INTERFACES

Una interfaz es una colección de métodos abstractos y propiedades constantes. En las interfaces se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describen la lógica del comportamiento de los métodos.

Ya hemos visto cómo definir clases normales, y clases abstractas. Si queremos definir un interfaz, se utiliza la palabra reservada interface, en lugar de class, y dentro declaramos (no implementamos), los métodos que queremos que tenga la interfaz:

```
public interface MiInterfaz
{
        public void metodoInterfaz();
        public float otroMetodoInterfaz();
}
```

Después, para que una clase implemente los métodos de esta interfaz, se utiliza la palabra reservada implements tras el nombre de la clase:

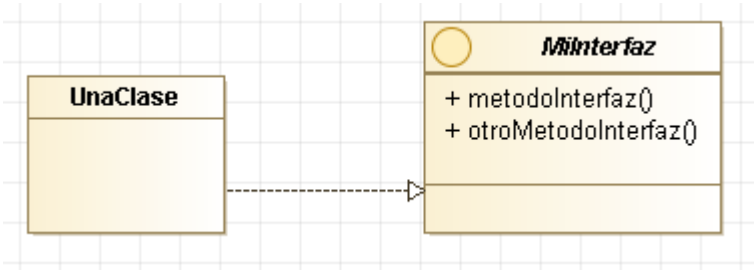
```
public class UnaClase implements MiInterfaz {
        public void metodoInterfaz(){
        ... // Código del método
        }
        public float otroMetodoInterfaz() {
        ... // Código del método
        }
}
```

Notar que, si en lugar de poner implements ponemos extends, en ese caso UnaClase debería ser una interfaz, que heredaría de la interfaz MiInterfaz para definir más métodos, pero no para implementar los que tiene la interfaz. Esto se utilizaría para definir interfaces partiendo de un interfaz base, para añadir más métodos a implementar.

Nota: Una clase puede heredar sólo de otra única clase,
pero puede implementar cuantos interfaces necesite:

```
public class UnaClase extends MiClase implements MiInterfaz1, MiInterfaz2, MiInterfaz3 {
...
}
```

A continuación vemos la representación de la interfaz en un diagrama UML y la implementación de la misma con una clase de ejemplo.



Cuando una clase implementa una interfaz se está asegurando que dicha clase va a ofrecer los métodos definidos en la interfaz. Se dice que la clase se compromete mediante un contrato a cumplir con todos los métodos que especifica la Interfaz.

Cuando heredamos de una clase abstracta, heredamos todos los campos y el comportamiento de la superclase, y además podemos definir algunos métodos que no habían sido implementados en la superclase.

> Desde el punto de vista del diseño, podemos ver la herencia como una relación **ES DEL TIPO**, mientras que la implementación de una interfaz sería una relación **ACTÚA COMO**.

PROBLEMATICA

Tenemos una aplicación para gestionar nuestra colección de DVDs, en la que podemos añadir películas a la colección, eliminarlas, o consultar la lista de todas las películas que poseemos. En esta aplicación tenemos una clase JDBCPeliculaDAO que nos ofrece los siguientes métodos:

```
Public class JDBCPeliculaDAO {

        public void addPelicula(PeliculaTO p);

        public void delPelicula(int idPelicula);

        public List<PeliculaTO> getAllPeliculas();

}
```

Esta clase nos permitirá acceder a los datos de nuestra colección almacenados en una Bd. *"Siempre que en algún lugar de la aplicación se haga un acceso a los datos se utilizará esta clase"*. Por ejemplo, si introducimos los datos de una nueva película en un formulario y pulsamos el botón para añadir la película, se invocaría un código como el siguiente:

```
JDBCPeliculaDAO jdbcPeliculadao = GestorDAO.getPeliculaDAO();
jdbcPeliculadao.addPelicula(pelicula);
```

Punto de Acceso

La clase auxiliar GestorDAO tiene un método estático que nos permitirá obtener una instancia de los objetos de acceso a datos desde cualquier lugar de nuestro código. En el caso anterior este método sería algo así como:

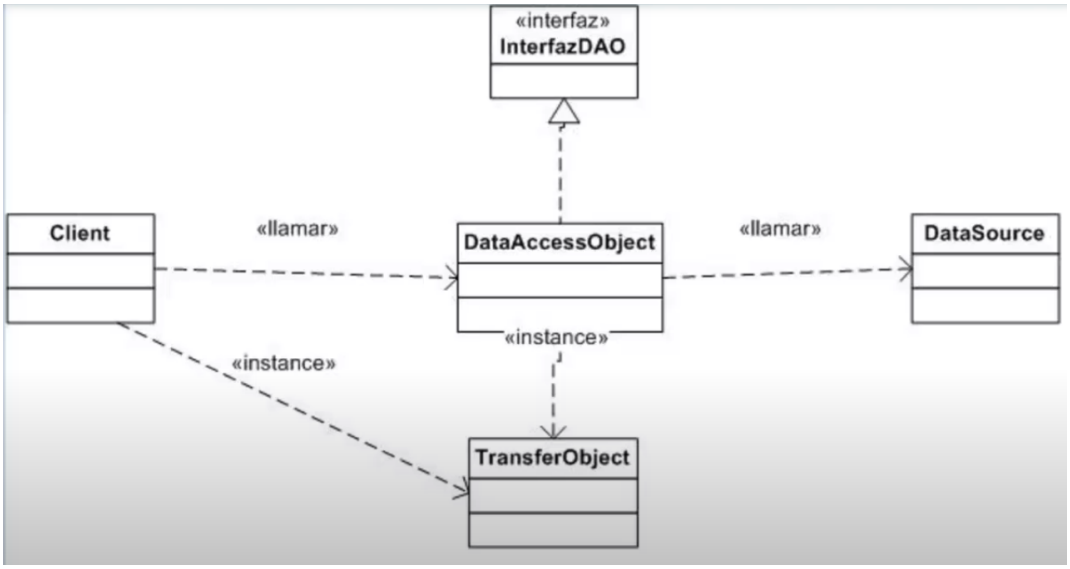
```
public static JDBCPeliculaDAO getPeliculaDAO() {
        return new JDBCPeliculaDAO();
}
```

Como la aplicación sufre modificaciones y decidimos pasar a almacenar los datos en un la web mediante una API REST en lugar de hacerlo en una BD. Para ello creamos una nueva clase RESTPeliculaDAO, que deberá ofrecer las mismas operaciones que la clase anterior. ¿Qué cambios tendremos que hacer en nuestro código para que nuestra aplicación pase a almacenar los datos usando una API REST? (Imaginemos que estamos accediendo a JDBCPeliculaDAO desde 20 puntos distintos de nuestra aplicación).

PATRON DAO:

Prácticamente todas las aplicaciones de hoy en día, requiere acceso al menos a una fuente de datos, dichas fuentes son por lo general base de datos relacionales, por lo que muchas veces no tenemos problema en acceder a los datos, sin embargo, hay ocasiones en las que necesitamos tener más de una fuente de datos o la fuente de datos que tenemos puede variar, lo que nos obligaría a refactorizar gran parte del código. Para esto, tenemos el patrón Arquitectónico Data Access Object (DAO), el cual permite separar la lógica de acceso a datos de los Bussines Objects u Objetos de negocios, de tal forma que el DAO encapsula toda la lógica de acceso de datos al resto de la aplicación.

Dado lo anterior, el patrón DAO propone separar por completo la lógica de negocio de la lógica para acceder a los datos, de esta forma, el DAO proporcionará los métodos necesarios para insertar, actualizar, borrar y consultar la información; por otra parte, la capa de negocio solo se preocupa por lógica de negocio y utiliza el DAO para interactuar con la fuente de datos.



Los compones que conforman el patrón son:

- Client: representa un objeto con la lógica de negocio.
- DataAccessObject: representa una capa de acceso a datos que oculta la fuente y los detalles técnicos para recuperar los datos.
- TransferObject: este es un objeto plano que implementa el patrón Data Transfer Object (DTO), el cual sirve para transmitir la información entre el DAO y el Business Service.
- DataSource: representa de forma abstracta la fuente de datos, la cual puede ser una base de datos, Webservices, LDAP, archivos de texto, etc.
- InterfazDAO es una interfaz java que especifica un contrato mediante operaciones a ser cumplidas por DataAccessObject.

SOLUCION

En una segunda versión como solución a la problemática de nuestra aplicación tenemos definida una interfaz **IPeliculaDAO** con los mismos métodos que comentamos anteriormente.

```
Public interfaz IPeliculaDAO {

        public void addPelicula(PeliculaTO p);

        public void delPelicula(int idPelicula);

        public List<PeliculaTO> getAllPeliculas();

}
```

Tendremos también la clase JDBCPeliculaDAO que en este caso implementa la interfaz IPeliculaDAO. En este caso el acceso a los datos desde el código de nuestra aplicación se hace de la siguiente forma:

```
IPeliculaDAO peliculadao = GestorDAO.getPeliculaDAO();

peliculadao.addPelicula(pelicula);
```

El GestorDAO ahora será como se muestra a continuación:

```
public static IPeliculaDAO getPeliculaDAO() {
      return new JDBCPeliculaDAO();
}
```

¿Qué cambios tendremos que realizar en este segundo caso para pasar a utilizar una BD? Por lo tanto, ¿qué versión consideras más adecuada?

INTERFAZ COLLECTION

Vamos a proceder a describir de manera detallada todas las interfaces y clases que consideramos más relevantes del paquete java.util. Para ello comenzaremos por la raíz de todas ellas que es la interface Collection.

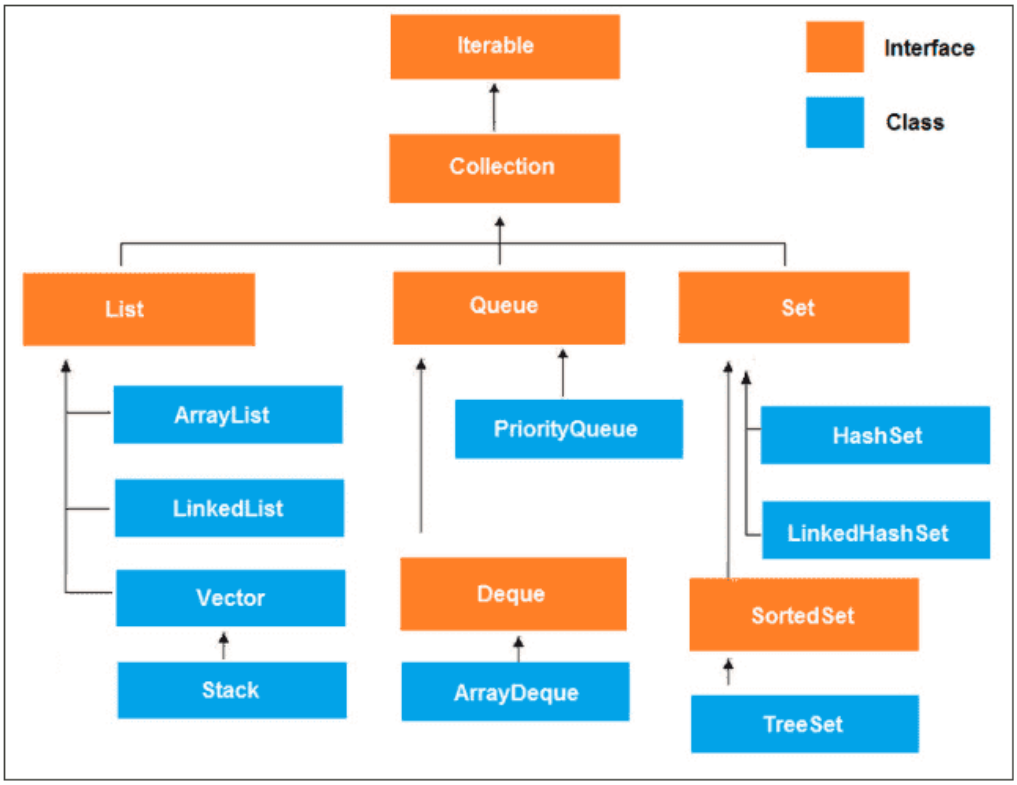
Esta interfaz es “la raíz” de todas las interfaces y clases relacionadas con colecciones de elementos. Algunas colecciones pueden admitir duplicados de elementos dentro de ellas, mientras que otras no admiten duplicados. Otras colecciones pueden tener los elementos ordenados, mientras que en otras no existe orden definido entre sus elementos. Java no define ninguna implementación de esta.

Una colección es de manera genérica un grupo de objetos llamados elementos. Esta interfaz por tanto será usada para pasar colecciones de elementos o manipularlos de la manera más general deseada.

En resumen, la idea es la siguiente: cuando queremos trabajar con un conjunto de elementos, necesitamos un almacén donde poder guardarlos.

En Java, se emplea la interface genérica Collection para este propósito. Gracias a esta interface, podemos almacenar cualquier tipo de objeto y podemos usar una serie de métodos comunes, como pueden ser: añadir, eliminar, obtener el tamaño de la colección… ya que se trata de métodos definidos por la interface que obligatoriamente han de implementar las subinterfaces o clases que hereden de ella.

Collection se ramifica en subinterfaces como Set, List y otras. Un tipo de colecciones de objetos en Java son los Maps. A pesar de que estas estructuras denominadas “mapas” o “mapeos” son colecciones de datos, en el api de Java no derivan de la interface Collection.



Como cualquier interface en Java, Collection nos obliga a implementar varios métodos de los que queremos destacar los siguientes:

- Boolean add (Object o)
- Boolean remove (Object o)
- Int size()

El primer método añade el elemento de la clase X a la colección, el segundo eliminaría el objeto o de la colección y el tercero nos devolvería el tamaño de la colección. Queda claro por tanto que la interface Collection sirve para trabajar con colecciones de objetos, no de tipos primitivos. Hay más métodos que podemos observar consultando la documentación, pero por simplicidad hemos querido destacar tan solo estos.

INTERFAZ SERIALIZACIÓN

Si queremos enviar un objeto a través de un flujo de datos, deberemos convertirlo en una serie de bytes. Esto es lo que se conoce como serialización de objetos, que nos permitirá leer y escribir objetos directamente en un medio de almacenamiento.

Los objetos que escribamos en dicho flujo deben tener la capacidad de ser serializables. Serán serializables aquellos objetos que implementan la interfaz Serializable. Cuando queramos hacer que una clase definida por nosotros sea serializable deberemos implementar dicho interfaz, que no define ninguna función, sólo se utiliza para identificar las clases que son serializables. Para

que nuestra clase pueda ser serializable, todas sus propiedades deberán ser de tipos de datos básicos o bien objetos que también sean serializables.

Un uso común de la serialización se realiza en los Transfer Objetcs. Este tipo de objetos deben ser serializables para así poderse intercambiar entre todas las capas de la aplicación, aunque se encuentren en máquinas diferentes.

```
public class Punto2D implements Serializable {
	private int x;
	private int y;
	public int getX() {
	return x;
	}
	public void setX(int x) {
		this.x = x;
	}

	public int getY() {
		return y;
	}
	public void setY(int y) {
		this.y = y;
	}
}
```

Podríamos enviarlo a través de un flujo, independientemente de su destino, de la siguiente forma:

```
Punto2D p = crearPunto();
FileOutputStream fos = new FileOutputStream(FICHERO_DATOS);
ObjectOutputStream oos = new ObjectOutputStream(fos);
oos.writeObject(p);
oos.close();
```

En este caso hemos utilizado como canal de datos un flujo con destino a un fichero, pero se podría haber utilizado cualquier otro tipo de canal (por ejemplo para enviar un objeto Java desde un servidor web hasta una máquina cliente). En aplicaciones distribuidas los objetos serializables nos permitirán mover estructuras de datos entre diferentes máquinas sin que el desarrollador tenga que preocuparse de la codificación y transmisión de los datos.

Cuando una clase implemente la interfaz Serializable veremos que las IDE nos da un warning si no añadimos un campo serialVersionUID. Este es un código numérico que se utiliza para asegurarnos de que al recuperar un objeto serializado éste se asocie a la misma clase con la que se creó. Así evitamos el problema que puede surgir al tener dos clases que puedan tener el mismo nombre, pero que no sean iguales (podría darse el caso que una de ellas esté en una máquina

cliente, y la otra en el servidor). Si no tuviésemos ningún código para identificarlas, se podría intentar recuperar un objeto en una clase incorrecta.

La IDE nos ofrece dos formas de generar este código pulsando sobre el icono del warning: con un valor por defecto, o con un valor generado automáticamente. Será recomendable utiliar esta segunda forma, que nos asegura que dos clases distintas tendrán códigos distintos.