

# **UNIDAD 6.**

## **Estructuras de datos avanzadas y POO.**



# Índice

|  |    |
|--|----|
| 1. Ficha unidad didáctica.....                 | 1  |
| 2. Contenidos.....                             | 2  |
| 2.1. Introducción.....                         | 2  |
| 2.2. Estructuras de datos avanzadas y POO..... | 3  |
| 2.2.1 Estructuras dinámicas básicas.....       | 3  |
| 2.2.1.1 Pila (Stack).....                      | 3  |
| 2.2.1.2 Cola.....                              | 8  |
| 2.2.1.3 Lista.....                             | 14 |
| 2.2.1.4 Árboles.....                           | 18 |
| 2.2.1.4.1. Binarios de búsqueda.....           | 21 |
| 2.2.1.4.2. Otros árboles.....                  | 31 |
| 2.2.1.5 Tablas Hash.....                       | 33 |
| 2.2.1.6 Grafos.....                            | 34 |
| 2.2.2 . Colecciones en Java.....               | 36 |
| 2.2.2.1 Jerarquía de clases, interfaces.....   | 36 |
| 2.2.2.2 Iteradores.....                        | 40 |
| 2.2.2.3 List.....                              | 42 |
| 2.2.2.4 Queue.....                             | 46 |
| 2.2.2.5 Set.....                               | 49 |
| 2.2.2.6 Map.....                               | 56 |
| 2.2.3 Otras estructuras de datos.....          | 59 |
| 2.2.3.1 Grafos en Java.....                    | 59 |
| 2.2.4 Generalización.....                      | 61 |
| 2.2.5 Interfaces funcionales.....              | 70 |
| 2.2.6 Streams.....                             | 76 |
| 3. Practicando.....                            | 81 |
| 3.1. Colecciones en Java.....                  | 81 |
| 3.1.1 Listas.....                              | 81 |
| 3.1.2 Queue.....                               | 83 |
| 3.1.3 Sets.....                                | 84 |
| 3.1.4 Maps.....                                | 85 |

|   |     |
|---|-----|
| 3.2. Interfaces funcionales.....                                    | 86  |
| 3.3. Streams.....   | 87  |
| 4. Ejercicios teóricos.....   | 94  |
| 4.1. Parte 1. Fundamentos.....                                      | 94  |
| 4.2. Parte 2. Colecciones en Java.....                              | 97  |
| 4.3. Parte 3. Generalización, interfaces funcionales y streams..... | 99  |
| 4.3.1 Generalización.....   | 99  |
| 4.3.2 Interfaces funcionales.....                                   | 100 |
| 4.3.3 Streams.....  | 102 |
| 5. Soluciones practicando.....                                      | 103 |
| 5.1. Colecciones Java.....  | 103 |
| 5.1.1 Listas.....   | 103 |
| 5.1.2 Queue.....  | 105 |
| 5.1.3 Set.....  | 106 |
| 5.1.4 Maps.....   | 106 |
| 5.2. Interfaces funcionales.....                                    | 112 |

# 1. Ficha unidad didáctica.

| OBJETIVOS DIDÁCTICOS  |                     |
|---|---------------------|
| <p>OD1: Justificar la necesidad de estructuras de datos dinámicas.</p> <p>OD2: Razonar la estructura de datos adecuada al problema a resolver.</p> <p>OD3: Analizar código fuente que haga uso de estructuras de datos avanzadas.</p> <p>OD4: Caracterizar las diferentes tipos de colecciones.</p> <p>OD5: Desarrollar clases y métodos que utilicen generalización de forma razonada.</p> <p>OD6: Enunciar las estructuras de datos dinámicos en los lenguajes OO más utilizados.</p> <p>EV2. Concienciar de la importancia de emplear hábitos respetuosos con el medio ambiente y control del gasto energético de las instalaciones informáticas.</p> <p>EV3. Caracterizar la importancia del trabajo en grupo en el ámbito empresarial.</p> <p>RL1. Utilizar de forma adecuada y ergonómica el mobiliario de oficina, evitando posturas incorrectas que conlleven lesiones.</p> <p>TIC1. Emplear Internet para la realización de la actividad laboral.</p> <p>TIC2. Usar Internet como forma de actualización.</p> <p>ID1. Interpretar documentación técnica en inglés.</p> |                     |
| <b>RESULTADOS DE APRENDIZAJE</b>  | RA6                 |
| CONTENIDOS  |                     |
| <p>Estructuras dinámicas básicas.</p> <p>Colecciones.</p> <p>Otras estructuras de datos.</p> <p>Estructuras dinámicas en otros lenguajes OO.</p>  |                     |
| ORIENTACIONES METODOLÓGICAS   |                     |
| <p>La planificación de las actividades se realiza teniendo en cuenta la problemática de comprensión del concepto de datos dinámico y la abstracción de los TAD.</p> <p>Fomentar mediante el desarrollo de actividades la capacidad de “aprender a aprender”.</p> <p>Se relacionan el proceso de enseñanza con su aplicación en el mundo laboral.</p>  |                     |
| <b>CRITERIO DE EVALUACIÓN</b>   | 6b, 6c, 6d, 6e, 6f. |

## 2. Contenidos.

### 2.1. Introducción.

Hasta ahora la única forma de gestionar grupos de objetos o datos eran los vectores. Si bien son sencillos de usar, poseen una serie de inconvenientes que no los hacen prácticos en la mayoría de las aplicaciones y/o librerías, siendo estos:

- NO SE ADAPTA AL TAMAÑO DEL PROBLEMA. Se ha de conocer el tamaño al crear el vector, implica que o se pierde algo de espacio o es posible que se quede pequeño..
- BÚSQUEDAS COMPLEJAS. La búsqueda de un elemento en un vector desordenado es costosa computacionalmente.
- POSIBLES PROBLEMAS DE MEMORIA. En caso de gran cantidad de objetos a gestionar se ha de reservar memoria contigua, y en ocasiones esto no es sencillo.
- MANEJO COMPLEJO. La inserción, borrado y obtención de elementos es compleja.

Por ejemplo, se tiene una aplicación para gestionar un restaurante de comida rápida, los pedidos pueden llegar desde el terminal del dependiente, desde terminales táctiles en el restaurante, desde la web o desde la aplicación para teléfono. Estos pedidos se han de gestionar con alguna estructura de datos, es decir manejar pedidos. Si se elige un array se plantean los problemas comentados anteriormente:

- ¿Para cuantos pedidos como máximo?
- ¿A las 4 de la tarde se tiene los mismos pedidos que a las 9 de la noche?
- ¿Es sencillo buscar un pedido, por ejemplo por hora de pedido, por importe u otras combinaciones?
- Al servir un pedido. ¿Qué sucede con el espacio que ocupa en el array? ¿Si se soluciona, esta solución tiene algún coste?

Para solucionar estos problemas se han definido las estructuras de datos dinámicas, también conocidas como tipos de datos abstractos o TAD.



**La principal característica de las estructuras dinámicas es que pueden crecer y decrecer en función del tamaño del problema en cada momento.**

Otra característica no menos importante es que no es necesario que la memoria de estas estructuras se encuentre contigua, siendo posible tener fragmentos de la misma en diferentes posiciones de la memoria.

El resto de características como **velocidad de búsqueda, inserción, existencia de duplicados, borrado o espacio en memoria** dependerá de forma concreta en cada una de las estructuras y su elección se determinará por el tipo de problema a resolver.

Se define un tipo abstracto de dato como:

- Un conjunto de valores.
- Un conjunto de operaciones sobre los valores.

## **2.2. Estructuras de datos avanzadas y POO.**

### **2.2.1 Estructuras dinámicas básicas.**

Existen una serie de estructuras ampliamente utilizadas en informática, no solo en programación, por ejemplo en la gestión de procesos en los sistemas operativos, la gestión de los sistemas de ficheros o el tratamiento de los paquetes en los routers.

La mayor parte de los lenguajes incluyen estas estructuras, ya sea de forma nativa o con librerías externas, incluso en ocasiones se pueden tener estructuras similares de forma nativa y en librerías externas.

Destacar los elementos que se almacena en las estructuras, estos elementos han de almacenar información para permitir las operaciones, normalmente referencias a otros elementos(objetos), para poder moverse y realizar las operaciones.



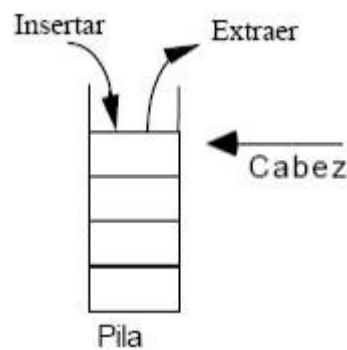
Introducción a los TAD. <https://www.youtube.com/watch?v=5k2DWMRTXMM>

#### **2.2.1.1 Pila (Stack).**

Este TAD representa un conjunto con valores(objetos de la misma clase o que hereden de

una común en Java) ordenados en función del momento en que se insertan, siendo el primer elemento de la pila el último en insertarse.

Dado su funcionamiento también se conoce como estructura LIFO (Last Input, First Output).



Las operaciones que posee una cola son:

- Create() ; //se crea una pila vacia
- Push(Element e); //apila un elemento en la cima de la pila.
- Element Pop(); //devuelve el elemento que se encuentre en la cima de la pila
- Boolean isEmpty(); //indica si la pila se encuentra vacia.

Algunos de los usos clásicos para las pilas son: Gestión de acción deshacer de alguno de los programas como procesadores de texto, gestión de páginas visitadas.

El elemento ha de contener a su vez dos componentes, el primero es el realmente a almacenar y el otro una **referencia** al anterior, de forma que al desapilar el anterior se convierta en la cima.



UCAM. Pilas. <https://www.youtube.com/watch?v=JRPKWsbjmmI>

Por ejemplo, se desea gestionar las páginas visitadas de forma que se pueda navegar hacia atrás (pero no hacia adelante).

En primer lugar se define el elemento a almacenar en la pila, una clase llamada StackNode que contiene como atributos una cadena que representa la url y una referencia (atributo) de tipo StackNode que sirve para apuntar o referenciar al siguiente. **En caso de ser el último elemento este atributo, llamado por ejemplo next será igual a null para**

indicar que es el último.

```
public class StackNode {  
    private String url;  
    private StackNode next;  
    public StackNode(String url){  
        this.next=null;  
        this.url=url;  
    }  
    public StackNode(String url, StackNode next){  
        this.next=next;  
        this.url=url;  
    }  
  
    /**  
     * @return the url  
     */  
    public String getUrl() {  
        return url;  
    }  
  
    /**  
     * @return the next  
     */  
    public StackNode getNext() {  
        return next;  
    }  
  
    /**  
     * @param url the url to set  
     */  
    public void setUrl(String url) {  
        this.url = url;  
    }  
  
    /**  
     * @param next the next to set  
     */  
    public void setNext(StackNode next) {  
        this.next = next;  
    }  
}
```



```
}
```

Ahora se define la clase Stack con las operaciones básicas vistas anteriormente.

```
public class Stack {  
    private StackNode top;  
  
    /**  
     * Constructor  
     */  
    public Stack() {  
        this.top = null;  
    }  
    /**  
     * Anyade un elemento a la pila  
     * @param node  
     */  
    public void push(StackNode node) {  
        node.setNext(this.top);  
        this.top = node;  
    }  
    /**  
     * desapila devolviendo la cima  
     * @return  
     */  
    public StackNode pop() {  
        StackNode tempo = null;  
        //si no es el final  
        if (this.top != null) {  
            tempo = top;  
            //pasa al siguiente, si el siguiente es nulo no pasa nada  
            this.top = this.top.getNext();  
            tempo.setNext(null);  
        }  
        return tempo;  
    }  
    /**  
     * devuelve si la pila está vacia  
     * @return  
     */  
    public boolean isEmpty() {  
        return this.top == null;  
    }  
}
```

```
}  
  
}
```

Un ejemplo de ejecución con el método toString sobrescrito en el Stack y en StackNode:

```
public static void main(String[] args) {  
    Stack pila= new Stack();  
    System.out.println("Al construir la pila contiene:");  
    System.out.println(pila);  
    pila.push(new StackNode("localhost"));  
    pila.push(new StackNode("www.google.com"));  
    pila.push(new StackNode("www.facebook.es"));  
    pila.push(new StackNode("www.ieslaencanta.com"));  
    System.out.println("Al insertar unos cuentos, el estado es:");  
    System.out.println(pila);  
    StackNode tempo=pila.pop();  
    System.out.println("Al desapilar se obtiene:");  
    System.out.println(tempo.toString());  
    System.out.println("Y el estado de la pila es:");  
    System.out.println(pila);  
}
```

El resultado:

```
Al construir la pila contiene:  
empty  
Al insertar unos cuentos, el estado es:  
www.ieslaencanta.com  
www.facebook.es  
www.google.com  
localhost  
  
Al desapilar se obtiene:  
www.ieslaencanta.com  
Y el estado de la pila es:  
www.facebook.es  
www.google.com  
localhost
```



Ejercicio propuesto. Implementar usando recursión el método length() que devuelve

la longitud de la pila, de igual forma implementar el método `toString()` que devuelve una cadena con la pila.



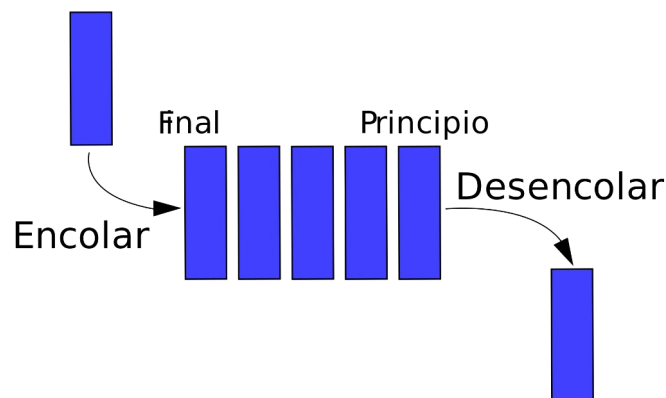
En caso de querer gestionar una pila de objetos de una clase que representa una coordenada ¿Qué cambios introducir?



Modificar el código para que se pueda añadir una cadena sin necesidad de crear o pasar como parámetro un `StackNode`, de igual forma para desapilar que devuelva solo la cadena.

### 2.2.1.2 Cola.

Las colas al igual que las pilas modelan situaciones en el que el orden de llegada determina el tratamiento de los datos, en este caso el primero en entrar es el primero en salir, por lo que también se conocen FIFO (First Input, First Output).



Las operaciones básicas son:

- `Create();`
- `Enqueue(Element e);` //encola el elemento
- `Element Dequeue();` //desencola el elemento y lo devuelve
- `boolean isEmpty();` //si la cola esta vacia
- `Element peek();` //devuelve el primer elemento de la cola, sin desencolar.

Las colas se utilizan ampliamente, desde los routers a la hora de gestionar los paquetes que llega, las solicitudes de uso de recursos o las peticiones de entradas en los portales

de espectáculos entre otros.

En el siguiente ejemplo se gestionan peticiones para procesar listas de la compra de personas, para simplificarlo se tiene simplemente la clase Pedido, con el nombre del cliente, la fecha y el importe.

**En este caso se opta por ocultar la estructura Node, ver que es muy similar, por no decir idéntica a la de la cola.**

Clase Order:

```
public class Order {
    private String client;
    private Date date;
    private float amount;
    public Order(String client, Date date, float amount) {
        this.client = client;
        this.date = date;
        this.amount = amount;
    }
    /**
     * @return the client
     */
    public String getClient() {
        return client;
    }

    /**
     * @return the date
     */
    public Date getDate() {
        return date;
    }

    /**
     * @return the amount
     */
    public float getAmount() {
        return amount;
    }

    /**
     * @param client the client to set
     */
}
```

```
*/  
public void setClient(String client) {  
    this.client = client;  
}  
/**  
 * @param date the date to set  
 */  
public void setDate(Date date) {  
    this.date = date;  
}  
/**  
 * @param amount the amount to set  
 */  
public void setAmount(float amount) {  
    this.amount = amount;  
}  
public String toString(){  
    SimpleDateFormat dateFormat = new SimpleDateFormat("dd/MM/yyyy");  
    return "Client: "+this.client+" Date:"+dateFormat.format(this.date)+ "  
Amount:"+this.amount;  
}  
}
```

Clase Node:

```
public class Node {  
    private Order order;  
    //referencia al siguiente  
    private Node next;  
  
    public Node() {  
        this.order = null;  
        this.next = null;  
    }  
    public Node(Order order) {  
        this.order = order;  
        this.next = null;  
    }  
    public Node(Order order, Node next) {  
        this.order = order;  
        this.next = next;  
    }  
}
```

```
}  
  
public Node getNext() {  
    return next;  
}  
  
public void setNext(Node next) {  
    this.next = next;  
}  
  
public Order getOrder() {  
    return order;  
}  
  
public void setOrder(Order order) {  
    this.order = order;  
}  
  
}
```

Clase Queue:

```
public class Queue {  
    //inicio de la cola, por donde sale  
    private Node top;  
    //final de la cola  
    private Node end;  
  
    public Queue() {  
        this.top = null;  
        this.end = null;  
    }  
  
    public void enqueue(Order order) {  
        Node n = new Node(order);  
        //la cola esta vacia  
        if (this.top == this.end && this.top == null) {  
            this.top = n;  
            this.end = n;  
        } else {  
            // se apunta al anterior  
            this.end.setNext(n);  
            this.end = n;  
        }  
    }  
}
```

```
}

public Order dequeue() {
    Order o = null;

    //si queda un elemento
    if (this.top != null) {
        o = this.top.getOrder();
        this.top = this.top.getNext();
    }
    return o;
}

public boolean isEmpty() {
    return this.top == null && this.end == null;
}

public Order peek() {
    Order o = null;
    if (this.top != null) {
        o = this.top.getOrder();
    }
    return o;
}
}
```

Programa principal, **se han implementado los métodos toString() necesarios para que funcione:**

```
public class QueuePrincipal {

    public static void main(String[] args) {
        Queue cola = new Queue();
        System.out.println("Al construir contiene:");
        System.out.println(cola);
        cola.enqueue(new Order("Paco", new Date(), 45));
        cola.enqueue(new Order("Paca", new Date(), 9));
        cola.enqueue(new Order("Pepe", new Date(), 234));

        System.out.println("Al insertar unos cuentos, el estado es:");
        System.out.println(cola);
    }
}
```

```
Order tempo = cola.dequeue();  
System.out.println("Al desencolar se obtiene:");  
System.out.println(tempo.toString());  
System.out.println("Y el estado de la cola es:");  
System.out.println(colas);  
}  
}
```

Salida:

```
Al construir contiene:  
empty  
Al insertar unos cuentos, el estado es:  
Client: Paco Date:22/02/2022 Amount:45.0  
Client: Paca Date:22/02/2022 Amount:9.0  
Client: Pepe Date:22/02/2022 Amount:234.0  
Al desencolar se obtiene:  
Client: Paco Date:22/02/2022 Amount:45.0  
Y el estado de la cola es:  
Client: Paca Date:22/02/2022 Amount:9.0  
Client: Pepe Date:22/02/2022 Amount:234.0
```

Las colas se utilizan mucho cuando se tiene un producto y un consumidor, por ejemplo las colas de impresión en que se van dejando los trabajos por parte del productor(diferentes programas como procesadores de texto, retoque fotográfico...) y un demonio va consultando la cola y procesando por orden de llegada, el consumidor. **Esto puede ser problemático ya que en un instante de tiempo dos o más programas o en caso de programación dos o más hilos (Thread) intenta acceder a la cola (puede suceder igual en la pila) modificando ambas al mismo tiempo los nodos, pudiendo ocasionar fallos.**



Investigar como se puede mitigar el acceso concurrente a las colas,pilas... usando synchronized, aplicarlo al código de la cola.



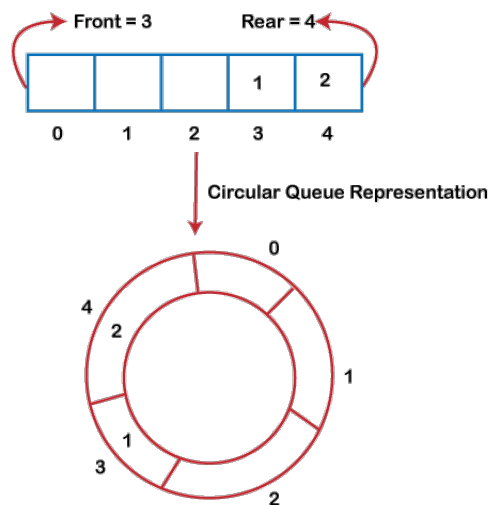
Ejercicio propuesto. Implementar usando recursión el método length() que devuelve la longitud , de igual forma implementar el método toString() que devuelve una cadena con la cola.





Añadir un método boolean `Exists(Order o)`, que indique si una orden existe en la cola.

La tratada es la cola básica, existen otro tipo de colas como las colas dobles cuyos nodos apuntan al siguiente y el anterior o colas circulares con un tamaño definido, la elección de una u otra depende del problema a trata.

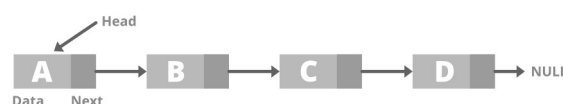


### 2.2.1.3 Lista.

Caso más general de pilas y colas, ya que una pila o cola son un tipo de lista. Una lista se puede ver como un vector pero de tamaño variable y no siendo necesario que la memoria sea continua. Las características a grandes rasgos son:

- Acceso a cualquier posición que exista en la lista.
- Insertar y borrar elementos en cualquier posición.
- Es posible concatenar (unir) y dividir listas a través de sublistas.

#### Singly Linked List



Las operaciones básicas con una lista son:

- `Insert(Element e, int i);` //inserta un elemento en la posición i-ésima
- `Append(Element e);` //añadie el elemento e al final de la lista.

- `getByIndex(int i);` //devuelve el elemento que se encuentra en la posición i-ésima.
- `Delete(int i);` //elimina el elemento de la posición i-esima
- `int Lenght();` //devuelve la longitud de la lista
- `void Join(List l);` //une dos listas
- `List l subList(int from, int to);` //extrae una sublista indicando los índices
- `boolean Exists(Element e);` //indica si un elemento se encuentra en la lista.

La clase más genérica es la de insertar los elementos según va siendo necesario en el inicio de la lista, de forma similar a las colas y pilas, siendo la diferencia más significativa con estas la posibilidad de obtener un elemento a partir del índice, insertar en un lugar concreto o borrar un elemento a partir de su índice.

Existen otros tipos de listas como las circulares, o las doblemente enlazadas que facilitan ciertas operaciones (en especial la inserción y el borrado) aunque las más conocidas son las listas ordenadas que como su nombre indica se encuentran ordenadas.



UCAM. Tipos de listas. <https://www.youtube.com/watch?v=ffriJilROZM>

Siguiendo con el ejemplo anterior, ahora se desea gestionar los pedidos por precio con una lista, la clase `Order` y `Node` son iguales que las anteriores.

Código de lista ordenada:

```
public class List {  
    private Node start;  
    public List() {  
        this.start = null;  
    }  
    public void insert(Order o) {  
        //se necesita dos temporales, el actual y el anterior al actual  
        Node now = this.start;  
        Node last = now;  
        Node new_node = new Node(o);  
        new_node.setNext(null);  
        //la lista esta vacia  
        if (this.start == null) {  
            this.start = new_node;  
        } else {
```

```
//se tiene que buscar la posición a insertar
while (now != null && now.getOrder().compareTo(o) < 0) {
    last = now;
    now = now.getNext();
}
//se ha llegado al nulo, se inserta en el anterior
if (now == null) {
    last.setNext(new_node);
} else {
    //si es el primero en el que se tiene que insertar
    if (now == start) {
        this.start = new_node;
        new_node.setNext(last);
    } else {
        //el anterior apunta al nuevo
        last.setNext(new_node);
        //el nuevo apunta al actual
        new_node.setNext(now);
    }
}
}

}

public Order getByIndex(int index) {
    Order o = null;
    Node tempo = this.start;
    //mientras no llegue al index o no queden nodos
    for (int i = 0; i <= index && tempo != null; i++) {
        if (i == index) {
            o = tempo.getOrder();
        }
        tempo = tempo.getNext();
    }
    return o;
}

public int length() {
    int c = 0;
    Node tempo = this.start;
    for (c = 0; tempo != null; c++) {
        tempo = tempo.getNext();
    }
}
```

```
    }  
    return c;  
}  
public Order remove(int index) {  
    Order o = null;  
    //se necesita dos temporales, el actual y el anterior al actual  
    Node now = this.start;  
    Node last = now;  
    if (this.start != null) {  
        //si es el primero  
        if (index == 0) {  
            o=this.start.getOrder();  
            this.start = this.start.getNext();  
        } else {  
            //la lista no esta vacia  
            for (int i = 0; i < index && now != null; i++) {  
                last = now;  
                now = now.getNext();  
            }  
            //si no es nulo se obtiene, el anteriorr ahora apunta al  
siguiente  
            //del actual (saltandolo)  
            if (now != null) {  
                last.setNext(now.getNext());  
                o = now.getOrder();  
                now = null;  
            }  
        }  
    }  
    return o;  
}  
@Override  
public String toString() {  
    if (this.start == null) {  
        return "empty";  
    } else {  
        return this.start.toString();  
    }  
}
```



Se ha utilizado el método `compareTo` de la clase `Order`, ¿Cómo sería la implementación de ese método? ¿Existe alguna otra clase que la utilice, por ejemplo ordenar elementos? ¿Qué peculiaridad tiene?



¿Cuál es el coste temporal de la búsqueda? ¿Es factible usar listas para almacenar registros en una base de datos, por ejemplo una tabla con un millón de registros, o para gestionar el sistema de ficheros de un sistema operativo?. Debatir en clase posibles soluciones para hacer la búsqueda más rápida.



Crear los métodos `getByName` y `removeByName`, en el que se utiliza el nombre para obtener un pedido y para borrar un pedido.



Video de la UPV sobre listas, pilas y colas. [https://www.youtube.com/watch?v=-](https://www.youtube.com/watch?v=-Shr2s0gYao)

[Shr2s0gYao](https://www.youtube.com/watch?v=-Shr2s0gYao)

#### 2.2.1.4 Árboles.

Las estructuras anteriores mejoran los clásicos vectores, pero existe una operación que no es viable manejando gran cantidad de datos, **la búsqueda**. Las búsquedas en las estructuras anteriores tienen un coste lineal, si se tiene una lista de 10 elementos en el peor de los casos se ha de buscar en los 10 elementos, en caso de un millón de elementos es necesario analizar el millón en el peor caso.

Para mejorar el coste de la búsqueda, se definen los árboles. La definición de un árbol es la siguiente:

- Los árboles se componen de **nodos**.
- Un sólo nodo es por sí solo un árbol.
- Al primer nodo de un árbol se le denomina raíz.
- Un nodo puede tener uno o más descendientes (las listas son árboles con nodos con un único descendiente), a los nodos descendientes se les denomina **hijos** y al antecesor de un nodo se le denomina **padre**.

- Los nodos terminales reciben el nombre de **hoja**.

Algunos conceptos relacionados con los árboles son:

- Los nodos con el mismo padre se denomina hermanos.
- Los nodos se unen mediante **aristas**.
- Se define un camino en el árbol como la secuencias de aristas para llegar de un nodo A a un nodo B.
- La **profundidad** de un nodo es la longitud del camino de la raíz a ese nodo.
- La **altura** de un árbol es el camino más largo desde la raíz a su hoja más profunda.
- Los árboles cuyos nodos pueden tener **n** hijos se denominan árboles n-arios. Una lista es un árbol unario. Los árboles binarios son aquellos cuyos



**Los árboles pueden estar ordenados o desordenados, en caso de ser ordenados suele hacerse de izquierda a derecha, es decir el subárbol de la izquierda es menor que el subárbol de la derecha.**



UCAM. Árboles. <https://www.youtube.com/watch?v=kzQ49lgWd68>

En el caso de árboles ordenados (**los más interesantes**) existen 3 formas de recorrerlos (**se utiliza recursividad**)

- Preorden: raíz, árbol de la izquierda, árbol de la derecha.
- Inorden: árbol de la izquierda, raíz, árbol de la derecha.
- Postorden: árbol de la izquierda, árbol de la derecha, raíz.

## Estructura de Datos

El resultado para los 3 métodos es el siguiente:

**Preorden**

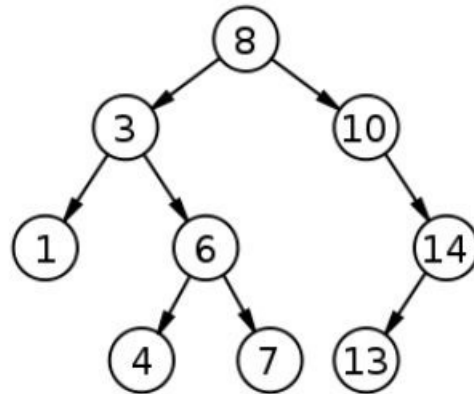
8 3 1 6 4 7 10 14 13

**Enorden**

1 3 4 6 7 8 10 13 14

**Postorden**

1 4 7 6 3 13 14 10 8



Las principales operaciones con los árboles son:

- Node Parent(Node n); //Devuelve el padre del nodo.
- Node Root(); //Devuelve la raíz del árbol.
- Node Search(parameters); //busca a partir de los parámetros un nodo devolviéndolo si lo encuentra.
- Insert(Element e); //inserta el elemento en el árbol
- Delete(Element e); //borra un elemento del nodo
- Preorden(); //Recorre el árbol en preorden
- Inorden();
- Posrorden();

Los árboles se pueden implementar con vectores o nodos con referencias a los hijos. La segunda opción es la más eficiente en memoria aunque algo más compleja.

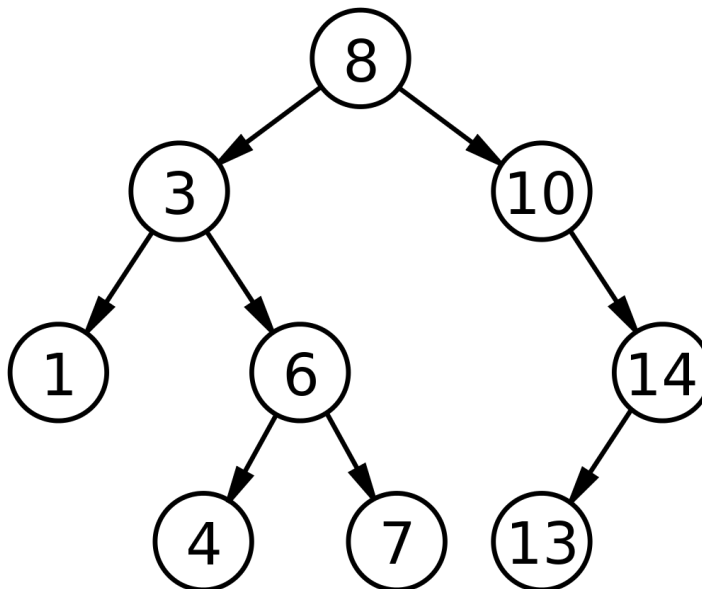
Existen multitud de árboles, desordenados y ordenados, entre los ordenados destacan:

- Árboles binarios.
- Árboles AVL.
- Árboles rojo-negro.
- Árboles B,B+,B\*

Mysql define diferentes motores de almacenamiento como MyISAM o memory. A partir del enlace <https://www.w3resource.com/mysql/mysql-storage-engines.php> indicar qué tipo de árbol utiliza MyISAM, memory e InnoDB para almacenar los datos de las tablas. ¿Cuál es la razón de usar árboles y no listas?

#### 2.2.1.4.1. Binarios de búsqueda.

Uno de los árboles más sencillos son los árboles binarios, en caso de estar ordenados se denominan árboles binarios de **búsqueda**. Estos árboles se encuentran ordenados de izquierda a derecha, y por lo tanto a la hora de buscar este orden marca el camino para encontrarlo.



UPV.Árboles binarios . <https://www.youtube.com/watch?v=mTMrszfrNtI>

En el siguiente ejemplo se define un árbol binario de búsqueda.

Nodo:

```
public class Node {  
  
    private Order order;  
    //referencia al siguiente  
    private Node left;  
    private Node right;
```



```
public Node() {
    this.order = null;
    this.left = null;
    this.right=null;
}

public Node(Order order) {
    this.order = order;
    this.left = null;
    this.right=null;
}

public Order getOrder() {
    return order;
}

public void setOrder(Order order) {
    this.order = order;
}

public Node getLeft() {
    return left;
}

/**
 * @return the right
 */
public Node getRight() {
    return right;
}

/**
 * @param left the left to set
 */
public void setLeft(Node left) {
    this.left = left;
}

/**
 * @param right the right to set
```

```
*/  
  
public void setRight(Node right) {  
    this.right = right;  
}  
  
}
```

Árbol binario de búsqueda:

```
public class SortBinaryTree {  
  
    private Node root;  
  
    public SortBinaryTree() {  
        this.root = null;  
    }  
  
    public SortBinaryTree(Order o) {  
        this.root = new Node();  
        this.root.setOrder(o);  
    }  
  
    public Order search(int amount) {  
        Order vuelta = null;  
        Node actual = null;  
        if (this.root != null) {  
            actual = this.root;  
            while (actual != null && actual.getOrder().getAmount() != amount) {  
                if (actual.getOrder().getAmount() > amount) {  
                    actual = actual.getLeft();  
                } else {  
                    if (actual.getOrder().getAmount() < amount) {  
                        actual = actual.getRight();  
                    }  
                }  
            }  
        }  
        return vuelta;  
    }  
  
    public Node remove(int amount) {  
        Node vuelta = null;
```

```
Node padre = null, actual = null, menor_derecha = null, padre_derecha =
null, mayor_izquierda = null, padre_izquierda = null;

if (this.root != null) {
    actual = this.root;
    while (actual != null && actual.getOrder().getAmount() != amount) {

        if (actual.getOrder().getAmount() > amount) {
            padre = actual;
            actual = actual.getLeft();
        } else {
            if (actual.getOrder().getAmount() < amount) {
                padre = actual;
                actual = actual.getRight();
            }
        }
    }
}

//borrar nodo con 3 situaciones:
//nodo hoja
//hijos a izquierda o derecha
//hijos a izquierda y derecha
if (actual != null) {
    //es una hoja
    if (actual.getLeft() == null && actual.getRight() == null) {
        if (padre.getLeft() == actual) {
            padre.setLeft(null);
            vuelta = actual;
        }
        if (padre.getRight() == actual) {
            padre.setRight(null);
            vuelta = actual;
        }
    }
    //si solo tiene el hijo de la derecha
    if ((actual.getLeft() == null && actual.getRight() != null)) {
        if (padre.getLeft() == actual) {
            padre.setLeft(actual.getRight());
            vuelta = actual;
        }
        if (padre.getRight() == actual) {
            padre.setRight(actual.getRight());
        }
    }
}
```

```
        vuelta = actual;
    }

}

//si solo tiene el hijo de la izquierda
if ((actual.getLeft() != null && actual.getRight() == null)) {
    if (padre.getLeft() == actual) {
        padre.setLeft(actual.getLeft());
        vuelta = actual;
    }
    if (padre.getRight() == actual) {
        padre.setRight(actual.getLeft());
        vuelta = (actual);
    }
}

//si tiene hijos en los dos , se sustituye por el menor de la
derecha
if ((actual.getLeft() != null && actual.getRight() != null)) {

    menor_derecha = actual.getRight();
    padre_derecha = actual;
    while (menor_derecha.getLeft() != null) {
        padre_derecha = menor_derecha;
        menor_derecha = padre_derecha.getLeft();
    }

    //en caso de que el menor a la derecha tenga hijos a su
derecha, se cuelgan del padre a la izquierda
    if (padre_derecha != actual) {
        padre_derecha.setLeft(menor_derecha.getRight());
    }

    //se actualiza el valor del nodo a borrar
    actual.setOrder(menor_derecha.getOrder());

    if (actual.getRight() != null &&
actual.getOrder().compareTo(actual.getRight().getOrder()) == 0) {
        actual.setRight(null);
    }

    vuelta = menor_derecha;
}

}
```

```
        return vuelta;
    }

    public String preorden() {
        if (this.root != null) {
            return this.preorden(this.root);
        } else {
            return "empty";
        }
    }

    private String preorden(Node n) {
        StringBuilder vuelta = new StringBuilder();
        if (n.getOrder() != null) {
            vuelta.append(n.getOrder());
        }
        if (n.getLeft() != null) {
            vuelta.append(this.preorden(n.getLeft()));
        }
        if (n.getRight() != null) {
            vuelta.append(this.preorden(n.getRight()));
        }
        return vuelta.toString();
    }

    public String inorden() {
        if (this.root != null) {
            return this.inorden(this.root);
        } else {
            return "empty";
        }
    }

    private String postorden(Node n) {
        StringBuilder vuelta = new StringBuilder();

        if (n.getLeft() != null) {
            vuelta.append(this.postorden(n.getLeft()));
        }
    }
}
```

```
if (n.getRight() != null) {
    vuelta.append(this.postorden(n.getRight()));
}
if (n.getOrder() != null) {
    vuelta.append(n.getOrder());
}
return vuelta.toString();
}

public String postorden() {
    if (this.root != null) {
        return this.preorden(this.root);
    } else {
        return "empty";
    }
}

private String inorden(Node n) {
    StringBuilder vuelta = new StringBuilder();

    if (n.getLeft() != null) {
        vuelta.append(this.inorden(n.getLeft()));
    }
    if (n.getOrder() != null) {
        vuelta.append(n.getOrder());
    }
    if (n.getRight() != null) {
        vuelta.append(this.inorden(n.getRight()));
    }
    return vuelta.toString();
}

public Node insert(Order o) {
    Node tempo = null, actual, last = null;
    //árbol vacío;
    if (this.root == null) {
        this.root = new Node(o);
    } else {
        actual = root;
        //se busca en el lugar que insertar
```

```
while (actual != null) {
    if (actual.getOrder().compareTo(o) < 0) {
        last = actual;
        actual = actual.getRight();
    } else {
        last = actual;
        actual = actual.getLeft();
    }
}

//ya se encuentra actual apuntando a un valor nulo, y anterior a un
nodo hoja


//ahora se inserta en el nodo hoja anterior en nuevo nodo
tempo = new Node(o);
if (last.getOrder().compareTo(o) < 0) {
    last.setRight(tempo);
} else {
    last.setLeft(tempo);
}

return tempo;
}

public String toGraphviz() {
    if (this.root != null) {
        return "digraph BST {\n" + this.toGraphviz(this.root) + "\n}";
    } else {
        return "empty";
    }
}

private String toGraphviz(Node n) {
    StringBuilder vuelta = new StringBuilder();
    if (n.getOrder() != null) {
        vuelta.append(n.getOrder().getAmount() + "\n");
        if (n.getLeft() != null) {
            vuelta.append(n.getOrder().getAmount() + "->" +
n.getLeft().getOrder().getAmount() + "\n");
            vuelta.append(this.toGraphviz(n.getLeft()));
        }
        if (n.getRight() != null) {
            vuelta.append(n.getOrder().getAmount() + "->" +
n.getRight().getOrder().getAmount() + "\n");
            vuelta.append(this.toGraphviz(n.getRight()));
        }
    }
}
```

```
    }  
    }  
    return vuelta.toString();  
}  
}
```

 UPV. Insercció en arbre binari de cerca. [https://www.youtube.com/watch?v=Yy\\_0EqTpFPs](https://www.youtube.com/watch?v=Yy_0EqTpFPs)

Un ejemplo con la clase anterior:

```
public class OrderBinaryTreePrincipal {  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        SortBinaryTree tree;  
        tree = new SortBinaryTree();  
        tree.insert(new Order("Paco", new Date(), 45));  
        tree.insert(new Order("Paca", new Date(), 9));  
        tree.insert(new Order("Pepe", new Date(), 234));  
        tree.insert(new Order("Pepa", new Date(), 2));  
        tree.insert(new Order("Pedro", new Date(), 231));  
        tree.insert(new Order("Petra", new Date(), 1));  
        System.out.println(tree.toGraphviz());  
    }  
}
```



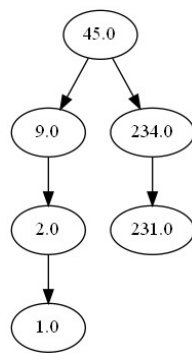
Investigar que es el software Graphviz, teniendo en cuenta que la salida del programa es:

```
digraph BST {  
45.0  
45.0->9.0  
9.0  
9.0->2.0  
2.0
```

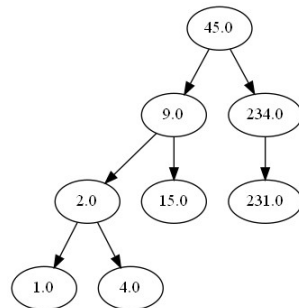


```
2.0->1.0
1.0
45.0->234.0
234.0
234.0->231.0
231.0
}
```

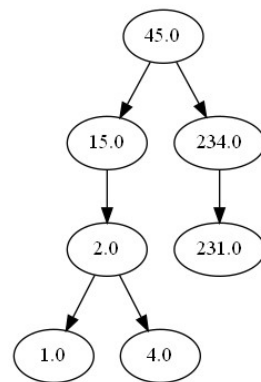
Y el árbol resultado:



Al insertar una nueva orden con amount 4 y otra con 15, el árbol resultado es:



Pór último, al borrar el 9, el nuevo árbol es:



Implementar un método que devuelva el número de elementos del árbol.



Dibujar el árbol binario de búsqueda resultado del siguiente código:

```
SortBinaryTree tree;
tree = new SortBinaryTree();
tree.insert(new Order("Paco", new Date(), 1));
tree.insert(new Order("Paca", new Date(), 2));
tree.insert(new Order("Pepe", new Date(), 3));
tree.insert(new Order("Pepa", new Date(), 4));
tree.insert(new Order("Pedro", new Date(), 5));
tree.insert(new Order("Petra", new Date(), 6));
tree.insert(new Order("Pilar", new Date(), 7));
```

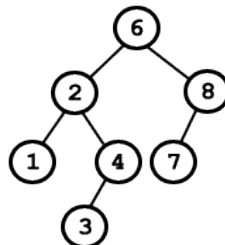
¿Existe algún problema con el árbol anterior? ¿Alguna idea para solucionarlo?

#### 2.2.1.4.2. Otros árboles.

Los árboles binarios de búsqueda no son los únicos árboles que se utilizan en programación y en la informática en general, del resto los más destacados son:

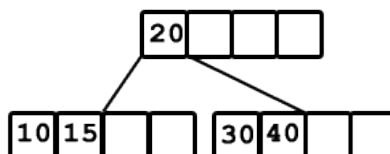
#### Árboles AVL.

Es similar a un árbol binario de búsqueda, pero con la característica de que cada rama se encuentra balanceada, es decir la diferencia entre las profundidades de cada rama de un nodo es como mucho 1. La ventaja principal es que la búsqueda de cualquier nodo es del orden  $\log_2(n)$ , siendo  $n$  el tamaño del árbol.



#### Árboles B.

Árboles  $n$ -arios, cada nodo puede tener como máximo  $2n$  hijos y como mínimo  $n$ , definiendo  $n$  al crear el árbol. Garantizando una ocupación de un 50%.



Se utilizan principalmente en los sistemas de ficheros (NTFS,EXT...) ya que es posible cargar en memoria secciones del árbol. La información se almacena en los nodos

intermedios y en las hojas.

Se necesitan las operaciones división para dividir un nodo en 2 y fusionar que uno de los nodos para garantizar los requisitos de operación



Vídeo árboles B. <https://www.youtube.com/watch?v=cLq3YdORfXs>



Vídeo árboles B II. <https://www.youtube.com/watch?v=7w-4h-XK7wc>

Árboles B+.

Similar al árbol B, pero en este caso la información se almacena únicamente en las hojas. Los intermedios solo almacena referencias y claves de búsqueda. Suelen tener referencias a la siguiente hoja aunque no formen parte de la misma rama.

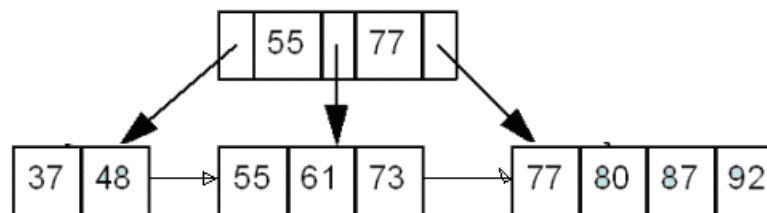
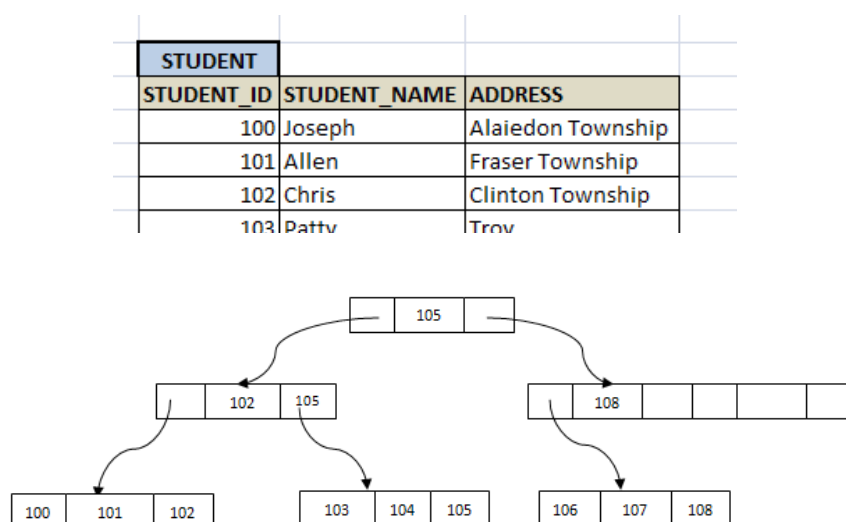


Fig. 6 Árbol B+



Uso de árboles B+ en base de datos. <https://www.youtube.com/watch?v=MNLXQgAqtwo>

Por ejemplo, al almacenar datos en una base de datos con una clave principal, realmente se crea un árbol con el índice.



Árboles B\*.

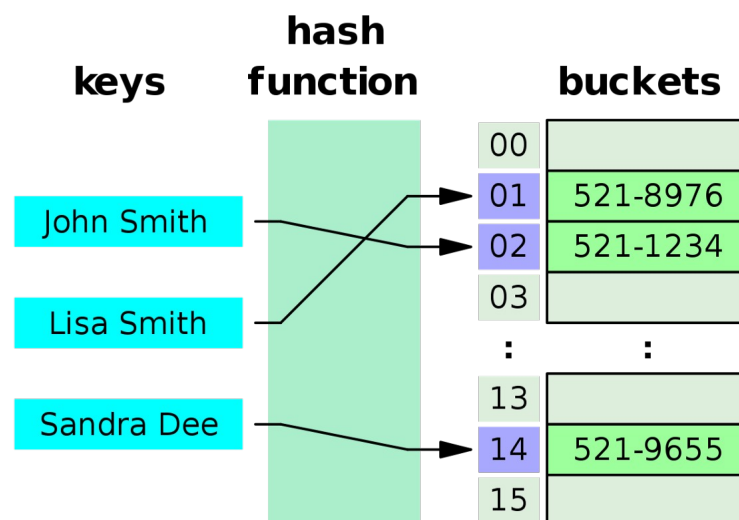
Modificación del árbol B, aumentando los requisitos para el mínimo de ocupación de un nodo.

### 2.2.1.5 Tablas Hash.

Las tablas hash o diccionarios son ampliamente utilizadas en la actualidad, no se va a entrar en detalles de implementación, pero combinan el uso de un vector de un tamaño dado con el cálculo de su índice a partir de una llave, de forma que el acceso a un elemento o su búsqueda se realiza por su clave.



**Las operaciones en las tablas Hash (en especial la búsqueda) es constante, aunque las insercciones pueden ser problemáticos por posibles colisiones.**



Por ejemplo en la imagen anterior se tiene una tabla hash de 15 elementos cuyas claves son cadenas y su par o valor es el teléfono **(puede ser cualquier objeto de una clase dada o incluso cualquier objeto si el par es de tipo Object)**, al insertar a John Smith se pasa la cadena a una función hash HASH("John Smith") que devuelve el entero 1, por lo que se inserta el teléfono en la posición 1 del vector.

Ahora se desea obtener el teléfono de Lisa Smith, para ello se realiza la función hash sobre el nombre devolviendo 2, accediendo al índice.



¿Qué sucede si el hash de "Juan Palomo" es el mismo que el de "John Smith"?

En esos casos se produce una colisión, y se han de establecer mecanismos para solucionar estas colisiones, la más común es que cada elemento del vector sea a su vez una lista.

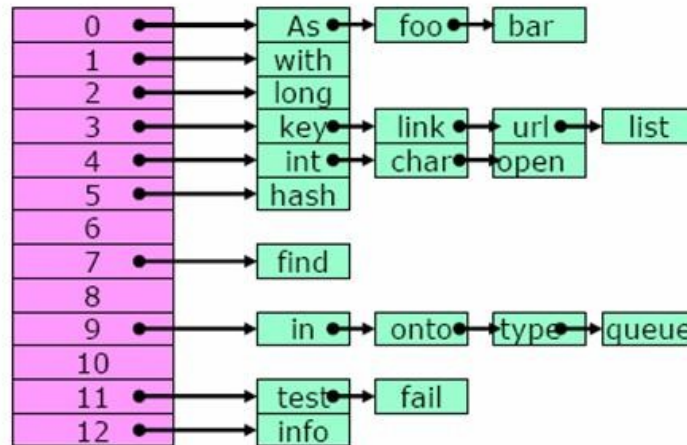


Tabla Hash UPV. <https://www.youtube.com/watch?v=WnLdu8OHA3Q>

### 2.2.1.6 Grafos.

El caso más general, las listas, pilas y colas son particularizaciones de los árboles y estos a su vez son particularizaciones de los grafos. En este punto simplemente se definen el concepto de grafo, queda fuera del curso profundizar en la implementación y algoritmos utilizados en los grafos.

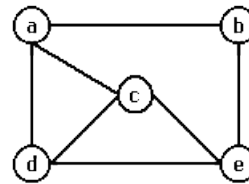
Un grafo es un conjunto de nodos unidos por un conjunto de arcos. Un ejemplo de grafo que podemos encontrar en la vida real es el de un plano de trenes. El plano de trenes está compuesto por varias estaciones (nodos) y los recorridos entre las estaciones (arcos) constituyen las líneas del trazado.

Una definición más formal de grafos. Un grafo  $G=(V,E)$  consiste en un conjunto  $V$  de nodos (vértices) y un conjunto  $E$  de aristas (arcos). Cada arista es un par  $(v,w)$ , siendo  $v$  y  $w$  un par de nodos pertenecientes al conjunto  $V$  de nodos. Podemos distinguir entre grafos dirigidos y no dirigidos. En un grafo dirigido los pares  $(v,w)$  están ordenados, traduciéndose la arista en una flecha que va desde el nodo  $v$  al nodo  $w$ .

En el caso de un grafo no dirigido, los nodos están unidos mediante líneas sin indicación de dirección.

$$V = \{ a, b, c, d, e \}$$

$$E = \{ (a, b), (a, c), (a, d), (b, e), (c, d), (c, e), (d, e) \}$$

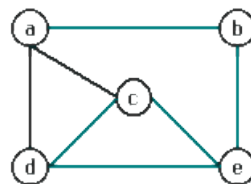


Por último se puede definir una función que asocie a cada arco un coste:  $\text{coste}(\text{arco})$

Terminología común en los grafos.

Se habla de dos vértices adyacentes cuando estén unidos por un arco. El número de vértices adyacentes de un nodo constituye el grado del mismo. En el ejemplo los vértices adyacentes al nodo 3 son el 1, 4 y 5, siendo éste por tanto un nodo de grado tres por tener tres vértices adyacentes.

Un camino entre dos vértices, es una secuencia de vértices tal que dos vértices consecutivos son adyacentes. En el siguiente ejemplo el camino entre el vértice a y el vértice e será la secuencia de vértices abecde.



Cuando este camino no tiene vértices repetidos se dice que es simple. Salvo en el caso de que el primer y último vértice del camino sean el mismo, en cuyo caso hablaremos de un ciclo

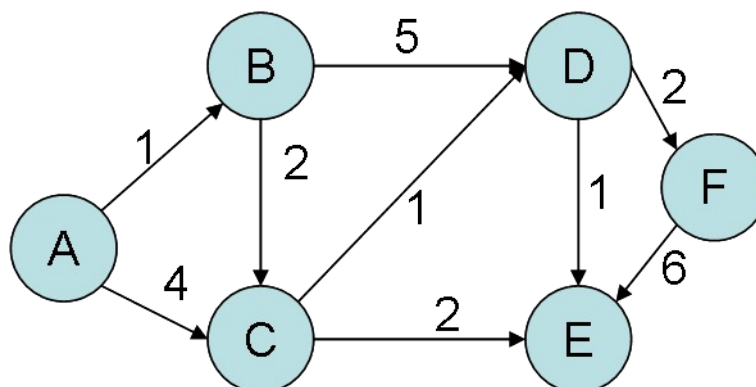
Existen diferentes representaciones de los grafos como matrices de adyacencia, listas de adyacencia o matrices dispersas, aunque lo común es utilizar librerías.



Fundamentos de los grafos UPV. <https://www.youtube.com/watch?v=pzca71UtH-A>



Uno de los algoritmos más conocidos que utiliza de Dijkstra, investigar que hace y exponer al menos 2 situaciones en las que se utilice.



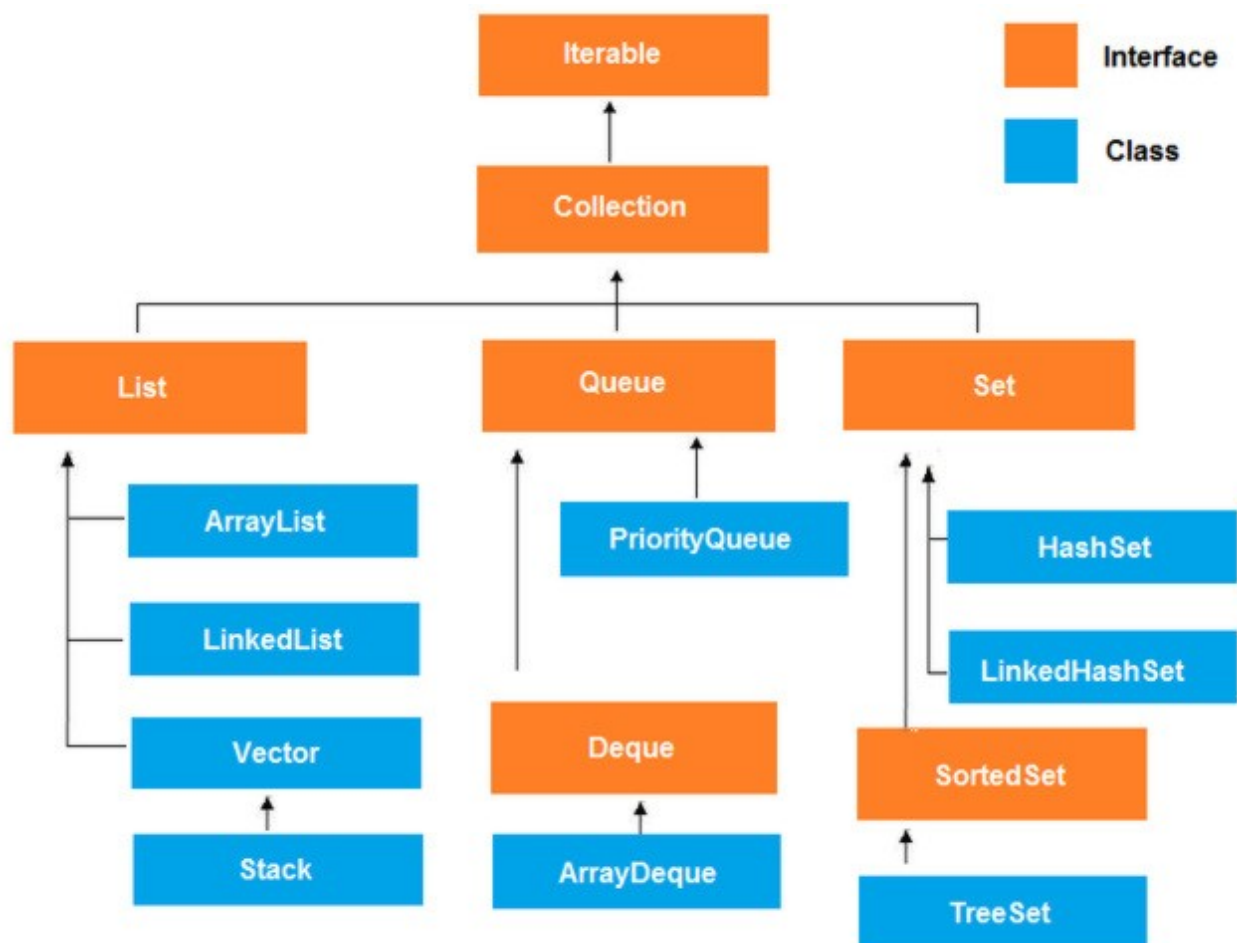
Algoritmo de Dijkstra UPV. <https://www.youtube.com/watch?v=eLFEIxDEphA>

## 2.2.2 . Colecciones en Java.

Los principales lenguajes de programación poseen librerías con código generico para (es posible su uso con cualquier clase o heredados de esta). Se basa en la definición de una serie de interfaces y de las implementación de estas en clases concretas, que implementan con algunas modificaciones las estructuras de datos anteriores (listas, árboles....). En java las colecciones se definen en `java.util.Collections`.

### 2.2.2.1 Jerarquía de clases, interfaces.

Se define una jerarquía de clases para la implementación de las diferentes estructuras de datos en Java con el uso de herencia e interfaces.



La imagen anterior es una versión reducida de la jerarquía de colecciones en Java. La jerarquía comienza con la interfaz **Iterable**, que permite recorrer el resto de las

estructuras.

A continuación se define la interfaz Collection, con los métodos como:

```
• boolean add(E e)
• void clear()
• boolean isEmpty()
• boolean remove(Object o)
• default Stream<E> parallelStream()
• int size()
• Object[] toArray()
```

En el siguiente nivel de la herencia se detallan las estructuras clásicas vistas en puntos anteriores:

- **List**. A los métodos anteriores añade algunos como void add(int index, E element), E get(int index), remove(int index), **operaciones típicas de una lista**, posee las siguientes implementaciones.
  - **ArrayList**. Implementa las listas usando arrays (vectores).
  - **LinkedList**. Lista enlaza, se implementa con referencias.
  - **Vector**. Similar al arraylist, pero en este caso el acceso al vector es sincronizado, en caso de arraylist varios hilos pueden acceder al mismo, en el vector solo un hilo puede acceder al vector.
    - **Stack**, pila implementada con vector. Con métodos como: push, pop o peek.
- **Queue**. En este caso una cola, con operaciones como add, element, peek o poll entre otras, clásicas en las colas.
  - **PriorityQueue**. Cola con prioridad, el orden se marca al comparar los elementos insertados. Por ejemplo:

```
PriorityQueue<String> queue=new PriorityQueue<String>();
queue.add("Amit");
queue.add("Vijay");
queue.add("Karan");
queue.add("Jai");
queue.add("Rahul");
Iterator itr=queue.iterator();
while(itr.hasNext()){
```



```
System.out.println(itr.next());  
}
```

Produce la salida:

```
Amit  
Jai  
Karan  
Vijay  
Rahul
```

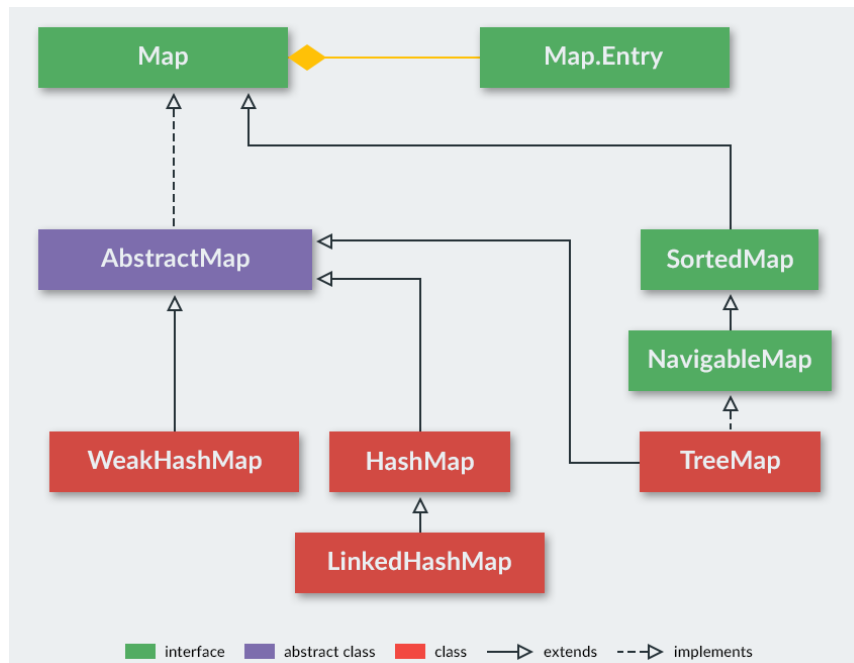


Si se tienen, por ejemplo personas, ¿cómo se indica la prioridad? ¿A qué tipo de lista sería equivalente esta cola de prioridad.

- **Deque:** Interfaz que define cola de dos extremos, es decir permite la inserción y borrado en la cabeza y la cola. Las implementaciones existentes son:
  - **ArrayDeque.** Se implementa usando un array.
  - **ConcurrentLinkedDeque.** Con referencias.
  - **LinkedBlockingDeque.** En este caso la doble cola tiene un límite máximo de crecimiento.
- **Set.** Este caso no se ha tratado, pero consiste en una colección de elementos desordenados que no puede tener duplicados, es decir **modela conjuntos matemáticos**. Las principales implementaciones son:
  - **HashSet.** En este caso el conjunto se implementa con una tabla hash.
  - **TreeSet.** En este caso la estructura interna se basa en un árbol de tipo rojo-negro.
  - **EnumSet.** En este caso se almacenan enumerados. La estructura interna es un vector de bits.

Los anteriores son las interfaces y colecciones más destacadas, existiendo más clases como `ConcurrentSkipListSet` o `LinkedTransferQueue` entre otros.

También se ha de tener en cuenta las tablas Hash conocidas por Map, con la siguiente jerarquía:



Se ha de tener en cuenta la naturaleza del problema a resolver para seleccionar la colección a utilizar, teniendo también en cuenta la complejidad temporal de las operaciones más comunes.

### Listas y conjuntos

| Estructura    | get      | add      | remove   | contains |
|---------------|----------|----------|----------|----------|
| ArrayList     | O(1)     | O(1)     | O(n)     | O(n)     |
| LinkedList    | O(n)     | O(1)     | O(1)     | O(n)     |
| HashSet       | O(1)     | O(1)     | O(1)     | O(1)     |
| LinkedHashSet | O(1)     | O(1)     | O(1)     | O(1)     |
| TreeSet       | O(log n) | O(log n) | O(log n) | O(log n) |

### Mapas:

| Estructura    | get      | put      | remove   | containsKey |
|---------------|----------|----------|----------|-------------|
| HashMap       | O(1)     | O(1)     | O(1)     | O(1)        |
| LinkedHashMap | O(1)     | O(1)     | O(1)     | O(1)        |
| TreeMap       | O(log n) | O(log n) | O(log n) | O(log n)    |



Quando se define y se instancian algunas de las colecciones anteriores es posible indicar la clase de los objetos que gestionara, esto determina a su vez los

tipos de datos que algunos de sus métodos devuelven, que serán del mismo tipo que el indicado en la definición e instanciación. Esto se conoce como **Generics**, utilizándose simplemente en la primera parte del tema y definiéndose en un apartado de la segunda parte.

Un ejemplo extraído del tutorial de Java de Oracle:

Uso de una lista enlazada en la que no se define el tipo de dato a almacenar:

```
List myIntList = new LinkedList(); // 1
myIntList.add(new Integer(0)); // 2
Integer x = (Integer) myIntList.iterator().next(); // 3
```

Y definiendo el tipo de dato que almacena:

```
List<Integer> myIntList = new LinkedList<Integer>(); // 1'
myIntList.add(new Integer(0)); // 2'
Integer x = myIntList.iterator().next(); // 3'
```



¿Qué diferencia se observa en los dos fragmentos anteriores? ¿Qué sucede si se intenta insertar en el segundo fragmento una cadena?

### 2.2.2.2 Iteradores.

La interfaz **Iterable** define una serie de métodos para poder recorrer las colecciones, estos métodos son:

**void foreach(Consumer <? super T> action):** Permite recorrer la estructura sin necesidad de un **for**, además de almacenar el elemento en un tipo de dato concreto.

**Iterator<T> iterator():** Devuelve un iterador del tipo indicado. **Se profundiza en los iteradores y generalizaciones en puntos siguientes del tema.**

**default Splitter<T> spliterator().** Similar al anterior, pero con la característica de que pueden **procesarse en paralelo**. **Pudiendo obtener información del proceso de procesado**, por ejemplo cuantos quedan por procesar un conjunto de imágenes (cambiar el tamaño) en la que cada tarea es independiente.

El segundo método devuelve un objeto que implementa la interfaz **Iterator** (<https://docs.oracle.com/javase/8/docs/api/java/util/Iterator.html>), ampliamente utilizada para trabajar con colecciones, la cual obliga a implementar los métodos:

- **boolean hasNext():** Indica si queda algún elemento por recorrer en el iterador.
- **<T> next():** Devuelve el siguiente elemento de la colección en caso de no quedar elementos lanza una excepción.
- **default void remove():** Borra el elemento actual del iterador.
- **default void forEachRemaining(Consumer<? super E> action).** Recorrer el iterador ejecutando una acción, pudiendo usar programación funcional.

Un ejemplo de uso de iteradores en el que se usan los métodos anteriores, extraído de w3schools:

```
import java.util.ArrayList;
import java.util.Iterator;

public class Main {
    public static void main(String[] args) {

        // Make a collection
        ArrayList<String> cars = new ArrayList<String>();
        cars.add("Volvo");
        cars.add("BMW");
        cars.add("Ford");
        cars.add("Mazda");
        cars.add("Renault");
        cars.add("Seat");
        // Get the iterator
        Iterator<String> it = cars.iterator();

        it.next();
        it.remove();
        System.out.println("-----");
        while (it.hasNext()) {
            String value = it.next();
            System.out.println(value);
        }
        it = cars.iterator();
        it.next();
        // Print the first item
        System.out.println(it.next());
        System.out.println("-----");
        // for
```

```
it.forEachRemaining(System.out::println);  
}  
}
```

### 2.2.2.3 *List.*

La interfaz lista define estructuras para gestionar elementos con **un orden y puede contener duplicados**, tal y cómo se ha comentado. Esta interfaz incluye métodos para las siguientes operaciones:

- Acceso por posición, con los métodos: get,set,add, addAll y remove.
- Búsquedas, con los métodos indexOf o lastIndexOf que devuelven los índices de los objetos buscados en caso de existir.
- Iteracion, extiende la semántica de los iteradores con ListIterator.
- Rango, extraer sublistas con el método sublist.

Además la **clase estática Collections** incluye una serie de algoritmos con operaciones comunes con las colecciones (aplicable también a Queue y Set), los algoritmos más destacados son:

- Sort: Ordena una lista de objetos, es necesario implementar la interfaz comparable en los objetos.
- Shuffle: Permuta aleatoriamente una lista.
- Swap: Intercambia dos elementos en las posiciones indicadas.
- ReplaceAll. Reemplaza **todas** las apariciones por un valor.
- Fill: Sobreescribe todos los elementos con un valor especificado.
- Copy: copia la lista origen a la lista destino.
- Binarysearch: Realiza una búsqueda binaria en una lista **ordenada**.
- indexOfSubList: devuelve el índice de la primera sublista de una Lista que es igual a otra.
- lastIndexOfSubList: devuelve el índice de la última sublista de una Lista que es igual a otra.

Se puede consultar la lista completa de algoritmos que posee la clase estática Collections en <https://docs.oracle.com/javase/8/docs/api/java/util/Collections.html>

Las clases que implementan las listas son:

**ArrayList.** Se implementa la interfaz List con un array, esto implica que en caso de quedarse sin espacio es necesario redimensionar el array, destacar que es posible indicar la capacidad del array en uno de los constructores personalizados. **Una de las colecciones más utilizadas.**

**CopyOnWriteArrayList.** Utilizada para procesos concurrentes, en el que se pueden tomar instantáneas de la lista en cada momento de forma que no se vea afectada por otras operaciones. Por ejemplo se obtiene un iterador y a continuación se borran elementos. En un ArrayList, estos cambios afectan al iterador, en caso de CopyOnWriteArrayList el iterador se encuentra apuntando a la instantánea y no se ve afectado por los cambios.

**LinkedList.** La interfaz lista se implementa con referencias, en concreto doblemente enlazadas. No es una implementación sincronizada, es decir, en programas con concurrencia pueden darse problemas de acceso.

**Stack.** La implementación de una típica pila, internamente es una extensión de la clase Vector, posee las operaciones clásicas de push y pop.

**Vector.** Similar al ArrayList, pero cuya principal diferencia es la sincronización al acceder de forma concurrente, siendo por tanto más segura pero más lenta.

Un ejemplo de funcionamiento de las listas implementadas con ArrayList, Vector y LinkedList en el que se implementa una lista de alumnos

```
public class Alumno implements Comparable<Object> {  
  
    private String nombre;  
    private String apellidos;  
    private String curso;  
  
    public Alumno() {  
    }  
  
    public Alumno(String nombre, String apellidos, String curso) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
    }  
}
```

```
this.curso = curso;

}

@Override
public int compareTo(Object o) {
    if (!(o instanceof Alumno)) {
        throw new UnsupportedOperationException("Not supported yet.");
    }
    return (this.nombre.compareTo(((Alumno) o).apellidos));
}

public String getNombre() {
    return nombre;
}

public void setNombre(String nombre) {
    this.nombre = nombre;
}

public String getApellidos() {
    return apellidos;
}

public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
}

public String getCurso() {
    return curso;
}

public void setCurso(String curso) {
    this.curso = curso;
}

public static void main(String[] args) {
    long inicio, fin;
    long tiempo;
    int tam = 1500000;
    //arraylist
```

```
List<Alumno> alumnos = new ArrayList<Alumno>();

inicio = System.currentTimeMillis();
for (int index = 0; index < tam; index++) {
    alumnos.add(new Alumno("Paco" + Math.random() * 1000000, "Uno" +
index, "1DAW" + index));
}
alumnos.get(4);
alumnos.remove(5);
alumnos.remove(1);
Iterator<Alumno> i = alumnos.iterator();
while (i.hasNext()) {
    i.next();
}
Collections.sort(alumnos);
fin = System.currentTimeMillis();
tiempo = ((fin - inicio));

System.out.println("un arraylist tarda en hacer las operaciones " +
tiempo + " milisegundos");

//vector
alumnos = new Vector<Alumno>();
inicio = System.currentTimeMillis();
for (int index = 0; index < tam; index++) {
    alumnos.add(new Alumno("Paco" + Math.random() * 1000000, "Uno" +
index, "1DAW" + index));
}

alumnos.get(4);
alumnos.remove(5);
alumnos.remove(1);
i = alumnos.iterator();
while (i.hasNext()) {
    i.next();
}
Collections.sort(alumnos);
fin = System.currentTimeMillis();
tiempo = ((fin - inicio));

System.out.println("un vector tarda en hacer las operaciones " + tiempo
+ " milisegundos");

//lista enlazada
alumnos = new LinkedList<Alumno>();
```



```
        inicio = System.currentTimeMillis();
        for (int index = 0; index < tam; index++) {
            alumnos.add(new Alumno("Paco" + Math.random() * 1000000, "Uno" +
index, "1DAW" + index));
        }
        alumnos.get(4);
        alumnos.remove(5);
        alumnos.remove(1);
        i = alumnos.iterator();
        while (i.hasNext()) {
            i.next();
        }
        Collections.sort(alumnos);
        fin = System.currentTimeMillis();
        tiempo = ((fin - inicio));

        System.out.println("un lista doblemente enlazada tarda en hacer las
operaciones " + tiempo + " milisegundos");
    }
}
```

El resultado del código anterior es:

```
un arraylist tarda en hacer las operaciones 1058 milisegundos
un vector tarda en hacer las operaciones 1040 milisegundos
un lista doblemente enlazada tarda en hacer las operaciones 1541 milisegundos
```



La lista enlazada es la más lenta, ¿ofrece alguna ventaja?



Ya sea el ArrayList, el vector o la lista doblemente enlazada el tipo de la variable que los referencia es de tipo List. ¿Cómo es posible? ¿Qué principio de la POO se utiliza?

#### 2.2.2.4 Queue.

Define, ya que es una interfaz el tipo abstracto de datos cola, la interfaz es la siguiente:

```
public interface Queue<E> extends Collection<E> {
    E element();
    boolean offer(E e);
    E peek();
}
```

```
E poll();  
E remove();  
}
```

Esta a su vez hereda de Collection, por tanto quien implemente la interfaz Queue también han de implementar los métodos de Collection.



¿Qué diferencia existe entre remove y poll? Buscar la respuesta en la API de Java.

Al igual que con las listas a partir de esta interfaz se definen una gran cantidad de colas, cada una de ellas con características propias.

[AbstractQueue](#),

[DelayQueue](#),

[PriorityBlockingQueue](#),

[ArrayBlockingQueue](#),

[LinkedBlockingDeque](#),

[PriorityQueue](#),

[ArrayDeque](#),

[LinkedBlockingQueue](#),

[SynchronousQueue](#)

[ConcurrentLinkedDeque](#),

[LinkedList](#),

[ConcurrentLinkedQueue](#),

[LinkedTransferQueue](#),



¿Cómo es posible que se tenga en las listas la clase LinkedList y también en las colas?

Por ejemplo las PriorityQueue permiten tener una cola ordenada por prioridad de objetos que se encuentran en la misma, para indicar la prioridad las clases de los objetos que se almacenan han de implementar la interfaz Comparable, de forma que esta cola sepa como ordenarlo.



A partir de las clases anteriores ¿En qué situaciones se usan las colas?

Un ejemplo del uso de la cola con prioridad de personas, que se ordena según la edad de la misma (mayor prioridad el menor):

Clase persona que implementa la interfaz Comparable:

```
/**  
 *  
 * @author Pedro  
 */
```

```
public class Person implements Comparable {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    @Override
    public int compareTo(Object o) {
        if (o instanceof Person) {

            return -1*( this.getAge() - ((Person) o).getAge());
        }
        throw new ClassCastException("the object is not of the Person class ");
    }
    public String getName() {
        return name;
    }
    public int getAge() {
        return age;
    }

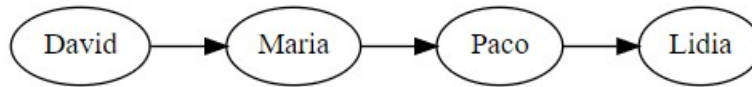
    public void setName(String name) {
        this.name = name;
    }
    public void setAge(int age) {
        this.age = age;
    }
}
```

El programa principal:

```
public class Principal {
    public static void main(String args[]) {
        PriorityQueue<Person> cola;
        Person last = null;
        cola = new PriorityQueue<Person>();
        cola.add(new Person("Lidia",19));
        cola.add(new Person("Paco", 20));
        cola.add(new Person("David", 76));
        cola.add(new Person("Maria", 22));
    }
}
```

```
}
```

La cola que se obtiene del código anterior es:



A partir de la interfaz Queue se define otra interfaz llamada Deque, es una cola doblemente enlazada, que permite la inserción y borrado en los dos extremos de la colección, pudiendo funcionar como una pila o una cola si se desea

Los principales métodos de esta interfaz son:

|                | First Element (Head)          |                               | Last Element (Tail)          |                              |
|----------------|-------------------------------|-------------------------------|------------------------------|------------------------------|
|                | <i>Throws exception</i>       | <i>Special value</i>          | <i>Throws exception</i>      | <i>Special value</i>         |
| <b>Insert</b>  | <a href="#">addFirst(e)</a>   | <a href="#">offerFirst(e)</a> | <a href="#">addLast(e)</a>   | <a href="#">offerLast(e)</a> |
| <b>Remove</b>  | <a href="#">removeFirst()</a> | <a href="#">pollFirst()</a>   | <a href="#">removeLast()</a> | <a href="#">pollLast()</a>   |
| <b>Examine</b> | <a href="#">getFirst()</a>    | <a href="#">peekFirst()</a>   | <a href="#">getLast()</a>    | <a href="#">peekLast()</a>   |

Si se compara con la interfaz cola:

| Queue Method              | Equivalent Deque Method       |
|---------------------------|-------------------------------|
| <a href="#">add(e)</a>    | <a href="#">addLast(e)</a>    |
| <a href="#">offer(e)</a>  | <a href="#">offerLast(e)</a>  |
| <a href="#">remove()</a>  | <a href="#">removeFirst()</a> |
| <a href="#">poll()</a>    | <a href="#">pollFirst()</a>   |
| <a href="#">element()</a> | <a href="#">getFirst()</a>    |
| <a href="#">peek()</a>    | <a href="#">peekFirst()</a>   |

Si se compara con una pila:

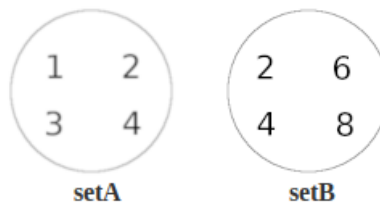
| Stack Method            | Equivalent Deque Method       |
|-------------------------|-------------------------------|
| <a href="#">push(e)</a> | <a href="#">addFirst(e)</a>   |
| <a href="#">pop()</a>   | <a href="#">removeFirst()</a> |
| <a href="#">peek()</a>  | <a href="#">peekFirst()</a>   |
|                         |                               |

Esta interfaz la heredan las siguientes clases:

[ArrayDeque](#), [ConcurrentLinkedDeque](#), [LinkedBlockingDeque](#), [LinkedList](#)

### 2.2.2.5 Set.

Esta interfaz modela conjuntos matemáticos, y al igual que esto **no es posible tener elementos duplicados** y las operaciones que define Set son similares a las operaciones matemáticas con conjuntos:

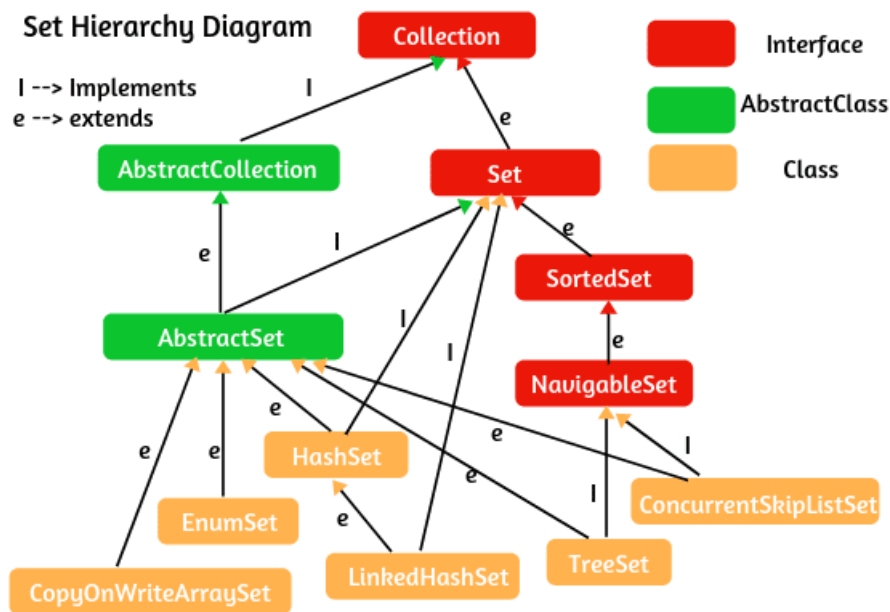


Como en cualquier conjunto no se puede garantizar una iteración en los elementos que siempre sea la misma. Además se entiende que dos Set's son iguales si tienen los mismos elementos, por lo que es necesario prestar atención a los métodos hashCode y equals de los elementos almacenados.

Los métodos son:

| Methods                        |   |
|--------------------------------|---|
| Modifier and Type              | Method and Description  |
| boolean                        | <code>add(E e)</code><br>Adds the specified element to this set if it is not already present (optional operation).  |
| boolean                        | <code>addAll(Collection&lt;? extends E&gt; c)</code><br>Adds all of the elements in the specified collection to this set if they're not already present (optional operation). |
| void                           | <code>clear()</code><br>Removes all of the elements from this set (optional operation).   |
| boolean                        | <code>contains(Object o)</code><br>Returns true if this set contains the specified element.   |
| boolean                        | <code>containsAll(Collection&lt;?&gt; c)</code><br>Returns true if this set contains all of the elements of the specified collection.   |
| boolean                        | <code>equals(Object o)</code><br>Compares the specified object with this set for equality.  |
| int                            | <code>hashCode()</code><br>Returns the hash code value for this set.  |
| boolean                        | <code>isEmpty()</code><br>Returns true if this set contains no elements.  |
| <code>Iterator&lt;E&gt;</code> | <code>iterator()</code><br>Returns an iterator over the elements in this set.   |
| boolean                        | <code>remove(Object o)</code><br>Removes the specified element from this set if it is present (optional operation).   |
| boolean                        | <code>removeAll(Collection&lt;?&gt; c)</code><br>Removes from this set all of its elements that are contained in the specified collection (optional operation).               |
| boolean                        | <code>retainAll(Collection&lt;?&gt; c)</code><br>Retains only the elements in this set that are contained in the specified collection (optional operation).                   |
| int                            | <code>size()</code><br>Returns the number of elements in this set (its cardinality).  |
| <code>Object[]</code>          | <code>toArray()</code><br>Returns an array containing all of the elements in this set.  |
| <code>&lt;T&gt; T[]</code>     | <code>toArray(T[] a)</code><br>Returns an array containing all of the elements in this set; the runtime type of the returned array is that of the specified array.            |

La jerarquía de interfaces, clases abstractas y clases relacionada con los Set es:



A partir de set se definen otras interfaces como:

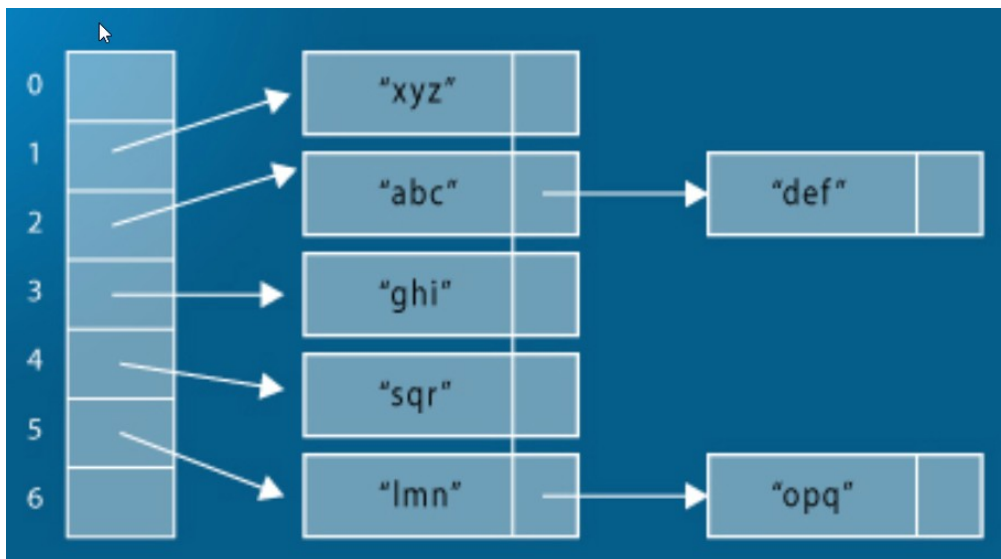
- **SortedSet:** Permite tener un Set ordenado, usando el orden natural o implementando compareTo, siendo el orden ascendente. Define los métodos **comparator**, **first**, **headSet**, **last**, **subSet** y **tailSet**.
- **NavigableSet:** Amplia a su vez a SortedSet, define métodos para obtener los elementos mayores y menores dado un elemento, borrar elementos más altos o más bajos entre otras operaciones, algunos de los métodos de NavigableSet:
  - E ceiling(E e): Devuelve el elemento menor de los mayores del elemento que se pasa.
  - E floor(E e): El mayor de los menores.
  - SortedSet<E> headSet(E toElement): Devuelve una vista de la parte de este conjunto cuyos elementos son estrictamente menores que toElement.
  - E higher(E e): Devuelve el elemento mínimo de este conjunto estrictamente mayor que el elemento dado, o nulo si no existe tal elemento.
  - E pollFirst(): Recupera y elimina el primer elemento (el más bajo), o devuelve un valor nulo si este conjunto está vacío.

De las clases finales destacan:

- **HashSet.** Se almacena el conjunto con una tabla hash, no se garantiza el orden de

la iteración.

- **TreeSet.** En este caso la estructura interna es un árbol rojo-negro, más lento que HashSet, implementa la interfaz NavigableSet.
- **LinkedHashSet.** En este caso se usa una tabla hash que contiene una lista enlazada en la que se almacena los elementos con colisión, en las listas internas se ordena por orden de llegada.



Otras clases menos usadas que implementan Set son:

- **ConcurrentSkipListSet:** Implementa la interfaz NavigableSet y su principal característica es que las operaciones de borrado, acceso e inserción se realizan de forma segura ante la existencia de diferentes hilos.
- **EnumSet:** Permite almacenar objetos de un enumerado concreto, es una estructura extremadamente compacta y eficiente. No permite elementos nulos.
- **CopyOnWriteArraySet.** Esta estructura se recomienda para casos en que las lecturas son mucho más frecuentes que las escrituras, borrados o inserciones, además de que se garantiza un acceso seguro ante varios hilos/subprocesos.
- **ConcurrentSkipListSet.** Implementación concurrente de un NavigableSet, ordenados por el ordena natural o la comparación de objetos. Las operaciones dede inserción eliminación y acceso se ejecutan de forma segura con múltiples hilos, el resto de operaciones no se puede garantizar que se realiza de forma atómica.



¿Qué significa que una operación se realiza de forma atómica? Indicar al menos otra situación en el día a día en que las operaciones sea necesario realizarlas de forma atómica.

Un ejemplo de uso de ConcurrentSkipListSet con diferentes hilos funcionando, en la que  $\frac{1}{3}$  son consumidores y el resto productores, la clase almacena objetos de tipo Order visto en ejemplos anteriores:

Los hilos que heredan de Thread:

```
/**
 *
 * @author Pedro
 */
public class Hilo extends Thread {

    private volatile boolean run = true;
    private Tiype tiype;
    private final NavigableSet<Order> orders;
    private int id;

    public enum Tiype {
        PRODUCER,
        CONSUMER,
    }

    public Hilo(NavigableSet<Order> orders, Tiype tiype, int id) {
        this.orders = orders;
        this.tiype = tiype;
        this.id = id;
    }

    public void stopThread() {
        this.run = false;
    }

    @Override
    public void run() {
        Order tempo;
        while (this.run) {
```



```
try {
    Thread.sleep((long) (Math.random() * 1000));
    if (this.tiype == Tiyte.CONSUMER) {
        //para borrar el ultimo

        tempo = this.orders.first();
        this.orders.remove(tempo);

        System.out.println("Soy " + this.id + " " + this.tiype + "
y he consumido:" + tempo);

    } else {
        tempo = new Order("dfsfs", new Date(), (float)
(Math.random() * 1000000));
        //productor
        this.orders.add(tempo);
        System.out.println("Soy " + this.id + " " + this.tiype + "
y he producido:" + tempo);

    }
} catch (InterruptedException ex) {
    Logger.getLogger(Hilo.class.getName()).log(Level.SEVERE, null,
ex);
}
}
```

El programa principal:


```
/**
 *
 * @author Pedro
 */
public class Principal {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) throws InterruptedException {
        ArrayList<Hilo> hilos = new ArrayList<Hilo>();
        ConcurrentSkipListSet<Order> orders = new
ConcurrentSkipListSet<Order>();
        Hilo tempo;
```

```
orders.add(new Order("Paco", new Date(), 4.0f));  
orders.add(new Order("Luis", new Date(), 1.0f));  
for(int i=0;i<10;i++){  
    //se crean los hilos  
    tempo=new Hilo( orders,i%3==0?  
Hilo.Tiype.CONSUMER:Hilo.Tiype.PRODUCER,i);  
    hilos.add(tempo);  
    //se inician  
    tempo.start();  
}  
Thread.sleep(1000);  
//se paran  
for(int i=0;i<hilos.size();i++){  
    hilos.get(i).stopThread();  
}  
}
```

La salida (dependerá de cuando entre cada uno, es decir cada ejecución es diferente) es:

```
Soy 6 CONSUMER y he consumido:Client: Luis Date:14/03/2022 Amount:1.0  
Soy 3 CONSUMER y he consumido:Client: Luis Date:14/03/2022 Amount:1.0  
Soy 2 PRODUCER y he producido:Client: dfsfs Date:14/03/2022 Amount:767481.5  
Soy 9 CONSUMER y he consumido:Client: dfsfs Date:14/03/2022 Amount:104751.445  
Soy 5 PRODUCER y he producido:Client: dfsfs Date:14/03/2022 Amount:353579.4  
Soy 6 CONSUMER y he consumido:Client: dfsfs Date:14/03/2022 Amount:70639.984  
Soy 1 PRODUCER y he producido:Client: dfsfs Date:14/03/2022 Amount:324487.3  
Soy 8 PRODUCER y he producido:Client: dfsfs Date:14/03/2022 Amount:685588.3  
Soy 0 CONSUMER y he consumido:Client: dfsfs Date:14/03/2022 Amount:24190.75  
Soy 7 PRODUCER y he producido:Client: dfsfs Date:14/03/2022 Amount:739824.56  
Soy 4 PRODUCER y he producido:Client: dfsfs Date:14/03/2022 Amount:668769.2  
Soy 9 CONSUMER y he consumido:Client: dfsfs Date:14/03/2022 Amount:244038.73  
Soy 1 PRODUCER y he producido:Client: dfsfs Date:14/03/2022 Amount:455745.5  
Soy 5 PRODUCER y he producido:Client: dfsfs Date:14/03/2022 Amount:211205.9
```

 Modificar el código anterior y sustituir el ConcurrentLinkedDeque por  
LinkedHashSet y ejecutar el código ¿Qué sucede? ¿A qué se debe?



Se ha usado la palabra reservada volatile para definir un atributo del hilo, ¿para qué

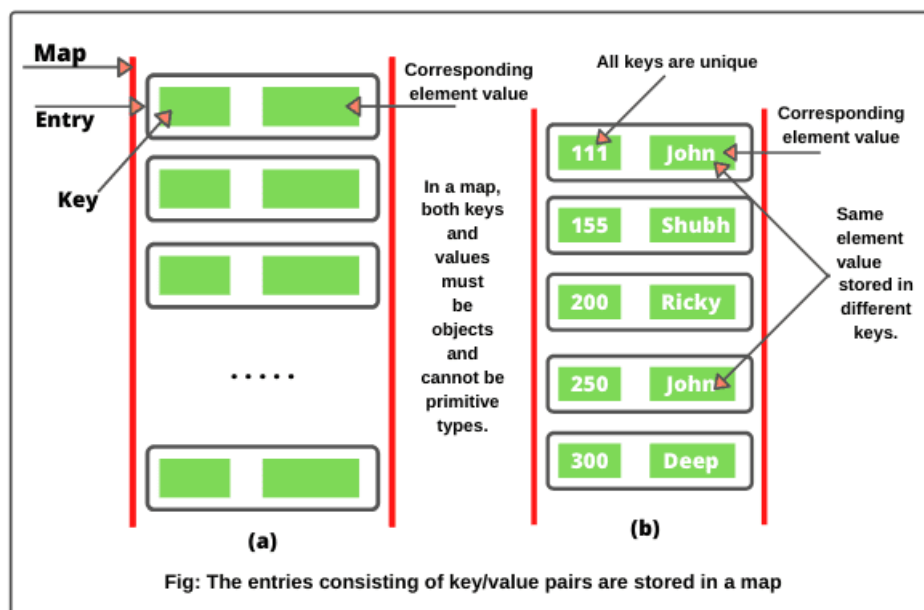
se utiliza?

### 2.2.2.6 Map.

A pesar de no ser una colección pura (no hereda de Collection) los Maps son considerados otro tipo de colección, siendo una de las colecciones más usadas en programación por su principal característica:



**Acceder a los elementos por la clave que puede ser un entero, una cadena o un objeto entre otros.**



Al igual que las colecciones anteriores, la base de la jerarquía de clases de los Maps es una interfaz llamada  $\text{Map}\langle K, V \rangle$ , donde K y V son respectivamente la clase que indica el tipo de objetos que será la llave y V que es la clase que indica el tipo de objetos que gestionará el mapa.

Sustituyo a la clase abstracta Dictionary, por lo que es posible que ambos términos se usen indistintamente para referirse al concepto de tabla Hash aunque internamente.

Dependiendo de la implementación se puede o no garantizar el orden, si se implementa internamente con un árbol es posible garantizar el orden, en el caso de usar una tabla Hash no es posible garantizar siempre el mismo orden al iterar. Dependiendo de la implementación también se imponen restricciones con respecto a los valores de clave válidos, como puede ser permitir claves nulas.

Algunos de los métodos que define esta interfaz son:

- `void clear()`. Elimina todas las asignaciones de este mapa (operación opcional).
- `boolean containsKey(Object key)`. Devuelve verdadero si este Map contiene algún objeto con la clave dada.
- `boolean containsValue(Object value)`. Devuelve verdadero si este mapa asigna una o más claves al valor especificado.
- `V get(Object key)`. Devuelve el objeto V asociado a la clave key.
- `V put(K key, V value)`. Asocia el valor especificado con la clave especificada.
- `V remove(Object key)`. Elimina el mapeo de una clave de este mapa si está presente.
- `default V replace(K key, V value)`. Reemplaza la entrada de la clave especificada solo si actualmente está asignada a algún valor.
- `int size()`. Devuelve el número de asignaciones de clave-valor en esta asignación.
- `Collection<V> values()`. Devuelve un objeto de tipo interfaz Collection con los valores.



En algunos métodos se tiene la palabra default ¿qué significa?

De esta interfaz es heredada por las siguientes interfaces:

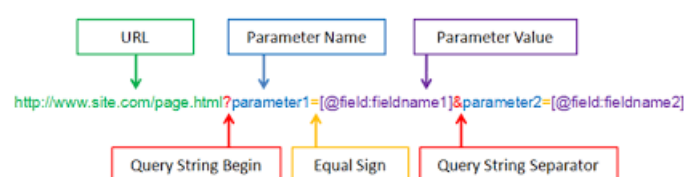
- Bindings.
- ConcurrentMap<K,V>.
- ConcurrentNavigableMap<K,V>.
- LogicalMessageContext.
- MessageContext.
- NavigableMap<K,V>.
- SOAPMessageContext,
- SortedMap<K,V>

Y las clases que implementan la interfaz son las siguientes:

- AbstractMap.
- Attributes.
- AuthProvider.
- ConcurrentHashMap.
- ConcurrentSkipListMap.
- EnumMap.
- HashMap.
- Hashtable.

- IdentityHashMap.
- **LinkedHashMap.**
- PrinterStateReasons.
- Properties.
- Provider.
- RenderingHints.
- **HashMap.** Utiliza una tabla de dispersión para almacenar la información del mapa. Las operaciones básicas (get y put) se harán en tiempo constante siempre que se dispersen adecuadamente los elementos. El coste de la iteración dependerá del número de entradas de la tabla y del número de elementos del mapa. No se garantiza que se respete el orden de las claves.
- **TreeMap.** Utiliza un árbol rojo-negro para implementar el mapa. El coste de las operaciones básicas será logarítmico con el número de elementos del mapa  $O(\log n)$ . En este caso los elementos se encontrarán ordenados por orden ascendente de clave.
- **Hashtable.** Es una implementación similar a HashMap, pero con alguna diferencia. Mientras las anteriores implementaciones no están sincronizadas, esta sí que lo está. Además en esta implementación, al contrario que las anteriores, no se permitirán claves nulas (null). Este objeto extiende la obsoleta clase Dictionary, ya que viene de versiones más antiguas de JDK.
- **EnumMap.** La clave ha de ser de un tipo de enumerado concreto. No permitiendo claves nulas y no se encuentra sincronizado.
- SimpleBindings.
- TabularDataSupport.
- **TreeMap.**
- UIDefaults.
- WeakHashMap.

Un ejemplo de uso de la interfaz Map muy usado en el desarrollo web es obtener el query string por GET o el cuerpo del mensaje por POST y pasar las variables a un Map de forma que sea posible su acceso por el nombre de la variable.



```
public class Principal {

    /**
     * @param args the command line arguments
     */
    public static void main(String[] args) {
        String cadena = "https://example.com/path/to/page?
name=ferret&color=purple";
        HashMap<String, String> parameters = new HashMap<String, String>();
        //se obtiene la parte del querystring
        String querystring = cadena.substring(cadena.indexOf('?') + 1);
        //se trocean los parametros
        StringTokenizer st = new StringTokenizer(querystring, "&");
        //temporales necesarios
        String parametro, clave, valor;
        //se recorren los trozos y se inserta en un HashMap
        while (st.hasMoreElements()) {
            parametro = st.nextToken();
            clave = parametro.substring(0, parametro.indexOf("="));
            valor = parametro.substring(parametro.indexOf("=") + 1);
            parameters.put(clave, valor);
        }
        //se recorre, es posible también consultar por clave con get
        for (String cclave : parameters.keySet()) {
            System.out.println("Clave:" + cclave + " valor:" +
parameters.get(cclave));
        }
    }
}
```

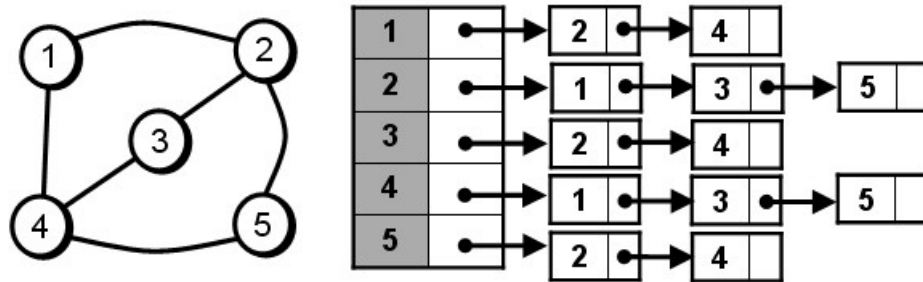
La salida del programa anterior es:

```
Clave:color valorpurple
Clave:name valorferret
```

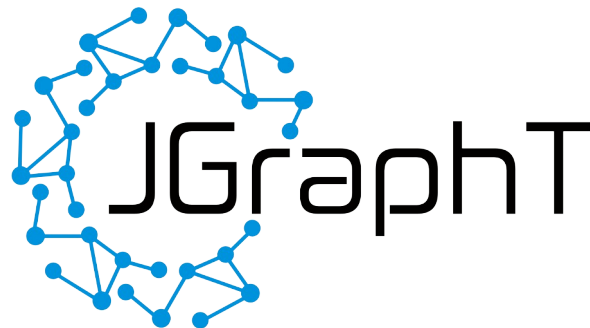
## 2.2.3 Otras estructuras de datos.

### 2.2.3.1 Grafos en Java.

Java no posee de forma nativa colecciones que representen grafos, representandose como matrices de adyacencia, listas de adyacencia o matrices dispersas.

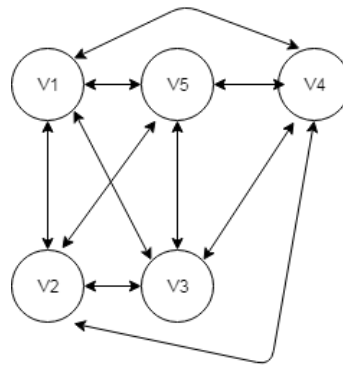


Lo común común es utiliza librerías. Una de las más conocidas es JgraphT(  
<https://jgrapht.org/>), aunque existen muchas otras.



Por ejemplo, para definir un grafo completo:

```
public void createCompleteGraph() {
    completeGraph = new SimpleWeightedGraph<>(DefaultEdge.class);
    CompleteGraphGenerator<String, DefaultEdge> completeGenerator
        = new CompleteGraphGenerator<>(size);
    VertexFactory<String> vFactory = new VertexFactory<String>() {
        private int id = 0;
        public String createVertex() {
            return "v" + id++;
        }
    };
    completeGenerator.generateGraph(completeGraph, vFactory, null);
}
```



Se recomienda la lectura además de la documentación técnica, el artículo <https://www.baeldung.com/jgrapht> sobre el uso de Jgrapht.

## 2.2.4 Generalización.

En las colecciones en Java es posible restringir las clases permitidas al definir y crear un Set, un Queue o un List entre otros. Esto es posible al utilizar los Generics, pero no se limita únicamente a las colecciones ya creadas, sino que es posible su utilización en clases desarrolladas a posteriori.

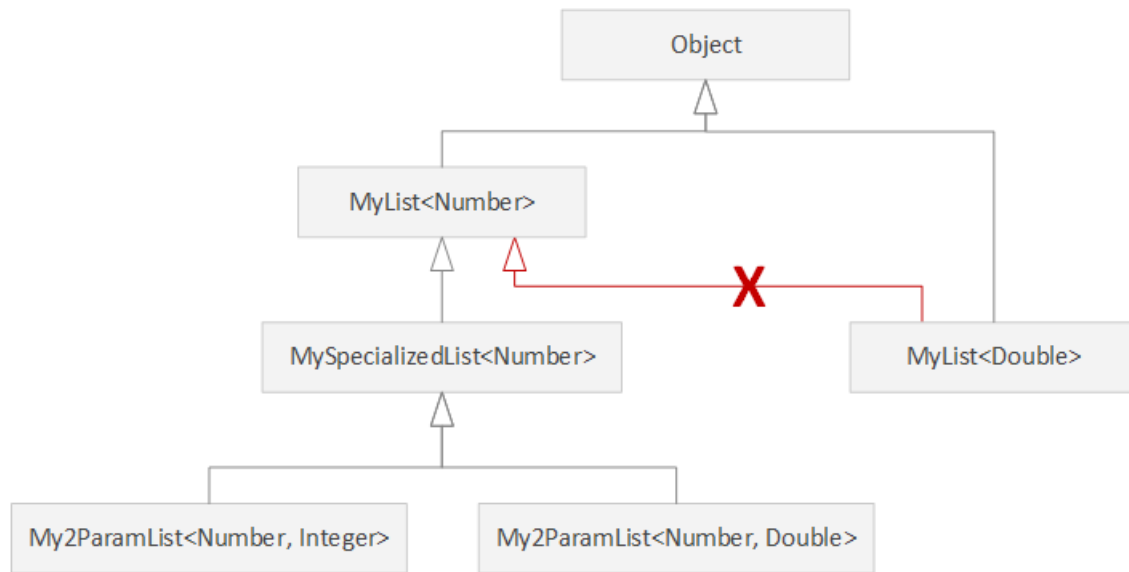
Los tipos genéricos (Generics) permiten establecer restricciones a nivel de tipo, haciendo que ciertas clases, interfaces o métodos acepten únicamente los tipos estipulados.

Su uso está ligado, mayoritariamente, a las colecciones (Collections), donde ayudan a realizar comprobación de tipos en tiempo de compilación.

Los beneficios son:

- Comprobación de tipos más fuerte en tiempo de compilación.
- Eliminación de casts aumentando la legibilidad del código.
- Posibilidad de implementar algoritmos genéricos, con tipado seguro.





Cuando se trata con jerarquías los genéricos **son invariantes**, esto es, dado un Tipo1 subtipo de otro Tipo2, para los genéricos, List<Tipo1> no es considerado como subtipo ni un supertipo de List <Tipo2> excepto en el caso trivial de que A y B sean idénticos.

Una clase genérica puede tener múltiples argumentos de tipos y los argumentos pueden ser a su vez tipos genéricos. Después del nombre de la clase se puede indicar la lista de parámetros de tipos con el formato <T1, T2, T3, ...>.

Según las convenciones los nombres de los parámetros de tipo usados comúnmente son los siguientes:

- E: elemento de una colección.
- K: clave.
- N: número.
- T: tipo.
- V: valor.
- S, U, V etc: para segundos, terceros y cuartos tipos.

Además de las clases los métodos también pueden tener su propia definición de tipos genéricos.

```

public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {

    return p1.getKey().equals(p2.getKey()) &&

```

```
p1.getValue().equals(p2.getValue());  
  
}
```

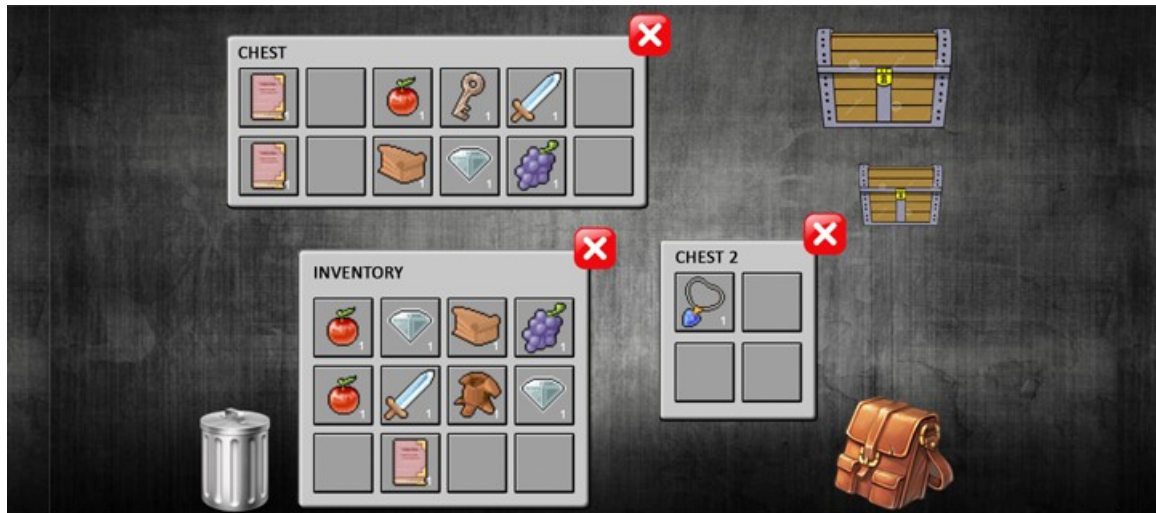
Por ejemplo si se desea tener una clase bolsa con la característica que solo puede almacenar objetos de una clase concreta se define la siguiente clase utilizando genéricos:

```
public class Bolsa<T> implements Iterable<T>{  
  
    private ArrayList<T> lista= new ArrayList<T>();  
    private int tope;  
    public Bolsa(int tope) {  
        super();  
        this.tope = tope;  
    }  
    public void add(T objeto ) {  
        if (lista.size()<=tope) {  
            lista.add(objeto);  
        }else {  
            throw new RuntimeException("bolsa llena");  
        }  
    }  
    public Iterator<T> iterator() {  
        return lista.iterator();  
    }  
  
}
```

Para instanciar la clase anterior:

```
Bolsa<Chocolatina> bolsa= new Bolsa<Chocolatina>();  
Chocolatina c= new Chocolatina("milka");  
Chocolatina c2= new Chocolatina("ferrero");
```

Ejemplo uso Generics: Se necesita implementar una estructura de datos para gestionar inventarios, por ejemplo un juego en el que cada personaje posee un conjunto de armas, además de vestuario, comida y objetos de diferente uso.



Se ha decidido implementar una clase generics reutilizable en otros juegos con la limitación de que en cada inventario solo es posible tener elementos de un tipo o que hereden de este, por ejemplo en el inventario de armas solo es posible almacenar objetos cuyas clases deriven de la clase arma y se desea también poder ordenar de forma sencilla el inventario.

El inventario ha de implementar al menos los métodos genéricos siguientes:

- addElement
- getElement
- joinInventory
- order
- getElementByName
- removeElementByName
- removeElementByIndex

Se definen a su vez la clase base Arma, y dos hijas: Pistola y Rifle y la clase base Vestuario, con dos clases hijas: Coraza y CotadeMaya.

Clase arma:

```
public abstract class Arma implements Comparable<Arma> {

    protected int balas;
    protected int distancia;
    protected int danyo;
```

```
protected String nombre;
public Arma() {
}
public Arma(int balas, int distancia, int danyo) {
    this.balas = balas;
    this.distancia = distancia;
    this.danyo = danyo;
}
public abstract void recargar();
public abstract void recargar(int balas);
public abstract void disparar(int distancia);
@Override
public int compareTo(Arma o) {
    return this.nombre.compareTo(o.getNombre());
    // throw new UnsupportedOperationException("Not supported
yet."); //To change body of generated methods, choose Tools | Templates.
}
public String toString(){
    return this.nombre+" balas:"+this.balas;
}
/**
 * @return the balas
 */
public int getBalas() {
    return balas;
}

/**
 * @return the distancia
 */
public int getDistancia() {
    return distancia;
}

/**
 * @return the danyo
 */
public int getDanyo() {
    return danyo;
}
```

```
/**
 * @return the nombre
 */
public String getNombre() {
    return nombre;
}

/**
 * @param balas the balas to set
 */
public void setBalas(int balas) {
    this.balas = balas;
}

/**
 * @param distancia the distancia to set
 */
public void setDistancia(int distancia) {
    this.distancia = distancia;
}

/**
 * @param danyo the danyo to set
 */
public void setDanyo(int danyo) {
    this.danyo = danyo;
}

/**
 * @param nombre the nombre to set
 */
public void setNombre(String nombre) {
    this.nombre = nombre;
}
}
```

Clase Pistola:

```
public class Pistola extends Arma {
    public Pistola() {
        super(7, 50, 50);
    }
}
```

```
}  
  
@Override  
public void recargar() {  
    this.setBalas(7);  
}  
  
@Override  
public void recargar(int balas) {  
    if(balas<7)  
        this.balas=balas;  
    else  
        throw new UnsupportedOperationException("Not supported yet."); //To  
change body of generated methods, choose Tools | Templates.  
}  
  
@Override  
public void disparar(int distancia) {  
    if(this.distancia>distancia)  
        this.balas--;  
}  
  
}
```

Clase Rifle:

```
public class Rifle extends Arma{  
    boolean miratelescopica;  
    boolean rafaga;  
    public Rifle() {  
        super(50, 200, 30);  
        this.miratelescopica=false;  
        this.rafaga=false;  
    }  
    @Override  
    public void recargar() {  
        this.setBalas(50);  
    }  
  
    @Override  
    public void recargar(int balas) {  
        if(balas<50)
```

```
        this.balas=balas;
    else
        throw new UnsupportedOperationException("Not supported yet."); //To
change body of generated methods, choose Tools | Templates.
    }

    @Override
    public void disparar(int distancia) {

        if(this.distancia+ (this.miratelescopica?20:0)>distancia)
            if(this.rafaga)
                this.balas-=10;
            else
                this.balas--;
    }

}
```

Clase inventario:

```
public class Inventario<T extends Comparable<T>> implements Iterable<T> {
    private ArrayList<T> elementos;
    public Inventario(){
        this.elementos= new ArrayList<T>();
    }
    public void addElement(T element){
        this.elementos.add(element);
    }
    public T getElement(int index) {
        return this.getElement(index);
    }
    public void joinInventory(Inventario<T> inventory){
        this.elementos.addAll(elementos);
    }
    public void sort(){
        //TODO
    }
    @Override
    public Iterator<T> iterator() {
        return elementos.iterator();
    }
}
```

```
}
```

Programa principal:

```
public class Principal {  
  
    /**  
     * @param args the command line arguments  
     */  
    public static void main(String[] args) {  
        Inventario<Arma> armas= new Inventario<Arma>();  
        Pistola p= new Pistola();  
        p.setNombre("Beretta 92 F");  
        armas.addElement(p);  
        p= new Pistola();  
        p.setNombre("Smith & Wesson MP 9 ");  
        armas.addElement(p);  
        Rifle r= new Rifle();  
        r.setNombre("AK-47");  
        armas.addElement(r);  
        r= new Rifle();  
        r.setNombre("M-16");  
        armas.addElement(r);  
        //armas.sort();  
        for (Arma a:armas) {  
            System.out.println(a);  
        }  
    }  
}
```



¿De qué forma se obliga a que todas clases que se metan en un inventario implementen Comparable?



Pensar si es posible hacer un new en los métodos o clases generics.



Implementar el método sort.



Implementar los métodos restantes del inventario.



- getElementByName
- removeElementByName
- removeElementByIndex



Crear la jerarquía de clases para vestuario y usarla.



¿Qué sucede si se desea almacenar en un inventario de armas un objeto de la clase vestuario?

## 2.2.5 Interfaces funcionales.

Las interfaces funcionales se basan en poder cambiar el código a ejecutar en tiempo de ejecución a través de las expresiones lambdas, que no son más que funciones sin nombre ni clase asociada, también conocidas como funciones anónimas.



Introducción UPM sobre programación funcional en Java.

<https://www.youtube.com/watch?v=8LVdhVMNQSw>

La sintaxis de una función lambda es similar a la de las funciones, excepto por que no posee nombre, en caso de usar un único parámetro no es necesario usar los paréntesis y la cabecera de la función se separa del cuerpo de la función con el operador lambda →.



(parámetros) → {cuerpo-lambda}

Algunos ejemplo, observar el segundo en el que al tener un único parámetro no son necesarios los paréntesis:

- `() -> System.out.println("Hello Lambda")`
- `x -> x + 10`
- `(int x, int y) -> { return x + y; }`
- `(String x, String y) -> x.length() - y.length()`
- `(String x) -> {  
    listA.add(x);  
    listB.remove(x);  
    return listB.size();  
}`

Un ejemplo del uso de las interfaces personalizada y funciones anónimas:

```
public class funcional1 {  
    //interfaz de calculadora con las operaciones clásicas  
  
    public interface ICalculadora<T> {  
        public T Suma(T op1, T op2);  
        /*public T Resta (T op1, T op2);  
        public T Multiplicacion (T op1, T op2);  
        public T Division (T op1, T op2);*/  
    }  
  
    public static void main(String[] args) {  
        /*creación de un objeto que implementa la interzaz asignando el método  
        suma solo a entereros y usandolo*/  
        System.out.println(  
            new ICalculadora<Integer>() {  
                @Override  
                public Integer Suma(Integer op1, Integer op2) {  
                    return op1.intValue() + op2.intValue();  
                }  
            }.Suma(5, 6).toString());  
  
        /* similar al anterior pero se crea el objeto y se indica la lógica al  
        crearlo*/  
  
        ICalculadora<String> c= (String op1, String op2)->{  
            return op1.concat(op2);  
        };  
  
        System.out.println(c.Suma("Hola ", "mundo"));  
    }  
}
```

Java define una serie de interfaces funcionales a las cual se les facilita funciones lambda con una condiciones en el paquete **java.util.function**., estas interfaces solo poseen un **único método abstracto** (puede poseer otros no abstractos por defecto), la concrección en tiempo de ejecución de el método será proveída por una función lambda.

Las características de las interfaces funcionales en Java son:

- Son interfaces , como su nombre indica.
- Tienen un **único** método abstracto.

- Se pueden introducir métodos por defecto.
- Se pueden introducir métodos estáticos.
- Se indican con la anotación `@FunctionalInterface`.

Estas interfaces son:

- **Consumidores. Consumer <T>**. Aceptan un valor y no devuelven nada  $(x) \rightarrow \{\dots\}$ , existen las bicosumidoras que aceptan 2 valores  $(x,y) \rightarrow \{\dots\}$ . Posee un método para composición de funciones `andThen`.

```
@FunctionalInterface
public interface Consumer <T>{
    void accept(T t);
}
```

Un ejemplo de uso de consumer para imprimir una lista:

```
public static void main(String[] args) {
    ArrayList<Integer> lista = new ArrayList<Integer>();
    for (int i = 0; i < 20; i++) {
        lista.add((int) (Math.random() * 10000));
    }
    Consumer<Integer> c = (Integer i) -> System.out.println(i);
    for (Integer in : lista) {
        c.accept(in);
    }
}
```

Haciendo uso de `andThen`:

```
public static void main(String[] args) {
    ArrayList<Integer> lista = new ArrayList<Integer>();
    for (int i = 0; i < 20; i++) {
        lista.add((int) (Math.random() * 10000));
    }
    Consumer<Integer> c = (Integer i) -> System.out.print(i);
    Consumer concatenado = c.andThen(j -> System.out.println("->" + (j+1)));
    for (Integer in : lista) {
        concatenado.accept(in);
    }
}
```

Parte de la salida del código anterior es:

2985->2986

8199->8200

7636->7637

- **Proveedores.** No recibe parámetros pero devuelve uno (`()`){...return x;}. Util para la creación de objetos (factorías).

```
@FunctionalInterface  
public interface Supplier <T>{  
    T get();  
}
```

En el siguiente ejemplo se implementa una **factoría** de Objetos en la que se pueden llamar pasando el nombre de la clase o bien pasándole un método al que no se le pase nada y devuelva algo, en este caso el objeto, los objetos de la factoría o son personas o hederan de esta.

```
public class Funcional3Producer {  
  
    public class Factoria {  
        private static HashMap<String,Supplier<Persona>> clases;  
        static{  
            clases= new HashMap<>();  
            clases.put("persona",Persona::new);  
            clases.put("estudiante", Estudiante::new);  
        }  
        public static Persona get(Supplier<? extends Persona> s){  
            return s.get();  
        }  
        public static Persona create(String nombre) {  
            if(Factoria.clases.get(nombre)!=null)  
                return Factoria.clases.get(nombre).get();  
            else  
                return null;  
        }  
    }  
  
    public static void main(String[] args) {  
        Estudiante e=(Estudiante) Factoria.create("estudiante");  
        System.out.println(e.getClass().getName());  
    }  
}
```

- **Funciones.** Aceptan un argumento y devuelve un valor  $(x) \rightarrow \{ \text{return } y; \}$ , existiendo

bifunciones con 2 argumentos. Los métodos que implementa la interfaz son `apply`, `andThen`, `compose` e `identity`. Para llamar a función usar `apply`. Es posible concatenar diferentes llamadas a interfaces `Function` con el método `andThen`, o usar composición con el método `compose`.

La interfaz es la siguiente:

```
@FunctionalInterface
public interface Function<T, R> {
    R apply(T t);

    default <V> Function<V, R> compose(Function<? super V, ? extends T> before) {
        Objects.requireNonNull(before);
        return (V v) -> apply(before.apply(v));
    }

    default <V> Function<T, V> andThen(Function<? super R, ? extends V> after) {
        Objects.requireNonNull(after);
        return (T t) -> after.apply(apply(t));
    }

    static <T> Function<T, T> identity() {
        return t -> t;
    }
}
```

- **Predicados.** Aceptar un parámetro y devuelve un valor lógico, existiendo también los bipredicados. Posee diferentes métodos, siendo posible crear expresiones booleanas con los métodos, **and**, **or** y **negate**, al que se le pasa otros predicados, para evaluar el predicado usar el método `test` con el valor a evaluar. Un ejemplo de uso, extraído de [Javatpoint.com](http://Javatpoint.com):

```
import java.util.function.Predicate;

public class PredicateInterfaceExample {
    static Boolean checkAge(int age) {
        if (age > 17)
            return true;
        else return false;
    }

    public static void main(String[] args) {
        // Using Predicate interface
        Predicate<Integer> predicate =
        PredicateInterfaceExample::checkAge;

        // Calling Predicate method
    }
}
```

```
boolean result = predicate.test(25);  
System.out.println(result);  
  
}  
}
```

- **Operadores.** Recibe un parámetro y devuelve otro del mismo tipo, extiende la interfaz Function, la definición es:

```
@FunctionalInterface  
public interface UnaryOperator<T>  
extends Function<T,T>
```

Y es posible usar los métodos then, apply y compose vistos en interfaces anteriores, existiendo también el operador binario con 2 parámetros y con la condición que tanto los dos parámetros como el valor devuelto sea del mismo tipo.



Realizar un pequeño resumen de las interfaces funcionales, sus parámetros y su uso.

Existen muchas otras interfaces funcionales definidas en Java, siendo variaciones de las anteriores.

| Interface            | Description  |
|----------------------|--|
| BiConsumer<T,U>      | Represents an operation that accepts two input arguments and returns no result.                                  |
| BiFunction<T,U,R>    | Represents a function that accepts two arguments and produces a result.  |
| BinaryOperator<T>    | Represents an operation upon two operands of the same type, producing a result of the same type as the operands. |
| BiPredicate<T,U>     | Represents a predicate (boolean-valued function) of two arguments.   |
| BooleanSupplier      | Represents a supplier of boolean-valued results.   |
| Consumer<T>          | Represents an operation that accepts a single input argument and returns no result.                              |
| DoubleBinaryOperator | Represents an operation upon two double-valued operands and producing a double-valued result.                    |
| DoubleConsumer       | Represents an operation that accepts a single double-valued argument and returns no result.                      |
| DoubleFunction<R>    | Represents a function that accepts a double-valued argument and produces a result.                               |
| DoublePredicate      | Represents a predicate (boolean-valued function) of one double-valued argument.                                  |
| DoubleSupplier       | Represents a supplier of double-valued results.  |
| DoubleToIntFunction  | Represents a function that accepts a double-valued argument and produces an int-valued result.                   |
| DoubleToLongFunction | Represents a function that accepts a double-valued argument and produces a long-valued result.                   |
| DoubleUnaryOperator  | Represents an operation on a single double-valued operand that produces a double-valued result.                  |
| Function<T,R>        | Represents a function that accepts one argument and produces a result.   |
| IntBinaryOperator    | Represents an operation upon two int-valued operands and producing an int-valued result.                         |
| IntConsumer          | Represents an operation that accepts a single int-valued argument and returns no result.                         |
| IntFunction<R>       | Represents a function that accepts an int-valued argument and produces a result.                                 |
| IntPredicate         | Represents a predicate (boolean-valued function) of one int-valued argument.                                     |
| IntSupplier          | Represents a supplier of int-valued results.   |
| IntToDoubleFunction  | Represents a function that accepts an int-valued argument and produces a double-valued result.                   |
| IntToLongFunction    | Represents a function that accepts an int-valued argument and produces a long-valued result.                     |
| IntUnaryOperator     | Represents an operation on a single int-valued operand that produces an int-valued result.                       |
| LongBinaryOperator   | Represents an operation upon two long-valued operands and producing a long-valued result.                        |
| LongConsumer         | Represents an operation that accepts a single long-valued argument and returns no result.                        |
| LongFunction<R>      | Represents a function that accepts a long-valued argument and produces a result.                                 |
| LongPredicate        | Represents a predicate (boolean-valued function) of one long-valued argument.                                    |
| LongSupplier         | Represents a supplier of long-valued results.  |

Muchas de los métodos de las colecciones vistas en el tema reciben interfaces funcionales, por ejemplo el método `removeIf` recibe una interfaz funcional de tipo predicado al que se le pasa un elemento, eliminándolo de la colección en caso de ser cierto el método `foreach` de la interfaz `Map`. Por ejemplo, en un juego de aviones para eliminar las balas que salen de la pantalla:

```
private void moveBullets() {  
    for (Bullet b : this.bullets) {  
        b.move();  
    }  
    this.bullets.removeIf(b -> (b.getPosicion().getX() <= b.getInc() || !b.isLive() ||  
b.hasCollided()));  
}
```



¿A qué método de la interfaz funcional llamara `removeIf` para evaluar?



Pensar en cómo sería el código anterior sin usar interfaces funcionales.

## 2.2.6 Streams.

Los streams o flujo de datos son otro de los elementos de la programación funcional que permiten gestionar de forma sencilla operaciones con colecciones que de forma tradicional tendría un alto coste de implementación y/o diseño.

Estos flujos de datos se pueden obtener de diferentes fuentes como arrays, **colecciones**, generadores, ficheros u otros canales.

Una vez creado los streams es posible realizar un conjunto de operaciones sobre cada uno de los elementos del flujo que puede ser sin estado (statless) o con estado.



**Las operaciones tienen como parámetro interfaces funcionales de los tipos vistos en el punto anteriores.**

Las operaciones se pueden encontrar para Java 8 en <https://docs.oracle.com/javase/8/docs/api/java/util/stream/Stream.html>.

| All Methods       | Static Methods | Instance Methods  | Abstract Methods | Default Methods |
|-------------------|----------------|---|------------------|-----------------|
| Modifier and Type |                | Method and Description  |                  |                 |
| boolean           |                | <b>allMatch</b> (Predicate<? super T> predicate)<br>Returns whether all elements of this stream match the provided predicate.   |                  |                 |
| boolean           |                | <b>anyMatch</b> (Predicate<? super T> predicate)<br>Returns whether any elements of this stream match the provided predicate.   |                  |                 |
| <R, A> R          |                | <b>collect</b> (Collector<? super T, A, R> collector)<br>Performs a <b>mutable reduction</b> operation on the elements of this stream using a Collector.  |                  |                 |
| <R> R             |                | <b>collect</b> (Supplier<R> supplier, BiConsumer<R, ? super T> accumulator, BiConsumer<R, R> combiner)<br>Performs a <b>mutable reduction</b> operation on the elements of this stream.   |                  |                 |
| long              |                | <b>count</b> ()<br>Returns the count of elements in this stream.  |                  |                 |
| Stream<T>         |                | <b>distinct</b> ()<br>Returns a stream consisting of the distinct elements (according to <code>Object.equals(Object)</code> ) of this stream.   |                  |                 |
| Stream<T>         |                | <b>filter</b> (Predicate<? super T> predicate)<br>Returns a stream consisting of the elements of this stream that match the given predicate.  |                  |                 |
| Optional<T>       |                | <b>findAny</b> ()<br>Returns an <b>Optional</b> describing some element of the stream, or an empty <b>Optional</b> if the stream is empty.  |                  |                 |
| Optional<T>       |                | <b>findFirst</b> ()<br>Returns an <b>Optional</b> describing the first element of this stream, or an empty <b>Optional</b> if the stream is empty.  |                  |                 |
| <R> Stream<R>     |                | <b>flatMap</b> (Function<? super T, ? extends Stream<? extends R>> mapper)<br>Returns a stream consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.                |                  |                 |
| DoubleStream      |                | <b>flatMapToDouble</b> (Function<? super T, ? extends DoubleStream> mapper)<br>Returns an <b>DoubleStream</b> consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element. |                  |                 |
| IntStream         |                | <b>flatMapToInt</b> (Function<? super T, ? extends IntStream> mapper)<br>Returns an <b>IntStream</b> consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.          |                  |                 |
| LongStream        |                | <b>flatMapToLong</b> (Function<? super T, ? extends LongStream> mapper)<br>Returns an <b>LongStream</b> consisting of the results of replacing each element of this stream with the contents of a mapped stream produced by applying the provided mapping function to each element.       |                  |                 |

Existen alrededor de 20 operadores no terminales o recolectores, destacando:

| Nombre   | Definición  | Descripción   |
|----------|---|---|
| Distinct | Stream<T> <b>distinct()</b>   | Devuelve un stream con los elementos que sean diferentes en el stream.  |
| Filter   | Stream<T> <b>filter</b> (Predicate<? super T> predicate)                                      | Devuelve un stream con los elementos que sean ciertos para la interfaz de tipo Predicado que se le pasa   |
| FlatMap  | <R> Stream<R> <b>flatMap</b><br>( Function <? super T, ? extends Stream<? extends R>> mapper) | Aplana un flujo, une diferentes flujos de datos en uno solo, por ejemplo procesar una lista cuyos elementos a su vez contienen una colección que se ha de procesar. |
| Limit    | Stream<T> <b>limit</b> (long maxSize)   | Un nuevo flujo con un tamaño máximo   |
| Map      | <R> Stream<R> <b>map</b> (Function<? super T, ? extends R> mapper)                            | Devuelve un nuevo flujo con elementos a los que se les ha aplicado la interfaz funcional indicada.  |
| Peek     | Stream<T> <b>peek</b> (Consumer<? super T> action)  | Recibe una interfaz consumidor, es útil para depuración.  |
| Skip     | Stream<T> <b>skip</b> (long n)  | Elimina los n primeros elementos del flujo de datos.  |
| Sorted   | Stream<T> <b>sorted()</b>   | Devuelve un flujo ordenado.   |



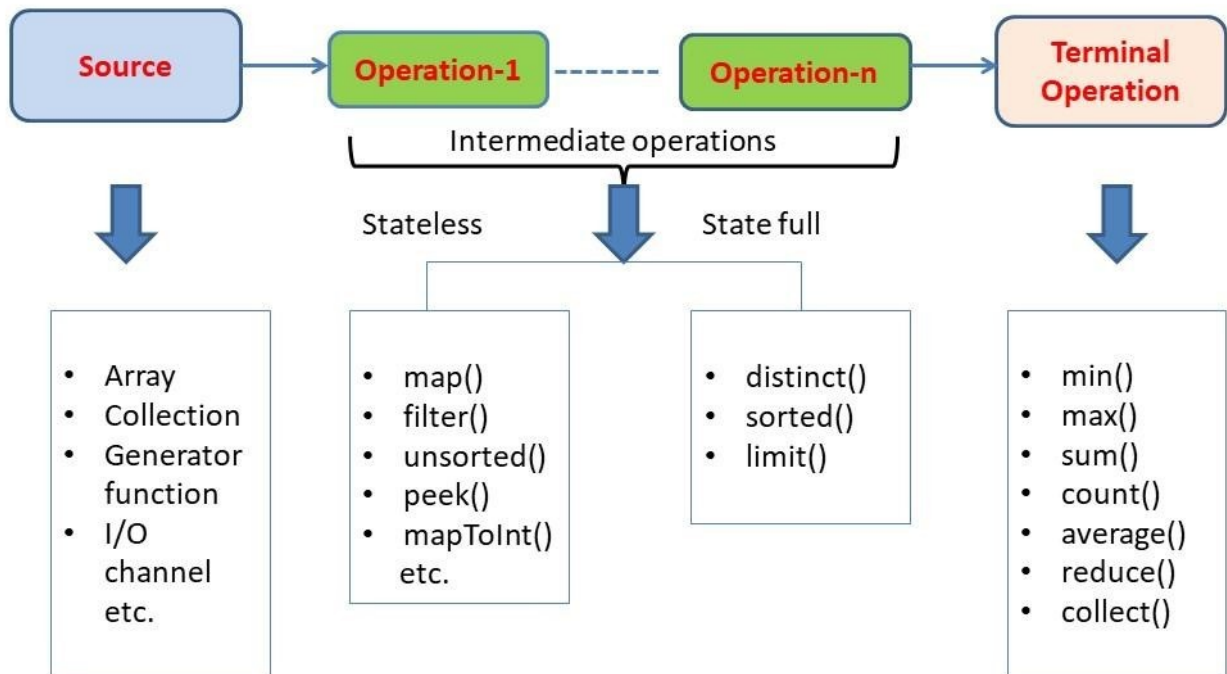
**Destacar que las operaciones intermedias devuelven a su vez un stream sobre el cual se puede volver a aplicar otra operación y así una vez tras otra.**

Los stream permiten la ejecución en paralelo llamando al método `parallel()` antes de ejecutar la operación necesaria o creando un stream paralelo con `parallelStream()` sobre un stream.

Por último se tiene una serie de operaciones que pasan de streams a valores concretos, como un número, una cadena o una colección, denominadas colectores u operaciones terminales.

Un ejemplo de concatenación de operadores extraído de la API de Java, que concatena el filtrado y el mapeo para finalizar calculando la suma:

```
int sum = widgets.stream()
    .filter(w -> w.getColor() == RED)
    .mapToInt(w -> w.getWeight())
    .sum();
```



En cuanto a las operaciones terminales:

| Nombre                | Definición                                       | Descripción                                      |
|-----------------------|--|--|
| <code>allMatch</code> | boolean <code>allMatch(Predicate&lt;?&gt;</code> | Cierto si todos los elementos del stream cumplen |

|               |   |   |
|---------------|---|---|
|               | super T> predicate)                                 | con el predicado  |
| anyMatch      | boolean anyMatch(Predicate<? super T> predicate)    | Cierto si alguno de los elementos cumple con le predicado   |
| Count         | long count()  | Devuelve el número de elementos del stream  |
| NoneMatc<br>h | boolean noneMatch(Predicate<? super T> predicate)   | Cierto si ningún elemento cumple con el predicado   |
| reduce        | T reduce(T identity, BinaryOperator<T> accumulator) | Reduce una colección a un único objeto de tipo T aplicando una interfaz funcional de tipo función con 2 parámetros de tipo T. |
| Max           | Optional<T> max(Comparator<? super T> comparator)   | Devuelve el elemento máximo, aplicando un comparador  |
| Min           | Optional<T> min(Comparator<? super T> comparator)   | Igual que el anterior, pero obtiene el mínimo   |
| FindAny       | Optional<T> findAny()                               | Devuelve un elemento del flujo.   |
| FindFirst     | Optional<T> findFirst()                             | Devuelve el primer elelemnto del stream   |
| Collect       | <R,A> R collect(Collector<? super T,A,R> collector) | Se realiza una selección de los elementos del stream, donde R es el tipo del resultado, A es el acumulador y T es el tipo.    |

Algunos ejemplo de métodos estáticos de la clase Collections utilizados en el método Collect:

```
// Accumulate names into a List
List<String> list =
people.stream().map(Person::getName).collect(Collectors.toList());

// Accumulate names into a TreeSet
Set<String> set =
people.stream().map(Person::getName).collect(Collectors.toCollection(HashSet::new));

// Convert elements to strings and concatenate them, separated by commas
String joined = things.stream()
                        .map(Object::toString)
                        .collect(Collectors.joining(", "));

// Compute sum of salaries of employee
```

```
int total = employees.stream()
                        .collect(Collectors.summingInt(Employee::getSalary));
;
// Group employees by department
Map<Department, List<Employee>> byDept
    = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment));
// Compute sum of salaries by department
Map<Department, Integer> totalByDept
    = employees.stream()
                .collect(Collectors.groupingBy(Employee::getDepartment,
                                                Collectors.summingInt(Employee::getSalary)));
// Partition students into passing and failing
Map<Boolean, List<Student>> passingFailing =
    students.stream()
              .collect(Collectors.partitioningBy(s -> s.getGrade() >=
PASS_THRESHOLD));
```

Por último mencionar que es posible transformar los streams en array con los métodos `toArray` y su versión sobrecargada `<A> A[] toArray(IntFunction<A[]> generator)` en la que se pasa un generador de elementos, normalmente un constructor. Por ejemplo:

```
public static void main(String[] args)
{
    List<Employee> employeeList = new ArrayList<>(Arrays.asList(
        new Employee(1, "A", 100),
        new Employee(2, "B", 200),
        new Employee(3, "C", 300),
        new Employee(4, "D", 400),
        new Employee(5, "E", 500),
        new Employee(6, "F", 600)));

    Employee[] employeesArray = employeeList.stream()
        .filter(e -> e.getSalary() < 400)
        .toArray(Employee[]::new);

    System.out.println(Arrays.toString(employeesArray));
}
```

Construye un stream a partir de la lista, realiza un filtrado por salario y con el resultado crea un **nuevo array de Empleados**.

 Colectores y Stream OpenWebinars. <https://www.youtube.com/watch?v=Jt1aV6gS80Q>

 Uso práctico de Stream UPM. [https://www.youtube.com/watch?v=6iaqT58\\_PnM](https://www.youtube.com/watch?v=6iaqT58_PnM)

## 3. Practicando.

### 3.1. Colecciones en Java.

#### 3.1.1 Listas.

Crear la clase corredor, que posee un nombre, unos apellidos, un dorsal (número), el número de minutos y segundos de la carrera y la posición de la misma, puede ser necesario definir nuevos métodos e implementar interfaces.

```
public class Corredor {  
    private String nombre;  
    private String apellidos;  
    private int dorsal;  
    private int minutos;  
    private int segundos;  
    private int posicion;  
    public Corredor() {  
        this.nombre="";  
        this.apellidos="";  
        this.dorsal=-1;  
        this.minutos=-1;  
        this.segundos=-1;  
        this.posicion=-1;  
    }  
    public Corredor(String nombre, String apellidos, int dorsal, int minutos,  
int segundos, int posicion) {  
        this.nombre = nombre;  
        this.apellidos = apellidos;  
        this.dorsal = dorsal;  
        this.minutos = minutos;
```

```
this.segundos = segundos;
this.posicion = posicion;
}
public String getNombre() {
    return nombre;
}
public String getApellidos() {
    return apellidos;
}
public int getDorsal() {
    return dorsal;
}
public int getMinutos() {
    return minutos;
}
public int getSegundos() {
    return segundos;
}
public int getPosicion() {
    return posicion;
}
public void setNombre(String nombre) {
    this.nombre = nombre;
}
public void setApellidos(String apellidos) {
    this.apellidos = apellidos;
}
public void setDorsal(int dorsal) {
    this.dorsal = dorsal;
}
public void setMinutos(int minutos) {
    this.minutos = minutos;
}
public void setSegundos(int segundos) {
    this.segundos = segundos;
}
public void setPosicion(int posicion) {
    this.posicion = posicion;
}
}
```

1. Definir una variable llamada lista\_abstracta que contenga objetos de la clase Corredor.
2. Asignar a la variable anterior un nuevo arrayList de tipo String
3. Insertar al menos 5 nuevos corredores, que posean solo nombre, apellidos y dorsal (no consecutivo ni ordenado).
4. Obtener el número de elementos de la lista.
5. Definir un iterador y recorrer la lista y mostrar cada uno de los elementos.
6. Clonar el primer corredor e insertarlo, ¿Qué sucede?
7. Utilizando for (E elemento: List lista) asignar a los corredores una posición por el orden de inserción.
8. Mezclar la lista anterior y mostrar la lista desordenada.
9. Duplicar la lista, ordenarla y mostrarla. Obtener la lista ordenada.
10. En la lista original obtener una sublista con los elementos 2 y 3 (desde 0).
11. Borrar el último elemento en una única línea.
12. Crear un corredor temporal y sustituir el primero.

### 3.1.2 Queue.

Se desea crear una pequeña aplicación para la gestión de atención al ciudadano de una empresa eléctrica. Los servicios que ofrece son:

Información general.

Reclamaciones.

Altas y bajas.

Cada vez que una persona llega ha de pulsar en uno de los servicios, indicando su nombre, apellidos, número ticket (consecutivo para esa cola), y la edad. Insertándose en una de las tres colas, además se tiene preferencia por la edad, de forma que mayores de 65 años tienen preferencia con respecto al resto.

Además se tiene n mesas para atender a cada uno de los servicios.

1. Definir la clase o clases necesarias para gestionar los servicios, no olvidar implementar lo necesario para comparar, en caso de mayor de 65 años tiene preferencia, y dentro de .
2. ¿Qué tipo de cola seleccionar teniendo en cuenta que se necesita sincronización y prioridades?
3. Definir los 3 servicios con los tipos de colas indicados anteriormente e instanciar.
4. Encolar en la cola de información general 3 tickets (el número de ticket se ha de hacer de forma automática, no se ha de colocar a mano):
  - Nombre: Paco, Apellidos: Masco, n.º ticket 1, edad 45.
  - Nombre: Antonia, Apellidos: Nia, n.º ticket 2, edad 65.
  - Nombre: Luís , Apellidos: Lis, n.º ticket 3, edad 23.
5. Desencolar simulando ser una mesa al pulsar el botón de siguiente y comprobar que devuelve el ticket 2 al tener más de 65 años.
6. Encolar los siguiente tickets:
  - Nombre: Juan, Apellidos: Magan, n.º ticket 3, edad 68.
  - Nombre: Rita, Apellidos: Santa, n.º ticket 4, edad 69.
7. Desencolar y comprobar que se obtiene el ticket 4, ya que en caso de la misma prioridad la prioridad es por el ticket.
8. Vaciar la cola para el próximo día.

### 3.1.3 Sets.

Se desea crear un pequeño predictor ortográfico para un aplicación de forma que al insertar a medida que se insertar los caracteres.

1. Investigar los diferentes Set que pueden soportar las operaciones.
2. Crear el set seleccionado en el punto 1.
3. Añadir las siguientes palabras al set marcado, por ejemplo: Alicante, Almería, Asturias, Austria, Astorga, Almoradí, Algorfa y Almansa.

4. Obtener a partir de una palabra los menores alfabéticamente, por ejemplo Almansa.
5. Igual que el 4 pero incluyendo la palabra Almansa.
6. Obtener a partir de una palabra los mayores o alfabéticamente, siguiendo con el ejemplo anterior, almansa.
7. Devolver el mayor de los menores de Astorga.
8. Devolver el menor de los mayores de Astorga.

### 3.1.4 Maps.

Se desea gestionar una pequeña página web que contiene categorías y productos, se tiene una estructura para almacenar todos los productos y otra estructura para almacenar las categorías, de forma que se pueda realizar búsquedas por productos y búsquedas por categorías, este ejercicio se centra en el segundo caso.

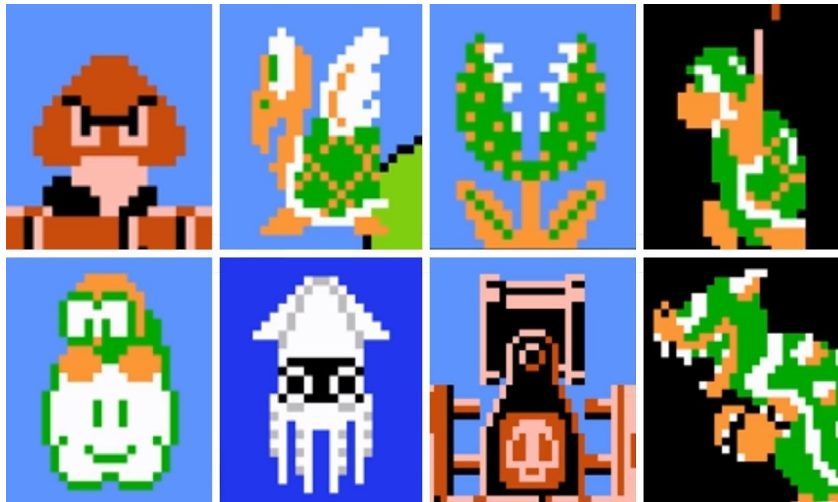
1. Pensar en que tipo de estructura de datos usar para tener categorías y dentro productos.
2. Crear la clase categoría (nombre, estado(enabled,disabled), img (en este caso una cadena) y estructura para soportar los productos) y la clase producto (nombre, precio, imagen, característica y estado), ya que tienen elementos en común usar herencia y clases abstractas.
3. Definir la estructura necesaria para gestionar las categorías.
4. Insertar la categoría: Portátiles y asignarle 2 productos:
  - Nombre: Lisa, precio 100€, características: Uno de los primeros ordenadores de Apple.
  - Nombre: IBMPC, precio 50€, características: El primer equipo para el gran público.
5. Insertar la categoría: Impresoras y asignar 2 productos:
  - Nombre: Brother Colo1234, precio 10€, características: Impresora a color.
  - Nombre: HP Laser, precio 20€, características: Impresora laser.



6. Obtener la lista de categorías de la tienda.
7. Listar los productos de portátiles.
8. Borrar las impresoras.
9. Volver a listar las categorías.

## 3.2. Interfaces funcionales.

Se tiene el juego de Super Mario Bros y se desea tener una factoria de enemigos de forma que cuando Mario llegue la primera vez a una coordenada se cree ese enemigo.



Los enemigos heredan de la clase Abstracta Enemy:

- Coordinadas.
- Path de la imagen.
- Estado (Enumerado o booleano)

Además se definen dos interfaces:

- IEnemyShoot:
  - object Shoot. //se devuelve un objeto, en la implementación real sería una bala, bola de fuego...
- IEnemyDinamic:
  - Move(Object board); //idem que el anterior, ahora no se entra en los detalles
  - Colision(Object other); //igual que anterior
  - Jump(Object board); //idem

1. Crear las estructura necesaria.

2. Definir los enemigos Mushroom, Turtle, FlyTurtle, CarnivorousPlant y Canyon heredando e implementando de las clases e interfaces anteriores (no es necesario completar los métodos).
3. Crear una clase FactoryEnemies que permita:
  - Añadir nuevas clases a crear facilitando un nombre y la interfaz funcional que lo creara.
  - Crear objetos a partir del nombre.
4. Definir una colección que almacene los enemigos creados, y usando el método remove y una interfaz funcional eliminar a aquellos cuyo estado sea muerto.
5. Idéntico al anterior pero que se hayan salido por la parte izquierda de la pantalla.
6. Se desea dotar de inteligencia al enemigo, al crearlo se ha de facilitar un algoritmo que recibe el tablero (en este caso un objeto, lo importante es usar las interfaces funcionales) y cambia las coordenadas, dispara, salta... en función del algoritmo, centrarse en la inteligencia de FlyTurtle.
7. Definir ahora una interfaz funcional que recibe 3 parámetros y devuelve otro. Usar generalización.

### 3.3. Streams.

Muchas administraciones y organismo ofrecen datos conocidos como fuentes abiertas, ya sea de forma consciente o de forma no consciente, siendo posible su análisis de forma automatizada, por ejemplo tendencias de moda a partir de Instagramo tendencias políticas en foros, twitter u otros. Los streams permiten realizar análisis de datos de fuentes abiertas en combinación con acceso a la red.

En el siguiente ejemplo se obtiene a partir de una petición HTTP el JSON los datos de los ciclos formativos de la Comunidad Valenciana:

[https://dadesobertes.gva.es/es/api/3/action/datastore\\_search?resource\\_id=b76b111f-b454-426e-9423-ecd5f0e001dc&limit=10000](https://dadesobertes.gva.es/es/api/3/action/datastore_search?resource_id=b76b111f-b454-426e-9423-ecd5f0e001dc&limit=10000)

Se utiliza una librería de Google para transformar el texto JSON a unos objetos especiales de esa misma librería JSONObject y JSONArray para arrays.

Clase ciclo:

```
public class Ciclo {  
    private int id;
```

```
private int anyo_academico;
private int cod_prov;
private String provincia;
private long cod_mun;
private String nom_mun;
private long cod_centro;
private String nom_centro;
private int cod_familia;
private String nom_familia;
private String cod_grado;
private String nom_grado;
private String cod_ciclo;
private String nom_ciclo;
private String cod_curso;
private String curso;
private int cod_turno;
private String nom_turno;
private int num_grupos;
private int num_alumnos;

public Ciclo() {
}

public int getId() {
    return id;
}

public void setId(int id) {
    this.id = id;
}

public int getAnyo_academico() {
    return anyo_academico;
}

public void setAnyo_academico(int anyo_academico) {
    this.anyo_academico = anyo_academico;
}

public int getCod_prov() {
```

```
        return cod_prov;
    }

    public void setCod_prov(int cod_prov) {
        this.cod_prov = cod_prov;
    }

    public String getProvincia() {
        return provincia;
    }

    public void setProvincia(String provincia) {
        this.provincia = provincia;
    }

    public long getCod_mun() {
        return cod_mun;
    }

    public void setCod_mun(long cod_mun) {
        this.cod_mun = cod_mun;
    }

    public String getNom_mun() {
        return nom_mun;
    }

    public void setNom_mun(String nom_mun) {
        this.nom_mun = nom_mun;
    }

    public long getCod_centro() {
        return cod_centro;
    }

    public void setCod_centro(long cod_centro) {
        this.cod_centro = cod_centro;
    }

    public String getNom_centro() {
```

```
        return nom_centro;
    }

    public void setNom_centro(String nom_centro) {
        this.nom_centro = nom_centro;
    }

    public int getCod_familia() {
        return cod_familia;
    }

    public void setCod_familia(int cod_familia) {
        this.cod_familia = cod_familia;
    }

    public String getNom_familia() {
        return nom_familia;
    }

    public void setNom_familia(String nom_familia) {
        this.nom_familia = nom_familia;
    }

    public String getCod_grado() {
        return cod_grado;
    }

    public void setCod_grado(String cod_grado) {
        this.cod_grado = cod_grado;
    }

    public String getNom_grado() {
        return nom_grado;
    }

    public void setNom_grado(String num_grado) {
        this.nom_grado = num_grado;
    }

    public String getCod_ciclo() {
```

```
        return cod_ciclo;
    }

    public void setCod_ciclo(String cod_ciclo) {
        this.cod_ciclo = cod_ciclo;
    }

    public String getNom_ciclo() {
        return nom_ciclo;
    }

    public void setNom_ciclo(String nom_ciclo) {
        this.nom_ciclo = nom_ciclo;
    }

    public String getCod_curso() {
        return cod_curso;
    }

    public void setCod_curso(String cod_curso) {
        this.cod_curso = cod_curso;
    }

    public String getCurso() {
        return curso;
    }

    public void setCurso(String curso) {
        this.curso = curso;
    }

    public int getCod_turno() {
        return cod_turno;
    }

    public void setCod_turno(int cod_turno) {
        this.cod_turno = cod_turno;
    }

    public String getNom_turno() {
```

```
        return nom_turno;
    }

    public void setNom_turno(String nom_turno) {
        this.nom_turno = nom_turno;
    }

    public int getNum_grupos() {
        return num_grupos;
    }

    public void setNum_grupos(int num_grupos) {
        this.num_grupos = num_grupos;
    }

    public int getNum_alumnos() {
        return num_alumnos;
    }

    public void setNum_alumnos(int num_alumnos) {
        this.num_alumnos = num_alumnos;
    }
}
```

Programa principal:

```
public class EjercicioStreams {
    //pasa de OBJECTJSON A OBJETO CICLO
    public static Ciclo ParseCiclo(JSONObject json){
        Ciclo ciclo= new Ciclo();
        ciclo.setId(Integer.valueOf(json.get("_id").toString()));
        ciclo.setAnyo_academico(Integer.valueOf(json.get("ANYO_ACADEMICO").toString()));
        ciclo.setCod_prov(Integer.valueOf(json.get("COD_PROV").toString()));
        ciclo.setProvincia(json.get("NOM_PROV").toString());
        ciclo.setCod_mun(Integer.valueOf(json.get("COD_MUN").toString()));
        ciclo.setNom_mun(json.get("NOM_MUN").toString());
        ciclo.setCod_centro(Integer.valueOf(json.get("COD_CENTRO").toString()));
        ciclo.setNom_centro(json.get("NOM_CENTRO").toString());
    }
}
```

```
ciclo.setCod_familia(Integer.valueOf(json.get("COD_FAMILIA").toString()));
    ciclo.setNom_familia(json.get("NOM_FAMILIA").toString());
    ciclo.setCod_grado(json.get("COD_GRADO").toString());
    ciclo.setNom_grado(json.get("NOM_GRADO").toString());
    ciclo.setCod_ciclo(json.get("COD_CICLO").toString());
    ciclo.setNom_ciclo(json.get("NOM_CICLO").toString());
    ciclo.setCod_curso(json.get("COD_CURSO").toString());
    ciclo.setCurso(json.get("CURSO").toString());
    ciclo.setCod_turno(Integer.valueOf(json.get("COD_TURNO").toString()));
    ciclo.setNom_turno(json.get("NOM_TURNO").toString());

ciclo.setNum_grupos(Integer.valueOf(json.get("NUM_GRUPOS").toString()));

ciclo.setNum_alumnos(Integer.valueOf(json.get("NUM_ALUMNOS").toString()));
    return ciclo;
}
/**
 * @param args the command line arguments
 */
public static void main(String[] args) throws MalformedURLException,
IOException, ParseException {
    String url =
    "https://dadesobertes.gva.es/es/api/3/action/datastore_search?
    resource_id=b76b111f-b454-426e-9423-ecd5f0e001dc&limit=4500";

    String texto =
    Jsoup.connect(url).ignoreContentType(true).execute().body();

    JSONParser jsonParser = new JSONParser();
    JSONObject jsonObject = (JSONObject) jsonParser.parse(texto);
    JSONObject resultado = (JSONObject) jsonObject.get("result");
    JSONArray resultados= (JSONArray) resultado.get("records");
    /*Se transforma usando una función lambada/interface funcional y el
    método map*/
    List<Ciclo> listado=resultados.stream().map( ciclo-> {
        return EjercicioStreams.ParseCiclo((JSONObject) ciclo);
    }).toList();
    /* se filtran los ciclos por localidades y se suma*/
    long t=listado.stream().filter(c-> { return
    c.getNom_mun().toLowerCase().equals("rojaes"); }).count();
    System.out.println(t);

}
```



}

## 4. Ejercicios teóricos.

### 4.1. Parte 1. Fundamentos.

1. ¿Qué problema tiene el uso de vectores para almacenar elementos?
2. ¿Qué característica poseen los TADs estructuras dinámicas?
3. Se desea gestionar el acceso al procesador por parte de un conjunto de procesos con igual prioridad. ¿Qué tipo de estructura seleccionar?
4. Indicar las operaciones de una estructura LIFO.
5. Se desea almacenar objetos de tipo Nivel\_de\_juego en una estructura de tipo lista, pila y cola. Explicar qué es y para qué se utilizan los nodos. Definir la clase nodo para almacenar un Nivel\_de\_juego.
6. Se tiene una pila vacía, dibujar el estado en que queda la cola al realizar las siguientes operaciones:
  - Push(1).
  - Push(3);
  - pop();
  - push(1);
  - push(0).
  - Pop();
7. Investigar qué es la pila del procesador y cómo funciona, ¿se comporta igual que las pilas en programación?
8. Se desea gestionar los trabajos de una impresora, justificar qué estructura básica (lista, pila o cola) utilizar.
9. Explicar el método para insertar un elemento en una cola y para sacar un elemento

10. ¿Qué puede suceder cuando existe un acceso concurrente a una cola?

11. Se tiene una cola vacía, dibujar el estado en que queda la cola al realizar las siguientes operaciones:

- Encolar(1).
- Encolar(3);
- Desencolar();
- Encolar(1);
- Encolar(0).
- Desencolar();

12. ¿Qué se puede hacer con una lista y no se puede hacer con una cola o pila?

13. Explicar cómo implementar el método `Lista.join(Lista l)` que une dos listas.

14. ¿Y si la lista anterior es ordenada? Explicar de forma general el algoritmo.

15. ¿En qué se diferencia una lista enlazada de una doblemente enlazada? ¿Qué ventajas aporta la segunda? ¿E inconvenientes?

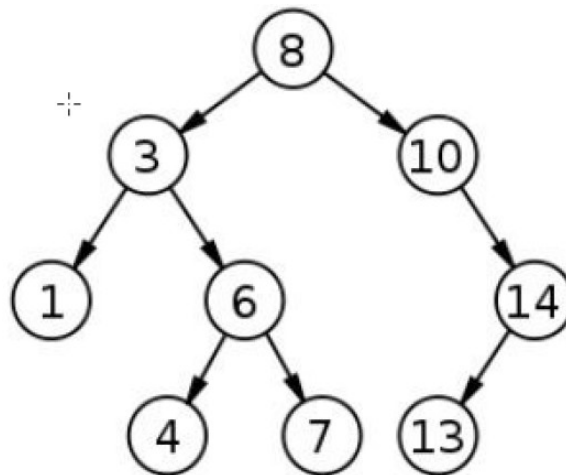
16. Explicar los casos posibles para insertar un nodo en una lista enlazada ordenada y cómo se haría la inserción.

17. ¿Qué característica tiene un nodo en una lista doblemente enlazada?

18. Explicar los casos posibles para borrar un nodo en una lista doblemente enlazada.

19. ¿Cuál es la principal ventaja de los árboles con respecto a las estructuras lineales?

20. Indicar en el siguiente árbol:



La raíz del árbol.

Profundidad del nodo 14.

Número de nodos.

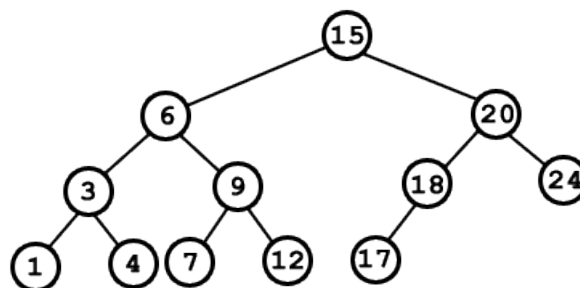
Nodos que son hojas.

Descendientes del nodo 3.

Altura del árbol.

Hijos del nodo 3.

21. A partir del siguiente árbol:



Escribir el resultado de realizar un recorrido preorden, inorden y postorden.

22. En un árbol binario de búsqueda: ¿Qué característica tiene los nodos hijos situados a la derecha de un padre? ¿Y los de la izquierda?

23. ¿Qué problema se tiene cuando un árbol binario no se encuentra balanceado?

24. Dibujar el árbol binario de búsqueda al ejecutar el siguiente código:

```
SortBinaryTree tree;
tree = new SortBinaryTree();
```

```
tree.insert(new Order("Paco", new Date(), 1));  
tree.insert(new Order("Paca", new Date(), 2));  
tree.insert(new Order("Pepe", new Date(), 3));  
tree.insert(new Order("Pepa", new Date(), 4));  
tree.insert(new Order("Pedro", new Date(), 5));  
tree.insert(new Order("Petra", new Date(), 6));  
tree.insert(new Order("Pilar", new Date(), 7));
```

25. Explicar las diferencias entre un árbol AVL y un árbol binario de búsqueda.

26. ¿En qué situaciones se usa árboles de tipo B? ¿Qué característica tiene?

27. Los árboles B+ se diferencian de los B en:\_\_\_\_\_.

28. Los árboles B\* añaden la característica de:\_\_\_\_\_.

29. Una tabla de dispersión o hash las búsquedas se realizan por una llave/key por un entero aunque es posible usar cadenas y objetos. Explicar cómo es posible. ¿Qué se tiene que garantizar para poder usar objetos como llaves en la clase que hace de llave?

30. ¿Qué sucede si se intenta insertar dos objetos que al transformarse dan la misma llave? Proponer alguna solución ante esta situación.

31. Explicar qué es un grafo.

## 4.2. Parte 2. Colecciones en Java.

1. Indicar las 3 interfaces principales de las colecciones en Java.

2. ¿Qué diferencia principal existe entre un ArrayList y un Vector?

3. ¿Qué colecciones tienen mejor tiempo al añadir elementos? Explicar la razón.

4. El TreeSet tiene complejidad  $\log_2 n$  en todas las operaciones. ¿Cuál es la razón?

5. Al crear una colección se le puede dar una clase entre los símbolos < y > ¿Qué significa esto?

6. Se tiene el siguiente código:

```
public class Vehiculo{...}  
public class Coche extends Vehiculo{...}  
public class Barco{ }  
public static void main (String [] args){  
List<Vehiculo> lista= new ArrayList<Vehiculo>();
```

```
lista.add(new Coche());  
lista.add(new Barco());  
}
```

Indicar si es correcto o no, y la razón.

7. ¿Qué son los iteradores?
8. Se tiene que procesar un conjunto de canciones almacenadas en una lista y se decide utilizar un iterador para procesar ¿Qué método usar si el procesamiento de las canciones es independiente entre ellas?
9. Escribir el código necesario para recorrer una LinkedList que contiene objetos de la clase Integer con un iterador.
10. ¿Qué dos características principales tiene la interfaz List?
11. Indicar las principales operaciones de List.
12. ¿Cómo ordenar de forma sencilla una lista independientemente de su implementación? ¿Y permutar de forma aleatoria?
13. Se desea crear un pequeño programa para la gestión de alumnos (entre otros atributos tiene en el nombre y los apellidos), los alumnos se encuentra en una lista, crear un programa que divida a los alumnos en grupos de 30 alumnos de forma alfabética. (Usar los métodos de la interfaz List).
14. ¿Qué ha de cumplir la clase alumno para poder insertarse correctamente en la lista?
15. ¿Qué dos operaciones posee una pila?
16. ¿Qué operaciones define la interfaz Queue?
17. Nos han encargado crear un programa para la atención al cliente de una empresa, el usuario entra y selecciona entre las peticiones de tipo: Reclamaciones, financiación, información general, asignándolo a una mesa y tiene que esperar su turno. Indicar qué estructura seleccionar.
18. Diseñar el diagrama de clases del ejercicio anterior.
19. Se decide modificar el programa anterior y ahora además de tipo de petición se ha de tener en cuenta la prioridad siendo esta:

- Personas mayores de 70 años.
- Mujeres embarazadas.
- Restor de personas.

Cambiar la estructura de datos seleccionadas para cumplir con los requerimientos anteriores. ¿Es suficiente con estos cambios para que funcione? En caso de que no sea así indicar qué hacer.

20. ¿Qué diferencia una Queue de una Deque?

21. Un set modela \_\_\_\_\_ y por tanto no puede tener \_\_\_\_\_

22. ¿Cuándo dos set's son iguales?

23. ¿Es posible instanciar un SortedSet?

24. Si se desea obtener elementos relacionados con uno de los almacenados en un set, por ejemplo, los menores o los mayores. ¿Qué tipo de set seleccionar?

25. Explicar qué es un Map, y el uso de key y value.

26. ¿Qué problema tiene los Maps? ¿Qué ventaja?

27. Indicar internamente que estructura implementa los siguientes Map's:

- HashMap.
- LinkdHashMap.
- TreeMap.

28. En clase se ha visto el uso de Map en las aplicaciones web con los métodos HTTP GET y POST, explicar qué son y cómo se usa el Map.

## 4.3. Parte 3. Generalización, interfaces funcionales y streams.

### 4.3.1 Generalización.

1. ¿Qué permiten los tipos genéricos?
2. ¿Qué ventajas ofrecen?
3. Los genericos son invariantes, indicar las consecuencias de esta característica.

4. Si se tiene una definición de generico del siguiente tipo, ¿Qué significa cada parámetro?

```
5. Public class Ejemplo <T,K,S,U.N>{...}
```

6. Se desea hacer una rifa, el sorteo consiste en seleccionar 5 elementos pueden ser cualquier tipo de objeto siempre y cuando implementen la interfaz Sortable. Definir un método estático que reciba una colección de elementos Sortables.
7. ¿Es posible crear nuevos objetos del tipo que se define?

#### 4.3.2 Interfaces funcionales.

1. Define una función lambda.
2. ¿Cuál es la principal característica de una interfaz funcional?
3. ¿Cómo indica Java que una interfaz es funcional?
4. Se desea tener flexibilidad para poder cambiar el algoritmo de movimiento de un juego de aviones, se define la siguiente interfaz

```
public interface IMove<T extends Element> {  
    public void move(T element);  
}
```

La interfaz Element y la clase avión:

```
public interface Element{  
    public int getX();  
    public int getY();  
    public void setX(int x);  
    public void setY(int y);  
}
```

```
public class Avion implements Element {  
    private int x;  
    private int y;  
    private IMove<Avion> algoritmo_mover;  
    public Avion() {  
        this.x=x;  
        this.y=y;  
        this.algoritmo_mover=null;  
    }  
}
```

```

    }

    public void setMove(IMove<Avion> algoritmo){
        this.algortimo_mover=algoritmo;
    }

    public void move(){
        this.algortimo_mover.move(this);
    }

    public int getX() {
        return x;
    }

    public void setX(int x) {
        this.x = x;
    }

    public int getY() {
        return y;
    }

    public void setY(int y) {
        this.y = y;
    }
}

```

5. Crear un objeto de clase avion asignar un algoritmo de movimiento en horizontal y a continuación cambiar el algoritmo para que solo se pueda mover en vertical. El primero con clases anónimas y el segundo con función anónima.
6. Pensar en el código necesario para poder hacer el ejercicio anterior sin funciones lambda.
7. Crear una interfaz funcional que reciba 3 parámetros, el primero que sea clonable, el segundo Iterable y el tercero Cloneable y devuelva un objeto que implemente la interfaz Idrawable vista en la práctica de la serpiente.
8. Completar la siguiente tabla:

| Interfaz  | Parámetros | Devuelve | Ejecutar | Uso |
|-----------|------------|----------|----------|-----|
| Consumer  |            |          |          |     |
| Supplier  |            |          |          |     |
| Function  |            |          |          |     |
| Predicate |            |          |          |     |



|               |  |  |  |  |
|---------------|--|--|--|--|
| UnaryOperator |  |  |  |  |
|---------------|--|--|--|--|

9. Se tiene una lista de objetos de clase Cliente de un Banco con los atributos nombre, edad, saldo, se desea añadir un método a la clase cliente de forma que se le pueda aplicar promociones, estas promociones cambian cada mes, por ejemplo, si es mayor de 65 se le ofrece un conjunto de sartenes (“se le envía un mensaje”), si es su santo un mensaje felicitandolo. Pensar como implementarlo teniendo en cuenta que se tiene un método estático en la clase Banco para enviar un mensaje de Telegram.
10. En el ejercicio anterior existe una promoción que posee 2 condiciones, mayor de 18 años y saldo de al menos 1000€, ¿Cómo se implementaría?
11. En un pequeño juego se tiene diferentes enemigos almacenados en una colección, en cada iteración del juego se ha de comprobar si el enemigo se encuentra muerto e.`IsDead()` o si ha salido fuera de la pantalla por la izquierda e.`getX()`. Pensar en qué método de colecciones permite borrar.
12. En el juego anterior se quiere que el jugador principal pueda tener diferentes tipos de armas, por ejemplo un hacha, una pistola, o una espada. Estas armas han de poder cambiar en el juego además de poder añadirse nuevas armas a lo largo del juego, pensar como implementar una Factorya de armas con interfaces funcionales.

### 4.3.3 Streams.

1. ¿De qué fuentes se pueden obtener los Streams?
2. ¿Cómo obtener un stream de la interfaz List? ¿Qué dos métodos de collection devuelve un tipo de stream?
3. ¿Qué tipo de parámetros reciben los streams?
4. Se tiene una lista de más de 100000 personas y se desea limitar a solo 1000, qué operador se ha de utilizar.
5. Se tiene un conjunto de clientes y se desea obtener solo aquellos que tenga un saldo mayor de 10000 euros. ¿Qué operador aplicar?

6. El ejercicio anterior desea obtener los valores ordenados por importe.
7. Al aplicar una operación no terminar a un stream lo que se obtiene es \_\_\_\_\_
8. ¿Cuántas veces se pueden “anidar” llamadas a operadores?
9. Explicar que se entiende por operaciones terminales.
10. ¿Cómo obtener el máximo de un stream? ¿Qué tipo de interfaz funcional se le pasa?
11. ¿Cómo contar el número de elementos de un stream?
12. Se tienen los clientes anteriores y se desea agrupar por localidad ¿Cómo hacerlo? ¿Qué estructura devuelve la operación?
13. ¿Y para obtener el saldo medio de los clientes con más de 10000€ por localidad?

## 5. Soluciones practicando.

### 5.1. Colecciones Java

#### 5.1.1 Listas.

```
List<Corredor> lista_abstracta;  
//punto 2  
lista_abstracta = new ArrayList<Corredor>();  
//punto 3  
Corredor tempo;  
tempo = new Corredor();  
tempo.setNombre("Paco");  
tempo.setApellidos("Gomez");  
tempo.setDorsal(5);  
lista_abstracta.add(tempo);  
  
tempo = new Corredor();  
tempo.setNombre("Luisa");  
tempo.setApellidos("Tisa");  
tempo.setDorsal(2);  
lista_abstracta.add(tempo);
```

```
tempo = new Corredor();
tempo.setNombre("Antonio");
tempo.setApellidos("Tonio");
tempo.setDorsal(34);
lista_abstrakta.add(tempo);

tempo = new Corredor();
tempo.setNombre("Marina");
tempo.setApellidos("Antunez");
tempo.setDorsal(4);
lista_abstrakta.add(tempo);

tempo = new Corredor();
tempo.setNombre("Juan");
tempo.setApellidos("Final");
tempo.setDorsal(1);
lista_abstrakta.add(tempo);
//punto 4
System.out.println("PUNTO 4: El número de elementos de la lista es " +
lista_abstrakta.size());
//punto 5
System.out.println("PUNTO 5");
Iterator it = lista_abstrakta.iterator();
while (it.hasNext()) {
    System.out.println(it.next());
}

//punto 6
Corredor clonado = lista_abstrakta.get(0);
lista_abstrakta.add(clonado);
//punto 7
int posicion = 0;
for (Corredor c : lista_abstrakta) {
    c.setPosicion(posicion);
    posicion++;
}
//punto 8
System.out.println("PUNTO 8");
Collections.shuffle(lista_abstrakta);
it = lista_abstrakta.iterator();
```

```
while (it.hasNext()) {
    System.out.println(it.next());

}
//punto 9
System.out.println("PUNTO 9");
List clonada = new ArrayList(lista_abstracta);
Collections.sort(clonada);
it = clonada.iterator();
while (it.hasNext()) {
    System.out.println(it.next());

}
//punto 10
List sublista = lista_abstracta.subList(2, 3);
//punto 11
lista_abstracta.remove(lista_abstracta.size() - 1);
//punto 12
lista_abstracta.set(0, new Corredor("Antono", " tempo", 67, -1, -1, -
1));
}
```

### 5.1.2 Queue.

```
public static void main(String[] args) {
    //punto 3
    PriorityBlockingQueue<Ticket> info_general,altas,reclamaciones;
    info_general= new PriorityBlockingQueue<>();
    altas= new PriorityBlockingQueue<>();
    reclamaciones= new PriorityBlockingQueue<>();

    //punto 4
    info_general.put(new Ticket("Paco","Masco",info_general.size()+1,45));
    info_general.put(new Ticket("Antonia","Nia",info_general.size()+1,65));
    info_general.put(new Ticket("Luís","Lis",info_general.size()+1,23));
    //punto 5
    System.out.println(info_general.poll());
    //punto 6
    info_general.put(new Ticket("Juan","Magan",info_general.size()+1,68));
    info_general.put(new Ticket("Rita","Santa",info_general.size()+1,69));
    //punto 7
```

```
System.out.println(info_general.poll());  
//punto 8  
info_general.clear();  
  
}
```

### 5.1.3 Set.

```
public class Sets {  
    public static void main(String[] args) {  
        //PUNTO 2  
        TreeSet<String> palabras = new TreeSet<>();  
        //PUNTO 3  
        palabras.add("Alicante");  
        palabras.add("Almeria");  
        palabras.add("Asturias");  
        palabras.add("Austria");  
        palabras.add("Astorga");  
        palabras.add("Almoradí");  
        palabras.add("Algorfa");  
        palabras.add("Almansa");  
        //PUNTO 4  
        System.out.println("Punto 4.");  
        Iterator<String> it=palabras.headSet("Almansa", false).iterator();  
        while(it.hasNext()){  
            System.out.println(it.next());  
        }  
        System.out.println("Punto 6.");  
        //PUNTO 6  
        it=palabras.tailSet("Almansa").iterator();  
        while(it.hasNext()){  
            System.out.println(it.next());  
        }  
    }  
}
```

### 5.1.4 Maps.

Punto 2. Definir las clases Categoría y Producto que hereda de la clase Abstracta Element.

Clase Element:

```
public abstract class Element implements Cloneable {  
    public enum ElementState{  
        ENABLE,  
        DISABLED  
    }  
    String name;  
    String img;  
    private boolean state;  
    public Element() {  
    }  
    public Element(String name) {  
        this.name=name;  
    }  
    public Element (String name,boolean state){  
        this.name=name;  
        this.state=state;  
    }  
    public Element (String name,boolean state,String path){  
        this.name=name;  
        this.state=state;  
        this.img=path;  
    }  
  
    public String getName() {  
        return name;  
    }  
  
    public void setName(String name) {  
        this.name = name;  
    }  
  
    public String getImg() {  
        return img;  
    }  
  
    public void setImg(String img) {  
        this.img = img;  
    }  
}
```

```
/**
 * @return the state
 */
public boolean isState() {
    return state;
}

/**
 * @param state the state to set
 */
public void setState(boolean state) {
    this.state = state;
}

public String toString(){
    return "Nombre:"+this.name+" "+" estado:"+this.state+ "
Imagen:"+this.img+"";
}

}
```

Clase Producto:

```
public class Producto extends Element {

    private float price;
    private String características;

    public Producto() {
        super();
        this.setName("");
    }

    public Producto(String name, float price) {
        super(name);
        this.price = price;
    }

    public Producto(String name, boolean state, float price) {
        super(name, state);
        this.price = price;
    }

    public Producto(String name, boolean state, String path, float price) {
```

```
        super(name, state, path);
        this.price = price;
    }

    public Producto(String name, boolean state, String path, float price,
String características) {
        super(name, state, path);
        this.price = price;
        this.caracteristicas = características;
    }

    public float getPrice() {
        return price;
    }

    @Override
    public String toString() {
        return "Nombre:" + this.name + " Precio:" + this.price + "
Características:" + this.caracteristicas;
    }

    public void setPrice(float price) {
        this.price = price;
    }

    @Override
    public Object clone() {
        Producto p = new Producto(this.name, this.isState(), this.price);
        //la imagen es copia superficial
        p.img = this.img;

        return p;
    }
}
```

Clase Categoría:

```
public class Categoria extends Element {

    List<Producto> productos;

    public Categoria() {
        super();
        this.setName("");
        this.productos = new LinkedList<>();
    }

    public Categoria(String nombre) {
```



```
super(nombre);
this.productos = new LinkedList<>();
}

public Categoria(String name, boolean state) {
    super(name, state);
    this.productos = new LinkedList<>();
}

public Categoria(String name, boolean state, String path) {
    super(name, state, path);
    this.productos = new LinkedList<>();
}

public void addProducto(Producto p) {
    this.productos.add(p);
}

public List<Producto> getAllProductos() {
    return this.productos;
}

public String toString() {
    StringBuilder sb= new StringBuilder();
    sb.append("Nombre:"+this.name+"\n");
    Iterator it=this.productos.iterator();
    while(it.hasNext())
        sb.append(it.next()+"\n");
    return sb.toString();
}

@Override
public Object clone() {
    Categoria c = new Categoria(this.name, this.isState());
    c.setImg(this.img);
    for (Producto p : this.productos) {
        c.addProducto((Producto) p.clone());
    }
    return c;
}
```

```
}
```

Resto de puntos:

```
public static void main(String[] args) {  
    //PUNTO 3  
    HashMap<String, Categoria> categorias;  
    categorias = new HashMap<>();  
    //PUNTO 4  
    categorias.put("portatiles", new Categoria("Portatiles", true,  
"portatil.png"));  
    Categoria c = categorias.get("portatiles");  
    c.addProducto(new Producto("Lisa", true, "e1.png", 100, "Uno de los  
primeros ordenadores de Apple."));  
    c.addProducto(new Producto("IBM_PC", true, "e2.png", 50, "El primer  
equipo para el gran público."));  
    //PUNTO 5  
    categorias.put("impresoras", new Categoria("Impresoras", true,  
"impresora.png"));  
    c = categorias.get("impresoras");  
    c.addProducto(new Producto("Brother Colo1234", true, "i1.png", 10, "  
Impresora a color."));  
    c.addProducto(new Producto("HP Laser", true, "i2.png", 20, "Impresora  
laser."));  
    //PUNTO 6  
    System.out.println("Punto 6.");  
    Iterator it = categorias.keySet().iterator();  
    while (it.hasNext()) {  
        System.out.println(it.next());  
    }  
    //PUNTO 7.  
    System.out.println("Punto 7.");  
    it = categorias.get("portatiles").getAllProductos().iterator();  
    while (it.hasNext()) {  
        System.out.println(it.next());  
    }  
    //PUNTO 8  
    categorias.remove("portatiles");  
    //PUNTO 9  
    System.out.println("Punto 9.");  
    it = categorias.keySet().iterator();  
    while (it.hasNext()) {  
        System.out.println(it.next());  
    }  
}
```

}

## 5.2. Interfaces funcionales.

1. Crear las estructura necesaria.

Clase Coordinate:

```
public class Coordinate {  
    private int x;  
    private int y;  
    public Coordinate() {  
        this.x=-1;  
        this.y=-1;  
    }  
    public Coordinate(int x,int y){  
        this.x=x;  
        this.y=y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setY(int y) {  
        this.y = y;  
    }  
}
```

Clase Enemy:

```
public class Enemy {  
    public enum State{  
        DEAD,  
        ALIVE
```

```
}  
  
private Coordinate position;  
private String img_path;  
private State state;  
public Enemy() {  
}  
  
public Coordinate getPosition() {  
    return position;  
}  
  
public void setPosition(Coordinate position) {  
    this.position = position;  
}  
  
public String getImg_path() {  
    return img_path;  
}  
  
public void setImg_path(String img_path) {  
    this.img_path = img_path;  
}  
  
public State getState() {  
    return state;  
}  
  
public void setState(State state) {  
    this.state = state;  
}  
  
}
```

Interface IenemyShoot:

```
public interface IEnemyShoot {  
    public Object Shoot();  
}
```

Interface IenemyDinamic:

```
public interface IEnemyDinamic {  
    public void Move(Object board);  
}
```

```
public void Collision(Object other);  
public void Jump (Object board);  
}
```

2. Definir los enemigos Mushroom, Turtle, FlyTurtle, CarnivorousPlant y Canyon heredando e implementando de las clases e interfaces anteriores (no es necesario completar los métodos).

Algunas clases:

```
public class Mushroom extends Enemy implements IEnemyDinamic{  
    public Mushroom(){  
        super();  
        this.setImg_path("mushroom.png");  
    }  
    @Override  
    public void Move(Object board) {  
    }  
  
    @Override  
    public void Collision(Object other) {  
    }  
  
    @Override  
    public void Jump(Object board) {  
    }  
}
```

```
public class Canyon extends Enemy implements IEnemyShoot{  
    public Canyon(){  
        super();  
        this.setImg_path("carnivorousPlant.png");  
    }  
  
    @Override  
    public Object Shoot() {  
        throw new UnsupportedOperationException("Not supported yet."); //To  
change body of generated methods, choose Tools | Templates.  
    }  
}
```

```
}
```

### 3. Crear una clase FactoryEnemies que permita:

- Añadir nuevas clases a crear facilitando un nombre y la interfaz funcional que lo creara.
- Crear objetos a partir del nombre.

Se tiene un map en el que se guardan por nombre las interfaces funcionales de tipo Supplier, y cuando se llama a create con el nombre devuelve un objeto del nombre de la clase:

```
public class FactoryEnemies {  
    private static HashMap<String, Supplier<Enemy>> enemigos;  
    private static ArrayList<String> names;  
    static {  
        enemigos = new HashMap<>();  
        names= new ArrayList<>();  
    }  
    public static void addEnemy(String name,Supplier< Enemy> s){  
        FactoryEnemies.enemigos.put(name, s);  
        FactoryEnemies.names.add(name);  
    }  
    public static Enemy get(Supplier<? extends Enemy> s) {  
        return s.get();  
    }  
    public static List<String> getKeyNames(){  
        return FactoryEnemies.names;  
        // return new ArrayList<String>(FactoryEnemies.enemigos.keySet());  
    }  
    public static Enemy create(String nombre) {  
        if (FactoryEnemies.enemigos.get(nombre) != null) {  
            return FactoryEnemies.enemigos.get(nombre).get();  
        } else {  
            return null;  
        }  
    }  
}
```

Al crear un main se configura la factoria y se añaden los supplier que serán los constructores de las diferentes clases, recordar que un supplier no recibe ningún parámetro y devuelve un objeto.

```
public static void main(String[] args) {  
    FactoryEnemies.addEnemy("Canyon", Canyon::new);  
    FactoryEnemies.addEnemy("CarnivorousPlant", CarnivorousPlant::new);  
    FactoryEnemies.addEnemy("FlyTurtle", FlyTurtle::new);  
    FactoryEnemies.addEnemy("Mushroom", Mushroom::new);  
    FactoryEnemies.addEnemy("Turtle", Turtle::new);  
  
    //se crean dos enemigos a partir de la factoria  
    Enemy e= FactoryEnemies.create("Canyon");  
    System.out.println(e.getClass().getName());  
  
    e= FactoryEnemies.create("Turtle");  
    System.out.println(e.getClass().getName());  
    // TODO code application logic here  
}
```

La salida del código anterior es:

```
interfacesfuncionales.Canyon  
interfacesfuncionales.Turtle
```

4. Definir una colección que almacene los enemigos creados, y usando el método removelf y una interfaz funcional eliminar a aquellos cuyo estado sea muerto.

```
LinkedList<Enemy> lista = new LinkedList<>();  
    //se crean dos enemigos a partir de la factoria  
    for (int i = 0; i < 1000; i++) {  
        lista.add(FactoryEnemies.create(  
            clases[(int) (Math.random() * (clases.length - 1))])  
        );  
    }  
    System.out.println("Antes de borrar tiene un tamaño de : " +  
lista.size());  
    lista.get(100).setState(Enemy.State.DEAD);  
    lista.get(300).setState(Enemy.State.DEAD);  
    lista.get(500).setState(Enemy.State.DEAD);  
    //se le pasa un predicado que se evalua para cada elemento de la lista
```

```
lista.removeIf(e -> e.getState() == Enemy.State.DEAD);  
System.out.println("Al eliminar tiene un tamaño de:" + lista.size());
```

5. Idéntico al anterior pero que se hayan salido por la parte izquierda de la pantalla.

```
System.out.println("Al eliminar tiene un tamaño de:" + lista.size());
```

6. Se desea dotar de inteligencia al enemigo, al crearlo se ha de facilitar un algoritmo que recibe el tablero (en este caso un objeto, lo importante es usar las interfaces funcionales) y cambia las coordenadas, dispara, salta... en función del algoritmo, centrarse en la inteligencia de FlyTurtle.

Es de tipo recibe un parámetro y no devuelve nada es decir un consumer **Consumer <T>**, se prepara la clase FlyTurtle:

```
public class FlyTurtle extends Enemy implements IEnemyDinamic{  
    BiConsumer<FlyTurtle, Object> IA;  
    public FlyTurtle(){  
        super();  
        this.setImg_path("flyturtle.png");  
    }  
    public void setIA(BiConsumer<FlyTurtle, Object> IA) {  
        this.IA=IA;  
    }  
    @Override  
    public void Move(Object board) {  
        this.IA.accept(this, board);  
    }  
    @Override  
    public void Collision(Object other) {  
    }  
    @Override  
    public void Jump(Object board) {  
    }  
}
```

Como se ve no se tiene ninguna lógica en la tortuga voladora, ahora desde otro punto del código se le puede pasar en cualquier momento o cambiar el algoritmo de movimiento y al llamar a Move(board) se ejecutará el algoritmo asignado:



```
//se crean dos enemigos a partir de la factoria
for (int i = 0; i < 1000; i++) {
    lista.add(FactoryEnemies.create(
        clases[(int) (Math.random() * (clases.length - 1))]
    ));
    if (lista.getLast() instanceof FlyTurtle) {
        //asignar los algoritmos de forma aleatoria
        if (Math.random() > 0.5f) {
            ((FlyTurtle) lista.getLast()).setIA((a, b) -> {
                System.out.println("Soy el algoritmo 1");
            });
        } else {
            ((FlyTurtle) lista.getLast()).setIA((a, b) -> {
                System.out.println("Soy el algoritmo 2");
            });
        }
    }
}
```

Por último se recorre la lista de enemigos comprobando los que implementan el enemigo dinámico llamando a Move(board) y en el caso de tortugas voladoras se ejecutará el algoritmo 1 o 2 dependiendo del que se le asigne

```
lista.forEach(enemy -> {
    if (enemy instanceof IEnemyDinamico) //en la realida se tendría que
    pasar el tablero, no null
    {
        ((IEnemyDinamico) enemy).Move(null);
    }
});
```

Observar que existe un método llamado Foreach en las colecciones que recibe un Consumidor.

Una porción de la salida:

```
Soy el algoritmo 2
Soy el algoritmo 2
Soy el algoritmo 1
Soy el algoritmo 2
Soy el algoritmo 1
Soy el algoritmo 2
```

7. Definir ahora una interfaz funcional que recibe 3 parámetros y devuelve otro. Usar generalización.