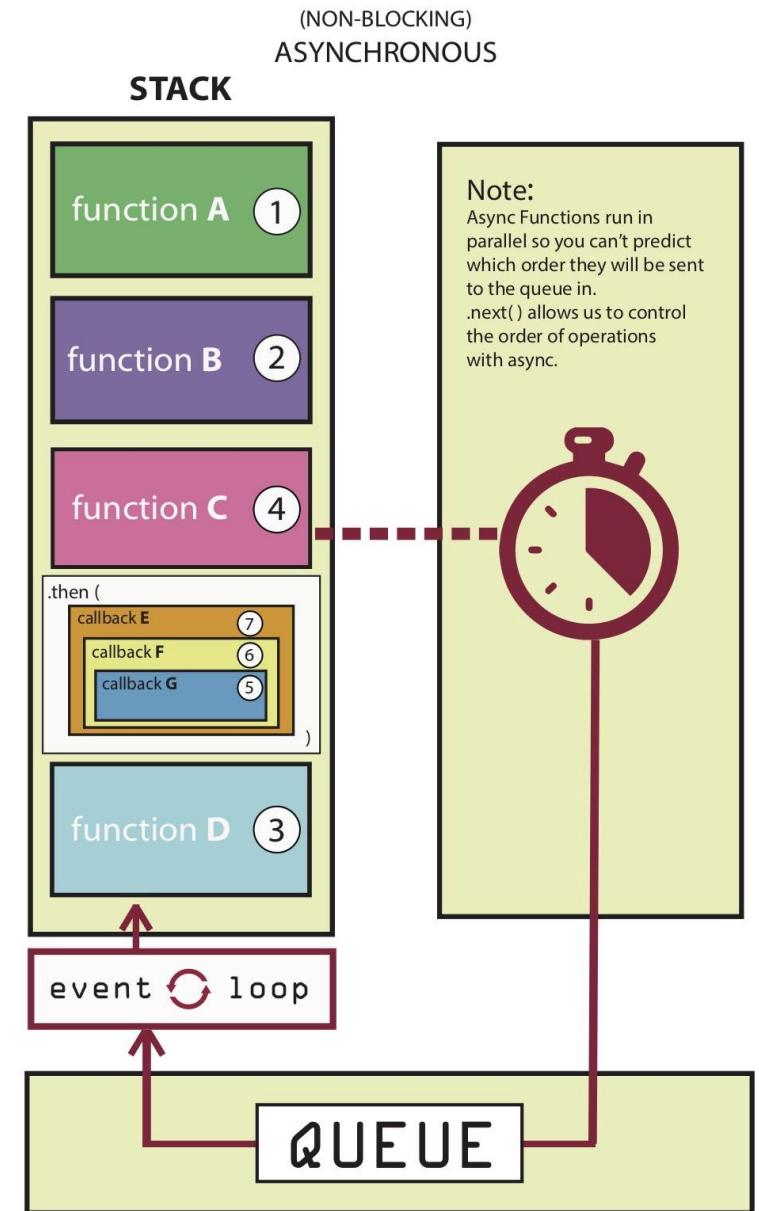


Asynchronous Programming in JS

“The” language of the Web

Fulvio Corno
Luigi De Russis



Outline

- Callbacks
- Functional Programming
- Asynchronous Programming
- Database Access with SQLite
- Promises
- `async/await`



JavaScript – The language of the Web

CALLBACKS

Callbacks

Callback è una funzione che viene passata ad un'altra funzione

https://developer.mozilla.org/en-US/docs/Glossary/Callback_function

posso fare anche databaseQuote, fileQuote... e altre cose

- A callback function is a function passed into another function as an argument, which is then invoked inside the outer function to complete some kind of routine or action.
 - Synchronous
 - Asynchronous

```
function logQuote(quote) {  
    console.log(quote);  
}  
  
function createQuote(quote,  
callback) {  
    const myQuote = `Like I always  
say, '${quote}'`;  
    callback(myQuote);  
}  
  
createQuote("WebApp I rocks!",  
logQuote);
```

Synchronous Callbacks

- Used in functional programming
 - e.g., providing the sort criteria for array sorting

```
let numbers = [4, 2, 5, 1, 3];

numbers.sort(function(a, b) {
  return a - b;
});

console.log(numbers);
```

```
let numbers = [4, 2, 5, 1, 3];

numbers.sort((a, b) => a - b);

console.log(numbers);
```

sincrona: viene eseguita interamente quando viene richiamata!!!!

Synchronous Callbacks

- Example: filter according to a criteria
 - filter() creates a **new** array with all elements for which the callback returns true

```
const market = [  
  { name: 'GOOG', var: -3.2 },  
  { name: 'AMZN', var: 2.2 },  
  { name: 'MSFT', var: -1.8 }  
];
```

JavaScript come motore è : Sincrono, pero riesce a simulare un comportamento asincrono

Non bisogna bloccare il programma, ex: vado sul database per un minuto, quindi simula un comportamento asincrono

funzione generica che filtra secondo qualche criterio

```
const bad = market.filter(stock => stock.var < 0);  
// [ { name: 'GOOG', var: -3.2 }, { name: 'MSFT', var: -1.8 } ]
```

```
const good = market.filter(stock => stock.var > 0);  
// [ { name: 'AMZN', var: 2.2 } ]
```

verrà poi definita dopo simulando un'asincronità



JavaScript: The Definitive Guide, 7th Edition
Chapter 6. Array
Chapter 7.8 Functional Programming

JavaScript – The language of the Web

FUNCTIONAL PROGRAMMING

Functional Programming: A Brief Overview

- A programming paradigm where the developer mostly construct and structure code using *functions*
 - not JavaScript's main paradigm, but JavaScript is well suited
- More “declarative style” rather than “imperative style” (e.g., for loops)
- Can improve program readability:

```
new_array =  
    array.filter ( filter_function ) ;
```

metodo funzionale : riga,
metodo non funzionale : un ciclo (if ec..) che poi avremo nella callback

programmazione funzionale



```
new_array = [] ;  
for (const el of list)  
    if ( filter_function(el) )  
        new_array.push(el) ;
```



Notable Features of the Functional Paradigm

funzioni sono cittadini di prima classe

- Functions are *first-class citizens*
 - functions can be used as if they were variables or constants, combined with other functions and generate new functions in the process, chained with other functions, etc.
- *Higher-order functions*
 - a function that operates on functions, taking one or more functions as arguments and typically returning a new function
- Function *composition* si devono poter comporre le funzioni
 - composing/creating functions to simplify and compress your functions by taking functions as an argument and return an output
- Call *chaining*
 - returning a result of the same type of the argument, so that multiple functional operators may be applied consecutively

Functional Programming in JavaScript

- JavaScript supports the features of the paradigm “out of the box”
- Functional programming requires *avoiding mutability*
 - i.e., do not change objects in place!
 - e.g., if you need to perform a change in an array, return a new array

Non si cambiano metodi in Place, si evitano nella programmazione funzionale pura

ex: meglio copiare gli array... ecc..

Iterating over Arrays

- Iterators: `for ... of`, `for (...;...;...)`
- Iterators: `forEach(f)` ^{iteratore}
 - Process each element with callback `f`
- Iterators: `every(f)`, `some(f)`
 - Check whether all/some elements in the array satisfy the Boolean callback `f`
- Iterators that return a new array: `map(f)`, `filter(f)` ^{operazioni sui array}
 - Construct a new array
- `reduce`: callback function on all items to *progressively* compute a result
`reduce(callback(accumulator, currentValue[, index[, array]])[, initialValue])`
si calcola progressivamente un risultato

.forEach()

iteratore

chiama in maniera sincrona la callback una volta per ogni elemento un oggetto che è iterabile come array

- `forEach()` invokes your (synchronous) callback function once for each element of an **iterable**

operatore di spread

```
const letters = [..."Hello world"] ;  
let uppercase = "" ;  
letters.forEach(letter => {  
    uppercase += letter.toUpperCase();  
});  
console.log(uppercase); // HELLO WORLD
```

come fare FOR letter in letters....
invece di fare FOR nel metodo
classico lo faccio al contrario così:

.forEach()

- `forEach()` invokes your (synchronous) callback function once for each element of an **iterable**

- The callback may have 3 parameters

- **currentValue**: The current element being processed in the array. elemento su cui stiamo iterando sull'array
- **index** (Optional): The index of `currentValue` in the array
- **array** (Optional): The array `forEach()` was called upon. copia dell'array intero

- **Always returns *undefined* and is not chainable**

- No way to stop or break a `forEach()` loop other than by throwing an exception

- `forEach()` does not mutate the array on which it is called

- however, its callback *may* do so

non è contabeneabile con altri metodi, non si può fare nome.qualcosa ec...

forEach non si può fermare, non ha il controllo sull'array e non può avere una callback che modifica quando sta ciclando

`.every()` testa se tutti gli elementi di un array passano una certa condizione

- `every()` tests whether **all elements** in the array **pass the test** implemented by the provided function
 - Callback: Same 3 arguments as `forEach`
 - It returns a Boolean value (*truthy/falsy*)
 - tutti passano : True
 - basta anche solo 1 che non lo passa : False
 - It executes its callback once for each element present in the array until it finds the one where the callback returns a falsy value
 - If such an element is found, **immediately** returns false

```
let a = [1, 2, 3, 4, 5];  
a.every(x => x < 10); // => true: all values are < 10  
a.every(x => x % 2 === 0); // false: not all even values
```

.some()

logicamente fa la stessa cosa:

Ma in questo caso mentre every diceva o tutti o niente questo VERIFICA se almeno 1 lo soddisfa

- **some()** tests whether **at least one** element in the array **passes the test** implemented by the provided function
 - It returns a Boolean value
 - It executes its callback once for each element present in the array until it finds the one where the callback returns a truthy value
 - if such an element is found, **immediately** returns true

```
let a = [1, 2, 3, 4, 5];  
a.some(x => x%2===0); // => true; a has some even numbers  
a.some(isNaN);
```

.map()

Permette di costruire un nuovo Array, passa ogni elemento dell'Array alla funzione che è chiamata
La callback deve restituire qualcosa, e Map torna sempre un nuovo Array contenente i valori della Callback

- `map()` passes each element of the **array** on which it is invoked to the **function** you specify
 - the callback should return a value
 - `map()` always returns a **new array** containing the values returned by the callback

```
const a = [1, 2, 3];  
  
const b = a.map(x => x*x);  
  
console.log(b); // [1, 4, 9]
```

```
const letters = [..."Hello world"];  
  
const uppercase = letters.map(letter  
=> letter.toUpperCase());  
  
console.log(uppercase.join(''));
```

restituisce un nuovo Array } Unisce tutto in un'unica stringa
perchè map va iterato

.filter()

manipola l'array ma il suo scopo è fare un Filtro, passa tutti gli Array di un elemento definito da una funzione
Se non c'è nessun elemento che lo passa viene restituito un Array vuoto

- **filter()** creates a **new array** with all elements that pass the test implemented by the provided function
 - the callback is a function that returns either true or false
 - if no element passes the test, an empty array is returned

```
const a = [5, 4, 3, 2, 1];
```

```
a.filter(x => x < 3); // generates [2, 1], values less than 3
```

```
a.filter((element, index) => index%2 == 0); // [5, 3, 1]
```

seleziona quelli con indice pari

usato: Molto Meno

Reduce combina gli elementi di un Array usando la callback specificata e produce un valore SINGOLO

.reduce()

```
reduce(  
    callback(accumulator, currentValue[, index[, array]])  
    [, initialValue]  
)
```

- **reduce()** combines the elements of an **array**, using the specified function, to produce a **single value**
 - this is a common operation in functional programming and goes by the names “inject” and “fold”
- **reduce** takes two arguments:
 1. the “**reducer function**” (callback) that performs the reduction/combination operation (combine or **reduce 2 values into 1**) il “lavoro” dove riduce due valori in uno solo, REDUCE
 2. an (optional) **initialValue** to pass to the function; if not specified, it uses the first element of the array as initial value (and iteration starts from the next element)
valore iniziale passato alla funzione, un “seed”, che serve per fare la combinazione e se non viene passato usa il primo valore iniziale dell'array

.reduce()

- Callbacks used with `reduce()` are different than the ones used with `forEach()` and `map()`
 - the *first* argument is the **accumulated result** of the reduction so far
 - on the first call to this function, its first argument is the initial value
 - on subsequent calls, it is the value returned by the previous invocation of the reducer function

Se non ci fosse lo stato lo 0 sarebbe partito col 5

```
const a = [5, 4, 3, 2, 1];  
  
                callback di due parametri  
a.reduce( (accumulator, currentValue) =>  
  accumulator + currentValue, 0); par iniziale a 0  
// 15; the sum of the values  
    Restituisce: la somma tra il valore corrente e il comulatore  
  
a.reduce((acc, val) => acc*val, 1);  
// 120; the product of the values  
  
a.reduce((acc, val) => (acc > val) ? acc  
  : val);  
// 5; the largest of the values
```



Array methods cheatsheet

JS tips
@sulco

 `.map( \rightarrow )` \rightarrow 

 `.filter()` \rightarrow 

 `.find()` \rightarrow 

 `.findIndex()` \rightarrow 3

 `.fill(1, )` \rightarrow 

 `.copyWithin(2, 0)` \rightarrow 

 `.some()` \rightarrow true

 `.every()` \rightarrow false

 `.reduce(acc + curr)` \rightarrow 



JavaScript: The Definitive Guide, 7th Edition Chapter 11. Asynchronous JavaScript

Mozilla Developer Network

- [Learn web development JavaScript » Dynamic client-side scripting » Asynchronous JavaScript](#)
- [Web technology for developers » JavaScript » Concurrency model and the event loop](#)
- [Web technology for developers » JavaScript » JavaScript Guide » Using Promises](#)

JavaScript – The language of the Web

ASYNCHRONOUS PROGRAMMING

Asynchronicity

Rimettono in coda per l'esecuzione (Asincronia) i elementi che richiedono un qualcosa (tipo un input) e dopo continua ad eseguire il codice:

-Ex-Sotto

Esegue il codice completo e dopo due secondi ritorna di nuovo lì, e dopo i due secondi ed esegue `deleteAfterTimeout`

- JavaScript is single-threaded and inherently synchronous
 - i.e., code cannot create threads and run in parallel in the JS engine
- Callbacks are the most fundamental way for writing asynchronous JS code
- How can they work asynchronously?
 - e.g., how can `setTimeout()` or other async callbacks work?
- Thanks to the Execution Environment
 - e.g., browsers and Node.js
- and the Event Loop

```
const deleteAfterTimeout = (task) =>
{
  // do something
}
// runs after 2 seconds
setTimeout(deleteAfterTimeout, 2000,
task)
```

fa questo dopo i 2 sec

Fa questo: Grazie a Node

Non-Blocking Code!

- Asynchronous techniques are very useful, particularly for web development
- For instance: when a web app runs executes an intensive chunk of code without returning control to the browser, the browser can appear to be frozen
 - this is called blocking, and it should be the exception!
 - the browser is blocked from continuing to handle user input and perform other tasks until the web app returns control of the processor
- This may happen outside browsers, as well
 - e.g., reading a long file from the disk/network, accessing a database and returning data, accessing a video stream from a webcam, etc.
- Most of the JS execution environments are, therefore, deeply asynchronous
 - with non-blocking primitives
 - JavaScript programs are event-driven, typically

Asynchronous Callbacks

- The most fundamental way for writing asynchronous JS code
- Great for “simple” things!
- Handling user actions
 - e.g., button click
- Handling I/O operations
 - e.g., fetch a document
- Handling time intervals
 - e.g., timers
- Interfacing with databases

solo su node, legge da tastiera -> Dal terminale

-> Assincrona!

```
const readline = require('readline');

const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('How old are you? ', (answer) => {
  let description = answer;

  rl.close();
});
```


Timers

- To delay the execution of a function:

-> Chiama la funzione dopo un certo tempo

- `setTimeout()` runs the callback function after a given period of time
- `setInterval()` runs the callback function periodically

-> Chiama la funzione ogni volta, dopo un intervallo di tempo

```
const onesec = setTimeout(()=> {  
  console.log('hey') ; // after 1s  
}, 1000) ;
```

```
console.log('hi') ;
```

```
const myFunction = (firstParam,  
secondParam) => {  
  // do something  
}  
// runs after 2 seconds  
setTimeout(myFunction, 2000,  
firstParam, secondParam) ;
```

Note: timeout value in ms, $< 2^{31}-1$ (about 24 days)

Timers

- `clearInterval()`: for stopping the periodical invocation of `setInterval`

```
const id = setInterval(() => {}, 2000) ;  
// «id» is a handle that refers to the timer  
  
clearInterval(id) ;
```

Handling Errors in Callbacks

Una BEST PRACTIS

- No “official” ways, only best practices!
- Typically, the first parameter of the callback function is for storing any error, while the second one is for the result of the operation
 - this is the strategy adopted by Node.js, for instance

```
fs.readFile('/file.json', (err, data) => {  
  if (err !== null) {  
    console.log(err);  
    return;  
  }  
  //no errors, process data  
  console.log(data);  
});
```

Data Persistence

DATABASE ACCESS WITH SQLITE

Server-Side Persistence

- A web server should normally store data into a persistent database
- Node supports most databases
 - Cassandra, Couchbase, CouchDB, LevelDB, MySQL, MongoDB, Neo4j, Oracle, PostgreSQL, Redis, SQL Server, **SQLite**, Elasticsearch
- An easy solution for simple and **small-volume** applications is **SQLite**
 - in-process on-file relational database

SQLite



- Uses the 'sqlite' npm module
- Documentation: <https://github.com/mapbox/node-sqlite3/wiki>

```
npm install sqlite3
```

```
const sqlite = require('sqlite3');  
const db = new sqlite.Database('exams.sqlite', // DB filename  
  (err) => { if (err) throw err; });  
  
...  
db.close();
```

SQLite: Queries

```
rows.forEach((row) => {  
    console.log(row.name);  
});
```

- `const sql = "SELECT...";`
- `db.all(sql, [params], (err, rows) => { })`
 - Executes sql and **returns all the rows** in the callback
 - If err is true, some error occurred. Otherwise, **rows** contains the result
 - **rows** is an array. Each item contains the fields of the result

select*from utenti = Se non produce risultati??!!
Allora -> Errore? o rows? No da rows

Quando le righe sono vuote rows da Undefined

<https://www.sqlitetutorial.net/sqlite-nodejs/>

SQLite: Queries

```
rows.forEach((row) => {  
    console.log(row.name);  
});
```

solo il primo risultato in risposta alla query

- db.get(sql, [params], (err, row) => { })
 - Get only **the first row** of the result (e.g., when the result has 0 or 1 elements: primary key queries, aggregate functions, ...)
- **db.each**(sql, [params], (err, row) => { })
 - Executes the callback **once per each result row** (no need to store all of them)

ed.each esegue la callback per ogni risultato

=> ARROW FUNCTIONS OVUNQUEEEEEEEEEEEEE

<https://www.sqlitetutorial.net/sqlite-nodejs/>

SQLite: Other Queries

Usassi un ArrowFuction qua alcune cose non funzionerebbero:

Se si utilizzano `this.changes` o `lastID` i valori sarebbero undefined
e con `=>` ridifenirebbe il loro ID


• `db.run`(`sql`, [`params`], `function` (`err`) { })

- For statement that do not return a value
- CREATE TABLE
- INSERT
- UPDATE
- In the callback function
 - `this.changes` == number of affected rows
 - `this.lastID` == number of inserted row ID (for INSERT queries)
 - Note: To make `this` work correctly in the callback, the arrow function syntax cannot be used here

<https://www.sqlitetutorial.net/sqlite-nodejs/>

Parametric Queries

Viene riempito in ordine con quello che c'è nell'Array di parametri

- The SQL string may contain parameter placeholders: 
- The placeholders are replaced by the values in the [params] array
 - in order: one param per each ?

```
const sql = 'SELECT * FROM course WHERE code=?';  
db.get(sql, [code], (err, row) => {
```

i parametri si passano attraverso il campo parametri

- Always use parametric queries – never string+concatenation nor
`template strings`

---> !! SQL Injection - (Operazioni non volute o malevole)

Example

Table: course

	code	name	CFU
	Filter	Filter	Filter
1	01TYMOV	Information systems security	6
2	02LSEOV	Computer architectures	10
3	01SQJOV	Data Science and Database Technology	8
4	01OTWOV	Computer network technologies and services	6
5	04GSPOV	Software engineering	8
6	01TXYOV	Web Applications I	6
7	01NYHOV	System and device programming	10

Table: score

	coursecode	score	laude	datepassed
	Filter	Filter	Filter	Filter
1	02LSEOV	25	0	2021-02-01

Example

transcript.js

```
const sqlite = require('sqlite3');
const db = new sqlite.Database('transcript.sqlite',
  (err) => { if (err) throw err; });

let sql = "SELECT * FROM course LEFT JOIN score ON course.code=score.coursecode" ;
db.all(sql, (err,rows)=>{
  if(err) throw err ;
  for (let row of rows) {
    console.log(row);
  }
});
```

Example

```
const sqlite = require('sqlite3');
const db = new sqlite.Database('transcript.sqlite',
  (err) => { if (err) throw err; });

let sql = "SELECT * FROM course LEFT JOIN score ON cou
db.all(sql, (err,rows)=>{
  if(err) throw err ;
  for (let row of rows) {
    console.log(row);
  }
});
```

```
{
  code: '01TYMOV',
  name: ' Information systems security ',
  CFU: 6,
  coursecode: null,
  score: null,
  laude: null,
  datepassed: null
}
{
  code: '02LSEOV',
  name: ' Computer architectures ',
  CFU: 10,
  coursecode: '02LSEOV',
  score: 25,
  laude: 0,
  datepassed: '2021-02-01'
}
```

But...

```
const sqlite = require('sqlite3');
const db = new sqlite.Database('transcript.sqlite', (err) => { if (err) throw err; });

let result = [];
let sql = "SELECT * FROM course LEFT JOIN score ON course.code=score.coursecode" ;
db.all(sql, (err,rows)=>{
    if(err) throw err ;
    for (let row of rows) {
        console.log(row);
        result.push(row);
    }
});
console.log('*****');
for (let row of result) {
    console.log(row);
}
```

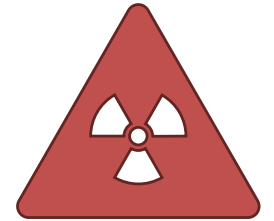
Queries Are Executed Asynchronously

```
CREATE TABLE IF NOT EXISTS "numbers" (  
    "number"    INTEGER  
);  
INSERT INTO "numbers" ("number") VALUES (1);
```

number
1

```
insert into numbers(number) values(1);  
-- Add a new line
```

```
select count(*) as tot from numbers;  
-- Count how many lines we have
```



Queries Are Executed Asynchronously

```
const sqlite = require('sqlite3');
const db = new sqlite.Database('data.sqlite',
  (err) => { if (err) throw err; });

for(let i=0; i<100; i++) {
  db.run('insert into numbers(number) values(1)',
    (err) => { if (err) throw err; });

  db.all('select count(*) as tot from numbers',
    (err, rows) => {
      if(err) throw err;
      console.log(rows[0].tot);
    }) ;
}
db.close();
```

queries.js

...

389

390

391

392

396

396

396

397

398

399

399

400

400

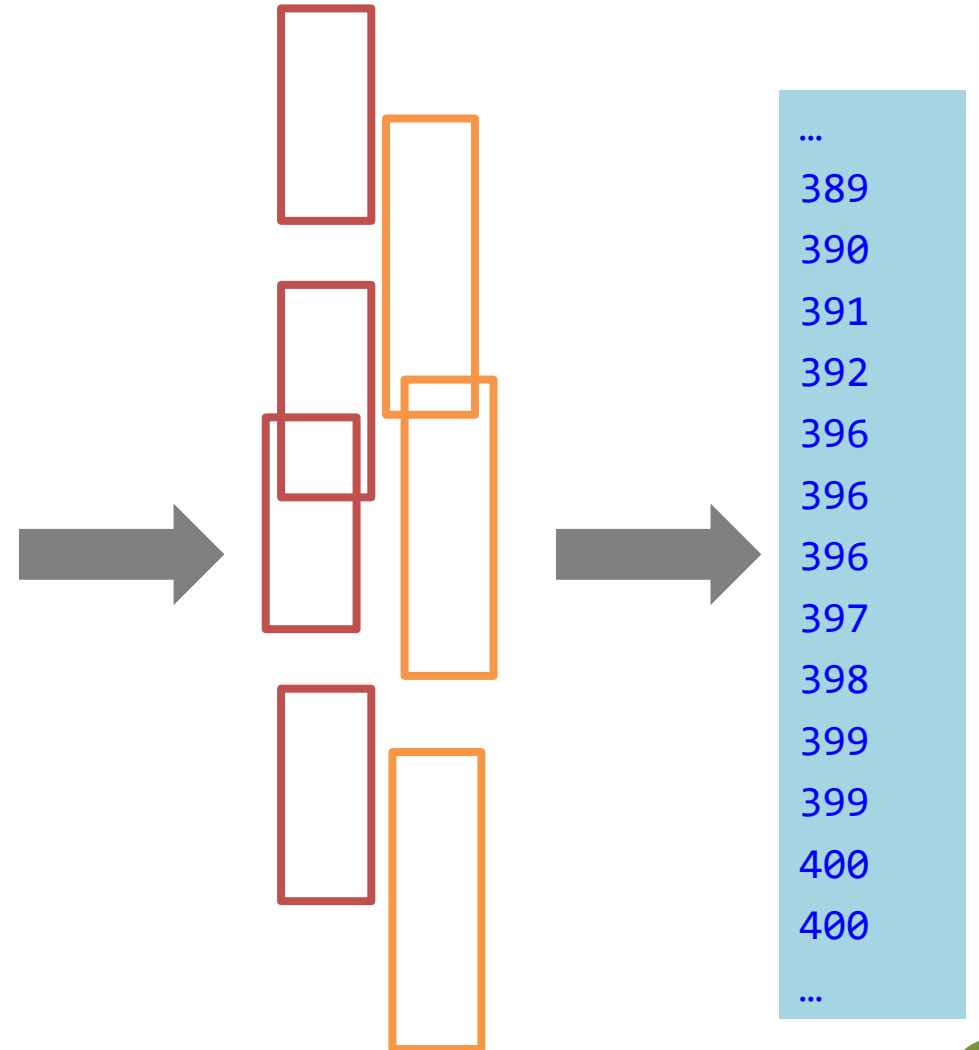
...

Queries are Executed Asynchronously

```
const sqlite = require('sqlite3');
const db = new sqlite.Database('data.sqlite',
  (err) => { if (err) throw err; });

for(let i=0; i<100; i++) {
  db.run('insert into numbers(number) values(1)',
    (err) => { if (err) throw err; });

  db.all('select count(*) as tot from numbers',
    (err, rows) => {
      if(err) throw err;
      console.log(rows[0].tot);
    }) ;
}
db.close();
```



Solution?



```
for(let i=0; i<100; i++) {  
  db.run('insert into numbers(number) values(1)',  
    (err) => { if (err) throw err;  
              else  })  
  db.all('select count(*) as tot from numbers',  
    (err, rows) => {  
      if(err) throw err;  
      console.log(rows[0].tot);  
        
    }) ;  
}
```

A possible solution is in `queries_sync.js`, but it's **not** recommended



JavaScript: The Definitive Guide, 7th Edition Chapter 11. Asynchronous JavaScript

Mozilla Developer Network

- [Learn web development JavaScript » Dynamic client-side scripting » Asynchronous JavaScript](#)
- [Web technology for developers » JavaScript » Concurrency model and the event loop](#)
- [Web technology for developers » JavaScript » JavaScript Guide » Using Promises](#)

JavaScript – The language of the Web

PROMISES

Beware: *Callback Hell*!

- If you want to perform multiple asynchronous actions in a row using callbacks, you must keep passing new functions to handle the continuation of the computation after the previous action
 - every callback adds a level of nesting
 - when you have lots of callbacks, the code starts to be complicated very quickly

```
const readline = require('readline');
const rl = readline.createInterface({
  input: process.stdin,
  output: process.stdout
});

rl.question('Task description: ', (answer) => {
  let description = answer;

  rl.question('Is the task important? (y/n)', (answer) => {
    let important = answer;

    rl.question('Is the task private? (y/n)', (answer) => {
      let privateFlag = answer;

      rl.question('Task deadline: ', (answer) => {
        let date = answer;
        ...
        rl.close();
      })
    })
  })
});
```

Promises

promessa che un oggetto verrà riempito dove poi si potranno fare delle operazioni
Forniscono in modo standard un modo per togliere le callback infinite

- A core language feature to “**simplify**” **asynchronous programming**
 - a possible solution to callback hell, too!
 - a fundamental building block for “newer” functions (async, ES2017)
- It is an **object** representing the **eventual completion** (or **failure**) of an asynchronous operation
 - i.e., an asynchronous function returns *a promise to supply the value* at some point in the future, instead of returning immediately a final value
- Promises standardize a way to handle errors and provide a way for errors to propagate correctly through a chain of promises

Promises

- Promises can be created or consumed
 - many Web APIs expose Promises to be consumed!
- When consumed:
 - a Promise starts in a pending state
 - the caller function continues the execution, while it waits for the Promise to do its own processing, and give the caller function some “responses”
 - then, the caller function waits for it to either return the promise in a fulfilled state or in a rejected state

Creating a Promise

Pending: Gestito in maniera Automatica

- A Promise object is created using the **new** keyword
- Its constructor takes an *executor function*, as its parameter
- This function takes **two functions** as parameters:
 - **resolve**, called when the asynchronous task completes successfully and returns the results of the task as a value
 - **reject**, called when the task fails and returns the reason for failure (an error object, typically)

```
const myPromise =  
  new Promise((resolve, reject) => {  
  
    // do something asynchronous which  
    eventually call either:  
  
    resolve(someValue); // fulfilled  
  
    // or  
  
    reject("failure reason"); // rejected  
  
  });
```

Creating a Promise

- You can also provide a function with “promise functionality”
- Simply have it return a promise!

```
function waitPromise(duration) {  
  // Create and return a new promise  
  return new Promise((resolve, reject) => {  
    // If the argument is invalid,  
    // reject the promise  
    if (duration < 0) {  
      reject(new Error('Time travel not yet  
implemented'));  
    } else {  
      // otherwise, wait asynchronously and then  
      // resolve the Promise; setTimeout will  
      // invoke resolve() with no arguments:  
      // the Promise will fulfill with  
      // the undefined value  
      setTimeout(resolve, duration);  
    }  
  });  
}
```


Consuming a Promise

- When a Promise is fulfilled, the `then()` callback is used
- If a Promise is rejected, instead, the `catch()` callback will handle the error
- `then()` and `catch()` are instance methods defined by the Promise object
 - each function registered with `then()` is invoked only once
- You can omit `catch()`, if you are interested in the result, only

Stayci

si possono concatenare luccy kapss

```
waitPromise().then((result) => {  
    console.log("Success: ", result);  
}).catch((error) => {  
    console.log("Error: ", error);  
});  
  
// if a function returns a Promise...  
waitPromise(1000).then(() => {  
    console.log("Success!");  
}).catch((error) => {  
    console.log("Error: ", error);  
});
```

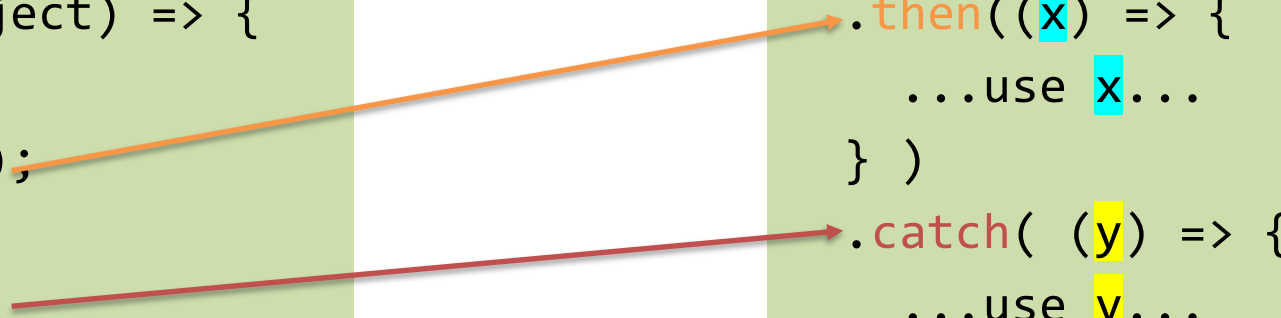
Consuming a Promise

- `p.then(onFulfilled[, onRejected]);`
 - Callbacks are executed asynchronously (inserted in the event loop) when the promise is either fulfilled (success) or rejected (optional)
- `p.catch(onRejected);`
 - Callback is executed asynchronously (inserted in the event loop) when the promise is rejected
- `p.finally(onFinally);`
 - Callback is executed in any case, when the promise is either fulfilled or rejected.
 - Useful to avoid code duplication in then and catch handlers
- All these methods return **Promises**, too! \Rightarrow They can be **chained**

Promise: Create & Consume

```
const prom = new Promise(  
  (resolve, reject) => {  
    ...  
    resolve(x);  
    ...  
    reject(y);  
    ...  
  }  
)
```

```
prom  
  .then((x) => {  
    ...use x...  
  } )  
  .catch( (y) => {  
    ...use y...  
  } ) ;
```



Chaining Promises

- One of the most important benefits of Promises
- They provide a natural way to express a sequence of asynchronous operations as a **linear chain of `then()`** invocations
 - **without having to nest** each operation within the callback of the previous one
 - the "callback hell" seen before
- **Important:** always return results, otherwise callbacks won't get the result of a previous promise

```
getRepoInfo()  
  .then(repo => getIssue(repo))  
  .then(issue => getOwner(issue.ownerId))  
  .then(owner => sendEmail(owner.email,  
    'Some text'))  
  .catch(e => {  
    // just log the error  
    console.error(e)  
  })  
  .finally(_ => logAction());  
});
```

Example: Chaining

- Useful, for instance, with I/O API such as `fetch()`, which returns a Promise

```
const status = (response) => {  
  if (response.status >= 200 && response.status < 300) {  
    return Promise.resolve(response) // static method to return a fulfilled Promise  
  }  
  return Promise.reject(new Error(response.statusText))  
}  
const json = (response) => response.json()
```

```
fetch('/todos.json')  
  .then(status)  
  .then(json)  
  .then((data) => { console.log('Request succeeded with JSON response', data) })  
  .catch((error) => { console.log('Request failed', error) })
```

Promises... in Parallel

```
Promise.all(promises)
  .then(results => console.log(results))
  .catch(e => console.error(e));
```

- What if we want to execute several asynchronous operations in parallel?

- **Promise.all()** prende tutte le promis inserite come argomento della funzione e se tutte le promis sono state

basta 1
non risolta
non ritorna
nessun risultato
l'argomento delle
promise all
puo essere qualcosa
che non è una
promise

– takes an array of Promise objects as its input and returns a Promise true filed: questo metodo ritorna un array con tutte le promise risolte

– the returned Promise will be rejected if at least one of the input Promises is rejected

– otherwise, it will be fulfilled with an **array of the fulfillment values** for each of the input promises

– the input array can contain non-Promise values, too: if an element of the array is not a Promise, it is simply copied unchanged into the output array

- **Promise.race()** ritorna una promis che viene fulfilled o rifiutata quando viene fulfilleda o rifiutata una
 - returns a Promise that is fulfilled or rejected when **the first** of the Promises in the input array is fulfilled or rejected
 - if there are any non-Promise values in the input array, it simply returns the first one



React!



JavaScript: The Definitive Guide, 7th Edition Chapter 11. Asynchronous JavaScript

Mozilla Developer Network

- Learn web development JavaScript » Dynamic client-side scripting » Asynchronous JavaScript
- Web technology for developers » JavaScript » Concurrency model and the event loop
- Web technology for developers » JavaScript » JavaScript Guide » Using Promises

JavaScript – The language of the Web

ASYNC/AWAIT

2 keyword semplificate per la scrittura di codice asincrono

Sono due keyword da mettere prima delle funzioni

Simplifying Writing With `async / await`

- ECMAScript 2017 (**ES8**) introduces two new keywords, **`async`** and **`await`**
 - write promise-based asynchronous code that *looks like* synchronous code
- Prepend the **`async`** keyword to any function means that it will return a **Promise**
- Prepend **`await`** when calling an **async function** (or a function returning a Promise) makes the calling code stop until the promise is resolved or **rejected**

```
const sampleFunction = async () => {  
  return 'test'  
}
```

non torna esplicitamente una promise, quindi il then funziona

```
sampleFunction().then(console.log) // This will log 'test'
```


async Functions

- The **async** function declaration defines an **asynchronous function**
- Asynchronous functions operate in a separate order than the rest of the code (via the event loop), **returning an implicit Promise** as their **result**
 - but the syntax and structure of code using async functions looks like standard synchronous functions.

```
async function name([param[, param[, ...param]]]) {  
    statements  
}
```

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/async_function

await

- The `await` operator can be used to wait for a Promise. It can *only be used inside an async function*
- `await` **blocks** the code execution within the async function **until the Promise is resolved**
- When resumed, the **value** of the `await` expression is that of the **fulfilled Promise**
- If the Promise is rejected, the `await` expression throws the rejected value
 - If the value of the expression following the `await` operator is not a Promise, it's converted to a resolved Promise

```
returnValue = await expression ;
```

<https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/await>

Example: async / await

```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}  
  
async function asyncCall() {  
  console.log('calling');  
  const result = await resolveAfter2Seconds();  
  console.log(result);  
}  
  
asyncCall();
```

} Return a
promise

} async is needed to use await

} Looks like
sequential
code

```
> "calling"  
//... 2 seconds  
> "resolved"
```

Example: async / await

```
function resolveAfter2Seconds() {  
  return new Promise(resolve => {  
    setTimeout(() => {  
      resolve('resolved');  
    }, 2000);  
  });  
}  
  
async function asyncCall() {  
  console.log('calling');  
  const result = await resolveAfter2Seconds();  
  return 'end';  
}  
  
asyncCall().then(console.log);
```

} Implicitly returns a Promise

} Can use Promise methods

```
> "calling"  
//... 2 seconds  
> "end"
```

Examples... Before and After

```
const makeRequest = () => {  
  return getAPIData()  
    .then(data => {  
    console.log(data);  
    return "done";  
  })  
};  
}
```

```
let res = makeRequest();
```

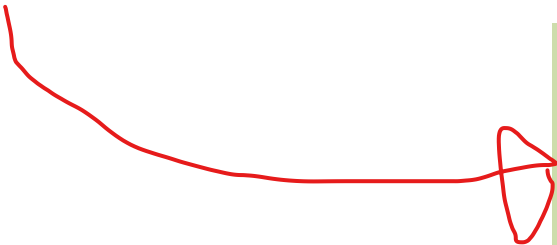
```
const makeRequest = async () => {  
  console.log(await getAPIData());  
  return "done";  
};
```

```
let res = makeRequest();
```

Examples... Before and After

```
function getData() {  
  return getIssue()  
    .then(issue => getOwner(issue.ownerId))  
    .then(owner => sendEmail(owner.email, 'Some text'));  
}
```

// assuming that all the 3 functions above return a Promise



```
async function getData = {  
  const issue = await getIssue();  
  const owner = await getOwner(issue.ownerId);  
  await sendEmail(owner.email, 'Some text');  
}
```

Chaining with async/await

- Simpler to read, easier to debug
 - debugger would not stop on asynchronous code

```
const getFirstUserData = async () => {  
  const response = await fetch('/users.json'); // get users list  
  const users = await response.json(); // parse JSON  
  const user = users[0]; // pick first user  
  const userResponse = await fetch(`/users/${user.name}`); // get user data  
  const userData = await user.json(); // parse JSON  
  return userData;  
}  
getFirstUserData();
```

Promises or async/await? Both!

asy-w è piu utilizzato e piu comodo
asy-w non ha meccanismi di utilizzo o di attesa quindi in quel caso mejo le promis

- If the output of `function2` is dependent on the output of `function1`, use `await`.
- If two functions can be run in parallel, create two different async functions and then run them in parallel `Promise.all(promisesArray)`
- Instead of creating huge async functions with many `await asyncFunction()` in it, it is better to create **smaller** async functions (not too much blocking code)
- If your code contains blocking code, it is better to make it an async function. The callers can decide on the level of asynchronicity they want.

<https://medium.com/better-programming/should-i-use-promises-or-async-await-126ab5c98789>

SQLite... revisited

```
function insertOne() {  
    return new Promise( (resolve, reject) => {  
        db.run('insert into numbers(number) va  
lues(1)', (err) => {  
            if (err) reject(err);  
            else resolve('Done');  
        });  
    });  
}
```

```
function printCount() {  
    return new Promise( (resolve, reject) => {  
        db.all('select count(*) as tot from nu  
mbers', (err, rows) => {  
            if(err)  
                reject(err);  
            else {  
                console.log(rows[0].tot);  
                resolve(rows[0].tot);  
            }  
        });  
    });  
}
```

SQLite... revisited

```
function insertOne() {  
    return new Promise( (resolve, reject) => {  
        db.run('insert into numbers(number) va  
lues(1)', (err) => {  
            if (err) reject(err);  
            else resolve('Done');  
        });  
    });  
}
```



```
async function main() {  
    for(let i=0; i<100; i++) {  
        await insertOne();  
        await printCount();  
    }  
    db.close();  
}
```

```
main() ;
```


```
function printCount() {  
    return new Promise( (resolve, reject) => {  
        db.all('select count(*) as tot from nu  
mbers',  
            (err, rows) => {  
                if(err)  
                    reject(err);  
                else {  
                    console.log(rows[0].tot);  
                    resolve(rows[0].tot);  
                }  
            }) ;  
    }) ;
```

Beware The Bug!

```
async function main() {  
  for(let i=0; i<100; i++) {  
    await insertOne();  
    await printCount();  
  }  
  db.close();  
}  
  
main() ;
```



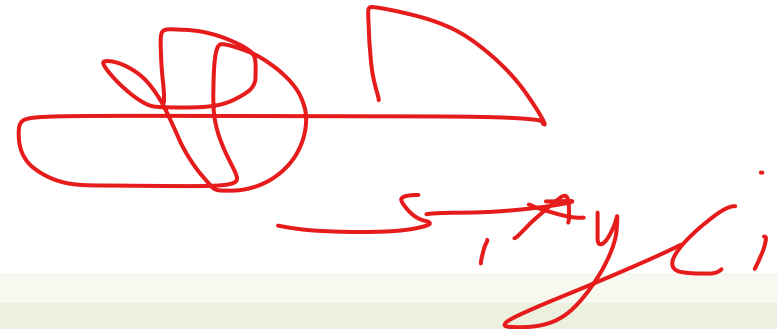
```
async function main() {  
  for(let i=0; i<100; i++) {  
    await insertOne();  
    await printCount();  
  }  
}  
  
main() ;  
db.close();
```



chiuso immediatamente senza l'esito delle 2 funzioni

SQLite Libraries: Various Options

- `sqlite3`: the basic SQLite interface (JS wrapper of the SQLite C library)
- `sqlite`: This module has the same API as the original `sqlite3` library, except that all its API methods **return ES6 Promises**.
 - internally, it wraps `sqlite3`; written in TypeScript
- `sqlite-async`: ES6 Promise-based interface to the `sqlite3` module.
- `better-sqlite3`: Easy-to-use synchronous API (they say it's faster...)
- ... search on <https://www.npmjs.com/>



License

- These slides are distributed under a Creative Commons license “**Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0)**”
- **You are free to:**
 - **Share** — copy and redistribute the material in any medium or format
 - **Adapt** — remix, transform, and build upon the material
 - The licensor cannot revoke these freedoms as long as you follow the license terms.
- **Under the following terms:**
 - **Attribution** — You must give [appropriate credit](#), provide a link to the license, and [indicate if changes were made](#). You may do so in any reasonable manner, but not in any way that suggests the licensor endorses you or your use.
 - **NonCommercial** — You may not use the material for [commercial purposes](#).
 - **ShareAlike** — If you remix, transform, or build upon the material, you must distribute your contributions under the [same license](#) as the original.
 - **No additional restrictions** — You may not apply legal terms or [technological measures](#) that legally restrict others from doing anything the license permits.
- <https://creativecommons.org/licenses/by-nc-sa/4.0/>

