CLASE 04 | ESTRUCTURA DE DATOS I

POR ALEJO BENGOECHEA

ESTRUCTURA DE DATOS I

▼ **OBJETIVOS DE APRENDIZAJE**

En la clase de hoy aprenderás:

- Recursión
- Estructura de Datos
  - Array
  - Set
  - Stack (pila)
  - Queue (cola)

▼ **INTRODUCCIÓN**

La problemática más común a la que nos enfrentamos los programadores es crear un código prolijo y escalable. Existen distintas herramientas, como la **recursión**, que nos permite mejorar el rendimiento y la eficacia de nuestro código. Lo mismo sucede con las **estructuras de datos**. Estas nos permiten guardar y ordenar nuestra información de la manera más conveniente. Es por esto que hoy analizaremos estos temas y aprenderemos qué son y cómo implementarlos.

▼ **MATERIAL TEÓRICO**

▼ **RECURSIÓN**

▼ **¿Qué es?**

La **recursión** es una metodología que nos sirve para realizar una misma acción muchas veces. Más específicamente, para dividir un problema grande en muchos problemas pequeños, e iterar sobre cada uno de ellos de manera más sencilla.

Cuando hablamos de recursión hay que pensar en ***una función que se llama a sí misma.*** Es decir, que repetirá su propio comportamiento una y otra vez. Veamos un ejemplo:

```
function sumar(num) {
    var total = num + 2;
    return sumar(total);
  };
```

En este ejemplo podemos ver que la función **sumar** recibe un número por parámetro. A este número le sumamos dos, y luego retornamos la misma función pero con el nuevo valor. La diferencia es que la segunda vez que se repita, el parámetro será un nuevo número (*el anterior + 2*). En este caso el proceso se repetirá de manera infinita. *¿Eso está bien? ¡No!*

▼ **Características de la recursión**

Como dijimos en un principio, esta herramienta busca resolver un problema de forma rápida y eficiente. Para que esto sea así, la función recursiva debe cumplir tres condiciones muy importantes.

▼ **La función debe llamarse a sí misma.**

Como vimos anteriormente, la idea de este tipo de funciones es iterar su comportamiento.

▼ **La función siempre debe tener una (o más) sentencia de base o cierre.**

Si nosotros ejecutamos una función recursiva en la que no hay una sentencia de base, este código entrará en un *loop infinito* y no nos servirá de nada. Lo que buscamos es que la función se repita hasta que cumpla nuestro objetivo. Veamos un ejemplo:

```
function sumar(num) {
    if(num >= 10) return num;
    var total = num + 2;
    return sumar(total);
  };
```

Observemos que el código es el mismo de antes. Lo único que hicimos fue agregar un condicional al comienzo de la función. La

diferencia es que ahora, cuando el número recibido por parámetro sea mayor o igual a diez, la función dejará de retornase a sí misma: ***sentencia base o de cierre***.

Puede que esta clase de funciones tengan más de una sentencia de cierre. Esto dependerá del contexto y la problemática.

▼ **El argumento de la función retornada debe ser distinto al de su función padre.**

Pensemos que si estamos repitiendo una función una y otra vez, y siempre lo hacemos con el mismo input, el resultado nunca cambiará. Otra vez entraremos en un *loop infinito*.

Para evitar esto debemos asegurarnos que la función retornada lleve como argumento un valor distinto que el recibido. Veamos un ejemplo:

```
function sumar(num) {
    if(num >= 10) return num;
    var total = num + 2;
    total = total - 2;
    return sumar(total);
  };
```

En este caso el argumento de la función retornada es igual al recibido por la función padre. Por esto la variable **num** nunca llegará a ser mayor o igual a 10. En cambio, si dejamos la función a como estaba antes…

```
function sumar(num) {
    if(num >= 10) return num;
    var total = num + 2;
    return sumar(total);
  };
```

Ahora el valor del argumento nunca será el mismo ya que este irá incrementando de dos en dos.

▼ **¿Qué es mejor, usar recursividad o métodos iterativos?**

Durante la cursada en Henry aprendimos algunos métodos iterativos, como los bucles **for** o **while**. Estos nos permitieron recorrer alguna estructura de datos (arreglos por ejemplo), y realizar alguna acción en cada uno de esos elementos. Hay que tener en claro que los métodos iterativos y la recursión tienen fines distintos, por lo que uno no es mejor que otro. El contexto del problema nos dirá qué es más conveniente usar.

Hay que utilizar recursividad sólo cuando sea realmente necesario. Estas situaciones se dan cuando podemos dividir un problema en muchos subproblemas, y combinando la solución de cada uno de estos pequeños problemas logramos resolver el problema original.

▼ **ESTRUCTURA DE DATOS**

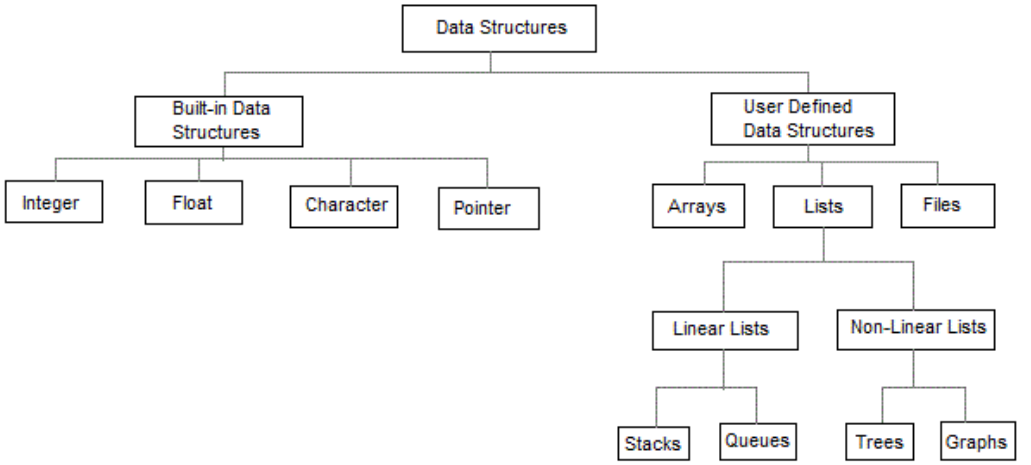
▼ **¿Qué es una estructura de datos?**

Las estructuras de datos son el estudio de las distintas formas en las que se puede organizar la información. El objetivo es que la búsqueda e inserción de datos sea lo más eficiente posible.

Dentro del mundo de la programación existen muchísimas estructuras de datos. Una que ya conocemos, por ejemplo, son los arreglos. Estos pueden guardar valores de forma lineal (uno después de otro).

> Pensemos en que los datos que tenemos que manejar son libros. En un principio tenemos muchísimos libros desordenados por toda la casa. Cada vez que queremos leer un libro tardamos cerca de dos horas buscando uno por uno hasta encontrarlo. ¡Esto es ineficiente! Para solucionar este problema podríamos

> crear una biblioteca. Allí acomodaremos todos los libros por orden alfabético, y encontrarlos será mucho más sencillo. La biblioteca es nuestra estructura de datos.



▼ ¿Cuáles son algunas de las estructuras más comunes?

▼ ARRAY

Los **arreglos**
son una estructura de datos nativa que tienen todos los lenguajes de programación (¡y que ya conoces!). Estos son una colección finita de elementos que ocupan espacios contiguos de la memoria. Se puede acceder a cada uno a través de su índice.

```
        [ “Soy string”, function(){}, { prop: “Comida” }, 45358 ]
Índices:        0            1               2             3
```

Dentro de los arreglos podemos guardar cualquier tipo de dato (incluso otros arreglos). Una de sus ventajas es que la inserción de elementos al principio o al final es muy sencilla de hacer. Pero buscar o eliminar elementos dentro de él tiene un poco más de dificultad.

▼ SET

Un **Set** es una colección de elementos sin un orden en particular, en donde cada elemento puede aparecer una sola vez. Es decir, es

muy parecido a un arreglo, pero con la diferencia de que ***no hay elementos repetidos***.

Esta estructura de datos ya está implementada de forma nativa en JavaScript, y la podemos utilizar con la clase **Set**. Veamos un ejemplo:

```
var arreglo = [ 1, 2, 2, 3, 4, 4, 2, 5, 6, 5 ]
var miSet   = new Set(arreglo)
// [ 1, 2, 3, 4, 5, 6 ]
```

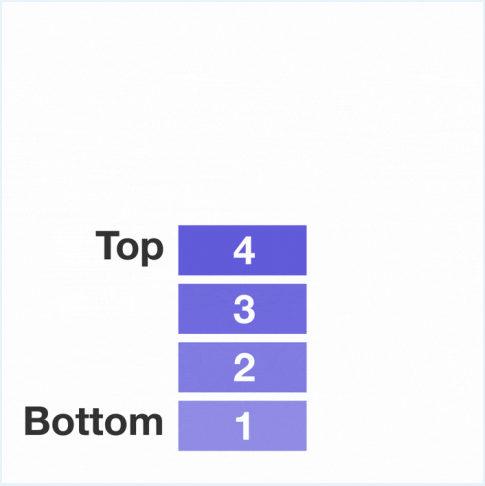
Al ser una estructura de datos distinta a los *arrays* también tiene métodos distintos. Por ejemplo:

```
var arreglo = [ 1, 2, 2, 3, 4, 4, 2, 5, 6, 5 ]
var miSet   = new Set(arreglo)
// [ 1, 2, 3, 4, 5, 6 ]

comp.add("Hola");
// [ 1, 2, 3, 4, 5, 6, "Hola"]
```

▼ STACKS (Pila)

Dentro de los **Stack** se cumple el principio **Last in - First Out (LIFO)**. Esto significa que el último en entrar a la pila es el primero en salir. Pensemos, por ejemplo, en el mesero de un restaurante. En la cocina recogerá los platos que pondrá en la mesa. Los apila uno arriba del otro para poder llevarlos a todos juntos. Pero cuando tiene que servir los platos en la mesa, el plato que está más arriba de su pila (el último en agregar) será el primero que servirá.



Los **stacks** tienen ***dos operaciones***: push( ) y pop( ). Esto es porque sólo se puede poner y sacar elementos en un orden definido. Si bien los *Stacks* no son estructuras nativas de JavaScript, existen múltiples formas de implementarlas. Por ejemplo, con arreglos, clases o listas enlazadas.

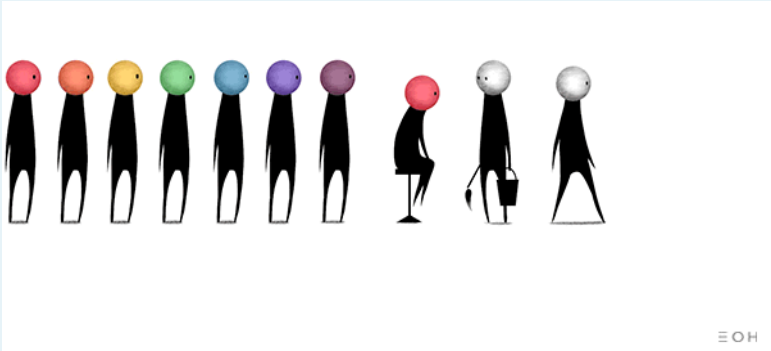
Si lo quisiéramos hacer con arreglos…

```
var stack = ["Plato 1", "Plato 2", "Plato 3"];

stack.push("Plato 4");
// ["Plato 1", "Plato 2", "Plato 3", "Plato 4"]
stack.pop()
// ["Plato 1", "Plato 2", "Plato 3"]
```

▼ QUEUE (Cola)

Las **queue** cumplen con el principio **First In - First Out (FIFO)**. Esto puede compararse con una fila para ir al baño por ejemplo. La primera persona que esté en la fila, va a ser la primera persona en irse (al baño). "*El primero en entrar es el primero en salir*".



Las **queue** tienen dos operaciones: ***enqueue( )*** y ***dequeue( )***. Estas sirven para agregar un elemento al final de la fila, y eliminar el primer elemento de la fila. Este tipo de estructura de datos, al igual que los ***stack***, no son nativas de JavaScript, pero se pueden implementar con arreglos, clases y listas enlazadas.

Si quisiéramos implementarlo con arreglos…

```
var queue = ["Persona 3", "Persona 2", "Persona 1"];

queue.unshift("Persona 4");
// ["Persona 4", "Persona 3", "Persona 2", "Persona 1"]
```

```
queue.pop()
// ["Persona 4", "Persona 3", "Persona 2"]
```

▼ **CONCLUYENDO…**

En la clase de hoy vimos dos conceptos importantes: **recursividad** y **estructuras de datos**.

Aprendimos que la recursividad es una herramienta que nos permite resolver código complejo y repetitivo de una manera eficiente y rápida. Cumple con las condiciones:

a. Es una función que se llama a sí misma.
b. La función debe tener una (o más) sentencia base o de cierre.
c. El argumento de la función retornada debe ser distinto al recibido por la función padre.

Si bien la recursividad es muy útil, hay que saber en qué contextos es adecuado usarla, y tener cuidado de no crear un *loop infinito*.

Por otro lado, hoy también descubrimos que en programación utilizamos distintos tipos de estructuras de datos. Estas estructuras nos permiten ordenar y encontrar información de forma mucho más rápida. Las que aprendimos con mayor profundidad fueron: **arrays**, **sets**, **stacks** y **queues**.

▼ **MEJOREMOS NUESTRO CONOCIMIENTO🤓**

¡Repasemos cómo funcionan los arreglos y cuáles son sus métodos! AQUÍ

¿Cómo podemos implementar la clase Set en JavaScript? AQUÍ