Lezione03:08-Asynchronous Programming in JS

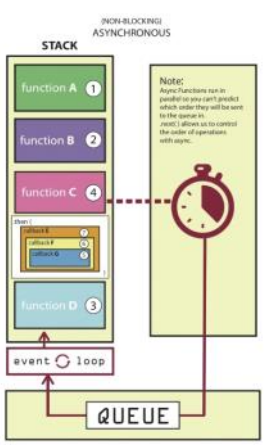
Tuesday, March 9, 2021 9:07 AM

<WA1/>
<AW1/>
2021

Asynchronous Programming in JS

"The" language of the Web

Fulvio Corno
Luigi De Russis
Enrico Masala



Outline

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

Typical synchronous programming style in js: pass functions and closures

Continue execution without waiting for the function to return

Every js library uses callbacks and asynchronous techniques!

JavaScript: The Definitive Guide, 7th Edition
11.1 Asynchronous Programming with Callbacks

JavaScript – The language of the Web

CALLBACKS

Callbacks

Functions as arguments of other functions

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

- 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);
```

Function log2Quote(quote){ console.log(`second version of the ${quote}`}

CreateQuote("WebApp I rocks!", log2Quote);

Or I can create a functional expression on the fly!
CreateQuote("WebApp I rocks!", (quote)=>{console.log(`second version of the ${quote}}});

Function does the formatting and uses the function parameter I give to it to do the printing

Synchronous Callbacks

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

Sorting.js

```
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);
```


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 }
];

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 } ]
```


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 stile" rather than "imperative style" (e.g., for loops)
- Can improve program readability:

Can dynamically adapt better to the data structure that is being passed (more polimorphic)

Extrac t subset of array that satisfies certain criteria

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

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

Array.filter scans the array and for each elements it calls the Callback function: if it returns true the element is added to the new array, otherways it is ignored.
There is an unique filtering algorithm (array.filter) that will work for any filtering criteria
Each different filtering criteria will be implemented by a different (filter_function1,2,3) that will be easily swapped (change passed callback) -> higher readability! (less code)

Array.filter is designed to be easier to parallelize because each elements is different from others, while a for loop is intrinsecally sequential, even if the operations are separated (needs smart interpreter /compiler/architecture to identify if they are dependant or not)

Notable Features of the Functional Paradigm

- 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*
  - 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

Iterating over Arrays

- Iterators: `for ... of`, `for (..;..;..)`

Functional style

- Iterators: `forEach(f)`
  - Process each element with callback f
- Iterators: `every(f)`, `some(f)`
  - Check whether all/some elements in the array satisfy the Boolean callback f

Easier than having a loop with a flag

- Iterators that return a new array: `map(f)`, `filter(f)`
  - Construct a new array

Very powerful

- `reduce`: callback function on all items to *progressively* compute a result

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

.forEach()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/forEach

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

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


.forEach()

https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/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.
    - `index` (Optional): The index of `currentValue` in the array
    - `array` (Optional): The array forEach() was called upon.
  - 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

Only argument is the callback to execute

.every()

"Is it true that every elements satisfies this condition?"

- `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*)
  - 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()

"Is it true that at least one element satisfies this condition?"

- `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()

Returns an array with length = to the original, with different elements!

- `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(''));
```

Could convert numbers to strings, strings to date, strings to output message.... Very powerful! Often used also in react to create list of items iterating over a list of elements using a method defined only once.

.filter()

Returns an array with length <= to the original, with the same elements!

- `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

Let lessThan3Array=

```
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]
```


.reduce()

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

Takes an array and returns 1 value combining an accumulated value with all the elements! Convenient for sum of elements, find Max, Min, when we can define an operation over the entire array as an operation over the summary of a part of an array+another argument

- `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**)
  2. an (optional) **`initialValue`** to pass to the function; if not specified, it uses the first element of the array as initial value

.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

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

a.reduce( (accumulator, currentValue) =>
accumulator + currentValue,     0);
// 15; the sum of the values

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
```

Next value of the accumulator will be the previous value of the accumulator + the currentValue (value of the current elements)

Example: average price of all SUVs

```
const vehicles = [
  { make: 'Honda',  model: 'CR-V',     type: 'suv',   price: 24045 },
  { make: 'Honda',  model: 'Accord',   type: 'sedan', price: 22455 },
  { make: 'Mazda',  model: 'Mazda 6',  type: 'sedan', price: 24195 },
  { make: 'Mazda',  model: 'CX-9',     type: 'suv',   price: 31520 },
  { make: 'Toyota', model: '4Runner',  type: 'suv',   price: 34210 },
  { make: 'Toyota', model: 'Sequoia',  type: 'suv',   price: 45560 },
  { make: 'Toyota', model: 'Tacoma',   type: 'truck', price: 24320 },
  { make: 'Ford',   model: 'F-150',    type: 'truck', price: 27110 },
  { make: 'Ford',   model: 'Fusion',   type: 'sedan', price: 22120 },
  { make: 'Ford',   model: 'Explorer', type: 'suv',   price: 31660 }
];

const averageSUVPrice = vehicles
  .filter(v => v.type === 'suv')
  .map(v => v.price)
  .reduce( (sum, price, i, array) => sum + price / array.length, 0);

console.log(averageSUVPrice); // 33399
```

https://opensource.com/article/17/6/functional-javascript

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

Until now callbacks are just normal functions, called in specific places. (synchronous functions)
Asynchronous Programming will revolutionize our view of the program, creating many opportunities (bugs :D)

JavaScript – The language of the Web

ASYNCHRONOUS PROGRAMMING

Key pattern for JS.

Asynchronicity

Different from multi-threading and concurrent programming!!!
Different thread in different tabs, but single tab (single js file) has a single thread!
I don't know When it will be executed, BUT IT CAN'T BE EXECUTED IN PARALLEL!
Sinchronous execution engine executes 1 instruction at a time. (slow function, when it's executed will slow all the operations) BUT the order of the instructions can be different from the synchronous one (depending on user actions for example)

- 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)
```

Callback to be executed after 2000ms!
Flow of operations goes after setTimeout, and after 2000ms the callback function will finally execute!
(even after the main function has finished because internally there is still a reference to that function, that will be executed by the execution engine after the triggering of one event)

Non-Blocking Code!

N.B. if asynchronous callback function has an infinite loop it will never end->browser/node.js will be frozen

- 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

There can be very long chain of asynchronous triggers!

Asynchronous Callbacks

Since WebApp behaviour is typically asynchronous (browser requests, clicks) depends on the environment around (event-driven by user action/server returns results of computations)

- 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

Every operation that interacts with user/I/O must be run asynchronously because we CAN'T wait for used action without doing any other action on the page! (Libraries are all asynchronous)

```
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

- Useful to delay the execution of a function. Two possibilities from the runtime environment
  - `setTimeout()` runs the callback function once after a given period of time
  - `setInterval()` runs the callback function periodically

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

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

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

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


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

2 Problems: already difficult handle error when they happen, but here even more difficult because the callback will run later (I don't know when! My program is in a totally different state when the error happens!)
The callback function should have the capability to deal with the error itself!

- 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

From Disk

I keep going even if the read has not completed!
When file is fully read, call this callback function: (err,data)=>{...}
Callback handles error itself when operation is complete/ when operation raised an error! Inside the callback I distinguish the two possible way in which the function readFile terminates (error or ok)

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

Here data contains file content, but outside "data" is not defined!

Result=data -> I can't do this because this will be executed AFTER readFile started BUT BEFORE readFile terminates (I don't know when!!) I can't reliably know when the data will be available outside the callback! We can only know from INSIDE the callbacks! That's the reason behind the chains of callbacks! Creating an event-driven code! (If readFile returned correctly, it will call another callback to

Result=data -> I can't do this because this will be executed AFTER readFile started BUT BEFORE readFile terminates (I don't know when!!) I can't reliably know when the data will be available outside the callback! We can only know from INSIDE the callbacks! That's the reason behind the chains of callbacks! Creating an event-driven code! (If readFile returned correctly, it will call another callback to process those files…) Most operation will be managed by chains of callbacks, while the main function will not do much (Just starting callbacks, not computing further)
Since callbacks could themself generate errors and they are in an uncomfortable environment, it is better to create them quick and clean!

Data Persistence

DATABASE ACCESS WITH SQLITE

Access to DB is an asynchronous operation!

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

Client<->DBMS(MySQL, Oracle)
SQLite simpler!
Embedded Database!! Simpler to use!
Application that runs in node.js+1 file containing SQLite database managed directly by our program!
SQLite is a small C library that understand a subset of SQL and do the operations directly in a binary file!
Client manages directly the file containing the database!

SQLite

Numbers.js

Each js project has a file package.json that contains information, also all modules that will be used by the project.
npm init creates the initial package.json file asking some info.

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

To access the database

How to declare project dependancies:

We import a module outside of standard library!

1)

```
npm install sqlite3
```

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

...
db.close();
```

Node modules manager that can install new modules using install:
This will:

- add to package.json the dependancies of this project from this library
- create the folder node_modules inside the project containing all sqlite3 modules and the modules it depends on!
- Create package-lock.json -> actual versions of the packages which are in the node_modules

When you want to distribute a project you distribute only package.json (and package-lock.json), and who needs to use it will then do npm install, which will read that file and install the needed dependancies

2) npm install –g sqlite3 -> global: dependancies for all the projects (in the operating system) NOT RECOMMENDED because each project will use libraries that may or may not be installed on the system

3) npm install --dev sqlite3 -> dependancies not used in runtime but only during development (by toolset as linters, parsers, code checkers, debuggers,…)

SQLite: Queries

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

- `const sql = "SELECT…";`

Query wrote as a normal string, executed using all function.

Callback function

- `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

Inside the callback function we are able to use the result WHEN the query terminates!

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

because each project will use libraries that may or may not be installed on the system

3) npm install --dev sqlite3 -> dependancies not used in runtime but only during development (by toolset as linters, parsers, code checkers, debuggers,...)

SQLite: Queries

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

- `const sql = "SELECT…";`

Query wrote as a normal string, executed using all function.

Callback function

- `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

Inside the callback function we are able to use the result WHEN the query terminates!

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

SQLite: Queries

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

- `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, …)

(Count result has always 1 row)

- `db.each(sql, [params], (err, row) => { } )`
  - Executes the callback once per each result row (no need to store all of them)

Callback receives 1 row at a time (callback is called many times, once for every row->doesn't need to wait for all result to arrive form the db!) (faster to process!)

Internal memory saving! Less memory footprint!

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

SQLite: Other Queries

- `db.run(sql, [params], (err) => { } )`

Insert, Update, Delete

  - 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)

We could explore libraries, but for our scope direct queries are enough

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

Parametric Queries

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

[code] is an ARRAY of all values which will be substituted to the ? In the query

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

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

To avoid SQL-Injection problems!

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'
}
```

Err, rows are 2 parameters which will be filled by the db.all function

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);
}
```

Save result for later

But this is not actually later!!! (We don't know if it will be later!)

Above != Before
Below != After

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; });
    (or get)

    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
...
```

Number increasing, but not consecutives!!!
3 run are finished before the next all, then 3 all are finished before the next run!
(partial ordering)

Queries are Executed Asynchronously

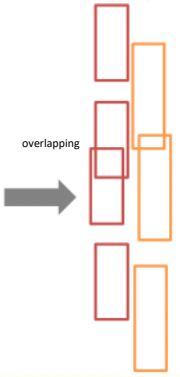
```
const sqlite = require('sqlite3');
const db = new sqlite.Database('data.sqlite',
    (err) => { if (err) throw err; });

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();
```

overlapping



```
...
389
390
391
392
396
396
396
397
398
399
399
400
400
...
```


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

(state machine)

Yellow is not enough, because it only guarantees that the all will run after the insert, BUT I have not solved the problem that the next run can start (and even finish!) BEFORE the previous run has finished!
To solve it I would also need to have the red part: execute the next run only before the previous run (and the previous all) has finished!
But this creates an impossible snake that eats its tail -> we can't write this code!

This problem is artificial (we won't call queries inside a for loop because IF WE WANT A SYNCHRONOUS SEQUENCE OF QUERIES, WE WILL USE A NESTED CHAIN OF ASYNCHRONOUS CALLBACKS which has a start and a finish, but still this could be a problem!)

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

Suggested solution for when the callback chain becomes too long to handle and to understand: PROMISES!
Recent addition to js that will semplify our way to write code in this circumstances -> use asynchronous code but call them in a synchronous way.
Not all libraries adopt promises and awaits instructions, but some still uses the old way of asynchronous callbacks, so we need to be able to use and understand them!

JavaScript – The language of the Web

PROMISES

Type of Object that was designed to avoid the callback hell! (put all functions inside callbacks just to be sure that they are executed in order! Adds complexity and unreadability for something that should be simple!)

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(...);

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 private = answer;

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

  rl.close();

});
```


Promises

- 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

When I call an asynch function, it won't return immediately the result, but it returns immediately an object (the Promise) that will tell me "the result of this operation sooner or later will be available". This object(the Promise) has callbacksthat will be executed when the operation is completed.
We can use the object as if it already contained the result of the operation! We don't pass the real return value of the asynch function, we pass the Promise of the result to other functions, which "load" the Promise with all the function that we want to execute on the result of the Promise, once it will be available!

Without Promises we either GO ON and don't care about the results of the asynch function OR CONTINUE the operations using the return value of the asynch function INSIDE THE CALLBACK.
With Promises we can also CONTINUE the operations using the PROMISE of the return value of the asynch function OUTSIDE THE CALLBACK.

Second advantage: standardized way to handle error which binds with the normal synchronous eception handling (try catch)

Promises

Asynch function returns a Promise

- 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

correct result

Error happened (Promise of all I would have done with the result is rejected)

By convention parameters called like this, but this is the name of the internal function that will make the promise RESOLVE in one of the 2 states

Constructor takes only 1 parameter: a callback function.

46

By convention parameters called like this, but this is the name of the internal function that will make the promise RESOLVE in one of the 2 states

Creating a Promise

Simple as creating new object of type Promise

- 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

});
```

Constructor takes only 1 parameter: a callback function.
This callback function will be called internally by the Promise Mechanism.
It takes 2 parameters:
Resolve= function defined inside the Promise (provided by the language) object that when is executed tells the promise object to change state from pending to fullfilled
Reject= function defined inside the Promise (provided by the language) object that when is executed tells the promise object to change state from pending to rejected
WE CAN DECIDE someValue and "failure reason" because they will be the return value of my asynch function and the error (exception code) that it will generate if it will fail.
These are just the functions provbided by JS to update the promise state

N.B. No code should be written after resolve and reject because then the caller won't be aware of it in any way

MY ASYNCH CODE will eventually RESOLVE TO a return value or REJECT for a failure reason

47

Creating a Promise

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

```
function wait(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);
    }
  });
}
```

Synchronously the wait function creates a Promise function and returns immediately (flow continues), while the callback inside the promise will execute and fill the return/error value asynchronously.

SetTimeout callback will call the resolve method when timer expires

48

Consuming a Promise

Use the result of 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

```
waitPromise().then((result) => {
  console.log("Success: ", result);
}).catch((error) => {
  console.log("Error: ", error);
});

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

Synchronously the then/catch function creates a Promise function and returns immediately (flow continues), while the callback inside the promise will execute and fill the return/error value asynchronously.

Promise object has methods (then and catch) that take as argument the callback to be executed when the promise changes its state to fulfilled or to rejected.
I call both method synchronously ON THE PROMISE OBJECT IN PENDING STATE, specifying what to do when the promise will change state.
The arguments of the callback to be executed are the values returned from the Asynch function inside the promise.

This is the code to be executed after the asynch function! But this time this isn't inside the callback function! No nesting! Can be outside, everywhere!

The asynchronous code can start only after the synchronous part will finish executing! (creation of promise object, creation of then promise object, creation of catch promise object). After those operation, at any time the asynch function can be executed.
A function can never be interrupted (single-thread!)

49

Consuming a Promise

Less used

- `p.then(onFulfilled[, onRejected]);`
  - Callbacks are executed asynchronously (inserted in the event loop) when the promise is either fulfilled (success) or rejected (optional)

better

- `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! ⇒ They can be chained

OnFulfilled function takes someValue as parameter, OnRejected takes "failure reason", OnFulfilled doesn't take any argumnets

This onFinally argument

p.then().then() -> second then applies on the promise returned by the first then!
p.then().then().catch() -> catch is applied on the promise returned by the SECOND THEN! But since exceptions propagate, if p or the promise returned by the first then or the promise returned by the

50

- Useful to avoid code duplication in then and catch handlers
- All these methods return Promises, too! ⇒ They can be chained

p.then().then() -> second then applies on the promise returned by the first then!
p.then().then().catch() -> catch is applied on the promise returned by the SECOND THEN! But since exceptions propagate, if p or the promise returned by the first then or the promise returned by the second then generate an exception, in all cases the catch will take it.

Not recommended nesting then and catch in various orders!

Promise: Create & Consume

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

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


Chaining is better than Nesting! Clearer and simpler!

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());
});
```

Convention to tell that the parameter will not be used

Hipotetical API GitHub

Promise RESOLVES to a value that contains the return value

("Promise returns object" for brevity, but it is not what happens!)

GetIssue could be a synchronous function that just extracts some data, BUT it needs to be executed asynchronously because the time at which the parameter will arrive IS NOT KNOWN at first! (or it may just be an asynchronous function)

GetIssue(repo) returns the promise which, when fulfilled, will contain the return value "issue"

Event Loop inside Browser will explain what happens behind the scenes (there is a scheduler inside js that gives an order of execution using a stack and updating it for each new premise) (not preemptive like in operating system but as request arrives)

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));
```

Do something when tot promises all completes

- What if we want to execute several asynchronous operations in parallel?
- `Promise.all()`
  - takes an array of Promise objects as its input and returns a Promise
  - 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()`
  - 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

Promise.all([insertNumber(), countRows()]).then(([_, tot]) => {console.log(tot))

Promise returned by the function insertNumber,	Promise returned by the function countRows
Returns _	Returns tot

One fastest

Now we have a better sintax, but we still lack a mechanism to wait for a promise to return.
The last step is to have a mechanism to wait for an asynchronous function (promise) to complete.
Stop execution until that finishes!

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

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

Async removes the need to write "return new Promise .."! Every return will be converted to a Promise!

Stops the synchronous execution (of the asynchronous function) until promise is resolved

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

Must be inside async function!

Regular (synchronous) javascript exception

Without Promises Blood and Sweat! (And simpler web applications!) Evolution in language permitted higher complexity web applications)

Often main is asynch, but functions to be executed are synchronous (when thery're time comes!)

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!

- 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

```
async 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 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

```
async 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 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);
                }
            }) ;
    ) ;
```

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

main() ;
```


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();
```


SQLite Libraries: Various Options

We used this-> this uses callbacks

- `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…)

Other people tried to create more Promise-friendly libraries (extensions)! Look them up! Different interface but same concepts!

Since sqlite3 is just C code which runs in local file cached in memory, so all this asynch stuff is not useful, "mine is faster!"

We can use whichever we prefer!

- … search on https://www.npmjs.com/

License