



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Introduzione a Javascript V parte

Fabio Vitali

Corsi di laurea in Informatica e
Informatica per il Management

Alma Mater – Università di Bologna

Oggi parleremo di...

Javascript

- Innovazioni sintattiche di ES 2015
- Gestione della asincronicità in Javascript





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

EcmaScript 2015

EcmaScript 2015

- EcmaScript è la standardizzazione presso ECMA di Javascript.
- Nata come necessità per permettere un po' di codice comune a Javascript di Netscape, Jscript di Microsoft, and VBA Javascript di Microsoft.
- La prima versione è del 1997, poi la versione 3 (1999) ha aggiunto qualche struttura di controllo (come try... catch, errors, etc.) e la versione 5 (2009) ha aggiunto il supporto per JSON.
- La versione 6 (aka EcmaScript 2015) è una major release, con molte nuove feature.
 - Alcune sono solo zucchero sintattico
 - Altre sono feature completamente nuove.
- Non tutto è stato inserito negli interpreti attuali.



ES 2015: funzioni freccia

Un nuovo modo per definire funzioni inline:

Sintassi tradizionale

```
var power = function(a,b) {  
    return Math.pow(a,b);  
}  
var c = power(5,3)
```

Nuova sintassi

```
var power = (a,b) => {  
    return Math.pow(a,b);  
}  
var c = power(5,3)
```

Utile per definire semplici funzioni callback:

Sintassi tradizionale

```
var arr = [1, 2, 3];  
var squares = arr.map(function (x) { return x * x });
```

Nuova sintassi

```
var arr = [1, 2, 3];  
var squares = arr.map(x => x * x);
```



ES 2015: template literals

Una nuova sintassi per definire stringhe multi-linea con interpolazione di variabili.

New way

```
var firstName = 'Jane';  
var x = `Hello ${firstName}!  
How are you  
today?`;
```

```
x vale "Hello Jane!  
How are you  
today?"
```

Tre elementi fondamentali: :

- Backticks come delimitatori: `
- I new line fanno parte della stringa
- Interpolatori: `${varName}`

Si tratta sostanzialmente di zucchero sintattico, ma molto utile per sbarazzarsi dell'incubo delle virgolette annidate.



ES 2015: Definizioni di classe

Un modo semplificato per creare oggetti in maniera retro-compatibile con Java e C++:

Sintassi tradizionale

```
var Shape = function(id, x, y) {  
    this.id = id;  
    this.x = x ;  
    this.y = y  
    this.move = null;  
};  
Shape.prototype.move=function(x, y) {  
    this.x = x;  
    this.y = y;  
};
```

Nuova sintassi

```
class Shape {  
    constructor (id, x, y) {  
        this.id = id  
        this.x = x;  
        this.y = y;  
        this.move = null;  
    }  
    move (x, y) {  
        this.x = x  
        this.y = y  
    }  
}
```

Gli oggetti sono ancora basati sui prototipi, ma le definizioni sono più semplici.



Optional chaining (ES 2020) - 1

Sia dato un array contenente due oggetti come al solito:

```
var persone = [{
  nome:      'Giuseppe',
  cognome:   'Rossi',
  altezza:   180,
  nascita:   new Date(1995,3,12),
  indirizzo: {
    via: { strada: 'Via Indipendenza', numero: '15'
  },
    citta: 'Bologna',
    nazione: 'Italia'
  }
}, {
  nome:      'Andrea',
  cognome:   'Verdi',
  altezza:   175,
  nascita:   new Date(1994,7,9),
  email:     'andrea.verdi@gmail.com'
}] ;
```

il primo oggetto contiene il membro `indirizzo`,
il secondo `no`.



Optional chaining (ES 2020) - 2

Quindi

```
for (var i in persone) {  
    console.log( persone[i].indirizzo.via.numero )  
}
```

mi dà errore:

Uncaught TypeError: Cannot read property 'via' of undefined

Per evitare l'errore dovrei controllare sistematicamente la catena dei campi dell'oggetto:

```
for (var i in persone) {  
    if (persone[i])  
        if (persone[i].indirizzo)  
            if (persone[i].indirizzo.via)  
                console.log( persone[i].indirizzo.via.numero )  
}
```

Optional chaining è un'introduzione di ES 2020 che mi permette di evitare questi controlli sistematici: l'operatore di sequenza '?.', appena uno degli elementi della chain è `undefined`, restituisce `undefined`.

```
for (var i in persone) {  
    console.log( persone[i]?.indirizzo?.via?.numero )  
}
```





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Programmazione asincrona in Javascript

Programmazione asincrona (1/16)

La caratteristica più peculiare e tipica di Javascript, evidente immediatamente, è la asincronicità come filosofia di design.

1. Una richiesta Ajax viene eseguita asincronicamente rispetto alla navigazione della pagina HTML
2. La gestione dei dati ricevuti via Ajax viene eseguita asincronicamente rispetto alla emissione della richiesta
3. La gestione degli eventi dell'utente viene eseguita asincronicamente rispetto alla specifica della funzione callback
4. `setTimeout()` posticipa di n millisecondi l'esecuzione di una funzione
5. ...



Programmazione asincrona (2/16)

Ad esempio:

```
var msg = document.getElementById('msg') ;  
msg.innerHTML = '<p>via!</p>' ;  
window.setTimeout(function() {  
    msg.innerHTML += '<p>1 secondo è passato</p>' ;  
}, 1000) ;
```

Il paragrafo `<p>via</p>` compare giustamente prima di quello della funzione. Ma anche se fosse:

```
var msg = document.getElementById('msg') ;  
window.setTimeout(function() {  
    msg.innerHTML += '<p>0 secondi sono passati</p>' ;  
}, 0) ;  
msg.innerHTML = '<p>via!</p>' ;
```

il paragrafo `<p>via</p>` comparirebbe prima lo stesso.

L'interprete Javascript prima esegue completamente lo script in corso, e DOPO esegue quelli in callback.



Programmazione asincrona (3/16)

Questa è una domanda di un compito di TW di tempo fa:

Qual è il contenuto dell'elemento con id test dopo l'esecuzione della funzione test() dell'esempio seguente? Si assuma che la connessione Ajax abbia restituito un codice "200";

```
let e1 = document.getElementById('test')
e1.innerHTML = ''
function test() {
  e1.innerHTML += "Ho messo ";
  callAjax({
    "url": "http://www.google.com",
    "success": function(data) {
      e1.innerHTML += "l'arrosto
    },
    "error": function (data) {
      e1.innerHTML += "la verdura
    }
  });
  e1.innerHTML += "nel forno " ;
}
```

Hanno risposto TUTTI:

"Ho messo l'arrosto nel forno"

Invece la risposta esatta è

"Ho messo nel forno l'arrosto"

perché la funzione del risultato della chiamata Ajax viene eseguita asincronicamente rispetto allo script di invocazione

Programmazione asincrona (4/16)

Abbiamo esigenze di asincronicità ogni volta che abbiamo l'esigenza di chiamare un servizio sulla cui disponibilità o sui cui tempi di esecuzione non abbiamo controllo.

```
var database = remoteService.setDatabaseAccessData() ;  
var result = database.query("SELECT * FROM hugetable");  
var output = prepareDOM(result);  
document.getElementById("display").appendChild(output);
```

Non ho controllo sui tempi di esecuzione del comando ***database.query***, che potrebbe metterci molto tempo.

Nel frattempo il processo è bloccato in attesa del ritorno della funzione, e l'utente percepisce un'esecuzione a scatti, non fluida, non responsive.



Programmazione asincrona (5/16)

La situazione potrebbe peggiorare se l'esecuzione avesse necessità, a catena, di tante altre richieste esterne non controllabili.

```
function searchProducts(query) {  
  var async = true;  
  var productDB = services.setDBAccess('products', async) ;  
  var opinionDB = services.setDBAccess('opinions', async) ;  
  var twitter = services.setDBAccess('twitter', async) ;  
  
  var products = productDB.search(query);  
  var opinions = opinionDB.search(products) ;  
  var tweets = twitter.search(opinions) ;  
  
  var output = prepareDOM(products, opinions, tweets);  
  document.getElementById("display").appendChild(output);  
}
```



Programmazione asincrona (6/16)

Soluzione 0: codice bloccante e amen

Aspettiamo, ci fumiamo una sigaretta e chiacchieriamo coi vicini.

```
function searchProducts(query) {  
  var async = false;  
  var productDB = services.setDBAccess('products', async) ;  
  var opinionDB = services.setDBAccess('opinions', async) ;  
  var twitter = services.setDBAccess('twitter', async) ;  
  
  var products = productDB.search(query);  
  var opinions = opinionDB.search(products) ;  
  var tweets = twitter.search(opinions) ;  
  
  var output = prepareDOM(products, opinions, tweets);  
  document.getElementById("display").appendChild(output);  
}
```



Programmazione asincrona (7/16)

Soluzione 1: logica server-side

Potrei usare un linguaggio multi-threaded server-side, e fare un'unica richiesta al server, che si occupi di tutti i dettagli.

```
function searchProducts(query) {  
  var DB = services.setDBAccess('all') ;  
  var allData = DB.search(query);  
  var output = prepareDOM(allData.products,  
                           allData.opinions,  
                           allData.tweets);  
  document.getElementById("display").appendChild(output);  
}
```

Ho comunque un'attesa, e non ho nessun particolare vantaggio da Ajax. Inoltre distribuisco la logica dell'applicazione in due luoghi, con evidente complessità della gestione:

- Chi si occupa della gestione delle eccezioni e degli errori?
- Chi si occupa del filtro delle opinioni negative o false?



Programmazione asincrona (8/16)

Soluzione 2: codice asincrono e callback (1/3)

Posso passare una funzione callback come argomento di chiamata a funzione, che viene eseguita alla conclusione del servizio.

```
function searchProducts(query) {  
    var productDB = services.setDBAccess('products') ;  
    productDB.search(query, function(products) {  
        var out = prepareDOM(products);  
        document.getElementById("products").appendChild(out);  
    });  
}
```

Però attenzione! Le callback:

- Non possono restituire valori alla funzione chiamante, ma solo eseguire azioni coi dati ottenuti.
- Sono funzioni indipendenti, e vengono eseguite alla fine dell'esecuzione della funzione che le chiama.
- Non hanno accesso alle variabili locali della funzione chiamante, ma solo a variabili globali e closure.



Programmazione asincrona (9/16)

Soluzione 2: codice asincrono e callback (2/3)

- L'approccio delle callback è comunissimo, ma Javascript è *single thread*
- Anche quando il servizio è molto veloce, le funzioni callback vengono comunque eseguite alla fine del flusso di esecuzione del thread chiamante, e hanno un ambiente indipendente.
- Quindi i flussi asincroni vengono eseguiti sempre e solo alla fine dell'esecuzione del flusso principale.



Programmazione asincrona (10/16)

Soluzione 2: codice asincrono e callback (3/3)

Dopo un po' la cosa si complica. Entriamo nel *callback hell*

```
function searchProducts(query) {
  var productDB = services.setDBAccess('products') ;
  productDB.search(query, function(products) {
    var output = prepareDOM(products);
    document.getElementById("products").appendChild(output);
    var opinionDB = services.setDBAccess('opinions') ;
    opinionDB.search(products, function(opinions) {
      var output = prepareDOM(opinions);
      document.getElementById("opinions").appendChild(output);
      var twitter = services.setDBAccess('twitter') ;
      twitter.search(opinions, function(tweets) {
        var output = prepareDOM(tweets);
        document.getElementById("tweets").appendChild(output);
      }) ;
    }) ;
  }) ;
}
```



Programmazione asincrona (11/16)

Soluzione 3: le promesse (1/2)

Una promessa è un oggetto che, si promette, tra un po' conterrà un valore. La promessa viene creata dalla funzione chiamante e mantenuta dalla funzione chiamata.

```
function search(query) {
    var db = 'products' ;
    return new Promise(function(resolve) {
        var DB = services.setDBAccess(db);
        DB.search(query, function(data) {
            resolve(data);
        })
    });
}
search(query).then(function(data) {
    var output = prepareDOM(data);
    document.getElementById("display").appendChild(output);
});
```



Programmazione asincrona (12/16)

Soluzione 3: le promesse (2/2)

La cosa interessante delle promesse è che possiamo evitare il callback hell con una gestione molto più semplice delle chiamate a catena:

```
function search(data,query) {  
    var productDB = services.setDBAccess('products') ;  
    var opinionDB = services.setDBAccess('opinions') ;  
    var twitter = services.setDBAccess('twitter') ;  
    productDB.search(query)  
    .then(function(data) {  
        return opinionDB.search(data)  
    }).then(function(data) {  
        return twitter.search(data)  
    }).then(function(data) {  
        var output = prepareDOM(products, opinions, tweets);  
        document.getElementById("display").appendChild(output);  
        return output ;  
    });  
};
```



Programmazione asincrona (13/16)

Soluzione 4: generator/yield (1/2)

- Alcune librerie (iniziando con Q) avevano introdotto il concetto di generatore e di yield. Questa è stata poi standardizzata in ES 2016.
- Il generatore è una metafunzione (una funzione che restituisce una funzione che può essere chiamata ripetutamente ed interrotta fino a che ne hai nuovamente bisogno).
- Ha una sintassi particolare con * dopo `function`.
- Il comando `yield` mette in attesa la assegnazione di valore fino a che non si chiude l'esecuzione della funzione chiamata.
- la funzione `next()` fa proseguire l'esecuzione della funzione fino al prossimo `yield`



Programmazione asincrona (14/16)

Soluzione 4: generator/yield

```
function *main(query) {  
  var products = yield productDB.search(query) ;  
  var opinions = yield opinionDB.search(products);  
  var tweets   = yield twitter.search(opinions);  
  var output   = prepareDOM(products, opinions, tweets);  
  document.getElementById("display").appendChild(output);  
}  
var m = main();  
m.next();
```



Programmazione asincrona (15/16)

Soluzione 5: async/await

E' un'introduzione di ES 2017, e sta rapidamente prendendo piede.

```
async function search(data,query) {  
  var productDB = services.setDBAccess('products') ;  
  var opinionDB = services.setDBAccess('opinions') ;  
  var twitter = services.setDBAccess('twitter') ;  
  var products = await productDB.search(query) ;  
  var opinions = await opinionDB.search(products) ;  
  var tweets = await twitter.search(opinions) ;  
  var output = prepareDOM(products, opinions, tweets);  
  document.getElementById("display").appendChild(output);  
  return output ;  
}
```

E' un misto di sincronia e asincronia: l'esecuzione di questa funzione è bloccata finché ogni await risponde, ma non prosegue al successivo finché non risponde la prima.



Programmazione asincrona (16/16)

Soluzione 6: Promise.all

Se ho molte chiamate asincrone indipendenti (cioé in cui non debbo aspettare il risultato di una per richiedere la seconda) posso usare Promise.all().

Nell'esempio seguente assumo che la seconda e la terza chiamata non dipendano dal risultato delle chiamate precedenti:

```
var p1 = new Promise(productDB.search(query));  
var p2 = new Promise(opinionDB.search(query));  
var p3 = new Promise(twitter.search(query));
```

```
Promise.all([p1, p2, p3]).then(function(values) {  
    var output = prepareDOM(values[0], values[1], values[2]);  
    document.getElementById("display").appendChild(output);  
});
```





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Fabio Vitali

Dipartimento di Informatica – Scienze e Ingegneria
Alma mater – Università di Bologna

Fabio.vitali@unibo.it

www.unibo.it