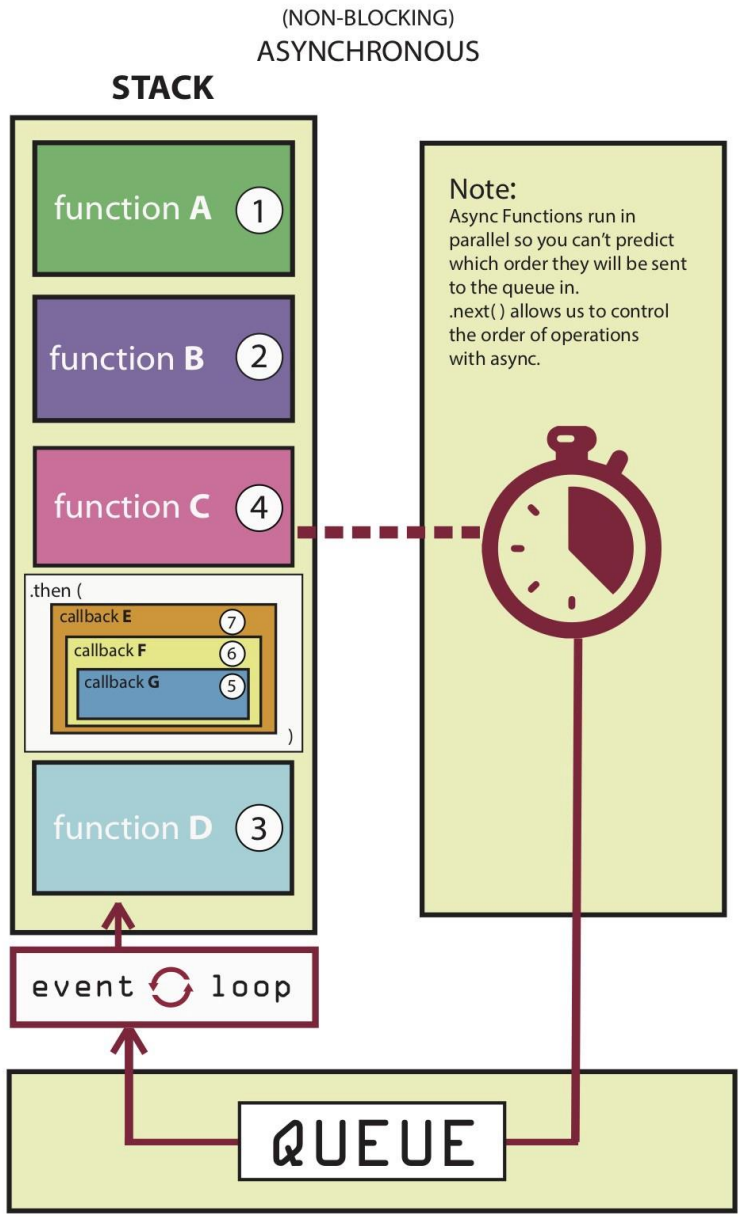
<WA/>
2026

Asynchronous Programming in JS

“The” language of the Web

Fulvio Corno
Luigi De Russis
Enrico Masala



Outline

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

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

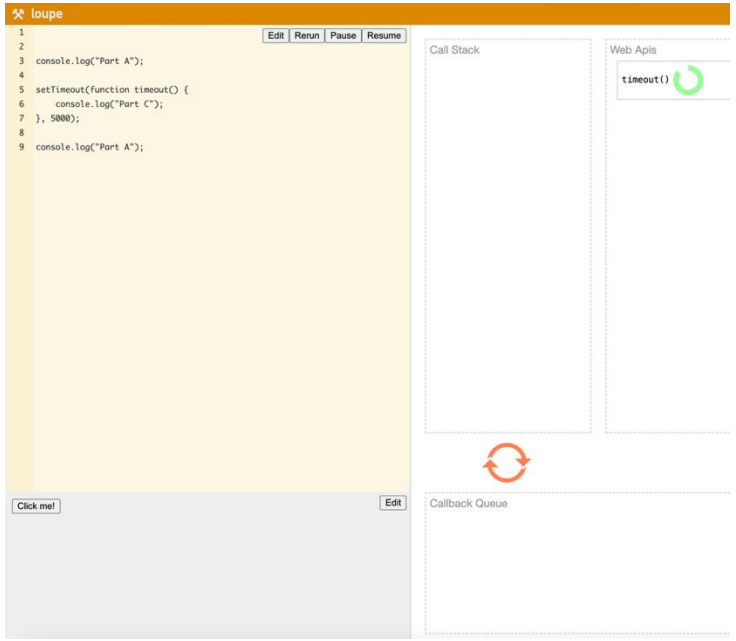
- JavaScript is single-threaded and inherently synchronous
  - i.e., code cannot create threads and run in parallel in the JS engine
- Callbacks are the most fundamental way for writing asynchronous JS code
- How can they work asynchronously?
  - e.g., how can setTimeout() or other async callbacks work?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

```
const doAfterTimeout = (task) => {
  // do something
}

// runs after 2 seconds
setTimeout(doAfterTimeout, 2000, task)
```

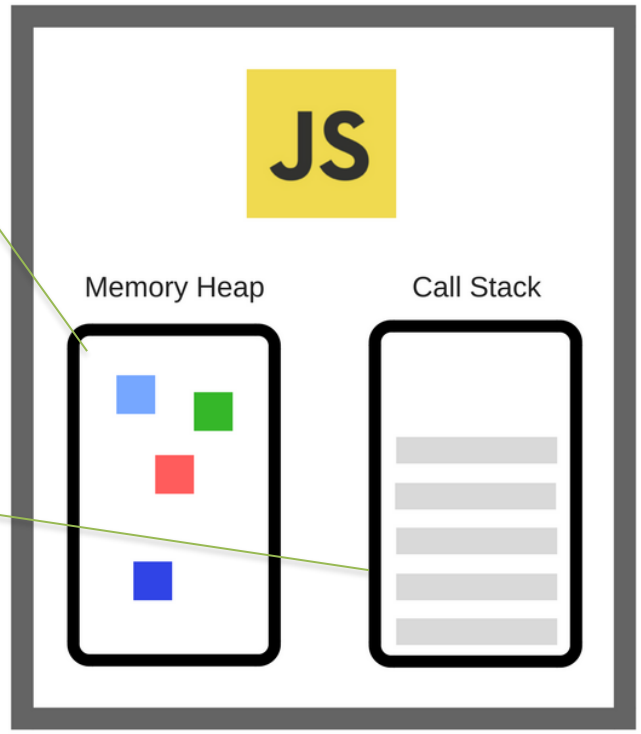
Execution Environment

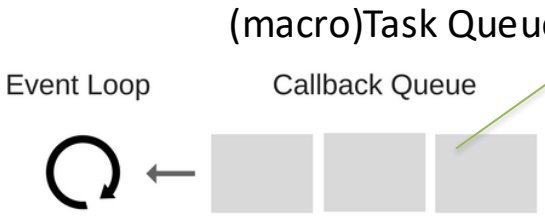
[Try me!](#)



Allocated Objects

Running function calls





Pending callbacks to be executed

https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#what-is-the-event-loop

Event Loop

- During code execution you may
  - Call functions → the function call is pushed to the call stack
  - Schedule events → the call to the event handler is put in the (macro)Task Queue (also called Callback Queue)
    - Events/Tasks may be scheduled also by external events (user actions, I/O, network, timers, …)
- At any step, the JS interpreter:
  - If the call stack is not empty, pop the top of the call stack and executes it
  - If the call stack is empty, pick the head of the (macro)Task Queue and executes it
- A function call / event handler is **never** interrupted
  - Avoid blocking code!

https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop

https://nodejs.org/en/docs/guides/event-loop-timers-and-nexttick/#what-is-the-event-loop

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

```
import readline from 'readline'; //comes with node.js

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 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

- 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
```

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

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

SQLite: SELECT 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 of objects. Each item contains the fields of the result

- `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, …)

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

SQLite: Other Queries

- `db.run(sql, [params], function (err) { } )`
  - For statement that do not return a value
  - INSERT
  - UPDATE
  - DELETE
  - 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

- 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) => {
```

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

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.mjs

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

Example

```
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 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…

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

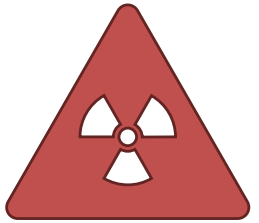
Queries Are Executed **Asynchronously**

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

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

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

number
1



Queries Are Executed Asynchronously

```
import sqlite from '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.mjs

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

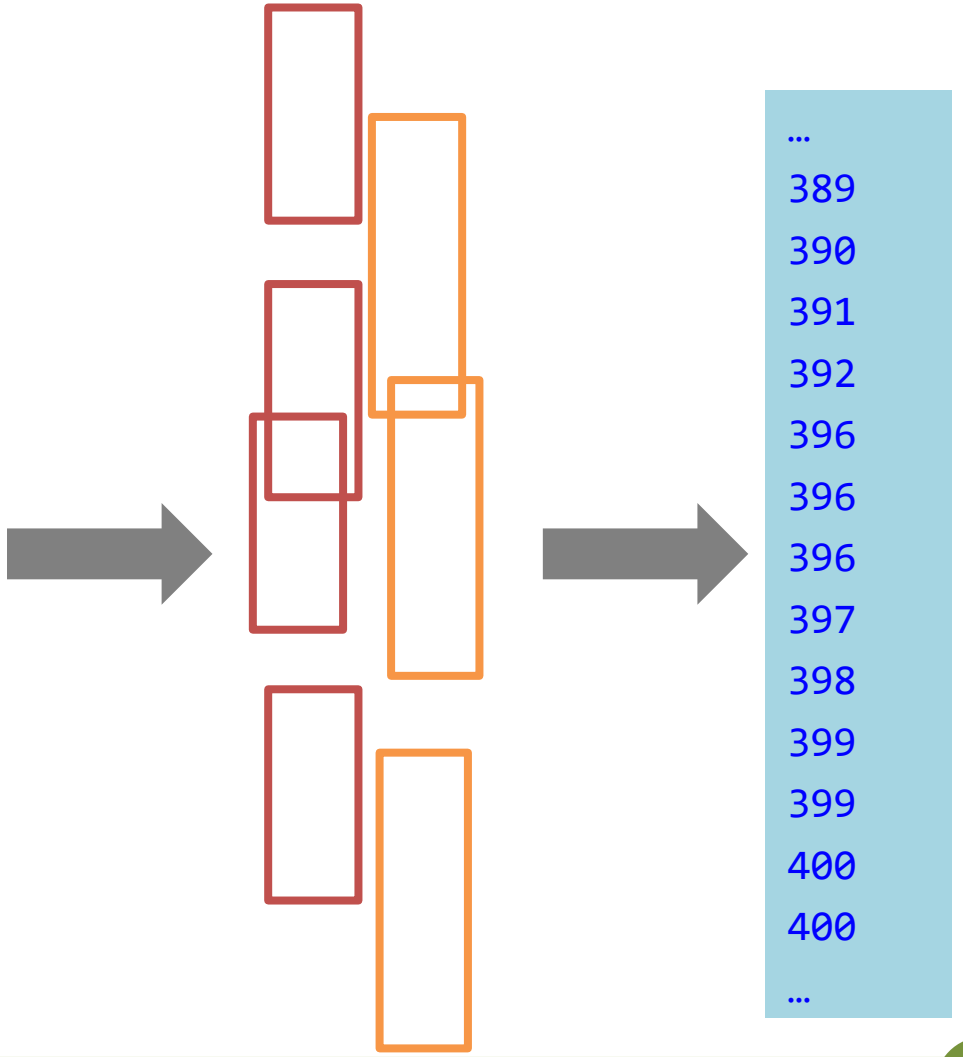
Queries are Executed Asynchronously

```
import sqlite from '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.mjs`, 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

https://javascript.info/promise-basics

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

```
import readline from '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

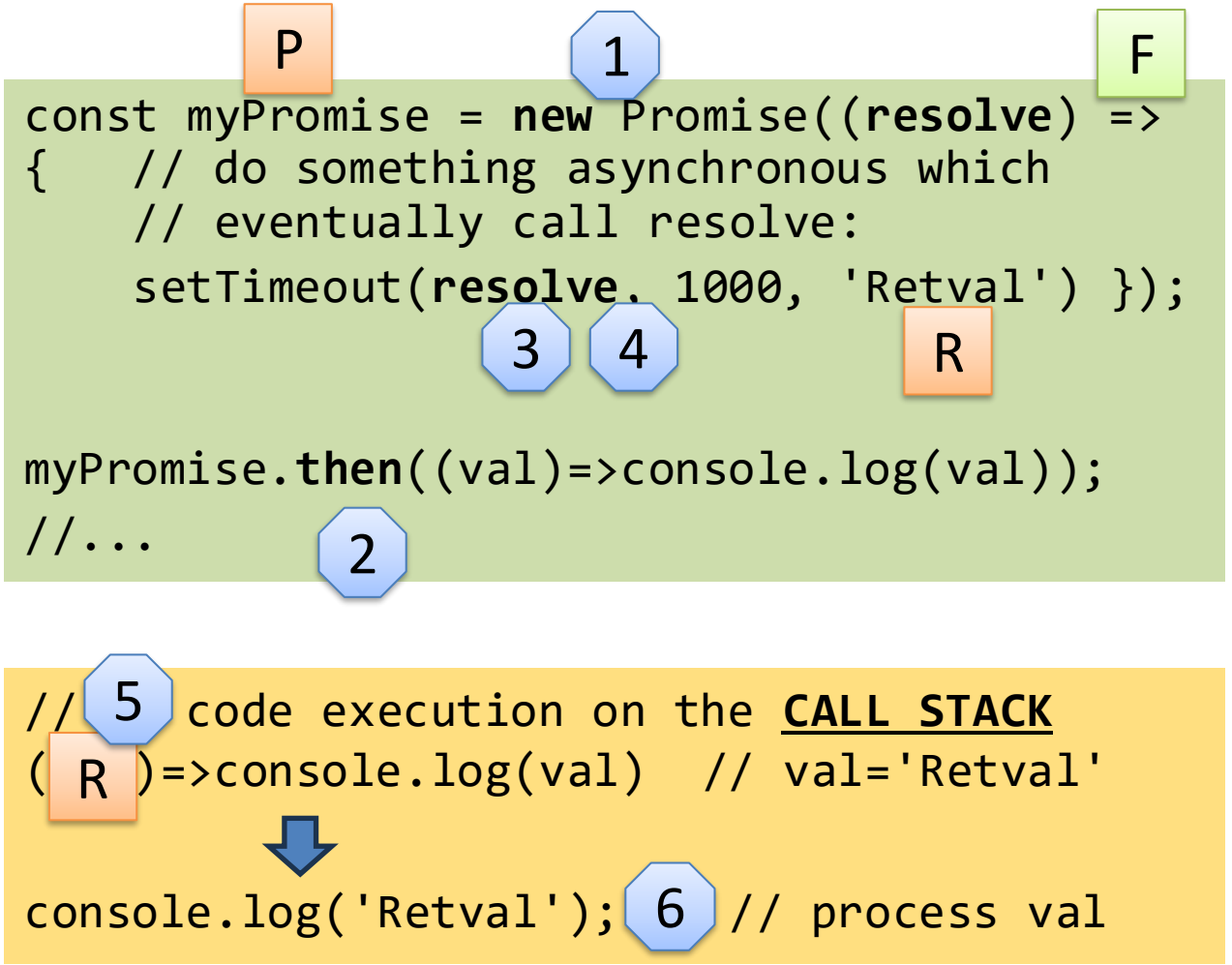
- A core language feature to "**simplify" asynchronous programming**
  - a possible solution to callback hell, too!
  - a fundamental building block for "newer" functions (`async`, ES8 = ES2017)
- It is an **object** representing the eventual **completion** (or **failure**) of an asynchronous operation, when the result of the operation is not yet known
  - 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: how they work

- A Promise object P (in pending state) is created and returned by an asynchronous function F
- Code execution then proceeds as usual
  - Typically, first the code immediately attaches a callback function to the resolution of the Promise
- At a certain time, the asynchronous operation, started within function F, ends, passing the result value R to the promise which stores it
- The Promise is now in resolved state: the JS environment put the callback attached to the promise in the (micro)task Queue (similar to microtask queue but higher priority)
- Once the code finishes its execution, the Event loop puts the callback on the call stack and executes it, passing the R value to the callback
- The R value is used for further processing according to the callback content

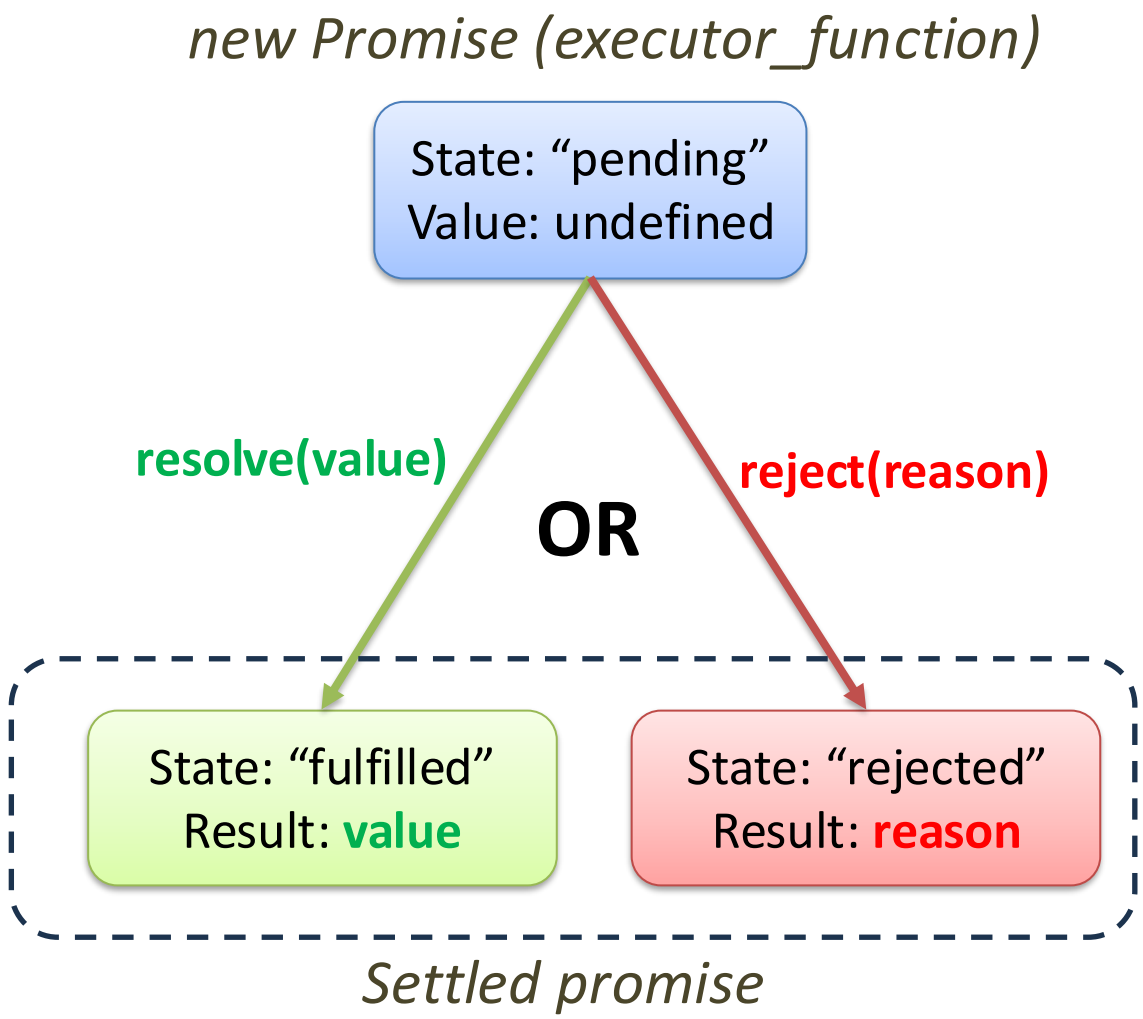
Promises: example

1. A Promise object P (in pending state) is created and returned by an asynchronous function F
2. Code processing proceed as usual
   - Typically, first the code immediately attaches a callback function to when the Promise will be resolved
3. At a certain time, the asynchronous operation, started within function F, ends, passing the result value R to the promise which stores it
4. The Promise is now in resolved state: the JS environment put the callback attached to the promise in the (micro)task Queue (similar to microtask queue but higher priority)
5. Once the code finishes its execution, the Event loop puts the callback on the call stack and executes it, passing the R value to the callback
6. The R value is used for further processing according to the callback content



Promise states

- Promise starts in a pending state
- Promise evolve into a final state (they are ”settled”). The state can be only one of the two:
  - Fulfilled (”success”)
  - Rejected (“failure”)
- In both cases they can carry a JS value



https://javascript.info/promise-basics

Creating a Promise

- A Promise object is created using the **new** keyword
- The constructor takes an *executor function* as parameter
- The executor function takes two *functions* as parameters:
  - `resolve`, to be called when the asynchronous task completes successfully; the parameter is the **value** of the fulfilled promise
  - `reject`, to be called when the task fails; the parameter is the **reason** of the rejected promise (reason for failure, typically an error object)

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

Function returning a Promise

- Very common, just 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

- How to use the value carried by the fulfilled or rejected Promise?
- Attach a callback to the Promise object p, just after it is created:
- `p.then(onFulfilled);`
- `p.catch(onRejected);`
- Callbacks are executed asynchronously (queued in the microtask queue) when the promise is either fulfilled (success) or rejected (failure)
- Both callbacks take, as the only **parameter**, the value of the fulfilled or rejected promise

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

Promise: Create & Consume

```
const promise = new Promise(
   (resolve, reject) => {
      ...
      resolve(value);
      ...
      reject(reason);
      ...
  }
)
```

```
promise
   .then((value) => {
      ...use value ...
   } )
   .catch((reason) => {
      ...use reason ...
   } ) ;
```

Consuming a Promise: methods

- `p.then(onFulfilled[, onRejected]);`
- `p.catch(onRejected);`
- `p.finally(onFinally);`
  - The callback is executed in any case, when the promise is either fulfilled or rejected
  - Useful to avoid code duplication in then and catch handlers
- Note: `catch` and `finally` can be omitted if not needed

- All these methods return Promises, too! ⇒ They can be chained

https://javascript.info/promise-chaining

Chaining Promises

```
const p1 = getRepoInfo();
const p2 = p1.then( c1  repo => getIssue(repo));
const p3 = p2.then( c2  issue => getOwner(issue.ownerId));
const p4 = p3.then( c3  owner => sendEmail(owner.email, 'Some text'));
const p5 = p4.catch( c4 e => { console.error(e) });
           p5.finally( c5 _ => logAction());
```

- All promises p1…p5 objects are created immediately when the code is first executed
- All promises have a callback c1…c5 attached immediately (to be used for either fulfill or reject)
- When p1 is fulfilled, c1 is (asynchronously) executed
- If c1 returns a Promise, if the promise is fulfilled, its value is passed to c2, etc.; if it is just a value, it is converted into a fulfilled Promise
- If any Promise is rejected and no catch callback was attached, the same status and reason is transferred to the next Promise in the chain (also, the “then” callbacks (c1…c3) are NOT executed)
- In the example: any p1…p4 rejected promise arrives at the catch callback c4 which is (asynchronously) executed

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
  - No need to nest operations in callbacks
- **Important:** always return results, otherwise next callbacks will NOT 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());
});
```

- Full specifications of Promise behavior and constraints: https://promisesaplus.com/

Another chaining example

- Useful, for instance, with I/O API such as `fetch()`, which returns a Promise
- Note the static methods to return an (already settled) 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()    // return is implicit in this arrow function

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

SQLite… revisited

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

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

```
function main() {
    let p = Promise.resolve();
    for(let i=0; i<100; i++) {
        p = p.then(() => {return insertOne();} );
        p = p.then(() => printCount() );
    }
    p.then(() => db.close()); // NOT: db.close();
}
main() ;
```

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()`
  - 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

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 its return value will be a Promise (regardless of the actual returned value type)
- Prepend `await` when calling an async function (or a function returning a Promise) makes the calling code stop until promise is resolved or rejected

```
const sampleFunc = async () => {
  return readTxtFile('./file.txt');  // e.g., return a string  }
```

```
// Log readTxtFile returned value
sampleFunc().then(console.log);
```

```
// Log readTxtFile returned value
const s = await sampleFunc();
console.log(s);
```

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*

```
returnValue = await expression ;
```

- await **blocks** the code execution within the async function **until the Promise is resolved, then the code resumes asynchronously**
- 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
  - Need Javascript `try { } catch` statement to handle
- If the value of `expression` is not a Promise, it is converted to a resolved Promise

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"
```

Handling rejected promises with await

```
function resolveAfterTime(ms) {
    return new Promise((resolve, reject) => {
        if (ms < 0)
            reject('Negative delay requested');
        else
            setTimeout(() => resolve('resolved') , ms);
    });
}
async function asyncCall() {
    try { // need to use try-catch otherwise an exception will propagate up
        const result = await resolveAfterTime(-1);
        console.log(result);
    } catch (e) {
        console.log('Exception: ', e);
    }
}
asyncCall();
```

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 userResponse.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

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

```
function printCount() {
    return new Promise( (resolve, reject) => {
        db.all('select count(*) as tot
                from numbers',
            (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)
                values(1)', (err) => {
            if (err) reject(err);
            else resolve('Done');
        });
    }) ;
}
```

```
function printCount() {
    return new Promise( (resolve, reject) => {
        db.all('select count(*) as tot
                from numbers',
            (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() ;
```

SQLite… await vs then

```
async function main() {

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

main() ;
```

```
function main() {
   let p = Promise.resolve();
   for(let i=0; i<100; i++) {
       p = p.then(() => {return insertOne();} );
       p = p.then(() => printCount() );
   }
   p.then( () => db.close() ); // NO: 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

- **`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…)
  - Avoid! (for the purpose of this course due to blocking operations on server side)
- … 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/