TEMA 5. DESARROLLO DE LA ESTRUCTURA DE UNA APLICACIÓN BASADA EN POO: OBJETOS Y CLASES. UTILIZACIÓN DE CLASES PREDEFINIDAS.

INDICE

DEFINICIÓN AVANZADA DE CLASES.

En este tema vamos a ver como añadir a las clases más funcionalidades. Para ello vamos a tomar como referencia las clases ya definidas con anterioridad.

Vamos a guardar todas las clases dentro del mismo paquete para facilitar el trabajo.

MÉTODOS DE ACCESO A LOS ATRIBUTOS DE UNA CLASE. SETTERS y GETTERS

Al definir un atributo de una clase como **private** impedimos que desde otras clases se pueda acceder a él.

Si queremos permitir que desde otras clases se pueda acceder a él debemos crear unos métodos public que realicen la función.

A los métodos public que se utilizan para **cambiar el valor** de un atributo de la clase se les denomina Setters ya que su nombre suele ser ***setNombreatributo***. Estos métodos reciben como parámetro un valor del mismo tipo que el del atributo al que queremos cambiar el valor y no devuelven nada (son de tipo void). Por ejemplo, si queremos crear un método que permita cambiar el valor del atributo real de la clase complejo escribimos

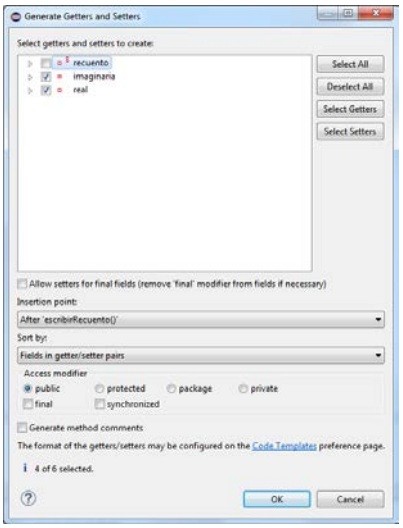
```
public void setReal(double r) {
 real = r;
}
```

A los métodos public que se utilizan para **obtener el valor** de un atributo de la clase se les denomina Getters ya que su nombre suele ser ***getNombreatributo***. Estos métodos no reciben ningún parámetro y devuelven un valor del mismo tipo que el del atributo del que queremos obtener el valor. Por ejemplo, si queremos crear un método que permita obtener el valor del atributo real de la clase complejo escribimos

```
public double getReal() {
 return real;
}
```

En Eclipse podemos generar los Setters y Getters automáticamente haciendo clic con el botón derecho sobre el código de la clase y seleccionando ***Source -> Generate Setters and Getters …***

Una vez allí seleccionamos los atributos para los que queremos generar los setters y getters el método de acceso (por defecto public) y pulsamos OK.



Estos métodos de tipo public los lo que debemos hacer es crear Un método o función miembro es un trozo de código que permite realizar una operación sobre uno o varios atributos de la clase.

MÉTODOS DE UNA CLASE

Un método o función miembro es un trozo de código que permite realizar una operación sobre uno o varios atributos de la clase.

Si queremos que un método de una clase pueda ser llamado desde otra clase debemos escribir el identificador **public** delante de su declaración.

Por ejemplo, si en la clase **Complejo** queremos crear un método de nombre **escribir** que nos permita escribir el valor de los atributos de la clase por pantalla con el formato ***real " + " imaginaria + "i"*** (si el complejo tiene como parte real 3.0 y como parte imaginaria 5.0 escribiría ***3.0 + 5.0 i***) dentro de la clase Complejo escribimos

```
public void escribir (){
 System.out.print(real + " + " + imaginaria + "i");
}
```

Para utilizar el método escribir de la clase Complejo desde otra clase, por ejemplo desde la clase **ComplejoMain**, basta con definir un objeto de la clase complejo y llamar a ese método.

```
Complejo c = new Complejo(3.0,5.0);
c.escribir();
```

PASO DE PARÁMETROS POR VALOR Y POR REFERENCIA

En Java, a diferencia de otros lenguajes como c++, **todos los parámetros se pasan por valor**. Cuando los parámetros son de tipos predefinidos (por ejemplo int, double, ...) se pasa directamente el valor mientras que si los parámetros son de tipo Objeto (por ejemplo un objeto de tipo String) se pasa el valor de la dirección del objeto.

En Java al pasar siempre los parámetros **por valor** si queremos guardar el cambio de un parámetro dentro de un método tendremos que hacer que el método devuelva un valor del mismo tipo que el del parámetro que queramos modificar y recoger ese valor devuelto modificado en la variable que hemos pasado como parámetro.

Por ejemplo, si tenemos una variable de tipo int y de nombre n que vale 5 y queremos usar un método de nombre incrementar que recibe la variable n como parámetro, la incrementa y devuelve el valor incrementado escribimos.

Por ejemplo, si tenemos una clase de nombre **Parametros** que tiene un método **main** con una variable de tipo int y de nombre n que vale 5 y otro de nombre **incrementar** que recibe la variable n como parámetro, la incrementa y devuelve el valor incrementado escribimos

```
public static void main(String[] args) {
 int n=5;
 n=incrementar(n);
}
```

```
private static int incrementar(int n){
 n = n + 1;
 return (n);
}
```

Para controlar la modificación de varios parámetros dentro de un método el método debe devolver un tipo complejo de datos (por ejemplo un array).

RECURSIVIDAD

La recursividad es una técnica que permite que un elemento se llame a sí mismo. Un ejemplo típico de recursividad es cuando un método se llama a sí mismo para resolver un problema.

Las soluciones recursivas constan de un caso general y de un caso base cuyo resultado es conocido y permite la finalización del proceso recursivo.

Un ejemplo típico de recursividad es el cálculo del factorial. En el cálculo del factorial el caso base es **F(1)=1** y el caso general es **F(N)=N*F(N-1)**.

Por ejemplo, si queremos calcular el factorial de manera recursiva en Java nos creamos una clase de nombre **FactorialRecursivo** y escribimos

```
public static void main(String[] args) {
 int n=5;
 int fn=factorial(n);
}

private static int factorial(int n){
 if(n==0){
  return 1;   // Caso Base
 }
 else{
  return (n*factorial(n-1));  // Caso General
 }
}
```

No todos los problemas se pueden resolver de forma recursiva. Además, las soluciones recursivas consumen más recursos del sistema (procesador, memoria, ...) que las soluciones no recursivas y pueden provocar un cuelgue del sistema. Esto es así porque hasta que no finaliza la ejecución de una instancia ésta sigue cargada en memoria.

SOBREESCRITURA DE MÉTODOS. SOBRECARGA DE OPERADORES

A la hora de definir una clase derivada de otra mediante herencia podemos modificar el comportamiento de alguno de los métodos heredados.

Por ejemplo, en Java todas las clases descienden de la clase Object. Esta clase Object dispone de una serie de métodos que son heredados por todas las clases Java. Estos métodos los podemos modificar para ajustar su comportamiento predefinido al comportamiento que necesitemos.

Los métodos que más se suelen sobrescribir son el método **toString** que permite mostrar el valor de los atributos de una clase como un String y los métodos **equals** y **hashCode** que permiten comparar objetos de una misma clase.

Eclipse nos permite generar automáticamente estos métodos. Para ello hacemos clic con el botón derecho sobre el código de la clase y seleccionando ***Source -> Generate toString ...*** o ***Source -> hashCode and equals ...***

Antes de la definición de cada método se añade la línea **@Override** que indica que el método sobrescribe la definición de otro método anterior.

El código de los métodos para la clase Complejo queda de la siguiente manera

CONVERTIR LOS DATOS DE UNA CLASE A STRING. toString

En algunos casos puede ser interesante convertir el valor de los atributos de una clase a un String. Por ejemplo para mostrar el valor de los atributos de una clase por pantalla.
Por ejemplo, si en la clase Complejo queremos sobrescribir el método **toString** para que devuelva el valor de los atributos de la clase como un String con el formato ***real*** + " + " + ***imaginaria*** + "***i***" (si el complejo tiene como parte real 3 y como parte imaginaria 5 escribiría ***3 + 5 i***) escribimos

```
@Override
public String toString() {
 return ("real + " + " + imaginaria + "i"");
}
```

COMPARAR LOS OBJETOS DE UNA CLASE. equals. hashCode

Para comparar dos objetos de una misma clase NO podemos usar el operador de comparación ==. Si queremos permitir que dos objetos de una misma clase se puedan comparar debemos sobrescribir los métodos equals y hashCode para esa clase.

El método **hashCode** genera un código especial (hash) que sirve para determinar si dos objetos de una misma clase son o no iguales. Es fundamental para que el método equals funcione correctamente.

Por ejemplo, el método hashCode de la clase Complejo sería de la forma

```
@Override
public int hashCode() {
 final int prime = 31;
 int result = 1;
 long temp;
 temp = Double.doubleToLongBits(imaginaria);
 result = prime * result + (int) (temp ^ (temp >>> 32));
 temp = Double.doubleToLongBits(real);
 result = prime * result + (int) (temp ^ (temp >>> 32));
 return result;
}
```

El método **equals** compara dos objetos de una misma clase. Por ejemplo, el método equals de la clase Complejo sería de la forma

```
@Override
public boolean equals(Object obj) {
 if (this == obj)    // si es el mismo objeto
```

```
 return true;
if (obj == null)   // si el objeto es nulo
 return false;
if (getClass() != obj.getClass()) // si los objetos no son de la misma clase
 return false;
Complejo other = (Complejo) obj; // creo un objeto temporal
if (Double.doubleToLongBits(imaginaria) != Double.doubleToLongBits(other.imaginaria))
// comparo el primer campo
 return false;
if (Double.doubleToLongBits(real) != Double.doubleToLongBits(other.real))
// comparo el segundo campo
 return false;
return true;
}
```

COMPARAR LOS OBJETOS DE UNA CLASE. Comparable. compareTo

Para comparar si dos objetos de una misma clase son iguales, mayores o menores, debemos usar el interfaz **Comparable**. Los interfaces los veremos con más detalle cuando veamos los mecanismos de herencia de clases.

Para declarar que una clase va a usar un interfaz debemos escribir en la declaración de la clase la palabra reservada **implements**.

Por ejemplo, para indicar que en la clase Complejo queremos usar el interfaz Comparable escribimos

```
public class Complejo implements Comparable<Complejo>{...}
```

Una vez hecho esto dentro de la clase debemos sobrescribir el método **compareTo** indicando como parámetro un objeto de la clase

```
@Override
public int compareTo(Complejo c1) {
...
}
```

Dentro del método compareTo debemos escribir el código que realice la comparación de los objetos de la misma clase.

La función **compareTo** devuelve un **entero** que será **mayor que 0** si el objeto que llama al método es mayor que el objeto que se pasa como parámetro, **menor que 0** si el objeto que llama al método es menor que el objeto que se pasa como parámetro, o **0** si los dos objetos tienen el mismo valor.

Por ejemplo, para comparar dos objetos de la clase Complejo dentro del método compareTo escribimos

```
@Override
public int compareTo(Complejo c) {
 if(real > c.real){
  return 1;
 }
 else if(real < c.real){
  return -1;
```

```
 }
 else {
  if (imaginaria > c.imaginaria){
   return 1;
  }
  else if(imaginaria < c.imaginaria){
   return -1;
  }
  else {
   return 0;
  }
 }
}
```

PROPIEDADES O ATRIBUTOS STATIC

Los valores de los atributos **static** son comunes a todos los objetos de una clase. Es decir, todos los objetos de la clase comparten el valor de ese elemento. Si un objeto cambia su valor lo cambia para todos los objetos de la clase.

Un ejemplo de atributo static sería un atributo que llevara el recuento de los objetos que se han creado.

Por ejemplo, podemos crear la clase Recuento que lleva un recuento de los objetos de esa clase que se han creado. Para ello tiene una variable static de tipo int y de nombre recuento que inicialmente vale 0.

```
public class Recuento{
...
private static int recuento=0;
...
}
```

Hay que tener en cuenta que cada vez que se cree un nuevo objeto de la clase (en cada constructor de la clase) deberemos incrementar el valor de esa variable y que cada vez que se destruya un objeto de la clase (en el destructor de la clase) deberemos decrementar el valor de esa variable.

En el caso de la clase Recuento dentro del constructor por defecto escribimos

```
Recuento(){
 recuento++;
}
```

En el caso de la clase Recuento dentro del destructor escribimos

```
public void finalize(){
 recuento--;
}
```

Para mostrar por pantalla el número de objetos de la clase creamos el método **escribirRecuento**. Para ello escribimos

```
public void escribirRecuento(){
 System.out.print("Hay "+recuento+" objetos de la clase.");
```

```
}
```

EJERCICIOS DE CLASES

1. Crea la clase Recuento que tiene un atributo static de tipo int y nombre recuento que permite conocer el número de objetos de la clase. Crea o modifica los métodos de la clase para que el atributo recuento tenga siempre el valor correcto. Añade un método de nombre escribirRecuento que muestre por pantalla el número de objetos de la clase con el formato ***"Hay "+recuento+" objetos de la clase."***.

2. Crea la clase Java RecuentoMain para probar la clase Recuento.

3. Añade a la clase Java Complejo los Setters y Getters necesarios para su correcto funcionamiento.

4. Añade a la clase Java Complejo el método toString que permita mostrar el valor de sus atributos por pantalla con el formato ***real*** + " + " + ***imaginaria*** + "***i***"

5. Añade a la clase Java Complejo los métodos hashCode y equals que permitan comparar objetos de la clase complejo.

6. Añade a la clase Java Complejo el interfaz comparable y el método compareTo para permitir comparar objetos de la clase complejo.

7. Añade a la clase Java Complejo el método leer para permitir la lectura de objetos de la clase complejo por teclado usando objetos de la clase Scanner.

8. Añade a la clase Java ComplejoMain instrucciones para probar las nuevas características de la clase Complejo.

9. Crea una nueva clase Java de nombre **Racional** que constará de dos atributos de tipo int numerador y denominador y de los siguientes métodos:
   - Un constructor por defecto que tenga como valores por defecto 0 para el numerador y 1 para el denominador.
   - Un constructor que permita dar valores a cada uno de los atributos de la clase.
   - Un constructor que permita dar valor sólo al numerador.
   - Un constructor copia.
   - Setters y Getters necesarios para el correcto funcionamiento de la clase.
   - Método toString que devuelve un String con el formato **numerador** + "/" + **denominador**. Ej: 2/3
   - Métodos equals, hashCode, y compareTo para comparar objetos de la clase.
   - Método **leer** que lea por teclado usando un objeto de la clase Scanner los datos necesarios para crear un objeto de la clase controlando los posibles errores.

10. Crea una nueva clase Java de nombre **RacionalMain** que pruebe todas las nuevas características de la clase Racional.

11. Crea una nueva clase Java de nombre **Fecha** que constará de los atributos **dia**, **mes**, y **año** de tipo **int** además de los siguientes métodos:

    - Un constructor por defecto que tenga como valores por defecto 1 para dia, 1 para mes, y 2016 para año.
    - Un constructor que permita dar valores a cada uno de los atributos de la clase.
    - Un constructor copia.
    - Setters y Getters necesarios para el correcto funcionamiento de la clase.
    - Método toString que devuelve un String con el formato **dia** + "/" + **mes** + "/" + **año**. Ej: 1/1/2016.

- Métodos equals, hashCode, y compareTo para comparar objetos de la clase.
- Método **leer** que lea por teclado usando un objeto de la clase Scanner los datos necesarios para crear un objeto de la clase controlando los posibles errores.

12. Crea una nueva clase Java de nombre **FechaMain** que pruebe todas las nuevas características de la clase Fecha. Compruebo que solo se pueda cambiar el mes si se introduce un valor de mes correcto.

13. Crea una nueva clase Java de nombre **Persona** que constará de los atributos dni, nombre, y apellidos de tipo String, y fechanacimiento de tipo Fecha además de los siguientes métodos:

- Un constructor por defecto que tenga como valores por defecto "" para los datos de tipo String y el valor por defecto que corresponda para la fecha.
- Un constructor que permita dar valores a cada uno de los atributos de la clase. Para los atributos de tipo objeto se crearán nuevos objetos con los valores que se pasen. (Cuidado con Fecha).
- Un constructor copia. Para los atributos de tipo objeto se crearán nuevos objetos con los valores que se pasen.
- Setters y Getters necesarios para el correcto funcionamiento de la clase.
- Método toString que devuelve un String con el formato **dni** + " " + **nombre** + " " + **apellidos** + " " + **fechanacimiento** . Ej: "11111111A José María De Miguel Ajamil 1/1/2014"
- Métodos equals, hashCode. Dos personas son iguales si su dni es el mismo.
- Método compareTo que devuelve los valores correspondientes a comparar únicamente el campo dni de los objetos que se comparan.
- Método **leer** que lea por teclado usando un objeto de la clase Scanner los datos necesarios para crear un objeto de la clase controlando los posibles errores.

14. Crea una nueva clase Java de nombre **PersonaMain** que pruebe todas las nuevas características de la clase Persona.

15. Crea una nueva clase Java de nombre **Asignatura** que constará de los atributos **codigo**, y **descripcion** de tipo String, y **nota** de tipo double además de los siguientes métodos:

- Un constructor por defecto que tenga como valores por defecto "" para los datos de tipo String y el valor por defecto que corresponda para la nota por ejemplo 0.0.
- Un constructor que permita dar valores a cada uno de los atributos de la clase. Para los atributos de tipo objeto se crearán nuevos objetos con los valores que se pasen.
- Un constructor copia. Para los atributos de tipo objeto se crearán nuevos objetos con los valores que se pasen.
- Setters y Getters necesarios para el correcto funcionamiento de la clase.
- Método toString que devuelve un String con el formato **codigo** + " - " + **nota**. Ej: "PROG - 8.0"
- Métodos equals, hashCode. Dos objetos de la clase Asignatura son iguales si el codigo y la nota son iguales
- Método compareTo que devuelve los valores correspondientes a comparar el campo codigo de los objetos que se comparan y en caso que el código coincida los valores correspondientes a comparar el campo nota.
- Método **leer** que lea por teclado usando un objeto de la clase Scanner los datos necesarios para crear un objeto de la clase controlando los posibles errores. Presta atención al hecho de que después de leer el valor de la nota el buffer del teclado debe de quedar vacío para permitir la lectura de otra asignatura sin problemas (*teclado.nextLine();*).

16. Crea una nueva clase Java de nombre **AsignaturaMain** que pruebe todas las nuevas características de la clase Asignatura. Comprueba que se pueden leer sin problemas los datos de dos asignaturas de manera consecutiva.

17. Crea una nueva clase Java de nombre **ComplejoBucle** que pide números complejos hasta que se introduce un número complejo con la parte real negativa.

18. Crea una nueva clase Java de nombre **RacionalBucle** que pide números racionales hasta que se introduce un número racional con el numerador negativo.

19. Crea una nueva clase Java de nombre **RacionalBucleCero** que pide números racionales hasta que se introduce un número racional con el denominador igual a 0.

20. Crea una nueva clase Java de nombre **FechaBucle** que pide fechas hasta que se introduce el año 0.

21. Crea una nueva clase Java de nombre **AsignaturaBucle** que pide asignaturas hasta que se introduce una nota negativa.

22. Crea una nueva clase Java de nombre **ComplejoArray** que pide números complejos hasta que se introduce un número complejo con la parte real negativa que no será tenido en cuenta. Cada complejo introducido lo mete en la última posición del array, si el array no está lleno. Al finalizar la introducción muestra el contenido del array por pantalla. Inicialmente el array tendrá 10 posiciones.

23. Crea una nueva clase Java de nombre **RacionalArray** que pide números racionales hasta que se introduce un número racional con el numerador negativo que no será tenido en cuenta. Cada racional introducido lo mete en la última posición del array, si el array no está lleno. Al finalizar la introducción muestra el contenido del array por pantalla. Inicialmente el array tendrá 10 posiciones.

24. Crea una nueva clase Java de nombre **FechaArray** que pide números fechas hasta que se introduce el año 0 que no será tenido en cuenta. Cada fecha introducida la mete en la última posición del array, si el array no está lleno. Al finalizar la introducción muestra el contenido del array por pantalla. Inicialmente el array tendrá 10 posiciones.

25. Crea una nueva clase Java de nombre **AsignaturaArray** que pide asignaturas hasta que se introduce una nota negativa que no será tenida en cuenta. Cada asignatura introducida la mete en la última posición del array, si el array no está lleno. Al finalizar la introducción muestra el contenido del array por pantalla. Inicialmente el array tendrá 10 posiciones.

26. Crea la clase **ComplejoMenu** que muestra un menú por pantalla con las siguientes opciones para manipular datos de la clase Complejo
    - Añadir Complejo. Pide un Complejo y lo añade al array de manera ordenada si queda sitio en el array.
    - Buscar Complejo. Pide un Complejo y lo busca en el array. Muestra un mensaje con la posición en que se encuentra o un mensaje de error si no se encuentra en el array.
    - Borrar Complejo. Pide un Complejo y lo elimina, si es que existe, del array.
    - Listar Array. Muestra todos los elementos del array por pantalla, si es que tiene elementos.
    - Salir. Realiza las operaciones necesarias para la correcta finalización del programa.

27. Crea la clase **RacionalMenu** que muestra un menú por pantalla con las siguientes opciones para manipular datos de la clase Racional
    - Añadir Racional. Pide un Racional y lo añade al array de manera ordenada si queda sitio en el array.
    - Buscar Racional. Pide un Racional y lo busca en el array. Muestra un mensaje con la posición en que se encuentra o un mensaje de error si no se encuentra en el array.
    - Borrar Racional. Pide un Racional y lo elimina, si es que existe, del array.
    - Listar Array. Muestra todos los elementos del array por pantalla, si es que tiene elementos.
    - Salir. Realiza las operaciones necesarias para la correcta finalización del programa.

28. Crea la clase **FechaMenu** que muestra un menú por pantalla con las siguientes opciones para manipular datos de la clase Fecha
    - Añadir Fecha. Pide una Fecha y la añade al array de manera ordenada si queda sitio en el array.
    - Buscar Fecha. Pide una Fecha y la busca en el array. Muestra un mensaje con la posición en que se encuentra o un mensaje de error si no se encuentra en el array.
    - Borrar Fecha. Pide una Fecha y la elimina, si es que existe, del array.
    - Listar Array. Muestra todos los elementos del array por pantalla, si es que tiene elementos.
    - Salir. Realiza las operaciones necesarias para la correcta finalización del programa.

29. Crea la clase **AsignaturaMenu** que muestra un menú por pantalla con las siguientes opciones para manipular datos de la clase Asignatura
    - Añadir Asignatura. Pide una Asignatura y la añade al array de manera ordenada si queda sitio en el array.
    - Buscar Asignatura. Pide una Asignatura y la busca en el array. Muestra un mensaje con la posición en que se encuentra o un mensaje de error si no se encuentra en el array.
    - Borrar Asignatura. Pide una Asignatura y la elimina, si es que existe, del array.
    - Listar Array. Muestra todos los elementos del array por pantalla, si es que tiene elementos.
    - Salir. Realiza las operaciones necesarias para la correcta finalización del programa.