Resumen de estudio

POO → Programación orientada a objetos.

↳ es un paradigma (es decir un método para realizar determinados tipos o proyectos)

* Organiza el código en objetos
* Estos objetos se agrupan en datos (atributos) y comportamientos (METODOS)

Conceptos Claves

↳ **CLASES** → Es una plantilla ó modelo para crear objetos

↘ Define qué atributos y métodos tendrán los objetos que se crean a partir de ella.

a) Ejemplo de Clase:

```
Class Persona {
    //CONSTRUCTOR: SE EJECUTA AL CREAR EL NUEVO OBJETO
    Constructor (nombre, edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    //METODO: COMPORTAMIENTO QUE PUEDE REALIZAR EL OBJE
    Saludar () {
        Consok.log (`Hola, mi nombre es ${this.nombre} y
                    tengo ${this.edad} años.`);
    }
}
```

NOTA

b- **Objetos** → Es una instancia de una clase. Es una entidad que tiene los atributos y métodos definidos por la clase.

Ej crear un objeto

```
// CREAR UN OBJETO 'PERSONA1' DE LA CLASE 'PERSONA'
const persona1 = new Persona ('Juan', 30);
persona1.saludar(); → SALIDA: Hola, mi nombre es Juan y tengo 30 años.
```

c- **ATRIBUTOS** → Son variables que almacenan datos sobre el objeto. Son definidos en la clase y accesibles desde los métodos de la misma.

Ej de atributos

```
// ATRIBUTOS DE LA CLASE 'PERSONA': 'nombre' y 'edad'
console.log(persona1.nombre); → SALIDA → JUAN
console.log(persona1.edad); → SALIDA → 30
```

D- **METODOS** → Son funciones definidas dentro de una clase que realizan acciones o comportamientos sobre los datos del objeto.

Ej de metodos

```
// METODO "SALUDAR" de la clase 'PERSONA'
persona1.saludar(); → SALIDA → 'Hola, mi nombre es Juan y tengo 30 años'.
```

E- **HERENCIA** → permite crear una nueva clase basada en una clase existente.
La nueva clase hereda atributos y métodos de la clase base.

NOTA

Ej de herencia

```
// CLASE BASE PERSONA
Class Persona {
constuctor ( nombre, edad ) {
 this. nombre = nombre;
 this. edad = edad;
 }
 saludar() {
  console.log('Hola mi nombre es ${this.nombre} y tengo ${this.edad} años.');
  }
}

// CLASE DERIVADA 'ESTUDIANTE' que HEREDA DE 'PERSONA'

Class Estudiante extends Persona {
constructor ( nombre, edad, curso ){
super( nombre, edad) ---> // LLAMA AL CONSTRUCTOR DE 'PERSONA'
this. curso = curso;
}
 estudiar ( ){
 console.log (' ${ this. nombre} está estudiando ${this.curso}.');
  }
}

// CREAR UN OBJETO 'ESTUDIANTE' DE LA CLASE 'ESTUDIANTE'.

Const estudiante1 = new Estudiante ('Ana', 22, 'Matemáticas')
estudiante1. saludar ( ) ; -> SALIDA-> Hola, mi nombre es Ana y tengo 22 años
esludiante1. ~~saludar~~ estudiar ( ); -> SALIDA- Ana está estudiando.
```

NOTA

CLASES Y VECTORES EN POO

Modelo que define las propiedades y comportamientos de los objetos.

o Array se utiliza para almacenar multiples valores en una sola variable y se puede usar dentro de una Clase para organizar colecciones de objetos o valores relacionados.

Ej: Clase con un Vector

```
Class Curso {
 constructor(nombre) {
 this. nombre = nombre;
 this. estudiante = [ ]; // INICIALIZAMOS UN VECTOR VACIO PARA ESTUDIANTES.
}

// METODO PARA AGREGAR UN ESTUDIANTE AL CURSO

agregarEstudiante (estudiante) {
 this. estudiante.push (estudiante);
 console.log (`${estudiante} ha sido agregado al curso ${this. nombre}.`);
}

// METODO PARA MOSTRAR LA LISTA DE ESTUDIANTES INSCRIPTOS

mostrarEstudiantes ( ){
console.log (`Estudiantes inscriptos en ${this. nombre}:`);
this. estudiante .forEach ((estudiante, index) => {
console.log (`${index +1}. ${estudiante}`);
});
}
}
```

NOTA

Eliminar un estudiante

```
Class Curso {
constructor (nombre) {
this.nombre = nombre;
this.estudiantes = [ ];
}
```

asigno el parámetro nombre al atributo nombre del objeto.
propiedad estudiantes → guarda el nombre del curso → vector vacío para almacenar los alumnos.

```
agregarEstudiante (estudiante) {
this.estudiantes.push (estudiante);
console.log (`${estudiante} ha sido agregado al curso ${this.nombre}.`);
}
```

→ agregamos el estudiante al [ ]

```
eliminarEstudiante (estudiante) {
const index = this.estudiantes.indexOf (estudiante)
if (index !== -1) {
this.estudiantes.splice (index, 1);
console.log (`${estudiante} ha sido eliminado del curso ${this.nombre}.`);
} else {
console.log (`${estudiante} no está inscripto en el curso.`);
}
}
```

indexOf (estudiante) → este método busca el valor especificado (nombre del estudiante) dentro del array y devuelve la posición (índice) donde se encuentra.

→ si index === -1 (menos uno) significa que el estudiante no está en el [ ]

Se verifica si el estudiante fué encontrado en el [ ]
Si index es distinto a (-1) es decir existe y se elimina.

Si el estudiante está en el array, devuelve su índice un nº ≥ 0 y si no está - 1

this.estudiantes.splice (index, 1)

splice ( ) es un método de los [ ] que permite eliminar o modificar elementos.

es el índice desde donde queremos empezar a eliminar (en este caso el valor index que obtenemos de indexOf( )

Cuántos elementos queremos eliminar (en este caso 1 un solo estudiante)

NOTA

HOJA Nº

FECHA

Ej: Si estudiantes es ['Juan', 'Manu', 'Pedro'] y quiero eliminar a Manu
0 1 2

indexOf('Manu') devolverá 1

splice(1, 1) elimina el elemento en la posición 1 (Manu) dejando el array ['Juan', 'Pedro']

Buscar estudiante por el nombre

```
Class Curso {
constructor (nombre){
this. nombre = nombre;
this. estudiantes = [ ];

agregarEstudiante (estudiante){
this. estudiantes.push (estudiante);
console.log (`${estudiante} ha sido agregado al curso ${this. nombre}.`);

buscarEstudiante (estudiante)
return this. estudiantes.includes (estudiante),

mostrarEstudiantes ( ){
console.log ('Estudiantes inscriptos en ${this. nombre}:');
this. estudiantes. forEach ((estudiante, index) => {
 console.log (` ${index + 1}. ${estudiante}`);
});
}
}
```

Este metodo recibe el nombre del estudiante y lo agregal al [ ]

Agrega un nuevo estudiante al final del [ ]

recibe el nombre del estudiante y devuelve TRUE SI ESTA INSCRIPTO SINO FALSE

metodo que verifica si un valor esta presente en el [ ] si esta TRUE SINO FALSE

NOTA

```
let cursoJavascript = new Curso ('JAVASCRIPT AVANZADO');
cursoJavascript.agregarEstudiante ('Juan');
cursoJavascript.agregarEstudiante ('Manu');
cursoJavascript.agregarEstudiante ('Pecho');

// BUSCAR ESTUDIANTE

console.log (cursoJavascript.buscarEstudiante ('Manu')); TRUE
console.log (cursoJavascript.buscarEstudiante ('Ana')); FALSE

// ELIMINAR ESTUDIANTE

cursoJavascript.eliminarEstudiante ('Manu');

// MOSTRAR ESTUDIANTES RESTANTES

cursoJavascript.mostrarEstudiantes ();
```

OPERADOR SPREAD (...)

Se utiliza para expandir los elementos de un array o objeto, es decir, "extiende" sus contenidos para que puedan combinarse o usarse de manera más flexible.

COMBINAR ARRAYS

```
Const VectorAcumulado = [...clientesSucursal1, ...clientesSucursal2];
```

Aquí en este (vector) se combinan (los arrays)

COMBINAR OBJETOS

```
const objetoEntero = {...persona1, ...cuentaBancaria};
```

Aquí combinó (2 objetos) si hay propiedades iguales las copias no las elimina ej: (cuit)

REDUCE ()

→ permite acumular un resultado a lo largo de las iteraciones.

Es un método del array que reduce el array a un solo valor al iterar sobre cada elemento.

Ejemplo (sin) REDUCE → contando productos con bajo stock

```
let contadorEnPtoReposicion = 0;

productos.forEach (element => {
  if (element.stock < 10 ) {
    contadorEnPtoReposicion = contadorPtoReposición + 1;
  }
});
```

NOTA

Contando productos con REDUCE

```
let resultado1 = productos.reduce((contador, elemento) => {
    if (elemento.stock < 10){
        ++contador;   // incremento el contador antes de devolverlo
    }
    return contador;   // Retorno el contador incrementado o no, según corresponda
}, 0);   → VALOR INICIAL

console.log(resultado1)
```

- El valor inicial del acumulador es cero. Es decir, antes de comenzar a iterar sobre el array de productos el contador comienza en 0
- En cada iteración si la condición (elemento.stock < 10) es verdadera. Se incrementa el contador
- El resultado será el nº total de productos cuyo stock es menor a 10
- Se puede un array vacío [ ] para construir un nuevo array o un objeto vacío { } para construir un objeto acumulativo.

Otros ejemplos

Sumar números

```
let suma = [1, 2, 3, 4].reduce((total, numero) =>
        total + numero, 0);

console.log(suma);   → 10
```

NOTA

HOJA N°

FECHA

Concatenar cadenas

```
let concatenado = ['Hola', ' ', 'Mundo'].reduce
    ((texto, palabra) => texto + palabra, '');
console.log(concatenado);  → 'Hola Mundo'
```

Construir un Objeto

```
let productos = [
  {id: 1, nombre: 'notebook'}
  {id: 2, nombre: 'teclado'}
];

let catalogo = productos.reduce ((objeto, producto) => {
    objeto[producto.id] = producto.nombre;
return objeto;
}, {});
console.log(catalogo); → // {'1': 'notebook', '2': 'Teclado'}
```

NOTA

REDUCE CON OPERADOR TERNARIO

```
let resultado2 = productos.reduce((contador, elemento) => {
    return elemento.stock < 10 ? ++contador : contador;
}, 0);
```

- `++contador` → antes de devolverlo, incrementa el valor del contador en 1
- `contador` → si es falso, devuelve el valor actual del contador sin modificarlo.

`Condición ? valorSiEsVerdadero : valorFalso` → compara si es V o F

REDUCE CON OBJETOS LITERALES

```
let resultado3 = facturas.reduce((objetoLiteral, elemento) => {
    if (elemento.cantidad >= 4) {
        objetoLiteral.contadorFacturas = objetoLiteral.totalFacturas + elemento.total;
    }
    return objetoLiteral;
}, { contadorFacturas: 0, totalFacturas: 0 });
```

`{ contadorFacturas: 0, totalFacturas: 0 }` → (VALOR INICIAL)

- `objetoLiteral` → acumulador (comienza en 0)
- `elemento` → elemento actual del array que está siendo procesado en la iteración
- `=> { ... }` función flecha que dice cómo se va a transformar el acumulador (objetoLiteral) en c/iteración

CONDICIÓN →

`if (elemento.cantidad >= 4)`: Se verifica si la cantidad en el elemento actual (elemento.cantidad) es >= 4

Si es VERDADERO → Se actualizan las propiedades del acumulador (objetoLiteral)

• es para acceder o modificar una propiedad.

HOJA Nº

FECHA

objetoLiteral.contadorFacturas → Se incrementa en 1. Esto cuenta el número de facturas ≥ 4.

objetoLiteral.totalFacturas → Suma el total de la factura actual (elemento.total). Esto acumula el total de las facturas que cumplen con la condición.

RETURN.objetoLiteral → Al final de c/iteración la f devuelve el acumulador ACTUALIZADO (objetoLiteral)

Y este valor se usa como acumulador para la siguiente iteración.

Resultado Final → Después que REDUCE haya procesado todos los elementos del array facturas, resultado contendrá el objeto final con las propiedades contadorFacturas y totalFacturas actualizadas según las condiciones especificadas.