UNIDAD DIDÁCTICA 4
PROGRAMACIÓN ORIENTADA A OBJETOS 2

INTERFACES

De forma genérica una interfaz se define como un dispositivo o sistema utilizado por entidades inconexas para interactuar, por ejemplo, un control remoto, el idioma inglés, etc. Una interfaz Java es un dispositivo que permite interactuar a objetos **"no relacionados entre si".** Definen un conjunto de mensajes que se puede aplicar a muchas clases de objetos, a los que cada una de ellas debe responder de forma adecuada. (Hay veces que se le llama protocolo).

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, es decir, no por herencia).

> NOTA: JDK 8 agregó una función a **interface** que hizo un cambio significativo en sus capacidades. Antes de JDK 8, una interfaz no podía definir ninguna implementación de ningún tipo. Por lo tanto, antes de JDK 8, una interfaz podría definir solo el qué, pero no el cómo. **JDK 8** cambió esto siendo posible agregar una implementación predeterminada a un método de interfaz. Además, ahora se admiten los **métodos de interfaz estática** y, a partir de JDK 9, una interfaz también puede incluir **métodos privados, métodos default..**. Por lo tanto, ahora es posible que la interfaz especifique algún comportamiento.
> Se verá todo esto en la unidad 5 y en la actual, se estudiarán las interfaces basándonos en el antiguo concepto de las mismas, ya que este perdura hoy día.

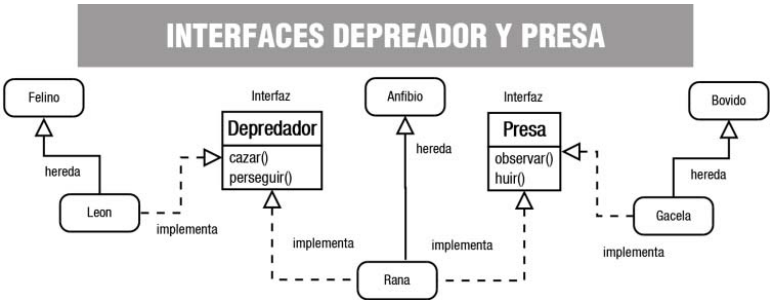
1. CONCEPTO DE INTERFAZ

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**.

U**na interfaz especifica qué se debe hacer, pero no cómo hacerlo**. Una vez que se define una interfaz, cualquier cantidad de clases puede implementarla. Además, una clase puede implementar cualquier cantidad de interfaces.

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 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 **guion** 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.

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 (a partir de JDK 8 sí, como hemos dicho en la nota de inicio).

- ✓ **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:

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

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 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.

3. DEFINIR UNA INTERFAZ

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.
- Desde la versión Java 8, se pueden incluir métodos los llamados “Default methods” que permiten utilizar de una mejor forma las interfaces para la herencia múltiple (prohibida en java). Estos métodos sí se pueden implementar en la interfaz, es decir, tendrán cuerpo y este será común para todas aquellas clases que implemente esa interfaz.

Tienen dos propósitos:

1. ***No modificar las clases que usen esa interface***.
2. ***Simular una "pseudo herencia múltiple"***, ya que java no dispone de herencia múltiple como tal.

El punto 1 (aunque puede sonar chapucero) tiene bastante sentido y más hoy en día en que es muy importante la rápida respuesta al cambio, ya que suponer un caso en el que hayamos desarrollado una interface que la implementan 20 clases y necesitemos a última hora, añadir una nueva funcionalidad que deben de tener todas las clases que implementan nuestra interface. Pues en este caso con implementar un método "default" en la interface nos valdría para solucionar el problema y adaptarnos rápidamente al cambio. Esta sería una solución rápida, aunque si fuésemos muy puristas, con más tiempo deberíamos reescribir esos métodos en las clases que implementan la interface.

Si te fijas, la declaración de los métodos “no por defecto”, 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 IDepredador {
        void localizar (Animal presa);
        void cazar (Animal presa);
}
```

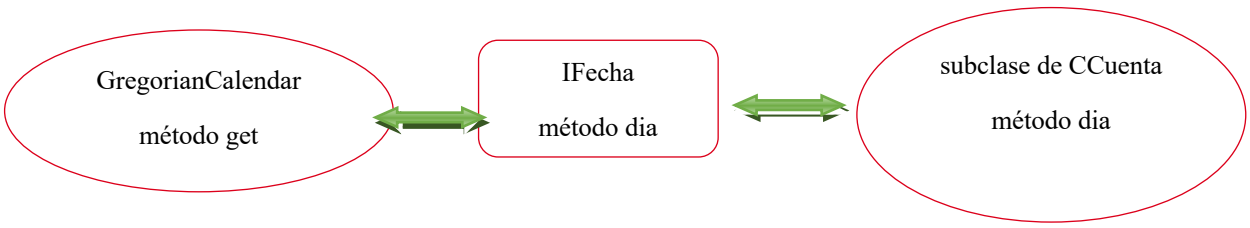
Serán las clases que implementen esta interfaz (León, Leopardo, Cocodrilo, Rana, Lagarto, Hombre, etc.) las que definan cada uno de los métodos por dentro.

Ejemplo: la interfaz IFecha (aunque en Java no se gestionan fechas de esta manera, usaremos la forma antigua para no tener que explicar ahora nada más).

Si continuamos con el ejemplo de los bancos, recordaremos que, en un momento, definíamos CCuenta como abstracta porque contenía métodos abstractos (comisiones e intereses) que no tenían sentido definir en la superclase (debían ser luego particularizados para cada una de las subclases).

Las interfaces, al igual que las clases y métodos abstractos, proporcionan plantillas de comportamiento que se espera, sean implementadas por otras clases.

En el ejemplo, vamos a definir una interfaz IFecha que va a ser utilizada para que dos clases (GregorianCalendar y una subclase de CCuenta) interactúen entre sí.



Aunque la clase GregorianCalendar ya es antigua y ha sido reemplazada por LocalDate, seguiremos usándola en este ejemplo a nivel académico (no se trabaja así con las fechas, se verá en el tema siguiente). La clase GregorianCalendar es de Java. En nuestro ejemplo notificará el día a los objetos derivados de CCuenta cuando intenten ejecutar sus métodos comisiones e intereses. Para ello, proporciona el método **get** que devuelve el tipo de dato (días, mes, etc.) solicitado:

```
public class GregorianCalendar extends Calendar {
        //…
        public final int get (int tipo_de_dato) {…}
}
```

Según este planteamiento, cualquier objeto derivado de CCuenta que quiera utilizar un objeto GregorianCalendar debe implementar el método **dia ()** proporcionado por la interfaz IFecha. Es el medio usado por el objeto GregorianCalendar para notificar al objeto derivado de CCuenta el día actual.

El código de IFecha podría ser:

```
import java.util.*;
public interface IFecha {
        public final static int DIA_DEL_MES = Calendar.DAY_OF_MONTH;
        public final static int MES_DEL_AÑO = Calendar.MONTH;
        public final static int AÑO = Calendar.YEAR;

        public abstract int día();
        public abstract int mes();
        public abstract int año();
}
```

- Una interfaz solo declara métodos y constantes.
- No define los métodos.
- Las declaraciones terminan en punto y coma.
- Todos los métodos declarados en una interfaz son implícitamente públicos y abstractos (public y abstract).
- **Todas las constantes son públicas, finales y estáticas.**
- El uso de los modificadores de acceso es solamente por estilo y claridad.
- Cualquier clase puede acceder a las constantes de la interfaz a través del nombre de la misma: IFecha.año;
- En cambio, una clase que implemente la interfaz puede tratar las constantes como si las hubiese heredado, es decir, accediendo directamente a su nombre.

4. UTILIZAR UNA INTERFAZ

Para utilizar una interfaz hay que añadir el nombre de la misma precedido por la palabra **implements** a la definición de la clase. La palabra implements sigue a la palabra **extends** si esta existe.

Si seguimos con el ejemplo, una subclase como CCuentaAhorro que utilice la interfaz IFecha, debe definirse así:

```
import java.util.*;
public class CCuentaAhorro extends CCuenta implements IFecha{
	private double cuotaMantenimiento;
	public CCuentaAhorro() {}
	public CCuentaAhorro(String nom, String cue, double sal, double tipo, double mant){
		super(nom, cue, sal, tipo); // invoca al constructor CCuenta
		asignarCuotaManten(mant); // inicia cuotaMantenimiento
	}
	public void asignarCuotaManten(double cantidad){
		if (cantidad >= 0){
			cuotaMantenimiento = cantidad;
		}
	}
	public double obtenerCuotaManten(){
		return cuotaMantenimiento;
	}
	public void comisiones(){
		// Se aplican mensualmente por el mantenimiento de la cuenta
		if (día() == 1) {
			reintegro(cuotaMantenimiento);
		}
	}
	public double intereses(){
		double interesesProducidos = 0.0;
		if (día() != 1)
		return 0.0;
		// Acumular los intereses por mes sólo los días 1 de cada mes
		interesesProducidos = estado() * obtenerTipoDeInteres() / 1200.0;
		ingreso(interesesProducidos);
		// Devolver el interés mensual por si fuera necesario
		return interesesProducidos;
	}
	// Implementación de los métodos de la interfaz IFecha
	public int día(){
		GregorianCalendar fechaActual = new GregorianCalendar();
		return fechaActual.get(DIA_DEL_MES);
	}
	public int mes() { return 0; } // no se necesita pero hay que implementarlo
	public int año() { return 0; } // no se necesita pero hay que implementarlo
```

```
}
```

OJO: Como esta interfaz solo aporta declaraciones de métodos abstractos, es obligatorio definir TODOS los métodos en cada una de las clases que utilice la interfaz. No podemos elegir y definir solo aquellos métodos que necesitemos. De no hacerlo, Java obliga a que la clase sea abstracta.

Si una clase implementa una interfaz, todas sus subclases heredarán los nuevos métodos que se hayan implementado en la superclase, así como las constantes definidas por la interfaz. Por ejemplo, modifiquemos la clase CCuentaCorriente para que utilice también la interfaz IFecha:

```
import java.util.*;
public class CCuentaCorriente extends CCuenta implements IFecha{
        // Atributos
        private int transacciones;
        private double importePorTrans;
        private int transExentas;

        // Métodos
        public CCuentaCorriente() {} // constructor sin parámetros

        public CCuentaCorriente(String nom, String cue, double sal,
        double tipo, double imptrans, int transex){
                super(nom, cue, sal, tipo);      // invoca al constructor CCuenta
                transacciones = 0;               // inicia transacciones
                asignarImportePorTrans(imptrans); // inicia importePorTrans
                asignarTransExentas(transex);     // inicia transExentas
        }
        public void decrementarTransacciones(){
                transacciones--;
        }
        public void asignarImportePorTrans(double imptrans){
                if (imptrans >= 0){
                        importePorTrans = imptrans;
                }

        }
        public double obtenerImportePorTrans(){
                return importePorTrans;
        }
        public void asignarTransExentas(int transex){
                if (transex >= 0){
                        transExentas = transex;
```

```
		}

	}
	public int obtenerTransExentas(){
		return transExentas;
	}
	public void ingreso(double cantidad){
		super.ingreso(cantidad);
		transacciones++;
	}
	public void reintegro(double cantidad){
		super.reintegro(cantidad);
		transacciones++;
	}

	public void comisiones(){
		// Se aplican mensualmente por el mantenimiento de la cuenta
		int n=0;
		if (día() == 1){
			n = transacciones - transExentas;
			if (n > 0) {
				reintegro(n * importePorTrans);
			}
			transacciones = 0;
		}
	}

	public double intereses(){
		double interesesProducidos = 0.0;
		if (día() != 1)
			return 0.0;
		// Acumular los intereses por mes sólo los días 1 de cada mes
		// Hasta 3000 euros al 0.5%. El resto al interés establecido.
		if (estado() <= 3000)
			interesesProducidos = estado() * 0.5 / 1200.0;
		else
		{
			interesesProducidos = 3000 * 0.5 / 1200.0 +(estado() - 3000) * obtenerTipoDeInterés() / 1200.0;
		}
```

```
                ingreso(interesesProducidos);
                // Este ingreso no debe incrementar las transacciones
                decrementarTransacciones();
                // Devolver el interés mensual por si fuera necesario
                return interesesProducidos;
        }
        // Implementación de los métodos de la interfaz IFecha
        public int día(){
                GregorianCalendar fechaActual = new GregorianCalendar();
                return fechaActual.get(DIA_DEL_MES);
        }
        public int mes() { return 0; } // no se necesita pero hay que implementarlo
        public int año() { return 0; } // no se necesita pero hay que implementarlo
}
```

Como la clase CCuentaCorriente implementa la interfaz IFecha, su subclase CCuentaCorrienteConIn heredará los nuevos métodos y constantes, por lo tanto, no es necesario agregar a la definición de esta clase la palabra implements más el nombre de la interfaz.

```
public class CCuentaCorrienteConIn extends CCuentaCorriente
{
 // Métodos
 public CCuentaCorrienteConIn() {} // constructor sin parámetros
 public CCuentaCorrienteConIn(String nom, String cue, double sal, double tipo, double imptrans, int transex){
        // Invocar al constructor de la superclase
        super(nom, cue, sal, tipo, imptrans, transex);
 }
 public double intereses(){
        if (día() != 1 || estado() < 3000) return 0.0;

        // Acumular interés mensual sólo los días 1 de cada mes
        double interesesProducidos = 0.0;
        interesesProducidos = estado() * obtenerTipoDeInterés() / 1200.0;
        ingreso(interesesProducidos);
        // Este ingreso no debe incrementar las transacciones
        decrementarTransacciones();

        // Devolver el interés mensual por si fuera necesario
        return interesesProducidos;
 }
```

```
}
```

NOTA: Se obtendría el mismo resultado implementando IFecha en la superclase CCuenta y así se tendría para todos. Se ha hecho de esta manera por motivos meramente didácticos, para ver que cada clase podría implementar sus métodos de la interfaz de forma diferente.

Podría volver a preguntarnos ¿En qué se diferencian entonces una interfaz de una clase abstracta? ¿Podríamos escribir entonces IFecha como una clase abstracta para hacer lo mismo?

```
import java.util.*;
public abstract class IFecha
{
        public final static int DIA_DEL_MES = Calendar.DAY_OF_MONTH;
        public final static int MES_DEL_AÑO = Calendar.MONTH;
        public final static int AÑO = Calendar.YEAR;

        public abstract int día();
        public abstract int mes();
        public abstract int año();

        }
```

NOOOOO. Si IFecha es una clase abstracta, todas las subclases de CCuenta, como CCuentaAhorro, que quisieran utilizar su funcionalidad para interactuar con GregorianCalendar tendrían que derivarse de ella, es decir, heredarían de CCuenta y de IFecha, y Java NO permite la herencia múltiple, pero sí permite que una interfaz se derive de múltiples interfaces.

5. UTILIZAR UNA INTERFAZ COMO UN TIPO

Una interfaz es un nuevo tipo de datos, por tanto, el nombre de una interfaz se puede utilizar en cualquier lugar donde pueda aparecer el nombre de cualquier otro tipo de datos (pero no podemos instanciar un objeto de ese tipo), es decir, **se puede declarar una referencia de un tipo de interface, y puede referirse a cualquier objeto que implemente su interfaz.**

Por ejemplo, se puede declarar un array clientes que sea del tipo IFecha y asignar a cada elemento un objeto de algunas de las subclases de CCuenta.

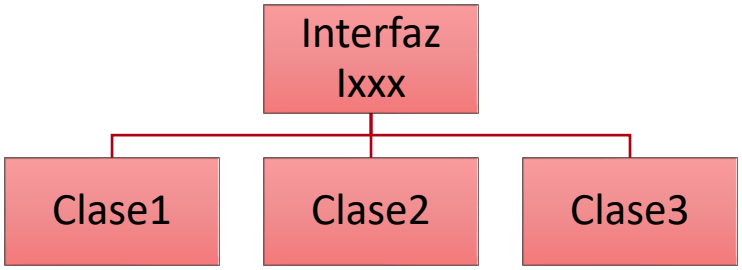
```
IFecha [ ] clientes = new IFecha [3]; //Correcto
IFecha nuevafecha= new IFecha ( );//Incorrecto
CCuentaAhorro cliente0 = new CCuentaAhorro ();

clientes[0]= cliente0;
```

La variable del tipo IFecha espera referenciar un objeto que tenga implementada dicha interfaz, de lo contrario Java mostrará un error de compilación. En el ejemplo, la variable clientes [0] de tipo IFecha, hace referencia a un objeto CCuentaAhorro que implementa dicha interfaz.

Con una referencia a la interfaz solo se tiene acceso a los métodos y constantes declarados en dicha interfaz. Pero es posible convertir implícitamente referencias a objetos que implementan una interfaz y viceversa, en este último caso, de manera explícita.

El siguiente ejemplo, muestra cómo tres clases no relacionadas por herencia, por el hecho de implementar la misma interfaz (Ixxx), nos permite definir un array de objetos de esas clases y aplicar la definición de polimorfismo:

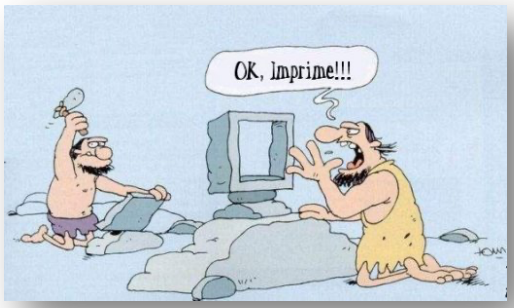



```
public interface Ixxx {
        public abstract void m ();
        public abstract void p ();
}
public class Clase1 implements Ixxx{
        public void m (){
                System.out.println("Método m de Clase1");
        }
        public void p (){};
}
public class Clase2 implements Ixxx{
        public void m (){
                System.out.println("Método m de Clase2");
        }
        public void p (){};
}
public class Clase3 implements Ixxx{
        public void m (){
                System.out.println("Método m de clase3");
        }
```

```
        public void p (){};
}

public class Test {
        public static void main (String [] args){
                Ixxx[] objs = new Ixxx[3];
                objs[0]= new Clase1 ();
                objs[1]= new Clase2 ();
                objs[2]= new Clase3 ();
                for (int i=o; objs.length; i++){
                        objs[i].m();
//Invoca al método m del objeto Clase1, Clase2, Clase3 referenciado por objs[i]
                }
        }
}
```

6. INTERFACES FRENTE A HERENCIA MÚLTIPLE



Aunque las interfaces pueden resolver problemas de herencia múltiple, no es buena alternativa usarlo. Son diferentes:

- ✓ Desde una interfaz, una clase solo hereda constantes.
- ✓ Desde una interfaz, una clase no puede heredar definiciones de métodos.
- ✓ La jerarquía de interfaces es independiente de la jerarquía de clases. De hecho, varias clases pueden implementar la misma interfaz y no pertenecer a la misma jerarquía de clases. Pero cuando se habla de herencia múltiple, todas las clases pertenecen a la misma jerarquía.

NOTA: Evitar la herencia múltiple soluciona el problema.

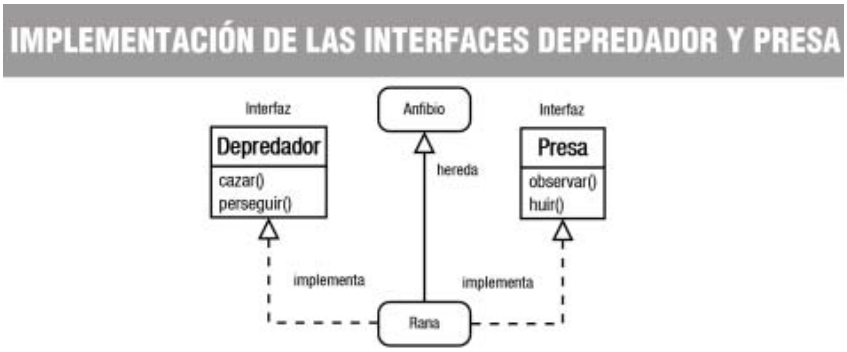
Una **interfaz** no tiene **espacio de almacenamiento** asociado en atributos (no se van a declarar objetos de un tipo de interfaz), es decir, no tiene **implementación**.
En algunas ocasiones es posible que interese representar la situación de que "una clase **X** es de tipo **A**, de tipo **B**, y 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).

7. PARA QUÉ SIRVE UNA INTERFAZ

Una interfaz se utiliza para definir un protocolo de conducta que puede ser implementado por cualquier clase en una jerarquía. La utilidad de esto puede resumirse en:

- Captar similitudes entre clases no relacionadas sin forzar entre ellas una relación artificial.
- Declarar métodos que una o más clases deben implementar en determinadas situaciones.
- Publicar la interfaz de programación de una clase sin descubrir cómo está implementada. Otros desarrolladores recibirían la clase compilada y la interfaz correspondiente.

Se pueden implementar una o más interfaces. Después de la palabra implements, van los nombres de las interfaces, separados por comas. En este caso, puede ocurrir que dos o más interfaces diferentes implementen el mismo método. Si esto ocurre, debemos hacer algo de lo siguiente:

- Si los métodos tienen el mismo protocolo, basta con definir uno en la clase.
- Si los métodos difieren en el número o tipo de parámetros, estamos en un caso de sobrecarga del método. Hay que implementar TODAS las sobrecargas.
- Si los métodos solo difieren en el tipo de valor de retorno, no existe sobrecarga y el compilador produce un error, ya que dos métodos pertenecientes a la misma clase no pueden diferir solo en el tipo de resultado.

Otro ejemplo:

```
public abstract class Figura {

        public abstract double area ( );

}

public interface Dibujable {

        public void dibujar ( );

}

public interface Rotable {

        public void rotar (double grados);

}

public class Circulo extends Figura implements Dibujable, Rotable {

…

}
```

8. MÁS EJEMPLOS DE CLASES ABSTRACTAS E INTERFACES

Os dejo varios ejemplos para lectura, por si alguien se aburre y quiere seguir repasando, aunque no añaden nada nuevo a lo ya visto.

```
importjava.util.*;

public abstract class Instrumento {

   public abstract void tocar();//Método abstracto que se implementa en clases hijas

   public String tipo() {
            return "Instrumento";
   }
   public abstract void afinar();
}

public class Guitarra extends Instrumento {

   public void tocar() {
            System.out.println("Guitarra.tocar()");
   }


   public String tipo() { return "Guitarra"; }
   public void afinar() {}
}

public class Piano extends Instrumento {

   public void tocar() {
            System.out.println("Piano.tocar()");
   }
   public String tipo() { return "Piano"; }
   public void afinar() {}
}

public class Saxofon extends Instrumento {
   public void tocar() {
            System.out.println("Saxofon.tocar()");
   }
   public String tipo() { return "Saxofon"; }
   publicvoid afinar() {}
}

// Un tipo de Guitarra
public class Guzla extends Guitarra {

     public void tocar() {
       System.out.println("Guzla.tocar()");
     }
```

```
    public void afinar() {
      System.out.println("Guzla.afinar()");
    }
}

// Un tipo de Guitarra
public class Ukelele extends Guitarra {
   public void tocar() {
            System.out.println("Ukelele.tocar()");
   }
   public String tipo() { return "Ukelele"; }
}

public class Musica2 {

   // No importa el tipo de Instrumento,
   // seguirá funcionando debido a Polimorfismo:
   static void afinar(Instrumento i) {
            // ...
            i.tocar();
   }

   static void afinarTodo(Instrumento[] e) {

            for(int i = 0; i<e.length; i++)
            afinar(e[i]);
   }


   public static void main(String[] args) {
      // Declarar un ARRAY SIN INSTANCIAS es válido en Clases Abstractas
            Instrumento[] orquesta = new Instrumento[5];
      // Generar una INSTANCIA de una la Clase Abstracta no es válido
      // Instrumento nuevo = new Instrumento(); Error
            int i = 0;
            // Up-casting al asignarse el Arreglo
            orquesta[i++] = new Guitarra();
            orquesta[i++] = new Piano();
            orquesta[i++] = new Saxofon();
            orquesta[i++] = new Guzla();
            orquesta[i++] = new Ukelele();
            afinarTodo(orquesta);
   }
}
```

Algunos detalles de esta definición:

- Nótese que los métodos definidos como *abstract* no contienen ningún tipo de código dentro de ellos, inclusive no declaran ni llaves ({ }).
- Cuando es definido un método o más de un método como *abstract*, es necesario que la Clase como tal sea definida también como *abstract*.

La característica de hacer una Clase/Método *abstract* reside en que no puede ser generada una **instancia** de la misma, este comportamiento se demuestra en el método principal (main):

- Aunque dentro del método sea generado un array de esta Clase abstracta, recuerda que un array es únicamente un contenedor de Objetos, esto permite que sea generado sin ningún error.
- Dentro de comentarios se encuentra la generación de una **instancia** del tipo Instrumento la cual generaría un error al ser compilada la Clase.

Una de las características de las Clases que Heredan de una Clase abstracta, es que estas deben definir los mismos métodos definidos en la Clase Base; en Java existe otro mecanismo que permite llevar acabo diseños donde se parte de una Estructura o Cápsula, estas son las interfaces.

Mismo ejemplo, pero con Interfaces

```
import java.util.*;

public interface Instrumento {
   // Constante al compilar, automáticamente static y final
   int i = 5;
   // Métodos Automáticamente Públicos
   void tocar();
   String tipo();
   void afinar();
 }

public class Guitarra implements Instrumento {
    public void tocar() {
                System.out.println("Guitarra.tocar()");
    }
    public String tipo() { return "Guitarra"; }
    public void afinar() {}
}

public class Piano implements Instrumento {
    public void tocar() {
                System.out.println("Piano.tocar()");
    }
    public String tipo() { return "Piano"; }
    public void afinar() {}
}

public class Saxofon implements Instrumento {
    public void tocar() {
                System.out.println("Saxofon.tocar()");
    }
    public String tipo() { return "Saxofon"; }
    public void afinar() {}
}

// Un tipo de Guitarra
public class Guzla extends Guitarra {
```

```
      public void tocar() {
         System.out.println("Guzla.tocar()");
      }
      public void afinar() {
        System.out.println("Guzla.afinar()");
      }
}

// Un tipo de Guitarra
public class Ukelele extends Guitarra {
   public void tocar() {
        System.out.println("Ukelele.tocar()");
   }
   public String tipo() { return "Ukelele"; }
}

public class Musica3 {

    // No importa el tipo de Instrumento,
    // seguirá funcionando debido a Polimorfismo:
    static void afinar(Instrumento i) {
                // ...
                i.tocar();
    }

       static void afinarTodo(Instrumento[] e) {

                for (int i = 0; i<e.length; i++){
                  afinar(e[i]);
                }
}

    public static void main(String[] args) {

// Declarar un Array SIN INSTANCIAS es válido en Clases Abstractas
                Instrumento[] orquesta = new Instrumento[5];
                int i = 0;
                // Up-casting al asignarse el Arreglo
                orquesta [i++] = new Guitarra();
                orquesta [i++] = new Piano();
                orquesta [i++] = new Saxofon();
                orquesta [i++] = new Guzla();
                orquesta [i++] = new Ukelele();
                afinarTodo(orquesta);
    }
}
```

Algunos detalles de esta definición:

- Nótese que los métodos definidos dentro de la interfaz no contienen ningún tipo de código dentro de ellos, incluso no declaran ni llaves ({ }).

- A diferencia de los métodos definidos en Clases Abstractas, los métodos definidos en Interfaces no utilizan ningún calificador de acceso (public, private, protected).
- Cuando una Clase hace uso de una Interfaz esta utiliza la palabra implements, además **es necesario definir todos los métodos definidos en la Interfaz** dentro de la Clase.

Herencias Múltiples a través de Interfaces

```
import java.util.*;


public interface PuedeCurar {
   void curar();
}

public interface PuedeConsultar {
   void consultar();
}

public interface PuedeRecetar{
   void recetar();
}

public class Cirujano {
   public void operar() { System.out.println("Cirujano.operar()"); }
}

public class Medico extends Cirujano implements PuedeCurar, PuedeConsultar, PuedeRecetar {

   public void curar() { System.out.println("Medico.curar()"); }
   public void consultar() { System.out.println("Medico.consultar()"); }
   public void recetar() { System.out.println("Medico.recetar()"); }
}

public class Cardiologo {
    static void r (PuedeCurar x) { x.curar(); }
    static void s (PuedeConsultar x) { x.consultar(); }
    static void t (PuedeRecetar x) { x.recetar(); }
    static void u (Cirujano x) { x.operar(); }

public static void main (String[] args) {
                Medico m = new Medico ();
                r(m);
                s(m);
                t(m);
                u(m);
    }
}
```

En este archivo fuente son definidas tres Interfaces y tres Clases las cuales son descritas a continuación:

Interfaces PuedeCurar, PuedeConsultar, PuedeRecetar

Cada una de estas interfaces define un método, este deberá ser implementado en las respectivas Clases que hagan uso de las interfaces.

Clase Cirujano

Esta Clase simplemente define un método llamado operar que imprime un mensaje a pantalla.

Clase Médico

Contiene la mayor parte de código funcional de este archivo fuente:

- Es definida al igual que cualquier otra Clase con el vocablo class.
- Se utiliza el vocablo extends para heredar el comportamiento de la Clase Cirujano.
- Después se implementan las interfaces:

  PuedeCurar,PuedeConsultar,PuedeRecetar a través de implements, esto obliga a que sean definidos los métodos de las diversas interfaces.

- Son escritas las diversas implementaciones de los métodos de cada interfaz, los cuales envían un mensaje a pantalla.

Clase Cardiólogo

- Dentro del método principal (main) de esta Clase se genera una instancia de la Clase Medico.
- A través de dicha referencia son invocados los diversos métodos locales r, s, t, u.
- El método r manda llamar la función curar, nótese que aunque el dato de entrada aparece como PuedeCurar (Interfaz) la invocación se lleva acabo directamente de la Clase Medico (método curar).
- El método s manda llamar la función consultar, nótese que aunque el dato de entrada aparece como PuedeConsultar (Interfaz) la invocación se lleva acabo directamente de la Clase Medico (consultar).
- El método t manda llamar la función recetar, nótese que aunque el dato de entrada aparece como PuedeRecetar (Interfaz) la invocación se lleva acabo directamente de la Clase Medico (recetar).
- El método u invoca la función operar disponible en la Clase Cirujano.

SEGUNDO EJEMPLO DE HERENCIA MÚLTIPLE CON INTERFACES

Como ya hemos visto, un interfaz no es solamente una forma más pura de denominar a una clase abstracta, su propósito es mucho más amplio que eso. Como un interfaz no tiene ninguna implementación, es decir, no hay ningún tipo de almacenamiento asociado al interfaz, no hay nada que impida la combinación de varios interfaces. Esto tiene mucho valor porque hay veces en que es necesario que un objeto X sea a y b y c. En C++ es donde esto se acuñó como *herencia múltiple* y no es sencillo de hacer porque cada clase puede tener su propia implementación. En Java, se puede hacer lo mismo, pero solamente una de las clases puede tener implementación, con lo cual los problemas que se presentan en C++ no suceden con Java cuando se combinan múltiples interfaces.

```
import java.util.*;

public interface Luchar {
        void luchar();
}

public interface Nadar {
        void nadar();
}

public interface Volar {
        void volar();
}
public class Accion {
        public void luchar() {}
}

public  class Heroe extends Accion implements Luchar,Nadar,Volar {
        public void nadar() {}
        public void volar() {}
   }

public class Test {

        public static void a( Luchar x ) {
           x.luchar();
        }
        public staticvoid b( Nadar x ) {
           x.nadar();
        }
        public static void c( Volar x ) {
           x.volar();
        }
        public static void d( Accion x ) {
           x.luchar();
        }

        public static void main( String args[] ) {
           Heroe i = new Heroe();
           a( i ); // Trata al Heroe como Luchar
           b( i ); // Trata al Heroe como Nadar
           c( i ); // Trata al Heroe como Volar
           d( i ); // Trata al Heroe como Accion
        }
}
```

Se puede observar que Heroe combina la clase concreta Accion con los interfaces Luchar, Nadar y Volar. Cuando se combina la clase concreta con interfaces de este modo, la clase debe ir la primera y luego los interfaces; en caso contrario, el compilador dará un error.

La autentificación, *signature*, para *luchar()* es la misma en el interfaz Luchar y en la clase **Accion**, y *luchar()* no está proporcionado con la definición de **Heroe**. Si se quiere crear un objeto del nuevo tipo, la clase debe tener todas las definiciones que necesita, y aunque **Heroe** no proporciona una definición explicita para *luchar()*, la definición es proporcionada automáticamente por **Accion** y así es posible crear objetos de tipo **Heroe**.

En la clase **Test**, hay cuatro métodos que toman como argumentos los distintos interfaces y la clase concreta. Cuando un objeto **Heroe** es creado, puede ser pasado a cualquiera de estos métodos, lo que significa que se está realizando un *upcasting* al interfaz de que se trate. Debido a la forma en que han sido diseñados los interfaces en Java, esto funciona perfectamente sin ninguna dificultad ni esfuerzo extra por parte del programador.

La razón principal de la existencia de los interfaces en el ejemplo anterior es el poder realizar un *upcasting* a más de un tipo base. Sin embargo, hay una segunda razón que es la misma por la que se usan clases abstractas: evitar que el programador cliente tenga que crear un objeto de esta clase y establecer que solamente es un interfaz. Y esto plantea la cuestión de qué se debe utilizar pues, un interfaz o una clase abstracta.

Un interfaz proporciona los beneficios de una clase abstracta y, además, los beneficios propios del interfaz, así que es posible crear una clase base sin la definición de ningún método o variable miembro, por lo que se deberían utilizar mejor los interfaces que las clases abstractas. De hecho, si se desea tener una clase base, la primera elección debería ser un interfaz, y utilizar clases abstractas solamente si es necesario tener alguna variable miembro o algún método implementado.