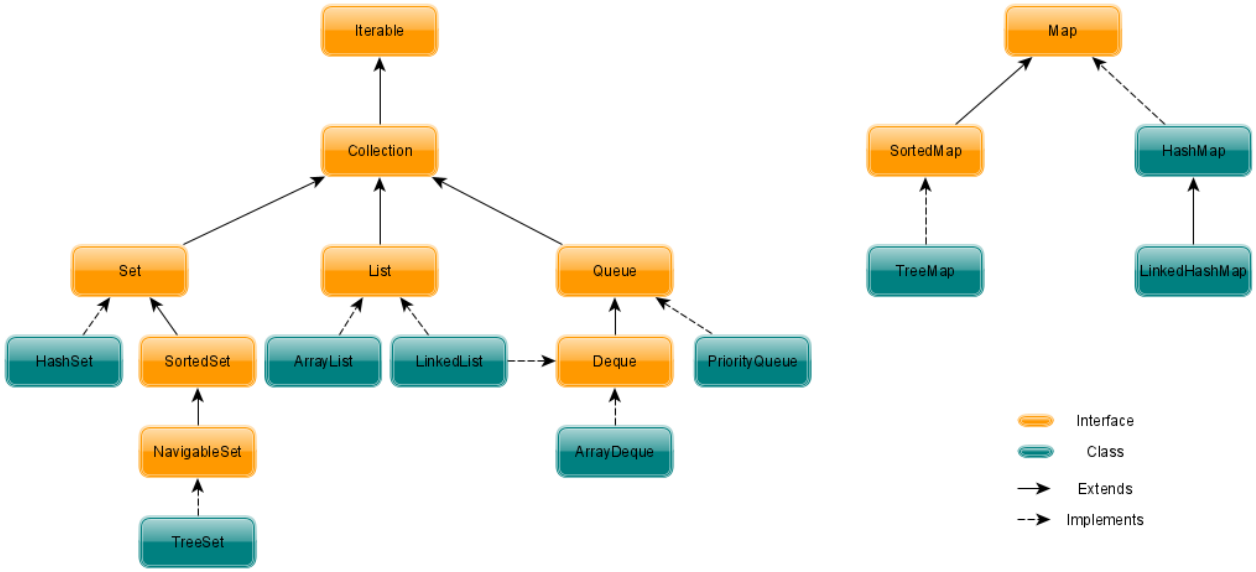
Almacenamiento de Objetos en Java

1. Introducción

Cuando se usan de estructuras de datos estáticas como los arrays, su tamaño se establece en el momento de su creación y no puede variar en tiempo de ejecución. Esto hace que no resulten adecuadas cuando se desconoce a priori la cantidad máxima de elementos que van a almacenar. En ese caso es preferible el uso de estructuras de datos dinámicas, ya que su tamaño varía en tiempo de ejecución a medida que se van realizando inserciones y borrados.

A lo largo de la historia de la computación se han ido desarrollando distintos tipos de estructuras de datos dinámicas, tales como listas, pilas, colas, tablas hash o diferentes tipos de árboles, que han sido clave a la hora de crear algoritmos eficientes para solucionar muchos tipos de problemas. En la actualidad, los lenguajes de programación incluyen en sus librerías estándar la implementación de varias estructuras de datos dinámicas que facilitan el desarrollo de aplicaciones de calidad. Dentro del API de Java se encuentra el *Java Collections Framework*, que define una amplia variedad de clases para crear contenedores de objetos, cada una de las cuales implementa una determinada estructura de datos dinámica subyacente. En este *framework* se definen un conjunto de interfaces y clases en dos jerarquías independientes para representar dos tipos de contenedores de objetos, las colecciones y los mapas[1]:



A partir de la versión 5 de Java, las interfaces y clases de estas dos jerarquías se definen mediante tipos genéricos. Por tanto, cualquier referencia a una colección se declara mediante la clase correspondiente, parametrizada con el tipo de elementos que van a almacenar en ella. Por ejemplo, la sentencia siguiente declara una referencia a una lista enlazada en la que se van a almacenar cadenas de caracteres:

```
LinkedList<String> lista;
```

Parámetro tipo

Tipo parametrizado

[1] Aunque los mapas no están en la misma jerarquía que las colecciones (no implementan la interface Collection), pertenecen al mismo framework debido a que desempeñan una función similar como contenedores de objetos. En el apartado de mapas se aclara la razón por la cual se definen en su propia jerarquía de clases.

2. Colecciones

En la raíz de la jerarquía de clases del *Java Collections Framework* está la interface Collection, que establece las operaciones básicas para cualquier colección con independencia de los detalles de su implementación. La tabla siguiente recoge algunos de sus métodos:

Método	Descripción
`add`	Añade un objeto a la colección.
`remove`	Elimina un objeto de la colección.
`size`	Retorna el número de objetos que almacena la colección.
`clear`	Elimina todos los objetos de la colección.
`contains`	Comprueba si un objeto está en la colección.
`isEmpy`	Comprueba si una colección está vacía.

Dado que todas las colecciones implementan la interface Collection, es posible manejar cualquiera de ellas mediante estas operaciones básicas. En el ejemplo siguiente se muestra cómo se manipulan dos tipos distintos de colecciones con una única referencia polimórfica, realizando con cada una de ellas las mismas operaciones básicas:

```
Collection<String> c;

c = new ArrayList<String>();
c.add("Carmen");
c.add("Fernando");
System.out.println("La lista contiene a Roberto: " + c.contains("Roberto"));
c.remove("Fernando");
System.out.println("Número de nombres en la lista: " + c.size());
c.clear();

c = new HashSet<String>();
c.add("Elisa");
c.add("Roberto");
System.out.println("El conjunto contiene a Roberto: " +
c.contains("Roberto"));
c.remove("Roberto");
System.out.println("Numero de nombres en el conjunto: " + c.size());
c.clear();
```

Sin embargo, si fuese necesario acceder desde una referencia polimórfica a la funcionalidad definida en alguna subclase, habría que convertir dicha referencia al tipo adecuado. Por ejemplo, en la interface List se sobrecarga el método get para poder obtener cualquier elemento a partir de un índice. Así pues, para invocar a este método con la referencia polimórfica declarada en el ejemplo anterior, tendríamos que hacerlo de la forma siguiente:

```
Collection<String> l = new ArrayList<String>();
l.add("Carmen");
l.add("Fernando");
System.out.println(((List<String>) c).get(1));
```

Obviamente, si no vamos a necesitar referencias polimórficas será preferible declarar las referencias usando tipos más específicos para evitar la conversión:

```
List<String> c = new ArrayList<String>();
c.add("Carmen");
c.add("Fernando");
System.out.println(c.get(1));
```

2.1. Colecciones de datos y tipos de datos primitivos

Las colecciones son contenedores de objetos. Por tanto, para poder almacenar en ellas datos de tipo primitivo es necesario recurrir a clases envoltorio como Integer, Float, etc. En el ejemplo siguiente se crea una lista para almacenar números enteros y se añade un número a la lista:

```
List<Integer> lista = new ArrayList<Integer>();
lista.add(100);
```

El compilador sustituye la expresión lista.add(100) por la expresión lista.add(Integer.valueOf(100)) en un proceso denominado *boxing* que se realiza de forma transparente. Esto ocurre de forma automática en cualquier contexto en el que se espera un objeto de una clase envoltorio, pero en su lugar se especifica un dato de tipo primitivo.

El proceso inverso se denomina *unboxing* y ocurre en cualquier contexto en el que se espera un tipo de dato primitivo, pero en su lugar se especifica una referencia a un objeto del tipo envoltorio correspondiente. Por ejemplo, el compilador sustituye de forma transparente la expresión int valor = lista.get(0) por la expresión int valor = lista.get(0).intValue().

2.2. Iterar sobre una colección

Al iterar sobre una colección se recorren en un orden determinado todos los objetos almacenados en ella. El orden de iteración depende del tipo de colección, por ejemplo, en una lista los elementos se recorren en el mismo orden en que fueron insertados. En cualquier caso, el proceso de iteración se puede hacer de varias formas:

- Con iteradores, disponibles a partir de la versión 1.2 de Java en respuesta a las limitaciones que presenta el bucle for clásico a la hora de recorrer cualquier tipo de colección.
- Utilizando la versión mejorada del bucle for, disponible a partir de la versión 5 de Java. Utiliza una sintaxis específica para iterar tanto sobre colecciones como sobre arrays. De hecho, puede iterar sobre cualquier objeto que implemente la interface Iterable.
- Las dos anteriores representan formas de iteración externa. Con la llegada de la versión 8 de Java y la incorporación al lenguaje de elementos de la programación funcional, se hace posible una nueva forma de iteración denominada iteración interna. Se basa en la utilización de expresiones lambda junto con el nuevo método forEach declarado en la interface Iterable implementada por todas las colecciones.

2.2.1. Iteradores

La interface Collection declara el método iterator para que cualquier tipo de colección retorne un objeto Iterator con el que iterar sobre los elementos que contiene. Iterator es una interface que declara los tres métodos siguientes:

Método	Descripción
`boolean hasNext()`	Retorna true si quedan elementos sobre los que aún no se ha iterado y false en caso contrario.
`E next()`	Retorna el siguiente objeto de tipo E en el orden de iteración, siendo E el tipo de los objetos almacenados en la colección subyacente.
`void remove()`	Elimina de la colección subyacente el objeto retornado en la última llamada al método next.

En el ejemplo siguiente muestra como iterar sobre una colección con un iterador:

```
List<String> alumnos = new ArrayList<String>();
alumnos.add("Carmen");
alumnos.add("Fernando");
alumnos.add("Elisa");
alumnos.add("Roberto");
Iterator<String> i = alumnos.iterator();
while (i.hasNext()) {
 String nombre = i.next();
 System.out.println(nombre);
}
```

Cualquier modificación del contenido de una colección mientras se está iterando provoca el lanzamiento de la excepción ConcurrentModificationException. Ejemplo:

```
Iterator<String> i = alumnos.iterator();
while (i.hasNext()) {
 String nombre = i.next();
 if (nombre.equals("Elisa"))ç
        // la ejecución de la sentencia siguiente provoca que se lance
        // la excepción ConcurrentModificationException
        alumnos.add("Felix");
}
```

La excepción a esta regla está en la posibilidad de eliminar objetos de la colección con el método remove del iterador. El ejemplo siguiente muestra cómo eliminar de la colección anterior todos los alumnos cuyo nombre empieza por 'F' utilizando un iterador:

```
Iterator<String> i = alumnos.iterator();
while (i.hasNext())
 if (i.next().startsWith("F"))
        i.remove();
}
```

2.2.2. Bucle for mejorado

Este bucle está diseñado para iterar sobre arrays y en general sobre cualquier instancia de una clase que implemente la interface Iterable. En las colecciones se exige que la referencia correspondiente se declare con un tipo parametrizado. El ejemplo siguiente muestra cómo iterar sobre una colección:

```
List<String> alumnos = new ArrayList<String>();
alumnos.add("Carmen");
alumnos.add("Fernando");
alumnos.add("Elisa");
for (String nombre: alumnos)
 System.out.println(nombre);
```

Con este método de iteración tampoco es posible añadir o eliminar objetos de la colección mientras se itera. Hacerlo también provocará el lanzamiento de la excepción ConcurrentModificationException.

2.2.3. Método forEach

Esta forma de iterar reduce el número de sentencias debido a que se implementa en el método forEach toda la lógica necesaria (iteración interna). Lo único que se necesita es una expresión lambda que determina qué hacer con cada elemento de la colección. Por ejemplo:

```
List<String> alumnos = new ArrayList<String>();
alumnos.add("Carmen");
alumnos.add("Fernando");
alumnos.add("Elisa");
alumnos.forEach(nombre -> System.out.println(nombre));
```

2.3. Listas

Las colecciones que implementan la interface List representan una lista de elementos, es decir, una secuencia ordenada en la que:

- El orden de iteración se corresponde con el orden de inserción de los elementos.
- Se admiten tanto valores duplicados como el valor null.
- El método add(objeto) añade cada objeto a continuación del almacenado en la última posición, pasando a ser el último elemento.

La interface List extiende a la interface Collection con métodos que basan su funcionalidad en la utilización de índices. Los índices son números enteros que identifican las posiciones dentro de la colección de forma similar a como lo hacen en los arrays: el índice 0 corresponde a la primera posición, el 1 a la segunda, y así sucesivamente hasta la posición size() – 1, en la que se almacena el último elemento. Además, al igual que ocurre con los arrays, la utilización de índices fuera de rango también provoca el lanzamiento de la excepción IndexOutOfBoundsException.

En el ejemplo siguiente se añaden varios objetos a una lista y después se muestra por pantalla. Al mostrar el contenido, se puede ver que los objetos están ordenados según el orden de inserción:

```
List<String> alumnos = new ArrayList<String>();
alumnos.add("Carmen");
alumnos.add("Fernando");
alumnos.add("Elisa");
System.out.println(alumnos);
```

Salida:

```
[Carmen, Fernando, Elisa]
```

El método add(índice, objeto) inserta un objeto en una posición dada, desplazando los objetos que se encuentran a partir de dicha posición. Por ejemplo, si insertamos "Manuel" en la posición 1 de la lista anterior, "Fernando" y "Elisa" se desplazan a la posición 2 y 3 respectivamente:

```
alumnos.add(1, "Manuel");
System.out.println(alumnos);
```

Salida:

```
[Carmen, Manuel, Fernando, Elisa]
```

Además de inserciones, en una lista también es posible realizar operaciones de consulta, modificación y borrado basadas en índices con los métodos siguientes:

- get(indice): retorna el objeto almacenado en la posición especificada.
- remove(indice): elimina de la colección el elemento almacenado en la posición especificada.
- set(índice, objeto): reemplaza el objeto almacenado en la posición especificada.

En este documento se describen dos tipos de listas, ArrayList y LinkedList. La diferencia entre ambas radica en la estructura de datos subyacente que implementa cada una de ellas y, por tanto, también en el rendimiento de las operaciones de consulta, inserción, modificación y borrado. En general, las operaciones de inserción –add– y borrado –remove– son más eficientes en una LinkedList que en un ArrayList. Sin embargo, con las operaciones de consulta –get– y modificación –set– ocurre lo contrario.

2.3.1. ArrayList

La estructura de datos subyacente en este tipo de lista es un array. La naturaleza estática de los arrays obliga a implementar toda la lógica necesaria para proporcionar la funcionalidad de una estructura de datos dinámica. Así pues, esta colección se crea un array con una determinada capacidad inicial. Cuando esta se agota, se crea un array nuevo con un 50% más de capacidad y se le transfirieren los elementos del array original, dejando este último a merced del recolector de basura.

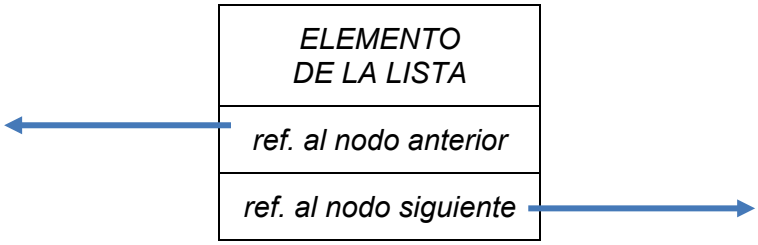
Las operaciones de inserción y borrado de elementos en posiciones intermedias son costosas incluso aunque no impliquen redimensionar el array, ya que obligan a mover una determinada cantidad de elementos hacia nuevas posiciones dentro del array.

La clase ArrayList proporciona tres constructores:

- `ArrayList()`: crea una lista con un tamaño inicial predeterminado.
- `ArrayList(int initialCapacity)`: crea una lista con el tamaño inicial especificado.
- `ArrayList(Collection<? extends E> c)`: crea una copia de la lista especificada.

2.3.2. LinkedList

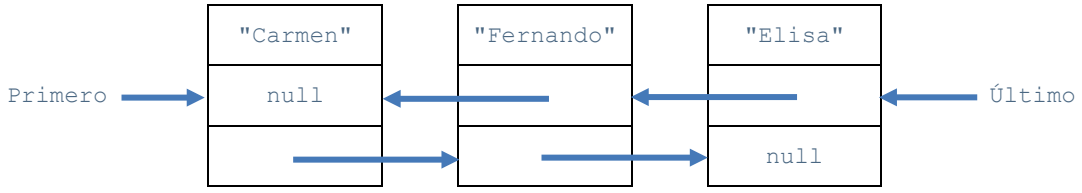
La clase LinkedList utiliza como estructura de datos subyacente una lista doblemente enlazada. Esta estructura de datos está basada en la utilización de subestructuras denominadas nodos que tienen una doble función: almacenar cada elemento de la lista a la vez que apuntan al nodo anterior y al nodo siguiente:



En una lista doblemente enlazada se guardan dos referencias especiales: una que apunta al primer nodo y otra que apunta al último nodo. En una lista vacía estas dos referencias tendrán el valor null y en una lista con un solo elemento harán referencia al mismo nodo.

En el primer nodo de la lista, la referencia que apunta al anterior tendrá el valor nulo. En el último nodo de la lista, la referencia que apunta al siguiente tendrá el valor nulo.

La representación gráfica de una lista que almacena varios elementos podría ser algo así:



Las diferencias más notables con ArrayList son:

- La lista crece y decrece de forma natural con las inserciones y borrados. Por tanto, es innecesario un constructor que especifique un tamaño inicial.
- Define nuevos métodos que realizan de forma más eficiente las operaciones con el primer y último elemento de la lista: addFirst, addLast, getFirst, getLast, removeFirst y removeLast. Estos métodos se definen por conveniencia, pero no incrementan la funcionalidad de la interface List. Por ejemplo, en una lista no vacía las expresiones lista.getFirst() y lista.get(0) son equivalentes.

2.3.3. Iterar sobre listas

Para las listas de ha definido un iterador avanzado que permite recorrer una lista en ambas direcciones, modificarla durante el proceso de iteración y obtener la posición actual. Este iterador se obtiene invocando al método listIterator de cualquier lista. Este retorna un objeto de tipo ListIterator, una interface que extiende a la interface Iterator con los métodos siguientes:

Método	Método y descripción
`void add(E e)`	Inserta el elemento especificado en la lista.
`boolean hasPrevious()`	Retorna true si este iterador tiene más elementos que preceden al elemento actual.
`E previous()`	Se sitúa en el elemento previo y lo retorna
`int nextIndex()`	Retorna el índice del elemento que retornará la siguiente llamada a next.
`int previousIndex()`	Retorna el índice del elemento que retornará la siguiente llamada a previous.
`void set(E e)`	Reemplaza el último elemento retornado por next o previous con el elemento especificado.

2.3.4. Ejemplo de un problema resueltos con listas

Se trata de desarrollar un programa que dadas varias secuencias de números enteros elimine de las mismas una determinada cantidad máxima de elementos, cada uno de ellos elegido con el algoritmo siguiente: se recorre desde el principio la secuencia y si se encuentra un número menor que el siguiente, ese número se elimina y finaliza el proceso. Si se llega al final sin encontrar un número menor que el siguiente, se elimina el último.

La entrada de datos se hará por teclado y se procesarán de la forma siguiente:

1. Se lee una línea con el número S de secuencias que se van a procesar.
2. Se obtienen los datos de cada secuencia de la forma siguiente:
    - Se lee una línea con el número N de elementos totales y el número K de elementos que hay que borrar, ambos separados por espacios en blanco.
    - Se lee una línea con la secuencia de elementos, cada uno de ellos separados por espacios en blanco
3. Se procesa la secuencia, se muestra el resultado y se pasa a la siguiente

Para simplificar el problema se asume que la entrada siempre es correcta, es decir, el número de elementos a borrar nunca será mayor que el número de elementos de la secuencia y todos los valores de entrada son números enteros.

Ejemplo de datos de entrada y datos de salida que se deberían de obtener:

ENTRADA	SALIDA	
3		Número de secuencias
3 1		Primera secuencia
3 100 1		
	[100, 1]	
5 2		Segunda secuencia
19 12 3 4 17		
	[19, 12, 17]	
5 3		Tercera secuencia
23 45 11 77 18		
	[77, 18]	

Solución:

```
import java.util.LinkedList;
import java.util.List;
import java.util.ListIterator;
import java.util.Scanner;

public class Main4 {

    public static void main(String[] args) {
        Scanner in = new Scanner(System.in);
        List<Integer> lista = new LinkedList<Integer>();
        int s = in.nextInt();
        for (int i=0; i<s; i++) {
            int n = in.nextInt();
            int k = in.nextInt();
            for (int j=0; j<n; j++)
                lista.add(in.nextInt());
            for (int j=0; j<k; j++)
                borrarElemento(lista);
            System.out.println(lista);
            lista.clear();
        }
        in.close();
    }

    static void borrarElemento(List<Integer> lista) {
        ListIterator<Integer> i = lista.listIterator();
        if (i.hasNext()) {
            int a = i.next();
            while (i.hasNext()) {
                int b = i.next();
                if (a < b) {
                    i.previous();
                    i.previous();
                    i.remove();
                    return;
                }
                else
                    a = b;
            }
            i.remove();
        }
    }

}
```

2.4. Colas y Pilas

En el *Collections Framework* se definen las clases e interfaces siguientes para el manejo de colas y pilas:

C/I	NOMBRE	DESCRIPCIÓN
Interface	`Queue`	Cola convencional.
Interface	`Deque`	Cola doblemente terminada.
Clase	`PriorityQueue`	Implementa la interface Queue para definir una cola de prioridad.
Clase	`LinkedList`	Implementa la interface Deque para definir tanto una cola convencional como una cola doblemente terminada.
Clase	`ArrayDeque`	Implementa la interface Deque para definir una cola doblemente terminada.

Las colas se crean instanciando las clases PriorityQueue, LinkedList o ArrayDeque. Las pilas se crean instanciando cualquiera de las dos últimas.

Tanto pilas como colas admiten valores duplicados, sin embargo, solamente aquellas colas o pilas que se creen instanciando la clase LinkedList admiten valores nulos.

2.4.1. Colas

Una **cola**, **queue** en inglés, es una lista de elementos sujeta a las restricciones siguientes:

- Los elementos se añaden por un extremo y se extraen por el otro. Por tanto, siempre se extrae el elemento de mayor antigüedad en la cola, por lo que también se conocen como estructuras FIFO (**F**irst **I**n **F**irst **O**ut, el primero en entrar el primero en salir).
- El elemento que se encuentra al principio de la cola de denomina **head** y el que se encuentra al final de la cola se denomina **tail**.
- Los elementos se insertan en el final de la cola mediante una operación denominada **enqueue**.
- Los elementos se retiran del principio de la cola mediante una operación denominada **dequeue**.
- El elemento head se puede consultar mediante una operación denominada **front** o **element**.

Además de las colas convencionales, existen otros dos tipos de colas:

- Cola doblemente terminada, en inglés **deque** (**D**ouble **E**nded **Que**ue), son colas en las que se pueden añadir y retirar los elementos por ambos extremos. Aúnan en una única estructura la funcionalidad de una cola y de una pila.
- Colas de prioridad, en las que los elementos tienen asociada una prioridad que determina el orden en el que se retiran de la cola. Siempre se retira el elemento que tenga asociada la mayor prioridad. En caso de que varios elementos compartan la mayor prioridad la extracción se realiza como se haría en una cola convencional.

La sentencia siguiente crea una **cola convencional** que se maneja a través de una referencia a un objeto de tipo Queue:

```
Queue<String> cola = new LinkedList<String>();
```

La tabla siguiente resume los métodos que se declaran en la interface Queue para realizar las operaciones propias de una cola convencional. Se distinguen dos tipos de métodos para cada operación, aquellos que lanzan una excepción si la operación no es posible por alguna razón, y aquellos que retornan un valor especial en la misma situación:

	Métodos	
Operaciones	Lanzan excepción	Valor de retorno especial
Enqueue	`add(e)`	`offer(e)`
Dequeue	`remove()`	`poll()`
Front	`element()`	`peek()`

Las sentencias siguientes crean **colas doblemente terminadas** instanciando las clases LinkedList y ArrayDeque respectivamente para manejarlas a través de una referencia a un objeto Deque:

```
Deque<String> cola1 = new LinkedList<String>();
Deque<String> cola2 = new ArrayDeque<String>();
```

La tabla siguiente resume los métodos que se declaran en la interface Deque para realizar las operaciones propias de una cola doblemente terminada. Se distinguen dos tipos de métodos para cada operación, aquellos que lanzan una excepción si la operación no es posible por alguna razón, y aquellos que retornan un valor especial en la misma situación:

	Métodos			
	Primer elemento (head)		Último elemento (tail)	
Operaciones	Lanzan excepción	Valor de retorno especial	Lanza excepción	Valor de retorno especial
Enqueue	`addFirst(e)`	`offerFirst(e)`	`addLast(e)`	`offerLast(e)`
Dequeue	`removeFirst()`	`pollFirst()`	`removeLast()`	`pollLast()`
Front	`getFirst()`	`peekFirst()`	`getLast()`	`peekLast()`

La sentencia siguiente crea una **cola de prioridad** que se maneja a través de una referencia a un objeto de tipo `PriorityQueue`:

```
PriorityQueue<String> cola = new PriorityQueue<String>();
```

La clase PriorityQueue ofrece varios constructores que determinan cómo se establece la prioridad de los elementos. Así, la prioridad de cada elemento puede estar basada en el orden natural definido en la clase correspondiente o en un objeto Comparator que se ha de especificar en el constructor y que determina el criterio de ordenación para los elementos que se van a insertar. En el primer caso no se permite la inserción de objetos no comparables (como se verá más adelante, un objeto es comparable si su clase implementa la interface Comparable).

La clase PriorityQueue implementa la interface Queue. Por tanto, una vez creada, la cola de prioridad se maneja utilizando los mismos métodos que se utilizan para manejar una cola convencional.

2.4.2. Pilas

Una **pila**, **stack** en inglés, es una lista de elementos sujeta a las restricciones siguientes:

- Los elementos se añaden y se eliminan por el mismo extremo, denominado **tope** de la pila. Por tanto, siempre se extrae el elemento que se ha añadido más recientemente, por

lo que también se conocen como estructuras LIFO (**L**ast **I**n **F**irs **O**ut, último en entrar primero en salir).

- Los elementos se insertan en el tope de la pila mediante una operación denominada ***push***.
- Los elementos se retiran del tope de la pila mediante una operación denominada ***pop***.
- El elemento del tope de la pila se puede obtener sin retirarlo mediante una operación denominada ***peek***.

En el *Collection Framework* se define la clase Stack que representa una pila. Sin embargo, en la documentación del API de java se aconseja instanciar las clases LinkedList o ArrayDeque para crear pilas, ya que implementan la interface Deque que representa una pila más completa y de mayor consistencia que la clase Stack.

Las sentencias siguientes crean **pilas** instanciando las clases LinkedList y ArrayDeque respectivamente para manejarlas a través de una referencia a un objeto tipo Deque:

```
Deque<String> pila1 = new LinkedList<String>();
Deque<String> pila2 = new ArrayDeque<String>();
```

La tabla siguiente resume los métodos que se utilizan para realizar las operaciones propias de una pila cuando se maneja mediante una referencia a un objeto Deque:

Operaciones	Métodos
Push	`addFirst(e) / offerFirst(e) / push(e)`
Pop	`removeFirst() / pollFirst() / pop()`
Peek	`getFirst() / peekFirst() / element() / peek()`

2.4.3. Ejemplos de problemas resueltos con colas y pilas

Ejemplo 1: Programa que simule la ejecución organizada de una serie de procesos. Los procesos se identifican con un número entero de 1 a N, siendo N el número de procesos. Los datos de entrada al programa son:

- El número N de procesos.
- Los N pares *(nombre_proceso*, *id_proceso)* especificados en un orden arbitrario.
- Los N identificadores de cada proceso en el orden ideal en que deberían ser ejecutados.

Cada proceso se representa mediante la clase siguiente, que simula su ejecución mostrando su nombre en la consola:

```
public class Proceso {

    private String nombre;
    private int id;

    public Proceso(String nombre, int id) {
        this.nombre = nombre;
        this.id = id;
    }

    public String getNombre() {
        return nombre;
    }

    public int getId() {
        return id;
```

```
    }

    public void ejecutar() {
        System.out.println("Ejecutando " + nombre);
    }

}
```

Ejemplo de datos de entrada y datos de salida que se deberían de obtener:

Entrada:

```
3
P3 3 P2 2 P1 1
1 3 2
```

Salida:

```
Ejecutando P1
Ejecutando P3
Ejecutando P2
```

Este problema se puede resolver con dos **colas**, una en la que se insertan los procesos que se crean con los datos introducidos en la segunda línea, y otra en la que se van almacenando los N identificadores de proceso introducidos en la tercera línea. Para simplificar la solución del problema se asume que los datos introducidos siempre cumplen las especificaciones dadas:

```
import java.util.LinkedList;
import java.util.Queue;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt();
        Queue<Proceso> procesos = new LinkedList<Proceso>();
        for (int i=0; i<n; i++)
            procesos.offer(new Proceso(s.next(), s.nextInt()));
        Queue<Integer> ideal = new LinkedList<Integer>();
        for (int i=0; i<n; i++)
            ideal.offer(s.nextInt());
        int esperado;
        Proceso proceso;
        while(!procesos.isEmpty()) {
            proceso = procesos.poll();
            esperado = ideal.poll();
            while (proceso.getId() != esperado) {
                procesos.offer(proceso);
                proceso = procesos.poll();
            }
            proceso.ejecutar();
        }
    }

}
```

Ejemplo 2: Programa que lee por teclado una línea con varios números enteros separados por espacios y los muestra en orden inverso.

La solución con una **pila** consiste en introducir en ella todos los números según se van leyendo. Cuando se hayan añadido todos, se van extrayendo y mostrando en la pantalla uno a uno.

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) throws IOException {
        Deque<Integer> pila = new ArrayDeque<Integer>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Números: ");
        Scanner in = new Scanner(br.readLine());
        while (in.hasNextInt())
            pila.push(in.nextInt());
        while (!pila.isEmpty())
            System.out.printf("%d ", pila.pop());
        in.close();
    }

}
```

Ejemplo 3: Programa que determina si los paréntesis, corchetes y llaves utilizados en una expresión introducida por teclado están balanceados. Este problema se puede resolver con una **pila** en la que se insertan los elementos de apertura y se extraen cuando se lee uno de cierre. Si el elemento de apertura extraído no casa con el elemento de cierre o la pila está vacía cuando se lee un elemento de cierre o la pila no está vacía cuando se termina de procesar la expresión, se puede concluir que estos elementos no están balanceados.

Ejemplos de expresiones balanceadas	Ejemplos de expresiones no blanceadas
([]{})	([}])
((({([])})())[]){})	((({([])})())[]){}

Solución:

```java
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.util.ArrayDeque;
import java.util.Deque;

public class Main {

    public static void main(String[] args) throws IOException {
        Deque<Character> pila = new ArrayDeque<Character>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        System.out.println("Expresión: ");
        String linea = br.readLine();
        char c;
        int i = 0;
        boolean balanceados = true;
        while (i < linea.length() && balanceados) {
            c = linea.charAt(i++);
            if (c == '(' || c == '[' || c == '{')
                pila.push(c);
            else if (c == ')' || c == ']' || c == '}')
                balanceados = casan(pila.pop(), c);
        }
        if (balanceados && !pila.isEmpty()) {
            balanceados = false;
            pila.clear();
        }
        if (!balanceada)
            System.out.print("no ");
        System.out.println("están balanceados");
    }
```

```
    static boolean casan(char apertura, char cierre) {
        return (apertura == '(' && cierre == ')') ||
               (apertura == '[' && cierre == ']') ||
               (apertura == '{' && cierre == '}');
    }

}
```

2.5. Conjuntos

La interface Set, implementada por las clases HashSet y TreeSet, es una abstracción del concepto matemático de conjunto. Por tanto, las colecciones que implementan esta interface no van a admitir que se almacene más de una vez un mismo objeto y las operaciones que cobran mayor relevancia van a ser las operaciones propias de conjuntos:

- Pertenencia al conjunto: métodos contains.
- Subconjunto: containsAll.
- Unión de conjuntos: método addAll.
- Intersección de conjuntos: método retainAll.
- Diferencia de conjuntos: método removeAll.

Las diferencias entre `HashSet` y `TreeSet` se resumen en la tabla siguiente:

	HashSet	TreeSet
Estructura de datos	Tabla hash	Árbol AVL
Admite valores nulos	Si	No
Ordena los elementos	No	Si

Ejemplo	`Set<String> s = new HashSet<String>();`
	`s.add("Fernando");`
	`s.add("Elisa");`
	`s.add("Mateo");`
	`s.add(null);`
	`System.out.println(s);`
		`Set<String> s = new TreeSet<String>();`
		`s.add("Fernando");`
		`s.add("Elisa");`
		`s.add("Mateo");`
		`System.out.println(s);`

Salida	`[Mateo, null, Fernando, Elisa]`	`[Elisa, Fernando, Mateo]`

2.5.1. Ejemplos de problemas resueltos con conjuntos

Ejemplo 1: Programa que lee una línea de texto de la entrada estándar y muestra qué caracteres están repetidos y qué caracteres aparecen una sola vez, ignorando los espacios en blanco.

Ejemplo de datos de entrada y datos de salida que se deberían de obtener:

ENTRADA	SALIDA
colecciones y mapas	**Caracteres sin repetir: [p, i, y, l, m, n]** **Caracteres repetidos: [a, c, s, e, o]**

Este problema se puede resolver con dos conjuntos, uno que vaya guardando los caracteres no repetidos y otro que vaya guardando los repetidos. Cada carácter se procesa de la forma siguiente:

- Si no pertenece a ninguno de los conjuntos, se añade al conjunto de los no repetidos.

- Si pertenece al conjunto de los no repetidos, se saca de ese conjunto y se añade al de los repetidos.
- Si pertenece al conjunto de los repetidos, no se hace nada.

Solución 1:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashSet;
import java.util.Set;

public class Main {

    public static void main(String[] args) throws IOException {
        Set<Character> repetidos = new HashSet<Character>();
        Set<Character> sinRepetir = new HashSet<Character>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String linea = br.readLine();
        for (Character c: linea.toCharArray())
            if (!Character.isSpaceChar(c)) {
                if (sinRepetir.contains(c)) {
                    repetidos.add(c);
                    sinRepetir.remove(c);
                }
                else if (!repetidos.contains(c))
                    sinRepetir.add(c);
            }
        System.out.println("Caracteres sin repetir: " + sinRepetir);
        System.out.println("Caracteres repetidos: " + repetidos);
    }

}
```

Solución 2:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashSet;
import java.util.Set;

public class Main {

    public static void main(String[] args) throws IOException {
        Set<Character> repetidos = new HashSet<Character>();
        Set<Character> sinRepetir = new HashSet<Character>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String linea = br.readLine();
        for (Character c: linea.toCharArray())
            if (!Character.isSpaceChar(c) && !repetidos.contains(c) &&
                !sinRepetir.add(c)) {
                repetidos.add(c);
                sinRepetir.remove(c);
            }
        System.out.println("Caracteres sin repetir: " + sinRepetir);
        System.out.println("Caracteres repetidos: " + repetidos);
    }

}
```

Solución 3:

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashSet;
import java.util.Set;
```

```java
public class Main {

    public static void main(String[] args) throws IOException {
        Set<Character> repetidos = new HashSet<Character>();
        Set<Character> sinRepetir = new HashSet<Character>();
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        String linea = br.readLine();
        for (Character c: linea.toCharArray())
            if (!Character.isSpaceChar(c) && !sinRepetir.add(c))
                repetidos.add(c);
        sinRepetir.removeAll(repetidos);
        System.out.println("Caracteres sin repetir: " + sinRepetir);
        System.out.println("Caracteres repetidos: " + repetidos);
    }

}
```

En este punto conviene resaltar que en la primera solución se podrían haber utilizado otro tipo de colecciones, como por ejemplo listas. Sin embargo, la segunda y la tercera solución solo son posibles usando conjuntos.

Ejemplo 2: Programa que lee de la entrada estándar varios compuestos químicos y, después de leerlos todos, muestra en la salida estándar todos los elementos químicos que forman dichos compuestos sin que se repita ninguno y en orden alfabético. La primera línea de texto contiene el número de compuestos. El resto de las líneas contienen los compuestos a razón de uno por línea. Cada compuesto se escribe como una secuencia de elementos químicos separados por espacios en blanco.

Ejemplo de datos de entrada y datos de salida que se deberían de obtener:

ENTRADA	SALIDA
4 Ce O Mo O Ce Ne Mo	[Ce, Mo, Ne, O]

Solución:

```java
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.Set;
import java.util.TreeSet;

public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Set<String> elementos = new TreeSet<String>();
        List<List<String>> compuestos = new ArrayList<List<String>>();
        int n = Integer.valueOf(br.readLine());
        for (int i=0; i<n; i++) {
            Scanner s = new Scanner(br.readLine());
            List<String> compuesto = new ArrayList<String>();
            while (s.hasNext())
                compuesto.add(s.next());
            compuestos.add(compuesto);
        }
        for (List<String> compuesto: compuestos)
            elementos.addAll(compuesto);
        System.out.println(elementos);
```

```
    }
}
```

2.6. Colecciones y arrays

Es posible que durante el desarrollo de una aplicación surja la necesidad de crear un array a partir de los elementos almacenados en una colección, o viceversa. En este apartado se exponen varias alternativas disponibles en Java.

2.6.1. Arrays a Colecciónes

Las clases del *Collection Framework* de Java no definen constructores para construir una colección a partir de un array. Si queremos crear una colección a partir del contenido de un array, además del procedimiento estándar -crear una colección vacía y recorrer el array para añadir cada uno de sus elementos a la colección-, también podemos usar otros métodos siempre que el tipo base del array no sea un tipo primitivo:

1. Usar el método asList de la clase Arrays. Por ejemplo:

   ```
   Integer [] array = {1, 2, 3, 4};
   List<Integer> list = Arrays.asList(array);
   ```

   Observaciones:

   - Sólo se pueden crear listas.
   - Crea listas inmodificables y, por tanto, cualquier operación que modifique la lista provocará en el lanzamiento de la excepción UnsupportedOperationException.

2. Usar el método addAll de la clase Collections. Por ejemplo:

   ```
   Integer [] array = {1, 2, 3, 4};
   Set<Integer> set = new HashSet<>();
   Collections.addAll(set, array);
   ```

   Observaciones:

   - La colección se ha de crear de forma explícita antes de invocar a addAll.

3. Usar el método of declarado en las interfaces List o Set. Por ejemplo:

   ```
   Integer [] array = {1, 2, 3, 4};
   List<Integer> list = List.of(array);
   ```

   Observaciones:

   - Crea colecciones inmodificables. Cualquier operación que modifique la colección derivará en el lanzamiento de la excepción UnsupportedOperationException.
   - Disponible a partir de la versión 9 de Java.

4. Usando el método stream de la clase Arrays, e invocando al método collect del *stream* retornado. Por ejemplo:

   ```
   Integer[] array = {1, 2, 3, 4};
   List<Integer> list = Arrays.stream(array).collect(Collectors.toList());
   Set<Integer> set2 = Arrays.stream(array).collect(Collectors.toUnmodifiableSet());
   ```

   Observaciones:

- La clase Collectors puede retornar diferentes *collectors* para crear varios tipos de colecciones, además de mapas, tanto modificables como inmodificables.
- Disponible a partir de la versión 8 de Java.

2.6.2. Colecciones a Arrays

Para crear un array a partir de una colección podemos crear un array del mismo tamaño que la colección e iterar sobre ella para añadir uno a uno sus elementos al array, pero en este caso también tenemos otras alternativas, pero de nuevo los arrays que se van a crear no pueden tener como tipo base un tipo primitivo:

1. Todas las colecciones tienen el método toArray para retornar un array que contenga sus elementos. Por ejemplo:

```
List<Integer> lista = List.of(1, 2, 3, 4);
Integer [] array = lista.toArray(new Integer[0]);
```

   Observaciones:

   - Para que cree un array del mismo tipo que los elementos de la colección, hay que pasarle al método toArray un array del mismo tipo y de longitud cero. En caso contrario retornará un array de elementos de tipo Object.

2. Usando el método toArray del *stream* retornado el método stream de la colección. Por ejemplo:

```
List<Integer> lista = List.of(1, 2, 3, 4);
Integer [] array = lista.stream().toArray(Integer[]::new);
```

   Observaciones:

   - Disponible a partir de la versión 8 de Java.

3. Usando el método copyOf de la clase Arrays. Por ejemplo:

```
Set<Integer> set = Set.of(1, 2, 3, 4);
Integer [] array = Arrays.copyOf(set.toArray(), set.size(), Integer[].class);
```

   Observaciones:

   - Disponible a partir de la versión 6 de Java.

4. Usando el método arrayCopy de la clase System. Por ejemplo:

```
List<Integer> lista = List.of(1, 2, 3, 4);
Integer [] array = new Integer[lista.size()];
System.arraycopy(lista.toArray(), 0, array, 0, lista.size());
```

   Observaciones:

   - El array tiene que estar creado antes de invocar al método arrayCopy.

3. Mapas

Un mapa es una colección en la que cada elemento almacenado tiene dos componentes: una clave y un valor. En el *Collections Framework* la interface Map está en la raíz de la jerarquía de clases en la que se definen los mapas. Esta interface no extiende a la interface Collection debido que sus métodos están orientados a las relaciones clave-valor.

Tanto claves como valores han de ser objetos y cada dupla clave-valor recibe el nombre de entrada. Las características básicas que definen la funcionalidad de un mapa son:

- Las claves tienen una función similar a las claves primarias de las tablas en el modelo relacional de datos, es decir, se define sobre ellas una restricción de unicidad.

- Un mismo valor puede estar asociado a más de una clave en varias entradas.
- No se puede utilizar null como clave, pero si como valor.

Por ejemplo, podemos almacenar nombres de personas en un mapa usando como clave su NIF. De esta forma no habrá problema a la hora de almacenar los datos de dos personas que tengan el mismo nombre:

CLAVE	VALOR	
11111111	Elena Fernández	*
11222333	Carlos González	
11444555	Rodrigo Rodíguez	
11321123	Elena Fernández	*
11777999	Pilar Ramos	

Se pueden crear tres tipos de mapas:

- HashMap: implementa la interface Map y se caracteriza por utilizar una tabla hash como estructura de datos subyacente y por no mantener ningún orden especifico en las entradas.
- TreeMap. Implementa la interface SortedMap que extiende a su vez a la interface Map. Por tanto, una de sus características básicas es que mantiene ordenadas las entradas atendiendo al orden natural de las claves. Utiliza como estructura de datos subyacente un árbol AVL.
- LinkedHashMap que extiende a la clase HashMap. Además de la tabla hash, emplea una lista enlazada para proveer de un orden de iteración basado en dos posibles criterios: el orden de inserción o el orden en el que se accedió a los elementos por última vez. Por defecto se utiliza el primer criterio de ordenación, pero esta clase define un constructor que permite elegir cualquiera de los dos.

Las operaciones más relevantes que se declaran en la interface Map se resumen en la tabla siguiente:

MÉTODO	DESCRIPCIÓN
`void clear()`	Elimina todas las entradas del mapa.
`boolean containsKey(Object key)`	Retorna true si el mapa contiene un elemento asociado a la clave especificada.
`boolean containsValue(Object value)`	Retorna true si el mapa contiene el valor especificado asociado a una o más claves.
`Set<Map.Entry<K,V>> entrySet()`	Retorna el conjunto de entradas del mapa en forma de objetos Map.Entry, una clase interna definida en la clase Map.
`default void forEach(BiConsumer<? super K,? super V> action)`	Iteración interna sobre las entradas del mapa, procesando cada una con la expresión lambda especificada.
`V get(Object key)`	Retorna el valor asociado a la clave especificada o null si en el mapa no se asocia ningún valor con dicha clave.
`V getOrDefault(Object key, V defaultValue)`	Retorna el valor asociado a la clave especificada o el valor por defecto especificado si en el mapa no se asocia ningún valor con dicha clave.
`boolean isEmpty()`	Retorna true si el mapa está vacío.
`Set<K> keySet()`	Retorna el conjunto de claves contenidas en el mapa.

`V put(K key, V value)`	Añade una nueva entrada asociando la clave especificada al valor especificado y retorna null. Si la clave ya está asociada a otro valor, se sustituye el valor antiguo por el valor especificado retornando el valor antiguo.
`void putAll(Map<? extends K,? extends V> m)`	Añade a este mapa todas las entradas del mapa especificado. Cada entrada se añade tal y como lo haría el método put.
`V putIfAbsent(K key, V value)`	Si la clave especificada no se encuentra aún asociada a un valor (o está asociada al valor null), se asocia al valor especificado y retorna null. Si la clave especificada ya tiene un valor asociado, no se modifica la entrada correspondiente y se retorna dicho valor.
`V remove(Object key)`	Elimina la entrada clave/valor correspondiente a la clave especificada.
`boolean remove(Object key, Object value)`	Elimina la entrada correspondiente a la clave especificada sólo si está asociada al valor especificado y retorna `true`. Retorna false en caso contrario.
`V replace(K key, V value)`	Reemplaza el valor asociado a la clave especificada con el valor especificado sólo si dicha clave está asociada con algún valor. Retorna el valor antiguo se produce el reemplazo, o null en caso contrario.
`boolean replace(K key, V oldValue, V newValue)`	Reemplaza el valor asociado a la clave especificada con el valor nuevo especificado sólo si dicha clave está asociada con el valor antiguo especificado. Retorna el valor antiguo se produce el reemplazo, o `null` en caso contrario.
`int size()`	Retorna el número de entrada en el mapa.
`Collection<V> values()`	Retorna una colección con todos los valores que contiene el mapa.

3.1. Iterar sobre mapas

Al contrario que en las colecciones, en los mapas no existe un método que retorne un iterador ni la posibilidad de utilizar el bucle for mejorado para iterar sobre las entradas. Sin embargo, la clase Map permite llevar a cabo una iteración interna sobre las entradas a través del método forEach.

También es posible iterar sobre la colección de entradas, la de claves o la de valores, retornadas por los métodos entrySet, keySet y values respectivamente. A continuación, se muestran varios ejemplos de iteración sobre el mapa siguiente:

```
Map<Integer, String> mapa = new TreeMap<Integer, String>();
m.put(11111111,      "Elena Fernández");
m.put(11222333,      "Carlos González");
m.put(11444555,      "Rodrigo Rodíguez");
m.put(11321123,      "Elena Fernández");
m.put(11777999,      "Pilar Ramos");
```

Iteración sobre las entradas con un iterador:

```
Iterator<Entry<Integer, String>> i = mapa.entrySet().iterator();
while (i.hasNext()) {
    Entry<Integer, String> e = i.next();
    System.out.printf("NIF: %d - Nombre: %s\n", e.getKey(), e.getValue());
```

```
}
```

Iteración sobre las entradas con bucle `for` mejorado:

```
for (Entry<Integer, String> e: mapa.entrySet())
    System.out.printf("NIF: %d - Nombre: %s\n", e.getKey(), e.getValue());
```

Iteración interna sobre las entradas:

```
mapa.forEach((nif, nombre) -> System.out.printf("NIF: %d - Nombre: %s\n", nif, nombre));
```

Iteración sobre las claves con un iterador:

```
Iterator<Integer> i = mapa.keySet().iterator();
while (i.hasNext())
    System.out.printf("NIF: %d\n", i.next());
```

Iteración sobre las claves con bucle `for` mejorado:

```
for (Integer nif: mapa.keySet())
    System.out.printf("NIF: %d\n", nif);
```

Iteración interna sobre las claves:

```
mapa.keySet().forEach(nif -> System.out.printf("NIF: %d\n", nif));
```

Iteración sobre los valores con un iterador:

```
Iterator<String> i = mapa.values().iterator();
while (i.hasNext())
    System.out.printf("Nombre: %s\n", i.next());
```

Iteración sobre los valores con bucle `for` mejorado:

```
for (String nombre: mapa.values())
    System.out.printf("Nombre: %s\n", nombre);
```

Iteración interna sobre los valores:

```
mapa.values().forEach(nombre -> System.out.printf("Nombre: %s\n", nombre));
```

3.2. Ejemplos de problemas resueltos con mapas

Ejemplo 1: Programa que lee una línea de texto de la entrada estándar y contabiliza el número de veces que se repite cada carácter que no es un espacio en blanco. Muestra el resultado en la salida estándar.

Este problema se puede resolver mediante un mapa en el que las claves son los caracteres leídos y los valores son el número de veces que se repiten.

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Map;
import java.util.TreeMap;

public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Map<Character, Integer> mapa = new TreeMap<Character, Integer>();
        System.out.println("Texto:");
```

```
        String linea = br.readLine().toLowerCase();
        for (char c: linea.toCharArray()) {
            if (!Character.isSpaceChar(c)) {
                if (mapa.containsKey(c))
                    mapa.put(c, mapa.get(c) + 1);
                else
                    mapa.put(c, 1);
            }
        }
        System.out.println(mapa);
    }

}
```

Ejemplo 2: Programa que lea de la entrada estándar dos o más líneas de texto. La primera línea ha de contener el número de líneas que le siguen. A medida que va leyendo el resto de las líneas tendrá que ir almacenando, sin que se repitan, las palabras que contiene cada una de ellas en diferentes colecciones. Cada colección ha de contener palabras de una longitud determinada y se creará únicamente cuando se necesite. Las colecciones de palabras se han de almacenar a su vez en una estructura de datos que permita recuperarlas fácilmente. Para poner a prueba la estructura de datos creada, el programa leerá de la entrada estándar longitudes y mostrará la lista de palabras de cada una o null si no hay palabras de dicha longitud. Finalizará cuando se introduzca una longitud menor que 1.

Este problema se puede resolver con un mapa en el que las claves son una longitud de palabra y los elementos una colección de palabras de dicha longitud. En concreto, el mapa puede ser un HashMap (no se considera necesario que las entradas estén ordenadas por la clave) y la colección un TreeSet (para que se muestre la lista de palabras en orden alfabético).

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.Map;
import java.util.Scanner;
import java.util.TreeSet;
public class Main {

    public static void main(String[] args) throws IOException {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        Map<Integer, TreeSet<String>> mapa = new HashMap<Integer, TreeSet<String>>();
        System.out.println("Lineas:");
        int n = Integer.valueOf(br.readLine());
        for (int i=0; i<n; i++) {
            Scanner s = new Scanner(br.readLine());
            while (s.hasNext())
                agregarPalabra(mapa, s.next());
        }
        int longitud;
        do {
            System.out.println("Longitud:");
            longitud = Integer.valueOf(br.readLine());
            if (longitud > 0)
                System.out.println(mapa.get(longitud));

        } while (longitud > 0);
    }

    static void agregarPalabra(Map<Integer, TreeSet<String>> mapa, String palabra) {
        int longitud = palabra.length();
        TreeSet<String> set = mapa.get(longitud);
        if (set == null) {
            set = new TreeSet<String>();
            mapa.put(longitud, set);
        }
        set.add(palabra);
    }
```

```
}
```

4. Comparación de objetos

Java define los operadores == y != para comparar datos de tipos primitivos. Entre los tipos de datos primitivos están las referencias a objetos, que no se han de confundir con los objetos a los que hacen referencia, y para las cuales también están definidos estos operadores. Esto hace que sea frecuente cometer el error de usar estos operadores para comparar objetos, como se pone de manifiesto en los ejemplos siguientes usando referencias a objetos de tipo String.

Ejemplo 1:

```
String a = "texto 1";
String b = "texto 2";
System.out.println(a == b);
```

Salida:

```
false
```

La salida se corresponde con el resultado que cabría esperar al comparar dos cadenas distintas, "texto 1" y "texto 2".

Ejemplo 2:

```
String a = "texto 1";
String b = "texto 1";
System.out.println(a == b);
System.out.println(a != b);
```

Salida:

```
true
```

De nuevo, la salida se corresponde con el resultado que cabría esperar al comparar dos cadenas iguales.

Ejemplo 3:

```
String a = new String("texto 1");
String b = new String("texto 1");
System.out.println(a == b);
```

Salida:

```
false
```

En este caso, si esperábamos ver el resultado de comparar dos cadenas iguales, la salida de este último ejemplo debería ser la misma que en el ejemplo 2. Sin embargo, no se obtiene la misma salida.

Entonces, ¿cómo se explica lo ocurrido en el tercer ejemplo? La respuesta es que en ninguno de los tres ejemplos se han comparado las cadenas de caracteres. Lo que realmente se han comparado son las referencias a los objetos que las representan.

En el ejemplo 1 las referencias a y b apuntan a objetos diferentes, es decir, almacenan valores distintos.

En el ejemplo 2 las variables a y b apuntan al mismo objeto, es decir, tienen el mismo valor. Esto es así porque el compilador de Java crea un solo objeto para un mismo literal de tipo cadena, independientemente de a cuantas variables distintas se lo asignemos dentro de nuestro programa.

En el ejemplo 3 las referencias a y b apuntan a objetos diferentes que se crean como copia del literal "texto 1", es decir, a y b tienen valores distintos.

Entonces, ¿cómo se comprueba la igualdad entre objetos? Los diseñadores del lenguaje Java han previsto una solución para comparar objetos definiendo en la clase Object el método equals. Dado que todas las clases son descendientes de Object, se espera que cada nueva clase redefina este método para poder determinar la igualdad entre sus instancias. El método equals retornará el valor true cuando los objetos que se comparan son iguales y false en caso contrario.

Por ejemplo, la forma de comprobar la igualdad entre dos cadenas es:

```
String a = new String("texto 1");
String b = new String("texto 1");

/*
 *  dos formas equivalentes de comprobar si el objeto String
 *  referenciado por la variable a es igual al objeto String
 *  referenciado por la variable b
 */
System.out.println(a.equals(b));
System.out.println(b.equals(a));

/*
 *  dos formas equivalentes de comprobar si el objeto String
 *  referenciado por la variable a es distinto del objeto String
 *  referenciado por la variable b
 */
System.out.println(!a.equals(b));
System.out.println(!b.equals(a));
```

Salida:

```
true
true
false
false
```

Si una clase no redefine el método equals, al invocarlo para cualquiera de sus instancias, estaremos invocando la versión definida en la clase Object, cuya implementación es la siguiente:

```
public boolean equals(Object obj){
    return this == obj;
}
```

El resultado es el mismo que si utilizásemos el operador ==. En definitiva, el método equals tal y como está definido en la clase Object no cumple con el propósito para el cual fue concebido, que es el de verificar si dos instancias diferentes de una clase son iguales.

En general, diremos que dos instancias de una clase son iguales si sus atributos tienen el mismo valor, es decir, su estado es idéntico. Por ejemplo, supongamos que tenemos que crear una clase para representar los libros que hay en una biblioteca. Si cada libro se describe mediante un título y un autor, se entiende que dos instancias de la clase libro son iguales si tienen el mismo título y autor. Según esto, la definición de la clase Libro redefiniendo el método equals será la siguiente:

```
public class Libro {

    private String titulo;
    private String autor;
```

```
    /*
     *  se omite la definición de constructores y otros métodos
     */

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null || getClass() != obj.getClass())
            return false;
        Libro libro = (Libro) obj;
        return ejemplar == libro.ejemplar &&
               titulo.equals(libro.titulo) &&
               autor.equals(libro.autor);
    }
}
```

Dentro del método equals, el primer condicional sirve para que la comprobación de igualdad de una instancia de la clase Libro consigo misma resulte más eficiente. El segundo condicional sirve para evitar la excepción NullPointerException si comparamos un objeto Libro con el valor null, o la excepción ClassCastException, si comparamos instancias de diferentes clases.

También, podemos resumirlo todo en una única sentencia return:

```
    @Override
    public boolean equals(Object obj) {
        return this == obj || (obj != null &&
               getClass() == obj.getClass() &&
               ejemplar == ((Libro) obj).ejemplar &&
               titulo.equals(((Libro) obj).titulo) &&
               autor.equals(((Libro) obj).autor));
    }
```

Este ejemplo sirve si estamos seguros que a ningún atributo de tipo referencia se le va asignar el valor null en ningún momento. Si no es así, tendremos que redefiner el método equals teniendo en cuenta esa posibilidad para evitar la excepción NullPointerException. El entorno de desarrollo Eclipse es capaz de generar de forma automática la redefinición del método equals a través de la opción *Source → generate hashCode() and equals()…* del menú principal teniendo en cuenta esta posibilidad. Para la clase Libro redefine el método equals de la forma siguiente:

```
@Override
public boolean equals(Object obj) {
    if (this == obj)
        return true;
    if (obj == null)
        return false;
    if (getClass() != obj.getClass())
        return false;
    Libro other = (Libro) obj;
    if (autor == null) {
        if (other.autor != null)
            return false;
    } else if (!autor.equals(other.autor))
        return false;
    if (titulo == null) {
        if (other.titulo != null)
            return false;
    } else if (!titulo.equals(other.titulo))
        return false;
    return true;
}
```

4.1. El compromiso general entre el método equals y el método hashCode

La clase object define el método hashCode con el objetivo de que lo usen clases que implementen tablas hash, como por ejemplo la clase HashSet. Así pues, desde la perspectiva del API de Java, se espera que cualquier clase nueva redefina hashCode para que este tipo de estructuras de datos desempeñen su labor de forma eficiente. Pero además, también se espera que la redefinición de este método se haga respetando un compromiso general con el método equals en virtud del cual **dos instancias cualesquiera consideradas iguales por** equals **han de retornar el mismo código hash al invocar al método** hashCode. Por tanto, en virtud de este compromiso es posible descartar la igualdad entre dos instancias de una clase si retornan códigos hash distintos. Se ha de tener en cuenta que por la propia definición de función hash es posible que dos instancias consideradas distintas por equals retornen el mismo código hash. Esto hace que el método hashCode sólo se pueda usar para descartar la igualdad, pero no para verificarla.

Veamos un ejemplo que pone de manifiesto lo que ocurre cuando no se respeta este compromiso. Como ya sabemos, la colección HashSet representa un conjunto de objetos, y como tal, no debería de permitir objetos duplicados. Sin embargo, si intentamos añadir a un HashSet objetos de la clase Libro tal y como ha sido definida anteriormente, ocurre lo siguiente:

```
Set<Libro> libros = new HashSet<>();
libros.add(new Libro("Thinking in Java", "Bruce Eckel"));
libros.add(new Libro("Java - The Complete Reference", "Herbert Schildt"));
libros.add(new Libro("Thinking in Java", "Bruce Eckel"));
System.out.println(libros);
```

Salida:

```
[[Thinking in Java, Bruce Eckel], [Java - The Complete Reference, Herbert
Schildt], [Thinking in Java, Bruce Eckel]]
```

Aclaremos esto: en la documentación del método add de la clase HashSet se dice que un elemento e2 se añade al conjunto solamente si éste no contiene un elemento e1 tal que e1.equals(e2) retorna true, que es justo lo que ocurre cuando se intenta añadir el tercer libro, y aun así se añade. Esto, que parece contradecir a la documentación, no se debe a ningún error en la implementación del método add. Lo que ha ocurrido es que este método ha descartado la igualdad entre el primer libro y el tercero comparando sus códigos hash que resultan ser distintos, a pesar de que el método equals habría confirmado la igualdad entre ellos. Si los codigos hash hubiesen sido iguales, se habría terminado de verificar la igualdad invocado al método equals y el tercer libro no se habría añadido al conjunto.

La razón por la que el primer y tercer libro retornan códigos hash distintos reside en el hecho de que en la clase Libro no se ha redefinido el método hashCode y por tanto, el HashSet ha invocado al de la clase Object, que calcula el código hash como una representación numérica de la dirección de memoria en la que se encuentra el objeto. En consecuencia, al tratarse de instancias distintas de la clase `Libro`, ambas retornan códigos hash distintos cuando debiera ser al contrario (el método hashCode de la clase Object respeta el compromiso general con el método equals definido en ella, pero no con el que redefine la clase Libro).

El objetivo de este documento no es entrar en detalles acerca de cómo definir funciones hash, más aún teniendo en cuenta que los entornos de desarrollo son capaces de generar los métodos equals y hashCode de forma automática y respetando el compromiso general entre ellos. En el caso de Eclipse esto se lleva a cabo mediante la opción *Source → generate hashCode() and equals()…* del menú principal. Además, a partir de Java 7, es posible utilizar el método hash de la clase Objects para generar de forma sencilla códigos hash a partir de los atributos de una clase, por ejemplo:

```
@Override
public int hashCode() {
```

```
    return Objects.hash(titulo, autor);
}
```

Una vez redefinido el método hashCode en la clase Libro, podremos comprobar al ejecutar de nuevo el ejemplo anterior que el tercer libro ya no se añade al conjunto.

5. Ordenación

El API de Java da soporte a la ordenación de objetos almacenados en arrays y colecciones a través de clases como Arrays, que define métodos de clase para la ordenación de arrays que contegan tanto tipos de datos primitivos como referencias a objetos, o Collections, que define métodos de clase para la ordenación de listas. Ademas, algunas colecciones como TreeSet o TreeMap almacenan los objetos ordenados por defecto según su orden natural.

La definición de los operadores de comparación del lenguaje Java (>, <, >=, <=) para los tipos de datos primitivos (excepto boolean) establece el orden natural para los valores de sus respectivos dominios. Sin embargo, estos operadores no están definidos para el resto de tipos de datos, por lo que se hace necesario establecer para cada nueva clase un orden natural basado en los criterios de comparación que corresponda en cada caso, y poder usar así los recursos del API de Java que permiten llevar a cabo procesos de ordenación. De no ser así, cualquier intento de usar estos recursos desembocará en el lanzamiento de la excepción ClassCastException.

Por ejemplo, veamos lo que ocurre si intentamos añadir un objeto de la clase Libro, tal y como ha sido definida en los ejemplos anteriores, a un TreeSet:

```
import java.util.Set;
import java.util.TreeSet;

public class Main {

    public static void main(String[] args) {
        Set<Libro> libros = new TreeSet<>();
        libros.add(new Libro(1, "Efective Java", "Joshua Bloch"));
    }

}
```

Salida:

```
Exception in thread "main" java.lang.ClassCastException: class Libro cannot be
cast to class java.lang.Comparable …
  at java.base/java.util.TreeMap.compare(TreeMap.java:1563)
  at java.base/java.util.TreeMap.addEntryToEmptyMap(TreeMap.java:768)
  at java.base/java.util.TreeMap.put(TreeMap.java:777)
  at java.base/java.util.TreeMap.put(TreeMap.java:534)
  at java.base/java.util.TreeSet.add(TreeSet.java:255)
  at Main.main(Main.java:8)
```

5.1. Orden natural de una clase de objetos.

El orden natural de una clase de objetos se establece implementando en ella la interface Comparable<T> parametrizada con la propia clase. Esta interface declara un único método:

```
int compareTo (T o)
```

Se espera que el método compareTo retorne uno de los valores siguientes:

- Un entero negativo si el objeto con el que se invoca es menor que el objeto que se recibe como parámetro según el criterio de ordenación elegido.
- Cero si ambos objetos son iguales.
- Un entero positivo si el objeto con el que se invoca es mayor que el objeto que se recibe como parámetro según el criterio de ordenación elegido.

Esto hace que se mantenga la semántica de los operadores de comparación (<, >, <=, >=, ==). Por ejemplo, para expresar que el objeto a es menor que el objeto b usamos a.compareTo(b) < 0, que para nosotros tendrá el mismo significado que a < b.

Como ejemplo, vamos a establecer el orden natural para la clase Libro usando como criterio de comparación el valor de los atributos autor y titulo:

```
public class Libro implements Comparable<Libro> {

    private String titulo;
    private String autor;

    ⋮

    @Override
    public int compareTo(Libro libro) {
        if (this == libro)
            return 0;
        int resultado = autor.compareTo(libro.autor);
        if (resultado == 0)
            resultado = titulo.compareTo(libro.titulo);
        return resultado;
    }
}
```

A continuación vamos a ver dos ejemplos de procesos de ordenación basados en el orden natural establecido para la clase Libro.

Ejemplo 1. Almacenamiento de libros en un TreeSet:

```
Set<Libro> libros = new TreeSet<>();
libros.add(new Libro("Java - The Complete Reference", "Herbert Schildt"));
libros.add(new Libro("Thinking in Java", "Bruce Eckel"));
libros.add(new Libro("Efective Java", "Joshua Bloch"));
libros.add(new Libro("On Java 8", "Bruce Eckel"));
for (Libro libro: libros)
    System.out.println(libro);
```

Salida:

```
[On Java 8, Bruce Eckel]
[Thinking in Java, Bruce Eckel]
[Java - The Complete Reference, Herbert Schildt]
[Efective Java, Joshua Bloch]
```

Ejemplo 2. Ordenación de un array de libros:

```
Libro [] libros = {
    new Libro("Java - The Complete Reference", "Herbert Schildt"),
    new Libro("Thinking in Java", "Bruce Eckel"),
    new Libro("Efective Java", "Joshua Bloch"),
    new Libro("On Java 8", "Bruce Eckel")
};
```

```
Arrays.sort(libros);
for (Libro libro: libros)
    System.out.println(libro);
```

Salida:

```
[On Java 8, Bruce Eckel]
[Thinking in Java, Bruce Eckel]
[Java - The Complete Reference, Herbert Schildt]
[Efective Java, Joshua Bloch]
```

A la hora de establecer el orden natural cuando definimos nuevas clases, es muy recomendable que la implementación de Comparable sea consistente con el método equals.

> *Se dice que la implementación de la interface Comparable en una clase C es consistente con el método equals definido en ella si dadas dos instancias cualesquiera de C, c1 y c2, el resultado de evaluar la expresión c1.compareTo(c2) == 0 es el mismo que el de evaluar la expresión c1.equals(c2).*

Una buena razón para asegurar la consistencia entre ambos métodos, reside en el hecho de que algunas clases del API de Java en lugar de determinar la igualdad entre dos objetos con equals, lo hacen con compareTo. Esto no solamente es perfectamente lícito a todas luces, sino que a estas clases les resulta muy conveniente.

Veamos un Ejemplo. Supongamos que mantenemos el método equals de la clase Libro tal y como lo definimos la última vez, pero implementamos en ella la interface Comparable usando como único criterio de comparación el valor del atributo autor:

```
public class Libro implements Comparable {

    private String titulo;
    private String autor;

    ⋮

    @Override
    public int compareTo(Libro libro) {
        return this == libro ? 0 : autor.compareTo(libro.autor);
    }
}
```

Si ahora volvemos a ejecutar el ejemplo en el que almacenamos libros en un TreeSet ocurre lo siguiente:

```
Set<Libro> libros = new TreeSet<>();
Libro libro1 = new Libro("Java - The Complete Reference", "Herbert Schildt");
Libro libro2 = new Libro("Thinking in Java", "Bruce Eckel");
Libro libro3 = new Libro("Efective Java", "Joshua Bloch");
Libro libro4 = new Libro("On Java 8", "Bruce Eckel");
libros.add(libro1);
libros.add(libro2);
libros.add(libro3);
libros.add(libro4);
for (Libro libro: libros)
    System.out.println(libro);
```

Salida:

```
[Thinking in Java, Bruce Eckel]
[Java - The Complete Reference, Herbert Schildt]
[Efective Java, Joshua Bloch]
```

Como se puede observar, el último libro no se ha añadido al conjunto, debido a que el método add determinó que libro4 era igual a libro2 evaluando la expresión libro4.compareTo(libro2) == 0, que resulta ser true, mientras que la expresión libro4.equals(libro2) resulta ser false. En conclusión, es esta inconsistencia la culpable de que el método add de la clase TreeSet se haya comportado de esa forma.

Casi todas las clases que del API de Java que establecen un orden natural aseguran su consistencia con el método equals, a excepción de algunas como la clase java.math.BigDecimal.

Por último, es conveniente aclarar que la comparación difiere de la comprobación de igualdad en que el método compareTo no admite el valor null como parámetro, al contrario que el método equals. Si x es una referencia no nula a un objeto, entonces x.equals(null) retornará false, mientras que x.compareTo(null) debería lanzar la excepción NullPointerException.

5.2. Comparadores

Los comparadores permiten comparar objetos cuando no se establece un orden natural en la clase correspondiente o cuando se desea usar un criterio de ordenación diferente al orden natural establecido. El procedimiento de creación de un comparador consiste en la definición de una clase que implemente la interface Comparator<T> parametrizada con la clase de objetos que se van a comparar.

Esta interface declara el método compare de la forma siguiente:

```
int compare(T o1, T o2)
```

Se espera que el método compare retorne un de los valores siguientes:

- Un entero negativo si o1 es menor que o2 según el criterio de ordenación elegido.
- Cero si ambos objetos son iguales.
- Un entero positivo si o1 es mayor que o2 según el criterio de ordenación elegido.

Se habrá de tener en cuenta de nuevo la posible conveniencia de que la implementación de la interface Comparator sea consistente con el método equals definido en la clase de objetos que se desean comparar.

Por ejemplo, suponiendo que deseamos ordenar libros por título, y en caso de libros de diferentes autores que tengan el mismo título, por autor, podemos definir un comparador de la forma siguiente:

```
import java.util.Comparator;

public class LibroComparator implements Comparator<Libro> {
    @Override
    public int compare(Libro libro1, Libro libro2) {
        if (libro1 == libro2)
            return 0;
        int resultado = libro1.getTitulo().compareTo(libro2.getTitulo());
        if (resultado == 0)
            resultado = libro1.getAutor().compareTo(libro2.getAutor());
        return resultado;
    }
}
```

Se puede comprobar que el nuevo comparador es consistente con el método equals definido en la clase Libro.

A continuación, lo ponemos a prueba con un par de ejemplos que mostrarán la misma salida.

Ejemplo 1. Ordenación de un array de libros:

```
Libro [] libros = {
    new Libro("Java 8", "Thierry Groussard"),
    new Libro("Thinking in Java", "Bruce Eckel"),
    new Libro("Efective Java", "Joshua Bloch"),
    new Libro("Java 8", "Herbert Schildt")
};
Arrays.sort(libros, new LibroComparator());
for (Libro libro: libros)
    System.out.println(libro);
```

Ejemplo 2. Almacenamiento de libros en un TreeSet:

```
Set<Libro> libros = new TreeSet<>(new LibroComparator());
libros.add(new Libro("Java 8", "Thierry Groussard"));
libros.add(new Libro("Thinking in Java", "Bruce Eckel"));
libros.add(new Libro("Efective Java", "Joshua Bloch"));
libros.add(new Libro("Java 8", "Herbert Schildt"));
for (Libro libro: libros)
    System.out.println(libro);
```

Salida:

```
[Efective Java, Joshua Bloch]
[Java 8, Herbert Schildt]
[Java 8, Thierry Groussard]
[Thinking in Java, Bruce Eckel]
```