6. POO: Desarrollo de clases.

1.- Introducción

"**Orientado a objetos**" hace referencia a una forma diferente de acometer la tarea del desarrollo de software, frente a otros modelos como el de la programación imperativa, la programación funcional o la programación lógica. Supone una reconsideración de los métodos de programación, de la forma de estructurar la información y, ante todo, de la forma de pensar en la resolución de problemas.

La **programación orientada a objetos (P.O.O.)** es un modelo para la elaboración de programas que ha impuesto en los últimos años. Este auge se debe, en parte, a que esta forma de programar está fuertemente basada en la representación de la realidad; pero también a que refuerza el uso de buenos criterios aplicables al desarrollo de programas, como son la **abstracción**, la **ocultación de información** y la **reusabilidad**, entre otros.

Naturalmente, el elemento fundamental en la P.O.O. es el **objeto**. Un **objeto** es la encapsulación en un solo elemento de datos y de la forma de manipular dichos datos. Los objetos combinan datos y comportamiento. Los datos se materializan a través de los **atributos** del objeto (variables) y su comportamiento queda definido por los **métodos** del objeto.

Los objetos pertenecen a una **clase**. Una **clase** es la descripción de cuales son los datos y el comportamiento de cada uno de los objetos de la clase. Un objeto es una instancia de una clase.

El lenguaje Java es un lenguaje orientado a objetos, por lo que se puede decir que programar en Java consiste en "escribir las definiciones de las clases y utilizar esas clases para crear objetos" de forma que se represente adecuadamente el problema que se desea resolver.

Los programas que hemos desarrollado hasta ahora estaban compuestos, generalmente, por una sola clase, que contenía al método main y otros métodos *static* a los que se llamaba desde main. En éstos, hemos hecho uso de clases ya existentes en Java: String, Random, PrintStream, Scanner, etc. En este tema vamos a crear nuestras propias clases, clases que representan elementos que aparecen en el problema a resolver y que utilizaremos para su resolución.

2.- Definición de clases. Cuestiones básicas

2.1.- Un problema sencillo

Una empresa de transportes quiere registrar información sobre los **trayectos** que realizan sus **vehículos**. En concreto, cuando un vehículo realiza un trayecto, quiere conocer la **hora** de salida desde el origen y la **hora** de llegada a su destino, para poder controlar, entre otras cosas, el tiempo tardado en el trayecto.

Vamos a diseñar una clase Tiempo para representar un hora (de salida o de llegada) y que nos facilite la resolución del problema que queremos resolver.

2.2.- Una clase sencilla.

```
public class Tiempo {
  // Atributos
  int hora;
  int minuto;
  int segundo;

  // Métodos
  public String toString(){
        return hora + ":" + minuto +":"+ segundo;
  }

} // de la clase
```

Usemos la clase Tiempo en la resolución del problema de control de trayectos que se nos planteaba al comienzo:

```
import java.util.*;
public class ControlTrayectos{
  public static void main(String arg[]){
        Scanner tec = new Scanner(System.in);

        // Definimos dos variables de tipo Tiempo
        Tiempo salida = new Tiempo();
        Tiempo llegada = new Tiempo();

        //Leemos las horas de teclado
        System.out.println("Hora de salida:");
        System.out.println("Hora: ");
        salida.hora = tec.nextInt()
        System.out.println("Minuto: ");
        salida.minuto= tec.nextInt()
        System.out.println("Segundos:");
        salida.segundo = tec.nextInt()
```

```
            System.out.println("Hora de llegada:");
            System.out.println("Hora: ");
            llegada.hora = tec.nextInt()
            System.out.println("Minuto: ");
            llegada.minuto = tec.nextInt()
            System.out.println("Segundos:");
            llegada.segundo = tec.nextInt()

            // Mostremos las horas por pantalla
            System.out.println ("Hora de salida: " + salida.toString());
            System.out.println("Hora de llegada: " + llegada.toString());
    }
}
```

¿Que significa *public class?*

Una clase definida como ***public*** puede ser usada desde otras clases. En nuestro ejemplo, la clase ControlTrayectos crea dos objetos de la clase Tiempo. Estamos usando una clase desde otra clase.

Si no definimos una clase como *public*, ésta solo podrá usarse desde clases que estén definidas en el mismo paquete.

Generalmente se escribe cada clase en un fichero .java distinto. En un mismo fichero puede haber varias clases, pero solo una de ellas puede ser pública. Es decir, solo una de ellas podría usarse desde otras clases. El resto sería de *uso interno*.

Atributos

La clase Tiempo define tres **atributos** *int:* hora, minuto y segundo. Los atributos de la clase permiten que un objeto almacene información.

Como vemos se declaran de la misma forma que hasta ahora las variables. Sin embargo se declaran fuera de todos los métodos.

En la clase *ControlTrayectos* hemos definido dos objetos de la clase Tiempo y hemos modificado sus atributos. Para referirnos a un atributo de un objeto usamos el punto entre el nombre de la variable y el del atributo: *salida.hora*

Métodos. El método toString.

La clase Tiempo, además de atributos, tiene definido un **método**.

Los métodos permiten definir el comportamiento de los objetos.

Observa que el método definido **no lleva la palabra *static*** en su cabecera. Más adelante repasaremos cual era el significado de este modificador, cuando hay que usarlo y cuando no.

El método ***toString*** es un método que tienen todas las clases en Java. Se utiliza para devolver un *String* que represente al objeto y, así, poder imprimirlo en pantalla o en un fichero.

En el ejemplo, se devuelve un *String* con la forma hora**:**minuto**:**segundo. Posteriormente, en el método *main* de *ControlTrayectos*, se llama a *toString*() para mostrar por pantalla la hora de salida y la de llegada.

Cuando usamos un objeto como si se tratase de un String, java llama automáticamente al método toString de dicho objeto. Por ejemplo, las dos siguientes instrucciones son equivalentes:

```
System.out.println ("Hora de salida: " + salida.toString());

System.out.println ("Hora de salida: " + salida);
```

En la seguda estamos concatenando el objeto *salida* a la derecha de un String. Es decir, estamos usando *salida* como si fuera un String. Java llamará automáticamente a salida.toString() para ejecutar la expresión.

Además hemos dicho que el método toString() está definido en todas las clases de Java. Si definimos una clase y no escribimos su método toString(), éste existirá en la clase, aunque con un comportamiento por defecto que muestra:

*nombreDeLaClase***@***direcciónDeMemoria*

2.3.- Ocultación de información, consultores y modificadores.

En el ejemplo anterior hemos visto como en la clase *ControlTrayecto* modificábamos los atributos (hora, minuto, segundo) de los objetos *salida* y *llegada* que había definido.

Esto (permitir la modificación directa de los atributos de un objeto) es una práctica no recomendable. **Como norma, los atributos de una clase serán privados, y se proporcionarán métodos públicos para permitir el acceso a dichos atributos.**

Lo anterior permite el cumplimiento de uno de los principios fundamentales en el campo de la programación: el **principio de ocultación de información**. Según este principio, se debe ocultar a los usuarios de una clase los detalles de implementación de dicha clase. La forma en que se tiene que utilizar la clase ha de ser independiente de sus detalles internos, de sus detalles de implementación.

Así, en nuestro ejemplo, para representar un *tiempo* hemos utilizado tres atributos: hora, minuto y segundo, pero podríamos haber representado un *tiempo* guardando, únicamente, el número de segundos transcurridos desde las 0 horas. Si modificaramos la clase Tiempo en este sentido, la clase ControlTrayectos dejaría de funcionar, y esto no es deseable.

Vamos a modificar nuestras clases para que cumplan con el principio de ocultación de información y que sean más robustas.

```
public class Tiempo {
   private int hora;
   private int minuto;
   private int segundo;

   // Metodos consultores
```

```
    public int getHora() {
       return hora;
    }
    public int getMinuto() {
       return minuto;
    }
    public int getSegundo() {
       return segundo;
    }

   // Metodos modificadores
    public void setHora(int hora) {
       if(hora <0 || hora >23) this.hora = 0;
       else this.hora = hora;
    }
    public void setMinuto(int minuto) {
       if(minuto < 0 || minuto >59) this.minuto = 0;
       this.minuto = minuto;
    }
    public void setSegundo(int segundo) {
       if(segundo <0 || segundo >59) this.segundo = 0;
       this.segundo = segundo;
    }

    public String toString(){
       return getHora() + ":" + getMinuto() + ":" + getSegundo();
    }
}
```

Atributos privados, métodos publicos

Como hemos adelantado, por norma definiremos los atributos privados. Esto se hace poniendo al atributo el modificador ***private***.

La consecuencia inmediata es que los atributos s**olo son accesibles en la clase en que están definidos**. Comprueba que ahora la clase ControlTrayectos contiene errores. En concreto, los atributos hora, minuto, segundo son inaccesibles, ya que han sido definidos en otra clase y son privados.

Esto nos obliga a definir métodos publicos que permitan consultar y modificar el valor de los atributos privados. Estos métodos reciben el nombre de **consultores y modificadores** y, en Java, se sigue el convenio de que comiencen por **get** y **set** respectivamente, aunque se les puede dar cualquier nombre válido.

Como podemos observar los consultores (getHora, getMinuto, getSegundo) se limitan a devolver el valor de sus respectivos atributos.

En el caso de los modificadores hemos introducido una mejora: No permiten que a los atributos se les de un valor incorrecto.

La clase ControlTrayectos se modifica como sigue:

```
import java.util.*;
public class ControlTrayectos {

   public static void main(String[] args) {
      Scanner tec = new Scanner(System.in);

      Tiempo salida = new Tiempo();
      Tiempo llegada = new Tiempo();

      //Hora de salida
      System.out.println("Hora de salida: ");
      System.out.println("Hora: ");
      salida.setHora(tec.nextInt());
      System.out.println("Minutos: ");
      salida.setMinuto(tec.nextInt());
      System.out.println("Segundos: ");
      salida.setSegundo(tec.nextInt());

      //Hora de llegada
      System.out.println("Hora de llegada: ");
      System.out.println("Hora: ");
      llegada.setHora(tec.nextInt());
      System.out.println("Minutos: ");
      llegada.setMinuto(tec.nextInt());
      System.out.println("Segundos: ");
      llegada.setSegundo(tec.nextInt());

      //Mostramos las horas introducidas.
      System.out.println("Hora de salida: " + salida.toString());
      System.out.println("Hora de llegada: " + llegada.toString());
   }
```

Por el momento todo parece trabajo extra. ¿Que ventajas tiene en nuestro ejemplo la ocultación de información?

- Podemos decidir **qué** atributos se pueden **consultar** y cuales no. Basta con implementar o no el correspondiente método consultor (o con hacerlo *public* o *private*)
- Podemos decidir **qué** atributos se pueden **modificar** y **cómo** se pueden modificar. Por ejemplo, los modificadores de la clase *Tiempo* impiden que la hora pueda ser una hora inválida como la siguiente: *25:61:89*
- Si decidiéramos que para representar una hora es mejor guardar el número de segundos transcurridos desde las 00:00 horas (en lugar

de la hora, el minuto y el segundo) solo habría que modificar la clase *Tiempo*. La clase *ControlTrayectos* no depende de sus detalles de implementación.

El uso de this

Siempre que se ejecuta un método (que no ha sido definido static), se ejecuta sobre un objeto:

llegada.setHora() ejecuta el método *setHora* del objeto *llegada*
salida.toString() ejecuta el método toString del objeto salida

En la clase *Tiempo*, **this** representa al objeto sobre el que se ha ejecutado el método.

Generalmente, el uso de this es opcional. Por ejemplo estas dos versiones del método getHora() son equivalentes.

```
public int getHora() {
   return hora;
}
public int getHora() {
   return this.hora;
}
```

Sin embargo en ocasiones puede ser necesario utilizarlo. Por ejemplo para deshacer la ambigüedad cuando un atributo y un parámetro tienen el mismo nombre:

```
public void setHora(int hora) {
   if(hora <0 || hora >23) this.hora = 0;
   else this.hora = hora;
}
```

En el método setHora hay dos "cosas "que se llaman *hora*:

- el parámetro hora que recibe el método.
- el atributo hora de la clase.

Para diferenciar al atributo del parámetro usamos this cuando nos referimos al atributo.

Si el parámetro tuviese un nombre distinto el uso de this no sería necesario, aunque se podría utilizar igualmente:

```
public void setHora(int h) {
   if(h <0 || h>23) hora = 0;
   else hora = h;
}


public void setHora(int h) {
   if(h <0 || h>23) this.hora = 0;
```

```
        else this.hora = h;
    }
```

2.4.- Constructores.

Un constructor es un método que se utiliza para construir un objeto, para inicializarlo, podríamos decir.

El método constructor de una clase es un método especial que ...

- tiene el mismo nombre que la clase.
- no devuelve nada (pero no se pone *void* en la cabecera del método)
- igual que otros métodos, puede llevar o no llevar parámetros.

Ya estamos acostumbrados a utilizar métodos constructores. Por ejemplo cuando escribimos

```
Scanner tec = new Scanner(System.in);
```

estamos llamando al método constructor de la clase Scanner, y le pasamos como parámetro el objeto *System.in*, perteneciente a la clase *java.io.InputStream*

Una clase puede tener varios constructores, siempre que estos se diferencien en el número o tipo de los parámetros que reciben. Esto último (que pueda haber varios constructores) recibe el nombre de **sobrecarga** y en realidad es aplicable a cualquier método. Es decir, podemos tener en una misma clase varios métodos con el mismo nombre, siempre que se diferencien en el número o tipo de sus parámetros.

Añadamos constructores a la clase Tiempo.

```
public Tiempo (int hora, int minuto, int segundo){
  this.hora = hora;
  this.minuto = minuto;
  this.segundo = segundo;
}
```

o, **mucho mejor**, para asegurarnos de que la hora es correcta....

```
/**
* Construye un nuevo tiempo con los la hora, minuto y segundos indicada
* @param hora La hora
* @param minuto El minuto
* @param segundo El segundo
*/
public Tiempo (int hora, int minuto, int segundo){
  this.setHora(hora);
  this.setMinuto(minuto);
  this.setSegundo(segundo);
}
```

Este constructor permite crear objetos de la clase tiempo de la siguiente forma:

```
Tiempo t = new Tiempo(20,15,25);
```

es decir, indicando la hora, minuto y segundos en el momento de la creación.

Pero podría interesar poder crear objetos sin tener que indicar los segundos, o los minutos, o incluso sin indicar nada. Veamos otros **constructores posibles**

```
/**
* Construye un nuevo tiempo con los la hora y  minuto indicados. Los segundos
* serán cero
* @param hora La hora
* @param minuto El minuto
*/
public Tiempo (int hora, int minuto){
  this.setHora(hora);
  this.setMinuto(minuto);
  this.setSegundo(0);
  // o también this(hora,minuto,0);
}

/**
* Construye un nuevo tiempo con los la hora indicada. Los minutos y segundos
* serán cero
* @param hora La hora
*/
public Tiempo (int hora, int minuto){
  this.setHora(hora);
  this.setMinuto(0);
  this.setSegundo(0);
  // o también this(hora,0,0);
}

/**
* Construye un nuevo tiempo correspondiente a las 00:00:00
* @param hora La hora
*/
public Tiempo (int hora, int minuto){
  this.setHora(0);
  this.setMinuto(0);
  this.setSegundo(0);
  // o también this(0,0,0);
}
```

Anteriormente hemos visto un uso de this para deshacer la ambigüedad originada por un atributo y un parámetro que se llaman igual. Aquí se está haciendo un uso distinto de this. Cuando this se usa como un método representa al

constructor de la clase. Por tanto en estas instrucciones estamos llamando un constructor desde otro constructor.

Modifiquemos la clase ControlTrayecto para que utilice el constructor de la clase Tiempo:

```
import java.util.*;
public class ControlTrayectos {

   public static void main(String[] args) {
      Scanner tec = new Scanner(System.in);
      int h, m, s;

      //Hora de salida
      System.out.println("Hora de salida: ");
      System.out.println("Hora: ");
      h =tec.nextInt();
      System.out.println("Minutos: ");
      m = tec.nextInt();
      System.out.println("Segundos: ");
      s = tec.nextInt();
      Tiempo salida = new Tiempo(h,m,s);

      //Hora de llegada
      System.out.println("Hora de llegada: ");
      System.out.println("Hora: ");
      h =tec.nextInt();
      System.out.println("Minutos: ");
      m = tec.nextInt();
      System.out.println("Segundos: ");
      s = tec.nextInt();
      Tiempo salida = new Tiempo(h,m,s);

      //Mostramos las horas introducidas.
      System.out.println("Hora de salida: " + salida.toString());
      System.out.println("Hora de llegada: " + llegada.toString());
   }
```

2.5.- Los metodos equals y compareTo

Al utilizar Strings aprendimos, entre otras cosas, que los objetos no se comparan con los operadores ==, !=, >, >=, ..., sino con equals y compareTo.

Eso era así porque, como sabemos, las variables que no son de tipo primitivo almacenan referencias a los objetos, y lo que estaríamos comparando sería dichas referencias y no los objetos.

Si queremos tener la posibilidad de comprobar la igualdad de dos objetos de la clase Tiempo tenemos que añadirle un método equals.

Si queremos tener la posibilidad de comprobar si un Tiempo es mayor o menor que otro, tenemos que añadir el método *compareTo* a la clase.

Aquí tenemos ambos métodos:

```
/*
* Comprueba si  este Tiempo es igual al objeto especificado
* @return True si son iguales
*/
public boolean equals( Object o) {
  if(this == o) return true;
  if(!(o instanceof Tiempo)) return false;
  Tiempo x = (Tiempo) o; //Cambio de tipo
        if (hora == x.hora && minuto == x.minuto && segundo == x.segundo)
              return true;
        else
              return false;
}

/*
* Compara este Tiempo con el objeto especificado
* Devuelve un entero
*       menor que cero si este Tiempo es menor
*       mayor que cero si este Tiempo es mayor
*       cero si son iguales.
*/
public boolean compareTo ( Object o) {
  Tiempo x = (Tiempo) o; //Cambio de tipo
  if (hora > x.hora) return 1;
  else if (hora<x.hora) return -1;
  else{
        // Horas iguales, comparamos los minutos
        if(minuto > x.minuto) return 1;
        else if(minuto < x.minuto) return -1;
        else {
              //Minutos iguales, comparamos segundos
              return (segundos-x.segundos);
              /* O lo que es lo mismo
              if(segundos > segundos) return 1;
              else if(segundos < segundos) return -1;
              else return 0;
              */
        }}
```

Ahora podriamos modificar ControlTrayectos para asegurarnos de que la hora de llegada es posterior a la hora de salida:

```
...
if (salida.compareTo(llegada) <= 0)
  System.out.println("La hora de llegada debe ser posterior a la de salida");
```

3.- Atributos y métodos estáticos

3.1.- Atributos estáticos

Cuando en una clase definimos un atributo no estático, cada objeto tiene su propio valor de dicho atributo. Así por ejemplo, dos objetos de la clase Tiempo tendrán sus propios valores de los atributos hora, minuto y segundo.

Sin embargo, si un atributo es estático (***static***), todos los objetos de la clase comparten el valor de dicho atributo. Cada objeto no tiene su propio atributo, sino que todos comparten el mismo atributo. Se dice que los atributos *static* son atributos de clase.

Así, por ejemplo:

```
class Ejemplo {
  int x; // x es un atributo
  static int y; // y es un atributo estático, un atributo de clase.
}
```

Cuando se declare un nuevo objeto de tipo *Ejemplo*, contendrá su propio valor del atributo *x*, mientras que *y* se comporta como un atributo colectivo para todos los objetos de la clase.

```
class Prueba {
  public static void main(String args[]){
        Ejemplo e1 = new Ejemplo();
        Ejemplo e2 = new Ejemplo();
        e1.x = 1;
        e2.x = 1;
        e1.x++; // e1.x = 2
        e2.x +=5; // e1.x = 6

        e1.y = 1; //e1.y = 1;
        e1.y++; //e1.y = 2;
        e2.y++; //e2.y = 3;
        ...
  }
}
```

Los atributos static se usan habitualmente para mantener información relativa a todos los objetos de la clase. Por ejemplo, en la clase Tiempo un atributo static nos permitiría saber cuantos objetos de la clase han sido instanciados (creados). Para ello definimos un atributo static con valor inicial 0 que incrementamos en el constructor de la clase:

```
public class Tiempo {
  private int hora;
  private int minuto;
  private int segundo;
  private static int creados = 0;

  public Tiempo (int hora, int minuto, int segundo){
        this.setHora(hora);
        this.setMinuto(minuto);
        this.setSegundo(segundo);

        creados ++;
  }
  ...
}
```

Cada vez que se crea con este constructor un nuevo objeto de la clase Tiempo, se incrementa el contador *creados* asociado a la clase.

Otro uso habitual de los atributos estáticos consiste en la definición de **constantes**. En dicho caso se utiliza también el modificador **final**, con el que se especifica la no modificabilidad.

Por ejemplo, en Java la clase predefinida Integer contiene la definición siguiente:

```
public final static int MAX_VALUE = 2147483647;
```

a la que haríamos referencia con `Integer.MAX_VALUE`

También es ya conocida `Math.PI`, definida en la clase Math como sigue

```
public final static double PI = 3.159·····;
```

3.2.- Métodos static

Cuando ejecutamos un método no estático tenemos que indicar en la llamada cual es el objeto sobre el que actúa el método:

objeto.metodo()

Si el método actúa (consulta o modifica) sobre algún atributo, lo hará sobre los atributos del objeto que se ha especificado

Sin embargo, los métodos ***static*** no se ejecutan sobre un objeto en concreto, sino que son métodos de la clase. Para ejecutarlos se antepone al nombre del método el nombre de la clase en la que se ha definido el método (salvo que sea la misma clase desde donde se llama al método)

```
double x = Math.sqrt(10);
```

sqrt no se ejecuta sobre objeto alguno. Simplemente indicamos que queremos ejecutar el método sqrt que pertenece a la clase Math. Sqrt es un método estático de la clase Math.

A lo largo del curso los hemos usado frecuentemente, tanto como métodos predefinidos de algunas clases de Java (como Math), como para definir funciones y procedimientos en la descomposición modular de un programa. El método main y los métodos que hemos usado hasta ahora para confeccionar programas eran static.

4.- Relaciones entre clases: Relaciones TIENE_UN

Desde la perspectiva de la POO, un programa está formado por un conjunto de clases. Estas clases no están aisladas, sino que interactúan, se relacionan entre sí.

Son varias las formas en las que las clases de un programa se relacionan unas con otras. La forma más sencilla de relación son las llamadas relaciones TIENE_UN

Según estas, los atributos de una clase pueden ser objetos de otras clases.

Por ejemplo, si definimos la clase *Alumno* con un atributo *nombre* de tipo String

```
public class Alumno{
  private int numExpediente;
  private String nombre;
  ...
}
```

las clases Alumno y String están relacionadas: *Alumno* TIENE_UN *String*

En el ejemplo estamos relacionando la clase que definimos con otra ya existente en Java, la clase String. Sin embargo es posible relacionar dos o más clases cualesquiera. Es decir, podemos relacionar de esta forma clases no predefinidas en Java como ocurre en el siguiente ejemplo:

Supongamos que queremos representar la clase *Grupo*,:

```
public class Grupo {
  privata String nombre;
  private Alumno delegado;
  private int numMatriculados;
  public Grupo(String nombre){
        this.nombre = nombre;
        this.numMatriculados = 0;
  }
  ...
  ...
  public void setDelegado(Alumno a) {
        this.delegado = a;
  }
  public void setDelegado(int expediente, String nombre) {
        this.delegado = new Alumno(expediente, nombre);
```

```
    }

}
...
```

En este ejemplo, el método setDelegado actua como modificador del atributo *delegado*, de tipo *Alumno.* Para designar un alumno como delegado en el ejemplo se trabaja con dos posibilidades:

- Que el modificador reciba un objeto de tipo *Alumno*, que asigna directamente al atributo delegado.
- Que el modificador reciba los datos de un alumno. En tal caso el objeto de la clase *Alumno* ha de ser creado.

Las dos versiones son válidas. Utilizar una u otra vendrá dado por cuestiones en el diseño de la clase.