

## TEMA8. PROGRAMACIÓN ORIENTADA A OBJETOS I

Programación  
CFGS DAW

Profesores:

Carlos Cacho

Raquel Torres

[carlos.cacho@ceedcv.es](mailto:carlos.cacho@ceedcv.es)

[raquel.torres@ceedcv.es](mailto:raquel.torres@ceedcv.es)

## Licencia



**Reconocimiento - NoComercial - CompartirIgual (by-nc-sa):** No se permite un uso comercial de la obra original ni de las posibles obras derivadas, la distribución de las cuales se debe hacer con una licencia igual a la que regula la obra original.

## Nomenclatura

A lo largo de este tema se utilizarán distintos símbolos para distinguir elementos importantes dentro del contenido. Estos símbolos son:



Importante



Atención



Interesante

# ÍNDICE DE CONTENIDO

<b>1.Introducción.....</b>	<b>4</b>
<b>2.Fundamentos de una clase.....</b>	<b>4</b>
<b>3.Objetos.....</b>	<b>6</b>
3.1 Declaración.....	6
3.2 Operador new.....	6
3.3 Asignación de variables de referencia a objetos.....	6
3.4 Recolector de basura.....	7
<b>4.Tipos de acceso a los miembros de una clase.....</b>	<b>7</b>
<b>5.Métodos.....</b>	<b>7</b>
<b>6.Constructores.....</b>	<b>8</b>
<b>7.Constantes de clase y de objeto.....</b>	<b>9</b>
<b>8.Arrays de objetos.....</b>	<b>9</b>
<b>9.Ejemplos.....</b>	<b>10</b>
9.1 Ejemplo 1.....	10
9.2 Ejemplo 2.....	12
9.3 Ejemplo 3.....	14
9.4 Ejemplo 4.....	19
9.5 Ejemplo 5.....	20
9.6 Ejemplo 6.....	21
<b>10.Agradecimientos.....</b>	<b>23</b>

## UD08. PROGRAMACIÓN ORIENTADA A OBJETOS I

### 1. INTRODUCCIÓN

La Programación Orientada a Objetos (POO) hace que los problemas sean más sencillos, al permitir dividir el problema. Esta división se hace en objetos, de forma que cada objeto funcione de forma totalmente independiente.



Un **objeto** es un elemento del programa que posee sus propios datos y su propio funcionamiento.



Una **clase** describe un grupo de objetos que contienen una información similar (atributos) y un comportamiento común (métodos).



Antes de poder utilizar un objeto, se debe definir su clase. La clase es la definición de un tipo de objeto.

Al definir una clase lo que se hace es indicar cómo funciona un determinado tipo de objetos. Luego, a partir de la clase, podremos crear objetos de esa clase.

Las propiedades de la POO son las siguientes:

- Encapsulamiento. Una clase se compone tanto de variables (atributos) como de funciones y procedimientos (métodos). De hecho no se pueden definir variables (ni funciones) fuera de una clase (es decir no hay variables globales).
- Ocultación. Hay una zona oculta al definir las clases (zona privada) que sólo es utilizada por esa clase y por alguna clase relacionada. Hay una zona pública (llamada también interfaz de la clase) que puede ser utilizada por cualquier parte del código.
- Polimorfismo. Cada método de una clase puede tener varias definiciones distintas. En el caso del juego parchís: partida.empezar(4) empieza una partida para cuatro jugadores, partida.empezar(rojo, azul) empieza una partida de dos jugadores para los colores rojo y azul; estas son dos formas distintas de emplear el método empezar, que es polimórfico.
- Herencia. Una clase puede heredar propiedades (atributos y métodos) de otra.

## 2. FUNDAMENTOS DE UNA CLASE

Una clase describe un grupo de objetos que contienen una información similar (atributos) y un comportamiento común (métodos).

Las definiciones comunes (nombre de la clase, los nombres de los atributos, y los métodos) se almacenan una única vez en cada clase, independientemente de cuántos objetos de esa clase estén presentes en el sistema.

Hay que pensar en una clase como un molde. A través de las clases se obtienen los objetos en sí. Es decir antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición incluye:

- Atributos. Los datos miembros de esa clase. Los datos pueden ser *public* (accesibles desde otra clase), *private* (sólo accesibles por código de su propia clase) o *protected* (accesibles por las subclases).
- Métodos. Las funciones miembro de la clase. Son las acciones u operaciones que puede realizar la clase. Al igual que los atributos pueden ser *public*, *private* o *protected*.
- Código de inicialización. Para crear una clase normalmente hace falta realizar operaciones previas (es lo que se conoce como el constructor de la clase).
- Otras clases. Dentro de una clase se pueden definir otras clases (clases internas).

Si hacemos uso de la notación UML<sup>1</sup> la **estructura de una clase** sería:

Nombre de la clase
Atributos
Métodos

Y la **sintaxis de una clase** es la siguiente:

```
[acceso] class nombreDeClase {  
    [acceso] [static] tipo atributo1;  
    [acceso] [static] tipo atributo2;  
    ...más atributos...  
  
    [acceso] [static] tipo método1(listaDeParametros) {  
        ...código del método...  
    }  
    [acceso] [static] tipo método2(listaDeParametros) {  
        ...código del método...  
    }  
    ... más métodos...
```

<sup>1</sup> [Unified Modeling Language](#)

```
}
```

Los *atributos* que se definen dentro de una clase se denominan *variables de instancia*, el *código* que contienen se llama *método*, y en conjunto, los *métodos* y *atributos* definidos en una clase se llaman *miembros de la clase*.

Las variables o atributos definidos en una clase se llaman variables de instancia porque cada instancia de clase, es decir cada objeto de la clase, contiene su copia de esas variables. Por lo tanto, los datos de un objeto están separados y son individuales de los datos de otro objeto.

La palabra opcional *static* sirve para hacer que el método o el atributo a la que precede se pueda *utilizar de manera genérica*<sup>2</sup> (más adelante se hablará de clases genéricas), los atributos y métodos así definidos se llaman *atributos de clase* y *métodos de clase* respectivamente.

### 3. OBJETOS

#### 3.1 Declaración

Cuando creamos una clase, se crea un nuevo tipo de datos que puede utilizarse para declarar objetos de ese mismo tipo. Sin embargo, para obtener los objetos de una clase es necesario un proceso de dos etapas:

- En primer lugar, hay que *declarar una variable del tipo de la clase*.
- En segundo lugar, se necesitará una *copia física del objeto y asignarla a esa variable*. Esto puede hacerse utilizando el operador *new*. Este operador asigna dinámicamente (es decir, durante tiempo de ejecución) memoria a un objeto y devuelve una referencia. Esta referencia se almacena en una variable.

Ejemplo:

```
Cubo miCubo; // declara la referencia a un objeto. Variable miCubo de tipo Cubo
miCubo = new Cubo(); // Reserva espacio para el objeto.
```

#### 3.2 Operador *new*

El operador *new* asigna automáticamente memoria para un objeto. Su forma general es ésta:



```
variable = new nombre_de_clase();
```

Donde *variable* es una variable del mismo tipo de la clase que se ha creado, y el *nombre\_de\_clase*, el nombre de la clase que se está instanciando.

El *nombre\_de\_clase* seguido de paréntesis especifica el *constructor* de la clase.

El *constructor* es el método que permite iniciar el objeto.

<sup>2</sup> Es decir, se puede hacer uso de una llamada estática sin necesidad de crear una instancia para acceder, por ejemplo, a algún método de la clase.

### 3.3 Asignación de variables de referencia a objetos

Las variables de referencia a objetos actúan de forma diferente a lo que cabría esperar cuando tiene lugar una asignación. Por [ejemplo](#), ¿qué hace el siguiente fragmento de código?

```
Cubo c1 = new Cubo();  
Cubo c2 = c1;
```

Podríamos pensar que a c2 se le asigna una referencia a una copia del objeto que referencia c1, es decir que c1 y c2 hacen referencia a objetos separados y distintos, pero esto no es así. Cuando este código se ejecute, c1 y c2 harán referencia al mismo objeto. Por lo tanto cualquier cambio que se haga al objeto referenciado a través de c2 afectará al objeto al que referencia c1.

### 3.4 Recolector de basura

En Java hay un recolector de basura (*garbage collector*) que se encarga de gestionar los objetos que se dejan de usar y de eliminarlos de memoria. Este proceso es automático e impredecible y trabaja en un hilo (*thread*) de baja prioridad. Por lo general ese proceso de recolección de basura, trabaja cuando detecta que un objeto hace demasiado tiempo que no se utiliza en un programa. Esta eliminación depende de la máquina virtual de Java, en casi todas las recolecciones se realiza periódicamente en un determinado lapso de tiempo.

(Ver [ejemplos 1 y 2](#)).

## 4. TIPOS DE ACCESO A LOS MIEMBROS DE UNA CLASE

Los principios de la programación orientada a objetos dicen que para mantener la encapsulación en los objetos, debemos aplicar el especificador **public** a las funciones miembro que formen la interfaz pública y denegar el acceso a los datos miembro usados por esas funciones mediante el especificador **private**. Se utilizará **protected** cuando queramos dar acceso a las subclases de la clase (herencia).

⚡ En un **paquete**, que es un agrupamiento lógico de clases en un mismo directorio, los atributos y los métodos de estas clases son **públicos** para el resto de clases existentes en el mismo paquete, y **privados** para cualquier clase que se encuentre fuera (en caso de no especificar lo contrario).

Cuando un paquete no está definido, se dice que la clase pertenece al paquete por defecto, por lo tanto se aplicará el calificador **public** al resto de las clases cuyos ficheros se encuentren en el mismo directorio.

(Ver [ejemplo 3](#)).

## 5. MÉTODOS

La forma general de un método es la siguiente:

```
tipo nombre_del_método(lista de parámetros) {  
    // cuerpo del método  
}
```

Donde:

- el *tipo* especifica el tipo de datos que devuelve el método. Puede tratarse de cualquier tipo válido, incluyendo los tipos de clases que crea el programador. Si el método no devuelve un valor, el tipo devuelto será *void*.
- el nombre del método lo especifica *nombre\_del\_método*, que puede ser cualquier identificador válido diferente a aquellos ya usados por otros elementos del programa.
- la *lista de parámetros* es una secuencia de pares de tipo e identificador separados por comas. Los parámetros son variables que reciben el valor de los argumentos que se pasa al método cuando se llama. Si el método no tiene parámetros, la lista de parámetros estará vacía.

Los métodos que devuelven un valor diferente a *void* utilizan la sentencia *return* para devolver el valor:

```
return valor;
```

Aunque sería perfectamente válido crear una clase que únicamente contenga datos, esto raramente ocurre. La mayoría de las veces, los métodos se utilizan para acceder a variables de instancia definidas por la clase.

Además de definir métodos que proporcionen acceso a los datos (se oculta o abstrae la estructura interna de los datos) pueden definirse métodos para ser utilizados internamente por la clase.

## 6. CONSTRUCTORES

Puede resultar una tarea bastante pesada inicializar todas las variables de una clase cada vez que se crea una instancia. Incluso cuando se añaden funciones adecuadas, es más sencillo y preciso realizar todas las inicializaciones cuando el objeto se crea por primera vez. Como el proceso de inicialización es tan común, Java permite que los objetos se inicialicen cuando se crean. Esta inicialización automática se lleva a cabo mediante un constructor.



Un **constructor** es un método especial que no devuelve nunca un valor,



siempre devuelve una referencia a una instancia de la clase y es llamado automáticamente al crear un objeto de una clase, es decir al usar la instrucción *new*.

Un *constructor* no es más que un método que tiene el mismo nombre que la clase. En general, cuando se crea una instancia de una clase, no siempre se desea pasar los parámetros de inicialización al construirla, por ello existe un tipo especial de constructores, que son los llamados **constructores por defecto**.

Estos constructores **no llevan parámetros asociados**, e **inician los datos** asignándoles valores por defecto.

Cuando no se llama a un constructor de forma explícita, Java crea un constructor por defecto y lo llama cuando se crea un objeto. Este constructor por defecto asigna a las variables de instancia un valor inicial igual a **cero a las variables numéricas** y **null** a todas las referencias a objetos.

Una vez se defina un constructor para la clase se deja de utilizar el constructor por defecto.

⚡ Existe la referencia *this*, que apunta a la instancia que llama al método, y siempre esta accesible, es utilizado por las funciones miembro para acceder a los datos del objeto.

(Ver [ejemplo 4](#)).

## 7. CONSTANTES DE CLASE Y DE OBJETO

Los miembros constantes se definen en Java a través de la palabra reservada *final*, mientras que los miembros de clase se definen mediante la palabra reservada *static*. De esta manera, si consideramos que los miembros son atributos, tenemos cuatro posibles combinaciones en Java para indicar si un atributo es constante:

- Atributos *static final*: son valores constantes de clase, asignados en la inicialización. Son comunes para todos los objetos de la clase, e incluso pueden invocarse, aunque no exista ninguna instancia.
- Atributos *final*: son valores constantes, pero potencialmente distintos en cada una de las instancias. Su valor se inicializa en la fase de construcción del objeto y no pueden ser modificados durante el tiempo de vida de éste.
- Atributos *static*: toman valores comunes a todos los objetos existentes y potencialmente variables. Se pueden considerar variables globales.
- Resto de atributos: atributos variables, diferentes en cada objeto de la clase.

⚡ Los **atributos globales** de la clase deben ser **limitados**, ya que pueden dar lugar a errores de muy difícil depuración, además de ir contra el concepto de la programación orientada a objetos.

Las constantes de objeto se definen mediante la palabra reservada *final*. Se trata de atributos que toman el valor en el constructor del objeto, un valor que no puede ser modificado en el resto del programa. De esta manera, a cada objeto corresponde un atributo de ese tipo, pero invariable para él. Este tipo de atributos sirve para identificar cada objeto de forma única.

Ver [ejemplo 5](#).

## 8. ARRAYS DE OBJETOS

En Java hay dos sintaxis posibles para definir un **array de objetos**:

```
NombreDeLaClase [] Objetos;  
NombreDeLaClase Objetos[];
```

De esta manera se crea la referencia al array de Objetos.

Para crear la instancia del objeto array, debemos especificar el tamaño deseado con la sintaxis:

```
Objetos = new NombreDeLaClase[n]; // n es el nº de referencias a Objetos
```

Podemos resumir estas dos sentencias en una sola:

```
NombreDeLaClase [] Objetos = new NombreDeLaClase[n];  
NombreDeLaClase Objetos[] = new NombreDeLaClase[n];
```

Hay que tener en cuenta que, cuando se crea el array, el compilador genera una referencia para cada uno de los elementos que lo integran, aunque no existan todavía las instancias de ninguno de ellos.

⚡ Para crear los objetos hay que **llamar** explícitamente al **constructor por cada elemento creado**:

```
Objeto[i] = new NombreDeLaClase();
```

(Ver [ejemplo 6](#)).

## 9. EJEMPLOS

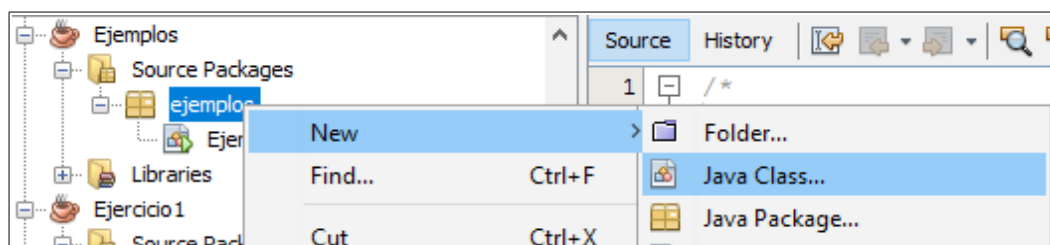
### 9.1 Ejemplo 1

En este ejemplo vamos a implementar la clase *Articulos*. Esta clase representa cada objeto con los siguientes atributos:

- codigo\_articulo
- titulo
- formato
- precio\_alquiler

y define tres métodos que permiten calcular, respectivamente, el precio de alquiler de un día, de dos días y una semana.

Lo primero que necesitaremos, y esto lo haremos por cada clase que necesitemos en todos los ejemplos, será crearnos una nueva clase en Java. Para ello pincharemos con el botón derecho sobre el paquete donde vayamos a tener las clases y luego en *NEW > Java Class*. Crearemos una clase llamada Artículo.



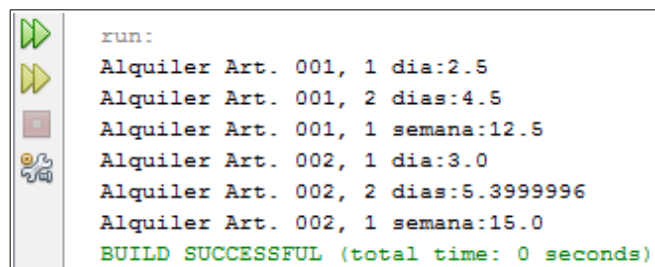
```
12 public class Artículo {
13     // Atributos de la clase
14     String cod;
15     String titulo;
16     String formato;
17     float precio_alquiler;
18
19     // Métodos de la clase
20     float precio1() {
21         return (precio_alquiler);
22     }
23
24     float precio2() {
25         float precio_total;
26
27         precio_total = precio_alquiler * 1.80f;
28
29         return (precio_total);
30     }
31
32     float precio_semana() {
33         float precio_total;
34
35         precio_total = precio_alquiler * 5;
36
37         return (precio_total);
38     }
39 }
40
```

El método *precio1* devuelve el valor del precio de alquiler del artículo. El método *precio2* calcula el precio de alquiler de dos días haciendo un descuento del 20%. Y por último el método *precio\_semana* calcula el precio de una semana multiplicando por 5 el precio de alquiler.

Con el operador *new* crearemos una instancia de la clase *Articulo*, como por el momento la clase no tiene constructores se invocará al constructor por defecto de la clase. RECUERDA: Cuando instanciamos una clase estamos creando un objeto de esta clase.

```
12 public class Ejemplos {
13
14     public static void main(String[] args) {
15         // Creamos dos artículos
16         Artículo articulo1 = new Artículo();
17         Artículo articulo2 = new Artículo();
18
19         // Le damos valores a sus atributos
20         articulo1.cod = "001";
21         articulo1.titulo = "Titulo1";
22         articulo1.formato = "DVD";
23         articulo1.precio_alquiler = 2.50f;
24
25         articulo2.cod = "002";
26         articulo2.titulo = "Titulo2";
27         articulo2.formato = "DVD";
28         articulo2.precio_alquiler = 3;
29
30         // Utilizamos sus métodos
31         System.out.println("Alquiler Art. " + articulo1.cod + ", 1 dia:" + articulo1.precio1());
32         System.out.println("Alquiler Art. " + articulo1.cod + ", 2 dias:" + articulo1.precio2());
33         System.out.println("Alquiler Art. " + articulo1.cod + ", 1 semana:" + articulo1.precio_semana());
34         System.out.println("Alquiler Art. " + articulo2.cod + ", 1 dia:" + articulo2.precio1());
35         System.out.println("Alquiler Art. " + articulo2.cod + ", 2 dias:" + articulo2.precio2());
36         System.out.println("Alquiler Art. " + articulo2.cod + ", 1 semana:" + articulo2.precio_semana());
37     }
38
39 }
```

La salida es:

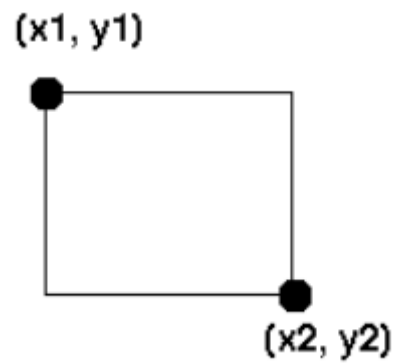


```
run:
Alquiler Art. 001, 1 dia:2.5
Alquiler Art. 001, 2 dias:4.5
Alquiler Art. 001, 1 semana:12.5
Alquiler Art. 002, 1 dia:3.0
Alquiler Art. 002, 2 dias:5.3999996
Alquiler Art. 002, 1 semana:15.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

## 9.2 Ejemplo 2

En este ejemplo vamos a implementar la clase *Cuadrado*, que representa cada objeto mediante dos puntos 2D, y define tres métodos que permiten calcular , respectivamente, la diagonal, el perímetro y el área.

El criterio de representación toma las coordenadas horizontales (x) crecientes de izquierda a derecha, y las verticales (y) crecientes de arriba abajo.

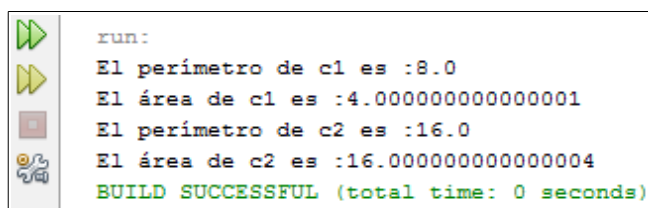


El código sería:

```
12 public class Cuadrado {
13     // Atributos
14     double x1, y1, x2, y2;
15
16     // Métodos
17     double CalcularDiagonal()
18     {
19         return Math.sqrt((x2-x1)*(x2-x1)+(y2-y1)*(y2-y1));
20     }
21
22     double CalcularPerimetro() {
23         double diagonal = CalcularDiagonal();
24         double lado = diagonal/Math.sqrt(2);
25
26         return (4*lado);
27     }
28     double CalcularArea()
29     {
30         double diagonal =CalcularDiagonal();
31
32         return (0.5*diagonal*diagonal);
33     }
34 }
```

```
12 public class Ejemplos {
13
14     public static void main(String[] args) {
15         // Creamos dos cuadrados
16         Cuadrado c1 = new Cuadrado();
17         Cuadrado c2 = new Cuadrado();
18
19         // Les damos valores
20         c1.x1=2; c1.y1=2; c1.x2=4; c1.y2=4;
21         c2.x1=1; c2.y1=1; c2.x2=5; c2.y2=5;
22
23         // Usamos sus métodos
24         System.out.println("El perímetro de c1 es :" + c1.CalcularPerimetro());
25         System.out.println("El área de c1 es :" + c1.CalcularArea());
26         System.out.println("El perímetro de c2 es :" + c2.CalcularPerimetro());
27         System.out.println("El área de c2 es :" + c2.CalcularArea());
28     }
29 }
```

Y la salida:



```
run:
El perímetro de c1 es :8.0
El área de c1 es :4.0000000000000001
El perímetro de c2 es :16.0
El área de c2 es :16.0000000000000004
BUILD SUCCESSFUL (total time: 0 seconds)
```

### 9.3 Ejemplo 3

En este ejemplo vamos a aplicar el principio de **encapsulamiento** haciendo *private* los atributos de la clase y *public* los métodos. Para ello vamos a modificar la clase Artículo del Ejemplo 1.

Pero ahora no podemos acceder a los atributos privados de la clase desde fuera de ella, para poder leer los datos almacenados en los atributos, debemos definir un método por cada atributo, que deseemos recuperar, que nos devuelva el contenido de dicho atributo (son los métodos “*get*”).

De la misma forma, para especificar los datos necesitamos un método, definido como público, que reciba los valores deseados y los utilice para modificar los atributos del objeto (*modificaValores*).

El código sería:

```
12 public class Artículo {
13     // Atributos de la clase
14     private String cod;
15     private String titulo;
16     private String formato;
17     private float precio_alquiler;
18
19     // Métodos de la clase
20     public float precio1(){
21         return (getPrecio_alquiler());
22     }
23
24     public float precio2(){
25         float precio_total;
26
27         precio_total = getPrecio_alquiler() * 1.80f;
28
29         return (precio_total);
30     }
31
32     public float precio_semana(){
33         float precio_total;
34
35         precio_total = getPrecio_alquiler() * 5;
36
37         return (precio_total);
38     }
39
40     public void modificaValores(String cod_p, String titulo_p, String formato_p, float precio_p)
41     {
42         cod=cod_p;
43         titulo=titulo_p;
44         formato=formato_p;
45         precio_alquiler=precio_p;
46     }
47
48     public String getCod()
49     {
50         return cod;
51     }
52
53     public String getTitulo()
54     {
55         return titulo;
56     }
57
58     public String getFormato()
59     {
60         return formato;
61     }
62
63     public float getPrecio_alquiler()
64     {
65         return precio_alquiler;
66     }
67 }
```



```

12 public class Ejemplos {
13
14     public static void main(String[] args) {
15         // Creamos dos articulos
16         Artículo articulo1 = new Artículo();
17         Artículo articulo2 = new Artículo();
18
19         // Le damos valores a sus atributos
20         articulo1.modificaValores("001","Titulo1","DVD",2.5f);
21         articulo2.modificaValores("002","Titulo2","DVD",3f);
22
23         // Utilizamos sus métodos
24         System.out.println("Alquiler Art. " + articulo1.getCod() + ", 1 dia:" + articulo1.precio1());
25         System.out.println("Alquiler Art. " + articulo1.getCod() + ", 2 dias:" + articulo1.precio2());
26         System.out.println("Alquiler Art. " + articulo1.getCod() + ", 1 semana:" + articulo1.precio_semana());
27         System.out.println("Alquiler Art. " + articulo2.getCod() + ", 1 dia:" + articulo2.precio1());
28         System.out.println("Alquiler Art. " + articulo2.getCod() + ", 2 dias:" + articulo2.precio2());
29         System.out.println("Alquiler Art. " + articulo2.getCod() + ", 1 semana:" + articulo2.precio_semana());
30     }
31 }
32

```

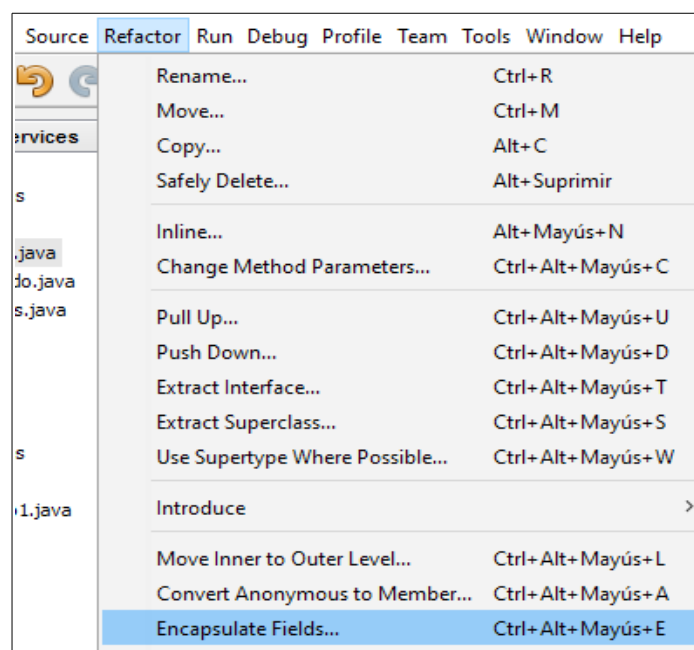
Y la salida:

```

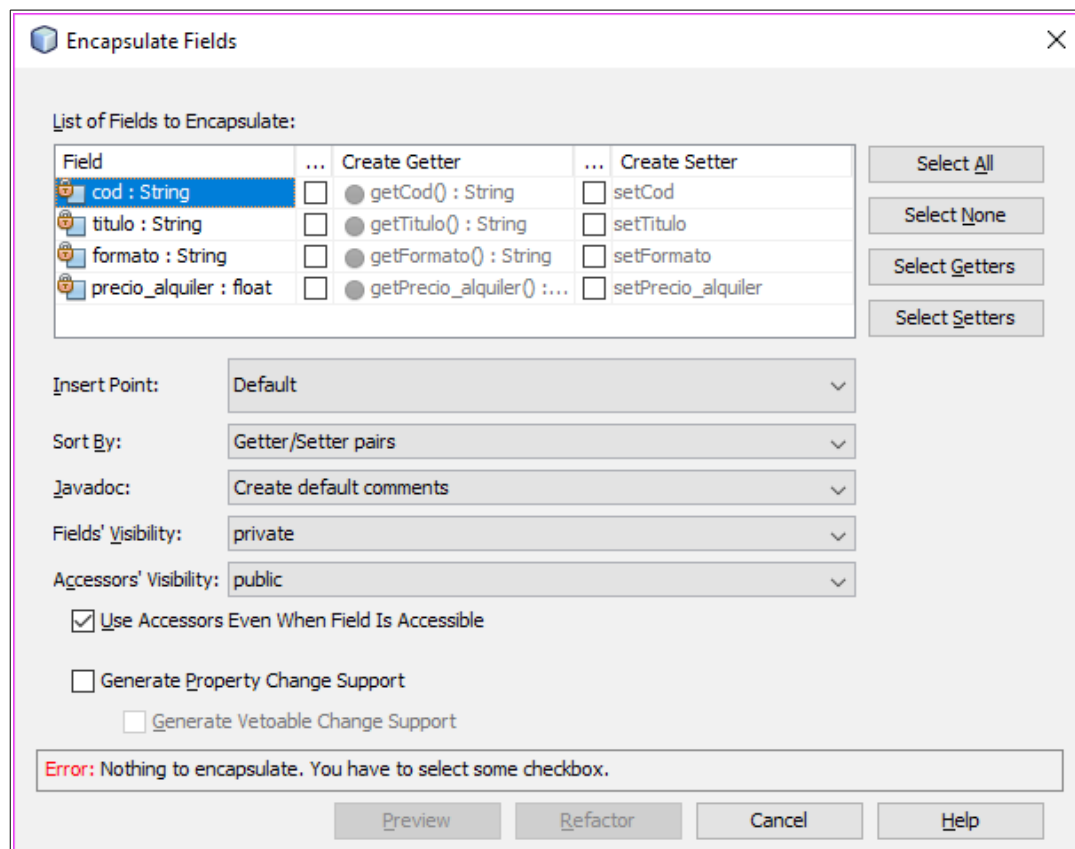
run:
Alquiler Art. 001, 1 dia:2.5
Alquiler Art. 001, 2 dias:4.5
Alquiler Art. 001, 1 semana:12.5
Alquiler Art. 002, 1 dia:3.0
Alquiler Art. 002, 2 dias:5.3999996
Alquiler Art. 002, 1 semana:15.0
BUILD SUCCESSFUL (total time: 0 seconds)

```

Otra forma de crear los métodos de encapsulamiento es utilizando el *Refactor* de Netbeans. Para ello tenemos que estar en una clase que tenga atributos, pinchamos en el menú *Refactor* y luego en *Encapsulate Fields*.



Luego seleccionamos los métodos para obtener los atributos (*Get*) que queramos o para establecer su valor (*Set*) y pinchamos en *Refactor*.



Automáticamente aparecerán los métodos en el código.

## 9.4 Ejemplo 4

En este ejemplo vamos a ver cómo crear un constructor de clase. Para ello vamos a basarnos en la clase *Articulo* creada anteriormente y le vamos a añadir el constructor.

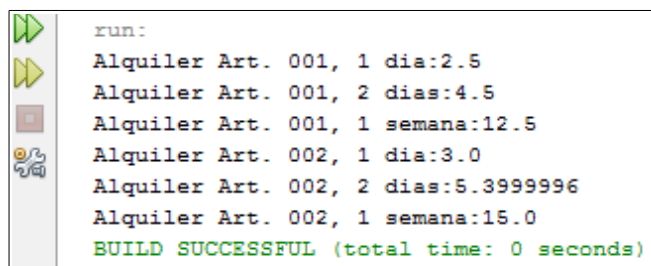
El código sería:

```
12 public class Articulo {
13     // Atributos de la clase
14     private String cod;
15     private String titulo;
16     private String formato;
17     private float precio_alquiler;
18
19     // Constructor de la clase
20     public Articulo(String cod, String titulo, String formato, float precio_alquiler)
21     {
22         // Con 'this' estamos accediendo al objeto de la clase
23         this.cod = cod;
24         this.titulo = titulo;
25         this.formato = formato;
26         this.precio_alquiler = precio_alquiler;
27     }
28     // Métodos de la clase
29     public float precio1(){
30         return (getPrecio_alquiler());
31     }
```

El resto del código de la clase sería el mismo.

```
12 public class Ejemplos {
13
14     public static void main(String[] args) {
15         // Creamos dos artículos
16         Articulo articulo1 = new Articulo("001", "Titulo1", "DVD", 2.5f);
17         Articulo articulo2 = new Articulo("002", "Titulo2", "DVD", 3f);
18
19         // Utilizamos sus métodos
20         System.out.println("Alquiler Art. " + articulo1.getCod() + ", 1 dia:" + articulo1.precio1());
21         System.out.println("Alquiler Art. " + articulo1.getCod() + ", 2 dias:" + articulo1.precio2());
22         System.out.println("Alquiler Art. " + articulo1.getCod() + ", 1 semana:" + articulo1.precio_semana());
23         System.out.println("Alquiler Art. " + articulo2.getCod() + ", 1 dia:" + articulo2.precio1());
24         System.out.println("Alquiler Art. " + articulo2.getCod() + ", 2 dias:" + articulo2.precio2());
25         System.out.println("Alquiler Art. " + articulo2.getCod() + ", 1 semana:" + articulo2.precio_semana());
26     }
27
28 }
```

Y la salida:



```
run:
Alquiler Art. 001, 1 dia:2.5
Alquiler Art. 001, 2 dias:4.5
Alquiler Art. 001, 1 semana:12.5
Alquiler Art. 002, 1 dia:3.0
Alquiler Art. 002, 2 dias:5.3999996
Alquiler Art. 002, 1 semana:15.0
BUILD SUCCESSFUL (total time: 0 seconds)
```

### 9.5 Ejemplo 5

Para ilustrar el uso de los cualificadores *final* y *static* dentro de nuestra clase *Articulo*, vamos a definir tres atributos principales:

- la constante *IVA*.
- un atributo compartido que contabilizará el número de instancias que se hayan definido hasta entonces.
- un atributo constante de cadena (String), distinto para cada objeto, que permita identificarlos.

Definiremos *IVA* como *static* y *final*, podremos acceder a su valor mediante la expresión *Articulo.IVA*, sin necesidad de haber creado ningún objeto de la clase *Articulo*:

```
public static final double IVA = 0.16;
```

El atributo privado con el número de instancia, en realidad, se trata de una variable global accesible para todos los objetos de tipo *Articulo*:

```
private static int numero = 0;
```

Y por último, el identificador constante de cada objeto se indica con un especificador *final* en su descripción.

```
final String identificador;
```

### 9.6 Ejemplo 6

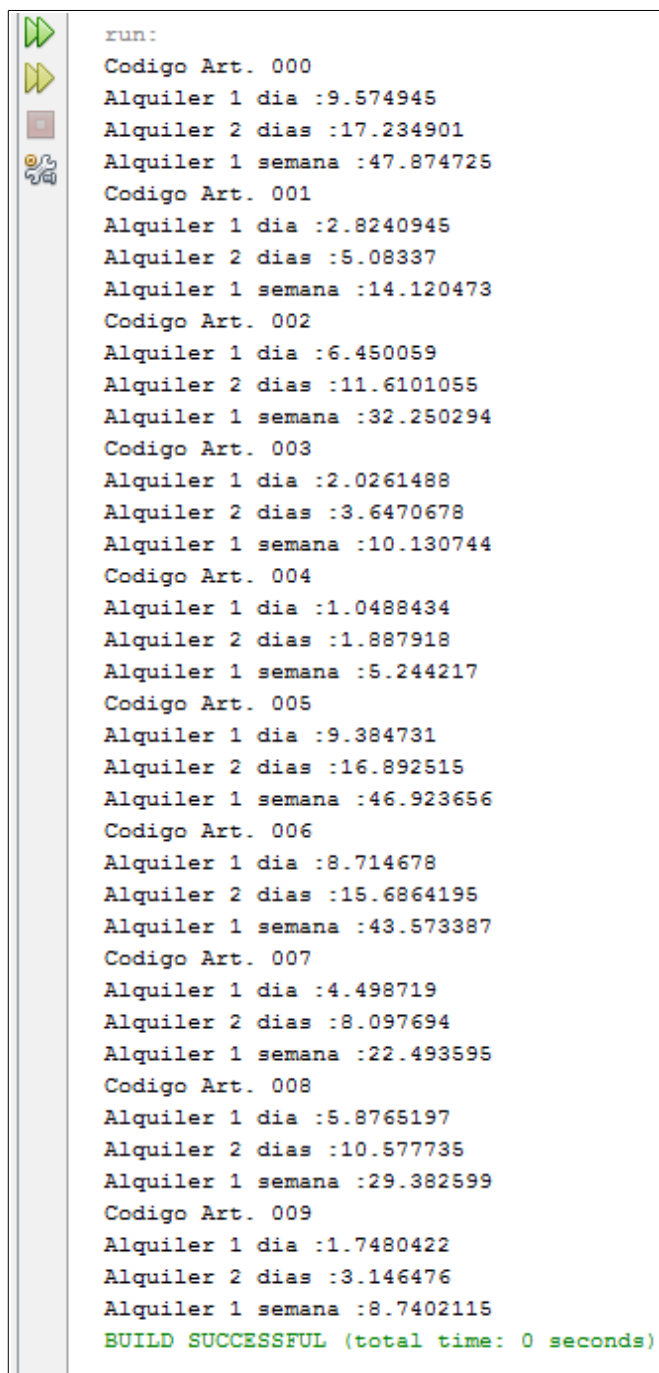
En este ejemplo vamos a crear un array (vector) que contenga diez objetos de tipo *Articulo*. Instanciaremos a todos con nombres genéricos y consecutivos y precios aleatorios.

Una vez instanciado *misArticulos* (un objeto array de referencias a *Articulo*, con tamaño 10), instanciamos cada uno de los objetos referenciados.

La clase *Articulo* no se vería modificada.

```
12 public class Ejemplos {
13
14     public static void main(String[] args) {
15
16         // Definimos el vector de Articulos de tamaño 10
17         Artículo[] misArticulos = new Artículo[10];
18         int cont;
19
20         // Para cada elemento del vector creamos el objeto Artículo
21         for (cont = 0; cont < misArticulos.length; cont++)
22             misArticulos[cont]=new Artículo("00"+cont,"Articulo "+cont,"DVD", (float)Math.random()*10);
23
24
25         // Recorremos el vector
26         for (cont = 0; cont < misArticulos.length; cont++)
27         {
28             System.out.println("Codigo Art. "+misArticulos[cont].getCod());
29             System.out.println("Alquiler 1 dia :"+misArticulos[cont].precio1());
30             System.out.println("Alquiler 2 dias :"+misArticulos[cont].precio2());
31             System.out.println("Alquiler 1 semana :"+misArticulos[cont].precio_semana());
32         }
33     }
34 }
35 }
```

La salida sería:



```
run:
Codigo Art. 000
Alquiler 1 dia :9.574945
Alquiler 2 dias :17.234901
Alquiler 1 semana :47.874725
Codigo Art. 001
Alquiler 1 dia :2.8240945
Alquiler 2 dias :5.08337
Alquiler 1 semana :14.120473
Codigo Art. 002
Alquiler 1 dia :6.450059
Alquiler 2 dias :11.6101055
Alquiler 1 semana :32.250294
Codigo Art. 003
Alquiler 1 dia :2.0261488
Alquiler 2 dias :3.6470678
Alquiler 1 semana :10.130744
Codigo Art. 004
Alquiler 1 dia :1.0488434
Alquiler 2 dias :1.887918
Alquiler 1 semana :5.244217
Codigo Art. 005
Alquiler 1 dia :9.384731
Alquiler 2 dias :16.892515
Alquiler 1 semana :46.923656
Codigo Art. 006
Alquiler 1 dia :8.714678
Alquiler 2 dias :15.6864195
Alquiler 1 semana :43.573387
Codigo Art. 007
Alquiler 1 dia :4.498719
Alquiler 2 dias :8.097694
Alquiler 1 semana :22.493595
Codigo Art. 008
Alquiler 1 dia :5.8765197
Alquiler 2 dias :10.577735
Alquiler 1 semana :29.382599
Codigo Art. 009
Alquiler 1 dia :1.7480422
Alquiler 2 dias :3.146476
Alquiler 1 semana :8.7402115
BUILD SUCCESSFUL (total time: 0 seconds)
```

## 10. AGRADECIMIENTOS

Apuntes actualizados y adaptados al CEEDCV a partir de la siguiente documentación:

[1] Apuntes Programación de José Antonio Díaz-Alejo. IES Camp de Morvedre.