

MP0485
Programación
UF7. Clases de uso general

7.2. Colecciones

Índice

☰	Objetivos	3
☰	Introducción	4
☰	Colecciones y mapas: estructura	7
☰	Las clases Vector y ArrayList	10
☰	LinkedList: listas enlazadas	25
☰	Interfaz Set	36
☰	Conjuntos sin orden: HashSet	38
☰	Conjuntos ordenados: TreeSet	42
☰	Los mapas	46
☰	HashMap	48
☰	TreeMap	53
☰	Ejemplo práctico: cuenta bancaria	55
☰	Amplía conocimientos	58
☰	¿Cómo elegir el tipo de colección?	59
☰	Resumen	60

Objetivos

Con esta unidad perseguimos los siguientes objetivos:

1

Crear programas Java utilizando el grupo de clases que implementan la interfaz *Collection*, es decir, colecciones de tipo lista (*List*) y conjunto (*Set*).

2

Crear programas Java utilizando el grupo de clases que implementan la interfaz *Map*, es decir, colecciones de datos donde cada elemento tiene una clave única.

3

Crear aplicaciones Java basadas en clases con interrelaciones de tipo agregación o composición con cardinalidad 1:0..* utilizando listas, conjuntos o mapas para guardar los objetos contenidos en el objeto contenedor.

¡Ánimo y adelante!

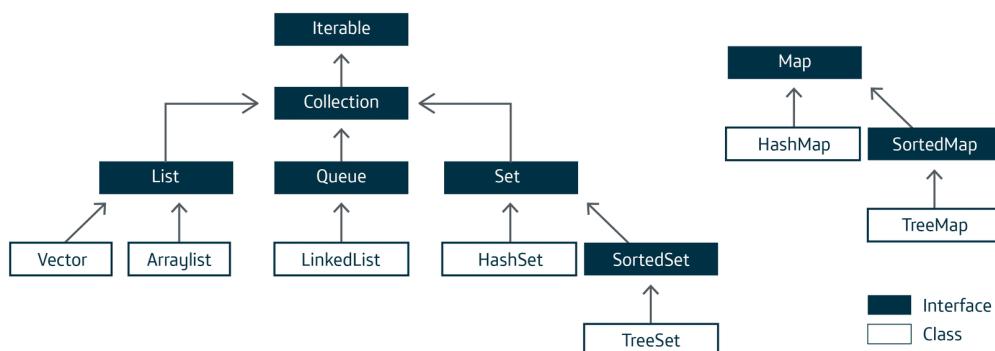
Introducción

Los arrays nos ofrecen una interesante forma de estructurar datos en la memoria, pero tienen el problema de que es necesario saber previamente la longitud o número de elementos que tendrá cada array.

Para resolver este problema podemos utilizar otro recurso que nos ofrece Java;
las clases que representan colecciones
y las clases que representan mapas.

Colecciones y mapas sirven para almacenar en la memoria un grupo o conjunto de elementos, al igual que los *arrays*, pero tienen la gran ventaja de crecer dinámicamente según las necesidades del programa en cada momento.

Java cuenta con muchísimas clases que sirven para representar colecciones de datos y mapas, clases que forman parte de una compleja jerarquía. En la siguiente imagen te mostramos la jerarquía que corresponde a las clases con las que vamos a trabajar:



Estructura jerárquica de las interfaces *Coleccion* y *Map*.

En la imagen hemos simplificado el árbol jerárquico de las colecciones y mapas, eliminando algunas ramas y dejando lo más relevante. Lo importante es que comprendas que cuando utilizas una clase, esta no está aislada, sino que forma parte de una estructura jerárquica más compleja.

Ten en cuenta que en la imagen los rectángulos con fondo negro son interfaces y los rectángulos con fondo blanco son clases. Así puedes fácilmente deducir que la clase *Vector* implementa la interfaz *List*, e indirectamente, también las interfaces *Collection* e *Iterable*. También puedes comprobar que la clase *LinkedList* implementa las interfaces *List* y *Queue*, y también, de manera indirecta, las interfaces *Collection* e *Iterable*.

Observa además que las clases que implementan la interfaz *Map* no implementan las interfaces *Collection* e *Iterable*, es decir, están separadas y no forman parte de la misma estructura o familia jerárquica. Aunque esto es relativo, porque ya sabes que todas las clases heredan de la superclase *Object*, así que de alguna manera cualquier clase está unida a otra a través de la superclase *Object*.

Diferencia entre una colección y un mapa

Tanto colecciones como mapas representan un conjunto o colección de elementos, pero hay una importante diferencia entre ambos:

1

Los mapas no implementan las interfaces *Iterable* y *Collection*, tal como puedes apreciar en la imagen. En el siguiente apartado aprenderás qué aportan estas interfaces a las clases que las implementan.

2

Técnicamente llamamos colecciones a las clases que implementan la interfaz *Collection*. Aunque en la práctica también se llama colecciones a los mapas, porque representan una colección o conjunto de elementos. De ahí que ambos conceptos se confundan a menudo.

3

La diferencia importante a nivel operativo es que, aunque tanto mapas como colecciones almacenan una colección o conjunto de objetos, **los mapas asocian una clave a cada elemento, que servirá después como llave para acceder a dicho elemento**.

Colección de triángulos		
lado1	lado2	lado3
5	5	5
5	5	8
5	8	9
5	2	7
1	2	8

Mapa de triángulos				
Clave		lado1	lado2	lado3
T1	→	5	5	5
T2	→	5	5	8
T3	→	5	8	9
T4	→	5	2	7
T5	→	1	2	8

La clave de cada elemento del mapa es como el carnet de identidad de una persona. Es única para cada elemento y sirve para identificarlo inequívocamente.

Tipos de colecciones

Observando de nuevo la imagen, puedes apreciar que hay colecciones que implementan la interfaz *List* (*Vector*, *ArrayList* y *LinkedList*) y colecciones que implementan la interfaz *Set* (*HashSet*, *TreeSet*). Clasificaremos las colecciones según se trate de listas o conjuntos.

1

Listas: objetos que pertenecen a una clase que implementa la interfaz *List*. Los elementos que se añaden a una lista estarán siempre situados según el orden en que se han añadido. Se admiten valores duplicados.

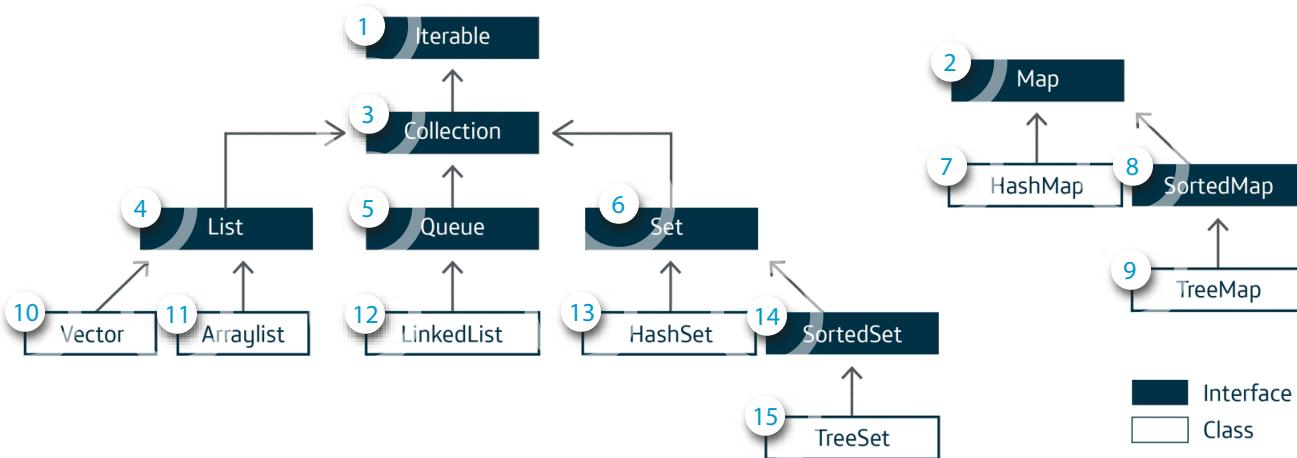
2

Conjuntos: objetos que pertenecen a una clase que implementa la interfaz *Set*, inspirada en la teoría de conjuntos de matemáticas. Los elementos que se añaden no guardan ningún orden específico, ni se sitúan en el orden en que se van introduciendo. La característica más importante de los conjuntos es que **no admiten valores duplicados**.

En el siguiente apartado estudiaremos un poco más detenidamente la jerarquía de las colecciones y mapas.

Colecciones y mapas: estructura

Todas las clases e interfaces que se encuentran en este esquema pertenecen al paquete `java.util`, a excepción de la interfaz `Iterable`, que se encuentra en `java.lang`.



1 Iterable

Las clases que implementan la interfaz `Iterable` tienen la característica de que sus elementos pueden recorrerse con una estructura de tipo `foreach` y además poseen el método `iterator()`. Trabajaremos con estos conceptos más adelante.

2 Map

Los mapas asocian una clave a cada elemento, que servirá después como llave para acceder a dicho elemento.

3 Collection

Las clases que implementan la interfaz `Collection` representan una colección dinámica de objetos. Internamente trabajan con `arrays`.

4 List

Interfaz que implementa las colecciones de datos que **se organizan en forma de lista, es decir, un elemento detrás de otro** en el orden en que se van introduciendo.

5 Queue

Las clases que implementan la interfaz **Queue** representan colecciones cuyos elementos **se organizan en forma de cola**.

Una cola es una estructura de datos en memoria RAM donde el primer elemento que entró, es el primero en salir. También se denominan estructuras FIFO, abreviatura del inglés “First In, First Out”.

6 Set

Conjuntos: los elementos que se añaden no se sitúan en el orden natural en que se van introduciendo. La característica más importante de los conjuntos es que **no admiten valores duplicados**.

7 HashMap

Colección de elementos donde cada elemento tiene asociada una clave. Los elementos no se sitúan en el orden en que se van añadiendo.

8 SortedMap

Interfaces cuyas clases que la implementan representan mapas cuyos elementos quedan ordenados según la clave.

9 TreeMap

Colección de elementos donde cada elemento tiene asociada una clave. Los elementos se sitúan en orden de clave.

10 Vector

Lista basada en un array tipo vector con una longitud inicial de 10 elementos, que irá creciendo en 10 elementos más cuando sea necesario, permitiendo que crezca dinámicamente. **Sus métodos son sincronizados**, es decir, es útil cuando trabajamos con programas multitarea utilizando distintos hilos de ejecución.

11 ArrayList

Lista basada en un array tipo vector con una longitud inicial de 10 elementos, que irá creciendo en 10 elementos más cuando sea necesario, permitiendo que crezca dinámicamente. A diferencia de la clase *Vector*, sus métodos no son sincronizados.

12 LinkedList

Basado en una lista enlazada y con los elementos organizados en forma de cola. Mucho más eficiente que *Vector* y *ArrayList* cuando hay que hacer muchas inserciones y eliminaciones.

13 HashSet

Colección de objetos que no guardan ningún tipo de orden. **Los elementos no se sitúan en el mismo orden que se han ido introduciendo.** Puesto que se trata de un conjunto, **no admite duplicados**, es decir, no admite ningún nuevo elemento que, aplicándole el método *equals* sobre cualquiera de los existentes, dé *true*.

14 SortedSet

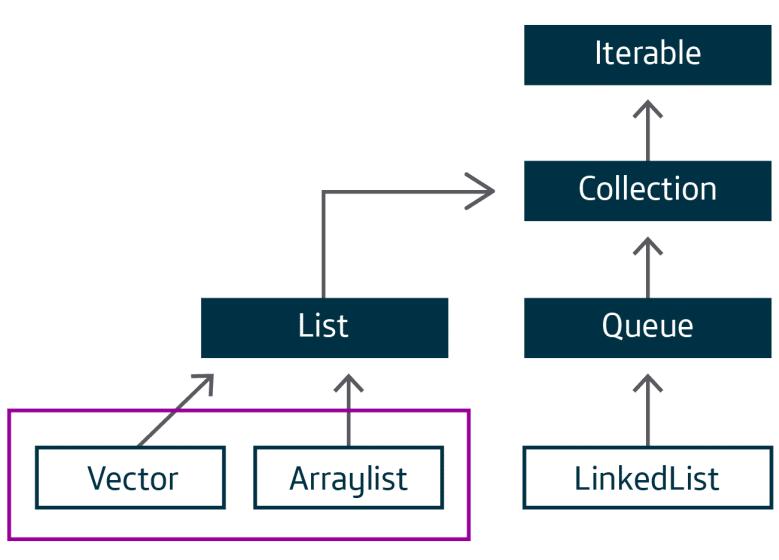
Las clases que implementan esta interfaz representan **colecciones de objetos que pueden ordenarse en función del método compareTo** de cada elemento con el resto de los elementos, y además **no admiten duplicados**, puesto que se trata de conjuntos.

15 TreeSet

Colección de objetos que **pueden ordenarse** en función del método *compareTo* de cada elemento con el resto de los elementos y además, **no admiten duplicados**, es decir, no admite ningún nuevo elemento que, aplicándole el método *equals* sobre cualquiera de los existentes, dé *true*.

Las clasees Vector y ArrayList

En este apartado estudiaremos las clases *ArrayList* y *Vector*. Ambas clases están basadas en un *array* que tiene la capacidad de crecer dinámicamente.



Listas basadas en arrays

Las clases *ArrayList* y *Vector* son muy similares, ya que ambas basan su estructura de datos interna en un *array* que crece dinámicamente.

Si no se especifica lo contrario, inicialmente crean un *array* de 10 elementos, que duplicará su tamaño cuando el número de elementos sobrepase su capacidad. Es posible indicar en el constructor el número de elementos iniciales. Además, el constructor de *Vector* también admite un segundo argumento para especificar el incremento.

Constructores principales de la clase *Vector* y *ArrayList*:

```
Vector()  
Vector(int initialCapacity)  
Vector(int initialCapacity, int capacityIncrement)  
  
ArrayList()  
ArrayList(int initialCapacity)
```

La diferencia entre **Vector** y **ArrayList** es que un **vector** tiene propiedad de sincronización, algo útil para la concurrencia, es decir, cuando realizamos programas multitarea utilizando distintos hilos de ejecución, ya que garantiza mayor seguridad.

ArrayList no es sincronizado, por lo que resulta más eficiente si no estamos trabajando con hilos y métodos sincronizados. Los ejemplos que veremos a lo largo de este apartado estarán basados en la clase **ArrayList**, pero podrías aplicarlos igual con un objeto de la clase **Vector**.

ArrayList y **Vector** son clases genéricas; luego hay que especificar la clase a la que pertenecen los elementos que portará la lista. Veamos varios ejemplos:

```
Vector<String> textos = new Vector<String>();
```

Lista inicial de 10 elementos de tipo *String*. En el momento que se añada el elemento 11, se añadirán otros 10, quedando la longitud del *array* en 20 elementos.

```
Vector<Triangulo> trians = new Vector<Triangulo>(15, 5);
```

Lista inicial de 15 elementos de tipo *Triangulo*. En el momento en que se añada el elemento 16, se añadirán 5 elementos más.

```
ArrayList<Triangulo> trians = new ArrayList<Triangulo>(15);
```

Lista inicial de 15 elementos de tipo *Triángulo*. En el momento en que se añade el elemento 16, se duplicará el array. En este caso, los métodos del objeto *trians* no son sincronizados.

i **Recuerda:** las clases genéricas tienen la posibilidad de declarar una o varias propiedades, cuyo tipo puede variar. El tipo de dicha propiedad será establecido en el momento de crear un objeto mediante la notación <tipo>. Si lo necesitas, puedes volver a revisar la documentación sobre **Clases genéricas** en la unidad anterior.

Ejemplo práctico

Crearemos una lista de objetos *Triángulo*. Para ello, comienza por crear un nuevo proyecto en Eclipse y añade la clase *Triángulo* con el siguiente código:

```
public class Triangulo {  
    private int lado1;  
    private int lado2;  
    private int lado3;  
  
    public Triangulo(int lado1, int lado2, int lado3) {  
        this.lado1 = lado1;  
        this.lado2 = lado2;  
        this.lado3 = lado3;  
    }  
  
    @Override  
    public String toString() {  
        return "Triangulo [" + lado1 + ", " + lado2 + ", " + lado3  
    }  
  
    public String vertTipo() {  
        if (this.lado1==this.lado2&&this.lado2==this.lado3) {  
            return this.toString() + " Equilátero";  
        }  
        else if (this.lado1==this.lado2||this.lado2==this.lado3||th  
            return this.toString() + " Isósceles";  
        }  
        else {  
            return this.toString() + " Escaleno";  
        }  
    }  
}
```

Ahora ya estamos preparados para construir una lista de objetos *Triangulo*.

OJO: igual que ocurría con los arrays, en las colecciones el primer elemento ocupa la posición 0.

```
import java.util.ArrayList;

public class Principal {
    public static void main(String[] args) {
        ArrayList<Triangulo> trians = new ArrayList<Triangulo>();

        // Añadimos tres elementos a la lista
        trians.add(new Triangulo(1, 2, 3));
        trians.add(new Triangulo(1, 1, 2));
        trians.add(new Triangulo(1, 1, 1));

        // trians.size() nos devuelve el número total de elementos
        System.out.println("Nº de elementos guardados: " + trians.size());
        System.out.println("Accediendo a la posición 1");
        // trians.get(1) nos devuelve el elemento que ocupa la posición 1
        System.out.println(trians.get(1).verTipo());

        System.out.println("Recorriendo todos los elementos");
        for (int i=0; i<trians.size(); i++) {
            System.out.println(trians.get(i).verTipo());
        }
    }
}
```

Nuestro objeto `ArrayList` tiene tres elementos que están situados en las posiciones 0, 1 y 2.

Vamos a estudiar los métodos más importantes de las clases `Vector` y `ArrayList`. Así comprenderás mejor el ejemplo anterior.

Algunos de los métodos que verás a continuación son comunes a todas las clases que implementan la interfaz `Collection` y otros son propios exclusivamente de la interfaz `List`.

Algunos métodos de la interfaz `Collection`

int size()

El método *size* devuelve el número de elementos que contiene la lista. En el ejemplo anterior estamos utilizando este método para controlar cuándo deber terminar el bucle *for*.

boolean add(Object element)

Añade el elemento pasado como argumento al final de la lista. Devuelve *true* si el elemento se ha añadido con éxito y devuelve *false* si, por alguna circunstancia, el elemento no se ha podido añadir.

boolean remove(Object obj)

Elimina de la colección el objeto pasado como argumento. Si el objeto no ha podido eliminarse por cualquier circunstancia, devuelve *false*, de lo contrario devuelve *true*.

void clear()

Elimina todos los elementos dejando la colección vacía.

boolean contains(Object o)

Devuelve *true* si la colección contiene el objeto especificado como argumento, de lo contrario devuelve *false*.

boolean isEmpty()

Devuelve *true* si la colección no contiene ningún elemento, de lo contrario devuelve *false*.

Algunos métodos de la interfaz *List* que no pertenecen a *Collection*

Object get(int pos)

El método *get* devuelve el elemento que ocupa la posición especificada como argumento. En el anterior ejemplo lo estamos utilizando para recuperar el objeto *Triangulo* que ocupa la posición 1 y después para recorrer cada uno de los objetos *Triangulo* por medio de un bucle *for*.

boolean add(int pos, Object element)

También es posible añadir un nuevo elemento en la posición deseada. Puedes probar a añadir el siguiente código al ejemplo:

```
trians.add(1, new Triangulo(7, 8, 9));
System.out.println("Recorriendo de nuevo todos los elementos");
for (int i=0; i<trians.size(); i++) {
    System.out.println(trians.get(i).verTipo());
}
```

El nuevo triángulo de lados 7, 8 y 9 es insertado en la posición 1 y los triángulos que antes ocupaban las posiciones 1 y 2 han sido desplazados una posición.

Object remove(int pos)

Elimina el elemento que ocupa la posición especificada en el argumento y retorna el elemento borrado. Puedes ponerlo en práctica añadiendo este código:

```
Triangulo t = trians.remove(2);
System.out.println("Se ha eliminado el triángulo: " + t.verTipo());
```

Se ha eliminado el elemento que ocupaba la posición indicada y los elementos que le seguían han tenido que desplazarse una posición hacia arriba.



Todas las clases que implementan la interfaz *Collection* también implementan indirectamente la interfaz *Iterable*, lo que convierte a sus objetos en colecciones de elementos iterables, es decir, que se pueden recorrer secuencialmente.

Y, ¿cómo podemos iterar?

Anteriormente ya has iterado de la siguiente forma:

```
for (int i=0; i<trians.size(); i++) {  
    System.out.println(trians.get(i).verTipo());  
}
```

Pero hay más formas de iterar, presta atención:

Iterar con la estructura *for each*

La estructura *for each* (para cada) es un formato especial de la instrucción *for* que permite repetir un grupo de sentencias tantas veces como elementos tenga una colección o un *array*. El formato es el siguiente:

```
for (Object obj : colección_o_array) {  
    // Bloque de sentencias  
}
```

Por ejemplo, para recorrer los elementos de una colección de objetos *Triangulo*:

```
import java.util.ArrayList;

public class Principal {
    public static void main(String[] args) {
        ArrayList<Triangulo> trians = new ArrayList<Triangulo>();

        trians.add(new Triangulo(1, 2, 3));
        trians.add(new Triangulo(1, 1, 2));
        trians.add(new Triangulo(1, 1, 1));

        for (Triangulo t : trians) {
            System.out.println(t);
        }
    }
}
```

El bucle se ejecuta tantas veces como elementos tenga la colección *trians*. En cada iteración guardará el elemento correspondiente en la variable *t*.

Iterar con el método *iterator()*

El método *iterator()* retorna un objeto de la clase *Iterator* que provee de un mecanismo para recorrer secuencialmente los elementos de una colección a través de un cursor. Llamamos cursor a una marca que se va desplazando y que va posicionándose en el siguiente elemento a leer dentro de la colección.

Observa este ejemplo:

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

public class Principal {
    public static void main(String[] args) {
        ArrayList<Triangulo> trians = new ArrayList<Triangulo>();

        trians.add(new Triangulo(1, 2, 3));
        trians.add(new Triangulo(1, 1, 2));
        trians.add(new Triangulo(1, 1, 1));

        Iterator<Triangulo> itera = trians.iterator();
        Triangulo cadaTriangulo;
        while (itera.hasNext()) {
            cadaTriangulo = itera.next();
            System.out.println(cadaTriangulo.verTipo());
        }
    }
}
```

Analicemos el código anterior:

Primero, hemos tenido que obtener el objeto *Iterator* asociado a la colección *trians*.

Iterator<Triangulo> itera = trians.iterator();

Nuestro nuevo objeto *Iterator* se llama *itera* y estamos utilizándolo para acceder secuencialmente a los elementos de la colección *trians*, de manera similar a como se recorren los registros de un fichero secuencial. Para lograrlo utilizamos los siguiente **métodos de la clase *Iterator*:**

boolean hasNext()

Esta función devuelve *true* mientras sigan quedando elementos para iterar en la colección. En el ejemplo lo estamos utilizando para ejecutar el bucle mientras haya más elementos que recorrer.

Object next()

Devuelve el siguiente elemento de la colección.

```
CadaTriangulo = itera.next();
```

Esta sentencia devuelve el objeto *Triangulo* que corresponde a la posición donde se encuentra el cursor de lectura.

```
while (itera.hasNext()) {  
    cadaTriangulo = itera.next();  
    System.out.println(cadaTriangulo.verTipo());  
}
```

La expresión *Itera.hasNext()* devuelve el valor *true* si el puntero no se encuentra al final de la colección, es decir, si todavía hay elementos que leer. La estructura *while* nos está permitiendo acceder secuencialmente a cada elemento mientras queden más elementos para leer.

 **Recuerda:** podrás iterar con cualquier tipo de colección que derive de la interfaz *Iterator*.

Pero, ¿cuándo utilizo la estructura *for ... each*
y cuándo un objeto *Iterator*?



La estructura *for ... each* parece más sencilla cuando solo quiero recorrer los elementos, por ejemplo, para realizar un listado. Pero si lo que deseamos es eliminar de la colección los elementos que cumplan una determinada condición, entonces podemos encontrarnos con una sorpresa.

Observa detenidamente el siguiente ejemplo donde, dentro de una colección de nombres de personas, queremos eliminar a aquellas personas que se llaman Carmen.

```
import java.util.ArrayList;

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

        nombres.add("Carmen");
        nombres.add("Rosa");
        nombres.add("Carmen");
        nombres.add("Miguel");
        nombres.add("Carmen");

        for (String n : nombres) {
            System.out.println(n);
            if (n.equals("Carmen")) {
                nombres.remove(n);
            }
        }
    }
}
```

Parece lógico y sencillo. Sin embargo, este programa provoca error de ejecución porque no permite eliminar el elemento sobre el que se está iterando.



Exception in thread "main" java.util.ConcurrentModificationException
at java.util.ArrayList\$Itr.checkForComodification(ArrayList.java:901)
at java.util.ArrayList\$Itr.next(ArrayList.java:851)
at Principal.main(Principal.java:14)

El método *Iterator.remove()*

Los objetos de la clase *Iterator* poseen el método *remove()*, que elimina el objeto que acaba de iterarse. Esto vendría a resolver el problema del ejemplo anterior.

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

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

        nombres.add("Carmen");
        nombres.add("Rosa");
        nombres.add("Carmen");
        nombres.add("Miguel");
        nombres.add("Carmen");

        Iterator<String> itera = nombres.iterator();
        String cadaNombre;
        while (itera.hasNext()) {
            cadaNombre = itera.next();
            System.out.println(cadaNombre);
            if (cadaNombre.equals("Carmen")) {
                itera.remove();
            }
        }

        System.out.println("Recorrido después de borrar las Carmene
for (String n : nombres) {
    System.out.println(n);

}
}
```

Para saber más ...

Más sobre la interfaz **Collection**

Pulsa el botón de la derecha para acceder a la documentación oficial de Oracle sobre la interfaz *Collection*.

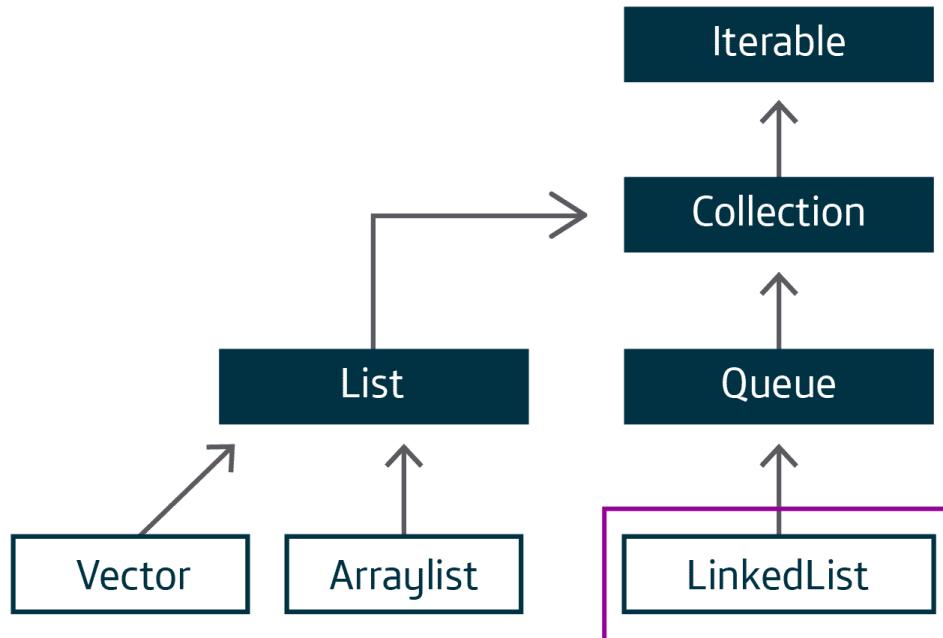
[ACCEDER](#)

Más sobre la interfaz **List**

Pulsa el botón de la derecha para acceder a la documentación oficial de Oracle sobre la interfaz *List*.

[ACCEDER](#)

LinkedList: listas enlazadas



En este apartado estudiaremos la clase *LinkedList*. Observando el diagrama, puedes comprobar que también deriva de *List*, *Collection* e *Iterable*. Todos los ejemplos que has realizado en el apartado anterior seguirían funcionando exactamente igual cambiando el tipo de referencia *ArrayList* por *LinkedList*, ya que comparten la mayoría de los métodos.

¿Exactamente igual?

Parece que funciona igual pero, en realidad, una colección *LinkedList* es diferente a una colección *ArrayList*. Los elementos están organizados en memoria RAM de una manera muy distinta. Descubriremos las ventajas de uno sobre otro.

Puedes poner en práctica estos dos ejemplos que ya utilizamos anteriormente, pero ahora con objetos *LinkedList*.

```
import java.util.LinkedList;

public class Principal {
    public static void main(String[] args) {
        LinkedList<String> nombres = new LinkedList<String>();

        nombres.add("Carmen");
        nombres.add("Rosa");
        nombres.add("Carmen");
        nombres.add("Miguel");
        nombres.add("Carmen");

        for (String n : nombres) {
            System.out.println(n);
        }
    }
}
```

Lista enlazada de objetos *String*.

```
import java.util.LinkedList;

public class Principal {
    public static void main(String[] args) {
        LinkedList<Triangulo> trians = new LinkedList<Triangulo>();

        trians.add(new Triangulo(1, 2, 3));
        trians.add(new Triangulo(1, 1, 2));
        trians.add(new Triangulo(1, 1, 1));

        for (int i=0; i<trians.size(); i++) {
            System.out.println(trians.get(i).verTipo());
        }
    }
}
```

Lista enlazada de objetos *Triangulo*.

Hay dos características muy importantes que hacen de *LinkedList* un tipo de colección muy especial:

1

Un objeto de tipo *LinkedList* es una lista enlazada.

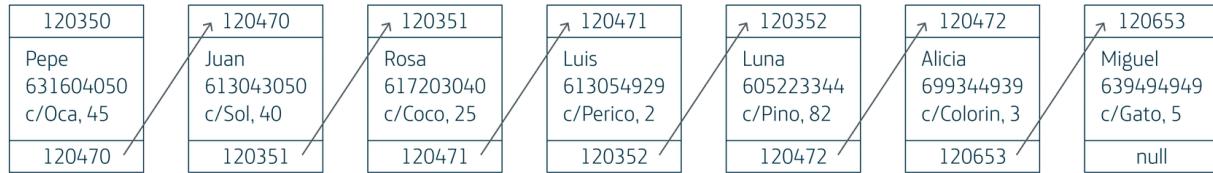
2

Un objeto de tipo *LinkedList* es una cola, característica que está obligada a cumplir al implementar la interfaz *Queue*.

Las listas enlazadas son estructuras de datos organizadas como conjuntos de registros en memoria RAM, donde cada registro está situado a partir de una determinada dirección de memoria y enlazado con el siguiente.

Cada registro contendrá, además de los datos necesarios (en el ejemplo de la imagen: nombre, teléfono y dirección), la dirección de memoria del siguiente registro, excepto en el último registro, que contendrá *null*.

Direcciones de memoria



Lista enlazada.

Existen listas enlazadas y listas doblemente enlazadas, donde cada elemento contiene la dirección de memoria, no solo del siguiente elemento, sino también del anterior. Los objetos *LinkedList* representan listas doblemente enlazadas.

¿Cuándo usar *LinkedList* y cuándo *ArrayList*?

El hecho de que los objetos *LinkedList* sean listas enlazadas resulta, en ocasiones, una ventaja y en otras, una desventaja.

1

Cuando insertamos un elemento en una posición determinada dentro de un *ArrayList*, el resto de los elementos deben desplazarse. Recuerda que un *ArrayList* está basado en un *array* interno cuyos elementos están situados en posiciones contiguas de memoria. Si el *array* es muy grande el tiempo de proceso puede ser grande. Si se añade un elemento al final no hay ningún problema.

Con *LinkedList* insertar un elemento no conlleva nada de tiempo, porque los elementos no ocupan posiciones contiguas de memoria, sino que pueden estar ubicados en cualquier parte, con tal de que apunten al anterior elemento y al siguiente elemento.

Es más eficiente usar *LinkedList* cuando tenemos una lista grande en la que hay que realizar muchas inserciones en cualquier posición.

2

Cuando eliminamos un elemento de posiciones intermedias en un *ArrayList*, todos los elementos que siguen deben recolocarse para no dejar huecos. Si el *array* es muy grande el tiempo de proceso puede ser un problema. Con un *LinkedList* esto no ocurre; si se elimina un elemento intermedio basta con hacer que el anterior apunte al siguiente.

Es más eficiente usar *LinkedList* cuando tenemos una lista grande, donde hay que realizar muchas eliminaciones en cualquier posición.

3

Cuando un objeto *ArrayList* necesita crecer porque se ha llegado al final del *array*, el sistema debe crear un nuevo *array* más grande y copiar todos los elementos del *array* antiguo. Esto no ocurre con un objeto *LinkedList*, ya que no está basado en *arrays*. Si vamos a utilizar un *ArrayList* y sabemos de antemano que crecerá mucho, puede ser interesante crearlo de antemano de un tamaño grande.

Los objetos *LinkedList* pueden crecer todo lo que sea necesario sin peligrar el rendimiento.

4

Si necesitamos acceder aleatoriamente a cualquier elemento, con *ArrayList* se accede directamente al elemento buscado sin tener que recorrer todos los elementos (acceso aleatorio).

```
buscado = lista.get(3);
```

Esta sentencia recupera el tercer elemento de un *ArrayList* sin necesidad de recorrer los elementos anteriores. La misma sentencia aplicada a un *LinkedList* también funciona, pero internamente tiene que recorrer todos los elementos hasta llegar al que ocupa la posición 3.

El acceso a un *LinkedList* solo se realiza secuencialmente, mientras que el acceso a un *ArrayList* puede ser aleatorio.

En conclusión: un *ArrayList* es mejor cuando tenemos una lista grande y debemos realizar muchas búsquedas aleatorias, pero no vamos a realizar inserciones o borrados.

LinkedList es mejor cuando tenemos una lista grande donde debemos realizar muchas inserciones y borrados.

LinkedList es también una cola. Una estructura de datos en memoria RAM donde el primer elemento que entró, es el primero en salir.

También se denominan **estructuras FIFO**, abreviatura del inglés “First In, First Out”.

ENTRADA

Pepe
631604050
c/Oca, 45

Juan
613043050
c/Sol, 40

Rosa
617203040
c/Coco, 25

Luis
613054929
c/Perico, 2

Luna
605223344
c/Pino, 82

Alicia
699344939
c/Colorin, 3

Miguel
639494949
c/Gato, 5

SALIDA

Para demostrar la forma en la que los objetos de tipo *LinkedList* se sitúan formando una estructura FIFO o cola puedes ejecutar este programa:

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

public class Principal {
    public static void main(String[] args) {
        Queue<String> mensajes = new LinkedList<String>();

        // Entrada de los elementos en la cola.
        mensajes.add("La cripta mágica");
        mensajes.add("El perro de San Roque");
        mensajes.add("Mente sana, cuerpo sano");

        // Salida de los elementos de la cola.
        while (!mensajes.isEmpty()) {
            String sms = mensajes.remove();
            System.out.println(sms);
        }
    }
}
```

El método *remove()* elimina o saca un elemento de la colección y lo devuelve. En nuestro ejemplo, la sentencia "String sms = mensajes.remove();" saca un elemento de la salida de la cola, pero antes lo almacena en la variable *sms*.

Con la sentencia

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

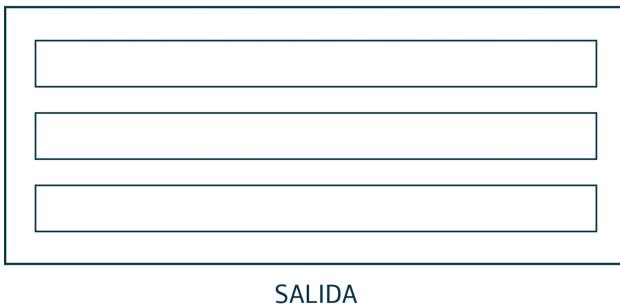
estamos aplicando el concepto de polimorfismo, ya que una referencia a un objeto *Queue* puede ser un objeto *LinkedList*. También podríamos haberlo hecho así:

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

El siguiente gráfico animado te ilustra sobre cómo funciona el anterior programa:

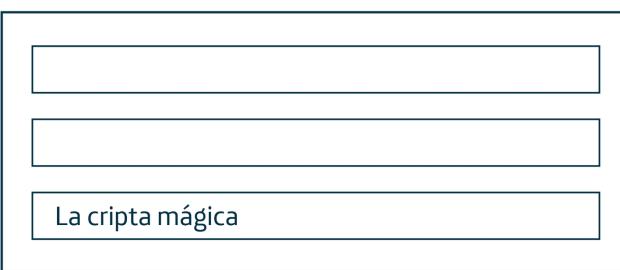
La cripta mágica

ENTRADA



SALIDA

ENTRADA



SALIDA

El perro de San Roque

ENTRADA



SALIDA

ENTRADA

El perro de San Roque

La cripta mágica

SALIDA

Mente sana, cuerpo sano

ENTRADA

El perro de San Roque

La cripta mágica

SALIDA**ENTRADA**

Mente sana, cuerpo sano

El perro de San Roque

La cripta mágica

SALIDA

ENTRADA

Mente sana, cuerpo sano

El perro de San Roque

SALIDA

La cripta mágica

ENTRADA

Mente sana, cuerpo sano

El perro de San Roque

SALIDA

ENTRADA

Mente sana, cuerpo sano

SALIDA

El perro de San Roque

ENTRADA

Mente sana, cuerpo sano

SALIDA

ENTRADA

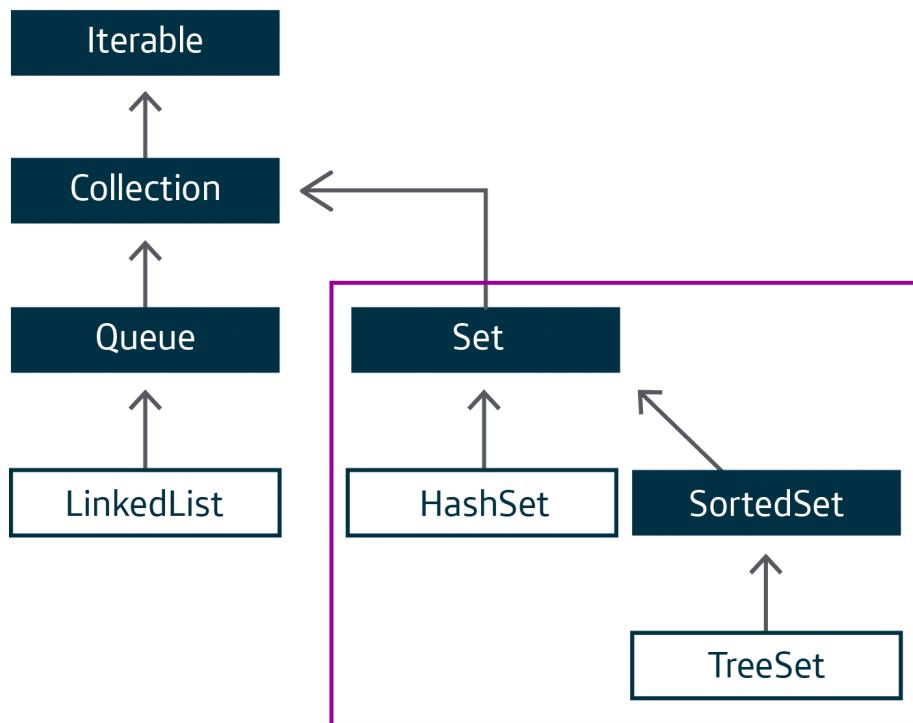
SALIDA

Mente sana, cuerpo sano

Interfaz Set

Todas las clases que derivan directa o indirectamente de la interfaz *Set* pertenecen a un tipo especial de colecciones llamadas *conjuntos*. Estas colecciones están basadas en la teoría de conjuntos de matemáticas.

La característica más importante de este tipo de colecciones es que **no admiten duplicados**.



Interfaz Set

Trabajaremos con las clases **HashSet** y **TreeSet**. Observa que derivan de *Collection* e *Iterable*, por lo que puedes deducir que puedes iterar con una estructura *for ... each*, usar el método *iterator()* y otros métodos como *add()*, *remove()*, *clear()*, etc. Sin embargo, no podrás utilizar el método *get()*, propio de la interfaz *List*.

¿Y cuál es la diferencia entre *HashSet* y *TreeSet*?

1

Ambas clases no admiten elementos duplicados, ya que son conjuntos. Para evitar los duplicados, por cada nuevo elemento que se va a almacenar se comprueba que la función *equals()* sobre el nuevo elemento comparado con todos los anteriores devuelva siempre el valor *false*. Recuerda que dos objetos se consideran iguales cuando tienen el mismo *hashCode*. Si crees que lo necesitas, repasa de nuevo el apartado sobre los métodos *hashcode()* y *equals()* de la unidad anterior.

2

En las colecciones de tipo *HashSet* los elementos no se colocan en el orden en que han sido insertados; no se puede garantizar en qué orden se recuperan los elementos.

3

En las colecciones de tipo *TreeSet* los elementos se ordenan atendiendo al criterio que dicta el método *compareTo()*.

Conjuntos sin orden: HashSet

En este apartado practicaremos con la clase *HashSet*.

Una colección de tipo *HashSet* representa un conjunto de elementos del mismo tipo, donde no pueden existir duplicados y los elementos no guardan el orden en que se introdujeron.

Para ponerlo en práctica ejecuta el siguiente programa:

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

public class Principal {
    public static void main(String[] args) {
        Set<String> nombres = new HashSet<String>();

        nombres.add("Rosa");
        nombres.add("Carlos");
        nombres.add("Miguel");
        nombres.add("Carlos"); // Este no se añadirá.
        nombres.add("Sole");
        nombres.add("Adrián");
        nombres.add("Angel");
        nombres.add("Amelia");
        nombres.add("Fernando");
        nombres.add("Sebas");
        nombres.add("Lucas");

        for (String n : nombres) {
            System.out.println(n);
        }
    }
}
```

Hemos añadido dos veces el nombre "Carlos", sin embargo, el segundo no se añadirá. Observa también que, a la hora de mostrar los elementos en pantalla con la estructura *for*, no se recuperan en el mismo orden en el que se insertaron.

Con las cadenas de texto está muy claro cuándo hay un duplicado, pero, **¿qué pasa con los objetos *Triangulo*?** **¿Cuándo se considera que dos objetos son iguales?** La respuesta está en los métodos *hashCode()* y *equals()*.

Recuerda que dos objetos se consideran iguales si la expresión *obj1.equals(obj2)* es *true*, y será *true* solo si tienen el mismo *hashCode*. Cada clase puede sobrescribir los métodos *hashCode()* y *equals()*. En conclusión; el programador que implementa una clase puede decidir qué criterio utiliza para decidir que dos objetos de dicha clase son iguales.

Vamos a modificar la clase *Triangulo* para **considerar que dos triángulos son iguales si la suma de sus lados es igual**.

```
public class Triangulo {  
    private int lado1;  
    private int lado2;  
    private int lado3;  
  
    public Triangulo(int lado1, int lado2, int lado3) {  
        this.lado1 = lado1;  
        this.lado2 = lado2;  
        this.lado3 = lado3;  
    }  
  
    @Override  
    public String toString() {  
        return "Triangulo [" + lado1 + ", " + lado2 + ", " + lado3  
    }  
  
    public String verTipo() {  
        if (this.lado1==this.lado2&&this.lado2==this.lado3) {  
            return this.toString() + " Equilátero";  
        }  
        else if (this.lado1==this.lado2||this.lado2==this.lado3||th  
            return this.toString() + " Isósceles";  
        }  
        else {  
            return this.toString() + " Escaleno";  
        }  
    }  
  
    @Override  
    public int hashCode() {  
        return this.lado1+this.lado2+this.lado3;  
    }  
    @Override  
    public boolean equals(Object obj) {  
        if (this.hashCode() == obj.hashCode())  
            return true;  
        else  
            return false;  
    }  
}
```

Ahora podemos crear una colección de objetos *Triangulo* donde no puedan existir dos triángulos cuya suma de los lados sea igual.

```
import java.util.HashSet;

public class Principal {
    public static void main(String[] args) {
        HashSet<Triangulo> trians = new HashSet<Triangulo>();

        trians.add(new Triangulo(1, 1, 1));
        trians.add(new Triangulo(2, 2, 1));
        trians.add(new Triangulo(3, 1, 1)); // No se guarda, se con-
        trians.add(new Triangulo(7, 1, 3));
        trians.add(new Triangulo(9, 8, 7));
        trians.add(new Triangulo(9, 8, 7));
        trians.add(new Triangulo(5, 5, 2));
        trians.add(new Triangulo(1, 1, 2));

        for (Triangulo t : trians) {
            System.out.println(t.verTipo());
        }
    }
}
```

De nuevo observa en el resultado de la ejecución que los objetos *Triangulo* mostrados no guardan el mismo orden en que se han ido añadiendo y, además, el tercer triángulo con lados 3, 1, 1 no se ha añadido.

Conjuntos ordenados: TreeSet

La clase *TreeSet* funciona igual que la clase *HashSet*, pero en este caso los elementos se ordenan según el criterio que dicte el método *compareTo()*. Además, la clase *TreeSet* implementa la interfaz *SortedSet*.

Puedes probar el mismo ejemplo de los nombres del apartado anterior, pero ahora con un objeto *TreeSet*.

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

public class Principal {
    public static void main(String[] args) {
        Set<String> nombres = new TreeSet<String>();

        nombres.add("Rosa");
        nombres.add("Carlos");
        nombres.add("Miguel");
        nombres.add("Carlos"); // Este no se añadirá.
        nombres.add("Sole");
        nombres.add("Adrián");
        nombres.add("Angel");
        nombres.add("Amelia");
        nombres.add("Fernando");
        nombres.add("Sebas");
        nombres.add("Lucas");

        for (String n : nombres) {
            System.out.println(n);
        }
    }
}
```

Comprueba que, igual que antes, no admite duplicados, pero ahora además los elementos están ordenados alfabéticamente.

¿Y qué pasa con los objetos?

Está muy claro cómo deberían ordenarse un grupo de objetos *String* (alfabéticamente) o un grupo de objetos *Integer* (numéricamente de menor a mayor). Sin embargo, no resulta tan claro cómo deben ordenarse los objetos. En este caso, es el programador que desarrolla la clase quien debe decidir qué tipo de ordenación resulta más interesante, y para ello debe implementar el método *compareTo()* de la interfaz *Comparable*.

Para implementar el método *compareTo()* hay que respetar las siguientes reglas:

1

Dada la expresión *obj1.compareTo(obj2)*, devolverá un número positivo si *obj1* es mayor que *obj2*.

2

Dada la expresión *obj1.compareTo(obj2)*, devolverá un número negativo si *obj1* es menor que *obj2*.

3

Dada la expresión *obj1.compareTo(obj2)*, devolverá un cero si *obj1* es igual que *obj2*.

Vamos a implementar el método *compareTo()* en la clase *Triangulo* para que los triángulos se ordenen según la suma de sus lados. Para lograrlo, la clase *Triangulo* deberá implementar la interfaz *Comparable*, así que lo primero que haremos será modificar la cabecera de la clase de la siguiente manera:

```
public class Triangulo implements Comparable<Triangulo> {  
}
```

Además, la interfaz *Comparable* es genérica, por eso incluimos la expresión *<Triangulo>*.

La nueva clase *Triangulo* quedará así:

```
public class Triangulo implements Comparable<Triangulo> {
    private int lado1;
    private int lado2;
    private int lado3;

    public Triangulo(int lado1, int lado2, int lado3) {

        this.lado1 = lado1;
        this.lado2 = lado2;
        this.lado3 = lado3;
    }

    @Override
    public String toString() {
        return "Triangulo [" + lado1 + ", " + lado2 + ", " + lado3
    }

    public String vertTipo() {
        if (this.lado1==this.lado2&&this.lado2==this.lado3) {
            return this.toString() + " Equilátero";
        }
        else if (this.lado1==this.lado2||this.lado2==this.lado3||th
            return this.toString() + " Isósceles";
        }
        else {
            return this.toString() + " Escaleno";
        }
    }

    @Override
    public int hashCode() {
        return this.lado1+this.lado2+this.lado3;
    }
    @Override
    public boolean equals(Object obj) {
        if (this.hashCode() == obj.hashCode())
            return true;
        else
            return false;
    }

    @Override
    public int compareTo(Triangulo o) {
        if (this.hashCode()==o.hashCode())
            return 0;
        else if (this.hashCode()>o.hashCode())
            return 1;
        else
            return -1;
    }
}
```

Ahora puedes crear una colección de objetos *Triangulo* exactamente igual que lo hiciste con un *HashSet*, pero ahora con un objeto *TreeSet*.

```
import java.util.TreeSet;

public class Principal {
    public static void main(String[] args) {
        TreeSet<Triangulo> trians = new TreeSet<Triangulo>();

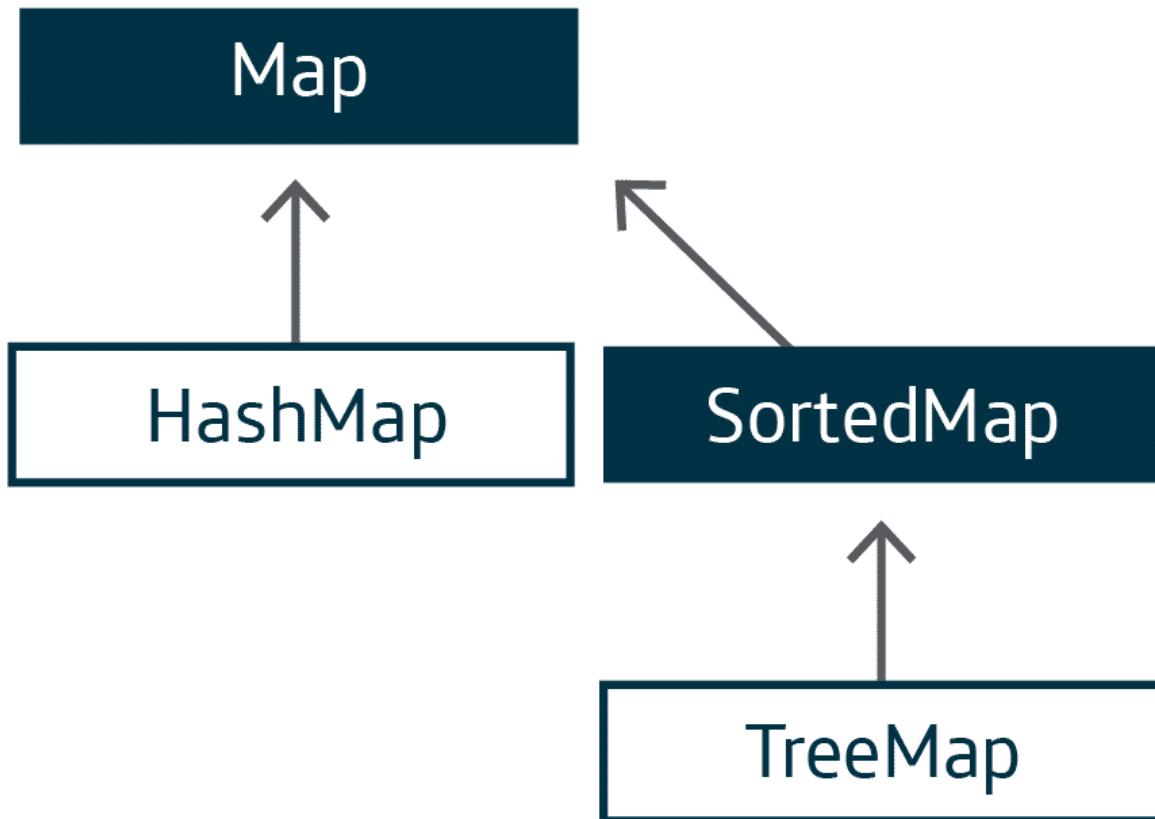
        trians.add(new Triangulo(1, 1, 1));
        trians.add(new Triangulo(2, 2, 1));
        trians.add(new Triangulo(3, 1, 1)); // No se guarda, se con-
        trians.add(new Triangulo(7, 1, 3));
        trians.add(new Triangulo(9, 8, 7));
        trians.add(new Triangulo(9, 8, 7));
        trians.add(new Triangulo(5, 5, 2));
        trians.add(new Triangulo(1, 1, 2));

        for (Triangulo t : trians) {
            System.out.println(t.verTipo());
        }
    }
}
```

Si has ejecutado ya, habrás comprobado que sigue sin admitir duplicado. El triángulo de lados 3, 1, 1 no ha sido añadido, pero ahora, además, los triángulos se muestran ordenados ascendente mente según la suma de sus lados.

Los mapas

Los mapas son un tipo especial de colecciones que asocian una clave a cada elemento, que servirá después como llave para acceder a dicho elemento. Las clases que representan mapas derivan de la interfaz *Map*.



En este apartado trabajaremos con las clases *HashMap* y *TreeMap*. Es importante tener en cuenta que estas clases **no derivan de Iterable y Collection**, por lo que muchos de los métodos que te has acostumbrado a usar no están disponibles.

Los mapas tienen ciertas similitudes con los conjuntos:

1

Igual que ocurre con los conjuntos, los elementos no se colocan en el orden en que se han añadido.

2

Los objetos *TreeMap* se colocan ordenados según la clave.

3

Los mapas no admiten valores duplicados en la clave o llave. Si añadimos un elemento con la misma clave que otro existente, sobrescribirá el elemento existente.

HashMap

Un objeto *HashMap* es una colección donde cada elemento tiene asociada una clave que sirve para distinguirlo de los demás elementos de la colección.

Vamos a comenzar por crear una colección de nombres, es decir, de objetos *String*.

```
import java.util.Map;
import java.util.HashMap;
import java.util.Set;

public class Principal {
    public static void main(String args[]) {
        Map<String, String> nombres = new HashMap<String, String>()

        nombres.put("51666443R", "Carlos Maldonado Gómez");
        nombres.put("51666443R", "Luis Santos Gómez");
        nombres.put("52664443A", "Alicia Torres Durán");
        nombres.put("31234443H", "Soledad Delgado Perico");
        nombres.put("45666443R", "Miguel Rubio gonzález");
        nombres.put("82333333T", "Alicia Pimiento Pérez");
        nombres.put("51777788Z", "Angel Ruiz Califato");
        nombres.put("91549494P", "Fernando García Solera");

        // Acceder a un elemento por la clave
        System.out.println(nombres.get("31234443H"));

        // Recorrer el mapa
        Set<String> claves = nombres.keySet();

        for (String key : claves) {
            System.out.println(key + " - " + nombres.get(key));
        }

        System.out.println("Hay " + nombres.size() + " personas");
    }
}
```

Vamos a analizar el ejemplo paso a paso:

1

Construir el objeto *HashMap*

Primero hemos creado el objeto *HashMap* *nombres* utilizando el siguiente formato:

```
Map<tipoClave, tipoElementos> nombreObjeto = new HashMap<tipoClave, tipoEle
```

HashMap es una clase genérica, maneja internamente dos objetos cuyo tipo puede variar. *tipoClave* corresponde al tipo de dato de las claves y *tipoElementos* corresponde al tipo de dato de los elementos.

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

En nuestro ejemplo tenemos una colección de elementos de tipo *String* (*nombres*), cuya clave es un DNI, que es también un texto. Lo más habitual es que la clave sea de tipo *Integer* o de tipo *String*.

2

Añadir elementos

Luego hemos añadido varios elementos utilizando el método *put*, que tiene el siguiente formato:

```
objMap.put(Object key, Object element);
```

El método *put()* recibe la clave (como objeto) y el elemento al que se asocia dicha clave (como objeto) y retorna el elemento que acaba de añadirse.

```
nombres.put("51666443R", "Carlos Maldonado Gómez");
nombres.put("51666443R", "Luis Santos Gómez");
nombres.put("52664443A", "Alicia Torres Durán");
nombres.put("31234443H", "Soledad Delgado Perico");
nombres.put("45666443R", "Miguel Rubio gonzález");
nombres.put("82333333T", "Alicia Pimiento Pérez");
nombres.put("51777788Z", "Angel Ruiz Califato");
nombres.put("91549494P", "Fernando García Solera");
```

Se han añadido dos elementos con la misma clave ("51666443R") a propósito para que puedas comprobar lo que ocurre en este caso. El método *put()* añade un elemento si la clave especificada no existe, pero si existe lo que hace es una modificación, es decir, sobrescribe el elemento asociado con dicha clave. Puedes comprobar que "Carlos Maldonado Gómez" no aparece en el listado, ha sido sustituido por "Luis Santos Gómez".

3

Buscar un elemento aleatoriamente

El método *get()* permite el acceso aleatorio a través de la clave. Si la clave buscada no existe devuelve *null*.

```
// Acceder a un elemento por la clave
System.out.println(nombres.get("31234443H"));
```

4

Iterando

Las colecciones de tipo *Map* no descenden de *Iterable*, luego es fácil deducir que no son colecciones iterables. No podemos hacer ninguna de estas operaciones con nuestro objeto nombres:

```
Iterator<String> nombres = nombres.iterator();
for(String n : nombres) {
}
```

Sin embargo, hay una solución. Las clases que implementan la interfaz *Map* tienen un par de métodos que vienen a salvarnos de esta situación:

```
objMap.keySet(); // Devuelve un objeto de tipo Set con las claves.
```

```
objMap.values(); // Devuelve un objeto de tipo Set con los valores, sin las claves.
```

Los objetos devueltos por estos métodos sí son iterables, ya que son conjuntos.

```
// Recorrer el mapa
Set<String> claves = nombres.keySet();

for (String key : claves) {
    System.out.println(key + " - " + nombres.get(key));
}
```

5

Obtener el número de elementos de la colección

```
System.out.println("Hay " + nombres.size() + " personas");
```

El método *size()* devuelve un valor *int* con el número de elementos que contiene el mapa.

TreeMap

Un objeto *TreeMap* funciona igual que un objeto *HashMap*, pero sus elementos estarán ordenados según la clave o llave.

Como prueba, vamos a utilizar el mismo ejemplo anterior, pero ahora con un objeto *TreeMap*.

```
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;

public class Principal {
    public static void main(String args[]) {
        Map<String, String> nombres = new TreeMap<String, String>()

        nombres.put("51666443R", "Carlos Maldonado Gómez");
        nombres.put("51666443R", "Luis Santos Gómez");
        nombres.put("52664443A", "Alicia Torres Durán");
        nombres.put("31234443H", "Soledad Delgado Perico");
        nombres.put("45666443R", "Miguel Rubio gonzález");
        nombres.put("82333333T", "Alicia Pimiento Pérez");
        nombres.put("51777788Z", "Angel Ruiz Califato");
        nombres.put("91549494P", "Fernando García Solera");

        // Acceder a un elemento por la clave
        System.out.println(nombres.get("31234443H"));

        // Recorrer el mapa
        Set<String> claves = nombres.keySet();

        for (String key : claves) {
            System.out.println(key + " - " + nombres.get(key));
        }

        System.out.println("Hay " + nombres.size() + " personas");
    }
}
```

En la ejecución puedes comprobar que los elementos aparecen ordenados por DNI.

Aprovecharemos la clase *Triangulo* creada anteriormente para crear un objeto *TreeMap*, que contendrá una colección de objetos *Triangulo*.

```
import java.util.Map;
import java.util.Set;
import java.util.TreeMap;

public class Principal {
    public static void main(String args[]) {
        Map<Integer, Triangulo> trians = new TreeMap<Integer, Trian

        trians.put(1, new Triangulo(1, 2, 2));
        trians.put(2, new Triangulo(2, 2, 3));
        trians.put(3, new Triangulo(1, 2, 3));
        trians.put(4, new Triangulo(4, 4, 4));

        Set<Integer> claves = trians.keySet();
        for (Integer key : claves) {
            System.out.println(trians.get(key).verTipo());
        }
    }
}
```

Ejemplo práctico: cuenta bancaria

Las colecciones de cualquier tipo son muy útiles en las interrelaciones de tipo agregación o composición donde la cardinalidad es de uno a varios. En el siguiente ejemplo práctico vamos a implementar una clase llamada *CuentaBancaria*, que estará compuesta por un número de cuenta, un nombre de cliente y una colección de movimientos.

La cardinalidad será de uno a varios (1/0..*). Una cuenta bancaria podrá tener de 0 a muchos movimientos. Utilizaremos una relación de tipo composición, donde la clase *CuentaBancaria* será el compuesto y la clase *Movimiento* un componente.

```
import java.time.LocalDate;

public class Movimiento {
    private LocalDate fecha;
    private String concepto;
    private double cantidad;
    private double saldo;

    public Movimiento(String concepto, double cantidad, double saldo) {
        this.concepto = concepto;
        this.cantidad = cantidad;
        this.saldo = saldo;
        this.fecha = LocalDate.now();
    }

    @Override
    public String toString() {
        return fecha + " Concepto=" + concepto + ", Cantidad=" + ca
    }
}
```

Clase que representa un movimiento en una cuenta bancaria, es decir, un gasto o un ingreso en una fecha determinada.

```
import java.util.ArrayList;

public class CuentaBancaria {
    private int numeroCuenta;
    private String cliente;
    private double saldo;

    private ArrayList<Movimiento> movimientos;

    public CuentaBancaria(int numeroCuenta, String cliente) {
        this.numeroCuenta = numeroCuenta;
        this.cliente = cliente;
        this.saldo = 0;
        this.movimientos = new ArrayList<Movimiento>();
    }

    public void agregarMovimiento(String concepto, double cantidad) {
        this.saldo = this.saldo + cantidad;
        this.movimientos.add(new Movimiento(concepto, cantidad, saldo));
    }

    @Override
    public String toString() {
        return "Número=" + numeroCuenta + ", Cliente=" + cliente +
    }

    public String listarMovimientos() {
        String listado = "";
        for (Movimiento mov : this.movimientos) {
            listado = listado + mov.toString()+"\n";
        }
        return listado;
    }
}
```

Clase que representa una cuenta bancaria compuesta por un número de cuenta, un cliente, un saldo y una colección de movimientos.

Ahora tendrás que crear una clase *Principal* que te permita poner en práctica el funcionamiento de un objeto *CuentaBancaria*.

```
public class Principal {  
    public static void main(String args[]) {  
        CuentaBancaria miCuenta = new CuentaBancaria(38143, "Amelia");  
        miCuenta.agregarMovimiento("Ingreso Nómina", 2000);  
        miCuenta.agregarMovimiento("Pago luz", -120);  
        miCuenta.agregarMovimiento("Compra supermercado", -27);  
        miCuenta.agregarMovimiento("Ingreso Efectivo", 8000);  
  
        System.out.println(miCuenta.toString());  
  
        System.out.println(miCuenta.listarMovimientos());  
    }  
}
```

Amplía conocimientos

En esta lección hemos visto buena parte de las clases Java que puedes utilizar para construir colecciones de objetos, pero aún hay más.

Ahora te invitamos, de manera opcional, a que investigues por tu cuenta y utilices algún tipo más de colección de las que no hemos visto en esta unidad. Por ejemplo, puedes probar con:

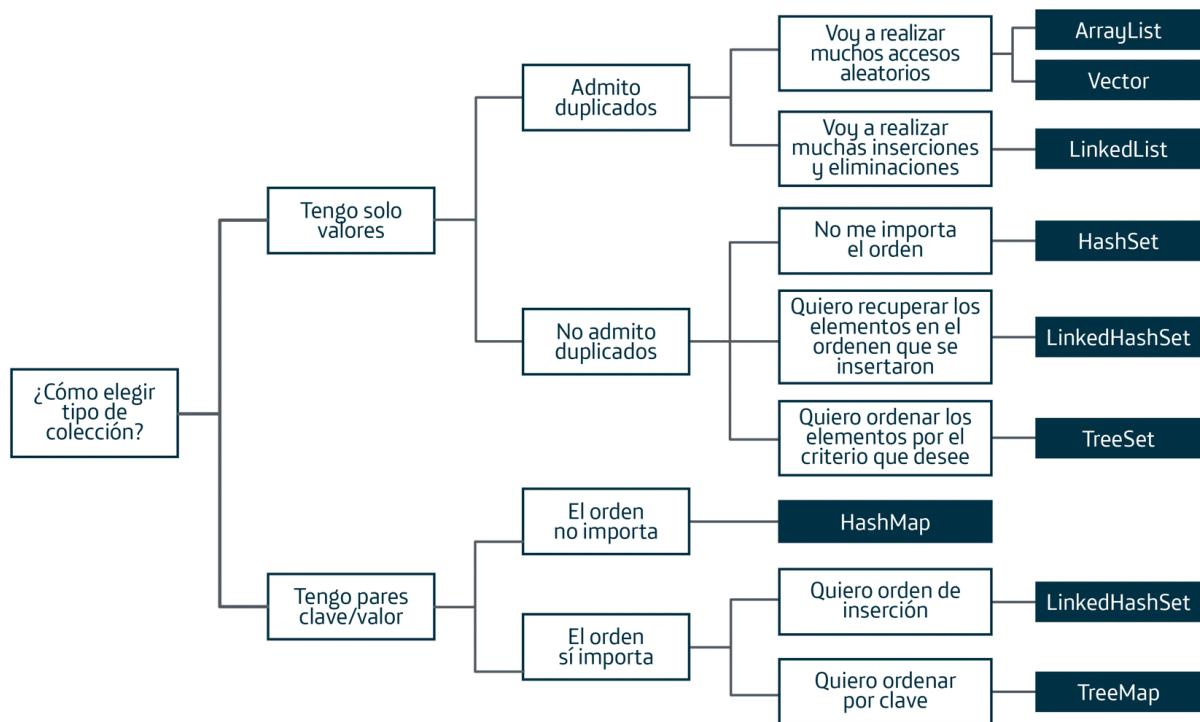
- **LinkedHashMap**: lista enlazada de elementos clave/valor (mapa). Puedes utilizarla igual que *HashMap*, pero al realizar una iteración los elementos se recuperan en el mismo orden en que se insertaron.
- **LinkedHashSet**: lista enlazada de elementos que no admite duplicados (conjunto). Puedes utilizarla igual que *HashSet*, pero al realizar una iteración los elementos se recuperan en el mismo orden en que se insertaron.
- **Stack**: lista de elementos basada en un *array* de tipo pila o estructura *LIFO* (Last In, First Out), donde el último en entrar es el primero en salir.

¿Te animas?

¿Cómo elegir el tipo de colección?

Con tantas clases disponibles para construir colecciones de datos,
¿cómo saber la que más
nos conviene?

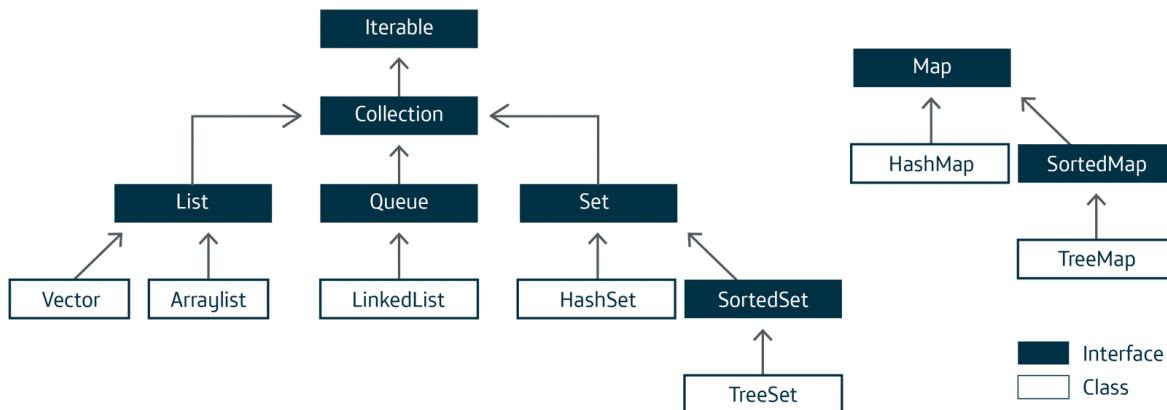
La siguiente imagen resume de manera esquemática cómo decidir el tipo de colección que necesitamos:



Resumen

Has terminado la lección, vamos a ver los puntos más importantes que hemos tratado.

- Java cuenta con una gran cantidad de clases que representan colecciones de datos. Todas ellas pertenecen al paquete `java.util`.
- Las colecciones se clasifican en listas, conjuntos y mapas.
- Todas ellas forman parte de una jerarquía. En la siguiente imagen puedes ver la jerarquía de las colecciones más utilizadas por la comunidad de programadores Java.





PROEDUCA