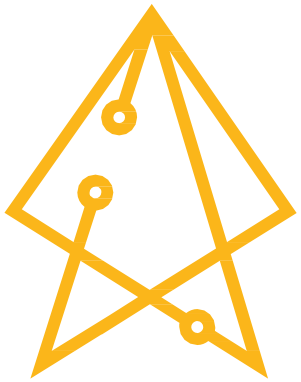


ACADEMIK

Programación funcional en JavaScript

ACADEMIK

Próximos cursos
Junio 2020

Martes 2

Introducción al desarollo web con CSS, HTML 5 Y JavaScript

Aprende los fundamentos de creación de sitios web

- **Duración: 36 horas**
- **Clases: Martes y Jueves - 19:00 - 21:00**

Martes 2

Aplicaciones empresariales con Angular

El framework más popular en entornos empresariales

- **Duración: 36 horas**
- **Clases: Martes y Jueves - 19:00 - 21:00**

Sábado 6

Introducción a la programación con Java 11

El lenguaje más popular desde 1996

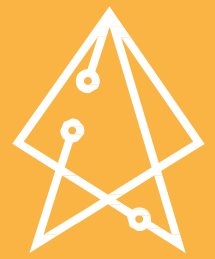
- **Duración: 36 horas**
- **Clases: Sábados - 14:00 - 18:00**

Sábado 6

Aplicaciones web con Java

Aprende los fundamentos de la programación backend

- **Duración: 36 horas**
- **Clases: Sábados - 14:00 - 18:00**



ACADEMIK

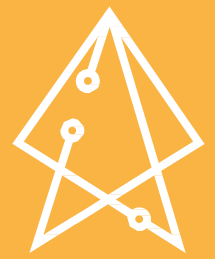
Paradigmas de la programación

Paradigmas de la programación

- Que es un paradigma de programación?
- Es una propuesta tecnológica que es adoptada por una comunidad de programadores cuyo núcleo central es incuestionable en cuanto a que únicamente trata de resolver uno o varios problemas claramente delimitados.
- Es un estilo de programación empleado

Paradigmas de la programación

- Programación imperativa
  - Ordenes
- Programación orientada a objetos
  - Objetos
- Programación funcional
  - Funciones

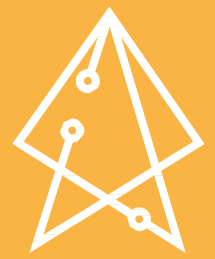


ACADEMIK

JavaScript

Funciones en JavaScript

- Las funciones en JavaScript son tratadas como ciudadanos de primera clase (first-class citizens). Dicho de otra manera, son tratadas como objectos, los cuales tienen las siguientes características:
- Funciones como argumentos de otras funciones
- Funciones retornadas por otra función
- Funciones que pueden ser almacenadas en variables o constantes



ACADEMIK

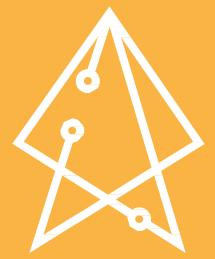
Programación Funcional

Programación Funcional

- En un sentido estricto, la programación funcional significa
  - Programación sin variables (mutables)
  - Sin iteración
  - Sin estructuras de control imperativas
- En un sentido mas amplio:
  - Centrado en funciones
- Las funciones deben ser ciudadanos de primera clase

¿Por qué usar programación funcional?

- Algunas ventajas de la programación funcional:
  - Adecuada para la paralelización
  - Fácil de mantener y de depurar
  - Una manera elegante de escribir código.



ACADEMIK

Características de la Programación Funcional

Centrado en funciones

- En la programación funcional todo gira en torno a funciones

```
//Programacion imperativa, no funcional
let a = 5;
let b = 4;
let c = a + b;
```

```
//Programacion funcional
const suma = (a, b) => {
    return a + b;
}

const c = suma(4, 5);
```

Recursión en lugar de iteración

- Evitar el uso de ciclos (for, while, etc.)

```
const sumIntegers = (i) => {
    let sum = 0
    do {
        sum += i;
        i--;
    } while(i > 0);

    return sum;
}
```

```
const sumIntegersFunctional = i => {
    if(i == 0){
        return i;
    }
    return (i + sumIntegers(i - 1));
}
```

Cómo se observa la recursión

- Supongamos i = 5
- 5 + sumIntegersFunctional(4)
- 5 + (4 + sumIntegersFunctional (3))
- 5 + (4 + (3 + sumIntegersFunctional (2)))
- 5 + (4 + (3 + (2 + (sumIntegersFunctional (1))))
- 5 + (4 + (3 + (2 + (1 + sumIntegersFunctional (0)))))
- 5 + (4 + (3+ (2+ (1 + 0))))
- 15

Evitando efectos secundarios

- En informática, se dice que una función o expresión tiene un efecto secundario (side effect) si modifica una variable fuera de su entorno.
- Al no tener efectos secundarios, la función puede ser paralelizada.
- La programación funcional evita los efectos secundarios y se concentra en funciones puras.
- Una función pura es aquella donde su salida depende solamente de su entrada.

Funciones puras vs no puras

- Incrementando un numero

```
//Efectos secundarios
let acc = 5;
const incrementAcc = (inc) => {
    return acc + inc
}
```

```
//funcion pura
const suma = (a, b) => {
    return a + b;
}
```

Inmutabilidad

- Supongamos el siguiente arreglo de objetos.

```
const obj = [{
        nombre: 'Jorge',
        pais: 'Guatemala',
        edad: 25
    }, {
        nombre: 'Eduardo',
        pais: 'El Salvador',
        edad: 20
    }, {
        nombre: 'Maria',
        pais: 'EEUU',
        edad: 24
    }
];
```

- Mutable

```
for(let i = 0; i < obj.length; i++){
    obj[i].edad += 1;
}
```

- Inmutable

```
let incremento = [];

for(let i = 0; i < obj.length; i++){
    let persona = obj[i];
    incremento.push({
        ...persona,
        edad: persona.edad + 1})
}
```

Funciones de orden superior

- Una función de orden superior es una función que cumple con:
  - Tomar una o mas funciones como entrada
  - Devolver una función como salida
- De acuerdo a lo conversado con anterioridad, esto solamente se puede alcanzar si las funciones son ciudadanos de primera clase (first-class citizens).

Funciones de orden superior en JavaScript

- Supongamos una función para sumar enteros en un rango (a, b)

```
const sumInts = (a, b) => {
    if(a > b) {
        return 0;
    } else {
        return a + sumInts(a + 1, b);
    }
}
```

- Ahora realicemos la creación de una función que reciba un numero y lo eleve al cubo

```
const cube = x => x * x * x;
```

Funciones de orden superior en JavaScript

- Tomando como referencia la suma de enteros en un rango, si queremos realizar la suma de cubos en un rango, podemos crear la función sumCubes:

```
const sumcubes = (a, b) => {
    if(a > b){
        return 0;
    } else {
        return cube(a) + sumcubes(a + 1, b);
    }
}
```

- ¿Y si deseamos, por ejemplo, una función para suma de cuadrados? Podríamos repetir los pasos anteriores y desarrollar dos nuevas funciones.
- Esto no es muy eficiente

Funciones de orden superior en JavaScript

- Y aquí es donde tiene un papel importante las funciones de orden superior. Supongamos la siguiente función:

```
const sum = (f, a, b) => {
    if(a > b){
        return 0;
    } else {
        return (f(a) + sum(f, a + 1, b));
    }
}
```

- Esta función recibe una función como parámetro, la cual será aplicada a la suma de enteros en un rango.
- Ahora podemos pasar cualquier función para los fines deseados.

Funciones de orden superior en JavaScript

- Realicemos la creación de las funciones identidad, cubo y factorial:

```
//identidad
const self = x => x;

//cubo
const cube = x => x * x * x;

//factorial
const fact = x => {
    if(x == 0){
        return 1
    } else {
        return x * fact(x - 1);
    }
}
```

- Ahora realizar la creación de las funciones suma de enteros, suma de cubos y suma de factoriales en un rango se limita a:

```
const sumInts = (a, b) => sum(self, a, b);
const sumCubes = (a, b) => sum(cube, a, b);
const sumFacts = (a, b) => sum(fact, a, b)
```

Map, filter, reduce

- Map, filter y reduce también son funciones de orden superior las cuales son recursivas.
- Son una manera elegante de recorrer colecciones de objetos (arreglos, listas, etc.)

Map

- Aplica una función a una colección de datos.

```
const obj = [{
        nombre: 'Jorge',
        pais: 'Guatemala',
        edad: 25
    }, {
        nombre: 'Eduardo',
        pais: 'El Salvador',
        edad: 20
    }, {
        nombre: 'Maria',
        pais: 'EEUU',
        edad: 24
    }
];
```

- Con iteración

```
let incremento = [];

for(let i = 0; i < obj.length; i++){
    let persona = obj[i];
    incremento.push({
        ...persona,
        edad: persona.edad + 1})
}
```

- Utilizando map

```
const incrementoMap = obj.map(persona => ({
    ...persona,
    edad: persona.edad + 1
}));
```

Filter, Reduce

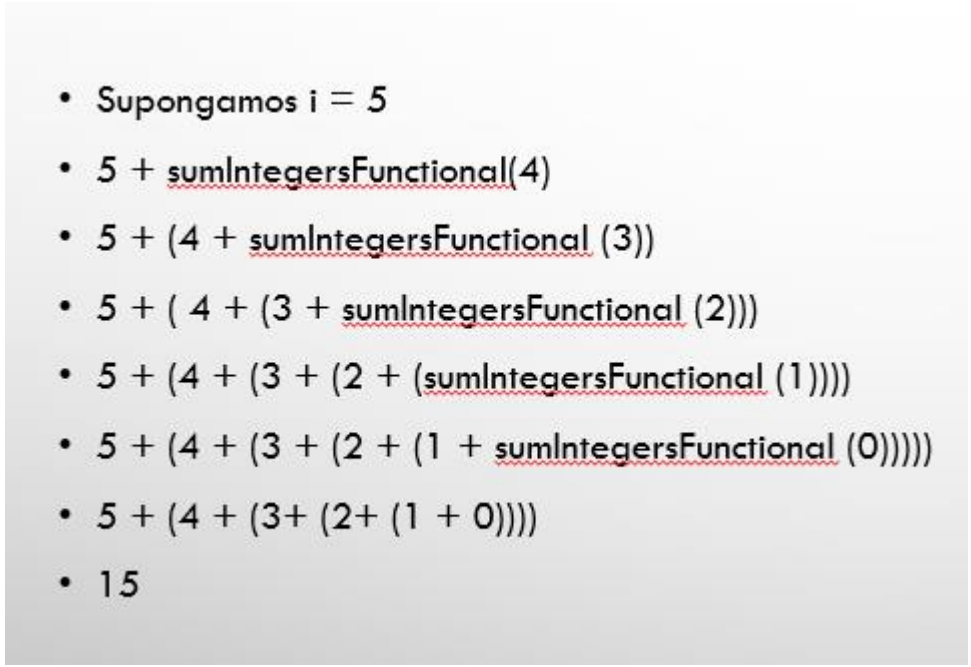
- Y finalmente ejemplos con reduce y filter.

```
//reduce
const suma = obj.reduce((total, persona) => total + persona.edad, 0);


//filter

//personas mayores a 22 años
const mayores = obj.filter(persona => persona.edad > 22);
```

- En el caso de reduce, podemos obtener la suma de las edades de todas las personas en el arreglo
- Filter nos devuelve una nueva estructura donde se cumpla la condición.

Bonus: Recursividad de cola

- Recuerdan este ejemplo?



- El resultado final se puede obtener hasta alcanzar la condición mínima de la función.
- Esto puede representar un problema de desbordamiento de pila (stack overflow).
- La solución para este problema es la recursividad de cola

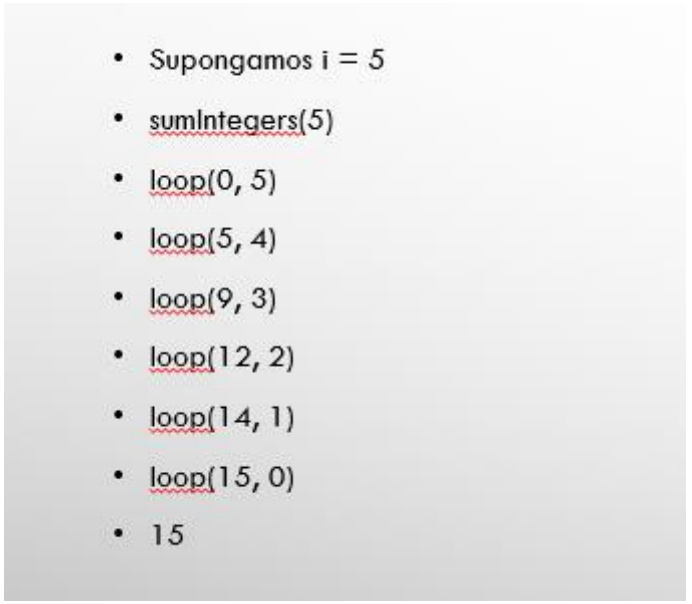
Recursividad de cola

- Si una función se llama a sí misma como su última acción, el marco de la pila de la función se puede reutilizar. Esto se llama recursión de cola.
- Regresando al caso de suma de enteros, podemos modificar la función de la siguiente manera:

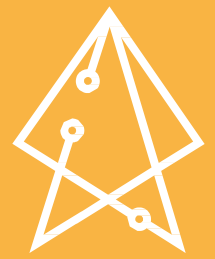
```
const sumIntegersTail = i => {
    const loop = (acc, i) => {
        if(i == 0){
            return acc;
        } else {
            return loop(acc + i, i - 1);
        }
    }
    return loop(0, i);
}
```

Recursividad de cola

- Como se observa la recursividad de cola?



- Esto permite evadir los posibles problemas por desbordamiento de pila



ACADEMIK

Muchas gracias por su atención
¿Preguntas, comentarios?