Apuntes

de

Java

Tema 4: Interfaces

Uploaded by

Ingteleco

http://ingteleco.webcindario.com

ingtelecoweb@hotmail.com

La dirección URL puede sufrir modificaciones en el futuro. Si no funciona contacta por email

TEMA 4: INTERFACES

4.1.- ¿EL PORQUÉ DE LOS INTERFACES?

Java no tiene herencia múltiple (C++ sí tiene, por ejemplo). La herencia múltiple consiste en que una clase puede tener más de un padre. En un principio la herencia múltiple es atractiva, pero se pueden dar situaciones extrañas y muy difíciles de manejar como la mostrada en la figura 4.1.

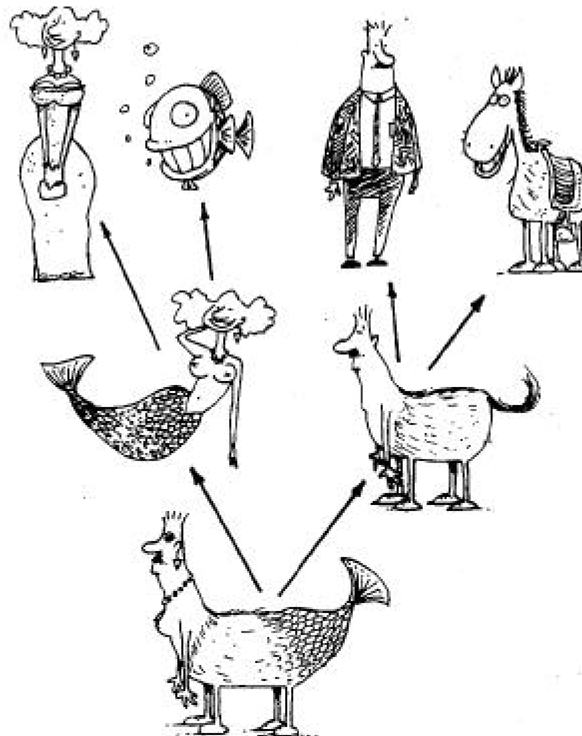

Figura 4.1: Jerarquía con herencia múltiple

La herencia múltiple presentará problemas cuando:

- Se herede varias veces de una misma clase base.
- Se hereden dos métodos implementados de forma diferente que se llamen igual

C++ y Eiffel (entre otros) admiten herencia múltiple. Esa flexibilidad deben pagarla en complejidad o ineficiencia. De forma genérica, el conflicto de la herencia múltiple ocurre porque una clase derivada puede recibir:

- Varias implementaciones para un mismo mensaje.
- Varias copias de un mismo atributo.

Por ejemplo, con la clase vehículo podría ocurrir algo así:

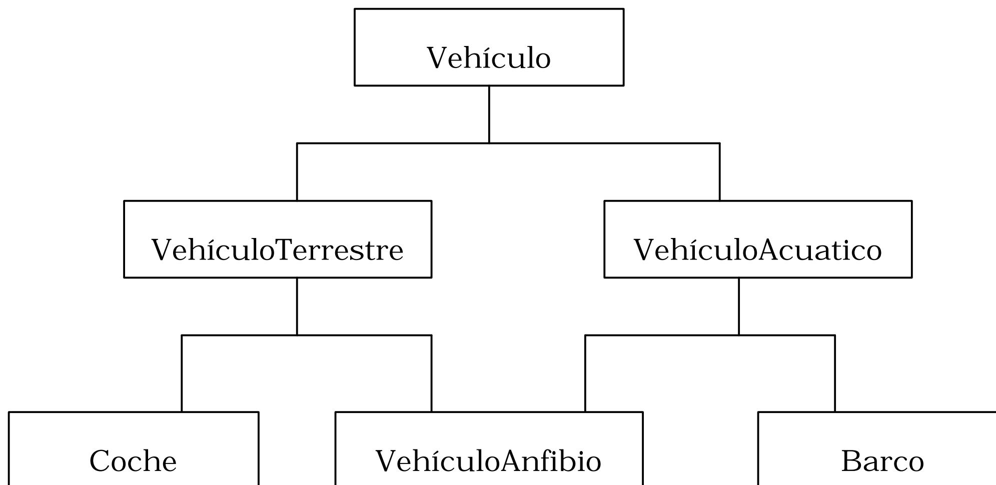

Figura 4.2: Jerarquía con herencia múltiple

¿Qué pasaría si la clase Vehículo tuviera un atributo llamado matrícula? ¿Cuántas matrículas tendría la clase VehículoAnfibio?

Y además, ¿qué pasaría si Vehículo tuviera un método que fuera implementado de forma diferente por VehículoTerrestre y por VehículoAcuatico.

Otro ejemplo: supongamos que tenemos la clase Animal, que describe el comportamiento genérico de todo animal y otra Volador, que describe el comportamiento de todo lo que vuela. ¿Cómo crearíamos la clase Pájaro? Lógicamente podría ser hija de ambas, pero Java nos lo prohibe.

Java opta por buscar la solución en otro mecanismo: los interfaces.

Un **interface** es una clase que describe sólo especificaciones de métodos, sin código ni atributos, y que sí se pueden indicar como "padres" de una jerarquía múltiple.

Es decir, con los interfaces se obtiene lo bueno de la herencia múltiple, eliminando lo malo. No se hereda codificación sólo definición de métodos.

4.2.- ¿QUÉ SON LOS INTERFACES?

Un interface es una lista de métodos (solamente cabeceras de métodos, sin implementación) que define un comportamiento específico para un conjunto de objetos. Cualquier clase que declare que implementa un determinado interface, debe comprometerse a implementar todos y cada uno de los métodos que ese interfaz define. Esa clase, además, pasará a pertenecer a un nuevo tipo de datos extra que es el tipo del interface que implementa.

Los interfaces actúan, por tanto, como tipos de clases. Se pueden declarar variables que pertenezcan al tipo del interface, se pueden declarar métodos cuyos argumentos sean del tipo del interface, asimismo se pueden declarar métodos cuyo retorno sea el tipo de un interface.

En cualquiera de estos casos, lo que realmente estamos indicando es que cualquier objeto que implemente ese interfaz puede ocupar el lugar de esa variable, de ese argumento o de ese valor de retorno; cualquier objeto que pertenezca al tipo del interface y por tanto, cualquier objeto que cumpla el comportamiento definido en ese interface.

Una clase puede implementar tantos interfaces como desee, pudiendo, por tanto, pertenecer a tantos tipos de datos diferentes como interfaces implemente. En este sentido, los interfaces suplen la carencia de herencia múltiple de Java (y además corrigen o evitan todos los inconvenientes que del uso de ésta se derivan).

4.3.- IMPLEMENTACION DE LOS INTERFACES

Un interface se parece, en cierto modo, a una clase abstracta pura (una clase en la que todos sus métodos son abstractos). Un interface se define mediante la palabra reservada interface seguida de una lista de métodos (solamente prototipos o cabeceras) sin implementación alguna:

```
interface Driveable {
    boolean startEngine();
    void stopEngine();
    float accelerate( float acc );
    boolean turn( Direction dir );
}
```

El ejemplo anterior define un interface llamado Driveable (capacitado para ser conducido) compuesto por cuatro métodos. Estos métodos podrían declararse con la cláusula abstract (por no tener implementación) pero al tratarse de un interface es algo que se sobreentiende (siempre todos los métodos de un interface van sin implementación) y no se ha indicado. De la misma forma, todos los métodos de un interface son siempre públicos y tampoco es necesario indicarlo explícitamente.

Los interfaces definen capacidades, por esta razón, suelen ser nombres adecuados aquellos que denoten capacidades (ejecutable, actualizable, ordenable, dirigible,...). Toda clase que vaya a implementar un determinado interface debe indicarlo mediante la cláusula implements, y a continuación está comprometida a implementar todos y cada uno de los métodos definidos en el interface.

```
class Automobile implements Driveable {
    ...
    public boolean startEngine() {
        if ( notTooCold )
            engineRunning = true;
        ...
    }

    public void stopEngine() {
        engineRunning = false;
    }

    public float accelerate( float acc ) {
        ...
```

```
        }

        public boolean turn( Direction dir ) {
            ...
        }
        ...
    }
```

La clase Automobile declara, mediante la cláusula implements, que implementa el interface Driveable y como se puede comprobar implementa todos los métodos definidos en el interface Driveable. Consecuentemente:

- En este momento se puede decir que la clase Automobile implementa el interface Driveable.
- La clase Automobile posee un tipo extra que es el tipo Driveable.
- Todos los objetos de la clase Automobile pertenecen además de al tipo de la clase a la que pertenecen (Automobile) al tipo Driveable.
- En cualquier lugar donde se requiera una variable del tipo Driveable podremos situar un objeto de la clase Automobile.

Si una clase declara mediante la cláusula implements que va a implementar un determinado interface y después deja algunos de los métodos de ese interface sin implementar, el compilador dará un error avisándole de tal circunstancia.

Cualquier otra clase, como por ejemplo la clase LawnMower, podría también implementar el interface Driveable. La figura 4.3 muestra esta circunstancia: el interface Driveable está siendo implementado por dos clases diferentes. Además, sería perfectamente posible que ambas clases, Automobile y Lawnmower, fuesen descendientes a su vez de cualquier otra clase base que represente un tipo básico de vehículo. En este supuesto se muestra la ventaja que proporcionan las interfaces frente a los problemas que ocasiona la herencia múltiple en C++ y como, en Java, con las interfaces, se puede prescindir completamente de ésta.

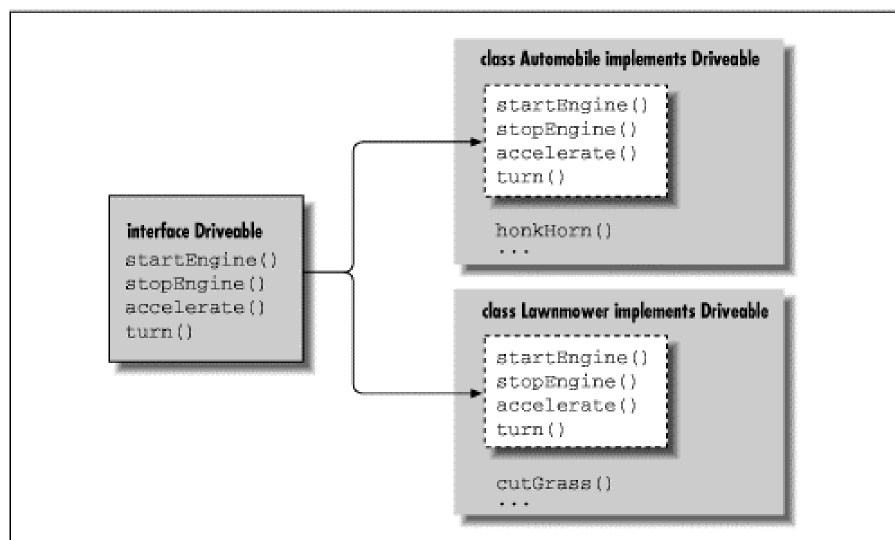

Figure 4.3: Implementación del interface Driveable

Como ya se ha comentado anteriormente, una vez que declaramos un interfaz, tenemos creado un nuevo tipo: el tipo Driveable. Ya podemos crear variables del tipo Driveable y asignárselas a objetos del tipo Driveable (objetos que pertenezcan a clases que implementen este interfaz). A continuación se muestra un ejemplo de ello:

```
Automobile auto = new Automobile();
Lawnmower mower = new Lawnmower();
Driveable vehicle;

vehicle = auto;
vehicle.startEngine();
vehicle.stopEngine();
...
vehicle = mower;
vehicle.startEngine();
vehicle.stopEngine();
```

Tanto la clase Automobile como la clase Lawnmower implementan el interfaz Driveable y por tanto todos sus objetos pueden considerarse, además, del tipo Driveable.

4.3.1.- Implementación de múltiples interfaces

Anteriormente hemos visto como dos o mas clases pueden implementar un mismo interfaz y éstas clases, a su vez, ser descendientes de una clase base. Así como una clase solamente puede derivar de una sola clase padre (Java no permite la herencia múltiple), una clase sí puede implementar más de un interfaz. Para ello lo único que debe hacer es especificar, a continuación de la palabra reservada implements, la lista de interfaces que implementa separadas por comas.

Supongamos que además del interfaz Driveable (definido anteriormente) tenemos un nuevo interfaz llamado Cloneable (capacidad para ser clonado). Este sería su código:

```
interface Cloneable {
    Object clone();
}
```

Y supongamos, además, que existe una clase base para todos los tipos de vehículos llamada Vehicle. Si la clase Automobile quisiese derivar de la clase Vehicle e implementar los interfaces Driveable y Cloneable, se haría de la siguiente manera:

```
class Automobile extends Vehicle implements Driveable, Cloneable {
    ...
    public boolean startEngine() {
        ...
    }

    public void stopEngine() {
        ...
    }

    public float accelerate( float acc ) {
        ...
    }
```

```
        public boolean turn( Direction dir ) {
            ...
        }

        public Object clone() {
            ...
        }
        ...
    }
```

Como puede observarse, la clase Automobile debe implementar los métodos de todos y cada uno de los interfaces que implemente. En estos momentos los objetos de la clase Automobile pertenecen a 4 tipos distintos:

- El tipo Automobile
- El tipo Vehicle
- El tipo Driveable
- El tipo Cloneable

4.3.2.- SubInterfaces: Herencia entre interfaces

Un interfaz puede derivar de otro interfaz, de la misma forma que una clase puede derivar de otra clase. Así, un interfaz que derive de otro parará a denominarse un subinterface:

```
interface DynamicallyScaleable extends Scaleable {
    void changeScale( int size );
}
```

El interface DynamicallyScaleable deriva del interfaz Scaleable definido anteriormente y añade un método adicional. Una clase que quiera implementar el interfaz DynamicallyScaleable debe implementar todos los métodos de ambos interfaces.

Un interfaz no puede especificar que él implementa a otro interfaz porque los interfaces no incluyen ningún tipo de implementación en sus métodos. Sin embargo, y como ya se ha comentado anteriormente, los interfaces pueden derivar de otros interfaces. De hecho, un interfaz puede derivar de tantos interfaces como quiera (¡se permite la herencia múltiple entre interfaces!), al contrario de lo que ocurre con la herencia entre clases donde sólo de permite heredar de una sola clase.

Si un interfaz quiere derivar de 2 ó más interfaces, se deben incluir éstos después de la palabra reservada extends y separados por comas. Este es un ejemplo:

```
interface DynamicallyScaleable extends Scaleable, SomethingElseable {
    ...
```

4.3.3.- Definición de variables en un interface

A pesar de que los interfaces nos permiten especificar el comportamiento de los objetos pero sin incluir nada de implementación, existe una excepción. Un interfaz puede contener identificadores de variables constantes; estas variables constantes serán "heredadas" por cualquier clase que implemente el interface. El siguiente código muestra un ejemplo:

```
interface Scaleable {
    static final int BIG = 0, MEDIUM = 1, SMALL = 2;

    void setScale( int size );
}
```

El interface Scaleable (escalable) define tres variables de tipo entero: BIG, MEDIUM y SMALL. Todas las variables que se definan en un interface son implícitamente (y de forma obligatoria) FINAL y STATIC. Por tanto, no es necesario incluir estas cláusulas en su definición, pero para mejorar la claridad se recomienda incluirlas.

Una clase que implemente el interface Scaleable tendrá acceso a estas tres variables, aunque solamente podrá consultar sus valores ya que al ser constantes (FINAL) no podrá modificarlas. El siguiente código muestra un ejemplo:

```
class Box implements Scaleable {

    void setScale( int size ) {
        switch( size ) {
            case BIG:
                ...
            case MEDIUM:
                ...
            case SMALL:
                ...
        }
    }
    ...
}
```

4.3.4.- Interfaces "vacíos"

En ocasiones los interfaces son creados únicamente para soportar constantes, de tal forma que cualquiera que implemente ese interfaz pueda tener acceso a esas constantes. Esta es una solución que trata de simular la directiva #INCLUDE de C/C++; una solución que es permitida por el lenguaje Java, pero en absoluto recomendable.

En otras ocasiones nos encontraremos con interfaces completamente "vacíos" (no poseen ni variables constantes ni métodos). Este tipo de interfaces son usados únicamente para "marcar" de alguna forma a una clase e indicar que cumple una determinada propiedad especial. El interfaz java.io.Serializable es un buen ejemplo de este tipo de interfaces. Las clases que implementan este interfaz no deben implementar método alguno ni heredan ninguna constante (el interfaz está vacío). Lo único que se consigue heredando de este interfaz es que

la clase que lo haga quede "marcada" como del tipo Serializable y por tanto, pueda ser serializada (ser convertida en un flujo de bytes para ser transportada por la red).

4.3.5.- Interfaces más característicos

- **java.io.Serializable** - permite serializar objetos
- **java.lang.Cloneable** - permite utilizar el método clone().
- **java.util.Enumeration** - permite tratar secuencias.

4.4.- DIFERENCIAS: CLASES VS. INTERFACES

4.4.1.- Clases

- Tipo que al extenderlo mediante herencia se obtienen sus mensajes y sus implementaciones (métodos y atributos).
- **Inconveniente:** Sólo se puede derivar de una de ellas.
- **Ventaja:** Menor codificación al crear nuevos subtipos ya que los mensajes vienen con sus implementaciones.

4.4.2.- Interfaces

- Tipo que al extenderlo mediante herencia se obtiene solamente mensajes.
- **Ventaja:** Se pueden heredar varios interfaces sin conflictos.
- **Inconveniente:** Hay que codificar el método de cada mensaje en cada subtipo.
- **Reglas de diseño:**
    - Preferiblemente se usarán interfaces antes que clases abstractas.
    - De esta manera no se compromete a los objetos con una jerarquía determinada.
    - Solo se usarán clases abstractas cuando se esté seguro de que los objetos a manipular no necesitan participar de más tipos y se desee reutilizar las implementaciones.

Según lo anterior: ¿Cuando se justificaría una clase con todos los métodos abstractos?

4.5.- EJEMPLOS BÁSICOS

- Creación de un interface e implementación de éste por una clase.

```
interface unInterface {
      void mensaje1();
      void mensaje2();
}

class UnaClase implements unInterface {
      void mensaje1() { System.out.println("1"); }
      void mensaje2() { System.out.println("2"); }
```

- Creación de dos interfaces e implementación de ambos por una clase.

```
interface A {
      void mensaje1();
      void mensaje2();
}

interface B {
      void mensaje3();
      void mensaje4();
}

class C implements A, B {
      void mensaje1() { System.out.println("1"); }
      void mensaje2() { System.out.println("2"); }
      void mensaje3() { System.out.println("3"); }
      void mensaje4() { System.out.println("4"); }
}
```

- Herencia simple entre interfaces.

```
interface A {
      void mensaje1();
}

interface B extends A {
      void mensaje2();
}

class C implements B {
      void mensaje1() { System.out.println("1"); }
      void mensaje2() { System.out.println("2"); }
}
```

- Herencia múltiple de interfaces.

```
interface A {
      void mensaje1();
}

interface B extends A {
      void mensaje2();
}
```

```
interface C extends A {
      void mensaje3();
}

interface D extends B, C {
      void mensaje4();
}

class E implements D {
      void mensaje1() { System.out.println("1"); }
      void mensaje2() { System.out.println("2"); }
      void mensaje3() { System.out.println("3"); }
      void mensaje4() { System.out.println("4"); }
}
```

4.6.- EJEMPLO: INTERFACES Y CLASES ABSTRACTAS

El ejemplo que se va a presentar en este apartado muestra una jerarquía de dos interfaces y una jerarquía de dos clases (una de ellas abstracta) que los implementan. Partamos de un interface ElementoVolador con un único método acelerar():

```
interface ElementoVolador
{
      void acelerar( int vel );
}
```

Pueden crearse jerarquías de interfaces heredando como habitualmente. Seguirán siendo únicamente definiciones de métodos, sin nada de implementación:

```
interface ElementoMecanicoVolador extends ElementoVolador
{
      void arreglarMecanismos();
}
```

Las clases que implementen un interface deben aportar el código para todos los métodos. Si no es así serán a su vez clases abstractas. En el siguiente ejemplo la clase Avion debe ser declarada abstracta porque no ha proporcionado ninguna implementación para el método acelerar():

```
abstract class Avion implements ElementoMecanicoVolador
{
      boolean elMotorFunciona;
      public void arreglarMecanismos()
      {
            elMotorFunciona = true;
      }
}
```

Finalmente alguna clase acabará por implementar todos los métodos del interface y será una clase no abstracta. En el siguiente ejemplo la clase AvionComercial ha implementado el método acelerar() que la clase abstracta Avion dejó sin implementar y por tanto, es una clase no abstracta:

```
class AvionComercial extends Avion
```

```
{
      int velocidad;
      public void acelerar( int vel )
      {
            velocidad += vel;
      }
}
```

4.7.- EJEMPLO PRÁCTICO DEL USO DE INTERFACES

Supongamos que queremos diseñar un método genérico y reutilizable que dado como parámetro un array con cualquier tipo de objetos dentro (incluso un array heterogéneo que contenga distintos tipos de objetos al mismo tiempo) nos devuelva ese array con los objetos ordenados según una determinada magnitud.

Existen múltiples algoritmos de ordenación de arrays que podríamos usar, pero el hecho de que nuestro método vaya a poder ordenar arrays con cualquier tipo de elementos dentro hace que surjan una serie de cuestiones:

- ¿De qué tipo tendríamos que definir el array si no sabemos el tipo objetos que va a contener?
- ¿Cómo podemos obtener el valor de cada objeto del array según el cual vamos realizar la ordenación?

Todas estas preguntas se responden mediante el uso de interfaces.

En realidad nos debería dar igual el tipo del que sean los objetos contenidos en el array, lo único que necesitamos es que todos los objetos posean un método que nos permita consultar el valor de la magnitud por la que vamos a realizar la ordenación, y para que el método de ordenación que estamos diseñando pueda usarse de manera genérica, que ese método sea el mismo para todos los objetos. Para ello podríamos crear un interfaz llamado, por ejemplo, **Ordenable** que contenga una única operación **getValor()**:

```
interface Ordenable{
      int getValor();
}
```

De esta forma, obligaremos a que todas las clases cuyos objetos quieran ser ordenados mediante nuestro método tengan que implementar el interfaz Ordenable. Cada clase implementará el método getValor() de la forma que crea más apropiada. Por ejemplo:

```
public class Persona implements Ordenable{
            ...
            public int getValor() { return estatura; }
}

public class Avion implements Ordenable{
            ...
            public int getValor() { return alas + cola; }
}
```

La clase Persona considera que la magnitud por la cual quiere ser ordenada debe ser la estatura e implementa la operación getValor() para que devuelva la estatura de la persona. La clase Avion considera que debe ser ordenada por su envergadura (alas + cola).

Al definir el interfaz Ordenable hemos definido un nuevo tipo de datos. Ahora el tipo que debe tener nuestro array está claro: deberá ser un array de objetos de tipo Ordenable.

```
Ordenable[] array;
```

Nos da igual qué objetos contenga el array, nuestro método de ordenación lo único que necesita es que todos los objetos implementen el interfaz Ordenable. Sólo tiene que tener la garantía de que todos los objetos poseen la operación getValor() que es la que él va a usar, el resto de características del objeto (ni siquiera la clase a la que pertenece) no le interesan.

La implementación del método de ordenación podría ser la siguiente:

```
public void ordena(Ordenable[] array)
{
      for (int i=0; i < array.length - 1; i++)
            for (j=i+1; j < array.length; j++)
                  if (array[j].getValor() < array[i].getValor()){
                        Ordenable temp = array[i];
                        array[i] = array[j];
                        array[j] = temp;
                  }
}
```

Esto mismo se podría haber hecho declarando una clase abstracta Ordenable, haciendo que todas las clases hereden de ella y sobreescribiendo el método getValor(). Pero, ¿qué pasaría si la clase Avión o Persona ya estuviesen heredando de otra superclase? o ¿qué pasaría si estas clases quisiesen participar en otros métodos de ordenación aparte del nuestro? Necesitaríamos que esas clases heredasen de más de una clase. Java no permite herencia múltiple entre clases y por tanto, si no fuese por las interfaces no podríamos hacerlo.

4.8.- RESUMEN

Los interfaces, en general, entonces:

- Se caracterizan porque no tienen implementación, sólo declara mensajes:
  - No tiene código asociado a los mensajes.
  - No tiene atributos (exceptuando las constantes).
- Al no tener implementación no se pueden instanciar (como las clases abstractas).
- Al igual que las clases, son una forma de crear tipos.
- Son como clases (de hecho, se compilan y generan un .class).

- Pueden definir una jerarquía de interfaces; y en ella se pueden definir padres múltiples (ya que en esa jerarquía no se hereda más que una lista de definiciones de métodos, sin código).
- Si no extienden de ningún otro interface, se definen desde cero. No hay un Object para interfaces.
- Pueden definirse variables de tipo interface; pero lo que querrá decir es que se referenciará a cualquier objeto que pertenezca a una de las clases no abstractas que implementen ese interface.

4.9.- EJERCICIOS

4.9.1.- Implementación de Enumeration

Queremos incorporar en la clase ListaEnlazada (vista en el capítulo del API) el interface de java.util.Enumeration. Para ello:

- Definiremos una clase de paquete EnumeradorLista que implemente el interface de Enumeration sobre una lista enlazada. Esto es, tiene que tener un constructor que inicialice la secuencia, un método de comprobación de final hasMoreElements() y un método de extracción del siguiente elemento de la secuencia nextElement().
- Para evitar que esta clase la tenga que manipular directamente el usuario incorporaremos a la clase ListaEnlazada un método enumeracion() que devuelva una instancia de EnumeradorLista inicializada sobre la propia lista.
- Así, desde un programa podremos para recorrer cualquier lista y hacer algo así como

```
java.util.Enumeration e = l.enumeracion();
while (e.hasMoreElements())
{
     System.out.print( "  " + e.nextElement() );
}
```

- Esta clase EnumeradorLista es el clásico ejemplo de uso de clase interna. Probar a definirla así.
- Cambiar la clase ListaEnlazada vista en anteriores capítulos de acuerdo a todo esto. Probar en el main que efectivamente la enumeración funciona.

4.9.2.- Jerarquía de figuras

Crear una clase llamada Figura de la que extiendan las clases Cuadrado, Triángulo y Círculo. Figura debe ser una clase abstracta que contenga dos métodos void dibujar() y void borrar(). Por otro lado, cada clase descendiente de Figura va a redefinir estos métodos visualizando por pantalla, en cada caso, un mensaje. (P. ej.: "Dibujando cuadrado/triángulo/círculo", "Borrando cuadrado/triángulo/círculo").

Crear por otro lado una clase Pizarra que tenga un atributo que sea un array de Figuras. Además, tendrá un método para añadir Figuras al array y otro para borrar todas las figuras del array.

Para probar el funcionamiento, realizar una clase Ejemplo que añada varias figuras (de cualquier tipo) a la pizarra. Ir viendo lo que saldría por pantalla para ver cómo funciona el polimorfismo. También se puede llamar al método borrar de la clase Pizarra para ver cómo se irían borrando todas las figuras.

4.9.3.- Jerarquía de figuras + Interfaces

Modificar el ejercicio anterior de la siguiente forma:

> Definir un interface llamado Coloreable que hace referencia a todos los objetos que admiten un color, definiendo los métodos void cambiaColor(Color c) que cambia el color del objeto y el Color queColor() que devuelve el color del objeto.
>
> Hacer que Figura implemente el interfaz Coloreable, añadiendo los métodos y atributos necesarios. ¿Dónde implementarías estos métodos? Hazlo.

4.9.4.- Jerarquías de objetos habladores

Observa la jerarquía de clases que se muestra en la figura 4.4.

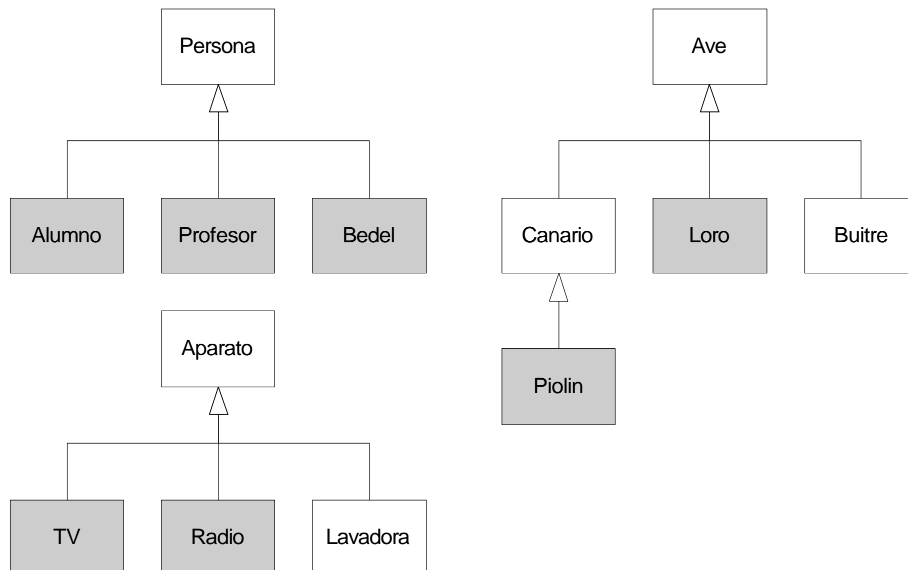

Figura 4.4: Jerarquías de clases

En la siguiente tabla se muestran las características mínimas que deben poseer cada una de estas clases:

CLASE PERSONA	Atributos: Nombre y Edad Métodos: Constructor(nombre,edad)
CLASE ALUMNO	Atributos: Nombre, Edad, Carrera y Curso Métodos: Constructor(nombre,edad,carrera,curso)

CLASE PROFESOR	Atributos: Nombre, Edad, Despacho y Email Métodos: Constructor(nombre,edad,despacho,email)
CLASE BEDEL	Atributos: Nombre, Edad, Turno y Antiguedad Métodos: Constructor(nombre,edad,turno,antiguedad)
CLASE APARATO	Atributos: Consumo y Precio Métodos: Constructor(consumo,precio)
CLASE TV	Atributos: Consumo, Precio, Teletexto(si/no) y Antiguedad Métodos: Constructor(consumo,precio,teletexto,antigüedad)
CLASE RADIO	Atributos: Consumo, Precio, Casette(si/no) y Potencia Métodos: Constructor(consumo,precio,casette,potencia)
CLASE LAVADORA	Atributos: Consumo, Precio, Alto y Ancho Métodos: Constructor(consumo,precio,alto,ancho)
CLASE AVE	Atributos: Sexo y Edad Métodos: Constructor(sexo,edad)
CLASE CANARIO	Atributos: Sexo, Edad y Canta Métodos: Constructor(sexo,edad,canta)
CLASE LORO	Atributos: Sexo, Edad, Region y Color Métodos: Constructor(sexo,edad,region,color)
CLASE BUITRE	Atributos: Sexo, Edad, VelocidadVuelo y Peso Métodos: Constructor(sexo,edad,velocidadVuelo,peso)
CLASE PIOLIN	Atributos: Sexo, Edad, Canta y NºPeliculas Métodos: Constructor(sexo,edad,canta,NºPeliculas)

Se pide:

- Implementar la jerarquía de clases de la figura 4.4 junto con los atributos y métodos de cada una de estas clases.

- Construir un interface llamado "Hablador" que posea un único método "hablar()" (sin parámetros y sin valor de retorno).

- Hacer que todas las clases que representen a entidades con la capacidad de hablar implementen este interface (éstas son las clases que en la figura 4.4 aparecen sombreadas).

  Cada una de estas clases debe implementar este interfaz de manera que el método "hablar()" visualice por pantalla el mensaje "Hola, soy un <CLASE> y sé hablar", junto con los valores de los atributos del objeto (ver la salida por pantalla al final de este ejercicio para orientarse).

- Una vez hecho esto, construir un programa que realice lo siguiente:

  - Crear un array de 7 posiciones que permita almacenar únicamente a objetos con la capacidad de hablar.

  - Crear los siguientes objetos: un LORO, un PIOLIN, un ALUMNO, un PROFESOR, un BEDEL, una TV y una RADIO. Asignar valores a los atributos de estos objetos (puedes tomar los que se muestran en el ejemplo del final).

  - Introducir estos objetos en el array.

- Recorrer el array e invocar el método "hablar()" sobre cada uno de los objetos que has introducido en él.

Una vez hecho esto, el programa debería generar una salida por pantalla parecida a esta:

```
Hola, soy un LORO y sé hablar.
        Sexo: Macho     Edad: 2
        Region: Europa  Color: Azul

Hola, soy PIOLIN y sé hablar.
        Sexo: Macho     Edad: 6
        Canta: En la ducha      Peliculas: 10

Hola, soy un ALUMNO y sé hablar.
        Nombre: Marta   Edad: 22
        Carrera: Informatica    Curso: 3

Hola, soy un PROFESOR y sé hablar.
        Nombre: Jesus   Edad: 35
        Despacho: 555-D Email: txus@eside.deusto.es

Hola, soy un BEDEL y sé hablar.
        Nombre: Dani    Edad: 40
        Turno: Tarde    Antiguedad: 10

Hola, soy una TV y sé hablar.
        Consumo: 100    Precio: 30000
        Teletexto: Si   Pulgadas: 28

Hola, soy una RADIO y sé hablar.
        Consumo: 50     Precio: 15000
        Casette: No     Potencia: 25
```