

**SUPSI**

# JavaScript: Events, Promises

Patrick Ceppi

## Eventi

- Comportamento tipico di una applicazione web:
  - la maggior parte delle applicazioni reagisce alle azioni dell'utente o a qualche altro avvenimento
  - esegue scambi di dati con uno o più server
  - modifica anche drasticamente la struttura (DOM), i contenuti e il comportamento della propria interfaccia in base ai primi due punti
- Quando la destinazione di un evento è l'oggetto `Window` o un altro oggetto autonomo, il browser risponde a un evento semplicemente invocando gli *handlers* appropriati su quell'oggetto.
- Quando il target dell'evento è un `Document` o un `Element`, tuttavia, la situazione è più complicata.

## Eventi: esempio

- Handler assegnato a `<table>`, ma viene eseguito anche se si clicca su qualsiasi tag innestato, come `<tbody>`, `<tr>`, `<td>`, `<em>` o `<code>`

```
<table onclick="alert('Mi ha cliccato!')">
  <tbody>
    <tr>
      <td>
        <em>Se clicchi <code onclick="alert('Hai cliccato code!')">QUI</code>, il
        gestore (handler) su <code>TABLE</code> viene eseguito.</em>
      </td>
    </tr>
  </tbody>
</table>
```

- In che sequenza verranno eseguiti i due handler?
- Perché l'handler su `<table>` è eseguito, se il click è stato fatto su `<code>`?

## Come si intercetta un evento?

### 1. Direttamente nel markup (bad practice)

```
<div onclick="alert('hello!')"></div>
```

### 2. Aggiungendo la funzione direttamente all'elemento interessato

```
document.querySelector('div').onclick = function() {alert('hello!')}
```

### 3. Utilizzando `addEventListener` (fortemente consigliato)

```
document.querySelector('div').addEventListener('click', function() { alert('hello!') })
```

## Come si intercetta un evento?

1. La prima modalità (`onclick` nel markup) accoppia fortemente il contenuto con il comportamento della pagina e non promuove il riuso del codice
2. La seconda (`elemento.onclick`) non consente di aggiungere facilmente più gestori allo stesso evento e rende molto problematica l'eventuale rimozione di un singolo gestore (*handler*) di eventi
3. La terza (`addEventListener`) consente di aggiungere e rimuovere un numero arbitrario di *event handlers* e di poter scegliere la fase di propagazione in cui eseguire l'azione.

## AddEventListener

- Al metodo `addEventListener` si può passare un ulteriore argomento `useCapture` che se non specificato di default è `false`

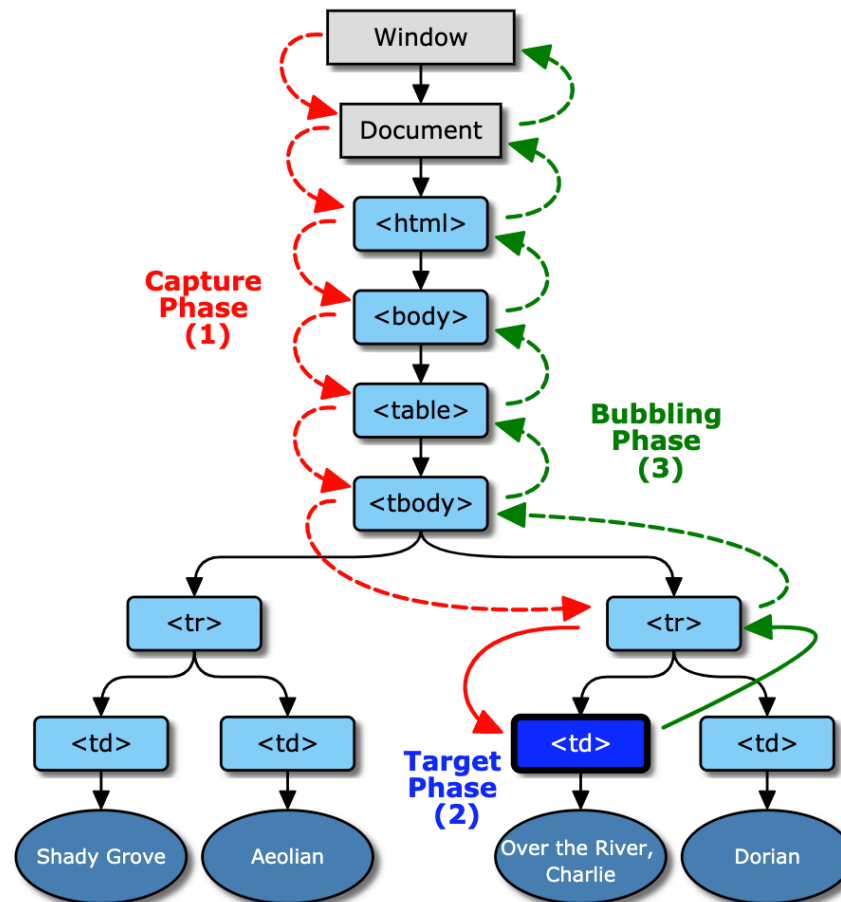
```
elem.addEventListener(..., {capture: true})  
// oppure, l'alias "true"  
elem.addEventListener(..., true)
```

- Indica in che **fase** l'evento deve essere gestito dall'*handler*
- Ma cosa significa fase? ci sono delle fasi?

## Eventi: 3 fasi

- Quando si verifica un evento, l'elemento più annidato in cui si verifica viene chiamato **target element** (`event.target`)
- Cosa succede? (<https://www.w3.org/TR/DOM-Level-3-Events/>)
  - **Capturing** - l'evento si propaga dal contenitore verso i propri discendenti.
    - L'evento si sposta dalla *root* del document fino all'elemento *target* chiamando gli *handlers* assegnati con `addEventListener(..., true)`
  - **At target** - l'evento è esattamente sull'oggetto su cui è stato richiesto.
    - Tutti gli handlers assegnati all'elemento target vengono chiamati
  - **Bubbling** - l'evento si propaga dal nodo su cui è stato agganciato fino alla radice del documento
    - L'evento risale dall'elemento *target* su fino alla *root* del documento, chiamando gli *handlers* assegnati con `on<event>`, attributi HTML o `addEventListener`

## Interazione - eventi e loro propagazione





## Interazione - eventi e loro propagazione

- Ogni handler può accedere alle proprietà dell'oggetto `event`:
  - `event.target`
    - l'elemento che ha originato l'evento (quello in più profondità)
  - `event.currentTarget (=this)`
    - l'elemento corrente che gestisce l'evento
  - `event.eventPhase`
    - la fase corrente (capturing=1, target=2, bubbling=3)
  - `event.stopPropagation()`
    - ogni handler può fermare la propagazione (non raccomandato)

## Interazione - eventi e loro propagazione

```
<table onclick="alert('Mi ha cliccato!')">
  <tbody>
    <tr>
      <td>
        <em>Se clicchi <code>QUI</code>, il gestore (handler) su <code>TABLE</code>
        viene eseguito.</em>
      </td>
    </tr>
  </tbody>
</table>
<script>
  for(let elem of document.querySelectorAll('*')) {
    elem.addEventListener("click", e => alert(`Capturing -> Target:
      ${e.target.tagName} CurrentTarget:${e.currentTarget.tagName}`), true);
    elem.addEventListener("click", e => alert(`${e.eventPhase} -> Target:
      ${e.target.tagName} CurrentTarget: ${e.currentTarget.tagName}`));
  }
</script>
```

## Bubbling, Capturing and removeEventListener

- Bubbling
  - Quasi tutti gli eventi hanno la fase di bubbling, ma non tutti
  - Per esempio l'evento `focus` non viene propagato dal basso verso alto della gerarchia ([https://developer.mozilla.org/en-US/docs/Web/API/Element/focus\\_event](https://developer.mozilla.org/en-US/docs/Web/API/Element/focus_event))
- Capturing
  - la fase di capturing è usata raramente
  - *handlers* aggiunti con `on<event>` o tramite attributi HTML non conoscono questa fase
- `removeEventListener`
  - per rimuovere un handler possiamo usare il metodo `removeEventListener`
  - se l'evento era assegnato alla fase di *capturing* allora bisogna aggiungere l'argomento anche al metodo di rimozione
    - `removeEventListener(..., true)`

## Event delegation

- Bubbling e capturing gettano le basi per la ***event delegation***, una tecnica di gestione degli eventi estremamente potente e efficace
- **Consiste nel collegare l'*handler* di un evento a un solo elemento invece che ai discendenti**
- L'idea:
  - Se abbiamo molti elementi gestiti in modo simile, invece di assegnare ad ognuno un *handler* specifico (e molto simile agli altri), associamo un *handler* unico ad un loro comune elemento padre

## Event delegation: esempio

```
<ul>
  <li data-id="2">Elemento 2</li>
  <li data-id="3">Elemento 3</li>
</ul>
<script>
  document.querySelector("ul").addEventListener('click', function(event) {
    if (event.target.dataset.id != undefined) { // se data-id esiste
      alert(event.target.dataset.id)
    }
  });
</script>
```

- In questo caso riconosciamo l'elemento cliccato utilizzando l'attributo `data` dell'elemento `target`
- In generale, possiamo sfruttare l'API del DOM per capire quale elemento è stato cliccato e se l'elemento è quello che ci interessa gestire l'evento

## Event delegation

- Benefici
  - Semplifica l'inizializzazione e risparmia memoria: non è necessario aggiungere molti gestori
  - Meno codice: quando si aggiungono o rimuovono elementi, non è necessario aggiungere / rimuovere gestori
  - Modifiche DOM: possiamo aggiungere / rimuovere in massa elementi con `innerHTML` e simili (per esempio dopo una chiamata ajax)
- Limitazioni
  - l'evento deve essere in bubbling/capturing. Attenzione all'utilizzo di `event.stopPropagation()`
  - la delegation potrebbe aggiungere un po' di carico alla CPU. Ma di solito il carico è trascurabile

## Javascript asincrono

- Il codice JS in un browser web è in genere **guidato dagli eventi** (si attendono click dall'utente per esempio)
- Anche i server basati su JS (Node.js) in genere aspettano che le richieste dai vari client arrivino prima di fare qualsiasi cosa o devo aspettare una risposta da un database interrogato
- Questo tipo di programmazione asincrona è molto comune in JS (soprattutto per essere efficienti in un'architettura mono thread)
- La modalità di base di **programmazione asincrona** in JS viene eseguita con il meccanismo delle **callbacks**, come abbiamo già visto.
- Un problema ricorrente con codice che deve eseguire chiamate asincrone una dopo l'altra e che il codice diventa difficile da gestire, poco intuitivo e poco leggibile. In questo caso si parla comunemente di **callback hell**.

## Callback hell

```
const a = (callback) => {return callback(0, "a")};
const b = (value, callback) => {return callback(0, value + "b")};
const c = (value, callback) => {return callback(0, "c" + value)};
const handleError = (error) => {console.log(error)}
const doSomething = (value) => {console.log(value)}

a(function(err, resultFromA) {
  if (err) {
    return handleError(err);
  }
  b(resultFromA, function(err, resultFromB) {
    if (err) {
      return handleError(err);
    }
    c(resultFromB, function(err, resultFromC) {
      if (err) {
        return handleError(err);
      }
      doSomething(resultFromC);
    });
  });
});
```

js console  
>> cab

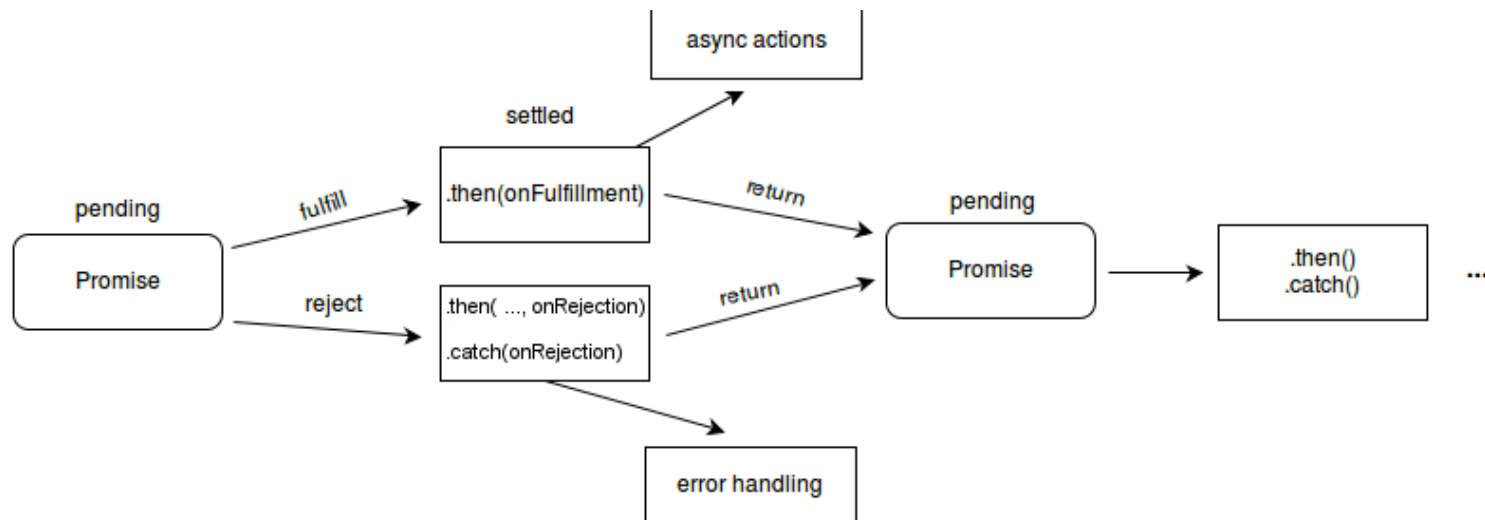


## Promises

- Se non si vuole cadere nella trappola del *callback hell*, le **Promises** sono un altro modo per chiamare un codice di lunga esecuzione e attendere il ritorno di un risultato
- Una Promise è un oggetto che rappresenta il risultato di un calcolo asincrono
  - questo risultato può essere o non essere pronto e non c'è modo di ottenerlo in modo sincrono
  - si può solo chiedere alla Promise di chiamare una funzione di callback quando il valore sarà pronto
- Le Promises sono solo un modo diverso di lavorare con le callback, ma con dei **vantaggi**:
  - consentono di **esprimere callback annidate come una più lineare catena di Promises**
  - **standardizzano un modo per gestire gli errori**, propagandoli correttamente attraverso la catena di Promises

## Promises

- Con le Promises possiamo creare dei metodi asincroni che ritornano un valore come i metodi sincroni, proprio facendoci ritornare una Promise
- Una Promise può essere in uno dei seguenti stati:
  - **pending**: stato iniziale, sospeso: ne soddisfatta, ne rifiutata ancora.
  - **fulfilled**: l'operazione è avvenuta con successo completamente.
  - **rejected**: l'operazione è fallita.



## Esempio: Promise

```
const promise = new Promise((resolve) => {  
  setTimeout(()=> resolve(123), 10000);  
});
```

```
> promise ◀ Eseguito entro 10 secondi  
< Promise {<pending>}  
  > [[Prototype]]: Promise  
    [[PromiseState]]: "pending"  
    [[PromiseResult]]: undefined  
> promise  
< Promise {<fulfilled>: 123}  
  > [[Prototype]]: Promise  
    [[PromiseState]]: "fulfilled"  
    [[PromiseResult]]: 123
```

## Esempio: fetch

- Abbiamo già utilizzato senza saperlo delle Promises, con la funzione fetch:

```
fetch("api/category")
  .then(function(response){
    return response.json();
  })
  .then(function(categories){
    changeDOM(categories);
  })
  .catch(function(error){ // promise rejected
    hanldeError(error);
  })
```

funzioni asincrone  
che ritornano  
una Promise

- Una Promise in sospeso può essere *fulfilled* con un valore o *rejected* con un motivo (errore). Quando si verifica una di queste opzioni, viene chiamata la callback associata ai metodi **then/catch** della Promise.

## From hell to heaven?

```
const a = () => new Promise(function(resolve){resolve("a")});
const b = (value) => value + "b";
const c = (value) => "c" + value;
const handleError = (error) => {console.log(error)}
const doSomething = (value) => {console.log(value)}

export default function nohell() {
  a().then(b).then(c).then(doSomething).catch(handleError);
  return "done";
}
```

```
js console
>> cab
```

- Da notare che il metodo *then* restituisce sempre una *Promise*
- Quando dalla callback passata come argomento nella funzione *then* viene ritornato **un valore** al posto di una *Promise*, allora il valore ritornato sarà:

```
Promise.resolve(<il valore ritornato>)
```

## Promise.all

- `Promise.all()` accetta un array di oggetti `Promise` come input e restituisce una `Promise`. La `Promise` restituita verrà rifiutata se una delle `Promises` viene *rejected*. In caso contrario, verrà *fulfilled* con un array dei valori ritornati da ciascuna `Promise` di input.

```
const urls = [ "/api/products", "/api/users" ];
promises = urls.map(url => fetch(url).then(r => r.text()));
Promise.all(promises).then(bodies => { changeDOM(bodies) }).catch(e => console.error(e));

// bodies è un array, contiene il corpo delle risposte HTTP nell'ordine in cui sono state passate
```

## Promises sequenziali e callback hell again

- Esempio di una fetch che deve aspettare il risultato di una fetch precedente:

```
fetch("api/category/123")
  .then(function(response){
    return response.json();
  })
  .then(function(category){
    fetch("api/products?category="+category.name)
      .then(function(response){
        return response.json();
      })
      .then(function(products){
        changeDOM(products);
      })
  })
```

- Se avete dei pezzi codice simile a questo, non state utilizzando le Promises nel modo corretto

## Promises sequenziali


- Esempio di una fetch che deve aspettare il risultato di una fetch precedente:

```
const fetch1 = () => {  
  return fetch("api/category/123").then((response) => response.json());  
}  
  
const fetch2 = (category) => {  
  return fetch("api/products?category="+category.name)  
    .then((response) => response.json());  
}  
  
fetch1().then(fetch2).then((products) => console.log(products));
```



## async / await

- Le funzioni **async** e la keyword **await** permettono di rendere il codice più leggibile quando abbiamo una concatenazione di Promises
- Rendono il codice asincrono più simile al codice sincrono/procedurale, semplificando l'uso delle Promises
- la keyword **await** prende una Promise e la trasforma in un valore ritornato o in una eccezione lanciata



```
let response = await fetch("/api/user/123");  
  
let profile = await response.json();
```

funzioni che ritornano una Promise

- È importante capire che **await** non causa il blocco del programma e non fa nulla fino a quando la Promise non è risolta (o va in eccezione). Il **codice rimane asincrono** e ciò significa che qualsiasi codice che utilizza **await** è anch'esso asincrono.

## async / await

- Poiché qualsiasi codice che utilizza *await* è asincrono, esiste una regola fondamentale:

**è possibile utilizzare la parola chiave *await* solo all'interno di funzioni dichiarate con la parola chiave *async***

- Esempio:

```
async function getUsername() {  
  let response = await fetch("/api/user/123");  
  let profile = await response.json();  
  return profile.username;  
}
```

## async / await

- Dichiarare una funzione *async* significa che il valore restituito dalla funzione sarà un Promise anche se nel corpo della funzione non vi è alcun codice relativo a una Promise
- Si possono annidare espressioni *await* con funzioni *async* come si desidera, non ci sono limiti
- Ma se si è all'interno di una funzione non *async* allora non si può più usare *await* e bisogna gestire la Promise ritornata:

```
async function getUsername() {  
  let response = await fetch("/api/user/123");  
  let profile = await response.json();  
  return profile.username;  
}  
  
getUsername().then(useUsername).catch(console.error);
```

## 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)
- **Mozilla Develop Network**: <https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide>