



ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Introduzione a Javascript V parte

Fabio Vitali + Angelo Di Iorio

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.



ES 2015: Ereditarietà

La keyword **extends** permette di esprimere anche ereditarietà in modo più semplice

Anche in questo caso è **zucchero sintattico**, gli oggetti continuano ad essere basati su prototipi

```
class Persona{  
  
    constructor(nome) {  
        this.nome= nome;  
    }  
  
    get nome() {  
        return this.nome;  
    }  
}
```

```
class Studente extends Persona{  
  
    constructor(nome, matricola) {  
        super(nome);  
        this.matricola = matricola;  
    }  
  
    getMatricola() {  
        return this.matricola;  
    }  
}
```



Esercizio

- Scrivere il codice JS per modellare i seguenti dati
 - Ogni calciatore è una persona
 - Ogni calciatore ha segnato 0+ goal in carriera
 - Alcuni calciatori ricoprono il ruolo di portiere
 - Ogni portiere subisce 0+ goal in carriera
- Scrivere un metodo che prende in input un oggetto *calciatore* (o *portiere*) e ne mostra le informazioni all'interno di un DIV che si assume abbia attributo *@id = "info"*





ALMA MATER STUDIORUM
UNIVERSITÀ DI BOLOGNA

Programmazione asincrona in Javascript

Programmazione asincrona

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. ...



Cosa stampa in console?

```
function primo() { console.log("primo"); }
```

```
function secondo() {  
  setTimeout(function() {  
    console.log("secondo");  
  }, 0);  
}
```

```
function terzo() { console.log("terzo"); }
```

```
primo();
```

```
secondo();
```

```
terzo();
```



JS Runtime

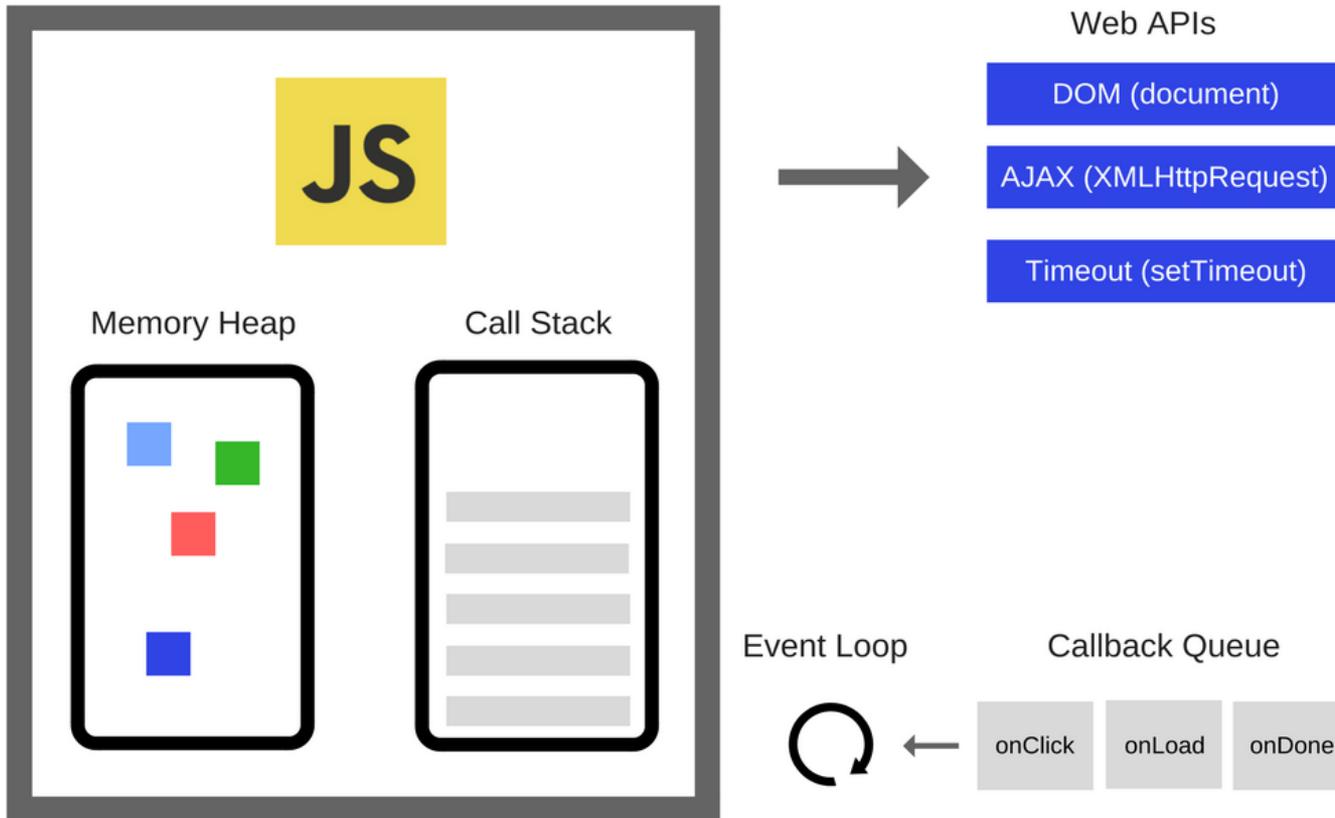
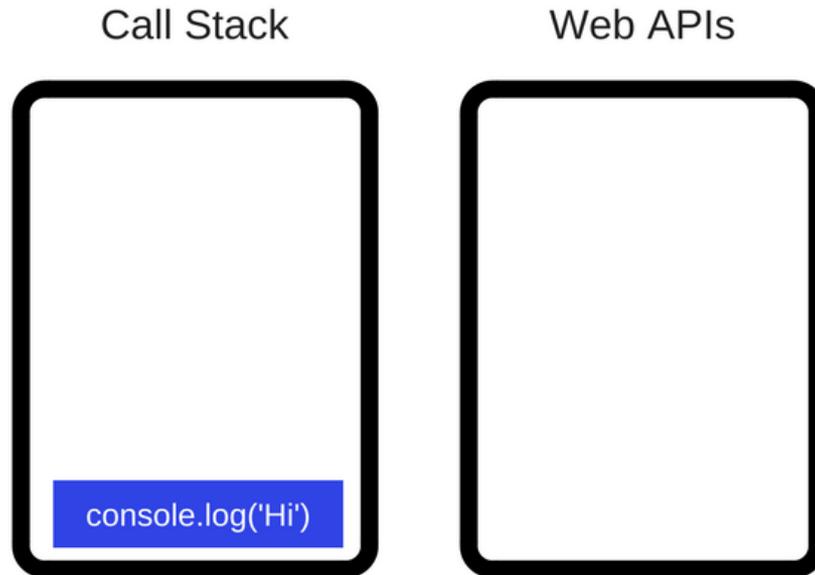
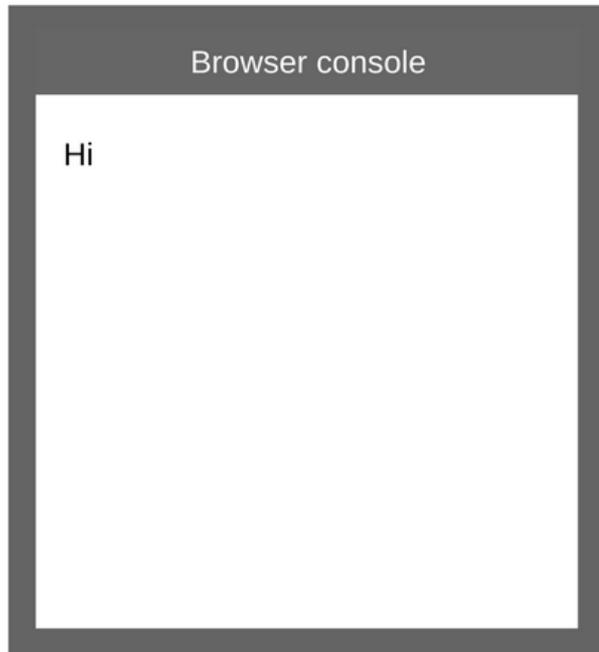


Immagine da “What the is the event loop anyway?” Philip Roberts



Esempio

```
console.log('Hi')  
setTimeout(cb1, 5000)  
console.log('Bye')
```



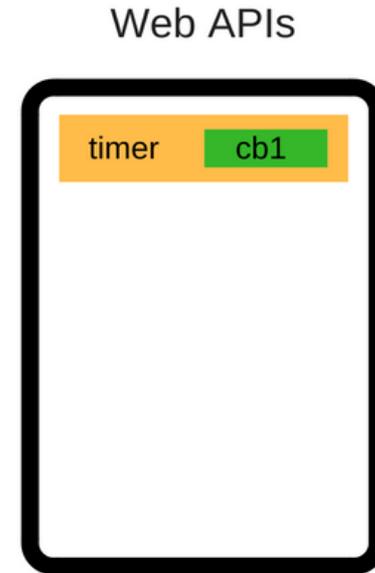
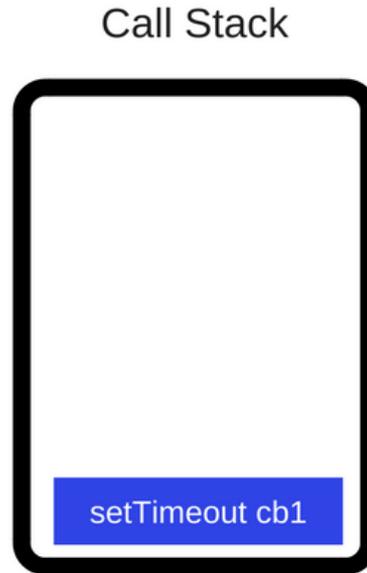
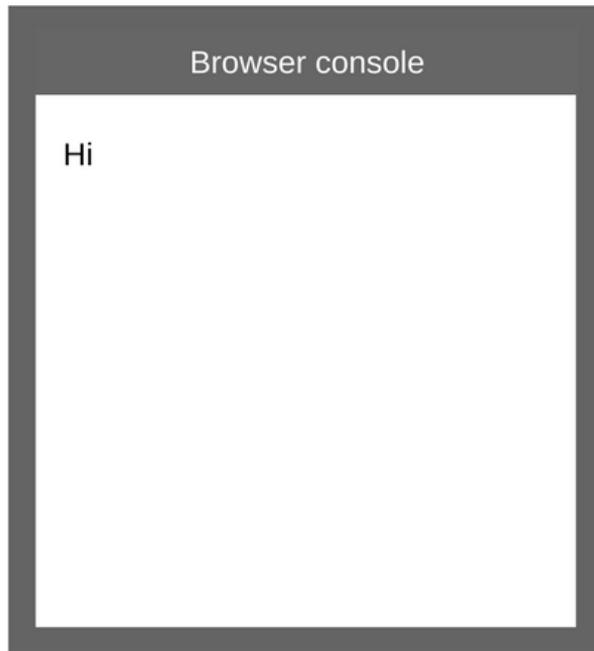
Event Loop

Callback Queue



Esempio

```
console.log('Hi')  
setTimeout(cb1, 5000)  
console.log('Bye')
```



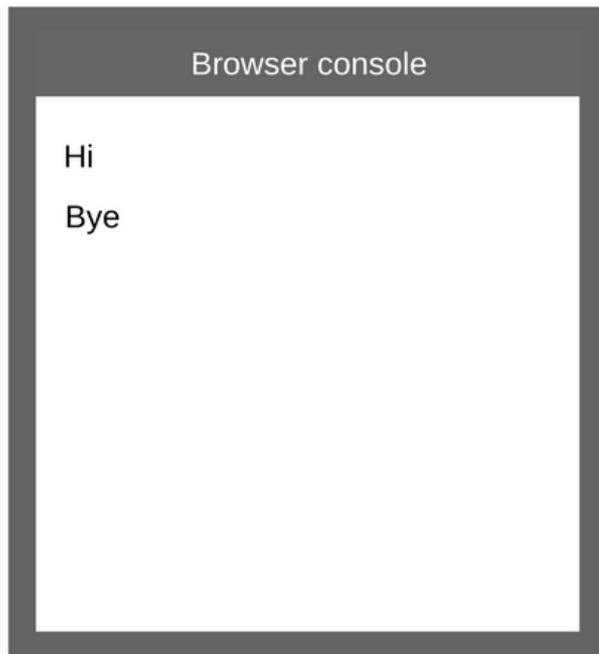
Event Loop

Callback Queue

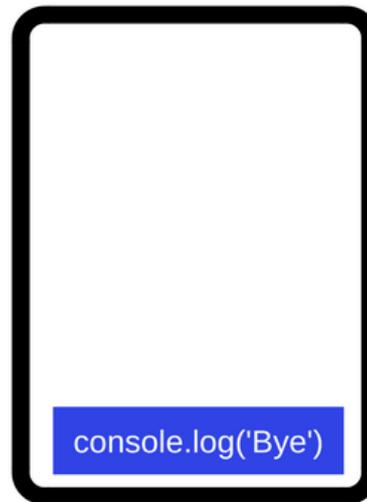


Esempio

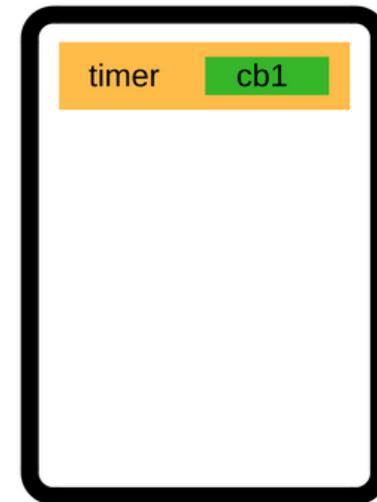
```
console.log('Hi')  
setTimeout(cb1, 5000)  
console.log('Bye')
```



Call Stack



Web APIs



Event Loop

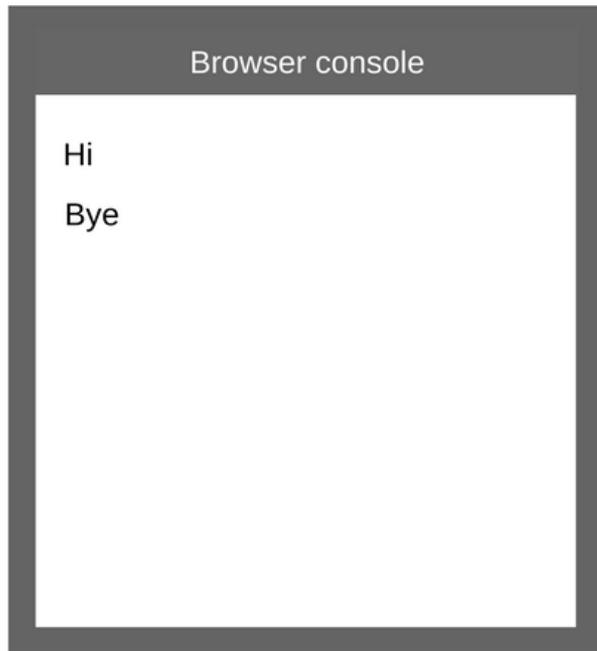


Callback Queue

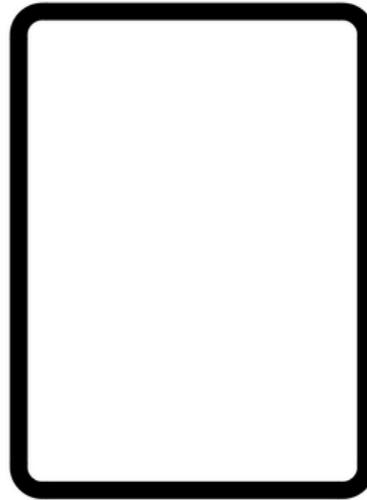


Esempio

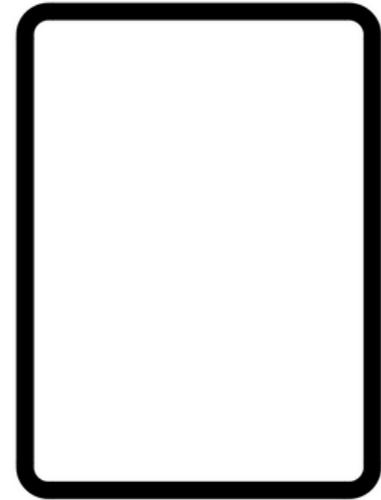
```
console.log('Hi')  
setTimeout(cb1, 5000)  
console.log('Bye')
```



Call Stack



Web APIs



Event Loop



Callback Queue



Programmazione asincrona

Riprendiamo un altro 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

Questa è una domanda di un compito di TW di tempo fa (usa JQuery che ancora non abbiamo visto ma qui non è rilevante):

Qual è il contenuto dell'elemento con id test di una funzione test() dell'esempio seguente? Si assume che abbia restituito un codice "200";

```
function test() {  
    $("#test").append("Ho messo ");  
    var ajax = $.ajax({  
        "url": "http://www.google.com"  
        "success": function(data) {  
            $("#test").append("l'arrosto");  
        },  
        "error": function (data) {  
            $("#test").append("la verdura ");  
        }  
    });  
    $("#test").append("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

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

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

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

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

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

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

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

Soluzione 3: le promesse

Una promessa è un oggetto che, si promette, tra un po' conterrà un valore.

Per operazioni asincrone, quindi, non si restituisce direttamente il risultato ma si promette che l'oggetto restituito conterrà il risultato

La promessa viene:

- creata dalla funzione chiamante, che consumerà i dati prodotti se e quando saranno pronti
- mantenuta dalla funzione chiamata, che produrrà i dati (executor/producer)

Al momento della creazione di una promessa si specifica la funzione da eseguire per produrre i dati

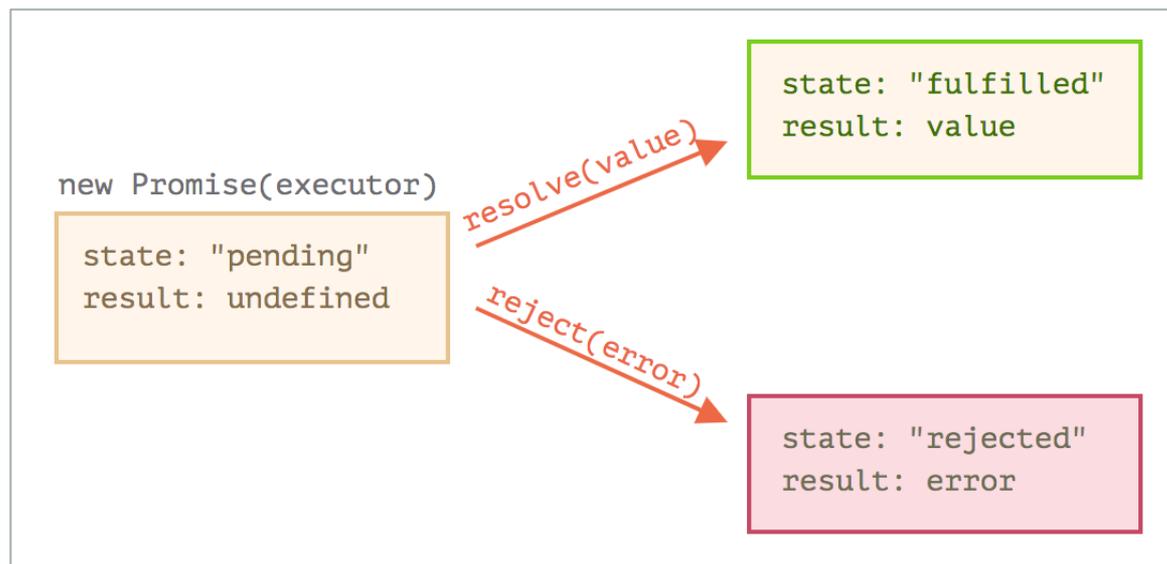


Struttura e stato di una Promise

```
var promise = new Promise( function(resolve, reject) {  
  // executor ("producing" code)  
  // eseguito al momento della creazione di una promessa  
  ...  
  resolve(value) // passa il parametro value alla callback  
  ...  
  reject(error) // passa il parametro error alla callback  
} );
```

resolve: callback da invocare se la promessa è mantenuta

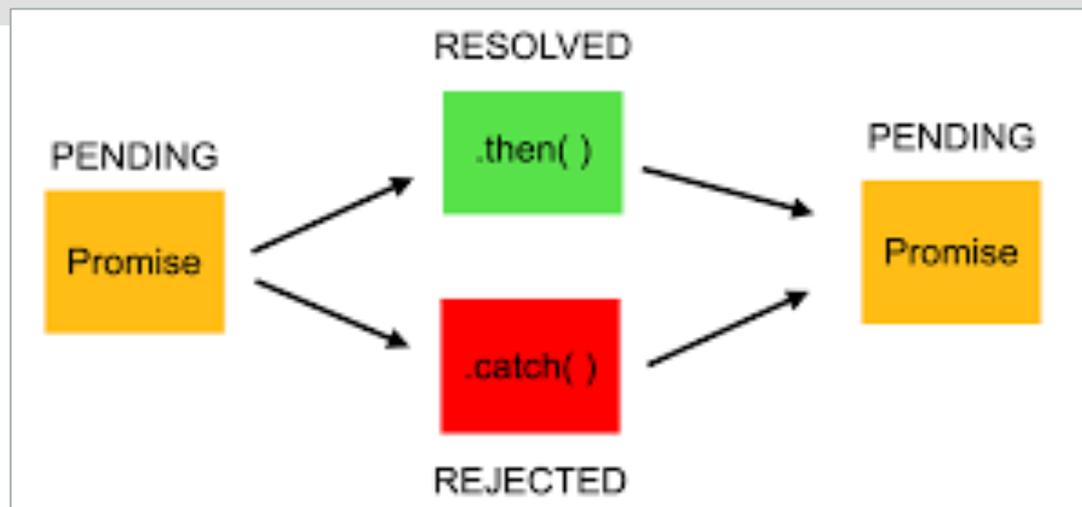
reject: callback da invocare se la promessa non è mantenuta



Promessa mantenuta?

```
promise.then(  
  (res) => { ... }, // funzione eseguita se la promessa è stata  
                    // mantenuta, riceve res dal Producer  
  (err) => { ... } // funzione eseguita se la promessa non è  
                    // stata mantenuta, riceve err dal Producer  
)
```

```
promise.then(  
  (res) => { ... })  
.catch(  
  (err) => { ... }  
)
```



Esempio

```
console.log("primo")
```

```
let promise = new Promise(  
  function(resolve, reject) {
```

//NOTA. Esempio semplice, pensato per guardare struttura e uso di una promessa.
Non ci sono operazioni asincrone!
Nella pratica in questo punto
l'executor produrrà i dati per poi
invocare resolve o reject

```
    resolve("ultimo");  
  });
```

```
console.log("secondo")
```

```
promise.then(  
  (res) => {console.log(res)},  
  (error) => {console.log("ERROR" + error)}  
);
```

Promise Chaining

- Il metodo `.then` restituisce a sua volta una Promise, su cui si può invocare di nuovo `.then` e concatenare promesse
- Il metodo `.catch` può essere invocato alla fine della catena per gestire in un unico punto gli errori

```
promise.then(  
  (res) => {return res + "!!!"} ,  
).then(  
  (res) => {console.log(res)}  
).catch(  
  (err) => { ... }  
)
```



Programmazione asincrona

Soluzione 3: le promesse (1/2)

Tornando all'esempio:

```
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

Soluzione 3: le promesse (1/2)

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(data);
        document.getElementById("display").appendChild(output);
        return output ;
    });
};
```



Programmazione asincrona

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

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

Soluzione 5: async/await

- La keyword `async` trasforma una funzione in una promessa: non restituisce un valore ma "promette" di restituirlo
- Si può usare con tutte le sintassi viste per dichiarare funzioni

```
async function hello(){console.log("Hello")}  
var hello = async function(){console.log("Hello")}  
var hello = async () => {console.log("Hello")}
```
- La keyword `await` può essere usata prima di qualunque funzione che restituisce una promessa e sospende l'esecuzione fino a quando la promessa non è mantenuta
- In caso di errore viene sollevata un'eccezione, come se ci fosse un'istruzione `throw`
- Si può usare SOLO all'interno di una funzione asincrona (`async`)



Programmazione asincrona

Soluzione 5: async/await

```
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 `await` finché non risponde il precedente.



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