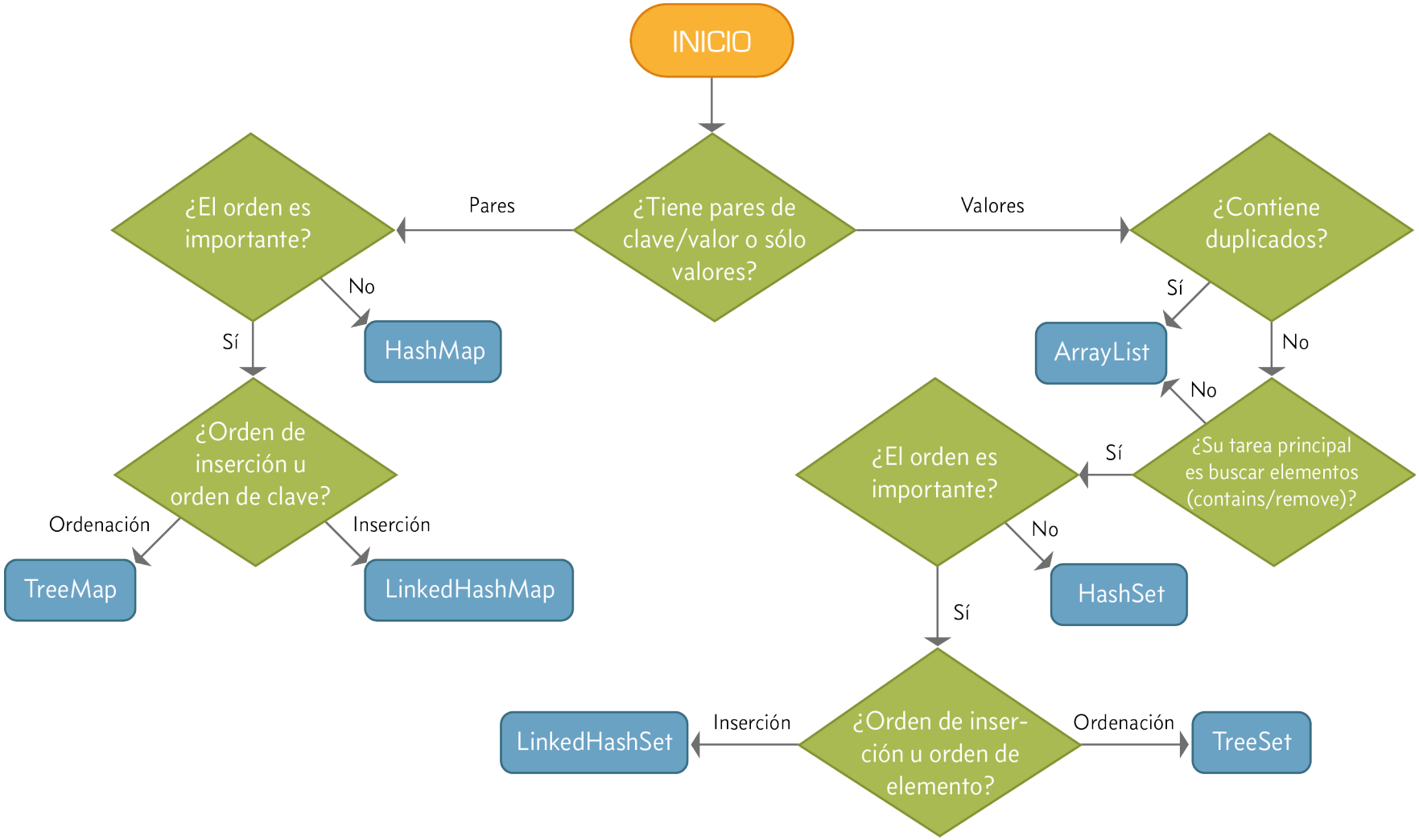
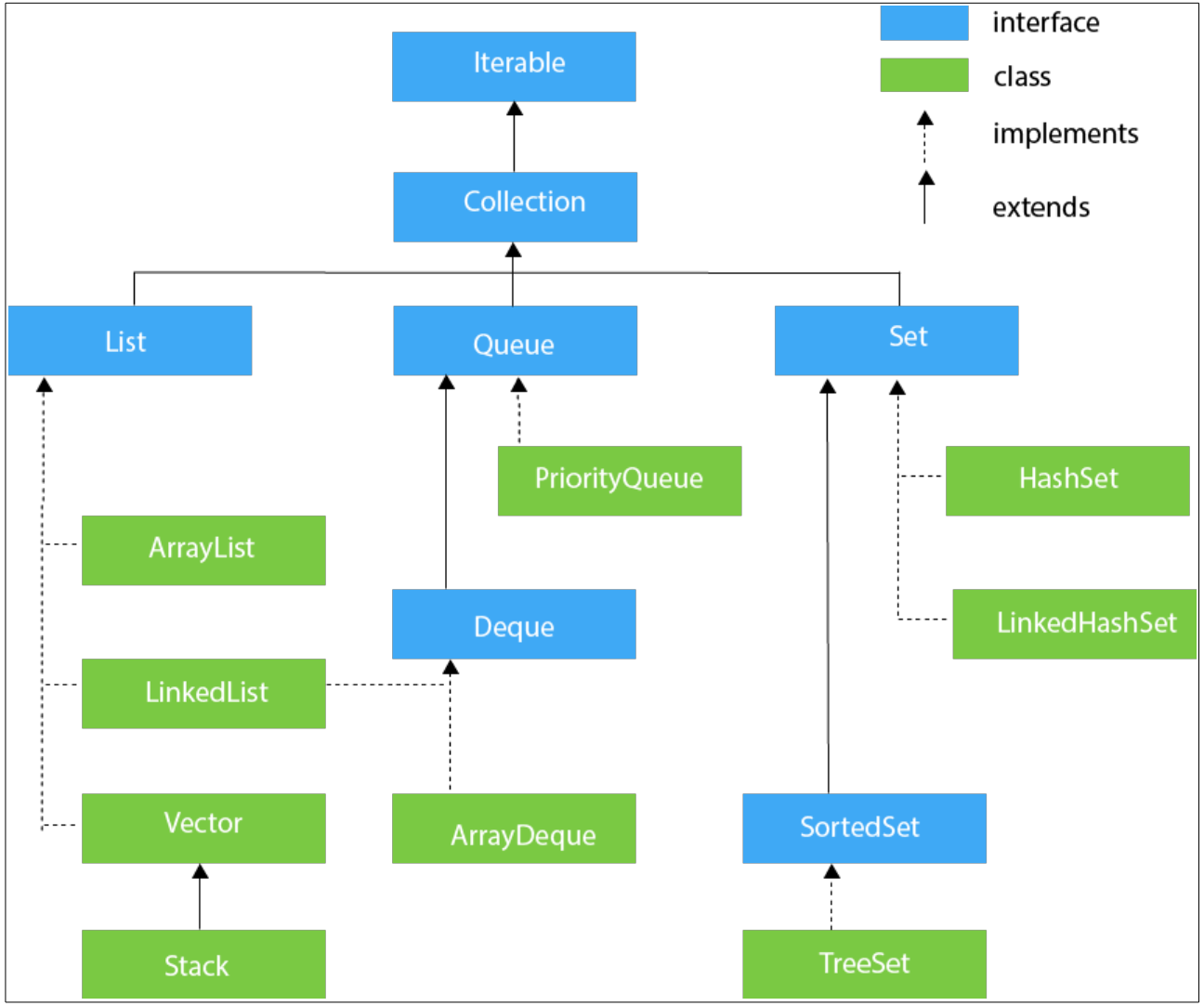
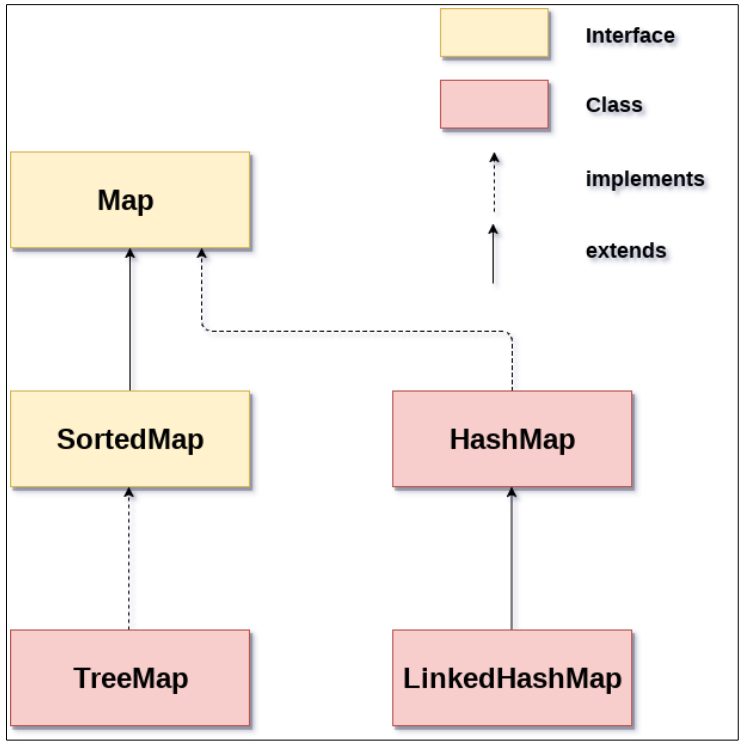
Diagrama de decisión para uso de colecciones Java



Jerarquía de `Collection` y `Map` en Java





2. Colecciones

Definiremos una colección como un objeto que representa un grupo sin tamaño determinado de elementos.

Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos. En Java, se emplea la interfaz genérica *Collection* para este propósito. Gracias a esta interfaz, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección, etc.

Para poder emplear las clases que implementan las colecciones es necesario importar el paquete que las contiene al comienzo del programa, todas están en *java.util*.

Partiendo de la interfaz *Collection* se extienden otra serie de interfaces genéricas, que aportan distintas funcionalidades sobre la interfaz anterior.

2.1. Operaciones básicas collection

add(Object o)

Añade un elemento, devuelve true si la operación finaliza correctamente.

remove(Object o)

Elimina un determinado elemento (objeto) de la colección, devolviendo true si dicho elemento estaba contenido en la colección, y false en caso contrario.

size()

Obtiene la cantidad de elementos que esta colección almacena, lo devuelve como un entero.

contains(Object o)

Pregunta si el elemento t ya está dentro de la colección. Devuelve un boolean.

isEmpty()

Indica si la colección está vacía (no tiene ningún elemento).

clear()

Elimina todos los elementos de la colección.

iterator()

Obtiene un "iterador" que permite recorrer la colección visitando cada elemento una vez.

Recorrer una collection

Las anteriores operaciones son métodos ya implementados, aunque recorrer no es un método, mostramos un código que nos permitiría el recorrido

```
for(Object o : col)
                    System.out.println(o);
```

Collection también permite interactuar con subconjuntos de la colección:

c1.containsAll(Collection c2)

Devuelve true si **c2** es un subconjunto de la colección

```
arrList1.containsAll(arrList2)      // True si arrList2 está en arrList1
```

c1.addAll(Collection c2)

Añade la colección **c2** a la colección **c1**, devuelve true si se realiza correctamente

```
boolean b = arrList1.addAll(arLlist2);
```

c1.removeAll(Collection c2)

Elimina el subconjunto **c2** de la colección **c1**

```
arrList1.removeAll(arrList2)
```

c1.retainAll(Collection c2)

Elimina de la colección **c1** los elementos que no estén en el subconjunto **c2**

```
arrList1.retainAll(arrList2)        // True si arrList2 está en arrList1
```

2.2. Interfaz Set

La interfaz Set define una colección que no puede contener elementos duplicados. Esta interfaz contiene, únicamente, los métodos heredados de Collection añadiendo la restricción de que los elementos duplicados están prohibidos.

Es importante destacar que, para comprobar si los elementos son elementos duplicados o no lo son, es necesario que dichos elementos tengan implementada, de forma correcta, los métodos **equals** y **hashCode**. Para comprobar si dos *Set* son iguales, se comprobarán si todos los elementos que los componen son iguales sin importar en el orden que ocupen dichos elementos.

La interfaz *Set* puede implementarse de varias formas:

HashSet: esta implementación almacena los elementos en una tabla hash (almacena un conjunto de pares "(clave, valor)"). Es la implementación con mejor rendimiento de todas, pero no garantiza ningún orden a la hora de realizar iteraciones.

```
HashSet <String> setNombres = new HashSet<>();
HashSet personas = new HashSet();
```

TreeSet: esta implementación almacena los elementos ordenándolos en función de sus valores. Es bastante más lento (*log(N)*) que **HashSet**. Los elementos almacenados deben implementar la interfaz **Comparable**.

LinkedHashSet: esta implementación almacena los elementos en función del orden de inserción. Es un poco más costosa que **HashSet**.

Ninguna de estas implementaciones es sincronizada, es decir, no se garantiza el estado del Set si dos o más hilos acceden de forma concurrente al mismo. Esto se puede solucionar empleando una serie de métodos que actúan de wrapper (envoltura)

```
Set set = Collections.synchronizedSet(new HashSet());
```

Ejemplo

```
import java.util.HashSet;
```

importamos el paquete HashSet

```
HashSet personas = new HashSet();
personas.add("Manolo Pérez");
personas.add("Juan López");
personas.add("Ana Gómez");

for(Object o : personas)
    System.out.println(o);
}
```

Instanciamos el objeto personas.

Añadimos elementos a la colección

Recorremos y mostramos los elementos

3. Colecciones

3.3. Interfaz List

Representa un conjunto "*lista* "de elementos ordenados, que pueden estar duplicados y que permite acceder a los elementos a partir de la posición que ocupan. Hereda los métodos de *Collection.*

3.3.1. ArrayList

Amacena los elementos en un array dinámico que va variando su tamaño en función del número de elementos a contener. Es la implementación *list* más eficiente en la mayoría de los casos. Permite valores *null*.

add(int posición, E elemento)

Añade un elemento en la posición indicada, devuelve true si la colección se ha modificado

get(int posicion)

Devuelve el elemento de la *posición* indicada. Si se pasa de rango, devuelve la excepción *IndexOutOfBoundsException*.

indexOf(Object o)

Devuelve la posición de la primera del elemento indicado, o -1 si no existe.

lastIndexOf(Object o)

Devuelve la posición de la última ocurrencia del elemento indicado, o -1 si no existe.

remove(int posicion)

Elimina el elemento de la *posición* y lo devuelve.

set(int posicion, E elemento)

sustituye el elemento situado en *posición* por el nuevo elemento, la operación devuelve el elemento que ocupaba la posición.

Collection.sort(ArrayList arrList). (método estático)

Ordena el ArrayList que recibe como parámetro

```java
import java.util.ArrayList;
import java.util.Collections;

class Main {
    public static void main(String[] args) {
        ArrayList<String> students = new ArrayList<>();

        students.add("Lindsay");
        students.add("Peter");
        students.add("Lucas");
        students.add("Athena");

        System.out.println("Student list: " + students);

        Collections.sort(students);
        System.out.println("Sorted student list: " +
students);
    }
}
```

Iterator<String> iter = list.iterator();

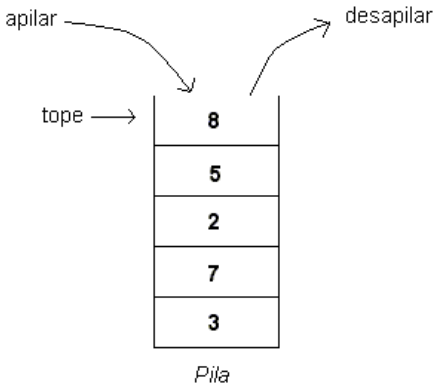
Devuelve un iterador que permite recorrer el arrayList en ambos sentidos y modificar la lista durante el recorrido. Hay que importar *java.util.Iterator.*

```
Iterator it = arrList1.iterator();
while(it.hasNext())
 System.out.println(it.next());

while(it.hasPrevious())
 System.out.println(it.previous());
```

3.3.2. Stack

Implementa la interface *List*, aunque en realidad representa al TAD (Tipo Abstracto de Datos) **pila** en el que se utiliza **LIFO** como método de acceso a los datos, siempre se accede por la parte superior, Last In - First Out, o último en entrar - primero en salir. Sus accesos son sincronizados, lo cual ralentiza las operaciones. Hay que importar *java.util.Stack.*



push (E item)

Introduce un elemento en la pila.

peek ()

Consulta el primer elemento de la cima de la pila

pop ()

Recupera un elemento de la pila y lo elimina.

search (Object o)

Busca un determinado elemento dentro de la pila y devuelve su posición dentro de ella.

empty ()

Comprueba si la pila está vacía.

Crear objeto de la clase Stack que almacenará objetos de la clase String:

```
Stack <String> pila = new Stack<String>();
```

Agregar elemento a la pila:

```
pila.push("ana");
```

Cambiar el contenido o valor de un elemento:

```
pila.set(3, "Manuel");
```

Obtener un elemento del Stack:

```
String nombre = pila.get(3);
```

Conocer la cantidad de elementos que hay en la pila:

```
int elementos = pila.size();
```

Extraer un elemento de la pila:

```
String nombre = pila.pop();
```

Consultar el primer elemento de la pila sin extraerlo (eliminarlo):

```
String nombre = pila.peek();
```

Recorrer la pila:

```
while (!pila.isEmpty())
        System.out.println(pila.pop());
```

```
for (String elemento : pila)
        System.out.println(elemento);
```

Borrar la pila:

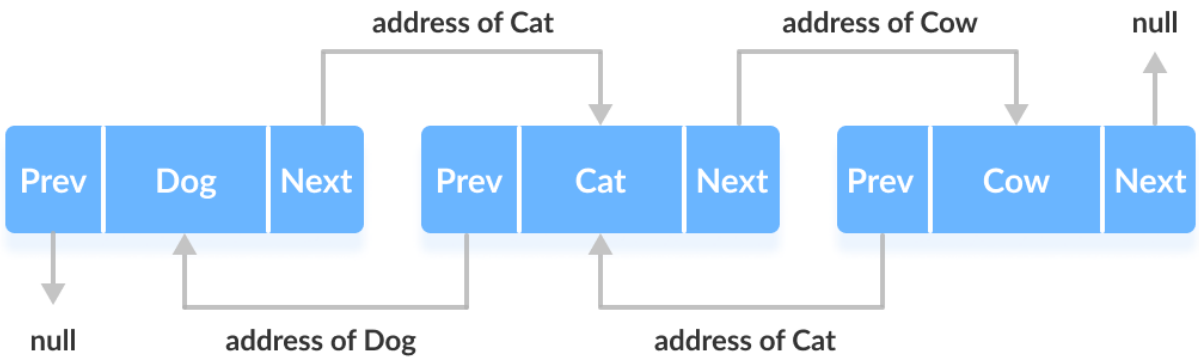
```
pila.clear();
```

Su uso es ineficiente en comparación con *LinkedList* o colecciones con el intefaz *Deque*.

Sólo aconsejable en implementaciones multihilo que necesiten accesos sincronizados.

3.3.3. LinkedList

Es una colección en la que se implementan los interfaces *List* y *Deque*. Se compone de una lista doblemente enlazada en la que cada elemento almacena una referencia a los elementos siguiente y anterior.



LinkedList Implementation in Java

- El almacenamiento de sus elementos en memoria **no** es contiguo.
- Permite almacenar valores null.
- Alto coste al acceder a un elemento aleatorio, ya que tiene que recorrer todos los anteriores o posteriores hasta posicionarse en él.
- Una vez posicionados, el coste de añadir es menor, ya que solo se realiza un cambio de referencias.
- Se trata de la colección ideal para implementar los TAD *pila* y *cola*.

Usando LinkedList

Importar clase LinkedList:

```
import java.util.LinkedList;
```

Crear objeto de la clase LinkedList:

```
LinkedList <String> pista = new LinkedList <String>();
```

Añadir nodos, puede indicarse la posición donde se tiene que insertar: *add()*

```
lista.add("Pedro");
lista.add(1, "María");
```

Añadir nodos en la primera o última posición: *addFirst(), addLast()*

```
lista.addFirst("Pedro");
lista.addLast("María");
```

Eliminar un nodo de la lista, ya sea pasando cierta información del nodo o su posición: *remove()*

```
lista.remove("Pedro");
lista.remove(1);
```

Cantidad de nodos: *size()*

```
lista.size();
```

Buscar un valor: *contains()*

```
lista.contains("Pedro");
```

Recuperar un dato de un nodo, sin eliminarlo: *get()*

```
lista.get(1);
```

Reescribir un elemento determinado: *set()*

```
lista.set(1, "Eduardo");
```

Eliminar todos los nodos de la lista: *clear()*

```
lista.clear();
```

Ver si la lista se encuentra vacía: *isEmpty()*

```
lista.isEmpty();
```

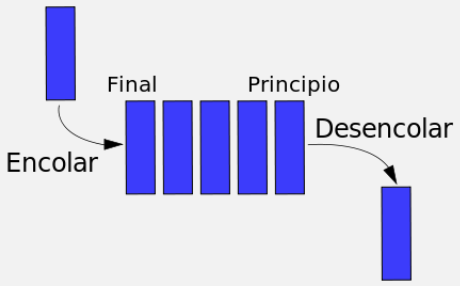
3.3.4. Cola

Una cola es un *tipo abstracto de datos* que permite almacenar y recuperar información siguiendo un esquema FIFO (First Input, First Output). Toda cola debe contar, al menos, con las operaciones *encolar* (add) y *desencolar* (remove).

Se llama *frente de la cola* al elemento que más tiempo lleva almacenado.

```
package colas;
import java.util.LinkedList;

public class Colas {
   private final LinkedList cola;
   public Colas(){
      this.cola = new LinkedList();
   }
   public  void encolar(Object elemento){
      cola.add(elemento);
   }
   public Object desencolar(){
      return cola.remove();
   }
   public Object frente(){
      return cola.peek();
   }
   public static void main(String[] args) {
      Colas c = new Colas();
      System.out.println("Elemento frente: " + c.frente());
      for (int i = 1; i<=3;i++){
         c.encolar(i);
      }
      System.out.println("Elemento frente: " + c.frente());
      System.out.println("desencolar : " + c.desencolar());
      System.out.println("Elemento frente: " + c.frente());
   }
}
```



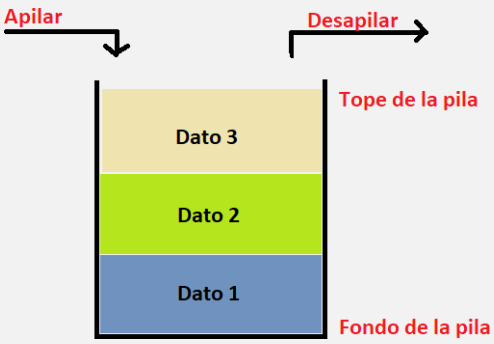
3.3.5. Pila

Una pila también es un *tipo abstracto de datos* que permite almacenar y recuperar información siguiendo un esquema LIFO (Last Input, First Output). Toda pila debe contar, al menos, con las operaciones *apilar* (push) y *desapilar* (pop).

Se denomina *TOS* al último elemento apilado.

```java
package pilas;
import java.util.LinkedList;

public class Pilas {
   private final LinkedList pila;
   public Pilas(){
      this.pila = new LinkedList();
   }
   public  void apilar(Object elemento){
      pila.push(elemento);
   }
   public Object desapilar(){
      return pila.pop();
   }
   public Object frente(){
      return pila.peek();
   }
   public static void main(String[] args) {
      Pilas c = new Pilas();
      System.out.println("Elemento frente: " + c.frente());
      for (int i = 1; i<=3;i++){
         c.apilar(i);
      }
      System.out.println("Elemento frente: " + c.frente());
      System.out.println("desenpilar : " + c.desapilar());
      System.out.println("Elemento frente: " + c.frente());
   }
}
```



3.4. Interfaz Queue

Conocida como *cola*, representa otra estructura de almacenamiento FIFO. Existen varias implementaciones que permiten crear colas con comportamientos diferentes a los vistos anteriormente.

3.4.1. PriorityQueue

Los elementos de una cola de prioridad son ordenados de acuerdo con su orden natural, independientemente del momento en el que fuesen encolados, aunque se puede emplear un comparador que se pasa como parámetro.

- No se permiten elementos nulos.
- No se permiten elementos no comparables.
- Se permiten duplicados.

Algunos de los métodos disponibles en esta colección son *add*, *clear*, *contains*, *iterator*, *offer*, *peek*, *poll*, *remove* y *size*.

3.4.1. ArrayDeque

Es un tipo de array que implementa la subinterfaz *Deque*. Esta clase y su iterador implementan todos los métodos de las interfacess *Collection* e *Iterator*.

- Usa una matriz de tamaño variable.
- No permite elementos nulos.
- Puede ser usada como fila, cola o ambos a la vez (puhs() inserta por delante, y add() por detrás).

https://www.instintoprogramador.com.mx/2019/12/java-linkedlist.html

3. Colecciones

2.4. Interfaz Map

La **interfaz** Map permite almacenar pares clave-valor de manera qque sea fácil buscar un valor a partir de su clave. Las colecciones que implementan Map no pueden contener claves duplicadas, cada clave únicamente se corresponde con un valor.

Map<Integer, String> nombreMap = new HashMap<Integer, String>();

Declaración de un Map (un HashMap) con clave "Integer" y Valor "String".

Las claves pueden ser de cualquier tipo de objetos, aunque los más utilizados como clave son los objetos predefinidos de Java como: String, Integer, Double !!!!CUIDADO los Map no permiten datos atómicos

Los principales métodos para trabajar con los Map son los siguientes:

nombreMap.size();

Devuelve el número de elementos del Map.

nombreMap.isEmpty();

Devuelve *true* si no hay elementos en el Map y *false* si los hay.

nombreMap.put(K clave, V valor);

Añade el par clave-valor al Map nombreMap.

nombreMap.get(K clave);

Devuelve el valor de la clave que se le pasa como parámetro o *'null'* si la clave no existe.

nombreMap.clear();

Borra todos los componentes del Map.

nombreMap.remove(K clave);

Borra el par clave/valor de la clave que se le pasa como parámetro.

nombreMap.containsKey(K clave);

Devuelve true si en el Map hay una clave que coincide con K.

nombreMap.containsValue(V valor);

Devuelve true si en el Map hay un Valor que coincide con V.

nombreMap.values();

Devuelve una "Collection" con los valores del Map.

nombreMap.keySet();

Devuelve un conjunto que contiene todas las claves del mapa.

nombreMap.putAll(Map m);

Copia el mapa m al mapa nombreMapa.

nombreMap.replace(K clave, V valor);

Sustituye el valor del par correspondiente a la clave pasada por argumento. Devuelve el valor anterior si se ha realizado algún cambio, o *null* en caso contrario.

Otro elemento importante a la hora de trabajar con los *Map* son los "Iteradores" (*Iterator*), con los que se recorrerán los Map.

Los Iteradores solo tienen tres métodos:

hasNext()

Para comprobar que siguen quedando elementos en el iterador,

next()

Para que nos dé el siguiente elemento del iterador

remove()

Que sirve para eliminar el elemento del Iterador.

En realidad, se puede prescindir de los iteradores para trabajar con los Map, ya que la gran ventaja de los Map frente a los ArrayList, es que estos tienen una clave asociada al objeto y se pueden buscar por la clave, aunque nunca está de más saber utilizar los iteradores para manejar los Map.

Java tiene implementadas varias "clases Map". Las tres que más importantes y útiles son la clase "*HashMap*", "*TreeMap*" y "*LinkedHashMap*":

HashMap: Los elementos que inserta en el Map no tendrán un orden específico. No aceptan claves duplicadas ni valores nulos.

```
Map<Integer, String> map = new HashMap<Integer, String>();
map.put(1, "Casillas");
map.put(3, "Pique");
map.put(11, "Capdevila");
map.put(16, "Busquets");
map.put(18, "Pedrito");
map.put(7, "Villa");

// Imprimimos el Map con un Iterador
Iterator it = map.keySet().iterator();
while(it.hasNext()){
        Integer key = (Integer) it.next();
        System.out.println ("Clave: " + key + " -> Valor: " + map.get(key));
}
```

El resultado obtenido no sigue un orden lógico.

```
Clave: 16 -> Valor: Busquets
Clave: 1  -> Valor: Casillas
Clave: 18 -> Valor: Pedrito
Clave: 3  -> Valor: Pique
Clave: 7  -> Valor: Villa
Clave: 11 -> Valor: Capdevila
```

TreeMap: Es bastante más lento que hashMap, ya que almacena las claves ordenadas ascendentemente en un árbol. Esta colección no adminte valores *null*.

```
Map<Integer, String> treeMap = new TreeMap<Integer, String>();
map.put(1, "Casillas");
map.put(3, "Pique");
map.put(11, "Capdevila");
map.put(16, "Busquets");
map.put(18, "Pedrito");
map.put(7, "Villa");

// Imprimimos el Map con un Iterador
Iterator it = treeMap.keySet().iterator();
while(it.hasNext()){
        Integer key = (Integer) it.next();
        System.out.println ("Clave: " + key + " -> Valor: " + treeMap.get(key));
}
```

El resultado obtenido no sigue un orden lógico.

```
Clave: 1  -> Valor: Casillas
Clave: 3  -> Valor: Pique
Clave: 7  -> Valor: Villa
Clave: 11 -> Valor: Capdevila
Clave: 16 -> Valor: Busquets
Clave: 18 -> Valor: Pedrito
```

LinkedHashMap: Esta implementación mantiene una lista con un enlace doble en todas sus entradas. Inserta en el Map los elementos en el orden en el que se van insertando; es decir, que no tiene una ordenación de los elementos como tal, por lo que esta clase realiza las búsquedas de los elementos de forma más lenta que las demás clases.

Empleando el mismo código (cambiando treeMap por LinkedHashMap), el resultado es que ordena los objetos tal y como se han ido introduciendo:

```
Clave: 1  -> Valor: Casillas
Clave: 3  -> Valor: Pique
Clave: 11 -> Valor: Capdevila
Clave: 16 -> Valor: Busquets
Clave: 18 -> Valor: Pedrito
Clave: 7  -> Valor: Villa
```