



# Colecciones

## Objetivos

- Implementar clases, interfaces y métodos con tipos genéricos.
- Conocer la interfaz Collection y sus métodos básicos y globales.
- Conocer la interfaz List y sus métodos.
- Utilizar las implementaciones de List: ArrayList y LinkedList.
- Conocer la interfaz Set.
- Usar las implementaciones de Set: HashSet, TreeSet y LinkedHashSet.
- Emplear las conversiones entre distintas implementaciones de List y Set.
- Conocer la interfaz Map.
- Usar los métodos de Map a través de sus implementaciones HashMap, TreeMap y linkedHashMap.
- Emplear las vistas Collection de los mapas.

## Contenidos

- 12.1. Tipos parametrizados o genéricos
- 12.2. Interfaz Collection
- 12.3. Métodos específicos de la interfaz List
- 12.4. Interfaz Set
- 12.5. Conversiones entre colecciones
- 12.6. Clase Collections
- 12.7. Interfaz Map

# Introducción

A menudo necesitamos guardar información, pero no sabemos de antemano el espacio que va a ocupar en la memoria. En estos casos, las tablas no son la solución adecuada, ya que su tamaño debe permanecer fijo una vez creadas. Al redimensionarlas, lo que hacemos es crear otras nuevas y copiar todos los datos de la antigua, con la sobrecarga que ello supone para la gestión de memoria. En su lugar, necesitamos estructuras dinámicas de datos, es decir, objetos que alberguen datos que se puedan insertar y eliminar, cambiando el tamaño de la estructura sin alterar los datos restantes, todo ello en tiempo de ejecución. Para este fin, Java nos proporciona una serie de estructuras dinámicas que comparten un conjunto de métodos declarados en la interfaz `Collection`. Todas ellas implementan dicha interfaz, aunque de distinta forma.

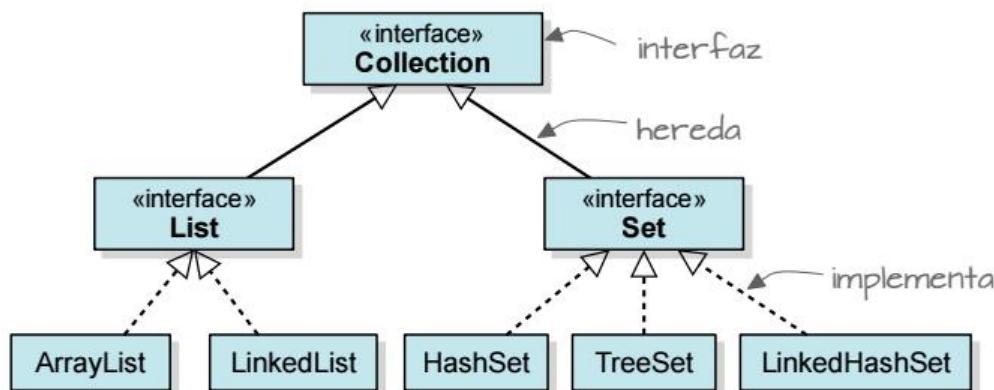


Figura 12.1. Estructuras de interfaces ligadas a colecciones.

Una colección es un objeto que sirve para agrupar un conjunto de objetos, llamados *elementos*, que generalmente guardan una relación entre ellos. Los métodos de las colecciones nos permiten llevar a cabo distintas operaciones con sus elementos, como la inserción, la eliminación, la búsqueda o la ordenación.

Hay colecciones de diferentes tipos, adaptadas a distintos fines más o menos específicos. Por eso no existe una clase colección, sino todo un marco de trabajo (framework), con una estructura jerárquica de interfaces (véase Figura 12.1) que se implementan en distintas clases, y con un conjunto de algoritmos que permiten la manipulación de los datos almacenados en las colecciones.

Los tipos fundamentales de estructuras dentro del framework de las colecciones son tres:

- **Listas:** responden a la necesidad de manejar sucesiones de datos que pueden estar repetidos y cuyo orden puede ser relevante. De alguna forma sustituyen a las tablas, con la diferencia de que podemos insertar o eliminar datos en ellas sin limitaciones de espacio. Implementan la interfaz `List` que, al heredar de `Collection`, incorpora todas sus funcionalidades y añade alguna más.
- **Conjuntos:** el orden de los datos no es relevante y lo que realmente importa es la mera pertenencia a la estructura, con lo que las repeticiones ni son posibles ni tienen sentido. Implementan la interfaz `Set`, que también hereda de `Collection`.

- **Mapas o diccionarios:** están dentro del framework de las colecciones, aunque no implementan la interfaz `Collection`. Sirven para guardar datos identificados por claves que no se repiten. Los mapas implementan la interfaz `Map`, que no hereda de `Collection`.

De todas estas estructuras, la más conocida y usada es la lista, aunque como vemos, no es la única.

Todas ellas son dinámicas, ya que permiten añadir y quitar unidades de información —objetos llamados *nodos* o *elementos*— en tiempo de ejecución, sin más límite que la memoria del ordenador. Nosotros llamaremos *colecciones* a todas las clases que implementen la interfaz `Collection`, aunque también usaremos el nombre del tipo especial de colección: lista o conjunto.

### Argot técnico



Hablamos del framework `Collection` como el entorno de trabajo formado por un conjunto de interfaces y clases relacionadas y que interactúan entre sí, pudiéndose obtener unas de otras según el modelo de datos que buscamos.

El conjunto de interfaces y clases del marco de trabajo `Collection` se halla en el paquete `java.util`.

## 12.1. Tipos parametrizados o genéricos

Una particularidad de `Collection` es que trabaja con tipos genéricos de datos. Por eso, antes de tratar las colecciones propiamente dichas, vamos a hacer una introducción a los tipos genéricos.

El uso de los tipos genéricos obedece a la necesidad de disponer de clases, interfaces o métodos que se puedan usar con muchos tipos de datos distintos, pero haciendo comprobaciones de tipo en tiempo de compilación. Ejemplos importantes son los métodos de comparación, `compareTo()` y `compare()`, que aparecen en las interfaces `Comparable` y `Comparator` respectivamente. Ambas interfaces están pensadas para comparar objetos de cualquier clase. De hecho, tendremos que implementar `Comparable` para cualquier clase de objetos que insertemos en cualquier tabla o colección que pretendamos ordenar. Por eso el método `compareTo()`, antes de que aparecieran los genéricos con Java 5, recibía como parámetro una variable de tipo `Object`, que es la clase más general de todas. Si tenemos una tabla de objetos `Cliente` que queremos ordenar por su DNI, implementamos la interfaz `Comparable` en la clase `Cliente` basándonos en su atributo `dni`. La interfaz, cuyo único elemento es el método `compareTo()`, antes de la aparición de los genéricos, tenía el prototipo

```
int compareTo(Object ob)
```

El parámetro de tipo `Object` garantizaba la total generalidad del método. En la implementación de `compareTo()` para la clase `Cliente`, como ya vimos en la Unidad 9, introducimos un cast `Cliente` delante de la variable `ob`.

```
int compareTo(Object ob) {
    return dni.compareTo(((Cliente)ob).dni);
}
```

El problema con este enfoque, aparte de la incomodidad de aplicar el cast, es que un valor del parámetro `ob` con tipo erróneo no se manifiesta durante la compilación, sino en tiempo de ejecución, lanzando una excepción. Esta es la razón para el uso de los tipos genéricos, que permiten la implementación con tipos tan generales como `Object`, pero comprobándolos y detectando sus errores antes de ejecutar el programa.

## 12.1.1. Clases con parámetros genéricos

Supongamos que queremos definir la clase `Contenedor`, que permita guardar un solo objeto de cualquier tipo. Los únicos métodos serán `guardar()` y `extraer()`. Habrá un único atributo `objeto` que, en principio, podría ser de tipo `Object`, para que el objeto para guardar pueda ser de cualquier clase. Pero así no tenemos control sobre el tipo del objeto guardado. Desde luego, siempre podríamos implementar una clase `Contenedor` para `Integer`, otra para `Double` y así sucesivamente. Pero si lo que queremos es una clase `Contenedor` que sirva para todo tipo de objetos y que, a la vez, permita controlar en cada caso ese tipo, tenemos que recurrir a los tipos genéricos. Una clase `Contenedor` con tipo genérico `T` podría ser:

```
class Contenedor<T> {
    private T objeto; //se inicializa a null: contenedor vacío
    public Contenedor() {
    }
    void guardar(T nuevo) {
        objeto = nuevo;
    }
    T extraer() {
        T res = objeto;
        objeto = null; //el contenedor vuelve a estar vacío
        return res;
    }
}
```

`T` representa el tipo de datos que se va a usar en la clase en cada declaración concreta, y tiene que ser una clase o interfaz, nunca un tipo primitivo.

Ahora podemos crear un `Contenedor` para enteros. La sintaxis es:

```
Contenedor<Integer> c = new Contenedor<Integer>();
```

El segundo `Integer` (el del lado derecho de la sentencia de asignación) puede ser omitido, ya que el compilador puede inferirlo a partir del lado izquierdo, con lo que pondremos

```
Contenedor<Integer> c = new Contenedor<>();
```

La expresión `<>` se llama *operador diamante*.

En este `Contenedor` vamos a guardar un 5 y luego lo volvemos a extraer y lo mostramos por consola.

```
c.guardar(5);
Integer n = c.extraer();
System.out.println(n); //aparece un 5 por consola
```

El compilador comprueba el tipo del valor que pasamos al método `guardar()`, que tiene que ser `Integer`. Si hubiéramos pasado el valor 7.4 o la cadena «silla», habría dado un error en la compilación. También hace una comprobación de tipos en la asignación a la variable `n`, que se ha declarado `Integer`. Si hubiéramos implementado la clase `Contenedor` con una variable `Object` en vez de un tipo genérico, habríamos tenido que poner un cast `Integer` delante de `c.extraer()`, y si el objeto devuelto fuera de tipo distinto a `Integer`, el error se habría producido durante la ejecución del programa.

La misma clase nos sirve para crear un `Contenedor` de números reales

```
Contenedor<Double> c1 = new Contenedor<>();
```

o de clientes

```
Contenedor<Cliente> c2 = new Contenedor<>();
```

En general, para definir una clase con un tipo genérico `T`, se escribe `<T>` después del nombre de la clase.

```
class nombreClase<T> {
    ...
}
```

Se suele usar la letra `T` para el tipo genérico, pero puede ser cualquier otra, aunque es costumbre reservar `E` para elementos de colecciones, `K` para claves, `V` para valores o `N` para números.

En la implementación de una clase puede intervenir más de un tipo genérico `U, V...`. En ese caso, se especifican los parámetros separados por comas.

```
class nombreClase<U, V...> {
    ...
}
```

Las clases definidas con tipos genéricos, como nuestro `Contenedor`, también pueden usarse sin ellos, en cuyo caso el compilador trabaja por defecto con variables de tipo `Object`. Eso significa que no hace comprobaciones de tipos y se pueden guardar objetos de distintas clases mezclados. Podemos escribir:

```
Contenedor c = new Contenedor();
c.guardar(7); //el atributo objeto guarda un 7 como Object
c.guardar("roca"); //ahora guarda la cadena "roca" como Object
```

Este código compilará sin problema, incluso si añadimos

```
Double x = (Double) c.extraer();
```

aunque sabemos que el `Object` extraído es una cadena y no un `Double`. El compilador se limita a darnos un aviso de que estamos realizando operaciones sin comprobación de tipo. El error se producirá al ejecutar el programa, con una excepción `ClassCastException` al aplicar el cast.

Muchos métodos de clases o interfaces de `java.lang`, como `compareTo()` o `compare()`, que actualmente están implementados con tipos genéricos, siguen soportando la implementación antigua con parámetros `Object`, aunque debe evitarse cuando sea posible, ya que eso sería renunciar a la ventaja de los tipos genéricos, que es la comprobación en tiempo de compilación de los tipos de los datos pasados como parámetro a las funciones o asignados a las variables.

## ■■■ 12.1.2. Interfaces con genéricos

También se pueden definir interfaces con tipos genéricos y la sintaxis es idéntica,

```
interface nombreInterfaz<T> {
    ...
}
```

Como ya hemos comentado, un ejemplo de interfaz con tipo genérico, definida de esta forma desde Java 5 y presente en `java.lang`, es la interfaz `Comparable`.

```
public interface Comparable<T> {
    int compareTo(T o);
}
```

Si queremos implementarla en la clase `Cliente` para que tenga un orden natural basado en el DNI, escribiremos,

```
public class Cliente implements Comparable<Cliente> {
    String dni;
    String nombre;
    ...
    public int compareTo(Cliente o) {
        return dni.compareTo(o.dni);
    }
}
```

donde se ha prescindido del cast que aparecía en la versión antigua. Otra interfaz que ha sido redefinida con tipos genéricos es `Comparator`.

```
public interface Comparator<T> {
    int comparable(T o1, T o2);
}
```

Como ejemplo de esta última, vamos a implementar una clase comparadora para ordenar los clientes por orden alfabético de nombres.

```
class ComparaNombres implements Comparator<Cliente> {
    public int compare(Cliente o1, Cliente o2) {
        return o1.nombre.compareTo(o2.nombre);
```

```
}
```

Cuando queremos invertir el criterio de orden natural de una clase `T` que implemente la interfaz `Comparable`, podemos implementar una clase comparadora nosotros mismos o bien extraerla de `T` por medio del método estático `naturalOrder()` de la interfaz `Comparator`. Por ejemplo, para conseguir un comparador con el criterio de orden de la clase `Integer`

```
Comparator<Integer> ordenInteger = Comparator.naturalOrder();
```

Java infiere del lado izquierdo la clase (el tipo `Integer`) de la que debe extraer el criterio de ordenación.

A partir de `ordenInteger` se puede obtener el criterio de ordenación inverso para `Integer`.

```
Comparator<Integer> ordenIntegerInverso = ordenInteger.reversed();
```

### ■ ■ ■ 12.1.3. Parámetros genéricos limitados

La implementación de una clase con el tipo genérico `T` implica que, en sus métodos, se van a realizar operaciones con variables de dicho tipo. Pero, a veces, dichas operaciones solo tienen sentido para determinados tipos de `T`. Por ejemplo, si aparece alguna operación aritmética entre valores de tipo `T`, sabemos que este no puede ser `String` o `Cliente`. Para casos así, existen los tipos genéricos limitados. La idea es que se limiten los posibles tipos de `T` a una determinada clase `claseLímite` y todas sus subclases (si `claseLímite` es un límite superior) o todas sus superclases (si `claseLímite` es un límite inferior). En el primer caso,

```
class nombreClase<T extends claseLímite> {  
    ...  
}
```

`<T extends claseLímite>` significa que `T` puede ser `claseLímite` o cualquiera de sus subclases.

O bien

```
class nombreClase<T super claseLímite> {  
    ...  
}
```

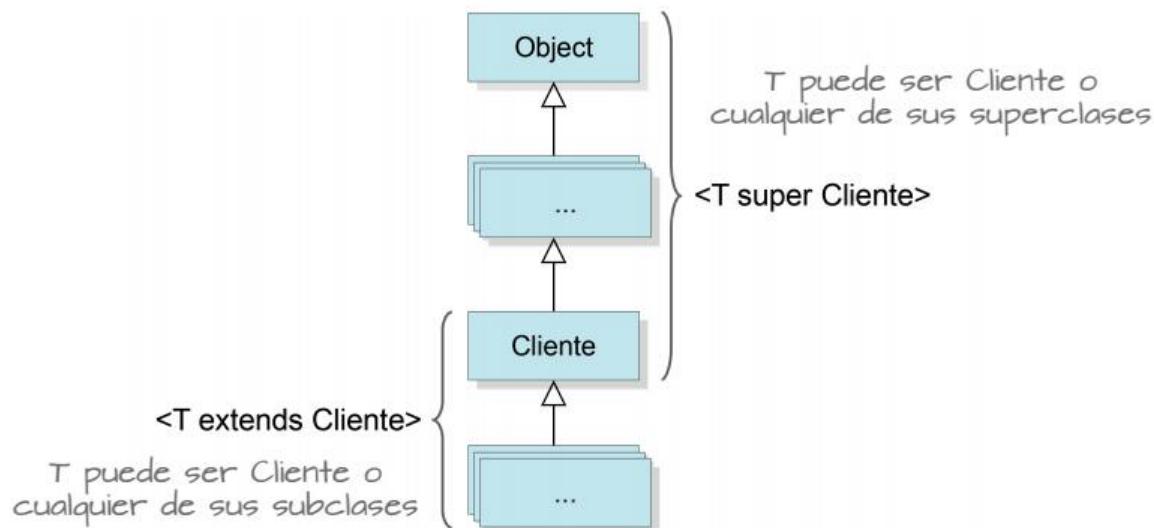
`<T super claseLímite>` significa `claseLímite` y todas sus superclases.

#### Argot técnico



Hablamos de límite superior de un parámetro genérico cuando el rango de los tipos admitidos en una declaración incluye al tipo límite y todos los subtipos.

Hablamos de límite inferior, cuando incluye al tipo límite y todos los supertipos.



**Figura 12.2.** Representación de los límites inferior y superior de la clase Cliente.

Como ejemplo, supongamos que queremos implementar la clase `Calculadora` con el tipo genérico `T`, donde se realizarán operaciones aritméticas. En este caso debemos limitarnos a las clases envoltorio que heredan de la clase abstracta `Number` (`Integer`, `Double`...). La clase `Calculadora` sería de la forma:

```

class Calculadora<T extends Number> {
    T a, b;
    //operaciones con a y b
}
    
```

Cuando declaremos una variable o usemos el constructor de una clase con tipo genérico, el compilador comprobará si el tipo declarado está dentro de los límites del parámetro genérico que aparece en la definición de la clase. Por ejemplo, la sentencia

```
Calculadora<Double> c = new Calculadora<>();
```

será correcta, ya que `Double` es una subclase de `Number`.

En cambio, se producirá un error de compilación con

```
Calculadora<Object> c = new Calculadora<>();
```

ya que `Object` no hereda de `Number`.

También se pueden limitar los tipos genéricos a aquellos que implementan una o más interfaces. En este caso no se usa la palabra `implements`, sino `extends`, como si fuera una herencia. Esta es una particularidad exclusiva de la sintaxis de los parámetros genéricos.

Por ejemplo, si estamos definiendo una clase `MiClase` con un parámetro genérico `T`, que sea válido solo para valores que tengan implementada la interfaz `MiInterfaz`, escribiríremos

```

class MiClase<T extends MiInterfaz> { //no ponemos implements!
    ...
}
```

## ■ ■ ■ 12.1.4. Métodos genéricos

Los parámetros genéricos de una clase o interfaz suelen aparecer en los métodos implementados dentro de ella. Por ejemplo, en los métodos `guardar()` y `extraer()` de la clase `Contenedor`, aparece el parámetro `T`. Sin embargo, dentro de cualquier clase, tanto si está definida con tipos genéricos como si no, podemos implementar métodos con sus propios parámetros genéricos, distintos de los que pueda tener la clase. Dichos métodos se llaman *métodos genéricos*.

Como ejemplo, vamos a implementar un método que nos devuelve el número de elementos `null` que hay en una tabla que se le pasa como argumento. El tipo `U` de la tabla es genérico, y se declara en la definición del método, justo antes del tipo devuelto.

```
static <U> int numeroDeNulos(U[] tabla) {
    int cont = 0;
    for (U e : tabla) {
        if (e == null) {
            cont++;
        }
    }
    return cont;
}
```

Este método se puede incluir en cualquier clase y no depende para nada de los parámetros propios de ella.

El tipo asociado a un método genérico también puede estar limitado. Por ejemplo, si quisieramos que nuestro método `numeroDeNulos()` solo funcionara para tablas numéricas, pondríamos `<U extends Number>` en vez de `<U>`.

### Actividad resuelta 12.1

Implementar un método genérico estático que realice la inserción de un objeto al final de una tabla, ambos del mismo tipo genérico, que se pasan como parámetros. Devuelve una nueva tabla con el resultado de la inserción.

#### Solución

```
static <E> E[] guardar(E elem, E[] tabla) {
    E[] nuevaTabla = Arrays.copyOf(tabla, tabla.length + 1);
    nuevaTabla[nuevaTabla.length - 1] = elem;
    return nuevaTabla;
}
/*Programa principal para probarlo. Insertamos dos cadenas en una tabla y
la mostramos*/
String cadenas[] = {};//o bien new String[0]
System.out.println(Arrays.toString(cadenas));
cadenas = guardar("coche", cadenas);
cadenas = guardar("avión", cadenas);
System.out.println(Arrays.toString(cadenas));
```

## Actividad propuesta 12.1

Implementa un método genérico estático al que se le pasan como parámetro dos tablas con elementos del mismo tipo genérico y devuelve una nueva tabla con los elementos de ambas concatenados (los de la segunda después de los de la primera).

### 12.1.5. Comodines

Los **comodines** —o *wildcards*— se suelen usar en la declaración de atributos, variables locales o parámetros pasados a una función. Un comodín se representa con el símbolo «?», que significa cualquier tipo. Por ejemplo, si escribimos

```
Contenedor<?> c;
```

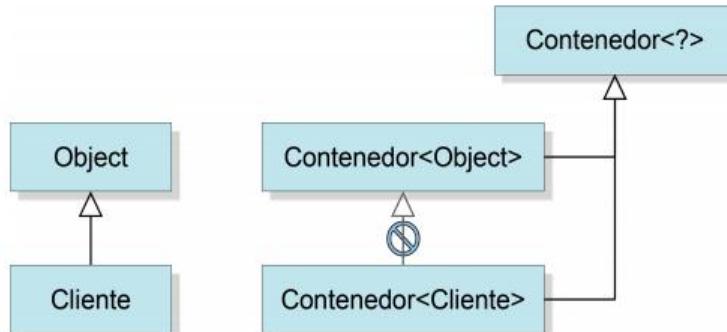
hemos declarado una variable `c` de tipo `Contenedor` cuyo parámetro genérico asociado puede ser cualquiera. La variable `c` puede referenciar un `Contenedor` de `Integer`, de `String` o de cualquier otra clase. Esto significa que todos los objetos `Contenedor` pertenecen a alguna subclase de `Contenedor<?>`.

Aquí tenemos que llamar la atención sobre un error común cuando se usan los tipos genéricos. El hecho de que `Integer` sea subclase de `Number` no implica que `Contenedor<Integer>` sea subclase de `Contenedor<Number>`. De igual modo `Contenedor<Cliente>` no es subclase de `Contenedor<Object>`. La relación de herencia entre los valores del parámetro genérico no tiene nada que ver con la relación entre las instancias de la clase `Contenedor`.

Por ejemplo, la sentencia

```
Contenedor<Object> c0bj = new Contenedor<Integer>(); //!Error!
```

producirá un error de compilación, a pesar de que `Integer` hereda de `Object`. Si queremos una variable `c` de tipo `Contenedor` que pueda referenciar a cualquier objeto `Contenedor`, tendremos que usar `Contenedor<?> c`.



**Figura 12.3.** La relación de herencia entre clases usadas como tipos genéricos en otras clases (como las colecciones) no implica que estas últimas mantengan la misma relación de herencia.

Con el comodín también se puede usar la clase límite superior `<? extends T>`, que significa cualquier clase que herede de `T`, incluyendo a esta. E inferior `<? super T>`, que significa `T` o cualquier superclase de `T`.

Cuando escribimos

```
Contenedor<? extends Number> c;
```

estamos declarando una variable de tipo Contenedor de clase numérica. Puede referenciar un objeto Contenedor<Number>, Contenedor<Integer>, Contenedor<Double>, etc. Todas ellas son subclases de Contenedor<? extends Number>.

Ejemplos con límite inferior se presentan con frecuencia en los comparadores, como veremos a lo largo de la unidad.

## ■ ■ ■ 12.1.6. Cosas que no se pueden hacer con parámetros genéricos

A pesar de todas las aportaciones de las clases genéricas a la programación en java, por la forma en que estas han sido implementadas, tienen una serie de limitaciones, algunas de las cuales puede que sean subsanadas en el futuro. En concreto, hay varias operaciones que, de momento, están prohibidas. Las más importantes son:

- Los tipos genéricos nunca pueden ser primitivos. Cuando estos hagan falta, usaremos sus correspondientes clases envoltorio Integer, Character...
- No se pueden crear instancias de tipo genérico, como en new T();
- No se pueden crear tablas de tipos genéricos, como en new T[10]. Cuando hagan falta para un tipo concreto, se pasan como argumento al método donde van a ser usadas, que puede ser un constructor, o deberán ser devueltos por algún método definido fuera de la clase. Las tablas genéricas siempre tienen que venir construidas fuera de nuestra clase, interfaz o método genérico.
- Tampoco se pueden crear tablas de clases parametrizadas, como  

```
new Contenedor<Integer>[5];
```
- No se pueden usar excepciones genéricas.

### Actividad resuelta 12.2

Implementar, con tipos genéricos, la clase Contenedor, donde podremos guardar tantos objetos como deseemos. Para ello utilizaremos una tabla, que inicialmente tendrá tamaño cero y se irá redimensionando según añadamos o eliminemos elementos. La clase, además del constructor y `toString()`, tendrá los siguientes métodos:

- void insertarAlPrincipio(T nuevo)
- void insertarAlFinal(T nuevo)
- T extraerDelPrincipio()
- T extraerDelFinal()
- void ordenar()

#### Solución

```
/*El tipo T debe tener implementada la interfaz Comparable  
para que se pueda ordenar la tabla*/
```

```
class Contenedor<T extends Comparable<T>> {
    private T[] objetos;
    /* Como no se puede instanciar una tabla de tipo genérico, para cada caso
particular deberá crearse fuera del constructor y se le pasa como parámetro*/
    public Contenedor(T[] objetos) {
        this.objetos = objetos;
    }
    void insertarAlFinal(T nuevo) {
        objetos = Arrays.copyOf(objetos, objetos.length + 1);
        objetos[objetos.length - 1] = nuevo;
    }
    void insertarAlPrincipio(T nuevo) {
        objetos = Arrays.copyOf(objetos, objetos.length + 1);
        /*desplazamos todos los elementos un lugar hacia el final para hacer un hueco
al principio: */
        System.arraycopy(objetos, 0, objetos, 1, objetos.length - 1);
        objetos[0] = nuevo;
    }
    T extraerDelFinal() {
        T res = null;
        if (objetos.length > 0) {//si la tabla no está vacía
            res = objetos[objetos.length - 1];
            objetos = Arrays.copyOf(objetos, objetos.length - 1);
        }
        return res;
    }
    T extraerDelPrincipio() {
        T res = null;
        if (objetos.length > 0) {
            res = objetos[0];
            objetos = Arrays.copyOfRange(objetos, 1, objetos.length);
        }
        return res;
    }
    void ordenar() {
        Arrays.sort(objetos);
    }
    public String toString() {
        return Arrays.deepToString(objetos);
    }
}
/*Código para probar la clase: */
public static void main(String[] args) {
    Contenedor<Integer> c = new Contenedor<>(new Integer[0]);
    for (int i = 0; i < 20; i++) {
        c.insertarAlFinal((int) (Math.random() * 20));
    }
    System.out.println("Sin ordenar: " + c);
    c.ordenar();
    System.out.println("Ordenado: " + c);
    Integer n = c.extraerDelPrincipio();
    System.out.println("Elemento extraído: " + n);
    System.out.println("Después de extraer: " + c);
}
```

## Actividad resuelta 12.3

Definir la interfaz `Pila` con parámetros genéricos. A continuación, implementar la interfaz `Pila` genérica en la clase `Contenedor`. Por último, escribir un programa donde se utilice un objeto contenedor como pila. En ella apilamos números enteros positivos leídos del teclado hasta que se introduzca un `-1`. Después, mediante un bucle, se desapilan todos los números hasta que la pila esté vacía y los mostramos por consola.

### Solución

```
/*Interfaz Pila: */
interface Pila<T> {
    void apilar(T nuevo);
    T desapilar();
}

/*Clase Contenedor implementando Pila*/
class Contenedor<T> implements Pila<T> {
    private T[] objetos;
    public Contenedor(T[] objetos) {
        this.objetos = objetos;
    }
    /*Resto de la implementación de Contenedor de la actividad anterior*/
    /*Implementación de Pila:*/
    @Override
    public void apilar(T nuevo) {
        this.insertarAlPrincipio(nuevo);
    }
    @Override
    public T desapilar() {
        return this.extraerDelPrincipio();
    }
}
/*Programa principal: Utilizamos un objeto Contenedor como Pila. Esto es posible
porque Contenedor implementa dicha interfaz. Como la variable es de tipo Pila,
los miembros accesibles son apilar() y desapilar(), es decir, el contenedor se
comporta como una Pila*/
Pila<Integer> p = new Contenedor<>(new Integer[0]);
Scanner sc = new Scanner(System.in);
System.out.print("Introducir entero positivo (-1 para terminar): ");
Integer n = sc.nextInt();
while (n != -1) {
    p.apilar(n);
    System.out.print("Introducir entero positivo (-1 para terminar): ");
    n = sc.nextInt();
}
System.out.print("Desapilamos: ");
n = p.desapilar();
while (n != null) {
    System.out.print(n + " ");
    n = p.desapilar();
}
System.out.println("");
```

## Actividad propuesta 12.2

Define la interfaz Cola con parámetros genéricos. A continuación, implementa la interfaz Cola genérica en la clase Contenedor (no hace falta suprimir la implementación de Pila de la Actividad resuelta 12.3). Por último, escribe un programa donde se utilice un objeto Contenedor como cola. En ella encolamos números enteros positivos leídos del teclado hasta que se introduzca un -1. Después, mediante un bucle, se desencolan todos los números hasta que la cola esté vacía y los mostramos por consola.

## 12.2. Interfaz Collection

La interfaz `Collection` define las funcionalidades comunes a todas las colecciones, ya sean listas o conjuntos. Sin embargo, las clases de la API que la implementan son listas o conjuntos, es decir, implementan la interfaz `List` o `Set`, que son extensiones de `Collection`. Ninguna implementa `Collection` directamente. Por ello, para poner ejemplos prácticos de ella, tendremos que usar listas o conjuntos. Nosotros usaremos listas, para lo cual adelantaremos aquí los conocimientos mínimos para crearlas y manipularlas como colecciones, aunque el estudio detallado de la interfaz `List` se hará más adelante.

### Argot técnico



Llamaremos *colección* a toda instancia de alguna de las clases que implementan la interfaz `Collection`. Estas incluyen: `ArrayList`, `LinkedList`, `HashSet`, `TreeSet` y `LinkedHashSet`. Los mapas (`HashMap`, `TreeMap` y `LinkedHashMap`) no son colecciones, aunque guardan relación con ellas.

### 12.2.1. Breve presentación de las listas

Las listas son clases que implementan la interfaz `List`, y sirven para almacenar datos que se pueden repetir y cuyo orden de inserción puede ser relevante. Hay dos implementaciones de `List`, las clases `ArrayList` y `LinkedList`. Las dos proporcionan los mismos métodos y funcionalidades. La diferencia entre `ArrayList` y `LinkedList` radica en la implementación interna y solo afecta levemente al rendimiento. La primera es más rápida en las operaciones que supongan recorrer la lista para la lectura o modificación de elementos, mientras que la segunda tiene mejor rendimiento en las operaciones de inserción y eliminación de elementos.

Nosotros usaremos la primera en nuestros ejemplos, aunque la segunda nos valdría exactamente igual. Por tanto, en todo el código que vamos a escribir, podemos sustituir `ArrayList` por `LinkedList` sin alterar ningún resultado.

La sintaxis para construir un objeto `ArrayList` con un tipo genérico de datos `E` (`Cliente`, `Empleado`, `Integer`...) es:

```
ArrayList<E> lista = new ArrayList<E>();
```

O bien, de forma más general, dado que tanto la clase `ArrayList` como `LinkedList` implementan todos los métodos de la interfaz `List`, es posible utilizar una variable de este tipo para referenciar objetos de ambas clases.

```
List<E> lista = new ArrayList<E>();
```

En esta lista solo se podrán insertar objetos (elementos) del tipo `E`.

La construcción de un `ArrayList` que nos sirva para guardar objetos de tipo `Cliente`, será:

```
List<Cliente> listaClientes = new ArrayList<Cliente>();
```

Una vez creada la colección `listaClientes`, disponemos de una estructura dinámica donde insertar o eliminar objetos `Cliente`. Antes vamos a definir una clase `Cliente` que nos permita probar los distintos métodos con ejemplos concretos:

```
class Cliente implements Comparable<Cliente> {
    String dni;
    String nombre;
    LocalDate fechaNacimiento;
    Cliente(String dni, String nombre, String fechaNacimiento) {
        this.dni = dni;
        this.nombre = nombre;
        DateTimeFormatter formatoFechas=
            DateTimeFormatter.ofPattern("dd/MM/yyyy");
        this.fechaNacimiento = LocalDate.parse(fechaNacimiento, formatoFechas);
    }
    int edad(){
        return (int)fechaNacimiento.until(LocalDate.now(), ChronoUnit.YEARS);
    }
    @Override
    public boolean equals(Object ob) {
        return dni.equals(((Cliente) ob).dni);
    }
    @Override
    public int compareTo(Cliente otro) {
        return dni.compareTo(otro.dni);
    }
    @Override
    public String toString() {
        return "DNI: " + dni + " Nombre: " + nombre + " Edad: " + edad() + "\n";
    }
}
```

Para estudiar la interfaz `Collection`, dado que la interfaz `List` hereda de ella, vamos a referenciar la lista `listaClientes` con la variable `colecciónClie` del tipo `Collection` (véase la Figura 12.4). Con ello conseguiremos tener acceso únicamente a las funcionalidades de `Collection`, que son las que vamos a estudiar a continuación, y no a las específicas de la interfaz `List`, que estudiaremos después.

```
Collection<Cliente> colecciónClie = listaClientes;
```

Mientras usemos la variable `colecciónClie`, el objeto referenciado (una lista) se comportará como una simple colección, sin ninguna de las particularidades específicas de las listas (véase Apartado 9.5, en la unidad referente a interfaces).



Figura 12.4. Representación de las referencias de variables de distintos tipos relacionados mediante herencia.

## ■ ■ ■ 12.2.2. Métodos básicos de la interfaz Collection

Los métodos de la interfaz `Collection` son de dos tipos. Unos afectan a elementos individuales y otros a grupos de elementos. A los primeros los llamaremos *métodos básicos* y a los segundos, *métodos globales*. En primer lugar, nos ocuparemos de los básicos.

### ■ ■ ■ Método de inserción

Es aquel que sirve para añadir elementos nuevos en una colección.

- `boolean add(E elem)`: se le pasa el objeto que se va a insertar. Si la inserción tiene éxito, devuelve `true`. En caso contrario, `false`. En general, es común que un método devuelva `true` cuando, al ejecutarse, cambia la estructura de una colección y `false` si la colección queda inalterada. Ya veremos otros ejemplos. Si la colección es una lista, el nuevo elemento siempre se insertará, y además lo hará al final. En cambio, como veremos más adelante, con los conjuntos será distinto.

Como hemos declarado la colección con un tipo genérico, el compilador no nos va a permitir insertar un objeto de otro tipo que no sea el declarado, en nuestro caso `Cliente`.

```
Cliente cliente = new Cliente("111", "Marta", "12/02/2000");
colecciónClie.add(cliente);
```

### ■ ■ ■ Métodos de eliminación

- `boolean remove(Object ob)`: elimina un elemento `ob` de una colección. Si está repetido, elimina solo el primero que encuentra. Devuelve `true` si la eliminación ha tenido éxito y `false` en caso contrario, por ejemplo, si el objeto no estaba en la colección. Por otra parte, se puede observar que no se exige a `ob` que sea del tipo genérico `E` con el que se ha declarado la colección. Esto se debe a que el método no va a añadir ningún elemento, y no hay peligro de que se inserte un objeto de una clase no permitida. Para eliminar a Marta de nuestra colección escribiremos,

```
colecciónClie.remove(cliente);
```

`void clear()`: nos permite eliminar todos los elementos de una colección y dejarla vacía. Esto no significa eliminar la propia colección, del mismo modo que vaciar una bolsa de caramelos no significa destruir la bolsa. La colección, simplemente, queda vacía y disponible para volver a insertar nuevos elementos.

```
colecciónClie.clear();
```

## Métodos de comprobación

Nos permiten comprobar el estado de una colección. Como hemos dejado la colección vacía, vamos a empezar insertando algunos elementos para seguir experimentando:

```
colecciónClie.add(new Cliente("111", "Marta", "12/02/2000"));  
colecciónClie.add(new Cliente("115", "Jorge", "16/03/1999"));  
colecciónClie.add(new Cliente("112", "Carlos", "01/10/2002"));
```

- `int size()`: nos permite saber, en cada momento, el número de elementos (o nodos) insertados en una colección. Por ejemplo,

```
colecciónClie.size(); //devuelve 3
```

- `boolean isEmpty()`: permite saber si una colección está vacía. Devuelve `true` si está vacía y `false` en caso contrario.

```
colecciónClie.isEmpty(); //devolverá false
```

- `boolean contains(Object ob)`: nos dice si un elemento `ob` determinado está en una colección. Devuelve `true` si `ob` pertenece a la colección y `false` en caso contrario. En nuestro ejemplo,

```
colecciónClie.contains(new Cliente("115", "Jorge", "16/03/1999")); //true
```

En la búsqueda del objeto `ob` (llamado *clave de búsqueda*), `contains()` usa el método `equals()` para compararlo con los elementos de la colección. En nuestro ejemplo con objetos `Cliente`, ese método está basado en el atributo `dni`. Por tanto, la expresión

```
colecciónClie.contains(new Cliente("115", "", ""));
```

también devolverá `true`, ya que, en la búsqueda del objeto, el programa solo va a comparar el atributo `dni` de la clave de búsqueda con los de los distintos elementos de la colección. Esto nos permite hacer búsquedas sin conocer ni tener que escribir toda la información de un elemento. Basta conocer el o los atributos que usa el método `equals()`, en nuestro ejemplo el DNI del cliente. Si tenemos que hacer muchas búsquedas puede ser útil implementar un constructor de `Cliente` con el DNI como único parámetro. Así, la línea de código anterior quedaría

```
colecciónClie.contains(new Cliente("115"));
```

que resulta más cómoda para el programador.

## Otros métodos

- `String toString()`: devuelve una cadena que representa la colección. Todas las colecciones tienen implementado este método, que muestra los elementos entre

corchetes y separados por comas. Cada elemento se muestra, a su vez, según la implementación de `toString()` que tenga la clase `E`, en nuestro caso la clase `Cliente`. Por tanto, para mostrar los elementos de `colecciónClie`, podemos escribir

```
System.out.println(colecciónClie);
```

Con nuestra implementación de `Cliente`, obtendríamos por pantalla

```
[DNI: 111 Nombre: Marta Edad: 20  
, DNI: 115 Nombre: Jorge Edad: 21  
, DNI: 112 Nombre: Carlos Edad: 18  
]
```

donde las edades pueden diferir, ya que se calculan a partir de la fecha de nacimiento y dependen de cuándo se ejecute el programa. Por otra parte, los datos de los elementos aparecen en líneas distintas porque pusimos un carácter «\n» al final de la cadena devuelta por `toString()` en la clase `Cliente`.

A menudo necesitamos recorrer una colección elemento a elemento. Una de las formas de hacerlo es por medio de iteradores, que son objetos que van apuntando sucesivamente a los elementos de la colección, empezando por el primero. Para usar un iterador con una colección concreta, primero hay que crearlo. El método

- `Iterator<E> iterator()`: invocado por la colección, nos devuelve un iterador asociado a ella. Aquí `E` será del mismo tipo que el de la colección.

Por ejemplo, si queremos recorrer nuestra colección de clientes, creamos el iterador con la sentencia

```
Iterator<Cliente> it = colecciónClie.iterator();
```

`it` es el iterador que sirve para recorrer `colecciónClie`. Para ello dispone de los métodos `hasNext()` y `next()`, que se complementan y emplean conjuntamente. Inicialmente `it` apunta al principio de la colección, justo antes del primer elemento (véase Figura 12.5).



Figura 12.5. Posición inicial del iterador.

- `boolean hasNext()`: comprueba si hay un elemento siguiente, es decir, si quedan elementos por visitar, y nos devuelve `true` o `false`, según el caso.
- `E next()`: el iterador avanza al siguiente elemento, si existe, y nos lo devuelve. En caso de que no haya siguiente, porque estemos al final de la colección o porque esta estuviera vacía, `next()` lanzará la excepción `NoSuchElementException`. La primera llamada a `next()` nos devuelve el primer elemento de la colección si esta no está vacía (Figura 12.6).

En la práctica, para evitar la excepción, los dos métodos se usan conjuntamente, de forma que solo se llama al método `next()` si antes se ha comprobado, con `hasNext()`, que hay

elemento siguiente. Veamos un trozo de código donde se crea un iterador `it` que empieza apuntando al principio de la colección `colecciónClie` y luego la recorre con un bucle `for`, mostrando todos sus elementos:

```
Iterator<Cliente> it = colecciónClie.iterator();
for ( ; it.hasNext(); ) {
    Cliente p = it.next();
    System.out.println(p);
}
```

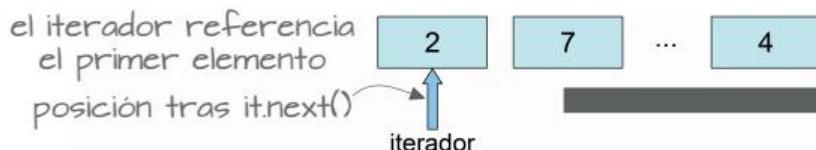


Figura 12.6. Posición del iterador tras el primer `next()`.

En el bucle hay que observar varias cosas. En primer lugar, la declaración e inicialización del iterador `it` se podría haber incluido en la parte de inicialización del bucle `for`. En segundo lugar, la parte de los incrementos está vacía. Esto se debe a que el método `next()` incrementa automáticamente el iterador para que apunte al siguiente elemento y después lo devuelve.

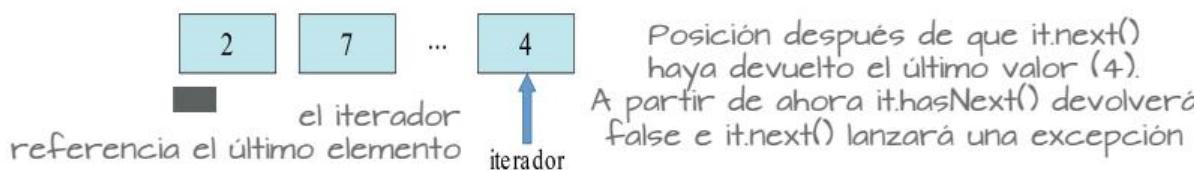


Figura 12.7. Posición del iterador tras el último `next()`.

Los iteradores tienen un tercer método que permite eliminar elementos de una colección.

- `void remove()`: elimina de la colección el último elemento devuelto por `next()`, que es el apuntado por el iterador en cada momento. Así podemos eliminar un elemento de una colección dependiendo de alguna condición mientras se está recorriendo con un iterador. Esta es la única forma de eliminar un elemento preservando la integridad de la colección. Aunque tenga el mismo nombre, no debemos confundirlo con el método `remove(Object ob)` de la interfaz `Collection`, que se invoca desde un objeto colección y no debe usarse dentro del bucle de un iterador. En este contexto, debemos invocar el del iterador, al que no se le pasa ningún parámetro. Por ejemplo, si queremos eliminar de nuestra colección aquellos clientes nacidos antes del año 2000,

```
Iterator<Cliente> it = colecciónClie.iterator();
while(it.hasNext()) {
    Cliente p = it.next();
    if (p.fechaNacimiento.compareTo(LocalDate.of(2000, 1, 1)) < 0) {
        it.remove(); /*elimina p, ultimo cliente devuelto por next()
        ;No usar colecciónClie.remove(p)!*/
    }
}
```

Si mostramos los clientes, veremos que Jorge ha sido eliminado de la colección. Una forma mucho más simple de recorrer una colección es por medio de la estructura `for` extendido o `for-each`,

```
for (Cliente c : colecciónClie) {
    System.out.println(c);
}
```

que se puede leer algo así como: «para cada cliente `c` que pertenece a `colecciónClie`, mostrar `c`». En este bucle, la variable local `c`, de tipo `Cliente`, va tomando todos los valores de la colección. Sin embargo, hay operaciones para las que un bucle `for` extendido no vale. Por ejemplo, no podemos eliminar elementos, ya que `c` es siempre la referencia a una copia de un elemento de la colección. Para la eliminación de elementos durante el recorrido de una colección, tenemos que usar un iterador.

## Actividad resuelta 12.4

Implementar una aplicación que pida por consola números enteros no negativos hasta que se introduce `-1`. Los números se irán insertando en una colección, pudiéndose repetir. Al terminar, se mostrará la colección por pantalla.

A continuación, se mostrarán todos los valores pares. Por último, se eliminarán todos los múltiplos de 3 y se mostrará por pantalla la colección resultante.

### Solución

```
Collection<Integer> numeros = new ArrayList<>(); /*las listas permiten repetidos.
Más adelante veremos que los conjuntos, no*/
System.out.print("Introducir entero: ");
Integer n = new Scanner(System.in).nextInt();
while (n >= 0) {
    numeros.add(n);
    System.out.print("Introducir entero: ");
    n = new Scanner(System.in).nextInt();
}
System.out.println("Lista completa: " + numeros);
System.out.print("Pares: ");
for (Integer a : numeros) {
    if (a % 2 == 0) {//si es par
        System.out.print(a + " ");
    }
}
System.out.println("");
for (Iterator<Integer> it = numeros.iterator(); it.hasNext();) {
    n = it.next();
    if (n % 3 == 0) {//si es múltiplo de 3
        it.remove(); //elimina el último valor devuelto por next()
    }
}
System.out.println("No múltiplos de 3: " + numeros);
```

## Actividad resuelta 12.5

Implementar una aplicación en la que se insertan 20 números enteros aleatorios entre 1 y 10 (incluidos), que pueden estar repetidos, en una colección. A continuación, se crea una lista con los mismos elementos sin repeticiones.

**Solución**

```
Collection<Integer> lista = new ArrayList<>();//admite repetidos
for (int i = 0; i < 20; i++) {
    lista.add((int) (Math.random() * 10 + 1));
}
System.out.println(lista);
Collection<Integer> listaSinRepetidos = new ArrayList<>();
for (Integer e : lista) {
    if(!listaSinRepetidos.contains(e)){
        listaSinRepetidos.add(e);
    }
}
```

**Actividad resuelta 12.6**

Implementar una aplicación donde se insertan 100 números enteros aleatorios entre 1 y 10 (incluidos), que pueden estar repetidos, en una colección. Después se eliminan todos los elementos que valen 5. Mostrar la colección antes y después de la eliminación.

**Solución**

```
Collection<Integer> lista = new ArrayList<>();//admite repetidos
for (int i = 0; i < 100; i++) {
    lista.add((int) (Math.random() * 10 + 1));
}
System.out.println(lista);
boolean eliminado = lista.remove(5);
while (eliminado) {
    eliminado = lista.remove(5);
}
/*podríamos haber prescindido de la variable eliminado y del propio cuerpo del bucle con,
   while(lista.remove(5));*/
ya que, al evaluar la condición se ejecuta la eliminación de un 5, hasta que ya no quede ninguno y devuelva un valor false, con lo cual se termina el bucle. Este código es más corto, pero el otro es más claro*/
System.out.println(lista);
```

**Actividad propuesta 12.3**

Repite la Actividad resuelta 12.6 usando un iterador para eliminar los elementos cuyo valor es 5.

**Actividad propuesta 12.4**

Implementa una aplicación donde se piden por consola números reales hasta que se introduce un 0. A medida que se leen del teclado, los valores positivos se insertan en una colección y los negativos en otra. Al final, se muestran ambas colecciones y las sumas de los elementos de cada una de ellas. Por último, se eliminan de ambas todos los valores que sean mayores que 10 o menores de -10 y se vuelven a mostrar.

## ■ ■ ■ 12.2.3. Métodos globales de la interfaz Collection

Hasta ahora hemos visto métodos de las colecciones que afectan a un solo elemento. Existen otros métodos, llamados *métodos globales*, en los que intervienen más elementos, incluso más de una colección.

- `boolean containsAll(Collection<?> c)`: se le pasa como parámetro otra colección que es de un tipo genérico cualquiera, independientemente del tipo `E` de la colección que hace la llamada. Devuelve `true` si todos los elementos de `c` están en la colección que hace la llamada y `false` si hay al menos un elemento de `c` que no está en ella.

Para ilustrarlo vamos a crear una segunda colección de objetos `Cliente`, referenciada con la variable `otrosClientes`.

```
Collection<Cliente> otrosClientes = new ArrayList<>();
otrosClientes.add(new Cliente("111", "Marta", "12/02/2000"));
otrosClientes.add(new Cliente("112", "Carlos", "01/10/2002"));
```

Hemos insertado dos clientes que ya estaban en la primera colección. Por tanto, esta contiene a todos los elementos de la segunda.

La expresión

```
colecciónClie.containsAll(otrosClientes);
```

devolverá `true`. Si ahora añadimos un elemento nuevo a `otrosClientes`,

```
otrosClientes.add(new Cliente("211", "Ana", "07/12/2001));
```

la misma expresión

```
colecciónClie.containsAll(otrosClientes);
```

devolverá `false`, ya que el nuevo elemento de `otrosClientes` no está contenido en `colecciónClie`.

- `boolean addAll(Collection<? extends E> c)`: añade a la colección que hace la llamada todos los elementos de la colección `c`. La forma en que se añaden depende de la implementación concreta de la colección. Si es una lista, se añadirán todos al final, aunque estén repetidos. Más adelante veremos que, si la implementación es un conjunto, no se añaden los repetidos ni tienen por qué insertarse al final. El tipo de `c` es `E` o una subclase. En nuestro caso, como `colecciónClie` se ha declarado como una colección del tipo genérico `Cliente` (esa sería `E`), podemos pasárle como parámetro cualquier colección que sea del tipo `Cliente` o de una subclase de `Cliente`. En particular, `otrosClientes` cumple la condición, ya que es del tipo `Cliente`.

```
colecciónClie.addAll(otrosClientes);
```

Si mostramos `colecciónClie` tal como ha quedado, se obtiene

```
[DNI: 111 Nombre: Marta Edad: 20
, DNI: 112 Nombre: Carlos Edad: 18
, DNI: 211 Nombre: Ana Edad: 19
, DNI: 111 Nombre: Marta Edad: 20
```

```
, DNI: 112 Nombre: Carlos Edad: 18
```

```
]
```

Vemos que, al tratarse de una lista, los dos elementos se han añadido al final, aunque están repetidos.

Hay un método que hace lo contrario del anterior:

- `boolean removeAll(Collection<?> c)`: elimina de la colección invocante todos los elementos que estén contenidos en `c`. Después de ejecutar el método no habrá elementos comunes a las dos colecciones.

Para probarlo en nuestro ejemplo, vamos a eliminar de `colecciónClie` todos los elementos incluidos en `otrosClientes`. Pero antes vamos a reducir esta última eliminando a Marta.

```
otrosClientes.remove(new Cliente("111","","0"))
```

Como ya hicimos con el método `contains()`, solo necesitamos el DNI en el constructor del objeto clave que pasamos a `remove()`, ya que este es el atributo que usa `equals()` para buscar e identificar el elemento que tiene que eliminar (si en `otrosClientes` estuviera repetido el elemento de Marta, solo se eliminaría una de las copias, la que aparece antes). Después de esta operación únicamente quedarán en `otrosClientes` los elementos de Ana y Carlos. Pues bien, vamos a eliminar de `colecciónClie` los elementos de `otrosClientes`,

```
colecciónClie.removeAll(otrosClientes);
```

con lo cual desaparecen todas las ocurrencias de Ana y Carlos, quedando solo los dos elementos de Marta.

Cuando queramos eliminar de una colección todas las ocurrencias de un elemento repetido, debemos tener en cuenta que `remove(Object ob)` solo elimina la primera que encuentra, empezando por el principio de la colección. Si usáramos este método para eliminar todas las ocurrencias de `ob`, tendríamos que implementar un bucle. Sin embargo, `removeAll(Collection<?> c)` elimina de la colección que hace la llamada a la función todas las ocurrencias de sus elementos que también se hallen en `c`. Más adelante veremos cómo usarlo para eliminar todas las ocurrencias de un elemento determinado.

Otro método global es:

- `boolean retainAll(Collection<?> c)`: elimina todos los elementos de la colección invocante, salvo aquellos que también estén en `c`.

## Actividad resuelta 12.7

Repetir la Actividad resuelta 12.6 usando métodos globales.

### Solución

```
Collection<Integer> lista = new ArrayList<>();
for (int i = 0; i < 100; i++) {
    lista.add((int) (Math.random() * 10 + 1));
```

```

}
System.out.println(lista);
Collection<Integer> c = new ArrayList<>();
c.add(5); //colección con un único elemento
lista.removeAll(c); //elimina todas las ocurrencias de 5
System.out.println(lista);

```

## ■ ■ ■ 12.2.4. Métodos de tabla de la interfaz Collection

Hay una tercera categoría de métodos que comparten todas las colecciones: aquellos que sirven para volcar sus datos en tablas:

- `Object [] toArray()`: devuelve una tabla de tipo `Object` con los mismos elementos de la colección. En el caso de listas, como en estas el orden importa, la tabla alberga los mismos elementos, incluyendo las repeticiones, en el mismo orden.

Para obtener en una tabla los elementos de `otrosClientes` escribiremos

```
Object [] t1 = otrosClientes.toArray();
```

La tabla `t1` tiene longitud 2 y contiene los objetos correspondientes a Ana y Carlos, pero para acceder a sus nombres, tendremos que poner un cast.

```
((Cliente)t1[0]).nombre // devolverá "Ana"
```

Como vemos, este método tiene el inconveniente de que devuelve una tabla de tipo `Object`, aunque sabemos que son clientes. Esto nos obliga a emplear un cast para acceder a los miembros de la clase a la que pertenece. Para solucionar este problema, podemos usar otra versión del método:

- `<T>T [] toArray(T [] t)`: es igual que el anterior, pero devuelve una tabla de tipo genérico `T`. El método es invocado por la colección de tipo genérico `T` y, como parámetro, le pasamos una tabla del mismo tipo. No hay que inicializar la tabla ni importa su tamaño. El método la devuelve redimensionada con el tamaño necesario para albergar todos los elementos de la colección. De hecho, es costumbre definirla con tamaño 0.

```
Cliente [] t2 = otrosClientes.toArray(new Cliente[0]);
System.out.println(t2[0].nombre); // "Ana"
```

Ahora `t2` es de tipo `Cliente` y recoge los elementos de `otrosClientes`. Con este método, no es necesario el cast delante de `t2[0]`.

La operación contraria, la de crear una colección a partir de los elementos de una tabla, es posible con el método estático de la clase `Arrays`,

```
static <T> List<T> asList(T ... a): recibe una tabla como argumento y devuelve una lista inmutable con los elementos de la tabla en el mismo orden. Al ser inmutable, no podemos insertar ni eliminar ningún elemento, pero podemos insertarla en otra colección con addAll(). Por ejemplo, si queremos crear una colección no inmutable con los elementos de la tabla de enteros tabla,
```

```
Integer[] tabla={1,2,3,4,5,6};
Collection<Integer> lista=new ArrayList<>();//no es inmutable
lista.addAll(Arrays.asList(tabla));
System.out.println(lista);
```

se mostrará por pantalla

```
[1, 2, 3, 4, 5, 6]
```

## Actividad resuelta 12.8

Implementar un programa en el que se insertan 20 números aleatorios en una colección. Esta se ordenará de menor a mayor convirtiéndola antes en tabla y volviéndola a convertir en colección. Repetir el proceso para ordenarla de mayor a menor.

### Solución

```
Collection<Integer> lista = new ArrayList<>();
for (int i = 0; i < 20; i++) {
    lista.add((int) (Math.random() * 10 + 1));
}
System.out.println(lista);
Integer[] tabla = lista.toArray(new Integer[0]);
Arrays.sort(tabla);
Collection<Integer> listaCreciente = new ArrayList<>();
listaCreciente.addAll(Arrays.asList(tabla));
System.out.println(listaCreciente);
Comparator<Integer> ordenDecreciente = new Comparator<>() {
    public int compare(Integer e1, Integer e2) {
        return e2 - e1;
    }
};
/*O bien:
Comparator<Integer> ordenEnteros = Comparator.naturalOrder();
ordenDecreciente = ordenEnteros.reversed();
*/
Arrays.sort(tabla, ordenDecreciente);
Collection<Integer> listaDecreciente = new ArrayList<>();
listaDecreciente.addAll(Arrays.asList(tabla));
System.out.println(listaDecreciente);
```

## ■ 12.3. Métodos específicos de la interfaz List

Todos los métodos vistos hasta ahora pertenecen a la interfaz `Collection` y, aunque los hemos probado con listas, son implementados por todas las colecciones, tanto listas como conjuntos. En realidad, las listas implementan la interfaz `List`, que hereda de `Collection`, añadiéndole una serie de métodos y funcionalidades específicas, que no comparten los conjuntos.

La funcionalidad más importante exclusiva de las listas (ya sean de la clase `ArrayList` como de `LinkedList`) es el acceso posicional a sus elementos por medio de índices. El primer elemento tiene índice 0, el segundo índice 1 y así sucesivamente, como en las

tablas. Las listas se inspiran en las sucesiones matemáticas. Sus elementos pueden estar repetidos y el orden en el que se encuentran es relevante.

Vamos a crear una nueva lista, ahora con números enteros de la clase `Integer`, y la vamos a asignar a una variable de tipo `List` para acceder a todas las funcionalidades de las listas.

```
List<Integer> listaEnteros = new ArrayList<>();
```

Aquí podríamos haber creado una lista de tipo `LinkedList` sin que cambie nada en el resto del epígrafe.

Insertamos algunos elementos:

```
listaEnteros.add(3);  
listaEnteros.add(1);  
listaEnteros.add(-2);  
listaEnteros.add(0);  
listaEnteros.add(3);  
listaEnteros.add(7);
```

Si mostramos la lista con

```
System.out.println(listaEnteros);
```

obtendremos

```
[3, 1, -2, 0, 3, 7]
```

En la lista el método `add()` inserta el nuevo elemento al final. Es decir, una lista, en principio, mantiene el orden de inserción —cuando hablamos del orden, a secas, de los elementos de una colección, nos referimos al orden en el que los obtenemos al recorrerla con un iterador que, en general, no tiene por qué coincidir con el orden de inserción—.

Los métodos más importantes aportados por la interfaz `List` son:

- `E get(int indice)`: devuelve el elemento que ocupa el lugar `indice` en la lista, siendo 0 el índice del primer elemento, como en las tablas.

Por ejemplo,

```
listaEnteros.get(2)
```

devolverá -2, que es el valor del elemento que ocupa el tercer lugar de la lista.

- `E set(int indice, E elem)`: guarda el elemento `elem` en la posición `indice`, machacando el valor que hubiera previamente en esa posición, que es devuelto. Con el siguiente código, vamos a poner el valor 10 en el elemento de índice 3, sustituyendo su valor actual (0), que es devuelto y asignado a la variable `y`.

```
Integer y = listaEnteros.set(3, 10);  
System.out.println("y: " + y);  
System.out.println(listaEnteros);
```

Se mostrará por pantalla

```
y: 0  
[3, 1, -2, 10, 3, 7]
```

- `void add(int indice, E elem)`: inserta el valor `elem` en la posición `indice`. Todos los elementos que ocupaban una posición igual o mayor que `indice`, se desplazan una posición hacia el final de la lista, para dejar hueco al nuevo elemento. Por ejemplo, si queremos insertar el valor 5 entre el -2 y el 10, se insertará en la posición 3, que actualmente ocupa el 10. Este y los elementos que le siguen se desplazarán un lugar hacia el final de la lista.

```
listaEnteros.add(3, 5);
System.out.println(listaEnteros);
```

Veremos por pantalla

[3, 1, -2, 5, 10, 3, 7]

- `boolean addAll(int indice, Collection<? extends E> c)`: inserta todos los elementos de la colección `c`, en el mismo orden que tengan, en la lista que invoca al método, empezando por el lugar `indice` y desplazando hacia el final todos los elementos de la lista original a partir de `indice`, incluido este, tantos lugares como sean necesarios. Los elementos de la colección `c` deben ser del mismo tipo `E` que los de la lista original, o de un subtipo de `E`.

Vamos a crear una segunda lista de enteros `Integer`:

```
ArrayList<Integer> otrosEnteros = new ArrayList<>();
otrosEnteros.add(20);
otrosEnteros.add(30);
otrosEnteros.add(40);
```

Ahora insertamos esta lista en el lugar de índice 2 de `listaEnteros`, es decir, donde se encuentra el valor -2. Este elemento, junto con los que le siguen se desplazan hacia el final para hacer sitio a los tres valores insertados.

```
listaEnteros.addAll(2, otrosEnteros);
System.out.println(listaEnteros);
```

Aparecerá en pantalla

[3, 1, 20, 30, 40, -2, 5, 10, 3, 7]

Para eliminar un elemento hay una versión sobrecargada del método `remove()`, que ya conocíamos, de la interfaz `Collection`.

- `E remove(int indice)`: elimina el elemento que ocupa el lugar `indice` y lo devuelve.

En este caso, hay que tener cuidado si lo usamos en una lista de objetos `Integer`, como la de nuestro ejemplo, ya que una sentencia como

```
listaEnteros.remove(5);
```

no será interpretada como que queremos eliminar un elemento `Integer` con valor 5 de la lista, sino el elemento con índice 5, es decir, el elemento de valor -2, ya que, al ser de tipo `int` el valor pasado como parámetro, dado que hay una versión de `remove()` que admite un `int` como argumento, Java le da prioridad y no hace autoboxing. En el caso de que quisiéramos eliminar un elemento `Integer` cuyo valor es 5, escribiríamos

```
listaEnteros.remove(Integer.valueOf(5));
```

Así estaríamos pasando como argumento un objeto `Integer` y no un valor `int`, con lo cual Java ejecuta la versión del método correspondiente a la interfaz `Collection` y elimina el elemento de valor 5.

Además de los métodos de lectura, escritura, inserción y eliminación de elementos, heredados de la interfaz `Collection`, la interfaz `List` añade funciones de búsqueda, ordenación y comparación.

- `int indexOf(Object ob)`: devuelve el índice de la primera ocurrencia de `ob` en la lista. Si no está, devuelve -1.
- `int lastIndexOf(Object ob)`: hace lo mismo que `indexOf()`, pero empezando la búsqueda por el final, devolviendo la última ocurrencia de `ob`.
- `boolean equals(Object otraLista)`: compara dos listas, tanto si las dos son `ArrayList` como si son `LinkedList`, o una de cada, y devuelve `true` si ambas tienen exactamente los mismos elementos, incluidas las repeticiones, en el mismo orden.
- `void sort(Comparator<? super E> c)`: ordena la lista invocante con el criterio de `c`, cuya implementación compara objetos de la clase `E` o una superclase (para que no se utilicen atributos que no están en `E`). Para ordenar por el criterio de orden natural de `E`, caso de que exista, antes tendremos que obtener el comparador correspondiente, implementándolo nosotros mismos, o por medio del método `naturalOrder()`. También podremos recurrir a la clase `Collections` (con «s» al final, no confundir con la interfaz `Collection`), que estudiaremos más adelante.

## Recuerda



Ya vimos el método `indexOf()` en la unidad sobre cadenas, que sirve para hacer una búsqueda secuencial de la primera ocurrencia de un carácter. Sin embargo, no existe un equivalente para las tablas no ordenadas. De hecho, una forma de buscar un valor en una tabla no ordenada es convertirla antes en una lista con el método `asList()` de la clase `Arrays`.

## Actividad resuelta 12.9

Crear una lista de números enteros positivos introducidos por consola hasta que se introduzca uno negativo. A continuación recorrer la lista y mostrar por pantalla los índices de los elementos de valor par, que será multiplicado por 100.

### Solución

```
List<Integer> lista = new ArrayList<>();
System.out.print("Introducir número: ");
Integer n = new Scanner(System.in).nextInt();
while (n >= 0) {
    lista.add(n);
    System.out.print("Introducir número: ");
    n = new Scanner(System.in).nextInt();
```

```
    }
    System.out.println(lista);
    System.out.print("Índices de valores pares: ");
    for (int i = 0; i < lista.size(); i++) {
        if(lista.get(i)%2 == 0){
            System.out.print(i+" ");
            lista.set(i, lista.get(i)*100);
        }
    }
    System.out.println("");
    System.out.println(lista);
```

## 12.4. Interfaz Set

La interfaz `Set` trata los datos como un conjunto matemático, eliminando las repeticiones y sin un orden preestablecido; aunque, como veremos, una de sus implementaciones permite introducir un orden. Todos sus métodos los hereda de `Collection`. Lo único que añade es la restricción de no permitir duplicados. Esto significa que si intentamos insertar un elemento que ya existe, no lo hará.

El conjunto de métodos disponibles es el mismo que vimos en los apartados de métodos básicos y globales de las colecciones:

1. `int size()`
2. `boolean isEmpty()`
3. `boolean contains(Object element)`
4. `boolean add(E element)`
5. `boolean remove(Object element)`
6. `Iterator<E>iterator()`
7. `boolean containsAll(Collection<?>c)`
8. `boolean addAll(Collection<? extends E>c)`
9. `boolean removeAll(Collection<?>c)`
10. `boolean retainAll(Collection<?>c)`
11. `void clear()`
12. `Object[] toArray()`
13. `<T>T[] toArray(T[])`

Asimismo, podemos recorrer un conjunto con un iterador o con una estructura `for-each`, igual que las listas.

Las diferencias más importantes son el orden en que se van insertando los elementos nuevos y que un elemento que ya está en el conjunto no se puede volver a insertar, ya que no son posibles los elementos repetidos. Cuando intentemos insertar un elemento repetido con el método `add()` o con `addAll()`, no se producirá ningún error ni se arrojará

ninguna excepción; sencillamente, el elemento no se inserta y, como el conjunto no habrá cambiado, el método devuelve `false`.

Sin embargo, no tendremos los métodos propios de listas, que vienen declarados en la interfaz `List`. En particular, no es posible el acceso posicional por medio de índices, aunque sí las iteraciones.

Las implementaciones de `Set` son las clases: `HashSet`, `TreeSet` y `LinkedHashSet`.

- `HashSet`: tiene un buen rendimiento, aunque no garantiza ningún orden en la inserción.
- `TreeSet`: a pesar de tener peor rendimiento, garantiza el orden basado en el valor de los elementos insertados. El criterio de ordenación por defecto es el natural (el proporcionado por el método `compareTo()` de la clase genérica `E`) o bien se especifica por medio de un comparador pasado como parámetro al constructor.
- `LinkedHashSet`: inserta los elementos al final, con lo cual se garantiza un orden basado en la inserción.

Al contrario de lo que pasa con las listas, las tres implementaciones de `Set` tienen diferencias en su comportamiento. Es muy común utilizar variables de tipo `Set` para referenciarlos. Esto permite aprovechar el polimorfismo de las distintas implementaciones y, como veremos, hacer transformaciones de unas en otras.

Por ejemplo, vamos a declarar un conjunto con un orden natural, basado en la implementación de la interfaz `Comparable` del tipo de los elementos, en este caso `Cliente`.

```
TreeSet<Cliente> conjuntoClientes = new TreeSet<>();
```

Sabemos que la clase `Cliente` implementa el método `compareTo()` basado en el atributo `dni`. Por tanto, los elementos se insertarán ordenados por `dni`, que es la ordenación natural de los objetos `Cliente`. Podríamos haber declarado `conjuntoCliente` como una variable de tipo `Set` o incluso `Collection`, ya que los métodos disponibles son los mismos y su comportamiento depende del objeto referenciado, en este caso de la clase `TreeSet`.

Insertaremos los mismos elementos que en la lista `listaClientes` y los mostramos.

```
conjuntoClientes.add(new Cliente("111", "Marta", "12/02/2000"));
conjuntoClientes.add(new Cliente("115", "Jorge", "16/03/1999"));
conjuntoClientes.add(new Cliente("112", "Carlos", "01/10/2002"));
System.out.println(conjuntoClientes);
```

Veremos por pantalla

```
[DNI: 111 Nombre: Marta Edad: 20
, DNI: 112 Nombre: Carlos Edad: 18
, DNI: 115 Nombre: Jorge Edad: 21
]
```

Observamos que el orden en que aparecen no coincide con el orden de inserción, sino que están ordenados por DNI creciente.

Si ahora intentamos volver a insertar uno de los elementos anteriores, por ejemplo, el de Marta,

```
boolean insertado = conjuntoClientes.add(new Cliente("111", "Marta", "12/02/2000"));
System.out.println(insertado); /*false, ya que no se ha insertado */
System.out.println(conjuntoClientes);
```

aparece por pantalla

```
false
[DNI: 111 Nombre: Marta Edad: 20
, DNI: 112 Nombre: Carlos Edad: 18
, DNI: 115 Nombre: Jorge Edad: 21
]
```

donde vemos que la inserción ha devuelto `false` (no se ha insertado) y que el conjunto no ha cambiado. Pero si queremos otro conjunto de clientes ordenados por nombre, lo creamos pasando a su constructor, como argumento, un comparador basado en el atributo nombre.

## Actividad propuesta 12.5

A partir de `conjuntoClientes` del ejemplo, crea otro conjunto con los mismos elementos ordenados por edad y otro con los clientes ordenados por nombre.

## Actividad resuelta 12.10

Insertar en una lista 20 enteros aleatorios entre 1 y 10. A partir de ella, crear un conjunto con los elementos de la lista sin repetir, otro con los repetidos y otro con los elementos que aparecen una sola vez en la lista original.

### Solución

```
List<Integer> lista = new ArrayList<>();
for (int i = 0; i < 20; i++) {
    lista.add((int) (Math.random() * 10) + 1);
}
/*ordenamos la lista para facilitar la visualización de los elementos originales: */
Comparator<Integer> c = Comparator.naturalOrder();
lista.sort(c); /*más adelante veremos que la clase Collections nos ahorra este c*/
System.out.println("Lista original: " + lista);
Set<Integer> sinRepeticiones = new TreeSet<>();
sinRepeticiones.addAll(lista);
System.out.println("Sin repeticiones: " + sinRepeticiones);
Set<Integer> repetidos = new TreeSet<>();
for (Integer e : sinRepeticiones) {
    lista.remove(e); //solo elimina una ocurrencia de e
}
/*después de eliminar de la lista uno de cada, solo quedan en ella las repeticiones*/
repetidos.addAll(lista);
System.out.println("Repetidos: " + repetidos);
Set<Integer> unicos = new TreeSet<>();
unicos.addAll(sinRepeticiones);
unicos.removeAll(repetidos);
System.out.println("Elementos no repetidos: " + unicos);
```

Por otra parte, para que los elementos se vayan colocando por orden de inserción, como en las listas, en vez de un `TreeSet` crearíamos un objeto `LinkedHashSet`. Si el orden nos es indiferente, podemos usar un `HashSet`, que tiene un mejor rendimiento.

## Actividad resuelta 12.11

Implementar la clase `Socio`, cuyos atributos son `dni`, `nombre` y `fechaAlta`, que deberá incluir el método `equals()`, la interfaz `Comparable` basada en el `dni` y el método `antiguedad()`. Implementar un programa que gestione los socios de un club guardando los datos en el fichero `socios.dat`. Al arrancar la aplicación, se leen los datos del fichero y se abre un menú con las opciones: 1. Alta; 2. Baja; 3. Modificación; 4. Listado por DNI; 5. Listado por antigüedad, y 6. Salir.

Al salir de la aplicación se guardan en el fichero los datos actualizados.

### Solución

```
public class Socio implements Comparable<Socio>, Serializable {
    String dni;
    String nombre;
    LocalDate fechaAlta;
    public Socio(String dni, String nombre, String alta) {
        this.dni = dni;
        this.nombre = nombre;
        DateTimeFormatter f = DateTimeFormatter.ofPattern("dd/MM/yyyy");
        this.fechaAlta = LocalDate.parse(alta, f);
    }
    /*Constructor para las búsquedas*/
    public Socio(String dni) {
        this.dni = dni;
    }
    int antiguedad() {
        return (int) fechaAlta.until(LocalDate.now(), ChronoUnit.YEARS);
    }
    /*Ordenación natural por DNI*/
    @Override
    public int compareTo(Socio o) {
        return dni.compareTo(o.dni);
    }

    /*Definimos un criterio de igualdad basado en el DNI, que no se puede repetir, e identifica a los socios de forma única. Además es consistente con el criterio de comparación de compareTo(): */
    @Override
    public boolean equals(Object o) {
        return dni.equals(((Socio) o).dni);
    }
    @Override
    public String toString() {
        return "Socio{" + "dni=" + dni + ", nombre=" + nombre
               + ", antiguedad=" + antiguedad() + "}";
    }
}
```

```
/*Programa principal: */
public static void main(String[] args) {
    /*Como los socios no pueden repetirse, usamos un conjunto para guardarlos.
    Además, con un TreeSet se mantendrán ordenados por DNI, que es su clave
    única de identificación*/
    Set<Socio> socios = new TreeSet<>();
    try ( ObjectInputStream in = new ObjectInputStream(
        new FileInputStream("socios.dat"))) {
        /*Al leer del archivo, el compilador no tiene modo de saber si el objeto
        leído se corresponde con el tipo que figura en el cast ni con el de la
        variable 'socios' a la que es asignado. Por tanto, en esa operación,
        no puede hacer comprobación de tipos. De ahí el aviso generado en la
        compilación. Es responsabilidad del programador asegurarse de que los tipos
        son los correctos: */
        socios = (TreeSet<Socio>)in.readObject();
    } catch (IOException ex) {
        System.out.println("Lista de socios vacía");
    } catch (ClassNotFoundException ex) {
        System.out.println(ex);
    }
    int opcion;
    do {
        System.out.println("1.Alta");
        System.out.println("2.Baja");
        System.out.println("3.Modificación");
        System.out.println("4.Listado por dni");
        System.out.println("5.Listado por antigüedad");
        System.out.println("6.Salir");
        System.out.print("\nIntroducir opción: ");
        opcion = new Scanner(System.in).nextInt();
        switch (opcion) {
            case 1 -> {
                System.out.print("dni: ");
                String dni = new Scanner(System.in).next();
                alta(socios, dni);
            }
            case 2 -> {
                System.out.print("dni socio: ");
                String dni = new Scanner(System.in).next();
                socios.remove(new Socio(dni));
            }
            case 3 -> {
                System.out.print("dni: ");
                String dni = new Scanner(System.in).next();
                socios.remove(new Socio(dni));
                alta(socios, dni);
            }
            case 4 -> {
                System.out.println(socios);
            }
            case 5 -> {
                Comparator<Socio> c = new Comparator<>() {
                    @Override
                    public int compare(Socio o1, Socio o2) {
                        return o2.antiguedad() - o1.antiguedad();
                    }
                };
                TreeSet<Socio> sortedSocios = new TreeSet<Socio>(c);
                sortedSocios.addAll(socios);
                System.out.println(sortedSocios);
            }
        }
    } while (opcion != 6);
}
```

```

        }
    };
    Set<Socio> s = new TreeSet<>(c);
    s.addAll(socios);
    System.out.println(s);
}
}

}while (opcion != 6);
try ( ObjectOutputStream out = new ObjectOutputStream(
new FileOutputStream("socios.dat"))) {
    out.writeObject(socios);
} catch (IOException ex) {
    System.out.println(ex);
}
}

static boolean alta(Set<Socio> socios, String dni) {
    System.out.print("nombre: ");
    String nombre = new Scanner(System.in).next();
    System.out.print("fecha de alta: ");
    String fechaAlta = new Scanner(System.in).next();
    Socio nuevo = new Socio(dni, nombre, fechaAlta);
    return socios.add(nuevo);
}
}

```

## 12.5. Conversiones entre colecciones

Una característica interesante de los conjuntos y de todas las colecciones en general es la posibilidad de crear unas a partir de otras, del mismo o distinto tipo, por medio de los constructores. Por ejemplo, para obtener un conjunto ordenado (un `TreeSet`) a partir de otro que no lo está (un `HashSet` o `LinkedHashSet`) o, dicho de otra forma, si queremos ordenar un conjunto, disponemos de dos caminos que podemos seguir:

1. Construimos un `TreeSet` con el criterio de ordenación que deseamos y luego le añadimos con `addAll()` el conjunto que queremos ordenar.
2. Si el criterio de ordenación va a ser el natural (el de la interfaz `Comparable`), podemos construir un `TreeSet` pasando como argumento a su constructor el conjunto que queremos ordenar.

Para ilustrar los dos planteamientos, vamos a crear un `LinkedHashSet` de números enteros. Después le añadimos cinco elementos, que se irán colocando por orden de inserción.

```

Set<Integer> conjuntoEnteros = new LinkedHashSet<>();
conjuntoEnteros.add(4);
conjuntoEnteros.add(1);
conjuntoEnteros.add(5);
conjuntoEnteros.add(10);
conjuntoEnteros.add(3);
System.out.println(conjuntoEnteros);

```

Obtenemos

```
[4, 1, 5, 10, 3]
```

En la variable `conjuntoEnteros` hemos usado el tipo `Set`, es decir, el nombre de la interfaz. Esto es una práctica común si queremos tener la posibilidad de referenciar, con la misma variable, conjuntos con distintas implementaciones. Si la variable `conjuntoEnteros` fuera de tipo `LinkedHashSet`, solo serviría para referenciar objetos de esa clase, pero no un `TreeSet`. Es una forma más de polimorfismo en Java. Volviendo a nuestro ejemplo, si queremos obtener un conjunto ordenado a partir de los elementos de `conjuntoEnteros`, podemos crear un `TreeSet` y añadírselos.

```
Set<Integer> conjuntoEnterosOrdenados = new TreeSet<>();  
conjuntoEnterosOrdenados.addAll(conjuntoEnteros);  
System.out.println(conjuntoEnterosOrdenados);
```

obteniendo

```
[1, 3, 4, 5, 10]
```

Ahora podemos dejar las variables `conjuntoEnteros` y `conjuntoEnterosOrdenados`, referenciando cada una un tipo distinto de conjunto, o referenciar con `conjuntoEnteros` el nuevo conjunto ordenado.

```
conjuntoEnteros = conjuntoEnterosOrdenados;
```

con lo cual, a todos los efectos, habríamos ordenado `conjuntoEnteros`. A partir de ese momento `conjuntoEnteros` sería un `TreeSet` y mantendría el orden al insertar o eliminar elementos.

La segunda forma de ordenar un conjunto es pasarlo al constructor de un `TreeSet`.

```
Set<Integer> conjuntoEnterosOrdenados = new TreeSet<>(conjuntoEnteros);
```

con lo que obtendríamos el mismo resultado.

No obstante, este segundo método solo sirve si queremos un conjunto con el orden natural de los elementos. Si, en vez de enteros, tuviéramos un conjunto de clientes sin ordenar y quisieramos ordenarlos por nombre, tendríamos que usar el primer procedimiento, construyendo un `TreeSet` con un comparador `ComparaNombres` (véase su implementación en el Apartado 12.1.2).

```
Set<Cliente> conjuntoClientes = new LinkedHashSet<>(); /*Sin orden*/  
conjuntoClientes.add(new Cliente("111", "Marta", "12/02/2000"));  
conjuntoClientes.add(new Cliente("115", "Jorge", "16/03/1999"));  
conjuntoClientes.add(new Cliente("112", "Carlos", "01/10/2002"));  
Set<Cliente> conjuntoClientesOrdenados = new TreeSet<>(  
new ComparaNombres());  
//el mismo conjunto ordenado por nombres:  
conjuntoClientesOrdenados.addAll(conjuntoClientes);  
System.out.println(conjuntoClientesOrdenados);
```

Obtenemos así los clientes ordenados por nombre.

Utilizando los constructores, es posible hacer conversiones entre todo tipo de colecciones. Se pueden crear listas pasando conjuntos a su constructor y viceversa; también se pueden añadir a una lista todos los elementos de un conjunto. Un ejemplo útil es la creación de un conjunto a partir de una lista para eliminar elementos repetidos.

```
List<Integer> lista = new ArrayList<>();  
lista.add(5);  
lista.add(3);  
lista.add(5); //elemento repetido  
lista.add(2);  
lista.add(5); //elemento repetido  
Set<Integer> conjunto = new LinkedHashSet<>(lista); /*sin repetidos */  
System.out.println(conjunto);
```

Obtenemos

```
[5, 3, 2]
```

donde se han eliminado las repeticiones. Si queremos recuperar la lista original, pero sin repeticiones,

```
lista = new ArrayList<>(conjunto);
```

El orden de los elementos se ha mantenido en todas estas transformaciones porque tanto `ArrayList` como `LinkedHashSet` respetan el orden de inserción.

Cuando se trabaja con colecciones de distinto tipo, siempre que se usen constructores o métodos comunes a listas y conjuntos, es costumbre definir las variables de tipo `Collection` para permitir que la misma variable refiera a diferentes tipos de colección en caso de conversiones. El código anterior podría ser:

```
Collection<Integer> colección = new ArrayList<>();  
colección.add(5);  
...  
colección = new LinkedHashSet<>(colección); //de lista a conjunto  
colección = new ArrayList<>(colección); //de conjunto a lista  
System.out.println(colección);
```

En este caso, con `colección` podemos referenciar cualquier lista o conjunto. El comportamiento dependerá del objeto referenciado.

También es posible hacer las conversiones encadenadas con una única sentencia,

```
colección = new ArrayList<>(new LinkedHashSet<>(colección));
```

dando los mismos resultados. Aquí habría valido una variable de tipo `List`.

## Actividad resuelta 12.12

Implementar un método estático que lleve a cabo la unión de dos conjuntos de elementos genéricos. La unión es un nuevo conjunto con todos los elementos que pertenezcan, al menos, a uno de los dos conjuntos.

Hacer lo mismo con la intersección, formada por los elementos comunes a los dos conjuntos. Los prototipos de los métodos son:

- static <E> Set<E> union(Set<E> conjunto1, Set<E> conjunto2)
- static <E> Set<E> interseccion(Set<E> conjunto1, Set<E> conjunto2)

### Solución

```
/*Creamos un conjunto donde añadir los elementos de los dos conjuntos. Como los conjuntos no permiten repetidos, el conjunto resultante es la unión de ambos. */
static <E> Set<E> union(Set<E> conj1, Set<E> conj2) {
    Set<E> resultado = new HashSet<>(conj1);
    resultado.addAll(conj2); //añadimos los elementos de conj2
    return resultado;
}

/*Usamos el método retainAll() de Set, que elimina todos los elementos de un conjunto , salvo los pertenecientes al conjunto pasado como parámetro.*/
static <E> Set<E> interseccion(Set<E> conj1, Set<E> conj2) {
    /*creamos el conjunto resultante, que estará inicializado con los elementos de conj1: */
    Set<E> interseccion = new HashSet<>(conj1);
    /*borra todos los elementos de interseccion salvo los que estén en conj2. Solo quedan los comunes a ambos conjuntos: */
    interseccion.retainAll(conj2);
    return interseccion;
}
```

## ■ 12.6. Clase Collections

Además de los métodos aportados por las interfaces `Collection`, `List` y `Set`, la clase `Collections` (no confundirla con la interfaz `Collection`) reúne una serie de utilidades en forma de métodos estáticos que trabajan con tipos genéricos. En ellos, el primer parámetro de entrada es la colección sobre la que deseamos operar y comprende métodos de búsqueda, ordenación y manipulación de datos, entre otros. Casi todos ellos operan sobre listas, aunque algunos valen para cualquier colección.

### ■ ■ ■ Métodos de ordenación

Ya vimos que las listas se pueden ordenar por medio del método `sort()`, al que se le pasa un comparador como argumento. Sin embargo, la clase `Collections` también posee métodos `sort()` estáticos para ordenar listas. El primero es:

- static <T extends Comparable<? super T>> void sort(List<T> lista): ordena una lista que se le pasa como argumento (los conjuntos no se pueden ordenar. En todo caso, los TreeSet se mantienen ordenados de forma automática). Esta tendrá elementos de un tipo genérico `T` que tenga implementada la interfaz `Comparable`. El criterio de ordenación será el llamado «criterio natural», que es el que establecerá el método `compareTo()` de la clase `T`. Las clases envoltorio —wrapper— proporcionadas por Java, como `Integer`, `Character` o `Double`, así como la

clase `String` lo traen implementado, de forma que ordenan los números de menor a mayor, los caracteres según el orden de Unicode y los `String` por orden alfabético.

Veamos un ejemplo. Creamos una lista `ArrayList`, para objetos `Cliente`

```
List<Cliente> lista = new ArrayList<>();
```

e insertamos varios elementos

```
lista.add(new Cliente("111", "Marta", "12/02/2000"));
lista.add(new Cliente("115", "Jorge", "16/03/1999"));
lista.add(new Cliente("112", "Carlos", "01/10/2002"));
```

A diferencia del método `sort()` de `List`, el de `Collections` es estático, y se invoca con el nombre de la clase. La lista va como argumento.

Si queremos ordenarla, solo tenemos que escribir

```
Collections.sort(lista);
```

La lista se ordenará con el criterio natural del tipo con el que se declaró. En nuestra clase `Cliente`, por el atributo `dni`.

Si queremos ordenar con otro criterio, tendremos que usar comparadores. Para ello disponemos de otra versión sobrecargada de `sort()`.

- `static <T> void sort(List<T> lista, Comparator<? super T> c)`: ordena listas con el criterio del comparador que se le añade como segundo parámetro. Si queremos ordenar por nombre, escribiremos

```
Collections.sort(lista, new ComparaNombres());
```

## Métodos de búsqueda

Uno de los métodos más importantes de la clase `Collections` es:

- `static <T> int binarySearch(List<? extends Comparable<? super T>> list, T clave)`: hace una búsqueda binaria de un objeto, llamado clave de búsqueda, en una lista que debe estar ordenada previamente. Todo ello necesita un criterio de ordenación que, por defecto, es el natural del tipo genérico de la lista. Se exige que la implementación de `Comparable` sea también genérica. Esto, en la clase `Cliente` sería de la siguiente forma:

```
class Cliente implements Comparable<Cliente> {
    ...
    public int compareTo(Cliente ob) {
        return dni.compareTo(ob.dni);
    }
}
```

Vamos a hacer una búsqueda en la lista de clientes. En primer lugar, la volvemos a ordenar por DNI, que es el orden natural.

```
Collections.sort(lista); //ordenada por dni (orden natural)
```

Al método `binarySearch()` se le pasan como parámetros la lista en la que queremos hacer la búsqueda y el objeto clave que queremos buscar. Devuelve el índice de este último si lo encuentra. Por ejemplo, si queremos buscar a Carlos, cuyo DNI es «112»,

```
int indice = Collections.binarySearch(lista, new Cliente("112", null, null));
```

O bien, si tenemos implementado un constructor de `Cliente` con un único parámetro `dni`,

```
int indice = Collections.binarySearch(lista, new Cliente("112"));
```

Como ya vimos cuando estudiamos las tablas, en el caso de que la clave no esté en la lista, devolverá un entero negativo del que se puede deducir el índice `indiceInsercion` que le correspondería al elemento si lo insertáramos manteniendo la lista ordenada. La fórmula es:

```
indiceInsercion = -indice - 1;
```

Supongamos que queremos insertar a Eva en la lista en caso de que no esté. Para ello, primero la buscamos con `binarySearch()`. Como la búsqueda nos da un valor negativo, calculamos su índice de inserción y la insertamos.

```
Cliente nuevo = new Cliente("555", "Eva", "21/09/2003");
int indice = Collections.binarySearch(lista, nuevo);
if (indice < 0) { //no está en la lista
    lista.add(-indice - 1, nuevo); //lista sigue ordenada
}
```

Si queremos hacer una búsqueda en una lista ordenada con un criterio distinto al natural, tendremos que pasar a `binarySearch()`, como tercer parámetro, el mismo comparador que se usó para ordenarla.

- `static <T> int binarySearch(List<? extends T> list, T clave, Comparator<? super T> c)`: busca la clave en una lista ordenada con el criterio de ordenación del comparador c.

Por ejemplo, si ordenamos lista por orden alfabético de nombres,

```
Collections.sort(lista, new ComparaNombres());
```

para buscar a Carlos, ahora debemos hacerlo por nombre,

```
indice = Collections.binarySearch(lista, new Cliente(null, "Carlos", null), new
ComparaNombres());
```

que devolverá 0, ya que Carlos es el primero de la lista por orden alfabético de nombres.

El método `binarySearch()` es extremadamente eficiente y sus tiempos de búsqueda son muy cortos en comparación con otros métodos de búsqueda, como el secuencial. El inconveniente es que precisa que la lista esté ordenada previamente. Surge la duda de si no merece la pena ordenarla para acelerar las búsquedas. Sin embargo, esto es así solo si vamos a tener que hacer muchas búsquedas con el mismo criterio. En caso contrario, es más eficiente hacer una búsqueda secuencial.

## Recuerda



En la clase `Arrays` se implementa una batería de métodos `binarySearch()` para tablas de toda clase de datos y criterios de ordenación.

## Métodos para la manipulación de datos

Si queremos intercambiar dos elementos en una lista, usaremos

`static void swap(List<?> lista, int i, int j)`: intercambia los elementos con índices `i` y `j` entre sí. Pongamos un ejemplo con una lista de enteros:

```
List<Integer> datos = new ArrayList<>();
datos.add(1); //indice 0
datos.add(2);
datos.add(3);
datos.add(4); //indice 3
datos.add(5);
Collections.swap(datos, 0, 3); //cambia los elementos con índices 0 y 3
```

La lista quedaría

[4, 2, 3, 1, 5]

Para reemplazar todas las ocurrencias de un elemento determinado por otro,

`static <T> boolean replaceAll(List<T> lista, T antiguo, T nuevo)`: reemplaza el elemento antiguo, en todos los lugares en que aparezca en la lista, por el nuevo.

Por ejemplo, si queremos reemplazar los elementos que valgan 4 por el valor 100,

```
Collections.replaceAll(datos, 4, 100);
```

que da,

[100, 2, 3, 1, 5]

Podemos llenar todos los elementos que tiene una lista con un valor que pasamos como parámetro.

`static <T> void fill(List<? super T> lista, T valorRelleno)`: sustituye todos los valores de los elementos de la lista por el de `valorRelleno`. La lista debe ser de tipo `<? super T>`, es decir, de la clase `T` o cualquier superclase de `T`. Dicho de otra forma, el elemento de relleno debe ser de la clase genérica de la lista o de una subclase. Rellenemos la lista `datos` con el valor 7:

```
Collections.fill(datos, 7);
```

quedando `datos` como,

[7, 7, 7, 7, 7]

Para copiar una lista en otra usamos

`static <T> void copy(List<? super T> destino, List<? extends T> origen)`: copia los elementos de la lista `origen` en la lista `destino`, empezando por el principio y sobrescribiendo los valores que hubiera antes. La lista `destino` deberá ser, como mínimo, del tamaño de la lista `origen`. Los elementos de la lista `origen` deben ser de clase compatible con la lista `destino`. Por eso, los elementos de esta última deben ser de clase `T` o superclase de `T` (`<? super T>`) y los elementos de la lista `origen` deben ser de clase `T` o subclase

de `T (<? extends T>)`. En particular, si las dos listas son de elementos de la misma clase genérica, se podrán copiar sin problema. Construyamos una segunda lista de enteros:

```
List<Integer> otrosDatos = new ArrayList<>();  
otrosDatos.add(1);  
otrosDatos.add(2);  
otrosDatos.add(3);
```

Ahora vamos a copiarla en datos,

```
Collections.copy(datos, otrosDatos);
```

con lo que `datos` queda:

```
[1, 2, 3, 7, 7]
```

### Recuerda



En la clase `System` disponemos del método `arrayCopy()` para copiar una tabla en otra:

```
static void arraycopy(Object orig, int posOrig, Object dest, int posDest, int longitud)
```

Sin embargo, debemos tener cuidado, porque los parámetros origen y destino en el método `copy()` de `Collections` aparecen en el orden contrario.

## Otras utilidades

A veces hace falta que los elementos estén desordenados. Por ejemplo, en aplicaciones de juegos, es útil la siguiente función:

- `static void shuffle(List<?> lista)`: `shuffle` significa barajar en inglés; el método desordena los elementos de lista. Si escribimos

```
Collections.shuffle(datos);
```

la lista `datos` quedará desordenada, es decir, con un orden aleatorio. En realidad, Java utiliza una fórmula para generar valores pseudoaleatorios, con lo cual el desorden es aparente. Pero el efecto es el mismo, ya que el usuario es incapaz de predecir los resultados.

- `static int frequency(Collection<?> col, Object ob)`: nos devuelve el número de veces que aparece un elemento en una colección. Por ejemplo:

```
Collections.frequency(datos, 7)
```

devolverá 2, ya que 7 aparece dos veces en `datos`.

- `static <T extends Comparable<? super T>> T max(Collection<? extends T> col)`: busca y devuelve el elemento con valor máximo de una colección —no tiene por qué ser una lista—. Las comparaciones se basan en el orden natural, lo que exige que la clase genérica de los elementos tenga implementada la interfaz `Comparable`. Por ejemplo:

```
Integer maximo = Collections.max(datos);
```

nos dará 7.

Si queremos el valor máximo atendiendo a un criterio de ordenación distinto del natural, le pasaremos a `max()` un segundo parámetro con un comparador adecuado,

- `static <T> T max(Collection<? extends T> col, Comparator<? super T> comp)`: devuelve el máximo utilizando `comp` como criterio de comparación. Por ejemplo, volviendo al conjunto de elementos Cliente, `conjuntoClie`, con Marta, Carlos y Jorge, con el criterio de ordenación por nombres, el máximo es el elemento que ocuparía el último lugar si el conjunto estuviera ordenado por orden alfabético de nombres,

```
Cliente ultimo = Collections.max(conjuntoClie, new ComparaNombres());
```

obtendríamos a Marta.

Es importante resaltar que para llamar al método `max()` hace falta un criterio de ordenación, pero eso no implica que la colección tenga que estar ordenada. En ninguno de los dos ejemplos anteriores lo estaba.

Hay métodos análogos para calcular el mínimo de una colección, que funcionan exactamente igual:

```
Integer minimo = Collections.min(datos);
Cliente primero = Collections.min(conjuntoClie, new ComparaNombres());
```

También podemos invertir el orden de una lista con:

- `static void reverse(List<?> lista)`: invierte lista, colocando los elementos en orden inverso. La función no devuelve una nueva lista invertida, sino que invierte la lista original.

Por último, podemos crear un conjunto a partir de un elemento,

- `static <T> Set<T> singleton(T elem)`: devuelve un conjunto con `elem` como único elemento. Es un conjunto inmutable, es decir, no podemos añadir más elementos ni eliminar el que ya está. Se suele emplear para eliminar todas las ocurrencias de un elemento repetido de una lista sin necesidad de usar un bucle. Por ejemplo, para eliminar el 7, que aparece dos veces en `datos`, escribimos

```
datos.removeAll(Collections.singleton(7));
```

Ahora `datos` será:

```
[2, 1, 3]
```

### Actividad resuelta 12.13

Implementar la clase `Sorteo` con parámetros genéricos. Deberá guardar un conjunto de valores distintos de tipo genérico, suministrados por consola y será capaz de generar una combinación premiada de un tamaño determinado. Deberán implementarse, como mínimo, los métodos:

- `boolean add(T elemento)`, que añadirá un elemento nuevo al conjunto de valores posibles en una apuesta. Si el elemento se añade, devuelve `true` y, en caso contrario, debido a que ya estaba presente, `false`.
- `Set<T> premiados(int numPremiados)`, que devolverá una combinación ganadora de `numPremiados` elementos distintos.

**Solución**

```
/* Vamos a usar objetos ordenables en el sorteo para facilitar las lecturas por consola. Por la misma razón, y porque no admitimos elementos repetidos el conjunto de elementos los guardaremos en una estructura TreeSet.*/
public class Sorteo<T extends Comparable<T>> { //T ordenable
    private final Set<T> elementos;
    public Sorteo() {
        elementos = new TreeSet<>();
    }
    boolean add(T nuevo) {
        return elementos.add(nuevo);
    }
    /*Para escoger un subconjunto de numPremiados elementos al azar, los desordenamos todos y nos quedamos con los numPremiados primeros. Para poder desordenar con shuffle(), los pasamos temporalmente a lista. */
    Set<T> premiados(int numPremiados) {
        Set<T> premiados = null;
        List<T> temp = new ArrayList<>(elementos);
        Collections.shuffle(temp);
        if (numPremiados <= elementos.size()) {
            premiados = new TreeSet<>();
            for (int i = 0; i < numPremiados; i++) {
                premiados.add(temp.get(i));
            }
        }
        return premiados;
    }
    @Override
    public String toString() {
        return "Sorteo[elementos=" + elementos + ']';
    }
}

/*Código en Main para probar la clase Sorteo. Extraemos un conjunto de valores premiados:*/
public static void main(String[] args) {
    Sorteo<Integer> s = new Sorteo<>();
    for (int i = 0; i < 100; i++) {
        s.add(i);
    }
    System.out.println(s);
    System.out.println("Premiados: " + s.premiados(20));
}
```

**Actividad resuelta 12.14**

Implementar una aplicación que simula el registro de las temperaturas, a lo largo de un día, en una estación meteorológica. La aplicación mostrará un menú con las opciones:

1. Nuevo registro (que introduciremos manualmente, aunque se supone que, en el sistema original, estaría controlado por un reloj).
2. Listar registros.

3. Mostrar estadística (con los valores máximo, mínimo y promedio de las temperaturas registradas hasta el momento desde la primera lectura del día).
4. Salir.

Al salir, los datos se grabarán en un fichero binario cuyo nombre estará compuesto por la cadena «registros» concatenada con la fecha del día en el formato «yyyyMMdd» y extensión «.dat».

Cada registro constará de la temperatura en grados centígrados y la hora, que se leerá del sistema en el momento de la creación del registro.

### Solución

```
public class Registro implements Serializable {  
    LocalTime hora;  
    double temperatura;  
    public Registro(double temperatura) {  
        this.temperatura = temperatura;  
        this.hora = LocalTime.now();  
    }  
  
    public boolean equals(Object o) {  
        return hora.equals(((Registro) o).hora);  
    }  
  
    @Override  
    public String toString() {  
        DateTimeFormatter f=DateTimeFormatter.  
            ofLocalizedTime(FormatStyle.MEDIUM).  
            withLocale(Locale.getDefault());  
        return "Registro{" + "hora=" + hora.format(f) +  
            ", temperatura=" + temperatura + "°C}\n";  
    }  
}  
  
/*Programa principal:*/  
Set<Registro> temperaturas = new LinkedHashSet<>();  
int opcion;  
do {  
    System.out.println("1.Nuevo registro");  
    System.out.println("2.Listar registros del día");  
    System.out.println("3.Mostrar estadísticas");  
    System.out.println("4.Salir");  
    System.out.print("\nIntroducir opción: ");  
    opcion = new Scanner(System.in).nextInt();  
    switch (opcion) {  
        case 1 -> {  
            System.out.print("Introducir temperatura: ");  
            double temperatura = new Scanner(System.in).  
                useLocale(Locale.US).nextDouble();  
            temperaturas.add(new Registro(temperatura));  
        }  
        case 2 -> System.out.println(temperaturas);  
        case 3 -> {  
            Comparator<Registro> c = new Comparator<Registro>() {  
                @Override
```

```

        public int compare(Registro o1, Registro o2) {
            return (int) Math.signum(o1.temperatura -
o2.temperatura);
        }
    };
    System.out.println("Máxima: " +Collections.
max(temperaturas, c));
    System.out.println("Mínima: " + Collections.
min(temperaturas, c));
    double suma = 0;
    for (Registro t : temperaturas) {
        suma += t.temperatura;
    }
    System.out.println("Media: " + suma / temperaturas.size());
}
}
} while (opcion != 4);String nombreArchivo = "registros";
DateTimeFormatter f = DateTimeFormatter.ofPattern("yyyyMMdd");
nombreArchivo += LocalDate.now().format(f);
try(ObjectOutputStream out=new ObjectOutputStream(
new FileOutputStream(nombreArchivo))){
    out.writeObject(temperaturas);
} catch (FileNotFoundException ex) {
    System.out.println(ex);
} catch (IOException ex) {
    System.out.println(ex);
}
}

```

## 12.7. Interfaz Map

Los mapas o diccionarios son estructuras dinámicas cuyos elementos, que aquí se llaman **entradas** (objetos del tipo `Map.Entry`), son pares clave/valor en vez de valores individuales como en las colecciones. Todas ellas implementan la interfaz `Map`, que no hereda de `Collection`. Por tanto, los mapas no son colecciones, aunque están íntimamente relacionados con ellas y funcionan dentro del mismo entorno de trabajo. Vamos a usar tres implementaciones de `Map`: `HashMap`, `TreeMap` y `LinkedHashMap`, que se diferencian entre sí de forma similar a `HashSet`, `TreeSet` y `LinkedHashSet`.

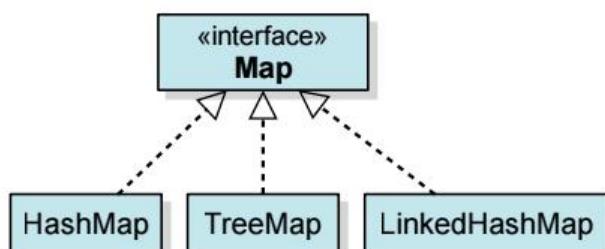


Figura 12.8. Clases más importantes que implementan la interfaz `Map`.

En un mapa se insertan entradas que constan de una clave, que no se puede repetir, y un valor asociado con ella, que sí puede estar repetido. Un mapa es una estructura semejante a la aplicación matemática. De hecho, la traducción al inglés de aplicación matemática es *mapping*.

Las operaciones fundamentales en un mapa son la inserción, la lectura y la eliminación de entradas, aunque veremos algunas más.

Para ilustrar el uso de mapas vamos a empezar utilizando la implementación `HashMap`, que no garantiza ningún orden de inserción de las entradas, aunque es muy eficiente en cuanto a la velocidad de acceso a los datos. El constructor más sencillo es de la forma

```
Map<K, V> m = new HashMap<>();
```

donde `K` es el tipo de las claves y `V`, el de los valores. Son tipos genéricos que, necesariamente, serán clases o interfaces y no tipos primitivos. Como hicimos con los conjuntos, hemos elegido `Map` como tipo de la variable `m` (podríamos haber puesto `HashMap`), con objeto de garantizar la posibilidad de un mayor polimorfismo. El comportamiento de `m` estará determinado por la clase del objeto referenciado. Como ejemplo, vamos a suponer que queremos mantener la información de las estaturas de un grupo de escolares, con entradas en las que figura el nombre del alumno (clase `String`) como clave y la estatura (clase envoltorio `Double`) como valor,

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

Para insertar entradas usamos el siguiente método:

`V put(K clave, V valor)`: se le pasan como parámetros la clave y el valor asociado con ella. Si no había ninguna entrada previa con la misma clave, se inserta en el mapa la nueva entrada con esa clave y ese valor, y el método devuelve `null`. Si ya había una entrada con la misma clave, se sustituye el valor antiguo por el nuevo, sin cambiar la clave, y la función devuelve el valor antiguo. Insertemos unas cuantas entradas:

```
m.put("Ana", 1.65);
m.put("Marta", 1.60);
m.put("Luis", 1.73);
m.put("Pedro", 1.69);
```

Con los mapas, igual que con los conjuntos, disponemos también de una implementación de `toString()`, de forma que podemos visualizarlos

```
System.out.println(m);
```

obteniéndose por pantalla,

```
{Marta=1.6, Ana=1.65, Luis=1.73, Pedro=1.69}
```

Si ahora queremos cambiar la estatura de Pedro, insertamos otra vez un elemento con la misma clave y el nuevo valor

```
m.put("Pedro", 1.71);
```

obteniéndose

```
{Marta=1.6, Ana=1.65, Luis=1.73, Pedro=1.71}
```

Si queremos eliminar un elemento:

- `V remove(Object k)`: elimina la entrada cuya clave es `k`, si existe. En este caso, devuelve el valor asociado con esa clave. En caso contrario, devuelve `null`.

Para eliminar todas las entradas de un mapa, llamamos a la función:

- `void clear()`: elimina todas las entradas, dejando el mapa vacío.

Si queremos conocer el valor de una entrada a partir de su clave,

- `V get(Object k)`: devuelve el valor asociado con la clave `k` o `null` si no hay ninguna entrada con esa clave.

Por ejemplo,

```
m.get("Ana");
```

devuelve 1,65.

Para saber si una determinada clave está presente en un mapa,

- `boolean containsKey(Object k)`: devuelve `true` si hay una entrada con la clave `k`.

Por ejemplo,

```
m.containsKey("Ana");
```

devolverá `true`.

Análogamente, para saber si hay alguna entrada con un valor determinado,

- `boolean containsValue(Object v)`: devuelve `true` si hay alguna entrada con valor `v`.

Dos mapas se pueden comparar entre sí con el método `equals()`, que devuelve `true` si ambos tienen exactamente las mismas entradas, independientemente del orden.

## ■ ■ ■ 12.7.1. Vistas Collection de los mapas

Aunque `Map` no hereda de `Collection`, los mapas están íntimamente ligados a las colecciones, de forma que se trabaja simultáneamente con ambas interfaces a través de distintas vistas con estructura de colección. Por vista de un mapa entendemos una colección respaldada por el mapa original, de forma que cuando accedemos a un elemento de la vista estamos accediendo a la entrada original en el mapa, y los cambios que se hagan en aquella se reflejarán en este. Hay tres tipos de vistas de un mapa. En primer lugar, podemos obtener una vista de las claves del mapa. Para ello disponemos del método:

- `Set<K> keySet()`: nos devuelve una vista, con estructura Set, de las claves presentes en un mapa.

Para obtener las claves del mapa del ejemplo escribimos:

```
Set<String> claves = m.keySet();
System.out.println(claves);
```

mostrará

```
[Marta, Ana, Luis, Pedro]
```

con corchetes, ya que es una colección.

También podemos obtener una vista de los valores del mapa.

- `Collection<V> values()`: devuelve una vista `Collection` de los valores. Si alguno se encuentra más de una vez en el mapa, también aparece repetido en la colección devuelta.

En nuestro ejemplo,

```
Collection<Double> estaturas = m.values();
System.out.println(estaturas);
```

devolverá

```
[1.6, 1.65, 1.73, 1.71]
```

Y, por último, disponemos de un método para obtener una vista de las entradas:

- `Set<Map.Entry<K, V>> entrySet()`: devuelve una vista conjunto de las entradas, objetos del tipo `Map.Entry`, de las que se puede obtener la clave con `getKey()` o el valor con `getValue()`. `Map.Entry` es una interfaz —no una clase— implementada en los elementos del mapa y del conjunto devuelto por `entrySet()`.

En nuestro mapa,

```
Set<Map.Entry<String, Double>> entradas = m.entrySet();
System.out.println(entradas);
```

se obtiene:

```
[Marta=1.6, Ana=1.65, Luis=1.73, Pedro=1.71]
```

Se puede usar la vista de entradas para acceder a las entradas individuales y obtener la clave o el valor, o bien para cambiar este último, con los métodos de la interfaz `Map.Entry`.

- `K getKey()`: devuelve la clave de la entrada.
- `V getValue()`: devuelve el valor de la entrada.
- `V setValue(V nuevoValor)`: asigna `nuevoValor` a la entrada y devuelve el valor antiguo.

Uno de los inconvenientes de los mapas es que no son iterables. Esto supone que, además de no poder usar iteradores para recorrerlos ni eliminar entradas, tampoco es posible el uso de la estructura `for-each`, cosa que sí podemos hacer con las vistas, que son colecciones.

Como hemos visto, los cambios que hagamos en ellas se reflejarán en el mapa. En particular, podemos eliminar entradas a través del conjunto de claves devuelto por `keySet()` con los métodos `remove()` de `Iterator`, `remove()` de `Collection`, `removeAll()` o `retainAll()`. Estos métodos, invocados por la vista, eliminarán las entradas correspondientes en el mapa. Por ejemplo, si eliminamos la clave «Marta» del conjunto claves,

```
claves.remove("Marta");
```

el mapa queda

```
{Ana=1.65, Luis=1.73, Pedro=1.71}
```

donde vemos que la entrada correspondiente a Marta ha desaparecido.

La única forma segura de eliminar entradas durante un proceso de iteración sobre cualquiera de las tres vistas es el método `remove()` de la interfaz `Iterator`. Veamos un ejemplo, pero antes vamos a añadir algunas entradas a nuestro mapa:

```
m.put("Lucas", 1.8);
m.put("Marta", 1.60);
m.put("Jorge", 1.75);
```

con lo que tenemos:

```
{Marta=1.6, Ana=1.65, Luis=1.73, Lucas=1.8, Pedro=1.71, Jorge=1.75}
```

Ahora vamos a filtrar el mapa eliminando todos aquellos alumnos con estatura mayor que 1,71. Para ello iteramos sobre el conjunto de las entradas:

```
Set<Map.Entry<String, Double>> entradas = m.entrySet(); /*Set de entradas */
Iterator<Map.Entry<String, Double>> it; //iterador de entradas
for (it = entradas.iterator(); it.hasNext();) {
    Map.Entry<String, Double> e = it.next();
    if (e.getValue() > 1.71) {
        it.remove();
    }
}
```

Obtendremos:

```
{Marta=1.6, Ana=1.65, Pedro=1.71}
```

Esto mismo podríamos haberlo hecho iterando sobre la vista de los valores.

```
Collection<Double> estaturas = m.values();
for (Iterator<Double> it = estaturas.iterator(); it.hasNext();) {
    Double v = it.next();
    if (v > 1.71) {
        it.remove();
    }
}
```

En cambio, no podemos añadir entradas a un mapa con `add()` o `addAll()` a través de ninguna de sus vistas. La única forma es con el método `put()` de la interfaz `Map`.

## 12.7.2. Implementaciones de Map

En nuestro ejemplo hemos utilizado la implementación `HashMap`, que destaca por su eficiencia, pero que no garantiza ningún orden en la inserción de las entradas. La interfaz `Map` tiene otras dos implementaciones, `TreeMap` y `LinkedHashMap`.

`TreeMap`, a semejanza de `TreeSet`, tiene una estructura de árbol que permite una inserción ordenada y una búsqueda rápida y eficiente de las entradas. Estas se insertan por orden natural creciente de las claves.

Por ejemplo,

```
TreeMap<String, Double> tm = new TreeMap<>();
```

```
tm.put("Ana", 1.65);
tm.put("Marta", 1.60);
tm.put("Luis", 1.73);
tm.put("Pedro", 1.71);
```

El mapa tm quedará:

```
{Ana=1.65, Luis=1.73, Marta=1.6, Pedro=1.71}
```

Podemos hacer que el orden de un TreeMap sea distinto. Para ello le pasamos un comparador al constructor como parámetro de entrada, igual que hacíamos con TreeSet. En cualquier caso, el orden siempre se refiere a las claves, nunca a los valores.

Por último, la implementación LinkedHashMap mantiene el orden en que se van insertando las entradas, de forma similar a lo que ocurre con LinkedHashSet. Es muy eficiente en las operaciones de inserción y eliminación de entradas y algo más lento en las búsquedas.

## Actividad resuelta 12.15

Implementar una aplicación para gestionar las existencias de una tienda de repuestos de automóviles. Cada producto se identifica por un código alfanumérico. La aplicación permitirá dar de alta o de baja productos y actualizar el número de unidades en stock de cada uno de ellos. Los datos se mantendrán en un fichero, que deberá actualizarse al cerrar el programa.

### Solución

```
/*Implementamos un mapa con un TreeSet que mantiene un orden basado en los
códigos*/
Map<String, Integer> existencias = new TreeMap<>();
try ( ObjectInputStream in = new ObjectInputStream(
    new FileInputStream("existencias.dat"))) {
    existencias = (TreeMap<String, Integer>) in.readObject();
} catch (FileNotFoundException ex) {
    System.out.println(ex);
} catch (IOException | ClassNotFoundException ex) {
    System.out.println(ex);
}
int opcion;
do {
    System.out.println("1.Alta producto");
    System.out.println("2.Baja producto");
    System.out.println("3.Cambio stock de producto");
    System.out.println("4.Listar existencias");
    System.out.println("5.Salir");
    System.out.print("\nIntroducir opción: ");
    opcion = new Scanner(System.in).nextInt();
    switch (opcion) {
        case 1 -> {
            System.out.print("Código producto: ");
            String codigo = new Scanner(System.in).next();
            /*Antes de dar de alta un código debemos asegurarnos
            de que no existe, ya que machacaría su valor: */
            if (!existencias.containsKey(codigo)) {
```

```

        existencias.put(codigo, 0);
    } else {
        System.out.println("El producto ya existe");
    }
}

case 2 -> {
    System.out.print("Código producto: ");
    String codigo = new Scanner(System.in).next();
    existencias.remove(codigo);
}

case 3 -> {
    System.out.print("Código producto: ");
    String codigo = new Scanner(System.in).next();
    System.out.print("Nuevo stock: ");
    int stock = new Scanner(System.in).nextInt();
    existencias.put(codigo, stock);
}

case 4 -> {
    System.out.println(existencias);
}

}

} while (opcion != 5);

try ( ObjectOutputStream out = new ObjectOutputStream(
        new FileOutputStream("existencias.dat"))
) {
    out.writeObject(existencias);
}

catch (FileNotFoundException ex) {
    System.out.println(ex);
}

catch (IOException ex) {
    System.out.println(ex);
}
}

```

### Actividad resuelta 12.16

Los miembros de la Real Academia de la Lengua ocupan sillones con las letras del abecedario español, minúsculas y mayúsculas (en la práctica, las letras v, w, x, y, z, Ñ, W, Y nunca se ocupan, pero nosotros no lo tendremos en cuenta). Cuando un sillón queda vacante, se nombra un nuevo académico para ocuparlo.

Implementar la clase `Academico`, cuyos atributos son el nombre y el año de ingreso. El criterio de ordenación natural será por nombres.

Implementar un programa donde se crean cinco objetos `Academico`, que se insertan en un mapa en el que la clave es la letra del sillón que ocupan, y el valor un objeto de la clase `Academico`. Para ello implementar el método estático:

```
static boolean nuevoAcademico(Map<Character, Academico> academia, Academico  
nuevo, Character letra),
```

donde se lleva a cabo la inserción después de comprobar que el carácter pasado como parámetro es una letra del abecedario.

Hacer diversos listados de los académicos: primero sin letra, por orden de nombre y de año de ingreso; y después con letra, por orden de letra (clave), nombre y fecha de ingreso. Debemos recordar que, en código Unicode, las mayúsculas van antes que las minúsculas.

**Solución**

```

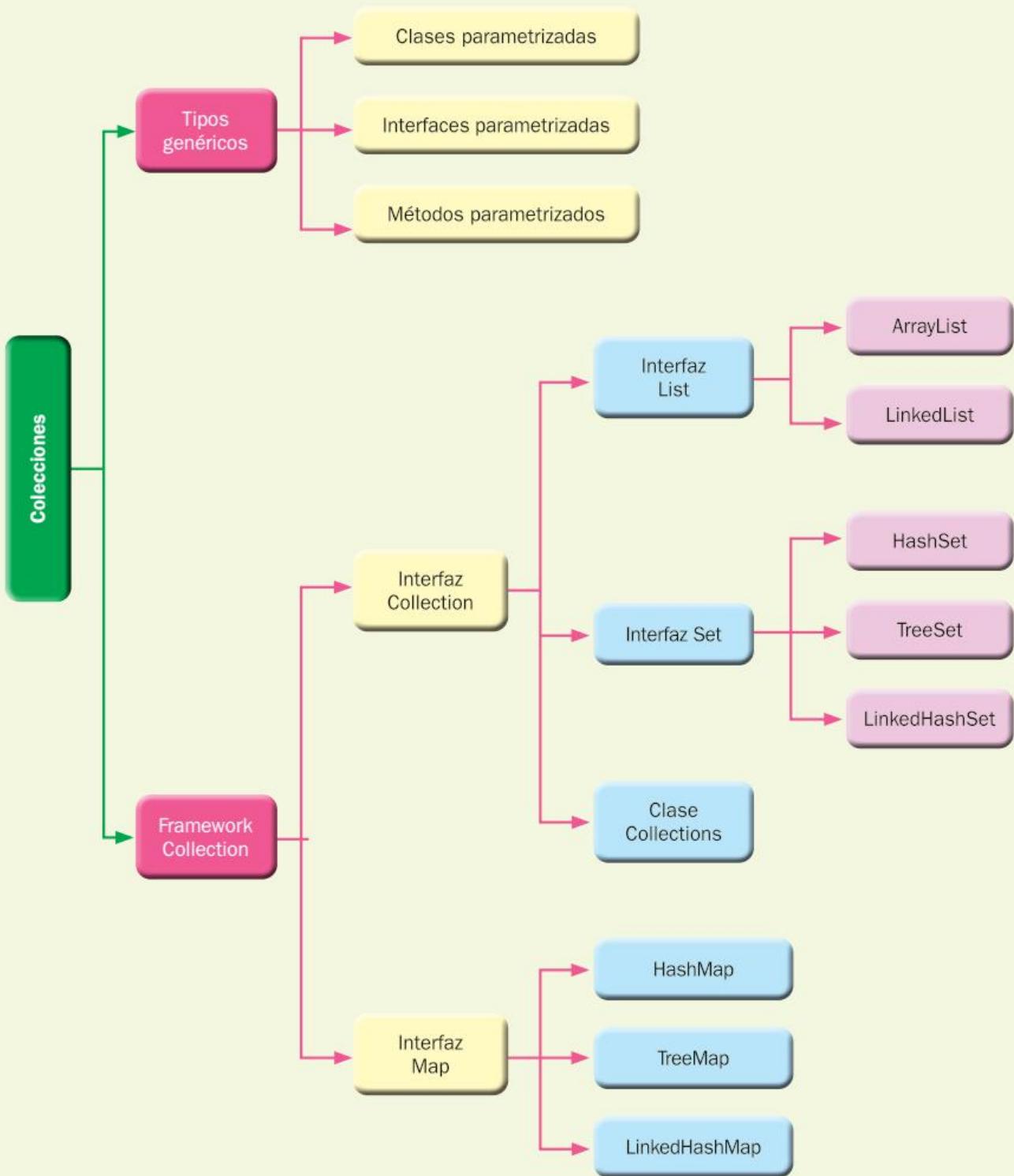
public class Academico implements Comparable<Academico> {
    String nombre;
    int aIngreso;
    public Academico(String nombre, int aIngreso) {
        this.nombre = nombre;
        this.aIngreso = aIngreso;
    }
    @Override
    public int compareTo(Academico o) {
        return nombre.compareTo(o.nombre);
    }
    @Override
    public String toString() {
        return "Academico{" + "nombre=" + nombre
            + ", año de ingreso=" + aIngreso + "}\n";
    }
}

/*Programa principal: Con TreeMap tenemos un mapa ordenado por las claves
(la letra), donde vamos a insertar cinco académicos */
Map<Character, Academico> academia = new TreeMap<>();
for (int i = 0; i < 5; i++) {
    System.out.print("Letra: ");
    Character letra = new Scanner(System.in).next().charAt(0);
    System.out.print("Nombre: ");
    String nombre = new Scanner(System.in).next();
    System.out.print("Año de ingreso: ");
    int ingreso = new Scanner(System.in).nextInt();
    nuevoAcademico(academia, new Academico(nombre, ingreso), letra);
}
System.out.println("Orden por letra: " + academia);
/*Para ordenar por los valores, tenemos que obtener una vista del mapa. Si
nos conformamos con mostrar solo los valores (nombre y año de ingreso de los
académicos, bastará una vista de los valores, transformada en lista para poder
ordenar: */
Collection<Academico> sinLetra = academia.values();
List<Academico> listaSinLetra = new ArrayList<>(sinLetra);
Collections.sort(listaSinLetra);
System.out.println("Por nombre sin letra: " + listaSinLetra);
/*por año de ingreso: */
Comparator<Academico> comparaIngresos = new Comparator<>() {
    @Override
    public int compare(Academico o1, Academico o2) {
        return o1.aIngreso - o2.aIngreso;
    }
};
Collections.sort(listaSinLetra, comparaIngresos);
System.out.println("Por año sin letra: " + listaSinLetra);
/*Si queremos que aparezca la clave (la letra) trabajaremos con una vista de
las entradas: */
Set<Map.Entry<Character, Academico>> conLetra = academia.entrySet();
/*Convertimos en lista para ordenar las entradas:*/
List<Map.Entry<Character, Academico>> listaConLetra
    = new ArrayList<>(conLetra);

```

```
/*Ordenamos por año de ingreso: */
Collections.sort(listaConLetra,
    new Comparator<>() { /*el tipo se infiere de listaConLetra*/
        @Override
        public int compare(Map.Entry<Character, Academico> o1,
                           Map.Entry<Character, Academico> o2) {
            return o1.getValue().aIngreso - o2.getValue().aIngreso;
        }
    });
System.out.println("Orden por año de ingreso: " + listaConLetra);
/*Ordenamos por orden natural (nombres) de los académicos*/
Collections.sort(listaConLetra,
    new Comparator<>() {/*el tipo se infiere de listaConLetra*/
        @Override
        public int compare(Map.Entry<Character, Academico> o1,
                           Map.Entry<Character, Academico> o2) {
            return o1.getValue().compareTo(o2.getValue());
        }
    });
System.out.println("Orden por nombre: " + listaConLetra);

static boolean nuevoAcademico(Map<Character, Academico> academia, Academico
nuevo, Character letra) {
    boolean insertado = false;
    if ((letra >= 'A' && letra <= 'Z')
        || (letra >= 'a' && letra <= 'z')
        || letra == 'ñ' || letra == 'Ñ') {
        academia.put(letra, nuevo);
        insertado = true;
    } else {
        System.out.println("Letra no válida");
    }
    return insertado;
}
```



## Actividades de comprobación

**12.1. ¿Qué es Collection?**

- a) Una interfaz.
- b) Una clase.
- c) Un sistema operativo.
- d) Un método.

**12.2. Los tipos genéricos sirven para:**

- a) Usar objetos de la clase Object.
- b) Usar variables primitivas.
- c) Usar tipos parametrizados.
- d) No tener que usar ningún tipo.

**12.3. ¿Para qué sirve una lista?**

- a) Guardar datos primitivos.
- b) Guardar datos que no se pueden repetir.
- c) No tener que ordenar un conjunto de datos.
- d) Guardar, de forma dinámica, datos que se pueden repetir y ordenar.

**12.4. Un conjunto es una colección de elementos:**

- a) Que no admiten orden.
- b) Que admiten repeticiones.
- c) Que no se pueden alterar.
- d) Cuyo criterio fundamental es el de pertenecer al conjunto.

**12.5. ArrayList y LinkedList se diferencian:**

- a) En el número de elementos.
- b) En el rendimiento.
- c) En el orden de los elementos.
- d) En nada.

**12.6. Los métodos de la interfaz Set:**

- a) Son los mismos que los de List.
- b) Son los mismos que los de Collection.
- c) Son implementados en la clase ArrayList.
- d) Esta interfaz no tiene métodos.

**12.7. Si la variable a referencia un objeto ArrayList, la expresión new TreeSet(a):**

- a) Devuelve un conjunto ordenado con los elementos de a.
- b) Es incorrecta.
- c) Devuelve una lista ordenada.
- d) Devuelve una tabla.

**12.8. ¿Qué es Collections?**

- a) Una clase cuyos objetos están repetidos.
- b) Una interfaz de la que heredan todas las colecciones.
- c) Una clase con métodos estáticos que sirven para gestionar colecciones.
- d) Nada, le sobra la ese.

**12.9. Un mapa en Java es:**

- a) Un gráfico con las relaciones de herencia entre interfaces.
- b) Una colección.
- c) Una representación de los datos por pantalla.
- d) Una estructura dinámica cuyos elementos son parejas clave-valor.

**12.10. Si queremos cambiar el valor de una entrada en un mapa, usaremos el método:**

- a) put().
- b) set().
- c) add().
- d) insert().

## Actividades de aplicación

**12.11.** Utilizando la clase `Contenedor` definida en la Actividad resuelta 12.2, implementa una aplicación donde se guardan 30 enteros aleatorios entre 1 y 10 y luego se ordenan de menor a mayor. La aplicación debe mostrar el contenedor antes y después de ordenar.

**12.12.** Añade a la clase `Contenedor` el método

```
void ordenar(Comparator<T> c),
```

que ordena los elementos del contenedor según el criterio de `c`.

**12.13.** Repite la Actividad de aplicación 12.11 ordenando los números de mayor a menor.

**12.14.** Añade a la clase `Contenedor` el método

```
T get(int indice),
```

que devuelve el elemento que ocupa el lugar `indice` dentro del contenedor.

**12.15.** Implementa un método genérico al que se le pasa una lista de valores de la clase genérica `T` y devuelve otra donde se han eliminado las repeticiones.

**12.16.** Implementa una aplicación que gestione los socios de un club usando la clase `Socio` implementada en la Actividad resuelta 12.11. En particular, se deberán ofrecer las opciones de alta, baja y modificación de los datos de un socio. Además, se listarán los socios por nombre o por antigüedad en el club.

**12.17.** Implementa la clase `Cola` genérica utilizando un objeto `ArrayList` para guardar los elementos.

**12.18.** Implementa la clase `Pila` genérica utilizando un objeto `ArrayList` para guardar los elementos.

**12.19.** Escribe un programa donde se introduzca por consola una frase que conste exclusivamente de palabras separadas por espacios. Las palabras de la frase se almacenarán en una lista. Finalmente, se mostrarán por pantalla las palabras que estén repetidas y, a continuación, las que no lo estén.

**12.20.** Utilizando colecciones, implementa la clase `Supercola`, que tiene como atributos dos colas para enteros, en las que se encola y desencola por separado. Sin embargo, si una de las colas queda vacía, al llamar a su método `desencolar()`, se desencola de la otra mientras tenga elementos. Solo cuando las dos colas estén vacías, cualquier llamada a `desencolar` devolverá `null`. Escribe un programa con el menú:

1. Encolar en `cola1`.
2. Encolar en `cola2`.
3. Desencolar de `cola1`.
4. Desencolar de `cola2`.
5. Salir.

Después de cada operación se mostrará el estado de las dos colas para seguir su evolución.

**12.21.** Implementa una aplicación donde se insertan 20 números enteros aleatorios distintos, menores que 100, en una colección. Deberán guardarse por orden decreciente a medida que se vayan generando, y se mostrará la colección resultante por pantalla.

**12.22.** Introduce por teclado, hasta que se introduzca «fin», una serie de nombres, que se insertarán en una colección, de forma que se conserve el orden de inserción y que no puedan repetirse. Al final, la colección se mostrará por pantalla.

**12.23.** Repite la Actividad de aplicación 12.22 de forma que se inserten los nombres manteniendo el orden alfabético.

**12.24.** Implementa una función a la que se le pasen dos listas de enteros ordenadas en sentido creciente y nos devuelva una única lista, también ordenada, fusión de las dos anteriores. Desarrolla el algoritmo de forma no destructiva, es decir, que las listas utilizadas como parámetros de entrada se mantengan intactas.

**12.25.** Implementa una aplicación que gestione un club donde se identifica a los socios por un apodo personal y único. De cada socio, además del apodo, se guarda el nombre y su fecha de ingreso en el club. Utiliza un mapa donde las claves serán los apodos y los valores, objetos de la clase `Socio`. Los datos se guardarán en un fichero llamado «club.dat», de donde se leerá el mapa al arrancar y donde se volverá a guardar actualizado al salir. Las operaciones se mostrarán en un menú que tendrá las siguientes opciones:

1. Alta socio.
2. Baja socio.
3. Modificación socio.
4. Listar socios por apodo.
5. Listar socios por antigüedad.
6. Listar los socios con alta anterior a un año determinado.
7. Salir.

**12.26.** Un centro educativo necesita distribuir de forma aleatoria a los alumnos de un curso entre los grupos disponibles para ese curso. Diseña la función

```
List<List<String>> repartoAlumnos(List<String> lista, int numGrupos)
```

que devuelve una lista de listas, cada una de las cuales corresponde a un grupo.

Cada nombre de la lista de alumnos se asigna a uno de los grupos.

### Actividades de ampliación

- 12.27. Implementa la función `leeCadena()`, con el siguiente prototipo:

```
List<Character> leeCadena(),
```

que lee una cadena por teclado y nos la devuelve en una lista con un carácter en cada elemento.

- 12.28. Implementa la función `uneCadenas`, con el siguiente prototipo:

```
List<Character> uneCadenas(List<Character> cad1, List<Character> cad2)
```

que devuelve una lista con la concatenación de `cad1` y `cad2`.

- 12.29. Añade a la clase `Contenedor` para tipos genéricos los métodos:

- `int[] buscarTodos(Object e)`: devuelve una tabla con los índices de todas las ocurrencias de `e`.
- `boolean eliminarTodos(Object e)`: elimina todas las ocurrencias de `e`. Devuelve `true` si la lista queda alterada.

- 12.30. Implementa una función

```
<T> List<T> eliminaRepetidos(List<T> lista)
```

a la que se pase una lista de objetos y devuelva una copia sin elementos repetidos.

- 12.31. Implementa las clases `Cola` y `Pila` genéricas heredando de `ArrayList`.

- 12.32. Implementa la función

```
static <E> List<E> clonaLista(List<E>)
```

que realice una copia exacta de una lista.

- 12.33. Define la clase `ListaOrdenada`, que almacena una serie de objetos de tipo genérico `E`, de forma ordenada, pudiéndose repetir. Los elementos se ordenarán según el orden natural de `E` o bien con el criterio de orden definido en un comparador que se le pasa al constructor.

- 12.34. Amplía la Actividad resuelta 12.14, de forma que se gestionen los registros de temperatura de diferentes días en la misma aplicación. Para ello, implementa un mapa cuyas entradas tendrán como clave la fecha y como valor el conjunto con los registros de un día. Implementa también un programa que gestione los registros del día actual y permita visualizar los de un día cualquiera, junto con sus estadísticas. Al arrancar el programa se cargará en memoria el mapa a partir del fichero correspondiente y, al terminar, se volverá a guardar actualizado.

- 12.35. Con la clase `Jornada` definida en la Actividad de ampliación 9.28, implementa una aplicación que gestione las jornadas de los trabajadores de una empresa por medio de colecciones, incluyendo altas y bajas de trabajadores y altas de jornadas, así como el listado de las jornadas de un trabajador. Los datos se guardarán en un fichero binario.

- 12.36. Repite la Actividad de ampliación 9.32 utilizando una colección para guardar y manipular las `Llamadas`.

**12.37.** Queremos gestionar la plantilla de un equipo de fútbol, en la que a cada jugador se le asigna un dorsal que no puede estar repetido. Para ello vamos a crear una estructura de tipo `Map` cuyas entradas corresponden a los jugadores, con el dorsal como clave y un objeto de la clase `Jugador` como valor. De cada jugador se guarda el DNI, el nombre, la posición en el campo —para simplificar, los jugadores pueden ser porteros, defensas, centrocampistas y delanteros— y su estatura.

Define la clase `Jugador` y un enumerado para la posición en el campo, e implementa los siguientes métodos estáticos:

- `static void altaJugador(Map<Integer, Jugador> plantilla, Integer dorsal)`, que añade una entrada al mapa con el dorsal pasado como parámetro y el jugador creado dentro del método, introduciendo sus datos por consola.
- `static Jugador eliminarJugador(Map<Integer, Jugador> plantilla, Integer dorsal)`, que elimina la entrada correspondiente al jugador cuyo dorsal se pasa como parámetro. Dicho dorsal desaparece del mapa hasta que se asigne a otro jugador por medio de un alta. El método devuelve el jugador eliminado.
- `static void mostrar(Map<Integer, Jugador> plantilla)`, que muestra una lista de los dorsales con los nombres de los jugadores correspondientes.
- `static void mostrar(Map<Integer, Jugador> plantilla, String posicion)`, que muestra una lista de los jugadores que comparten una misma posición. Por ejemplo, todos los defensas o todos los delanteros.
- `static boolean editarJugador(Map<Integer, Jugador> plantilla, Integer dorsal)`, que permite modificar los datos de un jugador, excepto su dorsal y su DNI. Devuelve `true` si el dorsal existe y `false` en caso contrario.