Functional JavaScript

- JavaScript soporta tanto paradigma OOP como el funcional
- Programación funcional
  - inmutabilidad
  - funciones de primera clase
  - funciones de orden superior
  - funciones anónimas (lambdas)
  - clausuras, currying, monads, combinators

Filosóficamente

- En lugar de uns secuencia de instrucciones (statements) que le dicen al computador exactamente que hacer ...
- Se conceptualiza el programa como una función matemática que transforma un input en un output
  - el programa usa composición de funciones menores
- Al ser menos concreto el código resulta mas legible y mas susceptible de correr de diversa maneras (paralelización)

Reducir Complejidad ...

- La programación funcional es una manera de pensar acerca de cómo construir programas para minimizar las complejidades inherentes a la creación de software.

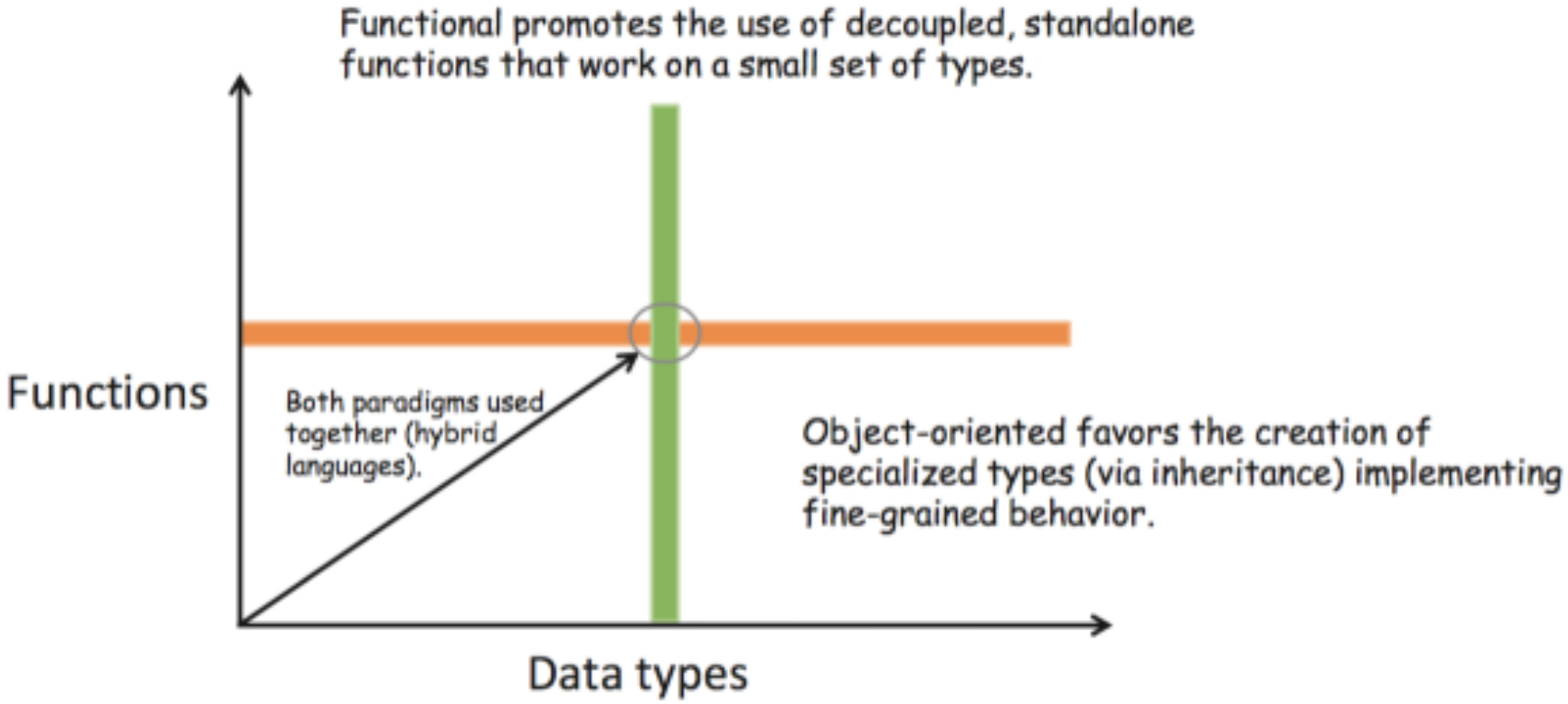
- Una forma de reducir la complejidad es reducir o eliminar (idealmente) la huella del cambio de estado que ocurre en los programas.

Un ejemplo simple

```
var books = [ "Gone with the Wind", "War and Peace" ];
// imperative way
for (var i = 0; i < books.length; i++)
  { console.log(books[i]); }

// functional way
books.forEach(book => console.log(book));
```

Lenguajes y Paradigmas



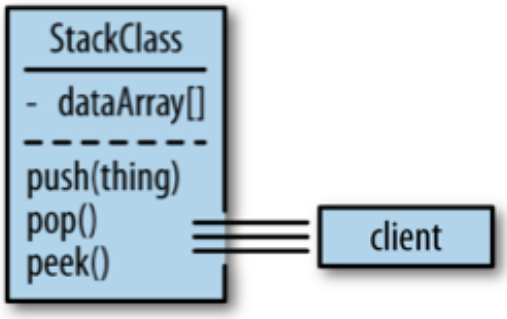
Lenguajes para FP

- Históricos
  - Haskell, Lisp, Erlang
- Nuevos
  - Clojure
  - F#
  - Scala
  - Mathematica
  - JavaScript

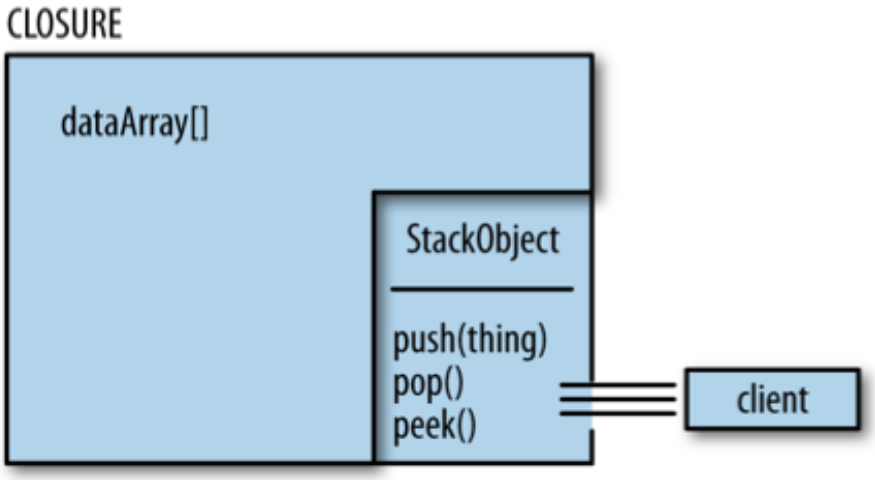
	Functional	Object-oriented
Unit of composition	Functions	Objects (classes)
Programming style	Declarative	Imperative
Data and behavior	Loosely coupled into pure, stand-alone functions	Tightly coupled in classes with methods
State management	Treats objects as immutable values	Favors mutation of objects via instance methods
Control flow	Functions and recursion	Loops and conditionals
Thread safety	Enables concurrent programming	Difficult to achieve
Encapsulation	Not needed because everything is immutable	Needed to protect data integrity

Encapsulación vía Clojures

- en OOP se usan los límites del objeto



- en FP se usan las clausuras



Closure

- funciones internas tienen acceso a variables declaradas en función que la engloba
- cuando una función interna vive mas allá de la externa que la engloba sigue teniendo visibilidad a las variables de la externa

```
function foo() {
  var a = 10;
  function bar() {
    a *= 2;
    return a;
   }
   return bar;
}

var baz = foo(); // baz is now a reference to function bar.
baz(); // returns 20.
baz(); // returns 40.
baz(); // returns 80.
var blat = foo(); // blat is another reference to bar.
blat(); // returns 20, because a new copy of a is being used.
```

Funciones de Primera Clase

- una función debe funcionar igual que por ejemplo,un número
  - debe poder ser guardada en una variable
  - debe poder ser guardada en un elemento de array
  - debe poder ser un atributo de un objeto
  - debe poder ser pasado como parámetro
  - debe poder ser devuelto como resultado

Funciones de Orden Superior en ES6

- forEach
  - array.forEach(callback)
- filter
  - array.filter(callback)
- map
  - array.map(callback)
- reduce
  - array.reduce(callback, initialValue)
- some
  - array.some(callback)
- every
  - array.every(callback)
- find
  - array.find(callback)
- findindex
  - array.findindex(callback)

Ejemplos

Calcular la suma de los cuadrados de todos los elementos de un array arr

```
var arr = [1, 2, 3, 4, 5];

// ES6 solution with arrow functions only
arr.map( x => x*x ).reduce( (x,y) => (x+y) );
> 55
```

Extraer un arreglo con los pares

```
arr.filter( x => x%2 === 0 );
[2, 4]
```

El pasar funciones permite hacer funciones + generales

```
function applyOperation(a, b, opt) {
   return opt(a,b);
}
var multiplier = (a, b) => a * b;
var adder = (a, b) => a + b;

applyOperation(2, 3, multiplier); // -> 6
applyOperation(2, 3, adder);      // -> 5


const repeat = (num, fn) =>
                 (num > 0) ? (repeat(num - 1, fn), fn(num)): undefined

repeat(3, function (n) {
  console.log(`Hello ${n}`)
})
  //=>
    'Hello 1'
    'Hello 2'
    'Hello 3'
    undefined
```

La librería Lodash

- Librería orientada a dar mayor soporte del paradigma funcional a JS https://lodash.com/
- Sucesor de underscore (http://underscorejs.org/)
- Mantiene el uso de objeto global "_" (underscore)
- Incluye muchas otras cosas que facilitan la programación con arrays, collections, etc

Ejemplos

```
var numbers = [1, 2, 3, 4, 5];
console.log(_.drop(numbers, 2));   // [3, 4, 5]


var basket = [
   { name: "Cable Lock", quantity: 12 },
   { name: "U Lock", quantity: 9 },
   { name: "Tail Light", quantity: 11 }
   ];

if (!_.every(basket, item => item.quantity > 0)) {
 alert("You must order at least 1 piece of each item");
}

if (_.some(basket, item => item.quantity == 0)) {
 alert("You must order at least 1 piece of each item");
}
```

Chaining

```
var safetyProducts = _.filter(products, p => p.category === "Safety");
var quantities = _.map(safetyProducts, p => p.quantity);
var bigQuantites = _.filter(quantities, q => q > 10);
```

```
var bigQuantites = _.chain(products)
    .filter(p => p.category === "Safety")
    .map(p => p.quantity)
    .filter(q => q > 10)
    .value();
```

ES6 hace innecesarias a varias de estas

- _map (map)
- _reduce (reduce)
- _filter (filter)
- _head, _tail   ([head, ...tail])
- _rest
- _spread

Currying

- Técnica que convierte una función multivariable en una secuencia de funciones unarias
- Normalmente para una función (a, b, c) =>{ ... }
  - f(a) es realmente f(a, undefined, undefined)
- Al currificar ...
  - f(a) devuelve una función (b, c) => { }
  - f(a, b) devuelve una función c => { }

```
const greet = (greeting, name) => {
                console.log(greeting + ", " + name);
               };
greet("Hello", "Heidi"); //"Hello, Heidi"


const greetCurried = (greeting) => {
  return (name) => {
    console.log(greeting + ", " + name);
  };
};

var greetHello = greetCurried("Hello");
greetHello("Heidi"); //"Hello, Heidi"
greetHello("Eddie"); //"Hello, Eddie"


greetCurried("Hi there")("Howard"); //"Hi there, Howard"
```

Un ejemplo Web

```
function getItem(kind, id) {
   return fetch(`http://api.example.com/${kind}/${id}`);
}

const bookIds = [1, 2, 3, 4, 5];
const productIds = [10, 11, 12, 13, 14];

bookIds.map(id => getItem("book", id));
productIds.map(id => getItem("product", id));

function getItemCurried(kind) {
   return function(id) {
      return fetch(`http://api.example.com/${kind}/${id}`);
   }
}

bookIds.map(getItemCurried("book"));
productIds.map(getItemCurried("product"));
```

Curry lodash vs ES6

```
function add(a, b) {
  return a + b;
}
var curriedAdd = _.curry(add);
var add2 = curriedAdd(2);
add2(1);
// 3

// becomes

const add = a => b => a + b;
const add2 = add(2);
add2(1);
// 3
```

Composición

```
const compose = (f,g) => (x) => f(g(x));

const f = x => x + 1;
const g = x => x * 2;

const h = compose(f, g);

> h(4) --> 9
```

Mas general

```
function compose(func1, func2) {
  return function() {
    return func1(func2.apply(null, arguments));
    };
}
```

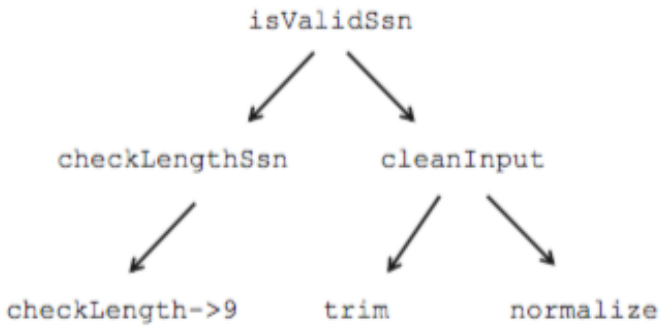
Primer argumento de apply especifica el objeto sobre el cual aplicar la función

Validación de Input

```
const trim = (str) => str.replace(/^\s*|\s*$/g, '');
const normalize = (str) => str.replace(/\-/g, '');
const validLength = (param, str) => str.length === param;
const checkLengthSsn = (str) => validLength(9, str);

const cleanInput = compose(normalize, trim);
const isValidSsn = compose(checkLengthSsn, cleanInput);

cleanInput(' 444-44-4444 ');  //-> '444444444'
isValidSsn(' 444-44-4444 ');  //-> true
```



Pipes

```
const pipe = functions => data => {
  return functions.reduce(
    (value, func) => func(value),
    data
  );
};

const pipeline = pipe( [x => x * 2, x => x / 3,  x => x > 5,  b => !b] );

pipeline(5);
// true
```

(5) → x => x * 2

(10) → x => x / 3

(3) → x => x > 5

(false) → b => !b

Combinators

- Funciones de orden superior usadas para orquestar el flujo de control
- Tienen nombres
- combinator de composición es un ejemplo
- se suele llamar B (bluebird)

```
const compose = (a, b) => (c) => a(b(c))

const addOne = (number) => number + 1;
const doubleOf = (number) => number * 2;

const doubleOfAddOne = compose(doubleOf, addOne);
```

Otros Combinators

- identity combinator
- K combinator
- ALT combinator
- S combinator
- Y combinator

K Combinator (kestrel)

- Para cualquier valor devuelve una funcion constante que devuelve ese valor
  - (K x) y -> x
  - (K x y) -> x
- Sirve para lidiar con funciones que retornan void

```
K(10, (it) => console.log(it)); //10
```

Ejemplo de K, I

Función abuild, toma un número y una función y devuelve un array del largo del número con la función aplicada a los elementos

```
const abuild = (n, f)=>[...new Array(n).keys()].map(f)
abuild (4, x=>x*x)
=> [ 0, 1, 4, 9 ]

const I = x => x
I(8)
=> 8

abuild(5, I)
=> [ 0, 1, 2, 3, 4 ]

const K = x => y => x
K(6) (1000)
=> 6

abuild(5, K('ja'))
=> [ 'ja', 'ja', 'ja', 'ja', 'ja' ]
```

S combinator

- también llamado secuencia
- toma dos o mas funciones como parámetros y retorna una función que las corre a todas ellas sobre un mismo valor
- puede llevarse a cabo una secuencia de operaciones relacionadas

```
const S = f => g => x => f(x)(g(x))
const f = (x) => (y) => x + y
const g = (x) => x * 3
S (f) (g) (10)
=> 40
```

I a partir de S y K

S K K x = K x K x = x = I x

Se puede demostrar que se puede construir cualquier expresión lambda solo con S y K

Lenguaje de expresiones lambda es equivalente al de las máquinas de Turing (computacionalmente completo)

Listas y recursividad

```
const [first, ...rest] = [];
first
  //=> undefined
rest
  //=> []:

const [first, ...rest] = ["foo"];
first
  //=> "foo"
rest
  //=> []

const [first, ...rest] = ["foo", "bar"];
first
  //=> "foo"
rest
  //=> ["bar"]

const [first, ...rest] = ["foo", "bar", "baz"];
first
  //=> "foo"
rest
  //=> ["bar","baz"]
const isEmpty = ([first, ...rest]) => first === undefined;
```

```
const length = ([first, ...rest]) =>
  first === undefined
    ? 0
    : 1 + length(rest);

const sumSquares = ([first, ...rest]) => first === undefined
                                         ? 0
                                         : first * first + sumSquares(rest);

sumSquares([1, 2, 3, 4, 5])
  //=> 55

const mapWith = (fn, [first, ...rest]) =>
  first === undefined
    ? []
    : [fn(first), ...mapWith(fn, rest)];

mapWith((x) => x * x, [1, 2, 3, 4, 5])
  //=> [1,4,9,16,25]


const flatten = ([first, ...rest]) => {
  if (first === undefined) {
    return [];
  }
  else if (!Array.isArray(first)) {
    return [first, ...flatten(rest)];
  }
  else {
    return [...flatten(first), ...flatten(rest)];
  }
}

flatten(["foo", [3, 4, []]])
  //=> ["foo",3,4]
```

Y Combinator

- Permite expresar funciones recursivas

```
factorial = n => (n === 0)? 1 : n * factorial(n - 1)
[Function factorial]

factorial_gen = f => (n => ((n === 0) ? 1 : n * f(n - 1)))
[Function factorial_gen]

factorial_gen (factorial) (19)
NaN

Y = f => (x => x(x))(x => f(y => x(x)(y)))
[Function Y]

Y(factorial_gen)(19)
121645100408832000
```

Funciones Puras

- Una función pura se adhiere a las siguientes propiedades:
  - Su resultado se calcula solamente a partir de los valores de sus argumentos.
  - No se puede confiar en los cambios externos.
  - No puede cambiar el estado de algo externo a su cuerpo.
  - En el ejemplo siguiente, la función *areaOfCircle* es **impura**

```
PI = 3.14;

function areaOfaCircle(radius) {
  return PI * sqr(radius);
}

areaOfaCircle(3)  //=> 28.26

PI = "magnum";
areaOfaCircle(3)  //=> NaN
```

¿ Que tiene de bueno ?

- Ante mismos parámetros, devuelve siempre el mismo resultado (técnica de optimización conocida como Memorization)
- La computación pura es menos propensa a errores.
- Elimina los efectos colaterales (side effects).
- Es más fácil de probar el código (test).

Inmutabilidad

- Es la cualidad de aquello que no cambia o la falta de cambio de estado explícita.
- La inmutabilidad esta relacionado con la pureza funcional.
- En programación funcional, lo ideal es que todo sea inmutable.
- En JS
  - Los strings no son mutables.
  - Los objetos son mutables.
  - Uso de const para explicitar

Encapsulando Estado

```
const stack = (() => {
  const obj = {
    array: [],
    index: -1,
    push (value) {
      return obj.array[obj.index += 1] = value
    },
    pop () {
      const value = obj.array[obj.index];

      obj.array[obj.index] = undefined;
      if (obj.index >= 0) {
        obj.index -= 1
      }
      return value
    },
    isEmpty () {
      return obj.index < 0
    }
  };

  return obj;
})();
```

```
stack.isEmpty()
  //=> true
stack.push('hello')
  //=> 'Hello'
stack.push('JavaScript')
 //=> 'JavaScript'
stack.isEmpty()
  //=> false
stack.array
['Hello', 'JavaScript']
stack.pop()
 //=> 'JavaScript'
stack.pop()
 //=> 'Hello'
stack.isEmpty()
  //=> true
```

- Se usa mecanismo de clausura
- Esta solución no oculta el estado

Mejor ...

```
const stack2 = (() => {
  let array = [],
      index = -1;

  const obj = {
    push (value) { return array[index += 1] = value },
    pop () {
      const value = array[index];

      array[index] = undefined;
      if (index >= 0) {
        index -= 1
      }
      return value
    },
    isEmpty () { return index < 0 }
  };

  return obj;
})();
```

Mejor aun ...

```
const Stack = () => {
  const array = [];
  let index = -1;

  return {
    push (value) { return array[index += 1] = value },
    pop () {
      const value = array[index];

      array[index] = undefined;
      if (index >= 0) {
        index -= 1
      }
      return value
    },
    isEmpty () { return index < 0 }
  }
}

const stack = Stack();
stack.push("Hello");
stack.push("Good bye");

stack.pop()
  //=> "Good bye"
stack.pop()
//=> "Hello"
```