

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**.

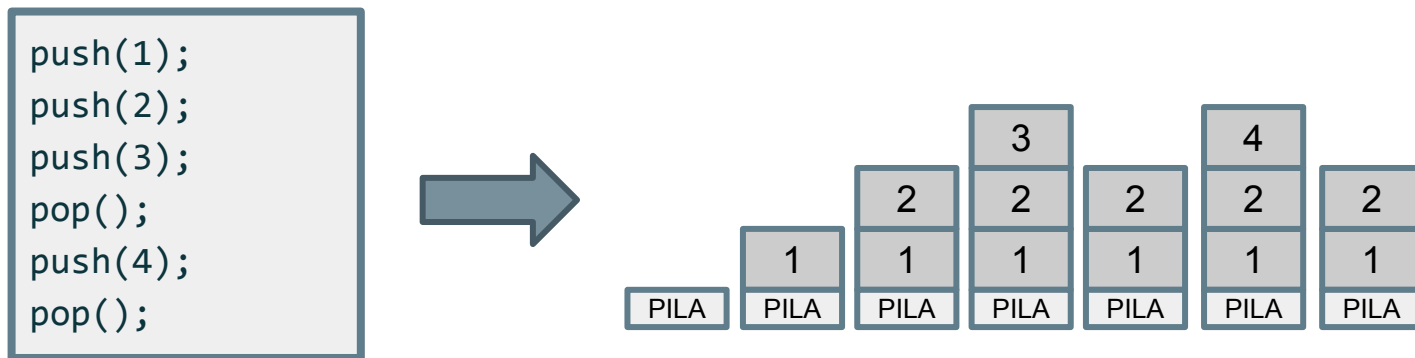
Estructuras de datos

- **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 de 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.

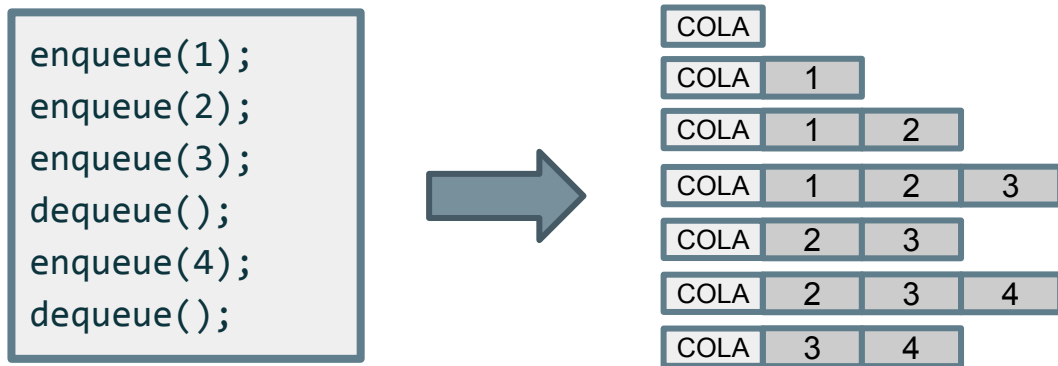


Las pilas son útiles en situaciones en las que se necesitan realizar operaciones en un orden específico, como en la evaluación de expresiones aritméticas. También se utilizan en la implementación de algoritmos recursivos y en la gestión de llamadas a funciones en una pila de llamadas.

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.



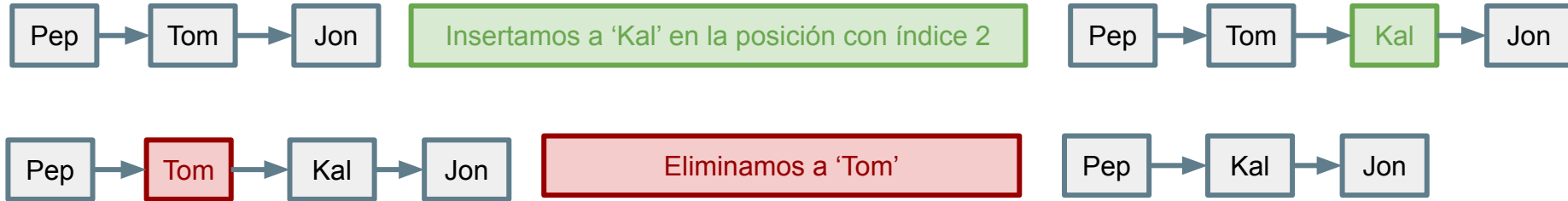
Las colas son útiles en situaciones en las que se necesitan procesar elementos en un orden específico, como en la gestión de tareas en un sistema operativo. También se utilizan en la gestión de recursos compartidos, como en la impresión de documentos en una impresora compartida, la típica cola de impresión.

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.

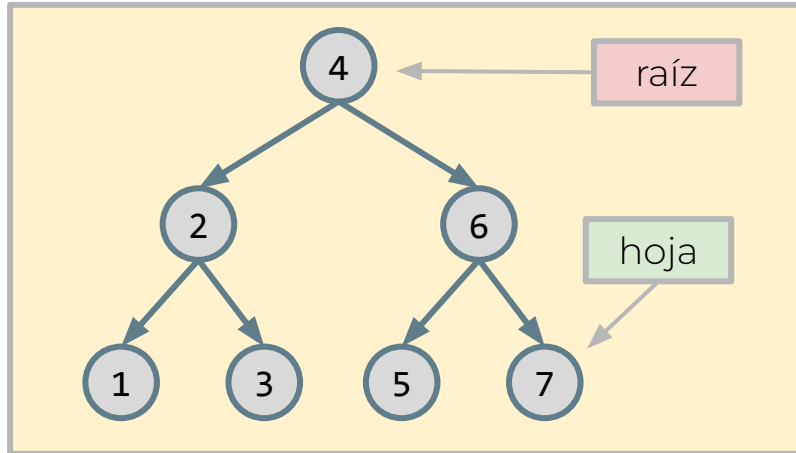
Las listas permiten agregar y eliminar elementos de la colección en cualquier parte de la lista.



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)

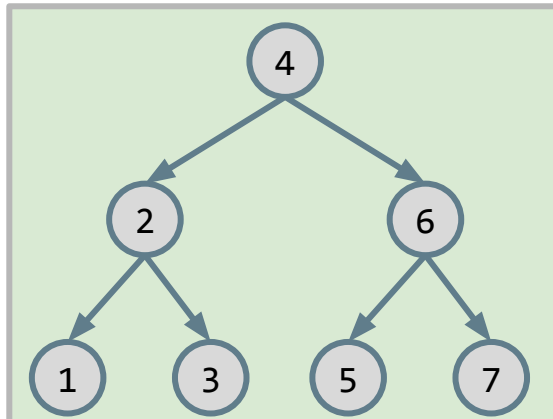
La forma más común de implementar un árbol es mediante punteros. Cada nodo del árbol se representa como un objeto que contiene un valor y punteros a sus nodos hijos. La estructura de punteros permite recorrer el árbol y acceder a sus nodos y elementos de manera eficiente.



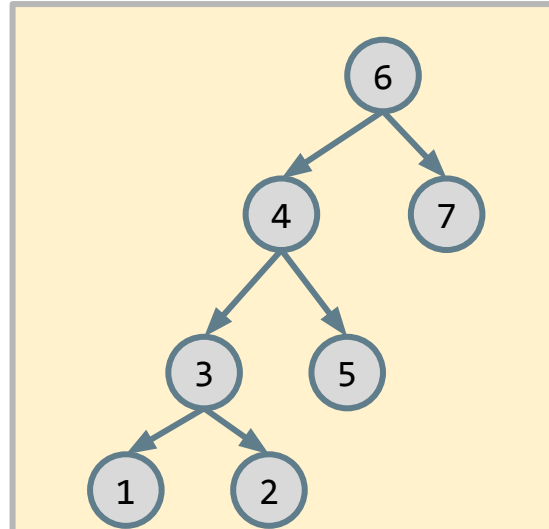
Los árboles se utilizan cuando se requiere una estructura de datos dinámica que pueda manejar grandes conjuntos de datos de manera eficiente. La elección del tipo de árbol depende de los requisitos específicos del problema y del conjunto de datos que se maneje.

Estructuras de datos: Árbol

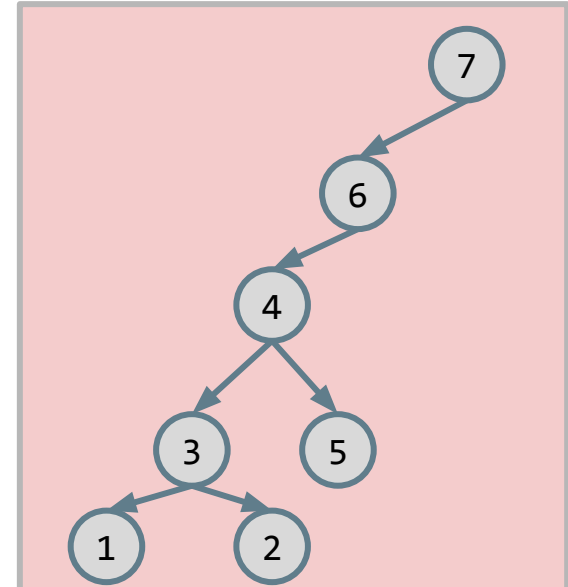
Aunque normalmente las operaciones típicas (búsqueda, inserción y borrado) en árboles suelen ser eficientes. El costo algorítmico de las operaciones en árboles varía dependiendo del tipo de árbol que se esté utilizando. Sin embargo, se debe tener en cuenta si el árbol está balanceado, de lo contrario 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



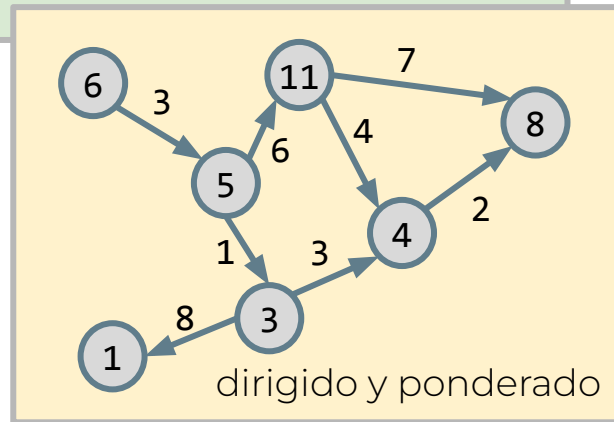
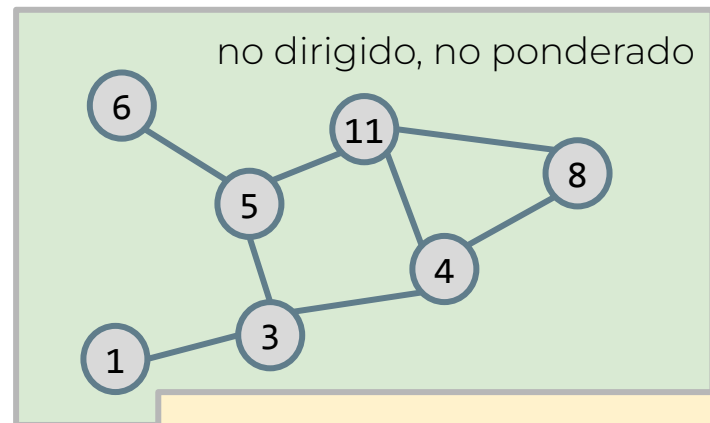
ÁRBOL DE ALTURA 5

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			

colisión

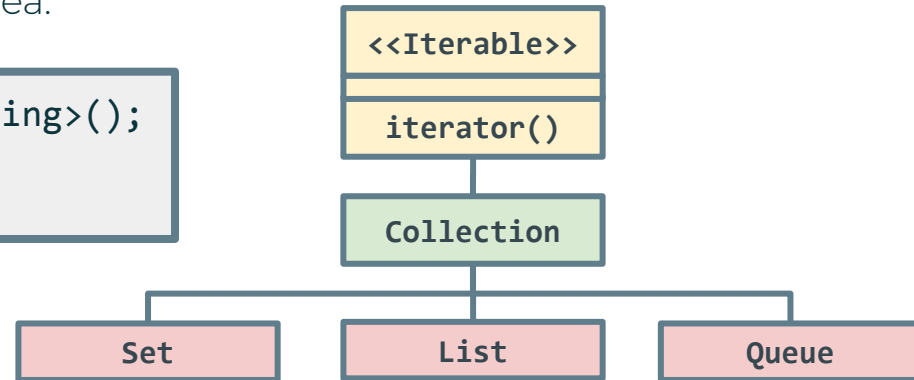
Iteradores en JAVA:

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.

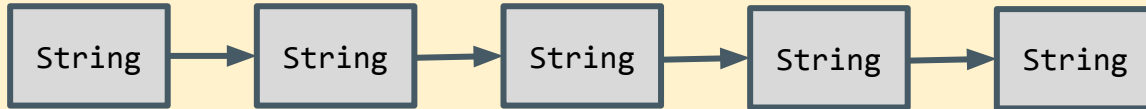
```
ArrayList<String> ciudades = new ArrayList<String>();  
...  
Iterator<String> it = ciudades.iterator();
```



Iteradores en JAVA: Iterable e Iterator

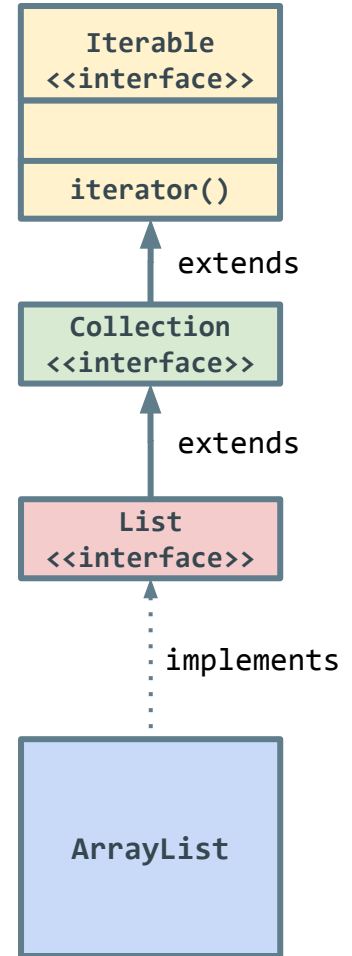
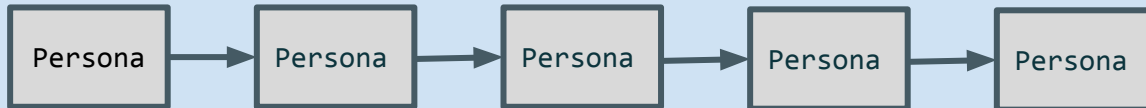
```
ArrayList<String> ciudades = new ArrayList<String>();  
...  
Iterator<String> it = ciudades.iterator();
```

CIUDADES



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

PERSONAS



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

- 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

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

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

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++);
    }
}
```

Interface Comparable<T> en JAVA

Interface Comparable<T>

En ocasiones puede que necesitemos ordenar colecciones de datos. Por ello, necesitamos alguna forma de poder comparar tanto tipos primitivos, como objetos. Por ejemplo, podemos comparar cadenas de texto con el método **compareTo(String s)** de la clase **String**. Dicho método nos devuelve un entero basado en la comparación lexicográfica de ambas cadenas.

```
String s1 = "ABC";
String s2 = "CDE";

int resultado = s1.compareTo(s2); //resultado devuelve un valor negativo
if(resultado < 0) System.out.println("s1 es menor que s2");
else if(resultado == 0) System.out.println("s1 es igual a s2");
else System.out.println("s1 es mayor que s2");

s2 = "AAA";
resultado = s1.compareTo(s2); //resultado devuelve un valor positivo
s1 = "AAA";
resultado = s1.compareTo(s2); //resultado devuelve un 0
```

Sin embargo, si queremos comparar instancias de nuestros propios objetos, debemos indicarle a JAVA cómo hacerlo. Para ello disponemos de la interface **Comparable<T>**.

Ejemplo de implementación Interface Comparable

La interface **Comparable<T>** tiene un único método abstracto que debemos implementar, dicho método es **compareTo(T obj)**, debe devolver un entero.

La implementación de compareTo() es importante para que los objetos de una clase puedan ser ordenados y comparados de manera coherente en todo tipo de Colecciones.

```
public class Alumno implements Comparable<Alumno> {  
  
    private String nombre;  
    private String nia;  
    private int edad;  
    // constructores, getters y setters  
  
    @Override  
    public int compareTo(Alumno a) {  
        int comparacion = Integer.compare(this.edad, a.edad);  
        if (comparacion == 0) comparacion = this.nombre.compareTo(a.nombre);  
        return comparacion;  
    }  
} // ordenamos por edad, de menor a mayor, y luego por nombre
```

Ordenar elementos en una colección tipo List

Si una clase implementa la interface `Comparable<T>`, una colección que implementa la interface `List<T>` puede ser ordenada fácilmente a través del método **`Collections.sort(List<T> list)`**.

Veamos un ejemplo ordenando un `ArrayList` de alumnos:

```
ArrayList<Alumno> alumnos = new ArrayList<>();
alumnos.add(new Alumno("Pep", "1111A", 15));
alumnos.add(new Alumno("Tom", "2222A", 17));
alumnos.add(new Alumno("Jon", "3333A", 14));

Collections.sort(alumnos); //ordenamos en base al método compareTo de la clase Alumno
System.out.println(alumnos); //Los muestra ordenados por edad → Jon, Pep, Tom

alumnos.add(new Alumno("Ben", "4444A", 14));

Collections.sort(alumnos); //ordenamos en base al método compareTo de la clase Alumno
System.out.println(alumnos); //Los muestra ordenados por edad → Ben Jon, Pep, Tom
```

Interface Comparator<T> en JAVA

Ejemplo de implementación Interface Comparator<T>

Tanto la interface **Comparator<T>** como la interfaz **Comparable<T>** se utilizan para comparar objetos, no obstante, la interface Comparator<T> nos ofrece una mayor flexibilidad. La interfaz Comparator<T> permite definir múltiples criterios de comparación para un objeto, e incluso, nos permite comparar objetos de diferentes clases sin necesidad de modificar la clase original.

La interface comparator nos obligará a implementar el método abstracto **compare(..)**, el cual recibirá dos instancias para que sean comparadas.

```
public class AlumnoPorEdadComparator implements Comparator<Alumno> {  
    public int compare(Alumno a1, Alumno a2) {  
        int comparacion = Integer.compare(a1.getEdad(), a2.getEdad());  
        if (comparacion == 0) comparacion = a1.getNombre().compareTo(a2.getNombre());  
        return comparacion;  
    }  
}
```

```
public class AlumnoPorNiaComparator implements Comparator<Alumno> {  
    public int compare(Alumno a1, Alumno a2) {  
        return a1.getNia().compareTo(a2.getNia());  
    }  
}
```

Ejemplo de implementación Interface Comparator<T>

A continuación podemos ver cómo utilizar las anteriores implementaciones de la interface Comparator<T> en un main.

```
public static void main(String[] args) {  
  
    ArrayList<Alumno> alumnos = new ArrayList<>();  
    alumnos.add(new Alumno("Pep", "222A", 25));  
    alumnos.add(new Alumno("Tom", "111A", 20));  
    alumnos.add(new Alumno("Jon", "444A", 21));  
    alumnos.add(new Alumno("Tim", "333A", 19));  
  
    Collections.sort(alumnos, new AlumnoPorEdadComparator());  
    System.out.println("Alumnos ordenados por edad");  
    System.out.println(alumnos);  
  
    Collections.sort(alumnos, new AlumnoPorNiaComparator());  
    System.out.println("Alumnos ordenados por NIA");  
    System.out.println(alumnos);  
  
}
```

Ejemplo de implementación Interface Comparator<T>

También tenemos la opción de implementar la interface sin crear directamente una clase.

```
public static void ordenarAlumnosPorEdad(ArrayList<Alumno> alumnos) {
    alumnos.sort(new Comparator<Alumno>() {
        @Override
        public int compare(Alumno a1, Alumno a2) {
            if (a1.getEdad() > a2.getEdad()) return 1;
            if (a1.getEdad() < a2.getEdad()) return -1;
            return a1.getNombre().compareTo(a2.getNombre());
        }
    });
}

public static void ordenarAlumnosPorNia(ArrayList<Alumno> alumnos) {
    alumnos.sort(new Comparator<Alumno>() {
        @Override
        public int compare(Alumno a1, Alumno a2) {
            return a1.getNia().compareTo(a2.getNia());
        }
    });
}
```

Interface Comparator<T> en JAVA: Implementación con distintas clases

Ejemplo de implementación Interface Comparator<T>

```
public class NombreComparator implements Comparator<Object> {  
    public int compare(Object obj1, Object obj2) {  
        String nombre1 = null;  
        String nombre2 = null;  
  
        if (obj1 instanceof Alumno) nombre1 = ((Alumno)obj1).getNombre();  
        else if (obj1 instanceof Docente) nombre1 = ((Docente)obj1).getNombre();  
  
        if (obj2 instanceof Alumno) nombre2 = ((Alumno)obj2).getNombre();  
        else if (obj2 instanceof Docente) nombre2 = ((Docente)obj2).getNombre();  
  
        if (nombre1 != null && nombre2 != null) return nombre1.compareTo(nombre2);  
        else if (nombre1 != null) return -1;  
        else if (nombre2 != null) return 1;  
        return 0;  
    }  
}
```

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

```
public class Docente {  
    private String nombre;  
    private int permanencia;  
    private double salario;  
    //constructor  
    //getters y setters  
}
```

Implementación de la interfaz
Comparator<T> para comparar
objetos de diferentes clases
basados en un atributo en común.

Ejemplo de implementación Interface Comparator<T>

Ejemplo de uso de Comparator ordenando instancias de distintas clases en base a un atributo común.

```
ArrayList<Object> instituto = new ArrayList<>();
instituto.add(new Alumno("Pep", "222A", 25));
instituto.add(new Alumno("Tom", "111A", 20));
instituto.add(new Alumno("Jon", "444A", 21));
instituto.add(new Alumno("Tim", "333A", 19));
instituto.add(new Alumno("Ada", "555A", 18));
instituto.add(new Docente("Kal", 5, 2000));
instituto.add(new Docente("Ana", 15, 3000));
instituto.add(new Docente("Sam", 2, 1800));
instituto.add(new Docente("Pol", 8, 2500));
instituto.add(new Docente("Ben", 10, 2700));

System.out.println("Alumnos y docentes ordenados por nombre");
Collections.sort(instituto, new NombreComparator());
System.out.println(instituto);

// Ada, Ana, Ben, Jon, Kal, Pep, Pol, Sam, Tim, Tom
```

Ejemplo de implementación Interface Comparator<T>

Implementación de la interface Comparator<T> haciendo uso de una interface auxiliar:

```
public interface PersonaCentroEducativo {  
    public String getNombre();  
}  
public class Alumno implements PersonaCentroEducativo {  
    //constructor, getters y setters  
    //...  
}  
public class Docente implements PersonaCentroEducativo {  
    //constructor, getters y setters  
    //...  
}
```

```
public class NombreComparatorCentroEducativo implements Comparator<PersonaCentroEducativo>  
{  
    @Override  
    public int compare(PersonaCentroEducativo p1, PersonaCentroEducativo p2) {  
        return p1.getNombre().compareTo(p2.getNombre());  
    }  
}
```

Ejemplo de implementación Interface Comparator<T>

Ejemplo de uso de Comparator ordenando instancias de distintas clases en base a una interface común.

```
ArrayList<PersonaCentroEducativo> instituto = new ArrayList<>();
instituto.add(new Alumno("Pep", "222A", 25));
instituto.add(new Alumno("Tom", "111A", 20));
instituto.add(new Alumno("Jon", "444A", 21));
instituto.add(new Alumno("Tim", "333A", 19));
instituto.add(new Alumno("Ada", "555A", 18));
instituto.add(new Docente("Kal", 5, 2000));
instituto.add(new Docente("Ana", 15, 3000));
instituto.add(new Docente("Sam", 2, 1800));
instituto.add(new Docente("Pol", 8, 2500));
instituto.add(new Docente("Ben", 10, 2700));

System.out.println("Alumnos y docentes ordenados por nombre");
Collections.sort(instituto, new NombreComparatorCentroEducativo());
System.out.println(instituto);

// Ada, Ana, Ben, Jon, Kal, Pep, Pol, Sam, Tim, Tom
```

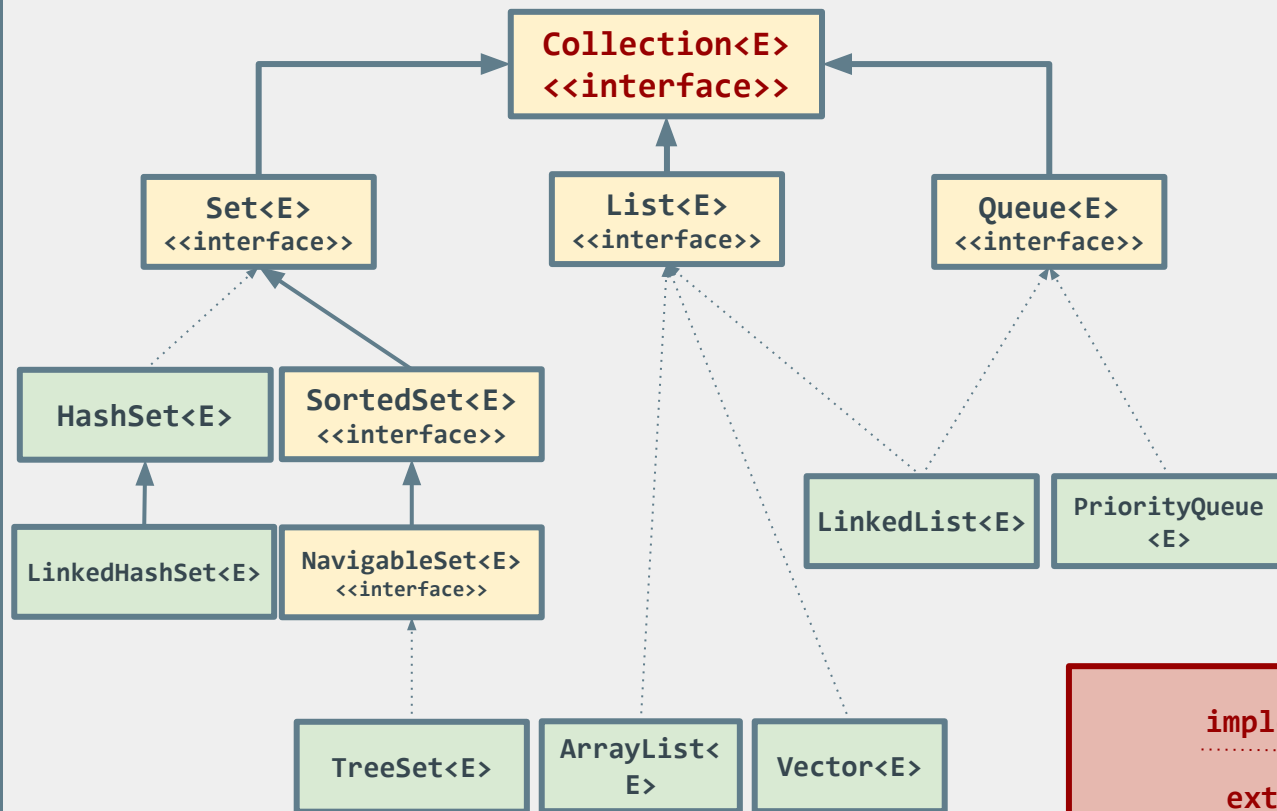
Colecciones en JAVA: Java Collections Framework

Colecciones en JAVA

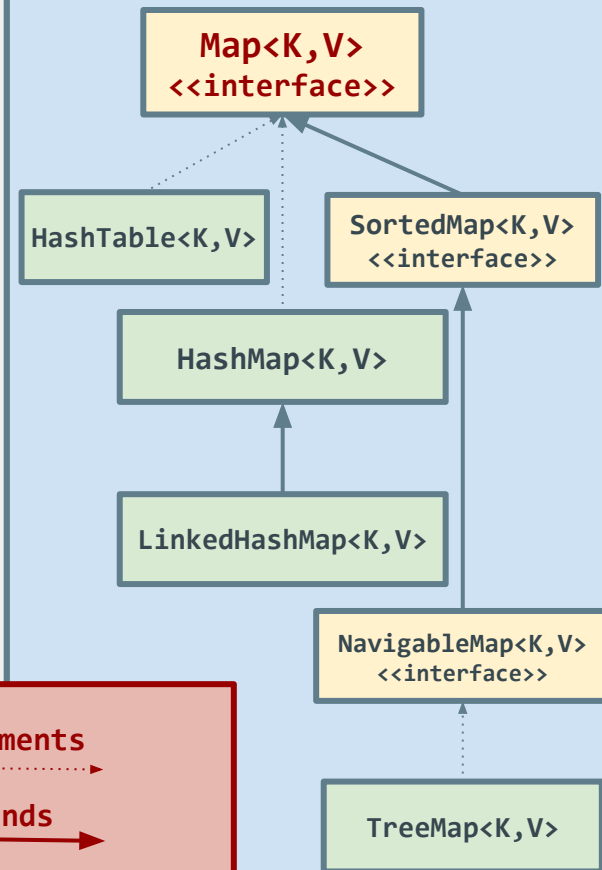
En JAVA, las colecciones de objetos se agrupan utilizando el **Java Collections Framework** (JCF), un conjunto de interfaces y clases que facilitan el trabajo con colecciones de objetos. El JCF proporciona una serie de estructuras de datos comunes y algoritmos para manipular colecciones de objetos, lo que permite a los desarrolladores centrarse en la lógica de la aplicación en lugar de implementar las estructuras desde cero. El JCF se organiza en torno a varias interfaces clave:

- Interfaz **Collection<E>**: es la raíz de la jerarquía de colecciones en JAVA. Define un conjunto básico de operaciones que cualquier colección de objetos debe soportar, como agregar, eliminar y verificar la existencia de elementos. Esta interfaz tiene varias subinterfaces que proporcionan más funcionalidad específica:
 - **List<E>**: colección ordenada de elementos que permite duplicados, clase ArrayList
 - **Set<E>**: presenta una colección de elementos únicos, es decir, no permite duplicados, clases HashSet y TreeSet
 - **Queue<E>**: representa una colección de elementos que se procesan siguiendo el principio FIFO, los elementos se insertan en un final de la cola y se eliminan por el inicio.
- Interfaz **Map<K, V>**: no es una subinterfaz de Collection, pero es una parte fundamental del JCF. Representa una colección de pares clave-valor donde cada clave está asociada a un valor. Las claves son únicas, pero los valores pueden duplicarse. HashMap, TreeMap y LinkedHashMap son implementaciones comunes de Map.

Collection Interface



Map<K, V> <<interface>>



Principales Métodos de la interface Collection<E> en JAVA

- **boolean add(Object o):** Agrega un elemento a la colección. Retorna true si la colección cambia como resultado de la llamada (es decir, si el elemento se agrega correctamente) y false en caso contrario (por ejemplo, si el elemento ya está en un conjunto que no permite duplicados).
- **boolean remove(Object o):** Elimina una instancia del objeto especificado de la colección, si está presente. Retorna true si la colección cambia y false en caso contrario.
- **void clear():** Elimina todos los elementos de la colección.
- **boolean contains(Object o):** Verifica si la colección contiene una instancia del objeto especificado. Retorna true si el objeto está presente y false en caso contrario.
- **boolean isEmpty():** Retorna true si la colección está vacía y false en caso contrario.
- **int size():** Devuelve el número de elementos en la colección.
- **Object[] toArray():** Retorna un array que contiene todos los elementos de la colección en el orden en que son devueltos por el iterador de la colección.
- **Iterator<E> iterator():** Retorna un iterador sobre los elementos de la colección.
- **boolean equals(Object o):** Compara la colección actual con el objeto especificado para verificar si son iguales. Normalmente, dos colecciones se consideran iguales si tienen el mismo tamaño y contienen los mismos elementos.
- **int hashCode():** Retorna el valor hash de la colección, que generalmente se calcula en función de los valores hash de los elementos contenidos en la colección.

Interface List<E> en JAVA

La interface List<E> en JAVA

La interfaz **List<E>** en JAVA es una subinterfaz de la interfaz **Collection<E>** y forma parte del JCF. Representa una colección ordenada de elementos, permitiendo duplicados. Los elementos en una lista están indexados, esto permite acceder, insertar y modificar elementos en posiciones específicas.

La letra <E> en la interfaz List<E> representa un parámetro de tipo genérico. Esto significa que al implementar la interfaz List, se puede especificar el tipo de elementos que se almacenarán en la lista. Por ejemplo, un List<String> contendría objetos de tipo String. Hay varias implementaciones comunes de la interfaz List<E> en el JCF. Algunas de las más utilizadas son:

- **ArrayList** es una implementación de List basada en un Array dinámico. Proporciona acceso rápido y constante en tiempo de ejecución a elementos por índice, pero puede ser ineficiente en operaciones de inserción y eliminación en el medio de la lista.
- **LinkedList** es una implementación de List basada en una lista doblemente enlazada. A diferencia de ArrayList, LinkedList proporciona un rendimiento más eficiente en operaciones de inserción y eliminación en el medio de la lista, pero el acceso a elementos por índice es más lento. LinkedList también implementa las interfaces Queue y Deque, lo que la hace útil para estructuras de datos basadas en colas.

Métodos de la interface List<E> en JAVA

La interfaz List<E> extiende Collection<E>, lo que significa que también hereda todos los métodos definidos en la interfaz Collection. Además de los métodos heredados de Collection<E>, List<E> define varios métodos adicionales específicos para listas ordenadas e indexadas, como:

- **E get(int index):** devuelve el elemento en la posición especificada en la lista.
- **E set(int index, E element):** reemplaza el elemento en la posición especificada en la lista con el nuevo elemento proporcionado y devuelve el elemento anteriormente en esa posición.
- **void add(int index, E element):** inserta el elemento proporcionado en la posición especificada en la lista.
- **E remove(int index):** elimina y devuelve el elemento en la posición especificada en la lista.
- **int indexOf(E o):** devuelve el índice de la primera aparición del objeto especificado en la lista, o -1 si la lista no contiene el objeto.
- **int lastIndexOf(E o):** devuelve el índice de la última aparición del objeto especificado en la lista, o -1 si la lista no contiene el objeto.
- **ListIterator<E> listIterator():** devuelve un objeto ListIterator<E> que permite recorrer la lista en cualquier dirección y modificar la lista durante la iteración.

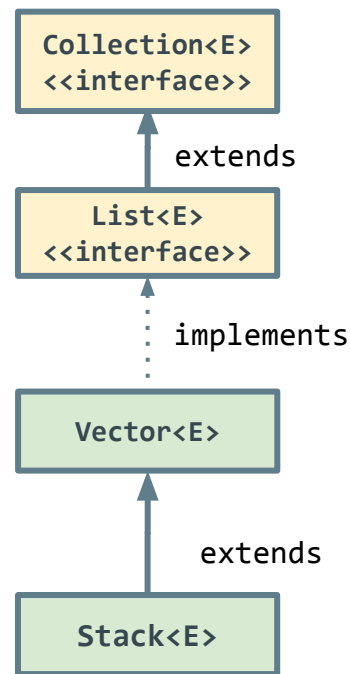
Clase Stack«E» en JAVA

Clase Stack<E>

Una pila (stack) es una estructura de datos lineal en la que se pueden agregar y eliminar elementos en la parte superior (top) de la estructura. La pila sigue el principio de "último en entrar, primero en salir" (LIFO, por sus siglas en inglés), lo que significa que el último elemento agregado a la pila será el primero en ser eliminado. En Java, la clase **Stack<E>** es una implementación de una pila a través de la interfaz **List<E>**. Los principales métodos que tenemos disponibles son:

- **push(E o)**: agrega un elemento a la pila
- **pop()**: elimina y devuelve el elemento en la parte superior de la pila. Si la pila está vacía, lanza una excepción `EmptyStackException`.
- **peek()**: obtiene el elemento en la parte superior de la pila sin eliminarlo.
- **size()**: obtiene el tamaño de la pila.
- **search(E o)**: devuelve la posición del elemento en la pila contando desde la parte superior. Si el elemento no se encuentra en la pila, devuelve -1.

La clase Stack en JAVA se considera algo obsoleta y se recomienda utilizar la interfaz Deque para implementar pilas y colas en su lugar. La interfaz Deque (cola de doble extremo) es más flexible y eficiente, y se puede utilizar como pila o cola según sea necesario, lo veremos más adelante.



Clase Stack«E»

```
import java.util.Stack;

//creamos una pila vacía
Stack<Integer> pila = new Stack<>();

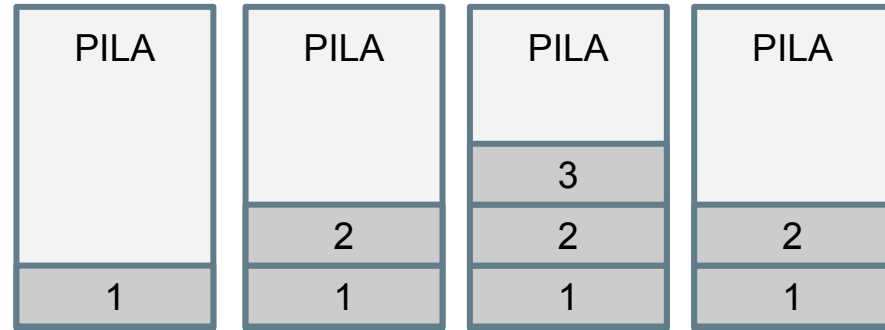
//apilamos un 1
pila.push(1);

//apilamos un 2
pila.push(2);

//apilamos un 3
pila.push(3);

//desapilamos
int elemento = pila.pop(); // → 3

//obtenemos el elemento de arriba de la pila
int elementoSuperior = pila.peek(); // → 2
```



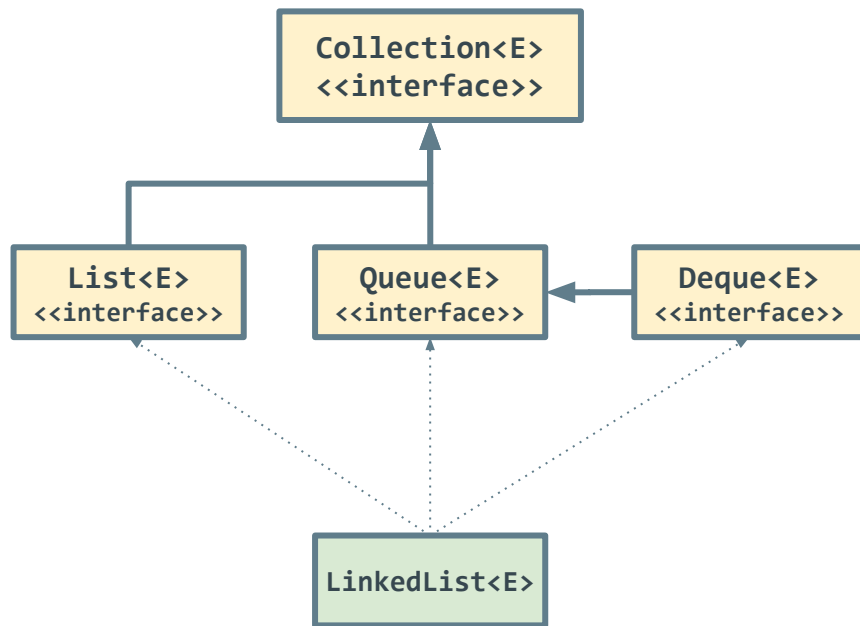
push(), **pop()** y **peek()** tienen una complejidad temporal de $O(1)$, lo que significa que su tiempo de ejecución no depende del tamaño de la pila y son consideradas eficientes.

Sin embargo, las operaciones de búsqueda, tienen una complejidad temporal de $O(n)$. Por ello, se consideran ineficientes si se usan repetidamente en pilas grandes.

Clase LinkedList<E> en JAVA

Ejemplo de implementación de la interface List<E>: LinkedList<E>

La clase **LinkedList<E>** en JAVA es una implementación de la interfaz **List<E>** y representa una lista doblemente enlazada de elementos. LinkedList<E> también implementa las interfaces **Queue<E>** y **Deque<E>**, por lo que proporciona funcionalidades de cola y cola de doble extremo además de las funcionalidades de lista. Por lo tanto, a parte de todos los métodos implementados a partir de la interface List<T>, también implementa todos los métodos de las interfaces Queue<E> y Deque<E>:



Métodos de LinkedList<E>

Métodos implementados por LinkedList<E> a partir de la interface **Queue<E>**:

- **boolean offer(E e)**: Agrega un elemento al final de la lista (similar a add(E e)).
- **E poll()**: Elimina y retorna el primer elemento de la lista, o null si la lista está vacía.
- **E peek()**: Retorna el primer elemento de la lista sin eliminarlo, o null si la lista está vacía.

Métodos implementados por LinkedList<E> a partir de la interface **Deque<E>**:

- **boolean offerFirst(E e)**: Inserta el elemento al comienzo de la lista.
- **boolean offerLast(E e)**: Inserta el elemento al final de la lista (similar a add(E e)).
- **E pollFirst()**: Elimina y retorna el primer elemento de la lista, o null si la lista está vacía.
- **E pollLast()**: Elimina y retorna el último elemento de la lista, o null si la lista está vacía.
- **E peekFirst()**: Retorna el primer elemento de la lista sin eliminarlo, o null si la lista está vacía.
- **E peekLast()**: Retorna el último elemento de la lista sin eliminarlo, o null si la lista está vacía.
- **E getFirst()**: Retorna el primer elemento de la lista sin eliminarlo.
- **E getLast()**: Retorna el último elemento de la lista sin eliminarlo.
- **E removeFirst()**: Elimina y retorna el primer elemento de la lista.
- **E removeLast()**: Elimina y retorna el último elemento de la lista.

getFirst(), getLast(), removeFirst() y removeLast() lanzan NoSuchElementException si está vacía.

Ejemplo de uso de la clase LinkedList<E>

Ejemplo de uso de métodos de la interface LinkedList.

```
LinkedList<Alumno> dam = new LinkedList<>();
```

```
//métodos interface Collection
```

```
dam.add(new Alumno("Pep", "222A", 25));  
dam.add(new Alumno("Tom", "111A", 20));  
dam.add(new Alumno("Jon", "444A", 21));  
dam.add(new Alumno("Tim", "333A", 19));  
dam.add(new Alumno("Ada", "555A", 18));  
dam.add(new Alumno("Sam", "666A", 18));
```

```
//métodos interface List
```

```
dam.set(2,new Alumno("Ana", "777A",20));  
dam.add(2,new Alumno("Bil", "777A",20));
```

```
//métodos interfaces Queue y Deque
```

```
dam.pollFirst();  
dam.pollLast();  
dam.offerFirst(new Alumno("Jud", "888A", 24));  
dam.offerLast(new Alumno("Kim", "999A", 28));  
System.out.println(dam.removeFirst());  
System.out.println(dam.removeLast());
```

```
public class Alumno {  
    //...  
    public String toString() {  
        return nombre;  
    }  
}
```

Ejemplo de uso de la clase LinkedList<E>

Ejemplo de uso de métodos de la interface LinkedList.

```
LinkedList<Alumno> dam = new LinkedList<>();

//métodos interface Collection
dam.add(new Alumno("Pep", "222A", 25)); //Pep
dam.add(new Alumno("Tom", "111A", 20)); //Pep, Tom
dam.add(new Alumno("Jon", "444A", 21)); //Pep, Tom, Jon
dam.add(new Alumno("Tim", "333A", 19)); //Pep, Tom, Jon, Tim
dam.add(new Alumno("Ada", "555A", 18)); //Pep, Tom, Jon, Tim, Ada
dam.add(new Alumno("Sam", "666A", 18)); //Pep, Tom, Jon, Tim, Ada, Sam

//métodos interface List
dam.set(2,new Alumno("Ana", "777A",20)); //Pep, Tom, Ana, Tim, Ada, Sam
dam.add(2,new Alumno("Bil", "777A",20)); //Pep, Tom, Bil, Ana, Tim, Ada, Sam

//métodos interfaces Queue y Deque
dam.pollFirst(); //Tom, Bil, Ana, Tim, Ada, Sam
dam.pollLast(); //Tom, Bil, Ana, Tim, Ada
dam.offerFirst(new Alumno("Jud", "888A", 24)); //Jud, Tom, Bil, Ana, Tim, Ada
dam.offerLast(new Alumno("Kim", "999A", 28)); //Jud, Tom, Bil, Ana, Tim, Ada, Kim
System.out.println(dam.removeFirst()); // Jud - dam → Tom, Bil, Ana, Tim, Ada, Kim
System.out.println(dam.removeLast()); // Kim - dam → Tom, Bil, Ana, Tim, Ada
```

Interface Set<E> en JAVA

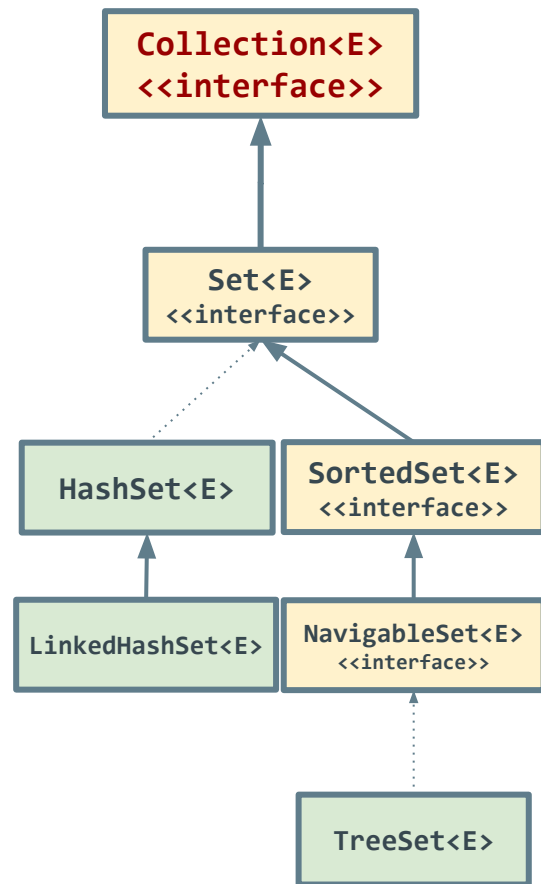
Interface Set<E> en JAVA

La interfaz Set<E> representa una colección de elementos únicos, es decir, no permite elementos duplicados. Como ya se ha visto, la letra <E> representa un parámetro de tipo genérico. Por ejemplo, un Set<Alumno> contendría una colección de instancias de tipo Alumno.

Dado que Set<E> no permite duplicados, su implementación del método add() garantiza que no se agreguen elementos duplicados. Si se intenta agregar un elemento que ya está presente en el conjunto, el método add() simplemente retorna false y no modificará el conjunto.

Al igual que List<E>, la interfaz Set<E> extiende Collection<E>, por ello, hereda todos los métodos definidos en la interfaz Collection. Sin embargo, **la interfaz Set<E> no introduce métodos propios.**

Es importante tener en cuenta que el comportamiento de estos métodos está adaptado a las propiedades de un conjunto, es decir, no permiten elementos duplicados. **Por ello, es necesario una correcta implementación de los métodos equals() y hashCode() sobre la clase Genérica utilizada en la colección.**



Implementaciones de la Interface Set<E> en JAVA

Hay varias implementaciones comunes de la interfaz Set<E> en el JCF. Algunas de las más utilizadas son HashSet, LinkedHashSet y TreeSet.

- **HashSet** es una implementación de Set que utiliza una tabla hash para almacenar los elementos. No garantiza ningún orden específico de los elementos, pero ofrece un rendimiento constante en tiempo de ejecución para las operaciones básicas (agregar, eliminar, buscar) en el caso promedio. HashSet permite valores nulos.
- **LinkedHashSet** es una extensión de HashSet que mantiene el orden de inserción de los elementos. Utiliza una estructura de datos de lista doblemente enlazada junto con la tabla hash para mantener el orden de inserción. El rendimiento de LinkedHashSet es ligeramente inferior al de HashSet debido a la sobrecarga adicional para mantener el orden de inserción.
- **TreeSet** es una implementación de Set basada en un árbol binario de búsqueda balanceado (red-black tree). Mantiene los elementos en un orden naturalmente ordenado (según el método compareTo() o un Comparator proporcionado). TreeSet no permite valores nulos y ofrece un rendimiento de tiempo logarítmico para las operaciones básicas.

Ejemplos de uso de HashSet<E>

```
Set<Integer> nums = new HashSet<>();  
nums.add(5);  
nums.add(8);  
nums.add(2);  
nums.add(3);  
nums.add(3);  
nums.add(4);  
nums.add(5);  
System.out.println(nums); // [2, 3, 4, 5, 8]  
System.out.println(nums.contains(1)); //false
```

```
List<Integer> nums = new ArrayList<>();  
nums.add(5);  
nums.add(8);  
nums.add(2);  
nums.add(3);  
nums.add(3);  
nums.add(4);  
nums.add(5);  
System.out.println(nums); // [5, 8, 2, 3, 3, 4, 5]  
Set<Integer> nums2 = new HashSet<>(nums);  
System.out.println(nums2); // [2, 3, 4, 5, 8]
```

Ejemplos de uso de HashSet<E>

```
Set<String> nombres = new HashSet<>();
nombres.add("Pep");
nombres.add("Tom");
nombres.add("Jon");
nombres.add("Pep");
nombres.add("Kal");
nombres.add("Tom");
System.out.println(nombres); // [Tom, Kal, Jon, Pep]
nombres.remove(2); // no borra ningún nombre
nombres.remove("Tom"); // [Kal, Jon, Pep]
nombres.get(2); // ERROR, no tenemos el método get
```

```
Set<Alumno> dam = new HashSet<>();
dam.add(new Alumno("Pep", "222A", 25));
dam.add(new Alumno("Sam", "666A", 18));
dam.add(new Alumno("Kal", "777A", 20));

Iterator<Alumno> iteratorAlumnos = dam.iterator();
while(iteratorAlumnos.hasNext()) {
    iteratorAlumnos.next().setNombre("AAA");
}
// todos tendrán de nombre AAA
```


Clase HashSet<E> en JAVA

Ejemplo de uso de HashSet<E>

Si la clase Alumno no tuviera implementados los métodos **equals()** y **hashCode()**, se podrían insertar elementos duplicados(mismo estado). Esto se debe a que, por defecto, los métodos equals() y hashCode() heredados de la clase Object están basados en las referencias de los objetos y no en su estado.

```
public class Alumno {  
    private String nombre;  
    private String nia;  
    private int edad;  
    public Alumno(...) {...}  
    public String toString() {...}  
    //sin equals() y hashCode implementados  
}
```

```
Set<Alumno> dam = new HashSet<>();  
Alumno a1 = new Alumno("Pep", "222A", 25);  
Alumno a2 = new Alumno("Sam", "666A", 18);  
Alumno a3 = new Alumno("Sam", "666A", 18);  
Alumno a4 = new Alumno("Kal", "666A", 20);
```

```
dam.add(a1); //Pep  
dam.add(a2); //Pep, Sam  
dam.add(a3); //Pep, Sam, Sam  
dam.add(a4); //Pep, Sam, Sam, Kal
```

```
dam.add(a1); //No se inserta Pep, tienen la misma referencia  
dam.add(new Alumno("Sam", "666A", 18)); //Se inserta Sam, no tienen la misma referencia  
//Estado final de dam → Pep, Sam, Sam, Kal, Sam
```

Ejemplo de uso de HashSet<E>

Para verificar la existencia de duplicados en un HashSet, primero se compara el resultado del método **hashCode()**. Si los hash son diferentes, se considera que los objetos son diferentes y no se verifica más. Sin embargo, si los hash son iguales, se procede a utilizar el método **equals()** para determinar si los objetos son realmente iguales.

```
public class Alumno {  
    private String nombre;  
    private String nia;  
    private int edad;  
    public Alumno(...) {...}  
    public String toString() {...}  
    //con equals() y hashCode correctamente  
    //implementados en base a los 3 atributos  
}
```

```
Set<Alumno> dam = new HashSet<>();  
Alumno a1 = new Alumno("Pep", "222A", 25);  
Alumno a2 = new Alumno("Sam", "666A", 18);  
Alumno a3 = new Alumno("Sam", "666A", 18);  
Alumno a4 = new Alumno("Kal", "666A", 20);
```

```
dam.add(a1); //Pep  
dam.add(a2); //Pep, Sam  
dam.add(a3); //Pep, Sam → no inserta Sam de nuevo, hay un Alumno igual  
dam.add(a4); //Pep, Sam, Kal
```

```
dam.add(a1); //No se inserta Pep, tienen la misma referencia  
dam.add(new Alumno("Sam", "666A", 18)); //No inserta Sam, hay un Alumno igual  
//Estado final de dam → Pep, Sam, Kal
```

Ejemplo de uso de HashSet<E>

Para verificar la existencia de duplicados en un HashSet, primero se compara el resultado del método **hashCode()**. Si los hash son diferentes, se considera que los objetos son diferentes y no se verifica más. Sin embargo, si los hash son iguales, se procede a utilizar el método **equals()** para determinar si los objetos son realmente iguales.

```
public class Alumno {  
    private String nombre;  
    private String nia;  
    private int edad;  
    public Alumno(...) {...}  
    public String toString() {...}  
    //con equals() y hashCode correctamente  
    implementados en base al nia  
}
```

```
Set<Alumno> dam = new HashSet<>();  
Alumno a1 = new Alumno("Pep", "222A", 25);  
Alumno a2 = new Alumno("Sam", "666A", 18);  
Alumno a3 = new Alumno("Sam", "666A", 18);  
Alumno a4 = new Alumno("Kal", "666A", 20);
```

```
dam.add(a1); //Pep  
dam.add(a2); //Pep, Sam  
dam.add(a3); //Pep, Sam → no inserta Sam de nuevo, hay un Alumno con el mismo nia  
dam.add(a4); //Pep, Sam → no inserta Kal, hay un Alumno con el mismo nia
```

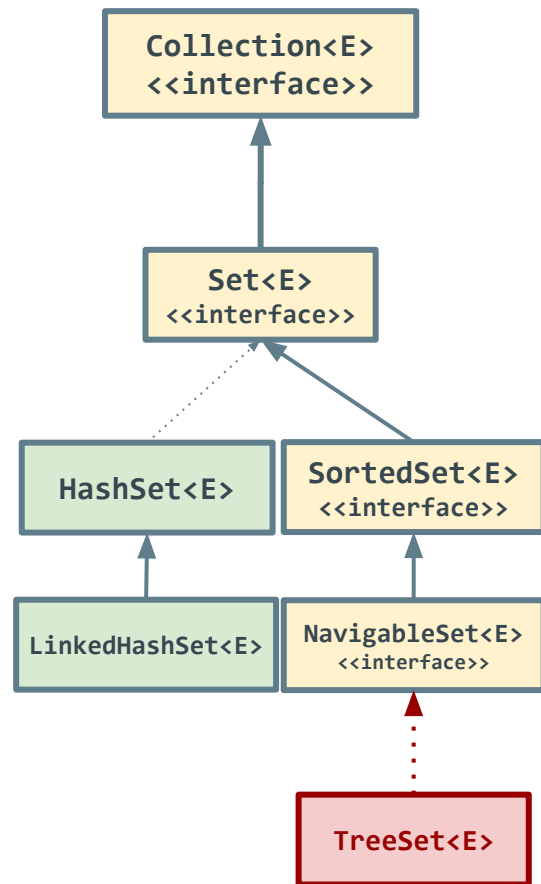
```
dam.add(a1); //No se inserta Pep, tienen la misma referencia  
dam.add(new Alumno("Sam", "777A", 18)); //Inserta Sam con nia(777A)  
//Estado final de dam → Pep, Sam(666A), Sam(777A)
```

Clase TreeSet<E> en JAVA

Clase TreeSet<E> en JAVA

La clase TreeSet en Java es una implementación de la interfaz SortedSet y también implementa la interfaz NavigableSet. Esta clase proporciona una estructura de datos basada en un árbol.

- **Ordenamiento:** TreeSet mantiene sus elementos en orden ascendente, según su orden natural o según el orden impuesto por un comparador. Si se proporciona un comparador, debe ser consistente con equals().
- **Elementos únicos:** Al igual que otras implementaciones de Set, TreeSet no permite elementos duplicados. Si se intenta agregar un elemento que ya está presente, el conjunto no se modifica y no se genera ninguna excepción.
- **Elementos no nulos:** TreeSet no permite elementos null. Intentar agregar un elemento null a un TreeSet arrojará una NullPointerException.
- **Rendimiento:** Las operaciones básicas (como agregar, eliminar y buscar elementos) tienen un tiempo de ejecución de $O(\log(n))$, gracias al árbol de búsqueda binaria autoequilibrado utilizado para almacenar los elementos.



Clase TreeSet<E> en JAVA

Además de los métodos de la interfaz Collection, TreeSet nos ofrece métodos adicionales para navegar y manipular el conjunto ordenado, algunos de los más utilizados son los siguientes:

- **E first():** Retorna el primer (más pequeño) elemento del conjunto. Arroja una NoSuchElementException si el conjunto está vacío.
- **E last():** Retorna el último (más grande) elemento del conjunto. Arroja una NoSuchElementException si el conjunto está vacío.
- **E lower(E e):** Retorna el elemento más grande en el conjunto que es menor que el elemento especificado, o null si no hay ninguno.
- **E floor(E e):** Retorna el elemento más grande en el conjunto que es menor o igual al elemento especificado, o null si no hay ninguno.
- **E ceiling(E e):** Retorna el elemento más pequeño en el conjunto que es mayor o igual al elemento especificado, o null si no hay ninguno.
- **E higher(E e):** Retorna el elemento más pequeño en el conjunto que es mayor que el elemento especificado, o null si no hay ninguno.
- **E pollFirst():** Elimina y retorna el primer (más pequeño) elemento del conjunto, o null si el conjunto está vacío.
- **E pollLast():** Elimina y retorna el último (más grande) elemento del conjunto, o null si el conjunto está vacío.

Comparator<T>, Comparable<T>, equals y hashCode en TreeSet<E>

En una colección de tipo TreeSet<E>, los métodos equals() y el Comparator() juegan roles importantes para garantizar la correcta organización y funcionamiento del conjunto

- El método **equals()** es utilizado para determinar si dos objetos son iguales en términos de su contenido o estado. En el caso de un TreeSet, si dos objetos son iguales según su método equals(), solo uno de ellos será almacenado en el conjunto, ya que un TreeSet no permite elementos duplicados.
- El método **hashCode()** es utilizado en estructuras de datos basadas en tablas hash, como HashSet y HashMap, para distribuir los elementos en distintos "buckets" según sus códigos hash, lo que permite una búsqueda y manipulación eficiente de los elementos. Sin embargo, en un TreeSet, la estructura es un árbol de búsqueda binaria autoequilibrado (árbol Red-Black), que no utiliza códigos hash para organizar los elementos, sino que depende del orden natural o del orden impuesto por un Comparator.
- **Comparator<T>** es una interfaz en Java que se utiliza para definir cómo se comparan dos objetos para determinar su orden. En un TreeSet, un Comparator es esencial para mantener un orden específico entre los elementos almacenados. Si no se proporciona un Comparator<T> personalizado al crear un TreeSet, se utilizará el orden natural de los elementos, que se determina mediante la implementación de la interfaz **Comparable<T>**.

Ejemplos sencillo TreeSet<E>

```
TreeSet<Integer> nums = new TreeSet<>();  
nums.add(5);  
nums.add(3);  
nums.add(1);  
nums.add(9);  
nums.add(2);  
nums.add(8);  
nums.add(7);  
nums.add(5); //no lo inserta
```

```
System.out.println(nums); // [1, 2, 3, 5, 7, 8, 9]  
System.out.println(nums.ceiling(4)); // 5  
System.out.println(nums.ceiling(5)); // 5  
System.out.println(nums.floor(4)); // 3  
System.out.println(nums.floor(5)); // 5  
System.out.println(nums.lower(8)); // 7  
System.out.println(nums.higher(2)); // 3  
System.out.println(nums.pollFirst()); // 1  
System.out.println(nums.pollLast()); // 9  
System.out.println(nums); // [2, 3, 5, 7, 8]
```

```
Set<String> nombres = new TreeSet<>();  
nombres.add("Pep");  
nombres.add("Tom");  
nombres.add("Sam");  
nombres.add("Ben");  
System.out.println(nombres);  
// [Ben, Pep, Sam, Tom]  
System.out.println(nombres.higher("Pep"));  
// ERROR, en teoría Sam, pero es un Set
```

Ejemplo de uso de TreeSet<E> con Comparable<T>: Clase Alumno

```
public class Alumno implements Comparable<Alumno> {
    private String nombre;
    private String nia;
    private int edad;
    public Alumno(...) {...}
    public String toString() {...}

    public int compareTo(Alumno a) {
        int comparacion = Integer.compare(this.edad, a.getEdad());
        if (comparacion == 0)
            comparacion = nombre.compareTo(a.getNombre());
        return comparacion;
    }
    public boolean equals(Object obj) {
        if (this == obj) return true;
        if (obj == null) return false;
        if (getClass() != obj.getClass()) return false;
        Alumno other = (Alumno) obj;
        if (nia == null && other.nia != null) return false;
        else if (!nia.equals(other.nia)) return false;
        return true;
    }
}
```

Ejemplo de uso de TreeSet<E> con Comparable<T>

En el caso de TreeSet, si no se proporciona un comparador externo, utiliza el método compareTo para comparar y ordenar los elementos, y también para verificar si ya existen elementos duplicados en el conjunto. Según el ejemplo de la diapositiva anterior, se ordenan de menor a mayor edad y luego por nombre. Además, el método equals() está programado para que no permita alumnos con el mismo NIA. Sin embargo, la implementación no funciona como debe. Veamos un ejemplo:

```
TreeSet<Alumno> dam = new TreeSet<>();
```

```
Alumno a1 = new Alumno("Pep", "222A", 25);  
Alumno a2 = new Alumno("Sam", "555A", 18);  
Alumno a3 = new Alumno("Pol", "666A", 18);  
Alumno a4 = new Alumno("Kal", "666A", 20);  
Alumno a5 = new Alumno("Tim", "777A", 20);  
Alumno a6 = new Alumno("Pep", "222A", 28);
```

```
dam.add(a1); //Pep  
dam.add(a2); //Sam, Pep  
dam.add(a3); //Pol, Sam, Pep  
dam.add(a4); //Pol, Sam, Kal, Pep → inserta a Kal, aunque no debería  
dam.add(a5); //Pol, Sam, Kal, Tim, Pep  
dam.add(a6); //Pol, Sam, Kal, Tim, Pep(25), Pep(28) → inserta a Pep, aunque no debería
```

Ejemplo de uso de TreeSet<E> con Comparable<T>

Una solución sería, que Comparable<T> únicamente compare la edad, sin tener en cuenta el nombre. Sin embargo, no nos permitirá insertar alumnos con la misma edad.

```
public class Alumno implements Comparable<Alumno> {  
    //...  
    public int compareTo(Alumno a) {  
        return Integer.compare(this.edad, a.edad);  
    }  
}
```

```
TreeSet<Alumno> dam = new TreeSet<>();  
Alumno a1 = new Alumno("Pep", "222A", 25);  
Alumno a2 = new Alumno("Sam", "555A", 18);  
Alumno a3 = new Alumno("Pol", "666A", 18);  
Alumno a4 = new Alumno("Kal", "666A", 20);  
Alumno a5 = new Alumno("Tim", "777A", 20);  
Alumno a6 = new Alumno("Pep", "222A", 28);  
dam.add(a1); //Pep  
dam.add(a2); //Sam, Pep  
dam.add(a3); //Sam, Pep → NO inserta a Pol, hay un alumno con la misma edad  
dam.add(a4); //Sam, Kal, Pep  
dam.add(a5); //Sam, Kal, Pep → NO inserta a Tim, hay un alumno con la misma edad  
dam.add(a6); //Sam, Kal, Pep(25), Pep(28)
```

Ejemplo de uso de TreeSet<E> con Comparable<T>

La siguiente implementación de compareTo lo soluciona para todos los casos:

```
public int compareTo(Alumno a) {  
  
    if(nia.compareTo(a.getNia()) == 0) return 0;  
    int comparacion = Integer.compare(edad, a.getEdad());  
    if (comparacion == 0) comparacion = nia.compareTo(a.getNia());  
    return comparacion;  
}
```

```
TreeSet<Alumno> dam = new TreeSet<>();  
Alumno a1 = new Alumno("Pep", "222A", 25);  
Alumno a2 = new Alumno("Sam", "555A", 18);  
Alumno a3 = new Alumno("Pol", "666A", 18);  
Alumno a4 = new Alumno("Kal", "666A", 20);  
Alumno a5 = new Alumno("Tim", "777A", 20);  
Alumno a6 = new Alumno("Pep", "222A", 28);  
dam.add(a1); //Pep  
dam.add(a2); //Sam, Pep  
dam.add(a3); //Sam, Pol, Pep  
dam.add(a4); //Sam, Pol, Pep → NO inserta a Kal, hay un alumno con el mismo nia  
dam.add(a5); //Sam, Pol, Tim, Pep  
dam.add(a6); //Sam, Pol, Tim, Pep → NO inserta a Pep, hay un alumno con el mismo nia
```

Ejemplo de uso de TreeSet<E> con Comparator<T>

Si al crear la instancia del TreeSet le pasamos al constructor como parámetro, una instancia de una clase que implementa la interface Comparator<T>. Será dicha implementación la que tendrá prioridad a la hora de ordenar los elementos. Por ejemplo:

```
public class AlumnoPorEdadComparator implements Comparator<Alumno> {  
    public int compare(Alumno a1, Alumno a2) {  
        if(a1.getNia().compareTo(a2.getNia()) == 0) return 0;  
        int comparacion = Integer.compare(a1.getEdad(), a2.getEdad());  
        if (comparacion == 0) comparacion = a1.getNia().compareTo(a2.getNia());  
        return comparacion;  
    }  
}
```

```
TreeSet<Alumno> dam = new TreeSet<>(new AlumnoPorEdadComparator());  
Alumno a1 = new Alumno("Pep", "222A", 25);  
Alumno a2 = new Alumno("Sam", "555A", 18);  
Alumno a3 = new Alumno("Pol", "666A", 18);  
Alumno a4 = new Alumno("Kal", "666A", 20);  
Alumno a5 = new Alumno("Tim", "777A", 20);  
Alumno a6 = new Alumno("Pep", "222A", 28);  
dam.add(a1); //Pep dam.add(a2); //Sam, Pep dam.add(a3); //Sam, Pol, Pep  
dam.add(a4); //Sam, Pol, Pep → NO inserta a Kal, hay un alumno con el mismo nia  
dam.add(a5); //Sam, Pol, Tim, Pep  
dam.add(a6); //Sam, Pol, Tim, Pep → NO inserta a Pep, hay un alumno con el mismo nia
```

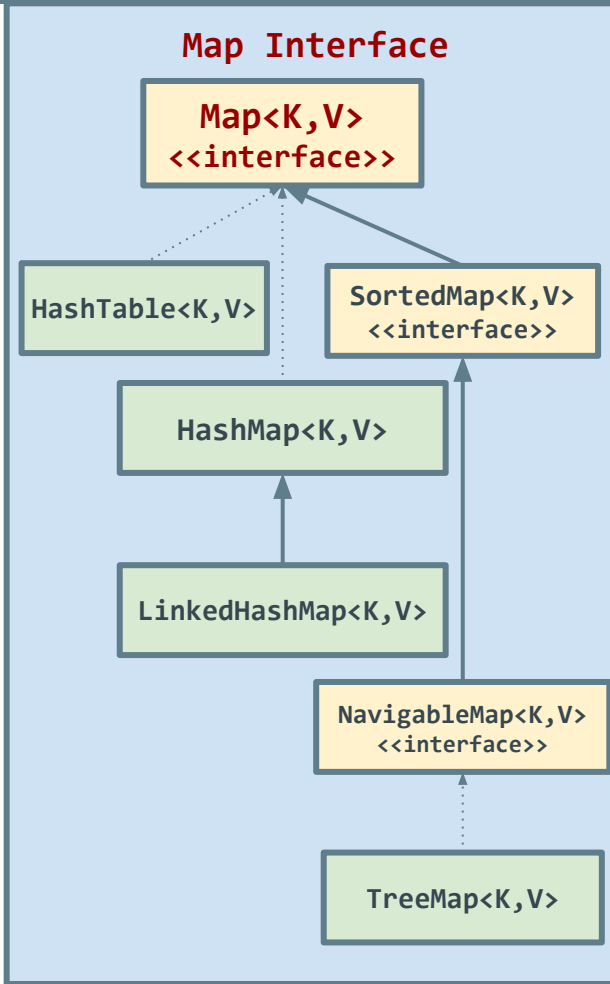
Interface Map<K, V> en JAVA

Interface Map<K,V> en JAVA

La interfaz Map<K, V> en Java es parte del Java Collections Framework (JCF) y representa una estructura de datos que almacena pares clave-valor. A diferencia de la interfaz Collection<E>, Map no extiende ni está relacionada con Collection directamente.

En un objeto Map, cada clave está asociada con un valor, y las claves deben ser únicas. Los valores, por otro lado, pueden repetirse. Los parámetros de tipo genérico <K> y <V>, representan el tipo de las claves y los valores respectivamente. Al implementar la interfaz Map, se puede especificar el tipo de las claves y los valores que se almacenarán en el mapa. Por ejemplo, un Map<String, Integer> almacenaría claves de tipo String y valores de tipo Integer.

```
Map<Character, Integer> letras = new HashMap<>();  
Map<String, Integer> palabras = new HashMap<>();  
Map<Alumno, Double> notasPrg = new LinkedHashMap<>();  
Map<Alumno, ArrayList<Double>> notasDam = new TreeMap<>();  
Map<Alumno, ArrayList<Calificacion>> notasDam = new TreeMap<>();
```



Implementaciones de la Interface Map<K,V> en JAVA

Hay varias implementaciones comunes de la interfaz Map<K, V> en el JCF. Algunas de las más utilizadas son, HashMap, LinkedHashMap, TreeMap y ConcurrentHashMap

- **HashMap** es una implementación de Map que utiliza una tabla hash para almacenar los pares clave-valor. No garantiza ningún orden específico de los elementos y ofrece un rendimiento constante en tiempo de ejecución para las operaciones básicas (agregar, eliminar, buscar) en el caso promedio. HashMap permite claves y valores nulos.
- **LinkedHashMap** es una extensión de HashMap que mantiene el orden de inserción de los pares clave-valor. Utiliza una estructura de datos de lista enlazada junto con la tabla hash para mantener el orden de inserción. El rendimiento de LinkedHashMap es ligeramente inferior al de HashMap debido a la sobrecarga adicional para mantener el orden de inserción.
- **TreeMap** es una implementación de Map basada en un árbol binario de búsqueda balanceado (red-black tree). Mantiene los pares clave-valor en un orden naturalmente ordenado (según el método compareTo() de las claves o un Comparator proporcionado). TreeMap no permite claves nulas y ofrece un rendimiento de tiempo logarítmico para las operaciones básicas.
- **HashTable** es una implementación de la interfaz Map<K, V> en Java que utiliza una tabla hash para almacenar pares clave-valor. Es una clase que está desde la versión 1.0 de Java y se considera obsoleta, su uso no se recomienda en favor de la implementación más reciente y flexible HashMap<K, V>.

Métodos Interface Map<K,V> en JAVA

- **V put(K key, V value)**: Asocia un valor con una clave en el mapa. Si la clave ya existe, el valor antiguo se reemplaza por el nuevo valor. Si la clave no existe, se agrega un nuevo par clave-valor al mapa.
- **V get(K key)**: Devuelve el valor asociado con la clave especificada. Si la clave no existe en el mapa, retorna null.
- **V remove(K key)**: Elimina el par clave-valor asociado con la clave especificada y devuelve el valor eliminado. Si la clave no existe en el mapa, retorna null.
- **boolean containsKey(K key)**: Verifica si el mapa contiene un par clave-valor con la clave especificada.
- **boolean containsValue(V value)**: Verifica si el mapa contiene uno o más pares clave-valor con el valor especificado.
- **clear()**: Elimina todos los pares clave-valor del mapa.
- **V getOrDefault(K key, V defaultValue)**: se utiliza para recuperar el valor asociado a una clave específica en el mapa. Si la clave no está presente en el mapa, el método devuelve un valor predeterminado especificado.
- **Set<K> keySet()**: Devuelve un objeto Set<K> que contiene todas las claves del mapa.
- **Collection<V> values()**: Devuelve una colección de tipo Collection<V> que contiene todos los valores del mapa.
- **Set<Map.Entry<K,V>> entrySet()**: Devuelve un objeto Set<Map.Entry<K, V>> que contiene todos los pares clave-valor del mapa como objetos de tipo Map.Entry<K, V>.

Ejemplo de uso de Map<String, Double>

A continuación podemos ver un ejemplo de uso de Map, para almacenar notas asociadas a un alumno:

```
Map<String, Double> notasDeAlumnos = new HashMap<>();

// Añadimos las notas de los alumnos
notasDeAlumnos.put("Tim", 9.7);
notasDeAlumnos.put("Bob", 8.5);
notasDeAlumnos.put("Jon", 7.8);
notasDeAlumnos.put("Bob", 8.8);
notasDeAlumnos.put("Bob", notasDeAlumnos.getDefault("Bob", 0.0) + 1); //Bob → 9.8
notasDeAlumnos.put("Jon", notasDeAlumnos.getDefault("Kal", 5.0) + 1); //Jon → 6.0
notasDeAlumnos.put("Kal", notasDeAlumnos.getDefault("Bob", 5.0)); //Kal → 9.8
notasDeAlumnos.put("Kal", notasDeAlumnos.getDefault("Sam", 0.0)); //Kal → 0.0

// Mostramos datos con entrySet()
System.out.println("Notas alumnos:");
for (Map.Entry<String, Double> pares : notasDeAlumnos.entrySet()) {
    System.out.println("La nota de " + pares.getKey() + " es " + pares.getValue());
}

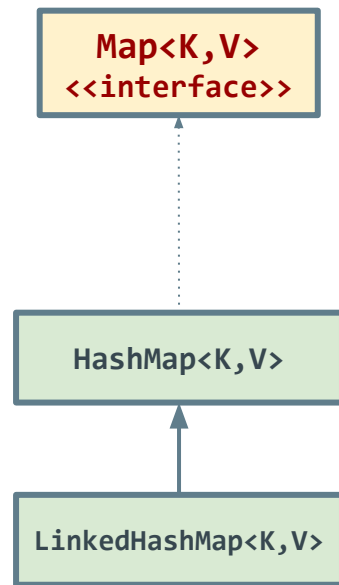
// Mostramos nombres de los alumnos con keySet() y nota media
System.out.println("Alumnos:"+notasDeAlumnos.keySet()); //Alumnos:[Bob, Kal, Jon, Tim]
double sumaNotas = 0;
for (Double nota : notasDeAlumnos.values()) sumaNotas += nota;
System.out.println("Nota media: " + sumaNotas / notasDeAlumnos.size()); //Nota media: 6.375
```

Clase HashMap<K, V> en JAVA

Clase HashMap<K,V> en JAVA

La clase HashMap en Java es una implementación de la interfaz Map que utiliza una tabla hash para almacenar pares clave-valor.

- **Ordenamiento:** no garantiza ningún orden específico al iterar sus elementos. El orden en que los elementos se devuelven al iterar sobre las claves, valores o entradas del HashMap puede variar y no está relacionado con el orden de inserción.
- **Claves únicas:** en todas las implementaciones de Map, se debe garantizar que no existirán claves repetidos. De hecho, no tenemos el típico método add(). Para agregar elementos lo debemos hacer a través de put(...) o putIfAbsent(...)
- **Elementos nulos:** al no permitir duplicados, el HashMap únicamente permite una clave nula, y múltiples valores nulos.
- **Métodos:** no tiene métodos exclusivos. Sin embargo al implementar la interface Map, tendrá disponibles todos los métodos de dicha interface.
- **Rendimiento:** ofrece un rendimiento de tiempo constante ($O(1)$) para las operaciones básicas (get, put, remove) en el mejor y el caso promedio.



Ejemplo de uso de HashMap<Character, Integer>

A continuación podemos ver un ejemplo de uso de Map, para obtener la cantidad de veces que aparece cada letra en una String:

```
String s = "Cadena de ejemplo!!\nHoy es miércoles día 29 de marzo de 2023\n\nFIN";

// Utilizamos HashMap para almacenar los caracteres y sus frecuencias
Map<Character, Integer> letras = new HashMap<>();

// Convertimos la cadena de entrada a minúsculas y la recorremos
for (char letra : s.toLowerCase().toCharArray()) {
    // Verificamos si el carácter es una letra
    if (Character.isLetter(letra)) {
        // Incrementamos la frecuencia del carácter en el mapa, o lo iniciamos a 0
        letras.put(letra, letras.getDefault(letra, 0) + 1);
    }
}

// Mostramos los resultados
System.out.println("Frecuencia de caracteres en la cadena de entrada:");
for (Map.Entry<Character, Integer> claveValor : letras.entrySet()) {
    System.out.println(claveValor.getKey() + ": " + claveValor.getValue());
}
```

Ejemplo de uso de HashMap<String, Integer>

A continuación podemos ver un ejemplo de uso de Map, para obtener la cantidad de veces que aparece cada palabra en una String:

```
String s = "Cadena    de ejemplo!!\nHoy    es miércoles día 29    de marzo de 2023\n\nFIN";

// Utilizamos HashMap para almacenar las palabras y sus frecuencias
Map<String, Integer> frecuenciaPalabras = new HashMap<>();

// Convertimos a minúsculas y dividimos la cadena en palabras con split
String[] palabras = s.toLowerCase().split("\\s+");

// Recorremos el array de palabras
for (String palabra : palabras) {
    // Incrementamos la frecuencia de la palabra en el mapa
    frecuenciaPalabras.put(palabra, frecuenciaPalabras.getOrDefault(palabra, 0) + 1);
}

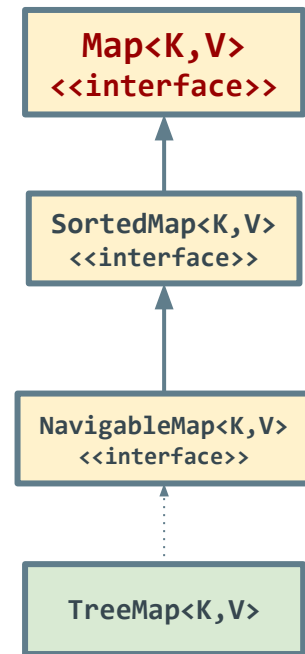
// Imprimimos los resultados
System.out.println("Frecuencia de palabras en la cadena de entrada:");
for (Map.Entry<String, Integer> claveValor : frecuenciaPalabras.entrySet()) {
    System.out.println(claveValor.getKey() + ": " + claveValor.getValue());
}
```

Clase TreeMap<K, V> en JAVA

Clase TreeMap<K,V> en JAVA

La clase TreeMap en Java es una implementación de la interfaz Map que utiliza un árbol binario autobalanceado(red-black) para almacenar pares clave-valor.

- **Ordenamiento:** A diferencia de HashMap, TreeMap almacena sus elementos de forma ordenada. El ordenamiento se realiza utilizando la clave de cada par clave-valor. Por defecto, las claves se ordenan según su orden natural, pero se puede proporcionar un comparador personalizado al constructor de TreeMap para definir un orden específico.
- **Claves únicas:** en todas las implementaciones de Map, se debe garantizar que no existirán claves repetidos.
- **Elementos nulos:** A diferencia de HashMap, TreeMap no permite claves nulas. Intentar insertar una clave nula resultará en una NullPointerException. Sin embargo, TreeMap permite valores nulos.
- **Métodos:** tiene métodos exclusivos al implementar las interfaces SortedMap y NavigableMap.
- **Rendimiento:** Un árbol Red-Black es un árbol de búsqueda binaria balanceado que garantiza que no haya ningún camino en el árbol que sea más del doble de largo que cualquier otro. Esta propiedad asegura que las operaciones básicas en el árbol tienen una complejidad de tiempo logarítmica.



Métodos clase TreeMap<K,V> en JAVA

Métodos de SortedMap:

- **K firstKey()**: Devuelve la primera (mínima) clave en el mapa ordenado.
- **K lastKey()**: Devuelve la última (máxima) clave en el mapa ordenado.
- **SortedMap<K, V> headMap(K key)**: Devuelve los elementos con las claves menores a key.
- **SortedMap<K, V> tailMap(K key)**: Devuelve los elementos con las claves mayores o iguales a key.
- **SortedMap<K, V> subMap(K fromKey, K toKey)**: Devuelve los elementos del mapa cuyas claves están en el rango desde fromKey hasta toKey.

Métodos de NavigableMap:

- **K lowerKey(K key)**: Devuelve la clave más grande, que es menor que la clave key.
- **K higherKey(K key)**: Devuelve la clave más pequeña, que es mayor que la clave key.
- **K floorKey(K key)**: Devuelve la clave más grande que es menor o igual a la clave key.
- **K ceilingKey(K key)**: Devuelve la clave más pequeña que es mayor o igual a la clave key.
- **Map.Entry<K,V> pollFirstEntry()**: Elimina y devuelve el par con clave mínima, null si el map está vacío.
- **Map.Entry<K,V> pollLastEntry()**: Elimina y devuelve el para con clave máxima, null si el map está vacío.
- **NavigableMap<K,V> descendingMap()**: Devuelve una vista del mapa en orden descendente.
- **NavigableSet<K> navigableKeySet()**: Devuelve un NavigableSet de las claves del mapa.
- **NavigableSet<K> descendingKeySet()**: Devuelve un NavigableSet de las claves del mapa en orden descendente.

Ejemplo de uso de TreeMap<String, Double>

A continuación podemos ver un ejemplo de uso de TreeMap, para almacenar notas asociadas a nombres:

```
TreeMap<String, Double> notas = new TreeMap<>();
notas.put("Pol", 9.8);           // {Pol=9.8}
notas.put("Pep", 8.3);          // {Pep=8.3, Pol=9.8}
notas.put("Tom", 7.0);          // {Pep=8.3, Pol=9.8, Tom=7.0}
notas.put("Sam", 9.6);          // {Pep=8.3, Pol=9.8, Sam=9.6, Tom=7.0}
notas.put("Pol", 6.5);          // {Pep=8.3, Pol=6.5, Sam=9.6, Tom=7.0}
notas.put("Kal", 9.1);          // {Kal=9.1, Pep=8.3, Pol=6.5, Sam=9.6, Tom=7.0}
System.out.println("Primero: " + notas.firstKey());      //Kal
System.out.println("Último: " + notas.lastKey());        //Tom
System.out.println("Antes de 'Sam': " + notas.headMap("Sam")); //{Kal=9.1,Pep=8.3,Pol=6.5}
System.out.println("Después de 'Sam': " + notas.tailMap("Sam")); //{Sam=9.6, Tom=7.0}
System.out.println("Entre 'Kal' y 'Pol': " + notas.subMap("Kal", "Pol")); //{Kal=9.1, Pep=8.3}
System.out.println("Anterior a 'Pol': " + notas.lowerKey("Pol")); //Pep
System.out.println("Anterior a 'Kal': " + notas.lowerKey("Kal")); //null
System.out.println("Anterior a 'Tim': " + notas.lowerKey("Tim")); //Sam
System.out.println("Posterior a 'Pol': " + notas.higherKey("Pol")); //Sam
```

Ejemplo de uso de Map<Alumno, ArrayList<Double>>

A continuación podemos ver un ejemplo de uso de **TreeMap** que permite almacenar varias notas asociadas a una instancia de la clase Alumno:

Se deberá tener en cuenta el método **compareTo()** para determinar si dos Alumnos son iguales. En el ejemplo, como el comparador impide insertar alumnos con el mismo NIA, no se podrá insertar a Kal, y la nota se agrega al alumno con el NIA de Kal. Además el **compareTo()** se encargará de mantener el orden a medida que se insertan nuevos alumnos, en este caso por edad.

```
Map<Alumno, ArrayList<Double>> notasDam = new TreeMap<>();

Alumno a1 = new Alumno("Pep", "111A", 20);
Alumno a2 = new Alumno("Jon", "222A", 18);
Alumno a3 = new Alumno("Sam", "333A", 21);
Alumno a4 = new Alumno("Bil", "444A", 19);
Alumno a5 = new Alumno("Kal", "111A", 22);

agregarNota(notasDam, a1, 9.2); //Pep
agregarNota(notasDam, a1, 7.5); //Pep
agregarNota(notasDam, a2, 8.1); //Jon, Pep
agregarNota(notasDam, a2, 9.0); //Jon, Pep
agregarNota(notasDam, a3, 7.5); //Jon, Pep, Sam
agregarNota(notasDam, a4, 8.2); //Jon, Bil, Pep, Sam
agregarNota(notasDam, a5, 10.0); //se agrega un 10 a Pep
```

// Método para añadir notas de los alumnos al map

```
public static void agregarNota(Map<Alumno, ArrayList<Double>> notasDam, Alumno a, double nota) {
    ArrayList<Double> notas = notasDam.getOrDefault(a, new ArrayList<>());
    notas.add(nota);
    notasDam.put(a, notas);
}
```

Ejemplo de uso de Map<Alumno, Map<String, ArrayList<Double>>>

A continuación podemos ver un ejemplo de uso de **TreeMap** que a cada instancia de la clase Alumno, permite almacenar de distintos módulos, con varias notas asociadas a cada módulo:

```
Map<Alumno, Map<String, ArrayList<Double>>> notasAlumnos = new TreeMap<>();
Alumno alumno1 = new Alumno("Pep", "111A", 20);
agregarNota(notasAlumnos, alumno1, "Programación", 8.5);
agregarNota(notasAlumnos, alumno1, "Programación", 7.1);
agregarNota(notasAlumnos, alumno1, "Bases de datos", 9.0);
```

```
public static void agregarNota(Map<Alumno, Map<String, ArrayList<Double>>>
notasAlumnos, Alumno alumno, String modulo, double nota) {
    Map<String, ArrayList<Double>> notasAlumno = notasAlumnos.get(alumno);
    if (notasAlumno == null) {
        notasAlumno = new HashMap<>();
        notasAlumnos.put(alumno, notasAlumno);
    }
    ArrayList<Double> notasModulo = notasAlumno.get(modulo);
    if (notasModulo == null) {
        notasModulo = new ArrayList<>();
        notasAlumno.put(modulo, notasModulo);
    }
    notasModulo.add(nota);
}
```

Ejemplo de uso de Map<Alumno, ArrayList<Calificacion>>

Implementación alternativa utilizando la clase auxiliar Calificación, y una clase enumerada para representar los módulos.

```
Map<Alumno, ArrayList<Calificacion>> notas = new TreeMap<>();
Alumno a = new Alumno("Pep", "111A", 20);
agregarNota(a, Modulo.PROGRAMACION, 8.5);
agregarNota(a, Modulo.PROGRAMACION, 9.0);
agregarNota(a, Modulo.BASES_DE_DATOS, 7.5);
```

```
public class Calificacion {
    private Modulo modulo;
    private ArrayList<Double> notas;

    public Calificacion(Modulo modulo) {
        this.modulo = modulo;
        this.notas = new ArrayList<>();
    }

    public Modulo getModulo() {
        return modulo;
    }

    public ArrayList<Double> getNotas() {
        return notas;
    }
}
```


```
public enum Modulo {

    PROGRAMACION("PRG", "Programación"),
    BASES_DE_DATOS("BDA", "Bases de Datos");
    private String nombre, abreviatura;

    Modulo(String abreviatura, String nombre) {
        this.abreviatura = abreviatura;
        this.nombre = nombre;
    }

    public String getAbreviatura() {
        return abreviatura;
    }

    public String getNombre() {
        return nombre;
    }
}
```



Ejemplo de uso de Map<Alumno, ArrayList<Calificacion>>

A continuación podemos ver la implementación del método agregarNota(...), dicho método nos permitirá agregar notas a un alumno especificando un módulo que siempre será válido.

```
static Map<Alumno, ArrayList<Calificacion>> notasAlumnos = new TreeMap<>();
//main {...}
public static void agregarNota(Alumno alumno, Modulo modulo, double nota) {
    ArrayList<Calificacion> calificacionesAlumno = notasAlumnos.get(alumno);
    if (calificacionesAlumno == null) {
        calificacionesAlumno = new ArrayList<>();
        notasAlumnos.put(alumno, calificacionesAlumno);
    }
    Calificacion calificacion = null;
    for (Calificacion c : calificacionesAlumno) {
        if (c.getModulo() == modulo) {
            calificacion = c;
            break;
        }
    }
    if (calificacion == null) {
        calificacion = new Calificacion(modulo);
        calificacionesAlumno.add(calificacion);
    }
    calificacion.getNotas().add(nota);
}
```

Documentación y uso de APIs en JAVA:

API de JAVA

Una API, o Interfaz de Programación de Aplicaciones, es un conjunto de reglas y especificaciones que permiten a los desarrolladores de software interactuar con otros sistemas, servicios o bibliotecas de manera estandarizada. Las API actúan como puentes de comunicación entre diferentes componentes de software, simplificando el proceso de intercambio de información y funcionalidades.

Las APIs son esenciales para facilitar el desarrollo de software, ya que permiten a los desarrolladores utilizar clases y métodos que son utilizados con frecuencia en todo tipo de aplicaciones. Dichos métodos han sido perfectamente diseñados y testeados para ofrecer un funcionamiento óptimo en todo tipo de situaciones. Esto ahorra tiempo y esfuerzo, y permite a los desarrolladores centrarse en la lógica y las funciones específicas de sus propias aplicaciones.

La API oficial de JAVA es el conjunto de librerías y recursos proporcionados para desarrollar aplicaciones. La API es parte del Java Development Kit (JDK) y consta de numerosas clases y paquetes organizados en librerías como de utilidad, de entrada/salida, redes, gráficos, etc.

Para obtener más información sobre la API oficial de Java y cómo usarla en tus proyectos, puedes visitar la documentación oficial en el sitio web de Oracle:

<https://docs.oracle.com/en/java/javase/>

Principales librerías de la API de JAVA

- **java.lang:** Contiene clases fundamentales como Object, String, Math, System, y las clases de envoltorio para tipos primitivos (Integer, Double, etc.).
- **java.util:** Incluye colecciones (List, Set, Map), utilidades de fecha y hora (Date, Calendar, TimeZone), clases para generar números aleatorios (Random), y otras utilidades generales.
- **java.io y java.nio:** Proporciona clases para entrada y salida de datos, como FileReader, FileWriter, BufferedReader, BufferedWriter, y clases para trabajar con archivos y directorios (File, FileInputStream, FileOutputStream)
- **java.net:** Contiene clases para trabajar con redes y comunicaciones, como Socket, ServerSocket, URL, HttpURLConnection, y InetAddress.
- **java.awt y javax.swing:** Proporciona un conjunto básico de componentes de interfaz gráfica de usuario (GUI). Swing: Amplía java.awt con un conjunto más avanzado de componentes como JFrame, JPanel, JButton, JLabel, etc.
- **java.sql:** Incluye clases para trabajar con bases de datos a través del estándar JDBC (Java Database Connectivity), como DriverManager, Connection, PreparedStatement, ResultSet, etc.
- **java.time:** Introduce una nueva API de fecha y hora más completa y fácil de usar que reemplaza las clases antiguas de java.util (LocalDate, LocalTime, LocalDateTime, Duration, Period, etc.).
- **java.security:** Contiene clases para realizar tareas relacionadas con la seguridad, como generación de claves, firmas digitales, cifrado y descifrado, y control de acceso.

Ejemplo de uso de API

Puedes utilizar cualquier librería/clase de la API de Java importando la clase o clases correspondientes a través de **import**, en este ejemplo podemos hacer:

import java.util.* ; importa todas las clases e interfaces del paquete java.util.

import java.util.ArrayList ;

importa solo la clase ArrayList del paquete java.util.

En general, se recomienda utilizar importaciones específicas en lugar de comodines para mantener un código más limpio y evitar posibles conflictos.

```
import java.util.ArrayList;

public class Ejemplo1ArrayList {
    public static void main(String[] args) {
        ArrayList<String> nums = new ArrayList<>();
        nums.add("Uno");
        nums.add("Dos");
    }
}
```

```
import java.util.*;

public class Ejemplo2ArrayList {
    public static void main(String[] args) {
        ArrayList<String> nums = new ArrayList<>();
        nums.add("Uno");
        nums.add("Dos");
    }
}
```

Ejemplo documentación en JAVA

Árbol de herencia de la clase, la clase Number hereda directamente de Object

Integer

Otras APIs y Frameworks

Existen otras APIs de JAVA populares y ampliamente utilizadas. Aunque no forman parte de la API oficial de JAVA, estas bibliotecas de terceros son fundamentales en el ecosistema de JAVA y pueden facilitar el desarrollo en diversas áreas:

- **Spring Framework:** Marco de trabajo integral para el desarrollo de aplicaciones Java empresariales que ofrece inversion of control (IoC), aspect-oriented programming (AOP), integración con bases de datos, seguridad, y mucho más.
- **Hibernate:** Framework de mapeo objeto-relacional (ORM) que facilita el trabajo con bases de datos relacionales en aplicaciones Java, al permitir que los desarrolladores se centren en la lógica de negocio en lugar de en el código SQL.
- **JavaFX:** JavaFX es un marco para desarrollar aplicaciones de escritorio y aplicaciones web ricas en contenido multimedia. JavaFX ofrece una serie de características, como una API de gráficos en 2D y 3D, animaciones, controles de interfaz de usuario avanzados, soporte para CSS y diseño, y una API de medios para reproducir audio y video. A partir de Java 11, JavaFX fue separado del JDK y se convirtió en un proyecto de código abierto independiente.
- **Maven:** Herramientas de construcción y gestión de proyectos para Java que permiten a los desarrolladores organizar, compilar, probar e implementar sus aplicaciones de manera eficiente.
- **ElasticSearch:** Motor de búsqueda y análisis distribuido que permite a las aplicaciones Java indexar, buscar y analizar grandes volúmenes de datos en tiempo real.

Ejemplos de uso de la API de JAVA

java.time

Principales clases de la librería java.time

El paquete **java.time** es parte de la API de fecha y hora de Java, que fue introducida en JAVA 8. Nos ofrece una amplia gama de clases para trabajar con fechas, horas, períodos y zonas horarias de una manera más clara y flexible que las clases de fecha y hora anteriores (`java.util.Date` y `java.util.Calendar`).

- **LocalDate**: Representa una fecha (año, mes, día) sin información de hora o zona horaria. Es útil cuando solo necesitas representar fechas, como cumpleaños o fechas de vencimiento.
- **LocalTime**: Representa una hora del día (hora, minuto, segundo, nanosegundo) sin información de fecha o zona horaria.
- **LocalDateTime**: Representa una fecha y hora (año, mes, día, hora, minuto, segundo, nanosegundo) sin información de zona horaria.
- **Instant**: Representa un punto específico en el tiempo, medido hasta los nanosegundos.
- **Period**: Representa un período de tiempo en términos de años, meses y días (por ejemplo, 2 años, 3 meses y 5 días).
- **Duration**: Representa una duración de tiempo en términos de segundos y nanosegundos (por ejemplo, 75 segundos y 500 nanosegundos).
- **DateTimeFormatter**: Proporciona funcionalidades para formatear y analizar fechas y horas. Permite personalizar la representación textual de fechas y horas y convertir entre objetos de fecha y hora y sus representaciones de cadena.

Ejemplos de uso de la librería java.time

Ejemplo que utiliza métodos de la clase **Instant** y **Duration** en JAVA para calcular la cantidad de tiempo que ha pasado desde una fecha con una hora en concreto. Para convertir a años y meses no es la opción adecuada.

```
// Fecha y hora de nacimiento
LocalDateTime fechaHoraNacimiento = LocalDateTime.of(1981, 9, 1, 8, 0); // 1 sep 1981, 08:00

// Convertir la fecha y hora de nacimiento a un Instant
Instant nacimientoInstant = fechaHoraNacimiento.atZone(ZoneId.systemDefault()).toInstant();

// Obtener el Instant actual
Instant ahoraInstant = Instant.now();

// Calcular la duración desde el Instant de nacimiento hasta el Instant actual
Duration duracionDesdeNacimiento = Duration.between(nacimientoInstant, ahoraInstant);

// Duración a diferentes unidades de tiempo
DecimalFormat df = new DecimalFormat("#,###");
System.out.println("En días: " + df.format(duracionDesdeNacimiento.toDays()));
System.out.println("En horas: " + df.format(duracionDesdeNacimiento.toHours()));
System.out.println("En minutos: " + df.format(duracionDesdeNacimiento.toMinutes()));
System.out.println("En segundos: " + df.format(duracionDesdeNacimiento.toSeconds()));
```


Ejemplos de uso de la librería java.time

Ejemplo que utiliza métodos de la clase **Period** y **Duration** en JAVA para calcular la cantidad de tiempo que ha pasado desde que naciste. Adecuado para calcular períodos de años y meses.

```
// Fecha y hora de nacimiento
LocalDate fechaNacimiento = LocalDate.of(1981, 9, 1);
LocalTime horaNacimiento = LocalTime.of(8, 0);
// Fecha y hora actual
LocalDateTime fechaHoraActual = LocalDateTime.now();
// Calcular el período desde la fecha de nacimiento hasta la fecha actual
Period p = Period.between(fechaNacimiento, fechaHoraActual.toLocalDate());
// Calcular la duración desde la hora de nacimiento hasta la hora actual en el último día
Duration d = Duration.between(horaNacimiento, fechaHoraActual.toLocalTime());
if (d.isNegative()) {
    p = p.minusDays(1);
    d = d.plusDays(1);
}
// Convertir la duración a años, meses, días, horas, minutos y segundos
long horas = d.toHours();
long minutos = d.toMinutes() % 60;
long segundos = d.getSeconds() % 60;
System.out.println("Han pasado " + p.getYears() + " años, "
    + p.getMonths() + " meses, " + p.getDays() + " días, " + horas
    + " horas, " + minutos + " minutos y " + segundos + " segundos desde tu nacimiento.");
```

Bibliografía:

Allen Weiss, M. (2007). *Estructuras de datos en JAVA*. Madrid: Pearson

Froufe Quintas, A. (2002). *JAVA 2: Manual de usuario y tutorial*. Madrid: RA-MA

J. Barnes, D. *Programación orientada a objetos en JAVA*. Madrid: Pearson

Desing Patterns. Elements of Reusable. OO Software

JAVA Limpio. Pello Altadi. Eugenia Pérez

Apuntes de Programación de Anna Sanchis Perales

Apuntes de Programación de Lionel Tarazón Alcocer



Ilustraciones:

<https://pixabay.com/>

<https://freepik.es/>

<https://lottiefiles.com/>

Preguntas

