**Academia Java – JAVA**

***Métodos Que Usan Variables de Instancias***

Tabla de contenido

[CÓMO SE COMPORTAN LOS OBJETOS 3](#_Toc310497946)

[1. Recuerda: una clase describe qué sabe y que hace un objeto 4](#_Toc310497947)

[1.1. Puede comportarse diferente un mismo método por cada objeto? 4](#_Toc310497948)

[1.2. El tamaño si influye en…el ladrido 4](#_Toc310497949)

[1.3. Enviando “cosas” a un método 5](#_Toc310497950)

[1.4. Puede que los métodos que devuelvan “cosas” 6](#_Toc310497951)

[1.5. Puedes enviar más de una “cosa” a un método 7](#_Toc310497952)

[1.5.1. Llamando un método con dos parámetros y enviándolo con dos argumentos 7](#_Toc310497953)

[1.6. Java pasa por valor 8](#_Toc310497954)

[1.7. Cosas interesantes que puedes hacer con los parámetros y tipos de retorno 9](#_Toc310497955)

[1.8. Los developers de verdad ¡encapsulan! 10](#_Toc310497956)

[1.9. Ocultar los datos 11](#_Toc310497957)

[1.10. Encapsulando la clase GoodDog 11](#_Toc310497958)

[1.11. Como se comportan los objetos en un array? 12](#_Toc310497959)

[1.12. La diferencia entre variables de instancia y variables locales 13](#_Toc310497960)

[1.13. Comparando variables (primitivos o referencias) 13](#_Toc310497961)

[1.14. ¿Es legal? 15](#_Toc310497962)

[1.15. Sé el compilador 15](#_Toc310497963)

# CÓMO SE COMPORTAN LOS OBJETOS

El estado afecta a la conducta, el comportamiento afecta al estado.



Ahora ya sabes que los objetos tienen estados y comportamientos, representado por las variables de instancia y métodos, respectivamente. Pero hasta ahora, no has visto cómo el estado y el comportamiento se relacionan.

Ya sabemos que cada instancia de una clase (cada objeto de un tipo) puede tener valores únicos para sus variables de instancia.

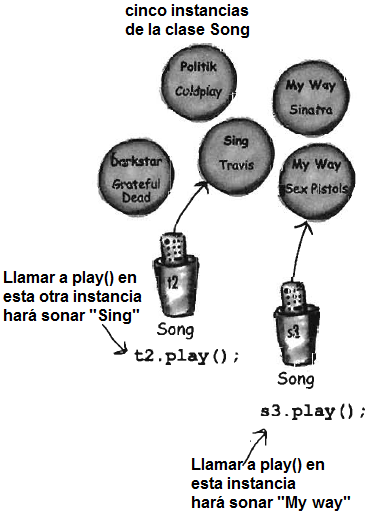
Por ejemplo, la instancia de nombre A de Dog puede llamarse "Fido" y tener un peso de 15 kilos. Dog B se llama "Killer" y pesa 9 kilos. Si la clase Dog tiene un método makeNoise(), bueno, será obvio que Fido ladrará un “poquito” más fuerte que Killer ¿no? Bueno, este es el punto de un objeto - tiene un comportamiento que se desarrolla según su estado. En otras palabras, los métodos usan los valores de las variables de las instancias. Manipular esos valores es como decir, "si un Dog tiene menos de 11 kilos, hace un sonido de nene, de lo contrario..." o "aumenta su peso en 5". Vamos a jugar con los cambios de estado.



# Recuerda: una clase describe qué sabe y que hace un objeto

Una clase es el proyecto para una clase. Cuando escribes una clase, estás describiendo como la JVM debe crear un objeto de un tipo. Ya sabes que cada objeto de un determinado tipo puede tener diferentes valores en sus variables de instancia. Pero ¿qué hay acerca de los métodos?

# Puede comportarse diferente un mismo método por cada objeto?



Cada instancia de una clase particular tiene los mismos métodos, pero los métodos pueden tener un comportamiento diferente dependiendo de los valores de las variables de la instancia.

La clase canción tiene dos variables de instancias, title y artist. El método play() toca una canción, pero la canción que sonará será la representada por el valor de la variable title. Entonces, si llamas al método play() de una instancia, oirás la canción “Politik”, mientras que en otra instancia se escuchará “Darkstar”. Aunque el código fuente sea el mismo:

**void** play( ) {

Si, por fin vemos señales de encapsulación.

soundPlayer.playSound(title);

}

Song t2 = **new** Song();

t2.setArtist("Travis");

t2.setTitle("Sing");

Song s3 = **new** Song();

s3.sstArtist("Sex Pistols");

s3.setTitie ("My Way") ;

# El tamaño si influye en…el ladrido

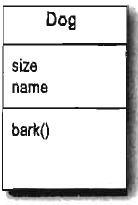
El ladrido de un perro pequeño es diferente al de un gran perro. La clase Dog tiene una variable de instancia de nombre size, que el método bark() para decidir qué tipo de ladrido usará.

**public** **class** Dog {

**int** size;

String name;

**public** **void** bark() {

 **if** (size > 60){

System.*out*.println("Wooof! Wooof!");

}

**else** **if** (size > 14) {

System.*out*.println("Ruff! Ruff!");

} **else** {

System.*out*.println("Yip! Yip!");

}

}

}

**public** **class** DogTestDrive {

**public** **static** **void** main(String[] args) {

Dog one = **new** Dog();

one.size = 70;

Dog two = **new** Dog();

two.size = 8;

Dog three = **new** Dog();

three.size = 35;

one.bark() ;

two.bark() ;

three.bark();

}

}

La salida es:



# Enviando “cosas” a un método

Como esperas de cualquier lenguaje de programación, en Java también puedes pasar valores a tus métodos. Por ejemplo, podrías querer decirle a tu objeto Dog cuantas veces llamará al método bark():

d.bark(3);

Estos valores que pasas a un métodos puedes llamarlos argumentos o parámetros. En realidad podrías llamarlos de muchas formas. Al final y al cabo lo que debes tomar en cuenta es:

**Un método utiliza parámetros. Una llamada pasa argumentos.**

Los argumentos con cosas que pasas a tus métodos. Un argumento se convierte en un parámetro cuando es una variable local a un método. Una variable que, recuerda, tiene un tipo y un nombre y puede ser usado en el cuerpo del método.

Pero aquí hay una parte importante: Si un método toma un parámetro, debes pasarle algo. Y ese algo debe ser un valor apropiado al tipo de dato indicado.

|  |  |  |
| --- | --- | --- |
|  | Llama al método bark() en la referencia a Dog y pásale el valor 3 (como argumento al método). | Dog d = **new** Dog();  d.bark(3); |
|  | Los bits que representan al valor int 3 son entregados en el método bark(). |  |
|  | Los bits en el parámetro numOfBarks. |
|  | Usa el parámetro numOfBarks como una variable en el código del método. |

# Puede que los métodos que devuelvan “cosas”

Los métodos pueden retornar valores. Cada método está declarado con un tipo de retorno, pero hasta ahora hemos creado métodos con tipo de retorno void, lo cual significa que no retorna nada.

**void** go(){

}

Pero podemos declarar un método para darle un tipo específico para el valor de retorno al caller:

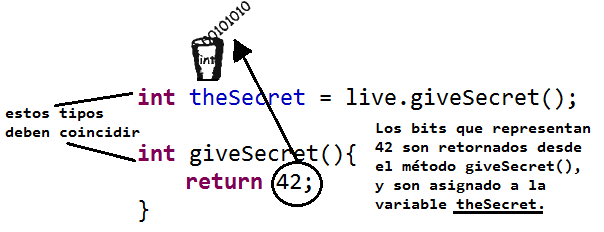
**int** giveSecret(){

**return** 42;

}

Si tu declaras que un método retorna un valor, debes retornar un valor del tipo declarado (o al menos compatible).

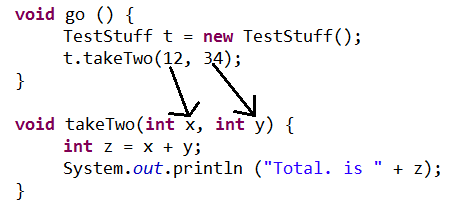
**Lo que dices que vas a retornar, es mejor que lo retornes!**



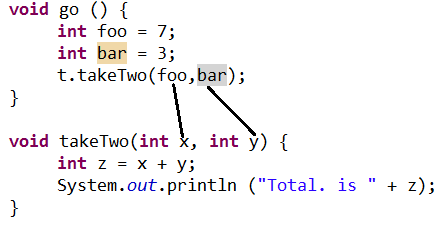
# Puedes enviar más de una “cosa” a un método

Los métodos pueden recibir muchos parámetros separados con comas cuando los declares. Y también debes separar a los parámetros con comas cuando los pases. Más importante aún debes pasar los argumentos en el orden y con los tipos correctos.

# Llamando un método con dos parámetros y enviándolo con dos argumentos



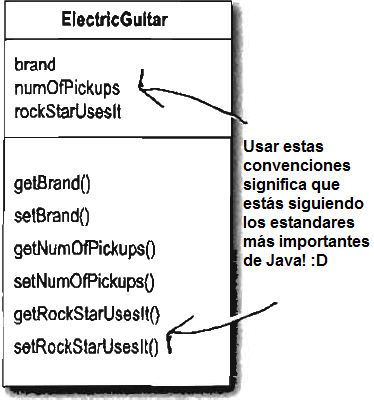
O también:



# Java pasa por valor

|  |  |  |
| --- | --- | --- |
|  | Lo que significa que pasa por copia. |  |
|  |  | Declara una variable int y asígnale el valor de 7. El patrón de bits para 7 va en la variable llamada x. |
|  |  | Declara un método con un parámetro int llamado z. |
|  |  | Llama al método go(), pasa la variable x como argumento. Los bits de x son copiados en z. |
|  |  | Cambia el valor de z dentro del método. El valor de x no cambia! El argumento que se pasa al parámetro z solo fue una copa de x. |

|  |  |
| --- | --- |
| C:\Users\ingas\Desktop\Atencion2.png | * Las clases definen que sabe y que hace un objeto. * Las cosas que un objeto sabe son variables de instancia (estado). * Las cosas que un objeto hace son métodos (comportamiento). * Los métodos pueden usar variables de instancia por ello objetos del mismo tipo pueden tener un comportamiento diferente. * Un método puede tener parámetros, lo cual significa que puedes pasarle uno o más valores. * La cantidad y el tipo de valores que pasas deben coincidir con el orden y el tipo de los parámetros declarados en el método. * Los valores que ingresan y salen de los métodos pueden ser implícitamente promovido a tipos más grandes o explícitamente casteados a tipos más pequeños. * El valor que pasas como argumento a un método puede ser un valor literal (2, ‘c’, etc) o una variable del tipo declarado (x donde esta es una variable int). * Un método debe declarar un tipo de retorno. Un tipo de retorno void significa que se devuelve nada. * Si un método declara un tipo de retorno no-void, debe retronar un valor compatible con el tipo de retorno. |



# Cosas interesantes que puedes hacer con los parámetros y tipos de retorno

Ahora que ya has visto como los parámetros y tipos de retorno funcionan, es hora que hagas un buen uso de los…Getters y Setters (si, por fin ☺). O si prefieres llamarlos formalmente accessors y mutators. Los getters y setters se ajustan perfectamente a las convenciones de nombres Java, entonces es correctísimo usarlos.

Los getters y setter te permiten, bueno…obtener y asignar “cosas” a un objeto. Los valores de las variables de instancia, generalmente. El único propósito en la vida de un Getter es retornar un valor del tipo solicitado. Y bueno, el propósito del Setter es tomar el valor del argumento que le pasan y asignarlo a una variable de la instancia.



**public** **class** ElectricGuitar {

String brand;

**int** numOfPickups;

**boolean** rockStarUsesIt;

**public** String getBrand() {

**return** brand;

}

**public** **void** setBrand(String brand) {

**this**.brand = brand;

}

**public** **int** getNumOfPickups() {

**return** numOfPickups;

}

**public** **void** setNumOfPickups(**int** numOfPickups) {

**this**.numOfPickups = numOfPickups;

}

**public** **boolean** isRockStarUsesIt() {

**return** rockStarUsesIt;

}

**public** **void** setRockStarUsesIt(**boolean** rockStarUsesIt) {

**this**.rockStarUsesIt = rockStarUsesIt;

}

}

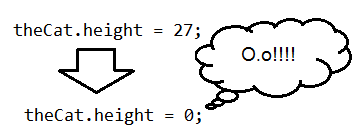


# Los developers de verdad ¡encapsulan!

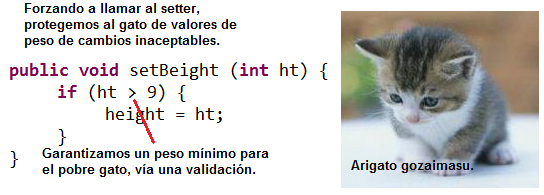
Hasta este momento hemos estamos aplicando una de las faltas más graves para en la POO:

**Exponer nuestra información!**

Algo que no podemos permitir es la modificación de la información por cualquier persona:



Por ello necesitamos construir métodos setter para todas las variables de instancia y encontrar la forma de hacer que el código llame más frecuentemente a ellos y no a la data directamente.



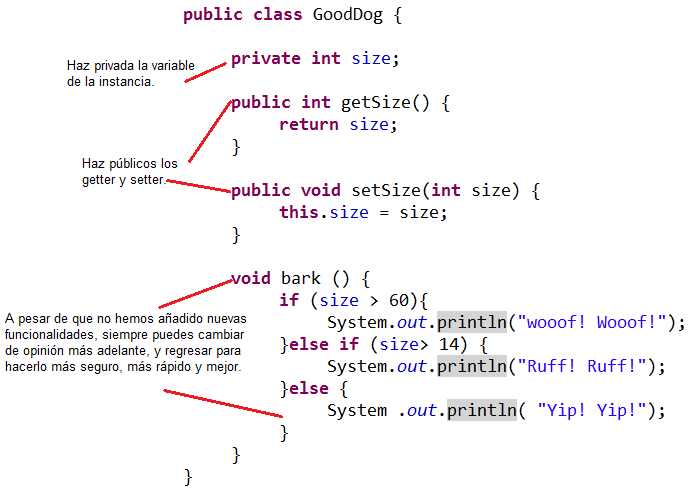
# Ocultar los datos

Sí, pasar de una aplicación que expone nuestros datos a una que protege los datos y protege la modificación de ellos: es sencillo.

Pero ¿cómo es exactamente lo que ocultan los datos? Con modificadores de acceso público y privado (public y private). Ya estás familiarizado con el público – ya lo has usado.

A continuación una regla inicial para encapsular: marcar las variables de instancia como private y generar los getters y setters públicos para el control del acceso.

# Encapsulando la clase GoodDog



# Como se comportan los objetos en un array?

Como cualquier otro objeto. La única diferencia es como los obtienes. En otras palabras, como obtienes el control remoto.

|  |  |  |
| --- | --- | --- |
|  | Declara y crea un array que tenga 7 referencias a Dog:  Dog[] pets;  pets = **new** Dog[7]; |  |
|  | Crea dos objetos y asígnalos a los dos primeros elementos del array:  pets[0] = **new** Dog();  pets[1] = **new** Dog(); |  |
|  | Llama a los métodos desde los dos elementos:  pets[0].setSize(30);  **int** x = pets[0].getsize();  pets[1].setSize(8); |

# La diferencia entre variables de instancia y variables locales

|  |  |
| --- | --- |
|  | Las variables de instancia son declaradas en una clase no en un método.  **public** **class** Horse {  **private** **double** height = 15.2;  **private** String breed;  //more code  } |
|  | Las variables locales son declaradas dentro de un método.  **public** **class** AddThing {  **int** a;  **int** b = 12;  **public** **int** add() {  **int** total = a + b;  **return** total;  }  } |
|  | Las variables locales DEBEN ser inicializadas antes de usarlas!    Incluso los IDE lo indican. |

# Comparando variables (primitivos o referencias)

A veces quieres sabes si dos variables de tipo primitivas son las mismas. Es facilísimo! Solo usa el operador ==.

Otras veces quieres sabes si dos variables de referencia…referencian a un mismo objeto en el heap. Igual de fácil, volvemos a usar el operador ==.

Pero a veces quieres saber si dos objetos son iguales. Y para eso, necesitas usar el método equals(). La idea de igualdad para objetos depende del tipo de objeto. Por ejemplo, si dos objetos String tienen los mismos caracteres, son significativamente equivalentes, independientemente si dos objetos distintos en el heap.

¿Pero qué hay acerca de un Dog? ¿Quieres operar dos Dogs como iguales si tienen el mismo tamaño y peso? Probablemente no. De esta forma, si dos objetos diferentes deben ser tratados como iguales mucho va a depender de lo que asumamos como igualdad para ese objeto. Este tema se verá más adelante. El punto central en esta parte es usar el operador == para comparar los bits en dos variables. Lo que representan, no importa.

Para comparar dos primitivos, usa el operador ==. Este puede ser usado para comparar dos variables de cualquier tipo, simplemente compara los bits.

**if** (a == b) {...} revisa los bits de a y b, para retornar true si el patrón es el mismo (sin importar el tamaño de la variable, los ceros a la izquierda no cuentan).

|  |  |
| --- | --- |
| **int** a = 3;  **byte** b = 3;  **if** (a == b) {  // true  } |  |

Para ver si dos referencias son las mismas (lo que significaría que referencias al mismo objeto en el heap) usa el operador ==.

Recuerda, el operador == solo toma en cuenta el patrón de bits en la variable. Las reglas son las mismas si la variable es una referencia o un primitivo.

Entonces el operador == retorna true si dos variables de referencia apuntan al mismo objeto! En ese caso, no sabemos qué patrón de bits tienen ya que eso depende de la JVM y está oculto a nuestros ojos, pero sabemos que todo es que podrían referencian al mismo objeto.

|  |  |
| --- | --- |
| Foo a = **new** Foo();  Foo b = **new** Foo();  Foo c = a;  **if** (a == b) {  // false  }  **if** (a == c) {  // true  }  **if** (b == c) {  //false  } |  |

# ¿Es legal?

De acuerdo al siguiente método, que está correcto en todos los sentidos:

**int** calcArea(**int** height, **int** width){

**return** height \* width;

}

Indica cuales de las siguientes líneas no son legales o tienen algún problema

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | **Es legal?** | |
| 1 | **int** a = calcArea(7, 12); | Si | No |
| 2 | **short** c = 7; | Si | No |
| 3 | calcArea(c, 15) ; | Si | No |
| 4 | **int** d = calcArea(57); | Si | No |
| 5 | calcArea(2,3) ; | Si | No |
| 6 | **long** t = 42; | Si | No |
| 7 | **int** f = calcArea(t, 17); | Si | No |
| 8 | **int** g = calcArea(); | Si | No |
| 9 | calcArea(); | Si | No |
| 10 | **byte** h = calcArea(4, 20); | Si | No |
| 11 | **int** j = calcArea(2, 3, 5); | Si | No |

# Sé el compilador

Para seguir ejercitándonos haremos unos pequeños ejercicios donde nos convertiremos (momentáneamente) en el compilador de Java:

|  |  |
| --- | --- |
| **public** **class** XCopy {  **public** **static** **void** main(String[] args) {  **int** orig = 42;  XCopy x = **new** XCopy();  **int** y = x.go(orig);  System.*out*.println(orig + " " + y);  }    **int** go(**int** arg){  arg = arg \* 2;  **return** arg;  }  } |  |
| **public** **class** Clock {  String time;    **void** setTime(String t){  time = t;  }    **void** getTime(){  **return** time;  }  }  **public** **class** ClockTestDrive {  **public** **static** **void** main(String[] args) {  Clock c = **new** Clock();  c.setTime("1245");  String tad = c.getTime();  System.*out*.println("time: " + tod);  }  } |  |