DESARROLLO WEB EN ENTORNO CLIENTE
TEMA2: ESTRUCTURAS DE DATOS. JAVASCRIPT AVANZADO.

JESUITAS
Sagrado Corazón
Jesuitas · Logroño

Índice

1. Tipos

Tipos y tipado dinámico

```
let variable = "Hola"

typeof variable
# Esto es "string"
```

```
let variable = 25

typeof variable
# Esto es "number"
```

```
let variable = new Date()

typeof variable
# Esto es "object"
```

```
let variable;

typeof variable
# Esto es "undefined"
```

```
let variable = () => {};

typeof variable
# Esto es "function"
```

1. Tipos

Tipos y tipado dinámico

- Decimos que el tipado de Javascript es dinámico **porque no se establecen tipos en tiempo de compilación sino de ejecución**
- Una variable en javascript puede almacenar cualquier valor y **su tipo será una inferencia sobre dicho valor**
- Esto es intuitivo para desarrolladores novatos pero p**uede resultar un peligro potencial** para desarrolladores que vengan de lenguajes con un tipado estático

1. Tipos

Tipado débil

```
let variable;

typeof variable
# Esto es "undefined"

variable = 5;

typeof variable
# Esto es "number"

variable = "hola";

typeof variable
# Esto es "string"
```

- La declaración de variables **no exige la asociación** con un tipo de datos de forma implícita y unívoca.
- Las variables **son declaradas sin tipo** y este depende del tipo del valor
- Los valores pueden **modificarse, compararse y operar entre ellos** sin necesidad de realizar una conversión previa

1. Tipos

Coerción de tipos

- Llamamos coerción de tipos al **proceso de convertir un dato de un tipo a otro tipo distinto**
- Existen dos clases de coerción de tipos: **La coerción explícita (O casting) y la coerción implícita**
- La coerción implícita **sólo es posible en lenguajes con tipado débil o blando**
- En Javascript existen tres tipos de conversiones: **A String, a Number y a Boolean**

1. Tipos

Coerción de tipos: String

```
// Coerción explícita

String(123)  // '123'
String(-14.4) // '-14.4'
String(null) // 'null'
String(undefined) // 'undefined'
String(true)  // 'true'
String(false) // 'false'
let num = 123;
num.toString(); // '123'
```

```
// Coerción implícita

123 + '' // '123'
'' + 123 // '123'
```

- La coerción explícita se realiza mediante la función **String()**
- Los objetos y los tipo Number tienen un **método toString() para coerción explícita**
- La coerción implícita ocurre siempre que usemos el operador **suma (+)** con un string y otro tipo

1. Tipos

Coerción de tipos: Boolean

```
// Coerción explícita

Boolean('') // false
Boolean(0)  // false
Boolean(-0) // false
Boolean(NaN) // false
Boolean(null) // false
Boolean(undefined) // false
Boolean(false) // false
```

```
// Coerción implícita

if (2) { ... } // Se evalúa true
!!2 // true
2 || 'falso' // Se evalúa true, devuelve 2
```

- La coerción explícita se realiza mediante la función **Boolean()**
- Es mejor **recordar los valores que devuelven false**, como se observa en la imagen
- La coerción implícita dependerá **del contexto o del operador**, el operador negación **(!)** siempre forzará el casting al valor booleano opuesto

1. Tipos

Coerción de tipos: Number

```
// Coerción explícita

Number(null) // 0
Number(undefined) // NaN
Number(true)  // 1
Number(false) // 0
Number(" 10 ") // 10
Number("-29.45") // -29.45
Number("\n") // 0
Number(" 28s ") // NaN
Number(123) // 123
```

```
// Coerción implícita

+'123' // 123
123 != '123' // false
4 > '5' // true
5/null // Infinity
true | 0 // 1
```

- La coerción explícita se realiza mediante la función **Number()**
- La coerción implícita es más frecuente y **tiene más detonadores**
- hay que tener en cuenta que conversiones como **null == 0** o **null == undefined** no producen coerción y **devuelven false**

1. Tipos

Evitando la coerción de tipos

```
'180' + 200 + 3 // '1802003'
'hola' ++ 'mundo' // 'holaNaN'
!+[]+[]+![] // 'truefalse'

if ('true' == true) {...} // Se evalúa false, no entra
if ('true') {...} // Se evalúa true, entra
if (!!"false" == !!"true") // Se evalúa true, entra
if (0) {...} // Se evalúa false, no entra
if (38) {...} // Se evalúa true, entra
```

- La coerción de tipos implícita **puede ser una fuente de errores**, como puede observarse en los ejemplos de arriba

1. Tipos

Evitando la coerción de tipos

```
0 == [] // true
0 == '' // true
1 == true // true
1 != '1' // false
1 != true // false
```

```
0 === [] // false
0 === '' // false
1 === true // false
1 !== '1' // true
1 !== true // true
```

- Si usamos el operador == o el operador != para validar igualdades, **se producirá coerción de tipos** para hallar la igualdad
- Si usamos los operadores === y !== estos validarán las igualdades **comprobando tanto valor como tipo de las variables**

1. Tipos

PARA RESUMIR

- Javascript es un lenguaje con tipado dinámico y débil, lo que significa que **el tipo de las variables se asigna en ejecución** y que además **puede variar a lo largo de la ejecución**
- Llamamos coerción de tipos a **la conversión de un tipo de datos a otro**, ya sea esta **explícita (casting)** o **implícita**
- Para **evitar la coerción de tipos** en la comprobación de igualdad podemos usar los operadores **=== y !==** que además de comprobar valor comprueban tipo

2. Operadores y expresiones en JavaScript

Operadores y expresiones en JavaScript

```
// operando1 operador operando2
5 * 6 // -> 30
73 - 38 // -> 35
```

Operaciones con
operadores binarios

```
// operando1 operador
10++ // -> 11

// operador operando1
!10 // -> false
```

Operaciones con
operadores unarios

2. Operadores y expresiones en JavaScript

Operadores de comparación

==	"1" == 1	true
!=	"2" != 1	true
===	"1" === 1	false
!==	"1" !== 1	true
>=, <=	2 <= 2	true
>, <	2 > 3	false

- **El operador de comparación** compara sus operandos y devuelve un booleano
- Los operadores == y != **realizan coerción de los operandos si es necesario**
- Los operadores === y !== **comparan tipo y valor evitando coerción**

2. Operadores y expresiones en JavaScript

Operadores de asignación

x += y	x = x + y
x %= y	x = x % y
x **= y	x = x ** y
x &&= y	x && (x = y)
x ||= y	x || (x = y)
>,<	2 > 3

- Las asignaciones se evalúan de **derecha a izquierda**
- Para asignaciones más complejas podemos **usar el destructuring**

```
let variable = [2, 4];
let uno = variable[0];
let dos = variable[1];

// Con destructuring
let [uno, dos] = variable;
```

2. Operadores y expresiones en JavaScript

Operadores de asignación

x += y	x = x + y
x %= y	x = x % y
x **= y	x = x ** y
x &&= y	x && (x = y)
x ||= y	x || (x = y)
>,<	2 > 3

- Las asignaciones se evalúan de **derecha a izquierda**
- Para asignaciones más complejas podemos **usar el destructuring**

```
let variable = [2, 4];
let uno = variable[0];
let dos = variable[1];

// Con destructuring
let [uno, dos] = variable;
```

2. Operadores y expresiones en JavaScript

Otros operadores importantes

```
// condicion ? valor1 : valor2;
x = 5 > 2 ? 'sí!' : 'no :(';
// x vale 'sí!'

x = 5 > 2 ? (2 < 1 ? 'es menor' : 'es mayor') : 'no :(';
// x vale 'es mayor'
```

- El operador ternario nos sirve para asignar un valor entre dos posibles **evaluando una condición**
- **Es posible anidar ternarios dentro de los ternarios** si fuera necesario

2. Operadores y expresiones en JavaScript

Otros operadores importantes

```
let a = null
let b = 3

let res = a || b // -> 3

a = 5
b = 4

let res = a || b // -> 5

a = undefined
b = 4

let res = a || b // -> 4
```

- Los operadores binarios && y || permiten asignar valores **dependiendo de los valores falsy**
- El más usado es || sobre todo para hacer asignaciones **default** en caso de que un parámetro sea **falsy**
- El operador && **tiene el efecto opuesto**

2. Operadores y expresiones en JavaScript

PARA RESUMIR

- Una expresión es **cualquier acción que nos arroja un resultado**, y en el caso de las operaciones se realizan entre **uno o más operandos y un operador**
- Los operadores **pueden ser binarios si se realizan entre dos operandos o unarios solo con uno**
- La coerción de tipos es un **factor importante a tener en cuenta a la hora de usar operadores**, y es gracias a ella que muchos operadores tienen un uso extra en ciertas circunstancias

3. Estructura de datos en programación

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**

3. Estructura de datos en JavaScript: Set

Estructura en Javascript: Set

- 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()**

3. Estructura de datos en JavaScript: Set

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**

3. Estructura de datos en JavaScript: MAP

Estructura en Javascript: Map

- 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()**

3. Estructura de datos en JavaScript: MAP

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}
```

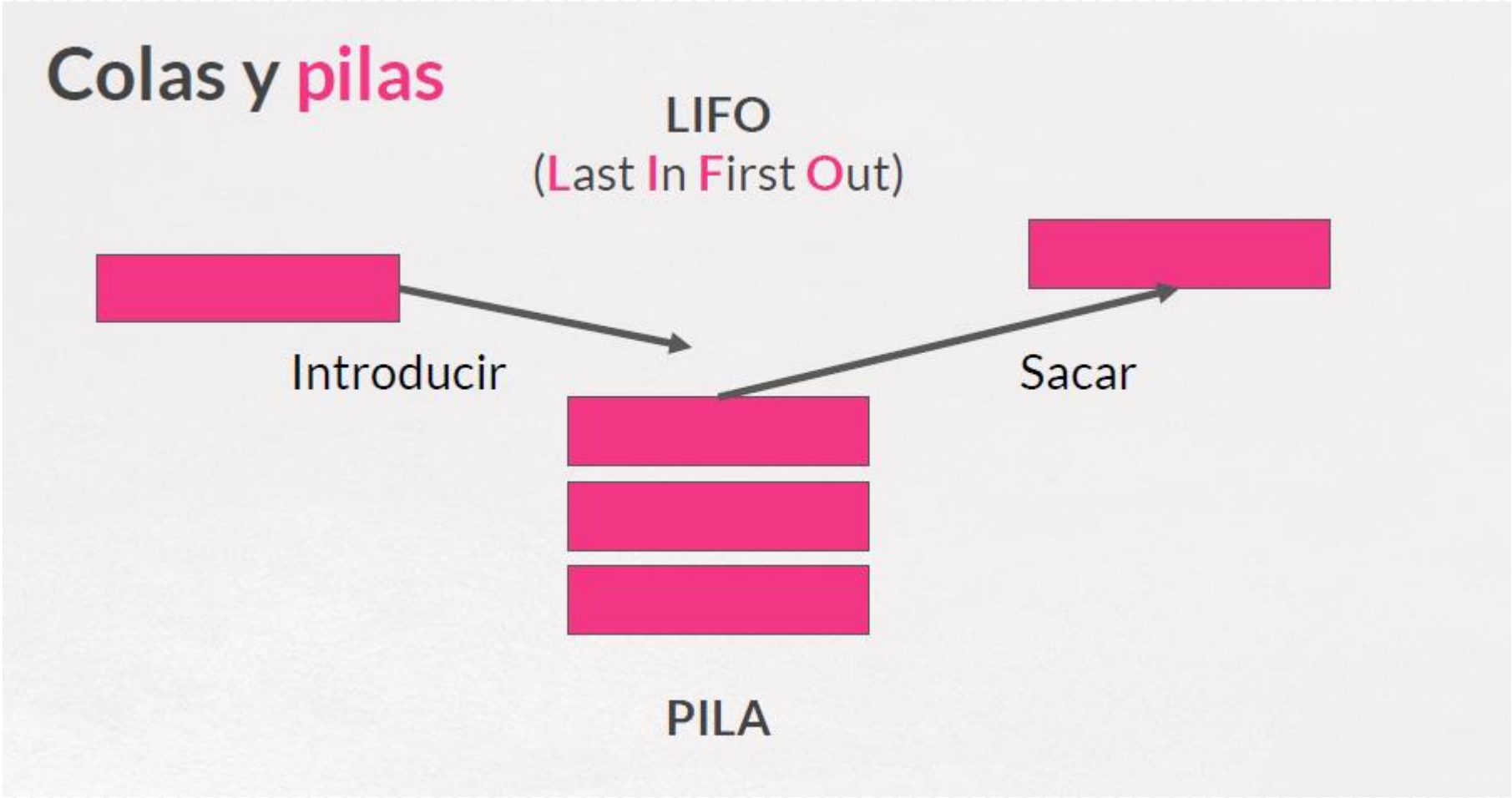
- A diferencia de un objeto **sus claves pueden ser objetos, funciones o cualquier tipo primitivo**
- También es sencillo sabe su tamaño usando la función **size()**
- Igualmente es posible iterar directamente sobre él, **ya que es un iterable**

3. Colas y pilas

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

3. Colas y pilas

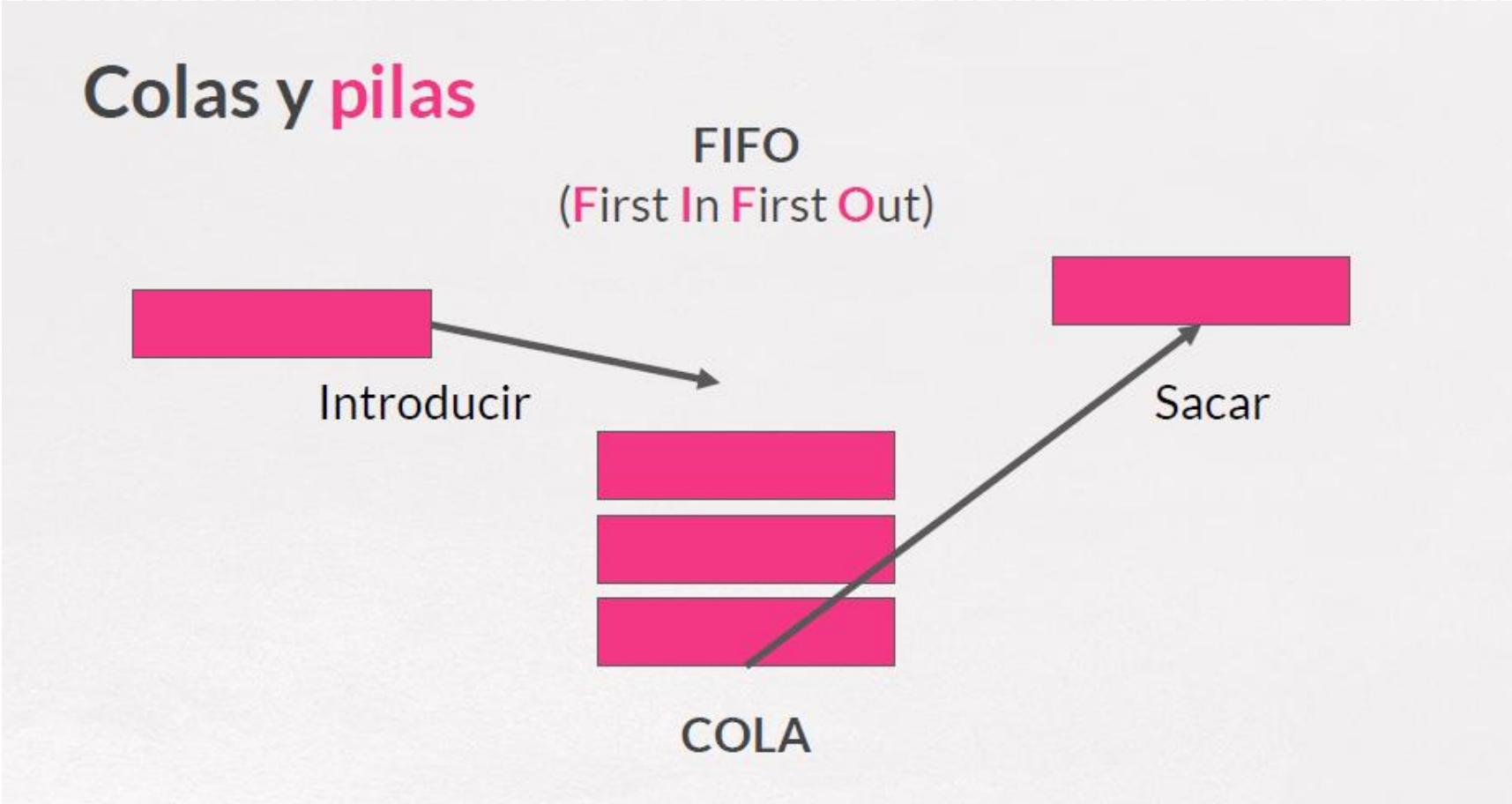


4. Colas y pilas

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;
    }
}
```

4. Colas y pilas

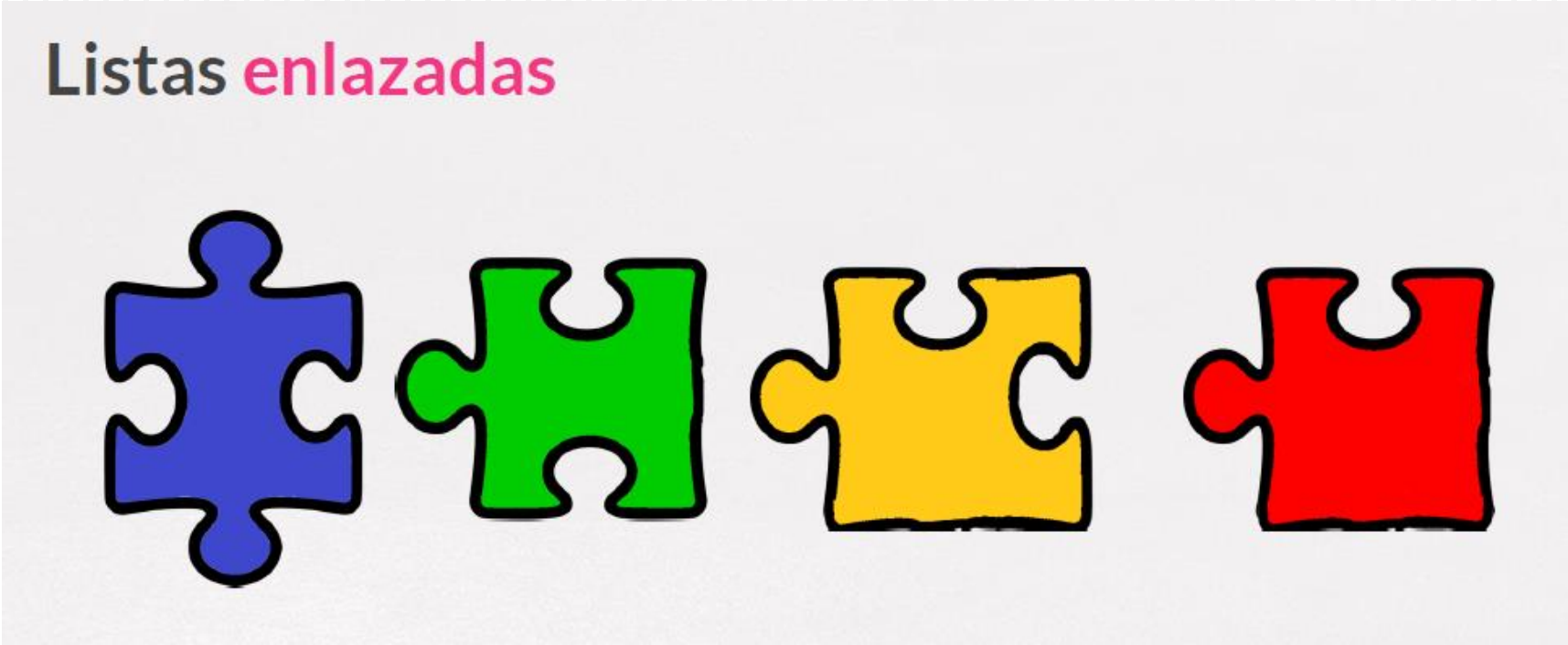


4. Colas y pilas

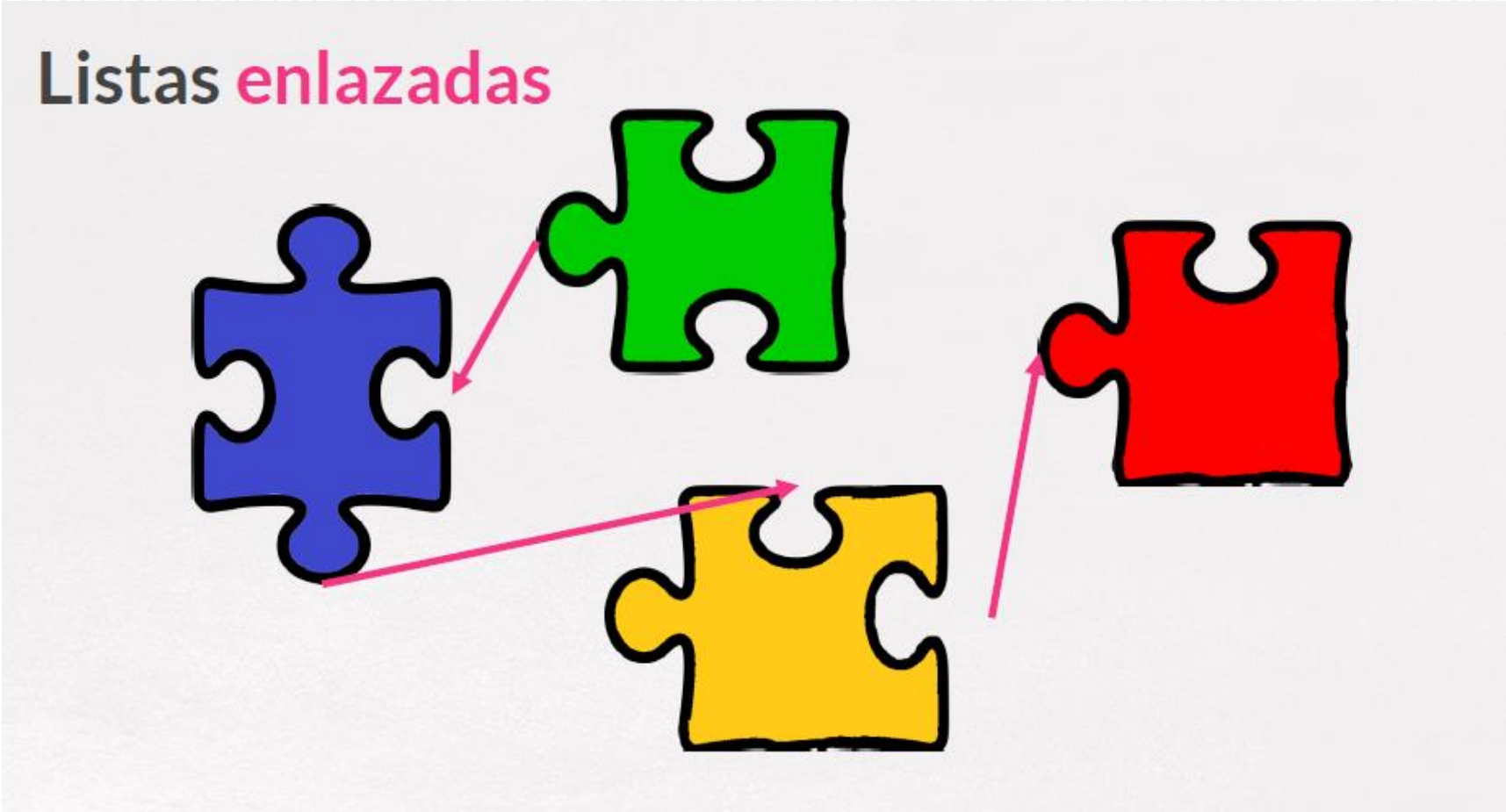
Colas y pilas

```
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;
    }
}
```

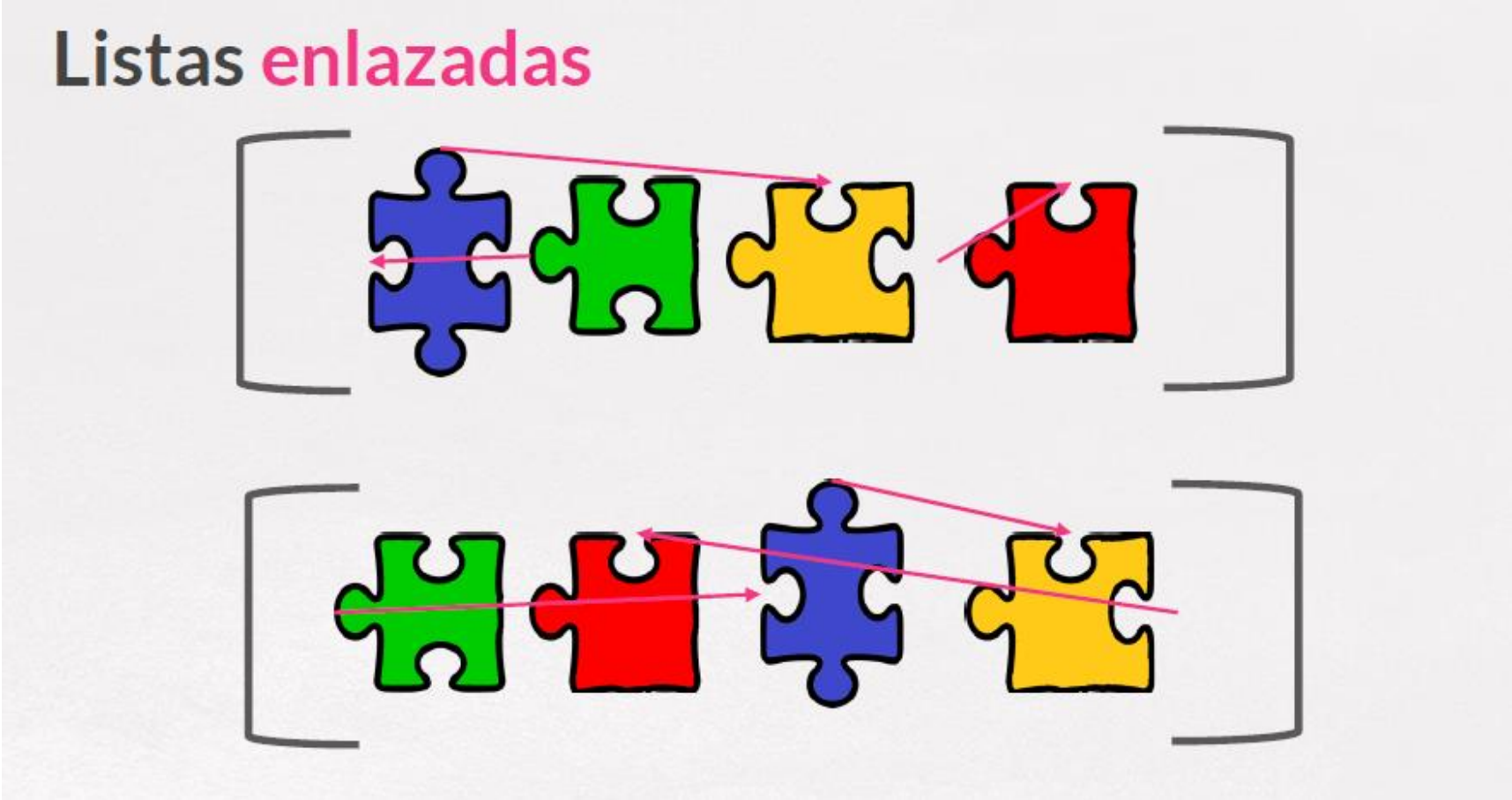
4. Listas enlazadas



4. Listas enlazadas



4. Listas enlazadas



4. 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**

4. Listas enlazadas

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;
    }
}
```

4. Resumen

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**