Capítulo 4

Interfaces y Herencia

Una de las grandes ventajas de los lenguajes orientados a objetos es que proporcionan muchas posibilidades de reutilización de código. El concepto reutilización se entiende desde el punto de vista de que el propio compilador se organiza de modo que sabe qué código debe usar sin necesidad de que el programador recurra a viejos métodos como el *copy-paste*. El concepto clase no es más que una evolución del concepto módulo. Por tanto, la programación orientada a objetos se basa en definiciones ya conocidas como *interface*.

4.1. Interfaces

En muchas situaciones es importante que diferentes grupos de programadores establezcan un acuerdo sobre cómo deben interactuar diferentes piezas de código. Esto permite a cada subgrupo trabajar de modo independiente sabiendo que el código producido va a funcionar correctamente en el momento de la integración. El acuerdo o contrato es lo que se conoce con el término interfaz.

Las interfaces suelen emplearse para definir APIs (Aplication Programming Interface), que son el mecanismo de comunicación entre desarrolladores de aplicaciones.

En Java el termino `interface` juega un rol fundamental. Un `interface` define un conjunto de comportamientos sin dar ninguna indicación de cómo deben programarse. De este modo, un `interface` define cómo deberían comportarse las piezas de código que quieran colaborar en resolver un problema. La existencia de un `interface` supone que existirán posteriormente clases que se comporten como el `interface` indica.

En Java se pueden definir interfaces. Los interfaces juegan un papel similar al de las clases. Es decir, se pueden considerar como tipos. Eso sí, un `interface` de Java puede sólo contener constantes, signaturas de métodos y tipos anidados. Nunca contienen cuerpos de métodos. Es obvio que, por tanto, un interfaz no puede instanciarse. Lo que si es posible es que el interfaz sea implementado por una clase o extendido por otro interfaz. Se dice que una clase implementa un interfaz cuando contiene todas los métodos que el interfaz define. Es decir, una clase implementa un iterfaz si los objetos que se instancien de esa clase verifican el comportamiento indicado por el interfaz. Se dice que un iterfaz extiende a otro cuando incluye nuevas *líneas de contrato* además de las del interfaz extendido.

Los interfaces tienen otro papel muy importante en Java. Java no permite múltiple herencia (se verá), pero los interfaces no forman parte de la jerarquía de clases, por lo que si es posible que una clase implemente múltiples interfaces, lo que permite programar algunas clases como si de hecho existiese la herencia múltiple.

4.1.1. Definición de una interfacce

Definir un interfaz en Java es similar a crear una nueva clase. Las diferencias son que se cambia la palabra reservada `class` por `interface`, que no define propiedades (salvo valores constantes) y que los métodos contienen sólo la signatura finalizada por `;`. Es decir un interfaz tiene una pinta similar a:

```
public interface ConductorDeCoche extends Interface1, Interface2{
    // declaracion de constantes, si hay alguna
    double PI = 3.141596;
    // signaturas de metodos
    // definicion de un enum con valores IZQUIERDA, DERECHA
    int girar(  Direccion direccion,
                double radio,
                double velocidadInical,
                double velocidadFinal);
    int cambiarDeCarril(Direccion direccion,
                        double velocidadInicial,
                        double velocidadFinal);
    // mas signaturas
}
```

La palabra `public` en la definición de la interfaz tiene el significado usual. Permite que la interfaz sea visible a todas las clases. Si se omite, la interfaz será sólo visible a las clases que estén en su mismo paquete.

Como se ha visto, la definición de una interfaz incluye información en su cabecera sobre si extiende otras interfaces. En tal caso, tras extends aparecerá un listado separado por comas de las intrerfaces extendidas.

Todos los métodos que se definen en una interfaz son implícitamente públicos,por lo que se puede omitir la palabra reservada `public`. De igual modo, la interfaz puede contener un listado de definición de constantes. Todas ellas son implicitamente `public`, `static` y `final`, por lo que pueden omitirse todos esos modificadores.

4.1.2. Implementación de una interface

Para implementar una interfaz debe crearse una clase que defina el cuerpo de todos los métodos definidos por la interfaz. La calse tendrá acceso a todas las constantes declaradas por la interfaz. Para indicar que la clase implementa la interfaz, tras el nombre de la clase se pone la palabra clave `implements` seguida del nombre de la interfaz. Como se ha indicado una clase puede implementar múltples interfaces, por lo que la palabra `implements` puede ir seguida de una lista separada por comas de las interfaces implementadas.

Vamos a ver un ejemplo de interfaz. Supongamos que queremos indicar que los objetos que puedan compararse en tamaño deben emplear todos la misma nomenclatura. Podemos definir la interface `Comparable` siguiente:

```
public interface Comparable {
    // Se emplean las siguientes constantes para indicar el
    // resultado de la comparacion
```

```
        public static final int MAS_PEQUENIO_QUE = -1;
        int MAS_GRANDE_QUE = 1;
        int IGUAL_QUE = 0;

        // El siguiente metodo se emplea para comparar el
        // objeto con el que se invoca contra otro. El
        // resultado indicara si el objeto es menor, mayor
        // o igual que el otro.

        public int esMasGrandeQue(Comparable otro);
    }
```

Cualquier clase que implemente `Comparable` debe permitir comparar de alguna manera los objetos de la clase. De hecho, podemos indicar esta característica simplemente afirmando que la clase implementa el interfaz `Comparable`. Por ejemplo, la siguiente clase permite comparar rectángulos.

```
public class Rectangulo implements Comparable{
    public int ancho = 0;
    public int largo = 0;
    public Punto origen;

    // Constructores
    public Rectangulo(){
        origen = new Punto(0,0);
    }
    public Rectangulo(Punto p){
        origen = p;
    }
    public Rectangulo(int larg, int anch){
        origen = new Punto(0,0);
        largo = larg;
        ancho = anch;
    }
    public Rectangulo(int larg, int anch, Punto p){
        origen = p;
        largo = larg;
        ancho = anch;
        alto = alt;
    }
    // Un metodo para mover el Rectangulo
    public void mueveACoordenadas(int x, int y) {
        origen.x = x;
        origen.y = y;
    }
    // Un metodo para calcular el area
    public int getArea() {
        return ancho*largo;
```

```
    }
    // Metodo requerido para implementar comparable
    public int esMasGrandeQue(Comparable otro){;
        Rectangulo elOtro = (Rectangulo) otro;
        if (this.getArea() <elOtro.getArea())
            return MAS_PEQUENIO_QUE;
        else if (this.getArea() == elOtro.getArea())
            return IGUAL_QUE;
        else
            return MAS_GRANDE_QUE;
    }
}
```

4.1.3. Las interfaces como tipos

Nótese que, en el programa anterior se está usando una operación de cast (la operación `(Rectangulo)`. El método `esMasGrandeQue` recibe un parámetro cuyo tipo es `Comparable`. Es decir, admite cualquier objeto de una clase que implemente `Comparable`. No podemos, por tanto, invocar `getArea` sobre `Otro` porque el compilador no sabe aún que `otro` es un `Rectangulo`. La operación de casting le informa al compilador de que el objeto recibido es un `Rectangulo`.

Cuando se define una interfaz, se está definiendo un nuevo nombre de tipo. De hecho, los nombres de una interfaz pueden aparecer en los mismo sitios que los de un tipo. Si se define una variable cuyo tipo sea el interfaz, sólo puede referenciar a objetos de clases que implementen la interfaz.

A veces podemos utilizar esta característica para hacer cosas *curiosas*. Supongamos que tenemos dos clases diferentes pero que ambas implementan `Comparable`. En tal caso, podemos programar sin problema un método como el siguiente:

```
public Object findLargest(Object objeto1, Object objeto2){
    Comparable obj1 = (Comparable)objeto1;
    Comparable obj2 = (Comparable)objeto2;
    if ((obj1).esMasGrandeQue(obj2) == MAS_GRANDE_QUE)
        return objeto1;
    else
        return objeto2;
}
```

Es decir, podemos comparar objetos que sean comparables aunque sean de clases diferentes (obviamente se necesita algo más, pues el método de cada uno de los objetos asume que el otro presenta una similitud).

4.1.4. Modificaciones de una inteface

La modificación de una interfaz es una pésima costumbre. Recuerde que cualquier clase que implemente la interfaz debe incorporar los métodos que la interfaz declara. Por lo tanto, la adición de algún método a la interfaz obliga a reprogramar las clases que implementen la interfaz antigua.

Lo habitual es que las interfaces no se modifiquen. En todo caso, si es preciso añadir nuevos elementos a una interfaz, se crea una nueva que extienda (en breve se le explica el concepto) a la anterior. De este modo cualquier nueva clase puede utilizar la nueva interfaz en lugar de la antigua

4.2. Herencia

Java permite que una clase derive de otra heredando todos sus métodos y propiedades. La clase derivada se denomina también extendida o clase hija. La clase de la que se hereda se denomina normalmente superclase, aunque a veces nos referimos a ella como clase base o madre.

La idea de la herencia es simple. Cuando creamos una clase parte de cuyo código ya existe en otra clase, podemos heredar de ella en lugar de volver a escribir el código. Es una forma muy rápida de reutilizar código.

Debe tenerse en cuenta que la clase derivada hereda tanto propiedades como métodos o subclases. Sin embargo, la clase heredada no hereda los constructores de la superclase (aunque puede usarlos, como se verá).

Para decir que una clase hereda de otra ya existente basta con poner `extends` seguido del nombre de la clase de la que se hereda en la declaración de la clase[1]:

```
public class CajaColoreada extends Caja {
    //La clase incluye un nuevo atributo
    public String color;
    // y su propio constructor
    public CajaColoreada(int largo, int ancho, int alto, String color){
        super(largo, ancho, alto);
        this.color = color;
    }
}
```

Con esto ya tenemos una `Caja` que tiene dimensiones, color, cuyo área se puede calcular .... Obviamente esto nos puede ahorrar mucho trabajo sobre todo cuando importamos métodos complejos y difíciles de testar.

Cabe destacar que en Java se define una clase muy especial denominada `Object` (definida en `java.lang`). Esta clase define e implementa el comportamiento que es común a todas las clases java. Toda clase que no hereda explicitamente de otra clase, hereda de `Object`. Dado esto, está claro que las clases java constituyen un árbol de herencia en el que `Object` es la clase raíz.

Como se ha comentado, una clase hereda lo que se haya definido en su superclase, aunque hay que aclarar algunos aspectos. Una clase hereda de su superclase todos los miembros públicos y protegidos (no importa si las clases están en paquetes diferentes). Los miembros *package privative* se heredan sólo si la clase está en el mismo paquete que la superclase. Una clase puede usar los miembros heredados, reemplazarlos, esconderlos o complementarlos con más miembros. Para ello:

[1] La clase Caja la cró en la práctica anterior

- Los miembros (campos y métodos) heredados pueden usarse como cualquier otro campo
- Podemos declarar un miembro con el mismo nombre que el de la superclase para esconder el de la superclase (no recomendado)
- Podemos escribir metodos de insatnacia con la misma signatura que los de la superclase, invalidándolos.
- Podemos declarar métodos `static` con la misma signatura que el de la superclase, escondiéndolos.
- Podemos construir constructores que invoquen el de la superclase
- Obviamente, podemos crear cosas que no están en la superclase

Cabe destacar que no se heredan los elementos `private` de la superclase, aunque si esta dispone de métodos públicos para acceder a los campos *private*, si pueden usarse.

Por último, es posible que recuerde que las clases anidadas tenían acceso a los miembros privados de la clase donde se declara. Por lo tanto una subclase publica o protected que se herede, permite acceder indirectamente a todos los miembros privados de la superclase (tomaaaaa).

Dado todo esto, cuando instanciamos un objeto de una clase que hereda de otra, ¿que estamos instanciando?. Si escribimos

`public CajaColoreada miCaja = new CajaColoreada();`

`miCaja` es de tipo `CajaColoreada`. Pero como este tipo hereda de `Caja` y este a su vez de `Object`, `miCaja` también se comporta como una caja y como un objeto. Es decir, puede usarse en cualquier entorno donde se prevea un objeto de estos tipos. Por ejemplo podemos hacer `Object obj = new CajaColoreada();` porque una caja coloreada es también un `Object`. A esto se le denomina casting implícito y puede llevar a problemas de comprensión. Por ejemplo, si después hiciésemos `CajaColoreada miCaja = obj` el compilador nos daría un error, porque `obj` no se declaró como `CajaColoreada`. Para poder hacer esa asignación necesitamos hacer previamete un casting explícito: `CajaColoreada miCaja = (CajaColoreada)obj;`. En el fondo, le estamos diciendo al compilador que le prometemos que `obj` se ha construido de modo que tiene lo necesario para ser una `CajaColoreada`.

Existe un operador lógico en java que nos ayuda a resolver el lío anterior. Es `instanceof`. Aplicado sobre un objeto y una clase, nos dice si realmente el objeto es una instancia de dicha clase. Gracias a el podemos curarnos en salud y escribir:

```
if (obj instanceof CajaColoreada) {
    CajaColoreada miCaja = (CajaColoreada)obj;
}
```

4.2.1. Escondiendo y sobreescribiendo métodos

Como se ha dicho, un método de instancia con la misma signatura que uno de la superclase sobreescribe (override) el de la superclase. Esto nos permite modificar el comportamiento definido por la superclase.

Si el método es de clase `static` lo esconde. La distinción es importante. En un método sobreescrito se accede siempre a la versión de la clase (no la de la superclase). Pero si el método es escondido depende. Pruebe, por ejemplo:

```
public class Animal{
    public static void testClassMethod(){
        System.out.println(''Metodo de clase en Animal'');
    }
    public void testInstanceMethod(){
        System.out.println(''Metodo de instancia en Animal'');
    }
```

Además cree la clase

```
public class Gato extends Animal{
    public static void testClassMethod(){
        System.out.println(''Metodo de clase en Gato'');
    }
    public void testInstanceMethod(){
        System.out.println(''Metodo de instancia en Gato'');
    }
public static void main (String[] args){
    Gato miGato = new Gato();
    Animal miAnimal = miGato; //Gato es Animal
    miGato.testClassMethod();
    miGato.testInstanceMethod();
    miAnimal.testClassMethod();
    miAnimal.testInstanceMethod();
```

Y ejecútela. Comprobará que cuado se comporta como la superclase se ejecuta el método escondido de la superclase, pero no el sobreescrito.

Por último, cuando sobreescribimos un método de la superclase podemos hacerlo más accesible que en la superclase, pero no menos (un método protected puede hacerse publico, pero no privado). Tampoco podemos convertir un método de instancia en uno de clase ni uno de clase en método de instancia.

4.2.2. Polimorfismo

Polimorfismo es un término tomado de la biología. Se refiere a la capacidad de un organismo de comportarse como si fuese organismos diferentes. Las clases de java presentan este comportamiento. Un objeto se conporta como miembro de su clase pero también puede hacerlo como miembro de todas las clase de las que hereda (hasta `Object`). Puebe el siguiente ejemplo:

Modifique la clase Caja de modo que incluya un nuevo método:

```
public void escribeDescripcion(){
    System.out.println(''Las dimensiones de la caja son '' + this.largo
        +'' por '' + this.ancho +'' por '' +this.alto);
}
```

Vamos a modificar también `CajaColoreada` sobrecargando (invalidando) este nuevo método, y añadiendo un constructor adecuado y un par de métodos:

```
public CajaColoreada(int larg, int anch, int alt, String col){
    super(larg, anch, alt);
    this.setColor(col);
}
public String getColor(){
    return this.color;
}
public void setColor(String col){
    this.color = col;
}
public void escribeDescripcion(){
    super.escribeDescripcion;
    System.out.println(''El color de la caja es '' +color);
}
```

Escriba ahora un programa ejecutable que instancie una caja y una cajaColoreada e invoque el método `escribeDescripcion` de cada una de ellas. Comprobará que la ejecución sabe cuál de los métodos debe invocar en cada caso. La máquina virtual decide el método mediante un metodo denominado invocación virtual. No se llama el propio de la variable, puede llamarse el de la superclase

4.2.3. La palabra reservada `super`

El ejemplo anterior muestra que `super` nos permite acceder a los métodos sobreescritos de la superclase. También permite acceder a campos escondidos (aunque recuerde que está mal visto esconderlos).

También hemos visto ejemplos en los que se accedía mediante super al constructor de la superclase. Para ello simplemente hay que escribir `super();` con los parámetros que sea necesario , aunque debe tenerse en cuenta que esta llamada al constructor de la superclase debe ser la primera línea del constructor de la subclase.

Debe tenerse en cuenta que si un constructor no llama explicitamente a un constructor de la superclase, el compilador de Java inserta una llamada al constructor sin parámetros de la superclase. Si la superclase no tiene dicho constructor se obtiene un error de compilación. `Object` tiene ese constructor, por lo que si la clase hereda sólo de `Object` no existe el problema. Sin embargo, en una jerarquía de herencia hay que tener cuidado con este aspecto. También debe considerarse que cada constructor de superclase llama al de su superclase hasta llegar a `Object`. Es decir, la instanciación de un objeto lleva a un encadanado de llamadas a constructores que debe tenerse en mente para evitar efectos colaterales.

4.2.4. Métodos y clases abstractas

Podemos emplear la palabra `abstract` para crear clases abstractas. Una clase abstracta es una clase que no se puede instanciar. Sn embargo, si es posible heredar de una clase abstracta e instanciar objetos de la clase que hereda de la clase abstracta.

En el fondo una clase abstracta es similar a un interfaz. Difieren en que una clase abstracta puede contener campos que no sean `static` ni `final`. Además puede contener métodos implementados. De este modo, la clase esta implementada *a medias* y deja partes de la implementación para las clases que hereden de ella.

Se usan para compartir partes de código. Por ejemplo, en aplicaciones de dibujo tenemos muchas clases diferentes (circulos, lineas, rectángulos, ...), pero todas comparten algunas propiedades (color, posición, ...) y métodos (moveTo, rotar, resize, ...). Algunos de estos métodos son iguales para todas ellas (por ejemplo moveTo). Por ello es habitual crear una clase abstracta que declare los métodos comunes e implemente los idénticos dejando el resto para las subclases.

Obviamente, si una clase abstracta implementa un interfaz, no es necesario que implemente todos los métodos del interfaz.

Por lo tanto, una clase abstracta contiene métodos implementados y otros que son abstractos. Es decir, es similare a:

```
public abstract class GraphicObject {
    // declaracion de campos
    //metodos no abstractos
abstract void metodoAbstracto();
}
```

4.3. Ejercicios

1. Cree la clase `CajaComparable`. Debe ser una clase similar a la clase `Caja` de la práctica anterior pero que implemente el interfaz `Comparable` que se definió en esta práctica. Cree otra clase ejecutable que instancie diferentes `CajasComparables` y las compare.

2. Vamos a modificar el ejercicio de la práctica 3 para que se puedan jugar a Ojos de Tigre con un dado trucado que caiga siempre en el 1. Piense un poco ¿Cómo lo haría? Seguro que se le ocurren varias maneras. Vamos a empezar por crear una clase DadoTrucado y una clase DadoBueno. Ambas clases tendrán un método lanzar(); y un método int getValor(). Con lo que ha visto de interfaces y herencia, puede imaginarse que podría crear una superclase Dado con dos métodos: lanzar() y int getValor(), pero también podría crear un interface Dado con dos métodos: lanzar() y int getValor(). ¿Qué opción cree que es mejor? Implemente un casino en el que puedan jugarse con un dado trucado si el usuario lo elije así. Fue fácil ¿no?.