Tema 8 Interfaces

1. CONCEPTO DE INTERFACE

En el tema anterior vimos los conceptos de métodos abstractos y clases abstractas. Una clase que contenga un método abstracto se debe declarar como abstracta y vimos que aunque se declaren como abstractas, pueden tener atributos, constructores y métodos no abstractos. De una clase abstracta no se pueden crear objetos y su finalidad es ser la base para construir una jerarquía de herencia entre clases y aplicar el polimorfismo.

Llevando estos conceptos un poco más allá apareció en concepto de Interface. Podemos considerar una **interface** como una clase que sólo puede contener:

a) Métodos abstractos
b) Atributos constantes

Este era el concepto inicial de interface. 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:

c) Métodos por defecto
d) Métodos estáticos
e) Tipos anidados

Y a partir de Java 9 se le ha añadido una nueva funcionalidad y también pueden contener:

f) 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

Pero se diferencia de una clase en lo siguiente:

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

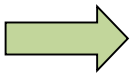
Conceptos importantes a tener en cuenta:

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

Explicándolo de una manera simple, implementar una interface sería equivalente a copiar y pegar el código de la interface dentro de la clase. 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 que todas tendrán implementados una serie de métodos en comunes.

Las interfaces juegan un papel fundamental en la creación de aplicaciones Java.



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.

Una interface se crea utilizando la palabra clave **interface** en lugar de class.

```
[public] interface NombreInterface [extends Interface1, Interface2, …]{
            [métodos abstractos]
            [métodos default]
            [métodos static]
            [métodos privados]
            [atributos constantes]
            [tipos anidados]
}
```

Algunas características:

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.

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*.

Los métodos privados se especifican mediante el modificador de acceso *private*.

Todos los atributos son constates 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 interface suelen acabar en *able* aunque no es necesario: Configurable, Arrancable, Dibujable, Comparable, Clonable, etc.

Ejemplo de interface:

Creamos una interface llamada *Relacionable* con tres métodos abstractos.

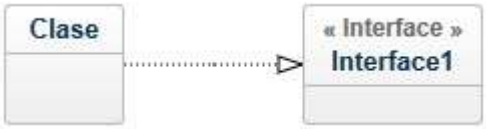
```
//Interface que define relaciones de orden entre objetos.
public interface Relacionable {
    boolean esMayorQue(Relacionable a);
    boolean esMenorQue(Relacionable a);
    boolean esIgualQue(Relacionable a);
}
```

2. IMPLEMENTACIÓN DE INTERFACES

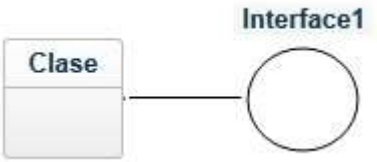
Para indicar que una clase implementa una interface se utiliza la palabra clave **implements**.

```
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:



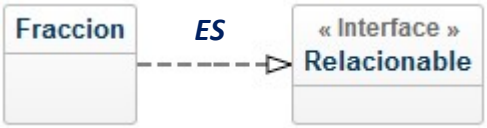
O también se puede representar de forma abreviada:



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

Cuando una clase implementa una interface se estable una relación ***es*** entre la clase y la inerface.

Por ejemplo, supongamos una clase *Fracción* que implementa la interface *Relacionable*:



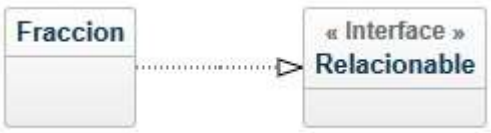
En ese caso se dice que la clase **Fraccion es Relacionable**

Ejemplo: Vamos a escribir la clase *Fracción* que implementa la interface *Relacionable*.

Los atributos de la clase son dos enteros num y den que contienen el valor del numerador y denominador de la fracción.

Además de los constructores, métodos get/set y toString, la clase contiene métodos para realizar la suma, resta, multiplicación y división de fracciones.

En la clase se deben implementar todos los métodos abstractos de la interface Relacionable.




```
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;
    }


    @Override
    public String toString() {
        simplificar();
        return num + "/" + den;
    }



   @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;
    }



    @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;
    }
```

Sobreescritura del método toString heredado de Object

Implementación del método abstracto de la interface

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;
    }
}
```

Implementación del método abstracto de la interface

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));

        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 anterior para ordenar las fracciones hemos utilizado un Comparator como parámetro de Collections.sort. También podríamos ordenarlas haciendo que la clase Fraccion implemente además la interfaz Comparable:

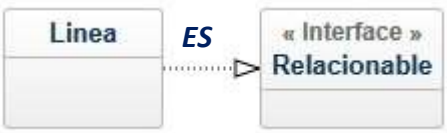
```
public class Fraccion implements Relacionable, Comparable<Fraccion> {
        ……
         //Código de la clase Fraccion
        ……

        //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 las fracciones de menor a mayor escribimos solamente:

```
Collections.sort(fracciones)
```

Una interface la puede implementar cualquier clase.

Por ejemplo, podemos tener una clase *Linea* que también implementa la interfaz *Relacionable*:



Ejemplo: Vamos a escribir la clase *Linea* que implementa la interface *Relacionable*.

Los atributos de la clase son cuatro enteros x1, y1, x2, y2 que contienen el valor de las coordenadas x e y de inicio y final de la línea. x1, y1 son las coordenadas donde empieza la línea. x2, y2 son las coordenadas del punto donde acaba la línea.

La clase contiene un conctructor, los métodos get/set, el método toString y un método para calcular la longitud de la línea.

En la clase se deben implementar todos los métodos abstractos de la interface 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 getX1() {
        return x1;
    }
    public void setX1(double x1) {
        this.x1 = x1;
    }
    public double getY1() {
        return y1;
    }
    public void setY1(double y1) {
        this.y1 = y1;
    }
    public double getX2() {
        return x2;
    }
    public void setX2(double x2) {
        this.x2 = x2;
    }
    public double getY2() {
        return y2;
    }
    public void setY2(double y2) {
        this.y2 = y2;
    }
    public double longitud() {
        return Math.sqrt((x2 - x1) * (x2 - x1) + (y2 - y1) * (y2 - y1));
    }
    @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();
    }

    @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

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();
    }
    @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();
    }
}
```

Implementación del método abstracto de la interface

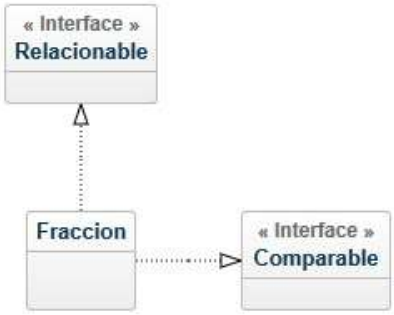
Sobreescritura del método toString heredado de Object

Ejemplo de utilización de la clase Linea:

```
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);
        }
}
```

3. HERENCIA ENTRE CLASES E IMPLEMENTACIÓN DE INTERFACES

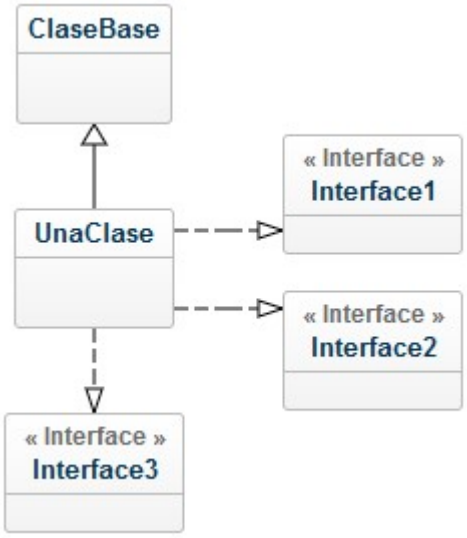
Como hemos visto en el ejemplo de la clase Fraccion, una clase puede implementar tantas interfaces como necesite.



Sabemos que Java NO permite herencia múltiple entre clases pero las interfaces proporcionan una alternativa para implementar algo parecido a la herencia múltiple de otros lenguajes.

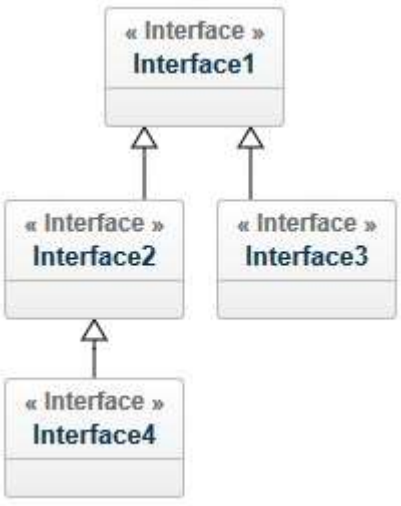
Una clase que además de implementar interfaces herede de otra se declara de esta forma:

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



4. HERENCIA ENTRE INTERFACES

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

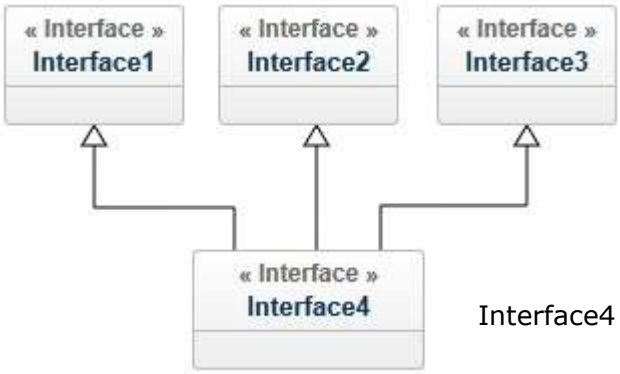



```
public interface Interface2 extends Interface1{
……
}

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.
Una interface que hereda de otra 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 SI se permite **herencia múltiple**:



Interface4 hereda todo el contenido de las tres interfaces.

```
public interface Interface4 extends Interface1, Interface2, Interface3{

    ……
}
```

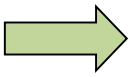
En Java todas las clases heredan de la clase Object pero **las interfaces no heredan de Object**.

Las interfaces en Java NO tienen una clase superior de la que heredan.

5. 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 tipo para declarar variables**.

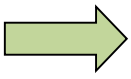
El nombre de una interfaz se puede utilizar en cualquier lugar donde pueda aparecer el nombre de un tipo de datos.



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 a los ejemplos anteriores, como 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);
```



Utilizando interfaces como tipo de datos 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 o ArrayLIst de tipo Relacionable y guardar objetos de clases que implementan la interfaz.

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

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

        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);
        }
}
```

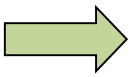
En este caso dos clases no relacionadas, *Linea* y *Fraccion*, por implementar la misma interfaz *Relacionable* las podemos manejar a través de referencias a la interfaz y aplicar polimorfismo.

En el tema anterior vimos que si manejamos objetos mediante una referencia de la clase base solo tenemos acceso a los métodos que se encuentran en la clase base.

Con las interfaces sucede lo mismo. Cuando manejamos objetos mediante una referencia a la interface solo tenemos acceso a los métodos que se encuentran en la interface.

Entonces… ¿cómo es posible que la instrucción `System.out.println(r);` ejecute por polimorfismo el método toString() de la clase Linea o el de la clase Fraccion si en la interface Relacionable no está el método toString()?

La respuesta a esto es que el método toString realmente sí está declarado en la interface.



Dentro de cada interface se declara implícitamente un método abstracto por cada método público de la clase Object. Métodos de la clase Object como *equals* y *toString* están declarados como abstractos de forma implícita en la interface.

6. 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 escribir su código. Solo lo incluirán en el caso de querer modificarlo.

Utilizando métodos default, si se modifica una interfaz añadiéndole una nueva funcionalidad (un nuevo método), 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 un String que contenga el nombre de la clase del objeto que la está utilizando.

Si este método lo añadimos como abstracto entonces tendremos obligatoriamente que modificar las clases Linea y Fraccion y añadir en cada una la implementación del método nuevo método abstracto heredado. Para evitar esto vamos a crear el método como *default* y de este modo las clases que implementan la interface lo pueden usar sin necesidad de modificar su código.

```
public interface Relacionable {
    boolean esMayorQue(Relacionable a);
    boolean esMenorQue(Relacionable a);
    boolean esIgualQue(Relacionable a);
    default String nombreClase(){ //método por defecto
        String clase = this.getClass().toString();
        int posicion = clase.lastIndexOf(".");
        return clase.substring(posicion+1);
    }
}
```

La instrucción `this.getClass().toString();` devuelve un String que contiene el nombre de la clase del objeto que ha invocado al método al que hace referencia this. Recuerda que this contiene la dirección de memoria del objeto que ha invocado al método. Si por ejemplo this hace referencia a una Fracción el método getClasss() devuelve:

```
class ejemplointerface.Linea
```

El nombre de la clase aparece al final después del punto. Lo que aparece antes del punto es el nombre del paquete donde se encuentra la clase.

Mediante los métodos *indexOf* y *substring* obtenemos un String que contiene solamente el nombre de la clase.

Ejemplo de uso:

```
public static void main(String[] args) {
       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));
       for (Relacionable r : array) {
            System.out.println(r.nombreClase()); //usamos el método por defecto
            System.out.println(r);
            System.out.println();
       }
}
```

Salida por pantalla:

```
Linea
Coordenadas inicio linea: 15.0 , 3.0
Coordenadas final linea: 9.0 , 5.0
Longitud: 6.324555320336759

Fraccion
10/7

Fraccion
6/25

Linea
Coordenadas inicio linea: 3.0 , 4.0
Coordenadas final linea: 10.0 , 15.0
Longitud: 13.038404810405298

Fraccion
8/3
```

Resultado de la instrucción `System.out.println(r.nombreClase());`

Resultado de la instrucción `System.out.println(r);`

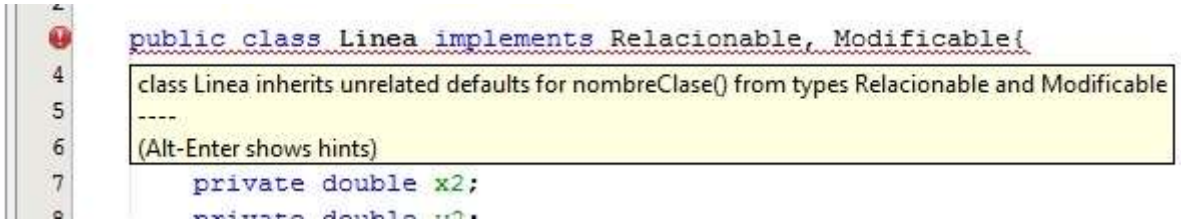
Se aplica polimorfismo y se ejecuta el método toString() de la clase correspondiente. En este caso el toString() de la clase Linea

Puede darse el caso de una clase que implemente varias interfaces y las interfaces contengan métodos default iguales (mismo nombre y lista de parámetros).

Ejemplo: La interface *Modificable* contiene dos métodos abstractos y un método default *nombreClase* que es igual que el contenido en la interface Relacionable:

```
public interface Modificable{

    void aumentar(int n);
    void disminuir(int n);

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

Supongamos que la clase Linea implementa las dos interfaces. En la clase Linea tendremos que escribir el código de los dos métodos abstractos, pero el método default provoca un error de compilación:



Al implementar las dos interfaces la clase Linea se encuentra con dos métodos default iguales. Esto provoca un *conflicto* que tenemos que resolver.

Lo podemos resolver de dos formas. La primera forma es redefinir el método nombreClase() en la clase Linea:

```
public class Linea implements Relacionable, Modificable{
      ……
       //Código de la clase Linea
      ……
    @Override
    public void aumentar(int n) {
        this.x1+=n;
        this.y1+=n;
        this.x2+=n;
        this.y2+=n;
    }

    @Override
    public void disminuir(int n) {
        this.x1-=n;
        this.y1-=n;
        this.x2-=n;
        this.y2-=n;
    }

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

}//Final de la clase Linea
```

Implementación del método abstracto de la interface Modificable

Implementación del método abstracto de la interface Modificable

Redefinir el método default

En este caso no se ha modificado el código del método aunque podríamos haberlo hecho. Nos hemos limitado a copiar y pegar el código cambiando el modificador *default* por *public*.

Para estos casos en los que no es necesario cambiar el método podemos usar la segunda forma de resolver el conflicto: indicar cuál de los dos métodos se tiene que ejecutar.

```
public class Linea implements Relacionable, Modificable{
          ……
          //Código de la clase Linea
          ……

    @Override
    public String nombreClase(){
        return Relacionable.super.nombreClase();
    }
}
```

Redefinir el método default.
Se ejecutará el método de la interface Relacionable

La forma de indicar en una clase el método de la interface que se debe ejecutar es:

`nombreInterface.super.metodo`

7. 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 **no pueden ser redefinidos** en las clases que la implementan.

Para utilizar en una clase un método static de una interface se escribe:

`nombreInterface.metodoStatic`

Ejemplo: Añadimos a la interfaz Relacionable un método estático *esNull*. El método comprueba si la variable *a* que recibe como parámetro 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 = this.getClass().toString();
        int posicion = clase.lastIndexOf(".");
        return clase.substring(posicion+1);
    }

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

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

```
public static void main(String[] args) {
        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 de la interface
                System.out.println(r.nombreClase());
                System.out.println(r);
                System.out.println();
            }
        }
```

8. MÉTODOS PRIVATE

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

Los métodos privados en una interface no pueden ser declarados como abstractos.

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

Los métodos privados se han incorporado para evitar la duplicidad de código en los métodos no abstractos de las interfaces. 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.

Ejemplo: Tenemos una interface llamada *Sumable* que contiene distintos métodos para sumar los elementos de un array de enteros. Contiene un método para sumar los números pares del array, otro para sumar los impares, otro para sumar los positivos y otro para sumar los negativos. El código que hay que escribir en cada uno es casi idéntico, hay que hacer un bucle para recorrer el array y si el elemento cumple una determinada condición se suma. Para no repetir el código en cada método se puede hacer un método privado que realice estas instrucciones.

```
public interface Sumable {

    public default int sumarPares(int[] A) {
        return sumar(1, A);
    }

    public default int sumarImpares(int[] A) {
        return sumar(2, A);
    }

    public default int sumarPositivos(int[] A) {
        return sumar(3, A);
    }

    public default int sumarNegativos(int[] A) {
        return sumar(4, A);
    }

    private int sumar(int n, int[] A) { //n contiene el tipo de elemento a sumar
        int suma = 0;
        for (int i = 0; i < A.length; i++) {
            if (n == 1 && A[i] % 2 == 0) { //suma solo los pares
                suma += A[i];
            } else if (n == 2 && A[i] % 2 != 0) { //suma solo los impares
                suma += A[i];
            } else if(n == 3 && A[i] > 0){ //suma solo los positivos
                suma += A[i];
            } else if(n == 4 && A[i] < 0){ //suma solo los negativos
                suma += A[i];
            }
        }
        return suma;
    }
}
```

9. ATRIBUTOS EN UNA INTERFACE

Una interface también puede contener atributos.

Todos los atributos contenidos en un interface son públicos estáticos y constantes.

No es necesario escribir *public static final* delante de los atributos ya que todos son de este tipo.

Al ser constantes se deben inicializar en la misma instrucción de declaración.

Ejemplo: Interface que contiene tres atributos correspondientes a 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;
}
```

Los tres atributos son public static final pero se omite porque todos los atributos dentro de una interface son de ese tipo, aunque podríamos escribirlo:

```
public static final int DIA_PAGO_INTERESES = 5;
```

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

```
public class Banco implements IDiasOperaciones{
    ......
    public void mostrarInformacionIntereses(){
        System.out.println("Día de cobro de la tarjeta: " + DIA_COBRO_TARJETA);

    }
    ......
}
```

Uso de la constante dentro de una clase que implementa la interface

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

```
public class ProyectoBanco{

 public static void main(String[] args) {
   ......
    System.out.println("Día de pago de intereses: " + IDiasOperaciones.DIA_PAGO_INTERESES);
    System.out.println("Día de cobro de la hipoteca: " + IDiasOperaciones.DIA_COBRO_HIPOTECA);
   ......
 }
}
```

Uso de las constantes dentro de una clase que no implementa la interface