

# Interfaces

## ¿Qué es un *interface*?

Un *interface* es una colección de declaraciones de métodos (sin definirlos) y también puede incluir constantes.

*Runnable* es un ejemplo de interface en el cual se declara, pero no se implementa, una función miembro *run*.

```
public interface Runnable {  
    public abstract void run();  
}
```

Las clases que implementen (**implements**) el interface *Runnable* han de definir obligatoriamente la función *run*.

```
class Animacion implements Runnable{  
    //..  
    public void run(){  
        //define la función run  
    }  
}
```

El papel del *interface* es el de describir algunas de las características de una clase. Por ejemplo, el hecho de que una persona sea un futbolista no define su personalidad completa, pero hace que tenga ciertas características que las distinguen de otras.

Clases que no están relacionadas pueden implementar el interface *Runnable*, por ejemplo, una clase que describa una animación, y también puede implementar el interface *Runnable* una clase que realice un cálculo intensivo.

## Diferencias entre un *interface* y una clase abstracta

Un *interface* es simplemente una lista de métodos no implementados, además puede incluir la declaración de constantes. Una clase abstracta puede incluir métodos implementados y no implementados o abstractos, miembros datos constantes y otros no constantes.

Ahora bien, la diferencia es mucho más profunda. Imaginemos que *Runnable* fuese una clase abstracta. Un applet descrito por la clase *MiApplet* que moviese una figura por su área de trabajo, derivaría a la vez de la clase base *Applet* (que describe la funcionalidad mínima de un applet que se ejecuta en un navegador) y de la clase *Runnable*. Pero el lenguaje Java no tiene herencia múltiple.

En el lenguaje Java la clase *MiApplet* deriva de la clase base *Applet* e implementa el interface *Runnable*

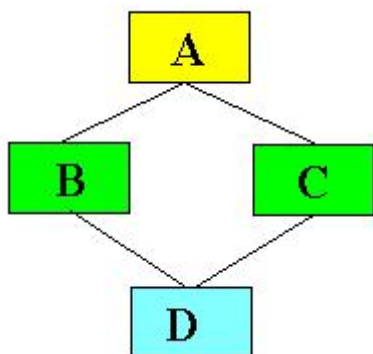
```
class MiApplet extends Applet implements Runnable{
//...
//define la función run del interface
    public void run(){
        //...
    }
//redefine paint de la clase base Applet
    public void paint(Graphics g){
        //...
    }
//define otras funciones miembro
}
```

Una clase solamente puede derivar **extends** de una clase base, pero puede implementar varios interfaces. Los nombres de los interfaces se colocan separados por una coma después de la palabra reservada **implements**.

El lenguaje Java no fuerza por tanto, una relación jerárquica, simplemente permite que clases no relacionadas puedan tener algunas características de su comportamiento similares.

## Los interfaces y el polimorfismo

En el lenguaje C++, es posible la herencia múltiple, pero este tipo de herencia presenta dificultades. Por ejemplo, cuando dos clases B y C derivan de una clase base A, y a su vez una clase D deriva de B y C. Este problema es conocido con el nombre de diamante.



En el lenguaje Java solamente existe la herencia simple, pero las clases pueden implementar interfaces. Vamos a ver en este apartado que la importancia de los interfaces no estriba en resolver los problemas inherentes a la herencia múltiple sin forzar relaciones jerárquicas, sino es el de incrementar el polimorfismo del lenguaje más allá del que proporciona la herencia simple.

Para explicar este aspecto importante y novedoso del lenguaje Java adaptaremos los ejemplos que aparecen en el artículo del Bill Venners "Designing with interfaces"

publicado en Java World ([www.javaWorld.com](http://www.javaWorld.com)) en Diciembre de 1998. Comparemos la herencia simple mediante un ejemplo similar al de la jerarquía de las figuras planas, con los interfaces.

## Herencia simple

Creamos una clase abstracta denominada *Animal* de la cual deriva las clases *Gato* y *Perro*. Ambas clases redefinen la función *habla* declarada abstracta en la clase base *Animal*.

```
public abstract class Animal {
    public abstract void habla();
}

class Perro extends Animal{
    public void habla(){
        System.out.println("¡Guau!");
    }
}

class Gato extends Animal{
    public void habla(){
        System.out.println("¡Miau!");
    }
}
```

El polimorfismo nos permite pasar la referencia a un objeto de la clase *Gato* a una función *hazleHablar* que conoce al objeto por su clase base *Animal*

```
public class PoliApp {
    public static void main(String[] args) {
        Gato gato=new Gato();
        hazleHablar(gato);
    }

    static void hazleHablar(Animal sujeto){
        sujeto.habla();
    }
}
```

El compilador no sabe exactamente que objeto se le pasará a la función *hazleHablar* en el momento de la ejecución del programa. Si se pasa un objeto de la clase *Gato* se imprimirá ¡Miau!, si se pasa un objeto de la clase *Perro* se imprimirá ¡Guau!. El compilador solamente sabe que se le pasará un objeto de alguna clase derivada de *Animal*. Por tanto, el compilador no sabe que función *habla* será llamada en el momento de la ejecución del programa.

El polimorfismo nos ayuda a hacer el programa más flexible, por que en el futuro podemos añadir nuevas clases derivadas de *Animal*, sin que cambie para nada el método *hazleHablar*. Como ejercicio, se sugiere al lector añadir la clase *Pajaro* a la jerarquía, y

pasar un objeto de dicha clase a la función *hazleHablar* para que se imprima ¡pio, pio, pio ...!.

## Interfaces

Vamos a crear un interface denominado *Parlanchin* que contenga la declaración de una función denominada *habla*.

```
public interface Parlanchin {  
    public abstract void habla();  
}
```

Hacemos que la jerrarquía de clases que deriva de *Animal* implemente el interface *Parlanchin*

```
public abstract class Animal implements Parlanchin{  
    public abstract void habla();  
}  
  
class Perro extends Animal{  
    public void habla(){  
        System.out.println(";Guau!");  
    }  
}  
  
class Gato extends Animal{  
    public void habla(){  
        System.out.println(";Miau!");  
    }  
}
```

Ahora veamos otra jerarquía de clases completamente distinta, la que deriva de la clase base *Reloj*. Una de las clases de dicha jerarquía *Cucu* implementa el interface *Parlanchin* y por tanto, debe de definir obligatoriamente la función *habla* declarada en dicho interface.

```
public abstract class Reloj {  
}  
  
class Cucu extends Reloj implements Parlanchin{  
    public void habla(){  
        System.out.println(";Cucu, cucu, ..!");  
    }  
}
```

Definamos la función *hazleHablar* de modo que conozca al objeto que se le pasa no por una clase base, sino por el interface *Parlanchin*. A dicha función le podemos pasar cualquier objeto que implemente el interface *Parlanchin*, este o no en la misma jerarquía de clases.

```
public class PoliApp {  
  
    public static void main(String[] args) {
```

```
Gato gato=new Gato();
hazleHablar(gato);
Cucu cucu=new Cucu();
hazleHablar(cucu);
}

static void hazleHablar(Parlanchin sujeto){
    sujeto.habla();
}
}
```

Al ejecutar el programa, veremos que se imprime en la consola ¡Miau!, por que a la función *hazleHablar* se le pasa un objeto de la clase *Gato*, y después ¡Cucu, cucu, ..! por que a la función *hazleHablar* se le pasa un objeto de la clase *Cucu*.

Si solamente hubiese herencia simple, *Cucu* tendría que derivar de la clase *Animal* (lo que no es lógico) o bien no se podría pasar a la función *hazleHablar*. Con interfaces, cualquier clase en cualquier familia puede implementar el interface *Parlanchin*, y se podrá pasar un objeto de dicha clase a la función *hazleHablar*. Esta es la razón por la cual los interfaces proporcionan más polimorfismo que el que se puede obtener de una simple jerarquía de clases.