IES DR LLUÍS SIMARRO

ENSENYAMENT PÚBLIC

XÀTIVA

CFGS Desarrollo de aplicaciones multiplataforma

Módulo profesional: Programación

GENERALITAT VALENCIANA

Conselleria d'Educació, Investigació, Cultura i Esport

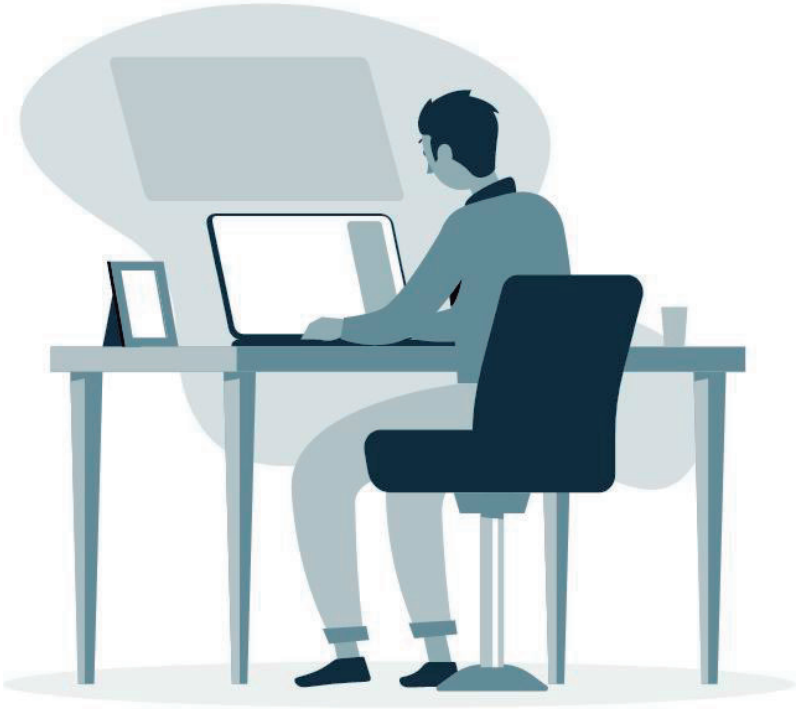
GOBIERNO DE ESPAÑA

MINISTERIO DE EDUCACIÓN Y FORMACIÓN PROFESIONAL

Unió Europea

Fons Social Europeu

L'FSE inverteix en el teu futur



Material elaborado por:

Edu Torregrosa Llácer

(aulaenlanube.com)

Estructuras de datos,colecciones, APIs y programación funcional en JAVA

1. Estructuras de datos
    a. Pila
    b. Cola
    c. Lista
    d. Árbol
    e. Tabla hash
    f. Grafo
2. Iteradores y comparadores
3. Colecciones en JAVA
    a. Set
    b. List
    c. Map
4. Clases y paquetes
5. Documentación y uso de APIs en JAVA
6. Interfaces funcionales y expresiones lambda

Estructuras de datos y colecciones en JAVA

Una estructura de datos es una forma de organizar y almacenar datos en la memoria. Las estructuras de datos están diseñadas para optimizar la eficiencia en la manipulación y acceso a los datos. Cada estructura de datos tiene sus propias propiedades y métodos para manipular y acceder a los datos almacenados. La elección de un tipo de estructura u otra depende del tipo de algoritmo que vayamos a diseñar.

Las estructuras de datos están diseñadas con el objetivo de buscar la máxima eficiencia posible para un tipo de operación en concreto. Por ejemplo, hay estructuras muy eficientes a la hora de recuperar información pero lentas a la hora de modificar. Y otras muy eficientes para modificar, pero lentas para acceder a los datos.

Por otro lado, las colecciones en JAVA son **interfaces** que representan una agrupación de objetos y que utilizan distintas estructuras de datos para poder almacenar las colecciones de datos. Algunas de dichas interfaces son **List**, **Set**, y **Map**.

A partir de dichas interfaces, JAVA proporciona distintas clases que implementan las interfaces, por ejemplo la clase **ArrayList** implementa la interface **List**.

Estructuras de datos

- **Pilas**: estructura de datos que permite almacenar una colección de elementos en un orden específico, tipo FIFO, primero en entrar, primero en salir.
- **Colas**: estructura de datos que permite almacenar una colección de elementos en un orden específico, tipo LIFO,último en entrar, primero en salir.
- **Listas**: estructura de datos que permite almacenar una colección de elementos en orden. Los elementos de una lista pueden ser accedidos secuencialmente.
- **Árboles**: cada elemento se conecta a uno o más elementos que están por debajo de él en la jerarquía. Se suelen utilizar si necesitamos almacenar los datos ordenados. Existen distintos tipos de árboles, uno es los más típicos es el árbol binario de búsqueda (BST)
- **Grafos**: consiste en un conjunto de vértices (nodos) y un conjunto de aristas (conexiones) que los conectan. Se utilizan en la representación de redes de relaciones complejas.
- **Tablas Hash**: permite el acceso eficiente a los elementos mediante el uso de una función hash. Dicha función, convierte una clave en un índice en la tabla hash. Se utilizan en la búsqueda de valores en grandes conjuntos de datos.

Estructuras de datos: Pila

Una pila es una estructura de datos lineal que sigue el principio de "último en entrar, primero en salir" (LIFO, por sus siglas en inglés). Esto significa que el último elemento que se agregó a la pila es el primer elemento que se elimina de la pila. Tiene dos operaciones principales:

- **push** (empujar): agrega un nuevo elemento a la pila.
- **pop** (sacar): elimina el último elemento agregado.

```
push(1);
push(2);
push(3);
pop();
push(4);
pop();
```

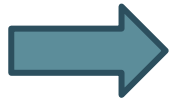
Las pilas son útiles en situaciones en las que se necesitan realizar operaciones en un orden específico, como en la evaluación de expresiones aritméticas. También se utilizan en la implementación de algoritmos recursivos y en la gestión de llamadas a funciones en una pila de llamadas.

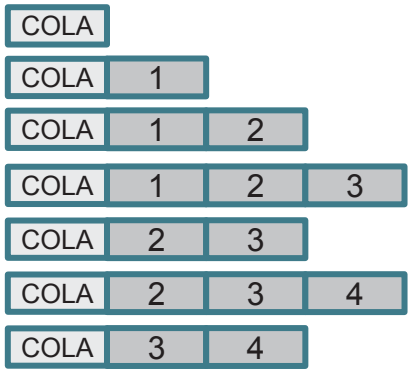
Estructuras de datos: Cola

Una cola es una estructura de datos lineal que sigue el principio de "primero en entrar, primero en salir" (FIFO, por sus siglas en inglés). Esto significa que el primer elemento que se agregó a la cola es el primer elemento que se elimina de la cola.

- **enqueue** (encolar): agrega un nuevo elemento al final de la cola.
- **dequeue** (desencolar): elimina el primer elemento de la cola.

```
enqueue(1);
enqueue(2);
enqueue(3);
dequeue();
enqueue(4);
dequeue();
```





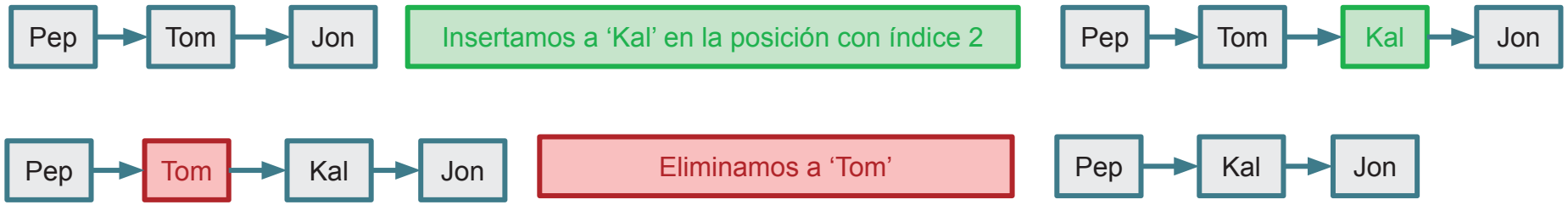
Las colas son útiles en situaciones en las que se necesitan procesar elementos en un orden específico, como en la gestión de tareas en un sistema operativo. También se utilizan en la gestión de recursos compartidos, como en la impresión de documentos en una impresora compartida, la típica cola de impresión.

Estructuras de datos: Lista

Una lista es una estructura de datos lineal que consiste en una colección de elementos que se organizan en un orden específico. Cada elemento de la lista está compuesto por dos partes: un valor y un puntero que apunta al siguiente nodo de la lista.

Las listas se pueden implementar de diferentes maneras, siendo las más comunes las listas enlazadas y las listas doblemente enlazadas. En las listas enlazadas, cada nodo contiene un puntero que apunta al siguiente nodo de la lista, mientras que en las listas doblemente enlazadas cada nodo contiene dos punteros, uno que apunta al nodo anterior y otro que apunta al nodo siguiente.

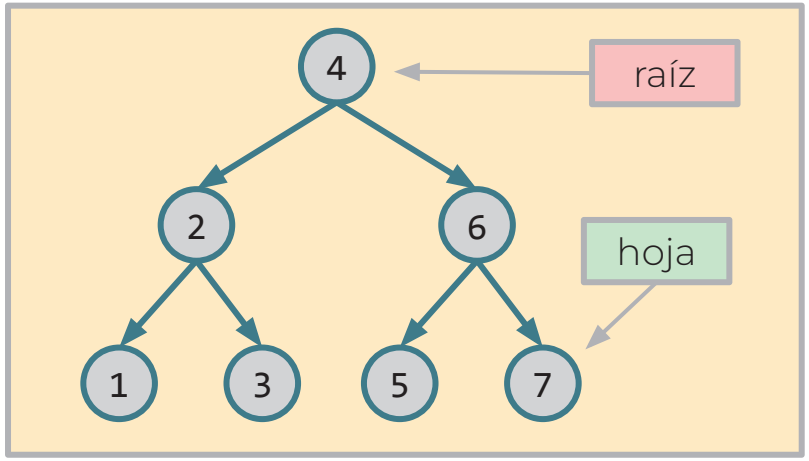
Las listas permiten agregar y eliminar elementos de la colección en cualquier parte de la lista.



Estructuras de datos: Árbol

Un árbol es una estructura de datos no lineal que consta de nodos conectados por aristas. Cada nodo tiene un valor o elemento y cero o más nodos hijos, que son otros nodos del árbol. El nodo sin nodos padres se conoce como raíz del árbol, y los nodos sin hijos se conocen como hojas. Los árboles se pueden clasificar en diferentes tipos según sus propiedades, como los árboles binarios de búsqueda (2 nodos hijo como máximo)

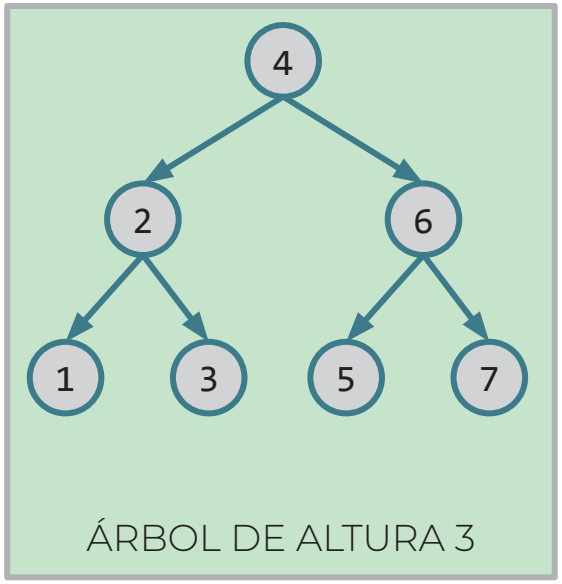
La forma más común de implementar un árbol es mediante punteros. Cada nodo del árbol se representa como un objeto que contiene un valor y punteros a sus nodos hijos. La estructura de punteros permite recorrer el árbol y acceder a sus nodos y elementos de manera eficiente.



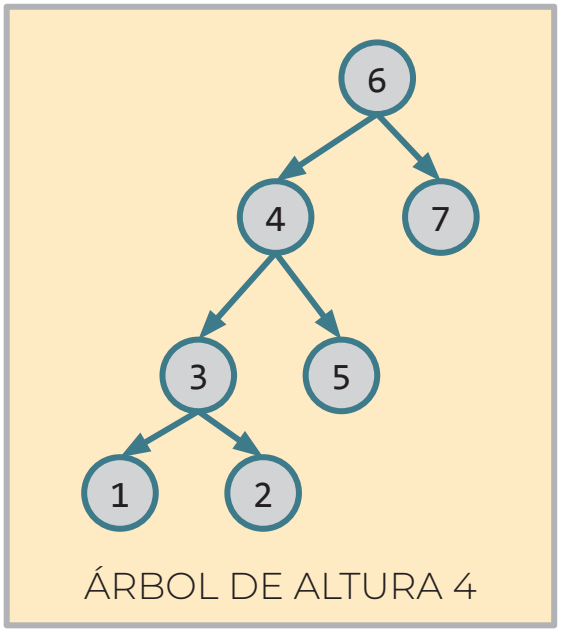
Los árboles se utilizan cuando se requiere una estructura de datos dinámica que pueda manejar grandes conjuntos de datos de manera eficiente. La elección del tipo de árbol depende de los requisitos específicos del problema y del conjunto de datos que se maneje.

Estructuras de datos: Árbol

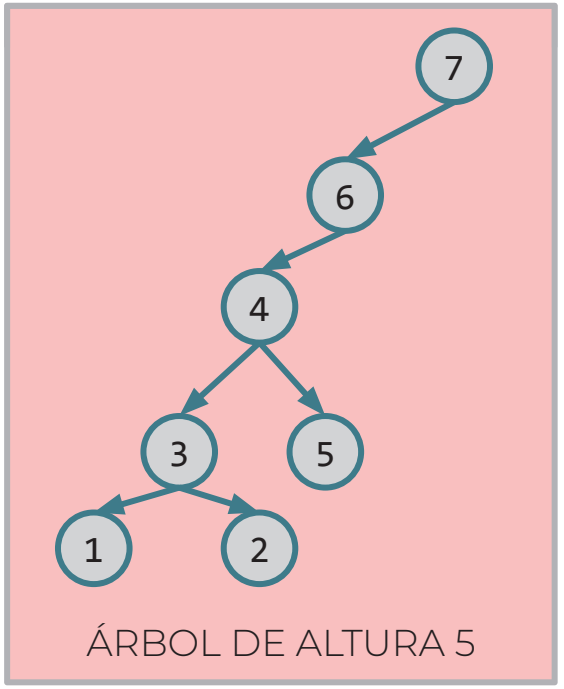
Aunque normalmente las operaciones típicas (búsqueda, inserción y borrado) en árboles suelen ser eficientes. El costo algorítmico de las operaciones en árboles varía dependiendo del tipo de árbol que se esté utilizando. Sin embargo, se debe tener en cuenta si el árbol está balanceado, de lo contrario el coste algorítmico podría empeorar. Veamos los siguientes ejemplos utilizando los mismos datos en distintos árboles binarios de búsqueda.



ÁRBOL DE ALTURA 3



ÁRBOL DE ALTURA 4



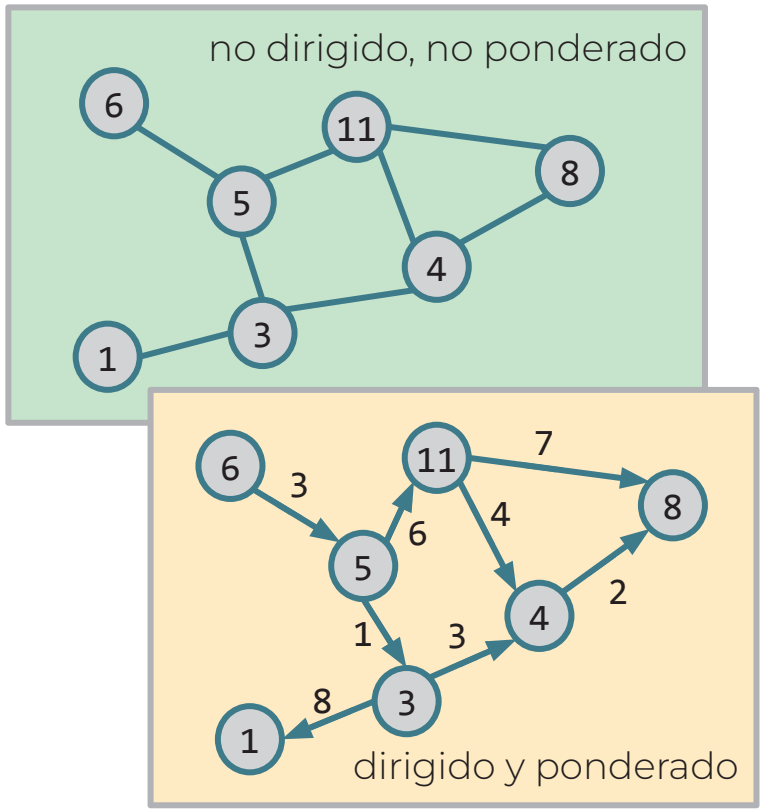
ÁRBOL DE ALTURA 5

Estructuras de datos: Grafo

Un grafo es una estructura de datos no lineal que consta de un conjunto de nodos (también conocidos como vértices) y un conjunto de aristas que conectan estos nodos. Cada arista representa una relación o conexión entre dos nodos.

Los grafos pueden ser **dirigidos** o **no dirigidos**. En un grafo no dirigido, las aristas no tienen una dirección asociada y la relación que representan se considera bidireccional. En un grafo dirigido, las aristas tienen una dirección asociada y la relación que representan se considera unidireccional.

Los grafos también pueden ser **ponderados** o **no ponderados**. En un grafo no ponderado, todas las aristas tienen el mismo peso o costo, mientras que en un grafo ponderado, cada arista tiene un peso o costo asociado que representa la magnitud de la relación que representa.



Estructuras de datos: Tabla Hash

Una tabla hash es una estructura de datos que permite el acceso rápido a un conjunto de elementos utilizando una función de hash. La función de hash a partir de la información del elemento, transforma una clave de búsqueda en una posición dentro de la tabla hash, donde se almacena el valor correspondiente.

La tabla hash consta de una matriz de "**buckets**" o casillas, donde cada bucket es una lista enlazada que contiene los valores correspondientes a una posición de la tabla. Cuando se inserta un elemento en la tabla hash, se aplica la función de hash a la clave de búsqueda para determinar la posición en la tabla donde se almacenará el valor. Si la posición ya está ocupada, el valor se agrega a la lista enlazada correspondiente al bucket.

Una buena función hash debe generar posiciones aleatorias y uniformemente distribuidas en la tabla para minimizar las colisiones, es decir, cuando dos elementos diferentes generan la misma posición en la tabla. Si la función de hash no es uniforme, pueden producirse colisiones frecuentes y disminuir el rendimiento de la tabla hash. Las tablas hash proveen tiempo constante de búsqueda promedio O(1), sin importar el número de elementos en la tabla. Sin embargo, una mala función hash puede llegar a empeorar el rendimiento llegando incluso a un coste O(n)

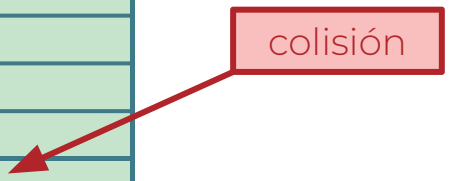
Estructuras de datos: Tabla Hash

Las tablas hash se utilizan en una variedad de aplicaciones, como la búsqueda de elementos en grandes bases de datos, la indexación de archivos y la implementación de estructuras de datos como conjuntos y mapas. La tabla hash ofrece un acceso rápido a los datos y una buena eficiencia en términos de tiempo y espacio, pero **su rendimiento depende en gran medida de la calidad de la función de hash y la técnica de resolución de colisiones utilizada.**

```
hash("Pep Garcia") → 334
hash("Tom Torres") → 128
hash("Khal Drogo") → 536
hash("Kim Dotcom") → 536
```

```
//mala función hash
hash("Pep Garcia") → 10
hash("Tom Torres") → 10
hash("Khal Drogo") → 10
hash("Kim Dotcom") → 10
```

TABLA HASH			
buckets	[0]	[1]	[2]
0			
1 2			
...			
128	Tom Torres		
...			
334	Pep Garcia		
...			
536	Khal Drogo	Kim Dotcom	
....			
1024			

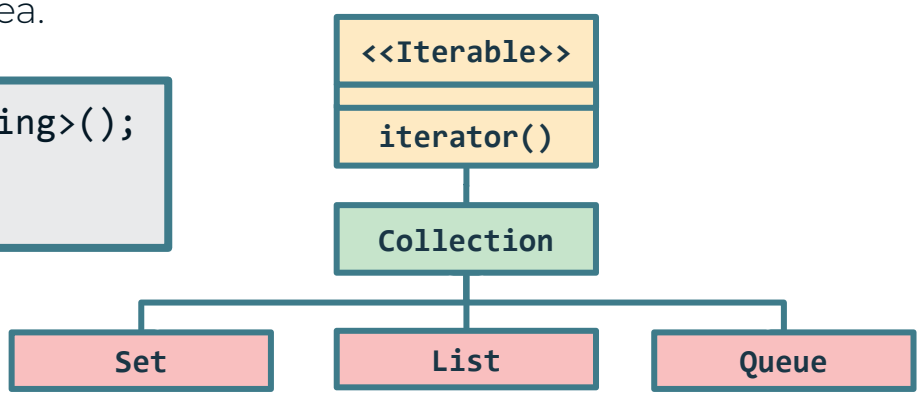


Iteradores en JAVA:
Interfaces Iterable‹T› e Iterator‹T›

Iteradores en JAVA: Iterable e Iterator

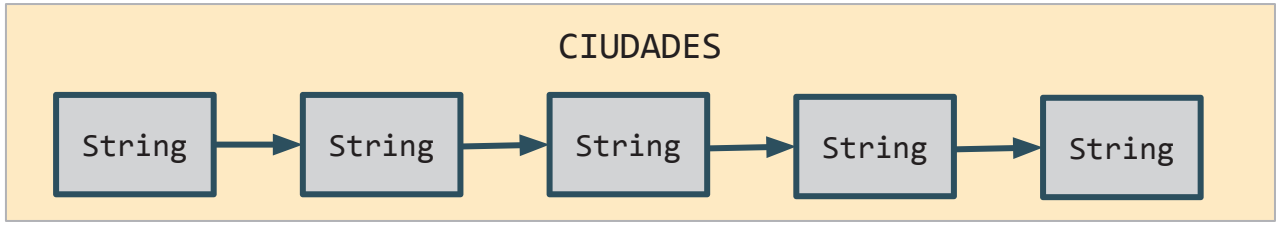
- De forma genérica un iterador es un objeto que podemos usar para obtener todos los objetos de una colección uno a uno. Para ello se utiliza la programación genérica.
- La programación genérica permite definir clases, interfaces y métodos que pueden trabajar con diferentes tipos de datos sin necesidad de crear una versión separada para cada tipo. En lugar de especificar el tipo de dato concreto en el momento de la definición, se utiliza un tipo de dato genérico, que se especifica entre corchetes **<T>**.
- En JAVA, **Iterable<T>** es una interfaz que puede ser implementada por una clase de colección. El único método abstracto de dicha interfaz es **public Iterator<T> iterator( )**
- Cualquier colección puede crear un objeto de tipo **Iterator<T>**. Este le proveerá una forma fácil de obtener uno a uno todos los objetos que posea.

```
ArrayList<String> ciudades = new ArrayList<String>();
...
Iterator<String> it = ciudades.iterator();
```

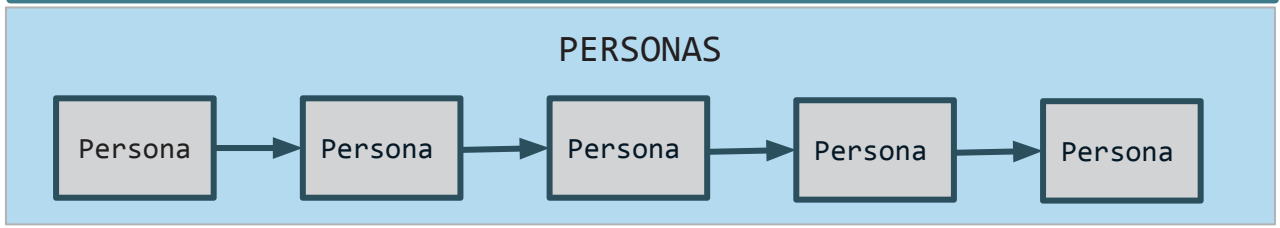


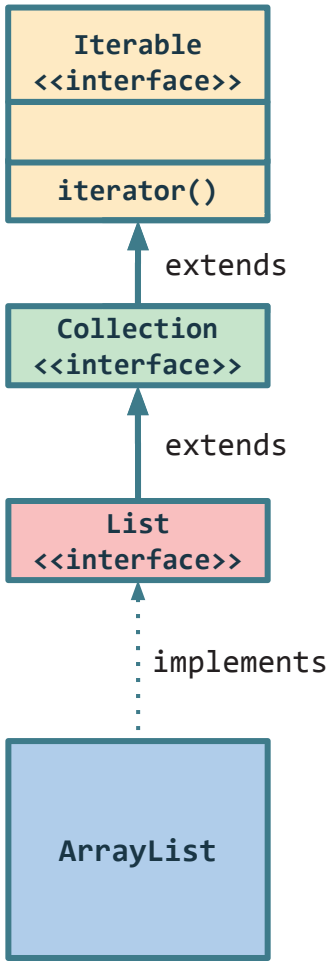
Iteradores en JAVA: Iterable e Iterator

```
ArrayList<String> ciudades = new ArrayList<String>();
...
Iterator<String> it = ciudades.iterator();
```




```
ArrayList<Persona> personas = new ArrayList<Persona>();
...
Iterator<Persona> it = personas.iterator();
```





Interfaces Iterable e Iterator

- La interface **Iterable<T>** hace referencia a una colección de elementos que se puede recorrer, así de simple.
- Dicha interface solo necesita que implementemos un método para poder funcionar de forma correcta, este método es **public Iterator<T> iterator( )**
- En JAVA, un Iterator es una interfaz que puede ser implementada por una clase que implementa la interface Collection.
- Principales métodos:
  - → **next( )**: retorna un objeto de tipo Object empezando por el primero y establece el iterator para que retorne el próximo objeto en la siguiente llamada a este mismo método. Si no existe próximo objeto y se invoca next() se produce una NoSuchElementException.
  - → **hasNext( )**: retorna true si existe un próximo objeto a retornar a través de la llamada a la función next().
  - → **remove( )**: Elimina el último objeto retornado por la función next(). Si no se invoca next() antes de remove() o se invoca dos veces después de next(), se produce una IllegalStateException.

Ejemplo de uso de Iterator

```
ArrayList<String> ciudades = new ArrayList<String>();
ciudades.add("New York");
ciudades.add("Tokyo");
ciudades.add("París");
System.out.print("Ciudades: ");
Iterator<String> it = ciudades.iterator();
it.remove(); // IllegalStateException
while(it.hasNext()) {
     System.out.println(it.next());
}
```

```
ArrayList<String> ciudades = new ArrayList<String>();
ciudades.add("New York");
ciudades.add("Tokyo");
ciudades.add("París");
System.out.print("Ciudades: ");
Iterator<String> it = ciudades.iterator();
it.next();
it.remove();
while(it.hasNext()) {
     System.out.print(it.next() + " ");
} // Ciudades: Tokyo París
```

Interfaces Iterable e Iterator

En Java, existen tres tipos de iteradores:

- **Iterator**: es el iterador básico que se utiliza para recorrer colecciones. No permite la modificación de la colección original.
- **ListIterator**: es un tipo de iterador que se utiliza específicamente para recorrer listas y permite la modificación de la lista original (por ejemplo, añadir o eliminar elementos).
- **Spliterator**: es un iterador que se utiliza para recorrer colecciones grandes y paralelizar el procesamiento de los elementos. También permite la modificación de la colección original.

Pero entonces, si únicamente queremos recorrer una colección, no será mejor hacerlo con un for each, y nos olvidamos de crear el iterador?

Respuesta rápida, si. Sin embargo, es importante tener en cuenta que no todas las colecciones en JAVA soportan el uso de un bucle for-each. En particular, las colecciones que no implementan la interfaz Iterable (por ejemplo, Map) no pueden ser recorridas utilizando un bucle for-each. En estos casos, es necesario utilizar un Iterator o un bucle for convencional.

Además podemos crear nuestros propios iteradores programando los métodos a nuestro gusto. Esto nos permitirá recorrer las colecciones en base a ciertas condiciones. Veremos ejemplos de ello.

Interfaces Iterable e Iterator

- Veamos un ejemplo, en primer lugar creamos un ArrayList

```
ArrayList<String> clientes = new ArrayList<>();
clientes.add("Pepe García");
clientes.add("Toni Pérez");
clientes.add("Marta Gómez");
clientes.add("Sara Martínez");
```

- El siguiente bucle generará una excepción **ConcurrentModificationException** ya que estamos modificando el tamaño de la colección a medida que lo recorremos.

```
for (String c : clientes) {
     if (c.equals("Toni Pérez")) clientes.remove(c);
     System.out.println(c);
}
// debería mostrar a todos menos a Toni Pérez, pero muestra
// Pepe García
// Toni Pérez
// ConcurrentModificationException
```

Interfaces Iterable e Iterator

- Aquí es donde un iterador puede resultarnos útil .

```
ArrayList<String> clientes = new ArrayList<>();
clientes.add("Pepe García");
clientes.add("Toni Pérez");
clientes.add("Marta Gómez");
clientes.add("Sara Martínez");
```

- El siguiente código ya no genera una excepción **ConcurrentModificationException**.

```
Iterator<String> clienteIterator = clientes.iterator();
while (clienteIterator.hasNext()) {
     String cliente = clienteIterator.next();
     if (cliente.equals("Toni Pérez")) clienteIterator.remove();
     System.out.println(cliente);
}
// muestra los 4 clientes, en el ArrayList final no tendrá a Toni Pérez
```

Implementación Interface Iterable en JAVA

Ejemplo de clase que implementa la interfaz Iterable

```
public class Grupo implements Iterable<Alumno> {

     private String nombre;
     private ArrayList<Alumno> alumnos;

     public Grupo(String nombre) {
          this.nombre = nombre;
          this.alumnos = new ArrayList<>();
     }

     //...

     @Override
     public Iterator<Alumno> iterator() {
          return new alumnos.iterator();
     }

}
```

```
public class Alumno {
     private String nombre
     private String nia;
     private int edad;
     //constructor
     //getters y setters
}
```

Utilizamos el iterator() de una clase que ya lo tiene implementado, en este caso, la clase ArrayList

EJEMPLO1

Ejemplo de clase que implementa la interfaz Iterable

```
public class Grupo implements Iterable<Alumno> {

     private String nombre;
     private ArrayList<Alumno> alumnos;

     public Grupo(String nombre) {
          this.nombre = nombre;
          this.alumnos = new ArrayList<>();
     }
     //...
     @Override
     public Iterator<Alumno> iterator() {
          return new IteratorGrupo();
     }
     private class IteratorGrupo implements Iterator<Alumno> {
          private int posicion = 0;
          //métodos abstractos interfaz Iterator<T>
          public boolean hasNext()    { return posicion < alumnos.size();   }
          public Alumno next()        { return alumnos.get(posicion++);     }
     }
}
```

```
public class Alumno {
     private String nombre
     private String nia;
     private int edad;
     //constructor
     //getters y setters
}
```

Creamos una clase interna que implementa la interfaz Iterator, y devolvemos una instancia de dicha clase

EJEMPLO2

Ejemplo de clase que implementa la interfaz Iterable

```java
public class Grupo implements Iterable<Alumno> {

    private String nombre;
    private ArrayList<Alumno> alumnos;

    public Grupo(String nombre) {
        this.nombre = nombre;
        this.alumnos = new ArrayList<>();
    }
    //...
    @Override
    public Iterator<Alumno> iterator() {
        return new Iterator<Alumno>() {
            private int posicion = 0;
            //métodos abstractos interfaz Iterator<T>
            public boolean hasNext()    { return posicion < alumnos.size();   }
            public Alumno next()        { return alumnos.get(posicion++);     }
        };
    }
}
```

```java
public class Alumno {
    private String nombre
    private String nia;
    private int edad;
    //constructor
    //getters y setters
}
```

Creamos nuestro propio iterador sin definir ninguna clase, únicamente implementando los métodos abstractos de la interfaz Iterator

EJEMPLO3

Ejercicio ampliación interfaz Iterator

Configurar el iterador para que el método next() únicamente devuelva alumnos que tengan NIA, es decir el NIA no sea nulo.

```
private class IteratorGrupo implements Iterator<Alumno> {
     private int posicion = 0;

     @Override
     public boolean hasNext() {
        while (posicion < alumnos.size() && alumnos.get(posicion).getNia() == null) {
              posicion++;
        }
        return posicion < alumnos.size();
     }

     @Override
     public Alumno next() {
        return alumnos.get(posicion++);
     }
}
```