Conceptos de JavaScript

Con toques del paradigma de Programación Funcional

Quién soy

Pepe Fabra Valverde

Líder y Arquitecto de Front

Disclaimer

- La sesión se va a grabar
- Al final de la sesión se compartirán las slides
- Pregunta sin miedo

No se requieren ni JavaScript ni TypeScript para esta charla, pero serán los lenguajes de elección para los ejemplos y toma de tierra de conceptos.

Agenda

- Pureza, efectos secundarios e idempotencia
  - **Ir a la sección**
- Memoización, serialización y comparación de valores
  - **Ir a la sección**
- Closures, alto orden y ciudadanía de primera clase
  - **Ir a la sección**

Orientado a...

- Quienes estén empezando
- Quienes que quieran adentrarse en el mundo de JavaScript
- Quienes quieran empezar a adentrarse en la programación funcional
- Quienes quieran descubrir algo nuevo hoy

Pureza, efectos secundarios e idempotencia

Qué veremos en esta sección

- Introducción a algunos conceptos de JavaScript
- Breve introducción a conceptos de Programación funcional
  - Pureza de las funciones
  - Efectos secundarios de una función
- Idempotencia

¿Qué utilidad nos aportan?

- Nos ayudan a entender mejor nuestra herramienta de trabajo, JavaScript, Java, Python
- Pueden orientarnos a un código de más claro y de mayor calidad
- Algunos de estos conceptos son preguntas de entrevistas técnicas
- Sirven de base para aplicar técnicas “más complejas”
- Son aplicables a otros lenguajes: Java, C#, Python, etc.

Programación funcional

La programación funcional es un paradigma de programación.

¿Qué es un paradigma de programación?

¿Qué es un paradigma de programación?

Un paradigma de programación no es más que un estilo de programación, con ventajas e inconvenientes. OOP/POO es uno muy conocido.

La checklist de la Programación Funcional

La “chicha”, ¿qué es la programación funcional?

- Programación declarativa
- Divide y vencerás
- Funciones más matemáticas
    - Con resultados determinísticos

Pureza

Definición de pureza

La pureza (pure, pureness) de una función es inversamente proporcional a la cantidad de efectos secundarios que tiene.

Menos efectos secundarios hacen de una función que sea más pura

Pureza y efectos secundarios

Las funciones puras tienen la menor cantidad posible de efectos secundarios.

La programación funcional entiende que los efectos secundarios a veces son necesarios

Pureza y el determinismo

Una función pura tiene un resultado **determinístico**.

Su resultado es esperable, es decir, puedes *determinar* su resultado si sabes qué argumentos le estás pasando.

Pureza, unificando conceptos

Una función pura es aquella con un resultado determinístico que tiene la menor cantidad posible de efectos secundarios.

Pero...

¿Qué es eso de un efecto secundario?

Efectos secundarios

Definiendo los efectos secundarios

Los efectos secundarios (side effects) de una función se pueden entender como acciones que van más allá del alcance (scope) de una función.

La checklist de un efecto secundario

- Depende de un estado (valor, atributo, propiedad, constante) no proporcionado como parámetro.
- Muta (modifica) un estado no local, fuera del cuerpo de la función.

Si cumple una de estas propiedades, la acción (instrucción) pasa a ser un efecto secundario

Entendiendo los efectos secundarios

Pero entonces, ¿no hay que tenerlos? Sí que hay que tener efectos secundarios, lo que hay que hacer es minimizar su uso.

Un efecto secundario **"daña" la traza**, es un **mock extra** dentro de un test, hace que el resultado de una función sea algo **más impredecible**.

Los efectos secundarios son como distracciones

Por eso la pureza es no tener efectos/distracciones **innecesarias**, es tener el contenido (código) sin adulterar

Ejemplos

Con ejemplos, todo se entiende mejor.

Pongamos que hemos recuperado información acerca de un usuario de la BDD, pero no queremos devolver toda la información, para ello, tenemos una función que filtra el cuerpo de la respuesta.

Es un efecto secundario cuando...

Un efecto secundario sería recuperar la información del usuario en la misma función que hace el filtro.

O guardar en una variable global la información del usuario que se ha recuperado y acceder a esta para filtrar los campos a devolver.

En TypeScript, ***formatCurrentUser(): ResponseUser***

En Java, ***ResponseUser formatCurrentUser()***

No sería un efecto secundario si...

Siguiendo el ejemplo de antes, no sería un efecto secundario si nuestra función recibe la información del usuario y devuelve la respuesta.

En TypeScript, ***formatUser(user: User): ResponseUser***

En Java, ***ResponseUser formatUser(User user)***

Contextualizando en JavaScript

En caso de estar en el frontend, este mismo caso puede darse.

Hemos recuperado la información del usuario del endpoint adecuado. Suponiendo que nuestra app no es multidioma.

No recuperaríamos el usuario directamente de un store, tendríamos una función que se encargaría de formatear la información del usuario, ***formatUser(user: User): string***

Idempotencia

Qué es la idempotencia

La idempotencia (idempotence, idempotent) es la propiedad de una función matemática de ser completamente pura.

Es decir, no tiene efectos secundarios y su resultado será siempre el mismo, dado que le pasemos los mismos argumentos.

Beneficios de la idempotencia

- Pureza de las funciones (trazabilidad)
- Posibilidad de ahorrarse computaciones
  - Un mismo resultado con los mismos argumentos podría cachearse…
- Decir una palabra no tan conocida para algo que es comúnmente conocido

“Pega” de la idempotencia

No todo puede, ni ha de ser, idempotente.

Ejemplos de idempotencia

- **Suma**
  - 1 + 1 a veces da 7, pero 2 + 2 siempre dará 4
- **Factorial**
  - El factorial de un número será siempre el mismo, *factorial(3) == 6*

Ejemplos que no son idempotentes

- **Math.random**
  - Casi siempre devolverá un número diferente
- **Time.now** o **Date.now**
  - El tiempo está en constante cambio

Recapitulemos...

Hemos visto...

- **Programación funcional**
  - Un paradigma de programación, *divide y vencerás*
- **Pureza**
  - Una función con la menor cantidad de efectos secundarios y un resultado determinístico
- **Efectos secundarios**
  - Mutaciones fuera del alcance de la función
- **Idempotencia**
  - Una función pura, *mismos parámetros == mismo resultado*

Lo cuál nos permite...

- Escribir código más claro y de mayor calidad
- Entender mejor el ecosistema
- Hacer buenas entrevistas técnicas
- Nuevos caminos ante los problemas de siempre

Y ahora... ¿Por dónde podría continuar?

En siguientes episodios…

Pasos a seguir

- Memoización orientada a JavaScript
  - Serialización
    - Comparación de valores

- Closure
- Ciudadanía de primera clase
  - Alto orden
    - Funciones de alto orden
    - Componentes de alto orden

Preguntas

Memoización, serialización y comparación de valores

Hasta ahora sabemos que...

Programación funcional	**Pureza**	**Efectos secundarios**	**Idempotencia**
Un paradigma de programación	Una función determinística con la menor cantidad de efectos secundarios	Acciones fuera del alcance de una función	Término matemático para la pureza de una función, generalizando

¿Qué vamos a ver a continuación?

- Memoización
- Serialización en JavaScript
- Comparación de valores en JavaScript

¿Por qué? ¿Qué aportará?

- Podemos aumentar el rendimiento de las soluciones
- Serialización puede abrir puertas a diferentes soluciones
- Saber comparar valores en JavaScript tiende a evitar quebraderos de cabeza
- Comprender mejor la pesadilla de tipados en JavaScript ayuda a entender ciertas decisiones
- Algunos de estos conceptos se preguntan en pruebas técnicas

Memoización

Qué es

Una solución de cache para funciones idempotentes.

Es decir, si los mismos parámetros dan el mismo resultado, calcular los mismos parámetros una vez garantiza que su resultado se puede almacenar sin riesgo de error.

¿Qué se tiene que dar?

Las condiciones que se tienen que dar son las siguientes:

- Tenemos una función idempotente
- Dicha función consume recursos para computarse
- Podemos serializar sus argumentos correctamente
- El tiempo de serialización y comparación es menor que el de computación

Antes que nada

Para conseguir implementar la memoización se tienen que dar condiciones, pero también tenemos unos requisitos:

- Poder serializar valores
- …y para poder serializar, necesitamos poder comparar valores

Comparación de valores

La mayoría de lenguajes comprende los valores entre dos tipos:

- Primitivos
- No primitivos

Entiendiendo el problema

Primitivos

- Undefined, number, string, boolean, Bigint, Symbol

No primitivos

- Objects, functions (también objects), classes (esto también ocurre en otros lenguajes)

Curiosidad de los tipados en JavaScript

¿No se echa en falta algún tipo primitivo del listado anterior?

Resolviendo la Curiosidad de los tipados en JavaScript

- Null

El primitivo null en realidad no tiene tipo, typeof null === “object”.

Se entiende que null es la ausencia de instanciación de un objeto, por tanto, su tipo es el de un objeto:

https://v8.dev/blog/react-cliff

Métodos de comparación en JavaScript

Existen diferentes métodos de comparación:

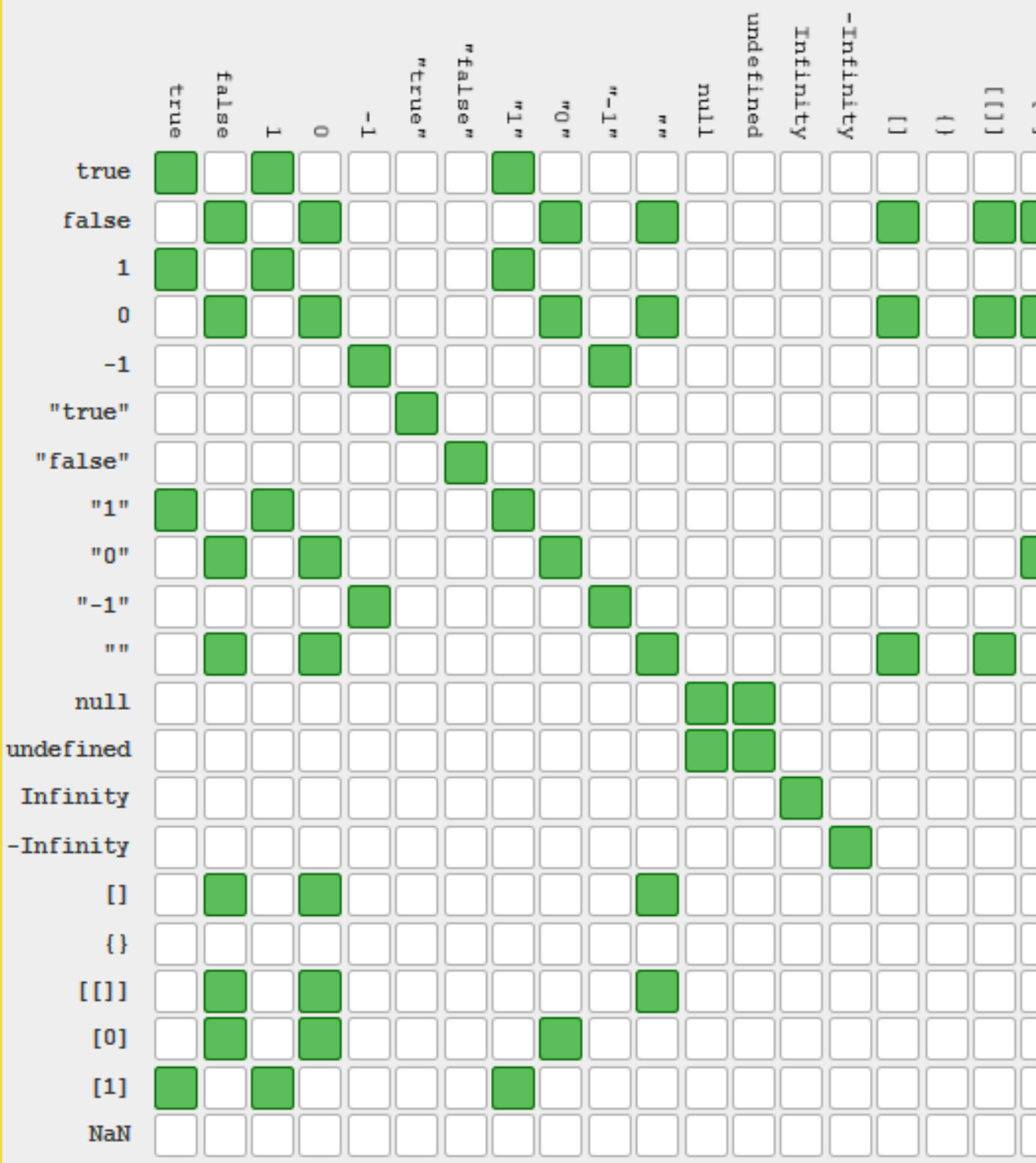
- Loosely equal (==)
- Strictly equal (===)
- Same Value (Object.is)

Más información en: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Equality_comparisons_and_sameness

Comparación simple (loosely equal)

Evitar a toda costa esta comparación.

La siguiente table expresa cómo funciona



Comparación exacta (strictly equal)

Entonces, cómo deberíamos comparar en JavaScript, o bien con strictly equal (===) o con same value (Object.is).

Strictly equal ofrece el comportamiento esperado en la mayoría de lenguajes por el ==

Comparación por mismo valor (same value)

Existen valores numéricos especiales gracias a IEEE 754-2008

- Infinity
- -Infinity
- NaN
- -0

Y esta locura existe en todos los lenguajes, pero aún queda algo más:

NaN es diferente de sí mismo...

Aquí es donde JavaScript viene al rescate, ***Object.is*** puede ayudarnos con este tipo de comparaciones.

Serialización

¿Qué es serializar?

La definición más clara y concisa es la de:

Persistir objetos, bien sea para almacenarlos en un fichero, base de datos, o enviarlos por red.

¿Por qué lo necesitamos?

- Porque los valores no primitivos son puñeteros en comparaciones
- Necesitamos poder comparar valores primitivos y no primitivos sin ningún tipo de discriminación…

Y en este último punto, la serialización nos puede echar una mano.

¿Cuál será el resultado de esta expresión?

```
{ foo: "bar" } === { foo: "bar" };
```

Vamos a simplificarlo...

```
{} === {};
```

Más todavía...

```
new Object() === new Object();
```

La raíz del problema en las comparaciones

Cuando usamos la *keyword* ***new*** estamos asignando una nueva dirección en memoria.

Entonces, lo que estamos comparando, no son los “valores”, sino sus referencias en memoria, dos direcciones completamente diferentes.

Esto se debe a cómo se pueden pasar los argumentos en programación

Argumentos en programación

Los argumentos se pueden pasar de dos maneras diferentes:

- Por **valor**, tipos primitivos (number, string, bool, etc.)
- Por **referencia**, tipos no primitivos (objects, functions, etc.)

Los valores por **referencia** son los costosos de comparar

Situando el problema

Como hemos visto anteriormente, la comparación de valores, y por tanto la serialización, trae quebraderos de cabeza, en especial en JavaScript.

Pero necesitamos poder serializar para memoizar…

Entonces, ¿cómo lo hacemos?

Tenemos varias opciones:

- JSON.stringify, que no es de fiar
- Implementar una solución a mano
- Buscar soluciones existentes, *no adivinarás cuál es la más recomendable…*

Por qué no deberías fiarte de JSON.STRINGIFY

Si bien es cierto que puede serializar valores, es demasiado preciso.

```
JSON.stringify({a:”B”, b:”A”}) !== JSON.stringify({b:”A”, a:”B”})
```

Por qué no deberías fiarte de JSON.STRINGIFY

Si bien es cierto que puede serializar valores, es demasiado preciso.

```
JSON.stringify({a:”B”, b:”A”}) !== JSON.stringify({b:”A”, a:”B”})
```

Al compararlo, nos dirá que no es el mismo valor, porque el orden de las keys ha cambiado, entonces ha comparado

El orden de las claves influye en el resultado de la comparación

```
'{"a":"B", "b":"A"}' !== '{"b":"A", "a":"B"}’
```

El orden de los factores **sí** altera el producto

Solución a mano

Si descartamos el uso de JSON.stringify, una solución a mano no parece tan mala idea, ¿verdad?

Todo el código hay que mantenerlo, probarlo bien, y cubrir los casos límite, máxime en secciones genéricas… y todavía no hemos hablado de los objetos.

```
function serializeValue(value) {
  if (value === null || value === undefined) {
    return value;
  } else if (typeof value === "object") {
    return serializeObject(value);
  } else if (typeof value === "number") {
    return value;
  } else if (value instanceof Set) {
    return Array.from(value.values()).map(serializeValue);
  } else if (value instanceof Map) {
    return Array.from(value.entries()).map(([key, value]) => [
      key,
      serializeValue(value),
    ]);
  }

  return value.toString();
}
```

Devalue, implementa lo JSON.stringify que no

Una de las reglas del desarrollo de software es, no reinventar la rueda.

> *A wise engineering solution would produce—or better, exploit—reusable parts.*
>
> Doug McIlroy, More shell, less egs

Nuestra solución base será del creador de Svelte, Rich Harris

https://github.com/Rich-Harris/devalue

Implementación

Código en vivo

Es bien sabido que, probar cosas en una presentación, es de las mejores prácticas.

Empecemos por lo fácil

Una función que sume dos números

```
function suma(a: number, b: number): number {
  return a + b;
}
```

Vamos a complicarlo

Ahora vamos a ejecutar el factorial, con recursividad

```
function factorial(n: number): number {
  if (n <= 1) return 1;
  return factorial(n - 1) * n;
}
```

Posibilidad de mejora

¿Hay alguna cosa que, vistos los ejemplos, podría mejorarse?

Propuesta de mejora

La lógica para memoizar comparte elementos en común.

Un closure podría ayudarnos a memoizar más fácilmente.

Mejora: construye un closure que memoice la función que le pases, sin recursividad.

Recapitulando...

Hemos visto...

- **Memoización**
  - Solución de cache para funciones idempotentes
- **Serialización**
  - Convertir a un primitivo cualquier elemento en memoria
- **Comparación de valores**
  - Valor y referencia, null es un pritimitivo, pero no tiene tipo

Y en episodios anteriores...

- **Programación funcional**
  - Un paradigma de programación, *divide y vencerás*
- **Pureza**
  - Una función con la menor cantidad de efectos secundarios y un resultado determinístico
- **Efectos secundarios**
  - Mutaciones fuera del alcance de la función
- **Idempotencia**
  - Una función pura, *mismos parámetros == mismo resultado*

Lo cuál nos permite...

- Entender mejor las comparaciones
- Solucionar problemas de rellamadas
- Evitar computaciones pesadas
  - Sin abusar, o traerás nuevos problemas encima de la mesa
- Mejores entrevistas
- Y puede que ahora **React**ciones mejor a cierto framework

Siguientes pasos...

- Closure
- Ciudadanía de primera clase
  - Alto orden
    - Funciones de alto orden
    - Componentes de alto orden

Preguntas

Closures, alto orden y ciudadanía de primera clase

Contextualizando...

- **Pureza**
  - Funciones determinísticas que mitigan la cantidad de efectos secundarios
- **Efectos Secundarios**
  - Acciones más allá del cuerpo de la función
- **Idempotencia**
  - Término matemático para la pureza
- **Memoización**
  - Cache para funciones idempotentes
- **Serialización (en JavaScript)**
  - Conversión de valores primitivos y no primitivos a string
- **Comparación de valores**
  - Problemática, pero parcheable

¿Qué veremos?

- Closures
  - Contextos
  - Encapsulación privada
- Ciudadanía de primera clase
- Alto orden
  - Funciones de alto orden
  - Componentes de alto orden (React)

Closure

¿Qué es un Closure?

Un closure en JavaScript es una función creada dentro de otra

Pero también es:

- Un contexto
- Encapsulación privada

Contexto

A veces, a lo largo del desarrollo, hay partes que querríamos extraer, pero que no podemos, porque necesitan de un ***contexto***.

¿Y si te dijese que eso se puede hacer?

Solo necesitamos algo que nos ayude a ***proveer*** ese *contexto* y listo.

Veamos más beneficios de los closures.

Encapsulación privada

La encapsulación privada es un concepto común en lenguajes orientados a objetos (Java, C#, etc.)

Se dice que JavaScript no tiene propiedades privadas

- Y esto es cierto… a medias

¿Qué entendemos por encapsulación privada?

¿Qué entendemos por encapsulación privada?

Ocultar la implementación, controlar el acceso a propiedades, en definitiva, proteger la implementación.

Esto es algo que podemos conseguir con los closures, no tienen el mismo comportamiento que las propiedades privadas, pero casi

Ejemplo de closure

```ts
type SetState<T> = [T | undefined, (value: T) => void];

const useState = function <TValue>(defaultValue?: TValue): SetState<TValue> {
  let value = defaultValue;
  const setValue = (newValue: TValue) => (value = newValue);
  return [value, setValue];
};
```

```ts
const [count, setCount] = useState(1);
// count es "inaccesible" desde fuera
console.log(count); // 1
setCount(3);
console.log(count); // 3
```

Generación de funciones y eventos

En runtime, es decir, podemos “enriquecer” el ecosistema, si vamos con control

```
const generadorDeMultiplicaciones =
(multiplicador: number) => {
  return (a: number) => a * multiplicador;
};
```

```
const multiplicatePorCero = generadorDeMultiplicaciones(0);
multiplicatePorCero(1); // 0
multiplicatePorCero(2); // 0
multiplicatePorCero(4); // 0
```

```
const duplica = generadorDeMultiplicaciones(2);
duplica(2); // 4
```

Ciudadanía de primera clase

¿Qué es la ciudadanía de primera clase?

¿Qué es la ciudadanía de primera clase?

Una entidad que permite la mayoría de operaciones disponibles a otras entidades:

- Pasarse como argumento de una función (input)
- Devolverse de una función (output)
- Asignarse una variable

Ciudadanía de segunda clase

Un término un poco absurdo, que tiene ciertas restricciones en comparación:

- No puede ser asignado a una variable
- O pasado como parámetro
- O devuelto de una función

Una clase sería el ejemplo idóneo de esta casuística

Antes de nada

¿Se diferencia en algo de un closure?

No debería...

La ciudadanía de primera clase y un closure…

- Son diseño e implementación en JavaScript

Esto significa que el concepto se implementa con closures, en JavaScript, diferentes lenguajes proveen de diferentes herramientas.

¿Cómo implementarías el concepto?

Adentrándonos en el concepto

Pero entonces, si la ciudadanía es de primera clase, ¿cómo llamamos a esos ciudadanos?

Es decir, si son funciones ¿cómo distinguimos a una función de primera clase con otra de segunda clase?

Alto orden

¿Qué es algo de alto orden?

No, no tiene que ver con la orden 66

- Es un elemento de primer nivel en la programación
- Un estilo de programación que puede usar primitivos y no primitivos como valores
- Para este contexto, es similar a la ciudadanía de primera clase

La entidad más conocida de este grupo son las Funciones de Alto Orden, que veremos a continuación

Elementos de alto orden

Existen distintos elementos de alto orden, veremos los siguientes:

- Funciones de alto orden
- Componentes de alto orden (orientado a React)

Función de alto orden

High-**O**rder **F**unctions o **HOF**s

Su definición oficial dice que cumplen una o más de estas condiciones:

- Reciben como input una o más funciones
- Devuelven como output una o más funciones

Es decir, closures, un closure puede recibir una función o devolver otra.

Componente de Alto Orden

High-**O**rder **C**omponents o **HOC**s, proveen de contexto al children

```
<Context.Provider>
    <ChildrenComponentWithoutContext />
</Context.Provider>
```

Recapitulemos...

Conceptos revisitados

Hemos visto ciudadanías, alto orden y closures.

La ciudadanía de primera clase permite algunas operaciones comunes (actuar como un valor) que la ciudadanía "de segunda" no permite (una clase no puede ser devuelta de una función, una instancia sí).

El alto orden, para pasar entidades y devolverlas, y como un closure en JavaScript es una función de alto orden

Dudas

Son conceptos complejos y algo teóricos, ahora podría ser un muy buen momento para comentar las dudas.

Caminos abiertos

Ahora podemos entender que trabajar con contextos puede ser más sencillo, y que las funciones se pueden adaptar según los valores del runtime.

El contexto de un closure puede dar ideas sobre cómo implementar algunos conceptos, o incluso crear algunos nuevos.

Se acerca el fin de la sesión

Ya solo quedan un par de extras y el cierre :D

Bonus

Currify

- Qué es
- Ejemplos y casos de uso
- Disclaimer

MapReduce

- Adopción general
- Uso en Big Data para pipelines
  - Spark, Hadoop

- **Map** -> modifica elemento a elemento de la colección
  - mantiene longitud de la colección
- **Reduce** -> modifica la colección recorriendo elemento a elemento
  - Crea una nueva haciendo un forEach de la colección que no tiene por qué mantener la misma longitud

La buena solución

- Es la que acaba pareciendo obvia en retrospectiva
- Usa las partes que realmente aporten valor

Conclusiones

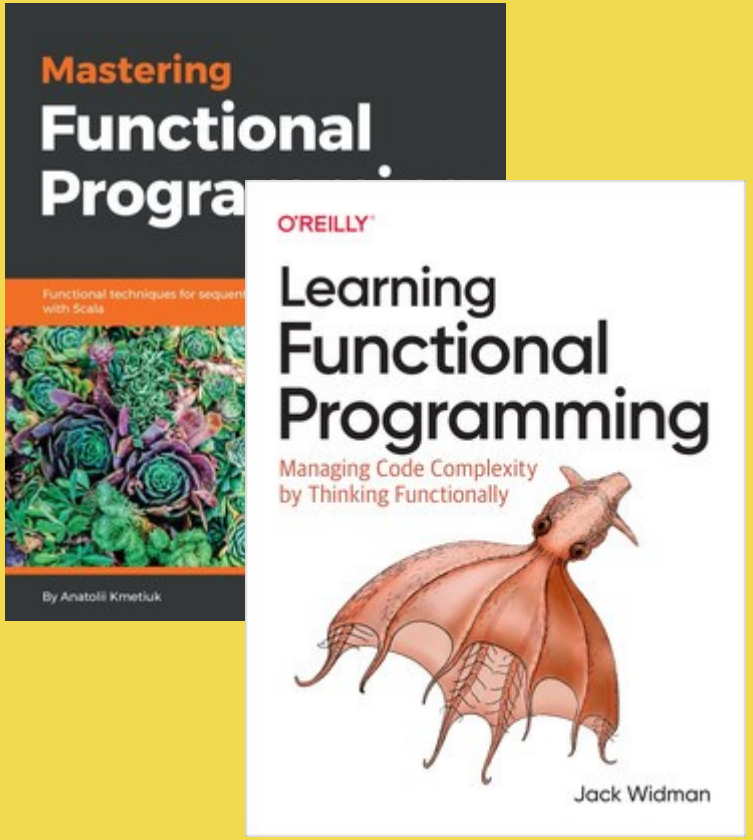
Libros

Learning Functional Programming
by Jack Widman, O'Reilly

O'Reilly: https://www.oreilly.com/library/view/learning-functional-programming/9781098111748/
Amazon: https://amzn.eu/d/0CVqCi6

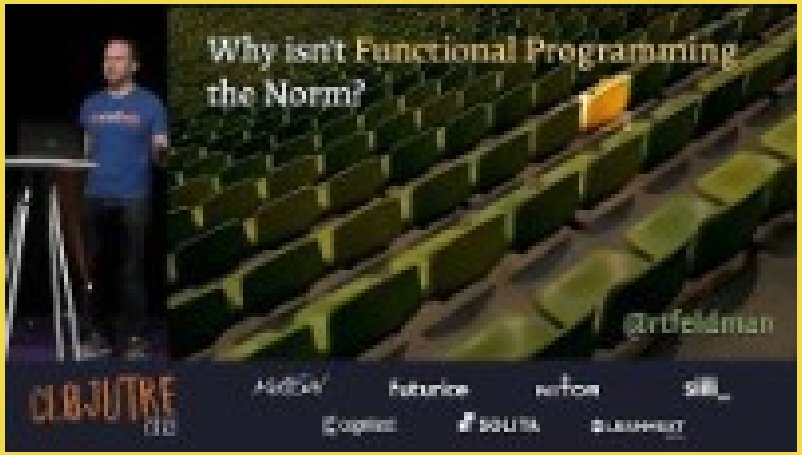
Mastering Functional Programming
by Anatolii Kmetiuk, Packt Publishing

Packt: https://www.packtpub.com/product/mastering-functional-programming/9781788620796
Amazon: https://a.co/d/dBo8L3m



Charlas

Anjana Vakil: Aprendiendo Programación Funcional con JavaScript - JSUnconf 2016



Why Isn't Functional Programming the Norm? – Richard Feldman

Functional Programming for Pragmatists • Richard Feldman • GOTO 2021

Artículo

Puedes leer el artículo al respecto en:

- https://medium.com/@jofaval/a60130f073ef

¡¡CUIDADO!! El contenido es muy parecido al de esta charla

Artículo

Puedes encontrar el artículo más detallado y con ejemplos en:

- https://medium.com/@jofaval/d26fc09e149

Hay algunas implementaciones a mano a modo de ejemplo.

Just Javascript

con Dan Abramov && Maggie Appleton

https://justjavascript.com/

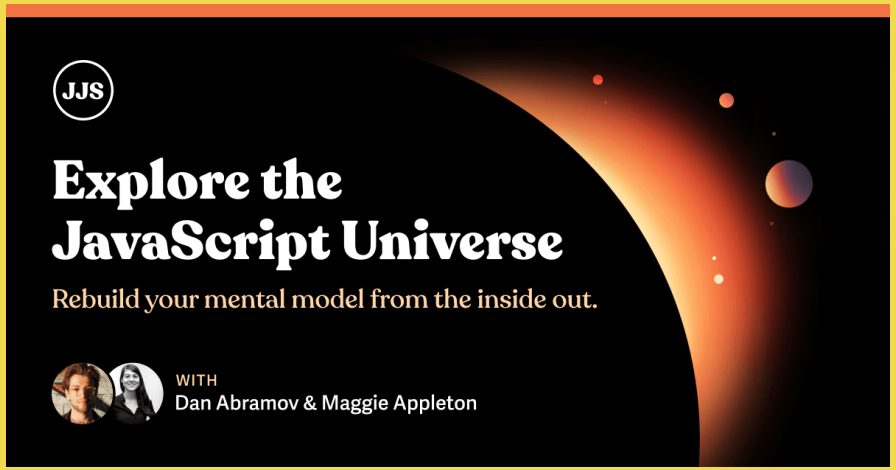
Artículos

- https://medium.com/@jofaval/a60130f073ef
- https://medium.com/@jofaval/d26fc09e149

Libros

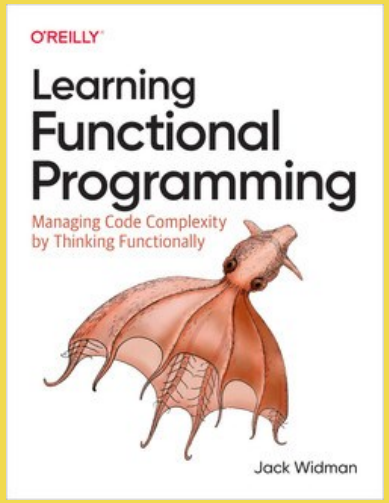
https://amzn.eu/d/0CVqCi6

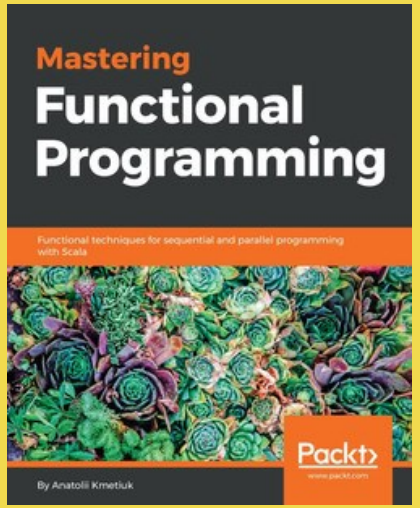
Cursos



https://justjavascript.com/

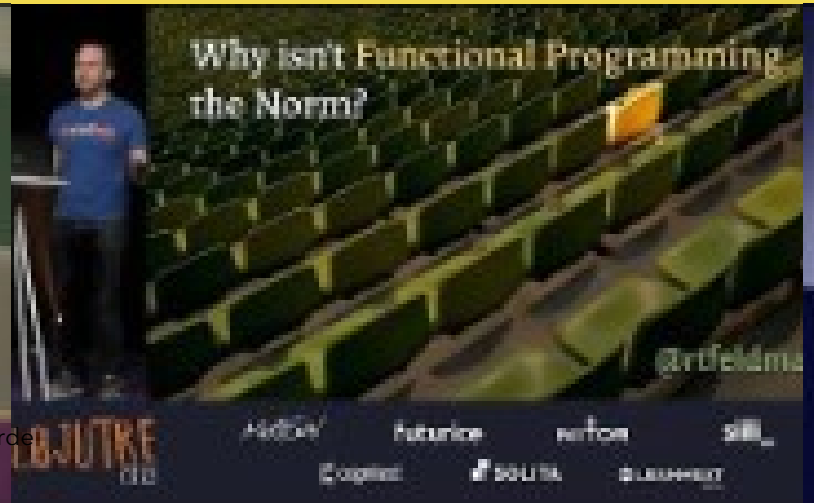
Programando…





Charlas

https://a.co/d/dBo8L3m



¿Por dónde podría investigar más?

- Localidad
- Inmutabilidad (programación funcional y estructuras de datos)
- Libros y charlas
- Reevalúa código escrito recientemente

QR de las slides

Enlace al repositorio

Encuéntrame en

- Linkedin - **linkedin.com/in/jofaval/**
- Github - **github.com/jofaval**

Preguntas

Conceptos de JavaScript

Con toques del paradigma de Programación Funcional

Gracias por la atención