Interfaces en Java. Ejemplos de Interfaces.

Antes de **Java 8,** una interface era considerada como una **clase abstracta pura:** todos sus métodos son abstractos y si tiene atributos son todos constantes.

En general, podemos considerar una **interface** como una clase que sólo puede contener:

- Métodos abstractos
- Atributos constantes

A partir de **Java 8** el concepto de Interface se ha ampliado y a partir de esa versión de Java las interfaces también pueden contener:

- Métodos por defecto
- Métodos estáticos
- Tipos anidados

Y a partir de **Java 9** se les ha añadido una nueva funcionalidad a las interfaces y también pueden contener:

- Métodos privados

Una **Interface tiene en común con una clase** lo siguiente:

- Puede contener métodos.
- Se escribe en un archivo con extensión .java que debe llamarse exactamente igual que la interface.
- Al compilar el programa, el byte-code de la Interface se guarda en un archivo .class.

Una **Interface se diferencia de una clase** en lo siguiente:

- No se puede instanciar. No podemos crear objetos directamente a partir de una Interface.
- No contiene constructores.
- Si contiene atributos todos deben ser *public static final.* Lo son por defecto.
- Una interface puede heredar de varias interfaces. Con las interfaces se permite la herencia múltiple.

Conceptos importantes a tener en cuenta cuando trabajamos con interfaces:

- Las clases no heredan las interfaces. Las clases **implementan** las interfaces.
- Una clase puede implementar varias interfaces.

Cuando una clase implementa una interface está obligada a implementar todos los métodos abstractos que contiene ya que de otra forma debería declararse como clase abstracta.

Usando Interfaces, si varias clases distintas implementan la misma interface, nos aseguramos de que todas tendrán implementados una serie de métodos comunes.

Las interfaces definen un protocolo de comportamiento y proporcionan un formato común para implementarlo en las clases.

Utilizando interfaces es posible que clases no relacionadas, situadas en distintas jerarquías de clases sin relaciones de herencia, tengan comportamientos comunes.

Se crean utilizando la palabra clave **interface** en lugar de class.

```
[public] interface NombreInterface [extends Interface1, Interface2, …] {

        [métodos abstractos]
        [métodos default]
        [métodos static]
        [métodos private] (a partir de java 9)
        [tipos anidados]
        [atributos constantes]
}
```

La interface puede definirse public o sin modificador de acceso, y tiene el mismo significado que para las clases. Si tiene el modificador public el archivo .java que la contiene debe tener el mismo nombre que la interfaz.

Todos los métodos que aparecen en la interfaz son públicos (si no ponemos lo contrario, ya que a partir de la 9 se permite poner private), por lo que se puede omitir el modificador de acceso *public*.

En los métodos abstractos no es necesario escribir *abstract.*

Los métodos por defecto se especifican mediante el modificador *default*.

Los métodos estáticos se especifican mediante la palabra reservada *static*.

Todos los atributos son constantes, públicos y estáticos. Por lo tanto, se pueden omitir los modificadores *public static final* cuando se declara el atributo. Se deben inicializar en la misma instrucción de declaración.

Los nombres de las interfaces suelen acabar en *able,* aunque no es necesario: configurable, arrancable, dibujable, comparable, clonable, etc.

Ejemplo:

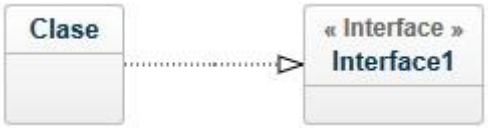
```
//Interfaz que define relaciones de orden entre objetos.
public interface Relacionable {
   boolean esMayorQue(Relacionable a);
   boolean esMenorQue(Relacionable a);
   boolean esIgualQue(Relacionable a);
}
```

IMPLEMENTACIÓN

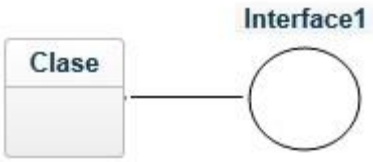
Para indicar que una clase implementa una interface se utiliza la palabra clave **implements**.

```
public class UnaClase implements Interface1{
……
}
```

En **UML** una clase que implementa una interface se representa mediante una flecha con línea discontinua apuntando a la interface:



O también se puede representar de forma abreviada:



Las clases que implementan una interfaz deben **implementar todos los métodos abstractos**. De lo contrario serán clases abstractas y deberán declararse como tal.

Ejemplo: La clase Fraccion implementa la interfaz Relacionable. En ese caso se dice que la clase **Fraccion es Relacionable**


```
public class Fraccion implements Relacionable {

   private int num;
   private int den;

   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;
   }
```

```
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;
}
```

```
//Cálculo del máximo común divisor por el algoritmo de Euclides
  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;
  }

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

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

Ejemplo de utilización de la clase Fraccion:

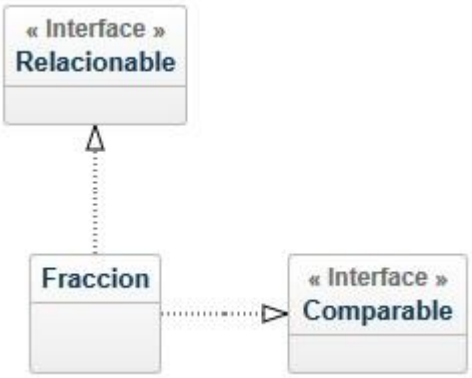
```
public static void main(String[] args) {
      //Creamos dos fracciones y mostramos cuál es la mayor y cuál menor.
      Fraccion f1 = new Fraccion(3, 5);
      Fraccion f2 = new Fraccion(2, 8);

      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));

// Con clase anónima:
       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{
…….
}
```

En el ejemplo, para ordenar las fracciones hemos utilizado un **Comparator** como parámetro de Collections.sort. También podríamos ordenarlas de otra forma haciendo que la clase Fraccion implemente la interfaz **Comparable**, además de Relacionable:



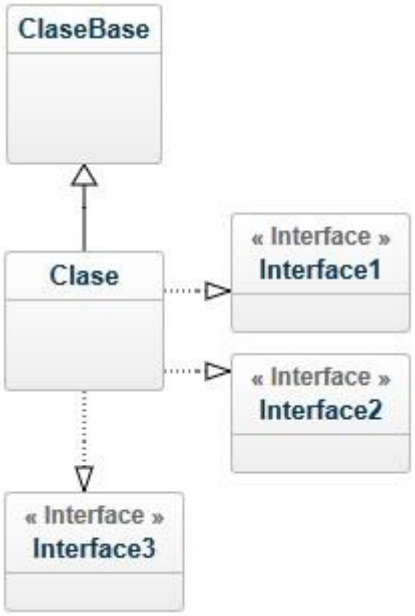

```
public class Fraccion implements Relacionable, Comparable<Fraccion> {

        ……
        //Código de la clase Fraccion
        ……

         //Añadimos a la clase el método compareTo
        @Override
        public int compareTo(Fraccion o) {
          if(this.esMenorQue(o)){
              return -1;
          }else if(this.esMayorQue(o)){
              return 1;
          }else{
              return 0;
          }
    }
} //fin de la clase Fraccion
```

De este modo para ordenar escribiríamos: Collections.sort(fracciones)

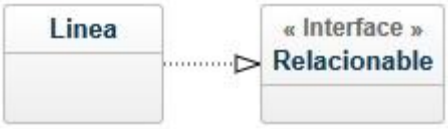
Una clase solo puede tener una clase base, pero puede implementar múltiples interfaces. El lenguaje Java no permite herencia múltiple entre clases, pero las interfaces proporcionan una alternativa para implementar algo parecido a la herencia múltiple de otros lenguajes.

Una clase que además de implementar interfaces herede de otra clase, se declarará de esta forma:

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



Una interfaz la puede implementar cualquier clase. Por ejemplo, podemos tener una clase Linea que también implementa la interfaz Relacionable:




```
public class Linea implements Relacionable {

   private double x1;
   private double y1;
   private double x2;
   private double y2;

   public Linea(double x1, double y1, double x2, double y2) {
      this.x1 = x1;
      this.y1 = y1;
      this.x2 = x2;
      this.y2 = y2;
   }
```

```
    public double longitud() {
        double l = Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
        return l;
    }


//Implementación del método abstracto de la interface
    @Override
    public boolean esMayorQue(Relacionable a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Linea)) {
            return false;
        }
        Linea linea = (Linea) a;
        return this.longitud() > linea.longitud();
    }

    //Implementación del método abstracto de la interface
    @Override
    public boolean esMenorQue(Relacionable a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Linea)) {
            return false;
        }
        Linea linea = (Linea) a;
        return this.longitud() < linea.longitud();
    }

    //Implementación del método abstracto de la interface
    @Override
    public boolean esIgualQue(Relacionable a) {
        if (a == null) {
            return false;
        }
        if (!(a instanceof Linea)) {
            return false;
        }
        Linea linea = (Linea) a;
        return this.longitud() == linea.longitud();
    }

    //Sobreescritura del método toString heredado de Object
    @Override
    public String toString() {
        StringBuilder sb = new StringBuilder();
        sb.append("Coordenadas inicio linea: ");
        sb.append(x1);
        sb.append(" , ");
        sb.append(y1);
        sb.append("\nCoordenadas final linea: ");
        sb.append(x2);
        sb.append(" , ");
        sb.append(y2);
        sb.append("\nLongitud: ");
        sb.append(longitud());
        return sb.toString();
    }
}
```

Podemos hacer un ejemplo de utilización de la clase Linea similar al que hemos hecho para la clase Fracción:

```
public static void main(String[] args) {
	Linea l1 = new Linea(2, 2, 4, 1);
	Linea l2 = new Linea(5, 2, 10, 8);
	if (l1.esMayorQue(l2)) {
	   System.out.println(l1 + "\nes mayor que" + l2);
	} else {
	   System.out.println(l1 + "\nes menor o igual que" + l2);
	}

	ArrayList<Linea> lineas = new ArrayList();

	lineas.add(new Linea(0, 7, 1, 4));
	lineas.add(new Linea(2, -1, 3, 5));
	lineas.add(new Linea(1, 9, 0, -3));
	lineas.add(new Linea(15, 3, 9, 5));
	Collections.sort(lineas, new Comparator<Linea>(){

	   @Override
	   public int compare(Linea o1, Linea o2) {
	      if(o1.esMayorQue(o2)){
	         return 1;
	      }else if(o1.esMenorQue(o2)){
	         return -1;
	      }else{
	         return 0;
	      }
	   }
	});
	System.out.println("\nLineas ordenadas por longitud de menor a mayor");
	for (Linea l : lineas) {
	   System.out.println(l);
	}
}
```

INTERFACES Y POLIMORFISMO

La definición de una interface implica una definición de un nuevo tipo de referencia y por ello **se puede usar el nombre de la interface como nombre de tipo**.

El nombre de una interfaz se puede utilizar en cualquier lugar donde pueda aparecer el nombre de un tipo de datos.

Si se define una variable cuyo tipo es una interface, se le puede asignar un objeto instancia de una clase que implementa la interface.

Volviendo al ejemplo, las clases Linea y Fraccion implementan la interfaz Relacionable. Podemos escribir las instrucciones:

Relacionable r1 = new Linea(2,2,4,1);

Relacionable r2 = new Fraccion(4,7);

En Java, utilizando interfaces como tipo se puede aplicar el polimorfismo para clases que no están relacionadas por herencia.

Por ejemplo, podemos escribir:

System.out.println(r1); //ejecuta toString de Linea

System.out.println(r2); //ejecuta toString de Fraccion

También podemos crear un array de tipo Relacionable y guardar objetos de clases que implementan la interfaz.

```
ArrayList<Relacionable> array = new ArrayList();
array.add(new Linea(15, 3, 9, 5));
array.add(new Fraccion(10, 7));
array.add(new Fraccion(6, 25));
array.add(new Linea(3, 4, 10, 15));
array.add(new Fraccion(8, 3));
array.add(new Linea(0, 7, 1, 4));
array.add(new Linea(2, -1, 3, 5));
array.add(new Fraccion(1, 9));
array.add(new Linea(1, 9, 0, -3));
array.add(new Fraccion(3, 8));
array.add(new Fraccion(-2, 3));

for (Relacionable r : array) {
   System.out.println(r);
}
```

En este caso dos clases no relacionadas, *Linea* y *Fraccion*, por implementar la misma interfaz *Relacionable* podemos manejarlas a través de referencias a la interfaz y aplicar polimorfismo.

MÉTODOS DEFAULT

A partir de **Java 8** las interfaces además de métodos abstractos pueden contener **métodos por defecto o métodos default**.

En la interfaz se escribe el código del método. Este método estará disponible para todas las clases que la implementen, no estando obligadas a rescribir su código. **Solo lo incluirán en el caso de querer modificarlo.**

De este modo, si se modifica una interfaz añadiéndole una nueva funcionalidad, se evita tener que modificar el código en todas las clases que la implementan.

Ejemplo: Vamos a añadir un nuevo método a la interfaz Relacionable que devuelva el nombre de la clase (String) del objeto que la está utilizando. Si lo añadimos como abstracto, tendremos que modificar las clases Linea y Fraccion y añadir en cada una el nuevo método. En lugar de esto vamos a crear el método como default

y de este modo las clases Linea y Fraccion lo pueden usar sin necesidad de escribirlo.

```
public interface Relacionable {
   boolean esMayorQue(Relacionable a);
   boolean esMenorQue(Relacionable a);
   boolean esIgualQue(Relacionable a);

   default String nombreClase(){ //método por defecto
      String clase = getClass().toString();
      int posicion = clase.lastIndexOf(".");

      return clase.substring(posicion+1);
   }
}
```

Ejemplo de uso:

```
      ArrayList<Relacionable> array = new ArrayList();

      array.add(new Linea(15, 3, 9, 5));
      array.add(new Fraccion(10, 7));
      array.add(new Fraccion(6, 25));
      array.add(new Linea(3, 4, 10, 15));
      array.add(new Fraccion(8, 3));
      array.add(new Linea(0, 7, 1, 4));
      array.add(new Linea(2, -1, 3, 5));
      array.add(new Fraccion(1, 9));
      array.add(new Linea(1, 9, 0, -3));
      array.add(new Fraccion(3, 8));
      array.add(new Fraccion(-2, 3));

      for (Relacionable r: array) {
         System.out.println(r.nombreClase()); //usamos el método por defecto
         System.out.println(r);
         System.out.println();
      }
```

MÉTODOS STATIC

A partir de **Java 8** las interfaces también **pueden contener métodos static**. En la interfaz se escribe el código del método. Los métodos static de una interfaz **NO pueden ser redefinidos en las clases que la implementan**.

Para utilizarlos se escribe:

nombreInterface.metodoStatic

Ejemplo: Añadimos a la interfaz Relacionable un método estático *esNull*. El método comprueba si la variable *a* contiene o no la dirección de un objeto.

```
interface Relacionable {

   boolean esMayorQue(Relacionable a);
   boolean esMenorQue(Relacionable a);
   boolean esIgualQue(Relacionable a);

   default String nombreClase(){
      String clase = getClass().toString();
      int posicion = clase.lastIndexOf(".");
      return clase.substring(posicion+1);
   }

   static boolean esNull(Relacionable a){
      return a == null;
   }
}
```

Ejemplo de uso de un método static en una interfaz Java: supongamos que disponemos de un Array en el que algunos elementos pueden ser null, es decir, no se les ha asignado un objeto. Utilizaremos el método *esNull* para comprobarlo y evitar que se produzca un error al intentar acceder a uno de estos elementos.

```
      Relacionable[] array = new Relacionable[20];

      array[1] = new Linea(15, 3, 9, 5);
      array[5] = new Fraccion(10, 7);
      array[9] = new Fraccion(6, 25);
      array[11] = new Linea(3, 4, 10, 15);
      array[14] = new Fraccion(8, 3);
      array[15] = new Linea(0, 7, 1, 4);
      array[18] = new Linea(2, -1, 3, 5);


      for (Relacionable r : array) {
         if (!Relacionable.esNull(r)) { //usamos el método static
            System.out.println(r.nombreClase());
            System.out.println(r);
            System.out.println();
         }
      }
```

MÉTODOS PRIVADOS

A partir de **Java 9** las interfaces también **pueden contener métodos privados**.

Estos métodos **solo son accesibles dentro de la propia interface**, por lo tanto, **las clases que la implementen no podrán utilizarlos**.

Estos métodos se han incorporado para evitar la duplicidad de código en los métodos no abstractos de la interface. Si hay métodos no abstractos dentro de la interface que tienen una serie de líneas de código iguales, se puede crear un método privado que realice estas instrucciones e invocarlo y de esta forma se evita tener el mismo código repetido en varios métodos.

ATRIBUTOS CONSTANTES

Una interfaz también puede contener atributos constantes (public static final por defecto). **Esto es anterior a la versión 8**.

Ejemplo:

```
//Interfaz que contiene días relevantes en una aplicación de banca.

public interface IDiasOperaciones {
   int DIA_PAGO_INTERESES = 5;
   int DIA_COBRO_HIPOTECA = 30;
   int DIA_COBRO_TARJETA = 28;
}
```

Una clase que la implemente puede usar las constantes como si fuesen propias:

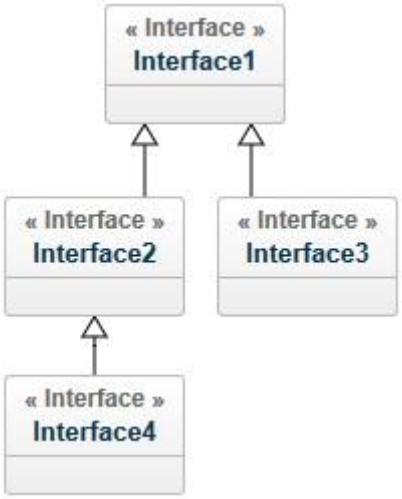
```
public class Banco implements IDiasOperaciones{
   ……..
   public void mostrarInformacionIntereses(){
      System.out.println("El día " + DIA_PAGO_INTERESES
                         + " de cada mes se realiza el pago de intereses");
   }
   ……..
}
```

Una clase que no implemente la interfaz puede usar las constantes escribiendo antes el nombre de la interfaz.

```
public static void main(String[] args) {
      ……
      System.out.println("Los intereses se pagan el día "
                         + IDiasOperaciones.DIA_PAGO_INTERESES);
      System.out.println("La hipoteca se paga el día "
                        + IDiasOperaciones.DIA_COBRO_HIPOTECA);
      ……
}
```

HERENCIA ENTRE INTERFACES

Se puede establecer una jerarquía de herencia entre interfaces igual que con las clases.

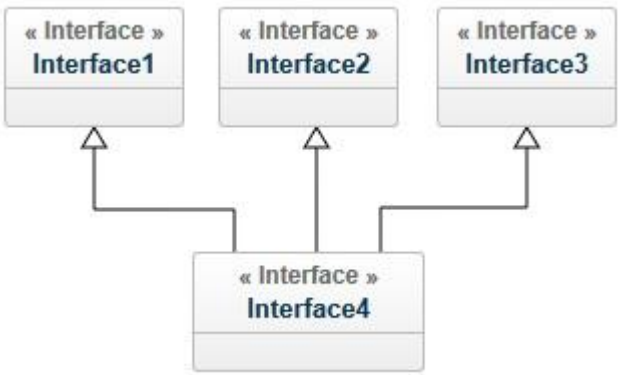



```
public interface Interface2 extends Interface1{
…….
}
public interface Interface3 extends Interface1{
…….
}
public interface Interface4 extends Interface2{
…….
}
```

Cada interface **hereda** el contenido de las interfaces que están por encima de ella en la jerarquía y puede añadir nuevo contenido o modificar lo que ha heredado siempre que sea posible.

- Los métodos static no se pueden redefinir.
- Los métodos abstract heredados se pueden convertir en métodos default.
- Los métodos default se pueden redefinir o convertir en abstract.

En las interfaces sí se permite herencia múltiple:



```
public interface Interface4 extends Interface1, Interface2, Interface3 {
    …….
}
```

Interface4 hereda todo el contenido de las tres interfaces.

Extraído de:

https://puntocomnoesunlenguaje.blogspot.com/2013/09/java-interfaces.html

Para más información:

http://aitorrm.github.io/java%208/java_8_interfaces/

https://javamagician.com/java-metodos-static-default/