# UNIDAD 3: PROGRAMACIÓN ORIENTADA A OBJETOS 1

## Índice



# 1. Introducción a la Programación Orientada a Objetos (POO)

En los años 60 la programación se realizaba de un modo "clásico" (no orientado a objetos). Un programa era un código que se ejecutaba, los trozos de código que se podían emplear en varias ocasiones a lo largo del programa (reutilizar) se escribían en forma de procedimientos que se invocaban desde el programa, y esta era la única capacidad de reutilización de código posible.

Según los códigos se fueron haciendo más grandes y complejos este estilo de programación se hacía más inviable: es difícil programar algo de grandes dimensiones con este estilo de programación. La única posibilidad de repartir trozos de código relativamente independientes entre programadores son los procedimientos, y al final hay que juntar todos estos con el programa central que los llama, siendo frecuente encontrar problemas al unir estos trozos de código.

En los años 70 se empezó a imponer con fuerza otro estilo de programación: POO, programación orientada a objetos (en la literatura suele aparecer como OOP, (Object Oriented Programming). Aquí un programa no es un código que llama a procedimientos, sino que un programa es un montón de objetos, independientes entre si, que dialogan entre ellos pasándose mensajes para llegar a resolver el problema en cuestión.

A un objeto no le importa en absoluto cómo está implementado otro objeto, qué código tiene o deja de tener, qué variables usa.... solo le importa a qué mensajes es capaz de responder. Un mensaje es la invocación de un método de otro objeto, es decir, cuando un objeto recibe un mensaje lo que hace es ejecutar el método asociado. Un método es muy semejante a un procedimiento de la programación clásica: a un método se le pasan uno, varios o ningún dato y nos devuelve un dato a cambio.

Si hay que repartir un programa de grandes dimensiones entre varios programadores, a cada uno se le asignan unos cuantos objetos, y en lo único que tendrán que ponerse de acuerdo es en los mensajes que se van a pasar. La forma en que un programador implemente sus objetos no influye en absoluto en lo que los demás programadores hagan. Esto es así gracias a que los objetos son independientes unos de otros (cuanta mayor sea la independencia entre ellos de mayor calidad serán).

Si analizamos lo que hemos dicho hasta aquí de los objetos, veremos que estos parecen tener dos partes bastante diferenciadas: la parte que gestiona los mensajes, que ha de ser conocida por los demás, y que no podremos cambiar en el futuro sin modificar los demás objetos (*sí es posible añadir nuevos métodos para dar nuevas funciones al objeto sin modificar los métodos ya existentes*). La otra parte es, el mecanismo por el cual se generan las acciones requeridas por los mensajes, esto es, el conjunto de variables que se emplean para lograr estas acciones. Esta segunda parte es, en principio, totalmente desconocida para los demás objetos (*a veces no es así, pero es lo ideal en un buena POO*). Por ser desconocida para los demás objetos podemos en cualquier momento modificarla sin que a los demás les importe, y además cada programador tendrá total libertad para llevarla a cabo como él considere oportuno.

La POO permite abordar con más posibilidades de éxito y con un menor coste temporal grandes proyectos de software, simplificándole la tarea al programador.

¿Por qué programar con objetos?

Objetos, objetos, objetos. Parece que ésta es la palabra clave de este nuevo paradigma de programación que estamos introduciendo. Pero, ¿qué son los objetos y por qué son tan especiales? Mira a tu alrededor, vivimos en un mundo de objetos, las personas pensamos en términos de objetos. Existen coches, aviones, personas, animales, edificios, semáforos, ascensores, etc. Todos ellos son objetos. Ahora la pregunta es, ¿qué tienen en

común todos ellos? ¿Qué describe a un objeto, pero no a un objeto concreto, sino a un objeto en general? Si reflexionamos, nosotros, las personas, aprendemos acerca de los objetos al estudiar sus atributos y al observar su comportamiento. Eso es precisamente lo que define a los objetos:

Todos los objetos tienen atributos que los describen (propiedades o atributos). Así, por ejemplo, un objeto "pelota" puede venir definida por su peso, su color, el material del que está hecha y su tipo (si es de fútbol, baloncesto, etc.). Un objeto bebé puede venir definido por su edad, su peso, su altura, su color de piel, su color de ojos y su color de pelo; y un objeto "coche" puede venir definido por su marca, su color, el número de puertas y el tipo de combustible que usa.

Representación lógica del "Soldado" -> Clase Soldado

Todos los objetos de la misma clase o tipo vendrán definidos por los mismos atributos; es decir, todas las pelotas vienen definidas por los cuatro atributos anteriormente citados: su peso, su color, el material y su tipo. Sin embargo, cada pelota particular, cada objeto pelota, tendrá unos valores concretos para dichos atributos. Así, por ejemplo, la pelota número uno pesa 1 Kilogramo, es de color rojo, está hecha de plástico y es una pelota de fútbol, mientras que la pelota número dos, pesa 1.5 Kilogramos, es de color rojo, está hecha de cuero y es una pelota de baloncesto.

Todos los objetos exhiben un comportamiento o realizan operaciones que especifican lo que hacen o para qué sirven (Métodos). Así, por ejemplo, el objeto pelota rueda, rebota, se infla, se desinfla; el objeto bebé duerme, come, llora y un coche acelera, frena y gira. Al igual que antes, todos los objetos de la misma clase o tipo tendrán el mismo comportamiento.

Con esta filosofía de programación, para plantear una solución a un problema dado, el programador tendrá que determinar los objetos involucrados en él, sus características comunes y las acciones que se pueden realizar con esos objetos *(esta parte la estudiaréis en el módulo de Entornos)*. Una vez localizados y analizados los objetos que intervienen en el problema real, tan sólo tendrá que trasladar éstos de manera directa al programa informático. Toda la potencia del paradigma de programación proviene precisamente de esta correspondencia directa entre el dominio del problema y el dominio de la solución, es decir, la correspondencia directa entre la realidad modelada y el programa que la modela. La programación orientada a objetos nos brinda una forma natural de ver el proceso de programación, a saber, mediante el modelado de objetos reales, sus atributos y su comportamiento. Con esta base, la resolución del problema se convierte en una tarea sencilla y bien organizada:

• El enfoque orientado a objetos trata de manera conjunta los procesos y los datos dentro del concepto de objeto y trata de realizar una abstracción lo más cercana al mundo real a través de objetos.

• Estos objetos encierran, no sólo los datos que le dan forma y entidad al objeto, sino también la funcionalidad necesaria para manipular dicho objeto, formando una entidad compacta y, por tanto, reutilizable.

• La programación orientada a objetos es una nueva manera de atacar los problemas de programación en la que un problema se divide en pequeñas unidades lógicas de código, que incluyen datos y funciones, que son independientes del resto del programa y que interactúan entre sí.

• A estas pequeñas unidades lógicas de código se les ha denominado objetos para establecer una analogía entre las mismas y los objetos materiales del mundo real.

*Por ejemplo, una vez inventado un objeto para apretar tornillos, podremos utilizarlo siempre en uno o en otro programa, es decir, cada vez que necesitemos un objeto que apriete tornillos (no vamos a inventar una herramienta cada vez que necesitemos apretar un tornillo).*

Con la teoría orientada a objetos, construimos la mayoría del software del futuro mediante la combinación de "partes estándares e intercambiables" llamadas clases:

• Veremos que las clases son a los objetos, lo que los planos son a las casas. Podemos construir muchas casas a partir de un plano, de igual manera que podemos crear instancias de muchos objetos a partir de una clase.

• Asimismo, veremos que el software organizado en clases, se puede reutilizar en futuros sistemas de software.

A partir de aquí comenzaremos a aprender los conceptos y técnicas de la programación orientada a objetos que nos lleven al desarrollo de software de calidad, extensible y reutilizable, utilizando para ello el lenguaje de programación.

Java como hilo conductor. No obstante, antes de continuar, puede que te estés planteando la siguiente pregunta: si la programación orientada a objetos parece ser el mejor camino para conseguir software de calidad, ¿por qué empezamos el curso viendo los conceptos y técnicas de la programación estructurada en lugar de empezar directamente por la programación orientada a objetos? La respuesta es sencilla, para construir los objetos haremos uso de las técnicas de la programación estructurada, de manera que necesitábamos establecer primero las bases de ésta.

# 2. Las clases en Java

## 1. Concepto de clase

¿Has jugado alguna vez en la playa a hacer castillos de arena? Para jugar sólo necesitamos un cubo, el cual llenamos de arena húmeda y a partir del cual podemos hacer multitud de castillos. El cubo es como un molde y su diseño será lo que marcará el aspecto de todos los castillos de arena que se hagan a partir de él. Si queremos otro tipo de castillo, necesitaremos otro cubo distinto que tenga otro diseño. Podemos pensar en una clase como en el cubo del juego descrito, mientras que los objetos serían los distintos castillos de arena concretos que se crean a partir del molde o clase. Cada uno de los castillos de arena es un objeto distinto, con entidad propia, pero todos ellos tienen la misma forma al proceder del mismo molde.

Una clase es una plantilla que define la forma de un tipo de objetos. En esta plantilla se especifican los atributos y el comportamiento con los que van a contar los objetos que se construyan a partir de dicha plantilla. Por tanto, podemos afirmar que una clase describe objetos que van a tener la misma estructura y el mismo comportamiento.

Además, el hecho de crear un objeto a partir de una clase, que equivaldría a la acción de crear un castillo de arena a partir de un cubo, recibe el nombre de **instanciar un objeto**. Así, podemos afirmar que los objetos son las instancias de las clases, cumpliéndose durante la ejecución de una aplicación que:

- Todo objeto es instancia de una única clase.
- Toda clase que forma parte del programa tiene, en un instante dado, cero o más objetos que son instancia de ella.

Pero, para el programador, ¿qué son más importantes: las clases o los objetos? Evidentemente, ambos son importantes, pues un programa orientado a objetos es:

- Una colección estructurada de clases que definen los distintos tipos de objetos que van a intervenir en la resolución del problema.
- Una especificación de qué objetos concretos, cuántos y de qué tipo se van a utilizar en la resolución de un problema y cómo van a colaborar dichos objetos para ello.

Una cosa que debemos tener en cuenta para entender la verdadera naturaleza de los objetos es que los objetos no existen hasta que el programa no empieza a ejecutarse.

Como diría nuestro querido niño de la película Matrix “NO HAY CUCHARA”.

https://www.youtube.com/watch?v=ZvznNu1Dse0

A partir de ese momento los objetos empiezan a crearse, a interactuar y a desaparecer según indique el propio programa. Por su parte, cuando el programa está en ejecución lo único que existe son los objetos. Por eso se dice que las clases representan la parte estática de la aplicación, el modelo a partir del cual crear objetos que se van a interrelacionar. Estos objetos y las interrelaciones constituyen la parte dinámica que es en sí la ejecución de la aplicación. Es decir, mientras las clases son estáticas, con semántica, relaciones y existencia previa a la ejecución de un programa, los objetos se crean y destruyen rápidamente durante la actividad de una aplicación.

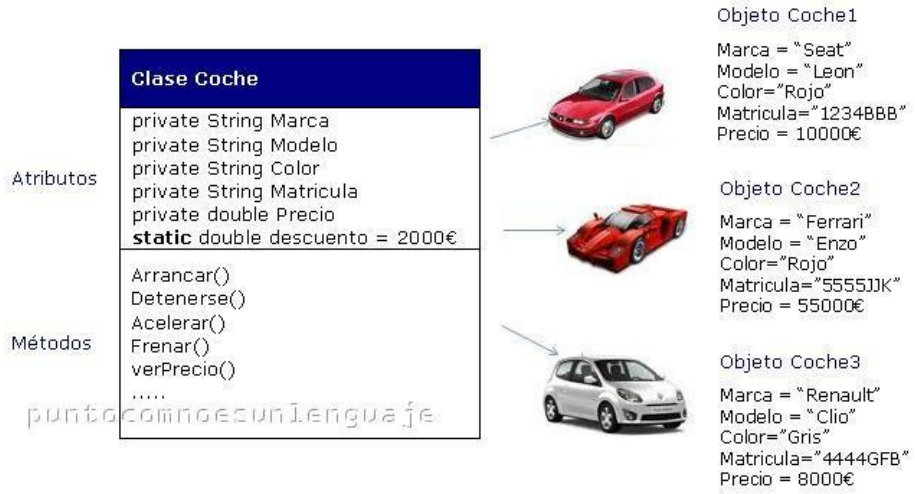
De un modo un tanto simplista, podemos ver una clase como un tipo de datos. Igual que tenemos datos de tipo número entero, datos de tipo número real o datos de tipo carácter, podemos definir también datos de tipo "trabajador" o datos de tipo "departamento", siendo "trabajador" y "departamento" dos clases distintas. Asimismo, de igual manera que declaramos una variable de un cierto tipo de datos, podemos instanciar objetos de una cierta clase.

En la programación orientada a objetos, el tipo de dato y las operaciones asociadas a dicho tipo de dato, aquéllas que pueden usar y/o modificar sus valores, van encapsuladas en la misma entidad, llamada clase.

Por tanto, podemos afirmar que, desde el punto de vista semántico, una clase es un mecanismo de definición de nuevos tipos de datos, donde al mismo tiempo que se describe una estructura de datos para representar valores de un dominio, también se describen las operaciones aplicables sobre las entidades de dicho tipo de datos.

Así, los componentes o miembros de una clase son los siguientes:

- **Los atributos o propiedades.** Son las características y la estructura de almacenamiento que tendrán los objetos de la clase. Nos ofrecen la información sobre el "estado" del objeto.
- **Los métodos.** Son el conjunto de funcionalidades que describen la naturaleza y el comportamiento que tendrán los objetos de la clase; es decir, las operaciones aplicables a dichos objetos. Un método bien construido debería ejecutar una única tarea. Además, los métodos deberían ser el único modo de acceder a los atributos de los objetos.



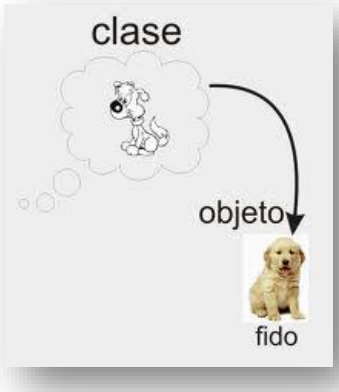

Pero diseñar correctamente una clase, identificarla primero y establecer sus atributos y métodos después, no es una tarea sencilla. Una clase bien diseñada debe definir una única entidad, que agrupe todas sus características y comportamiento, pero que sea una única entidad del mundo o problema que se está modelando.

*De nuevo, decir que estudiaréis más a fondo el diseño de clases en el módulo de ENTORNOS.*

Imagina que para la resolución de un problema necesitamos modelar un coche y la radio del mismo. Todos sabemos que todo coche viene acompañado de un aparato de radio más o menos sofisticado, por lo que podemos caer en la tentación de modelar ambas entidades en una misma clase, estableciendo, por ejemplo, el volumen como atributo del coche, o añadiéndole a la clase "coche" métodos para subir o bajar el volumen de la radio. Sin embargo, el coche y el equipo de radio son entidades a todas luces distintas y, por lo tanto, deben modelarse en clases distintas: debe haber una clase que modele los objetos de tipo "coche" y otra clase que modele los objetos de tipo "radio". Hacer otra cosa es un error muy grave y desafortunadamente, muy frecuente cuando se está empezando y no se tiene experiencia en el modelado orientado a objetos.

## 2. Declaración de clases

Si repasas los distintos ejercicios que hemos ido realizando a lo largo del curso te darás cuenta de que los programas creados siempre eran una clase (dos con la clase Leer).




Ahora, una vez que hemos aprendido el verdadero significado y utilidad de las clases y los objetos, ha llegado el momento de aprender a definir e implementar las nuestras propias, de manera más seria.

A la hora de definir una clase en Java debemos tener en cuenta los siguientes aspectos:

- La **definición e implementación** de una clase en Java se realiza en el **mismo archivo**.
- Es conveniente crear **una sola clase en cada archivo**. Puede haber más de una clase en un archivo siempre que **sólo haya una public**.
- **El archivo** de una clase Java debe tener el **mismo nombre que la clase public que contiene** el archivo.
- En Java, la **palabra clave class** es la que nos va a permitir definir una clase.
- En la definición de la clase se deben incluir los atributos que contiene y los métodos que operan sobre ellos.

- Los nombres de las clases deben ser sustantivos escritos con la primera letra de cada palabra en mayúscula y el resto en minúscula. Los nombres deben contener palabras completas, debe evitarse el uso de siglas y abreviaturas.

Ejemplos:

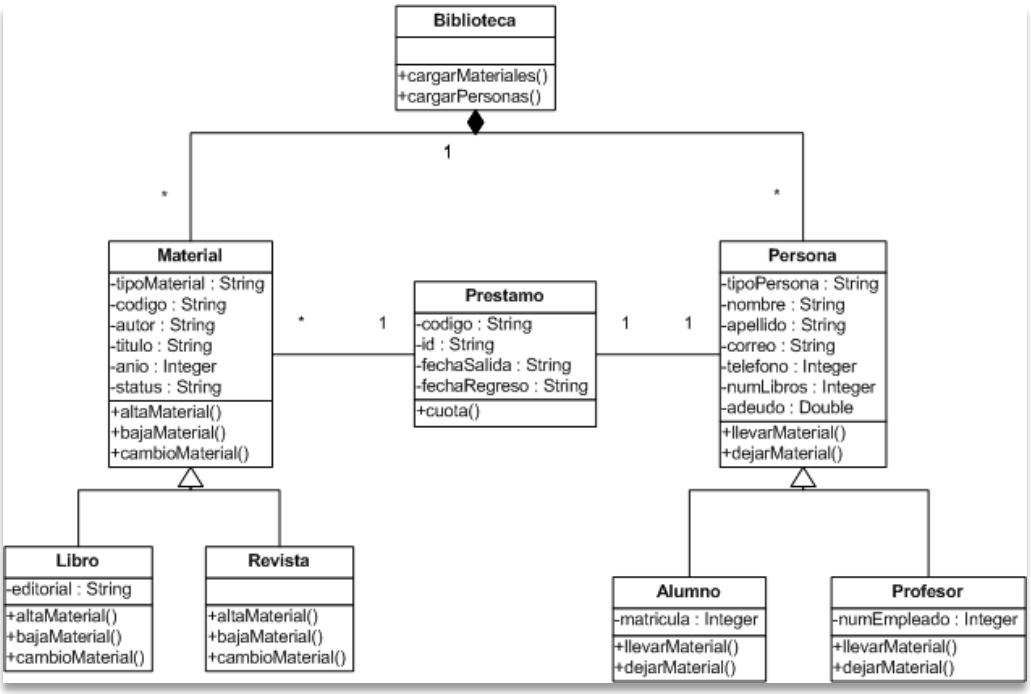
Agenda, Trabajador, CuentaAhorro…

La definición de una clase en Java va a constar de las siguientes partes básicas:

- Cabecera de la clase. En la cabecera de la clase se indica el nombre de la clase y una serie de características de la misma, como son: la clase de la que deriva (es decir, su superclase), los privilegios de acceso a la clase y si la clase implementa, o no, uno o varios interfaces. No te preocupes si no has entendido bien todo esto, pues lo iremos viendo poco a poco a lo largo de este tema y siguientes. De momento, lo que sí debemos tener presente, que aparecerá siempre en la cabecera de la clase, es la palabra reservada class, así como el nombre de la propia clase. Además, por convención, el nombre de la clase debe empezar con una letra mayúscula.

- Cuerpo de la clase. En el cuerpo de la clase se incluye el contenido de la clase, es decir, se definen sus atributos y se declaran e implementan sus métodos. El cuerpo de la clase va entre llaves y al final de la llave de cierre NO se pone punto y coma (;).

- Comentarios. Es conveniente escribir comentarios en cada clase, para facilitar la comprensión. Para ello, usaremos los dos ya vistos anteriormente. Existe otro comentario en java, llamado comentarios de documentación, es el que aparecía cuando habéis abierto una clase ya creada con Eclipse. Java tiene una herramienta llamada *javadoc* que genera documentación de la biblioteca de clases. Este tipo de comentarios deben comenzar con /** y terminar con */ y suelen ir al comienzo de la clase, antes de la cabecera.




Veamos cada una de estas partes más detenidamente, empezando por la cabecera de la clase.

## 1. La cabecera de la clase

Como hemos dicho anteriormente, en la cabecera se indica tanto el nombre de la clase, como una serie de características de la clase que vienen determinadas por lo que se conoce con el nombre de modificadores de la clase. Existen varios modificadores de clase, los cuales pueden aparecer antes de la palabra reservada class o bien después del nombre de la clase.

En POO las clases permiten a los programadores abstraer el problema a resolver ocultando los datos y la manera en que estos se manejan para llegar a la solución (se oculta la implementación). En un programa orientado a objetos es impensable que desde el mismo programa se acceda directamente a las variables internas de una clase si no es a través de métodos getters y setters (por ejemplo, getEdad ( ) o setEdad ()) que ya estudiaremos más adelante.

Algunos de los modificadores de clase que pueden aparecer delante de la palabra reservada *class* son los siguientes:

**MODIFICADORES DE ACCESO**

1. La palabra clave **public**. Indica un nivel de acceso a la clase de tipo público. Cuando una clase ha sido definida como pública, cualquier clase puede hacer uso de ella o relacionarse con ella, bien sea definiendo variables u objetos que sean de esa clase, o por herencia, tal y como veremos en el tema siguiente.

   *OJO: El hecho de que una clase sea pública nos obliga a que el archivo que contiene esa clase se llame igual que la clase pública.*

2. **Cuando no aparece la palabra clave public** (ni ningún modificador), entonces se dice que la clase **es no pública** y **su nivel de acceso es de tipo default (acceso a paquete), lo que quiere decir que sólo pueden relacionarse con ella o hacer uso de ella las clases pertenecientes a su mismo paquete**. Es decir, ninguna clase que no pertenezca al mismo paquete podrá declarar variables u objetos de dicha clase ni podrá heredar de ella.

*Recuerda: Debemos de tener siempre presente que, en un mismo archivo no puede haber definida más de una clase pública, aunque sí pueden coexistir en un mismo archivo una clase pública y otras clases no públicas. Se estudiará más adelante.*

**OTROS MODIFICADORES**

• La palabra clave final. Indica que con esta clase se termina la cadena de herencia; es decir, no va a haber clases que hereden de ella (se verá en temas posteriores).

• La palabra clave abstract. Como veremos en profundidad más adelante, hay algunas clases que pueden tener algún método que esté declarado, pero para el que no se dé su implementación (no dar todo su

contenido), la cual será aportada por las clases que hereden de ella. Puede que ahora mismo esto te suene un tanto extraño, ¡una clase con métodos que no tienen código!, pero no te preocupes, pues en el próximo tema veremos la utilidad y la potencia que nos ofrecen este tipo de clases, de las cuales no se pueden instanciar objetos, y que son conocidas con el nombre de clases abstractas.

Los modificadores de clase que pueden aparecer detrás del nombre de la clase son los siguientes:

• La palabra clave extends. Indica que la clase hereda de otra clase, llamada superclase o clase madre. Puede que esto no te quede muy claro ahora, pero no te preocupes porque abordaremos la herencia en profundidad en el tema siguiente.

• La palabra clave implements. Indica que la clase implementa los métodos de una o varias interfaces. Puede que esto no te quede nada claro ahora, pero no te preocupes porque abordaremos las interfaces, qué son, su utilidad y su uso en profundidad en el tema siguiente.

A continuación, se muestra el aspecto base de una clase con modificadores, donde la notación utilizada en la misma se debe interpretar de la siguiente forma:

• Los corchetes [ ] indican que lo que va dentro de ellos puede aparecer o no. Es, por tanto, optativo ponerlo. Lo que no va entre corchetes, entonces, deberá aparecer obligatoriamente.

• Las tuberías | indican posibilidad de elección entre lo que hay a la izquierda o a la derecha de la tubería, pero sólo se hace uso de uno de ellos.

```
[public] [final | abstract] class Nombre_de_la_clase [extends superclase][implements Interface_1 [, Interface_2]
{
        //Cuerpo de la clase
}
```

Ejemplo:

```
public class Datos {
        //Declaración de los atributos
        String nombre;
        //Declaración de métodos
        Public void imprime ( ) {
                System.out.println (“Me llamo: “ +nombre);
        }
}
```

Estos modificadores también se utilizan para atributos y métodos. De momento, nosotros pondremos SIEMPRE prívate a los atributos y public a métodos y constructores.

## 2. El cuerpo de la clase

Por su parte, el cuerpo de la clase es donde se declaran los atributos que caracterizan a los objetos de la clase y donde se define e implementa el comportamiento de dichos objetos; es decir, donde se declaran e implementan los métodos. La declaración de los atributos de una clase es idéntica a la vista en temas anteriores para las variables "comunes":

```
public class Cuenta {
        private long numero;
        private String titular;
        private float saldo, interesAnual;
        public void ingresar (float cantidad) {
                saldo= saldo+cantidad;
        }
        public void retirar (float cantidad) {
                saldo =saldo-cantidad;
        }
        public void ingresarInteresMes ( ) {
                saldo += interesAnual * saldo / 1200;
        }
        public boolean enRojos ( ) {
                return saldo< 0;
        }
        public float leerSaldo () {
                return saldo;
        }
}
//No es recomendable escribir todo el código en una línea.
```

Se define el tipo de dato de la variable y su nombre.

Sin embargo, además de esto, a los atributos de una clase se le añaden una serie de modificadores que indican una serie de características del atributo. Usando la misma notación que anteriormente, este es el aspecto base de cada atributo de la clase:

[private | protected | public] [static] [final] [transient] [volatile] tipo_atributo Nombre_atributo;

Ejemplo: private int teléfono;

Más adelante en este tema veremos el significado de los modificadores private, protected, public y static. Además, en el tema siguiente veremos el significado del modificador final. Por su parte, los modificadores transient (Utilizado para indicar que los atributos de un objeto no son parte persistente del objeto o bien que estos no deben guardarse y restaurarse utilizando el mecanismo de serialización estándar) y volatile (para indicar al compilador que es posible que dicho atributo vaya a ser modificado por varios threads de forma simultánea y asíncrona, y que no queremos guardar una copia local del valor para cada thread a modo de caché, sino que queremos que los valores de todos los threads estén sincronizados en todo momento, asegurando así la visibilidad del valor actualizado a costa de un pequeño impacto en el rendimiento). NO los veremos en el desarrollo de este módulo, pues el mismo no aborda en ningún momento conceptos de permanencia de objetos ni de programación concurrente, respectivamente. Por el momento es suficiente con saber que la definición de cada atributo de una clase tendrá el siguiente aspecto genérico:

modificador_de_atributo tipo_atributo Nombre_atributo;

Al igual que sucede en la declaración de variables “comunes”, cuando se van a definir varios atributos que son del mismo tipo y que tienen los mismos modificadores, se podrán declarar en una sola sentencia de la siguiente manera:

modificador_de_atributo tipo_atributo Nombre_atributo1, . . ., Nombre_atributoN;

No obstante, por cuestiones de claridad en el código, muchas veces es preferible declarar cada atributo por separado.

Ejemplo:

```
public class Pajaro {
        private char color;
        private int edad;
}
```

# 3. Métodos de una clase

Los métodos asociados a una clase se ubican en el cuerpo de la misma.

Por su parte, la definición de cada método de una clase está formada por dos partes: la cabecera del método, que es donde se declara, y el cuerpo del mismo, que es donde reside su implementación. Esta definición seguirá el siguiente formato básico:

## 1. Cabecera del método

La cabecera del método está formada básicamente por:

• Modificadores del método, que indican una serie de características del método y, aunque ahora simplemente los mencionemos, algunos de ellos los veremos en profundidad más adelante en este tema o en el tema siguiente.

[private | protected | public] [static] [abstract] [final] [native][synchronized]

Más concretamente, hablaremos de los modificadores private, protected, public y static en este tema, y de los modificadores abstract y final en el siguiente. Por su parte, los modificadores native (modificador utilizado cuando un determinado método está escrito en un lenguaje distinto a Java, normalmente C, C++ o ensamblador para mejorar el rendimiento) y synchronized (como volatile pero para métodos y no para variables). NO los veremos en el desarrollo de este módulo, pues el mismo no aborda en ningún momento los conceptos de método nativo ni de programación concurrente (varias partes de un programa se ejecutan simultáneamente) respectivamente.

• Tipo de datos del valor devuelto por el método. Los métodos en Java pueden devolver un valor (SÓLO UNO) y, como Java es un lenguaje fuertemente "tipado", hay que indicar qué tipo de valor se devuelve. Por otra parte, puede ser que un método no devuelva nada, en cuyo caso se indica usando la palabra clave *void* como tipo devuelto por el método. No debes olvidar que omitir el tipo de valor de retorno en la declaración de un método es un error de sintaxis.

• Nombre del método. El nombre del método debe ser lo más indicativo posible de lo que realiza dicho método. Además, por convención, el nombre de un método debe comenzar por *minúscula*.

• Parámetros del método. Entre paréntesis, y separados por comas, después del nombre del método, se puede especificar una lista de parámetros, donde para cada uno de ellos se deberá establecer tanto su tipo como su nombre. Al igual que sucedía con los nombres de los métodos, deberemos elegir los nombres de los parámetros de tal manera que éstos sean significativos, pues esto hará que los programas sean más legibles. Los parámetros del método vendrán definidos de la siguiente forma:

(tipo_parámetro1 nombre_parámetro1,…, tipo_parámetro Nnombre_parámetroN)

Ejemplo:
```
public int sumar (int numero1, int numero2) {
        //Cuerpo del método
}
```

*¡Ojo! Colocar un punto y coma después del paréntesis derecho que encierra a los parámetros es un error de sintaxis. Además, debemos tener muy presente que cada parámetro se declara por separado siguiendo el formato anterior y que no se puede unirla declaración de parámetros que sean del mismo tipo, como se hace en la declaración de variables.*

Por lo tanto, sería un error de sintaxis una declaración de este tipo:

(tipo_parámetro nombre_parámetro1, nombre_parametro2,…,tipo_parámetro Nnombre_parámetroN)

public int suma (int numero1, numero2,…, int numero6)

En los métodos, los parámetros sirven para pasar información al método, la cual podrá ser utilizada por el cuerpo del mismo a través del nombre que se le haya dado. Por su parte, cuando se invoque dicho método deberá haber un argumento en la llamada para cada uno de los parámetros que haya en la declaración del mismo. Además, cada argumento deberá ser compatible con el tipo del parámetro correspondiente, pues no hacerlo así supone un error de sintaxis. Es decir, el encabezado del método y las llamadas al mismo deben coincidir en cuanto al número, tipo y orden de los parámetros y argumentos, pues lo contrario es un error de sintaxis.

Los parámetros son sólo de entrada o, dicho de otro modo, se pasan por valor (realmente se pasa una copia del valor). Es decir, no se devuelven al terminar el método, de tal manera que actúan casi como si fuesen variables locales dentro del método. De hecho, cualquier modificación de su valor que se haga dentro del método no tendrá efecto una vez se salga del mismo. Como veremos más adelante, esto no sucede así cuando lo que se está pasando a un método como parámetro es un objeto, en cuyo caso, se pasa por referencia y las modificaciones a sus valores sí se mantienen al salir del método.

- Las excepciones. Por último, deberá indicarse qué excepciones va a generar el método. Una excepción es un error en tiempo de ejecución que puede ser capturado y tratado para evitar que el programa acabe de forma abrupta. Para indicar qué excepciones puede generar un método, se usa la palabra clave throws, seguida de una lista con las excepciones que puede lanzar, siguiendo la siguiente sintaxis:

throws excepción1, excepción2, …, excepción

*NOTA: De momento sólo pondremos las que el compilador Eclipse nos indique. Hay un tema más adelante dedicado al estudio de las excepciones en Java.*

## 2. Cuerpo del método

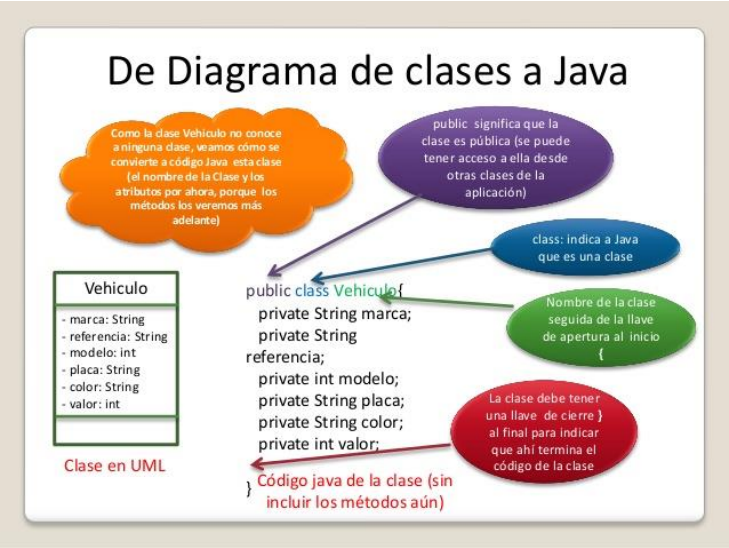
Por su parte, el cuerpo del método contiene las sentencias que llevan a cabo o implementan el comportamiento del método. Estas sentencias comprenden tanto las instrucciones que declaran las variables locales del método, como las instrucciones que llevan a cabo la tarea para la que está diseñado, teniendo en cuenta que:

• Declarar nuevamente el parámetro (lo que va entre los paréntesis) de un método como variable local en el cuerpo del método es un error de sintaxis.

• Si se ha especificado un tipo de valor de retorno en la cabecera del método, entonces no debemos de olvidarnos de devolver dicho valor en el cuerpo del método, pues lo contrario es un error de sintaxis. Por lo tanto, si se especifica un tipo de valor de retorno que no sea void, el método debe contener una instrucción return que devuelva un valor del tipo de valor de retorno del método.

• De manera similar, devolver un valor de un método cuyo tipo de valor de retorno se haya declarado como void es un error de sintaxis.

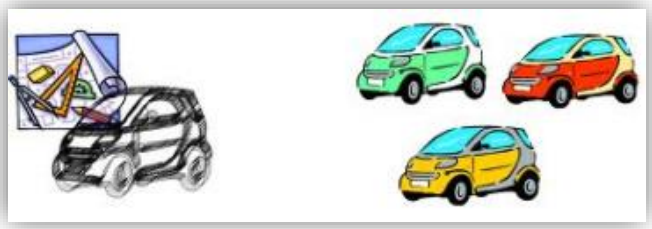
Otro ejemplo de clase a modo de resumen:



# 4. Objetos de una clase

## 1. Objetos o instancias

A lo largo del tema no hemos hecho otra cosa que hablar de lo que son los objetos y de los moldes de los que proceden: las clases; con lo que, llegados a este punto, ya deberíamos tener una idea muy clara de lo que son ambas cosas y para qué se usa cada una de ellas. Es entonces el momento de tratar aspectos un poco más concretos relativos a los objetos y al manejo de los mismos.



Un objeto es una instancia de una clase, creada en tiempo de ejecución. Pero para obtener la instancia de la clase, primero tenemos que declarar una referencia al objeto, que no es más que declarar el objeto igual que se haría con cualquier variable de un tipo básico del lenguaje, con la diferencia de que el tipo será la clase a partir de la cual se va a obtener el objeto. Para ello se usará una sentencia de definición de este tipo:

Nombre_clase nombre_objeto;

Así, por ejemplo, para declarar un objeto de tipo Trabajador, tendríamos que utilizar una sentencia como ésta:

Trabajador unTrabajador; **//Esto es una referencia a un objeto, pero no un objeto en sí mismo (todavía no hay cuachara)**


Una referencia es un valor en tiempo de ejecución que está o vacío o conectado. Si está conectado, una referencia identifica a un único objeto. Nada más crear una referencia, está se encuentra vacía; es decir, el valor inicial de la referencia al objeto es null, que es un valor nulo válido para Java, queriendo decir que la referencia está creada pero que el objeto no está instanciado todavía, por eso la referencia apunta a un objeto inexistente llamado “nulo”. Por lo tanto, una vez declarada la referencia al objeto es el momento de crear la

instancia de la clase, el objeto, que se va a guardar en la referencia creada. Es decir, es el momento de hacer que la referencia pase al estado “conectada”. Para hacerlo se usará una sentencia de este tipo:

nombreclase nombre_objeto= new constructor_de_la_clase (parámetros); //Del igual a la derecha es la instanciación del objeto y se asigna a una referencia

• El operador new se encarga de reservar memoria suficiente para crear una instancia al objeto, asignando a la referencia la dirección de memoria en la que se encuentra el objeto recién instanciado. Si no hubiera memoria suficiente para el objeto, Java daría un error en tiempo de ejecución, pero es difícil que esto ocurra porque Java gestiona muy eficientemente la memoria por medio del recolector de basura (que es un elemento de Java que se encarga de reciclar la memoria de los objetos que han dejado de ser necesitados para que ésta pueda volver a ser utilizada). Ésta es una operación que requiere tiempo, por lo que Java la realiza únicamente cuando lo considera necesario.

• El método constructor de la clase es un método especial de la misma, que se ejecuta exclusivamente en el instante de la creación de una instancia u objeto y que se encarga de inicializar dicho objeto; es decir, de asignar los valores iniciales a los atributos del objeto.

*NOTA: De momento, esto es todo lo que vamos a comentar sobre los métodos constructores, ya que más adelante dedicaremos un apartado exclusivamente a ello.*

En el caso de nuestra clase Trabajador, y suponiendo que el constructor de la clase es un método de nombre “Trabajador” que no necesita parámetros, una vez definida la referencia llamada “unTrabajador”, instanciaríamos el objeto en sí mediante la siguiente sentencia:

unTrabajador = new Trabajador ( );

Es posible realizar la declaración de la referencia al objeto y la instanciación del propio objeto en una única sentencia, que será de la siguiente forma:

Nombre_clasenombre_objeto = new constructor_de_la_clase (parámetros_del_constructor);

En nuestro ejemplo del Trabajador, se traduciría en una sentencia como la siguiente:

Trabajador unTrabajador = new Trabajador ( );

Una vez creado un objeto, éste tiene identidad propia que lo distingue de los demás. Con objetos que son de clases distintas, poder distinguir uno de otro es muy sencillo. Aún con objetos de la misma clase, que tienen los mismos atributos y los mismos patrones de comportamiento, es muy probable que tengan valores distintos

para los atributos y relaciones con objetos distintos, con lo que podremos también distinguirlos fácilmente. Sin embargo, son posibles objetos de la misma clase que tengan unos valores idénticos para sus atributos y también las mismas relaciones con otros objetos. Si esto sucede, ¿son objetos distintos o es que son el mismo objeto?

Todos los objetos, incluso los que son de la misma clase, poseen su propia identidad y se pueden distinguir entre sí, independientemente de los valores de sus atributos y de sus relaciones con otros objetos. El término identidad significa que los objetos se distinguen por su existencia inherente y no por las propiedades descriptivas que puedan tener. Es decir, es posible que dos objetos distintos de la misma clase tengan exactamente los mismos valores para los atributos, pero eso no quiere decir que sean el mismo objeto.

Imagina dos coches exactamente iguales: misma marca, mismo modelo, mismo color, mismos accesorios; todo igual. ¿Dejan por ello de ser dos entidades distintas? No, siguen siendo dos coches distintos, aunque las características que los definen sean idénticas. Internamente, podemos obtener un identificador único para cada objeto a través de su "firma" con la función hashCode ( ). *Se verá en su momento.*

## 2. Miembros de objetos o miembros de clase

Acabamos de recordar en el apartado anterior lo que supone instanciar un objeto: reservar espacio en memoria para los miembros del objeto, es decir, para sus atributos y su comportamiento, tal y como venía definido en la clase a partir de la cual ha sido instanciado. Además, sabemos que:


• Los atributos no son más que variables que van a almacenar los valores que caracterizan al objeto.

• El comportamiento del objeto viene dado por sus métodos, que actúan sobre los atributos del objeto consultando o modificando sus valores.

Cada objeto que se instancia tiene su propia zona de almacenamiento en memoria donde está almacenada una copia, de uso exclusivamente suya, de los atributos que caracterizan a un objeto de su clase. Es por ello que, a este tipo de atributos o variables, que son propios y particulares de cada objeto que se instancia, se les conoce con el nombre de variables de instancia, y así es como los llamaremos nosotros a partir de ahora. Por su parte, a los métodos que hacen uso de estas variables de instancia, ya sea para su consulta o modificación, reciben el nombre de métodos de instancia. En general, tanto a las variables como los métodos de instancia, reciben el nombre de miembros de objeto.

Pero existe otro tipo de variables y métodos de una clase que no son de instancia. Veamos cómo se llaman y para qué sirven.

*Imagina que estamos programando un videojuego con marcianos y otras criaturas espaciales. Cada marciano tiende a ser valiente y deseoso de atacar a otras criaturas espaciales cuando sabe que hay, al menos, otros cinco marcianos presentes en pantalla. Por el contrario, si están presentes menos de cinco marcianos, cada marciano se vuelve cobarde.*

*Por tanto, si tenemos la clase "Marciano" y cada marciano del videojuego es una instancia de dicha clase, necesitamos algún mecanismo para que cada marciano sepa en todo instante cuántos marcianos hay en pantalla y así saber si tiene que comportarse como un valiente o como un cobarde. Podríamos optar por dotar a la clase "Marciano" con una variable de instancia "cuentaMarcianos", pero esto supondría que cada vez que se crease o se matase un marciano habría que actualizar dicha variable "cuentaMarcianos" en todos y cada uno de los objetos marciano. Este mecanismo de copias redundantes desperdicia tanto espacio como tiempo, además de ser un proceso propenso a errores, por lo que no es un mecanismo adecuado. En este caso, para poder solucionar este problema de manera eficiente, necesitaríamos poder contar con una variable que fuese compartida por todos los marcianos. Pues bien, estas variables existen y reciben el nombre de variables de clase.*

Una variable de clase representa información en toda la clase; es decir, es una variable que será compartida por todos los objetos de la clase. Así, cuando un objeto modifique dicha variable, esta modificación será visible a todos los objetos de la misma clase, pues la variable es la misma para todos ellos al ser una variable compartida. Como programadores, al diseñar una clase, definiremos una variable de clase cuando todos los objetos de la clase tengan que utilizar la misma copia de la variable.

En el ejemplo del videojuego de marcianos, como ya hemos dicho, usar una variable de clase para llevar la cuenta de los marcianos que hay en pantalla nos ahorra espacio y tiempo; al haber sólo una copia de la variable compartida por todos los objetos, no tenemos que incrementar cada una de las copias de cada uno de los objetos, que es lo que pasaría si hubiésemos usado una variable de instancia, ya que cada objeto tendría la suya propia.

En nuestra clase Trabajador necesitamos tener al menos dos variables de clase:

• Una que lleve la cuenta del número de trabajadores que tiene la empresa, información que necesitamos saber para aplicar el complemento de productividad al sueldo de los mismos, pues este complemento viene en función del número de trabajadores.

• Otra que almacene las posibles categorías profesionales que pueden tener los trabajadores de la empresa y entre las cuales tomará su valor la variable de instancia categoriaProfesional de cada objeto Trabajador.

Para crear una variable de clase haremos uso del modificador static cuando declaremos la variable, de ahí que también reciban el nombre de variables estáticas, y se usará para ello una sentencia como ésta:

static tipo_variable_clase nombre_variable_clase;

En el código de nuestra clase Trabajador, la definición de las variables de clase anteriormente descritas, se hace de la siguiente manera:

private static int numTrabajadores=0;

private static String[][]categorias={{"empleado","encargado","directivo","prácticas"},{"25","50","500","0"}};

No debemos olvidar que, aunque las variables de clase pueden parecer variables globales, tienen alcance a nivel de clase; es decir, sólo son compartidas por los objetos que sean de la clase en la cual se ha definido dicha variable y no por todos los objetos del programa.

Hasta ahora hemos hablado únicamente de las variables de clase, pero también existen métodos de clase, que junto a las primeras forman lo que se conoce como miembros de clase. Si las variables de clase se usan para manejar datos que deben compartirse entre todos los objetos de la clase, por su parte, los métodos de clase se usan solamente para acceder a las variables de clase. Para definir un método de clase, también se hace usando el modificador static en la cabecera del método, de ahí que también reciban el nombre de métodos estáticos, y se usará para ello una sentencia como ésta:

static tipo_devuelto_metodo nombre_metodo_clase (parámetros_metodo)

En nuestra clase Trabajador, tenemos un método de clase que servirá para añadir categorías a la variable de clase *categorias*. Dicho método tiene la declaración siguiente:

```
public static void addCategoria (String categoria, int complementoSueldo){
        // Código del método
}
```

Se puede acceder a los miembros de clase a través de una referencia a cualquier objeto de dicha clase, de igual manera que se accede a cualquier otro miembro de la clase. Suponiendo que “unObjeto” es una referencia conectada a un objeto, suponiendo que “unaVariableClase” es una variable de clase y suponiendo que “unMetodoClase” es un método de clase de dicho objeto, el acceso a estos miembros de clase se haría de la siguiente manera:

unObjeto.unaVariableClase

unObjeto.unMetodoClase()

Por lo tanto, suponiendo que existiese una variable llamada *unTrabajador* que fuese una referencia a un objeto Trabajador, para acceder a la variable de clase numTrabajadores, suponiendo que ésta fuese pública, se usaría una sentencia como ésta:

unTrabajador.numTrabajadores

Por su parte, para acceder al método de clase addCategoria(), se usaría una sentencia como la siguiente:

unTrabajador.addCategoria(“gerente”,200);

Como podemos ver, para acceder a un miembro de clase se utiliza la notación punto; es decir, interponiendo un punto entre el nombre del objeto al que pertenece el miembro al que se desea acceder y el propio miembro. No obstante, cómo acceder a los miembros de la clase lo veremos en profundidad más adelante, cuando hablemos de los métodos.


Los miembros de clase, tanto las variables como los métodos, tienen una particularidad en cuanto a su uso, y es que pueden utilizarse incluso aunque no se haya instanciado objetos de esa clase y están disponibles tan pronto como la clase se carga en memoria, en tiempo de ejecución. Esto es así ya que, realmente, los miembros de clase están ligados a la clase en sí y no a los objetos de la misma. En este caso, para acceder a un miembro de clase, lo haríamos anteponiendo el nombre de la clase y un punto al miembro de la clase al que queremos acceder. Suponiendo que la clase del ejemplo anterior se llama “NombreClase”, se accedería a sus miembros estáticos de la siguiente manera:

NombreClase.unaVariableClase

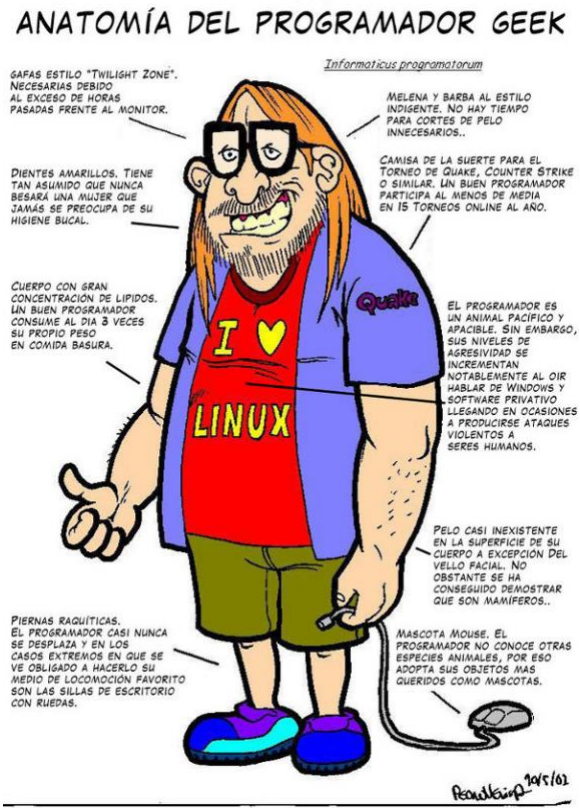
NombreClase.unMetodoClase()

Así, podríamos acceder a la variable de clase numTrabajadores, si ésta fuese pública, de la siguiente forma:

Trabajador.numTrabajadores;

Y podemos invocar al método de clase addCategoria() con la siguiente sentencia:

Trabajador.addCategoria ("gerente",200);




*Expertos programadores recomiendan acceder siempre a los miembros de clase de esta última manera, pues esto enfatiza que el miembro al que se está accediendo es un miembro de clase y hace que otros programadores obtengan esta información en el primer vistazo que echen al código del programa.*

Los miembros de clase, tanto las variables como los métodos, también pueden ir acompañados de un modificador de control de acceso, principalmente el modificador

public o el modificador private:

- Los miembros que sean privados sólo podrán ser accedidos desde los métodos de la propia clase.
- Los miembros que sean públicos podrán ser accedidos desde cualquier clase.
- Al igual que se hace con los miembros de instancia, se recomienda que las variables de clase sean privadas y que los métodos sean públicos, así como que se proporcionen métodos estáticos públicos para acceder a dichas variables estáticas privadas.

## 3. Cuestiones importantes sobre las referencias a objetos

Como hemos visto, para crear un objeto, primero hay que definir una referencia, que inicialmente será una referencia vacía, y luego hay que ligar ésta a la instanciación en sí del objeto. A partir de ese momento la referencia "apunta" al objeto creado (apunta a la dirección de memoria en la que se encuentra), al cual podemos acceder a través de ella.

El hecho de que manejemos los objetos a través de referencias a los mismos tiene varias implicaciones que debemos tener muy en cuenta para no llevarnos sorpresas desagradables. Veamos cuáles son estas implicaciones:

*NOTA: Las referencias y los objetos se asimilan a un televisor (objeto) y un mando a distancia (referencia). A medida que se hace uso de la referencia, se está conectado a la televisión, pero cuando alguien dice "cambia de canal" o "baja el volumen", lo que se manipula es la referencia, que será la que manipule el objeto. Si desea moverse por la habitación y seguir controlando la televisión, se toma el mando a distancia (la referencia), en vez de la televisión.*

*Además, el mando a distancia puede existir por sí mismo, aunque no haya televisión. Es decir, el mero hecho de tener una referencia no implica necesariamente la existencia de un objeto conectado al mismo. De esta forma si se desea tener una palabra o frase, se crea una referencia String:*

*String S;*

*Pero esta sentencia solamente crea la referencia, y no el objeto. Si se decide enviar un mensaje a S en este momento, se obtendrá un error (en tiempo de ejecución) porque S no se encuentra, de hecho, vinculado a nada (no hay televisión). Una práctica más segura, por consiguiente, es inicializar la referencia en el mismo momento de su creación:*

*String S = "asdf";*

*Sin embargo, esta sentencia hace uso de una característica especial de Java: las cadenas de texto pueden inicializarse con texto entre comillas. Normalmente, es necesario usar un tipo de inicialización más general para los objetos (usando new, que ya veremos más a fondo).*

- La asignación entre referencias no implica copia de valores sino de referencias. Suponiendo que tengo dos objetos "Objeto1" y "Objeto2" que son de la misma clase, si tengo una referencia "a" al objeto "Objeto1" y una referencia "b" al objeto "Objeto2", la sentencia de asignación "a = b" no produce una copia de valores de las variables de instancia del objeto "b" a las variables de instancia del objeto "a", sino una copia de referencias.

Esto implica que el objeto "Objeto2" estaría doblemente referenciado, pudiendo acceder a él desde la referencia "a" y desde la referencia "b", mientras que el objeto "Objeto1" quedaría sin referencia y ya no tendríamos ninguna manera de acceder a él.

- Cuidado con el aliasing, que es la posibilidad de tener referencias múltiples a un mismo objeto, como hemos visto en el caso anterior. Si varias referencias apuntan a un mismo objeto, si cualquiera de ellas cambia un valor del objeto, cambiará para todas las referencias. Aunque en ciertos casos nos puede

resultar útil tener múltiples referencias, merece la pena ser cuidadosos con este aspecto, ya que es un posible foco de errores.

- Si la asignación no produce una copia del objeto, sino una copia de referencias, ¿cómo podemos duplicar un objeto? Si queremos duplicar un objeto, es decir, queremos obtener una copia de un objeto que sea independiente del objeto original, hay que definir un método duplicador que se encargue de duplicar el objeto, de forma que en el código de dicho método se cree un nuevo objeto de la misma clase que el que se quiere duplicar y se copien uno a uno los valores de los campos del objeto original en los campos del nuevo objeto. Dicho método devolverá entonces este nuevo objeto, que sí que será independiente del original. Java provee un mecanismo para hacer esto, mediante el interfaz Cloneable y la redefinición del método clone () de la clase Object. (Se verá en su momento).

- Sabemos que cuando se pasa un parámetro a un método, si éste es de un tipo primitivo del lenguaje Java, el parámetro se pasa por valor, por lo que al salir del método sigue manteniendo su valor original, aun cuando el parámetro hubiese sido modificado dentro del método. Pero, ¿qué sucede cuando lo que se pasa como parámetro a un método es la referencia a un objeto? Cuando se pasa un objeto como parámetro a un método, lo que se va a pasar al interior del método es la referencia al mismo, con la implicación que esto tiene:

- Cualquier cambio que se haga en el parámetro va a hacerse en la referencia, con lo que el cambio quedará registrado en el objeto incluso al salir del método. La anterior afirmación es cierta sólo en el caso de objetos no inmutables (OJO los objetos de la clase String y los envoltorios de los tipos básicos son objetos inmutables). Por lo tanto, tenemos que ser cuidadosos con esto. En algunos casos nos interesará duplicar el objeto y pasar la copia como parámetro al método para no modificar así el original.

- Al igual que cuando realizamos una asignación entre referencias no estamos haciendo una copia de valores sino de referencias, ¿qué sucede cuando comparamos dos objetos con el operador de igualdad "==" o el operador de desigualdad "!="? Evidentemente, al igual que sucedía con la asignación, cuando comparamos dos referencias no estamos comparando los valores de los objetos a los que apuntan, sino las propias referencias. Por lo tanto, la igualdad será cierta si y sólo si ambas referencias comparadas apuntan al mismo objeto. Sin embargo, si apuntan a objetos distintos, aun cuando éstos contienen los mismos valores para todos sus atributos, la igualdad no será cierta.

- Para poder realizar la comparación entre objetos por los valores de sus atributos, deberemos hacer lo mismo que para la copia: definir un método comparador que se encargue de comparar uno a uno los valores de las variables de instancia de los dos objetos y nos diga si los objetos son iguales campo a campo. Java provee un mecanismo para realizar esto, por medio de los interfaces Comparable y Comparator. Ejemplo:

La comparación de strings nos da la oportunidad de distinguir entre el operador lógico == y la función miembro *equals* de la clase *String*. En el siguiente código

```
String str1="El lenguaje Java";
String str2=new String("El lenguaje Java");
        if(str1==str2){
                System.out.println("Los mismos objetos");
        }else{
                System.out.println("Distintos objetos");
        }

        if(str1.equals(str2)){
                System.out.println("El mismo contenido");
        }else{
                System.out.println("Distinto contenido");
        }
```

# 5. Métodos y Mensajes

## 1. Generalidades

La cuestión clave que queda por resolver es la siguiente: ¿cómo se manda un mensaje a un objeto? Bien, para mandar un mensaje desde un objeto a otro objeto:

• En primer lugar, el objeto emisor debe poseer una referencia al objeto receptor, pues sólo se podrá enviar un mensaje a un objeto desde otro objeto que sea cliente del mismo. Una clase es cliente de otra si tiene algún atributo, parámetro o variable local de alguno de sus métodos que sea de ese tipo.

• Una vez que tenemos la referencia al objeto receptor del mensaje, para enviarle un mensaje utilizaremos lo que se conoce con el nombre de notación punto, que no

es más que interponer un punto entre el nombre de la variable que es referencia al objeto receptor y el nombre del método que deseamos invocar para su ejecución. Suponiendo que la variable "unObjeto" es una referencia a un objeto que está definida en el objeto desde el que vamos a enviar el mensaje, en el código de este último objeto debería aparecer una sentencia como ésta:

```
unObjeto.unMetodoInstancia();
```

Suponiendo que existiese una variable llamada *unTrabajador* que fuese una referencia a un objeto Trabajador, a continuación, se presentan una serie de mensajes para ese objeto:

```
//Para obtener el NIF, sueldo y edad

unTrabajador.getNif();
unTrabajador.getSueldo();
unTrabajador.getEdad();
```

## 2. Sobrecarga de métodos

Observa el código de la clase Trabajador que se te ha proporcionado y fíjate el en método setNif ( ). ¡Oye, hay dos métodos setNif ( )! ¿Cómo es esto posible?, te estarás preguntando. Vamos a verlo.

Imagina que sólo existiese el primero de los métodos setNif(). Este método establece el NIF del trabajador a partir de una cadena de caracteres que deberá estar formada por ocho números y una letra. Sin embargo, puede ser que queramos tener la posibilidad de establecer el NIF del trabajador a partir del DNI, que no lleva letra, y la letra, proporcionada al método de manera independiente y no como una única cadena junto a los ocho números del DNI.

Ante una situación como ésta, ¿qué puedo hacer? En principio, según la filosofía de programación tradicional, la única manera sería definir en la clase otro método, llamado por ejemplo setNif2 (), que recibiese como parámetros una cadena de ocho números por un lado y una letra por otro. De esa manera, según los parámetros a partir de los cuales quisiera establecer el NIF del trabajador, llamaría a un método o a otro.

Sin embargo, Java permite declarar en la misma clase varios métodos distintos con el mismo nombre, siempre y cuando éstos tengan distintos conjuntos de parámetros, los cuales se determinan mediante el número y los tipos de los parámetros. A esta técnica se la conoce con el nombre de sobrecarga de métodos. Por lo tanto, podría tener dos métodos setNif(), cada uno recibiendo unos parámetros distintos, pero teniendo el mismo efecto: establecer el NIF del trabajador.

*¿Para qué puedo querer sobrecargar un método?* Generalmente, la sobrecarga se utiliza para crear varios métodos con el mismo nombre que realizan tareas similares, pero en tipos de datos distintos. Además, sobrecargar métodos que realizan tareas estrechamente relacionadas, pero no idénticas, puede hacer que los programas sean más legibles y comprensibles. En nuestro caso de la clase Trabajador, si ambos métodos van a establecer el NIF del trabajador, ¿por qué tienen que llamarse de manera distinta? Al tener ambos métodos el mismo nombre, todo se vuelve más sencillo para el programador.

Cuando como programadores, decidimos sobrecargar algún método, debemos de tener en cuenta los siguientes factores:

• Los métodos sobrecargados se distinguen por su firma: una combinación del nombre del método, el número y tipo de sus parámetros. De hecho, cuando se llama a un método sobrecargado, el compilador de Java selecciona el método apropiado examinando el número y los tipos de los argumentos en la llamada. Así, por ejemplo, observa las distintas firmas de los dos métodos setNif () de la clase Trabajador:

```
public void setNif (String nif){
        // Código del método
}
public void setNif (String dni, char letra){
        // Código del método
}
```

• De lo anterior se deriva que no puede haber en una misma clase dos métodos que se llamen igual y que además tengan el mismo número y tipo para sus parámetros.

*Declarar métodos con el mismo nombre y con listas idénticas de parámetros es un error de sintaxis y daría un error en tiempo de compilación.*

• Observa también que a la hora de hablar de métodos sobrecargados no hemos mencionado en absoluto el valor de retorno de los mismos, ya que éstos no pueden distinguirse en base al tipo de valor de retorno. Es decir, los métodos sobrecargados pueden tener el mismo o distinto tipo de valor de retorno, pero en lo que no pueden coincidir nunca es en su firma: declarar dos métodos con la misma firma produce un error de sintaxis, tengan éstos el mismo tipo de valor de retorno o un tipo de valor de retorno distinto.

# 6. Constructores

## 1. Generalidades

Mira el siguiente video:

https://www.youtube.com/watch?v=oGiv71BdKAc&list=PL4qVHq3C8rcHtXHtHtmI5D2pjmCQOezpA&index=11

Antes de entrar de lleno en lo que son los constructores y para qué se utilizan, detengámonos un instante para recordar qué sucede cuando declaramos una variable para poder almacenar en ella algún dato. Básicamente lo que sucede es lo siguiente:

• Se reserva en memoria espacio suficiente para albergar el dato que debe almacenar la variable, donde el tamaño del espacio que se reserva dependerá del tipo de dato que se haya elegido para la propia variable.

• Esa zona de memoria es donde va a estar almacenado el contenido de la variable; es decir, su valor. Por lo tanto, cada vez que leamos el contenido de esa zona de memoria estaremos leyendo el valor almacenado en la variable, y cada vez que escribamos en esa posición de memoria estaremos modificando su valor.

• A la posición de memoria en la que empieza el espacio reservado se le asocia un identificador, que será el nombre que le hayamos dado a la variable en nuestro código y el que, como programadores, usaremos para referirnos a dicha porción de memoria cada vez que queramos leer de ella su contenido o escribir en ella algún valor.

Es como si la memoria fuese un enorme casillero dividido en celdas y el proceso de definir una variable equivaliese a asignar una de esas celdas a una persona para que pueda guardar en ella sus pertenencias. Lo normal es que antes de asignar una celda a una persona, el personal de limpieza se encargue antes de vaciar el contenido que pudiese albergar de su anterior propietario, para que el nuevo dueño se la encuentre como nueva.

Ahora, la cuestión es: cuando se declara una nueva variable, ¿quién le garantiza al programador que, en la posición de memoria asignada para la misma, no hay datos (basura) anteriores?

Para solucionar este problema, las buenas técnicas de programación recomiendan al programador escribir expresamente un valor inicial en las variables declaradas en el programa antes de comenzar a usarlas, para así poder estar seguro de que éstas no contienen ningún valor basura sino el valor que se les acaba de dar.

A la acción de dar un valor inicial a las variables recién declaradas se le denomina inicialización. Si revisamos el código de cualquier programa estructurado, casi con total seguridad encontraremos al principio una zona con las sentencias de declaración de variables e, inmediatamente después, las sentencias de inicialización de las mismas, si bien es cierto que en la mayoría de los lenguajes es posible realizar ambas acciones, la declaración y la inicialización, de manera conjunta en una misma sentencia. Revisa algunos de los programas que ya has hecho a lo largo del curso. ¿Has estado inicializando todas tus variables? Seguro que sí, pues ya estamos convirtiéndonos en buenos programadores. Pues bien, al igual que sucede en la programación estructurada, en la programación orientada a objetos también es recomendable inicializar los objetos que se creen antes de usarlos, y es ahí donde intervienen unos elementos que reciben el nombre de métodos constructores o, simplemente, constructores.

Anteriormente en este mismo tema ya vimos el proceso de creación de un objeto. Si recordamos, para instanciar un objeto:

• Teníamos que declarar primero una referencia al mismo; es decir, declarar la variable que va a representar al objeto y a través de la cual vamos a acceder al mismo.

• Una vez definida la referencia, se creaba o instanciaba el objeto en cuestión, que quedaba ligado a la referencia anterior.

• Una vez creado el objeto hay que inicializarlo, dando valores iniciales a los atributos que lo conforman.

Para instanciar un objeto hacíamos uso de una sentencia de este tipo:

nombre_objeto=new constructor_de_la_clase (argumentos_que_se_pasan_al_constructor);

Si hacemos un poco de memoria, recordaremos que:

• La variable con el nombre del objeto es la referencia (mando a distancia) que hemos creado para conectarla al objeto (televisor), una vez instanciado éste.

• El operador new se encarga de reservar memoria suficiente para crear una instancia al objeto, asignando a la referencia la dirección de memoria en la que se encuentra el objeto recién instanciado.

• El propio operador new invoca a lo que se llama constructor de la clase, que inicializa el mismo y completa así su creación.

El método constructor de la clase es un método especial de la misma que se ejecuta exclusivamente en el instante de la creación de una instancia u objeto y que se encarga de inicializar dicho objeto, es decir, de asignar los valores iniciales a todos sus atributos y de realizar, en general, todas las tareas que sean necesarias para poder empezar a utilizar el objeto. Por lo tanto, como programadores, al diseñar una clase deberemos proporcionar un método creador que contenga el código necesario para inicializar un objeto recién creado de la clase.

Pero, ¿qué sucedería si ni siquiera en un método constructor se establece un valor inicial para un atributo? En esos casos, el propio compilador inicializará los atributos para los que no se haya establecido nada en el constructor con sus valores por defecto o valores predeterminados, siendo éstos:

• El valor 0 para los tipos numéricos primitivos,

• El valor false para los valores bolean, y

• El valor null para las referencias.

Por lo tanto, un constructor, al ser de la misma naturaleza, se comporta como cualquier otro método, si bien es cierto que hay ciertas particularidades que los hacen especiales:

• Primeramente, el método constructor es un método que se invoca única y exclusivamente inmediatamente después de la creación del objeto, pues su única misión es la de dar un estado inicial al objeto recién creado.

• El constructor de una clase obligatoriamente debe tener el mismo nombre que la clase, incluyendo las mismas letras en mayúsculas y minúsculas. Además, aunque *no es obligatorio*, dicho método suele aparecer como el primer método definido en la clase.

• Al contrario que cualquier otro método, un constructor no puede especificar un valor de retorno, por lo que en su declaración no aparecerá un tipo de retorno, ni siquiera aparecerá la palabra clave void que se utilizaba para indicar que un método no devolvía ningún valor. Tratar de declarar un tipo de valor de retorno para un constructor, o tratar de devolver mediante una sentencia return un valor en un constructor, es un error.

• Java permite que otros métodos de la clase tengan el mismo nombre de ésta y que especifiquen tipos de valor de retorno, aunque dichos métodos no serán constructores y no serán llamados al instanciar a un objeto de la clase. Java determina qué métodos son constructores localizando los métodos que tienen el mismo nombre que la clase y que no especifican un tipo de valor de retorno. De todas formas, y aunque el lenguaje Java lo permite, es recomendable no dar a ningún método, que no sea constructor, el mismo nombre que la clase.

• Por lo general, los constructores se declaran con un acceso público, por lo que en su declaración irán acompañados del modificador de acceso public.

• Al igual que cualquier otro método, un constructor puede definir parámetros en su declaración, de tal manera que en la sentencia de instanciación de un objeto se pueda pasar argumentos al constructor de la clase.

Estos argumentos o parámetros de los métodos constructores reciben el nombre de inicializadores, pues permitirán al programador indicar, en el momento de la creación del objeto, qué valores concretos debe establecerse para los atributos del objeto.

También es posible para el programador proporcionar un constructor sin parámetros y establecer directamente en el código de dicho constructor los valores con los que inicializar las variables de instancia del objeto recién creado.

• Se requiere que toda clase tenga cuando menos un constructor. Si no se declarasen constructores para una clase, y sólo en ese caso, el compilador crearía un constructor predeterminado que no tomaría argumentos.

Dicho constructor llamaría primero al constructor sin argumentos de la superclase (clase de la cual hereda) y luego procedería a inicializar las variables de instancia con los valores por defecto. Pero ¡cuidado!, si la superclase no tuviese un constructor sin argumentos, entonces el compilador generaría un mensaje de error. Se verá con más profundidad en el tema siguiente, con la herencia.

• Un constructor puede llamarse sin argumentos solamente si no hay constructores declarados para la clase, en cuyo caso se llama al constructor por defecto, o si hay un constructor público sin argumentos. Además, como el constructor por defecto sólo se construye cuando no se declara ningún otro constructor para la clase, si una clase tiene constructores públicos, pero ninguno de ellos es un constructor sin argumentos y un programa trata de llamar a un constructor sin argumentos para inicializar un objeto de la clase, se produce un error de compilación.

En definitiva, si declaramos constructores, tenemos que usar esos obligatoriamente y no el que se crea por defecto. Este último, sólo se usa cuando el programador no ha declarado ninguno.

• Un constructor puede llamar a otros métodos de la clase; sin embargo, debe ser consciente de que las variables de instancia del objeto tal vez no se encuentren aún en un estado consistente, ya que el constructor está en el proceso de inicializar el objeto, y que utilizar variables de instancia antes de que hayan sido inicializadas

apropiadamente es un error lógico. Por lo tanto, aunque se pueda llamar a otros métodos de la clase desde un constructor, no es recomendable hacerlo.

*Observa el código de la clase Trabajador que se te ha proporcionado. ¿Puedes localizar el método creador de la clase?*

Efectivamente, es el método cuyo nombre es también Trabajador:

```
public Trabajador (String nif, String nombre) {
        // Código del método
}
```

Además, nuevamente, verás que no es el único método con ese nombre, verás que el constructor también está sobrecargado.

Para nosotros los constructores seguirán siempre la siguiente estructura, por ejemplo, si tenemos una clase Alumno:

```
public class Alumno {
        private String nombre;
        private int edad;
        private double notaMedia;

        public Alumno(String nombre, int edad, double notaMedia) {
                super(); //Se verá más adelante
                this.nombre = nombre;
                this.edad = edad;
                this.notaMedia = notaMedia;
        }

}
```

## 2. Constructores sobrecargados

Anteriormente en este tema vimos que los métodos pueden estar sobrecargados; es decir, que en la misma clase puede haber varios métodos distintos con el mismo nombre, siempre y cuando estos tengan distintas firmas (distinto número y tipo de parámetros). Además, vimos que la utilidad de la sobrecarga no era otra que la de poder denominar con el mismo nombre métodos que realizan tareas similares, aunque sobre tipos de datos distintos.

Por otra parte, si hemos dicho en el apartado anterior que los constructores no son más que unos métodos especiales, la pregunta que surge es, evidentemente, la siguiente: ¿se pueden sobrecargar los constructores? La respuesta es evidente: sí.

Los constructores sobrecargados permiten a los objetos de una clase inicializarse de distintas formas. Para ello, simplemente hay que proporcionar varias declaraciones del constructor con distintas listas de parámetros, pues al igual que sucede con los métodos normales, tratar de sobrecargar un constructor con otro que tenga exactamente la misma firma es un error de sintaxis.

*Observa los constructores del código de la clase Trabajador y verás cómo cada uno de ellos inicializa el objeto recién creado de forma distinta:*

• Uno de los métodos constructores crea el objeto dándole valores iniciales al NIF y al nombre del trabajador.

• Otro de los métodos constructores crea el objeto dándole valores iniciales al NIF, al nombre del trabajador y a la fecha de alta en la empresa.

*NOTA: El operador this se explica en el siguiente apartado*

```
// Primer constructor de la clase que inicializa el NIF y el nombre del trabajador
publicTrabajador (String nif, String nombre) {
        this.nif=nif;
    this.nombre=nombre;
}

/* Segundo constructor sobrecargado, que invoca al primero para "extenderlo", es decir, para hacer  todo lo que hacía el primero, y algo más.*/

public Trabajador (String nif, String nombre, int diaAlta, int mesAlta, int añoAlta) {
// Lo primero que hacemos es llamar al constructor anterior
    this (nif, nombre);
// realmente llama al otro constructor, de forma que actualiza el total de
// personas y le asigna al trabajador el nif y el nombre que recibe como
// parámetro.

    // A partir de aquí, el código de este constructor
        this.diaAlta=diaAlta;
        this.mesAlta=mesAlta;
```

```
            this.añoAlta=añoAlta;
}
```

## 3. Uso del operador this y del método this ( )

Ya sabemos que cuando desde un método de una clase se quiere acceder a un miembro de la clase, ya sea éste una variable de instancia o uno de sus métodos, se puede hacer directamente utilizando el nombre del miembro. Observa el método setFechaNacimiento ( ) en el código de la clase Trabajador que se te ha proporcionado:

```
public void setFechaNacimiento (int dia, int mes, int año) {

if (comprobar Fecha (dia, mes, año)) {
/* Si se trata de una fecha válida procedemos a cambiar los valores de las variables de instancia asociadas a la fecha de nacimiento*/

        diaNacimiento=dia;
        mesNacimiento=mes;
        añoNacimiento=año;
}
else {

// Si la fecha no es válida, mostramos un mensaje de error.
System.out.println("La fecha especificada no correcta. No se ha asignado");
    }
  }
```

Como puedes ver, en él se invoca al método privado comprobarFecha() de la siguiente manera:

```
comprobarFecha (dia, mes, año)
```

Observa también cómo se accede a las variables de instancia diaNacimiento, mesNacimiento y añoNacimiento para establecerles un valor:

```
diaNacimiento=dia;
mesNacimiento=mes;
```

```
añoNacimiento=año;
```

En ambos casos no es necesario especificar nada más pues el compilador sabe que se está haciendo referencia a un método o a una variable de instancia del propio objeto, respectivamente. Esto es lo que se llama una referencia implícita al objeto, pues se está haciendo referencia al propio objeto, pero sin mencionarlo expresamente dentro de la misma clase. Sin embargo, aunque no sea obligatorio, es posible reflejar esta referencia al propio objeto de forma explícita mediante la palabra clave this.

Observa ahora el método setFechaAlta() de la clase Trabajador, que accede a algunas otras variables de instancia e invoca también al mismo método privado del ejemplo anterior:

```
public void setFechaAlta (int dia, int mes, int año){

if (this.comprobarFecha(dia, mes, año)==true){

// Si se trata de una fecha válida procedemos a cambiar los valores de las variables
//de instancia asociadas a la fecha de nacimiento.
this.diaAlta=dia;
this.mesAlta=mes;
this.añoAlta=año;
}
else {
// Si la fecha no es válida, mostramos un mensaje de error.
System.out.println ("La fecha especificada no es correcta y no se ha asignado");
}
   }
```

En este caso, aunque no era necesario, se ha explicitado con la palabra clave this que los métodos y las variables de instancia, a las que se accede, pertenecen al mismo objeto que lanza la petición. Por lo tanto, debemos tener siempre presente que todo objeto puede hacer referencia a sí mismo mediante la palabra clave this, la cual hace referencia al propio objeto en el que se está ejecutando la sentencia. Esta manera explícita de hacer referencia al propio objeto que realiza el acceso a uno de sus variables de instancia o la invocación a uno de sus métodos, es preferida por muchos programadores frente a la referencia implícita, pues puede incrementar la claridad del programa en algunos contextos en donde this es opcional, como en los ejemplos anteriores.

Sin embargo, hay ciertas ocasiones donde la referencia implícita no es posible, pues produce ambigüedad, y es imprescindible usar la referencia explícita this para poder acceder a un miembro de instancia. Observa el

código del método setNombre() de la clase Trabajador (o en cualquiera de los constructores que autogenera Eclipse):

```
public void setNombre (String nombre) {
        this.nombre=nombre;
}
```

Está claro que lo que hace el método es asignar a la variable de instancia nombre del propio objeto el valor del parámetro del método, que también se llama nombre. Pero, ¿qué pasaría si eliminásemos de la sentencia la palabra clave this? Es evidente que, en ese caso, ante una sentencia nombre=nombre, no tendríamos ninguna manera de saber que lo que hay a la izquierda del signo de asignación está haciendo referencia a la variable de instancia del objeto. De hecho, el compilador interpreta que en ambos lados de la asignación se está haciendo referencia al parámetro del método.

Por lo tanto, extraemos la siguiente conclusión: para un método en el cual un parámetro o variable local tenga el mismo nombre que una variable de instancia de la clase, debe usar obligatoriamente la referencia this si desea tener acceso a la variable de instancia; de no ser así, estará haciendo referencia al parámetro o variable local del método.

Para prevenir este tipo de errores sutiles y difíciles de localizar, se recomienda siempre evitar los nombres de parámetros o variables locales en los métodos que tengan conflicto con los nombres de las variables de instancia.

Hay una cosa que deberemos tener siempre en cuenta sobre el uso de this para no cometer errores de sintaxis: desde un método de clase (aquellos que llevan modificador static) no se puede utilizar la referencia this. Como ya dijimos cuando hablamos de los métodos static, los métodos de clase se usan solamente para acceder a las variables de clase y no pueden tener acceso a los miembros de instancia. Esto es así porque, recordemos, los miembros de clase no están ligados a ningún objeto concreto de la clase, sino a la propia clase.

De hecho, los miembros de clase existen y pueden ser accedidos aun cuando no exista ningún objeto creado de dicha clase. Por lo tanto, es un error de sintaxis que un método static llame a un método de instancia de manera directa o acceda a una variable de instancia directamente, incluido a través de la referencia this, al no haber objeto de la clase al cual hacer referencia.

Es cierto, no obstante, que un método static puede llamar a un método de instancia o acceder a una variable de instancia a través de una referencia a un objeto, si dicha referencia está disponible en el método.

Aparte de la referencia this, también existe lo que se conoce como el método this( ).

Veamos cuál es su uso:

Imagina una clase con un constructor sobrecargado, sin ir más lejos, nuestra clase Trabajador. En ella:

• Uno de los métodos constructores crea el objeto dándole valores iniciales al NIF y al nombre del trabajador.

• Otro de los métodos constructores crea el objeto dándole valores iniciales al NIF, al nombre del trabajador y a la fecha de alta en la empresa.

Como vemos, el segundo de los métodos constructores hace lo mismo que el primero, y además hace alguna otra cosa más. ¿No sería estupendo no tener que repetir en el segundo de los constructores el mismo código

que ya usamos en el primero? Pues esto es posible haciendo uso del método this(). El método this()se usa para hacer referencia dentro de un constructor de una clase a otro constructor sobrecargado de la misma clase, aquél que coincida con la lista de parámetros de la llamada.

Observa el código del segundo de los constructores de la clase Trabajador. Como primera instrucción aparece una llamada al método this(). De hecho, y esto es importante, de usarse el método this(), forzosamente tiene que ser la primera línea de código del constructor.

```
// Primer constructor de la clase que inicializa el NIF y el nombre del trabajador
public Trabajador (String nif, String nombre) {

        if (this.comprobarNif(nif)) {//si lo devuelto por comprobarNif es true
                this.nif=nif;
                this.nombre=nombre;
                this.categoriaProfesional=Trabajador.categorias[0][0];
                Trabajador.objetoCreado=true;
                numTrabajadores++;
        } else{
                Trabajador.objetoCreado=false;
        }
}

/* Segundo constructor sobrecargado, que invoca al primero para "extenderlo", es decir, para hacer todo lo
que hacía el primero, y algomás.*/

public Trabajador (String nif, String nombre, int diaAlta, int mesAlta, int añoAlta) {
/* Segundo constructor sobrecargado, que invoca al primero para "extenderlo", es decir, para hacer todo lo
que hacía el primero, y algo más lo primero que hacemos es llamar al constructor anterior.*/
this (nif, nombre);
// Realmente llama al otro constructor, de forma que actualiza
// El total de personas y le asigna al trabajador el nif y el
// Nombre que recibe como parámetro.
if (Trabajador.objetoCreado){
        // A partir de aquí, el código de este constructor
```

```
        if (this.comprobarFecha(diaAlta, mesAlta, añoAlta)) {


                this.diaAlta=diaAlta;
                this.mesAlta=mesAlta;
                this.añoAlta=añoAlta;
        }else {
        // Por defecto si la fecha introducida no es correcta se la asigna el valor 0
        this.diaAlta=0;
        this.mesAlta=0;
        this.añoAlta=0;
        }
}
```

> *Estos ejemplos anteriores, son académicos, por lo cual, normalmente no se suele añadir más código en los constructores que las líneas que asignan el valor del parámetro al atributo (this.edad=edad) o alguna cosa más para la creación, es decir, no se suelen añadir operaciones, condicionales, etc.*