

# JavaScript asincrono

- Modello asincrono
  - Callback
  - Promise
- Progetto di riferimento
  - <https://github.com/egalli64/nesp> (*modulo 3d*)
    - Node.js
    - VS Code

# Timer

- Esecuzione differita di una funzione
  - Dopo  $n$  millisecondi: **setTimeout()**
  - Ripetutamente, ogni  $n$  millisecondi: **setInterval()**
- Parametri per entrambe le funzioni timer
  - La funzione che si vuole eseguire in futuro
  - Il delay in millisecondi
  - Eventuali parametri da passare alla funzione che verrà chiamata
- Ritornano entrambe un id usato per annullare l'operazione
  - **clearTimeout()**
  - **clearInterval()**

# Callback

- Funzione passata ad un'altra funzione
  - Non implica necessariamente asincronicità
- Nel modello asincrono il chiamato
  - Viene messo in coda per l'esecuzione e si ritorna il controllo al chiamante
  - Quando il chiamato viene eseguito, può usare callback per notifiche al chiamante
- Esempi di codice standard asincrono che si basa su questo meccanismo
  - `setTimeout(f, 1000)`: aspetta quanto indicato dal secondo parametro, poi invoca la callback
    - passata come primo parametro – in questo caso `f()`
  - AJAX: `XMLHttpRequest`, quando arriva la response viene invocata la callback indicata in
    - `onload`, in caso di successo
    - `onerror`, in caso di errore

# Callback Hell

- Si può rapidamente arrivare a situazioni molto complicate
  - Passaggio di più callback
    - Una per il successo, una per errori, ...
  - Necessità di più chiamate asincrone in sequenza
    - Pipeline, esecuzione di più task in un ordine specificato

Calc esegue operazioni asincronicamente  
Esempio:  $((7 * 5) + 1) / 3$

```
calc(7, 5, mult,  
    res => calc(res, 1, add,  
    res => calc(res, 3, div, printResult, printError),  
    printError),  
    printError);
```

# Promise

- Azione da completare asincronicamente
  - Può produrre un valore, notificando chi interessato del risultato
  - Messa in una coda di eventi, che mantiene l'ordine di esecuzione
- Quando una promessa è creata, è nello stato “**pending**”
- Diventa “**settled**” quando
  - ha avuto successo, è “**fulfilled**”, ritorna un valore che può essere acceduto via **then()**
    - Possiamo avere più callback, ognuna gestita da un **then()**
      - Vede il risultato della precedente, ritorna una nuova promessa che viene a sua volta “settled”
  - o, alternativamente, è stata “**rejected**”
    - il valore ritornato è detto “**reason**” e può essere acceduto via **catch()**
    - Il catch è unico, anche se abbiamo più then(), ogni possibile rejection termina lì


# resolve() / reject()

- `Promise.resolve()` ritorna una promessa *settled*
  - Come parametro può accettare
    - Un valore (o una funzione che ritorna un valore)
      - Le eccezioni vanno gestite sincronicamente in un try-catch dedicato
    - Un oggetto “thenable” → ha un metodo `then()`
      - Se ritorna correttamente la promessa è fulfilled, altrimenti è rejected
    - Una promessa
      - Riduce promesse su più livelli ad un singolo livello
- `Promise.reject()` ritorna una promessa *rejected*
  - La ragione indicata dovrebbe essere un Error
  - Gestita nel metodo **`catch()`** o, in alternativa, nel secondo parametro di **`then()`**

# new Promise

- Pensato per wrappare codice legacy
- Si passa al ctor di Promise una funzione eseguita immediatamente
  - Usata per risolvere la promessa
- Nei then() si mettono le callback
- Nel catch() si gestiscono eventuali errori

```
let delay = ms => new Promise(resolve => setTimeout(resolve, ms));  
  
delay(1000)  
  .then(() => sayHello("Bob"))  
  .catch(err => console.log(err));
```



# async / await

- Keyword che semplificano la gestione delle promesse
- **async**
  - modifica una funzione, assicurando che ritorni una promessa
    - direttamente o per conversione implicita
- All'invocazione di una funzione async possono seguire then() / catch()
- **await**
  - Può riferirsi solo a una promessa
  - Può essere usata solo in una funzione async
  - Resta in attesa del compimento della promessa e ne ritorna risultato
- Vedi l'esempio della fetch per confrontare il codice con e senza async /await