



Paradigmas de Programación (EIF-400) Nociones de Asincronía- Concurrencia y FP

CARLOS LORÍA-SÁENZ LORACARLOS@GMAIL.COM

AGOSTO 2017

EIF/UNA

Objetivos

2

- ▶ Mostrar el uso del paradigma de FP en los casos de asincronía y concurrencia
- ▶ Uso en arquitecturas SPA-Restfull
- ▶ Introducir el manejo asincronía en JS
- ▶ Nociones de arquitectura de JS en el tema non-blocking

Objetivos Específicos

3

- ▶ Event Loop JS
- ▶ CPS (y el Callback hell)
- ▶ Promesas en ES6 con FP
- ▶ Workers
- ▶ Segunda parte: async/await

Material

4

- ▶ Ver work en rar
- ▶ Ver Ejercicio_mejsn.rar

Síncrono vs Asíncrono en programación

5

- ▶ Síncrono: cada “paso” de la computación espera por el paso anterior en el control
- ▶ El siguiente paso está bloqueado por el previo
- ▶ Asíncrono es lo contrario: el control puede ser distinto al orden textual
- ▶ Un paso no bloquea necesariamente al siguiente

JS es mono-hilo

6

- ▶ Todo el JS corre en un solo hilo
- ▶ Eso minimiza problemas de acceso concurrente (por ejemplo al DOM)
- ▶ En Node sucede lo mismo. Pero por “debajo” maneja pool de hilos ([libuv](#))
- ▶ La idea es usar “delegación en la plataforma”: *“todo corre en concurrente excepto el código del programador”*
- ▶ El programador debe procurar no bloquear el hilo principal.
- ▶ Técnica: callbacks

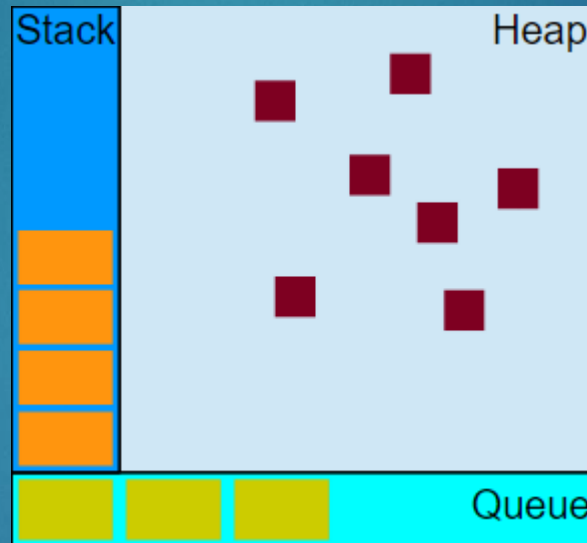
Arquitectura: asumimos una cola nada más

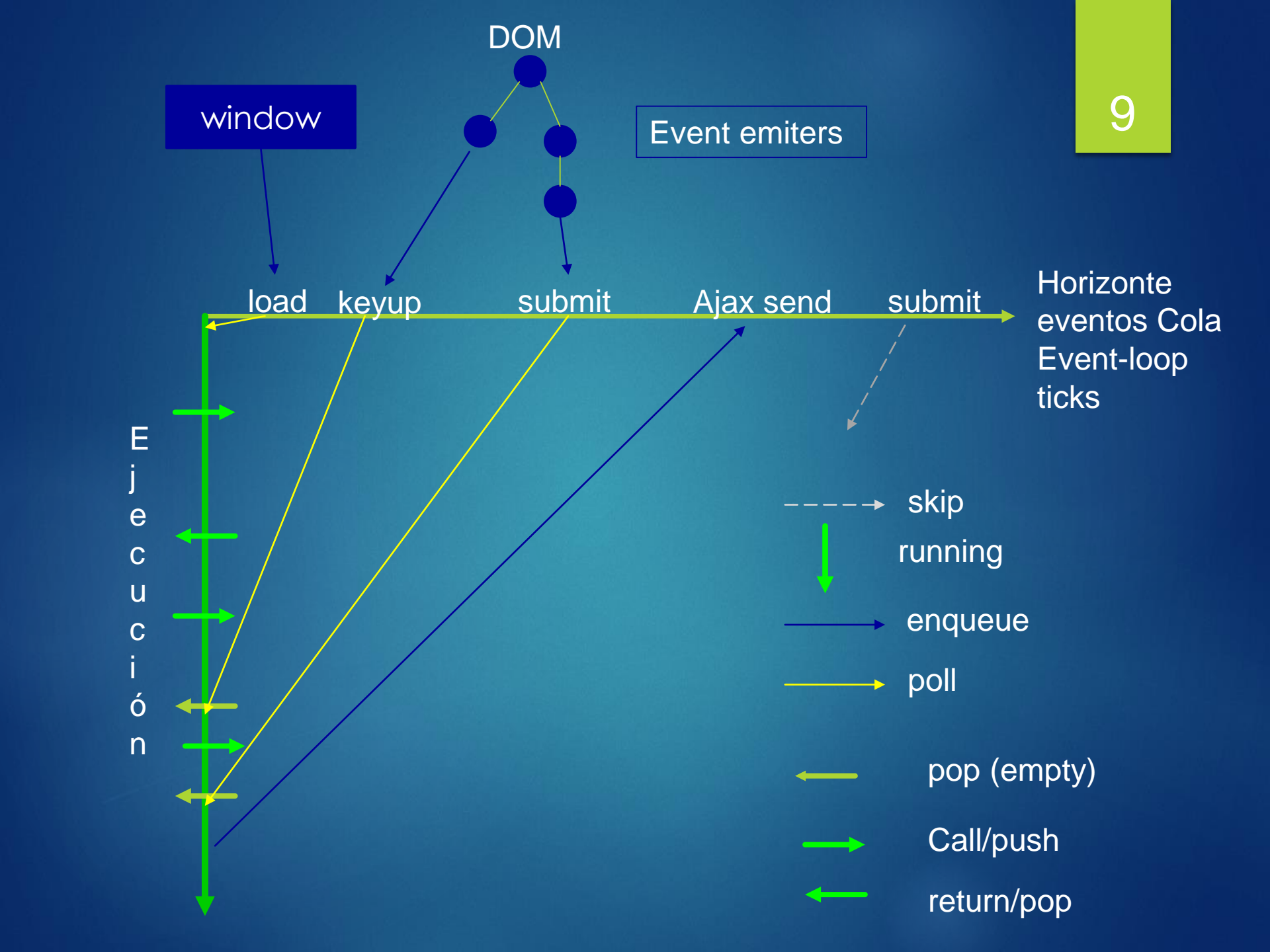
- ▶ Heap; memoria de objetos (Garbage-collected)
- ▶ (Call) Stack: pila de llamadas sincrónicas
- ▶ Event Queue(s): cola(s) de llamadas asincrónicas asociadas con eventos: timers, IO,
- ▶ En el browser: `setInterval`, `setTimeout` (`setImmediate`) ponen en cola la función pasada (`callback`)
- ▶ La cola se consulta hasta que el stack se vacíe
- ▶ Cada consulta a la cola se llama un “*tick*”

MDN Event Loop simplificado

8

- ▶ Ver [EventLoop](#)





Ejemplo: Eventos desde JS Browser

10

- ▶ Pruebe fireEvent.html



Ejemplo: Eventos en Node

11

- Considere `life.js`

```
PP:node life.js 3
>>> Testing Events (3 listeners) <<<
----- Starting Life -----
{ '0': { value: 7, listener: [Function: listener] },
  '1': { value: 5, listener: [Function: listener] },
  '2': { value: 7, listener: [Function: listener] } }

Listener 0 starts working with 7
Listener 1 starts working with 5
Listener 2 starts working with 7

>>> There are 3 listening to event <<<
Listener 0 starts working with 6
Listener 1 starts working with 4
Listener 2 starts working with 6

>>> There are 3 listening to event <<<
Listener 0 starts working with 5
Listener 1 starts working with 3
Listener 2 starts working with 5

>>> There are 3 listening to event <<<
Listener 0 starts working with 4
Listener 1 starts working with 2
Listener 2 starts working with 4

>>> There are 3 listening to event <<<
Listener 0 starts working with 3
Listener 1 starts working with 1
Listener 2 starts working with 3

>>> There are 3 listening to event <<<
Listener 0 starts working with 2
Listener 1 starts working with 0
*** Listener 1 is now dying ***
Listener 2 starts working with 2
```

Ejercicio

12

- ▶ Corra `foo_events.js`
- ▶ Explique la salida observada

```
PP:node foo_events.js  
foo_1  
foo_2  
hoo_1  
foo_3  
goo_1
```

Ejercicio

13

- ▶ Corra y estudie `block.js`
- ▶ Explique la salida observada

PP

```
PP:node block.js
```

```
In main at: 20:20:59 GMT-0600 (Hora estándar, América Central)
```

```
Exit blocking loop at: 20:21:04 GMT-0600 (Hora estándar, América Central)
```

```
Loop took 25091831 iterations.
```

```
Function Timed-out now starts at 20:21:04 GMT-0600 (Hora estándar, América Central)
```

```
PP:
```

Continuaciones (CPS)

14

- ▶ En JS/Node se usan call-backs
- ▶ Problema: composición. Solución: Promesas

```
PP
PP:node read_file.js bigfile.txt
Reading bigfile.txt

*** Look! Not blocked ***

OK 5647342 bytes were read
```

```
> var p = readFileAsPromise('read_file.js', 'utf8')
undefined
> p.then(text => console.log(text.length))
Promise {
  <pending>,
  domain:
    Domain {
      domain: null,
      _events: { error: [Function: debugDomainError] },
      _eventsCount: 1,
      _maxListeners: undefined,
      members: [] } }
> 591
```

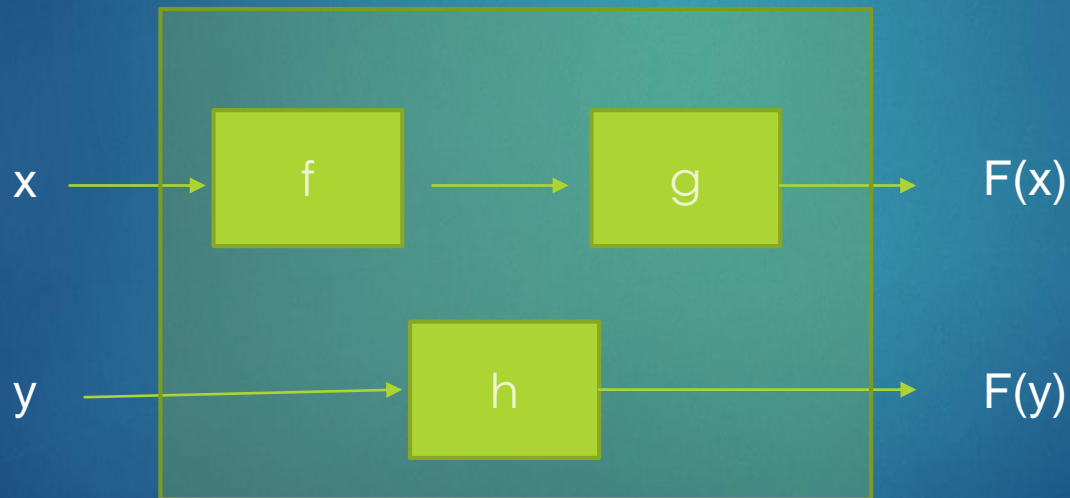

Promesas (en JS)

15

- ▶ Técnica para manejo de la `asincronía`
- ▶ Es un objeto que representa un valor no computado aún, pero que lo será eventualmente
- ▶ Es ya estándar en ES6
- ▶ Hay una especificación: `Promises/A+`
- ▶ Conceptos análogos: futuro, observable
- ▶ Promesa: lugar donde poner el valor
- ▶ Futuro: el valor a obtener (en Java)
- ▶ Observable: continuo de valores (`stream`, en Java)
- ▶ Computación `Lazy` vs `Eager`

Ejemplo

16



Asuma: f es
“pesada” en
CPU

Problema: composición

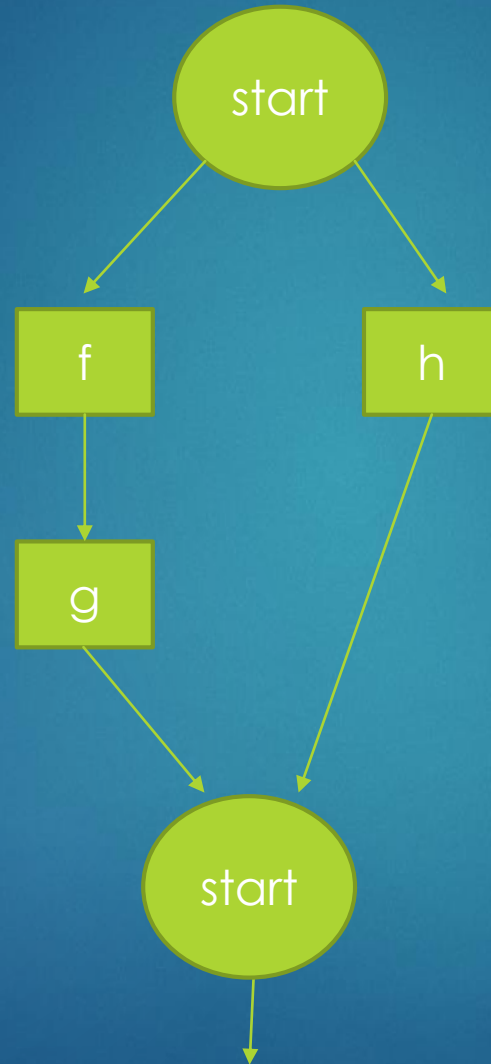
17

- ▶ ¿Cómo decir que g espere a f
- ▶ Y que el resultado final espere a los dos
- ▶ ¿Cómo componer asincrónicamente?

```
function F(x, y) {  
  let f = x => console.log('f -->' + x);  
  let g = x => console.log('g -->' + x);  
  let h = x => console.log('h -->' + x);  
  return [g(f(x)), h(y)];  
}
```

Diagrama: flujo de trabajo

18



Ver intro_F.js

```
PROMISE:node  
> .load intro_F.js  
> val  
Promise { [ 'g(f(1))', 'h(2)' ] }
```

Promise (ver API)

19

Promise

SEE ALSO

Standard built-in objects

Promise

▼ Properties

Promise.prototype

▼ Methods

Promise.all()

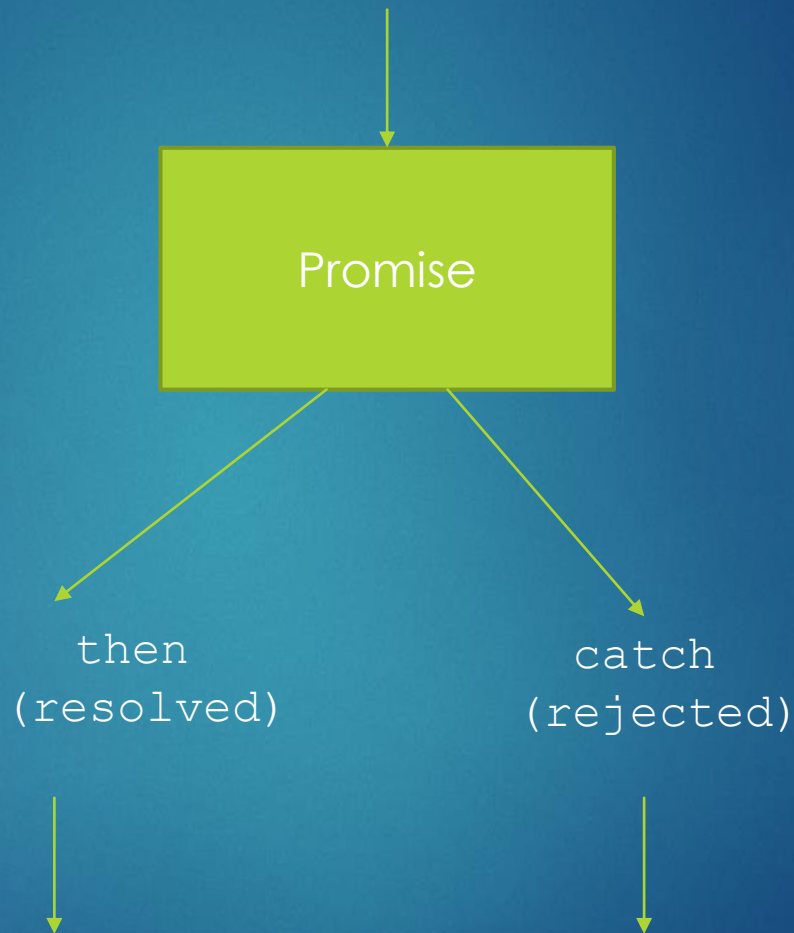
Promise.prototype.catch()

Promise.prototype.then()

Promise.race()

Promise.reject()

Promise.resolve()



Webworkers

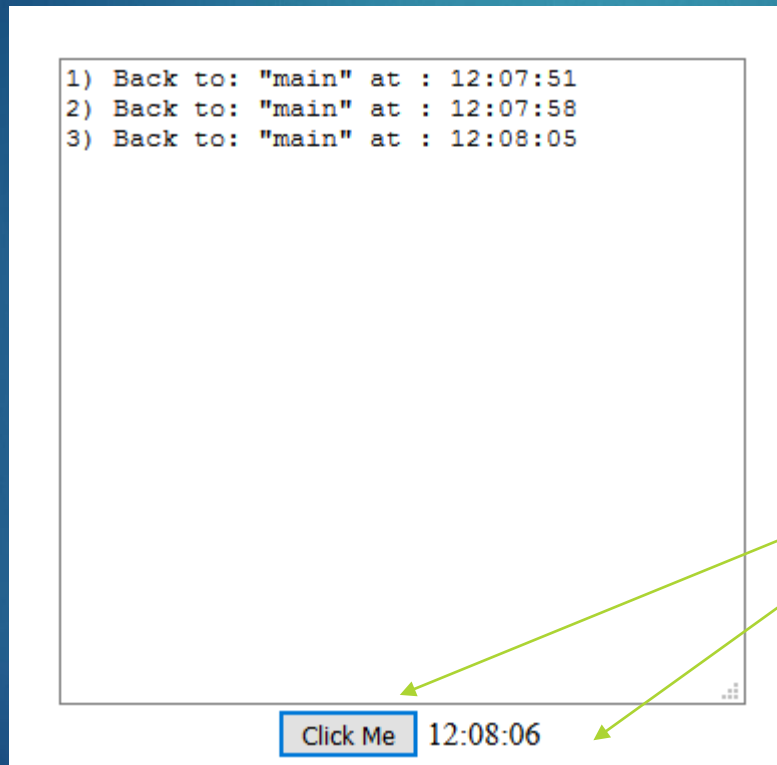
20

- ▶ Disponibles en HTML5
- ▶ El browser administra su ejecución en hilos
- ▶ No pueden alterar el DOM
- ▶ Tienen su propio stack y mandan mensajes a cola de eventos
- ▶ Se comunican con el hilo principal por eventos
- ▶ Modelo de “message-passing”

Web Workers

21

- ▶ Pruebe el demo `workerTest.html`



Se mantiene
responsiva

Ejercicio

22

- ▶ Mueva el código en JS en `workerTest.html` a Promesas

Continuará...

23

