7.E. Interfaces.

Sitio: Formación Profesional a Distancia
Curso: Programación
Libro: 7.E. Interfaces.
Imprimido por: Iván Jiménez Utiel
Día: lunes, 10 de febrero de 2020, 15:52

Tabla de contenidos

1. Interfaces.

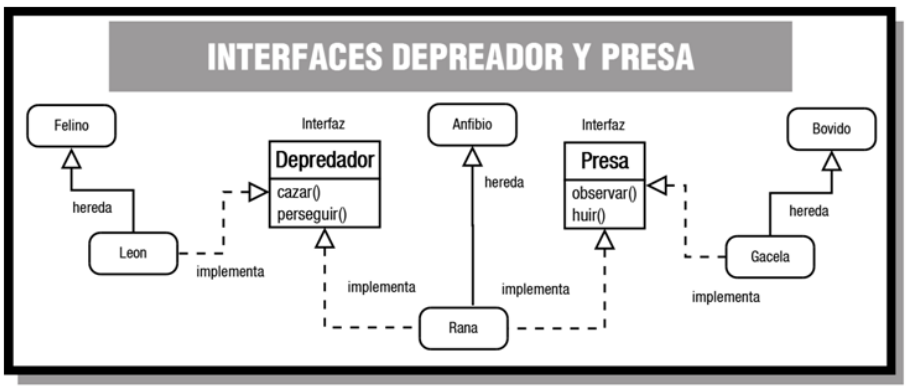
Has visto cómo la **herencia** permite definir **especializaciones** (o **extensiones**) de una **clase base** que ya existe sin tener que volver a repetir de todo el código de ésta. Este mecanismo da la oportunidad de que la nueva **clase especializada** (o **extendida**) disponga de toda la **interfaz** que tiene su clase base.

También has estudiado cómo los **métodos abstractos** permiten establecer una **interfaz** para marcar las **líneas generales de un comportamiento común de superclase** que deberían compartir de todas las **subclases**.

Si llevamos al límite esta idea de **interfaz**, podrías llegar a tener una **clase abstracta** donde todos sus métodos fueran abstractos. De este modo estarías dando únicamente el **marco de comportamiento**, sin ningún método implementado, de las posibles **subclases** que heredarán de esa **clase abstracta**. La idea de una **interfaz** (o **interface**) es precisamente ésa: **disponer de un mecanismo que permita especificar cuál debe ser el comportamiento que deben tener todos los objetos que formen parte de una determinada clasificación** (no necesariamente jerárquica).

Una **interfaz** consiste principalmente en una lista de declaraciones de métodos sin implementar, que caracterizan un determinado comportamiento. Si se desea que una clase tenga ese comportamiento, tendrá que implementar esos métodos establecidos en la **interfaz**. En este caso no se trata de una relación de **herencia** (la clase **A** es una especialización de la clase **B**, o la subclase **A** es del tipo de la superclase **B**), sino más bien una relación "de implementación de comportamientos" (la clase **A** implementa los métodos establecidos en la **interfaz B**, o los comportamientos indicados por **B** son llevados a cabo por **A**; pero no que **A** sea de clase **B**).

Imagina que estás diseñando una aplicación que trabaja con clases que representan distintos tipos de animales. Algunas de las acciones que quieres que lleven a cabo están relacionadas con el hecho de que algunos animales sean **depredadores** (por ejemplo: **observar** una **presa**, **perseguirla**, **comérsela**, etc.) o sean **presas** (**observar**, **huir**, **esconderse**, etc.). Si creas la clase **León**, esta clase podría implementar una interfaz **Depredador**, mientras que otras clases como **Gacela** implementarían las acciones de la interfaz **Presa**. Por otro lado, podrías tener también el caso de la clase **Rana**, que implementaría las acciones de la interfaz **Depredador** (pues es cazador de pequeños insectos), pero también la de **Presa** (pues puede ser cazado y necesita las acciones necesarias para protegerse).



1.1. Concepto de interfaz.

Una **interfaz** en Java consiste esencialmente en una lista de declaraciones de métodos sin implementar, junto con un conjunto de constantes.

Estos métodos sin implementar indican un **comportamiento**, un tipo de conducta, aunque no especifican cómo será ese **comportamiento** (**implementación**), pues eso dependerá de las características específicas de cada clase que decida implementar esa **interfaz**. Podría decirse que una **interfaz** se encarga de establecer qué **comportamientos** hay que tener (qué **métodos**), pero no dice nada de cómo deben llevarse a cabo esos **comportamientos** (**implementación**). Se indica sólo la **forma**, no la **implementación**.

En cierto modo podrías imaginar el concepto de **interfaz** como un **guión** que dice: "éste es el protocolo de comunicación que deben presentar todas las clases que implementen esta interfaz". Se proporciona una lista de **métodos públicos** y, si quieres dotar a tu clase de esa **interfaz**, tendrás que definir todos y cada uno de esos **métodos públicos**.

En conclusión: **una interfaz se encarga de establecer unas líneas generales sobre los comportamientos (métodos) que deberían tener los objetos de toda clase que implemente esa interfaz, es decir, que no indican lo que el objeto es** (de eso se encarga la clase y sus **superclases**)**, sino acciones (capacidades) que el objeto debería ser capaz de realizar**. Es por esto que el nombre de muchas interfaces en Java termina con sufijos del tipo "**-able**", "**-or**", "**-ente**" y cosas del estilo, que significan algo así como **capacidad o habilidad** para hacer o ser receptores de algo (**configurable**, **serializable**, **modificable**, **clonable**, **ejecutable**, **administrador**, **servidor, buscador**, etc.), dando así la idea de que se tiene la capacidad de llevar a cabo el conjunto de acciones especificadas en la **interfaz**.

Imagínate por ejemplo la clase **Coche**, **subclase** de **Vehículo**. Los coches son **vehículos a motor**, lo cual implica una serie de acciones como, por ejemplo, **arrancar el motor** o **detener el motor**. Esa acción no la puedes heredar de **Vehículo**, pues no todos los vehículos tienen porqué ser a motor (piensa por ejemplo en una clase **Bicicleta**), y no puedes heredar de otra clase pues ya heredas de **Vehículo**. Una solución podría ser crear una **interfaz** **Arrancable**, que proporcione los métodos típicos de un **objeto** **a motor** (no necesariamente vehículos). De este modo la clase **Coche** sigue siendo subclase de **Vehículo**, pero también implementaría los comportamientos de la interfaz **Arrancable**, los cuales podrían ser también implementados por otras clases, hereden o no de **Vehículo** (por ejemplo una clase **Motocicleta** o bien una clase **Motosierra**). La clase **Coche** implementará su método **arrancar** de una manera, la clase **Motocicleta** lo hará de otra (aunque bastante parecida) y la clase **Motosierra** de otra forma probablemente muy diferente, pero todos tendrán su propia versión del método **arrancar** como parte de la interfaz **Arrancable**.

Según esta concepción, podrías hacerte la siguiente pregunta: **¿podrá una clase implementar varias interfaces?** La respuesta en este caso sí es afirmativa.

Una clase puede adoptar distintos modelos de comportamiento establecidos en diferentes interfaces. Es decir una clase puede implementar varias interfaces.

Autoevaluación

Una interfaz en Java no puede contener la implementación de un método mientras que una clase abstracta sí. ¿Verdadero o Falso?

- Verdadero
- Falso

1.2. ¿Clase abstracta o interfaz?.

Observando el concepto de **interfaz** que se acaba de proponer, podría caerse en la tentación de pensar que es prácticamente lo mismo que una **clase abstracta** en la que **todos sus métodos sean abstractos**.

Es cierto que en ese sentido existe un gran **parecido formal** entre una **clase abstracta** y una **interfaz**, pudiéndose en ocasiones utilizar indistintamente una u otra para obtener un mismo fin. Pero, a pesar de ese gran parecido, existen algunas diferencias, no sólo formales, sino también conceptuales, muy importantes:

- **Una clase no puede heredar de varias clases**, aunque sean abstractas (**herencia múltiple**). Sin embargo sí puede **implementar una o varias interfaces** y además seguir heredando de una clase.

- **Una interfaz no puede definir métodos** (**no implementa su contenido**), tan solo los declara o enumera.

- **Una interfaz puede hacer que dos clases tengan un mismo comportamiento** independientemente de sus ubicaciones en una determinada jerarquía de clases (no tienen que heredar las dos de una misma superclase, pues no siempre es posible según la naturaleza y propiedades de cada clase).

- **Una interfaz permite establecer un comportamiento de clase sin apenas dar detalles**, pues esos detalles aún no son conocidos (dependerán del modo en que cada clase decida implementar la **interfaz**).

- **Las interfaces tienen su propia jerarquía**, diferente e independiente de la jerarquía de clases.

De todo esto puede deducirse que **una clase abstracta proporciona una interfaz disponible sólo a través de la herencia**. Sólo quien herede de esa **clase abstracta** dispondrá de esa **interfaz**. Si una clase no pertenece a esa misma jerarquía (no hereda de ella) no podrá tener esa **interfaz**. Eso significa que para poder disponer de la **interfaz** podrías:

1. Volver a escribirla para esa jerarquía de clases. Lo cual no parece una buena solución.

2. Hacer que la clase herede de la superclase que proporciona la **interfaz** que te interesa, sacándola de su jerarquía original y convirtiéndola en **clase derivada** de algo de lo que conceptualmente no debería ser una **subclase**. Es decir, estarías forzando una relación "**es un**" cuando en realidad lo más probable es que esa relación no exista. Tampoco parece la mejor forma de resolver el problema.

Sin embargo, **una interfaz sí puede ser implementada por cualquier clase**, permitiendo que clases que no tengan ninguna relación entre sí (pertenecen a distintas jerarquías) puedan compartir un determinado comportamiento (una interfaz) sin tener que forzar una relación de herencia que no existe entre ellas.

A partir de ahora podemos hablar de otra posible relación entre clases: la de **compartir un determinado comportamiento (interfaz)**. Dos clases podrían tener en común un determinado conjunto de

comportamientos sin que necesariamente exista una relación jerárquica entre ellas. Tan solo cuando haya realmente una relación de tipo "**es un**" se producirá **herencia**.

Recomendación

Si sólo vas a proporcionar una lista de **métodos abstractos** (**interfaz**), sin definiciones de métodos ni atributos de objeto, suele ser recomendable definir una **interfaz** antes que **clase** **abstracta**. Es más, cuando vayas a definir una supuesta **clase** **base**, puedes comenzar declarándola como **interfaz** y sólo cuando veas que necesitas definir métodos o variables miembro, puedes entonces convertirla en **clase abstracta** (no instanciable) o incluso en una **clase** **instanciable**.

Autoevaluación

En Java una clase no puede heredar de más de una clase abstracta ni implementar más de una interfaz. ¿Verdadero o Falso?

- Verdadero
- Falso

1.3. Definición de interfaces.

La **declaración de una interfaz** en Java es similar a la declaración de una clase, aunque con algunas variaciones:

- Se utiliza la palabra reservada `interface` en lugar de `class`.
- Puede utilizarse el modificador `public`. Si incluye este modificador la **interfaz** debe tener el mismo nombre que el archivo .java en el que se encuentra (exactamente igual que sucedía con las clases). Si no se indica el modificador `public`, el acceso será por omisión o "**de paquete**" (como sucedía con las clases).
- Todos los **miembros** de la **interfaz** (atributos y métodos) son `public` de manera implícita. No es necesario indicar el modificador `public`, aunque puede hacerse.
- Todos los **atributos** son de tipo `final` y `public` (tampoco es necesario especificarlo), es decir, **constantes** y **públicos**. Hay que darles un **valor inicial**.
- Todos los **métodos** son **abstractos** también de manera implícita (tampoco hay que indicarlo). No tienen cuerpo, tan solo la cabecera.

Como puedes observar, una **interfaz** consiste esencialmente en una lista de **atributos finales** (**constantes**) y **métodos abstractos** (**sin implementar**). Su sintaxis quedaría entonces:

```
[public] interface <NombreInterfaz> {

                [public] [final] <tipo1> <atributo1>= <valor1>;

                [public] [final] <tipo2> <atributo2>= <valor2>;

                ...

                [public] [abstract] <tipo_devuelto1>  <nombreMetodo1>
([lista_parámetros]);

                [public] [abstract] <tipo_devuelto2>  <nombreMetodo2>
([lista_parámetros]);

                ...

}
```

Si te fijas, la declaración de los métodos termina en punto y coma, pues no tienen cuerpo, al igual que sucede con los **métodos abstractos** de las **clases abstractas**.

El ejemplo de la interfaz **Depredador** que hemos visto antes podría quedar entonces así:

```
public interface Depredador {

void localizar (Animal presa);

void cazar (Animal presa);

...

}
```

Serán las clases que implementen esta interfaz (**León**, **Leopardo**, **Cocodrilo**, **Rana**, **Lagarto**, **Hombre**, etc.)

las que definan cada uno de los métodos por dentro.

Autoevaluación

Los métodos de una interfaz en Java tienen que ser obligatoriamente declarados como `public` y `abstract`. Si no se indica así, se producirá un error de compilación. ¿Verdadero o Falso?

- Verdadero
- Falso

Ejercicio resuelto

Crea una interfaz en Java cuyo nombre sea **Imprimible** y que contenga algunos métodos útiles para mostrar el contenido de una clase:

1. Método **devolverContenidoString**, que crea un `String` con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve. El formato será una lista de pares "nombre=valor" de cada atributo separado por comas y la lista completa encerrada entre llaves: "{<nombre_atributo_1>=<valor_atributo_1>, ..., <nombre_atributo_n>=<valor__atributo_n>}".

2. Método **devolverContenidoArrayList**, que crea un `ArrayList` de `String` con una representación de todo el contenido público (o que se decida que deba ser mostrado) del objeto y lo devuelve.

3. Método **devolverContenidoHashtable**, similar al anterior, pero en lugar devolver en un `ArrayList` los valores de los atributos, se devuelve en una `Hashtable` en forma de pares (nombre, valor).

Solución:

Se trata simplemente de declarar la interfaz e incluir en su interior esos tres métodos:

```
public interface  Imprimible {
                String devolverContenidoString ();
                ArrayList devolverContenidoArrayList ();
                HashTable devolverContenidoHashtable ();
}
```

El cómo se implementarán cada uno de esos métodos dependerá exclusivamente de cada clase que decida implementar esta interfaz.

1.4. Implementación de interfaces.

Como ya has visto, todas las clases que implementan una determinada **interfaz** están obligadas a proporcionar una **definición (implementación) de los métodos de esa interfaz**, adoptando el modelo de comportamiento propuesto por ésta.

Dada una **interfaz**, cualquier clase puede especificar dicha **interfaz** mediante el mecanismo denominado **implementación de interfaces**. Para ello se utiliza la palabra reservada `implements`:

```
class NombreClase implements NombreInterfaz {
```

De esta manera, la clase está diciendo algo así como "**la interfaz indica los métodos que debo implementar, pero voy a ser yo (la clase) quien los implemente**".

Es posible indicar varios nombres de **interfaces** separándolos por comas:

```
class NombreClase implements NombreInterfaz1, NombreInterfaz2,... {
```

Cuando una clase implementa una **interfaz**, tiene que redefinir sus métodos nuevamente con **acceso público**. Con otro tipo de acceso se producirá un **error de compilación**. Es decir, que del mismo modo que no se podían restringir permisos de acceso en la **herencia de clases**, tampoco se puede hacer en la **implementación de interfaces**.

Una vez implementada una **interfaz** en una clase, los métodos de esa interfaz tienen exactamente el mismo tratamiento que cualquier otro método, sin ninguna diferencia, pudiendo ser invocados, heredados, redefinidos, etc.

En el ejemplo de los depredadores, al definir la clase **León**, habría que indicar que implementa la **interfaz Depredador**:

```
class Leon implements Depredador {
```

Y en su interior habría que implementar aquellos métodos que contenga la **interfaz**:

```
void localizar (Animal presa) {
```

```
// Implementación del método localizar para un león
```

```
...
```

```
}
```

En el caso de clases que pudieran ser a la vez **Depredador** y **Presa**, tendrían que implementar ambas interfaces, como podría suceder con la clase **Rana**:

```
class Rana implements Depredador, Presa {
```

Y en su interior habría que implementar aquellos métodos que contengan ambas **interfaces**, tanto las de **Depredador** (**localizar**, **cazar**, etc.) como las de **Presa** (**observar**, **huir**, etc.).

Autoevaluación

¿Qué palabra reservada se utiliza en Java para indicar que una clase va a definir los métodos indicados por una interfaz?

- implements.
- uses.
- extends.
- Los métodos indicados por una interfaz no se definen en las clases pues sólo se pueden utilizar desde la propia interfaz.

Ejercicio resuelto

Haz que las clases `Alumno` y `Profesor` implementen la interfaz `Imprimible` que se ha escrito en el ejercicio anterior.

Solución:

La primera opción que se te puede ocurrir es pensar que en ambas clases habrá que indicar que implementan la interfaz Imprimible y por tanto definir los métodos que ésta incluye: devolverContenidoString, devolverContenidoHashtable y devolverContenidoArrayList.

Si las clases Alumno y Profesor no heredaran de la misma clase habría que hacerlo obligatoriamente así, pues no comparten **superclase** y precisamente para eso sirven las **interfaces**: para implementar determinados comportamientos que no pertenecen a la estructura jerárquica de herencia en la que se encuentra una clase (de esta manera, clases que no tienen ninguna relación de herencia podrían compartir interfaz).

Pero en este caso podríamos aprovechar que ambas clases sí son **subclases** de una misma **superclase** (heredan de la misma) y hacer que la interfaz Imprimible sea implementada directamente por la **superclase** (Persona) y de este modo ahorrarnos bastante código. Así no haría falta indicar explícitamente que Alumno y Profesor implementan la interfaz Imprimible, pues lo estarán haciendo de forma implícita al heredar de una clase que ya ha implementado esa interfaz (la clase Persona, que es padre de ambas).

Una vez que los métodos de la **interfaz** estén implementados en la clase Persona, tan solo habrá que redefinir o ampliar los métodos de la **interfaz** para que se adapten a cada **clase** **hija** específica (Alumno o Profesor), ahorrándonos tener que escribir varias veces la parte de código que obtiene los atributos genéricos de la clase Persona.

1. **Clase** Persona**.**

Indicamos que se va a implementar la interfaz Imprimible:

public abstract class Persona **implements Imprimible** {

...

Definimos el método devolverContenidoHashtable a la manera de como debe ser implementado para la clase Persona. Podría quedar, por ejemplo, así:

public Hashtable devolverContenidoHashtable () {

// Creamos la Hashtable que va a ser devuelta

Hashtable contenido= new Hashtable ();

// Añadimos los atributos de la clase

SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");

String stringFecha= formatoFecha.format(this.fechaNacim.getTime());

contenido.put ("nombre", this.nombre);

contenido.put ("apellidos", this.apellidos);

contenido.put ("fechaNacim", stringFecha);

// Devolvemos la Hashtable

return contenido;

}

Del mismo modo, definimos también el método **devolverContenidoArrayList**:

public ArrayList devolverContenidoArrayList () { ... }

Y por último el método **devolverContenidoString**:

public String devolverContenidoString () { ... }

2. **Clase** Alumno**.**

Esta clase hereda de la clase Persona, de manera que heredará los tres métodos anteriores. Tan solo habrá que redefinirlos para que, aprovechando el código ya escrito en la **superclase**, se añada la funcionalidad específica que aporta esta **subclase**.

public class Alumno extends Persona {

...

Como puedes observar no ha sido necesario incluir el implements Imprimible, pues el extends Persona lo lleva implícito dado que Persona ya implementaba ese **interfaz**. Lo que haremos entonces será llamar al método que estamos redefiniendo utilizando la referencia a la **superclase** **super**.

El método devolverContenidoHashtable podría quedar, por ejemplo, así:

public Hashtable devolverContenidoHashtable () {

```
// Llamada al método de la superclase

Hashtable contenido= super.devolverContenidoHashtable();

// Añadimos los atributos específicos de la clase

contenido.put ("grupo", this.salario);

contenido.put ("notaMedia", this.especialidad);

// Devolvemos la Hashtable rellena

return contenido;

}
```

3. **Clase** Profesor**.**

En este caso habría que proceder exactamente de la misma manera que con la clase Alumno: redefiniendo los métodos de la interfaz **Imprimible** para añadir la funcionalidad específica que aporta esta **subclase**.

La solución completa será:

Imprimible.java (Interfaz)

```
/*
 * Interfaz Imprimible
 */
package ejemplointerfazimprimible;
```

```
import java.util.Hashtable;
import java.util.ArrayList;

/**
 *
 * Interfaz Imprimible
 */
public interface Imprimible {
        String devolverContenidoString ();
        ArrayList devolverContenidoArrayList ();
        Hashtable devolverContenidoHashtable ();

}
```

Persona.java

```
/*
 * Clase Persona
 */
package ejemplointerfazimprimible;
import java.util.GregorianCalendar;
import java.util.Hashtable;
import java.util.ArrayList;
import java.util.Enumeration;
import java.text.SimpleDateFormat;

/**
 * Clase Persona
 */
public abstract class Persona implements Imprimible {
    protected String nombre;
    protected String apellidos;
```

```
protected GregorianCalendar fechaNacim;
// Constructores
// -------------
// Constructor
public Persona (String nombre, String apellidos, GregorianCalendar fechaNacim) {
    this.nombre= nombre;
    this.apellidos= apellidos;
    this.fechaNacim= (GregorianCalendar) fechaNacim.clone();
}

// Métodos get
// -------------

// Método getNombre
protected String getNombre (){
    return nombre;
}

// Método getApellidos
protected String getApellidos (){
    return apellidos;
}

// Método getFechaNacim
protected GregorianCalendar getFechaNacim (){
    return this.fechaNacim;
}

// Métodos set
// -------------
```

```
// Método setNombre
protected void setNombre (String nombre){
    this.nombre= nombre;
}

// Método setApellidos
protected void setApellidos (String apellidos){
    this.apellidos= apellidos;
}

// Método setFechaNacim
protected void setFechaNacim (GregorianCalendar fechaNacim){
    this.fechaNacim= fechaNacim;
}

// Implementación de los métodos de la interfaz Imprimible
// ------------------------------------------------------

// Método devolverContenidoHashtable
public Hashtable devolverContenidoHashtable () {
    // Creamos la Hashtable que va a ser devuelta
    Hashtable contenido= new Hashtable ();

    // Añadimos los atributos específicos
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
    String stringFecha= formatoFecha.format(this.fechaNacim.getTime());
    contenido.put ("nombre", this.nombre);
    contenido.put ("apellidos", this.apellidos);
    contenido.put ("fechaNacim", stringFecha);

    // Devolvemos la Hashtable
```

```
    return contenido;
}

// Método devolverContenidoArrayList
public ArrayList devolverContenidoArrayList () {
    ArrayList contenido= new ArrayList ();
    SimpleDateFormat formatoFecha = new SimpleDateFormat("dd/MM/yyyy");
    String stringFecha= formatoFecha.format(this.fechaNacim.getTime());

    contenido.add(this.nombre);
    contenido.add (this.apellidos);
    contenido.add(stringFecha);

    return contenido;
}


// Método devolverContenidoString
public String devolverContenidoString () {
    String contenido= Persona.HashtableToString(this.devolverContenidoHashtable());
    return contenido;
}

// Métodos estáticos (herramientas)
// --------------------------------

// Método HashtableToString
protected static String HashtableToString (Hashtable tabla) {
    String contenido;
    String clave;
    Enumeration claves= tabla.keys();
```

```
            contenido= "{";
            if (claves.hasMoreElements()) {
               clave= claves.nextElement().toString();
               contenido= contenido + clave + "=" + tabla.get(clave).toString();
            }
            while (claves.hasMoreElements()) {
                  clave= claves.nextElement().toString();
                  contenido += ",";
                  contenido= contenido.concat (clave) ;
                  contenido= contenido.concat ("=" + tabla.get(clave));
            }
            contenido= contenido + "}";

            return contenido;
         }
}
```

Alumno.java

```
/*
 * Clase Alumno.
 */
package ejemplointerfazimprimible;
import java.util.*;
import java.text.*;

/**
 *
 * Clase Alumno
 */
public class Alumno extends Persona  {
```

```
    protected String grupo;
    protected double notaMedia;



    // Constructor
    // -----------
    public Alumno (String nombre, String apellidos, GregorianCalendar fechaNacim, String grupo, double
notaMedia) {
        super (nombre, apellidos, fechaNacim);
        this.grupo= grupo;
        this.notaMedia= notaMedia;
    }

    // Métodos get
    // ------------

    // Método getGrupo
    public String getGrupo (){
        return grupo;
    }

    // Método getNotaMedia
    public double getNotaMedia (){
        return notaMedia;
    }

    // Métodos set
    // -----------

    // Método setGrupo
    public void setGrupo (String grupo){
```

```
    this.grupo= grupo;
}

// Método setNotaMedia
public void setNotaMedia (double notaMedia){
    this.notaMedia= notaMedia;
}

// Redefinición de los métodos de la interfaz Imprimible
// ------------------------------------------------------

// Método devolverContenidoHashtable
@Override
public Hashtable devolverContenidoHashtable () {
    // Llamada al método de la superclase
    Hashtable contenido= super.devolverContenidoHashtable();
    // Añadimos los atributos específicos
    contenido.put ("grupo", this.grupo);
    contenido.put ("notaMedia", this.notaMedia);
    // Devolvemos la Hashtable rellena
    return contenido;
}

// Método devolverContenidoArray
@Override
public ArrayList devolverContenidoArrayList () {
    // Llamada al método de la superclase
    ArrayList contenido= super.devolverContenidoArrayList ();
    // Añadimos los atributos específicos
    contenido.add(this.grupo);
    contenido.add (this.notaMedia);
```

```
        // Devolvemos el ArrayList relleno
        return contenido;
    }

    // Método devolverContenidoString
    @Override
    public String devolverContenidoString () {
        // Aprovechamos el método estático para transformar una Hashtable en String
        String contenido= Persona.HashtableToString(this.devolverContenidoHashtable());
        // Devolvemos el String creado.
        return contenido;
    }
}
```

Profesor.java

```
/*
 * Clase Profesor
 */
package ejemplointerfazimprimible;

/**

 */
import java.util.*;
import java.text.*;

/**
 *
 * Clase Profesor
 */
public class Profesor extends Persona {
```

```
    String especialidad;
    double salario;

// Constructor
// -----------

public Profesor (String nombre, String apellidos, GregorianCalendar fechaNacim, String especialidad,
double salario) {
    super (nombre, apellidos, fechaNacim);
    this.especialidad= especialidad;
    this.salario= salario;
}

// Métodos get
// -----------

// Método getEspecialidad
public String getEspecialidad (){
    return especialidad;
}

// Método getSalario
public double getSalario (){
    return salario;
}

// Métodos set
// -----------

// Método setSalario
public void setSalario (double salario){
```

```
    this.salario= salario;
}

// Método setESpecialidad
public void setESpecialidad (String especialidad){
    this.especialidad= especialidad;
}



// Redefinición de los métodos de la interfaz Imprimible
// ------------------------------------------------------

// Método devolverContenidoHashtable
@Override
public Hashtable devolverContenidoHashtable () {
    // Llamada al método de la superclase
    Hashtable contenido= super.devolverContenidoHashtable();
    // Añadimos los atributos específicos
    contenido.put ("salario", this.salario);
    contenido.put ("especialidad", this.especialidad);
    // Devolvemos la Hashtable rellena
    return contenido;
}

// Método devolverContenidoArrayList
@Override
public ArrayList devolverContenidoArrayList () {
    // Llamada al método de la superclase
    ArrayList contenido= super.devolverContenidoArrayList ();
    // Añadimos los atributos específicos
    contenido.add(this.salario);
```

```
            contenido.add (this.especialidad);
            // Devolvemos el ArrayList relleno
            return contenido;
        }

        // Método devolverContenidoString
        @Override
        public String devolverContenidoString () {
            // Aprovechamos el método estático para transformar una Hashtable en String
            String contenido= Persona.HashtableToString(this.devolverContenidoHashtable());
            // Devolvemos el String creado.
            return contenido;
        }
}
```

EjemploInterfazImprimible.java

```
/*
 * Ejemplo de utilización de clases que implementan la interfaz Imprimible.
 */
package ejemplointerfazimprimible;

import java.util.GregorianCalendar;
import java.util.ArrayList;
import java.util.Hashtable;

/**
 *
 * Ejemplo de utilización de clases que implementan la interfaz Imprimible
 */
public class EjemploInterfazImprimible {
```

```
    /**
     * Clase principal
     */
    public static void main(String[] args) {
        String stringContenidoAlumno, stringContenidoProfesor;
        Hashtable hashtableContenidoAlumno, hashtableContenidoProfesor;
        ArrayList listaContenidoAlumno, listaContenidoProfesor;

        // Creación de objetos alumno y profesor
        Alumno al1= new Alumno ("Juan", "Torres Mula", new GregorianCalendar (1990, 10, 6), "1DAM-B",
7.5);
        Profesor prof1= new Profesor  ("Antonio", "Campos Pin", new GregorianCalendar (1970, 8, 15),
"Informática", 2000);

        // Obtención del contenido del objeto alumno a través de los métodos del interfaz Imprimible
        stringContenidoAlumno= al1.devolverContenidoString();
        hashtableContenidoAlumno= al1.devolverContenidoHashtable();
        listaContenidoAlumno= al1.devolverContenidoArrayList();

        // Obtención del contenido del objeto profesor a través de los métodos del interfaz Imprimible
        stringContenidoProfesor= prof1.devolverContenidoString();
        hashtableContenidoProfesor= prof1.devolverContenidoHashtable();
        listaContenidoProfesor= prof1.devolverContenidoArrayList();

        // Impresión en pantalla del contenido del objeto alumno a través de las estructuras obtenidas
        System.out.printf ("Contenido del objeto alumno: %s\n", stringContenidoAlumno);

        // Impresión en pantalla del contenido del objeto alumno a través de las estructuras obtenidas
        System.out.printf ("Contenido del objeto profesor: %s\n", stringContenidoProfesor);
```

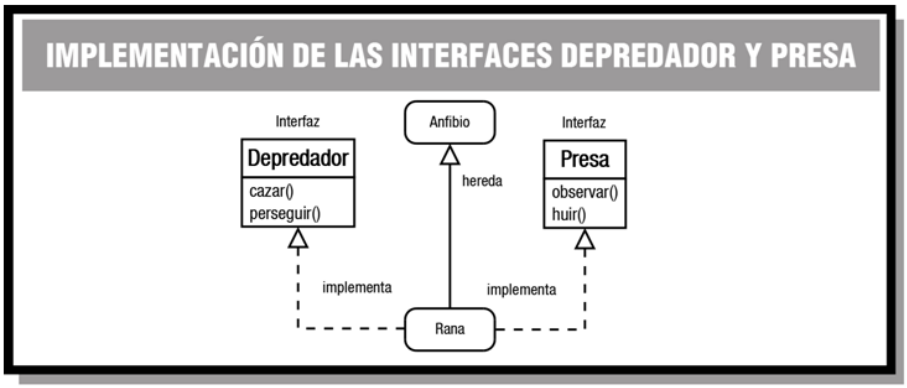
```
    }
}
```

1.5. Simulación de la herencia múltiple mediante el uso de interfaces.

Una **interfaz** no tiene **espacio de almacenamiento** asociado (no se van a declarar objetos de un tipo de interfaz), es decir, no tiene **implementación**.

En algunas ocasiones es posible que interese representar la situación de que "una clase **X** es de tipo **A**, de tipo **B**, y de tipo **C**", siendo **A**, **B**, **C** **clases disjuntas** (no heredan unas de otras). Hemos visto que sería un caso de **herencia múltiple** que Java no permite.

Para poder simular algo así, podrías definir tres **interfaces A**, **B**, **C** que indiquen los comportamientos (métodos) que se deberían tener según se pertenezca a una supuesta clase **A**, **B**, o **C**, pero sin implementar ningún método concreto ni atributos de objeto (sólo interfaz).



De esta manera la clase **X** podría a la vez:

1. Implementar las interfaces **A**, **B**, **C**, que la dotarían de los comportamientos que deseaba heredar de las clases **A**, **B**, **C**.
2. Heredar de otra clase **Y**, que le proporcionaría determinadas características dentro de su taxonomía o jerarquía de objeto (atributos, métodos implementados y métodos abstractos).

En el ejemplo que hemos visto de las interfaces **Depredador** y **Presa**, tendrías un ejemplo de esto: la clase **Rana**, que es subclase de **Anfibio**, implementa una serie de **comportamientos** propios de un **Depredador** y, a la vez, otros más propios de una **Presa**. Esos **comportamientos** (**métodos**) no forman parte de la **superclase** **Anfibio**, sino de las **interfaces**. Si se decide que la clase **Rana** debe de llevar a cabo algunos otros **comportamientos adicionales**, podrían añadirse a una **nueva** **interfaz** y la clase **Rana implementaría** una tercera **interfaz**.

De este modo, con el mecanismo "**una herencia pero varias interfaces**", podrían conseguirse resultados similares a los obtenidos con la **herencia** **múltiple**.

Ahora bien, del mismo modo que sucedía con la **herencia múltiple**, puede darse el problema de la **colisión de nombres** al implementar dos **interfaces** que tengan un **método** **con el mismo identificador**. En tal caso puede suceder lo siguiente:

- Si los dos métodos tienen **diferentes parámetros** no habrá problema aunque tengan el mismo nombre pues se realiza una **sobrecarga** de métodos.
- Si los dos métodos tienen **un valor de retorno de un tipo diferente**, se producirá un **error de compilación** (al igual que sucede en la sobrecarga cuando la única diferencia entre dos métodos es ésa).

Si los dos métodos son **exactamente iguales en identificador, parámetros y tipo devuelto**, entonces solamente se podrá **implementar uno de los dos métodos**. En realidad se trata de un solo método pues ambos

tienen la misma interfaz (mismo identificador, mismos parámetros y mismo tipo devuelto).

Recomendación

La utilización de nombres idénticos en diferentes **interfaces** que pueden ser implementadas a la vez por una misma clase puede causar, además del problema de la **colisión de nombres**, dificultades de **legibilidad** en el código, pudiendo dar lugar a confusiones. Si es posible intenta evitar que se produzcan este tipo de situaciones.

Autoevaluación

Dada una clase Java que implementa dos interfaces diferentes que contienen un método con el mismo nombre, indicar cuál de las siguientes afirmaciones es correcta.

- Si los dos métodos tienen un valor de retorno de un tipo diferente, se producirá un error de compilación.
- Si los dos métodos tienen un valor de retorno de un tipo diferente, se implementarán dos métodos.
- Si los dos métodos son exactamente iguales en identificador, parámetros y tipo devuelto, se producirá un error de compilación.
- Si los dos métodos tienen diferentes parámetros se producirá un error de compilación.

Ejercicio resuelto

Localiza en la API de Java algún ejemplo de clase que implemente varias interfaces diferentes (puedes consultar la documentación de referencia de la API de Java).

Solución:

Existen una gran cantidad de clases en la API de Java que implementan **múltiples interfaces**. Aquí tienes un par de ejemplos:

- La clase javax.swing.JFrame, que implementa las **interfaces** WindowConstants, Accessible y RootPaneContainer:
  - public class JFrame extends Frame
  - implements WindowConstants**,** Accessible**,** RootPaneContainer

- La clase java.awt.component, que implementa las **interfaces** ImageObserver, MenuContainer y Serializable:
  - public abstract class Component extends Object
  - implements ImageObserver, MenuContainer, Serializable

1.6. Herencia de interfaces.

Las **interfaces**, al igual que las **clases**, también permiten la **herencia**. Para indicar que una **interfaz** hereda de otra se indica nuevamente con la palabra reservada `extends`. Pero en este caso sí se permite la **herencia múltiple** **de interfaces**. Si se hereda de más de una **interfaz** se indica con la lista de **interfaces** separadas por comas.

Por ejemplo, dadas las interfaces **InterfazUno** e **InterfazDos**:

```
public interface InterfazUno {
```

```
                // Métodos y constantes de la interfaz Uno
```

```
}
```

```
public interface InterfazDos {
```

```
                // Métodos y constantes de la interfaz Dos
```

```
}
```

Podría definirse una nueva **interfaz** que heredara de ambas:

```
public interface InterfazCompleja extends InterfazUno, InterfazDos {
```

```
                // Métodos y constantes de la interfaz compleja
```

```
}
```

Autoevaluación

En Java no está permitida la herencia múltiple ni para clases ni para interfaces. ¿Verdadero o Falso?

- Verdadero
- Falso

Ejercicio resuelto

Localiza en la API de Java algún ejemplo de interfaz que herede de una o varias interfaces (puedes consultar la documentación de referencia de la API de Java).

Solución:

Existen una gran cantidad de **interfaces** en la API de Java que heredan de otras **interfaces**. Aquí tienes un par de ejemplos:

- La interfaz java.awt.event.ActionListener, que hereda de java.util.EventListener:

§ public interface ActionListener extends EventListener

- La interfaz org.omg.CORBA.Policy, que hereda de org.omg.CORBA.PolicyOperations, org.omg.CORBA.Object y org.omg.CORBOA.portable.IDLEntity:

§ public interface Policy extends PolicyOperations, Object, IDLEntity.