Interfaces

1 Introducción

Una de las características de la programación Orientada a Objetos es el encapsulamiento, y para su implementación Java utiliza los modificadores de acceso y las interfaces. Esto permite que el código interno de una clase no sea conocido por quienes la utilizan, simplemente se conocen los servicios o métodos que ella está en capacidad de prestar, sin necesitar conocer la forma en que lo hace. De manera que si se cambia la forma en la cual se hace, es decir el código, el funcionamiento de quienes lo usan no se ve afectado.

En este capítulo se estudiarán las interfaces, las cuales son una herramienta que usa java para separar el encabezado de un método de su implementación, en algunos casos en los que dicha separación es necesaria.

Como ya se vio en cursos anteriores, en Java pueden definirse clases abstractas[1], las cuales pueden tener varios métodos sin cuerpo, es decir, abstractos. Estos métodos tienen que ser definidos por las clases hijas. Cuando se crea una referencia a una clase abstracta, dicha referencia realmente apuntará a un objeto de una de sus clases hijas y podrá usar todos los métodos que se definieron en la clase padre, incluyendo los métodos abstractos. Esta es la primera forma de separar, de manera total, el encabezado de un método de de su implementación, pues cada clase hija puede tener una forma diferente de definir los métodos que heredó de su padre, pero a quien usa estos métodos le interesa saber que existen, el servicio que puede esperar de ellos, la información que debe suministrarle y el tipo de valor que retornarán.

La segunda forma de separar el encabezado de la implementación es el uso de *interfaces*, las cuales son elementos similares a las clases abstractas pero que definen, únicamente, métodos abstractos; por lo anterior las interfaces no pueden tener código para ningún método, sólo encabezados.

Pero las interfaces tienen otras utilidades adicionales, como por ejemplo, permitir una especie de "herencia múltiple" en las clases -lo cual se explicará más adelante en este capítulo -, e incluso el permitir agrupar clases que tienen algún comportamiento común.

2 Definición

Una interfaz en Java define un comportamiento, es decir, un conjunto de métodos que serán implementados por un grupo de clases. La sintaxis para definir una interfaz en Java es:

```
modificadorDeAcceso interface NombreInterfaz {
  /**  Listado de encabezados de métodos con el siguiente formato: .
  tipoRetorno nombreMétodoUno (lista de parámetros) ;     */
  …
}
```

Las reglas que hay para la definición de interfaces, en Java, son muy similares a las que existen para las clases, tanto en lo referente a lo que pueden contener, como en lo referente a la sintaxis; algunas de estas similitudes se presentan a continuación:

- Los modificadores de acceso permitidos para las interfaces son los mismos que para las clases, es decir, se puede emplear *public* u omitido (de paquete).

Respecto al nombre, se sigue el mismo estándar que para las clases, es decir los nombres de las interfaces deben comenzar con mayúscula; además, y dado que generalmente las interfaces definen un comportamiento, es frecuente que sus nombres incluyan la terminación "able" o "ible", como

[1] Se requiere conocer el concepto de **clase abstracta** para entender adecuadamente este capítulo.

“Moldeable”, “Comparable”, "Consumible”. Debe ser claro que esto es frecuente, pero no es obligatorio.

A diferencia de las clases, no es necesario incluir modificador de acceso para los métodos definidos en la interfaz, ya que todos ellos son públicos, incluso si este modificador de acceso se omite en su definición. Es decir, el hecho de que se omita el modificador no indica que el acceso sea de paquete; y no pueden colocarse modificadores como *protected* o *private*.

Como las interfaces no pueden contener el código para ningún método, no pueden definir métodos *static*, es decir asociados a ella, sólo pueden definir encabezados de método de instancias.

También debe quedar claro que todos los métodos definidos en una interfaz son abstractos, así no se incluya la palabra reservada *abstract*.

Ejemplo:

```
public interface Definible {
  public abstract void modificarDefinición(String descripción);
  // el siguiente método es público y abstracto:
  String obtenerDefinición();
  // Esto sería un ERROR:
  private void MostrarDefinición();
}
```

> Se debe tener presente este ejemplo, ya que será referenciado a lo largo del capítulo.

En el ejemplo anterior se está creando una interfaz llamada “Definible”, que tiene visibilidad pública. Esta interfaz tiene dos métodos, que deben ser definidos en las clases que la implementen. Estos métodos son *modificarDefinición* y *obtenerDefinición*, cada uno con el tipo de retorno y los parámetros indicados.

En una interfaz no pueden definirse atributos de instancia, pero sí pueden definirse constantes asociadas a la interfaz, es decir *static*; de hecho, si se define un “atributo” dentro de una interfaz, será visto como una constante asociada a la clase, así no se incluyan las palabras *final* y *static* en su definición. Por ejemplo:

```
public interface Motor {
  public String fuenteDePoder( );
  // definición de constantes
  public static final int VELOCIDAD_LIMITE = 200;
  double CAPACIDAD = 10.5;
}
```

En el ejemplo mostrado, la interfaz *Motor* tiene un método llamado fuenteDePoder, y también tiene **DOS** constantes, una llamada VELOCIDAD_LIMITE y otra llamada CAPACIDAD.

Así, todas las variables que se definan en una interfaz son implícitamente constantes, con modificador de acceso público y deben pertenecer a la interfaz; además deben tener un valor inicial desde el momento en que se definen, en caso contrario se genera un error en tiempo de compilación.

A diferencia de las clases, una interfaz si puede heredar de varias interfaces, es decir puede agrupar y especializar el comportamiento de varias interfaces previamente definidas. Ejemplo:

```
public interface PuedeVolar {
  public void volar();
}

public interface SuperFuerza {
  public void moverCasas();
  public void moverVehículos();
}

public interface SuperHéroe extends PuedeVolar, SuperFuerza {
  public void pelear();
}
```

De acuerdo con este ejemplo, las clases que implementen la interfaz *SuperHéroe* deberán definir los métodos *volar*, *moverCasas*, *moverVehículos* y *pelear*.

> La mayor parte de la bibliografía sobre Java hace referencia a clases, para referirse a éstas y a las interfaces. Sólo la experiencia y el contexto pueden servir de guía para identificar si una mención a clases excluye las interfaces o no.

3 Implementación de una interfaz

Para establecer que una clase implementa una interfaz se utiliza la palabra reservada *implements* en el encabezado de la clase, como se indica a continuación:

```
class NombreClase implements InterfazUno {
  // código clase
}
```

Si la clase que se va a definir hereda de una clase diferente a Object, entonces la palabra reservada implements, va después del nombre de la clase padre, como se muestra:

```
class NombreClase extends Padre implements InterfazUno {
  // código clase
}
```

Una clase puede implementar varias interfaces, simplemente colocando sus nombres separados por comas.

Ejemplo:

```
public class Artículo implements Definible, Comparable {
  …
}
```

Esto significa que la clase *Artículo* implementa dos interfaces: *Definible* y *Comparable*; por lo tanto tiene que implementar el comportamiento que se ha establecido en ellas, definiendo **todos** los métodos que tiene cada una de ellas.

Preguntas

- Hasta ahora se ha mencionado que una clase en java tiene un padre único, pero que puede implementar tantas interfaces como requiera. ¿Esto permitiría afirmar que las interfaces implementan la herencia múltiple en java? Para responder esta pregunta, evalúe las siguientes afirmaciones, que son ciertas para las clases:
  - Clase: define una estructura a partir de la cual se pueden crear objetos.
  - Herencia: Permite que los hijos compartan todo lo definido por el padre.

Cuando una clase implementa una interfaz está indicando que se compromete a tener el comportamiento definido en ella. Como todos los métodos de la interfaz son abstractos, la clase está obligada a definirlos.

Ejemplo:

```
public class Producto implements Definible {
  String nombre;
  double valor;
  // métodos que debe implementar por la interfaz Definible.
  public String obtenerDefinición(){
    return nombre + " $"+valor;
  }
  public void modificarDefinición(String descripción) {
    /* dado que el método no retorna un valor, la implementación de este método puede ser un bloque de código vacío */
  }
  // constructor de la clase
  public Producto(String nombre, double valor)  {
    this.nombre = nombre;
    this.valor = valor;
  }
}
// otra clase
public class Materia implements Definible {
  String nombre;
  String descripción;
  public String obtenerDefinición() {
    return descripción;
  }
  public void modificarDefinición(String descripción) {
    this.descripción = descripción;
  }
  // otros métodos de la clase
  public String getNombre( ) {
    return nombre;
  }
}
```

En el ejemplo anterior puede observarse que ambas clases (Producto y Materia) definen el código para los métodos *obtenerDefinición* y *modificarDefinición*, cuyos encabezados están definidos en la interfaz *Definible* que ambas clases implementan.

> Cuando una superclase implementa una interfaz, las subclases no necesitan utilizar la palabra *implements* para esa interfaz, pues ya heredan todos sus métodos y constantes a través de la superclase.

Cuando una clase implementa varias interfaces puede suceder que dos o más interfaces diferentes definan el mismo método. Si esto ocurre pueden darse los siguientes casos:

- Si los métodos tienen el mismo encabezado (tipo de retorno y parámetros) es suficiente con implementar uno de ellos en la clase.
- Si los métodos tienen diferente tipo de retorno o número y/o tipo de parámetros entonces deben implementarse todos los métodos (sobrecarga).
- Si los métodos tienen el mismo tipo y número de parámetros, pero un diferente tipo de retorno entonces se produce un error de compilación, pues una clase no puede tener dos métodos que sólo se diferencien en el tipo de retorno.

En muchos casos, como ya se mencionó, las interfaces se comportan como las clases abstractas, pues de ninguna de las dos pueden crearse instancias, pero si pueden crearse referencias, a través de las cuales se hará referencia a objetos de clases hijas, en el caso de las clases abstractas, o de clases que las implementen, en el caso de las interfaces.

Ejemplo:

```
public class EjemploInterfazDefinible {
  public static void main(java.lang.String[] args) {
    Definible objetoUno = new Producto("lápiz", 1000);
    Definible objetoDos = new Materia();

    /** Esto sería un error: Definible objeto3 = new Definible(); */
    System.out.println(objetoUno.obtenerDefinición());
    objetoDos.modificarDefinición("materia de prueba");
    System.out.println(objetoDos.obtenerDefinición());

    /* Esto también es un error:  System.out.println(objetoDos.getNombre());
       Pues se tiene una variable de tipo Definible, es decir, sólo pueden usarse los métodos definidos en esa interfaz.  Una
    posible solución sería:  */
    Materia materiaUno = (Materia) objetoDos;
    System.out.println( materiaUno.getNombre());
  }
}
```

Adicionalmente, se puede saber si un objeto implementa o no una interfaz, utilizando el operador *instanceof*:

```
if (materiaUno instanceof Definible) { /** según el código anterior, se obtendría true*/
  // lo que se deba hacer …
}
```

Preguntas

- De acuerdo con los ejemplos anteriores, ¿es posible definir un método con el siguiente encabezado:

  void pregunta(Definible objeto) { …} ?

- Si la respuesta es si, ¿objetos de qué clase podrían enviarse al método pregunta?

Ejercicio

- El siguiente código incluye un error. Identifíquelo y explique, claramente, por qué es un error.

```
public interface Medible {
  public double área( );
}
public class Habitación implements Medible {
  int largo = 10;
  int ancho = 5;
  public int área( ) {
    return (largo * ancho);
  }

}
```

4 Interfaces bandera

Algunas interfaces en Java no tienen métodos de instancias, ni definen constantes, es decir su cuerpo se limita a la definición de su encabezado. A estas interfaces se les denomina “interfaces bandera”.

Estas interfaces permiten que, al utilizar el operador *instanceof,* se puedan identificar los objetos que pertenecen a las clases que las implementan, lo cual se usa frecuentemente, para dar a esos objetos un “permiso especial” para realizar alguna actividad.

Ejemplo:

```
package java.lang;
public class Object {
. . .
  protected Object clone( ) throws CloneNotSupportedException{
    // lo que se deba hacer para clonar el objeto.
  }
}
```

La anterior es la definición del método *clone*; que, como puede verse, está definido en la clase Object, padre de todas las clases en java, sin embargo, si se intenta usar este método en un objeto de una clase cualquiera, que no lo haya sobrescrito, se presentará un error en tiempo de ejecución. Esto se debe a que el método sólo puede ser usado por objetos cuyas clases implementen la interfaz *Cloneable*, la cual es una interfaz bandera.

> Como el método clone es protegido, para que se pueda invocar este método sobre otro objeto, primero debe sobreescribirse con el modificador de acceso public.

El uso de interfaces banderas no es muy frecuente, sin embargo debe ser comprendido y recordado pues se requiere para el buen uso de varias herramientas que ofrece el lenguaje.

5 Uso de las constantes

Las constantes que están definidas en una interfaz pueden ser usadas en cualquier clase con igual sintaxis a la empleada para los atributos estáticos, es decir, con el nombre de la interfaz, un punto, y el nombre de la constante[2]. En este caso no es necesario que la clase donde se usa la constante implemente la interfaz. Ejemplo:

```
public interface Motor {
  public String fuenteDePoder( );
  public static final int VELOCIDAD_LIMITE = 200;
  double CAPACIDAD = 10.5;
}
public class EjemploUsoConstante {
  int velocidad;
  EjemploUsoConstante () {
    velocidad = Motor.VELOCIDADLIMITE;
  }
}
```

La clase *EjemploUsoConstante*, no implementa la interfaz *Motor*, pero esto no impide que pueda usar cualquiera de las constantes definidas en ella, ya que son públicas.

Cuando una clase implementa una interfaz que define una o más constantes, puede usarlas de la misma forma que las demás clases, es decir, con su nombre completo, pero el hecho de implementar la interfaz le permite usar la constante directamente, sin necesidad de adicionar el nombre de la interfaz.

Ejemplo:

```
public class Bus implements Motor {
  double nivel = CAPACIDAD;
  public String fuenteDePoder() {
    return "motor Diesel";
  }
}
```

[2] Se recomienda que los nombres de las constantes sean todos en mayúscula, y si son varias palabras se separan por una línea en la parte inferior, por ejemplo: UNA_CONSTANTE

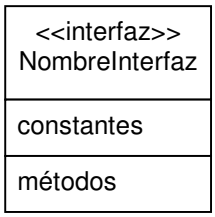
Pregunta

- Si una clase implementa dos interfaces, ¿tales interfaces pueden contener constantes con el mismo nombre?

6 Representación gráfica

Al efectuar el análisis de un problema, es muy útil realizar el diagrama de las clases e interfaces que se utilizarán en la solución. A continuación se explicará brevemente la forma de diagramar interfaces, clases y relaciones entre ellas[3].

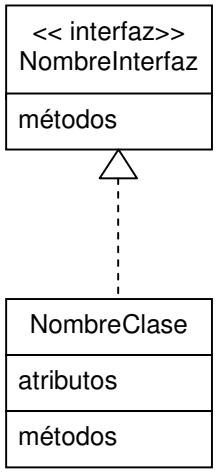
Una interfaz se representa gráficamente de la siguiente forma:



Es similar a la representación de una clase, pero tiene la palabra interfaz, encerrada entre símbolos de menor y mayor, sobre el nombre de la interfaz.

Muchas veces se omite el espacio de las constantes y simplemente se tienen dos recuadros: uno para el nombre de la interfaz y otro para los métodos.

Para indicar que una clase implementa una interfaz se representa de la siguiente forma:

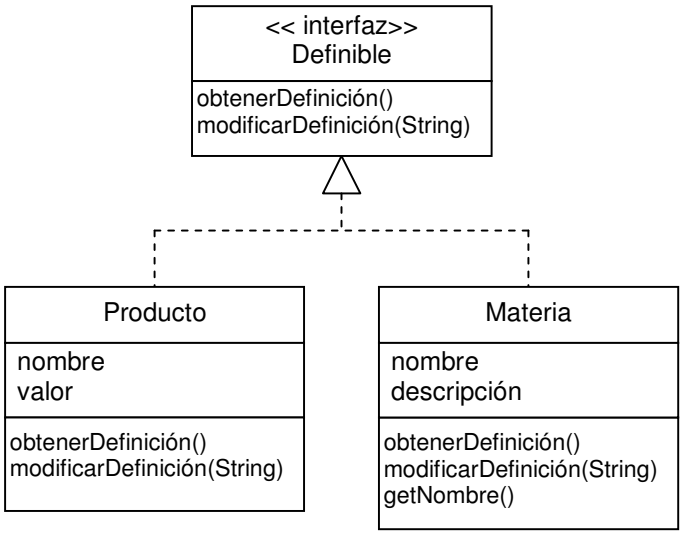


Se dibuja una línea que parte de la clase y termina en la interfaz. Dicha línea es discontinua y termina en un triángulo vacío (similar al de la herencia).

Ejemplo:

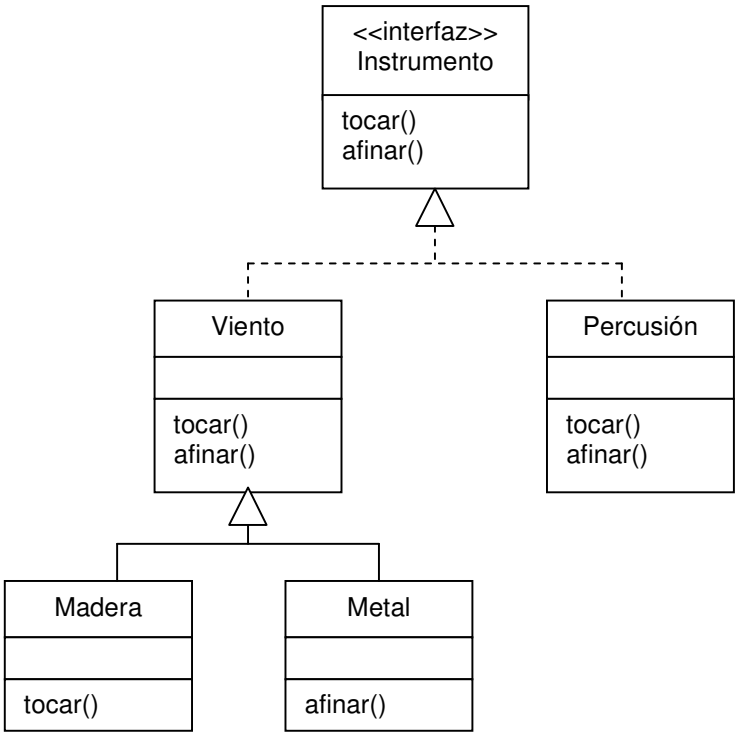
A continuación se hará el dibujo correspondiente a las clases Producto y Materia, que implementan la interfaz Definible, vista en un ejemplo anterior.

[3] Se supone un conocimiento básico de la forma de diagramar clases.



Ejercicios

- Escriba en Java el código correspondiente al siguiente dibujo de clases e interfaces:



- Teniendo en cuenta la figura anterior, ¿podría guardar varios objetos de estas clases en un arreglo? ¿De qué tipo sería el arreglo?
- Dada la siguiente descripción de un problema elabore el dibujo de clases e interfaces correspondiente:

  "Se necesita tener información de los diferentes productos que se venden en una tienda. Esta tienda en particular vende: comida (de la cual se desea saber las calorías que tiene), juguetes (de

los cuales se desea saber la edad mínima requerida) y libros (de los cuales se desea conocer su autor).

De estos productos, a los juguetes y los libros se les aplica un impuesto (IVA), pero no a la comida. Hay otros elementos a los cuales se les aplica el IVA, incluyendo algunos servicios, de manera que el concepto de impuesto está separado del concepto "Producto".

Se sabe que el IVA, para todos los elementos a los cuales se aplica, es del 12%, y que cualquier elemento al que se le pueda aplicar este impuesto debe tener un método "calcularValorIva", para poder saber el valor del IVA que le corresponde"

7 Clases abstractas e interfaces

Las interfaces se utilizan para definir un comportamiento que puede ser implementado por cualquier clase, por esto son útiles para:

- Declarar métodos que van a ser implementados por varias clases.
- Definir constantes que van a ser usadas por varias clases.
- Definir un comportamiento que pueden tener varias clases, sin importar que dichas clases no se encuentren en la misma jerarquía.
- Presentar el comportamiento que van a tener los objetos de un programa, sin revelar la clase a la cual pertenecen, es decir, un total encapsulamiento del código.

Es posible utilizar clases abstractas en lugar de interfaces para lograr algunos de los aspectos mencionados previamente, como definir métodos y constantes que van a ser utilizados por varias clases, aunque no son independientes de la herencia ni presentan un encapsulamiento total del código. Las clases abstractas son clases que no tienen todos los elementos necesarios para poder ser crear instancias de ellas. Aunque pueden definir atributos y métodos con cuerpo, también pueden definir métodos abstractos, que deben ser implementados por las clases hijas. Las clases abstractas, por lo general, se utilizan para iniciar una jerarquía de clases que se especializan cada vez más.

Las diferencias entre una interfaz y una clase abstracta se pueden sintetizar en:

- Una clase abstracta es una clase que requiere una posterior especialización, pero puede contener atributos y métodos, tanto abstractos como no abstractos; mientras que una interfaz es sólo una especificación de un comportamiento.
- Una interfaz no tiene ningún tipo de interferencia con la herencia. De hecho una clase puede ser subclase de otra e implementar tantas interfaces como necesite, incluso si su clase padre no las implementa.
- Mientras que una clase puede heredar solo de otra clase, una interfaz puede heredar de tantas interfaces como desee.

Probablemente se empleen más las interfaces que las clases abstractas. Una clase abstracta es utilizada cuando se desea iniciar una jerarquía de clases más especializadas. Las interfaces se utilizan siempre que se desea decir "necesito llamar un método que tenga este nombre, reciba esto, me devuelva esto y esté disponible para todas las clases que lo requieran".

Ejemplos

1 Elaboración de un diagrama de clases

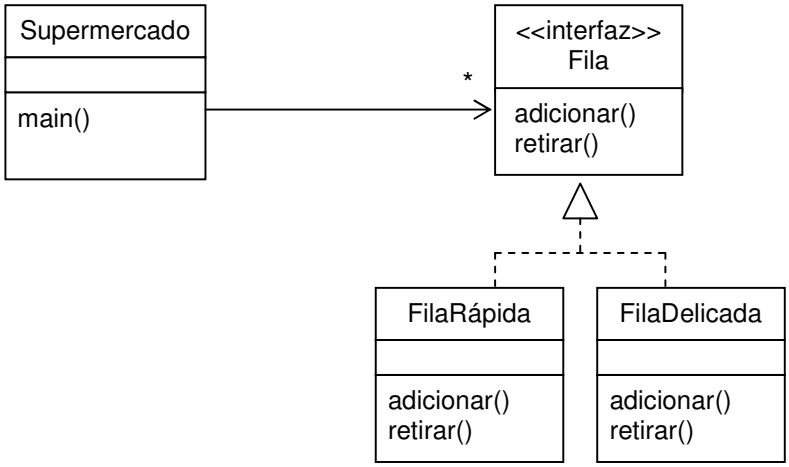
Se tiene que realizar un programa que controle las filas en las cajas de pago de un supermercado. El supermercado tiene dos tipos de cajas:

- Caja rápida, en ella es muy importante que la gente trabaje rápido. No importa que el proceso de empaque en el carrito no sea óptimo.

- Caja delicada, no es tan importante la rapidez, pero sí que los artículos contenidos en el carrito sean tratados con sumo cuidado.

> Cuando se habla de una "fila" se hace referencia a un grupo de objetos (algo similar a un arreglo). La característica principal es que todo objeto que se adiciona queda de último, y sólo puede sacarse el primer objeto de la fila. Esta estructura también se conoce como una estructura FIFO (por First In First Out - primero en entrar primero en salir).

Para realizar este programa se pensó, inicialmente, en tener dos clases: una llamada "Fila", con los métodos *adicionar* (que siempre adiciona al final) y *retirar* (que siempre obtiene el primero de la fila); y la otra clase llamada "Supermercado", la cual utiliza los métodos de la Fila, para controlar el comportamiento de los clientes en las cajas; sin embargo como las diferentes cajas tienen requerimientos distintos, se decidió que la estructura del programa debía ser:



Este esquema presenta las siguientes ventajas:

- La clase Supermercado tiene un arreglo de objetos de tipo "Fila", para contener en él objetos de las diferentes cajas, y se utilizan los métodos adicionar y retirar. En este arreglo pueden incluirse referencias a objetos de tipo FilaRápida o FilaDelicada. Si más adelante aparecen otros tipos de filas, el arreglo estará en capacidad de utilizarlos y la clase Supermercado no deberá sufrir modificaciones traumáticas, pues el único cambio debería ser en el punto donde se crean los objetos, así:

```
Fila[ ] cajas;
//  ….
int condición =  // … se establece la condición para saber que fila crear.
switch (condición) {
  case 1 : cajas[posición]  = new FilaRápida(); break;
  case 2 : cajas[posición] = new FilaDelicada(); break;
  //  Se pueden adicionar tanto cases como tipos de filas
}
```

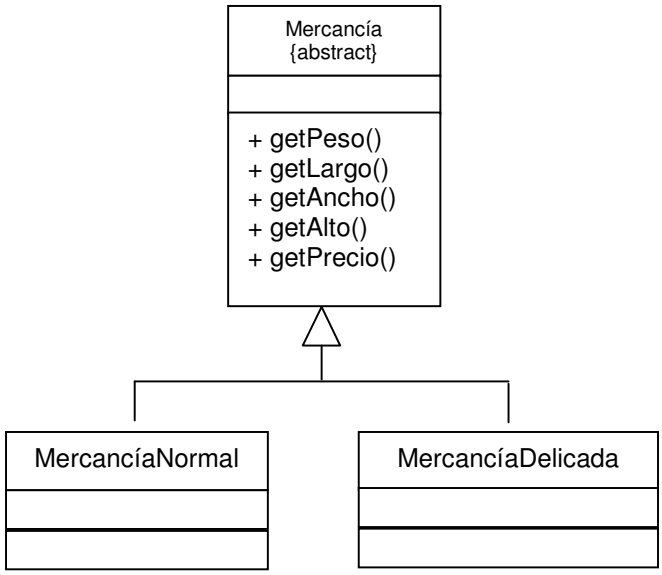
- Como Fila es una interfaz, puede ser implementada por cualquier clase que requiera garantizar este comportamiento, sin que su jerarquía de herencia se vea afectada. Por ejemplo, suponga que una persona elaboró una clase llamada "Colección", que permite manejar una colección de objetos. Los métodos de esta clase son muy rápidos y manejan con cuidado los objetos contenidos en ella, pero no corresponden exactamente a los métodos definidos en la interfaz Fila. Esta clase no debe ser modificada, pues ya está siendo utilizada, así que ¿cómo se puede aprovechar, para crear un nuevo tipo de fila? Simplemente se crea una nueva clase que herede de ella y que implemente la interfaz Fila; de esta forma basta con implementar los métodos

adicionar y retirar, tal como lo requiere la interfaz Fila, aunque al implementarlos lo que se haga sea usar los métodos de la clase Colección.

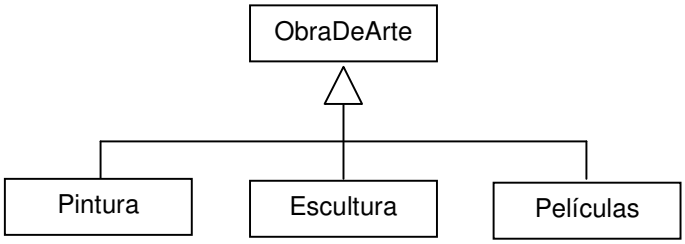
- La interfaz Fila, hasta ahora, ha facilitado el trabajo con la clase Supermercado, pero no está limitada a este uso. De hecho cualquier clase, en cualquier aplicación que se esté desarrollando puede hacer uso de ella, si esto es conveniente.

2 Interfaz Empacable

Una empresa de transportes está haciendo el análisis de un sistema que le permita guardar la información de las diferentes mercancías que envían. Ha definido que le interesa saber el peso, las medidas y el precio de envío de todos los objetos, y por lo tanto pensó en definir una clase abstracta de la cual heredan los diferentes tipos de mercancías, así:



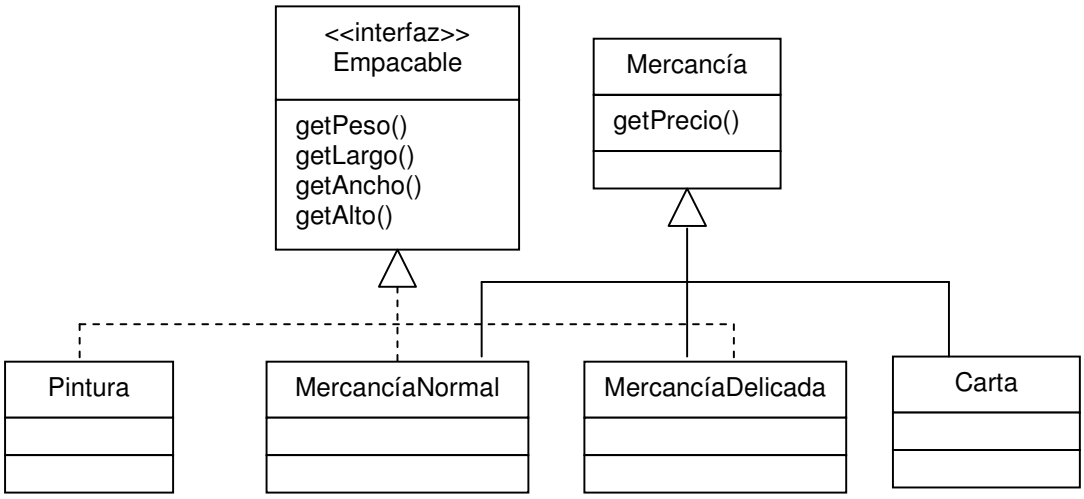
Sin embargo, un amigo del jefe, que es administrador de un museo de arte, le indicó que también le interesa saber el peso y medidas de las pinturas, pues en ocasiones deben enviarlas para exposiciones en otras ciudades. El museo de arte ya tiene definido su diagrama de clases de la siguiente forma (por simplicidad se omiten los atributos y métodos):



Como se han dado cuenta de que les interesa saber el peso y medidas de muchos objetos, sin importar la jerarquía de clases a la cual pertenecen, han concluido que lo mejor es definir una interfaz, denominada *Empacable*.

La solución con una interfaz es más útil para la empresa de transportes, pues ahora desean incluir el envío de correspondencia, y para esto no es necesario saber las medidas de los sobres, aunque sí su precio de envío. Es decir, la nueva clase *Carta* puede heredar de la clase *Mercancía*, pero no requiere implementar la interfaz *Empacable*.

El diagrama que representa la solución definitiva es:



Preguntas y ejercicios

- Indique qué errores existen en el siguiente código y cómo podrían solucionarse:

```
interface InterfazUno {
   void unMétodo( );              }
interface InterfazDos {
   int unMétodo(int valor);       }
interface InterfazTres {
  int  unMétodo( );               }
interface InterfazCuatro extends InterfazUno, InterfazTres {   }
class ClaseUno {
  public int unMétodo(int valor) {
    return 1;
  }
}
class ClaseDos implements InterfazUno, InterfazDos {
  public void unMétodo( ) {   }
  public int unMétodo(int valor) {
    return 1;
   }
}
class ClaseTres extends ClaseUno implements InterfazDos {
  public int unMétodo(int valor) {
    return 1;
  }
}
class ClaseCuatro extends ClaseUno implements InterfazTres {
  public int unMétodo( ) {
    return 1;
   }
}
class ClaseCinco extends ClaseUno implements InterfazUno {   }
```

- Seleccione la respuesta correcta. Cuando una clase implementa una interfaz ¿qué debe hacer?
  - Redefinir cada una de las constantes existentes en la interfaz.
  - Declarar y proporcionar un cuerpo para cada uno de los métodos existentes en la interfaz
  - Declarar una variable para cada una de las constantes existentes en la interfaz.
  - Incluir un método privado para cada uno de los métodos existentes en la interfaz.
- Seleccione la respuesta correcta. ¿Cuál de las siguientes afirmaciones es verdadera?

- Una clase hija puede extender a una padre o implementar una interfaz, pero NO las dos cosas.
- Una clase hija puede extender sólo a una padre y puede implementar sólo una interfaz.
- Una clase hija puede extender sólo a una padre y puede implementar cero o más interfaces.
- Una clase hija puede extender cero o más clases padre, y puede implementar cero o más interfaces.

- ¿Es posible tener una interfaz que sólo defina constantes? Dé un ejemplo de un conjunto de constantes que pueda ser definido en una interfaz, de forma que dichas constantes puedan ser usadas en varias clases de un programa.

- Realice el dibujo de las clases e interfaces para el siguiente enunciado, y escriba el correspondiente código en Java:

  En el diario *El planeta* se desea tener un registro de todas las personas que están trabajando actualmente, de las cuales se guarda su nombre, teléfono y cargo. De los empleados administrativos se desea guardar su nivel de educación, mientras que de los periodistas se guardan los años de experiencia y la sección del periódico donde trabajan actualmente. Por otro lado, para efectos de contabilidad, los empleados del diario (periodistas o administrativos) se consideran elementos que generan un costo para la empresa, es decir, puede saberse cuánto le cuesta al diario mensualmente cada uno de ellos. Esto mismo puede ser aplicado a otros elementos del diario (no necesariamente personas); por ejemplo, también pueden saber cuánto le cuesta al diario mensualmente el mantenimiento de cada uno de los vehículos que tiene para cubrir las noticias, o el mantenimiento de los equipos de impresión.

- Existe una interfaz llamada java.lang.*Comparable*, que define un método con el siguiente encabezado:

  public int compareTo(Object obj)

  Las clases que implementen esta interfaz deben definir el método compareTo, teniendo en cuenta las siguientes reglas: Si el objeto que llega como parámetro es menor que el objeto actual (el objeto sobre el cual se invoca el método), debe retornarse un entero positivo. Si es mayor se retorna un entero positivo, y si ambos objetos son iguales se retorna cero.

  Elabore un programa en Java que permita ordenar de mayor a menor varios equipos de fútbol. Usted debe definir la clase EquipoDeFútbol y utilizar la interfaz Comparable para establecer cuándo un equipo es mayor, menor o igual a otro, usando como criterio de comparación los puntos obtenidos en el torneo.