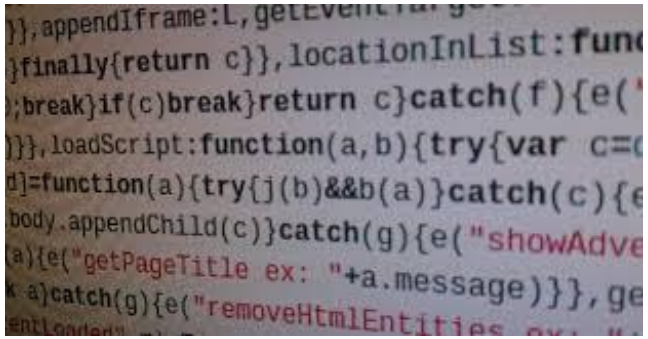


UT 12

COLECCIONES

IES JUAN DE LA CIERVA
DPTO. INFORMÁTICA

CONTENIDOS
DE LA
PRESENTACIÓN

Objetivos

La unidad permite al alumno conocer que son las colecciones en java, cual es su utilidad, los distintos tipos que tenemos y elegir la mejor opción para nuestro problema.

Contenidos

- Concepto de Colección.
- List. Tipos de Lists.
- Manejo y creación de listas (ArrayList, LinkedList, Vector...)
- Set. Tipos de sets.
- Map. Tipos de Maps.

TIPOS DE VARIABLES

Clasificación de las variables que se usan en los programas:

1. **Variables simples**: int, char, long, float, String…..

2. **Estructuras:** son “variables” donde cabe más de un dato del tipo especificado.

TIPOS DE ESTRUCTURAS

Cada estructura de datos se caracteriza por:

- Dónde se guardan los datos.
- La forma en que están organizados.
- Las operaciones que se pueden realizar.

Según dónde se guarden los datos, se pueden clasificar en:

- **Estructuras internas.** Sus datos se guardan en memoria RAM.
- **Estructuras externas.** Se guardan en soportes externos permanentes (HD, pendrive…) Ficheros y B.D.

ESTRUCTURAS INTERNAS

- Estáticas (tablas).

  Se dice que son estáticas porque tienen un tamaño fijo que se determina en la creación del objeto y no se puede variar.

- Dinámicas (Pilas, listas, árboles,….). **COLECCIONES**

  Se caracterizan porque no tienen un tamaño fijo, y se pueden ir ampliando o disminuyendo a lo largo de la ejecución del programa según las necesidades y espacio disponible en memoria.

  Las diferentes colecciones dinámicas que existen se diferencian por la forma en la que se relacionan los diferentes datos que las componen.

ESTRUCTURAS INTERNAS

- Estáticas (tablas).

DIFICULTADES CON LOS ARRAYS

- Conocer a priori el tamaño
- El tamaño no se puede modificar una vez creado.
- Problemas para insertar objetos en posiciones intermedias
- No son realmente objetos
- ...

- Dinámicas (Pilas, listas, árboles,….). **COLECCIONES**

Beneficios

- Menos esfuerzo de programación.
- Aumento de la calidad y velocidad.
- Interoperabilidad
- Curva de aprendizaje pequeña
- Reusabilidad.

Colecciones

COLECCIONES

- Colecciones: Estructuras internas de almacenamiento que **crecen dinámicamente**.
- Conjunto de clases e interfaces que **mejoran** notablemente las **capacidades del lenguaje** respecto a estructuras de datos.
- Son **contenedores de objetos genéricos de la clase Object**, no permiten el uso de tipos primitivos. Todas las colecciones se encuentran en el **paquete java.util.***

¿Qué son y para que sirven las colecciones?

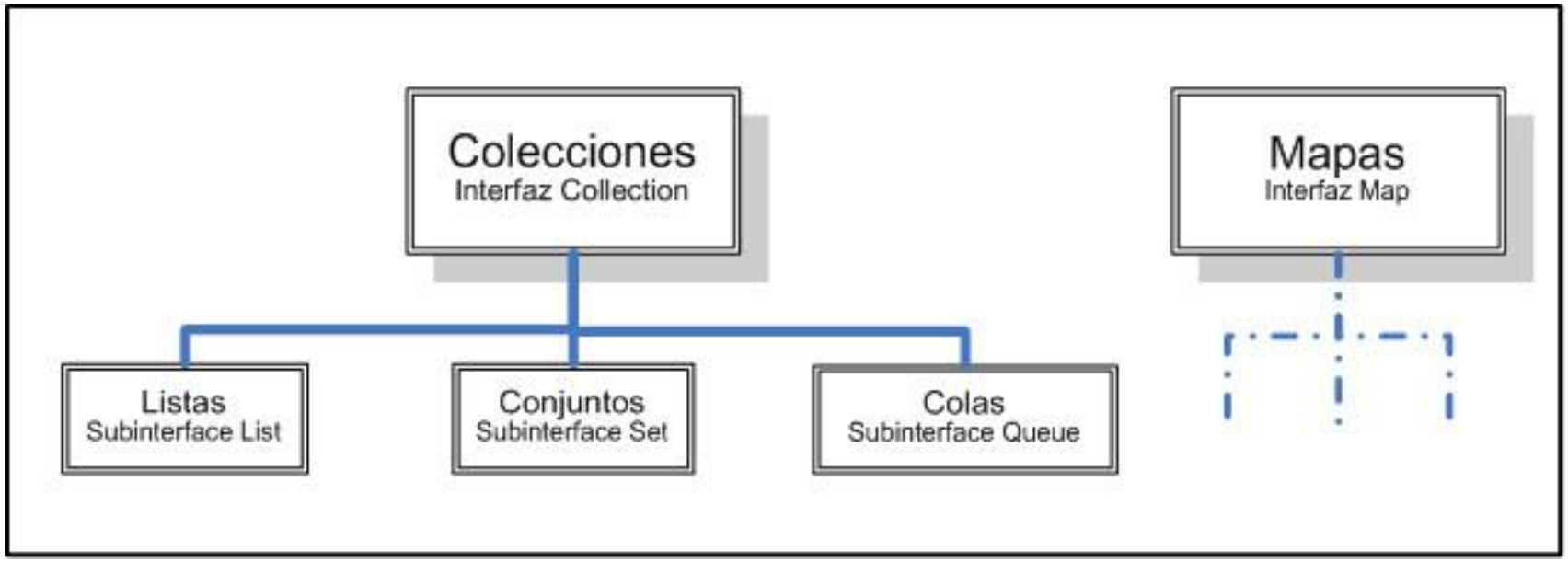
Ventajas con respecto a los arrays:

- Pueden cambiar de tamaño dinámicamente.
- Pueden ir provistas de ordenamiento.
- Se pueden insertar y eliminar elementos.

COLECCIONES

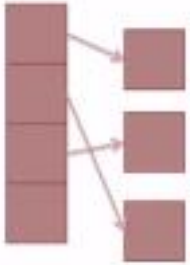
- Los datos que las forman son siempre **objetos** que reciben el nombre de **nodos.**
- Los datos **NO** se localizan de **forma contigua** en la **memoria**
- Estas estructuras dinámicas, según estén organizados sus nodos, se dividen de la siguiente forma:
  1) **Lineales:** listas, conjuntos, pilas y colas.
  2) **No Lineales:** árboles y grafos.

COLECCIONES



COLECCIONES

TIPOS DE COLECCIONES



List

- Lineal
- Posibilidad de orden.
- Con repetidos

Set

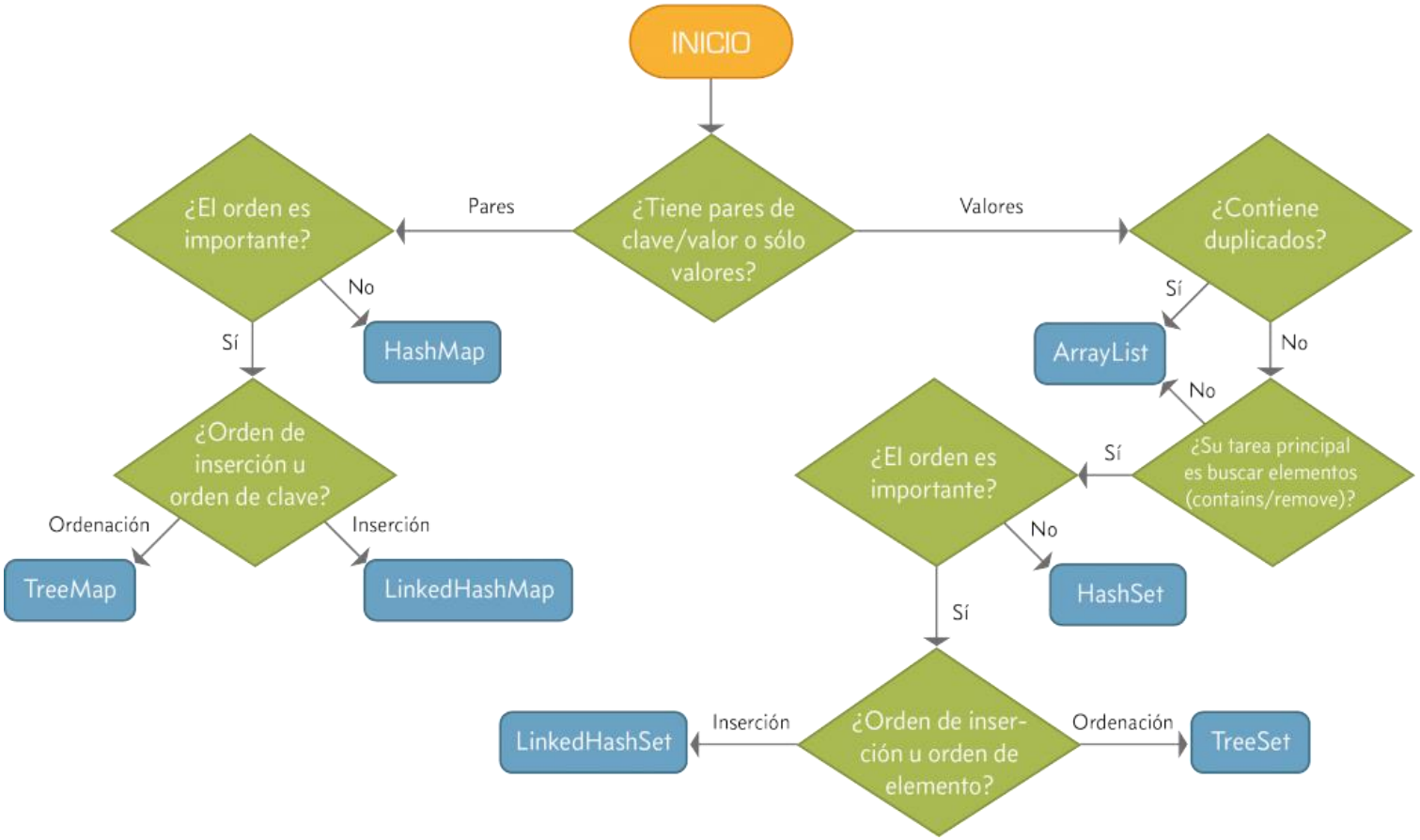
- No soporta duplicados.
- Posibilidad de orden de elementos

Map

- Estructura clave, valor.
- Posibilidad de orden de elementos

COLECCIONES

Diagrama de decisión para uso de colecciones Java



COLECCIONES DINÁMICAS LINEALES

- **Listas:** estructura de datos secuencial en la que los elementos tienen una posición o índice y que permite elementos duplicados.

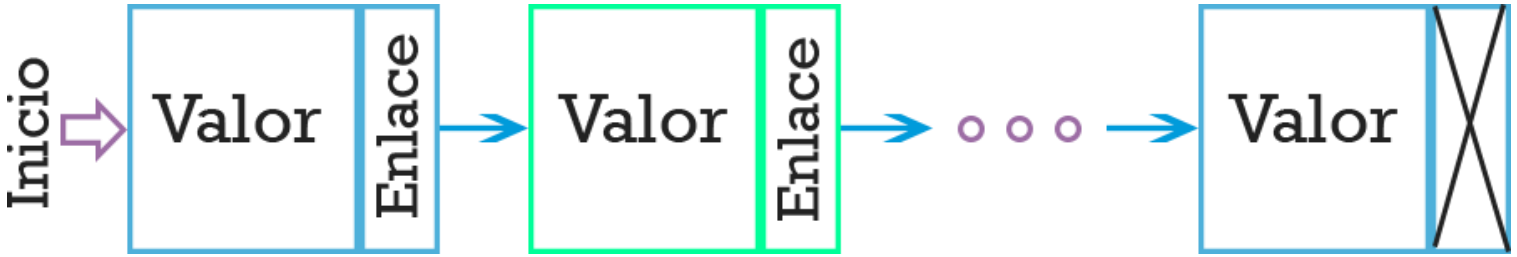
  Estructura en la que los nodos que se van insertando en la colección se pueden colocar donde quieras y se puede acceder a cualquier nodo para poder trabajar en la información que contiene.

  Sus nodos puede tener un campo de enlace o dos dando lugar a listas:

  - **simplemente enlazadas..**
  - **doblemente enlazadas.**

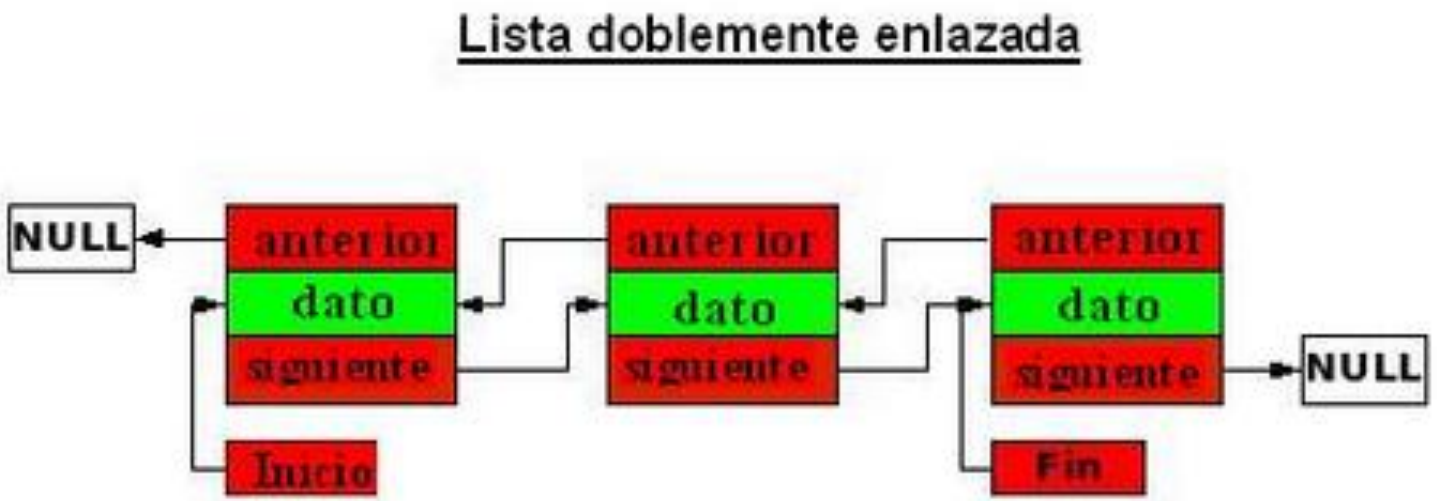
COLECCIONES DINÁMICAS LINEALES
Listas

Simplemente enlazadas. El campo de enlace apunta al siguiente nodo.



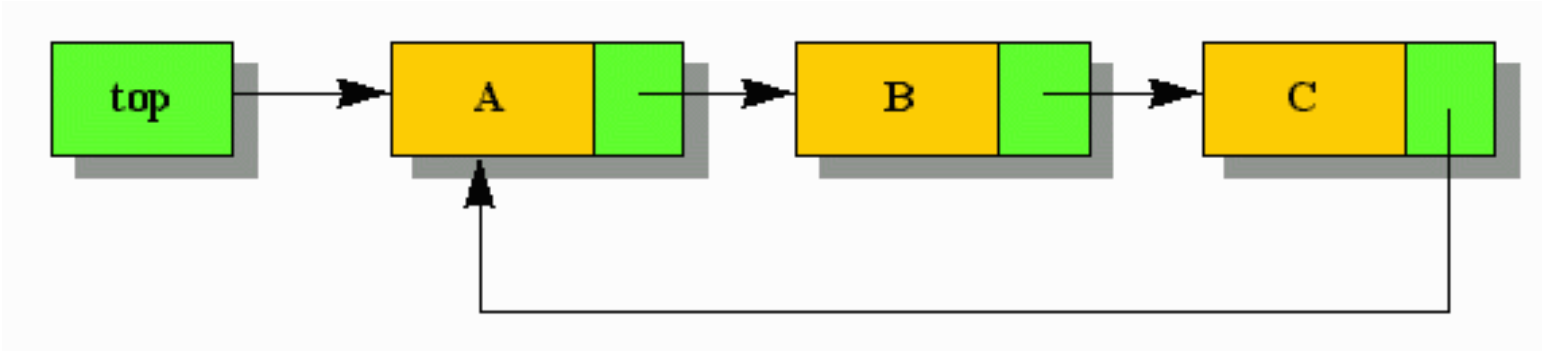
COLECCIONES DINÁMICAS LINEALES: Listas

Doblemente enlazadas. Un C.E. apunta al nodo anterior y el otro al nodo posterior.



COLECCIONES DINÁMICAS LINEALES
Listas

(*) Existen listas circulares: El último nodo apunta al primero.



COLECCIONES DINÁMICAS LINEALES
Pilas

PILAS:

Cuando los nodos que se van insertando en la colección NO se pueden colocar donde quieras.

Estas estructuras también se conocen como LIFO (Last In – First Out).

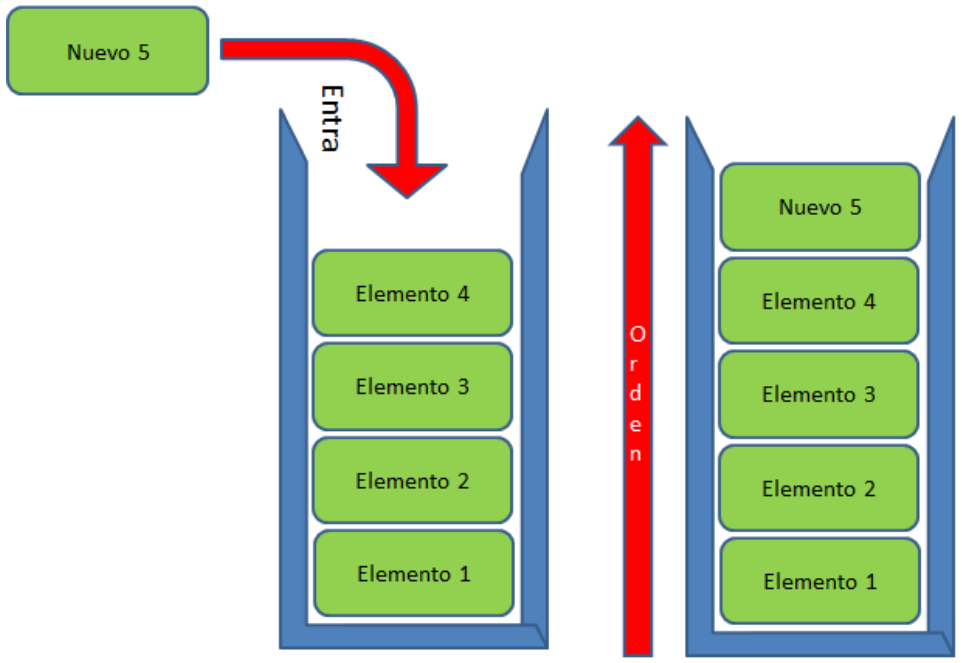
Los nodos nuevos que se insertan se sitúan delante del último nodo que se insertó.

A la hora de acceder a un nodo para trabajar con su información sólo se puede hacer sobre el último nodo que se insertó.

COLECCIONES DINÁMICAS LINEALES
Pilas

PILAS: Entrada de Datos

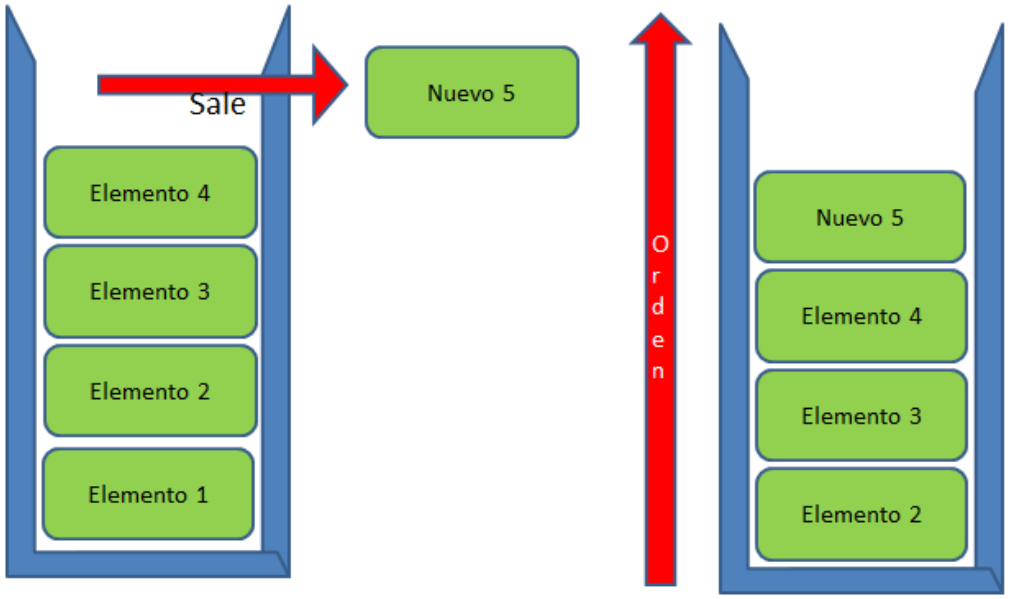
Los nodos nuevos que se insertan se sitúan delante del último nodo que se insertó.



COLECCIONES DINÁMICAS LINEALES
Pilas

PILAS: Salida de Datos

A la hora de acceder a un nodo para trabajar con su información sólo se puede hacer sobre el último nodo que se insertó.



COLECCIONES DINÁMICAS LINEALES
Colas

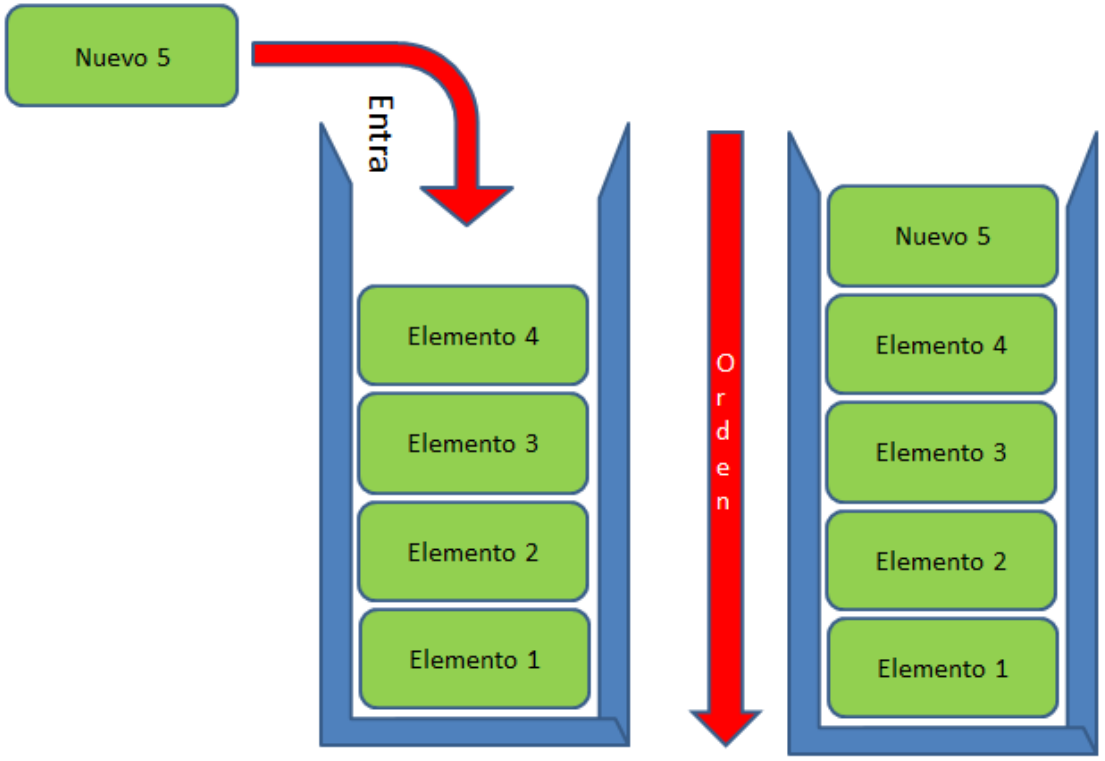
Estructura en la que los nodos que se van insertando en la colección NO se pueden colocar donde quieras.

Estas estructuras también se conocen como FIFO (First In – First Out).

COLECCIONES DINÁMICAS LINEALES
Colas

COLAS: Entrada de Datos.

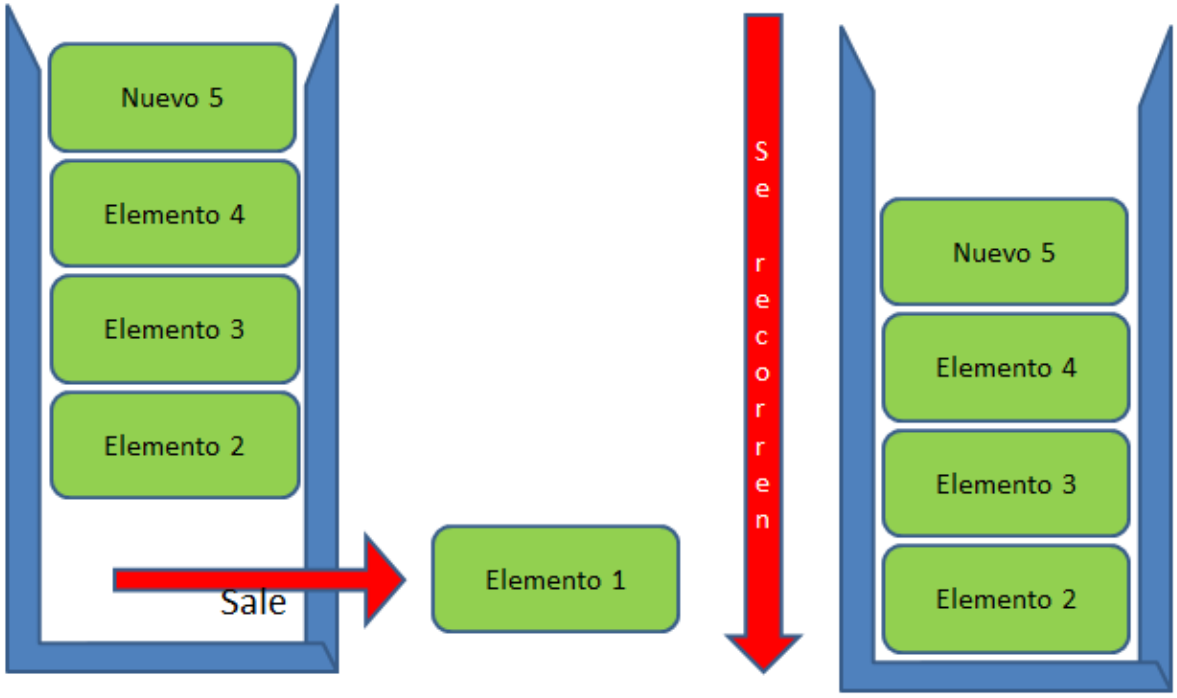
Los nodos nuevos que se insertan se sitúan después del último nodo que se insertó.



COLECCIONES DINÁMICAS LINEALES
Colas

COLAS: Salida de Datos:

A la hora de acceder a un nodo para trabajar con su información sólo se puede hacer al primero que llegó.



COLECCIONES DINÁMICAS NO LINEALES

Maps

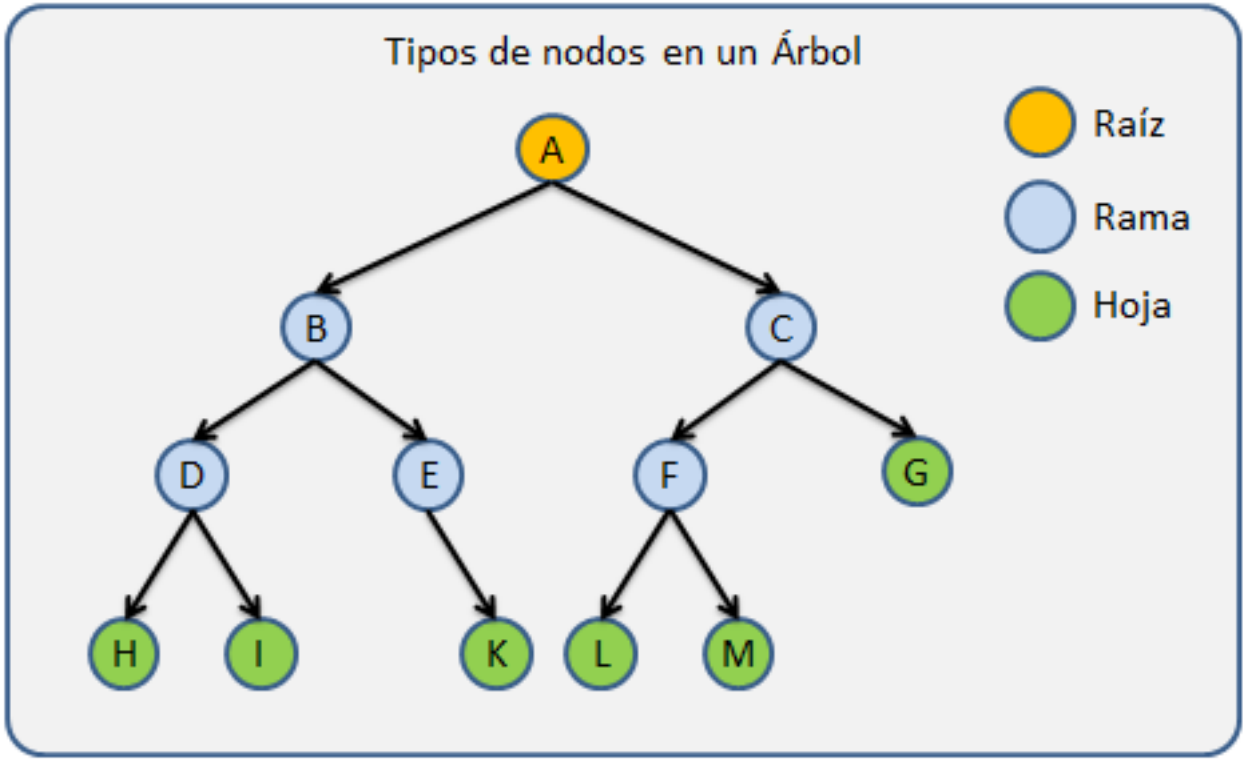
- **Árboles:** Los datos se relacionan formando un árbol invertido.

- **Grafos:** colección en la que los nodos se relacionan de cualquier forma que no sea ninguna de las anteriores.

COLECCIONES DINÁMICAS NO LINEALES

Maps

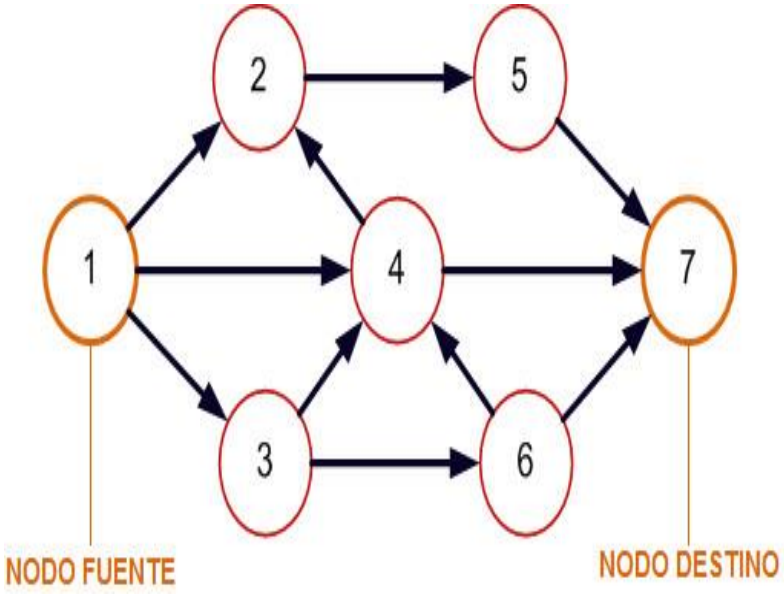
- **Árboles:** Los datos se relacionan formando un árbol invertido.

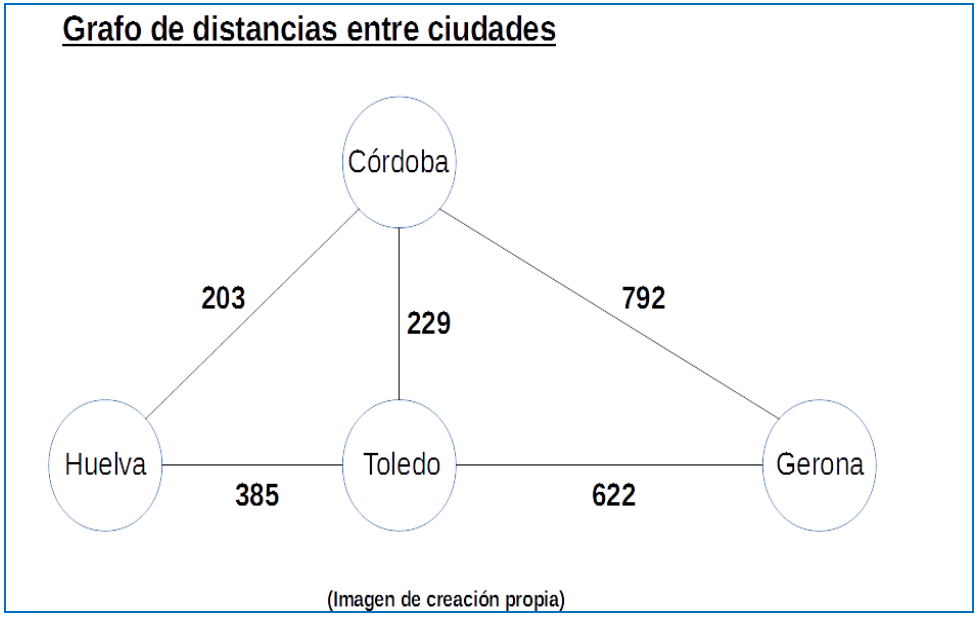


COLECCIONES DINÁMICAS NO LINEALES

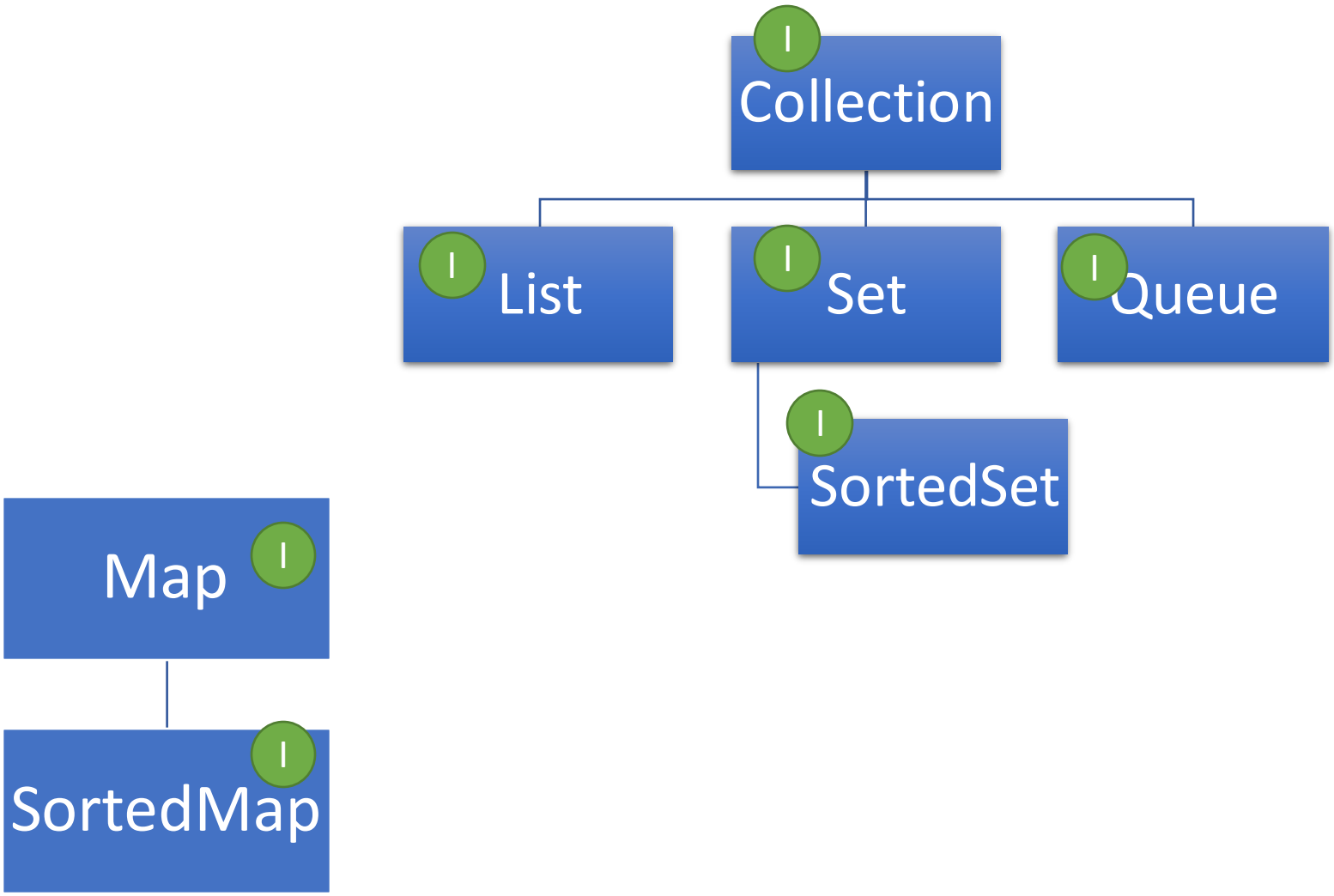
Maps

- **Grafos:** colección en la que los nodos se relacionan de cualquier forma que no sea ninguna de las anteriores





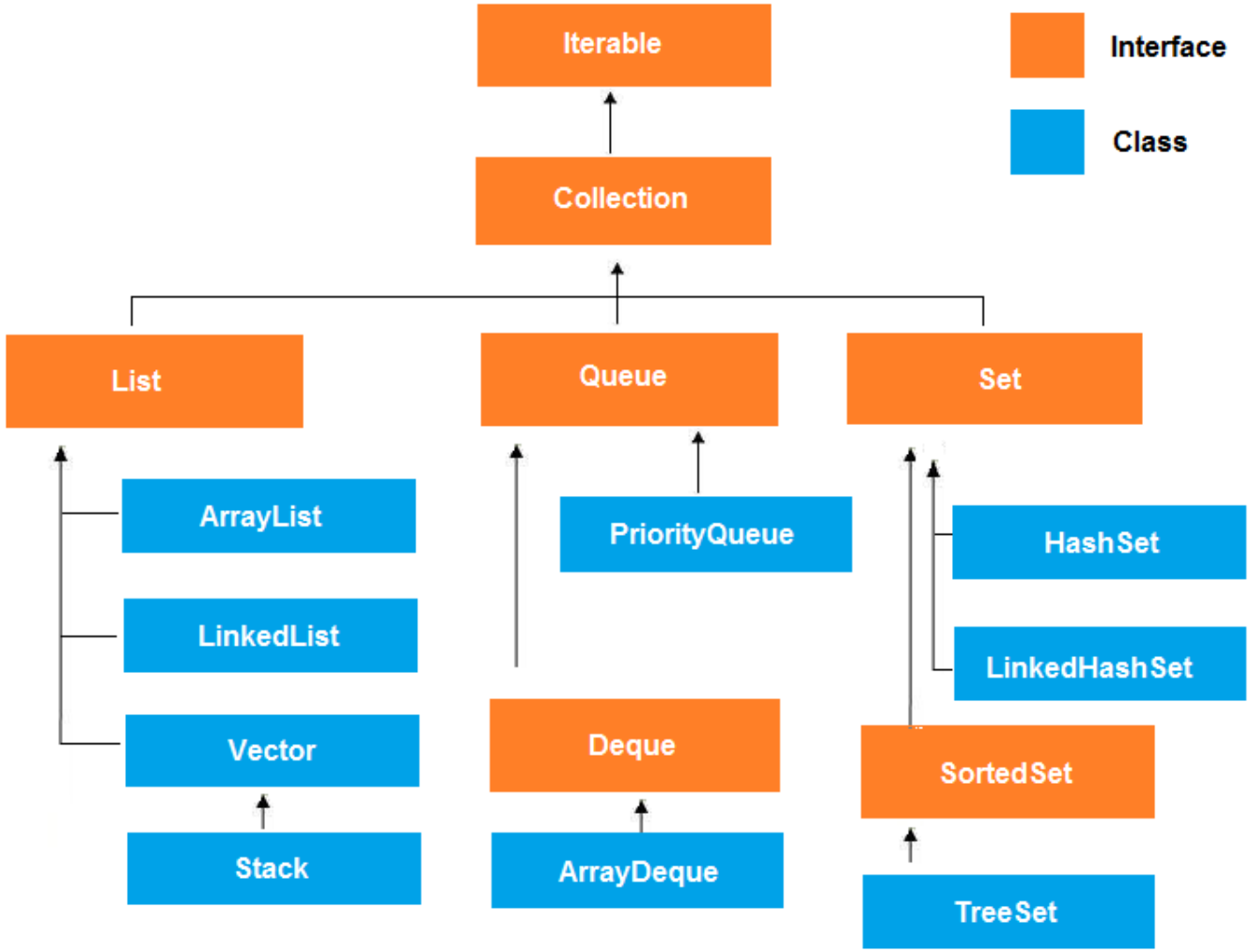
COLECCIONES DINÁMICAS



COLECCIONES DINÁMICAS LINEALES

- List :
  - Puede tener elementos repetidos
  - Los elementos están indexados
  - Permite acceso aleatorio
- Set :
  - No permite elementos repetidos
  - No están ordenados
- Queue :
  - No permite acceso aleatorio
  - Sólo se puede acceder al primer o último elemento
- Map :
  - Permite elementos repetidos
  - Elementos indexados por una clave única

JERARQUÍA DE COLECCIONES

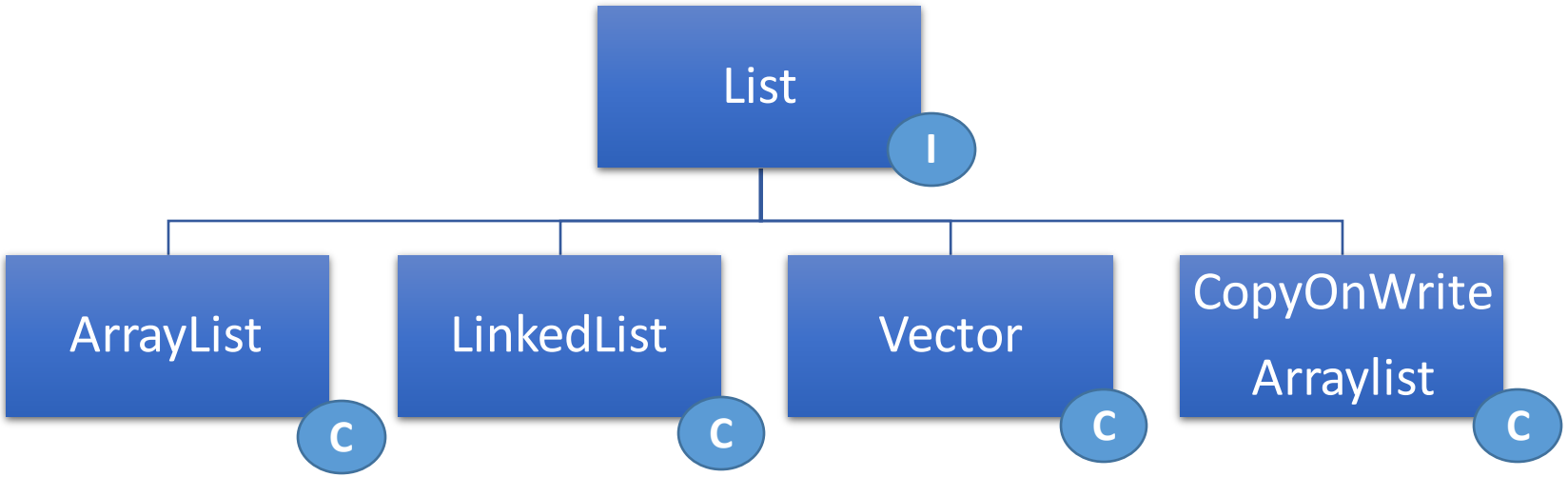


INTERFACE COLLECTION: Métodos

MÉTODOS son comunes a los distintos tipos de colecciones.

boolean add(E e)	Añade un nuevo elemento al final de la lista.
boolean remove(E e):	Elimina la primera ocurrencia del elemento indicado.
boolean contains(E e):	Comprueba si el elemento especificado está en la colección.
void clear():	Elimina todos los elementos de la colección
Int size():	Devuelve el número de elementos en la colección.
boolean isEmpty(Collection<?> C)	Comprueba si la colección está vacía.

INTERFAZ LIST



INTERFAZ LIST

Una **lista** es una colección donde los nodos que lo forman están dispuestos en un cierto **orden**.

A diferencia de una Collection, la lista **guarda información de como están colocados los nodos**, por esta **razón permite acceder a un nodo** concreto por su posición.

ArrayList:

- No tiene un tamaño fijo
- El acceso a sus elementos es rápido, indicando su posición.
- El principal inconveniente es borrar un elemento, requiere tiempo ya que tiene que reorganizar todos los elementos que van detrás del elemento eliminado.
- Mismo inconveniente para insertar un nodo.

LinkedList:

- Listas doblemente enlazadas.
- Cada nodo tiene dos campos de enlace,
- Permiten insertar y borrar nodos fácilmente.

INTERFAZ LIST. Métodos I

Acceso posicional:

Object **get**(int **indice**); // Obtiene el elemento almacenado en la colección en la posición que indica el índice.

Object **set**(int **indice**, Object **elemento**); // **Sustituye** el elemento número índice por uno nuevo. Devuelve además el elemento antiguo ese elemento.

void **add**(int **indice,** Object **elemento**); // **Añade** el elemento indicado en la posición índice de la lista

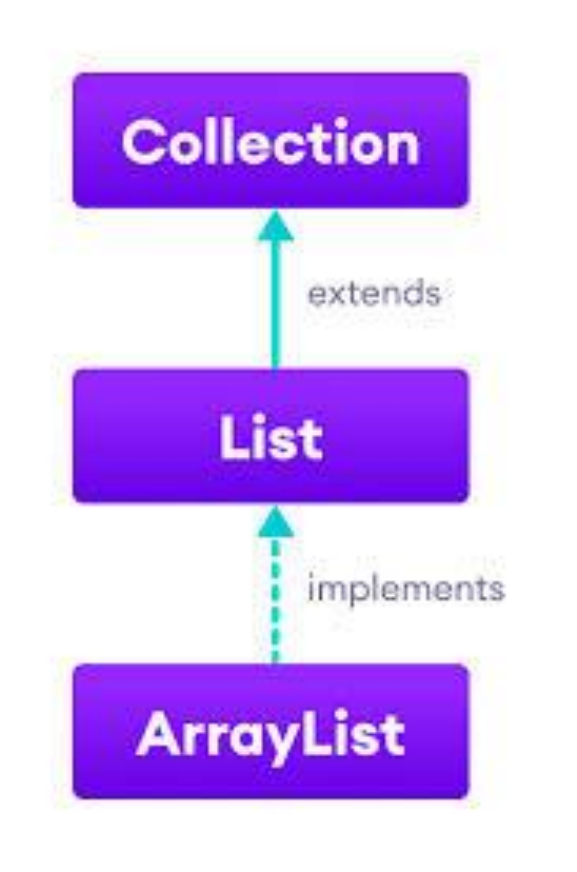
Object **remove**(int **indice**); // Elimina el elemento cuya posición en la colección la da el parámetro índice

INTERFAZ LIST. Métodos II

Búsqueda:

int **indexOf**(Object **elemento**); // Devuelve la posición del elemento. Si no lo encuentra, devuelve -1

int **lastIndexOf**(Object **elemento**); // Devuelve la posición del elemento comenzando a buscarle por el final. Si no lo encuentra, devuelve -1



ArrayList

ARRAYLIST: constructores

Posee tres **constructores**:

- **ArrayList()** : Constructor por defecto. Simplemente crea un ArrayList vacio.
- **ArrayList(int capacidadInicial).** Crea una lista con una capacidad inicial indicada.
- **ArrayList(Collection c):** Crea una lista a partir de los elementos de la colección indicada.

ARRAYLIST

Paso 1: Importar paquete útil.

import.java.util.*
import java.util.ArrayList

Paso 2: Declarar el ArrayList.

ArrayList <tipo_dato> “nombre del ArrayList” = new ArrayList<tipo_dato>();

Ejemplos:

ArrayList <Empleado> listaEmpleados=new ArrayList<Empleado>()
ArrayList <Empleado> listaEmpleados=new ArrayList<>()
ArrayList <String> listaNombres=new ArrayList<>()

ARRAYLIST

Sus elementos empiezan a numerarse desde O igual que en las tablas.

add(elemento)	añadir un elemento
add(numero,elemento)	añade un elemento en la posición indicada
size( )	indica el tamaño
get(numero)	devuelve el elemento que se encuentra en esa posición
contains(elemento)	nos indica si ese elemento existe. Devuelve un booleano.
indexOf(elemento)	devuelva la primera ocurrencia donde aparece
lastIndexOf(elemento)	devuelve la última ocurrencia donde aparece
remove(numero)	borra el elemento indicado
remove(elemento)	borra la primera ocurrencia del elemento especificado.
clear( )	borra el ArrayList
isEmpty( )	nos indica si está vacío devuelve un true
set(numero,elemento)	modifica el valor de la posición indicada

ARRAYLIST: añadir elementos

Método add

boolean add (e E)
void add (int index, E elemento)

Ej1: Empleado emp1 = new Empleado(“pepe”,2500);
listaEmpleados.add(emp1);
Ej2: listaEmpleados.add(new Empleado(“pepe”,2500));

ARRAYLIST: añadir elementos

•**Método set** :
Reemplaza el elemento existente en una posición determinada por el elemento enviado.

set(int index, **E** element)

Ejem: listaEmpleados.set(4, new Empleado(“pepe”,2500));

ARRAYLIST: recuperación/lectura de elementos

Método get:

get (int Index):

Devuelve la referencial al objeto.

Si queremos obtener el contenido utilizar un método de la clase para obtenerlo.

Ejem: String s = listaEmpleados.get (4).toString()

ArrayList: Recorrido

Bucle for-convencional

```
for (int i=0; i<listaEmpledo.size();i++){
        Empleado e = listaEmpleado.get(i);
}
```

Bucle for-each

Sirve para recorrer una colección de objetos de forma secuencial de principio a fin.

for (tipo <var_iteración> : colección) {bloque de instrucciones}

Ejemplo:

```
for (Empleado var : listaEmpleados){
                System.out.println (var.toString());
        }
```

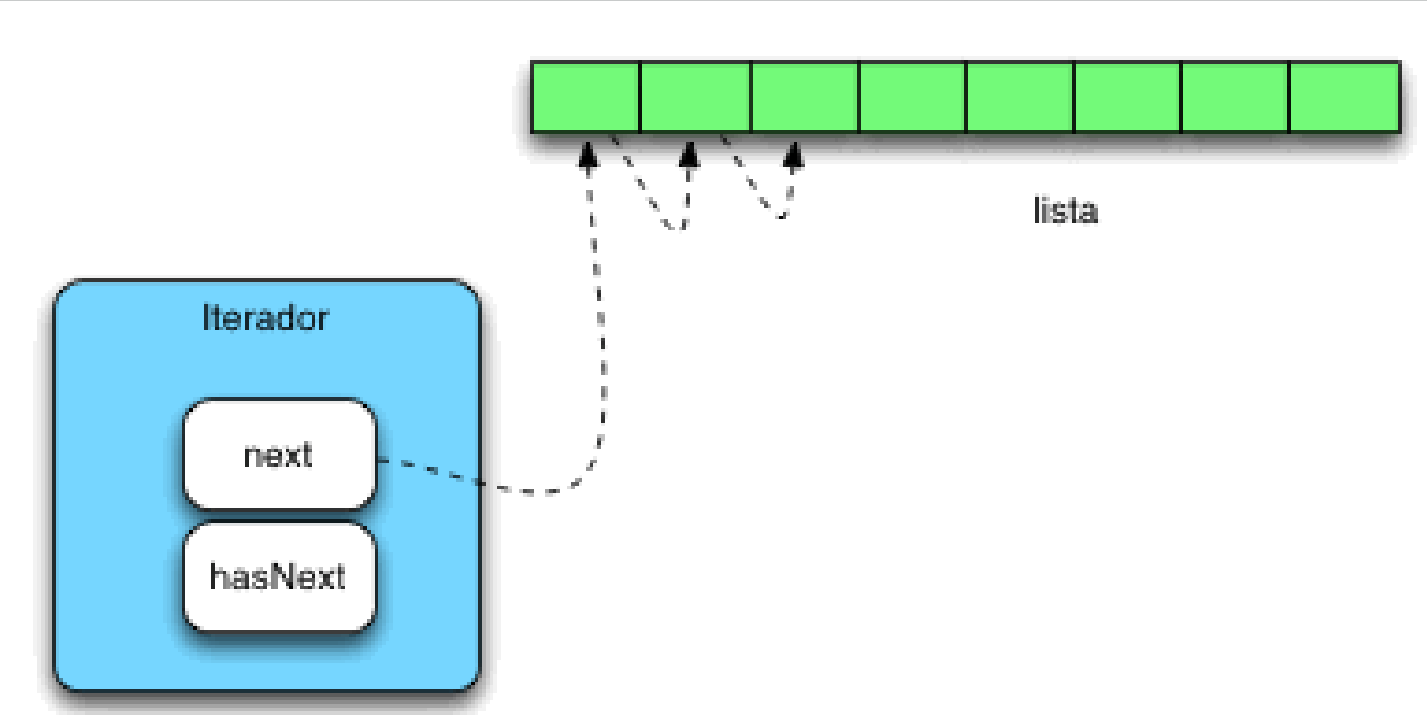
ArrayList

Copiar un ArrayList en un array estático.:
método toArray

T [ ] toArray(T[ ])

Ejem:

```
Empleado arrayEstatico[] = new Empleado[listaEmpleados.size()];
ListaEmpleado.toArray(arrayEstatico)
```



Iterator

ITERATOR

Iterator es un elemento que nos permite recorrer una lista de elementos para realizar las operaciones necesarias para ello.

Crear el iterador:

Iterator <tipo_Dato> “nombre iterador”=“nombre_coleccion”.iterator();

Método iterator() : método de las clases que implementan listas que nos devuelve un objeto de tipo Iterator.

Ejemplo :

```
Iterator <Empleado> miIterador = listaEmpleados.iterator();
```

ListIterator : Métodos

Modificador y tipo	Método y descripción
void	**add** (E e) Inserta el elemento especificado en la lista (operación opcional).
boolean	**hasNext** () Devuelve true si este iterador de lista tiene más elementos al atravesar la lista en la dirección de avance.
boolean	**hasPrevious** () Devuelve true si este iterador de lista tiene más elementos al atravesar la lista en la dirección inversa.
E	**next** () Devuelve el siguiente elemento de la lista y avanza la posición del cursor.
int	**nextIndex** () Devuelve el índice del elemento que sería devuelto por una llamada subsiguiente a **next()** .
E	**previous** () Devuelve el elemento anterior en la lista y mueve la posición del cursor hacia atrás.
int	**previousIndex** () Devuelve el índice del elemento que sería devuelto por una llamada posterior a **previous()** .
void	**remove** () Elimina de la lista el último elemento devuelto por **next()** o **previous()** (operación opcional).
void	**set** (E e) Reemplaza el último elemento devuelto por **next()** o **previous()** con el elemento especificado (operación opcional).

ITERATOR

Algunos de sus métodos son:

boolean hasNext():
Indica si hay un elemento siguiente (y así evita la excepción).

Object next():
Obtiene el siguiente objeto de la colección. Si se ha llegado al final de la colección y se intenta seguir, da lugar a una excepción de tipo:

NoSuchElementException (que deriva a su vez de RunTimeException)

void remove(); // Elimina el último elemento devuelto por next()

ITERATOR: Ejemplos

```
// Declaramos el Iterador e imprimimos los Elementos del ArrayList

 ArrayList<String> listaMarcasCoches = new ArrayList<String>();

Iterator<String> miIterator = listaMarcasCoches.iterator();

while(miIterator.hasNext()){
    String elemento = miIterator.next();
    System.out.print(elemento+" / ");
}
```

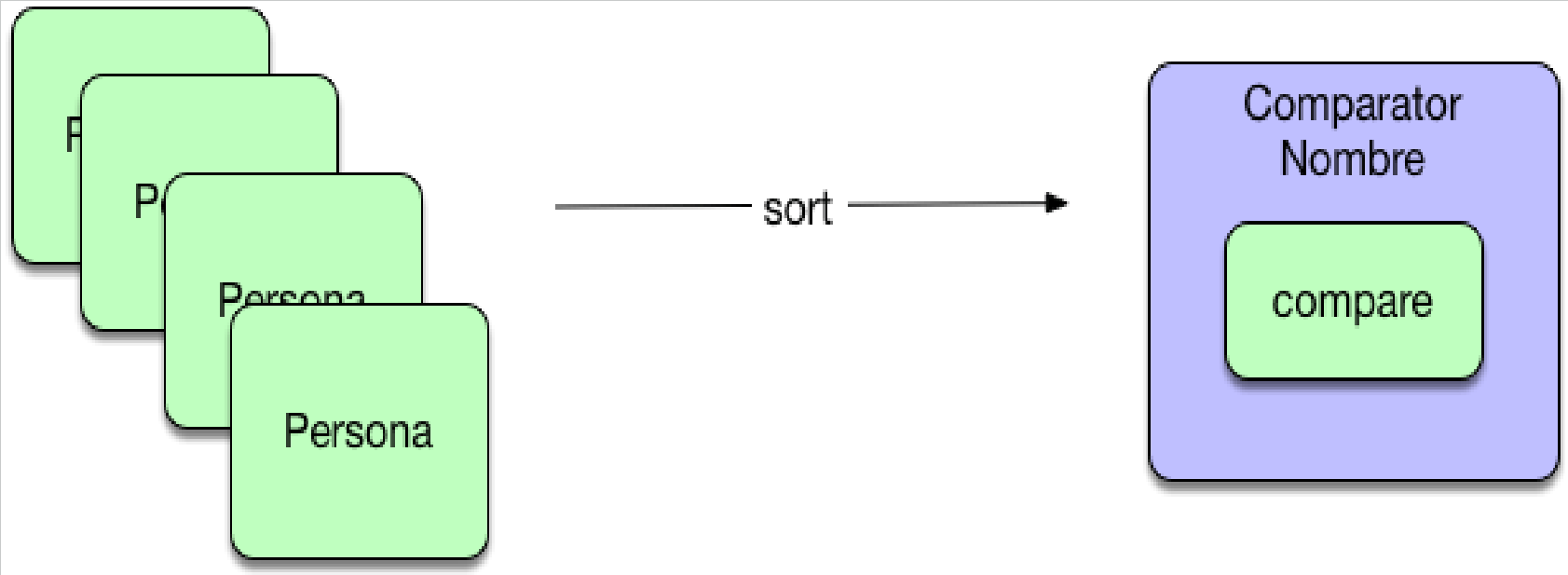
```
for (Iterator iterator = lC.iterator(); iterator.hasNext();) {
    Correo correo = (Correo) iterator.next();

}
```

EJERCICIO LinkedList e Iterator

Haz un programa que genere dos LinkedList :

- Paises : añadiremos 5 paises
- Capitales : añadiremos 5 capitales en el orden que corresponda a los países de la lista Paises.

1) Programar el proceso necesario para insertar los elementos de la lista Capitales en la de Paises. Cada capital se tiene que insertar a continuación del país que le corresponda. (utilizar hasNext(), next() y listIterator.add().
2) Eliminar de la lista Capitales los elementos que ocupan las posiciones pares.
3) Elimina de la lista Paises los elementos que quedan en la lista Capitales (removeAll)



Ordenación de listas

Utilización del método sort

```
public static void main(String[] args) {
    miLista= new ArrayList<>();

    miLista.add(new Persona("Maria","Lopez","8493893"));
    miLista.add(new Persona("Ana","Alvarez","8658453"));
    miLista.add(new Persona("Juan","Cansado","5689874"));
    miLista.add(new Persona("Luis","Vazquez","5691232"));

    miLista.sort(new Comparadores());

    miLista.forEach(System.out::println);

}

public static class Comparadores implements Comparator <Persona> {

        @Override
        public int compare(Persona p1, Persona p2) {
            return p1.getApellidos().compareTo(p2.getApellidos());
        }
}
```

LinkedList

LinkedList

Colección que permite elementos repetidos, tiene orden en sus elementos.

Ventaja:

- Más eficiente en la inserción y borrado de elementos en posiciones aleatorias.

Inconveniente:

- Desplazamiento en la lista más lento que ArrayList.

LinkedList

Paso 1: Importar paquete útil.

Import.java.util.*

Paso 2: Declarar el ArrayList.

LinkedList <tipo_dato> “nombre del LinkedList” = new LinkedList<tipo_dato>();

Ejemplo : LinkedList <Empleado> listaEmpleados=new LinkedList <Empleado>()

LinkedList: Métodos

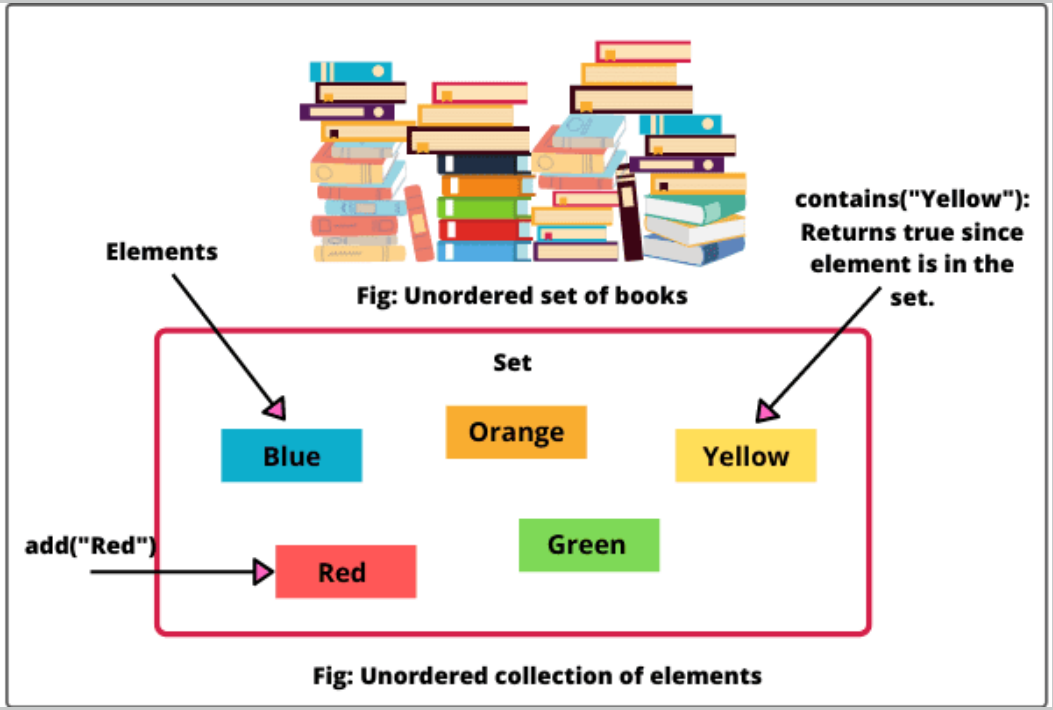
Sus elementos empiezan a numerarse desde 0, igual que en las tablas.

Métodos (ver API de Java) :

https://docs.oracle.com/javase/8/docs/api/

EJERCICIO LinkedList y ArrayList

Haz un programa que pruebe insertar 10000000 objetos de la clase Alumno dentro de un ArrayList y de un LinkedList y compruebe la diferencia en el tiempo que tarda en hacer la operación en cada una de las estructuras.



Conjunto - Set

Colecciones tipo Set

Colección que agrupa un conjunto de elementos que no permiten repetidos y que no admite un acceso posicional.

- No puede contener repetidos.
- Propone tres implementaciones: **HashSet**, **TreeSet** y **LinkedHastSet**.
- ***HashSet*** es la más eficiente, pero no nos asegura nada sobre el orden.
- ***TreeSet*** utiliza un árbol Red-Black, ordena según el valor.
- ***LinkedHashSet*** es un HashSet ordenado por orden de inserción.

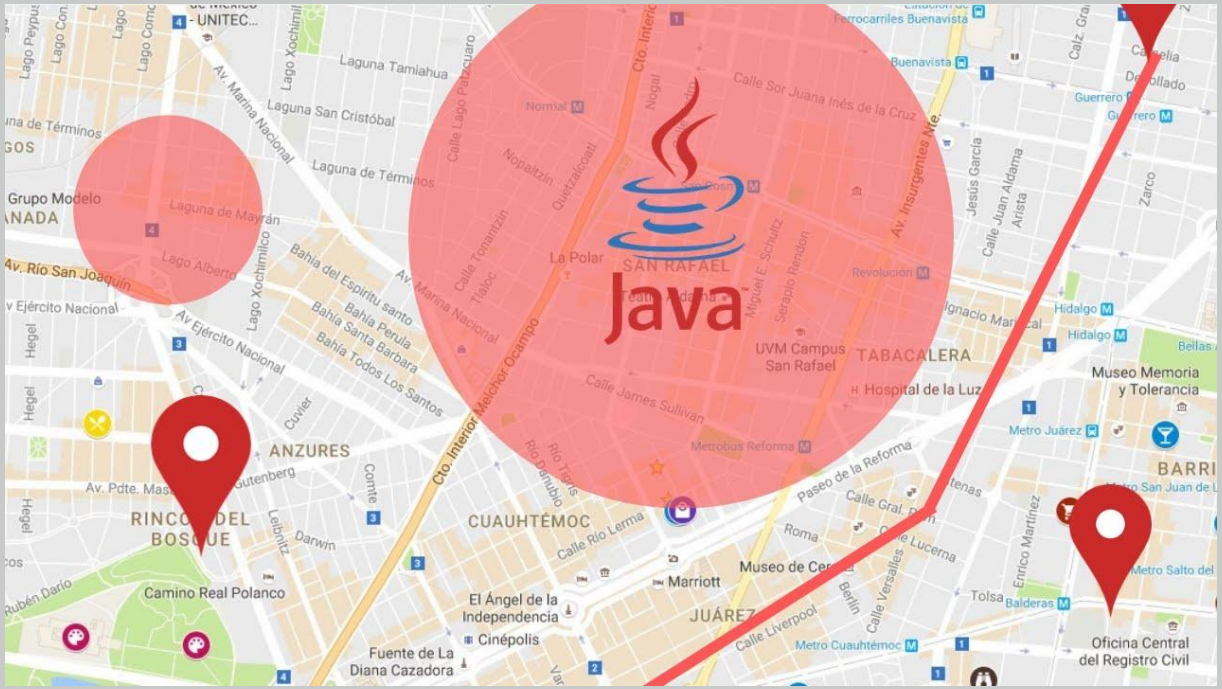
Colecciones tipo Set: métodos

Nombre	Uso
add	Añade un elemento al conjunto, si aun no está contenido
addAll	Añade todos los elementos de la colección pasada como argumento si es que aun no están presentes.
clear	Elimina todos los elementos del conjunto.
contains	Comprueba si un elemento está o no en el conjunto
isEmpty	Verifica si el conjunto está vacío
remove	Elimina un elemento del conjunto
size	Devuelve el número de elementos de la lista
toArray	Devuelve la lista como un array

Respecto a las listas, perdemos el método de acceso posicional get(posición)

Colecciones tipo Set: métodos

```
public class EjemploSet {
    public static void main(String[] args) {
        //Set <Persona> juntaDirectiva = new HashSet<>();
        //Set <Persona> juntaDirectiva = new TreeSet<>(); //nos obliga a implementar Comparable
        Set <Persona> juntaDirectiva = new LinkedHashSet<>();

        juntaDirectiva.add (new Persona (" 12345678A","Pepe","Pérez", LocalDate.of(1990,1,2 )));
        juntaDirectiva.add (new Persona (" 23456789B "," Juan ", " Martínez " , LocalDate.of( 1991,2,3 )));
        juntaDirectiva.add (new Persona (" 34567890C "," Ana ", " Ramírez " , LocalDate.of( 1992,3,4)));
        Persona p=new Persona (" 45678901D "," María ", " López " , LocalDate.of( 1993,4,5));
        juntaDirectiva.add (p);

        // Si tratamos de añadir un elemento repetido ...
        juntaDirectiva.add (p);

        // Comprobamos que al listarlos todos, no aparece duplicado
        for( Persona p1 : juntaDirectiva)
            System.out. println (p1);
```



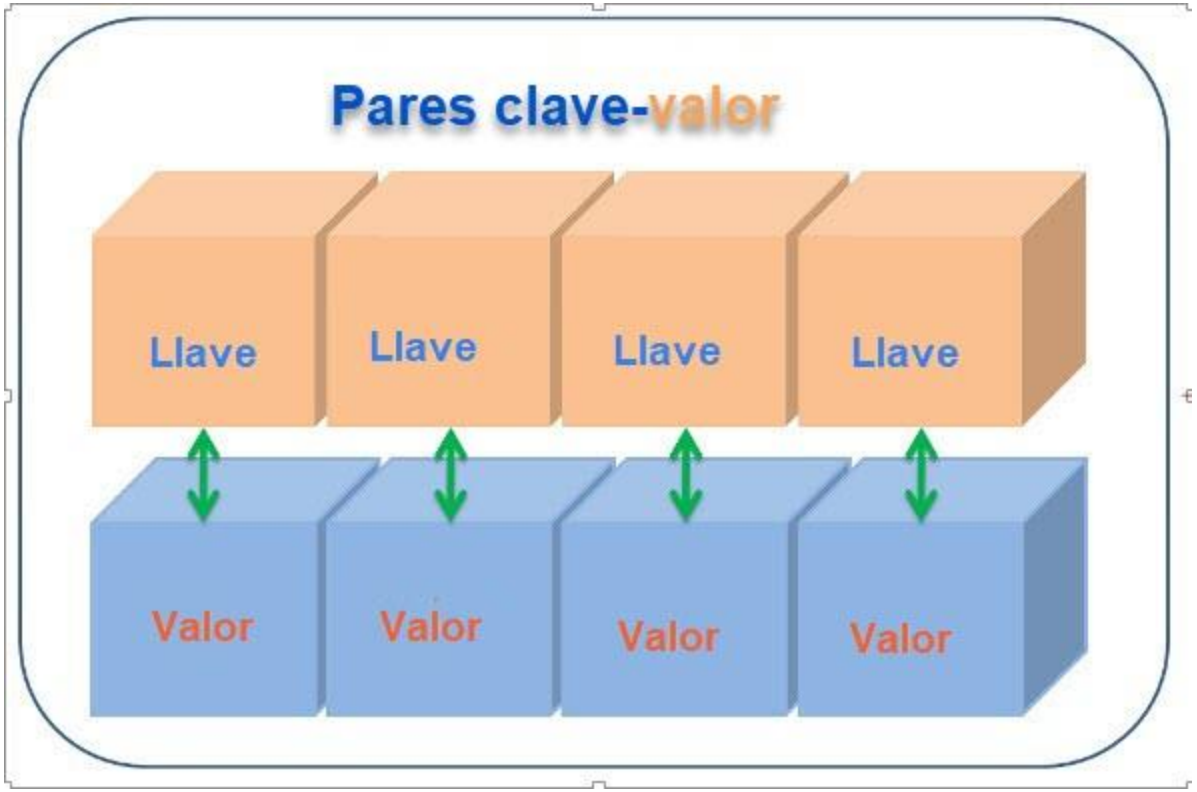
Collecciones tipo Map

Colecciones tipo Map

INTERFAZ MAP

- No es un subtipo de Collection (List y Set sí que lo son).
- Cada elemento tiene estructura clave, valor.
- La clave sirve para acceder directamente al valor.
- Las implementaciones son ***HashMap***, ***TreeMap*** y ***LinkedHashMap***. Las consideraciones son análogas a Set.

Colecciones tipo Map



Colecciones tipo Map

OPERACIONES CON MAP

Nombre	Uso
clear	Elimina todos los elementos del *diccionario*.
containsKey	Comprueba si una clave está presente en el *diccionario*.
containsValue	Comprueba si un valor está presente en el *diccionario*.
get	Devuelve el valor asociado a una clave.
isEmpty	Verifica si el conjunto está vacío
keySet	Devuelve un Set con todas las claves.
put	Permite insertar un par clave, valor
remove	Elimina un elemento del conjunto
size	Devuelve el número de elementos de la lista
values	Devuelve un Collection con los valores

(*) put, reemplaza a add

Colecciones tipo Map

```
//Map <String, Persona> miAgenda = new HashMap<>();
//Map <String, Persona> miAgenda = new TreeMap<>();
Map <String, Persona> miAgenda = new LinkedHashMap<>();


miAgenda.put("564897845",new Persona (" 12345678A","Pepe","Pérez", LocalDate.of(1990,1,2 )));
miAgenda.put ("654323121",new Persona (" 23456789B "," Juan ", " Martínez " , LocalDate.of( 1991,2,3 ))
miAgenda.put ("987652541",new Persona (" 34567890C "," Ana ", " Ramírez " , LocalDate.of( 1992,3,4)));
Persona p=new Persona (" 45678901D "," María ", " López " , LocalDate.of( 1993,4,5));
miAgenda.put("789456123",p);

miAgenda.put ("987652541",new Persona (" 34567890","Luisa", "Sánchez" , LocalDate.of( 1992,3,4)));

for (String clave : miAgenda.keySet()) {
    System.out.printf("%s %s %n",clave, miAgenda.get(clave));

}
```