

# *Taller de Taller de Programación Funcional en JavaScript*

Javier Vélez Reyes

@javiervelezreye

Javier.velez.reyes@gmail.com

Enero 2016



# Taller de Programación Funcional en JavaScript

## Presentación

### ¿Quién Soy?



Licenciado por la UPM desde el año 2001 y doctor en informática por la UNED desde el año 2009, Javier conjuga sus labores como profesor e investigador con la consultoría y la formación técnica para empresa. Su línea de trabajo actual se centra en la innovación y desarrollo de tecnologías para la Web. Además realiza actividades de evangelización y divulgación en diversas comunidades IT y es coordinador de varios grupos de ámbito local como NodeJS Madrid o Madrid JS. Forma parte del programa Polymer Polytechnic Speaker y es mentor del capítulo de Madrid de Node School.



[javier.velez.reyes@gmail.com](mailto:javier.velez.reyes@gmail.com)



[@javiervelezreye](https://twitter.com/@javiervelezreye)



[linkedin.com/in/javiervelezreyes](https://linkedin.com/in/javiervelezreyes)



[gplus.to/javiervelezreyes](https://gplus.to/javiervelezreyes)



[jvelez77](https://facebook.com/jvelez77)



[javiervelezreyes](https://github.com/javiervelezreyes)



[youtube.com/user/javiervelezreyes](https://youtube.com/user/javiervelezreyes)



**Javier Vélez Reyes**  
@javiervelezreyes  
Javier.veler.reyes@gmail.com

# 1 *Introducción*

- Introducción
- Objetivos de la Programación Funcional
- Principios de Diseño Funcional

*Taller de Programación Funcional en JS*

*Introducción*

# Taller de Programación Funcional en JavaScript

## Introducción

### I. Introducción

La programación funcional es un viejo conocido dentro del mundo del desarrollo. No obstante, en los últimos años está cogiendo tracción debido, entre otros factores, a la emergencia de arquitecturas reactivas, al uso de esquemas funcionales en el marco de Big Data y al creciente soporte que se le da desde diversas plataformas vigentes de desarrollo. JavaScript ha sido siempre un lenguaje con fuerte tendencia al diseño funcional. En este texto revisaremos sus objetivos y principios y discutiremos los mecanismos, técnicas y patrones empleados actualmente para construir arquitecturas funcionales haciendo uso de JavaScript.

#### Programación Funcional

*Arquitecturas  
centradas en la  
transformación*

*Variantes  
funcionales*

*Inmutabilidad*

*Transparencia  
Referencial*

*Compositividad*

*Arquitecturas  
dirigidas por flujos de  
datos*

#### Programación Orientada a Objetos

*Puntos de Extensión  
Polimórfica*

*Arquitecturas  
centradas en la  
abstracciones*

*Sustitutividad  
Liskoviana*

*Encapsulación de  
estado*

*Arquitecturas  
dirigidas por flujos de  
control*

# Taller de Programación Funcional en JavaScript

## Introducción

## II. Objetivos de la Programación Funcional

### A. Especificación Declarativa

La programación funcional persigue diseñar especificaciones de comportamiento abstracto que se centren en la descripción de las características de un problema más que en una forma particular de resolverlo. De acuerdo a esto, se entiende que la responsabilidad de encontrar una solución para el problema descansa no tanto en manos del programador sino en la arquitectura funcional subyacente que toma en cuenta dicha especificación.

#### Funcional Qué

*La descripción fluida del estilo funcional permite entender fácilmente la especificación*

```
function total (type) {  
  return basket.filter (function (e) {  
    return e.type === type;  
  }).reduce (function (ac, e) {  
    return ac + e.amount * e.price;  
  }, 0);  
}
```

```
function total (type) {  
  var result = 0;  
  for (var idx = 0; idx < basket.length; idx++) {  
    var item = basket[idx];  
    if (basket[idx].type === type)  
      result += item.amount * item.price;  
  }  
  return result;  
}
```

*Aunque operativamente es equivalente, el estilo imperativo es más confuso y resulta más difícil de entender*

```
var basket = [  
  { product: 'oranges', type: 'food', amount: 2, price: 15 },  
  { product: 'bleach', type: 'home', amount: 2, price: 15 },  
  { product: 'pears', type: 'food', amount: 3, price: 45 },  
  { product: 'apples', type: 'food', amount: 3, price: 25 },  
  { product: 'gloves', type: 'home', amount: 1, price: 10 }  
];
```

#### Cómo

#### Objetos

# Taller de Programación Funcional en JavaScript

## Introducción

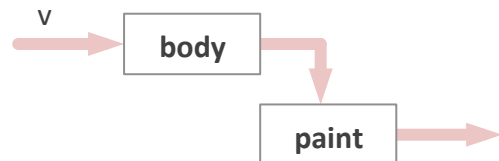
## II. Objetivos de la Programación Funcional

### B. Abstracción Funcional

La programación funcional persigue alcanzar una especificación de alto nivel que capture esquemas algorítmicos de aplicación transversal. Como veremos a continuación, así se fomenta la reutilización y la adaptación idiomática. Esto se consigue con arquitecturas centradas en la variabilidad funcional y contrasta ortogonalmente con la orientación a objetos.

#### Funcional

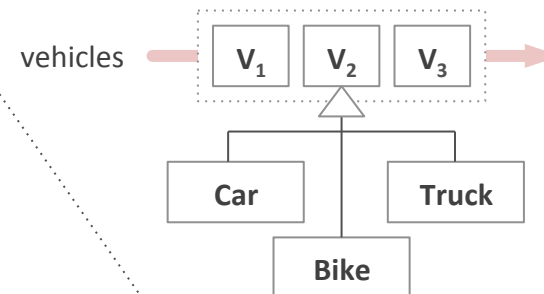
#### Abstracción Funcional



```
var phases = [
  function body (v) {...},
  function paint (v) {...}
];
var test = function (phases) {
  function (vehicle) {
    return phases.reduce(function (ac, fn) {
      return fn(ac);
    }, vehicle);
  }
};
```

*En FP la función es la unidad de abstracción ya que permite definir esquemas algorítmicos que dependen de funciones pasadas como parámetros*

```
var vehicles = [new Car(), new Truck(), ...];
var garage = function (vehicles) {
  for (v in vehicles)
    vehicles[v].test();
};
garage(vehicles);
```



*En OOP los algoritmos se distribuyen entre objetos con implementaciones variantes*

#### Abstracción de tipos Objetos

# Taller de Programación Funcional en JavaScript

## Introducción

## II. Objetivos de la Programación Funcional

### C. Reutilización Funcional

Como consecuencia de la capacidad de abstracción en la especificación declarativa, la programación funcional alcanza cotas elevadas en la reutilización de funciones. Este nivel de reutilización algorítmica no se consigue en otros paradigmas como en la orientación a objetos.

```
var get = function (collection) {  
    return function (filter, reducer, base) {  
        return collection  
            .filter(filter)  
            .reduce(reducer, base);  
    };  
};
```

*El esquema algorítmico se reutiliza sobre distintas estructuras de datos y con distintas aplicaciones funcionales*

```
var users = [  
    { name: 'jvelez', sex: 'M', age: 35 },  
    { name: 'eperez', sex: 'F', age: 15 },  
    { name: 'jlopez', sex: 'M', age: 26 }  
];  
var adult = function (u) { return u.age > 18; };  
var name = function (ac, u) {  
    ac.push(u.name);  
    return ac;  
};  
get(users)(adult, name, []);
```

```
var basket = [  
    { product: 'oranges', type: 'F', price: 15 },  
    { product: 'bleach', type: 'H', price: 15 },  
    { product: 'pears', type: 'F', price: 45 },  
];  
var food = function (p) { return p.type === 'F'; };  
var total = function (ac, p) {  
    return ac + p.price;  
};  
get(basket)(food, total, 0);
```

# Taller de Programación Funcional en JavaScript

## Introducción

## II. Objetivos de la Programación Funcional

### D. Adaptación Funcional

Una de las virtudes de la programación funcional es la capacidad de poder transformar la morfología de las funciones para adaptarlas a cada escenario de aplicación real. En términos concretos esto significa poder cambiar la signatura de la función y en especial la forma en que se proporcionan los parámetros de entrada y resultados de salida. Veremos que existen mecanismos técnicos y patrones para articular este proceso.

```
function greater (x, y) {  
  return x > y;  
}
```

**reverse**



*Se invierte el orden de aplicación de los parámetros de forma transparente*

```
function greater (x, y) {  
  return y > x;  
}
```

**curry**



*Se transforma la evaluación de la función para que pueda ser evaluada por fases*

```
(function greater (x) {  
  function (y) {  
    return y > x;  
  }  
})(18)
```

**partial**



*Se configura parcialmente la función para obtener el predicado de adulto*

```
function greater (x) {  
  function (y) {  
    return y > x;  
  }  
}
```



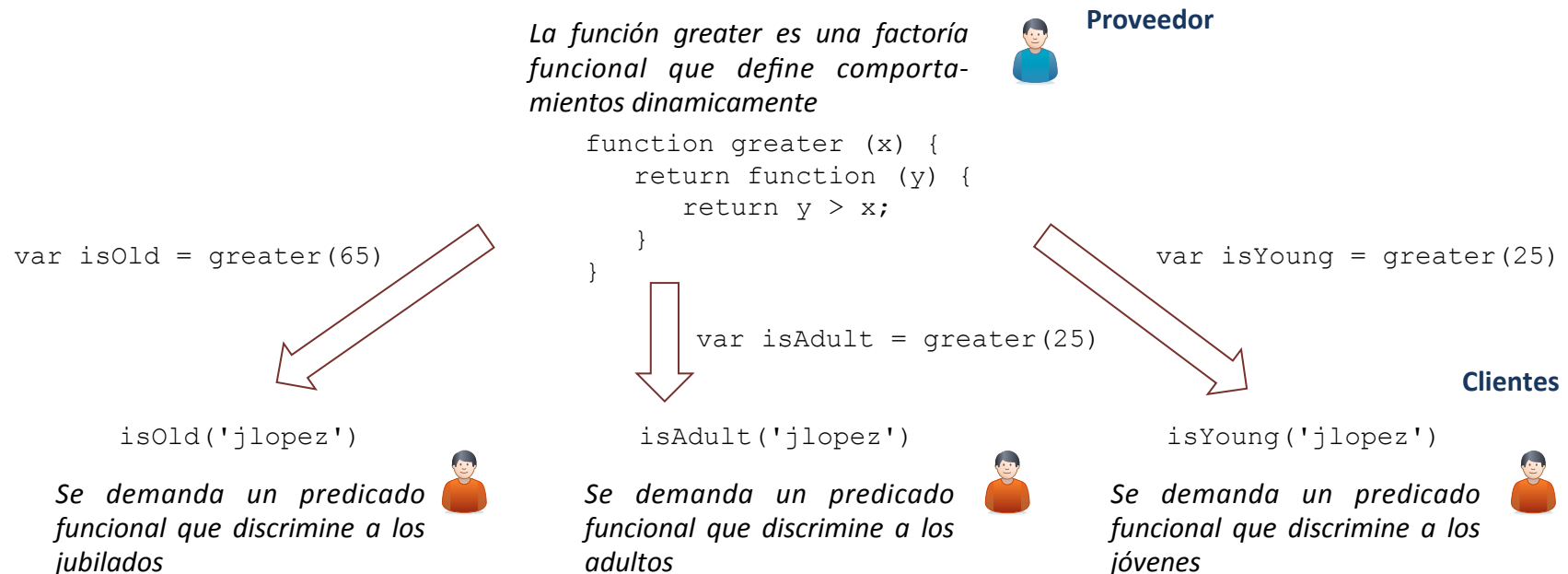
# Taller de Programación Funcional en JavaScript

## Introducción

## II. Objetivos de la Programación Funcional

### E. Dinamicidad Funcional

La programación funcional permite diseñar funciones que sean capaces de construir abstracciones funcionales de forma dinámica en respuesta a la demanda del contexto de aplicación. Como veremos, este objetivo se apoya en mecanismos de abstracción y también abunda en la adaptación funcional que acabamos de describir.



# Taller de Programación Funcional en JavaScript

## Introducción

### III. Principios de Diseño Funcional

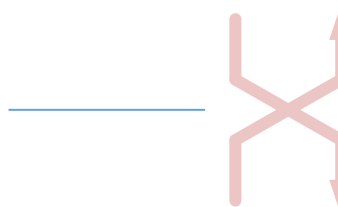
#### A. Principio de Transparencia Referencial

El principio de transparencia referencial predica que en toda especificación funcional correcta, cualquier función debe poder substituirse en cualquier ámbito de aplicación por su expresión funcional sin que ello afecte al resultado obtenido. La interpretación de este principio de diseño tiene tres lecturas complementarias.

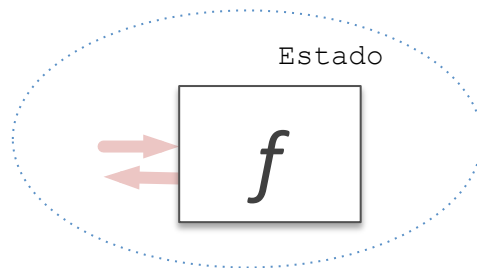
##### Funciones Puras

*Dado que el predicado greater sólo depende de sus parámetros de entrada  $x$  e  $y$ , se trata de una función pura. Por ello, mantiene la transparencia referencial permitiendo su sustitución sin impacto en la semántica del contexto de aplicación*

```
var old = greater(65)
```



```
var old = (function greater (x) {  
  function (y) {  
    return y > x;  
  }  
})(65);
```



Se dice que una función es pura – recuerda al estilo de comportamiento matemático – si su valor de retorno sólo depende de los parámetros de entrada y no del estado ambiental (variables globales, variables retenidas en ámbito léxico, operaciones de E/S, etc.).

# Taller de Programación Funcional en JavaScript

## Introducción

### III. Principios de Diseño Funcional

#### A. Principio de Transparencia Referencial

El principio de transparencia referencial predica que en toda especificación funcional correcta, cualquier función debe poder substituirse en cualquier ámbito de aplicación por su expresión funcional sin que ello afecte al resultado obtenido. La interpretación de este principio de diseño tiene tres lecturas complementarias.

##### Comportamiento Idempotente

###### Estilo Funcional

```
function Stack () {  
  return { stack: [] };  
}  
function push (s, e) {  
  return {  
    stack: s.stack.concat(e),  
    top: e  
  };  
}  
function pop (s) {  
  var stack = [].concat(s.stack);  
  var e = stack.pop();  
  return {  
    stack: stack,  
    top: e  
  };  
}
```

*La pila mantiene el estado interno y las operaciones push y pop no resultan idempotentes*

*El estado se mantiene externamente y se pasa como parámetro a cada operación con lo que el comportamiento es idempotente*

###### Estilo Orientado a Objetos

```
function Stack () {  
  var items = [];  
  return {  
    push: function (e) {  
      items.push(e);  
    },  
    pop: function () {  
      return items.pop();  
    }  
  };  
}
```

Se dice que una función es idempotente si siempre devuelve el mismo resultado para los mismos parámetros de entrada. En nuestro ejemplo debería verificarse esta igualdad:

```
pop + pop === 2 * pop
```

# Taller de Programación Funcional en JavaScript

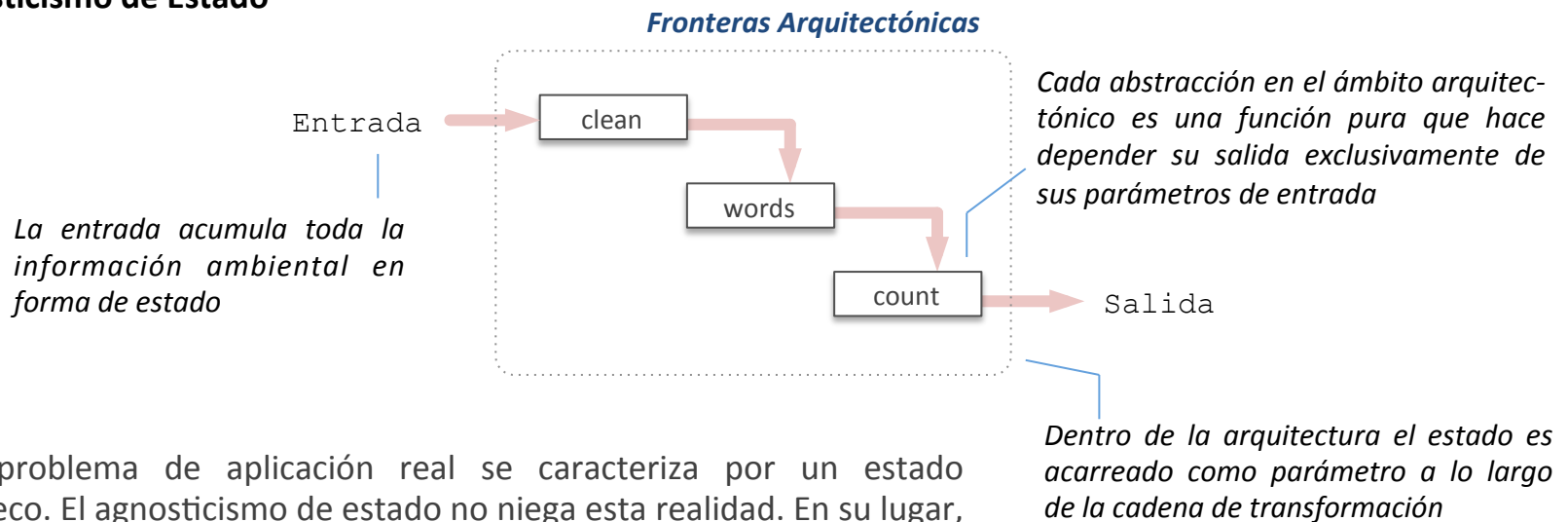
## Introducción

### III. Principios de Diseño Funcional

#### A. Principio de Transparencia Referencial

El principio de transparencia referencial predica que en toda especificación funcional correcta, cualquier función debe poder substituirse en cualquier ámbito de aplicación por su expresión funcional sin que ello afecte al resultado obtenido. La interpretación de este principio de diseño tiene tres lecturas complementarias.

##### Agnosticismo de Estado



Todo problema de aplicación real se caracteriza por un estado intrínseco. El agnosticismo de estado no niega esta realidad. En su lugar, el principio debe interpretarse como que dentro de las fronteras del sistema el comportamiento funcional sólo puede depender de los parámetros explícitamente definidos en cada función.

# Taller de Programación Funcional en JavaScript

## Introducción

### III. Principios de Diseño Funcional

#### B. Principio de Inmutabilidad de Datos

En el modelo de programación imperativa, la operativa de computo se entiende como un proceso secuencial y paulatino de transformación del estado mantenido por la arquitectura hardware de la máquina. En programación funcional – y en virtud de la transparencia referencial – las funciones no dependen del estado ambiental. De ello se colige que el concepto de variable entendido como depósito actualizable de información no existe.

##### Imperativo

```
var x = 1
...
x = x + 1
```



##### P. Funcional

```
x@(0) = 1
...
x@(t+1) = x@(t) + 1
```

*En programación imperativa las variables se entienden como depósitos de información en memoria que pueden actualizarse durante el tiempo de vida del programa. La dimensión tiempo queda oculta y esto dificulta el razonamiento*

*La programación funcional pone estrés en que los cambios en los datos de un programa deben manifestarse como funciones sobre la dimensión tiempo. Esto significa que el concepto de variables como depósito de información y las operaciones de actualización sobre ellas no están permitidas*

En términos prácticos la aplicación de este principio se traduce en que las funciones nunca deben actualizar los parámetros de entrada sino generar a partir de ellos resultados de salida

```
function push (s, e) {
  return s.push(e);
}
```

##### Estilo Imperativo

*Se modifica s*

```
function push(s, e) {
  return s.concat(e);
}
```

##### Estilo Funcional

*No se modifica s*

# Taller de Programación Funcional en JavaScript

## Introducción

### III. Principios de Diseño Funcional

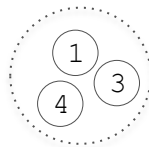
#### C. Principio de Computación Centrada en Colecciones

A diferencia de lo que ocurre en otros paradigmas como en orientación a objetos donde los datos se organizan en forma de grafos relacionales, en programación funcional éstos deben modelarse como colecciones – arrays o diccionarios en JavaScript – para que se les pueda sacar pleno partido<sup>1</sup>. Se dice que el modelo de computación de la programación funcional está centrado en colecciones.

##### Set

*Un conjunto se modela como un array o un diccionario de booleanos*

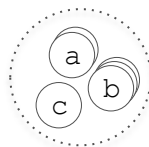
```
[true,
false,
true,
true]
```



##### Bag

*Una bolsa se modela como un diccionario de contadores*

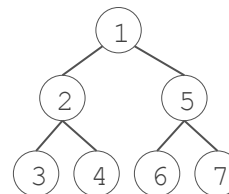
```
{a: 2,
b: 3,
c: 1}
```



##### Binary Tree

*Un array se aplana como una colección anidada de arrays de 3 posiciones, árbol izquierdo, nodo, árbol derecho*

```
[
  [
    [], 3, [],
    2,
    [], 4, []],
  1,
  [[], 6, []],
  5,
  [], 7, []]
]
```



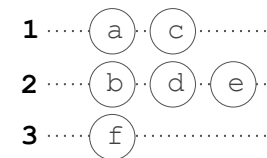
##### Heap

*Una cola con prioridad es un diccionario de arrays o un array de arrays*

```
{1: [a, c],
2: [b, c, d],
3: [f]}
```

```
[
  [a, c],
  [b, c, d],
  [f]
]
```

...



<sup>1</sup> Los diccionarios son una aproximación de representación aceptable en tanto que pueden procesarse como colecciones de pares clave-valor

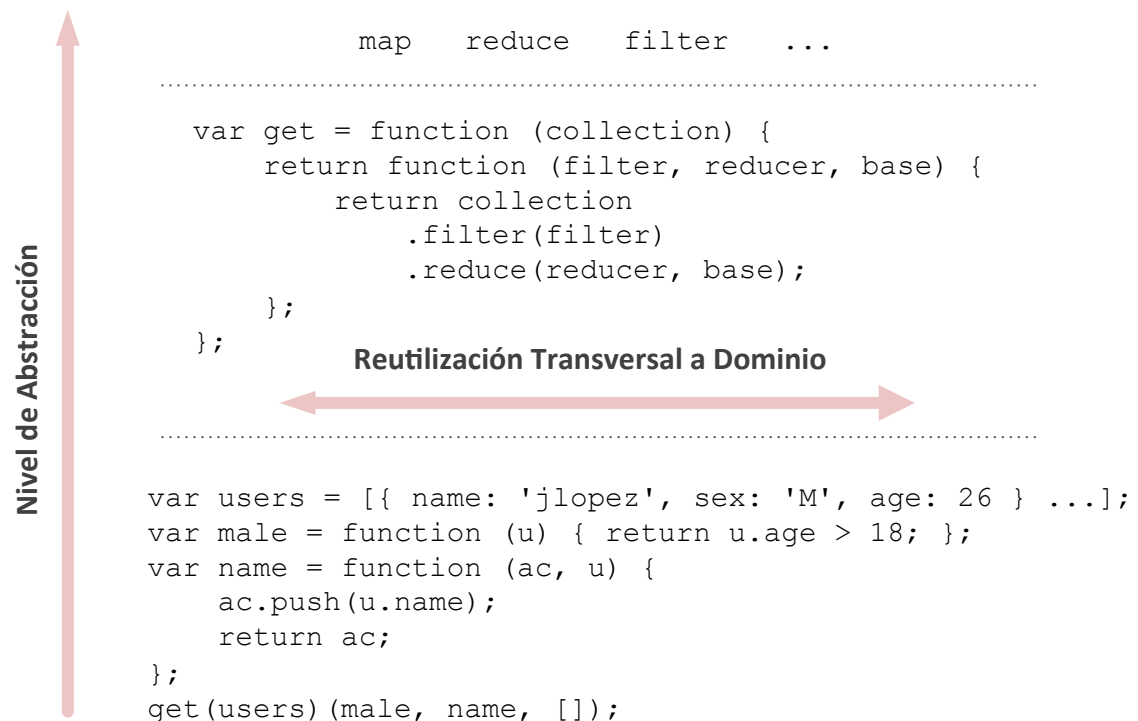
# Taller de Programación Funcional en JavaScript

## Introducción

### III. Principios de Diseño Funcional

#### D. Principio de Diseño Dirigido por Capas

La construcción de arquitecturas funcionales sigue un proceso de diseño dirigido por capas o niveles de especificación. Desde las abstracciones más generales encontradas en librerías funcionales se construye un nivel idiomático que propone un esquema abstracto de solución para el problema propuesto y sólo entonces se concreta en el dominio particular del problema.



#### Nivel de Librería

*Se utilizan funciones generales con un alto nivel de abstracción*

#### Nivel Idiomático

*Se crea un esquema funcional que resuelve la familia de problemas al que pertenece el problema planteado*

#### Nivel de Dominio

*Se contextualiza el esquema anterior dentro del dominio de aplicación concretando con datos y funciones específicas*

**Javier Vélez Reyes**  
@javiervelezreyes  
Javier.veler.reyes@gmail.com

# 2 *JavaScript como Lenguaje Funcional*

- Definición Funcional por Casos
- Definición por Recursión
- Expresiones Funcionales de Invocación Inmediata IIFE
- Definición en Orden Superior
- Clausuras & Retención de Variables

*Taller de Programación Funcional en JS*  
*JavaScript como Lenguaje Funcional*



# Taller de Programación Funcional en JavaScript

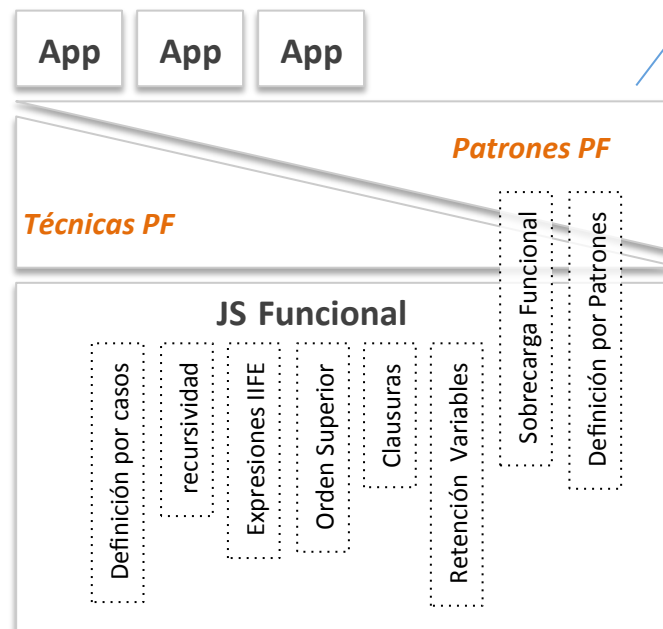
## JavaScript como Lenguaje Funcional

### I. Introducción

Como ya hemos comentado, los procesos de programación funcional consisten esencialmente en la definición de una colección de declaraciones funcionales que puedan adaptarse a distintos niveles de abstracción y aplicarse conjugadamente para resolver problemas de forma declarativa. Para articular estas tareas de definición el paradigma funcional ofrece una colección de mecanismos esenciales que revisamos en este capítulo.

#### Nivel de Lenguaje

*El nivel de lenguaje está formado por la colección de mecanismos de programación que proporciona el lenguaje de forma nativa*



#### Nivel Idiomático

*Las deficiencias que por su concepción presenta el lenguaje se deben suplir a este nivel por aplicación de técnicas y patrones de diseño que revisaremos en próximos capítulos*

# Taller de Programación Funcional en JavaScript

## JavaScript como Lenguaje Funcional

### II. Mecanismos de Programación Funcional

#### A. Definición Funcional por Casos

La forma más sencilla de definir una función es diseñando explícitamente la colección de valores de retorno que debe devolver para cada posible parámetro de entrada. Aunque este mecanismo parece poco flexible aisladamente, cuando se conjuga con otras formas de definición resulta de aplicación frecuente.

##### Única Expresión

*El uso anidado del operador condicional permite expresar los pares caso-resultado como una única expresión de retorno*

```
function f (parámetros) {  
  return caso-1 ? resultado-1 :  
         caso-2 ? resultado-2 :  
         resultado-defecto;  
}
```

##### Enumeración Explícita de casos

*La definición por casos se expresa declarando explícitamente el valor que debe devolver la función para cada posible valor de los parámetros de entrada*

```
function comparator(x) {  
  return x > 0    ? 1 :  
         x === 0 ? 0 :  
         -1 ;  
}
```

*Se utiliza el operador condicional anidado para definir cada caso funcional*

```
function even(n) {  
  return n % 2;  
}
```

*La definición mediante una única expresión es el escenario más sencillo de definición por casos*

# Taller de Programación Funcional en JavaScript

## JavaScript como Lenguaje Funcional

## II. Mecanismos de Programación Funcional

### B. Definición por Recursión

La definición de funciones por recursión consiste en la invocación de la propia función como parte del proceso de definición. El esquema de definición recursivo se apoya en el mecanismo anterior para distinguir entre casos base – aquéllos que devuelven un resultado final – de aquéllos casos recursivos – donde el valor de retorno vuelve a invocar a la propia función.

#### Casos base

*Los casos base son el final del proceso recursivo y se suponen soluciones inmediatas a problemas sencillos*

```
function f (parámetros) {  
  return caso-base-1 ? resultado-base-1 :  
         caso-base-2 ? resultado-base-2 :  
         caso-recursivo-1 ? resultado-recursivo-1 :  
         caso-recursivo-2 ? resultado-recursivo-2 :  
         resultado-defecto ;  
}
```

#### Casos recursivos

*Los casos recursivos se diseñan invocando a la propia función sobre valores de parámetros que convergen a los casos base*

#### I. Regla de cobertura

*Asegúrese de que todos los casos base del problema han sido incluidos explícitamente en la definición de la función*

#### II. Regla de convergencia

*Asegúrese de que cada resultado recursivo converge a alguno de los casos base*

#### III. Regla de autodefinition

*Para diseñar cada caso recursivo asuma que la función por definir está ya definida para valores de parámetros más próximos a los casos base*

# Taller de Programación Funcional en JavaScript

## JavaScript como Lenguaje Funcional

## II. Mecanismos de Programación Funcional

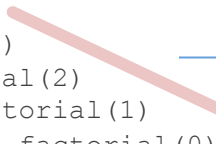
### B. Definición por Recursión

En función de cómo se expresan los casos recursivos de una función podemos distinguir entre dos tipos de situaciones. En la recursión directa, los casos recursivos de la función se expresan en términos de llamadas a ella misma. En la recursión indirecta, la función invoca a otra función que recurre sobre la primera.

#### Recursión Directa

```
function factorial (n) {  
  return n === 0 ? 1 :  
    n * factorial(n - 1);  
}
```


```
factorial(4)  
= 4 * factorial(3)  
= 4 * 3 * factorial(2)  
= 4 * 3 * 2 * factorial(1)  
= 4 * 3 * 2 * 1 * factorial(0)  
= 4 * 3 * 2 * 1 * 1  
= 24
```



*Los casos recursivos convergen hacia los casos base reduciendo el tamaño del problema en una unidad a cada paso*

*La convergencia hacia los casos base va incluyendo un operador de negación que se resuelven al llegar al caso base*

```
even(4)  
= !odd(3)  
= !!even(2)  
= !!!odd(1)  
= !!!!even(0)  
= !!!!true  
= true
```



#### Recursión Indirecta

```
function even (n) {  
  return n === 0 ? true :  
    !odd(n-1);  
}  
function odd (n) {  
  return n === 0 ? false :  
    !even(n-1);  
}
```

# Taller de Programación Funcional en JavaScript

## JavaScript como Lenguaje Funcional

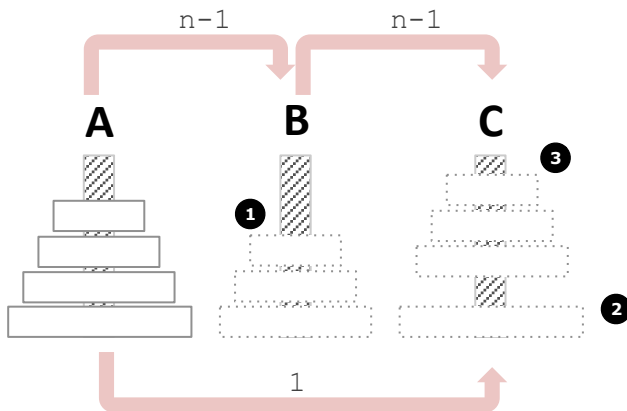
## II. Mecanismos de Programación Funcional

### B. Definición por Recursión

Dado que la programación funcional no dispone de estructuras de control de flujo, la única forma de resolver problemas que requieren un computo iterativo es expresarlas a partir de una formulación recursiva. Aunque puede resultar en principio más complejo, la expresividad funcional aumenta con este tipo de expresiones. Además hay problemas que tienen una resolución inherentemente recursiva.

#### Problema de las Torres de Hanoi

Mover discos de uno en uno para dejarlos en la misma posición de A en C usando B como auxiliar. Los discos por encima de uno dado deben ser siempre de menor tamaño que éste.



```
function hanoi (n, origen, aux, destino) {  
  if (n === 1) mover(origen, destino);  
  else {  
    hanoi(n-1, origen, destino, aux);  
    mover(origen, destino);  
    hanoi(n-1, aux, origen, destino);  
  }  
}  
  
function mover (origen, destino) {  
  destino.push (origen.pop());  
}  
  
var A = [4, 3, 2, 1], B = [], C = []  
hanoi(A, B, C);
```

# Taller de Programación Funcional en JavaScript

## JavaScript como Lenguaje Funcional

## II. Mecanismos de Programación Funcional

### C. Expresiones Funcionales e Invocación Inmediata

No en pocas ocasiones la necesidad de definir una función se restringe a un solo uso que se realiza a continuación de su definición. Si encerramos una definición de función entre paréntesis y aplicamos a la expresión resultante el operador de invocación – los parámetros actuales a su vez entre parentesis y separados por comas – obtenemos una expresión funcional de invocación inmediata IIFE.

#### Expresión Funcional

*La definición funcional se encierra entre paréntesis para convertirla en una expresión evaluable*

```
(function f (parámetros-formales) {  
    <<cuerpo de declaración funcional>>  
}) (parámetros-actuales);
```

#### Invocación Inmediata

*La expresión funcional se invoca directamente por aplicación de los parámetros actuales*

```
var cfg = (function config(user) {  
    // getting config from DB  
    return {...}  
}) ('jvelez');
```

*Un típico ejemplo de IIFE se da cuando se pretenden realizar cálculos complejos o tareas que se efectuarán una sola vez a lo largo del código*

*La ventaja de este tipo de construcciones es que la función se libera de la memoria tras su ejecución*

# Taller de Programación Funcional en JavaScript

## JavaScript como Lenguaje Funcional

## II. Mecanismos de Programación Funcional

### D. Definición en Orden Superior

En JavaScript las funciones son ciudadanos de primer orden. Esto quiere decir que a todos los efectos no existe ningún tipo de diferencia entre una función y cualquier valor de otro tipo primitivo. Esto permite a una función recibir otras funciones como parámetros o devolver funciones como retorno. Este mecanismo se le conoce con el nombre de orden superior.

#### Funciones como Parámetros a Otras Funciones

El paso de funciones como parámetros a una función permite que el comportamiento de ésta sea adaptada por medio de la inyección de código variante en cada invocación. Esta aproximación contrasta con la programación por tipos variantes propia de la orientación a objetos.

```
function once (fn) {  
  var called = false;  
  return function () {  
    if (!called) {  
      called = true;  
      return fn.apply(this, arguments);  
    }  
  }  
}
```

```
var greater = function (x, y) {...};  
var less    = function (x, y) {...};  
var even    = function (x) {...};  
  
[1, 4, 3, 2].sort(greater);  
[1, 4, 3, 2].sort(less);  
[1, 4, 3, 2].filter(even);
```

#### Funciones como Retorno de Otras Funciones

Las funciones son capaces de generar dinámicamente funciones que devuelven como resultado. Esto es de especial interés cuando la función devuelta es el resultado de transformar una función recibida como argumento. Este proceso justifica el nombre de orden superior ya que se opera con funciones genéricas.

# Taller de Programación Funcional en JavaScript

## JavaScript como Lenguaje Funcional

## II. Mecanismos de Programación Funcional

### E. Clausuras & Retención de Variables

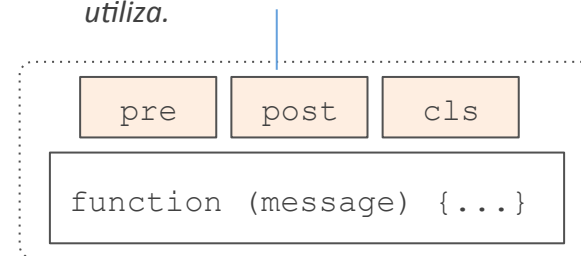
Las funciones que, por medio de los mecanismos de orden superior, devuelven funciones como resultado se pueden expresar en términos de variables locales o parámetros presentes en el entorno funcional donde se definen. La retención de variables es el mecanismo mediante el cual dichas variables y parámetros son mantenidos durante todo el tiempo de vida de la función de retorno. Este tipo de funciones se denominan clausuras.

```
function Logger(cls) {  
  var pre = 'Logger';  
  var post = '...';  
  return function (message) {  
    console.log ('%s[%s] - [%s]%s',  
                pre, cls, message, post);  
  }  
}
```

```
var log = Logger('My Script');  
log('starting');  
log(1234);  
log('end');
```

#### Variables Retenidas

*Al extraer una función fuera de su ámbito léxico de definición se mantiene el contexto de variables y parámetros que dicha función utiliza.*



#### Clausura

*Las clausuras son un mecanismos de construcción funcional que captura de forma permanente un estado de configuración que condiciona su comportamiento*



**Javier Vélez Reyes**  
@javiervelezreyes  
Javier.veler.reyes@gmail.com

# 3 *Técnicas de Programación Funcional*

- Abstracción
- Encapsulación
- Inmersión por Recursión y Acumulación
- Evaluación Parcial
- Composición Funcional & Composición Monádica

*Taller de Programación Funcional en JS*  
*Técnicas de Programación Funcional*

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### I. Introducción

A partir de los mecanismos que ofrece JavaScript como lenguaje de programación se puede escribir software bien centrado en el paradigma funcional. No obstante, para ello es necesario conocer una colección de técnicas básicas que describen formas canónicas de hacer frente a problemas recurrentes. A lo largo de este capítulo haremos una revisión de las técnicas más relevantes.

#### Abstracción

Las técnicas de abstracción permiten definir especificaciones funcionales con diferentes grados de generalidad y reutilización transversal

#### Encapsulación

En funcional la encapsulación de estado es un mal a veces necesario y en muchas ocasiones puede reemplazarse por encapsulación de comportamiento

#### Inmersión

Las técnicas de inmersión son una conjugación de abstracción, encapsulación y recursividad para obtener control de flujo dirigido por datos

#### Evaluación Parcial

La evaluación parcial permite evaluar una función en varias fases reduciendo paulatinamente el número de variables libres

#### Composición Funcional

La composición funcional permite articular procesos de secuenciamiento de funciones definidas como expresiones analíticas

#### Transformación Monádica

La transformación monádica persigue adaptar funciones para convertirlas en transformaciones puras que soporten procesos de composición

#### Inversión de Control

La inversión de control permite permutar entre las arquitecturas funcionales centradas en los datos y las centradas en las transformaciones

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

#### A. Técnicas de Abstracción

Una de las técnicas de programación funcional más comúnmente utilizadas, en alineamiento con los principios de diseño presentados anteriormente, es la abstracción funcional. Las tareas de abstracción deben entenderse como un proceso de transformación en el que la definición de una función se reexpresa en términos más generales para dar cobertura a un abanico más amplio de escenarios de aplicación. Podemos distinguir tres dimensiones de abstracción.

##### **add**

*Suma convencional de dos números pasados como parámetros*

```
add(x, y)
```

##### **Abstracción en Anchura**



##### **addAll**

*Suma todos los parámetros de la función independientemente de la aridad*

```
addAll(x, y, z, ...)
```

##### **Abstracción en Alcance**



##### **reduceFrom**

*Combina mediante la función c todos los parámetros a partir de uno dado en posición p*

```
reduceFrom(p, c)(x, y, ...)
```

##### **Abstracción en Comportamiento**



##### **addFrom**

*Suma todos los parámetros de la función a partir de aquél que ocupa una posición p*

```
add(p)(x, y, z, ...)
```

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

## II. Técnicas de Programación Funcional

### B. Técnicas de Encapsulación

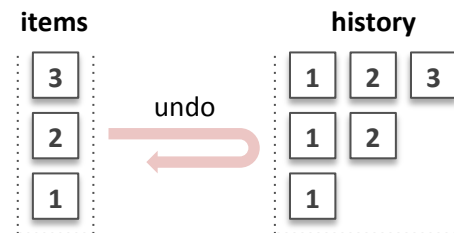
Mediante el uso de clausuras y la retención de variables es posible capturar, durante la fase de diseño o ejecución, cierta información que resulta de utilidad para adaptar el comportamiento de la función a lo largo del tiempo. Es posible distinguir entre encapsulación de estado y de comportamiento.

#### Encapsulación de Estado. Pila Undo

```
function Stack () {  
  var items    = [];  
  var history = [];  
  return {  
    push: function (e) {  
      history.push([].concat(items));  
      items.push(e);  
    },  
    pop: function () {  
      history.push([].concat(items));  
      return items.pop();  
    },  
    undo: function () {  
      if (history.length > 0)  
        items = history.pop();  
    }  
  };  
}
```

#### Retención de Estado

*Las variables que capturan el estado quedan retenidas dentro de la clausura lo que permite articular modelos de programación funcional que evolucionan con el tiempo*



#### Dependencia de Estado

*Ojo! Las funciones que dependen del estado son en realidad una mala práctica de programación de acuerdo a los principios de diseño del paradigma*

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

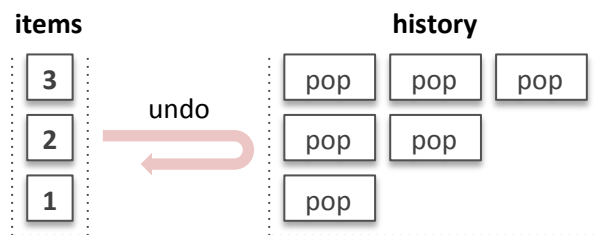
#### B. Técnicas de Encapsulación

Mediante el uso de clausuras y la retención de variables es posible capturar, durante la fase de diseño o ejecución, cierta información que resulta de utilidad para adaptar el comportamiento de la función a lo largo del tiempo. Es posible distinguir entre encapsulación de estado y de comportamiento.

##### Encapsulación de Comportamiento. Pila Undo

###### Retención de Comportamiento

*En este caso la variable de histórico captura la colección de operaciones inversas que permiten deshacer las transacciones según han ido sucediendo*



###### Operaciones Inversas

*Dentro de cada operación, se registran en el histórico las operaciones inversas para luego invocarlas desde la operación deshacer*

```
function Stack () {
  var items  = [];
  var history = [];
  return {
    push: function push (e) {
      history.push(function() {items.pop()});
      items.push(e);
    },
    pop: function pop () {
      var e = items.pop();
      history.push(function() {items.push(e)});
      return e;
    },
    undo: function undo() {
      if (history.length > 0)
        history.pop()(),
    }
  };
}
```

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

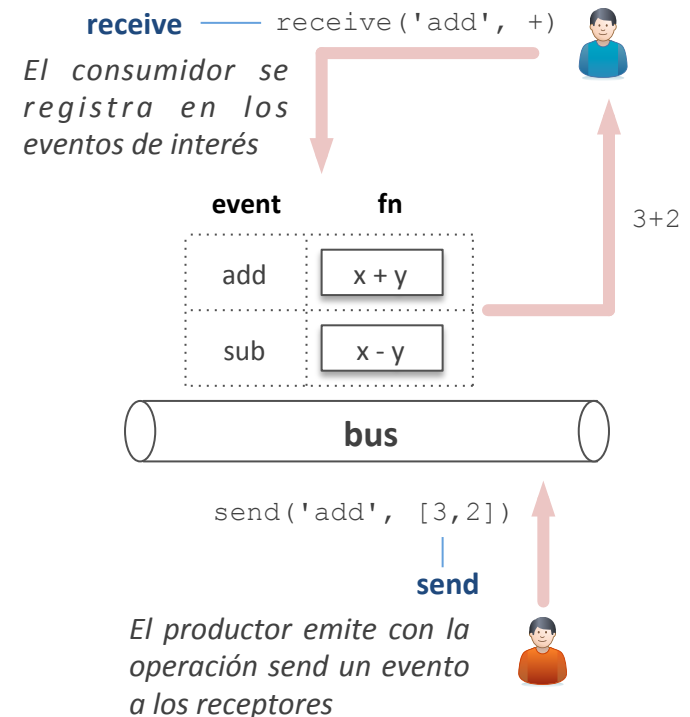
### II. Técnicas de Programación Funcional

#### B. Técnicas de Encapsulación

Mediante el uso de clausuras y la retención de variables es posible capturar, durante la fase de diseño o ejecución, cierta información que resulta de utilidad para adaptar el comportamiento de la función a lo largo del tiempo. Es posible distinguir entre encapsulación de estado y de comportamiento.

##### Encapsulación de Comportamiento. Bus

```
function Bus () {  
  var fns = {};  
  return {  
    receive: function (e, fn) {  
      fns[e] = fn;  
    },  
    send: function (e, ctx) {  
      return fns[e].apply(null, ctx);  
    }  
  };  
}  
  
var bus = Bus();  
bus.receive('add', function (x,y) {return x+y;});  
bus.receive('sub', function (x,y) {return x-y;});  
bus.send('add', [3,2]);  
bus.send('sub', [7,3]);
```



# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

#### C. Diseño por Inmersión

Dado que la programación funcional no soporta los esquemas iterativos propios de la programación imperativa, es preciso orientar los cálculos a la recurrencia apoyándose para ello en parámetros auxiliares. Las técnicas de inmersión conjugan encapsulación y abstracción con el uso de parámetros auxiliares. Podemos distinguir diferentes tipos de inmersión.

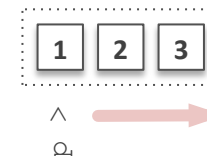
##### I. Inmersión por Recorrido

```
function find (v, e) {  
  var aux = function (v, e, p) {  
    return (p > v.length) ? -1 :  
           (v[p] === e)    ? p :  
           aux(v, e, p + 1);  
  };  
  return aux(v, e, 0);  
}
```

*La inmersión de recorrido extiende la función introduciendo un nuevo parámetro cuyo propósito es llevar la cuenta del punto hasta el que se ha avanzado en el vector.*

```
aux([1,2,3], 0) =  
1 + aux([1,2,3], 1) =  
1 + 2 + aux ([1,2,3], 2) =  
1 + 2 + 3 = 6
```

*La técnica se apoya es una abstracción que queda sumergida dentro de la más específica por encapsulación*



```
function addAll (v) {  
  return (function aux(v, p) {  
    if (p === v.length-1) return v[p];  
    else return v[p] + aux (v, p+1);  
  }) (v, 0);  
}
```

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

## II. Técnicas de Programación Funcional

### C. Diseño por Inmersión

Dado que la programación funcional no soporta los esquemas iterativos propios de la programación imperativa, es preciso orientar los cálculos a la recurrencia apoyándose para ello en parámetros auxiliares. Las técnicas de inmersión conjugan encapsulación y abstracción con el uso de parámetros auxiliares. Podemos distinguir diferentes tipos de inmersión.

#### II. Inmersión por Acumulación

```
function even (v) {  
  return (function aux(v, p, ac){  
    if (p === v.length) return ac;  
    else {  
      if (v[p] % 2 === 0) ac.push(v[p]);  
      return aux(v, p+1, ac);  
    }  
  })(v, 0, []);  
}
```

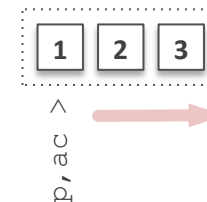
##### Reducción

*Se utiliza la técnica de acumulación para consolidar un vector formado por el subconjunto de los elementos pares del vector*

```
aux([1,2,3], 0, 0) =  
  aux([1,2,3], 1, 1) =  
    aux([1,2,3], 2, 3) =  
      aux([1,2,3], 3, 6) =  
        6
```

##### Tail Recursion

*La acumulación se aplica aquí para obtener recursividad final buscando la eficiencia en memoria*



```
function addAll (v) {  
  return (function aux(v, p, ac) {  
    if (p === v.length) return ac;  
    else return aux (v, p+1, v[p]+ac);  
  })(v, 0, 0);  
}
```



# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

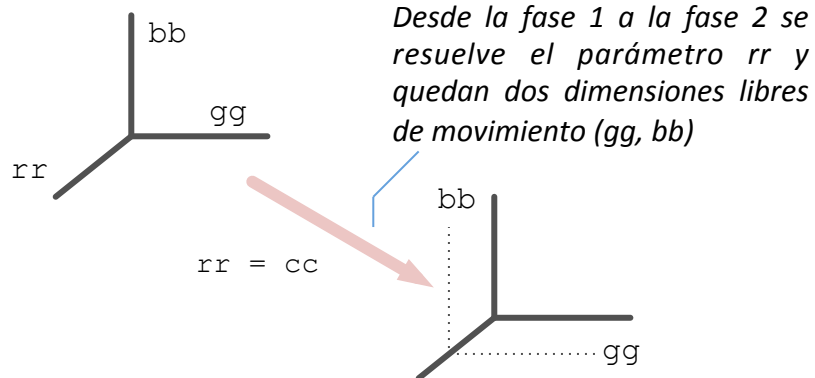
## II. Técnicas de Programación Funcional

### D. Evaluación Parcial

El uso de clausuras permite transformar una función para que pueda evaluarse de manera parcial resolviendo sólo algunos de sus parámetros. El resultado es una nueva función con menos grados de libertad donde los parámetros ya evaluados se fijan a un valor concreto y el resto quedan como variables libres.

#### I. Dimensiones Contractuales

A diferencia de lo que ocurre en otros paradigmas, el diseño funcional de abstracciones tiene dos dimensiones de definición, la dimensión espacial y la dimensión temporal.



*En la dimensión temporal cada fase conduce a una etapa de resolución parcial de parámetros*

*La dimensión espacial vincula convencionalmente valores actuales a parámetros*

```
function color(rr) {  
  return function (gg) {  
    return function (bb) {  
      return [rr, gg, bb] ;  
    };  
  };  
}
```

#### II. Reducción Dimensional

Cada fase de aplicación temporal de parámetros conduce a una reducción dimensional en el espacio del problema.

```
color('cc') ('a3') ('45')
```

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

#### D. Evaluación Parcial

El uso de la evaluación parcial permite que el programador vaya resolviendo de forma faseada la evaluación de una función a medida que va disponiendo de los argumentos actuales necesarios. En estos casos, el diseño de la función debe cuidarse para garantizar que el orden en el que se demandan los parámetros a lo largo del tiempo corresponde con el esperado en el contexto previsto de uso.

#### Ejemplo. Evaluación por Fases

```
function schema (scheme) {  
  var uri = { scheme : scheme };  
  return function (host) {  
    uri.host = host;  
    return function (port) {  
      uri.port = port;  
      return function (path) {  
        uri.path = path;  
        return function () {  
          return uri;  
        };  
      };  
    };  
  };  
}
```

*Se articula un proceso de construcción por fases que permite inyectar las partes de una Uri. Este esquema recuerda al patrón Builder de OOP pero usando únicamente funciones*

```
var host = schema('http');  
var port = host('foo.com');  
var path = port(80);  
var uri = path('index.html');  
uri();
```

*Aunque la aplicación de este esquema induce un estricto orden en la resolución paramétrica permite resolver por fases el proceso constructivo a medida que se dispone de los datos*

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

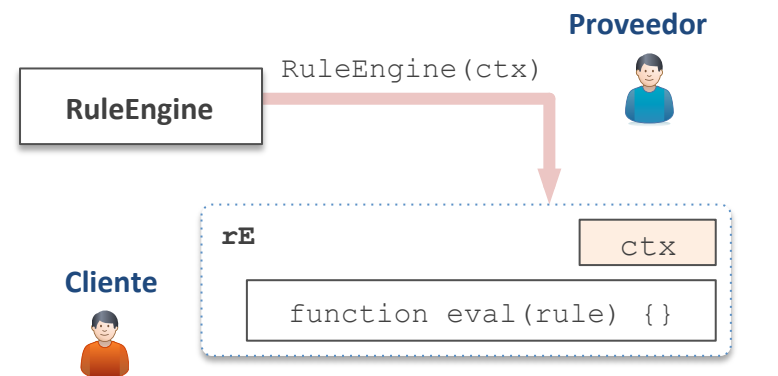
#### D. Evaluación Parcial

Otro escenario prototípico del uso de la evaluación parcial se da en la comunicación entre funciones proveedoras y clientes. El proveedor resuelve parcialmente una función reduciendo así el número de dimensiones y la entrega en respuesta a la demanda de un cliente que resolverá el resto de dimensiones con argumentos actuales en sucesivas invocaciones. En este caso, el diseño funcional debe fasearse teniendo en cuenta la intervención primera del proveedor y posterior del cliente.

##### Transferencia Proveedor - Consumidor

```
function RuleEngine (ctx) {  
  return function eval(rule) {  
    return function () {  
      var args = [].slice.call(arguments);  
      if (rule.trigger.apply(ctx, args))  
        return rule.action.apply(ctx, args);  
    };  
  };  
}  
var rE = RuleEngine({age: 18, login: false});
```

*El proveedor resuelve parcialmente una función eval y la entrega como resultado a la función cliente*



*La función obtenida del proveedor permite al cliente evaluar reglas que se apoyan en las variables retenidas*

```
rE ({ trigger: function (age) { return age > this.age; },  
      action: function () { return this.login = true; } })(19);  
rE ({ trigger: function () { return this.login; },  
      action: function () { return 'Bienvenido!'; } })();
```

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

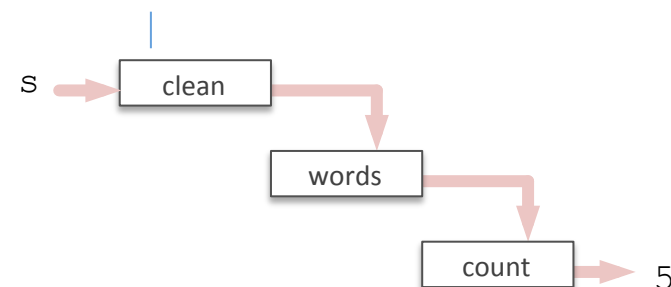
#### E. Composición Funcional

Dado que en programación funcional las abstracciones son meras transformaciones que no pueden articular un algoritmo secuencial, es frecuente modelar el secuenciamiento en base a la composición. De esta manera, el valor de retorno de una función sirve de entrada para la función subsiguiente.

```
function clean (s){ return s.trim(); }  
function words (s){ return s.split(' '); }  
function count (s){ return s.length; }  
  
count(  
  words(  
    clean('La FP en JS Mola!!!')  
  )  
);
```

*La cascada de composición funcional se recorre en sentido inverso al natural. Los datos comienzan por la función más interna (clean) y atraviesan la cadena de composición ascendentemente hasta la cima (count). En el capítulo siguiente estudiaremos patrones que conducen a una lectura descendente más natural*

*La organización compositiva de funciones conduce a arquitecturas centradas en las transformaciones que los datos atraviesan en cascada*



# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

#### E. Composición Funcional

En ocasiones la forma en la que se definen las abstracciones funcionales no es compatible con los procesos de composición. Para poder articular adecuadamente dichos procesos debemos tener en cuenta dos reglas de transformación funcional que puede ser necesario aplicar para conseguir funciones compositivas.

##### I. Dominio Simple

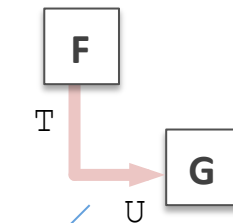
*Dado que una función  $f$  devuelve un único valor de retorno, el número de parámetros que acepta una función  $g$  cuando se compone con  $f$  debe ser exactamente uno. Para mitigar este problema es posible encapsular los resultados de  $f$  en una estructura de datos y/o resolver todos los parámetros de  $g$  menos uno por medio de evaluación parcial*

```
count(
  words(
    clean(s)
  )
);
```

[String] -> Number  
String -> [String]  
String -> String

*El tipo de salida  $T$  debe ser compatible con el tipo de entrada  $U$ . En términos OOP diríamos que  $T$  debe ser subtipo de  $U$*

```
neg(
  square(
    {stack:[3]}
  )
);
```



```
function square(r) {
  var e = r.stack.pop();
  return {
    value: e*e, stack: r.stack
  };
}
function neg(r) {
  r.stack.push(-r.value);
  return r;
}
```

##### II. Compatibilidad Rango-Dominio

*El tipo del parámetro de entrada de una función debe ser compatible con el tipo del valor de retorno devuelto por la función anterior para que la composición pueda articularse con éxito*

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

#### F. Transformación Monádica

El diseño de funciones debe hacerse garantizando la ausencia de efectos colaterales de manera que cada función sólo dependa de sus parámetros de entrada. Sin embargo, en el mundo real es frecuente requerir que una función realice operaciones que dependan del entorno (dependencias de estado, operaciones de E/S, etc.). Las técnicas de transformación monádica garantizan la ausencia de efectos colaterales a la vez que conservan la capacidad compositiva.

#### Ejemplo. Monada Escritor

*Supongamos que tenemos una librería de funciones matemáticas de un solo parámetro que operan con números y devuelven un número. Por motivos de depuración estas funciones emiten mensajes a la consola*

```
function inv (x) { console.log('invertir'); return 1/x; }  
function sqr (x) { console.log('cuadrado'); return x*x; }  
function inc (x) { console.log('incremento'); return x+1; }  
function dec (x) { console.log('decremento'); return x-1; }
```

```
function inv (x) { return { value: 1/x, log: ['invertir'] }; }  
function sqr (x) { return { value: x*x, log: ['cuadrado'] }; }  
function inc (x) { return { value: x+1, log: ['incrementar'] }; }  
function dec (x) { return { value: x-1, log: ['decrementar'] }; }
```

*Dado que la traza por consola se considera un efecto colateral indeseable se transforman las funciones en puras parametrizando la traza como valor anexo de salida*

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

## II. Técnicas de Programación Funcional

### F. Transformación Monádica

El diseño de funciones debe hacerse garantizando la ausencia de efectos colaterales de manera que cada función sólo dependa de sus parámetros de entrada. Sin embargo, en el mundo real es frecuente requerir que una función realice operaciones que dependan del entorno (dependencias de estado, operaciones de E/S, etc.). Las técnicas de transformación monádica garantizan la ausencia de efectos colaterales a la vez que conservan la capacidad compositiva.

#### Ejemplo. Monada Escritor

*Ahora las funciones son puras pero han perdido su capacidad compositiva puesto que incumplen la regla de compatibilidad Rango-Dominio. Es necesario, en primer lugar, crear una función unit que eleve valores unitarios desde el tipo simple (Number) al tipo monádico (valor + log)*

```
function unit (value) {  
  return {  
    value: value,  
    log  : []  
  };  
}
```

```
function bind (m, fn) {  
  var r = fn(m.value);  
  return {  
    value : r.value,  
    log   : m.log.concat(r.log)  
  };  
  return this;  
}
```

*Ahora podemos crear una función bind que permita componer nuestras funciones anteriores sin necesidad de alterarlas. Esta función recibe un valor monádico (entrada + log acumulado) y una de nuestras funciones. La función bind primero desenvuelve la monada, después aplica la función pasada como parámetro y anexa la traza de la misma al log acumulado. Finalmente devuelve la estructura monádica con los resultados obtenidos*

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

## II. Técnicas de Programación Funcional

### F. Transformación Monádica

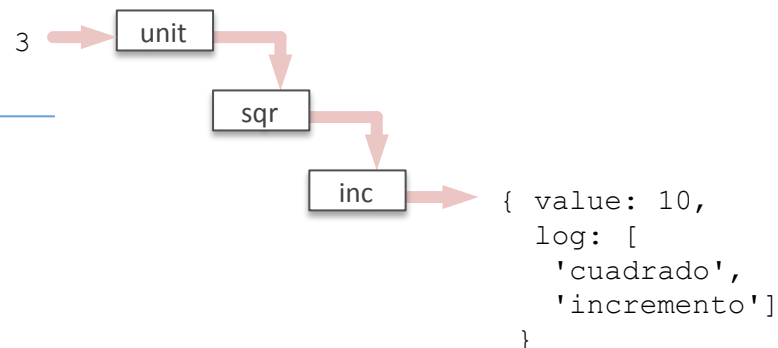
El diseño de funciones debe hacerse garantizando la ausencia de efectos colaterales de manera que cada función sólo dependa de sus parámetros de entrada. Sin embargo, en el mundo real es frecuente requerir que una función realice operaciones que dependan del entorno (dependencias de estado, operaciones de E/S, etc.). Las técnicas de transformación monádica garantizan la ausencia de efectos colaterales a la vez que conservan la capacidad compositiva.

#### Ejemplo. Monada Escritor

Con las funciones de transformación monádica *unit* & *bind* hemos creado una solución para garantizar la composición funcional eliminando los efectos colaterales de traza por pantalla

```
bind(  
  bind(  
    unit(3), sqr  
  ), neg  
);
```

El valor primitivo 3, se transforma primero al tipo monádico y luego se compone con la función *sqr* e *inc* por medio de la asistencia que proporciona la función *bind*





# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

#### F. Transformación Monádica

El diseño de funciones debe hacerse garantizando la ausencia de efectos colaterales de manera que cada función sólo dependa de sus parámetros de entrada. Sin embargo, en el mundo real es frecuente requerir que una función realice operaciones que dependan del entorno (dependencias de estado, operaciones de E/S, etc.). Las técnicas de transformación monádica garantizan la ausencia de efectos colaterales a la vez que conservan la capacidad compositiva.

#### Ejemplo. Monada Escritor

```
function Writer (value) {  
  this.value = value;  
  this.log   = [];  
}  
Writer.prototype.bind = function (fn) {  
  var m = fn(this.value);  
  var result = new Writer(m.value);  
  result.log = this.log.concat(m.log);  
  return result;  
};
```

*Existen muchos tipos de efectos colaterales que pueden evitarse por medio de la aplicación de técnicas de composición monádica. Dado que cada tipo de efecto colateral requiere una lógica unit y bind específica es buena idea expresar estas técnicas como objetos donde unit se codifique como constructor y bind como método miembro del prototipo*

*Aunque desde un punto de vista pragmático esta operativa resulta cómoda y natural debemos ser conscientes de que se apoya en encapsulación de estado*

```
var r = new Writer(3)  
  .bind(sqr)  
  .bind(inc);
```

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

#### G. Inversión de Control

La programación funcional permite articular dos tipos de arquitecturas complementarias. Las arquitecturas centradas en los datos fijan una colección de datos de entrada y hacen atravesar en torno a ellos una colección de transformaciones funcionales. Ortogonalmente, las arquitecturas centradas en la transformación establecen una cadena compositiva de funciones que es atravesada por diversas colecciones de datos. Las técnicas de inversión de control son una pasarela para convertir un esquema arquitectónico en el contrario y viceversa.

#### Arquitecturas Centradas en los Datos

```
function data (value) {  
  return {  
    value: value,  
    do: function (fn) {  
      this.value = fn(this.value);  
      return this;  
    }  
  };  
}
```

*Las arquitecturas centradas en los datos fijan un conjunto de datos y permiten realizar transformaciones pasadas como parámetros en orden superior*



```
data('La FP en JS Mola!!!')  
  .do(clean)  
  .do(words)  
  .do(count);
```

# Taller de Programación Funcional en JavaScript

## Técnicas de Programación Funcional

### II. Técnicas de Programación Funcional

#### G. Inversión de Control

La programación funcional permite articular dos tipos de arquitecturas complementarias. Las arquitecturas centradas en los datos fijan una colección de datos de entrada y hacen atravesar en torno a ellos una colección de transformaciones funcionales. Ortogonalmente, las arquitecturas centradas en la transformación establecen una cadena compositiva de funciones que es atravesada por diversas colecciones de datos. Las técnicas de inversión de control son una pasarela para convertir un esquema arquitectónico en el contrario y viceversa.

#### Arquitecturas Centradas en la Transformación

```
function clean (s) { return s.trim();      }  
function words (s) { return s.split(' '); }  
function count (s) { return s.length;     }  
  
count(  
  words(  
    clean('La FP en JS Mola!!!')  
  )  
);
```

*En el siguiente capítulo estudiaremos patrones dedicados a establecer transformaciones para convertir arquitecturas centradas en los datos a arquitecturas centradas en transformación y viceversa*

*Las arquitecturas centradas en la transformación establecen una cadena de composición de transformaciones funcionales por las que atraviesan distintas colecciones de datos*



# *Taller de Taller de Programación Funcional en JavaScript*

Javier Vélez Reyes

@javiervelezreye  
Javier.velez.reyes@gmail.com

Enero 2016

