significa che se viene fatta una modifica ad un oggetto interno dal riferimento originale, la copia non è influenzata in nessun modo.

- In JavaScript è possibile ottenere una deep copy, unicamente utilizzando un trucco; trasformare l'oggetto in una rappresentazione json (stringa) e riconvertirlo ad oggetto.

```
JSON.parse(JSON.stringify(oggettoDaClonare))
```

JavaScript - Objects deep copy, esempio

```
const player = {
    name: 'Darth Vader',
    type: 'Sith Lord',
    stats: {
        health: 15,
        power: 30
    }
}

const deep = JSON.parse(JSON.stringify(player));

player.stats.health=66;
console.log(player, deep);
```

JavaScript - Functions

- In JavaScript è possibile definire delle funzioni in 3 modi diversi:
  - Function Declarations
  - Function Expressions
  - Arrow Functions

- Solo le functions declarations possono essere chiamate prima della loro definizione (Hoisting).

- Le funzioni in JavaScript sono oggetti! La differenza è che le funzioni sono callable (possono essere eseguite).

JavaScript - Function Declarations

- La sintassi per una function declarations è la seguente:

  `function nomeFunzione(listaArgomenti){}`

- Le functions declarations sono l'unica tipologia di funzione che può essere chiamate prima della loro definizione (Hoisting).

```
// Function declaration
function playerCreator(){
    console.log("playerCreator Function Declaration");
}

// Function call
playerCreator();
```

JavaScript - Function Expressions

- La sintassi per una function expression è la seguente:

```
const nomeFunzione = function(listaArgomenti){}
```

- Di fatto si sta creando una funzione e assegnando la stessa ad una variabile.

- È possibile usare le function expression sia con funzioni anonime che con funzioni con nome (visibile solo al proprio interno)

- Le function expression permettono di creare funzioni che vengono invocate subito dopo la loro creazione (IIFE: Immediately Invoked Function Expressions).
  Lo scopo di una funzione di questo genere è di ottenere un livello di privacy (le variabili interne non sono accessibili dall'esterno).
  Vengono usate in particolare per implementare alcuni patterns in JavaScript.

JavaScript - Function Expressions, esempi

```
// 1. Function expression, funzione anonima
const playerCreatorAnonymous = function(){
    console.log("playerCreator Anonymous Function Expression")
}

// Function call
playerCreatorAnonymous();


// 2. Function expression, funzione con nome
const playerCreatorNominal = function playerCreatorNominal(){
    console.log("playerCreator Nominal Function Expression")
}

// Function call
playerCreatorNominal();
```

JavaScript - Function Expressions - IIFE example

- Le function expression permettono di creare funzioni che vengono invocate subito dopo la loro creazione (IIFE: Immediately Invoked Function Expressions).

  Lo scopo di una funzione di questo genere è di ottenere un livello di privacy (le variabili interne non sono accessibili dall'esterno).
  Vengono usate in particolare per implementare alcuni patterns in JavaScript.

```
// IIFE
(()=>{
    console.log("I'm an Immediately Invoked Function Expression");
})()
```

JavaScript - Arrow Functions

- La sintassi di base per una arrow function è la seguente:

  ```
  const nomeFunzione = (listaArgomenti) => { }
  ```

- Esistono alcune versioni compatte a dipendenza del numero di parametri ed istruzioni eseguite. Al seguente link maggiori approfondimenti sulle varianti:

  MDN - Arrow Functions

- La principale (ma non unica) differenza con le altre tipologie di funzioni è nel comportamento di `this` al suo interno.

  Una arrow function non ha il proprio `this`, ma si aggancia allo scope della funzione padre (non arrow) più vicina nella gerarchia.

JavaScript - Arrow Functions, esempi

```
// Arrow Function
const playerCreatorArrow = () => {
    console.log("playerCreator Arrow Function")
};

//Function call
playerCreatorArrow();

// Arrow Function Compatta (senza {})
const playerCreatorArrowCompact = () => console.log("playerCreator Arrow Function 2");

//Function call
playerCreatorArrowCompact();
```

JavaScript - Function scope

- In JavaScript ogni funzione crea un nuovo scope
- Le variabili definite all'interno di uno scope sono visibili dagli scope interni (annidati) ma non dagli scope esterni (padri)

  In pratica le variabili definite all'interno di una funzione non sono visibili al di fuori di essa.
- Se esistono variabili con lo stesso nome la precedenza è data a quella dello scope più vicino, partendo da quello locale ed andando verso l'esterno

```
// scope 1
const name = 'Yoda';

function testFunction(){
    //scope 2

    function testFunction2(){
        //scope 3
        const name2 = 'Anakin';
        console.log(name, name2);
    }

    testFunction2();

}

testFunction();
```

JavaScript - Function closures

- Una closure è la combinazione di una funzione ed uno scope contenente dei riferimenti (variabili)

- Tecnicamente, ogni funzione in javascript genera una closure, ma sono particolarmente interessanti quando la funzione viene chiamata da uno scope differente da quello di definizione.

  Essa mantiene i riferimenti allo scope di definizione e non quello di esecuzione!

- Vengono usate tipicamente per ottenere la privacy di variabili.

JavaScript - Function closures

```
function createStarWarsCharacter(name){
    // questa variabile è di fatto privata, non accessibile dall'esterno
    const description = `STAR_WARS_SHIP: ${name}`;

    return function(pilotName){
        // closure, description viene legata allo scope della funzione
        return description + ', PILOT: ' + pilotName;
    }

}

const shipFunction = createStarWarsCharacter('TIE-FIGHTER');
console.log(shipFunction("Trooper 1"));
console.log(shipFunction("Trooper 2"));
console.log(shipFunction("Trooper 3"));
console.log(shipFunction("Trooper 4"));
```

JavaScript - Callbacks

- In JavaScript una funzione è un oggetto, è quindi possibile passare una funzione come argomento di un altra funzione.
- Si parla di callbacks quando una funzione, invoca al suo interno una funzione che le è stata passata come argomento
- È un meccanismo che viene utilizzato tipicamente in scenari di asincronia (es timers, eventi, promises, ...)

JavaScript - Callbacks

```
// funzione che riceve una funzione come parametro (fn)
function manipulateName(name, fn){
    console.log(`name to manipulate: ${name}`);

    // esecuzione della callback
    return fn(name);
}

const result = manipulateName('Obi Wan Kenobi', (name) => {
    // logica della callback passata come argomento
    return `CHARACTER: ${name.toLowerCase().split(' ').join('_')}`;
});

console.log(result);
```

JavaScript - Callbacks, esempi

```
// la funzione setTimeout accetta una funzione come argomento
setTimeout(() => {
    console.log("I'm the callback being executed!")
}, 2000);
```

```
// addEventListener richiede una funzione di callback
// da chiamare allo scatenarsi dell'evento
document.querySelector('.logo').addEventListener('click', () => {
    console.log("Clicked on the logo!")
})
```

JavaScript - `this`

- La parola `this` all'interno di una funzione rappresenta il contesto di esecuzione (environment)
- Il contesto di esecuzione può essere modificato tramite funzioni come `call`, `apply` e `bind` (vedi dettagli su MDN)
- Le arrow functions non forniscono un proprio `this` ma si rifanno a quello del contesto padre
- Nel contesto globale `this` si riferisce al global object (window)
  In `strict mode` (default con webpack) `this` rimane invece `undefined`
- Attenzione viene chiamata come metodo di un oggetto, `this` si riferisce all'oggetto sul quale è chiamata, indipendentemente da dove è definita

JavaScript - `this` - esempio nelle funzioni

```
<script>
 // sloppy mode (non strict)
 var a = "A"

 function testThis(){
     // global object (window)
     console.log(this);
     console.log(this.a);

     function test2(){
         console.log(this);
     }
 }

 testThis();

 // cambiamo this
 testThis.call({a: 'AAA'});

 </script>
```

JavaScript - `this` - esempio in metodi

- Attenzione quando viene chiamata come metodo di un oggetto, `this` si riferisce all'oggetto sul quale è chiamata, indipendentemente da dove è definita

```
const charachter = {

    name: 'name',

    //definita direttamente nell oggetto
    myFunc(){
        console.log(this);
    }

}

charachter.myFunc();
```

```
const charachter = {
    name: 'name'
}

// definita all'esterno ma aggiunta
// e chiamata sull'oggetto
function myFunc() {
    console.log(this);
}

charachter.fn = myFunc;

myFunc();
charachter.fn();
```

JavaScript - Metodi Array: `map`, `filter`, `reduce`

- Esistono svariati metodi utili per lavorare con gli array in JavaScript.
  La lista completa può essere visionata al seguente link:

  https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array

- Tra quelli disponibili alcuni dei più utili per manipolare gli array sono `map`, `filter` e `reduce`

JavaScript - Metodi Array, `map`

- Il metodo `map` permette di generare un nuovo array che viene popolato applicando la **funzione** fornita (callback) ad ogni elemento dell'array di partenza. In pratica, gli elementi ritornati dalla funzione vengono collezionati nell'array risultante.

- Assumiamo ad esempio di avere un array di oggetti contenti nome ed età e di voler ottenere un array solo con le età.
  Come visibile nell'esempio si può fornire a `map` una funzione (callback) che di ogni elemento ritorna l'età

```
const people = [
    {name: 'Obi-Wan Kenobi', age: 57},
    {name: 'Darth Vader', age: 45},
    {name: 'Yoda', age: 900},
    {name: 'Luke Skywalker', age: 23},
]

const ages = people.map((element) => {
    return element.age;
});

console.log(ages);
```

JavaScript - Metodi Array, `filter`

- Il metodo `filter` permette di generare un nuovo array che viene popolato applicando una **condizione** espressa con una **funzione** fornita (callback) ad ogni elemento dell'array di partenza.
  In pratica, vengono filtrati e collezionati nell'array risultante solo gli elementi che passano il controllo.

- Assumiamo ad esempio di avere un array di oggetti contenti nome ed età e di voler ottenere un array solo con le persone di età maggiore a 50.
  Come visibile nell'esempio si può fornire a `filter` una funzione (callback) che esprime la condizione e funge da filtro

```
const people = [
    {name: 'Obi-Wan Kenobi', age: 57},
    {name: 'Darth Vader', age: 45},
    {name: 'Yoda', age: 900},
    {name: 'Luke Skywalker', age: 23}
]

const oldies = people.filter((element) => {
    return element.age > 50;
});

console.log(oldies);
```

JavaScript - Metodi Array, `reduce`

- Il metodo `reduce` permette di eseguire una funzione reducer su ogni elemento di un array. La funzione da fornire (callback) prende come parametri obbligatori un accumulatore (per default all'inizio è il primo elemento) e l'elemento corrente.

  In pratica ad ogni iterazione viene assegnato all'accumulatore il valore ritornato dalla funzione. Il nuovo accumulatore verrà passato alla prossima iterazione.

- Al termine delle iterazioni `reduce` ritorna il singolo valore accumulato.

- Assumiamo ad esempio di avere un array numeri e di voler ottenere la somma.
  Come visibile nell'esempio si può fornire a `reduce` una funzione (callback) che somma all'accumulatore il valore corrente

```
const ages = [57, 45, 900, 23]

const reducer = (accumulator, element) => {
    return accumulator + element;
}

const ageSum = ages.reduce(reducer);

console.log(ageSum);
```

Riepilogo

- Variabili: `var`, `let`, `const`
- Truthiness
- Operatore Ternario
- Template Literals
- Object Shorthand Properties
- Object Destructuring
- Shallow vs Deep Copy
- Functions: declaration, expressions, arrow
- Scope
- Closures
- Callbacks
- this
- Arrays methods: `map`, `filter`, `reduce`

Fonti e Link Utili

Fonti

- Todd Motto - UltimateCourses, JavaScript Basics: https://ultimatecourses.com/courses/javascript
- David Flanagan - JavaScript The Definitive Guide, Master the World's Most-Used Programming Language - O'Reilly Media (2020)
- Eric Elliot - Programming JavaScript Applications - O'Reilly Media
- Mozilla Develop Network: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide

Link Utili:

- nodejs: https://nodejs.org/en/
- babel: https://babeljs.io
- eslint: https://eslint.org
- webpack: https://webpack.js.org