UTILIZACIÓN AVANZADA DE CLASES(II)

En este documento trataremos:

- Clase enum
- Clase Abstracta
- Interfaces

4. CLASE ENUM.

4.1. Uso simple – contenedor de constantes.

Las enumeraciones o enum en un lenguaje de programación sirven para representar un grupo de constantes - "contenedores de constantes".

Una enumeración brinda una manera de definir con precisión un nuevo tipo de datos que tiene un número fijo de valores válidos.

```
enum Color
{
    ROJO, VERDE, AZUL;
}

public class Test
{
    // El método
    public static void main(String[] args)
    {
        Color c1 = Color.ROJO;
        System.out.println(c1);
    }
}
```

```
public enum EnumTestSimple {

    MESA,
    SILLA,
    ARMARIO;

}
```

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

    System.out.println(EnumTestSimple.ARMARIO); // Obtener el enunciado disponible.
    System.out.println(EnumTestSimple.ARMARIO.ordinal()); // Obtener el indice.

}
```

Métodos:

	Return	Descripción
name()	String	Devuelve un String con el nombre de la constante que contiene tal y como aparece en la declaración.
ordinal()	*int*	Devuelve un entero con la posición de la constante según está declarada. A la primera constante le corresponde la posición cero.
toString()	String	Devuelve un String con el nombre de la constante que contiene tal y como aparece en la declaración.
equals()	boolean	Devuelve true si el valor de la variable enum es igual al objeto que recibe.
compareTo()	int	Compara el enum con el que recibe según el orden en el que están declaradas las constantes. Devuelve un número negativo, cero o un número positivo según el objeto sea menor, igual o mayor que el que recibe como parámetro.
valueOf()	enumConstant	Devuelve la constante que coincide exactamente con el String que recibe como parámetro.
values()	enumConstant []	Devuelve un array que contiene todas las constantes de la enumeración en el orden en que se han declarado. Se suele usar en bucles for each para recorrer el enum.

4.2. Uso más completo – Contenedor de constantes con valores asociados.

Podremos asociar una o más variables, para ello es necesario integrar un constructor "privado" y los getters para cada una de las variables insertadas.

- *EJEMPLO de insertar 1 variable (velocidad).*

```
//Uso de un constructor, una variable de instancia y un método.
enum Transporte{
    COCHE(60), CAMION(50), AVION(600), TREN(70), BARCO(20);
    private int velocidad; //velocidad típica de cada transporte

    //Añadir un onstructor
    Transporte(int s){velocidad=s;}
    //Añadir un método
    int getVelocidad(){return velocidad;}
}

class Enumerados {
    public static void main(String[] args) {
        Transporte tp;
       //Mostrar la velocidad de un avión
        System.out.println("La velocidad típica para un avión es: "+
        Transporte.AVION.getVelocidad()+ " millas por hora.\n");

        //Mostrar todas las velocidades y transportes
        System.out.println("Todas las velocidades de transporte: ");
        for (Transporte t:Transporte.values())
            System.out.println(t+ ": velocidad típica es "+t.getVelocidad()+" millas por hora.");
    }
}
```

EJEMPLO de insertar múltiples variables (Producto y precio)

```
public enum EnumTest {

    MESA("Mesa",50),
    SILLA("Silla",20),
    ARMARIO("Casa",120);

    private final String producto;
    private final int precio;

    private EnumTest(String producto, int precio) {
        this.producto = producto;
        this.precio = precio;
    }

    public String getProducto() {
        return producto;
    }

    public int getPrecio() {
        return precio;
    }

}
```

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

    System.out.println(EnumTest.ARMARIO);
    System.out.println(EnumTest.ARMARIO.getProducto());
    System.out.println(EnumTest.ARMARIO.getPrecio());

}
```

```
<terminated> MainEnum [Java Application]
ARMARIO
Casa
120
```

4.3. DIAGRAMA UML DE CLASES: ENUM.

En la mayoría de los casos se representa mediante una relación de dependencia.

5. INTERFAZ Y CLASE ABSTRACTA.

5.1. INTERFAZ:

Es un contrato de compromiso.

Conjunto de operaciones que una clase se compromete a implementar, marcando qué métodos y sus argumentos implementar.

También puede incluir constantes.

- La interfaz únicamente define los métodos y argumentos que se deben implementar.
- La clase que implemente la interfaz desarrollará cada uno de dichos métodos.

Así pues, aunque no se define/codifica cada método se "obliga-fuerza" a implementar cada uno de los elementos que debe integrar y que serán comunes para todos y cada uno de las clases que utilicen dicha Interfaz.

5.1.1. ¿Cómo diferenciar una clase de una interfaz?

- En lugar de usar la palaba reservada "class" utilizaremos la palabra "interface"
- La clase podrá implementar uno o más interfaces.

InterfazProfesiones.java

```
public interface InterfazProfesiones {

    public void viajar() ;
    public void cantar(String cancion);

}
```

InterfazProfesiones.java

```
public class Profesiones implements InterfazProfesiones{

    @Override
    public void viajar() {
        // TODO Auto-generated method stub

    }

    @Override
    public void cantar(String cancion) {
        // TODO Auto-generated method stub

    }

}
```

Main.java

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

    Profesiones profesion = new Profesiones();
    profesion.cantar("La Macarena");
    profesion.viajar();

}
```

5.1.2. PECULIARIDADES:

1. POSIBILIDAD: INTERFAZ INCLUIR MÉTODOS POR DEFECTO.

Este método por defecto contendrá la implementación ya indicada en la propia interfaz.

Estas interfaces por defecto se podrán modificar en la clase que implemente la interfaz. Sino es modificada por la clase tendrá accesibilidad directamente al algoritmo situado en la interfaz.

2. POSIBLIDAD: INTERFAZ INCLUIR MÉTODO ESTÁTICOS. Funcionaria exactamente igual que si estuviera el método estático en una clase normal.

3. UNA INTERFAZ PUEDA EXTENDER OTRA INTERFAZ. Por lo tanto, la clase que extienda la interfaz primera extenderá también la segunda.

3.1. E INCLUSO, UNA INTERFAZ PERMITE HERENCIA MÚLTIPLE DE VARIAS INTERFACES.

```
public interface GroupedInterface extends Interface1, Interface2  {
```

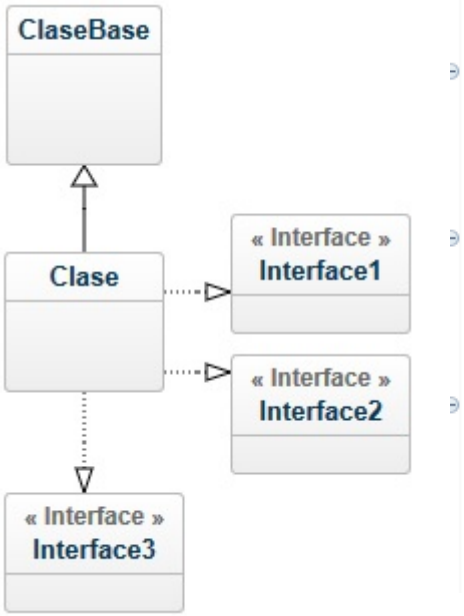
4. POSIBLIDAD UTILIZAR LA REFERENCIA DE UNA INTERFAZ A LA HORA DE INSTANCIAR UNA CLASE. Si la clase que instancia implementa dicha interfaz.

```
public class ClassInterface implements NewInterface{

    @Override
    public void doSomething(int i, double x) {
        // TODO Auto-generated method stub

    }

    @Override
    public int doSomething(String s) {
        // TODO Auto-generated method stub
        return 0;
    }

    @Override
    public void metodoInterfazPadre() {
        // TODO Auto-generated method stub

    }

}
```

```
public interface NewInterface extends NewInterfazPadre{

    static double PI = 1.1415;

    void doSomething(int i, double x);
    int doSomething(String s);

    default public void metodoPorDefecto() {
        System.out.println("Metodo interfaz x defecto");
    }

    public static void metodoEstatico() {
        System.out.println("Metodo interfaz estatico");
    }
}
```

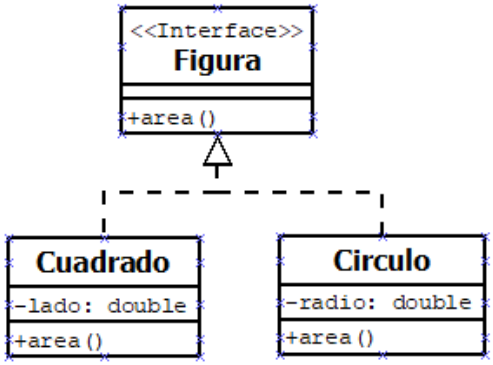
```
public interface NewInterfazPadre {

    public void metodoInterfazPadre();
}
```

5.1.3. INTERGRACIÓN EN UN DIAGRAMA DE CLASE UML




```java
public class Clase extends ClaseBase implements Interface1, Interface2, Interface3{

    @Override
    public void metodoInterface3() {
        // TODO Auto-generated method stub

    }

    @Override
    public void metodoInterface2() {
        // TODO Auto-generated method stub

    }

    @Override
    public void metodoInterface1() {
        // TODO Auto-generated method stub

    }

}
```

EJEMPLO: CALCULAR AREA DE UNA FIGURA. La interfaz Figura fuerza a que cada una de las clases que la implementen desarrollen como se calcula el área y según sea un círculo o un cuadrado dicho cálculo se realizará de forma diferente.




```java
public interface Figura
{
  public double area ();
}
```

```java
public class Circulo implements Figura
{
  private double radio;

  public Circulo (double radio)
  {
    this.radio = radio;
  }

  public double area ()
  {
    return Math.PI*radio*radio;
  }
}
```

```java
public class Cuadrado implements Figura
{
  private double lado;

  public Cuadrado (double lado)
  {
    this.lado = lado;
  }

  public double area ()
  {
    return lado*lado;
  }
}
```

DESDE JAVA 8 LAS INTERFACES YA PUEDEN INCLUIR CÓDIGO.

5.2. CLASE ABSTRACTA.

5.2.1. ¿Qué es una clase abstracta?

Las clases abstractas están en un paso intermedio entre la herencia y las interfaces. Una clase **puede** contener una o más métodos abstractos.

Su comportamiento es el siguiente:

- **Cercano a la herencia:** En las clases abstractas al igual que una clase padre se definen una serie de métodos y atributos que hereda la clase hija.
- **Cercano a la interfaz:**
  - Las clases abstractas definen/indican los métodos con su firma que se deben codificar por parte de la clase hija (uso de **abstract**)
  - No se pueden crear instancias de dicha clase. Actúa únicamente como plantilla para otra clase que extienda de ella. *[Al contrario que en la clase padre que cabe la posibilidad de crear un objeto dicha clase, nunca se puede instanciar un objeto de una clase abstracta]*

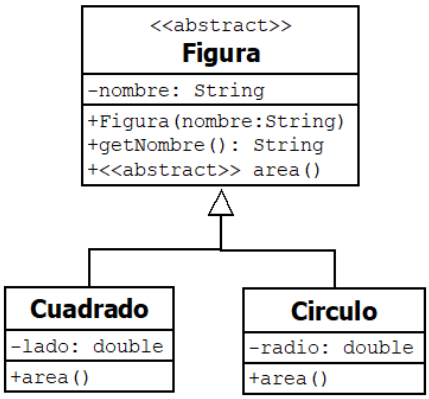
5.2.2. ¿Qué es un método abstracto?

- Deben estar dentro de una clase abstracta.
- Conviven dentro de la clase abstracta junto con otros métodos implementados / "normales".
- Definen la firma (argumentos) del método, pero sin implementarlos (similar al caso de la interfaz). Estos métodos deberán ser implementados por las clases hijas.

Precauciones:

- **Sólo puede existir dentro de una clase abstracta.** Si una clase incluye un método abstracto, forzosamente la clase será una clase abstracta.
- **Los métodos abstractos forzosamente habrán de estar sobreescritos en las subclases**. Si una subclase no implementa un método abstracto de la superclase tiene un método no ejecutable, lo que la fuerza a ser una subclase abstracta.

5.2.3. PRACTICA:




```java
public abstract class Figura {

    private String nombre;

    public Figura(String nombre) {
        super();
        this.setNombre(nombre);
    }

    public String getNombre() {
        return nombre;
    }

    public void setNombre(String nombre) {
        this.nombre = nombre;
    }

    public abstract double area();

}
```

```java
public class Circulo extends Figura{

    private double radio;


    public Circulo(String nombre, double radio) {
        super(nombre);
        this.radio = radio;
    }

    @Override
    public double area() {
        return Math.PI*radio*radio;

    }

}
```

```java
public class Cuadrado extends Figura{
    private double lado;


    public Cuadrado(String nombre, double lado) {
        super(nombre);
        this.lado = lado;
    }

    @Override
    public double area() {
        return lado*lado;

    }
}
```

5.3. ¿Cuándo usar interfaz y cuando usar una clase abstracta? Depende…

INTERFACES	CLASES ABSTRACTAS
No se pueden instanciar	No se pueden instanciar
Métodos sin implementación	Métodos sin implementación
Métodos con implementación por defecto	Métodos con implementación por defecto
Atributos estáticos o constantes	Cualquier tipo de atributos
Métodos públicos o por defecto	Métodos públicos, privados, protegidos o por defecto.
Una clase puede implementar varios interfaces	Una clase solo puede heredar de otra

DIFERENCIAS CLAVE:

DIF1:

- Interfaces no pueden tener atributos o constructores, y lo normal es que no tengan métodos codificados en su interior. Como desde Java8 se pueden codificar métodos, pero estos deben tener la palabra reservada "default" obligatoriamente y no pueden ser modificados por la clase que implemente la interfaz.
- Abstract es como una superclase que vimos en herencia, pero que implementa un método abstracto, en ese caso al implementar un método abstracto obliga que la clase implemente sí o sí ese método.

DIF2:

- Interfaces. Una clase puede implementar múltiples interfaces utilizando "implements".
- Abstract. Al igual que en herencia utiliza el operador **"extends"** y únicamente se puede extender una única clase ya sea abstracta o no.