INTERFACES EN JAVA

En la programación orientada a objetos, a veces es útil definir qué debe hacer una clase, pero no cómo lo hará. Ya has visto un ejemplo de esto: el método abstract. Un método abstracto define la firma de un método pero no proporciona ninguna implementación. Una subclase debe proporcionar su propia implementación de cada método abstracto definido por su superclase. Por lo tanto, **un método abstracto especifica la interfaz para el método pero no la implementación**.

Si bien las clases y métodos abstractos son útiles, es posible llevar este concepto un paso más allá. En Java, puede separar por completo la interfaz de una clase de su implementación utilizando la palabra clave `interface`.

1. Qué es una interface en Java

Una interfaz (`interface`) es sintácticamente similar a una clase abstracta, en la que puede especificar uno o más métodos que no tienen cuerpo (*{}*). Esos métodos deben ser implementados por una clase para que se definan sus acciones.

Por lo tanto, **una 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.

Para implementar una interfaz, una clase debe proporcionar cuerpos (implementaciones) para los métodos descritos por la interfaz. Cada clase es libre de determinar los detalles de su propia implementación. Dos clases pueden implementar la misma interfaz de diferentes maneras, pero **cada clase aún admite el mismo conjunto de métodos**. Por lo tanto, el código que tiene conocimiento de la interfaz puede usar objetos de cualquier clase, ya que la interfaz con esos objetos es la misma.

✕Al proporcionar la palabra clave **interface**, Java le permite utilizar completamente el aspecto de «una interfaz, múltiples métodos» del **polimorfismo**.

2. interface en el nuevo JDK

Antes de continuar, se necesita hacer un punto importante. 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, como se acaba de describir. **JDK 8** cambió esto.

Hoy, es 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**. Por lo tanto, ahora es posible que la interfaz especifique algún comportamiento.

Sin embargo, tales métodos constituyen lo que son, en esencia, características de uso especial, y la intención original detrás de la interfaz aún permanece. Por lo tanto, como regla general, con frecuencia creará y utilizará interfaces en las que no se utilizarán estas nuevas funciones. Por esta razón, comenzaremos discutiendo la interfaz en su forma tradicional. Las nuevas funciones de la interfaz se describen más adelante.

Aquí hay una forma general simplificada de una interfaz tradicional:

```
acceso interface nombre {

tipo-retorno metodo-nombre1(lista-parametros);
```

```
tipo-retorno metodo-nombre2(lista-parametros);
tipo var1 = valor;
tipo var2 = valor;
// ...
tipo-retorno metodo-nombreN(lista-parametros);
tipo varN = valor;
}
```

Para una interfaz de nivel superior, *acceso* es ***public*** o no se usa. Cuando no se incluye ningún modificador de acceso, los resultados de acceso predeterminados y la interfaz solo están disponibles para otros miembros de su paquete. Si se declara como **public**, la interfaz puede ser utilizada por cualquier otro código. (Cuando una interfaz se declara **public**,, debe estar en un archivo del mismo nombre.) *nombre* es el nombre de la interfaz y puede ser cualquier identificador válido.

En la forma tradicional de una interfaz, los métodos se declaran utilizando solo su tipo de devolución y firma. Son, esencialmente, métodos abstractos. Por lo tanto, cada clase que incluye dicha interfaz debe implementar todos sus métodos.

En una interfaz, los métodos son implícitamente públicos.

Las variables declaradas en una interfaz no son variables de instancia. En cambio, son implícitamente *public*, *final*, y *static*, y deben inicializarse. Por lo tanto, son esencialmente constantes.

Aquí hay un ejemplo de una definición de interfaz. Especifica la interfaz a una clase que genera una serie de números.

```
public interface Series {
    int getSiguiente(); //Retorna el siguiente número de la serie
    void reiniciar(); //Reinicia
    void setComenzar(int x); //Establece un valor inicial
}
```

Esta interfaz se declara pública para que pueda ser implementada por código en cualquier paquete.

3. Implementación de interfaces

Una vez que se ha definido una interfaz, una o más clases pueden implementar esa interfaz. Para implementar una interfaz, incluya la cláusula `implements` en una definición de clase y luego cree los métodos requeridos por la interfaz. La forma general de una clase que incluye la cláusula de **implements** se ve así:

```
class nombreclase extends superclase implements interface {
// cuerpo-clase
}
```

Para implementar más de una interfaz, las interfaces se separan con una coma.

✕Por supuesto, la cláusula extends es opcional.

Los métodos que implementan una interfaz deben declararse públicos. Además, la firma de tipo del método de implementación debe coincidir exactamente con la firma de tipo especificada en la definición de la interfaz.

Aquí hay un ejemplo que implementa la interfaz de *Series* mostrada anteriormente. Crea una clase llamada *DeDos*, que genera una serie de números, cada uno mayor que el anterior.

```
class DeDos implements Series {
    int iniciar;
    int valor;

    DeDos(){
        iniciar=0;
        valor=0;
    }



    public int getSiguiente() {
        valor+=2;
        return valor;
    }

    public void reiniciar() {
        valor=iniciar;
    }

    public void setComenzar(int x) {
        iniciar=x;
        valor=x;
    }
}
```

Observe que los métodos *getSiguiente()*, *reiniciar()* y *setComenzar()* se declaran utilizando el especificador de acceso público (`public`). Esto es necesario. Siempre que implemente un método definido por una interfaz, debe implementarse como público porque todos los miembros de una interfaz son implícitamente públicos.

Aquí hay una clase que demuestra *DeDos*:

```
class SeriesDemo {
    public static void main(String[] args) {
        DeDos ob=new DeDos();

        for (int i=0;i<5;i++)
```

```
            System.out.println("Siguiente valor es: "+ob.getSiguiente());

        System.out.println("\nReiniciando");
        ob.reiniciar();
        for (int i=0;i<5;i++)
            System.out.println("Siguiente valor es: "+ob.getSiguiente());
        System.out.println("\nIniciando en 100");
        ob.setComenzar(100);
        for (int i=0;i<5;i++)
            System.out.println("Siguiente valor es: "+ob.getSiguiente());
    }
}
```

Salida:

```
Siguiente valor es: 2
Siguiente valor es: 4
Siguiente valor es: 6
Siguiente valor es: 8
Siguiente valor es: 10

Reiniciando
Siguiente valor es: 2
Siguiente valor es: 4
Siguiente valor es: 6
Siguiente valor es: 8
Siguiente valor es: 10

Iniciando en 100
Siguiente valor es: 102
Siguiente valor es: 104
Siguiente valor es: 106
Siguiente valor es: 108
Siguiente valor es: 110
```

Es permitido y común para las clases que implementan interfaces definir miembros adicionales propios. Por ejemplo, la siguiente versión de *DeDos* agrega el método *getAnterior()*, que devuelve el valor anterior:

```
class DeDos implements Series {
    int iniciar;
    int valor;
    int anterior;
```

```
    DeDos(){
        iniciar=0;
        valor=0;
    }



    public int getSiguiente() {
        anterior=valor;
        valor+=2;
        return valor;
    }

    public void reiniciar() {
        valor=iniciar;
        anterior=valor-2;
    }

    public void setComenzar(int x) {
        iniciar=x;
        valor=x;
        anterior=x-2;
    }

    //Añadiendo un método que no está definido en Series
    int getAnterior(){
        return anterior;
    }
}
```

Observe que la adición de *getAnterior()* requirió un cambio en las implementaciones de los métodos definidos por *Series*. Sin embargo, dado que la interfaz con esos métodos permanece igual, el cambio es continuo y no rompe el código preexistente. Esta es una de las ventajas de las interfaces.

Como se explicó, cualquier cantidad de clases puede implementar una interfaz. Por ejemplo, aquí hay una clase llamada *DeTres* que genera una serie que consta de múltiplos de tres:

```
public class DeTres implements Series{
    int iniciar;
    int valor;
```

```
    DeTres(){
        iniciar=0;
        valor=0;
    }


    public int getSiguiente() {
        valor+=3;
        return valor;
    }

    public void reiniciar() {
        valor=iniciar;
    }

    public void setComenzar(int x) {
        iniciar=x;
        valor=x;
    }
}
```

Un punto más: si una clase incluye una interfaz pero no implementa completamente los métodos definidos por esa interfaz, **esa clase debe declararse como abstracta** (`abstrac`). No se pueden crear objetos de dicha clase, pero se puede usar como una **superclase abstracta**, lo que permite que las subclases proporcionen la implementación completa.

4. Uso de referencias a interface

Es posible que se sorprenda al descubrir que puede declarar una variable de referencia de un tipo de interfaz. En otras palabras, puede crear una **variable de referencia de interfaz**.

Dicha variable puede referirse a cualquier objeto que implemente su interfaz. Cuando llama a un método en un objeto a través de una referencia de interfaz, es la versión del método implementado por el objeto que se ejecuta. Este proceso es similar al uso de una referencia de superclase para acceder a un objeto de subclase.

El siguiente ejemplo ilustra este proceso. Utiliza la misma variable de referencia de interfaz para llamar a métodos en objetos de *DeDos* y *DeTres*.

```
//Demostración de referencia de interface
class DeDos implements Series {
    int iniciar;
    int valor;

    DeDos(){
        iniciar=0;
```

```
        valor=0;
    }


    public int getSiguiente() {
        valor+=2;
        return valor;
    }

    public void reiniciar() {
        valor=iniciar;
    }

    public void setComenzar(int x) {
        iniciar=x;
        valor=x;
    }
}
public class DeTres implements Series{
    int iniciar;
    int valor;

    DeTres(){
        iniciar=0;
        valor=0;
    }



    public int getSiguiente() {
        valor+=3;
        return valor;
    }

    public void reiniciar() {
        valor=iniciar;
    }

    public void setComenzar(int x) {
        iniciar=x;
        valor=x;
```

```
    }
}
class SeriesDemo {
    public static void main(String[] args) {
        DeDos dosOb=new DeDos();
        DeTres tresOb=new DeTres();
        Series ob;

        for (int i=0;i<5;i++) {
            ob = dosOb;
            System.out.println("Siguiente valor DeDos es: " + ob.getSiguiente());
            ob = tresOb;
            System.out.println("Siguiente valor DeTres es: " + ob.getSiguiente());
        }
    }
}
```

Salida:

```
Siguiente valor DeDos es: 2
Siguiente valor DeTres es: 3
Siguiente valor DeDos es: 4
Siguiente valor DeTres es: 6
[...]
```

En *main()*, **ob** se declara como una referencia a una interfaz de *Series*. Esto significa que se puede usar para almacenar referencias a cualquier objeto que implemente *Series*. En este caso, se utiliza para referirse a *dosOb* y *tresOb*, que son objetos de tipo *DeDos* y *DeTres*, respectivamente, que implementan *Series*.

Una variable de referencia de interfaz solo tiene conocimiento de los métodos declarados por su declaración de interfaz. Por lo tanto, ***ob*** no se podría usar para acceder a otras variables o métodos que puedan ser compatibles con el objeto.

5. Variables en interfaces

Como se mencionó, las variables se pueden declarar en una interfaz, pero son implícitamente públicas, estáticas y finales (*public*, *static*, y *final*). A primera vista, podría pensar que habría un uso muy limitado para tales variables, pero ocurre lo contrario.

Los programas grandes normalmente hacen uso de varios valores constantes que describen cosas como el tamaño de la matriz, diversos límites, valores especiales y similares. Dado que un programa grande generalmente se mantiene en una cantidad de archivos fuente separados, debe haber una forma conveniente de hacer que estas constantes estén disponibles para cada archivo. En Java, las variables de interfaz ofrecen una solución.

Para definir un conjunto de constantes compartidas, cree una interfaz que contenga solo estas constantes, sin ningún método. Cada archivo que necesita acceso a las constantes simplemente «implementa» la interfaz. Esto trae las constantes a la vista. Aquí hay un ejemplo:

```
//Una interfaz que contiene constantes
interface Constante {


    //Definiendo 3 constantes
    int MIN=0;
    int MAX=10;
    String MSJERROR="LIMITE ERROR";
}
class ConstanteD implements Constante{
    public static void main(String[] args) {
        int numeros[]=new int;


        for (int i=MIN; i<11; i++){
            if (i>=MAX) System.out.println(MSJERROR);
            else {
                numeros=i;
                System.out.println(numeros+ " ");
            }
        }
    }
}
```

La técnica de usar una interfaz para definir constantes compartidas es controvertida. Veremos más ejemplos más adelante.

6. Las interfaces pueden ser extendidas

Una interfaz puede heredar otra mediante el uso de la palabra clave **extends**. La sintaxis es la misma que para heredar clases. Cuando una clase implementa una interfaz que hereda otra interfaz, debe proporcionar implementaciones para todos los métodos requeridos por la cadena de herencia de la interfaz. Lo siguiente es un ejemplo:

```
//Una interface puede extender de otra
interface A{
    void metodo1();
    void metodo2();
}
//B ahora incluye metodo1() y metodo2() - y añade metodo3()
interface B extends A{
```

```
    void metodo3();
}
//Esta clase debe implementar los métodos de A y B
class MiClase implements B{

    public void metodo1() {
        System.out.println("Implementación de metodo1().");
    }

    public void metodo2() {
        System.out.println("Implementación de metodo2().");
    }

    public void metodo3() {
        System.out.println("Implementación de metodo3().");
    }
}
public class Extender {
    public static void main(String[] args) {
        MiClase mc=new MiClase();

        mc.metodo1();
        mc.metodo2();
        mc.metodo3();
    }
}
```

Salida:

```
Implementación de metodo1().
Implementación de metodo2().
Implementación de metodo3().
```

Como experimento, puede intentar eliminar la implementación de *metodo1()* en *MiClase*. Esto causará un error en tiempo de compilación. Como se dijo anteriormente, cualquier clase que implemente una interfaz debe implementar todos los métodos requeridos por esa interfaz, **incluidos los heredados de otras interfaces**.