INTERFACES

https://manuais.iessanclemente.net/index.php/Interfaces

Como ves un interfaz define "un comportamiento" y esto tiene varios usos como por ejemplo "posibilitar una suerte de herencia múltiple", pero ahora es importante ver un interfaz con la siguiente idea:

"es el protocolo de comunicación que deben presentar todas las clases que implementen esta interface".

Lo anterior a veces lo encontramos expresado como que una interfaz es "un contrato" en el se que indica lo que tiene que cumplir el que la implementa.

DEFINICIÓN DE UNA INTERFACE

Ejemplo: una interfaz para generar una serie de números. El diseñador detecta que vamos a tener que desarrollar clases que generan series de números y que todas esas clases deben cumplir lo siguiente:

```
interface Serie{
        int obtenerSiguiente(); //devuelve el siguiente número de la serie,
        void restablecer(); //reinicia
        void establecerInicio(int x); //establece el valor inicial
}
```

Por el momento observamos:

- en lugar de la palabra class estoy utilizando la palabra interface
- el acceso a los métodos de interfaces siempre debe ser public por lo que por defecto, aunque no se especifique, siempre es public
- En principio todos los métodos de un interface son abstractos pero no es necesario utilizar la palabra reservada *abstract* para indicarlo puesto que se presupone. Desde java 8 esto tiene dos excepciones: los métodos default y los métodos static que tienen implementación (y por tanto no son abstractos)

IMPLEMENTACIÓN DE INTERFACES

Una vez definido un interface una o más clases pueden implementarlo(implements)

Ejemplo: una clase *MasDos* que implementa el interface *serie.*

```
interface Serie{
   int obtenerSiguiente(); //devuelve el siguiente número de la serie,
   void restablecer(); //reinicia
   void establecerInicio(int x); //establece el valor inicial
}
class MasDos implements Serie{
   int inicio;
   int val;
   MasDos(){
      inicio=0;
      val=0;
```

```
    }
    public int obtenerSiguiente(){
        val +=2;
        return val;
    }
    public void restablecer(){
        inicio=0;
        val=0;
    }
    public void establecerInicio(int x){
        inicio=x;
        val=x;
    }
}
class App{
    public static void main(String[] args) {
        MasDos ob = new MasDos();
        for(int i=0;i<5;i++)
            System.out.println("el siguiente valor es: "+ ob.obtenerSiguiente());
        System.out.println("restableciendo ...");
        ob.restablecer();
        for(int i=0;i<5;i++)
            System.out.println("el siguiente valor es: "+ ob.obtenerSiguiente());
        System.out.println("empezando en 100 ...");
        ob.establecerInicio(100);
        for(int i=0;i<5;i++)
            System.out.println("el siguiente valor es: "+ ob.obtenerSiguiente());
    }
}
```

Comprueba con el código anterior que:

- si quiero declarar el interface como public tengo que escribirlo en otro fichero. Recuerda que un fichero .java sólo puede contener una clase/interface public, y que si existe una clase o interface public su nombre debe coincidir con el nombre del fichero.

- Recuerda que en un interface un método siempre es público y que en su declaración se ponga o no public un método siempre es public. Pero ojo, en su implementación también tiene que ser public pero hay poner el public explícitamente de forma obligatoria. Observa por ejemplo el error al retirar el acceso public a obtenerSiguiente().

    - De la misma manera para los métodos sin implementación podemos usar la palabra abstract pero no es necesario ya que se asume.

Por lo tanto se pudo haber escrito el interface Serie de forma equivalente de la siguiente manera

```
interface Serie{
    public abstract int obtenerSiguiente(); //devuelve el siguiente número de la serie,
    public abstract void restablecer(); //reinicia
    public abstract void establecerInicio(int x); //establece el valor inicial
}
```

Los programadores java no suelen escribir el *public* y el *abstract* en los interfaces

UN INTERFACE PUEDE SER IMPLEMENTADO POR VARIAS CLASES

Uno de los principios de diseño orientado a objetos es "un interface, muchas implementaciones" que no es más que un caso concreto de polimorfismo que analizaremos más adelante. El usuario de un conjunto de clases que comparten una interface sólo se tiene que preocupar de cómo es esa interfaz, puede haber cientos de clases que implementan dicha interfaz y todas tendrán un manejo similar. ¿Te parece poco?. Esto es muy importante y valorarás su importancia por ejemplo cuando estudiemos "colecciones".

Ejercicio U5_B5A_E1: Modifica el ejemplo anterior de forma que:

El main es ahora:

```
class App{
   public static void main(String[] args) {
      MasDos ob2 = new MasDos();
      MasTres ob3 = new MasTres();


      System.out.println("De dos en dos ...");
      for(int i=0;i<5;i++)
           System.out.println("el siguiente valor es: "+ ob2.obtenerSiguiente());


      System.out.println("De tres en tres ...");
      for(int i=0;i<5;i++)
          System.out.println("el siguiente valor es: "+ ob3.obtenerSiguiente());
   }
}
```

```
De dos en dos ...
el siguiente valor es: 2
el siguiente valor es: 4
el siguiente valor es: 6
el siguiente valor es: 8
el siguiente valor es: 10
De tres en tres ...
el siguiente valor es: 3
el siguiente valor es: 6
el siguiente valor es: 9
el siguiente valor es: 12
el siguiente valor es: 15
```

Y por tanto debo crear una clase MasTres que produzca la salida anterior.

Las implementaciones pueden tener miembros propios.

Supongamos que necesitamos que, por la razón que fuere la clase *MasDos* permita conocer en un momento dado si el atributo *val* que contiene el objeto es múltiplo de 10, además, esta característica no es interesante a nivel de interface. En este caso, simplemente, se define ese método para MasDos, es decir, MasDos implementa el interfaz pero puede tener además sus atributos y métodos propios.

```
public class MasDos implements Serie{
   int inicio;
   int val;
```

```
    MasDos(){
        inicio=0;
        val=0;
    }
    public int obtenerSiguiente(){
        val +=2;
        return val;
    }
    public void restablecer(){
        inicio=0;
        val=0;
    }
    public void establecerInicio(int x){
        inicio=x;
        val=x;
    }
    boolean esMultiploDe10(){
        if(val%10==0)
            return true;
        else
            return false;
    }
}
```

LAS INTERFACES PUEDEN EXTENDERSE (EXTENDS).

Similar a la herencia de clases.

Ejemplo: Observa como MiClase al implementar B tiene que forzosamente implementar met3() y los métodos "heredados" met1() y met2(). Comprueba el error si suprimes alguna implementación.

```
interface A{
    void met1();
    void met2();
}
interface B extends A {
    void met3();
}
class MiClase implements B{
    public void met1(){
        System.out.println("implementación de met1()");
    }
    public void met2(){
        System.out.println("implementación de met2()");
    }
    public void met3(){
        System.out.println("implementación de met3()");
    }
}
class App{
    public static void main(String[] args) {
    MiClase ob=new MiClase();
    ob.met1();
    ob.met2();
    ob.met3();
    }
}
```

VARIABLES EN INTERFACES

Sólo se permiten variables *public static final*. Son pues "constantes con nombre". En programas grandes es habitual ver interfaces de sólo variables.

Supongamos que tenemos un conjunto de clases que trabajan todas ellas con las mismas constantes . Para evitar que cada clase utilice sus nombres, colocamos las constantes en una interface, y las clases que quieran utilizarla implementan dicha interface.

Aunque es un poco simplón podemos decir que si una variable de clase static es una "especie de variable global a todos los elementos de la clase", si la definimos en un interface público pasa a ser una "variable global a todas las clases que implementan el interface".

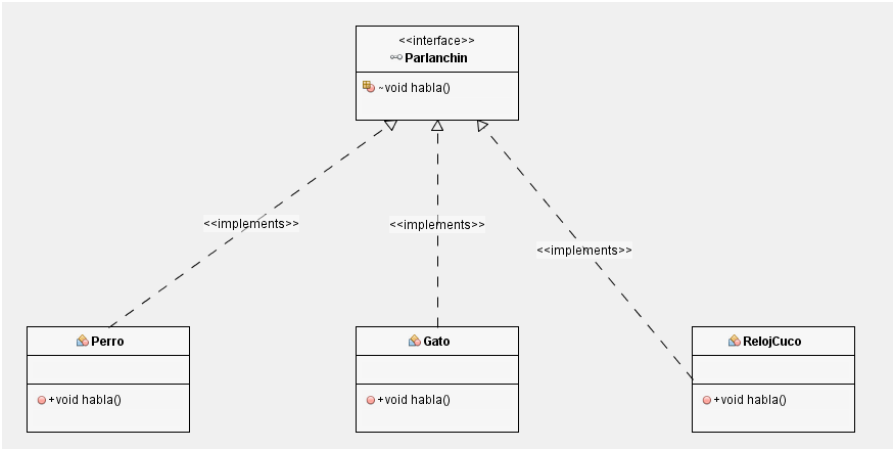
Ejemplo: Tenemos el siguiente interface.

```
interface IConstantes {
    int MIN=0;
    int MAX=10;
}
class App implements IConstantes{
    public static void main(String[] args) {
        System.out.println("Máximo: "+ MAX);
        System.out.println("Máximo: "+ App.MAX);
        System.out.println("Máximo: "+ IConstantes.MAX);
    }
}
```

MIN y MAX los podemos usar directamente asociados al nombre del interface o bien a través de una clase que lo implementa como es el caso de App. Además como main pertenece a App y es static, App.MAX se puede abreviar simplemente como MAX.

Ejercicio U5_B5A_E2:

Tenemos una serie de clases muy diferentes que implementan el mismo interface Parlanchin. Escribimos la jerarquía en el paquete *parlanchines*



El método habla() genera una descripción textual del sonido del objeto que habla. Probamos la estructura desde App

```
public class App {
    public static void main(String[] args) {
        Gato g = new Gato();
        Perro p = new Perro();
        RelojCuco rc = new RelojCuco();
        g.habla();
```

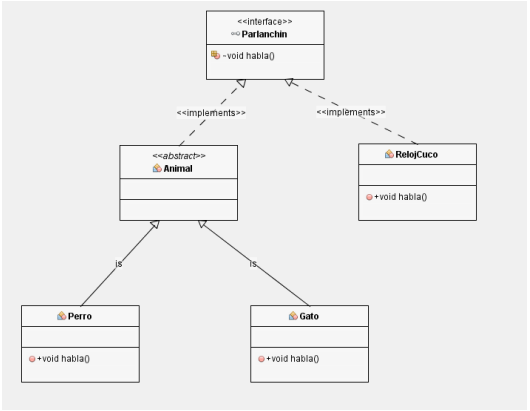
```
        p.habla();
        rc.habla();
    }
}
```
que genera la salida

```
¡Miau!
¡Guau!
¡Cucu, cucu, ..!
```

App en paquete por defecto. El resto de las clases pertenecen al paquete parlanchines. Cada clase en su .java. y se permite para simplificar que todas las clases hagan println()

Ejercicio U5_B5A_E3:

Ejercicio: ahora ampliamos el ejercicio anterior para incluir a Gato y Perro como subclase de la clase Abstracta Animal. Escribimos la jerarquía en el paquete *parlanchines*



MÉTODOS CON IMPLEMENTACIÓN EN INTERFACES:

Desde java 8, por fines prácticos se relaja la restricción de que una interface no puede tener métodos con código si son de uno de estos dos tipos:

- Métodos default
- Métodos static

LOS MÉTODOS DEFAULT.

La implementación de un método default es común para todas las clases que implementen esa interface. Esto no quita que dicha implementación pueda ser sobreescrita. Los métodos default pueden tener varios usos:

- Dar ya una implementación que se sabe que es casi siempre buena y permitir que las clases que implementen la interface no se vean obligados a sobrescribir dicho método.
- Permitir que el interface tenga un nuevo método (default) y las clases que usaban la versión anterior sigan funcionando. Si el método no es default las clases antiguas habría que tocarlas para sobreescribir el nuevo método.

Ejemplo: tenemos que la clase Calculadora que implementa ICalculadora

```
interface ICalculadora{
    public void sumar(int x, int y);
    public void restar(int x,int y);
}

class Calculadora implements ICalculadora{

    @Override
    public void sumar(int x, int y) {
        System.out.println("x + y es:"+ (x+y));
    }

    @Override
    public void restar(int x, int y) {
        System.out.println("x - y es:"+ (x-y));
    }

}
public class App{
    public static void main(String[] args) {
        Calculadora c = new Calculadora();
        c.sumar(2, 3);
        c.restar(2,3);
    }
}
```

Por la razón que sea, se quiere modificar ICalculadora agregando un método multiplicar. Si lo agregamos como default, la clase Calculadora no se ve obligada a sobreescribirlo.

```
interface ICalculadora{
    public void sumar(int x, int y);
    public void restar(int x,int y);
    default public void multiplicar(int x, int y) {
        System.out.println("x*y es:"+ (x*y));
    }

}

class Calculadora implements ICalculadora{

    @Override
    public void sumar(int x, int y) {
        System.out.println("x + y es:"+ (x+y));
    }

    @Override
    public void restar(int x, int y) {
        System.out.println("x - y es:"+ (x-y));
    }

}
public class App{
    public static void main(String[] args) {
        Calculadora c = new Calculadora();
        c.sumar(2, 3);
        c.restar(2,3);
    }
}
```

los métodos default:

- Son heredados

- Se pueden sobrescribir

Son heredados: Observa como Calculadora no tiene ningún método multiplicar pero puedo hacer
c.multiplicar(2,3);

```
interface ICalculadora{
    public void sumar(int x, int y);
    public void restar(int x,int y);
    default public void multiplicar(int x, int y) {
        System.out.println("x*y es:"+ (x*y));
    }

}

class Calculadora implements ICalculadora{

    @Override
    public void sumar(int x, int y) {
        System.out.println("x + y es:"+ (x+y));
    }

    @Override
    public void restar(int x, int y) {
        System.out.println("x - y es:"+ (x-y));
    }

}
public class App{
    public static void main(String[] args) {
        Calculadora c = new Calculadora();
        c.sumar(2, 3);
        c.restar(2,3);
        c.multiplicar(2,3);
    }
}
```

Se pueden sobrescribir

```
interface ICalculadora{
    public void sumar(int x, int y);
    public void restar(int x,int y);
    default public void multiplicar(int x, int y) {
        System.out.println("x*y es:"+ (x*y));
    }

}

class Calculadora implements ICalculadora{

    @Override
    public void sumar(int x, int y) {
        System.out.println("x + y es:"+ (x+y));
    }

    @Override
    public void restar(int x, int y) {
        System.out.println("x - y es:"+ (x-y));
    }
    public void multiplicar(int x, int y ){
        System.out.println("no sé multiplicar");
    }

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

```
        Calculadora c = new Calculadora();
        c.sumar(2, 3);
        c.restar(2,3);
        c.multiplicar(2,3);
    }
}
```

MÉTODOS STATIC EN INTERFACES

Ejemplo:

```
interface X{
    static void saludo(){
        System.out.println("hola");
    }
}


public class App{
    public static void main(String[] args)    {
        X.saludo();
          }
}
```