

Este resumen ha sido elaborado para ser leído después de haber asistido a clase: pretende fijar y profundizar en los conceptos ya explicados.

Interfaces

Introducción

Nos servirá de introducción recordar que las clases abstractas¹ *tienen la función de formalizar una especificación que cumplirán todas las clases derivadas a partir de ella*. Los que usen esa familia de clases en sus programas (los *clientes* de la clase) tienen la seguridad de que las subclasses cumplen esta especie de contrato. Estamos estableciendo un “protocolo” de comunicación entre nuestra jerarquía de clases y las clases que usen nuestra jerarquía de clases.

Consideremos lo que sucede, por ejemplo, con la clase abstracta `Reader` (de `java.io`) y sus descendientes: todas implementan el método `read()`, que lee un caracter de un stream. Las clases derivadas especializan a `Reader` (por ejemplo, `FileReader` lee de un archivo), algunas implementan una funcionalidad adicional (es el caso de `BufferedReader`, que añade `readLine()` para leer líneas). Aunque hagan un *override* de `read`, todas respetan la definición de `read()`: devuelve un `int` y puede arrojar una `IOException`. Los que usan `Reader` o alguno de sus descendientes en sus programas saben siempre a qué atenerse.

Qué es una Interface (*interfaz*)

Una interfaz es una clase completamente abstracta, *sin ninguna implementación*. Nos permite especificar a) los nombres de cada método; b) los argumentos del método; y, c) los tipos de retorno. No permite, en cambio, ninguna implementación de los métodos: todos sus métodos son abstractos.

La interfaz puede definir también atributos (variables), pero son siempre finales y estáticos. Los métodos, por su parte, son siempre publicos.²

Al declarar una interfaz estamos diciendo a los que la usarán: “todas las clases que implementen una interfaz se verán así”.

1 Clases abstractas son aquellas que se declaran con la palabra reservada `abstract`. Pueden definir métodos y variables, pero no se pueden usar directamente para crear objetos usando `new`. Es necesario crear una clase que extienda la clase abstracta y “levante la abstracción”. Con levantar la abstracción nos referimos a implementar todos los métodos que en la clase abstracta hayan sido declarados `abstract`. Por ejemplo:

```
abstract class ClaseAbstracta {
    public abstract void f();
    public toString() { return "ClaseAbstracta"; }
}
```

`f()` es un método abstracto que incluye la implementación (no tiene código), sólo se define la firma del método (nombre y parámetros). `toString()`, en cambio, no es un método abstracto y sí incluye implementación.

```
class SubClase extends ClaseAbstracta {
    public void f() { System.out.println("Soy f()"); }
}
```

La clase derivada `SubClase` levanta la abstracción, pues deriva de `ClaseAbstracta` e implementa `f()`.

2 Esto es lógico, pues los métodos privados de una clase sólo pueden ser usados por otros métodos de la misma clase. Si la interfaz no incluye la implementación de ninguno de sus métodos, y además los métodos privados no se heredan ¿quién podría usar un método privado dentro de la interfaz?

Sintaxis

Las interfaces se declaran de modo similar a las clases, pero usando la palabra reservada **interface** en vez de **class**. La interfaz puede ser **public** o no (según queramos que el acceso sea para todos o sólo para las otras clases del package). Al declarar los métodos, no es necesario especificar que son **public**, ni se especifica tampoco que las variables son **static** o **final**, porque de hecho lo son siempre en una interfaz.

Una interfaz definida para el ejemplo que usamos al explicar polimorfismo podría verse así:

```
public interface Instrumento {
    public void play(Nota n);
    public void ajustar();
}
```

Para derivar una clase a partir de la interfaz, se usa la palabra reservada **implements**. Se dice que la clase *implementa* una determinada interfaz. Así,

```
public class DeViento implements Instrumento {
    public void play(Nota n) {
        System.out.println(this+".play()"+n);
    }

    public void ajustar() {
        System.out.println(this+".ajustar()");
    }
}

public class DePercusion implements Instrumento {
    public void play(Nota n) {
        System.out.println(this+".play()"+n);
    }

    public void ajustar() {
        System.out.println(this+".ajustar()");
    }
}
```

La clase que implementa la interfaz (en este caso DeViento) debe implementar todos los métodos de la interfaz, salvo que declare alguno de ellos abstractos (y, por tanto, toda la clase DeViento sea siendo abstracta).

Además, una clase puede implementar varias interfaces, lo que en la práctica puede verse como un modo de heredar de diferentes clases³:

```
class Test implements interfazA, interfazB {
    (...)
}
```

La clase Test tendrá que implementar todos los métodos definidos en interfazA y también los de interfazB.

Es perfectamente válido combinar **extends** con **interface**:

```
class SubClase extends ClaseBase implements interfazA, interfazB {
    (...)
}
```

³ Aunque en realidad no es una herencia “completa”: no heredo la implementación de los métodos, pues la interfaz no implementa ninguno.

Uso de las interfaces

Las interfaces se usan del mismo modo que las clases: pueden especificar el tipos de parámetros en las declaraciones de métodos; se pueden declarar variables del tipo de la interfaz; se puede hacer *upcasting* y *downcasting* de subclases de la interfaz; las llamadas a métodos de subclases de la interfaz son polimórficas.

En ese sentido, siguiendo con el ejemplo de Instrumento, al cambiar Instrumento de una clase abstracta a una interfaz no necesitaríamos cambiar la definición de `afinar` ni de `afinarTodos` en Musica:

```
public class Musica
{
    static void afinar(Instrumento i) {
        i.play(Nota.DO);
    }

    static void afinarTodos(Instrumento[] instrumentos) {
        for (Instrumento i: instrumentos) {
            afinar(i);
        }
    }

    public static void main(String[] args) {
        Instrumento[] orquesta = {
            new DePercusion(),
            new DeViento(),
            new Flauta(),
            new Oboe(),
            new Tambor()
        };
        afinarTodos(orquesta);
    }
}
```

Sin embargo, no hemos ganado nada con respecto a cómo estaba antes. Incluso se podría decir que hemos perdido, porque para poder declarar Instrumento como interface, hemos tenido que duplicar la implementación de `play` y ajustar tanto en `DeViento` como en `DePercusion` para no malograr el resto de las clases.

Qué buscamos al definir una interfaz

Vamos a considerar la interfaz desde dos puntos de vista complementarios.

Por un momento dejemos de lado Instrumento (supongamos que es una clase que va a escribir otra persona) y fijémonos solamente en Musica.

Para afinar el instrumento necesitamos tocar la nota Do. Por eso, nuestro método `afinar` se ve algo así:

```
public static4 void afinar(           i ) {
    i.play(Nota.DO);
}
```

⁴ Estamos declarando `afinar` como `static` simplemente para poder llamarlo desde `main`, que es un método estático. (Recordemos que los métodos estáticos sólo pueden llamar otros métodos estáticos). Puedo declarar `afinar` estático porque `afinar` no modifica ningún atributo de la clase.

Hasta ahora hemos usado como parámetro de afinar un objeto *i* de tipo Instrumento (en unos casos una clase, en otro una interfaz). Pero realmente, ¿cuál es la mínima condición necesaria que debe cumplir *i* para que nuestro método funcione? La mínima condición es que *i* tenga un método `play(Nota n)`. Visto desde el punto de vista de afinar, realmente no me interesa si *i* tiene un método `ajustar()`, o si reemplaza `toString()`, pues no los necesito para escribir afinar.

El uso de interfaces me permite definir un tipo de objeto **que sólo exija los métodos estrictamente necesarios**, en este caso, un método `play`. A la vez, no restringe al objeto *i* en cuanto a qué otros métodos pueda tener (ya se porque los declara directamente, los hereda via `extends`, o porque implementa otras interfaces). Desde el punto de vista de afinar, sólo nos interesa que *i* implemente `play`.

Definamos la interfaz mínima que necesitamos para que nuestro método afinar funcione. Le llamaremos `Playable`⁵:

```
public interface Playable {
    void play(Nota n);
}
```

Ahora podemos completar la definición de nuestro método afinar:

```
public static void afinar( Playable i ) {
    i.play(Nota.DO);
}
```

y de `afinarTodos`:

```
public static void afinarTodos( Playable[] e ) {
    for( Playable i: e ) {
        i.play(Nota.DO);
    }
}
```

Nuevamente notamos cómo las interfaces se usan igual que si fueran una clase abstracta (antes `afinar` usaba un parámetro de tipo Instrumento, y `afinarTodos` un array de instrumentos).

La nueva clase `Musica`, completa, se vería así:

```
public class Musica
{
    public static void afinar(Playable i) {
        i.play(Nota.DO);
    }

    public static void afinarTodos(Playable[] e) {
        for(Playable i: e) {
            i.play(Nota.DO);
        }
    }

    public static void main(String[] args) {
        Instrumento[] orquesta = {
            new Tambor(),
            new Flauta(),
            new Oboe()
        };
        afinarTodos(orquesta);
    }
}
```

⁵ En el sentido de que las clases que implementen `Playable` serán capaces de tocar una nota.

```
    }
}
```

Bien, ahora nos ponemos del otro lado del problema: acabo de pagar \$50 por una clase `Musica` que permite afinar instrumentos. No sabemos cómo está implementada. Sólo nos dicen que la clase tiene dos métodos, que esperan un objeto que implemente `Playable` (y nos dan la declaración de `Playable`⁶).

```
public interface Playable {
    void play(Nota n);
}

public void afinar(Playable i)
public void afinarTodos(Playable[] e)
```

¿Cómo elaboramos una clase que pueda ser el parámetro de estos dos métodos? Declaramos una clase que implemente la interfaz `Playable`:

```
public abstract7 class Instrumento implements Playable {
    public void play(Nota n) {
        System.out.println8(this+".play() " + n);
    }
}
```

Como vemos, este `Instrumento` no implementa el método `ajustar` que tenía la clase `Instrumento`, pues la interfaz `Playable` no me lo exige. Para que los instrumentos que ya estaban definidos no dejen de funcionar, lo lógico es declarar una interfaz `Ajustable` que declare el método `ajustar`, y declarar que `Instrumento` implementa esta interfaz, de modo que la clase `Instrumento` no pierda la funcionalidad que tenía:

```
public interface Playable {
    void play(Nota n);
}

public interface Ajustable {
    void ajustar();
}

public abstract9 class Instrumento implements Playable, Ajustable {
    public void play(Nota n) {
        System.out.println(this+".play() " + n);
    }

    public void ajustar() {
        System.out.println(this+".ajustar()");
    }
}
```

`DeViento`, `DePercusion` y sus respectivas clases derivadas no necesitan ningún cambio.

En resumen, los dos puntos de vista distintos desde los que hemos enfocado el problema son:

a) Desde el punto de vista del que escribe un método, se usa una interfaz para dejar sentado

⁶ Y el enum `Nota`, se entiende.

⁷ Estamos declarando `Instrumento` como `abstract` porque la usaremos sólo para crear subclases de instrumentos concretos. Si quitamos el `abstract`, en este caso, igual funcionará el ejemplo.

⁸ Decir que `play()` imprime un texto en la consola es solamente un ejemplo... una clase para un instrumento por lo menos debería producir algún tipo de sonido.

⁹ Estamos declarando `Instrumento` como `abstract` porque la usaremos sólo para crear subclases de instrumentos concretos. Si quitamos el `abstract`, en este caso, igual funcionará el ejemplo.

qué métodos se espera que estén disponibles en el objeto que se recibe como parámetro. Se trata de los métodos que son estrictamente necesarios para el funcionamiento de mi método.

b) Desde el punto del que usa un método, la interfaz dice qué métodos deben implementar, como mínimo, los objetos que se pasen como parámetros del método.

Ya se entiende que las dos partes necesitan conocer la declaración de la interfaz.

BubbleSort e interfaces

Consideremos otro ejemplo: el algoritmo **BubbleSort**. Hasta ahora, cuando hemos usado el BubbleSort, nos hemos limitado a ordenar listas de enteros. ¿Cómo podríamos definir un sort que funcione con cualquier tipo de objetos?

La clase BubbleSort, que ordena de menor a mayor, es la siguiente:

```
public class BubbleSort {  
  
    public static int[] sort( int[] l ) {  
        int[] lista = l.clone();  
  
        for(int i=0; i<lista.length-1;i++) {  
            for(int j=i;j<lista.length;j++) {  
  
                1 ➡ if( lista[i] > lista[j] ) {  
  
                    int tmp = lista[i];  
                    lista[i] = lista[j];  
                    lista[j] = tmp;  
  
                }  
            }  
        }  
        return lista;  
    } // end sort  
  
} // end class
```

El punto neurálgico del sort es la línea que compara dos valores de la lista para determinar si deben intercambiar sus posiciones porque uno es mayor que el otro (1 ➡). Para una lista de enteros, está claro cómo se comparan dos números. Pero, ¿cómo comparo dos objetos de una clase que no conozco? Recordemos que estamos diciendo que nuestro sort debe funcionar con cualquier tipo de objetos.

Una solución es definir una interfaz con un método que permita comparar dos objetos del tipo de la interfaz, y ordenar objetos del tipo de esa interfaz en vez de enteros. Las clases que implementen esta interfaz usarán el método de la interfaz para definir cuál es el criterio para ordenar una lista de objetos de esta clase¹⁰.

Para facilitarnos las cosas, la librería estándar de Java ya trae una interfaz como la que queremos:

```
public interface Comparable {  
    int compareTo(Object o);  
}
```

¹⁰ Probablemente uno de los atributos de la clase será el que se use como criterio de ordenamiento.

Como esta interfaz es parte de `java.lang`¹¹, no hace falta volverla a escribir. Las clases que quieran implementar esta interfaz deben declarar un método `compareTo` tal que `o1.compareTo(o2)` devuelva:

```
-1    si o1 < o2
0     si o1 es igual a o2
1     si o1 > o2
```

Reescribamos el sort de modo que use una interfaz `Comparable` como tipo de la lista a ordenar:

```
public class BubbleSort
{
    // ordena ascendente
    static Comparable[] sort( Comparable[] lista ) {

        Comparable[] o = lista.clone();

        for(int i=0; i<o.length-1;i++) {
            for(int j=i;j<o.length;j++) {

                if( o[i].compareTo(o[j]) > 0 ) {

                    Comparable tmp = o[j];
                    o[j] = o[i];
                    o[i] = tmp;
                }
            }
        }
        return o;
    }
}
```

Ya tenemos un sort que ordena cualquier clase de objetos que implementen la interfaz `Comparable`. Ahora probemos el sort con alguna clase.

Vamos a definir una clase `Persona` que implemente `Comparable`. El criterio de ordenación será la edad de la persona.

```
public class PersonaPorEdad implements Comparable {
    String nombre;
    int edad;

    PersonaPorEdad(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }


    public int compareTo(Object p) {
        if(edad < ( (PersonaPorEdad) p).edad)
            return -1;
        else if ( edad > ((PersonaPorEdad) p).edad )
            return 0;
        else return 1;
    }
}
```

Hay que notar que el método `compareTo` espera como parámetro un `Object` (1), de modo que si lo declaramos `public int compareTo(PersonaPorEdad p)` se producirá un error de compilación. Esto nos complica un poco la sintaxis de la comparación, pues tenemos que hacer un


¹¹ `java.lang` se incluye en todos los programas automáticamente.


downcast para comparar las edades:

```
if(edad < ( (PersonaPorEdad) p ).edad)
```



Si p1 y p2 son dos objetos de tipo PersonaPorEdad, entonces si decimos p1.comparteTo(p2),

edad () es la edad de p1;

p () es de tipo Object y contiene p2;

El *downcast* (PersonaPorEdad) p le dice al compilador que p es realmente de tipo PersonaPorEdad;

((PersonaPorEdad) p).edad obtiene la edad de p2.

Esta sintaxis engorrosa se puede evitar gracias a que la interfaz Comparable puede usar *Generics*. Para eso, definimos la clase del siguiente modo:

```
public class PersonaGenerics implements Comparable<PersonaGenerics> {  
    (...)  
}
```

Es como si estuviéramos diciendo que en la interfaz Comparable se reemplace la palabra Object por PersonaGenerics. Por eso, el método comparteTo se puede declarar simplemente como int comparteTo(PersonaGenerics p). La nueva clase se vería así (hemos añadido un main que cree algunos objetos y use el BubbleSort para ordenarlos):

```
public class PersonaGenerics implements Comparable<PersonaGenerics> {  
  
    String nombre;  
    public int edad;  
  
    public PersonaGenerics(String nombre, int edad) {  
        this.nombre = nombre;  
        this.edad = edad;  
    }  
  
    public int compareTo(PersonaGenerics p) {  
        if(edad < p.edad)  
            return -1;  
        else if ( edad == p.edad )  
            return 0;  
        else return 1;  
    }  
  
    public static void main(String[] args) {  
        PersonaGenerics[] personas = {  
            new PersonaGenerics("Otto", 18),  
            new PersonaGenerics("Peter", 32),  
            new PersonaGenerics("Helga", 12),  
            new PersonaGenerics("Fritz", 9),  
            new PersonaGenerics("Elke", 45),  
            new PersonaGenerics("Max", 17),  
            new PersonaGenerics("Moritz", 22),  
            new PersonaGenerics("Walter", 11)  
        };  
        System.out.println("Antes:");  
        for( PersonaGenerics p : personas) {  
            System.out.println(p.edad + ", " + p.nombre);  
        }  
    }  
}
```

```

        PersonaGenerics[] ordenada =
            (PersonaGenerics[]) BubbleSort.sort(personas);

        System.out.println("\nDespués:");
        for( PersonaGenerics p : ordenada) {
            System.out.println(p.edad + ", " + p.nombre);
        }
    }
}

```

Nótese que el hecho de que en `PersonaGenerics` utilicemos *generics* no afecta para nada a la clase `BubbleSort`. De hecho, como `BubbleSort.sort` no acepta *generics* sino un objeto `Comparable`, al recibir el valor de retorno de `BubbleSort.sort` estamos haciendo un downcast para convertir el objeto de tipo `Comparable` a un objeto de tipo `PersonaGenerics`. Esto es necesario porque de lo contrario no podríamos acceder a `p.edad` ni a `p.nombre`, que no están declarados en la interfaz `Comparable`.

Hay un beneficio adicional que se obtiene usar `Comparable`, que ya está definida en Java, en vez de definir nuestra propia interfaz (“Ordenable”, por decir un algo). La documentación de `Comparable` dice: *Lists (and arrays) of objects that implement this interface can be sorted automatically by Collections.sort (and Arrays.sort)*. Traducido, hemos escrito el `BubbleSort` por gusto, porque si una clase implementa `Comparable`, los elementos de un arreglo de objetos de esa clase pueden ser ordenados usando `Arrays.sort`.

```

import java.util.Arrays;


public class Persona implements Comparable<Persona> {

    String nombre;
    public int edad;

    public Persona(String nombre, int edad) {
        this.nombre = nombre;
        this.edad = edad;
    }

    public int compareTo(Persona p) {
        if(edad < p.edad)
            return -1;
        else if ( edad == p.edad )
            return 0;
        else return 1;
    }

    public static void main(String[] args) {
        Persona[] personas = {
            new Persona("Otto", 18),
            new Persona("Peter", 32),
            new Persona("Helga", 12),
            new Persona("Fritz", 9),
            new Persona("Elke", 45),
            new Persona("Max", 17),
            new Persona("Moritz", 22),
            new Persona("Walter", 11)
        };
        System.out.println("Antes:");
        for( Persona p : personas) {
            System.out.println(p.edad + ", " + p.nombre);
        }

         Arrays.sort(personas);
    }
}

```

```

        System.out.println("\nDespués:");
        for( Persona p : personas) {
            System.out.println(p.edad + ", " + p.nombre);
        }
    }
}

```

Cabe notar que `Arrays.sort`, a diferencia de nuestro `BubbleSort.sort`, sí modifica el arreglo que se pasa como parámetro. Sin embargo es más eficiente que el `BubbleSort`.

Algunas notas sobre las interfaces

Interfaces que extienden otras interfaces

Es perfectamente lícito decir:

```

interface AnimalPeligroso {
    void muerde();
}

interface Perro {
    void ladra();
    void mueveCola();
}

```

y crear una nueva interfaz combinando las interfaces `Perro` y `AnimalPeligroso`:

```

interface PerroRabioso extends Perro, AnimalPeligroso {
}

```

Nótese que en el caso de las interfaces, sí se pueden especificar varias interfaces después de la palabra `extends`. Las clases que implementen `PerroRabioso` tendrán que implementar los métodos `ladra()`, `muerde()` y `mueveCola()`.

Atributos en las interfaces

Todos los atributos que se declaren en una interfaz son automáticamente estáticos y finales. Realmente, se podría discutir sobre si los atributos que declara la interfaz son o no parte de la interfaz, porque el método que implementa la interfaz no puede “implementar” los atributos. Vamos a dejar la discusión para los puristas del lenguaje. Pero sí podríamos decir que los atributos de la interfaz son constantes que acompañan siempre a la interfaz que los declara.

Muchas veces se usan los atributos de las interfaces para crear grupos de constantes:

```

public interface Nota {
    int DO =1, RE=2, MI=3, FA=4, SOL=5, LA=6, SI=7;
}

```

Es costumbre escribir los nombres de estos atributos con mayúscula.

Java 1.5 introduce `enum`, una construcción que permite definir grupos de constantes de un modo

más eficiente, y es preferible usar esa notación en vez de declarar las constantes dentro de las interfaces:

```
enum Nota {
    DO, RE, MI, FA, SOL, LA, SI
}
```

También hay que notar que las interfaces **no permiten atributos sin inicializar**. Se pueden usar expresiones para inicializar estas variables (es decir, no necesariamente hay que usar valores constantes de valor inicial, pueden usarse otros métodos), pero no pueden quedar en blanco.

```
public interface NumerosAleatorios {
    Random RAND = new Random();
    int RANDOM_INT = RAND.nextInt(10);
}
```

Clases que implementan más de una interfaz

Consideremos las dos interfaces siguientes:

```
interface AnimalPeligroso {
    void muerde();
}

interface Perro {
    void ladra();
    void mueveCola();
}
```

Si definimos la clase

```
public class PerroPeligroso implements AnimalPeligroso, Perro {
    public void muerde() { ... }
    public void ladra() { ... }
    public void mueveCola() { ... }
}
```

entonces podemos decir que todos los PerroPeligroso son AnimalPeligroso, y también que todo los PerroPeligroso son Perro. Gracias al polimorfismo, las siguientes expresiones son válidas:

```
Perro kujo = new PerroPeligroso();
kujo.ladra();
kujo.mueveCola();

AnimalPeligroso fido = new PerroPeligroso();
fido.muerde();
```

En cambio no puedo decir `kujo.muerde()` porque aunque `kujo` contenga un `PerroPeligroso` y el nombre de la variable dé la impresión de un cucho agresivo, `kujo` es de tipo `Perro`, y `Perro` sólo define `ladra()` y `mueveCola()`. Recordemos que sólo puedo llamar a los métodos que están definidos en la clase del tipo que almacena el objeto, aunque el objeto tenga más métodos definidos. Por eso tampoco puedo decir `fido.ladra()` ni `fido.mueveCola()`, pues `fido` está definido como `AnimalPeligroso`.

Colisión de nombres

¿Qué sucede si dos interfaces declaran métodos del mismo nombre y una clase intenta

implementar las dos interfaces? Mientras los métodos tengan firmas distintas (tipo o número de parámetros distintos), el compilador podrá diferenciar entre uno y otro método. En cambio, no es suficiente que los métodos se diferencien sólo por el tipo de retorno.

Un ejemplo:

```
public interface I1 {
    int read();
}

public interface I2 {
    int read();
    void write();
}
```

No puedo decir

```
public class Test implements I1, I2 {
    (...)
}
```

pues las dos interfaces definen el método read. En cambio, si

```
public interface I3 {
    int read(String name);
}
```

no hay ningún inconveniente en decir

```
public class Test implements I1, I3 {
    public int read() { ... }
    public int read(String nombre) { ... }
}
```

o también,

```
public class Test implements I2, I3 {
    public int read() { ... }
    public int read(String nombre) { ... }
    public void write() { ... }
}
```

pues la declaración de read en I3 tiene firma distinta a la de I1 e I2.

Comentarios, correcciones y sugerencias: Roberto Zoia (roberto.zoia@gmail.com)

This work is licensed under the Creative Commons Attribution-ShareAlike 2.5 License. To view a copy of this license, visit <http://creativecommons.org/licenses/by-sa/2.5/> or send a letter to Creative Commons, 543 Howard Street, 5th Floor, San Francisco, California, 94105, USA.