

Curso de JavaScript avanzado

Estructuras de datos

Estructuras de datos avanzadas


ÍNDICE

- Estructuras de datos en programación
- Estructura en JavaScript: Set
- Estructura en JavaScript: Map
- Colas y pilas
- Listas enlazadas

Estructuras de datos en programación

- Una estructura de datos es una organización concreta de los datos que **permite optimizar su uso**
- Cada estructura de datos es una abstracción útil para ciertas tareas, por lo tanto **es útil conocer su existencia y uso**
- La estructura de datos más común y utilizada en Javascript es el **Array o lista**

Estructura en Javascript: Set

`const A = [1, 2, 3, 2, 3]` ^{new Set(A)}  `[1, 2, 3]`

- Un **Set** es una estructura de datos compuesto por un conjunto de valores únicos, es decir, **no puede tener datos repetidos**
- Es posible crear un **Set** a partir de cualquier objeto de Javascript iterable (Array, DOM collection...etc)
- Los tres métodos principales para su uso son **add()**, **has()** y **delete()**

Estructura en Javascript: Set

```
const setEjemplo = new Set([2, 3, 3, 2]);  
// setEjemplo almacena [2, 3]  
  
setEjemplo.has(2); // -> true  
setEjemplo.has(1); // -> false  
  
setEjemplo.add(2); // -> [2, 3]  
setEjemplo.add(1); // -> [2, 3, 1]  
  
setEjemplo.delete(2); // -> [3, 1]  
  
const newObj = {};  
const otherObj = {};  
  
setEjemplo.add(newObj); // -> [2, 3, 1, {}]  
setEjemplo.add(newObj); // -> [2, 3, 1, {}]  
setEjemplo.add(otherObj); // -> [2, 3, 1, {}, {}]
```

- Al añadir nuevos valores, si estos son objetos, el set **comprueba sus referencias**
- Es posible iterar los sets utilizando **keys()**, **values()** o **entries()**
- Puede crearse un array a partir de un set **gracias al uso del destructuring**

Estructura en Javascript: Map

`const A = { a: 'v1', b: 'v2' }` $\xrightarrow{\text{new Map(A)}}$ `{ a: 'v1', b: 'v2' }`

- Un mapa o Map es una estructura de datos que, al igual que un objeto de JS, **almacena registros como clave -> valor**
- Al igual que el Set es posible crear un Map de **cualquier iterable o colección con clave -> valor**
- Los tres métodos principales para su uso son **get()**, **set()** y **delete()**

Estructura en Javascript: Map

```
const mapEjemplo = new Map({a: 1, b: 4})
// mapEjemplo almacena {a: 1, b: 4}

mapEjemplo.has('a') // -> true
mapEjemplo.has('c') // -> false

mapEjemplo.set('c', 5) // -> {a: 1, b: 4, c: 5}
mapEjemplo.get('c') // -> 5

mapEjemplo.delete('a') // -> {b: 4, c: 5}

const newObj = {}

mapEjemplo.set(newObj, 5) // -> {b: 4, c: 5, ref: 5}
mapEjemplo.delete({}) // -> {b: 4, c: 5, ref: 5}
mapEjemplo.delete(newObj) // -> {b: 4, c: 5}
```

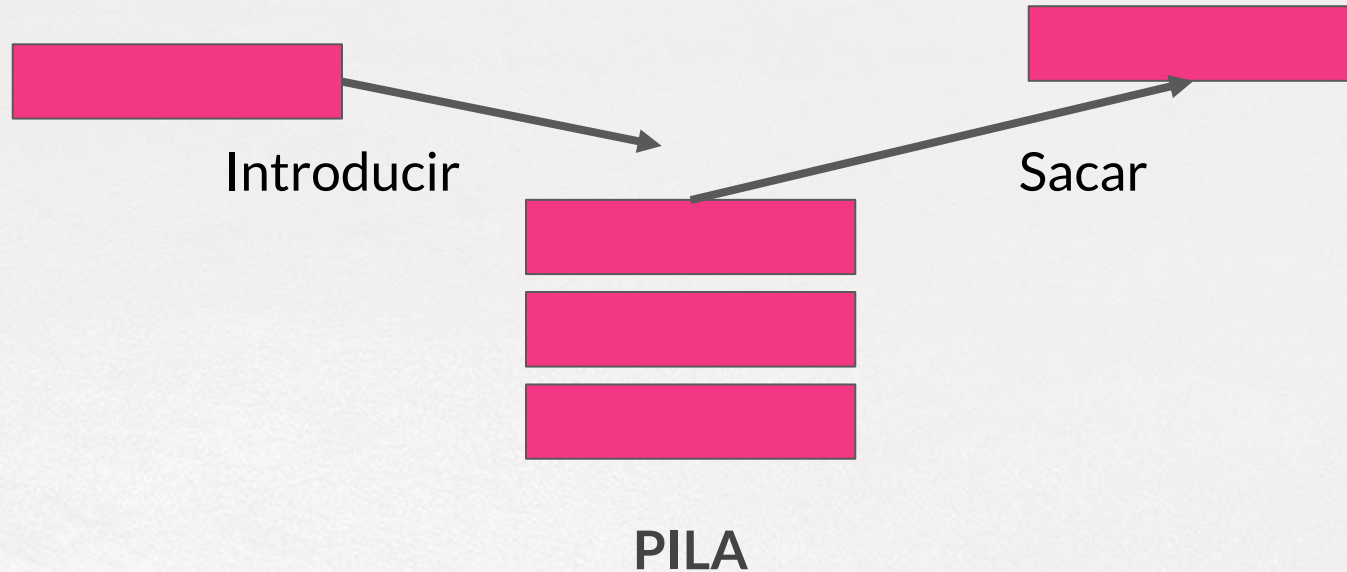
- A diferencia de un objeto sus claves pueden ser objetos, funciones o cualquier tipo primitivo
- También es sencillo saber su tamaño usando la función `size()`
- Igualmente es posible iterar directamente sobre él, ya que es un iterable

Colas y pilas

- Las colas y pilas son estructuras abstractas que almacenan una colección e implementan **un método para añadir y otro para extraer**
- La única diferencia entre ambas es el orden de extracción de **un elemento de la colección**
- Existen variaciones en las colas como las llamadas **colas circulares** o las **colas con prioridad**, donde se altera ligeramente el comportamiento

Colas y pilas

LIFO
(Last **I**n First **O**ut)



Colas y pilas

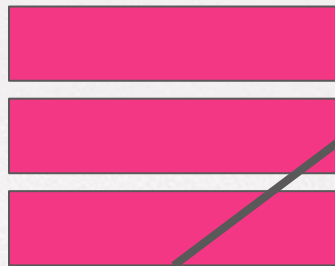
```
class Stack {  
  constructor(in_items) {  
    this.items = in_items || [];  
  }  
  length() {  
    return this.items.length;  
  }  
  stack(el) {  
    // Añade un elemento a items  
    this.items.push(el);  
  }  
  unstack() {  
    // Devuelve el ultimo elemento o undefined  
    return this.length() > 0 ? this.items.pop() : undefined;  
  }  
}
```

Colas y pilas

FIFO
(First In First Out)



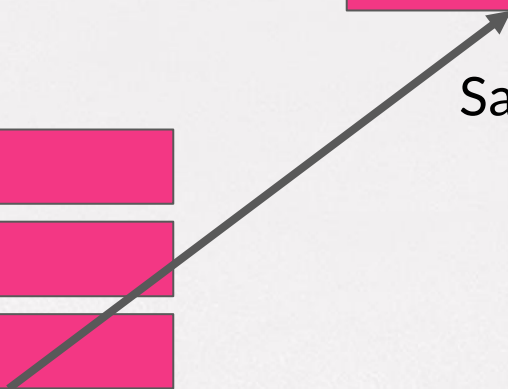
Introducir



COLA



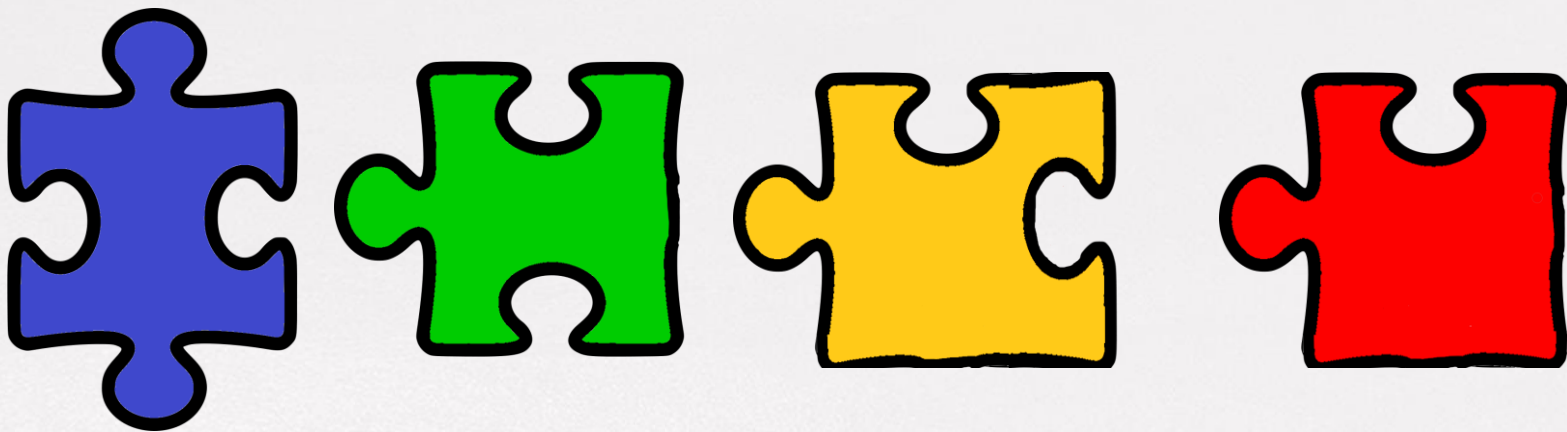
Sacar



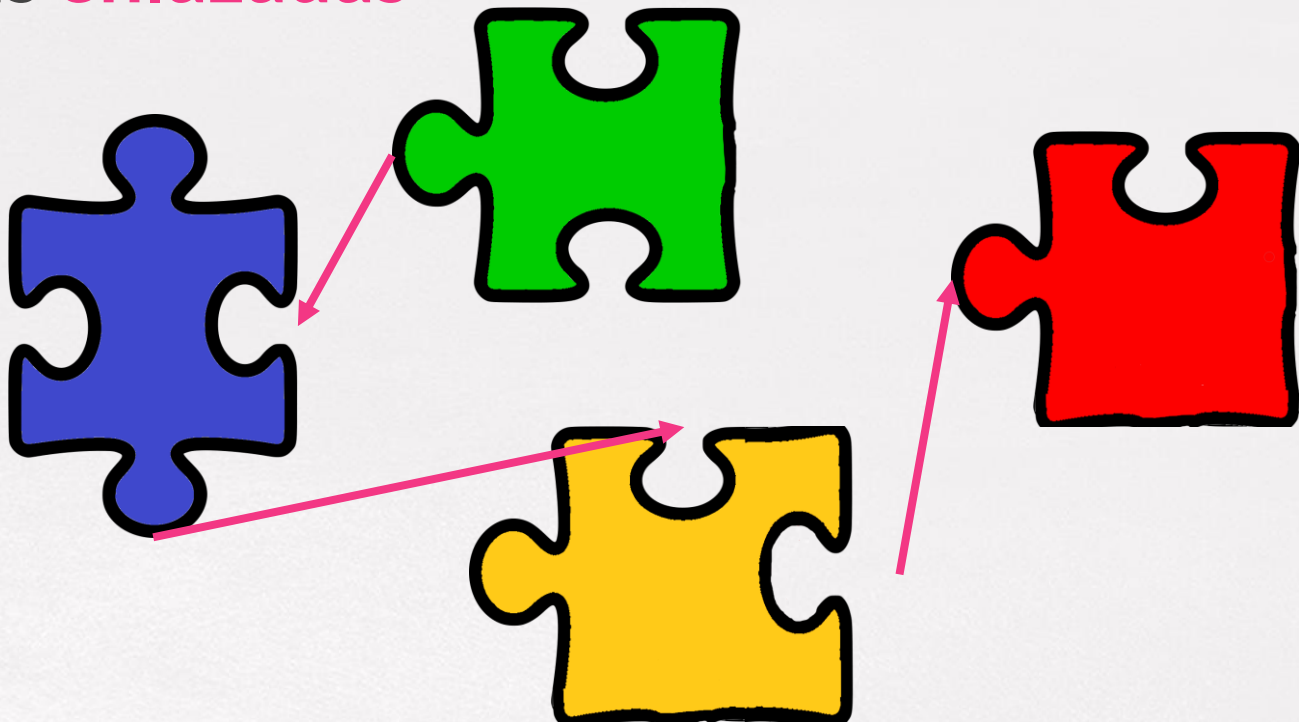
Colas y pilas

```
class Queue {  
  constructor(in_items) {  
    this.items = in_items || [];  
  }  
  length() {  
    return this.items.length;  
  }  
  enqueue(el) {  
    // Añade un elemento a items  
    this.items.push(el);  
  }  
  dequeue() {  
    // Devuelve el primer elemento o undefined  
    return this.length() > 0 ? this.items.shift() : undefined;  
  }  
}
```

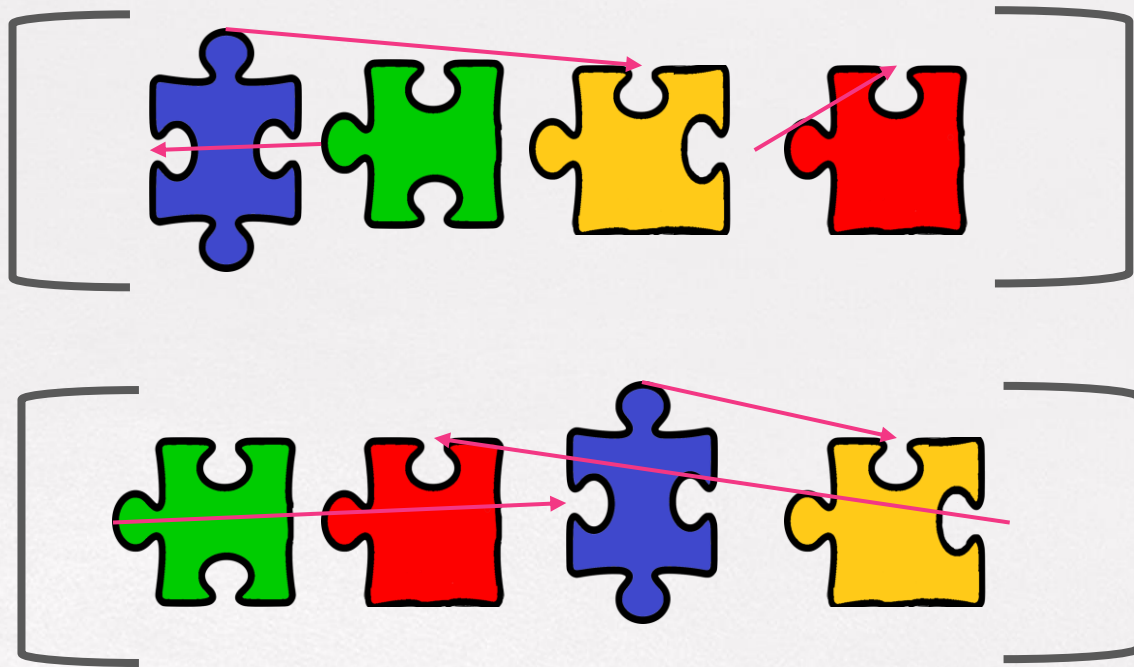
Listas **enlazadas**



Listas enlazadas



Listas enlazadas



Listas **enlazadas**

- Una lista enlazada es una estructura de datos **donde cada miembro tiene una referencia al miembro siguiente**
- Por lo tanto su orden no coincide con el orden de ordenación en memoria, sino que **está determinado por dichas referencias**
- En algunos casos son más eficientes que un array, pero sobre todo sus usos son útiles **para la representación de estructuras como grafos**

Listas enlazadas

```
class ListNode {  
  constructor(data) {  
    this.data = data;  
    this.nextId = null;  
  }  
}
```

```
let node1 = new ListNode(2)  
let node2 = new ListNode(5)  
node1.next = node2  
let list = new LinkedList(node1)
```

```
class LinkedList {  
  constructor(head = null) {  
    this.head = head;  
  }  
  getLast() {  
    let lastNode = this.head;  
    if (lastNode) {  
      while (lastNode.next) {  
        lastNode = lastNode.next  
      }  
    }  
    return lastNode  
  }  
  size() {  
    let count = 0;  
    let node = this.head;  
    while (node) {  
      count++;  
      node = node.next  
    }  
    return count;  
  }  
}
```

PARA RESUMIR

- ✓ Una estructura de datos es una **forma eficiente de organizar la información que nos ofrece ventajas en ciertos escenarios**
- ✓ Javascript tiene algunas estructuras útiles ya implementadas como Set o Map que son en ocasiones **más eficientes que usar listas**
- ✓ Otras estructuras de datos inexistentes en Javascript son en general **fácilmente implementables cuando comprendemos su funcionamiento**