1 Lezione del 21-10-24

1.1 JavaScript

Il linguaggio Javascript è fondamentale allo sviluppo web moderno. Il Javascript è un linguaggio di scripting (oggi perlopiù compilato JIT) orientato agli oggetti (attraverso il paradigma dei prototipi), a tipizzazione debole. Viene usato principalmente come linguaggio di scripting lato client, anche se alcuni framework permettono di usarlo come lingaggio lato server (Node.js).

Viene introdotto nel 1996 da Brendan Eich per Netscape Navigator (oggi Firefox). Nel 1997, l'ECMA standardizza l'ECMAScript, che è sià un sottoinsieme che un sovrainsieme del Javascript. L'ultima versione di rilievo in quanto a cambiamenti è l'ES6 (ha introdotto classi, iteratori e promesse), mentre la più recente è l'ES11 (o ES2020).

In Javascript una **variabile** è un'oggetto che può avere proprietà e metodi. Anche una funzione è considerata un'oggetto.

1.1.1 Scripting lato client

Per scripting lato client si intende codice che viene scaricato e quindi eseguito localmente sulla macchina client, e non sul server e restituito sotto forma di risposte HTTP.

- **Vantaggi:**
  - Elaborazione in locale significa meno carico sul server;
  - Il client può rispondere più velocemente all'utente di quanto potrebbe farlo un ciclo richiesta-risposta HTTP da parte del server;
  - Javascript può interagire direttamente con l'HTML della pagina, creando un'esperienza di sviluppo più simile a quella del software desktop.
- **Svantaggi:**
  - Non c'è garanzia che il client abbia abilitato Javascript, cioè ogni funzionalità assolutamente necssaria deve avere ridondanze lato server;
  - Le pagine che usano il Javascript in maniera massiccia sono difficili da mantenere;
  - Javascript non tollera errori. Se i file HTML e CSS possono essere invalidi, il browser annullerà l'esecuzione nel caso di errori del codice Javascript;
  - Javascript è correntemente supportato da quasi tutti i browser, ma gli standard e le API sono in continua evoluzione. Questo significa che le ultime funzionalità non potrebbero essere disponibili subito su tutti i browser.

1.2 Javascript e HTML

Il javascript può essere inserito in diversi modi nei nostri documenti HTML:

- **Inline:** si riferisce alla pratica di inserire del codice Javascript direttamente all'interno di attributi di elementi HTML. Ad esempio:

```
<a href="JavaScript:OpenWindow();">for more info</a>
<input type="button" onClick="alert('Are you sure?');"/>
```

- **Embedded:** contenuto all'interno dell'elemento `script`

```
<script>
  alert("Ciao vecchio!");
</script>
```

- **Esterno:** sempre indicato dall'elemento `script`, ma per riferimento, e contenuto effettivamente in un file separato.

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

1.2.1 Utenti senza Javascript

Nel caso l'utente non voglia usare Javascript, si può usare l'elemento `noscript`, che viene abilitato solamente nel caso Javascript sia disattivato.

1.3 Variabili e tipi di dato

Una variabile in Javascript è debolmente tipizzata, ergo non bisogna dichiararne il tipo. Si usano le keyword `var`, `const` (per variabili costanti) o `let`. L'assegnamento può avvenire in qualsiasi momento co l'operatore `=`.

Esistono due tipi di dato:

- **tipi riferimento:** cioè gli oggetti. Contengono effettivamente riferimenti al blocco di dati che contiene l'oggetto;
- **tipi primitivi:** alcuni tipi speciali, che possono comunque essere usati come oggetti. Vengono solitamente immagazzinati direttamente in memoria. Questi sono:
  - **Booleani**, vero o falso;
  - **Numeri**, solitamente double precision su 64 bit;
  - **Stringhe**, di caratteri delimitate da `'` o `"`;
  - **Null**, il tipo nullo;
  - **Undefined**, il tipo non definito (non inizializzato);
  - **Simboli**, nuovo in ES2015, rappresenta un valore unico che può essere usato come chiave.

1.3.1 Oggetti predefiniti

Oltre ad array e funzioni, esistono altri oggetti predefiniti. Alcuni dei più usati sono Object, Function, Boolean, Error, Number, Math, Date, String e Regexp. Useremo alcuni oggetti vitali che non fanno parte della specifica Javascript, ma sono comunque supportati dal browser: questi sono window, console e document.

1.3.2 Concatenazione

Per cooncatenare le stringhe si usa l'operatore `+`:

```
const country = "France";
const city = "Paris";
let msg = city + " is the capital of " + country;
```

1.3.3 Condizionali

Esistono i condizionali a cui siamo abituati dal C++, delimitati da graffe. Notiamo, negli operatori di confronto, i due:

- `===`, uguaglianza **strict**, controlla uguaglianze di tipo e valore (quindi dà falso da tipi uguali in valore ma che richiedono conversioni);
- `!==`, diseguaglianza **strict**, come sopra ma con la diseguaglianza.

1.3.4 Vero e falso

In Javascript, ogni oggetto ha un valore `true` o `false` predefinito. Tutti i valori sono `true` tranne `Null`, `""`, `0`, `NaN` e `undefined`.

1.3.5 Gestione delle eccezioni

Sono previste gestioni delle eccezioni, con i blocchi try-catch del C++

```
try {
  funzione_maligna("Ciao");
}
catch(err) {
  alert("Error: " + err);
}
```

1.3.6 Array

Le arrey possono essere create come letterali, o attraverso la funzione costruttore:

```
const countries = [ "Canada", "France", "Italy" ];
const days = new Array("Lun", "Mar", "Mer", "Gio", "Ven", "Sab", "Dom");
```

Sono previsti anche i loop for-each:

```
for(let day of days) {
  // ...
}
```

Si possono **destrutturare** array attraverso l'operatore di accesso (`[0]`, ...), o in sequenza come:

```
primo = array[0];
secondo = array[1];
// oppure:
let [primo, secondo] = array;
```

1.3.7 Oggetti

Gli oggetti in Javascript vengono gestiti attraverso il paradigma a prototipi, cioè nuovi oggetti vengono creati non dalle classi, ma da altri oggetti. Un oggett è composto da un'insieme di coppie chiave-valore dette **properietà**.

Il modo più tipico di accedere ad un oggetto è attraverso la notazione letterale:

```
const oggetto = {
  chiave1: valore1,
  // ...
  chiaveN: valoreN
}
```

Si possono poi usare o l'operatore punto o le parentesi quadre:

```
oggetto.chiave1;
ogetto["chiave1"];
```

Le proprietà possono poi essere **annidate**:

```
const country1 = {
  name: "Canada",
  languages: { "English", "French" },
  capital: {
    name: "Ottawa",
    population: "400000"
  }
  // ...
}
```

con array o altri oggetti.

Si possono, come le array, **destrutturare** gli oggetti con la notazione comoposta:

```
let {id, title} = photo;
let {id, title, location:{country, city}}
```

1.3.8 Spread

Si può usare la sintassi di **spread** per copiare oggetti da un'array ad un altra. Ad esempio:

```
const foo = {name: client.name, country: photo.country};
```

1.3.9 Notazione JSON

JSON sta per Javascript Object Notation, ed è un modo, analogo all'XML, di rappresentare oggetti del Javascript: La differenza principale fra il Javascript e il JSON e che le proprietà sono racchiuse fra virgolette:

```
const obj = '{
  "name1": "value1",
  // ...
  "nameN": "valueN"
}';
```

Gli oggetti JSON vengono letti attraverso l'oggetto predefinito JSON:

```
text = '{
  "name1": "value1",
  // ...
  "nameN": "valueN"
}';
const obj = JSON.parse(text);
console.log(obj.name1);
```

Il JSON viene usato oggi per rappresentare dati strutturati anche al di fuori del web.

1.3.10 Funzioni come oggetti

Le funzioni in Javascript vengono definite con la parola chiave `function`, seguita dalla nome di funzione ed eventuali parametri. Come in Python, le funzioni non richiedono né un tipo di ritorno né i tipi degli argomenti. Vediamo un'esempio di **dichiarazione di funzione**:

```
function subtotal(price, quantity) {
  return price * quantity;
  //ci si aspetta che prenda due numeri e restituisca un'altro numero
}
```

Potremo invocare questa funzione come:

```
let result = subtotal(2, 3);
```

Si possono definire anche **funzioni espressione**, o *funzioni anonime*. Questo perchè, in Javascript, una funzione è essenzialmente un'oggetto:

```
const warn = function(msg) { alert(msg); };
warn("Questo non restituisce nulla");
```

Si possono specificare **valori di default**. Ad esempio:

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

let bar = foo(3); // bar e' NaN

function foo1(a = 10, b = 0) {
  return a + b;
}

let bar = foo1(3); //bar e' 3
```

In Javascript possono esistere funzioni con **numero variabile di argomenti**, usando la parola chiave `args` e l'operatore di **rest** (...). Ad esempio, si può avere:

```
function conc(...args) {
  let s = "";
  for(let a of args){
    s += a + " ";
    return s;
  }
}
```

Si possono poi avere **funzioni annidate**, quando abbiamo bisogno di funzioni dentro altre funzioni:

```
function calculateTotal(price, quantity) {
  let subtotal = price * quantity;
  return subtotal + calculateTax(subtotal);

  function calculateTax(subtotal) {
    let rate = 0.05;
    return subtotal * rate;
  }
}
```

Attraverso un procedimento di **hoisting**, la funzione `calculateTax` viene automaticamente spostata all'inizio del suo ambito di visibilità corrente (la funzione `calculateTotal`)

Possiamo quindi passare le stesse funzioni come argomenti ad altre funzioni, ovvero fornire delle cosiddette **funzioni callback** (*funzioni di "richiamo"*). Ad esempio:

```
const calculateTotal = function(price, quantity, tax) {
  subtotal = price * quantity:
  return subtotal + tax(subtotal)
}

const calcTax = function calculateTax() {
```

```
    let rate = 0.05;
    return subtotal * rate;
}

calculateTotal(2, 5, calcTax);
```

1.3.11 Metodi

Quelli che erano **metodi** negli altri linguaggi, non sono altro che proprietà funzione in Javascript. Un'oggetto può quindi avere funzioni definite come proprietà al suo interno. Si può usare, come in C++, la parola chiave `this` per riferirsi all'oggetto stesso.

Si possono avere **funzioni costruttrici**, cioè funzioni che si usano con la parola chiave `new` per creare nuovi oggetti. Queste usano, al loro interno, la parola chiave `this` per riferirsi all'oggetto creato.

```
function Customer(name, address) {
  this.name = name;
  this.address = address;
  // ...
}

let customer = new Customer("Marietto Stromboldi", "Via del Guzzuldo 56");
```

Quello che succede quando si esegue questo codice è che la parola chiave `new` crea un'oggetto vuoto, e la funzione costruttrice popola poi questo oggetti con gli attributi necessari.

1.3.12 Funzioni freccia

Le **funzioni freccia**, introdotte in ES6, forniscono una sintassi più concisa per le funzioni anonime. Forniscono anche un metodo per gestire gli errori di visibilità causati dalla parola chiave `this`. Per iniziare, consideriamo una semplice espressione di funzione:

```
const taxRate = function() { return 0.05; };
// questo puo' diventare
const taxRate = () => 0.05;
```

Esistono poi diverse varianti, che ci permettono di esprimere funzioni con livelli variabili di complessità:

Notiamo che nel caso precedente abbiamo usato l'ultima contrazione (senza argomenti).

Le funzioni freccia non forniscono un valore specifico a `this`, ma usano quello dell'ambito parentale corrente.

1.3.13 Visibilità

Esistono 4 livelli di visibilità (o *scopo*) in JavaScript:

- **Visibilità a livello funzione**: anche detta visibiltà locale, specifica a funzioni (ad esempio lo sono gli argomenti di funzione). `var`, `let` e `const` dichiarano variabili a livello visibiltà di funzione. Dobbiamo fare attenzione però, in quanto visibilità a livello funzione non significa visibilità a livello blocco, cioè quello che ci aspetteremo in linguaggi come il C++. Ad esempio, il codice:

```
for (var i = 0; i < helpText.length; i++) {
  var item = helpText[i];
  document.getElementById(item.id).onfocus = function() {
    showHelp(item.help);
  }
}
```

ha un errore semantico, in quanto la variabile `item`, dichiarata come `var` in un blocco non funzione, è effettivamente visibile a livello funzione e va a sovrascriversi ad ogni esecuzione del ciclo for.

- **Visibilità a livello blocco**: incluso all'interno di un blocco di controllo. Dobbiamo stare attenti ad usare soltanto `let` o `const` quando si vogliono dichiarare variabili a visibilità a livello blocco, come appare chiaro dall'esempio precedente. La parola chiave `var` infatti, come abbiamo detto, dichiara variabili visibili a livello funzione o globale. Questa differenza deriva dal fatto che la visibilità a livello blocco, e quindi le relative parole chiave, sono state introdotte solo nella versione ES6 di JavaScript;
- **Visibilità di modulo:** riguarda il codice organizzato in file, o *moduli*, separati. Ogni variabile dicharata in un modulo è privata a quel modulo a meno che non venga **esportata** dal modulo e **importata** da un altro modulo;
- **Visibilità globale**: le variabili definite con visibilità globale (quindi abbiamo detto definite con `var` fuori dallo scopo) sono visibili da ogni parte del codice dopo la loro dichiarazione.

Notiamo che in JavaScript esite il meccanismo della cosiddetta **closure**: cioè, la chiusura di una funzione è un'oggetto che contiene la funzione e la sua catena di visibilità, ergo che conserva il valore delle variabili a lei visibili a tempo di dichiarazione. Possiamo fare l'esempio:

```
function foo() {
  var name = "Pino";
  function saluta() {
    console.log("Ciao " + );
  }
  saluta();
}

/* ... */
foo();
```

Alla chiamata della funzione foo(), viene istanziata la variabile `name`, e poi viene chiamata la funzione `saluta()`. Questa non ha variabili locali, ma vede comunque la variabile `name` dichiarata dalla funzione madre. Ergo, la chiamata di `foo()` risulta nella corretta esecuzione di saluta(), cioè nell'output della stringa `"Ciao Pino"`, anche se effettivamente la funzione `saluta()` chiamata da `foo()` non ha una dichiarazione interna di `name`. Questo meccanismo prende anche il nome di **scoping lessicale**.

1.3.14 Output

Ci vengono fornite alcune funzioni di output predefinite:

- `alert()`: visualizza un messaggio di errore o allerta;
- `prompt()`: visualizza un messaggio con un campo di input di testo;

- `confirm()`: visualizza un messaggio con restituzione di un opzione si/no.

Alternativamente, si può usare:

- `document.write()`: scrive contenuti direttamente nel file HTML;
- `console.log()`: visualizza contenuti nella console Javascript del browser.

Vedremo in seguito metodi più sofisticati di interagire con la pagina e con l'utente.

Sintassi tradizionale	Sintassi freccia

`function() {`
`  // statement`
`}`
	`() => {`
	`  // statement`
	`}`

`function(a, b) {`
`  // statement`
`}`
	`(a, b) => {`
	`  // statement`
	`}`

`function() {`
`  call();`
`}`
	`() => {`
	`  call();`
	`}`

`function(a) {`
`  return value;`
`}`
	`(a) => value;`
	`// o addirittura`
	`a => value;`