Sapienza Università di Roma
corso di laurea in Ingegneria informatica e automatica

**Linguaggi e tecnologie per il Web**

a.a. 2023/2024

# Parte 5

# JavaScript: approfondimenti su JSON, funzioni, classi, errori, regex, Web Storage

Lorenzo Marconi

# JSON

- Acronimo di JavaScript Object Notation
- Formato standard per la serializzazione e de-serializzazione degli oggetti JavaScript
- **Serializzazione**:  trasformazione dell'oggetto JavaScript in stringa
- **De-serializzazione**: trasformazione della stringa in oggetto JavaScript
- JSON è un formato standard (ECMA 404) per lo scambio dei dati

# JSON

- Esempio 1: rappresentazione di un array:

```
["ciao", 45, null, true]
```

- Esempio 2: rappresentazione di un oggetto con due proprietà `prop1` (stringa), `prop2` (booleano):

```
{"prop1": "ciao", "prop2": true}
```

- Esempio 3: rappresentazione di un oggetto con due proprietà: `p1`, il cui valore è l'array dell'Esempio 1, e `p2`, il cui valore è l'oggetto dell'Esempio 2:

```
{"p1": ["ciao", 45, null, true],
 "p2": {"prop1": "ciao", "prop2": true}}
```

# JSON: valori

- Un valore può essere:
  - numero
  - stringa
  - oggetto
  - array
  - `true`
  - `false`
  - `null`

```
valore := oggetto | array | numero | stringa | "true"
          | "false" | "null"
```

# JSON: numeri e stringhe

- I numeri sono rappresentati nella sintassi usuale dei linguaggi di programmazione (possono esere rappresentati sia numeri interi che in virgola mobile)
- Le stringhe sono sequenze di caratteri delimitate da **doppi** apici

# JSON: oggetti

- Un oggetto è una sequenza di coppie chiave-valore (separate da virgole) racchiusa tra parentesi graffe

```
oggetto     := "{" (coppia altreCoppie)? "}"
coppia      := stringa ":" valore
altreCoppie := ("," coppia )*
```

# JSON: array

- Un array è una sequenza di valori separati da virgole e racchiusi tra parentesi quadre

```
array       := "[" (valore altriValori)? "]"
altriValori := ("," valore )*
```

# L’oggetto JSON

- Oggetto JSON contenente due metodi:
  - Metodo per la de-serializzazione:

    `JSON.parse(stringa)`

    trasforma la stringa in oggetto
  - Metodo per la serializzazione:

    `JSON.stringify(oggetto)`

    trasforma l’oggetto in stringa

# JSON.stringify: esempio

```
var mioVerbaleEsame = {
    numero: 123,
    insegnamento: "LTW",
    esami: [ { matr:11, voto:30 },
             { matr:16, voto:27 },
             { matr:35, voto:28 } ]
}

JSON.stringify(mioVerbaleEsame)
```

restituisce la stringa

```
'{"numero":123,"insegnamento":"LTW","esami":[{"ma
tr":11,"voto":30},{"matr":16,"voto":27},{"matr":3
5,"voto":28}]}'
```

# JSON.parse: esempio

```
JSON.parse('{ "nome": "Mario", "annoDiNascita":
1979, "esamiSuperati": ["FI1", "FIS", "LTW"] }');
```

restituisce il seguente oggetto JavaScript:

```
{
      nome: "Mario",
      annoDiNascita: 1979,
      esamiSuperati: ["FI1", "FIS", "LTW"]
};
```

# JSON: escape

- Problema: rappresentare i doppi apici nelle stringhe
- Soluzione: fare l'**escape** del carattere " premettendo ad esso un backslash (\)
- Esempio: `{ "autore": "David Bowie", "titolo" : "\"Heroes\""}`
- Lo stesso problema vale, in generale, in JavaScript!
- Quindi, per rappresentare la stringa JSON di cui sopra in JavaScript (senza usare backtick o singoli apici) bisogna fare il doppio escape:

```
var album = JSON.parse("{ \"autore\": \"David Bowie\", \"titolo\" : \"\\\"Heroes\\\"\"}")
```

# Funzioni in JavaScript

- Anche se `typeof` è in grado di distinguere tra oggetti e funzioni, le funzioni in JavaScript di fatto sono anch'esse **oggetti**
- Possono pertanto, ad esempio, essere assegnate a delle proprietà
- Possono anche essere assegnate a delle variabili:

```
var p = function(a,b) {
  ...
}
```

- Il precedente è un esempio di **funzione anonima** (o **funzione lambda**)

# Funzioni in JavaScript

- La funzione precedente può essere invocata usando il nome della variabile:

```
var x = p(32,5);
```

oppure può essere invocata in questo modo:

```
(function(a,b) {
  ...
})(32,5)
```

# Funzioni in JavaScript

- Tutte le funzioni hanno proprietà e metodi definiti
- Proprietà:
  - **`name`** (nome della funzione)
  - **`length`** (numero di argomenti)
  - …
- Metodi (per invocare la funzione):
  - **`call`** / **`apply`**: da usare per "prendere in prestito" un metodo da un altro oggetto (il secondo passa gli argomenti come un array)
  - **`bind`**: da usare per "vincolare" gli argomenti a un certo contesto

# Funzioni variadiche

- JavaScript permette la definizione di funzioni variadiche, ovvero che prendono in input un numero qualunque di argomenti
- Esempio:

```
function sum(...args) { // args è un array!
  return args.reduce(
    (i,j) => { return i+j; }
  );
}
sum(1,2,3); // restituisce 6
```

# Valore di default

- Le funzioni ammettono valori di default per gli argomenti
- Esempio:

```
function saluta(nome, escl="Ciao") {
  window.alert(`${escl} ${nome}!`);
}
saluta("Mario");
saluta("Luigi","Hey");
```

# Funzioni di callback

- Come gli altri oggetti, le funzioni possono essere passate come parametri nella chiamata ad un'altra funzione (**callback**)
- Esempio:

```
function f(x,y) { … }
function oper(f,a,b) { return f(a,b); }
```

oppure

```
document.form1.bott.onclick = f;
```

# Funzioni restituite da funzioni

- Come gli altri oggetti, le funzioni possono anche essere restituite come risultato di una funzione
- Esempio:

```
var x = function(y) {
  return function (z) {
    return y*z;
  };
};
```

# Arrow functions

- Il simbolo `=>` è usato per una notazione alternativa (detta *fat arrow*) per le funzioni lambda
- Attenzione: *non* è del tutto equivalente alla notazione `function() {…}` (ad es. non ha il binding di `this`)
- Generalmente *non* va quindi usata per creare costruttori o metodi

# Arrow functions

Esempi:

```
x = (s,a,b) => { console.log(s); return a + b; };
```

corrisponde all'assegnazione

```
x = function(s,a,b) { console.log(s); return a + b; }
```

```
x = (a,b) => a + b;
```

corrisponde all'assegnazione

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

```
x = a => a + 1;
```

corrisponde all'assegnazione

```
x = function(a) { return a + 1; }
```

# Array e funzioni lambda

- Le funzioni anonime giocano un ruolo importante in relazione agli array
- I seguenti metodi di un oggetto `Array` accettano in input funzioni lambda:
  - `forEach`
  - `map`
  - `flatMap`
  - `filter`
  - `reduce`

# Metodi = funzioni = proprietà

- I metodi degli oggetti sono funzioni
- Ma le funzioni sono oggetti!
- In realtà quindi anche i metodi sono oggetti
- Più precisamente, ogni metodo è una proprietà che ha come valore una funzione
- In questo senso **non c'è più distinzione tra metodi e proprietà** in JavaScript
- Ogni oggetto è un contenitore di proprietà; i metodi dell'oggetto sono quelle proprietà valorizzate ad oggetti di tipo funzione

# Il metodo addEventListener

- Un esempio tipico di uso di funzioni di callback avviene per il metodo `addEventListener` del DOM
- Questo è un metodo sia dell'oggetto document che degli oggetti element
- Ha due parametri:
  - nome dell'evento (stringa)
  - gestore dell'evento o event handler (funzione)
- (c'è un terzo parametro booleano opzionale con cui si può decidere l'ordine di esecuzione degli event handler in elementi annidati)
- Nota: la funzione di callback non può avere argomenti

# Il metodo addEventListener

Esempio: aggiungo un event handler che gestisce il click sull'elemento che ha id uguale a d1:

```
function f() {alert("hai fatto click sull'elemento con ID=d1");}
document.getElementById("d1").addEventListener("click",f);
```

Lo stesso esempio si può scrivere usando una funzione anonima:

```
document.getElementById("d1").addEventListener("click",
function() {alert("hai fatto click sull'elemento con ID=d1");});
```

Aggiungiamo un event handler a livello di documento (questo gestirà un qualunque click sul documento):

```
document.addEventListener("click",
 function() {alert("hai fatto click sul documento");});
```

# Il metodo addEventListener

```
document.getElementById("d1").addEventListener("click",
  function() {alert("hai fatto click sull'elemento con ID=d1");});
document.addEventListener("click",
  function() {alert("hai fatto click sul documento");});
```

- Cosa succede quando sono stati aggiunti entrambi gli event handler e faccio click sull'elemento che ha id uguale a d1?
- Vengono eseguiti **entrambi** gli event handler
- L'ordine di esecuzione dei due event handler dipende dalla **modalità di propagazione** degli eventi

# Bubbling vs. capturing

- Un evento può verificarsi in due fasi:
  - Fase di **capturing**: l'evento è catturato prima a livello di documento, poi di elemento HTML più esterno e via via fino all'elemento più annidato
  - Fase di **bubbling**: l'evento è catturato prima dall'elemento più annidato, e poi da quelli più esterni, fino ad arrivare al livello del documento
- Il terzo argomento (opzionale) del metodo `addEventListener` può essere usato per decidere in quale delle due fasi eseguire l'event handler
- Se il terzo argomento è false (o è assente), si sceglie event bubbling
- Se il terzo argomento è true, si sceglie event capturing

# Bubbling vs. capturing: Esempio

```
<div id="div1" style="border:solid black">
  1
  <div id="div2" style="border:solid red">
    2
  </div>
</div>
<script>
  function b(event){ alert(event.currentTarget.id + " bubble!"); };
  function c(event){ alert(event.currentTarget.id + " capture!"); };
  for (id of ["div1", "div2"]) {
    document.getElementById(id).addEventListener("click", b, false)
    document.getElementById(id).addEventListener("click", c, true)
  }
</script>
```

# Classi in JavaScript

- Le **classi** sono state introdotte in ES6
- Esempio:

```
class Persona {
  constructor(n,c) {
    this.nome= n;
    this.cognome = c;
  }
}
var o = new Persona("Lorenzo","Marconi");
```

- Una classe in realtà è una funzione (invocabile con `new`)
- A differenza dei costruttori semplici, le classi supportano l'**ereditarietà**

# Classi in JavaScript

```
class Persona {
  constructor(n,c) {
    this.nome= n;
    this.cognome = c;
  }
}
class Studente extends Persona {
  constructor(n,c,voti) {
    super(n,c);
    this.media = …; // calcola la media dei voti
  }
}
var mario = new Persona("Mario","Rossi",[25,18,30]);
```

# L’operatore instanceof

- L’operatore `instanceof` di JavaScript restituisce true se un certo valore (eventualmente specificato tramite una variabile) è istanza di un certo tipo di oggetto
- Esempio:

```
             "" instanceof String; // false
    String("") instanceof String; // false
new String("") instanceof String; // true
```

- Non confondere con `typeof` (che è *unario* e controlla il "tipo", non la classe!)

# Classi, funzioni e hoisting

- Il costrutto `class` è sostanzialmente "zucchero sintattico"
- C'è però una differenza tra l'uso esplicito di class e quello della funzione
- Le classi devono essere definite prima di essere invocate
- Invece le funzioni, così come le variabili, possono essere definite anche dopo essere state invocate
- Infatti l'interprete JavaScript effettua il cosiddetto **hoisting** delle dichiarazioni delle funzioni e delle variabili: esegue tali dichiarazioni all'inizio del blocco in cui si trovano (programma o funzione), indipendentemente dalla posizione di tali dichiarazioni nel codice
- Questo non avviene per le dichiarazioni `class`, `let` e `const`

# Hoisting

Esempio:

```
x = 5;
(function() {
  x = 0;
})();
(function() {
  x = 4;
  var x;
})();
```

Qual è il valore della variabile x alla fine dell'esecuzione?

# Gestione degli errori

- Come in molti altri linguaggi, JavaScript permette di "catturare" gli errori
- Classico blocco try-catch-finally:

```
try {
  … // codice che può generare un errore
}
catch(err) {
  … // gestione dell'eventuale errore
}
finally {
  … // questo blocco verrà eseguito in ogni caso
}
```

- Anche l'utente può lanciare errori (tramite **`throw`**)

# Espressioni regolari

- Usate anche nell'attributo HTML `pattern`
- In JavaScript sono oggetti, definibili come:

```
/pattern/modificatori    //compilata a loadtime
new RegExp(pattern, mod) //compilata a runtime
```

- Modificatori più usati:
  - `g` (global)
  - `m` (multiline)
  - `s` (singleline o dotall)
  - `i` (case insensitive)

# Espressioni regolari: sintassi

- Alternation: `/a|b/` individua `a` o `b`
- Quantificatori: `*` (0 o più), `+` (1 o più), `?` (0 o 1), `{n}`, `{n,m}`
- Character set: `[abc]`, `[0-9]`, `[^A-Z0-9]`
- Alcuni caratteri speciali: `.`, `\w`, `\d`, `\s`, `\n`, `\r`, `\t`, `\b`, `^`, `$`
- Lookaround: `(?=exp)`, `(?<=exp)`, `(?!exp)`, `(?<!exp)`
- Capturing group: `/(a|b)c/`, oltre a `ac` o `bc`, "ricorda" anche di aver "catturato" `a` o `b`
- Non-capturing group: `(?:a|b)c`
- Named group: `(?<name>expr)` dà un nome al gruppo
- Backreference: `\1` si riferisce al primo gruppo
- Escape: `\*` individua l'asterisco, `\\` individua il backslash

# Espressioni regolari: applicazioni

Le espressioni regolari sono usate per vari scopi, ad es.:

1. pattern matching:

```
/b.*d/.test("abcd")   // restituisce true
"abcd".search(/ABCD/) // restituisce -1
"abcd".search(/E|C/i) // restituisce 2
```

2. estrazione di stringhe

```
/(\d)\1(\d)/.exec("007")  // ["007","0","7"]
"007".match(/(\d)\1(\d)/) // ["007","0","7"]
"a0b0c7".matchAll(/\d/g)  // rest. un iterable
```

3. trova e sostituisci

```
"Aba".replace(/(ab|a)a/i,"$1$1") // rest. "AbAb"
"abc".replace(/..?/g,"*")        // rest. "**"
```

# TypeScript

- TypeScript è una estensione di JavaScript (per la precisione di ECMAScript 6) proposta da Microsoft a partire dal 2012
- La principale caratteristica di TypeScript è la **tipizzazione forte**
- E’ realizzata mediante un sistema di annotazione dei tipi
- Il codice TypeScript può essere compilato in JavaScript (e quindi essere eseguito da qualsiasi web browser)

# HTML DOM

- Abbiamo visto finora solo alcuni aspetti del DOM HTML
- In realtà il DOM è una API molto più complessa
  - `https://dom.spec.whatwg.org/`
- Anche la parte degli eventi è molto complessa
- Proposta di standard più recente: UIEvents:
  - `https://w3c.github.io/uievents/`

# Memorizzazione persistente

Per la memorizzazione permanente in script lato client:

- Interazione con il web server (l'applicazione lato server memorizza in modo permanente i dati inviati dal client)
- Uso di cookie o file locali (Web Storage API)
- Uso di database locali (IndexedDB API)

# Web Storage API

- Nuove API in HTML5: Web Storage API
- Due nuovi oggetti:
  - `localStorage`
  - `sessionStorage`
  - Derivano entrambi dal nuovo oggetto `Storage`
- `localStorage` permette la memorizzazione persistente
- `sessionStorage` permette la memorizzazione a livello di sessione

# Memorizzazione persistente

- Entrambi gli oggetti sono utilizzati definendo nuove coppie chiave-valore
- Es. `localStorage.cognome="Rossi"` memorizza una nuova coppia
- I valori assegnabili ad una proprietà dello storage **devono essere di tipo String**
- Per poter memorizzare oggetti arbitrari JavaScript, è necessario serializzarli (ad es. tramite il metodo JSON.stringify)

# Esempio

```
localStorage.cognome = "Rossi";
localStorage.nome = "Mario";
var o = { s1: 5, p2: "ciao" };
localStorage.oggetto=JSON.stringify(o);
alert(localStorage.cognome + " " +
      localStorage["nome"]);
```

# Altri metodi dello storage

Altri metodi dell'oggetto `Storage` (e quindi degli oggetti `sessionStorage` e `localStorage`):

- `getItem(chiave)`
- `setItem(chiave,valore)`
- `removeItem(chiave)`
- `clear()`

L'esecuzione dei metodi `setItem`, `removeItem` e `clear` genera l'evento `storage` (che può essere gestito come gli tutti altri eventi del DOM)

# Gestione dei dati dello storage

Analogie e differenze con i cookie:

- Gli oggetti del Web Storage hanno un limite di dimensione di (almeno) 5 MB (molto maggiore dei cookie)
- I dati contenuti in questi oggetti NON vengono trasmessi automaticamente al server (a differenza dei cookie)
- Ogni dato (proprietà) contenuto negli oggetti storage è visibile solo dagli script provenienti dallo stesso dominio dello script che lo ha creato (same origin policy): questa politica è simile (anche se non esattamente uguale) a quella adottata per i cookie

# IndexedDB API

- Le IndexedDB API rispondono a necessità di memorizzazione permanente lato client che localStorage e sessionStorage non possono soddisfare
- Permettono di usare un Object store (database) lato client
- Non è un database relazionale e non si usa il linguaggio SQL
  - una proposta in tal senso, chiamata Web SQL Database API, non è stata adottata come standard W3C
- Proprietà indexedDB dell'oggetto window
- Per i dettagli della API si rimanda alla documentazione ufficiale: `https://www.w3.org/TR/IndexedDB/`

# User script

- JavaScript può essere usato per definire i cosiddetti **user script** dei browser
- Gli user script sono script che alcuni browser (tramite opportune estensioni) eseguono automaticamente in corrispondenza al caricamento di certe pagine web
- Ogni user script è definito per una classe di pagine (ad esempio, tutte le pagine provenienti da un dominio o da una porzione di un dominio)

# Userscript manager

Le estensioni/plugin dei browser che permettono l'esecuzione di user script sono chiamate **userscript manager**

Principali estensioni di questo tipo:

- Greasemonkey (Firefox)
- Tampermonkey (Chrome, Firefox, Safari, Edge)
- Ace Script (Firefox)
- ….

# Esempio di user script

Vogliamo scrivere uno user script che fa aggiungere automaticamente al browser la visualizzazione (come primo elemento della pagina) dell'immagine del logo Sapienza.

Vogliamo applicare tale script a tutte le pagine web la cui URL inizia con il seguente prefisso:

**`http://www.diag.uniroma1.it/rosati/`**

Tuttavia (come eccezione alla precedente regola) non vogliamo applicare tale script alle pagine web la cui URL inizia con il seguente prefisso:

**`http://www.diag.uniroma1.it/rosati/dmds/`**

# Esempio di user script (Greasemonkey)

```
// ==UserScript==
// @version     1.0
// @name        Esempio 1
// @description Esempio di UserScript
// @match       http://www.diag.uniroma1.it/rosati/*
// @exclude     http://www.diag.uniroma1.it/rosati/dmds/*
// ==/UserScript==
Function aggiungiLogoSapienza(){
  var body1 = document.getElementsByTagName("body");
  var body = body1[0];
  var logoSapienza = document.createElement("img");
  logoSapienza.setAttribute("src",
  "https://www.uniroma1.it/sites/default/files/images/logo/sapienza-big.png");
  body.insertBefore(logoSapienza,body.firstChild);
  alert("Fatto!");
}
aggiungiLogoSapienza();
```

# WebAssembly

- WebAssembly (Wasm o WA) è un linguaggio standardizzato dal World Wide Web Consortium alla fine del 2019
- Come dice il nome, è un linguaggio assembler per le applicazioni Web
- Linguaggio di programmazione di basso livello (per una stacked virtual machine)
- Codice binario
- Permette l'esecuzione efficiente di codice sul lato client
- E' supportato da tutti i principali web browser
- E' basato sul linguaggio asm proposto da Mozilla

# Riferimenti

- JSON:
  - https://www.json.org/json-it.html
- TypeScript:
  - https://www.typescriptlang.org/
- HTML DOM:
  - https://dom.spec.whatwg.org/
- UIEvents (eventi):
  - https://w3c.github.io/uievents/

# Riferimenti

Web storage API:

- https://www.w3.org/TR/webstorage/

Indexed Database API:

- https://www.w3.org/TR/IndexedDB/

WebAssembly:

- https://www.w3.org/TR/wasm-core-1/
- https://webassembly.org/

Greasemonkey:

- http://www.greasespot.net/

Tampermonkey:

- https://tampermonkey.net/