Bordeaux INP
ENSEIRB
MATMECA

Asynchronous Programming in JavaScript

Bordeaux INP
ENSEIRB
MATMECA

ASYNCHRONOUS PROGRAMMING

Synchronicity

Bordeaux INP
ENSEIRB
MATMECA

- Synchronous programming
  - Single threading: one operation after the other
  - If one operation takes time, execution waits
  - Problems: network, disk, user input, **timer**, *etc.*
- Questions
  - Why/where can this be a problem?
    - In general (give examples)
    - In a browser
  - When can this be a problem?
    - Can the situations be characterized?
  - What would be a solution?

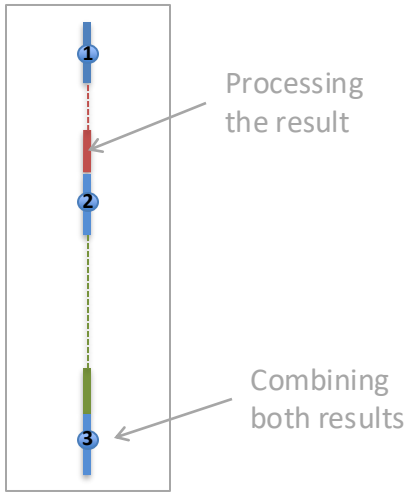
Asynchronicity

Bordeaux INP ENSEIRB MATMECA

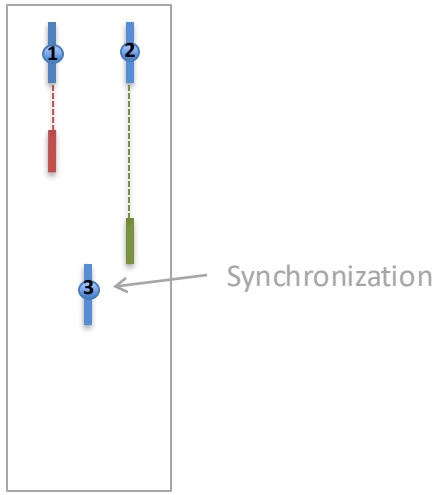
- Problem: network, disk, user input, *etc.*
- Asynchronous programming
  - Operations can be executed *simultaneously*
  - Asynchronous execution
    - Operation is launched
    - Program execution continues
    - When operation returns, execution processes result

Example: P3(P1, P2)

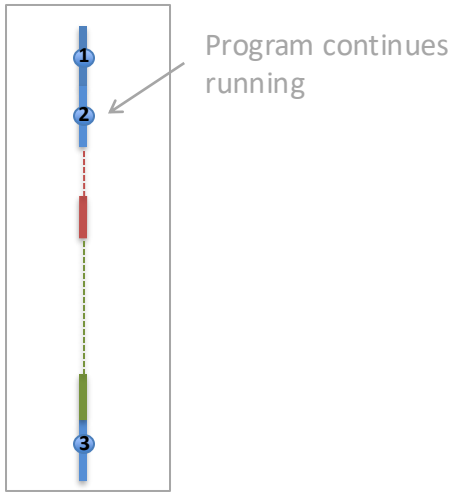
Synchronous, single threaded



Synchronous, two threads

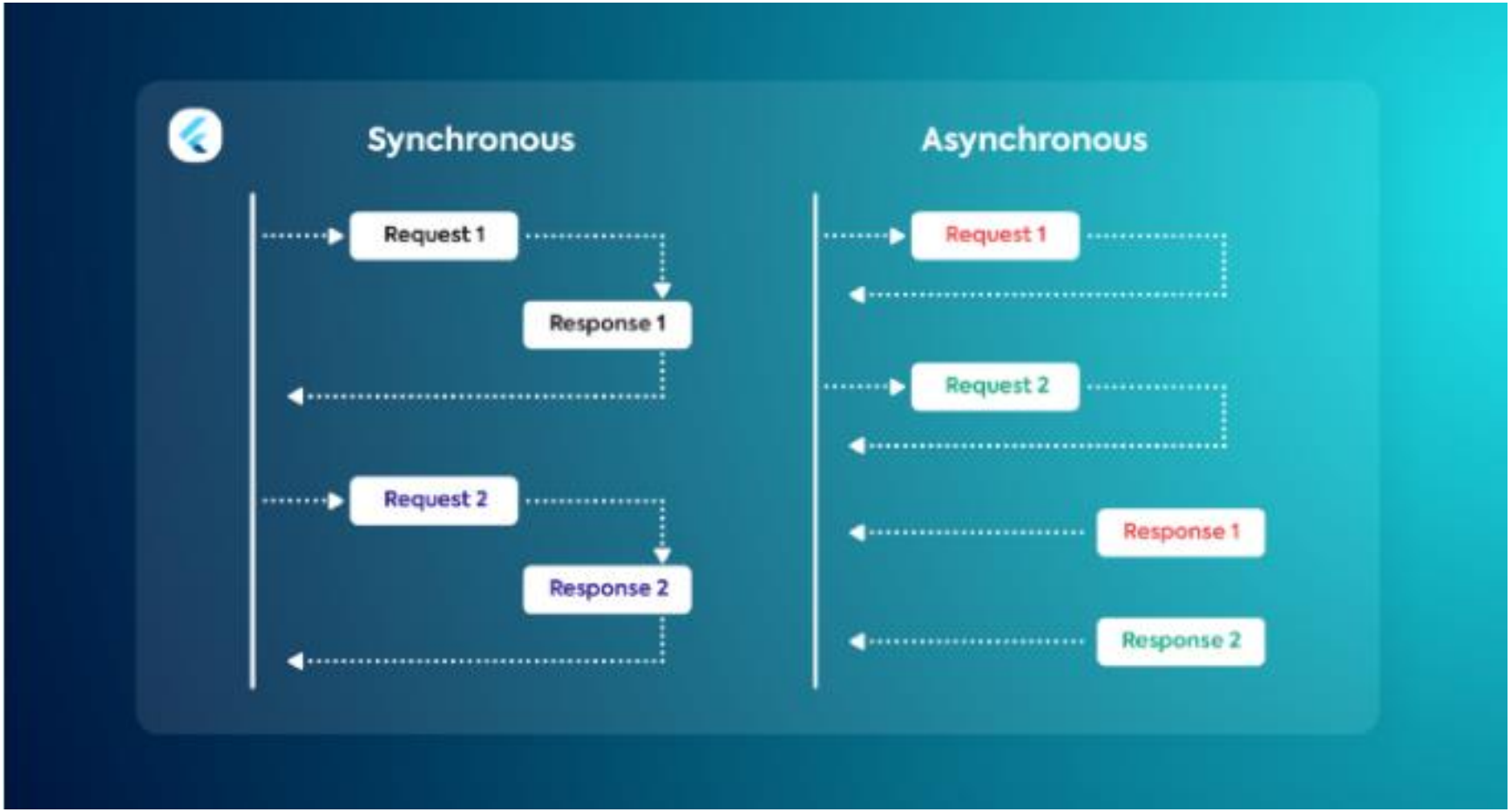


Asynchronous



- Synchronous model (single threaded)
  - Waiting *implicitly*
- Asynchronous model
  - Waiting *explicitly*

Implicit/Explicit waiting



JavaScript

Bordeaux INP
ENSEIRB
MATMECA

- Single-threaded language
  (≠ single-threaded implementation)
- JavaScript shares a thread with other tasks
  E.g., in a browser: updating styles and handling user actions.
- Example
  - `setTimeout(function, milliseconds, …)`
  - `setTimeout( ()=> console.log('Hello'), 3000);`

Creating An Oracle

Bordeaux INP
ENSEIRB
MATMECA

```
204  function getRandomInt(min, max) {
205      return Math.floor(Math.random() * (max - min)) + min;
206    }
207
208  function oracle (c) {
209      let time = getRandomInt(1000, 9000);
210      setTimeout(() => {
211                    console.log(`Waited ${Math.round(time/1000)} seconds`);
212                    c(time); },
213                  time);
214      }
```

- Produces a value after a random time [209]
- Calls the function passed as argument with that time [212]
- What is the similarity between `oracle` and `setTimeout`?

Exercise

Bordeaux INP
ENSEIRB
MATMECA

- Assign a value from the Oracle to variable `a`
  - Where should this assignment be placed?
- Define a function that sums two Oracle-generated numbers
  - What is the difference from the previous case?
- What are the different options to call the Oracle?
  - What are their effects?
- Define a function that returns the sum of four Oracle-generated numbers
  - What do you notice?

Limitations of Asynchronicity

```
231  oracle ((v1) => {
232      oracle((v2) => {
233          oracle((v3) => {
234              oracle((v4) => { console.log(`the result is ${v1 + v2 + v3 + v4}`); });
235          }); }); });
```

- Continuation-passing style: next step is made explicit
- Combining several asynchronous calls
- Contamination of the code
- Heavily nested callbacks – *AKA* `callback *hell*'
- Error prone
- Error handling '`if (err) return callback(err)`'
- Need for structuring
- A software layer to handle asynchronous processing

Bordeaux INP
ENSEIRB
MATMECA

PROMISES

Promises

- It is an asynchronous action that may complete at some point and produce a value.
- Creating a promise with `Promise.resolve(value)` [237]
  - Value wrapped with a promise
  - What does this `resolve` return?
  - Where is the resolved value?
- Getting the result
  - The `then` method is used with a callback [238]
  - The callback is invoked when the promise is resolved
  - A promise can be invoked multiple times (it delivers the same result)
- Moving into time with the `then` method [241, 242]
  - What operation is needed to move values into time?
  - What kind of processing schema is this approach realizing?

```
237    var fifteen = Promise.resolve(15);
238    fifteen.then(value => console.log(`Got ${value}`)); // Got 15
```

```
240    var fifteen = Promise.resolve(15);
241    var eighty = fifteen.then(value => { console.log(`Got ${value}`); return 80; }) // Got 15
242    eighty.then(value => console.log(`Got another one ${value}`)); // Got another one 80
```

Using `Promise` as a Constructor

- Constructor: `new Promise(`*`callback`*`)`
  where *`callback`* is a function with two arguments
  - `resolve`: it is called when data is successfully obtained
  - `reject`: it is called when the processing failed (optional)
- Simplify the definition of oracleP and show the equivalence
- How could the callback function be rewritten?

```
244  function oracleP() {
245    return new Promise((resolve) => oracle((v) => resolve(v)));
246  }
```

Exercise

Bordeaux INP
ENSEIRB
MATMECA

- Write a function that adds two Oracle-generated values using promises.
  - There are (at least) two ways to write this function.
  - How would you compare both versions?

Using Promises

- Nested asynchronous calls [239]
- Cascading/Pipeline processing [244, 246]
- Why is `Math.sqrt` used this way? [246]
- What equivalent expression could be used? (*hint: longer form*)

```
239  oracleP().then((v1) => { oracleP().then((v2) => {
240                                  let res = v1 + v2;
241                                  console.log(`Result is ${res}`);
242                                  return res; }); })
243
244  oracleP().then(v => v*1000).then((r) => { console.log(`r=${r}`); })
245
246  oracleP().then(Math.sqrt).then((v1) => oracleP().then((v2) => { console.log(`Result = ${v1+v2}`); return v1+v2} ))
```

Resolve and Reject

- Constructor invoked with a two-argument function: `resolve` and `reject` [253]
- `Promise.reject(`*`value`*`)` is the dual of `Promise.resolve(`*`value`*`)`
  - Write a simple example using it?

```
250   function odd (v) { v /= 2; return v !== Math.trunc(v); }
251
252   function oraclePF() {
253       return new Promise((resolve, reject) => { oracle((v) => {
254                                                   if(odd(v)) resolve(v);
255                                                   else reject(new Error(`Even number ${v}`)); }); });
256   }
```

Error Handling with Promises

Bordeaux INP ENSEIRB MATMECA

- Two possibilities
  - `catch` with a callback function
  - `then` with a *success* and a *failure* function
- `catch` callback will also receive a synchronous error
  - Is it an alternative to `try-catch`?
- Are the two possibilities for error handling completely equivalent?

```
258  oraclePF().then((v) => { console.log(`Succeed ${v}`); }).catch((v) => { console.log(`Fail ${v}`); });
259
260  oraclePF().then((v) => { console.log(`Success ${v}`); },
261                           (v) => { console.log(`Failure ${v}`); });
```

Exercise

Bordeaux INP ENSEIRB MATMECA

- Write a function that adds two Oracle-generated values, taking into account that `oraclePF` can now fail
  - Try both methods (see slide #17):
    - Two-function callback, and
    - `then`/`catch` callbacks
  - Are they equivalent?
- Assign to variable `p` a failed promise of an Oracle-generated value
  - Call promise `p` with a two-function callback. What happened?
  - Call promise `p` with only the resolve callback. What happened?
  - Call promise `p` only with a `then` callback. No `catch`. What happened?
  - Call promise `p` only with a `then` and `catch` callbacks. What happened?

Exercise

Bordeaux INP
ENSEIRB
MATMECA

- Write a function that
  - takes as argument a number `n`, and
  - produces a list with `n` promises, which will eventually yield Oracle values

Lists of Promises

Bordeaux INP
ENSEIRB
MATMECA

- A list of promises can be produced
  - When does it return its result?
  - Can the result be used immediately?
  - What operation needs to be invoked to use the result?

```
260  function oracleList(n) {
261      let l = [];
262      for(let i=0 ; i<n ; ++i) {
263          l.push(oracleP().then((v) => v));
264      }
265      return l;
266  }
```

Using a List of Promises

Bordeaux INP ENSEIRB MATMECA

- `Promise.all` allows a callback function to be invoked when all promises have yielded a value

```
268   var promList = oracleList(3)
269   Promise.all(promList).then((l) => console.log(`List of oracles = ${l}`))
```

Promises as a Race

- `Promise.race(list_of_promises)` returns the first promise that yielded a value
- Can Promise.race be used to collect values incrementally, as they become available?
- What difference is there between the two invocations?

```
276   var promList = oracleList(3)
277
278   Promise.race(promList).then((v) => console.log(`The winner is = ${v}`))
279
280   Promise.race(oracleList(3)).then((v) => console.log(`The winner is = ${v}`))
```

Algebraic Simplification of Promises

- `resolve(…resolve(v)) = resolve(v)`
- Combining resolved and regular values
- A value returned from a `then` becomes *resolved*

```
282   Promise.resolve(Promise.resolve(Promise.resolve(33))).then(v => console.log(`Value is ${v}`))
283
284   Promise.resolve(false? 1: Promise.resolve(333)).then(v => console.log(`It works ${v}`))
```

Bordeaux INP
ENSEIRB
MATMECA

ASYNCHRONOUS FUNCTIONS

Asynchronous Function

Bordeaux INP ENSEIRB MATMECA

An asynchronous function

- Returns its result via an implicit Promise
- Expresses computations synchronously [289, 290]
- Contains an `await` expression
  - Waits for the passed Promise's resolution
  - Resumes the Async function's execution and returns the resolved value
- Is the only function where `await` expressions are valid

```
288  async function synchAdd() {
289      let v1 = await oracleP();
290      let v2 = await oracleP();
291      console.log(`First call: ${v1}`);
292      console.log(`Second call: ${v2}`);
293  }
```

Exercise

- Explore variations of where to place `await`
  - What behavior do you get for each of them?
  - Does it matter how much time each asynchronous call takes?
- How to maximize responsiveness?

```
288  async function synchAdd() {
289      let v1 = await oracleP();
290      let v2 = await oracleP();
291      console.log(`First call: ${v1}`);
292      console.log(`Second call: ${v2}`);
293  }
```

Exercise

Bordeaux INP
ENSEIRB
MATMECA

- Write a function without `async` and `await` that asks two values from the Oracle, handling errors (version `oraclePF`)
  - Prints their sum if both calls succeed
  - Prints the first value if the second call fails
  - Prints a constant message if the first call fails
- Rewrite the function with `async` and `await`

Answer 1

Bordeaux INP
ENSEIRB
MATMECA

```
298  function asynchAddSF()  {
299
300      oraclePF().then((v1) => {
301                      oraclePF().then((v2) => { console.log(`Total success: ${v1 + v2}`); })
302                                .catch((e) => { console.log(`Second call failed ${e} / First call ${v1}`); }); })
303              .catch((e) => { console.log(`First call failed ${e}`); }) ;
304  }
```

Answer 2

```
308  async function synchAddSF() {
309      try {
310          let v1 = await oraclePF();
311          try {
312              let v2 = await oraclePF();
313              console.log(`Total success: ${v1 + v2}`)
314          } catch (e) {
315              console.log(`Second call failed ${e} / First call ${v1}`);
316          }
317      } catch (e) {
318          console.log(`First call failed ${e}`);
319      }
320  }
```

Bordeaux INP
ENSEIRB
MATMECA

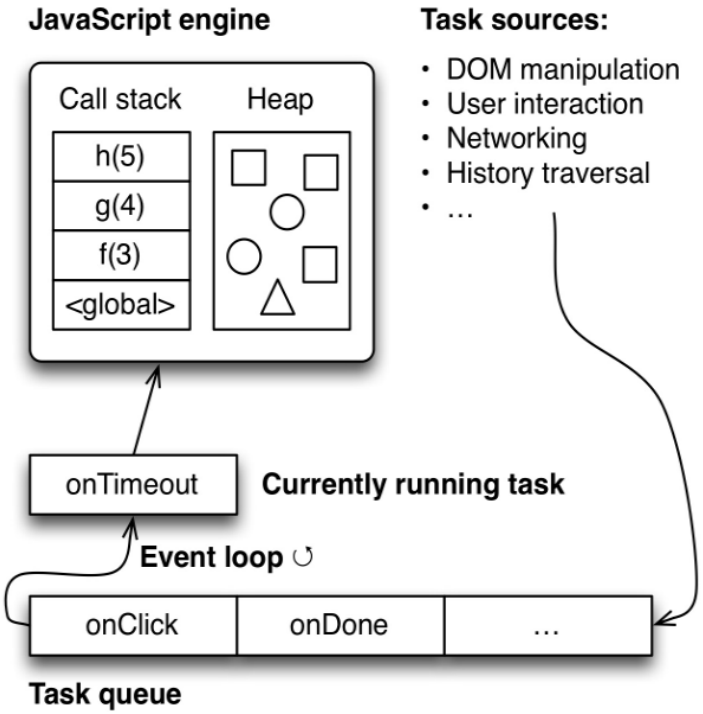
THE BROWSER EVENT LOOP

Event Loop

Bordeaux INP
ENSEIRB
MATMECA

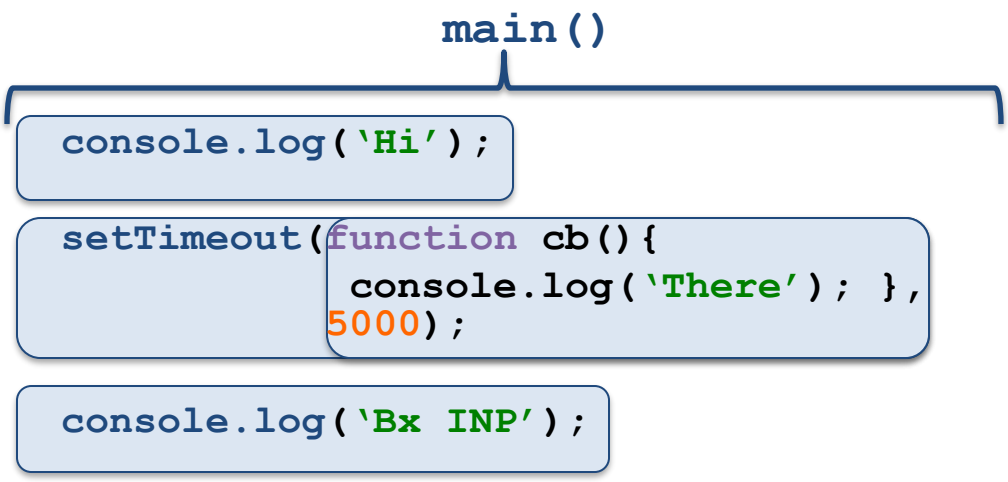
- A single process per browser tab
- Process = the event loop
- Execution of browser-related tasks from a task queue
  1. Parsing HTML
  2. Executing JavaScript code in script elements
  3. Reacting to user input (mouse clicks, key presses, *etc.*)
  4. Processing the result of an asynchronous network request
- Items 2-4
  - run JavaScript code via browser built-in engine
  - terminate when the code terminates (run-to-completion semantics)
  - then, next task from the queue is executed

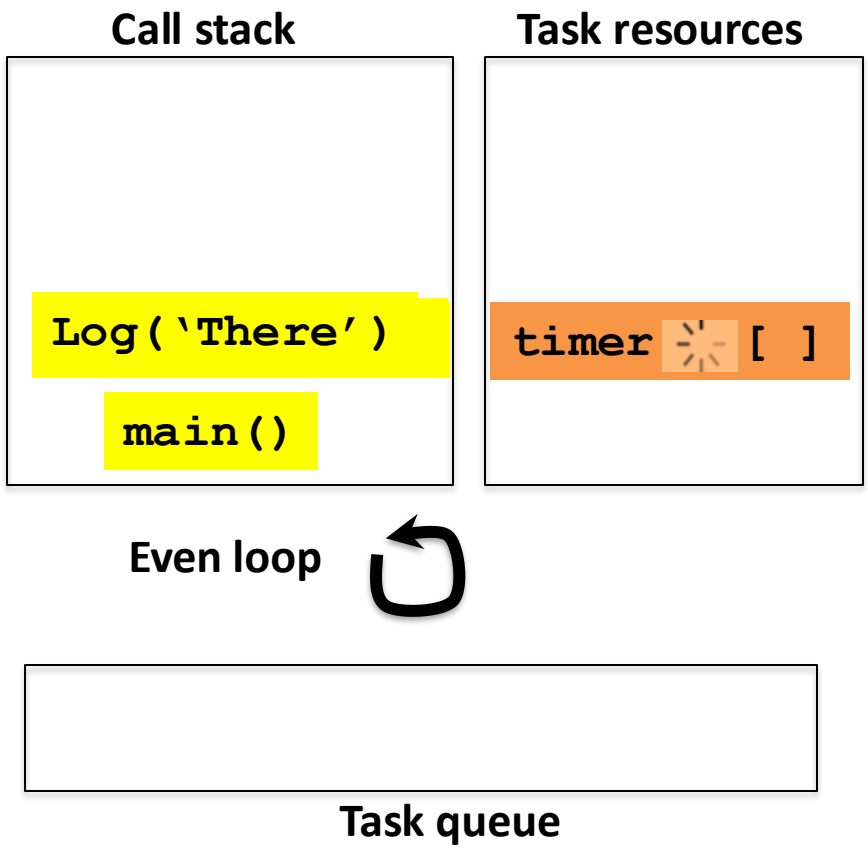
Browser Event Loop



- Other processes run in parallel (**timers**, input handling, *etc.*)
- They add tasks to its queue

Simulation With A Timer





Simulation With A Null Timer

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

setTimeout(function cb(){
    console.log('There'); },
0);

console.log('Bx INP');
```

Console

Hi

Bx INP

There