PROGRAMACIÓN

Desarrollo de Aplicaciones Multiplataforma

Manual del módulo

UD 8 - INTERFACES

Ander Gonzalez Ortiz

8.1. INTRODUCCIÓN

Supongamos que vamos a trabajar con clases de animales. De ellos, unos tienen la capacidad de emitir un sonido -por ejemplo, los perros, los gatos o los lobos- y otros no. Los primeros, por tanto, tendrán el método

```
void voz()
```

Sin embargo, la forma en que se ejecuta dicho método es distinta según la clase. En un objeto de la clase Gato, la ejecución de voz() hará que aparezca en la pantalla la cadena, ¡Miau!, . En cambio, en un objeto Perro mostrará, ¡Guau!». Pero todos ellos tienen en común que implementan el método voz() .

8.2. CONCEPTO DE INTERTAZ

La definición de la interfaz Sonido tendría la siguiente forma:

En Java disponemos de otra forma de expresar una funcionalidad común de algunas clases que no tienen otra relación entre ellas, en este caso las que tienen implementado el método `voz()`. A dichas funcionalidades las definiremos en las interfaces. En nuestro ejemplo hablaríamos de la interfaz `Sonido`, que consiste en implementar el método `voz()`. Diremos que las clases `Perro`, `Gato` y `Lobo` implementan la interfaz `Sonido`, ya que, entre sus métodos está `voz()`, mientras que las clases `Caracol` y `Lagartija` no.

8.2.1. Definición de una intertaz

La definición de la interfaz Sonido *tendría* la siguiente forma:

```
public interface Sonido {
     void voz(); //método de la interfaz
}
```

En la definición aparece el nombre del método, la lista de parámetros de entrada (aquí está vacía) y el tipo devuelto "`void`". Naturalmente, no se implementa el cuerpo del método, ya que este depende de la clase concreta que implemente `Sonido`.

8.2.2. Implementación de una interfaz

Es en la definición de cada clase donde se decide qué hace exactamente voz ( ) . Por ejemplo, la clase Perro se definiría así:

```
public class Perro implements Sonido {
     public void voz() {
          System.out.println{"¡Guau!")
     }
     ... //resto de la implementación de Perro
}
```

Como vemos, cuando una clase implementa una interfaz, la declara en la cabecera con la palabra reservada `implements` seguida del nombre de la interfaz, en nuestro caso, `Sonido`.

Con ello estamos declarando que la clase `Perro` implementa la interfaz `Sonido` y que, por tanto, entre sus métodos se encuentra `voz()`. La implementación de un método de una interfaz en una clase tiene que declararse `public`, como hemos hecho en nuestro ejemplo. La definición de la clase `Gato` sería:

```
public class Gato implements Sonido {
     public void voz() {
          System.out.println{"¡Miau!")
     }
     ... //resto de la implementación de Gato
}
```

En cambio, la clase `Caracol` no implementa la interfaz `Sonido` y entre sus métodos no encontraremos la implementación de `voz()`. En definitiva, cuando una clase declara en su encabezamiento que tiene implementada una interfaz concreta, se está comprometiendo a tener implementadas en su definición todas las funcionalidades de esa interfaz. En caso contrario, el compilador se encargará de mostrar un mensaje de error. Las interfaces, sin embargo, no son instanciables por sí mismas; solo podemos crear objetos de las clases que las implementan.

8.3. ATRIBUTOS DE UNA INTERFAZ

Una interfaz también puede declarar atributos. Por ejemplo, si queremos llevar la cuenta de las sucesivas versiones de nuestra interfaz `Sonido`, podemos definir el atributo entero `version`. La definición de la interfaz tendría la siguiente forma:

```
public interface Sonido {
    int version = 1;
    void voz();
}
```

El atributo `version` se convertirá automáticamente en un atributo de las clases `Gato` y `Perro`, y so lo podrá cambiarse en la definición de `Sonido`, pero no en las clases que la implementan, ya que los atributos definidos en una interfaz son `static` y `final` por defecto, sin necesidad de escribirlo explícitamente. Se accede a ellos, solo para lectura, a través del nombre de la interfaz o de una clase que la implemente, pero no desde una instancia. Por ejemplo, si queremos mostrar la versión actual de la interfaz `Sonido`, podemos poner

```
System.out.println(Sonido.version);
```

o bien

```
System.out.println(Perro.version);
```

Una interfaz puede incluso constar solo de atributos. Se pueden usar para añadir un conjunto de constantes compartidas por muchas clases distintas.

8.4. Métodos implementados en una intertaz

Hasta ahora solo hemos declarado un método en la interfaz `Sonido`. Pero, en general, una interfaz puede declarar más de uno. Entre ellos puede haber también métodos implementados en la propia interfaz. A los métodos declarados pero no implementados, como `voz()`, se les llama `abstractos`, igual que en las clases abstractas. Los que sí están implementados en la interfaz se llaman métodos de extensión y pueden ser tanto estáticos como no estáticos, públicos o privados. A los métodos de extensión no estáticos públicos se les llama métodos por defecto.

8.4.1. Métodos por defecto

Los métodos por defecto se declaran anteponiendo la palabra reservada `default` y son `public` sin necesidad de especificarlo. Vamos a añadir uno a nuestro ejemplo de los animales. Supongamos que, entre los sonidos de los animales, queremos incluir los ruidos que hacen durmiendo, y que todos emiten el mismo sonido cuando duermen. En este caso podemos incluir, dentro de la interfaz, el método `vozDurmiendo()`, que va a valer para todos los animales. Por tanto, lo implementamos en la misma interfaz.

```
public interface Sonido {
     int version = 1;
     public void voz();
     public default void vozDurmiendo() {
          System.out.println("Zzz");
     }
}
```

Este método, con su implementación, será incorporado por `Perro` y `Gato` automáticamente, y será accesible para todos los objetos de ambas clases.

```
Gato g = new Gato();
g.vozDurmiendo(); //muestra por pantalla <<ZZZ>>
Perro p = new Perro();
p.vozDurmiendo(); //también muestra <<ZZZ>>
```

No obstante, `vozDurmiendo()` se puede reimplementar en cada clase haciendo overriding de la implementación que se ha hecho en la interfaz. Supongamos que los leones son una excepción entre los animales, y rugen hasta cuando duermen.

```
public class Leon implements Sonido {
     public void voz() {
          System.out.println("¡Grrrr!");
     }
     @Overriding //de la implementación en Sonido
     public void vozDurmiendo() {
          System.out.println("¡Grrrr!");
     }
     ...//resto de la implementación de León
}
```

En este caso, el código

```
Leon le = new Leon();
le.vozDurmiendo();
```

mostraría por consola «¡Grrrr!».

8.4.2. Métodos estáticos en una intertaz

En una interfaz también se pueden implementar métodos estáticos. Si no se especifica un modificador de acceso, son `public`. Estos métodos pertenecen a la interfaz, y no a las clases que la implementan y mucho menos a sus objetos. Por ejemplo, supongamos que todos los animales, sin excepción, emitieran el mismo sonido al bostezar. Entonces podríamos implementar un método estático para los bostezos.

```
public interface Sonido {
     // ...
     public static void bostezo() { //public por detecto
          System.out.println("¡Aaaauuuh!");
     }
}
```

Este método será accesible directamente desde la interfaz `Sonido`. Para invocarlo tendremos que escribir

```
Sonido.bostezo();
```

8.4.3. Métodos privados

Además de los métodos abstractos, por defecto y estáticos, que son públicos por defecto, en una interfaz se pueden implementar métodos privados anteponiendo el modificador `private`.

Pueden ser tanto estáticos como no estáticos y no son accesibles fuera del código de la interfaz. Estos métodos están implementados (no son abstractos) y solo pueden ser invocados por el resto de los métodos no abstractos de la interfaz. En general, son métodos auxiliares para uso interno de otros métodos de la interfaz.

8.5. HERENCIA

Las interfaces pueden heredar unas de otras, heredando todos los atributos y métodos, salvo los privados, y pudiéndose añadir otros nuevos. A diferencia de las clases, una interfaz puede heredar de más de una interfaz -entre las interfaces es posible la herencia múltiple-.

8.6. VARIABLES DE TIPO INTERTAZ

También se pueden crear variables cuyo tipo es una interfaz. Con ellas podremos referenciar cualquier objeto de cualquier clase que la implemente -a ella o a una subinterfaz-.

Por ejemplo, podremos crear la variable de tipo Sonido.

```
Sonido son;
```

Con ella es posible referenciar objetos de cualquier clase que implemente Sonido.

```
son = new Gato();
```

La sentencia

```
son.voz();
```

mostrará en la pantalla el mensaje, ¡Miau!, ya que se ejecuta el método voz implementa- do en la clase Gato. Esa misma variable puede referenciar a un objeto Perro.

```
son = new Perro();
```

con lo cual la misma sentencia

```
son.voz();
```

escribirá «¡Guau!», que es el mensaje implementado en la clase Perro.

Este es otro ejemplo de selección dinámica de métodos. La misma línea de código produce efectos distintos -ejecuta métodos distintos-, dependiendo del objeto referenciado por la variable son. Se decide en tiempo de ejecución qué implementación concreta del método se va a ejecutar.

Es importante señalar que cuando usemos una variable de tipo `Sonido` para referenciar un objeto de la clase `Perro`, solo tendremos acceso a los métodos que pertenezcan a la interfaz `Sonido` y a los heredados de `Object`, pero no a los otros métodos implementados en la clase Perro. De esta forma, el objeto se manifestará exclusivamente como un objeto que emite un sonido. Esto va a ocurrir con todas las variables de tipo interfaz.

8.7. CLASES ANÓNIMAS

Lo más común es que las interfaces sean implementadas por clases, de las que luego se crearán diversos objetos. Sin embargo, hay ocasiones en que una determinada implementación de una interfaz es necesaria en un solo lugar. Cuando ocurre eso, no merece la pena definir una clase nueva para crear un solo objeto del que solo interesan las funcionalidades de la interfaz.

En estos casos, podemos crear sobre la marcha, en el mismo lugar del código donde se va a usar, un objeto de una clase sin nombre, es decir, anónima, en la que se implementan los métodos de la interfaz que nos interesa. Como no dispondremos de nombre para la clase, tampoco lo tendremos para el constructor. Por eso se usa uno con el nombre de la interfaz. Asimismo, el objeto será referenciado por una variable del tipo de la interfaz. Como

ejemplo, crearemos una clase anónima que implemente la interfaz `Sonido`. Imaginemos que alguien ha encontrado un animal de una especie desconocida, que emite un extraño ruido. Mientras se identifica o no, para describir su sonido crearemos un objeto de una clase anónima que implementa la interfaz `Sonido`.

```
Sonido son = new Sonido() {
    public void voz() {
```

```
            System.out.println("¡Jajeji jojuuuu!");
    }
}; //hasta aquí hemos escrito una sentencia, terminada en punto y coma
son. voz();
```

Por pantalla obtendremos el mensaje «¡Jajejijojuuuu!». En realidad, no hemos definido ninguna clase. Lo que hemos hecho es crear un objeto sin clase, que nos permite describir el sonido que emite, aunque es costumbre hablar de clases anónimas. Además, todo esto se implementa en una línea de código ejecutable, no en una clase ni archivo aparte. Por eso se dice que su definición es local. Vamos a ver un ejemplo algo más útil.

8.8. ACCESO ENTRE MIEMBROS DE UNA INTERTAZ

Es importante destacar que, dentro de una interfaz, todos los métodos `default` y `private` tienen acceso entre ellos y, a su vez, pueden acceder a los métodos abstractos. No olvidemos que estos últimos son invocados a través de un objeto de alguna clase (aunque esta sea anónima}, donde ya están implementados.

8.9. SINTAXIS GENERAL

La sintaxis general de la definición de una interfaz tiene la siguiente forma:

```
tipoDeAcceso interface NombreInterfaz {
    //atributos: son public, static y final por detecto:
    tipo atributo1 = valor1;
    //...resto atributos

    //métodos abstractos, sin implementar:
    tipo metodo1(listaParámetros1);
    //...resto métodos abstractos

    //métodos static, implementados:
    static tipo metodoEstatico1(listaParámetros1) {
        //...cuerpo de método Estatico1
    }
    //...resto métodos estáticos

    //métodos por defecto, implementados:
    default tipo metodoDefault1(listaParámetros1) {
        //...cuerpo de metodoDefault1
    }
    //...resto métodos por defecto

    //métodos privados, solo accesibles dentro de la interfaz
    prívate tipo metodoPrivado1(listaParametros1) {
        //...cuerpo de metodoPrivado1
    }
    //...resto métodos privados
}
```

Si se omite `tipoDeAcceso`, el acceso a la interfaz está restringido al paquete en el que está incluida. Si es `public`, la interfaz podrá ser importada desde otro paquete mediante una sentencia `import`.

Una clase puede implementar más de una interfaz, para lo cual deberá declararlas, separadas por comas, en el encabezamiento, e implementar todos los métodos abstractos de cada una de ellas. En general, la declaración de una clase que implementa una o más interfaces tiene la siguiente forma:

```
tipoDeAcceso class NombreClase implements Interfaz!, Interfaz2,... {

}
```

Cuando una clase implementa varias interfaces, puede haber algún método declarado en más de una de ellas con el mismo nombre y la misma lista de parámetros -el tipo devuelto tendría que ser también el mismo automáticamente; si no, daría error de compilación-. En este caso, en la clase se hará una única implementación del método.

En general, decimos que una clase implementa una interfaz si tiene implementados todos sus métodos abstractos, como hemos hecho en nuestros ejemplos. Sin embargo, se puede dejar alguno sin implementar con tal de que la clase se declare como abstracta.

Cuando una interfaz hereda de otra u otras, su definición tendrá la siguiente forma:

```
public interface nombrelnterfaz extends superlnterfaz1, superlnterfaz2,.. {
    //... definición de la interfaz
}
```

8.10. UN PAR DE INTERFACES DE LA API

En programación hay operaciones tan necesarias y tan frecuentes, que merecen entrar a formar parte de las interfaces de la API. Una de esas operaciones es la de comparar valores para buscar u ordenar objetos. Con este fin, los desarrolladores de Java han implementado un par de interfaces, `Comparable` y `Comparator`. Las estudiamos aquí a modo de ejemplo. Además, las vamos a usar frecuentemente a partir de ahora.

8.10.1 lnterfaz Comparable

Cuando queramos establecer un criterio de comparación natural o por defecto entre los objetos de una clase, haremos que implemente la interfaz `Comparable`, que consta de un único método abstracto,

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

Este método será el encargado de establecer un criterio de ordenación entre los objetos de la clase. Para comparar dos objetos `ob1` y `ob2` de una determinada clase que implemente `Comparable`, escribiremos

```
obl.compareTo(ob2)
```

que devolverá un número entero.

Si en una ordenación el objeto `ob1` debe ir antes que el objeto `ob2`, el método devolverá un número negativo; si debe ir después, devolverá un número positivo, y si deben ser iguales a efectos de ordenación, devolverá un cero.

- ob1.compareTo(ob2) < 0 si ob1 va antes que ob2.
- ob1.compareTo(ob2) > 0 si ob1 va después que ob2.
- ob1.compareTo(ob2) = 0 si ob1 es igual que ob2.

Si observamos la implementación de `compareTo()`, llama la atención el cast `Socio` aplicado al parámetro `ob` como ya hicimos con el método `equals()` de la clase `Object`, que vimos en la unidad dedicada a las clases. La razón es que, para que la interfaz sea útil para cualquier clase de Java, su parámetro de entrada debe ser de la clase `Object`, que es la más general posible. Esto obliga a que, en el cuerpo de definición del método, haya que aplicarle un cast que le diga al compilador que, en realidad, es un objeto `Socio`, y permita acceder así al atributo `id`. En la implementación del método, el `id` del objeto que llama a `compareTo()` es accesible directamente, ya que estamos accediendo al atributo desde el código del propio objeto (`this`) que hace la llamada. En cambio, `ob` es una referencia a un objeto externo, de tipo `Object`, que se le pasa como parámetro, y para acceder a su `id` deberemos poner el cast delante.

En casos como este, en que se usa un atributo de tipo numérico en la comparación, hay una implementación mucho más sencilla:

```
int compareTo(Object ob) {
     return id – {(Socio)ob}.id;
}
```

Debemos tener en cuenta que el número positivo -o negativo- devuelto no tiene por qué ser 1 o - 1. Lo que importa es si es positivo, negativo o 0.

Vamos a aplicar todo esto a la comparación de dos objetos `Socio`.

```
Socio s1 = new Socio(3, "Anselmo", "11-07-2002");
Socio s2 = new Socio(l, "Josefa", "21-11-2001");
int resultado = s1.compareTo(s2);
System.out.println(resultado);
```

Se mostrará por pantalla un número positivo, ya que se iría después de s2 en una ordenación por número id. En cambio, si la tercera línea fuera

```
int resultado = s2.compareTo(s1);
```

La interfaz `Comparable` ha sido creada por los desarrolladores de Java y la reconocen otras clases de la API. Entre ellas, la clase `Arrays`, donde se implementa el método `sort()`, que sirve para ordenar una tabla. Por ejemplo, si declaramos e inicializamos la tabla de números enteros

```
int[] tabla = {5,3,6,9,3,4,1,0,10};
```

la sentencia

```
Arrays.sort(tabla);
```

ordena la tabla `tabla` por el orden natural de sus elementos que, en el caso de los números enteros, es el orden creciente.

Es importante tener en cuenta que el método `sort()` no devuelve una copia ordenada de `tabla`, sino que ordena la tabla original. Si queremos mostrarla sin necesidad de un bucle que la recorra, podemos recurrir a otro método de la clase `Arrays`

```
System.out.println(Arrays.toString(tabla));
```

que mostrará por pantalla

```
[O,1,3,3,4,5,6,9,10]
```

En este ejemplo hemos usado `sort()` en una tabla cuyos elementos son de un tipo primitivo, pero también sirve para ordenar una tabla de objetos de cualquier clase, con tal de que esta tenga implementada la interfaz `Comparable`. Usará el orden natural de la clase, es decir, el establecido por el método `compareTo()`. A continuación, se muestra un ejemplo: vamos a crear una tabla de objetos `Socio` que, de paso, inicializaremos de una forma nueva:

```
Sonido son;
```

Para ordenar los utilizaremos el método `sort()`, ya que `Socio` tiene implementada la interfaz `Comparable` que, como acabamos de ver, se basa en el atributo `id` creciente, escribiremos:

```
Arrays.sort(t);
```

Podemos mostrar la tabla ya ordenada usando la sentencia

```
System.out.println(Arrays.deepToStirng(t));
```

donde, en lugar de `toString()`, válido para tablas de tipos primitivos, hemos utilizado `deepToString()` que sirve para tablas de objetos. Este método llamará al método `toString()` que hayamos implementa do en la clase `Socio` para mostrar los objetos individuales.

Se obtendrá por pantalla:

```
[Id:  1 Nombre: Juan Edad: 16
, Id: 2 Nombre: Ana Edad: 24
, Id: 5 Nombre: Jorge Edad: 18
]
```

Las edades variarán con la fecha en que se ejecute el programa.

Muchas clases de la API de Java, como la clase `String`, traen implementada la interfaz `Comparable`, con lo cual podemos comparar sus objetos y ordenarlos con el método `sort()`. En particular, las cadenas se comparan siguiendo el orden alfabético creciente.

Cuando se hacen búsquedas y ordenaciones de objetos de una clase definida por nosotros, además de la interfaz `Comparable`, debemos implementar el método `equals()`. Ambos identifican cuándo dos objetos son iguales --`compareTo()` devuelve 0 y `equals()` devuelve `true`-. Para evitar conflictos, debemos asegurarnos de que ambos resultados sean consistentes, es decir, que identifiquen como iguales las mismas parejas de objetos.

8.10.2 Intertaz Comparator

La interfaz `Comparable` proporciona un criterio de ordenación natural, que es el que usan por defecto los distintos métodos de la API, como `sort()`. Pero es frecuente que tengamos que ordenar los objetos de la misma clase con distintos criterios en el mismo programa. Por ejemplo, puede ser que necesitemos un listado de objetos `Socio` por orden alfabético de nombres, o por edades, en sentido creciente o decreciente. Para resolver este problema existe la interfaz `Comparator`, definida también en la API. Antes de usarla habrá que importarla con la sentencia:

```
import java.util.Comparator;
```

Esta interfaz tiene un único método abstracto:

```
int compare(Object obl, Object ob2)
```

Recibe como parámetros dos objetos que queremos comparar para determinar cuál va antes y cuál después en un proceso de ordenación. Devuelve un entero, que será negativo si `ob1` va antes de `ob2`, positivo si va después y cero si son iguales. Llamamos comparador a cualquier objeto de una clase que implemente la interfaz `Comparator`. Necesitaremos una clase de comparadores distinta para cada criterio de comparación que queramos emplear con objetos de una clase determinada. Pero, a diferencia de la interfaz `Comparable`, `Comparator` no se implementa en la clase de los objetos que queremos ordenar -en nuestro caso, `Socio`-, sino en una clase específica, cuyos objetos llamaremos comparadores. Por cada criterio de comparación adicional para la clase `Socio`, definiremos una nueva clase de comparadores.

Por ejemplo, si queremos ordenar una tabla de socios por orden creciente de edad manteniendo como orden natural el atributo `id`, implementaremos una clase comparadora para el atributo `edad`.

```
import java.util.Comparator;
public class ComparaEdades implements Comparator {
    @Override
    public int compare(Object ob1, Object ob2) {
         return ((Socio)o1).edad() - ((Socio)o2).edad();
    }
}
```

Igual que ocurre con la interfaz `Comparable`, `Comparator` se ha diseñado para que sirva para comparar objetos de cualquier clase. Por eso los parámetros de entrada son de tipo `Object`. Esto obliga a usar el cast correspondiente, en este caso (`Socio`).

Los métodos de ordenación de la API de Java, por ejemplo el método `sort()`, están sobrecargados y admiten como parámetro adicional un comparador que les diga el criterio de ordenación que deben emplear. Como ejemplo vamos a reordenar la lista de socios por orden de edades crecientes.

Empezamos por crear un objeto comparador de edades

```
ComparaEdades = new ComparaEdades();
```

que pasaremos al método `sort()` como argumento, junto con la tabla que queremos ordenar,

```
Arrays.sort(t, e);
```

Para mostrar la tabla ordenada escribimos

```
System.out.println(Arrays.deepToString(t));
```

obteniéndose por pantalla

```
[ Id:  1  Nombre   :  Juan Edad: 16
, Id: 5  Nombre: Jorge Edad: 18
, Id: 2 Nombre: Ana Edad: 24
]
```

En vez de declarar la variable e, podríamos haber creado el comparador en la propia llamada al método `sort()`

```
Arrays.sort(t, new ComparaEdades());
```

Otra opción, en caso de que se vaya a hacer la ordenación una sola vez, es crear una clase anónima en la llamada al método `sort()`:

```
Arrays.sort(t, new Comparator() {
    public int compare(Object obl, Object ob2) {
        return((Socio) obl).edad - ((Socio) ob2).edad;
    }
}) ;
```

Cuando queramos una ordenación en sentido decreciente, basta cambiar el signo del valor devuelto en el método `compare()` o `compareTo()`. Sin embargo, la interfaz `Comparator` lleva implementado el método por defecto,

```
default Comparator reversed()
```

que, invocado por un objeto comparador, devuelve otro con el criterio de comparación invertido.

En clases más complejas, con más atributos, podrá haber más criterios posibles de ordenación. Para cada uno de ellos habría que definir una clase comparadora específica.

La ordenación no es la única operación que precisa de un criterio de comparación. Cuando una tabla está ordenada, podemos hacer búsquedas rápidas, aprovechando el orden, por medio del método `binarySearch()`, que está implementado en la clase `Arrays`, igual que `sort()`. Para ello es indispensable que ambos métodos trabajen con el mismo criterio de ordenación. Por defecto, este criterio será el natural, implementado en la interfaz `Comparable`. Por ejemplo, si ordenamos una tabla t de números enteros por medio de la sentencia

```
Arrays.sort (t);
```

la tabla quedará ordenada en orden natural (creciente) y podremos buscar el índice del valor 10 en la tabla escribiendo

```
int índice = Arrays.binarySearch(t,10);
```

Pero si la ordenamos en orden decreciente por medio de un objeto comparador de la clase `ComparaEnterosinverso`, escribiremos:

```
Arrays.sort(t, new ComparaEnterosInverso());
```

Para buscar el valor 10, tendremos que pasar al método de búsqueda un argumento adicional con el mismo comparador:

```
int indice = Arrays.binarySearch(t, 10, new ComparaEnterosInverso());
```

En resumen, cuando en una aplicación tenemos que comparar objetos de una clase con distintos criterios, tanto si es para ordenarlos como para hacer búsquedas binarias, definimos un criterio de ordenación natural -por defecto a través de la interfaz `Comparable`, y añadimos el resto de los criterios de ordenación con clases comparadoras, es decir, que implementen la interfaz `Comparator`.