**Programación**

**CFGS DAW**



**UNIDAD 8**

**Programación orientada a objetos I**

Autores:

Carlos Cacho y Raquel Torres

Revisado por:

Lionel Tarazón – [lionel.tarazon@ceedcv.es](mailto:lionel.tarazon@ceedcv.es)

2019/2020

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

[1. Introducción 4](#__RefHeading___Toc5542_1483547622)

[2. Fundamentos de una clase 5](#__RefHeading___Toc5544_1483547622)

[3. Objetos 7](#__RefHeading___Toc5546_1483547622)

[3.1 Instanciación de un objeto (el operador new) 7](#__RefHeading___Toc5548_1483547622)

[3.2 Asignación de variables de referencia a objetos 8](#__RefHeading___Toc5552_1483547622)

[3.3 Recolector de basura 8](#__RefHeading___Toc5554_1483547622)

[4. Tipos de acceso a los miembros de una clase (VISIBILIDAD) 9](#__RefHeading___Toc5556_1483547622)

[5. Métodos 10](#__RefHeading___Toc5558_1483547622)

[6. Constructores 11](#__RefHeading___Toc5560_1483547622)

[7. Constantes de clase y de objeto 12](#__RefHeading___Toc5562_1483547622)

[8. Arrays de objetos 13](#__RefHeading___Toc5564_1483547622)

[9. Ejemplos 14](#__RefHeading___Toc1270_1026290097)

[9.1 Ejemplo 1 14](#__RefHeading___Toc1272_1026290097)

[9.2 Ejemplo 2 16](#__RefHeading___Toc1274_1026290097)

[9.3 Ejemplo 3 17](#__RefHeading___Toc1276_1026290097)

[9.4 Ejemplo 4 22](#__RefHeading___Toc1278_1026290097)

[9.5 Ejemplo 5 23](#__RefHeading___Toc1280_1026290097)

[9.6 Ejemplo 6 23](#__RefHeading___Toc1282_1026290097)

[10. Agradecimientos 25](#__RefHeading___Toc5588_1483547622)

UD08. Programación orientada a objetos I

# 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 la 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 la 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.**
* **Abstracción:**

# 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.

Una clase es como un molde. A partir de ella se pueden crear objetos.

Es decir antes de poder utilizar un objeto se debe definir la clase a la que pertenece, esa definición

incluye:

* **Atributos**. Las variables miembro de la clase. 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*.

En notación UML[[1]](#footnote-1) la estructura de una clase se define así:

|  |
| --- |
| Nombre de la clase |
| Atributos |
| Métodos |

Y la sintaxis de una clase en Java 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...  **}** |

Veamos un ejemplo:

La clase Persona contiene dos atributos (variables) para almacenar datos sobre una persona (nombre y edad) además de varios métodos (funciones) que hacen cosas con dichos datos.

|  |
| --- |
| *public class Persona {*  *String nombre;*  *int edad;*  *// Establece el nombre de la persona*  void *setNombre(String n) {*  nombre = n;  *}*  // Establece la edad de la persona  void setEdad(int e) {  edad = e;  *}*  *// Devuelve el nombre de la persona*  String *getNombre() {*  return nombre;  }  // Devuelve la edad de la persona  int getEdad() {  return edad;  }  // Muestra su nombre por pantalla  void *imprimeNombre() {*  System.out.println(nombre);  }  // Devuelve true si es mayor de edad, false en caso contrario  boolean esMayorEdad() {  return (edad >= 18)  }  } |

Hay que tener en cuenta que **la clase Persona nos servirá para crear tantos objetos Persona como necesitemos, cada uno con su nombre y edad**. Los métodos nos permitirán manipular los datos de cada objeto. Esto se explica en más detalle en el siguiente apartado.

También es importante entender que **cada clase se crea en un archivo Java diferente** (con el mismo nombre de la clase), **y se utilizan fuera de la clase**.

Por ejemplo, podríamos tener un archivo Persona.java (con la clase Persona del ejemplo anterior) además de un archivo Programa.java (que solo tendrá la función principal public static void main a la que estamos acostumbrados). **Desde la función main de Programa podremos crear objetos de tipo Persona además de cualquier otro código que necesitemos**. Esto se verá en ejemplos posteriores.

Los atributos y métodos de una clase se llaman *miembros de una clase*.

Los atributos (variables) de una clase se llaman *variables de instancia* porque cada instancia de la clase (es decir, cada objeto de la clase), contiene sus propias variable atributo. Por lo tanto, los datos de cada objeto son individuales e independiente de los demás objetos.

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

# Objetos

## Instanciación de un objeto (el operador new)

Cuando creamos una clase definimos un nuevo tipo de datos que puede utilizarse para instanciar (crear) objetos de esa clase. La instanciación se hace de la siguiente manera:

* 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 se hace utilizando el operador *new* que asigna dinámicamente (es decir, durante tiempo de ejecución) memoria a un objeto y devuelve una referencia. Esta referencia se almacena en una variable.

Por ejemplo, suponiendo que ya hemos creado la clase Cubo:

*Cubo c1; // Crea una variable referencia llamada c1 de tipo Cubo*

*c1 = new Cubo(); // Crea un objeto Cubo y lo asigna a c1*

También puede hacerse todo en una sola línea de código:

*Cubo c1 = new Cubo();*

## Asignación de variables de referencia a objetos

Las variables de referencia a objetos permiten acceder al objeto (son una referencia al objeto, no el objeto). Por ello, 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 copia del objeto c1, pero no es así. Lo que sucede es que la referencia c1 se copia en c2, por lo que c2 permitirá acceder al mismo objeto referenciado por c1. Por lo tanto cualquier cambio que se haga al objeto referenciado a través de c2 afectará al objeto al que referencie c1.

## 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 eliminarles 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 la recolecciones se realiza periódicamente en un determinado lapso de tiempo.

# Tipos de acceso a los miembros de una clase (VISIBILIDAD)

Los miembros de una clase (atributos y métodos, es decir, sus variables y funciones) pueden definirse como públicos, privados o protegidos. Es importante entender la diferencia:

* ***public***: Se puede utilizar desde cualquier clase.
* ***private***: Solo puede utilizarlo la propia clase.
* ***protected***: Puede utilizarlo la propia clase y también las subclases heredadas (por ahora no la utilizaremos, la herencia de clases se verá en la siguiente unidad).

Se le llama ***interfaz*** a los miembros de una clase (atributos y métodos) que son ***public***, porque son los que permiten interactuar con la clase desde fuera de ella.

Los principios de la programación orientada a objetos, dice que para mantener le 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 especificados ***private***.

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

Cuando un paquete no esta 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](#9.3.Ejemplo 3|outline)).

CLASES

* Tipos:
* Vecinas (Mismo paquete)
* Externas (Diferente paquete)
* Visibilidad:
  + Por defecto (sólo las vecinas)
  + Public (todas)(Necesario sentencia de importación 🡪 import…)

MIEMBROS

* Por defecto
* Public
* Private

# Métodos

Se llama método a una función de una clase. Su sintaxis general 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.

Cono en cualquier otra función, los métodos devuelven un valor con la sentencia **return** a no ser que el tipo devuelto se defina como void:

*return valor;*

Aunque sería perfectamente válido crear una clase que únicamente contenga datos, esto raramente ocurre. La mayoría de las veces es conveniente crear métodos para acceder a los atributos de 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.

# 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 inicializan los datos asignándoles valores por defecto.

Cuando no se llama a un constructor de forma explicita, 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](#9.4.Ejemplo 4|outline)).

# 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***: toman valores comunes a todos los objetos existentes y potencialmente variables. Pueden utilizarse aunque no exista ningún objeto instanciado.
* 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 final***: combinan las características de static y final.
* Resto de atributos (sin static ni final): atributos variables y diferentes para cada objeto.

 Los **atributos static** 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](#9.5.Ejemplo 5|outline).

# 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();*

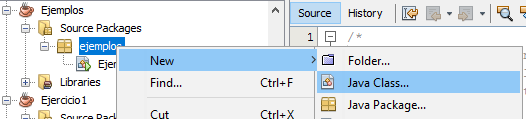
(Ver [ejemplo 6](#9.6.Ejemplo 6|outline)).

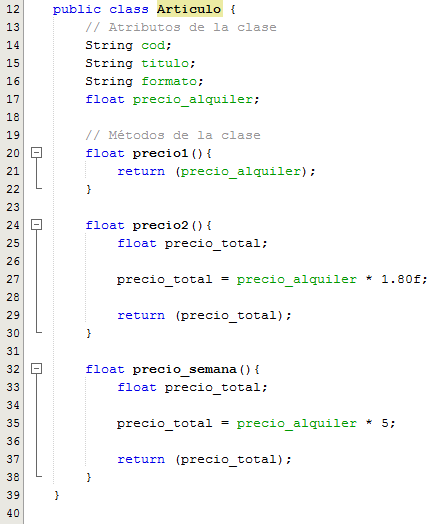
# Ejemplos

## Ejemplo 1

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

los siguientes atributos: codigo\_articulo, titulo, formato y precio\_alquiler. También 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 Clas*s. Crearemos una clase llamada Articulo.

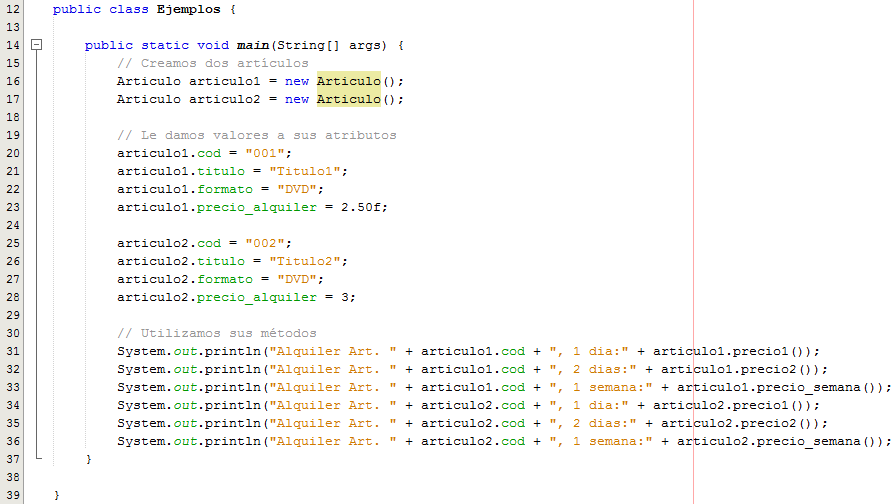




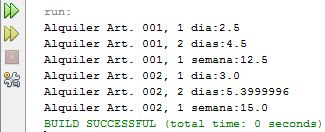
El método *precio1* devuelve el valor del precio de alquiler del artículo. El método *precio2* calcula el precio de alquiler de dos días haciendo un descuento del 20% (por eso se multiplica por 1,8). Por último el método *precio\_semana* calcula el precio de una semana multiplicando por 5 el precio de alquiler.

Vamos a crear una clase nueva llamada Ejemplos, con la función public static void main, donde vamos a instanciar (crear) objetos de Articulo y utilizarlos.

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.



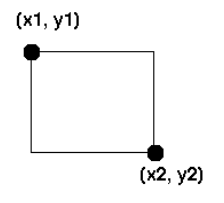
La salida es:

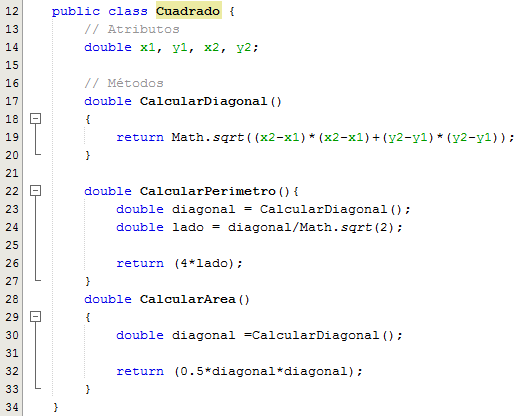


## Ejemplo 2

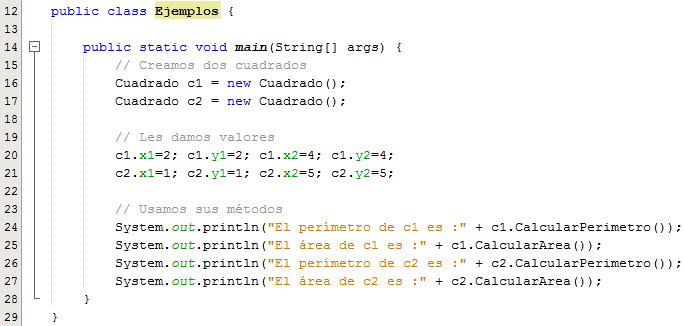
En este ejemplo vamos a implementar la clase *Cuadrado*, que representa cuadrados mediante dos coordenadas 2D, y define tres métodos que permiten calcular, respectivamente, la diagonal, el perímetro y el área.

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

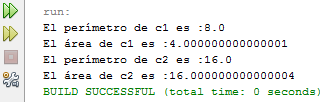
El código de Cuadrado.java sería:



El código de Ejemplos.java sería:



Y la salida:



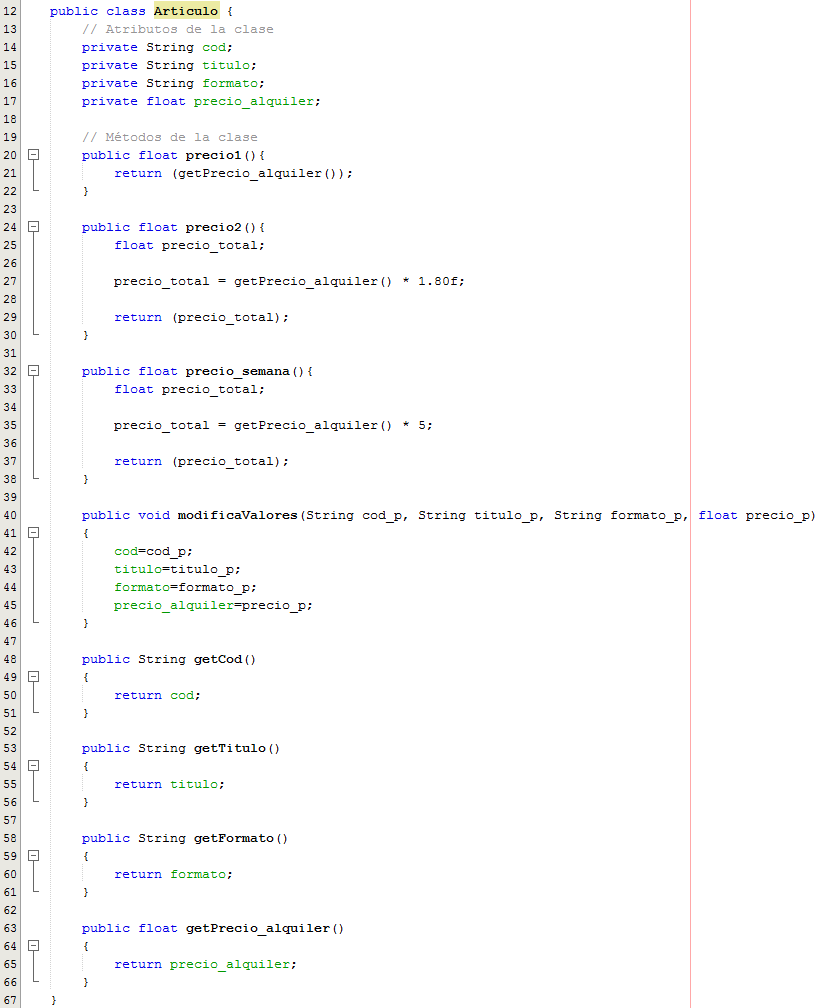
## Ejemplo 3

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 Articulo del Ejemplo 1.

Pero ahora no podremos leer ni modificar los atributos de la clase desde fuera de ella (porque son private). Así que vamos a definir métodos que nos permitan hacerlo:

* Crearemos métodos public (uno por atributo) que nos devuelva el valor de cada atributo. A esto se le llama métodos “***get***” o “***getters***” (del inglés coger).
* Del mismo módo, métodos que nos permitan modificar el valor de los atributos. A esto se le llama métodos “***set***” o “***setters***” (del inglés establecer). En el ejemplo se llama *modificaValores* y permite cambiar todos los valores en una sola llamada.

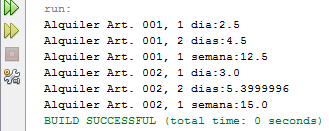
El código de Articulo.java sería:



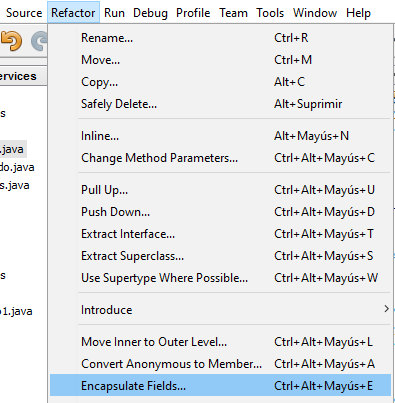
El código de Ejemplos.java con el main sería:



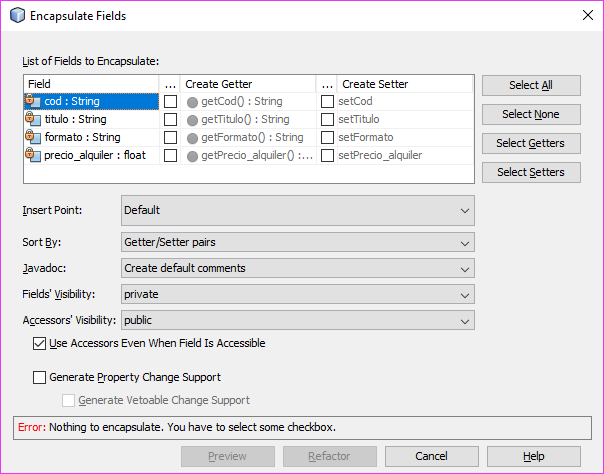
Y la salida:



Otra forma de crear los métodos *getters* y *setters* es utilizando el *Refactor* de Netbeans. Para ello pinchamos en el menú *Refactor* y luego en *Encapsulate Fields*.



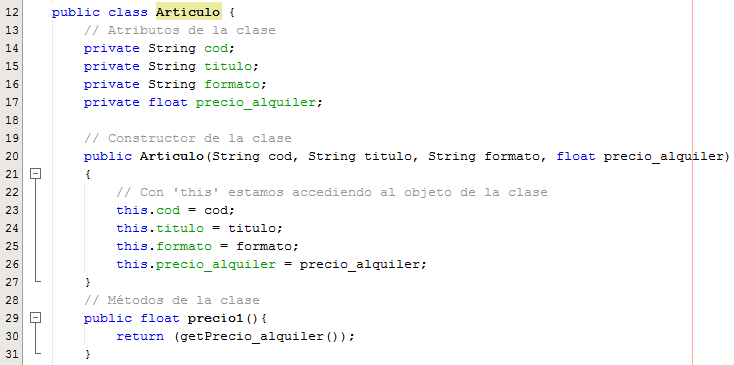
Seleccionamos los getters y setters que queramos crear y pinchamos en *Refactor*.



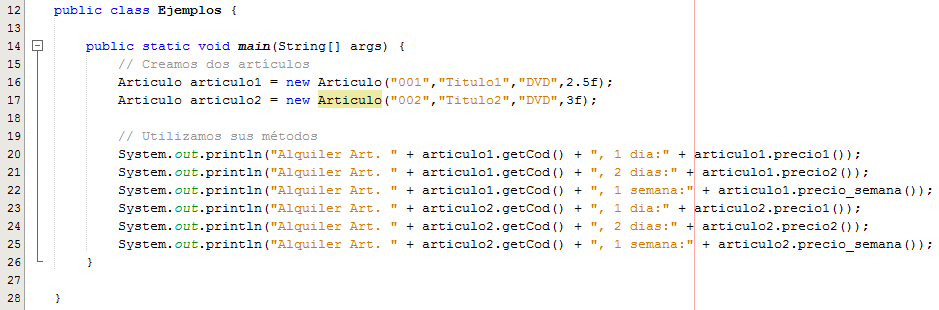
Automáticamente aparecerán los métodos en el código.

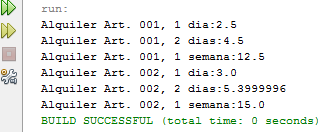
## Ejemplo 4

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 de Articulo.java:



Ahora en Ejemplos.java utilizamos el constructor (al instanciar el objeto con new). Esto permite instanciar el objeto y definir atributos en una sola instrucción.



Y la salida:

## 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 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;*

Se deja como ejercicio propuesto hacer estos cambios y probarlo.

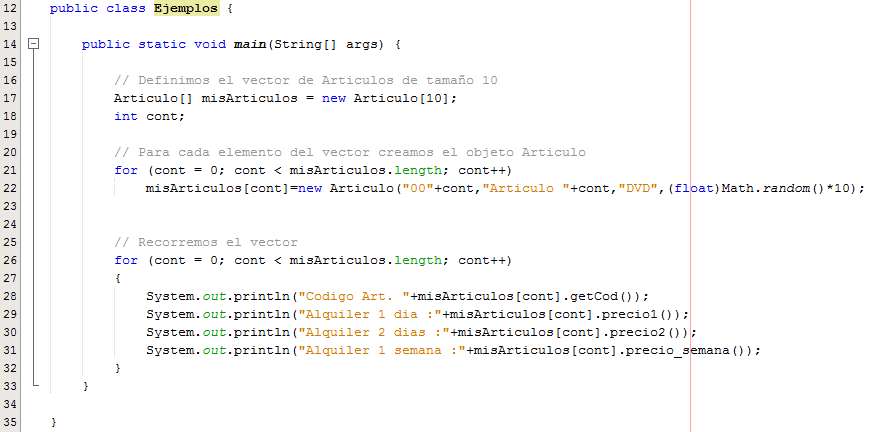
## Ejemplo 6

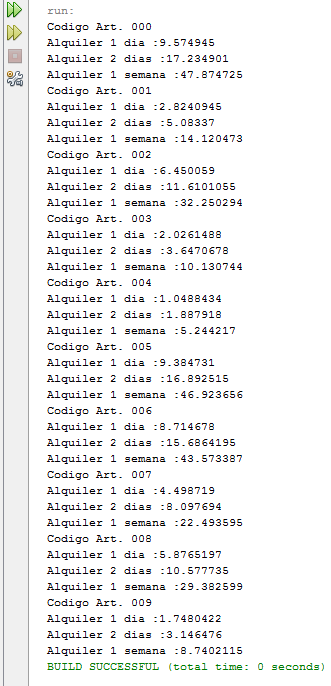
En este ejemplo vamos a crear un array (vector) que contenga diez objetos de tipo *Articulo*. Los instanciaremos todos con nombres genéricos, consecutivos y precios aleatorios.

Primero creamos un vector de Artículos llamado *misArticulos* (es un array de referencias a objetos Articulo). Luego instanciaremos los 10 objetos, guardando cada uno de los objetos referenciados.

La clase *Articulo* no se vería modificada.

El código de Ejemplos quedaría así:





La salida sería:

# 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.

1. [Unified Modeling Language](https://es.wikipedia.org/wiki/Lenguaje_unificado_de_modelado) [↑](#footnote-ref-1)
2. 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. [↑](#footnote-ref-2)