Programación Java
Clases y objetos

(Versión de fecha 8 de diciembre de 2024)

1 Concepto de clase y objeto

Anteriormente, hemos hablado de los *arrays* y explicado que se trata de un tipo de datos compuesto que permite agrupar, en una misma variable, un conjunto de valores del mismo tipo.

Una *clase* también es un tipo de datos compuesto que agrupa, en una misma entidad, un conjunto de variables y un conjunto de funciones. Las *clases* son el tipo de datos que sustenta la programación en Java y, por extensión, la *Programación Orientada a Objetos*.

Para definir una clase, se pone la palabra clave *class* seguida de un nombre adecuado y un bloque de código entre llaves en el que se declaran las variables y las funciones asociadas con la clase. La Figura 1 muestra el esquema del bloque de código utilizado para definir una clase:

```
class NombreClase {

    // Lista de variables de la clase (Propiedades)

    // Lista de funciones de la clase (Métodos)
}
```

Figura 1: Esquema del bloque de código de definición de una clase

La definición de una clase define un nuevo tipo de datos. Los valores concretos de las variables de dicho tipo son *objetos* o *instancias* de dicha clase.

Para los nombres de las clases, se sigue el criterio *PascalCase* que se ha explicado en otras ocasiones: una o más palabras con la primera letra mayúscula y sin separación entre ellas.

En la terminología que se usa en la Programación Orientada a Objetos, las variables incluidas en una clase se denominan *propiedades* o *atributos* de la clase, también a veces *campos*. A las funciones incluidas en una clase se les suele llamar *métodos* de la clase. Para referirse de manera conjunta a las propiedades y a los métodos de una clase, se hace referencia a los *miembros* de una clase. Por tanto, los miembros de una clase son sus propiedades y sus métodos.

Propiedades = variables definidas en la clase
Métodos = funciones definidas en la clase
Miembros = propiedades + métodos

El siguiente podría ser el código para definir una clase de nombre *Circulo*:

```
class Circulo {
   double radio;

   double area() {
      double resultado = 3.14*radio*radio;
      return resultado;
   }
}
```

La clase *Circulo* así definida tiene una variable del tipo *double* llamada *radio* y una función llamada *area*, que no recibe ningún parámetro y devuelve un valor *double* con el cálculo del área del circulo. La variable *radio* es una propiedad de la clase *Circulo* y la función *area()* es un método de *Circulo*.

Es habitual representar las clases utilizando esquemas *UML*[1]. En *UML*, una clase se representa mediante un rectángulo dividido en tres secciones: el nombre de la clase, sus propiedades y sus métodos. La Figura 2 muestra el esquema *UML* de la clase *Circulo*.

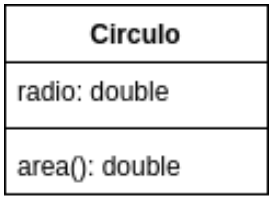

Figura 2: Esquema UML para representar la clase *Circulo*.

En el esquema de la Figura 2, la sección superior contiene el nombre de la clase, en fuente grande, vertical, negrita y centrado. La siguiente sección está reservada a las propiedades de la clase; en este caso solo hay una: *radio*. La sección inferior se reserva a los métodos de la clase. Dentro de los paréntesis se pone el tipo de datos de los parámetros y, tras los dos puntos, se pone el tipo de datos del valor devuelto.

[1] El estándar *UML* (*Unified Modelling Language*) se utiliza para representar procesos y, en particular, para representar las jerarquías de clases en la Programación Orientada a Objetos. Puede consultar la siguiente dirección Web:

`https://en.wikipedia.org/wiki/Class_diagram`

Observación: sintaxis UML

Observe que la forma de indicar el tipo de datos de las propiedades o el valor devuelto por un método es diferente en UML que en el código Java.
El tipo de datos de una propiedad se pone detrás del nombre de la misma, separado por dos puntos:

```
nombrePropiedad : tipoDeDatos
```

El tipo de datos del valor devuelto por una función también se especifica de manera parecida:

```
nombreMetodo() : tipoDevuelto
```

Si el método tiene parámetros, se ponen sus tipos de datos en los paréntesis, separados por comas (no se suele escribir el nombre del parámetro):

```
nombreMetodo(tipoParametro1,...) : tipoDevuelto
```

Tenga en cuenta que, hay numerosos lenguajes que soportan la Programación Orientada a Objetos, cada uno con su propia sintaxis para definir las clases, las propiedades y los métodos. UML se utiliza para describir clases en cualquier lenguaje y tiene su propia forma de definirlas.
Nota: en los diagramas *UML* hay que indicar también la *visibilidad* de los miembros de la clase, esto es, si son variables o funciones *publicas*, *privadas*, *protected*,... Estos conceptos y la forma de presentarlos en los esquemas de las clase se explicarán en el Apartado **??**.

2 El constructor de la clase

Todas las clases disponen de un método, implementado por defecto, que permite crear objetos de la clase. A este método se le denomina el *constructor* de la clase.

La construcción de los objetos es un proceso que se hace en dos pasos:

- Declaración de una variable del tipo de datos de la clase.
- Creación de un objeto de la clase y asignación a la variable anterior.

Para declarar una variable de una clase se usa la siguiente sintaxis:

```
NombreDeClase nombreDeVariable;
```

En este momento, todavía no se ha creado ningún objeto, solo se ha declarado que la variable *nombreDeVariable* guardará referencias a objetos de la clase *NombreDeClase*.

Una vez declarada la variable, se puede crear un objeto de la clase utilizando el operador *new* y asignar una referencia del mismo a la variable :

```
nombreDeVariable = new NombreDeClase();
```

En la expresión anterior, *NombreDeClase()* es el constructor de la clase y se utiliza a través del operador *new*. El resultado es la creación de un objeto de esa clase en memoria, donde se guardarán los valores de las propiedades del objeto. La dirección de memoria de dicho objeto, esto es, la *referencia* al objeto, es lo que se copiará en la variable *nombreDeVariable*.

En el mecanismo de creación de objetos indicado, hay dos aspectos que es muy importante tener in mente:

- La creación del objeto, esto es, la reserva de memoria para guardar los valores de las propiedades del objeto, se hace de manera *dinámica* (durante la ejecución del programa).

- Las variables guardan una *referencia* a la posición de memoria donde está guardado el objeto. Las clases son *tipos de datos por referencia*.

Al igual que vio cuando se explicó la creación de variables de tipos primitivos, es habitual juntar en una misma instrucción de código la declaración de la variable y la asignación del valor:

```
NombreDeClase nombreDeVariable = new NombreDeClase();
```

Por ejemplo, para crear una variable del tipo *Circulo*, que se mostró como ejemplo en el apartado anterior, se utiliza la siguiente sintaxis:

```
Circulo c = new Circulo();
```

Se podría haber hecho utilizando dos instrucciones: una para la declaración de la variable *c* y otra para la creación del objeto y la asignación de la referencia a la variable.

```
Circulo c;
c = new Circulo();
```

La primera de las dos instrucciones es una declaración de que la variable *c* guardará la *referencia* a un objeto del tipo *Circulo*, pero la declaración no crea ningún objeto, solo indica que la variable *c* guardará una referencia a un *Circulo*.

En la segunda instrucción es en la que crea un objeto de la clase *Circulo* y su referencia se asigna a la variable *c*. La variable *c*, guarda la dirección de memoria en la que está guardado el objeto recién creado. Se dice que la variable *c* guarda una *referencia* a un objeto de la clase Circulo o que la variable *c apunta* a un objeto de la clase *Circulo*.

La cláusula *new*, seguida del nombre de la clase y los paréntesis es la forma de llamar al constructor por defecto de cualquier clase. Se trata de un método especial que todas las clases implementan por defecto. En capítulos posteriores se explicará cómo personalizar el método constructor de la clase.

3 Objetos, instancias, estado y comportamiento

En el ejemplo que se ha visto en los aparatados anteriores, se distinguen los siguientes conceptos:

- La clase *Circulo* es la definición de un nuevo tipo de datos.
- La variable *c* es del tipo de datos *Circulo*.
- El valor que tiene asociado *c* es un *objeto* de la clase *Circulo*, es un *Circulo*. También se dice que es una *instancia* de la clase *Circulo*.

En general, a los valores asociados a las variables del tipo de datos de una clase determinada se les denomina *objetos* de esa clase, o también, que son *instancias* de dicha clase[2].

La intención de la programación orientada a objetos es que las clases representen las entidades del mundo real que se están modelizando en el programa. En ese sentido, el valor de las propiedades representa el *estado* de los objetos mientras que los métodos definen el *comportamiento* de dichos objetos.

[2] Aunque los *objetos* o las *instancias* son los valores que se asocian con las variables, es habitual utilizar indistintamente la denominación *objeto* o *instancia* para referirse a la variable o al valor asociado.

Clase = representación de una entidad modelizada por el programa
= propiedades + métodos.
Objeto = valor concreto de una de esas entidades,
= instancia de una clase.
Estado del objeto : valor de las propiedades en esa instancia. Comportamiento del objeto: queda definido por los métodos de la clase.

Para acceder a las propiedades de un objeto de una clase se utiliza la notación *punto*:

```
nombreVariable.nombrePropiedad
```

La misma notación se utiliza para acceder a los métodos de un objeto:

```
nombreVariable.nombreMetodo()
```

Se va a desarrollar un ejemplo completo en *jshell* que utilice la clase *Circulo* para crear un objeto, asignar valor a su propiedad *radio* y obtener el valor de su área utilizando el método *area()*. Lo primero es crear la clase que define el nuevo tipo de datos *Circulo*, como se muestra en la Figura 3

```
jshell> class Circulo {
   ...>     double radio;
   ...>
   ...>     double area() {
   ...>         return 3.14*radio*radio;
   ...>     }
   ...> }
jshell>
```

Figura 3: Creación de la clase *Circulo*.

El código anterior crea la clase *Circulo* con su propiedad *radio* del tipo *double* y su método *area()* que devuelve un *double* con el cálculo del área del circulo.

En el código de la Figura 4, se crea un objeto del tipo *Circulo* utilizando el constructor por defecto de la clase. A continuación, se asigna el valor *10.0* a su propiedad *radio*. Por último, se crea una variable llamada *s* a la que se le asigna el valor del área del círculo y se muestra en pantalla su valor.

```
jshell> Circulo c = new Circulo();
jshell> c.radio = 10.0;
$3 ==> 10.0
jshell> double s = c.area();
jshell> s
s ==> 314.0
jshell>
```

Figura 4: Utilización de la clase *Circulo*.

4 ÁMBITO DE LOS MIEMBROS DE LAS CLASES

En la clase *Circulo* que se ha utilizado como ejemplo en los apartados anteriores, la propiedad *radio* se dice que es una propiedad *de instancia*. Esto quiere decir que, para acceder a ella, primero hay que crear un objeto de la clase. Cada objeto de la clase *Circulo* tiene su propio *radio*.

Observe de nuevo el código de la clase *Circulo*:

```
class Circulo {
   double radio;

   double area() {
      double resultado = 3.14*radio*radio;
      return resultado;
   }
}
```

En el siguiente código, se crean dos objetos de la clase *Circulo*, *c1* y *c2*, cada uno de ellos con su propio *radio*.

```
Circulo c1 = new Circulo();
c1.radio = 3.0;

Circulo c2 = new Circulo();
c2.radio = 4.5;
```

El método *area()* también es un miembro de instancia. Para llamar al método *area()*, hay que usar un objeto *Circulo* concreto. En el siguiente código, se llama al método *area()* del círculo *c1* y del círculo *c2*:

```
jshell> c1.area()
$6 ==> 28.259999999999998

jshell> c2.area()
$7 ==> 63.585
```

Observe que el resultado del método *area()* es diferente para cada círculo. Si se fija en el código del método *area()* de la definición de la clase *Circulo*, verá que dentro del método se accede al valor de la propiedad *radio*. Cada objeto tiene su propio *radio* y, cuando se llama al método *area()* a través de un objeto círculo concreto, el *radio* que se usa en el método es el de ese círculo concreto:

`c1.area()` → el método *area()* usa el radio de *c1*

`c2.area()` → el método *area()* usa el radio de *c2*

Tanto la propiedad *radio* como el método *area()* son miembros *de instancia* de la clase *Circulo*. Para acceder a los métodos de instancia hay que utilizar una instancia concreta de la clase. Ademas, los métodos de instancia pueden acceder a otros miembros de instancia de la clase.

Es posible crear propiedades o métodos que no dependan de una instancia concreta de la clase, sino que pertenezcan a la propia clase y sean los mismos para cualquier instancia que se cree de la clase. Para ello, al definir la clase hay que escribir la cláusula *static* delante de la propiedad o método que se quiera declarar *de clase*, como se hace en el siguiente ejemplo:

```
static tipoDeDatos nombrePropiedad;
static tipoDeDatos nombreMetodo(){ ... }
```

Observe el ejemplo siguiente. Se trata de una clase llamada *Producto*, que modeliza productos de un comercio. Cada producto queda definido por su nombre y el precio neto de venta, representados por las propiedades *nombre*, del tipo *String* y *precioNeto*, del tipo *double*. La clase dispone también de un método *precioDeVenta()* que calcula el precio de venta

del producto, sumando al precio neto el importe del IVA. El porcentaje de IVA que hay que aplicar a cada producto es una propiedad *static* de la clase llamada *iva*:

```
jshell> class Producto {
   ...>     static double iva = 0.21;
   ...>
   ...>     String nombre;
   ...>     double precioNeto;
   ...>
   ...>     double precioDeVenta() {
   ...>         double result = (1+iva)*precioNeto;
   ...>         return result;
   ...>     }
   ...> }
|  created class Producto
```

Ahora se van a crear dos productos, *p1* y *p2*, calculando a continuación el precio de venta de cada uno de ellos:

```
jshell> Producto p1 = new Producto()
p1 ==> Producto@3d494fbf

jshell> p1.nombre = "Detergente"
$3 ==> "Detergente"

jshell> p1.precioNeto = 3.55
$4 ==> 3.55

jshell> p1.precioDeVenta()
$5 ==> 4.2955

jshell> Producto p2 = new Producto()
p2 ==> Producto@1e80bfe8

jshell> p2.nombre = "Lata de sardinas"
$7 ==> "Lata de sardinas"

jshell> p2.precioNeto = 1.40
$8 ==> 1.4

jshell> p2.precioDeVenta()
$9 ==> 1.694
```

Observe el código del método *precioDeVenta()* en la definición de la clase *Producto*. Se trata de un método de instancia, como lo era el método *area()* de la clase *Circulo*. Dentro del método *precioDeVenta()*, se accede al *precioNeto* del objeto *Producto* que llama al método y también se accede a la propiedad *iva*.

La propiedad *iva* lleva el modificador *static*. Esto quiere decir que hay una sola copia de *iva* para todas las instancias de *Producto* que se creen. La propiedad *iva* es una propiedad *de clase*, no de instancia. A pesar de ser una propiedad *static*, los métodos de instancia de la clase pueden acceder a ella.

Para acceder a una propiedad *static* desde fuera de la clase, no se puede utilizar una instancia concreta, se usa la propia clase, con la siguiente sintaxis:

```
NombreDeClase punto nombrePropiedad
```

Siguiendo con el ejemplo de la clase *Producto*, se va a modificar el valor de la propiedad *iva* de la clase, asignándole el valor *0.07*. A continuación se vuelve a calcular el precio de venta de *p1* y *p2*:

```
jshell> Producto.iva = 0.07
$10 ==> 0.07

jshell> p1.precioDeVenta()
$11 ==> 3.7985

jshell> p2.precioDeVenta()
$12 ==> 1.498
```

En el código anterior, observe cómo se modifica el valor de la propiedad *iva* usando el nombre de la clase; observe también que el precio de venta de los productos se calcula con el nuevo valor del IVA de la clase. Queda evidenciado que la propiedad *iva* es una propiedad de la clase que afecta a todas las instancias de la misma.

Más adelante se verá como modelizar la relación entre distintos objetos así como las formas habituales de modificar los estados de los objetos. También se explicarán más adelante los tres mecanismos que hacen de la Programación Orientada a Objetos un paradigma de programación muy potente: *herencia*, *encapsulación* y *polimorfismo*.

La Tabla 4 resume la terminología utilizada en la Programación Orientada a Objetos:

Terminología en Programación Orientada a Objetos

Término	**Explicación**
Clase	Tipo de datos compuesto de variables y funciones
Objeto, instancia	Variable o valor del tipo de datos definido por una clase
Propiedad, atributo	Cada una de las variables contenidas en la definición de una clase.
Método	Cada una de las funciones definidas en una clase
Miembros	Para referirse de manera conjunta a las propiedades y los métodos
Estado	Valor de las propiedades de un objeto en cada momento
Comportamiento	Se define mediante los métodos de los que disponen los objetos

5 Visibilidad de los miembros. Encapsulación

Al declarar los miembros de una clase, propiedades y métodos, es conveniente añadir un *modificador de visibilidad* que defina de manera clara las zonas del código del programa desde las que se podrá acceder a ellos.

Si no se especifica ningún modificador de visibilidad para un miembro de una clase, es posible acceder a dicho miembro desde el código de su misma clase o desde clases que pertenezcan al mismo paquete. En el Apartado ?? se explicará el funcionamiento de los *paquetes* en Java.

Cuando un miembro se declara como público, *public*, es accesible desde el código de cualquier clase. En cambio, si un miembro se declara privado, *private*, solo es posible acceder a él desde el código interior de su clase.

```
public tipo nombreMiembro;
private tipo nombreMiembro;
```

Hay otro tipo de modificador de visibilidad: *protected*. Cuando los miembros se declaran protegidos, se puede acceder a ellos desde el código de cualquier clase derivada de la clase que define el miembro protegido. En el Apartado ?? se explicará la *herencia* entre clases y la utilización del modo de visibilidad *protected* para los miembros.

```
protected tipo nombreMiembro;
```

En la Programación Orientada a Objetos, el valor de las propiedades de una clase representa su *estado*, mientras que los métodos modelizan el *comportamiento* de los objetos de la clase.

El estado de un objeto no se debe poder modificar directamente por otros objetos, solo se debe poder modificar utilizando los métodos que ofrece la propia clase para hacerlo. Siguiendo esta regla, las propiedades de las clases se deben declarar privadas o protegidas, de forma que solo se puedan modificar utilizando los métodos que proporcione la clase para hacerlo.

En cambio, los métodos suelen ser públicos, pues representan el comportamiento de los objetos de las clases, el *interface público* que exponen ofrecen a los objetos de otras clases para que puedan conocer o modificar el estado del objeto.

A esta forma de interpretar la visibilidad de los miembros de las clases se la conoce con el nombre de *encapsulación* y es uno de los tres pilares básicos en los que se fundamenta la Programación Orientada a Objetos, como se mencionó en el Apartado 4. La Figura 5 esquematiza el concepto de encapsulación.

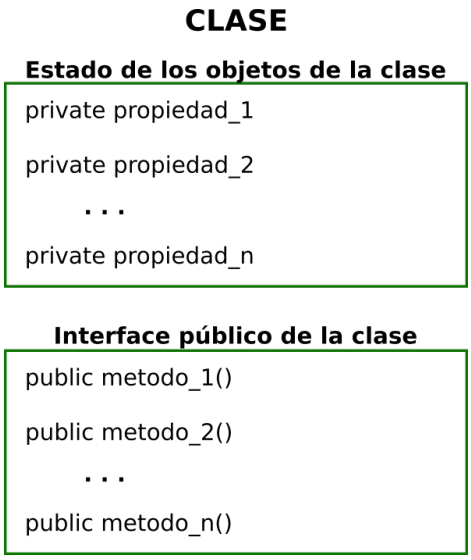

Figura 5: Criterio general de visibilidad de los miembros de las clases

En el siguiente ejemplo, la clase *Vehiculo* tiene una propiedad privada llamada *velocidad*. Dentro de la clase, los métodos pueden acceder o modificar el valor de la propiedad *velocidad*. Desde fuera de la clase, para modificar el valor de la velocidad del vehículo es necesario utilizar el método *setVelocidad()*. Para saber el valor de la velocidad del vehículo desde fuera de la clase, es necesario utilizar el método *getVelocidad()*:

```
jshell> class Vehiculo {
   ...>     private int velocidad;
   ...>     public void setVelocidad(int v) {
   ...>         velocidad = v;
   ...>     }
   ...>     public int getVelocidad() {
   ...>         return velocidad;
   ...>     }
   ...> }
|  created class Vehiculo

jshell> Vehiculo coche = new Vehiculo();
coche ==> Vehiculo@3d494fbf

jshell> coche.setVelocidad(20);

jshell> coche.getVelocidad();
$4 ==> 20
```

Es costumbre que los métodos que se utilizan para dar valor a una propiedad se llamen *setNombrePropiedad()*. Así mismo, los métodos que sirven para leer el valor que tiene una propiedad se suelen llamar *getNombrePropiedad()*. En conjunto, se les suele llamar *metodos getter-setter* de las propiedades de la clase.