11 Colecciones

1. Introducción

Una colección representa un grupo de objetos. Esto objetos son conocidos como elementos. Cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos.

Las colecciones son estructuras de datos con la peculiaridad de que son estructuras dinámicas. Esto quiere decir que pueden aumentar o disminuir su tamaño dependiendo de los elementos que almacenan, lo que suponen una mejora respecto a las estructuras de datos estáticas cuyo tamaño se define en su creación y no se puede alterar en tiempo de ejecución, como por ejemplo, los arrays.

El API de Java nos proporciona en el paquete `java.util` el framework de las colecciones, que nos permite utilizar diferentes estructuras de datos para almacenar y recuperar objetos de cualquier clase. Java tiene desde la versión 2 todo un juego de clases e interfaces para guardar colecciones de objetos donde todas las entidades conceptuales están representadas por interfaces y las clases se usan para proveer implementaciones de esas interfaces. Estas clases e interfaces están estructuradas en una jerarquía.

Pero ¿qué podemos almacenar dentro de una colección? Podemos almacenar cualquier objeto que herede de la clase `Object`. Pero esto presenta ciertos inconvenientes:

- Podríamos tener una colección con objetos completamente distintos, lo que puede dar lugar a problemas ya que en todo momento deberíamos saber qué tipo de objeto y en

qué posición de la colección se encuentra el elemento con el que queremos trabajar, de otro modo podríamos tener incongruencias en el código o incluso hacer saltar una excepción.
- Otro inconveniente es que tendríamos que hacer continuos castings para poder trabajar con los elementos de la colección, lo cual resulta tedioso y poco productivo.

Veamos un ejemplo para obtener la suma de los valores almacenados en una lista:

```
int total = 0;
ArrayList numbers = new ArrayList();//Creación de la lista
numbers.add(1);//Se añade el elemento 1 a la lista utilizando el método add
numbers.add(2);
numbers.add(3);
for (int i = 0; i < numbers.size(); i++) {
  // Nos vemos a obligados a hacer cast, dado que numbers.get(i) retorna un
Object:
  total += (int) numbers.get(i);
}
System.out.printf("Total: %d\n", total);
```

Como vemos en el ejemplo anterior, nos vemos obligados a hacer explícitamente un cast cuando obtenemos un elemento de la lista, dado que la lista internamente trabaja con elementos de la clase `Object`. No hay ningún contrato que permita a la clase `ArrayList` saber con qué tipo de datos queremos que trabaje.

Además, es posible añadir elementos de distinto tipo a la misma lista, con el agravante de que más adelante cuando se intenta acceder al elemento y se hace un *cast* sobre él se producirá un error en tiempo de ejecución. Así, si modificamos el ejemplo anterior de la siguiente manera:

```
int total = 0;
ArrayList numbers = new ArrayList();
numbers.add(1);
numbers.add(2);
numbers.add("Antonio");
for (int i = 0; i < numbers.size(); i++) {
  // Esta línea lanza una excepción en tiempo de ejecución cuando
  // se trata de convertir a entero el elemento "Antonio".
  total += (int) numbers.get(i);
}
System.out.printf("Total: %d\n", total);
```

Así pues, para resolver este problema, a partir de la versión 5 de Java se empezaron a utilizar los **genéricos**. Los genéricos nos permiten establecer un tipo con el que vamos a trabajar en esa colección, de esa manera podemos evitar los problemas mencionados anteriormente.

2. Genéricos

Desde su versión original 1.0, muchas nuevas características han sido añadidas a Java. Todas han mejorado y ampliado el alcance del lenguaje, pero una que ha tenido un impacto especialmente profundo y de gran alcance es el uso de genéricos porque sus efectos se sintieron en todo el lenguaje Java ya que añadieron un elemento de sintaxis completamente nuevo y causaron cambios en muchas de las clases y métodos de la API principal. Los genéricos de Java están basados en los famosos *templates* de C++.

En su esencia, el término genérico significa tipo parametrizado, es decir, el tipo de datos sobre el que se opera se especifica como parámetro. Muchos algoritmos son lógicamente los mismos, independientemente del tipo de datos a los que se apliquen. Por ejemplo, un algoritmo de ordenación es el mismo si está ordenando elementos de tipo entero, decimales o cadena. Con

los genéricos, se puede definir un algoritmo una vez, independientemente del tipo de datos, y luego aplicar ese algoritmo a una amplia variedad de tipos de datos sin ningún esfuerzo adicional.

Los tipo genéricos nos permiten definir una clase con uno o más tipos genéricos y al instanciar un objeto de dicha clase indicar el tipo concreto con el que queremos trabajar. Por ejemplo, la clase `ArrayList` es una colección de Java para listas de elementos. Si observamos en la API esta clase, se define como `ArrayList<E>`. Esto significa que es una clase genérica donde E es el tipo genérico. Cuando se instancie un objeto de esta clase, hay que indicar el tipo concreto que se va a utilizar:

```
ArrayList<String> list = new ArrayList<String>();
```

Cualquier método que se ejecute sobre la variable *list* se hará traduciendo la letra genérica *E* por *String* en el código del método.

Para mantener la compatibilidad, a partir de Java 5 podemos trabajar con colecciones parametrizadas con tipos genéricos y con colecciones sin tipo (ya existentes en versiones anteriores a Java 5), pero se recomienda **siempre utilizar las colecciones parametrizadas**, por ejemplo siempre utilizar `ArrayList<E>` y nunca `ArrayList`.

Así el ejemplo anterior podríamos modificarlo de la siguiente manera:

```
int total = 0;
ArrayList<Integer> numbers = new ArrayList<Integer>();
numbers.add(1);
numbers.add(2);
// Esta línea da un error de compilación, dado que el compilador
// detecta que estamos intentando añadir una cadena a una lista de enteros:
numbers.add("Antonio");
for (int i = 0; i < numbers.size(); i++) {
  // Ya no es necesario hacer un cast explícito, dado que el compilador
  // lo hará internamente por nosotros, al haberle informado de que queríamos
  // trabajar con una lista de enteros.
  total += numbers.get(i);
}
System.out.printf("Total: %d\n", total);
```

Como vemos en el ejemplo anterior, gracias a la información que le suministramos al compilador sobre el tipo de lista con el que queremos trabajar, en este caso `Integer`, el compilador es capaz de detectar en tiempo de compilación que no debería ser posible añadir una cadena a la lista, y además nos evita tener que hacer explícitamente el *cast* a entero cuando obtenemos los elementos de la lista, porque ya lo puede hacer él internamente por nosotros.

Lo que se use en un genérico debe ser un objeto, por lo tanto, los genéricos no funcionan con datos primitivos. Para resolver esta situación, la API de Java incorpora las clases envoltorio (wrapper class), que no son más que dotar a los datos primitivos con un envoltorio que permita tratarlos como objetos. Las clases envoltorio proporcionan métodos de utilidad para la manipulación de datos primitivos (conversiones de / hacia datos primitivos, conversiones a String, etc).

3. Interfaz Collection

Es la interfaz raíz de la jerarquía de las colecciones. Java no proporciona ninguna implementación directa de esta interfaz, sino que proporciona implementaciones de subinterfaces más específicas como *Set* y *List*. Esta interfaz se utiliza normalmente para pasar colecciones y manipularlas cuando se desea la máxima generalidad.

Veamos algunos de los métodos que podemos observar en la API:

- `boolean add(E e)`: añade el elemento *e* a la colección.
- `boolean addAll(Collection<? extends E> c)`: añade todos los elementos de la colección *c*.
- `void clear()`: elimina todos los elementos de la colección.
- `boolean contains(Object o)`: comprueba si el elemento *o* está en la colección.
- `boolean containsAll(Collection<?> c)`: comprueba si todos los elementos de *c* están en la colección.
- `boolean isEmpty()`: comprueba si la colección está vacía.
- `boolean remove(Object o)`: elimina de la colección el elemento *o*.
- `boolean removeAll(Collection<?> c)`: elimina de la colección todos los elementos de *c*.
- `boolean retainAll(Collection<?> c)`: elimina de la colección todos los elementos exceptos los que están en *c*, es decir, obtiene la intersección.
- `int size()`: devuelve el tamaño de la colección.

4. Interfaz List

Esta interfaz es una subinterfaz o interfaz hija de *Collection*, por lo tanto, tiene todos los métodos de *Collection* y además añade los suyos propios.

Este tipo de colección se refiere a listas en las que los elementos de la colección tienen un orden, es decir, existe una secuencia de elementos. Cada elemento tiene un índice o posición. El primer elemento ocupa la posición 0.

La interfaz List sí admite elementos duplicados.

Veamos algunos de los métodos que podemos observar en la API:

- `void add(int index, E element)`: inserta el elemento *E* en la posición *index*.
- `boolean add(E e)`: añade el elemento *e* al final de la lista.
- `boolean addAll(int index, Collection<? extends E> c)`: inserta todos los elementos de *c* en la posición *index*.
- `boolean addAll(Collection<? extends E> c)`: añade todos los elementos de *c* al final de la lista.
- `E get(int index)`: devuelve el elemento de la posición *index*.
- `int indexOf(Object o)`: devuelve el índice de la primera ocurrencia del elemento *o* en la lista, o -1 si la lista no contiene el elemento.
- `int lastIndexOf(Object o)`: devuelve el índice de la última ocurrencia del elemento *o* en la lista, o -1 si la lista no contiene el elemento.
- `E remove(int index)`: elimina el elemento que se encuentra en la posición *index*.
- `E set(int index, E element)`: reemplaza el elemento que se encuentra en *index* por el elemento *element*.
- `List<E> subList(int fromIndex, int toIndex)`: devuelve la sublista comprendida entre las posiciones *fromIndex* incluida y *toIndex* excluida.

Existen varios tipos de implementaciones realizadas dentro de la plataforma Java para la interfaz *List*, como por ejemplo, *ArrayList*.

5. Clase ArrayList

Se basa en un array redimensionable que aumenta su tamaño según crece la colección de elementos. Lo bueno es que el tiempo de acceso a un elemento en particular es ínfimo. Lo malo es que si queremos eliminar un elemento del principio o del medio, la clase debe mover todos los que le siguen a la posición anterior, para tapar el agujero que deja el elemento removido. Esto hace que sacar elementos del medio o del principio sea costoso.

ArrayList mantiene el orden de inserción, es decir, si recorremos la colección se nos mostrará en el mismo órden en que insertamos los objetos.

Veamos un ejemplo de declaración e inicialización:

```
ArrayList<Integer> list = new ArrayList<Integer>();
```

A partir de java7, no es necesario indicar el genérico en la inicialización:

```
ArrayList<Integer> list = new ArrayList<>();
```

Para hacer el código más genérico, se puede definir la variable de tipo interfaz, ya que dicho código podría funcionar con cualquier clase que implemente la interfaz, simplemente habría que cambiar el `new`:

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

Veamos un ejemplo de *ArrayList* donde utiliza métodos tanto de *Collection* como de *List*:

```
package tema11_Colecciones;

import java.util.ArrayList;
import java.util.List;

public class ShowArrayList {

    public void show() {

        List<Integer> list1 = new ArrayList<>();
        List<Integer> list2 = new ArrayList<>();
        List<Integer> list3 = new ArrayList<>();

        list1.add(1);
        list1.add(2);//Se añaden los elementos al final de la lista
        list1.add(6);
        list1.add(2, 5);//Se añade el 5 en la posición 2
        for (Integer i : list1) {//Recorremos la lista con un bucle for-each:
1  2  5  6
            System.out.printf(" %d ", i);
        }
        System.out.println();
        list2.add(3);
        list2.add(4);
        list1.addAll(2, list2);//Se inserta list2 en la posición 2 de list1
        for (Integer i : list1) {// 1  2  3  4  5  6
            System.out.printf(" %d ", i);
        }
        System.out.println();
        list3.add(7);
        list3.add(8);
        list1.addAll(list3);//Se inserta list3 al final de list1
        for (Integer i : list1) {// 1  2  3  4  5  6  7  8
            System.out.printf(" %d ", i);
```

```
        }
        System.out.printf("\nEl elemento 3 de list1 es: %d", list1.get(3));//4
        System.out.printf("\nLa posición del 4 en list1 es: %d",
list1.indexOf(4));//3
        list1.add(4);//Se añade un 4 al final de list1
        System.out.printf("\nLa posición del 4 en list1 por el final es:
%d\n", list1.lastIndexOf(4));//8
        list1.remove(8);//Se elimina el elemento de la posición 8, que es el
último 4 insertado
        for (Integer i : list1) {// 1  2  3  4  5  6  7  8
            System.out.printf(" %d ", i);
        }
        list1.set(6, 8);//Se reemplaza el elemento que se encuentra en la
posición 6 por un 8
        System.out.println();
        for (Integer i : list1) {// 1  2  3  4  5  6  8  8
            System.out.printf(" %d ", i);
        }
        System.out.printf("\nLa sublista comprendida entre las posiciones 2 y
5 es: ");
        for (Integer i : list1.subList(2, 6)) {// 3  4  5  6
            System.out.printf(" %d ", i);
        }
        System.out.printf("\nEl 4 %s se encuentra en list1", list1.contains(4)
? "sí" : "no");//sí
        System.out.printf("\nEl 9 %s se encuentra en list1", list1.contains(9)
? "sí" : "no");//no

        //list1: 1  2  3  4  5  6  8  8
        //list2: 3  4
        //list3: 7  8
        System.out.printf("\nTodos los elementos de list2 %s se encuentran en
list1",
                list1.containsAll(list2) ? "sí" : "no");//sí
        System.out.printf("\nTodos los elementos de list3 %s se encuentran en
list1\n",
                list1.containsAll(list3) ? "sí" : "no");//no
        list1.removeAll(list3);//Se eliminan de list1 todos los elementos de
list3, es decir, el 8
        for (Integer i : list1) {// 1  2  3  4  5  6
            System.out.printf(" %d ", i);
        }
        System.out.println();
        list1.retainAll(list2);//Intersección entre list1 y list2
        for (Integer i : list1) {// 3  4
            System.out.printf(" %d ", i);
        }
        System.out.printf("\nEl tamaño de list1 es: %d", list1.size());//2
        System.out.printf("\nlist1 %s está vacía", list1.isEmpty() ? "sí" :
"no");//no
        list1.clear();//Elimina todos los elementos de list1
        System.out.printf("\nlist1 %s está vacía", list1.isEmpty() ? "sí" :
"no");//sí

    }

    public static void main(String[] args) {

        new ShowArrayList().show();

    }

}
```

Salida por consola:

```
 1  2  5  6
 1  2  3  4  5  6
 1  2  3  4  5  6  7  8
El elemento 3 de list1 es: 4
La posición del 4 en list1 es: 3
La posición del 4 en list1 por el final es: 8
 1  2  3  4  5  6  7  8
 1  2  3  4  5  6  8  8
La sublista comprendida entre las posiciones 2 y 5 es:  3  4  5  6
El 4 sí se encuentra en list1
El 9 no se encuentra en list1
Todos los elementos de list2 sí se encuentran en list1
Todos los elementos de list3 no se encuentran en list1
 1  2  3  4  5  6
 3  4
El tamaño de list1 es: 2
list1 no está vacía
list1 sí está vacía
```

6. Iteradores

En diseño de software, el patrón de diseño *Iterador* (en inglés, *Iterator*) define una interfaz que declara los métodos necesarios para acceder secuencialmente a un grupo de objetos de una colección.

Este patrón debe ser utilizado cuando se requiera una forma estándar de recorrer una colección, es decir, cuando no sea necesario que un cliente sepa el tipo de colección que está recorriendo.

La interfaz `Iterable<T>` contine el método `iterator()` que devuelve una instancia de alguna clase que implemente la interfaz `Iterator<T>`:

- `Iterator<T> iterator()`: devuelve un iterador al comienzo de la colección.

La interfaz `Iterator<E>` permite el acceso secuencial a los elementos de una colección y realizar recorridos sobre la colección. Los métodos de `Iterator<E>` son:

- `boolean hasNext()`: comprueba si hay siguiente elemento.
- `E next()`: devuelve el siguiente elemento y mueve el iterador.
- `void remove()`: se invoca después de next() para eliminar el último elemento leído.

La interfaz `Collection<E>` es una subinterfaz o interfaz hija de `Iterable<E>`. La clase `ArrayList<E>` implementa `Collection<E>` y por lo tanto también `Iterable<E>`, así que dispone del método `iterator()`. Veamos un ejemplo de recorrido de un `ArrayList<E>` utilizando iteradores:

```java
package tema11_Colecciones;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class ShowIterator {

    public void show() {

        Iterator<String> it;
        List<String> list = new ArrayList<>();
```

```
        list.add("Juan");
        list.add("Antonio");
        list.add("Jaime");
        list.add("Vicente");

        it = list.iterator();
        while (it.hasNext()) {
            System.out.println(it.next());
        }

    }

    public static void main(String[] args) {

        new ShowIterator().show();

    }

}
```

Salida por consola:

```
Juan
Antonio
Jaime
Vicente
```

Si no hay siguiente, `next()` lanza en ejecución una excepción `NoSuchElementException`:

```
package tema11_Colecciones;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class NextException {

    public void show() {

        Iterator<String> it;
        List<String> list = new ArrayList<>();

        list.add("Juan");
        it = list.iterator();
        System.out.println(it.next());
        System.out.println(it.next());//Se lanza en ejecución una excepción
NoSuchElementException

    }

    public static void main(String[] args) {

        new NextException().show();

    }

}
```

Tal y como vemos en el ejemplo anterior, hay que comprobar si hay siguiente con un `hasNext()` para que el `next()` no lanze la excepción.

No se puede modificar la colección dentro del bucle for-each porque se lanza en ejecución la excepción `ConcurrentModificationException`, ya que estamos recorriendo y modificando la lista a la vez:

```
package tema11_Colecciones;

import java.util.ArrayList;
import java.util.List;

public class ModifyInsideForEach {

    public void show() {

        List<String> list = new ArrayList<>();

        list.add("Juan");
        list.add("Antonio");
        list.add("Jaime");
        list.add("Vicente");
        for (String s : list) {//Se lanza en ejecución la excepción
ConcurrentModificationException
            System.out.printf(" %s ", s);
            if (s.equals("Antonio")) {
                list.remove("Antonio");
            }
        }

    }

    public static void main(String[] args) {

        new ModifyInsideForEach().show();

    }

}
```

Para solucionarlo, podemos utilizar el método `remove()` de `Iterator<E>`. Si se modifica una colección mientras se recorre, los iteradores quedan invalidados, a excepción del método `remove()` de la interfaz `Iterator<E>`. El método `remove()` permite eliminar elementos de la colección siendo la única forma adecuada para eliminar elementos durante la iteración:

```
package tema11_Colecciones;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class RemoveIterator {

    public void show() {

        String s;
        Iterator<String> it;
        List<String> list = new ArrayList<>();

        list.add("Juan");
        list.add("Antonio");
        list.add("Jaime");
        list.add("Vicente");

        it = list.iterator();
```

```
        while (it.hasNext()) {
            s = it.next();
            if (s.equals("Antonio")) {
                it.remove();
            } else {
                System.out.println(s);
            }
        }

    }

    public static void main(String[] args) {

        new RemoveIterator().show();

    }

}
```

Salida por consola:

```
Juan
Jaime
Vicente
```

Solo puede haber una invocación a `remove()` por cada invocación a `next()`. Si no se cumple, se lanza en ejecución una excepción `IllegalStateException`. Por ejemplo, imaginemos que tenemos una lista de personas y queremos eliminar a *Antonio* y a la persona que venga detrás. Si cuando encontramos a *Antonio*, hacemos dos `remove()` seguidos, entonces salta la excepción:

```
package tema11_Colecciones;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class RemoveNext1 {

    public void show() {

        String s;
        Iterator<String> it;
        List<String> list = new ArrayList<>();

        list.add("Juan");
        list.add("Antonio");
        list.add("Jaime");
        list.add("Vicente");

        it = list.iterator();
        while (it.hasNext()) {
            s = it.next();
            if (s.equals("Antonio")) {
                it.remove();
                it.remove();//Se lanza en ejecución la excepción
IllegalStateException
            } else {
                System.out.println(s);
            }
        }

    }
```

```
    public static void main(String[] args) {

        new RemoveNext1().show();

    }

}
```

Tendríamos que hacer otro `next()` para el que venga detrás de *Antonio*:

```
package tema11_Colecciones;

import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class RemoveNext2 {

    public void show() {

        String s;
        Iterator<String> it;
        List<String> list = new ArrayList<>();

        list.add("Juan");
        list.add("Antonio");
        list.add("Jaime");
        list.add("Vicente");

        it = list.iterator();
        while (it.hasNext()) {
            s = it.next();
            if (s.equals("Antonio")) {
                it.remove();
                it.next();
                it.remove();
            } else {
                System.out.println(s);
            }
        }

    }

    public static void main(String[] args) {

        new RemoveNext2().show();

    }

}
```

Salida por consola:

```
Juan
Vicente
```

La interfaz `ListIterator<E>` es una subinterfaz o interfaz hija de `Iterator<E>`. Es un iterador para listas que permite al programador recorrer la lista hacia adelante y hacia atrás, modificar la lista durante la iteración y obtener la posición actual del iterador en la lista.

Hereda los métodos de `Iterator<E>` y además aporta otros métodos nuevos:

- `void add(E e)`: inserta el elemento en la lista antes del elemento que sería devuelto por *next()*, si lo hubiera, y después del elemento que sería devuelto por *previous()*, si lo hubiera. Una llamada posterior a next() no se vería afectada y una llamada posterior a *previous*() devolvería el nuevo elemento.
- `boolean hasPrevious()`: comprueba si hay un elemento anterior.
- `int nextIndex()`: devuelve el índice del elemento que sería devuelto por una llamada a *next()*. El índice del primer elemento es 0.
- `E previous()`: devuelve el elemento anterior de la lista y mueve la posición del cursor hacia atrás.
- `int previousIndex()`: devuelve el índice del elemento que sería devuelto por una llamada a *previous()*.
- `void set(E e)`: sustituye el último elemento devuelto por *next()* o *previous()* por el elemento *e*.

En la interfaz `List<E>`, hay dos métodos para crear este iterador:

- `ListIterator<E> listIterator()`: se coloca antes del primer elemento para que al hacer el primer *next()* se devuelva el primer elemento.
- `ListIterator<E> listIterator(int index)`: se coloca antes del elemento que se encuentra en la posición *index* para que al hacer un *next()* se devuelva dicho elemento. Para recorrer la lista al revés, hay que crearlo con el tamaño de la lista para que el primer *previous()* devuelva el último.

```
package tema11_Colecciones;

import java.util.ArrayList;
import java.util.List;
import java.util.ListIterator;

public class ShowListIterator {

    public void show() {

        ListIterator<Integer> it;
        List<Integer> list = new ArrayList<>();

        list.add(1);
        list.add(2);
        list.add(4);
        list.add(6);

        //Recorrido de la lista hacia atrás:
        it = list.listIterator(list.size());
        while (it.hasPrevious()) {
            System.out.printf(" %d ", it.previous());// 6  4  2  1
        }

        it = list.listIterator(1);//Se coloca antes del elemento que se
encuentra en la posición 1
        System.out.printf("\n %d ", it.next());//2
        System.out.printf("\n %d ", it.nextIndex());//2
        System.out.printf("\n %d ", it.previousIndex());//1
        it.add(3);
        System.out.printf("\n %d ", it.next());//4 Una llamada posterior a
next() no se ve afectada
        it.add(5);
        System.out.printf("\n %d ", it.previous());//5 Una llamada posterior a
previous() devuelve el nuevo elemento
        System.out.printf("\n %d \n", it.next());//5
        it.set(7);//sustituye el último elemento devuelto por next() por 7
```

```
        for (Integer i : list) {
            System.out.printf(" %d ", i);//1  2  3  4  7  6
        }

    }

    public static void main(String[] args) {

        new ShowListIterator().show();

    }

}
```

7. Métodos equals y hashCode

Vimos en el tema de herencia que la clase *Object* proporciona un cierto número de métodos de utilidad general que pueden utilizar todos los objetos ya que los heredan. Pero normalmente hay que sobrescribirlos para que funcionen adecuadamente adaptándolos a la clase correspondiente. Esto se hace con la idea de que todas las clases utilicen el mismo nombre y prototipo de método para hacer operaciones comunes. Como por ejemplo, `equals()` que comprueba si dos elementos son iguales y `hashCode()` que devuelve un número entero que identifica al objeto cuando se guarda en algunas estructura de datos.

Sin embargo, debemos tener en cuenta que al sobrescribir todos estos métodos debemos seguir cumpliendo con el comportamiento que se espera de ellos, ya que son usados internamente por muchas clases del propio lenguaje. De lo contratrio, las clases que dependen de ello, como `HashMap` y `HashSet` dejarían de funcionar correctamente.

Una primera opción es no sobrescribir el método `equals()`. En dicho caso una instancia de la clase sólo será igual a sí misma, es decir, si está situada en la misma posición de memoria (identidad). Esta es la implementación de `equals()` en la clase `Object`.

Debemos tener en cuenta que cuando un programador usa el método `equals()` sobre un objeto pasándole como argumento otro objeto lo que pretende es descubrir si ambos objetos son equivalentes lógicamente (representan el mismo "valor"), no si están almacenados en la misma posición de memoria (tienen la misma identidad).

No sobrescribir el método `equals()` en una determinada clase es la opción recomendada en los siguientes casos:

- Cuando cada instancia de la clase es intrínsecamente única, lo cual es cierto para clases como `Scanner` que representan entidades activas en lugar de valores.
- Cuando se considera que no hay necesidad de que la clase provea una prueba de "equivalencia lógica".
- Cuando una superclase de la clase ya lo ha sobrescrito y el comportamiento de la superclase es apropiado para las subclase.
- Cuando la clase es privada o friendly, y estamos completamente seguros de que su método `equals()` nunca será invocado, ni explícita ni implícitamente.
- Cuando la clase controla las instancias que pueden ser creadas para asegurarse que no pueden existir dos objetos que representen el mismo "valor", como por ejemplo con las clases `enum` o las clases que siguen el patrón de diseño *singleton*. Para estas clases la equivalencia lógica es igual que la igualdad de identidad, ya que no puede existir más de una instancia para un "valor". Dado que esta es la comprobación que realiza la implementación de `equals()` en la clase `Object`, no tiene sentido sobrescribirlo.

Si ** *nuestra* clase no se encuentra en ninguno de los casos anteriores es muy recomendable que sobrescribamos el *método* `equals()`. Un ejemplo muy característico es cuando la clase corresponda a una entidad que represente un valor (*value class*).

Como hemos comentado, cuando sobrescribamos el método `equals()` debemos seguir cumpliendo el comportamiento que el sistema espera de él, que incluye las siguientes propiedades:

- Reflexiva: Para todo objeto *x* distinto de *null* se debe cumplir que `x.equals(x)` sea *true*.
- Simétrica: Para todo par de objetos *x* e *y* distintos de *null* se debe cumplir que `x.equals(y)` sólo debe retornar *true* si `y.equals(x)` retorna *true*.
- Transitiva: Para todo trío de objetos *x*, *y* , *z* distintos de *null* se debe cumplir que si `x.equals(y)` retorna *true* y `y.equals(z)` retorna *true* entonces `x.equals(z)` debe retornar *true*.
- Consistente: Para todo par de objetos *x* e *y* distintos de *null* se debe cumplir que `x.equals(y)` siempre retorne el mismo valor si no hemos cambiado los atributos que se usan para comparar en alguno de los dos objetos.
- Para todo objeto *x* distinto de *null* se debe cumplir que `x.equals(null)` debe retornar *false*.

Así, para sobrescribir el método `equals()` cumpliendo con las propiedades anteriores se recomienda seguir los siguientes pasos:

1. Usar el operador `==` para comprobar si el argumento corresponde a otra referencia al mismo objeto, en cuyo caso retornar *true*.
2. Usar el operador `instanceof` para comprobar si el objeto recibido como argumento no es de la misma clase, en cuyo caso retornar *false*. También nos sirve para comprobar si dicho argumento es *null*, ya que en este caso *instanceof* retorna *false*.
3. Hacer *cast* del objeto recibido como argumento convirtiéndolo a la clase correspondiente. Dado que hemos hecho antes *instanceof*, el *cast* siempre tendrá éxito.
4. Para cada atributo significativo de la clase, comprobar que dicho atributo en el objeto argumento es equivalente al atributo en el objeto *this*. Si no tenemos éxito en alguno de ellos, retornar *false*. Si todas las comprobaciones se han hecho con éxito, retornar *true*. Para realizar las comprobación de cada atributo usar:
    - El operador `==` para valores primitivos que no sean *float* ni *double*.
    - Para valores *float* usar `Float.compare(value1, value2` y para valores *double* usar `Double.compare(value1, value2)`.
    - Para valores correspondientes a objetos llamar a `equals()` recursivamente. Si es válido que dichos objetos contengan *null*, entonces debemos usar `Objects.equals(object1, object2)` para que no se produzca la excepción `NullPointerException`.
    - Para los valores correspondientes a arrays, compara uno a uno los elementos significativos del array. Si todos los elementos son significativos, usa alguno de las versiones del método estático `Arrays.equals()`.

Veamos un ejemplo:

```
public final class PhoneNumber {

    private final short areaCode, prefix, lineNum;

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof PhoneNumber)) return false;
        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNum == lineNum && pn.prefix == prefix
```

```
                && pn.areaCode == areaCode;
    }

    // ...
}
```

Una aspecto muy importante es que no debemos cambiar el tipo del objeto recibido como argumento, que siempre debe ser `Object`, o no estaremos sobrescribiendo el método `equals()`, sino sobrecargándolo, lo que puede producir falsos positivos. El compilador no se quejará si no usamos la anotación `@Override` (por eso siempre se recomienda usarla). Por ejemplo, nunca hagas esto:

```
// NUNCA HAGAS ESTO. EL TIPO DE o DEBE SER Object
public boolean equals(MyClass o) {
    // ...
}
```

Escribir a mano los métodos `equals()` y `hashCode()` es bastante tedioso. Para facilitarnos esta tarea tenemos librerías como la librería AutoValue de Google que lo genera automáticamente para nosotros con tan sólo usar una determinada anotación. Otra opción es dejar que el IDE nos genere dichos métodos, aunque esto tiene el inconveniente de que no se generan automáticamente de nuevo conforme añadimos atributos a nuestro clase, por lo que debemos tener cuidado, algo que sí hace AutoValue. En todo caso, es mejor usar el IDE que hacerlo nosotros a mano, ya que el humano es más propenso a los errores. En el Eclipse se encuentra en Menú *Source → Generate hashCode() and equals()*.

Nota: Algunas veces, para comparar que una variable de tipo `String` es equivalente a una determinada constante de cadena se usa la construcción `"Hello".equals(message)`, ya que dicho construcción no puede lanzar `NullPointerException` si *message* es *null*, sino que tan sólo retornará *false*, mientras que `message.equals("Hello")` lanzaría `NullPointerException` en ese caso.

Un detalle muy importante que no debemos olvidar es que **si en una clase sobrescribimos el método `equals()` debemos obligatoriamente sobrescribir también el método `hashCode()`** o de lo contrario no se estará cumpliendo con el comportamiento esperado de este último, lo que impedirá que los objetos de dichas clase funcionen correctamente en colecciones como `HashMap` y `HashSet`.

El comportamiento que se espera de `hashCode()` es el siguiente:

- Debe ser consistente, es decir, que repetidas llamadas al método `hashCode()` deben retornar el mismo valor, siempre y cuando no se haya modificado ninguno de los atributos usados para las comparaciones.
- Si dos objetos son equivalentes, es decir, si `x.equals(y)` retorna *true*, entonces `x.hashCode()` e `y.hashCode()` deben retornar el mismo valor entero. Éste es el motivo por el que siempre que sobrescribamos `equals()` debemos sobrescribir `hashCode()`, ya que la implementación por defecto de `hashCode()` de la clase `Object` devuelve una representación numérica de la dirección de memoria en la que se encuentra ubicado el objeto.
- Si dos objetos no son equivalentes, es decir si `x.equals(y)` retorna *false*, no es estrictamente necesario, aunque sí recomendable, que `x.hashCode()` e `y.hashCode()` retornen valores diferentes, de manera que se mejore el rendimiento de las tablas *hash*. Idealmente el algoritmo de la función *hash* debe distribuir una colección de instancias de un tamaño considerable de forma uniforme entre todos los valores enteros.

La implementación característica al sobrescribir el método `hashCode()` en la clase `PhoneNumber` es la siguiente, usando los atributos *areaCode*, *prefix* y *lineNum*:

```
public final class PhoneNumber {

    private final short areaCode, prefix, lineNum;

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof PhoneNumber)) return false;
        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }

    @Override
    public int hashCode() {
        int result = Short.hashCode(areaCode);
        result = 31 * result + Short.hashCode(prefix);
        result = 31 * result + Short.hashCode(lineNum);
        return result;
    }

    // ...
}
```

Además de escribir nosotros a mano el código del método `hashCode()`, podemos usar la implementación proporcionada por algunas librerías, como Guava o AutoValue. Adicionalmente los IDEs nos proporcionan asistentes para la generación de este método, seleccionando los atributos que queremos usar para ello de entre los disponibles en la clase. En el Eclipse se encuentra en Menú *Source → Generate hashCode() and equals()*.

Por otra parte, podemos usar `Objects.hashCode(objeto...)` para sobrescribir el método con una sola línea. Desafortunadamente, este método es bastante menos eficiente de lo esperado, debido a que recibe un array de atributos y de que realiza *boxing* y *unboxing* de los atributos que sean de un tipo primitivo. Por ejemplo:

```
public final class PhoneNumber {

    private final short areaCode, prefix, lineNum;

    @Override
    public boolean equals(Object o) {
        if (o == this) return true;
        if (!(o instanceof PhoneNumber)) return false;
        PhoneNumber pn = (PhoneNumber) o;
        return pn.lineNum == lineNum && pn.prefix == prefix
                && pn.areaCode == areaCode;
    }

    @Override
    public int hashCode() {
        return Objects.hash(lineNum, prefix, areaCode);
    }

    // ...
}
```

Si una clase es inmutable y el coste de calcular el valor *hash* es significativo, podría considerar almacenar cacheado el código *hash* en el propio objeto, en lugar de recalcularlo cada vez que se solicite. Si cree que la mayoría de los objetos de esta clase se usarán como claves *hash*, entonces debería calcular el código *hash* cuando se cree la instancia. De lo contrario, podría elegir calcular perezosamente el código *hash* la primera vez que se invoque el método `hash()`.

Dos consideraciones finales: en primer lugar **no excluya atributos significativos del cálculo de valor *hash***, así logrará un mejor rendimiento, al no repetir tanto los valores. En segundo lugar, **no proporcione a los clientes de la clase demasiada información acerca de cómo se calcula el valor *hash***, de esta manera el código cliente no podrá depender de cómo se calcula, permitiéndonos modificar la implementación del método en el futuro sin afectar a los clientes.

8. Colecciones sin duplicados

La interfaz `Set`, que hereda de *Collection*, permite implementar listas de elementos sin duplicados, es decir, modela la abstracción matemática de los conjuntos.

Existen varios tipos de implementaciones realizadas dentro de la plataforma Java:

- **HashSet**: esta implementación almacena los elementos en una tabla *hash*. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones.
- **LinkedHashSet**: esta implementación almacena los elementos en función del orden de inserción. Es un poco más costosa que *HashSet*.
- **TreeSet**: esta implementación utiliza una estructura de árbol para ordenar los elementos. Es bastante más lenta que *HashSet*. La veremos más adelante en el apartado de los árboles.

8.1 Clase HashSet

Implementa la interfaz anterior. Es la clase más utilizada para implementar listas sin duplicados. Esta clase permite el elemento nulo. No garantiza ningún orden a la hora de realizar iteraciones.

Utiliza internamente una tabla de tipo *hash*:

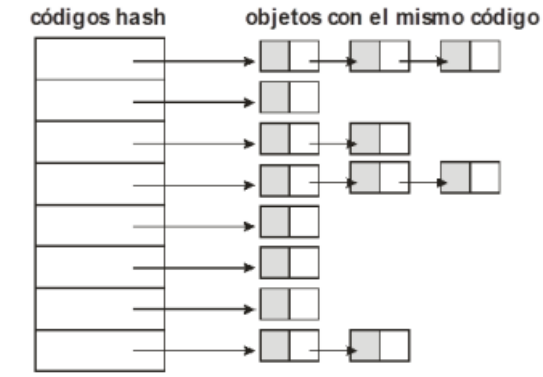
Hashtable

Al querer guardar un objeto en esta estructura, se llama al método `hashCode()` el cual devuelve un número entero que la estructura usará para decidir en qué cajón guardará este dato. Para recuperar el objeto se llama al método `hashCode()` para determinar de qué cajón debo recuperar el objeto. El objetivo de guardar los datos de esta forma y de llamar al método es lograr almacenar y recuperar información en tiempo constante (lo cual no ocurre siempre, pero se acerca). El que no suceda esto depende, casi siempre, del valor que devuelva el método `hashCode()` para cada objeto.

Supongamos que guardamos 3 objetos en esta estructura y el método `hashCode()` de los 3 devuelve 0, esto quiere decir que los 3 objetos se guardarán en el cajón 0. Cuando se necesite recuperar un objeto, hay que recorrer los objetos del cajón 0 para determinar cuál es el que se quiere recuperar. Por lo tanto, este método `hashCode()` no es útil ya que lo que se pretende al

guardar los elementos es que éstos queden dispersos de forma uniforme en toda la estructura quedando la menor cantidad de cajones vacíos y que no haya cajones donde se guarden muchos más elementos que en otros.

Si dos objetos tienen el mismo `hashCode()`, ambos objetos se guardarán en el mismo cajón. La estructura usa entonces el método `equals()` dentro de ese cajón para determinar cuál corresponde con el solicitado y para eso depende de que el programador haya sobreescrito el método, de lo contrario no garantiza un resultado correcto.



Los objetos *HashSet* se construyen con un tamaño inicial de tabla (el tamaño del array) y un factor de carga que indica cuándo se debe redimensionar el array. Es decir, si se creó un array de 100 elementos y la carga se estableció al 80%, cuando se hayan rellenado 80 valores, se redimensiona el array. Por defecto, el tamaño del array se toma con 16 y el factor de carga con 0,75 (75%). No obstante, se puede construir una lista *HashSet* indicando ambos parámetros.

Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función *hash* disperse de forma correcta los elementos dentro de la tabla *hash*. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación.

Veamos un ejemplo de `HashSet` con la clase `Vehicle`. Los atributos significativos a tener en cuenta para el `equals()` y el `hashCode()` son *wheelCount* y *colour*. La velocidad (*speed*) no se incluye ya que si comparamos el mismo coche pero con velocidades distintas, en realidad, no deja de ser el mismo coche.

```
package tema8_Colecciones.sinDuplicados;

public class Vehicle {

    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(int wheelCount, String colour) {
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }
```

```
    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    @Override
    public String toString() {
        return "Vehicle [wheelCount=" + wheelCount + ", speed=" + speed + ",
colour=" + colour + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((colour == null) ? 0 : colour.hashCode());
        result = prime * result + wheelCount;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj instanceof Vehicle))
            return false;
        Vehicle other = (Vehicle) obj;
        if (colour == null) {
            if (other.colour != null)
                return false;
        } else if (!colour.equals(other.colour))
            return false;
        if (wheelCount != other.wheelCount)
            return false;
        return true;
    }

}
```

```
package tema8_Colecciones.sinDuplicados;

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

public class ShowHashSet {

    public void show() {

        Set<Vehicle> set = new HashSet<>();
        set.add(new Vehicle(4, "azul"));
        set.add(new Vehicle(2, "rojo"));
```

```
        set.add(new Vehicle(4, "azul"));
        set.add(new Vehicle(2, "rojo"));
        set.add(new Vehicle(4, "verde"));
        for (Vehicle v : set) {
            System.out.println(v);//Se llama al toString del objeto
        }

    }

    public static void main(String[] args) {

        new ShowHashSet().show();

    }

}
```

Salida por consola:

```
Vehicle [wheelCount=4, speed=0.0, colour=verde]
Vehicle [wheelCount=4, speed=0.0, colour=azul]
Vehicle [wheelCount=2, speed=0.0, colour=rojo]
```

8.2 Clase LinkedHashSet

Almacena los elementos en función del orden de inserción. Es un poco más costosa que *HashSet*.

```
package tema8_Colecciones.sinDuplicados;

import java.util.LinkedHashSet;
import java.util.Set;

public class ShowLinkedHashSet {

    public void show() {

        Set<Vehicle> set = new LinkedHashSet<>();
        set.add(new Vehicle(4, "azul"));
        set.add(new Vehicle(2, "rojo"));
        set.add(new Vehicle(4, "azul"));
        set.add(new Vehicle(2, "rojo"));
        set.add(new Vehicle(4, "verde"));
        for (Vehicle v : set) {
            System.out.println(v);//Se llama al toString del objeto
        }

    }

    public static void main(String[] args) {

        new ShowLinkedHashSet().show();

    }

}
```

Salida por consola:

```
Vehicle [wheelCount=4, speed=0.0, colour=azul]
Vehicle [wheelCount=2, speed=0.0, colour=rojo]
Vehicle [wheelCount=4, speed=0.0, colour=verde]
```

8.3 Clase EnumSet

Es una implementación de conjuntos de alto rendimiento de tipos enumerados. Requiere que las constantes de enumeración colocadas en él pertenezcan al mismo tipo de enumeración. Veamos algunos de sus métodos:

- `static <E extends Enum<E>> EnumSet<E> allOf(Class<E> elementType)`: crea un conjunto de enumeraciones que contiene todos los valores del tipo de enumeración especificado.
- `static <E extends Enum<E>> EnumSet<E> complementOf(EnumSet<E> s)`: crea un conjunto de enumeraciones con el mismo tipo de elemento que el conjunto de enumeraciones especificado, conteniendo inicialmente todos los elementos de este tipo que no están contenidos en el conjunto especificado.
- `static <E extends Enum<E>> EnumSet<E> copyOf(Collection<E> c)`: crea un conjunto de enumeraciones a partir de una colección.
- `static <E extends Enum<E>> EnumSet<E> copyOf(EnumSet<E> s)`: crea un conjunto de enumeraciones a partir de otro.
- `static <E extends Enum<E>> EnumSet<E> noneOf(Class<E> elementType)`: crea un conjunto de enumeraciones vacío con el tipo de elemento especificado.
- `static <E extends Enum<E>> EnumSet<E> of(E e)`: crea un conjunto de enumeraciones que contiene el elemento especificado. Este método tiene varias sobrecargas para admitir más elementos.
- `static <E extends Enum<E>> EnumSet<E> range(E from, E to)`: crea un conjunto de enumeraciones que contiene inicialmente todos los elementos del rango definido por los dos elementos especificados.

```java
package tema11_Colecciones.enumSet;

public enum Operation {

    PLUS("+"), MINUS("-"), TIMES("*"), DIVIDE("/");

    private final String symbol;

    private Operation(String symbol) {
        this.symbol = symbol;
    }

    public String getSymbol() {
        return symbol;
    }

}
```

```java
package tema11_Colecciones.enumSet;

import java.util.EnumSet;

public class ShowEnumSet {

    public void show() {

        EnumSet<Operation> allOperations1, allOperations2, operations1,
operations2, operations3, operations4;

        allOperations1 = EnumSet.allOf(Operation.class);
        System.out.printf("allOf: %s", allOperations1);//allOf: [PLUS, MINUS,
TIMES, DIVIDE]

        allOperations2 = EnumSet.copyOf(allOperations1);
```

```
        System.out.printf("\ncopyOf: %s", allOperations2);//copyOf: [PLUS,
MINUS, TIMES, DIVIDE]

        operations1 = EnumSet.noneOf(Operation.class);
        operations1.add(Operation.PLUS);
        operations1.add(Operation.MINUS);
        System.out.printf("\nnoneOf y add: %s", operations1);//noneOf y add:
[PLUS, MINUS]

        operations2 = EnumSet.complementOf(operations1);
        System.out.printf("\ncomplementOf: %s", operations2);//complementOf:
[TIMES, DIVIDE]

        operations3 = EnumSet.of(Operation.DIVIDE, Operation.MINUS);
        System.out.printf("\nof: %s", operations3);//of: [MINUS, DIVIDE]

        operations4 = EnumSet.range(Operation.MINUS, Operation.DIVIDE);
        System.out.printf("\nrange: %s\n", operations4);//range: [MINUS,
TIMES, DIVIDE]
        System.out.println(operations4.contains(Operation.PLUS));//false
        System.out.println(operations4.contains(Operation.MINUS));//true

    }

    public static void main(String[] args) {

        new ShowEnumSet().show();

    }

}
```

9. Mapas

Las colecciones de tipo Set tienen el inconveniente de tener que almacenar una copia exacta del elemento a buscar. Sin embargo, en la práctica es habitual que haya datos que se consideran clave, es decir, que identifican a cada objeto (el dni de las personas por ejemplo) de tal manera que se buscan los datos en base a esa clave y por otro lado se almacenan el resto de los datos. Los mapas permiten definir colecciones de elementos que poseen pares de datos clave-valor. Esto se utiliza para localizar valores en función de la clave que poseen. Son muy interesantes y rápidos. Los mapas también son conocidos como diccionarios.

La interfaz `Map<K,V>` es la raíz de todas las clases que implementan mapas. Hasta la versión 5, los mapas eran colecciones de pares clave-valor donde tanto la clave como el valor eran de tipo Object. Desde la versión 5, esta interfaz tiene dos genéricos: `K` para el tipo de datos de la clave y `V` para el tipo de los valores. Esta estructura de datos nos permite obtener el objeto `V` muy rápidamente a partir de su clave `K`.

Esta interfaz no hereda de Collection por lo que no tiene los métodos vistos anteriormente. La razón es que la obtención, búsqueda y borrado de elementos se hace de manera muy distinta.

Las claves no se pueden repetir por lo que se implementan con una tabla hash para que no haya duplicados. Por lo tanto, la clase que se utilice como clave tiene que sobrescribir los métodos `equals()` y `hashCode()`.

Veamos algunos métodos de esta interfaz:

- `boolean containsKey(Object key)`: devuelve *true* si el mapa contiene dicha clave.
- `boolean containsValue(Object value)`: devuelve *true* si el mapa contiene dicho valor.

- `V get(Object key)`: devuelve el valor asociado a la clave o *null* si no existe esa clave en el mapa.
- `V getOrDefault(Object key, V defaultValue)`: devuelve el valor asociado a la clave o *defaultValue* si no existe esa clave en el mapa.
- `V put(K key, V value)`: añade un par clave-valor al mapa. Si ya había un valor para esa clave, se reemplaza. Devuelve el valor que tenía antes dicha clave o *null* si la clave no estaba en el mapa.
- `V putIfAbsent(K key, V value)`: si la clave especificada no está ya asociada a un valor o está asignada a *null*, se le asocia al valor dado y devuelve *null*, en caso contrario, devuelve el valor previamente asociado con la clave.
- `void putAll(Map<? extends K,? extends V> m)`: añade los pares clave-valor del mapa *m*.
- `V remove(Object key)`: elimina la clave y su valor asociado, el cual se devuelve. Si no existe dicha clave, devuelve *null*.

Existen varios tipos de implementaciones realizadas dentro de la plataforma Java:

- **HashMap**: este implementación almacena las claves en una tabla *hash*. Es la implementación con mejor rendimiento de todas pero no garantiza ningún orden a la hora de realizar iteraciones.
- **LinkedHashMap**: esta implementación almacena las claves en función del orden de inserción. Es un poco más costosa que *HashMap*.
- **TreeMap**: esta implementación utiliza una estructura de árbol para ordenar las claves. Es bastante más lenta que *HashMap*. La veremos más adelante en el apartado de los árboles.

9.1 Clase HashMap

Esta implementación proporciona tiempos constantes en las operaciones básicas siempre y cuando la función *hash* disperse de forma correcta las claves dentro de la tabla *hash*. Es importante definir el tamaño inicial de la tabla ya que este tamaño marcará el rendimiento de esta implementación. No garantiza ningún orden a la hora de recorrer el mapa.

Veamos un ejemplo de un mapa de vehículos donde la clave es la matrícula. Añadimos la matrícula como atributo por lo que hay que generar de nuevo los métodos `toString()`, `hashCode()` y `equals()`:

```
package tema8_Colecciones.mapas;

public class Vehicle {

    private String registration;//Atributo para almacenar la matrícula del
coche
    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(String registration, int wheelCount, String colour) {
        this.registration = registration;
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }
```

```
    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public String getRegistration() {//get de la matrícula
        return registration;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    /*Como se ha añadido el nuevo atributo para la matrícula, hay que generar
     * de nuevo los métodos toString(), hashCode() y equals():
     */

    @Override
    public String toString() {
        return "Vehicle [registration=" + registration + ", wheelCount=" +
wheelCount + ", speed=" + speed + ", colour="
                + colour + "]";
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + ((colour == null) ? 0 : colour.hashCode());
        result = prime * result + ((registration == null) ? 0 :
registration.hashCode());
        result = prime * result + wheelCount;
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (!(obj instanceof Vehicle))
            return false;
        Vehicle other = (Vehicle) obj;
        if (colour == null) {
            if (other.colour != null)
                return false;
        } else if (!colour.equals(other.colour))
            return false;
        if (registration == null) {
            if (other.registration != null)
                return false;
        } else if (!registration.equals(other.registration))
            return false;
        if (wheelCount != other.wheelCount)
            return false;
        return true;
```

```
    }

}
```

```
package tema8_Colecciones.mapas;

import java.util.HashMap;
import java.util.Map;

public class ShowHashMap {

    public void show() {

        Map<String, Vehicle> map = new HashMap<>();
        Map<String, Vehicle> map2 = new HashMap<>();
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        for (int i = 0; i < vehicles.length; i++) {
            map.put(vehicles[i].getRegistration(), vehicles[i]);
        }

        System.out.println(map.containsKey("1005SAW"));//false
        System.out.println(map.containsKey("1705UBG"));//true
        System.out.println(map.containsValue(new Vehicle("5930POI", 4,
"negro")));//false
        System.out.println(map.containsValue(new Vehicle("5930POI", 2,
"negro")));//true
        System.out.println(map.get("4554ASD"));//null
        System.out.println(map.get("1705UBG"));//Vehicle
[registration=1705UBG, wheelCount=4, speed=0.0, colour=blanco]
        System.out.println(map.getOrDefault("8080SAS", new Vehicle("4554ASD",
4, "negro")));//Vehicle [registration=4554ASD, wheelCount=4, speed=0.0,
colour=negro]
        System.out.println(map.getOrDefault("1705UBG", new Vehicle("4554ASD",
4, "negro")));//Vehicle [registration=1705UBG, wheelCount=4, speed=0.0,
colour=blanco]
        System.out.println(map.put("6320LPL", new Vehicle("6320LPL", 2,
"verde")));//null
        System.out.println(map.put("6320LPL", new Vehicle("6320LPL", 4,
"beis")));//Vehicle [registration=6320LPL, wheelCount=2, speed=0.0,
colour=verde]
        System.out.println(map.putIfAbsent("4687RTB", new Vehicle("4687RTB",
2, "blanco")));//null
        System.out.println(map.putIfAbsent("4687RTB", new Vehicle("4687RTB",
4, "naranja")));//Vehicle [registration=4687RTB, wheelCount=2, speed=0.0,
colour=blanco]
        System.out.println(map.remove("1234ABC"));//null
        System.out.println(map.remove("4687RTB"));//Vehicle
[registration=4687RTB, wheelCount=2, speed=0.0, colour=blanco]

        System.out.printf("El mapa tiene %d vehículos", map.size());
        map2.put("7410HJH", new Vehicle("7410HJH", 4, "rojo"));
        map2.put("8520FGF", new Vehicle("8520FGF", 2, "verde"));
        map.putAll(map2);//añade a map los pares clave-valor del mapa map2
        System.out.printf("\nDespués de añadirle map2, el mapa tiene %d
vehículos", map.size());

    }
```

```
    public static void main(String[] args) {

        new ShowHashMap().show();

    }

}
```

Salida por consola:

```
false
true
false
true
null
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=blanco]
Vehicle [registration=4554ASD, wheelCount=4, speed=0.0, colour=negro]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=blanco]
null
Vehicle [registration=6320LPL, wheelCount=2, speed=0.0, colour=verde]
null
Vehicle [registration=4687RTB, wheelCount=2, speed=0.0, colour=blanco]
null
Vehicle [registration=4687RTB, wheelCount=2, speed=0.0, colour=blanco]
El mapa tiene 7 vehículos
Después de añadirle map2, el mapa tiene 9 vehículos
```

Veamos las distintas maneras de recorrer un mapa:

- `Set<K> keySet()`: devuelve un conjunto con todas las claves. Como entre las claves no puede haber elementos duplicados , las claves forman un conjunto (*Set*).
- `Collection<V> values()`: devuelve una colección con todos los valores. Los valores sí pueden estar duplicados, por lo tanto, este método devuelve un *Collection*.
- `Set<Map.Entry<K,V>> entrySet()`: devuelve un conjunto de objetos *Map.Entry<K, V>*. Los pares de elementos (también llamados entradas) de los que está compuesto un *Map<K, V>* son de un tipo que viene implementado por la interfaz *Map.Entry<K, V>*. La interfaz *Map.Entry<K, V>* se define de forma interna a la interfaz *Map<K, V>* y representa un objeto de par clave-valor, es decir, mediante esta interfaz podemos trabajar con una entrada del mapa. Veamos algunos métodos de la interfaz *Map.Entry<K, V>*:
  - `K getKey()`: retorna la clave.
  - `V getValue()`: retorna el valor.
  - `V setValue(V value)`: reemplaza el valor por *value* y devuelve el valor anterior.

```
package tema8_Colecciones.mapas;

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

public class TraverseHashMap {

    public void show() {

        Map<String, Vehicle> map = new HashMap<>();
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
```

```
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        for (int i = 0; i < vehicles.length; i++) {
            map.put(vehicles[i].getRegistration(), vehicles[i]);
        }

        System.out.println("Claves del mapa:\n");
        for (String s : map.keySet()) {//keySet() devuelve un conjunto con
todas las claves
            System.out.println(s);
        }

        System.out.println("\nValores del mapa:\n");
        for (Vehicle v : map.values()) {//values() devuelve una colección con
todos los vehículos
            System.out.println(v);
        }

        System.out.println("\nPares clave-valor del mapa usando un
foreach:\n");
        for (Map.Entry<String, Vehicle> entry : map.entrySet()) {
            System.out.printf("Matrícula -> %s Vehículo -> %s\n",
entry.getKey(), entry.getValue());
        }

        System.out.println("\nPares clave-valor del mapa usando
iteradores:\n");
        Set<Map.Entry<String, Vehicle>> entrySet = map.entrySet();
        Iterator<Map.Entry<String, Vehicle>> it = entrySet.iterator();
        Map.Entry<String, Vehicle> entry;
        while (it.hasNext()) {
            entry = it.next();
            System.out.printf("Matrícula -> %s Vehículo -> %s\n",
entry.getKey(), entry.getValue());
        }

    }

    public static void main(String[] args) {

        new TraverseHashMap().show();

    }

}
```

Salida por consola:

```
Claves del mapa:

3495JZA
1705UBG
1235GTR
7314QWE
9685KMX
5930POI

Valores del mapa:

Vehicle [registration=3495JZA, wheelCount=2, speed=0.0, colour=naranja]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=blanco]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
```

```
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=5930POI, wheelCount=2, speed=0.0, colour=negro]

Pares clave-valor del mapa usando un foreach:

Matrícula -> 3495JZA Vehículo -> Vehicle [registration=3495JZA, wheelCount=2,
speed=0.0, colour=naranja]
Matrícula -> 1705UBG Vehículo -> Vehicle [registration=1705UBG, wheelCount=4,
speed=0.0, colour=blanco]
Matrícula -> 1235GTR Vehículo -> Vehicle [registration=1235GTR, wheelCount=2,
speed=0.0, colour=rojo]
Matrícula -> 7314QWE Vehículo -> Vehicle [registration=7314QWE, wheelCount=4,
speed=0.0, colour=verde]
Matrícula -> 9685KMX Vehículo -> Vehicle [registration=9685KMX, wheelCount=4,
speed=0.0, colour=azul]
Matrícula -> 5930POI Vehículo -> Vehicle [registration=5930POI, wheelCount=2,
speed=0.0, colour=negro]

Pares clave-valor del mapa usando iteradores:

Matrícula -> 3495JZA Vehículo -> Vehicle [registration=3495JZA, wheelCount=2,
speed=0.0, colour=naranja]
Matrícula -> 1705UBG Vehículo -> Vehicle [registration=1705UBG, wheelCount=4,
speed=0.0, colour=blanco]
Matrícula -> 1235GTR Vehículo -> Vehicle [registration=1235GTR, wheelCount=2,
speed=0.0, colour=rojo]
Matrícula -> 7314QWE Vehículo -> Vehicle [registration=7314QWE, wheelCount=4,
speed=0.0, colour=verde]
Matrícula -> 9685KMX Vehículo -> Vehicle [registration=9685KMX, wheelCount=4,
speed=0.0, colour=azul]
Matrícula -> 5930POI Vehículo -> Vehicle [registration=5930POI, wheelCount=2,
speed=0.0, colour=negro]
```

9.2 Clase LinkedHashMap

Almacena las claves en función del orden de inserción. Es un poco más costosa que *HashMap*.

```java
package tema8_Colecciones.mapas;

import java.util.LinkedHashMap;
import java.util.Map;

public class ShowLinkedHashMap {

    public void show() {

        Map<String, Vehicle> map = new LinkedHashMap<>();
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        for (int i = 0; i < vehicles.length; i++) {
            map.put(vehicles[i].getRegistration(), vehicles[i]);
        }

        for (Map.Entry<String, Vehicle> entry : map.entrySet()) {
            System.out.printf("Matrícula -> %s Vehículo -> %s\n",
entry.getKey(), entry.getValue());
        }

    }
```

```
    public static void main(String[] args) {

        new ShowLinkedHashMap().show();

    }

}
```

Salida por consola:

```
Matrícula -> 9685KMX Vehículo -> Vehicle [registration=9685KMX, wheelCount=4,
speed=0.0, colour=azul]
Matrícula -> 1235GTR Vehículo -> Vehicle [registration=1235GTR, wheelCount=2,
speed=0.0, colour=rojo]
Matrícula -> 7314QWE Vehículo -> Vehicle [registration=7314QWE, wheelCount=4,
speed=0.0, colour=verde]
Matrícula -> 5930POI Vehículo -> Vehicle [registration=5930POI, wheelCount=2,
speed=0.0, colour=negro]
Matrícula -> 1705UBG Vehículo -> Vehicle [registration=1705UBG, wheelCount=4,
speed=0.0, colour=blanco]
Matrícula -> 3495JZA Vehículo -> Vehicle [registration=3495JZA, wheelCount=2,
speed=0.0, colour=naranja]
```

Podemos observar que los datos se muestran en el mismo orden en el que se insertaron.

9.3 Ejemplo de uso de un mapa en un enum

Si hemos sobrescrito el método `toString()` o las instancias del *enum* tienen alguna forma adicional de referirnos a ellas, tiene bastante sentido que creemos un método estático parecido a `valueOf()`, pero que reciba dicha forma adicional de referirnos a las instancias.

```
package tema11_Colecciones.mapas;

import java.util.Map;

public enum Operation {

    PLUS("+") {
        @Override
        public double apply(double x, double y) {
            return x + y;
        }
    },
    MINUS("-") {
        @Override
        public double apply(double x, double y) {
            return x - y;
        }
    },
    TIMES("*") {
        @Override
        public double apply(double x, double y) {
            return x * y;
        }
    },
    DIVIDE("/") {
        @Override
        public double apply(double x, double y) {
            return x / y;
        }
    };
```

```
    private final String symbol;

    private static final Map<String, Operation> symbolToOperation =
Map.of(Operation.PLUS.getSymbol(), Operation.PLUS,
            Operation.MINUS.getSymbol(), Operation.MINUS,
            Operation.TIMES.getSymbol(), Operation.TIMES,
            Operation.DIVIDE.getSymbol(), Operation.DIVIDE);

    private Operation(String symbol) {
        this.symbol = symbol;
    }

    public String getSymbol() {
        return symbol;
    }

    public abstract double apply(double x, double y);

    public static Operation fromSymbol(String symbol) {
        return symbolToOperation.get(symbol);
    }

}
```

```
package tema11_Colecciones.mapas;

public class ExampleUseMapEnum {

    public void show() {

        Operation operation;
        operation = Operation.fromSymbol("+");//operation se asigna con la
instancia correspondiente al símbolo +
        System.out.printf("La variable operation es de tipo enum %s y su
símbolo es %s", operation, operation.getSymbol());

    }

    public static void main(String[] args) {

        new ExampleUseMapEnum().show();

    }

}
```

Salida por consola:

```
La variable operation es de tipo enum PLUS y su símbolo es +
```

Como vemos en el código anterior, creamos un mapa estático que relaciona cada símbolo con cada instancia, de manera que podamos obtener la instancia adecuada a partir del símbolo. Debemos tener en cuenta que no está permitido que los constructores de las instancias de un *enum* accedan a los atributos estáticos del *enum*, con la excepción de las constantes de las instancias, dado que los atributos estáticos aún no han sido inicializados cuando se están ejecutando los constructores de las instancias. Un caso especial de esta restricción es que en los constructores de las instancias tampoco se puede acceder a otras instancias del *enum*.

9.4 Clase EnumMap

Es una implementación de mapa muy eficiente donde las claves son elementos de una enumeración.

```
package tema11_Colecciones.enumMap;

public enum Operation {

    PLUS("+"), MINUS("-"), TIMES("*"), DIVIDE("/");

    private final String symbol;

    private Operation(String symbol) {
        this.symbol = symbol;
    }

    public String getSymbol() {
        return symbol;
    }

}
```

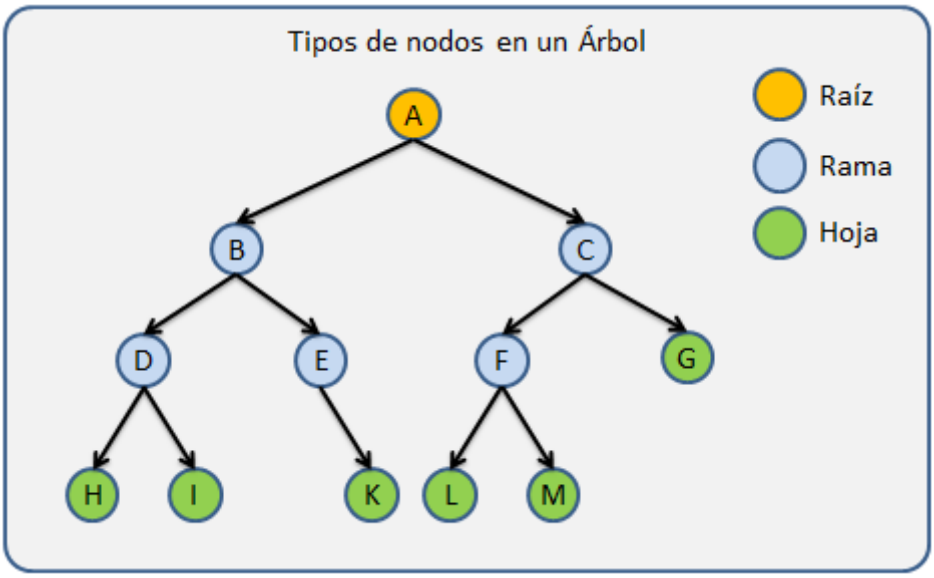
```
package tema11_Colecciones.enumMap;

import java.util.EnumMap;
import java.util.Map;

public class ShowEnumMap {

    public void show() {

        EnumMap<Operation, String> operationsMap = new EnumMap<>
(Operation.class);
        operationsMap.put(Operation.PLUS, "Esta operación se utiliza para
sumar");
        operationsMap.put(Operation.MINUS, "Esta operación se utiliza para
restar");
        operationsMap.put(Operation.TIMES, "Esta operación se utiliza para
multiplicar");
        operationsMap.put(Operation.DIVIDE, "Esta operación se utiliza para
dividir");
        for (Map.Entry<Operation, String> entry : operationsMap.entrySet()) {
            System.out.printf("%-6s: %s\n", entry.getKey(), entry.getValue());
        }

    }

    public static void main(String[] args) {

        new ShowEnumMap().show();

    }

}
```

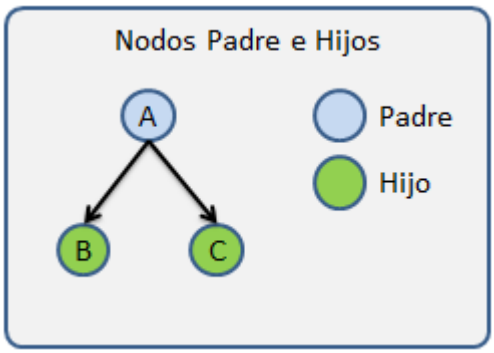
Salida por consola:

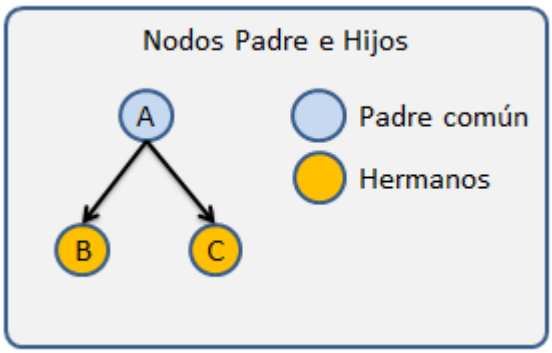
```
PLUS  : Esta operación se utiliza para sumar
MINUS : Esta operación se utiliza para restar
TIMES : Esta operación se utiliza para multiplicar
DIVIDE: Esta operación se utiliza para dividir
```

10. Árboles

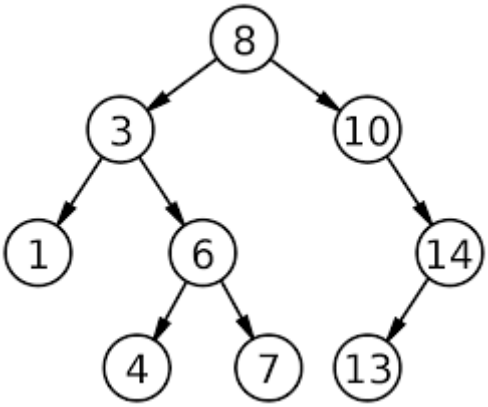
Los árboles se caracterizan por almacenar sus nodos en forma jerárquica y no en forma lineal como las listas.







Un árbol es una estructura en la que los datos se organizan en nodos. Los **árboles binarios** son aquellos en los que un nodo solamente puede tener dos hijos como máximo. Se utilizan para ordenar datos, de tal manera que a la izquierda se colocan los valores menores y a la derecha los valores mayores. Un árbol binario se puede recorrer de varias formas, siendo el recorrido **inorden** el que muestra los datos ordenados: subárbol izquierdo, raíz, subárbol derecho. Veamos un ejemplo:



Nos posicionamos en el 8. Mientras tenga subárbol iquierdo vamos avanzando hasta llegar al 1. Como es una hoja, la mostramos: el **1**. Luego, mostramos la raíz: el **3**. Después, continuamos con el subárbol derecho. Nos encontramos con el 6 pero tiene subárbol izquierdo, así que avanzamos hasta el 4. Como es una hoja, lo mostramos: el **4**. Luego, continuamos con la raíz: el

6. Después, continuamos con el subárbol derecho, avanzando hasta el 7. Como es hoja, lo mostramos: el **7**. Ya hemos tratado todo el subárbol izquierdo del 8, que es la raíz. Ahora mostramos la raíz: el **8**, y a continuación comenzamos con el subárbol derecho del 8. Y así sucesivamente...El resultado final es: 1, 3, 4, 6, 7, 8, 10, 13 y 14, es decir, los elementos ordenados de menor a mayor.

Si queremos introducir un nuevo nodo en el árbol, hay que tener cuidado de no romper la estructura ni el orden del árbol. Hay que tener en cuenta que cada nodo no puede tener más de dos hijos, por esta razón si un nodo ya tiene 2 hijos, el nuevo nodo nunca se podrá insertar como su hijo. Con esta restricción nos aseguramos mantener la estructura del árbol, pero aún nos falta mantener el orden. Para localizar el lugar adecuado del árbol donde insertar el nuevo nodo se realizan comparaciones entre los nodos del árbol y el elemento a insertar. El primer nodo que se compara es el nodo raíz, si el nuevo nodo es menor que el raíz, la búsqueda prosigue por el nodo izquierdo de éste. Si el nuevo nodo fuese mayor, la búsqueda seguiría por el hijo derecho. Y así sucesivamente hasta llegar a un nodo que no tenga hijo en la rama por la que la búsqueda debería seguir. En este caso, el nuevo nodo se inserta en ese hueco, como su nuevo hijo.

Por ejemplo, queremos insertar el elemento 9. Lo primero es comparar el nuevo elemento con el nodo raíz. Como 9 > 8, entonces la búsqueda prosigue por el lado derecho. Ahora el nuevo nodo se compara con el elemento 10. En este caso 9 < 10, por lo que hay que continuar la búsqueda por la rama izquierda. Como la rama izquierda de 10 no tiene ningún nodo, se inserta en ese lugar el nuevo nodo.

La interfaz `SortedSet<E>` es la encargada de definir esta estructura. Esta interfaz es hija de `Set<E>`, que a su vez es hija de `Collection<E>`, que a su vez es hija de `Iterable<E>`. Por lo tanto, tiene los métodos de todas y además añade sus propios métodos:

- `E first()`: devuelve el elemento más pequeño.
- `E last()`: devuelve el elemento más grande.
- `SortedSet<E> headSet(E toElement)`: devuelve un *SortedSet* que contendrá todos los elementos menores que *toElement*.
- `SortedSet<E> tailSet(E fromElement)`: devuelve un *SortedSet* que contendrá todos los elementos mayores que *fromElement*.
- `SortedSet<E> subSet(E fromElement, E toElement)`: devuelve un *SortedSet* que contendrá los elementos que van desde *fromElement* incluido hasta *toElement* excluido.

Pero ¿cómo ordenamos elementos como por ejemplo vehículos? Para ello, Java nos proporciona dos interfaces: `Comparable<T>` y `Comparator<T>`. La diferencia entre ambas es que *Comparable* se implementa desde la propia clase que se quiere ordenar y *Comparator* no.

10.1 Clase TreeSet

La clase `TreeSet<E>` es la que se utiliza prioritariamente para trabajar con árboles ordenados ya que implementa la interfaz `SortedSet<E>`.

Los objetos a incluir en un *TreeSet* deben implementar *Comparable* o bien crear el árbol con un constructor que reciba un *Comparator*.

La interfaz `Comparable` contiene un único método, el método `compareTo`, que recibe un objeto de la misma clase y que debe realizar una comparación entre ambos objetos, retornando un valor entero negativo, cero o positivo, dependiendo de si el objeto sobre el que se ejecuta es, respectivamente, menor, igual o mayor que el objeto recibido.

La definición de la interfaz es la siguiente:

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

Por ejemplo, si quisiéramos crear un árbol para ordenar vehículos, lo primero que tendríamos que hacer es que la clase *Vehicle* implemente la interfaz *Comparable* y que el método *compareTo* ordene por el atributo que deseemos. Por ejemplo, vamos a ordenar vehículos alfabéticamente por el color. Como el color es de tipo *String*, debemos utilizar el *compareTo* de la clase *String*:

```java
package tema11_Colecciones.arboles1;

public class Vehicle implements Comparable<Vehicle> {//La clase Vehicle
implementa Comparable

    private String registration;
    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(String registration, int wheelCount, String colour) {
        this.registration = registration;
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public String getRegistration() {
        return registration;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    @Override
    public String toString() {
        return "Vehicle [registration=" + registration + ", wheelCount=" +
wheelCount + ", speed=" + speed + ", colour="
                + colour + "]";
    }

    @Override
```

```
    public int compareTo(Vehicle o) {//Tenemos que definir en este método el
criterio de ordenación
        return colour.compareTo(o.colour);//Utilizamos el compareTo de la
clase String
    }

}
```

```
package tema11_Colecciones.arboles1;

import java.util.SortedSet;
import java.util.TreeSet;

public class TreeSet1 {

    public void show() {

        SortedSet<Vehicle> tree = new TreeSet<>();
        tree.add(new Vehicle("9685KMX", 4, "azul"));
        tree.add(new Vehicle("1235GTR", 2, "rojo"));
        tree.add(new Vehicle("7314QWE", 4, "verde"));
        tree.add(new Vehicle("5930POI", 2, "negro"));
        tree.add(new Vehicle("1705UBG", 4, "blanco"));
        tree.add(new Vehicle("3495JZA", 2, "naranja"));
        for (Vehicle v : tree) {
            System.out.println(v);
        }

    }

    public static void main(String[] args) {

        new TreeSet1().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=blanco]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0, colour=naranja]
Vehicle [registration=5930POI, wheelCount=2, speed=0.0, colour=negro]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
```

Pero ¿qué ocurriría si tuviéramos colores repetidos?:

```
package tema11_Colecciones.arboles1;

import java.util.SortedSet;
import java.util.TreeSet;

public class TreeSet2 {

    public void show() {

        SortedSet<Vehicle> tree = new TreeSet<>();
        tree.add(new Vehicle("9685KMX", 4, "azul"));
        tree.add(new Vehicle("1235GTR", 2, "rojo"));
        tree.add(new Vehicle("7314QWE", 4, "verde"));
```

```
        tree.add(new Vehicle("5930POI", 2, "azul"));
        tree.add(new Vehicle("1705UBG", 4, "rojo"));
        tree.add(new Vehicle("3495JZA", 2, "verde"));
        for (Vehicle v : tree) {
            System.out.println(v);
        }

    }

    public static void main(String[] args) {

        new TreeSet2().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
```

Los tres primeros vehículos se introducen en el árbol. Cuando se va a introducir el cuarto que es de color azul, el árbol lo va comparando con *compareTo* con los vehículos que ya existen en el árbol para encontrar la posición ordenada donde incluirlo. Pero cuando lo compara con el que es azul, el *compareTo* le devuelve 0, por lo que el árbol interpreta que ese objeto ya existe en el árbol, que es igual a otro, por lo tanto no lo incluye en el árbol. Lo mismo ocurre con el quinto y el sexto. Por lo tanto, lo que ocurriría es que los 3 últimos no se introducen en el árbol porque el *compareTo* devuelve 0 entre vehículos del mismo color, por lo que el árbol considera que son iguales. En estos casos, lo que se hace es que se incluye un segundo criterio de comparación: vamos a ordenar por el color, y en aquellos casos donde los vehículos tengan el mismo color, entonces vamos a ordenar por matrícula:

```
package tema11_Colecciones.arboles2;

public class Vehicle implements Comparable<Vehicle> {

    private String registration;
    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(String registration, int wheelCount, String colour) {
        this.registration = registration;
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }
```

```
    public void setColour(String colour) {
        this.colour = colour;
    }

    public String getRegistration() {
        return registration;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    @Override
    public String toString() {
        return "Vehicle [registration=" + registration + ", wheelCount=" +
wheelCount + ", speed=" + speed + ", colour="
                + colour + "]";
    }

    @Override
    public int compareTo(Vehicle o) {
        int result = colour.compareTo(o.colour);
        if (result == 0) {//Introducimos un segundo criterio de ordenación
            result = registration.compareTo(o.registration);
        }
        return result;
    }

}
```

```
package tema11_Colecciones.arboles2;

import java.util.SortedSet;
import java.util.TreeSet;

public class ShowTreeSet {

    public void show() {

        SortedSet<Vehicle> tree = new TreeSet<>();
        tree.add(new Vehicle("9685KMX", 4, "azul"));
        tree.add(new Vehicle("1235GTR", 2, "rojo"));
        tree.add(new Vehicle("7314QWE", 4, "verde"));
        tree.add(new Vehicle("5930POI", 2, "azul"));
        tree.add(new Vehicle("1705UBG", 4, "rojo"));
        tree.add(new Vehicle("3495JZA", 2, "verde"));
        for (Vehicle v : tree) {
            System.out.println(v);
        }

    }

    public static void main(String[] args) {

        new ShowTreeSet().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=5930POI, wheelCount=2, speed=0.0, colour=azul]
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=rojo]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0, colour=verde]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
```

Pero, ¿qué ocurriría si la clase *Vehicle* no implementase la interfaz *Comparable*? Pues que no contendría el método *compareTo*, entonces el árbol no tendría la información de cómo ordenar los vehículos. En este caso, se lanzaría una excepción `ClassCastException`:

```
package tema11_Colecciones.arboles3;

public class Vehicle {//No implementa Comparable por lo que no contiene el
método compareTo

    private String registration;
    private int wheelCount;
    private double speed;
    private String colour;

    public Vehicle(String registration, int wheelCount, String colour) {
        this.registration = registration;
        this.wheelCount = wheelCount;
        this.colour = colour;
        speed = 0;
    }

    public int getWheelCount() {
        return wheelCount;
    }

    public double getSpeed() {
        return speed;
    }

    public String getColour() {
        return colour;
    }

    public void setColour(String colour) {
        this.colour = colour;
    }

    public String getRegistration() {
        return registration;
    }

    public void accelerate(double amount) {
        speed += amount;
    }

    public void brake(double amount) {
        speed -= amount;
    }

    @Override
    public String toString() {
        return "Vehicle [registration=" + registration + ", wheelCount=" +
wheelCount + ", speed=" + speed + ", colour="
                + colour + "]";
```

```
    }

}
```

```
package tema11_Colecciones.arboles3;

import java.util.SortedSet;
import java.util.TreeSet;

public class ShowException {

    public void show() {

        SortedSet<Vehicle> tree = new TreeSet<>();
        tree.add(new Vehicle("9685KMX", 4, "azul"));
        tree.add(new Vehicle("1235GTR", 2, "rojo"));
        tree.add(new Vehicle("7314QWE", 4, "verde"));
        tree.add(new Vehicle("5930POI", 2, "negro"));
        tree.add(new Vehicle("1705UBG", 4, "blanco"));
        tree.add(new Vehicle("3495JZA", 2, "naranja"));
        for (Vehicle v : tree) {
            System.out.println(v);
        }

    }

    public static void main(String[] args) {

        new ShowException().show();

    }

}
```

Salida por consola:

```
Exception in thread "main" java.lang.ClassCastException: class
tema11_Colecciones.arboles3.Vehicle cannot be cast to class
java.lang.Comparable (tema11_Colecciones.arboles3.Vehicle is in unnamed module
of loader 'app'; java.lang.Comparable is in module java.base of loader
'bootstrap')
    at java.base/java.util.TreeMap.compare(TreeMap.java:1291)
    at java.base/java.util.TreeMap.put(TreeMap.java:536)
    at java.base/java.util.TreeSet.add(TreeSet.java:255)
    at tema11_Colecciones.arboles3.ShowException.show(ShowException.java:11)
    at tema11_Colecciones.arboles3.ShowException.main(ShowException.java:25)
```

Otra posibilidad es utilizar un objeto `Comparator<E>`. Esta es otra interfaz que define el método `compare` al que se le pasan los dos objetos a comparar y cuyo resultado es como el de *compareTo* (0 si son iguales, positivo si el primero es mayor y negativo si el segundo es mayor). Para definir un comparador de esta forma, hay que crear una clase que implemente esta interfaz y definir el método *compare*, después crear un objeto de ese tipo y usarlo en la construcción del árbol mediante un constructor que recibe un *Comparator*: `TreeSet(Comparator<? super E> comparator)`. En este caso, ésa será la forma prioritaria para ordenar la lista, por encima del método compareTo de la interfaz Comparable. Como ya dijimos anteriormente, la diferencia entre *Comparable* y *Comparator* es que *Comparable* se implementa desde la propia clase que se quiere ordenar y *Comparator* no, ya que *Comparator* se implementa desde otra clase distinta a la que se quiere ordenar:

```
package tema11_Colecciones.arboles3;
```

```java
import java.util.Comparator;

public class VehicleComparator implements Comparator<Vehicle> {

    @Override
    public int compare(Vehicle o1, Vehicle o2) {
        int result = o1.getColour().compareTo(o2.getColour());
        if (result == 0) {
            result = o1.getRegistration().compareTo(o2.getRegistration());
        }
        return result;
    }

}
```

```java
package tema11_Colecciones.arboles3;

import java.util.SortedSet;
import java.util.TreeSet;

public class ShowComparator {

    public void show() {

        SortedSet<Vehicle> tree = new TreeSet<>(new VehicleComparator());
        tree.add(new Vehicle("9685KMX", 4, "azul"));
        tree.add(new Vehicle("1235GTR", 2, "rojo"));
        tree.add(new Vehicle("7314QWE", 4, "verde"));
        tree.add(new Vehicle("5930POI", 2, "azul"));
        tree.add(new Vehicle("1705UBG", 4, "rojo"));
        tree.add(new Vehicle("3495JZA", 2, "verde"));
        for (Vehicle v : tree) {
            System.out.println(v);
        }

    }

    public static void main(String[] args) {

        new ShowComparator().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=5930POI, wheelCount=2, speed=0.0, colour=azul]
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=rojo]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0, colour=verde]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
```

Pero si esta comparación solamente la vamos a utilizar una vez, tenemos que crear una clase solamente para un uso. Y si necesitamos ordenar los vehículos de varias maneras, tenemos que tener una clase por cada criterio de ordenación. En estos casos, podemos utilizar una clase inline anónima:

```java
package tema11_Colecciones.arboles3;
```

```
import java.util.Comparator;
import java.util.SortedSet;
import java.util.TreeSet;

public class AnonymousComparator {

    public void show() {

        SortedSet<Vehicle> tree = new TreeSet<>(new Comparator<Vehicle>() {

            @Override
            public int compare(Vehicle o1, Vehicle o2) {
                int result = o1.getColour().compareTo(o2.getColour());
                if (result == 0) {
                    result =
o1.getRegistration().compareTo(o2.getRegistration());
                }
                return result;
            }

        });

        tree.add(new Vehicle("9685KMX", 4, "azul"));
        tree.add(new Vehicle("1235GTR", 2, "rojo"));
        tree.add(new Vehicle("7314QWE", 4, "verde"));
        tree.add(new Vehicle("5930POI", 2, "azul"));
        tree.add(new Vehicle("1705UBG", 4, "rojo"));
        tree.add(new Vehicle("3495JZA", 2, "verde"));
        for (Vehicle v : tree) {
            System.out.println(v);
        }

    }

    public static void main(String[] args) {

        new AnonymousComparator().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=5930POI, wheelCount=2, speed=0.0, colour=azul]
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=rojo]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0, colour=verde]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
```

Si quisiéramos ordenar de manera descendente, cambiamos el orden entre *o1* y *o2*, es decir, hacemos que sea *o2* el que ejecute el *compareTo*:

```
package tema11_Colecciones.arboles3;

import java.util.Comparator;
import java.util.SortedSet;
import java.util.TreeSet;

public class DescendingOrder {
```

```
    public void show() {

        SortedSet<Vehicle> tree = new TreeSet<>(new Comparator<Vehicle>() {

            /*
             * Si quisiéramos ordenar de manera descendente,
             * cambiamos el orden entre o1 y o2, es decir,
             * hacemos que sea o2 el que ejecute el compareTo
             */
            @Override
            public int compare(Vehicle o1, Vehicle o2) {
                int result = o2.getColour().compareTo(o1.getColour());
                if (result == 0) {
                    result =
o2.getRegistration().compareTo(o1.getRegistration());
                }
                return result;
            }

        });

        tree.add(new Vehicle("9685KMX", 4, "azul"));
        tree.add(new Vehicle("1235GTR", 2, "rojo"));
        tree.add(new Vehicle("7314QWE", 4, "verde"));
        tree.add(new Vehicle("5930POI", 2, "azul"));
        tree.add(new Vehicle("1705UBG", 4, "rojo"));
        tree.add(new Vehicle("3495JZA", 2, "verde"));
        for (Vehicle v : tree) {
            System.out.println(v);
        }

    }

    public static void main(String[] args) {

        new DescendingOrder().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
Vehicle [registration=3495JZA, wheelCount=2, speed=0.0, colour=verde]
Vehicle [registration=1705UBG, wheelCount=4, speed=0.0, colour=rojo]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=5930POI, wheelCount=2, speed=0.0, colour=azul]
```

Los comparadores de tipo Comparator permiten ordenar de diferentes formas, por eso en la práctica se utilizan mucho. Por ejemplo, el método *sort* de la clase *Arrays* también admite indicar un comparador para saber de qué forma deseamos ordenar el array.

Cuando creemos clases que representen valores que posean un determinado orden natural, como por ejemplo un orden alfabético, numérico o cronológico, deberemos hacer que dicha clase implemente la interfaz `Comparable`, permitiendo así que los objetos de dicha clase puedan trabajar con mucho algoritmos genéricos e implementaciones de colecciones que dependen de dicha interfaz.

La mayoría de las clases estándar que representan valores y de las clases *enums* incorporadas a Java, implementan la interfaz *Comparable*, como por ejemplo la clase *String*. Las clases que definamos nosotros que representen valores también deberían implementarla.

A la hora de realizar la implementación debemos respetar una serie de reglas:

- `x.compareTo(y) == -y. compareTo(x)` para todo valor de `x` e `y`.
- La relación es transitiva, es decir, que si `(x. compareTo(y) > 0 && y.compareTo(z) > 0)` entonces `x.compareTo(z) > 0`.
- Si `x.compareTo(y) == 0` entonces `x.compareTo(z) == y.compareTo(z)` para cualquier valor de `z`.
- Aunque no es obligatorio se recomienda que `(x.compareTo(y) == 0) == (x.equals(y))`.

Si para comparar los objetos debemos comparar un atributo de un tipo primitivo, se recomienda usar los métodos estáticos de comparación `compare` de las clases boxed correspondientes, como `Long.compare()`, `Float.compare()`, etc., disponibles a partir de Java 7, en vez de usar los operadores `<` o `>`, ya que son menos verbosos y propensos al error:

```
package tema11_Colecciones.arboles3;

import java.util.Comparator;
import java.util.SortedSet;
import java.util.TreeSet;

public class CompareBoxedClasses {

    public void show() {

        SortedSet<Vehicle> tree = new TreeSet<>(new Comparator<Vehicle>() {

            @Override
            public int compare(Vehicle o1, Vehicle o2) {
                int result = Integer.compare(o1.getWheelCount(),
o2.getWheelCount());
                if (result == 0) {
                    result = Double.compare(o1.getSpeed(), o2.getSpeed());
                }
                return result;
            }

        });
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[0].accelerate(100);
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[1].accelerate(150);
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[2].accelerate(200);
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[3].accelerate(80);
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[4].accelerate(75);
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        vehicles[5].accelerate(170);

        for (int i = 0; i < vehicles.length; i++) {
            tree.add(vehicles[i]);
        }
        for (Vehicle v : tree) {
            System.out.println(v);
        }
```

```
    }

    public static void main(String[] args) {

        new CompareBoxedClasses().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=5930POI, wheelCount=2, speed=80.0, colour=negro]
Vehicle [registration=1235GTR, wheelCount=2, speed=150.0, colour=rojo]
Vehicle [registration=3495JZA, wheelCount=2, speed=170.0, colour=naranja]
Vehicle [registration=1705UBG, wheelCount=4, speed=75.0, colour=blanco]
Vehicle [registration=9685KMX, wheelCount=4, speed=100.0, colour=azul]
Vehicle [registration=7314QWE, wheelCount=4, speed=200.0, colour=verde]
```

10.2 Clase TreeMap

Esta implementación utiliza una estructura de árbol que permite que los elementos del mapa se ordenen en sentido ascendente según la clave, por lo tanto, la clase de las claves tiene que implementar la interfaz *Comparable* o bien indicar un objeto *Comparator* durante la creación del *TreeMap* .

TreeMap implementa la interfaz `SortedMap` que, a su vez, es heredera de `Map`, por lo que todo lo dicho sobre los mapas funciona con las colecciones de tipo *TreeMap*.

```
package tema11_Colecciones.arboles3;

import java.util.Map;
import java.util.SortedMap;
import java.util.TreeMap;

public class ShowTreeMap {

    public void show() {

        SortedMap<String, Vehicle> sortedMap = new TreeMap<>();
        Vehicle vehicles[] = new Vehicle[6];
        vehicles[0] = new Vehicle("9685KMX", 4, "azul");
        vehicles[1] = new Vehicle("1235GTR", 2, "rojo");
        vehicles[2] = new Vehicle("7314QWE", 4, "verde");
        vehicles[3] = new Vehicle("5930POI", 2, "negro");
        vehicles[4] = new Vehicle("1705UBG", 4, "blanco");
        vehicles[5] = new Vehicle("3495JZA", 2, "naranja");
        for (int i = 0; i < vehicles.length; i++) {
            sortedMap.put(vehicles[i].getRegistration(), vehicles[i]);
        }

        for (Map.Entry<String, Vehicle> entry : sortedMap.entrySet()) {
            System.out.printf("Matrícula -> %s Vehículo -> %s\n",
entry.getKey(), entry.getValue());
        }

    }

    public static void main(String[] args) {
```

```
        new ShowTreeMap().show();

    }

}
```

Salida por consola:

```
Matrícula -> 1235GTR Vehículo -> Vehicle [registration=1235GTR, wheelCount=2,
speed=0.0, colour=rojo]
Matrícula -> 1705UBG Vehículo -> Vehicle [registration=1705UBG, wheelCount=4,
speed=0.0, colour=blanco]
Matrícula -> 3495JZA Vehículo -> Vehicle [registration=3495JZA, wheelCount=2,
speed=0.0, colour=naranja]
Matrícula -> 5930POI Vehículo -> Vehicle [registration=5930POI, wheelCount=2,
speed=0.0, colour=negro]
Matrícula -> 7314QWE Vehículo -> Vehicle [registration=7314QWE, wheelCount=4,
speed=0.0, colour=verde]
Matrícula -> 9685KMX Vehículo -> Vehicle [registration=9685KMX, wheelCount=4,
speed=0.0, colour=azul]
```

Como podemos observar, está ordenado ascendentemente por la matrícula ya que la matrícula es de tipo `String` que implementa la interfaz `Comparable`. Si la clave fuera una clase hecha por nosotros, tendríamos que hacer que implementara *Comparable* o bien indicar un *Comparator* en la creación del *TreeMap*.

11. Pilas y colas

Una cola es un tipo de dato que sigue el principio **FIFO (*first in, first out*)** que implica que el primer elemento en ser insertado en la cola es también el primero en ser eliminado de la misma.

En el mundo real podemos encontrar este ejemplo en las colas de un banco, la cadena de impresión de documentos, etc. En el caso de la cola en el banco, la primera persona en llegar es también la primera en irse (suponiendo una única ventanilla) y en los documentos a imprimir, la impresora imprime según el orden de llegada.

`Queue<E>` es una interfaz que hereda de *Collection* que proporciona operaciones para trabajar con una cola. Veamos alguna de ellas:

- `boolean add(E e)`: inserta el elemento al final de la cola.
- `E element()`: Devuelve, pero no elimina, el principio de la cola. Lanza la excepción `NoSuchElementException` si la cola está vacía.
- `E peek()`: Devuelve, pero no elimina, el principio de la cola. Devuelve *null* si la cola está vacía.
- `E poll()`: Devuelve y elimina el principio de la cola. Devuelve *null* si la cola está vacía.
- `E remove()`: Devuelve y elimina el principio de la cola. Lanza la excepción `NoSuchElementException` si la cola está vacía.

`Deque<E>` representa una cola de doble extremo, lo que significa que se puede insertar y eliminar elementos desde ambos extremos de la cola. El nombre *Deque* es una abreviatura de *Double Ended Queue*. Admite, por lo tanto, la implementación de la cola *FIFO* como la implementación de la pila *LIFO*, que implica que el último elemento que se ha insertado, es el primero en ser eliminado: **LIFO (*last in, first out*).**

Deque hereda de *Queue*, por lo que tiene todos sus métodos y además añade los suyos propios. Veamos algunos de ellos:

- `void addFirst(E e)`: inserta el elemento al principio.

- `void addLast(E e)`: inserta el elemento al final.
- `E getFirst()`: Devuelve, pero no elimina, el primer elemento. Lanza la excepción `NoSuchElementException` si el deque está vacío.
- `E getLast()`: Devuelve, pero no elimina, el último elemento. Lanza la excepción `NoSuchElementException` si el deque está vacío.
- `E peekFirst()`: Devuelve, pero no elimina, el primer elemento. Devuelve *null* si el deque está vacío.
- `E peekLast()`: Devuelve, pero no elimina, el último elemento. Devuelve *null* si el deque está vacío.
- `E pollFirst()`: Devuelve y elimina el primer elemento. Devuelve *null* si el deque está vacío.
- `E pollLast()`: Devuelve y elimina el último elemento. Devuelve *null* si el deque está vacío.
- `E removeFirst()`: Devuelve y elimina el primer elemento. Lanza la excepción `NoSuchElementException` si el deque está vacío.
- `E removeLast()`: Devuelve y elimina el último elemento. Lanza la excepción `NoSuchElementException` si el deque está vacío.

11.1 Clase ArrayDeque

La clase `ArrayDeque<E>` implementa la interfaz *Deque* y por lo tanto, también *Queue*, ya que *Deque* hereda de *Queue*.

Veamos un ejemplo de uso de *ArrayDeque* utilizado como una cola:

```
package tema11_Colecciones.pilasYColas;

import java.util.ArrayDeque;
import java.util.Queue;

public class ShowQueue {

    public void show() {

        Queue<Vehicle> queue = new ArrayDeque<>();
        queue.add(new Vehicle("9685KMX", 4, "azul"));
        queue.add(new Vehicle("1235GTR", 2, "rojo"));
        queue.add(new Vehicle("7314QWE", 4, "verde"));
        System.out.println(queue.element());//Devuelve pero no elimina:
9685KMX
        System.out.println(queue.peek());//Devuelve pero no elimina: 9685KMX
        System.out.println(queue.poll());//Devuelve y elimina: 9685KMX
        System.out.println(queue.remove());//Devuelve y elimina: 1235GTR
        System.out.println(queue.remove());//Devuelve y elimina el último, se
queda la cola vacía: 7314QWE
        System.out.println(queue.peek());//Devuelve null
        System.out.println(queue.poll());//Devuelve null
        System.out.println(queue.element());//Lanza NoSuchElementException
porque la cola está vacía
        System.out.println(queue.remove());//Lanza NoSuchElementException
porque la cola está vacía

    }

    public static void main(String[] args) {

        new ShowQueue().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
null
null
Exception in thread "main" java.util.NoSuchElementException
    at java.base/java.util.ArrayDeque.getFirst(ArrayDeque.java:402)
    at java.base/java.util.ArrayDeque.element(ArrayDeque.java:551)
    at tema11_Colecciones.pilasYColas.ShowQueue.show(ShowQueue.java:21)
    at tema11_Colecciones.pilasYColas.ShowQueue.main(ShowQueue.java:28)
```

Veamos otro ejemplo de uso de un *ArrayDeque* utilizado como una pila:

```
package tema11_Colecciones.pilasYColas;

import java.util.ArrayDeque;
import java.util.Deque;

public class ShowDeque {

    public void show() {

        Deque<Vehicle> deque = new ArrayDeque<>();
        deque.add(new Vehicle("9685KMX", 4, "azul"));
        deque.add(new Vehicle("1235GTR", 2, "rojo"));
        deque.add(new Vehicle("7314QWE", 4, "verde"));
        System.out.println(deque.getLast());//Devuelve pero no elimina:
7314QWE
        System.out.println(deque.peekLast());//Devuelve pero no elimina:
7314QWE
        System.out.println(deque.pollLast());//Devuelve y elimina: 7314QWE
        System.out.println(deque.removeLast());//Devuelve y elimina: 1235GTR
        System.out.println(deque.removeLast());//Devuelve y elimina el último,
se queda la pila vacía: 9685KMX
        System.out.println(deque.peekLast());//Devuelve null
        System.out.println(deque.pollLast());//Devuelve null
        System.out.println(deque.getLast());//Lanza NoSuchElementException
porque la pila está vacía
        System.out.println(deque.removeLast());//Lanza NoSuchElementException
porque la pila está vacía

    }

    public static void main(String[] args) {

        new ShowDeque().show();

    }

}
```

Salida por consola:

```
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
Vehicle [registration=7314QWE, wheelCount=4, speed=0.0, colour=verde]
Vehicle [registration=1235GTR, wheelCount=2, speed=0.0, colour=rojo]
Vehicle [registration=9685KMX, wheelCount=4, speed=0.0, colour=azul]
null
null
Exception in thread "main" java.util.NoSuchElementException
    at java.base/java.util.ArrayDeque.getLast(ArrayDeque.java:413)
    at tema11_Colecciones.pilasYColas.ShowDeque.show(ShowDeque.java:21)
    at tema11_Colecciones.pilasYColas.ShowDeque.main(ShowDeque.java:28)
```

12. Clase Collections

La clase `Collections` contiene numerosos métodos estáticos para utilizar con todo tipo de colecciones, como por ejemplo, para añadir, buscar, copiar, reemplazar, ordenar, obtener el máximo o el mínimo, etc.