

# Investigación

## ArrayList

Es una estructura de datos dinámica, que representa una lista ordenada de datos del mismo tipo... Es muy similar a lo que es una array, con la ventaja de que su tamaño se puede ampliar o reducir.

## Métodos

### Agregar Elementos

- `add(element)` : agrega el elemento al final de la lista
- `addAll` : agrega todos los elementos de la colección al final de la lista
- `add(index, element)` : agrega el elemento en la posición específica

### eliminar elementos

- `remove(elemento)` : remueve la primera aparición del elemento
- `remove(index)` : remueve el elemento en la posición especificada.
- `removeAll` : elimina todas las ocurrencias de los elementos de la colección específica de la lista
- `clear` : elimina todos los elementos del array list.

### Acceder a elementos

- `get(index)` : devuelve el elemento en la posición especificada de la lista.
- `set(index, element)` : Reemplaza el elemento en la posición
- `size()` : devuelve el número de elementos de la lista
- `isEmpty()` : devuelve un booleano indicando si la lista está vacía

### Modificar la lista

- `listIterator()` : devuelve un iterador de la lista que permite recorrerla y modificarla

- `listIterator()` : devuelve un iterador de la lista que permite modificar a partir de la posición indicada.
- `foreach()` : recorre la lista realizando una acción para cada elemento.

## Métodos para convertir la lista

- `contains(elemento)` : devuelve true si el elemento tiene la lista
- `containsAll()` : lo mismo pero para colecciones
- `indexOf(element)` : devuelve la primera posición del elemento indicado, es decir, la primera coincidencia
  - `lastIndexOf(elemento)` : devuelve la última posición de la lista donde aparece el elemento, si no lo encuentra, te devuelve -1

## igualdad y hash

- `equals(objeto)` : compara la lista con el objeto especificado para determinar la igualdad
- `hashCode` : Devuelve un código hash para la lista


## métodos adicionales

- `subList(fromindex, index)` : devuelve una sublista de la lista original, desde la posición fromindex hasta index
- `sort(comparador)` : ordena la lista utilizando el comparador especificado.
- `sort(lambda)` o `sort((o1,o2)-> o2.compare(o1))` : ordena la lista en orden descendente utilizando el método

# MAP

Un `MAP` en Java es una interfaz que representa una colección de pares clave-valor. Cada par consiste en una **clave única** y un **valor asociado**. Las claves deben ser únicas dentro del `MAP`, mientras que los valores pueden ser duplicados. La implementación más común de `MAP` es la clase `HashMap`.

- colección: una **colección** es una interfaz que representa un grupo de elementos del mismo tipo.
- interfaz: una interfaz se define como una colección de métodos abstractos y constantes públicas

 *diferencia*: una interfaz se diferencia de una clase en que ella no contiene implementación de métodos.

## Métodos

### Insertar pares clave-valor

- `put(key, value)` : insertar un nuevo par clave-valor en el map, si el valor ya existe se sobre escribe.
- `putAll()` : Inserta todos los pares clave-valor del `Map` especificado en el `Map` actual.

### obtener valores

- `get(key)` : obtiene el valor asociado a la clave especificada, si no existe devuelve `null`
- `containsKey(key)` : comprueba si la clave especificada existe en el `map`, retorna un boolean
- `getOrDefault(key, defaultValue)` : obtiene el valor asociado a la clave especificada si la clave no existe, devuelve el valor predeterminado asignado.

### eliminar pares clave-valor

- `remove(key)` : elimina el valor clave-valor asociado a la clave especificada devuelve el valor eliminado si existe, o `null` si no existe.
- `clear()` : Elimina todos los pares clave-valor del `Map`.

### iterar sobre elementos map

- `keySet()` : Devuelve un conjunto que contiene todas las claves del `Map`.
- `values()` : Devuelve una colección que contiene todos los valores del `Map`.
- `entrySet()` : Devuelve un conjunto que contiene todos los pares clave-valor del `Map` en forma de objetos `Map.Entry`.

### adicionales

- `equals(Object o)` : Compara dos `Map` para determinar si son iguales en términos de contenido.

- `hashCode()` : Devuelve un hash code para el `Map` .
- `toString()` : Devuelve una cadena que representa el contenido del `Map` .

## Ejemplo

```
List<String> productos = Arrays.asList("camiseta-roja", "pantalon-azul",
"zapatos-deportivos"); Map<String, Double> precios = new HashMap<>();
precios.put("camiseta-roja", 15.99); precios.put("pantalon-azul", 29.95);
precios.put("zapatos-deportivos", 59.99); double totalCompra =
calcularTotalCompra(productos, precios); System.out.println("Total de la
compra: " + totalCompra);
```

## POO

principios de OOP

### ¿Qué es y que no es oop?

aquí una discusión sobre el asunto

link: <https://wiki.c2.com/?DefinitionsForOo>

que define a un lenguaje opp?

1. capacidad de modelar problemas atreves de objetos

#### Requerimientos 🛠️

- asociación: objetos con la capacidad de referir a otros objetos
- agregación: capacidad de un objeto de referir a otros objetos independientes.
- composición: capacidad que tienen los objetos de referir a otros objetos dependientes.

2. soporte de algunos principios que soporten la modularidad y la reutilidad del código

#### requerimientos 🛠️

- encapsulación: capacidad de concentrar datos y códigos en una sola entidad.
- herencia: mecanismo por la que un objeto puede obtener las características de otro o mas objetos.
- polimorfismo: capacidad de procesar objetos con diferentes tipos de datos y estructuras pero que al final pueden darnos una respuesta.

modular: proyecto grande que se puede dividir en partes pequeñas las cuales se pueden reutilizar

reutilizable:

## Asociación

Los objetos estan relacionado: maria y jhon gracias a parent

```
// asociación

class Person{
  constructor(name, lastname){
    this.name=name

    this.lastname=lastname
  }
}

const jhon= new Person("jhon", "ray")

const maria= new Person("maria", "perez")

maria.parent=jhon // adición de otra clave... Esta es la relación

console.log(maria)
console.log(jhon)
```

## Agregación

en employes se encuentran los dos objetos que tienen vida independiente

```
// asociación

const company={

  name:"fazt tech",
  employes:[] // objeto que refiere a objetos
```

```
independientes
```

```
}
```

```
class Person{
```

```
    constructor(name, lastname){
```

```
        this.name=name
```

```
        this.lastname=lastname
```

```
    }
```

```
}
```

```
const jhon= new Person("jhon", "ray")
```

```
const maria= new Person("maria", "perez")
```

```
company.employees.push(maria)
```

```
company.employees.push(jhon)
```

```
maria.parent=jhon
```

```
console.log(company);
```

## composición

*No se puede referir al objeto adrees por fuera de company... Ni siquiera tiene sentido*

***El objeto que se encuentra dentro no tiene vida independiente ya que pertenece a otro que es company***

```
// composición.

const company={
  name:"ryan",
  lastname:"rya",
  ⚡ adress:{
    .....street:"123 baker street",
    .....country:"united kingdom"
    .....}
}

adress:{
  street:"123 baker street",
  country:"united kingdom" ';' exp
}
```

## Encapsulación

- simplificar el uso de un objeto, el usuario no tiene que saber como funciona internamente... El usuario solo debe saber que funciona y listo.

*concentrando datos en una sola entidad*

```
// encapsulacion

function Company(name){

  let employees=[] // variable por lo tanto no se hereda en las nuevas instancias

  this.name=name // atributo
  this.getEmployees=function(){
    return employees
  }
  this.addEmployee= function(employee){
    employees.push(employee)
  }
}
```

```
const company=new Company("coca cola")
const company2= new Company("pepsi")

console.log(company)
console.log(company2)

company.addEmployee({name:"ryan"})

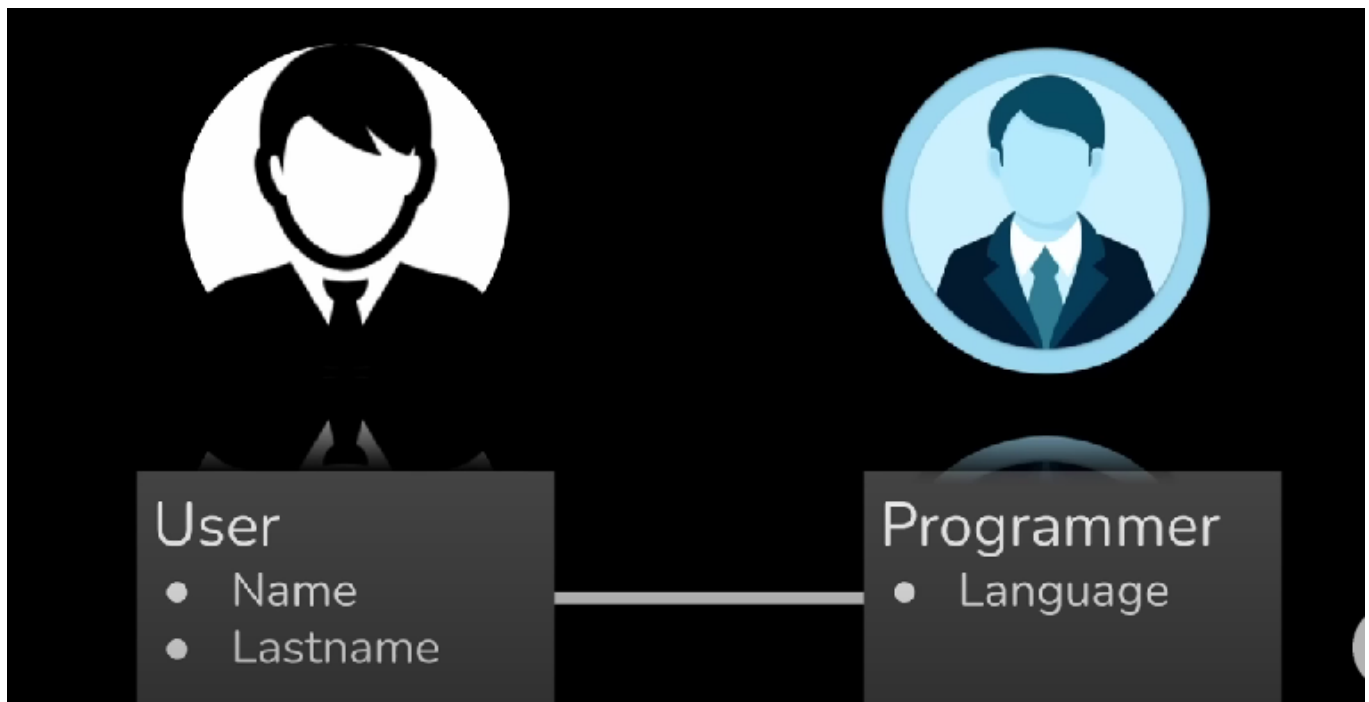
company2.addEmployee({name:"joe"})

console.log(company.getEmployees());

console.log(company2.getEmployees());
```

## Herencia

*principio de crear objetos más especializados a partir de uno mas genérico*



A lo siguiente se le conoce como herencia.

```
function Person(){

    this.name=""
```



```
this.lastname=""
```

```
}
```

```
function Programmer(){
```

```
    this.language=""
```

```
}
```

```
Programmer.prototype=new Person()
```

```
console.log(Programmer)
```

```
console.log(Person)
```

```
const person = new Person()
```

```
console.log(person)
```

```
const programmer= new Programmer()
```

```
programmer.name="robert"
```

```
programmer.lastname="zapata"
```

```
programmer.dato="experimental" // aqui se comprueba que tambien se puede
añadir uno dato nuevo
```

```
console.log(programmer)
```

```
[Function: Programmer]
[Function: Person]
Person { name: '', lastname: '' }
Person {
  language: '',
  name: 'robert',
  lastname: 'zapata',
  dato: 'experimental'
}
```

- **Herencia con clases** 🧑🧑🧑

```
class Person{

    constructor(name, lastname){
        this.name=name
        this.lastname=lastname
        this.age=null
    }
}

class Programmer extends Person{

    constructor(language, lastname, name){
        super(name, lastname)
        this.language=language
    }
}

const person= new Person("maria", "perez")

console.log(person)
```

```
const programmer= new Programmer("joe", "mcmillan", "python")

console.log(programmer)
```

## Polimorfismo

capacidad que tienen algunos objetos para poder manipular distintos tipos de datos de manera uniforme.

característica:

- sobre carga: Los métodos pueden tomar parámetros con diferentes tipos de datos
- polimorfismo paramétrico: gestionar tipos genéricos. no se conocen de antemano
- polimorfismo exclusivo: puede ser representado por una clase y derivado de ella

## sobrecarga

tomando la misma función o método para tomar 2 y luego 3 parametros... en el ejemplo se usa sobre carga.

```
public int Sum(int x, int y) {
    return Sum(x, y, 0);
}

public int Sum(int x, int y, int z) {
    return x + y + z;
}
```

**C# Example - Different Number of Arguments**



```
1 using System;
2
3 class Program {
4     public static void Main (string[] args) {
5         Console.WriteLine ("Hello World");
6
7         Program program = new Program();
8         Console.WriteLine(program.CountItems(3030303));
9         Console.WriteLine(program.CountItems("Hola mundo"));
10
11         Console.WriteLine(program.Sum(10, 20));
12         Console.WriteLine(program.Sum(10, 20, 30));
13     }
14
15     // Overloading
16     public int CountItems(int x) {
17         return x.ToString().Length;
18     }
19
20     public int CountItems(string x) {
21         return x.Length;
22     }
23
24     public int Sum(int x, int y) {
25         return x + y;
26     }
27
28     public int Sum(int x, int y, int z) {
29         return x + y + z;
30     }
31
32 }
```

```

1 using System;
2
3 class Program {
4     public static void Main (string[] args) {
5         Console.WriteLine ("Hello World");
6
7         Program program = new Program();
8         Console.WriteLine(program.CountItems(3333333));
9         Console.WriteLine(program.CountItems('Hola mundo'));
10
11         Console.WriteLine(program.Sum(10, 20));
12         Console.WriteLine(program.Sum(10, 20, 30));
13     }
14
15     // Overloading
16     public int CountItems(int x) {
17         return x.ToString().Length;
18     }
19
20     public int CountItems(string x) {
21         return x.Length;
22     }
23
24     public int Sum(int x, int y) {
25         return x + y;
26     }
27
28     public int Sum(int x, int y, int z) {
29         return x + y + z;
30     }
31 }

```

```

function sum(x,y,z){
    return x+y+z
}

```

```
console.log(sum(10,20))
```

// cuando se le pasan dos parametros en nan el console

```

function sum(x=0,y=0,z=0){
    return x+y+z
}

```

```
console.log(sum(10,20))
```

// cero sera el valor que tome por undefined z asi el valor resultante es 30

```
// esto es sobre carga.
```