PROGRAMMAZIONE ASINCRONA

Javascript di default è sincrono ma può avere un comportamento asincrono.
Callback sono uno dei modi principali per scrivere codice asincrono

Asincronicità possibile grazie a due elementi:

- Execution environment (*node, javascript nel browser*)
- Event loop

Query database, richieste http... vengono ottimizzati da asincronicità

Esempio di codice asincrono:

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

setTimeout aspetta (in parallelo al programma) **circa** 2 secondi e poi chiama la callback. Il programma non rimane bloccato.

Callback asincrone

Modo base per fare codice asincrono, eseguite in modo non bloccante.

- Gestire azioni utente (click bottone)
- Gestire operazioni di I/O (fetch di un documento)
- Gestire intervalli di tempo (timers)
- Interfacciarsi con il database

```
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

- **setTimeout()** → chiama una funzione dopo un certo tempo (max 2^{31}-1 ms)
  - callback
  - secondi
  - parametri della callback
- **setInterval()** → chiama una funzione ciclicamente ogni intervallo di tempo
- **clearInterval(id)** → ferma esecuzione della setInterval con id id

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

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

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

clearInterval(id) ;
```

Gestione errori callback, default:

- primo parametro contiene l'errore
- secondo parametro è il contenitore per i risultati, per i dati

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

DATABASE

In SqlLite, il database è contenuto interamente dentro un file (positivo per app piccole, no per app grosse)

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

Queries:

const sql=”SELECT …”;

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

db.all() → esegue e ritorna tutte le righe della callback

```
db.all(sql, [params], (err, rows) => { } )
```

- se **err**=true → c’è stato un errore, altrimenti rows contiene un risultato
- **rows** è un array, ogni item contiene i campi del risultato

db.get() → prende il primo risultato in risposta ad una query

```
db.get(sql, [params], (err, row) => { } )
```

- se c’è un errore sintatico nella query o database inaccessibile → err pieno
- se la query è sintatticamente corretta ma non produce risultati → row=undefined

db.each() → esegue la callback per ogni risultato

```
db.each(sql, [params], (err, row) => { } )
```

db.run() → per statement che non ritornano un valore

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

- create table
- insert
- update
- funzioni callback
  - this.changes → quante righe sono state modificate dalla nostra operazione
  - this.lastID → ultimo ID assegnato all’ultima riga inserita (vale solo per insert)

NOTA: con db.run bisogna usare una function normale perché un arrow function andrebbe a sovrascrivere i valori this.

PARAMETRI NELLA QUERY

Per passare un parametro alla query, si usa il **?** nella query e [] nella chiamata

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

```
import sqlite from '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);
    }
```

```
{
  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'
}
```

Importare librerie

```
import sqlite from 'sqlite3';
```

Collegarsi a database

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

Scrittura query

```
let sql = 'SELECT * FROM answer';
```

Db.all con stampa risultato

```
db.all(sql, [], (err, rows) => {
    if(err) throw err;
    for(let row of rows )
        console.log(row);
});
```

Esempi di query e come non farle tenendo conto che vengono eseguite in maniera asincrona.

PROMISES

Oggetto che rappresenta il **possibile completamento** di un'operazione asincrona. Contiene la promessa che questo oggetto verrà servito e quindi si potranno effettuare operazioni. → mi permette di evitare callback infinite.
Quando una callback viene letta può essere in **3 stati:**

- **pending** → sono ancora una promessa, non ancora soddisfatta
- **fulfilled, resolve** → soddisfatta con successo
- **rejected** → non andata a buon fine, rifiutata

```
const myPromise =
  new Promise((resolve, reject) => {

    // do something asynchronous which
    eventually call either:

      resolve(someValue); // fulfilled

    // or

      reject("failure reason"); // rejected

});
```

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

```
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);
});
```

Consumare la promise:

- **then:** se resolve
- **catch:** se rejected (si può omettere se non utile, tipo se si causa errore direttamente)
- **finally:** si esegue in ogni caso

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

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

```
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());
});
```

NOTA: le promise sono **concatenabili** in modo tale da eseguire delle funzioni in sequenza facendo in modo che si succedano in caso di risultato valido

- 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) })
```

Le promise possono anche essere eseguite in parallelo (ad esempio posso far eseguire 2 promise in parallelo e attendere la fine di entrambe per andare avanti)

- **Promise.all**()
  - Prende **tutte le promise** inserite come argomento
  - Se tutte le promise sono state fulfilled, ritorna un array con tutti i valori delle promise risolte
  - Se almeno 1 non viene risolta, nessuna viene risolta
  - Argomento di queste promise.all può anche non essere una promise (non farà mai rejected, sempre valida)
- **Promise.race**()
  - Ritorna una promise che è fulfilled o rejected quando **la prima delle promise** nell'array è fulfilled o rejected (NOTA: se c'è una funzione non promise, sicuro torna fulfilled)

ASYNC/AWAIT

Keyword per semplificare la scrittura di codice asincrono usando promise.

- **Async** prima di una funzione → indica che la funzione ritornerà una promise
- **Awayt** prima di una funzione chiamata → indica che la funzione che sto chiamando, ritorna una promise e che quindi devo aspettare che quella promise sia fulfilled o rejected.
  - Blocca esecuzione finchè non viene risolta la promise
    - In caso di errore → throw
  - **Nota: una awayt può essere usata solo in una funzione async**

```
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"
```

Promise vs async-aways

```
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();
```

- 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();
```

```
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');
}
```

- Se l'output della funzione 2, dipende dall'output della funzione 1 → **await**
- Se due funzioni possono girare in parallelo → due funzioni async che runnano in // con Promise.all
  - *Async-awayt non ha un meccanismo per fare promise.all() o promise.race() e quindi in quel caso bisogna usare le **promise**.*
- Generalmente è più comodo usare async-await ma a volte (sqlite3) non è possibile farlo.

NOTA: **Resolve** → ritorno promise fuori da funzione

NOTA: Essendo esecuzione asincrona, nel caso barrato, si chiude il database prima che sia effettivamente usato o durante l'utilizzo.

```
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();
```

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