UNIVERSIDAD TÉCNICA PARTICULAR DE LOJA

COMPUTACIÓN

MATERIA

TALLER

Integrantes: María Alejandra Paute

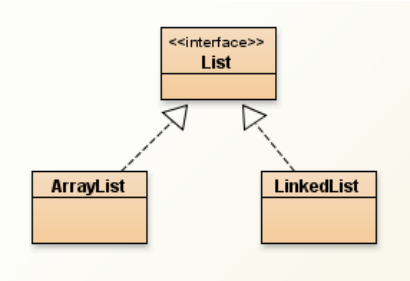
Fecha: 15 de julio de 2021

Profesor: Ing. René Elizalde

abril 2021 – agosto 2021

Uso de Interfaces en Java

Una interfaz en Java es una colección de métodos abstractos y propiedades constantes.



En las interfaces se especifica qué se debe hacer pero no su implementación. Serán las clases que implementen estas interfaces las que describen la lógica del comportamiento de los métodos.

La principal diferencia entre interface y abstract es que un interface proporciona un mecanismo de encapsulación de los protocolos de los métodos sin forzar al usuario a utilizar la herencia.

Ventajas

- Organizar el código.
- Permite declarar constantes que van a estar disponibles para todas las clases que queramos.
- Obligar a que ciertas clases utilicen los mismos métodos.
- Establecer relaciones entre clases que no estén relacionadas.

Uso

Java proporciona dos palabras reservadas para trabajar con interfaces: **interface** e **implements**.

Para declarar una interfaz se utiliza:

```
modificador_acceso interface NombreInterfaz{

...

}
```

modificador_acceso puede ser una clase de objetos que nos permite utilizar herencia en abstracción constante en las clases en las que se implementa.

Para implementar en una clase, se utiliza la forma:

```
modificador_acceso class NombreClase implements NombreInterfaz1 [, NombreInterfaz2]
```

Una clase puede implementar varias interfaces de los paquetes que se han importado dentro del programa, separando los nombres por comas.

Ejemplo:

- Definición de una interfaz.

```
interface Nave {
        public void moverPosicion (int x, int y);
        public void disparar();
        .....
    }
```

- Uso de la interfaz definida:

```
public class NaveJugador implements Nave {
        public void moverPosicion (int x, int y) {
            //Implementación del método
            posActualx = posActualx - x;
            posActualy = posActualy - y;
        }

        public void disparar() {
            //Implementación del método
        }

        ...
    }
```

Ejemplo:

```
public class Fraccion implements Relacionable {

    private int num;
    private int den;

    //Constructores
    public Fraccion() {
        this.num = 0;
        this.den = 1;
    }

    public Fraccion(int num, int den) {
        this.num = num;
        this.den = den;
        simplificar();
    }

    public Fraccion(int num) {
        this.num = num;
        this.den = 1;
    }

    //Setters y Getters
    public void setDen(int den) {
        this.den = den;
        this.simplificar();
    }

    public void setNum(int num) {
        this.num = num;
        this.simplificar();
    }

    public int getDen() {
        return den;
    }

    public int getNum() {
        return num;
    }

    //sumar fracciones
    public Fraccion sumar(Fraccion f) {
        Fraccion aux = new Fraccion();
        aux.num = num * f.den + den * f.num;
        aux.den = den * f.den;
        aux.simplificar();
        return aux;
    }
    //restar fracciones
    public Fraccion restar(Fraccion f) {
        Fraccion aux = new Fraccion();
        aux.num = num * f.den - den * f.num;
        aux.den = den * f.den;
        aux.simplificar();
        return aux;
    }
```

```
    //multiplicar fracciones
    public Fraccion multiplicar(Fraccion f) {
        Fraccion aux = new Fraccion();
        aux.num = num * f.num;
        aux.den = den * f.den;
        aux.simplificar();
        return aux;
    }

    //dividir fracciones
    public Fraccion dividir(Fraccion f) {
        Fraccion aux = new Fraccion();
        aux.num = num * f.den;
        aux.den = den * f.num;
        aux.simplificar();
        return aux;
    }

    //Método para simplificar una fracción
    private void simplificar() {
        int n = mcd(); //se calcula el mcd de la fracción
        num = num / n;
        den = den / n;
    }

    //Cálculo del máximo común divisor por el algoritmo de Euclides
    //Lo utiliza el método simplificar()
    private int mcd() {
        int u = Math.abs(num); //valor absoluto del numerador
        int v = Math.abs(den); //valor absoluto del denominador
        if (v == 0) {
            return u;
        }
        int r;
        while (v != 0) {
            r = u % v;
            u = v;
            v = r;
        }
        return u;
    }

    //Sobreescritura del método toString heredado de Object
    @Override
    public String toString() {
        simplificar();
        return num + "/" + den;
    }
```

```
    //Implementación del método abstracto esMayorQue de la interface
    @Override
    public boolean esMayorQue(Relacionable a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Fraccion)) {
            return false;
        }
        Fraccion f = (Fraccion) a;
        this.simplificar();
        f.simplificar();
        if ((num / (double) den) <= (f.num / (double) f.den)) {
            return false;
        }
        return true;
    }
   //Implementación del método abstracto esMenorQue de la interface
    @Override
    public boolean esMenorQue(Relacionable a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Fraccion)) {
            return false;
        }
        Fraccion f = (Fraccion) a;
        this.simplificar();
        f.simplificar();
        if ((num / (double) den) >= (f.num / (double) f.den)) {
            return false;
        }
        return true;
    }

    //Implementación del método abstracto esIgualQue de la interface
    @Override
    public boolean esIgualQue(Relacionable a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Fraccion)) {
            return false;
        }
        Fraccion f = (Fraccion) a;
        this.simplificar();
        f.simplificar();
        if (num != f.num) {
            return false;
        }
        if (den != f.den) {
            return false;
        }
        return true;
    }
}
```

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

        //Creamos dos fracciones y mostramos cuál es la mayor y cuál menor.
        Fraccion f1 = new Fraccion(3, 5);
        Fraccion f2 = new Fraccion(2, 8);

        if (f1.esMayorQue(f2)) {
            System.out.println(f1 + " > " + f2);
        } else {
            System.out.println(f1 + " <= " + f2);
        }

        //Creamos un ArrayList de fracciones y las mostramos ordenadas de
menor a mayor
        ArrayList<Fraccion> fracciones = new ArrayList();

        fracciones.add(new Fraccion(10, 7));
        fracciones.add(new Fraccion(-2, 3));
        fracciones.add(new Fraccion(1, 9));
        fracciones.add(new Fraccion(6, 25));
        fracciones.add(new Fraccion(3, 8));
        fracciones.add(new Fraccion(8, 3));

        Collections.sort(fracciones, new Comparator<Fraccion>(){

            @Override
            public int compare(Fraccion o1, Fraccion o2) {
                if(o1.esMayorQue(o2)){
                    return 1;
                }else if(o1.esMenorQue(o2)){
                    return -1;
                }else{
                    return 0;
                }
            }

        });

        System.out.println("Fracciones ordenadas de menor a mayor");
        for(Fraccion f: fracciones){
            System.out.print(f + " ");
        }
}
```

Una clase puede implementar más de una interface. Los nombres de las interfaces se escriben a continuación de *implements* y separadas por comas:

```
public class UnaClase implements Interface1, Interface2, Interface3{
   ......
}
```

Principio de segregación de interfaces

El principio de segregación de interfaces viene a decir que ninguna clase debería depender de métodos que no usa. Por tanto, cuando creemos interfaces que definan comportamientos, es importante estar seguros de que todas las clases que implementen esas interfaces vayan a necesitar y ser capaces de agregar comportamientos a todos los métodos. En caso contrario, es mejor tener varias interfaces más pequeñas.

Las interfaces nos ayudan a desacoplar módulos entre sí. Esto es así porque si tenemos una interfaz que explica el comportamiento que el módulo espera para comunicarse con otros módulos, nosotros siempre podremos crear una clase que lo implemente de modo que cumpla las condiciones.

El módulo que describe la interfaz no tiene que saber nada sobre nuestro código y, sin embargo, nosotros podemos trabajar con él sin problemas.

La problemática surge cuando esas interfaces intentan definir más cosas de las debidas, lo que se denominan *fat interfaces*.

Probablemente ocurrirá que las clases hijas acabarán por no usar muchos de esos métodos, y habrá que darles una implementación.

Muy habitual es lanzar una excepción, o simplemente no hacer nada.

Pero, al igual que vimos en algún ejemplo en el principio de sustitución de Liskov, esto es peligroso. Si lanzamos una excepción, es más que probable que el módulo que define esa interfaz use el método en algún momento, y esto hará fallar nuestro programa.

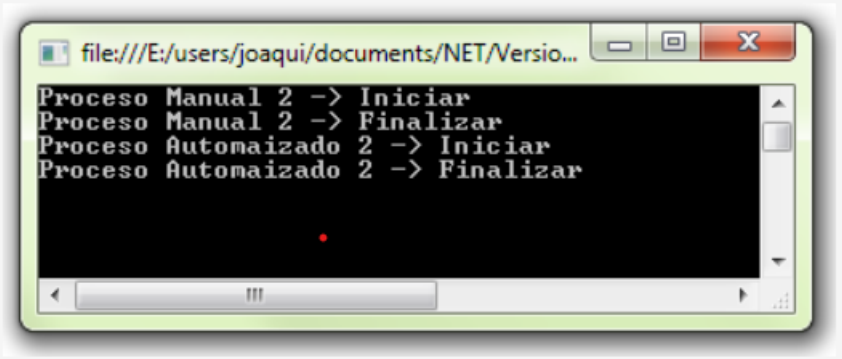
El resto de implementaciones “por defecto” que podamos dar, pueden generar efectos secundarios que no esperemos, y a los que sólo podemos responder conociendo el código fuente del módulo en cuestión, cosa que no nos interesa.

Ejemplo:

```
//---------------------------------
// Interfaz que deben cumplir los trabajos que
// se realicen de forma manual (por personas)
public interface IManual
{
    void Suspender();
    void Reiniciar();
}

//---------------------------------
// Interfaz que debe cumplir CUALQUIER TIPO de trabajo
public interface IProceso
{
    void Iniciar();
    void Finalizar();
}
```

```
//---------------------------------
// Clase que representa a un proceso realizado por personas
// Implementa la interfaz [IProceso] correspondiente a cualquier proceso
// e implementa la interfaz [IManual] procesos realizados por personas
public class ProcesoManual : IProceso, IManual
{
    public void Iniciar()   { Console.WriteLine("Proceso Manual 2 -> Iniciar");   }
    public void Suspender() { Console.WriteLine("Proceso Manual 2 -> Suspender"); }
    public void Reiniciar() { Console.WriteLine("Proceso Manual 2 -> Reiniciar"); }
    public void Finalizar() { Console.WriteLine("Proceso Manual 2 -> Finalizar"); }
}


//---------------------------------
// Clase que representa a un proceso realizado por maquinas
// Implementa la interfaz [IProceso] correspondiente a cualquier proceso
public class ProcesoAutomatizado : IProceso
{
    public void Iniciar()   { Console.WriteLine("Proceso Automatizado 2 -> Iniciar");
}
    public void Finalizar() { Console.WriteLine("Proceso Automatizado 2 -> Finalizar");
}
}
//---------------------------------
// Una clase para manejar procesos
public class GerenteProcesos
{
    // Campo de la clase que guarda la referencia a la interfaz [IProceso]
    // Este valor se tiene que proporcionar a través del constructor
    // La palabra clave readonly corresponde a un modificador que
    // se puede utilizar en campos.
    // Cuando una declaración de campo incluye un modificador readonly,
    // las asignaciones a los campos que aparecen en la declaración
    // sólo pueden tener lugar en la propia declaración o
    // en un constructor de la misma clase.
    private readonly IProceso unProceso;

    // Constructor
    public GerenteProcesos(IProceso w)
    {
        unProceso = w;
    }

    // propiedad para leer el proceso que está almacenado en la clase
    public IProceso Proceso
    {
        get
        {
            return unProceso;
        }
    }
    // Poner en ejecución el proceso
    public void Gestionar()
    {
        unProceso.Iniciar();
        //unProceso.Suspender(); // provoca un error
        //unProceso.Reiniciar(); // provoca un error
        unProceso.Finalizar();

    }
}
```

```
//---------------------------------
// Clase para probar las clases anteriores
public class Test
{
    public void TestGerente()
    {
        ProcesoManual PM1 = new ProcesoManual();
        ProcesoAutomatizado PA1 = new ProcesoAutomatizado();

        GerenteProcesos GP1 = new GerenteProcesos(PM1);
        GP1.Gestionar();

        GerenteProcesos GP2 = new GerenteProcesos(PA1);
        GP2.Gestionar();
    }
}
```



Bibliografía:

Es.wikipedia.org. 2020. *Interfaz (Java) - Wikipedia, la enciclopedia libre*. [online] Recuperado de <https://es.wikipedia.org/wiki/Interfaz_(Java)>

Walton, A., 2020. *Interfaces en Java con Ejemplos*. [online] javadesdecero. Recuperado de <https://javadesdecero.es/intermedio/interfaces-ejemplos/>

Interfaces., I., 2020. *Interfaces en Java. Ejemplos de Interfaces.*. [online] Puntocomnoesunlenguaje.blogspot.com Recuperado de <http://puntocomnoesunlenguaje.blogspot.com/2013/09/java-interfaces.html>

Guru99. 2021. *¿Qué es la interfaz en Java con un ejemplo? - Guru99*. [online] Recuperado de <https://guru99.es/java-interface/>

Leiva, A., 2019. *Principio de segregación de interfaces (SOLID 4ª parte)*. [online] DevExperto. Recuperado de <https://devexperto.com/principio-de-segregacion-de-interfaces/>

Alexandre Freire. 2020. ▷ 🎯【SOLID】 *Principio Segregación de Interfaces - Ejemplos*. [online] Recuperado de <https://alexandrefreire.com/principios-solid/segregacion-interfaces/>

Tribalyte Technologies. 2021. *PRINCIPIOS S.O.L.I.D. - INTERFACE SEGREGATION | Tribalyte Technologies*. [online] Recuperado de <https://tech.tribalyte.eu/blog-principios-solid-interface-segregation>

] Principio de segregación de interfaces. 2021. *[ISP] Principio de segregación de interfaces*. [online] Joaquin.medina.name. Recuperado de <http://joaquin.medina.name/web2008/documentos/informatica/documentacion/logica/OOP/Principios/Oop_Solid_ISP/2012_09_14_ISP_InterfaceSegregationPrincipe.html>