Paradigma de programación orientada a objetos

*Cada día que pasa **Víctor** se siente más capacitado para programar aplicaciones por sí solo y tras concluir la última, considera que está preparado para programar una aplicación sin ayuda. Pero **José** le dice que a la empresa **SI Andalucía**, lo que le interesa es mantener un método de programación y no sólo hacer programas para sus clientes. De lo que realmente se trata es de que la empresa invierta el menor tiempo posible en satisfacer a sus clientes aprovechando material elaborado con el esfuerzo ya realizado en otras aplicaciones. De este modo, además se consigue evitar la dependencia de determinadas personas. Esto significa que si por cualquier causa cualquiera de los programadores de la empresa se ve obligado a dejar un proyecto, éste puede ser retomado por otro trabajador de la empresa y terminarlo con el mínimo esfuerzo, ya que utilizan el mismo método de trabajo y emplean los mismos recursos, que en la mayoría de los casos ya están programados y forman parte de la biblioteca de objetos de programación.*

__Víctor__ no entiende muy bien a qué se refiere su compañero, y José le explica que el siguiente paso en su formación como programador, debe ser una iniciación a las técnicas de Programación Orientada a Objetos. Para iniciarle en estas técnicas __José__ le explica a __Victor__ el proceso con un __ejemplo__. Cuando queremos construir una vivienda, en un momento dado es necesario utilizar puertas o ventanas, que ya están fabricadas y que solamente hay que montar. Ante esta situación tenemos que pensar que hay muchos tipos de puertas, pero ante una de ellas, no nos cabe duda que se trata de este objeto y sabemos utilizarla sin problemas. Además todas funcionan igual y tienen la misma utilidad. Algo así ocurre con la programación orientada a objetos, definimos objetos (clases) a los que les asignamos unos atributos (características con las que lo identificamos respecto a otros objetos) y sobre los que podemos aplicar diferentes operaciones o acciones (métodos) sin necesidad de volver a aprenderlos. __José__ concluye diciendo que la Programación Orientada a Objetos es un mecanismo con el que pretendemos imitar el modelo natural de actuar ante la necesidad de construir algo útil.

Ya llevamos recorrido gran parte de este largo camino que hemos emprendido con el objetivo de aprender los secretos que encierra el mundo de la programación de ordenadores. A lo largo del trayecto hemos asimilado la metodología que debemos seguir para el desarrollo de programas, así como las principales técnicas de programación. Sin embargo, **cuando se desarrolla un programa informático**, al igual que en el resto de ámbitos de la vida, **siempre se busca que se trate de un producto de calidad**.

Ahora bien, ¿sabemos desarrollar software de calidad? ¿Qué debemos hacer para desarrollar programas informáticos de calidad? Y lo que es más importante, ¿qué significa el término "calidad" cuando hablamos de software? Comenzamos en esta unidad una nueva etapa de nuestro trayecto, etapa en la que trataremos de dar respuesta a las preguntas anteriormente formuladas, dando pasos en pos de la consecución del objetivo de desarrollar programas informáticos de calidad mediante la introducción de las **técnicas de la programación orientada a objetos para lograr mejoras significativas en la calidad de los productos software**, si bien teniendo siempre en mente las palabras del reconocido ingeniero de software Fred Brooks, en las que afirmaba que "la construcción de software siempre será una tarea difícil, pues *no hay soluciones mágicas* (lo que en inglés se llama bala de plata) para ello".

PARA SABER MÁS...
Si quieres leer íntegro, en inglés, el artículo en el que el ingeniero de software Fred Brooks afirmaba que no hay, ni habrá, soluciones mágicas (bala de plata) en lo concerniente a la construcción de software, consulta el siguiente enlace.
***No silver bullet:* *Essence and accidents of Software Engineering* [Versión en caché]**

Antes de comenzar a estudiar las técnicas que nos permitirán alcanzar el objetivo marcado, deberíamos clarificar cuál es dicho objetivo, analizando meticulosamente **qué es lo que se considera un producto software de calidad**. Pongámonos en la piel de un usuario de un programa. Si alguien nos preguntase qué aspectos generales nos gustaría que tuviese un programa informático que adquirimos, ¿qué contestaríamos?

Probablemente, cualquier usuario experimentado de ordenadores contestaría que le gustaría que el programa reuniese, en uno u otro orden de importancia, las características expuestas a continuación.

Empezamos con los **factores de calidad externos**. Todas las características que exponemos en los siguientes subaparatados son características deseables de los programas que se denominan factores de calidad externos, pues **pueden ser detectados por los usuarios.**

Una de las cosas principales que como usuarios esperamos de un producto software es que haga lo que se supone que debe hacer; es decir, si es un programa que hace las nóminas de los empleados de una empresa, que haga las nóminas de los empleados de la empresa. Si un programa no hace lo que se supone que debe hacer, poco importa que sea muy rápido o que tenga una deslumbrante interfaz de usuario, ¿verdad? **Definimos corrección como la capacidad de los productos software para realizar con exactitud sus tareas, tal y como se definen en las especificaciones**, y es una de las características que debe reunir un programa para que se considere un producto de calidad.


Corrección


Robustez

Si un programa es correcto, ante una situación contemplada en su especificación, hará lo que esté previsto en dicha especificación. Pero ¿qué sucede en un programa cuando se da una situación no contemplada en su especificación? Por ejemplo, el programa espera que se introduzca por teclado un número y se introduce una letra. Generalmente, este tipo de situación excepcional y no contemplada en la especificación suele producir eventos catastróficos en la ejecución del programa y una terminación abrupta del mismo. Un programa que sea capaz de detectar y reaccionar a este tipo de situaciones no previstas, por ejemplo, produciendo los mensajes de error apropiados y terminando su ejecución de manera limpia, se dice que es un programa robusto. **Definimos robustez como la capacidad de los sistemas software de reaccionar apropiadamente ante condiciones excepcionales**, y es otra de las características que debe reunir un programa para que se considere un producto de calidad.

Un programa que realiza lo que se establece en su especificación es un programa correcto. Pero ¿qué sucede si esa especificación cambia o se ve ampliada? Evidentemente, para que el programa siga siendo correcto, tendrá que ser también modificado para que recoja los nuevos elementos especificados. ¿Crees que esto suele ocurrir frecuentemente en el mundo del desarrollo del software? Pues el hecho es que sí, que es muy frecuente que una vez desarrollado y entregado un programa, el cliente se dé cuenta de que necesita añadir algo al mismo o que circunstancias de la vida lleven a tener

que modificar o ampliar un programa.

Imagina, por **ejemplo**, un software de gestión en el que la aprobación de una ley sobre retenciones fiscales puede invalidar de pronto los supuestos sobre los que el sistema se apoyaba para la realización de las nóminas de los trabajadores. En un programa pequeño, realizar estos cambios no suele ser tarea difícil; sin embargo, en proyectos grandes de software, esto que recibe el nombre de **mantenimiento del software**, puede convertirse en una tarea casi inabordable o, en el mejor de los casos, extremadamente costosa en tiempo y dinero. Es por tanto imprescindible que nuestros programas estén construidos de tal manera que sean fácilmente extensibles; es decir, que se les pueda añadir nueva funcionalidad o se les pueda realizar modificaciones de manera sencilla y poco traumática. **Definimos extensibilidad como la capacidad de los sistemas software de ser adaptados fácilmente a los cambios de las especificaciones**, y es otra de las características que debe reunir un programa para que se considere un producto de calidad.


Reutilización

El desarrollo de software es una tarea muy costosa y al programador, como a cualquier otra persona, no le suele gustar realizar las mismas cosas una y otra vez. Si inventaras una herramienta para apretar tornillos, y posteriormente en otro problema necesitaras apretar tornillos, ¿volverías a inventarla o usarías la que ya hiciste? La respuesta es evidente, ¿verdad? Sin embargo, para asegurarte de que luego vas a poder reutilizar la herramienta en otras situaciones distintas, a la hora de inventar la primera herramienta para apretar tornillos, deberías hacerla lo más genérica posible. Los sistemas software generalmente siguen patrones similares, por lo que realizar programas que puedan ser reutilizados de uno u otro modo es algo importante y que nos va a permitir ahorrar esfuerzos, los cuales podemos dedicar a otros aspectos del desarrollo. La idea es la de evitar reinventar soluciones para problemas ya resueltos. ¡Aprovechemos el trabajo que ya hicimos en su día! **Definimos reutilización como la capacidad de los elementos del software de servir para la construcción de muchas aplicaciones diferentes**, y es otra de las características que debe reunir un programa para que se considere un producto de calidad.


Compatibilidad

El software no es un elemento aislado, sino que es algo que tiene que **relacionarse** con otros sistemas externos a él. En muchas ocasiones los programas tienen dificultades para interactuar con otros sistemas porque se han hecho en ellos suposiciones contradictorias o erróneas sobre el resto del mundo. Un ejemplo es la amplia variedad de formatos de archivos soportados por muchos sistemas operativos. Un programa puede usar directamente como entrada los resultados de otro sólo si los formatos de archivos empleados por ambos programas son compatibles. La clave recae en la homogeneidad del diseño, en acordar convenciones estándares para la comunicación entre programas, de tal manera que éstos sean compatibles. **Definimos compatibilidad como la capacidad de combinar unos elementos software con otros**, y es otra de las características que debe reunir un programa para que se considere un producto de calidad.


Eficiencia

Otra cosa que un usuario le suele pedir a un programa informático es que éste le ofrezca un buen rendimiento, que sea un producto eficiente. Generalmente, cuando hablamos de un programa informático, esta eficiencia viene expresada en términos de ahorro de tiempo, memoria y, en general, de recursos. **Definimos eficiencia como la capacidad de un sistema software de requerir la menor cantidad posible de recursos hardware**, y es otra de las características que debe reunir un programa para que se considere un producto de calidad.

Cuando adquirimos un producto software en la tienda, lo que queremos es que dicho producto funcione en nuestro ordenador. Esto, que puede resultar evidente, implica más de lo que pudiese parecer en un principio. Somos muy dados a pensar que todos los ordenadores son como el nuestro, pues nuestro ordenador es el mejor, ¿no? Sin embargo, no todo el monte es orégano; no todo el mundo usa el mismo sistema operativo, unos usan Windows, otros usan Linux, ni todo el mundo usa la misma plataforma hardware. De hecho, cuando compramos un programa, generalmente, lo

compramos para una plataforma concreta, aquélla en la que va a ser ejecutado. Seguro que te suena la pegatina de "Requiere Windows XP" que tienen la mayoría de los programas que encuentras en las tiendas. Sería estupendo que un mismo programa pudiese funcionar en distintas plataformas, ¿verdad?; sería estupendo que el software fuese portable. **Definimos portabilidad como la capacidad que tienen algunos productos software para funcionar en diferentes entornos hardware y software**, y es otra de las características que debe reunir un programa para que se considere un producto de calidad.

¿Qué es lo primero que haces cuando adquieres un nuevo programa informático? ¿Leerte el manual de instrucciones o ponerte a toquetearlo a ciegas? Seguramente eres como yo y perteneces al segundo grupo. En esos casos, seguro que agradeces que el programa sea fácil de usar. **Definimos facilidad de uso como la capacidad de los programas para que personas con diferentes formaciones y aptitudes puedan aprender a usarlos y explotarlos**, y es ésta una cualidad que es indicativa de la calidad del producto.

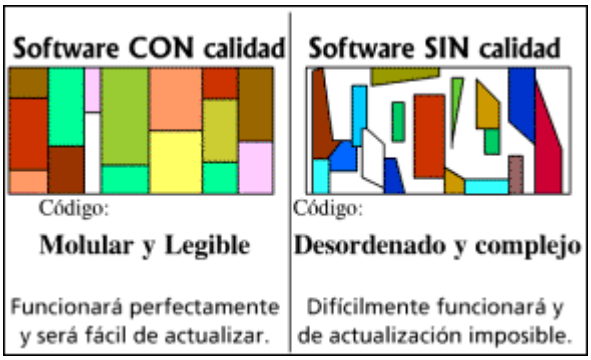
Por último, otra característica que siempre vamos mirando los usuarios, si no es la primera, es el precio de los productos, aunque sea éste un factor que no tiene por qué tener relación alguna con la calidad del producto.

Calidad del software: Factores internos

Ya mencionamos al principio del apartado anterior que todas esas características, expuestas en los subapartados anteriores, forman parte de lo que se denomina **factores de calidad externos**, pues pueden ser detectados por los usuarios.

Pero hay otros factores que determinan la calidad del software, los llamados **factores internos**, **perceptibles sólo por profesionales de la informática que tienen acceso al código fuente de los programas**.

Es cierto que en última instancia lo que importan son los factores externos, sin embargo, no es menos cierto que **los factores internos de calidad son el medio para conseguir esa calidad externa**.



En los siguientes subapartados destacamos los principales factores internos de calidad.

En unidades anteriores ya abordamos algunos de los aspectos relativos a la programación modular y, básicamente, aprendimos que un programa modular es aquél que está formado por un conjunto de módulos, donde un módulo es la unidad básica de descomposición de un sistema software. Sólo nos queda una pregunta pendiente, ¿cómo contribuye la programación modular a la producción de software de calidad? Como vimos, **seguir una metodología modular en el desarrollo de programas informáticos**

ayuda a producir sistemas software a partir de elementos autónomos interconectados por una estructura simple y coherente, y esto **favorece la consecución de los dos principales factores de calidad externos del software: la extensibilidad y la reutilización**. Es evidente que una correcta división en módulos de un programa, siguiendo las directrices que vimos en la unidad 6 en cuanto al bajo acoplamiento y alta cohesión de los mismos, hará más fácil la introducción de cambios, pues estos estarán localizados en módulos concretos, y también favorecerá la reutilización de ciertas partes del software, aquellas que me proporcionen ciertos módulos del programa.

Para conseguir una calidad interna en nuestros programas también es importante que éstos sean legibles. Básicamente, esto comprende que sus líneas de código tengan buenos comentarios que permitan entender rápidamente qué hace el programa y, por otra parte, que estén correctamente tabuladas para su rápida comprensión. Además, y ligando este factor de calidad con el anterior, es importante que la documentación interna de un módulo forme parte del propio módulo y pueda ser utilizada para comprender el funcionamiento y lo que ofrece dicho módulo.

La siguiente simulación te guía en el proceso para generar de la documentación del programa aprovechando un tipo de comentarios que el programador incorpora en el código del mismo.

DEMO: Visualiza cómo generar la documentación de la aplicación de forma automática con JAVADOC

Autoevaluación

La programación orientada a objetos

Una vez descrito qué entendemos por **calidad** en el mundo del software, es hora de empezar a estudiar cómo podemos alcanzar dicha calidad en nuestros programas, es hora de introducirnos en el paradigma de programación orientada a objetos.

Un paradigma de programación representa un enfoque particular o filosofía para la construcción de software. Cada paradigma ofrece un estilo de acercarse a los problemas y deriva en una manera particular de abordarlos y resolverlos. Paradigmas de programación hay muchos, si bien son dos los más importantes:

- El paradigma **imperativo**, que da lugar a la programación estructurada o basada en procedimientos.
- El paradigma **orientado a objetos**, que da lugar a la programación orientada a objetos.

Si recuerdas, en la unidad 5 vimos:

- Que **la programación estructurada o basada en procedimientos está orientada a acciones**. Los programadores agrupan acciones dentro de funciones y procedimientos, cada uno de los cuales realiza alguna tarea. Estas funciones y procedimientos son posteriormente agrupados para formar los programas.
- Que **en la programación estructurada** los datos son importantes, pero la idea es que **los datos existen primordialmente para apoyar las acciones que realizan las funciones o procedimientos**.
- Que **la programación estructurada fija su atención en las estructuras de control de flujo que se pueden usar**, imponiendo algunas limitaciones y restricciones para evitar generar algoritmos difíciles de seguir y entender y, por tanto, difíciles y costosos de mantener.

Las principales dificultades y problemas surgidos en las técnicas estructuradas

parten de su propio enfoque:

- Las metodologías estructuradas hacen una **división de procesos y datos**, donde los procesos son los que guían la lógica del programa.
- Además, **las dependencias existentes entre datos y procesos quedan reflejadas en el programa**, lo que implica que cualquier cambio en el proceso o en los datos suponga cambios importantes en el propio programa. Esto hace que los **programas** sean **poco extensibles**.
- Pero el problema principal de la programación estructurada o basada en procedimientos es que **las unidades de programación no reflejan de manera fácil y efectiva a las entidades del mundo real**, lo cual deriva en que estas **unidades no** sean particularmente **reutilizables**. Con gran frecuencia los programadores deben comenzar "de nuevo" cada nuevo proyecto y escribir código similar "desde cero". Esto significa un gasto de tiempo y de dinero, ya que la gente tiene que "reinventar las cosas" repetidamente.

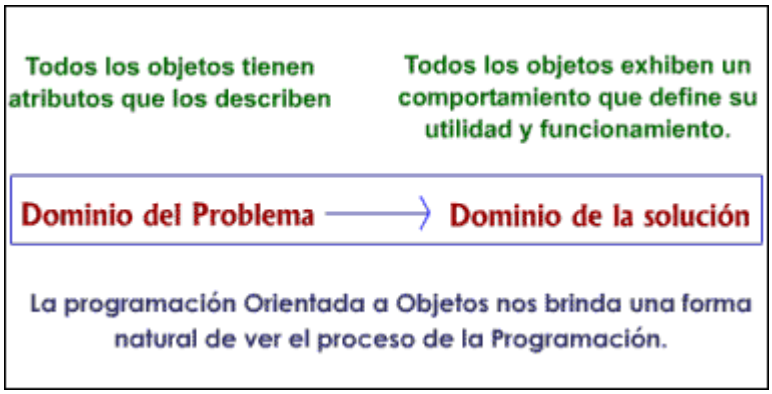
Y ahí es precisamente donde interviene la programación orientada a objetos, la cual se basa en el siguiente planteamiento: **si percibimos el mundo real como un mundo formado por objetos, entonces considerar los problemas en términos de objetos nos resultará más sencillo**. Además, **mediante la tecnología de objetos, las entidades de software creadas**, que veremos que reciben el nombre de clases, si se diseñan apropiadamente, **tienden a ser mucho más reutilizables en proyectos futuros**.

No obstante, algunas empresas indican que la reutilización de software no es, de hecho, el principal beneficio que obtienen de la programación orientada a objetos. Más bien mencionan que **la programación orientada a objetos tiende a producir software que es más comprensible, mejor organizado y fácil de mantener, modificar y corregir**. Esto puede ser importante debido a que se estima que el 80% de los costos de software no están asociados con los esfuerzos originales para el desarrollo de software, sino que están asociados con la continua evolución y mantenimiento de ese software durante su vida útil.

En definitiva, **la programación orientada a objetos nos proporciona un conjunto de técnicas de elaboración de programas informáticos destinadas a obtener calidad interna, siempre como medio para obtener calidad externa en los mismos**. Principalmente, **software reutilizable y extensible**.

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**. 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. **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**. Así, por ejemplo, el objeto pelota rueda, rebota, se infla, se desinfla; el objeto bebé duerme, come, llora y hace de vientre; 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. 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.

¿Recuerdas el símil que usamos al hablar de la reutilización como factor de calidad? 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. **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 puede reutilizarse 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.

Autoevaluación

Clases

*Parece que **Víctor** se ha convencido de que necesita adaptarse a este método de programación para incorporarse al equipo de programación de **SI Andalucía**, así que decide prepararse bien y pedir ayuda a **Carmen**, que siempre sabe cómo explicarle las cosas de manera que él lo entienda.*

__Carmen__ después de dos horas con __Víctor__, ya no sabe cómo hacerle entender que las clases sólo son un trozo de código que define un objeto, por llamarle de alguna manera. Y le explica que el modo que tenemos en Java de definir objetos, es mediante clases. Para crear objetos de una clase determinada utilizamos un mecanismo (código en Java) que llamamos __Constructor__. Con un constructor creamos __instancias__ ("individuos") de un objeto. Le pone el ejemplo de una supuesta fábrica de robots, todos iguales. Hay un elemento constructor que permite crear robots. Cada uno de los que crea será idéntico al resto, todos ellos podrán hacer las mismas cosas y tendrán el mismo comportamiento. Para crear otro tipo de robot, será necesario definir un nuevo Constructor.

En la aplicación que está trabajando actualmente __Carmen__ para la gestión de personal de una empresa, ha definido una clase Trabajador, con una serie de atributos que permiten definir a todos los trabajadores, los cuales tendrán un comportamiento en la empresa que se describe con los métodos de dicha clase.

Con lo visto hasta ahora, contesta a la siguiente pregunta:

¿cuál es el concepto central de la tecnología de objetos?

Seguramente, y sin dudarlo, habrás respondido que son los objetos, y no andas desencaminado, aunque no sea del todo cierto. Es cierto que los objetos son la piedra angular alrededor de la cual se construye este nuevo paradigma de programación que estamos estudiando. De hecho, fíjate si son importantes, que hasta dan nombre al propio paradigma. Sin embargo, y aunque no lo creas, **hay algo más importante que los objetos: las clases**. Veamos entonces qué es una clase.

Concepto de clase (I)

¿Has jugado alguna vez en la playa a hacer castillos de arena? Para jugar sólo necesitamos un cubo, que 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.

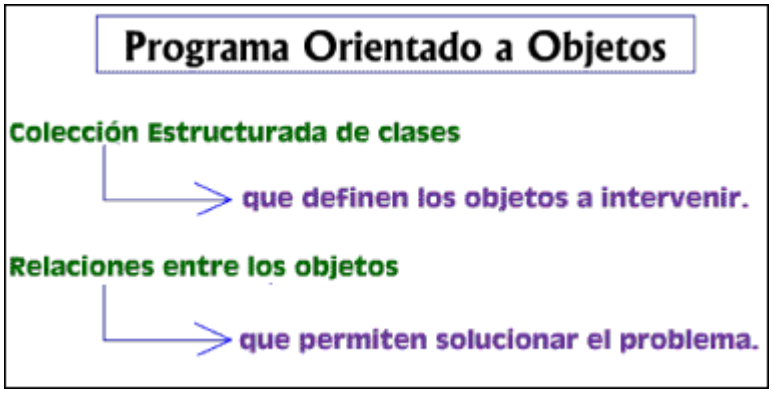
Una clase es una plantilla que define la forma de un tipo de objetos. En esta plantilla se especifican **los atributos y el comportamiento** con los que van a contar los objetos que se construyan a partir de dicha plantilla. Por tanto, podemos afirmar que **una clase describe objetos que van a tener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

Si has entendido bien lo que son las clases y la diferencia y la relación entre éstas y los objetos, entenderás a la perfección la afirmación anterior de que las clases son más importantes que los objetos, pues definen cómo van a ser éstos. Entonces, 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**. 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.

Concepto de clase (II)

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. Así mismo, de igual manera que declaramos una variable de un cierto tipo de datos, podemos instanciar objetos de una cierta clase. **Sin embargo, el concepto de clase va más allá del simple concepto de tipo de dato**:

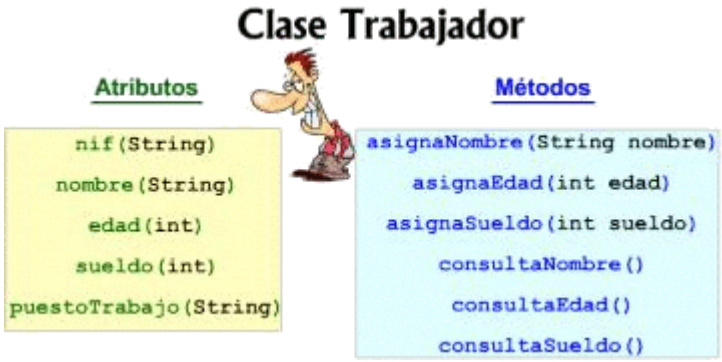
- Así, en la **programación tradicional,** el tipo de una variable indica qué cantidad de memoria debe reservarse para almacenar el contenido de la variable, es decir, su valor.
- Además, en la programación tradicional, hay una serie de funciones o procedimientos que actúan sobre las distintas variables definidas en su ámbito usando y/o modificando sus valores.
- Sin embargo, 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**. Son las características y la estructura de almacenamiento que tendrán los objetos de la clase.
- **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.

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.

Autoevaluación

Relaciones entre clases

El error descrito en el apartado anterior de modelar conjuntamente en la

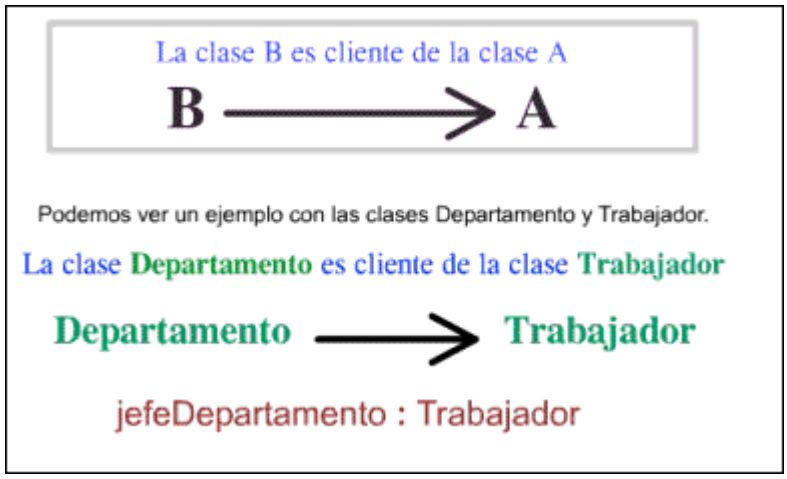
misma clase el coche y la radio se produce por una interpretación errónea de lo que no es más que una relación entre clases distintas: "un coche tiene una radio". Y es que **las clases, aunque deben presentar una alta cohesión y un bajo acoplamiento, tienen ciertas relaciones entre ellas, de forma que un programa bien estructurado contará con clases bien diseñadas que se relacionen entre sí**.

Pero, ¿qué relaciones puede haber entre dos clases? Básicamente, **las relaciones entre clases se reducen a dos tipos**:

(A veces nombrada **Composición**) Una clase B es cliente de una clase A si B contiene una declaración en la que se establezca que cierta entidad de B, ya sea ésta un atributo, parámetro o variable local de alguno de sus métodos, es de tipo A. Éste es el caso más común de relación entre clases, pues es imprescindible para que los objetos puedan interactuar entre sí. De hecho, ésta es precisamente la relación que hay entre el coche y la radio del ejemplo anterior. Un coche se compone de una serie de elementos, y uno de ellos es la radio. En definitiva, habrá que pensar en una relación de clientela o composición cuando la relación sea del tipo "tiene un". Es decir, **un coche tiene un equipo de radio.**

Pero veamos un ejemplo de clientela con la clase "Trabajador" que estamos modelando. Imagina que necesitásemos modelar también una clase "Departamento", para poder representar a los distintos departamentos de los que está compuesta la empresa. Supongamos que dicha clase, entre otros atributos que la caracterizan, tenga uno que sirva para reflejar qué trabajador de la empresa es el jefe del departamento; es decir, la relación "un departamento tiene un trabajador que es jefe del departamento". Entonces podremos afirmar que la clase "Departamento" será cliente de la clase "Trabajador", al tener un atributo, llamado "jefeDepartamento" que servirá para almacenar un objeto de tipo "Trabajador".

Este tipo de relaciones, las de clientela, se representan gráficamente como muestra la siguiente imagen:

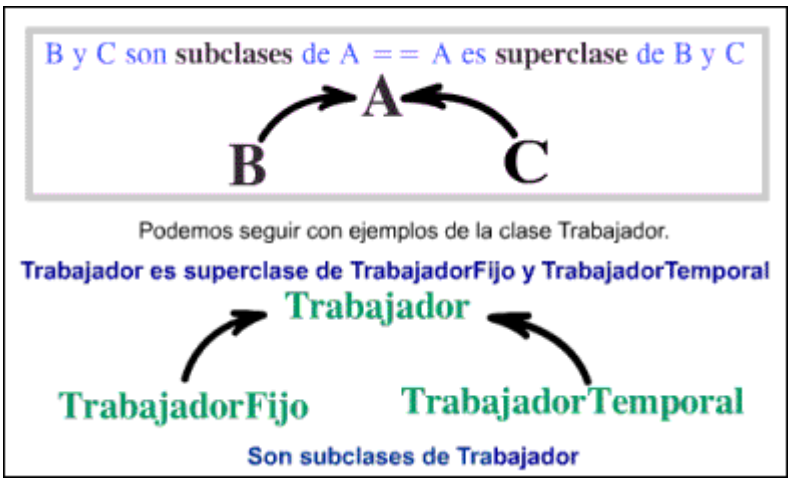


Una clase B hereda de una clase A cuando incorpora la estructura (atributos) y el comportamiento (métodos) de la clase A, pero puede incluir algunas adaptaciones, como añadir nuevos atributos o nuevos métodos. **Se dice entonces que la clase B es una versión especializada de la clase A, o más comúnmente, que la clase B es una subclase de la clase A o que deriva de la clase A y, por otra parte, se dice que la clase A es superclase de las clases B y C**.

Imagina la clase "Trabajador" que estamos modelando: trabajadores hay de varios tipos, por ejemplo, trabajadores fijos en la empresa y trabajadores con contrato temporal o trabajadores temporales. Aunque ambos tienen características comunes, cada tipo de trabajador tiene sus características propias que los hace diferentes. Así, por ejemplo, todos los trabajadores van a tener un NIF y un nombre, pero sólo los trabajadores fijos van a tener antigüedad en la empresa, mientras que sólo los trabajadores temporales van a tener una fecha de finalización del contrato, ya que los fijos tienen un contrato indefinido.

Esto podría ser modelado como una jerarquía de tres clases haciendo uso de la herencia, donde la clase madre o superclase sería la clase "Trabajador", que definiría la estructura y el comportamiento común a todos los tipos de trabajadores, y donde dicha clase madre tendría dos subclases, "TrabajadorFijo" y "TrabajadorTemporal", que tendrían las características de la clase "Trabajador" pero, además, cada una de ellas definiría sus características especiales que las diferencia del resto de tipos de trabajadores.

No te preocupes si ahora mismo esto de la herencia te resulta un poco confuso, pues volveremos a ello en el siguiente tema, donde estudiaremos en profundidad esta relación tan especial entre clases, analizando todas sus implicaciones y beneficios. Por el momento es suficiente con conocer esta definición de herencia y cómo este tipo de relación se representa gráficamente, tal y como puedes ver en la siguiente imagen:



A diferencia de la clientela, la herencia es la relación adecuada cuando la relación es de tipo "es un". Por ejemplo, **un TrabajadorFijo es un Trabajador.**

Definición de clases en Java

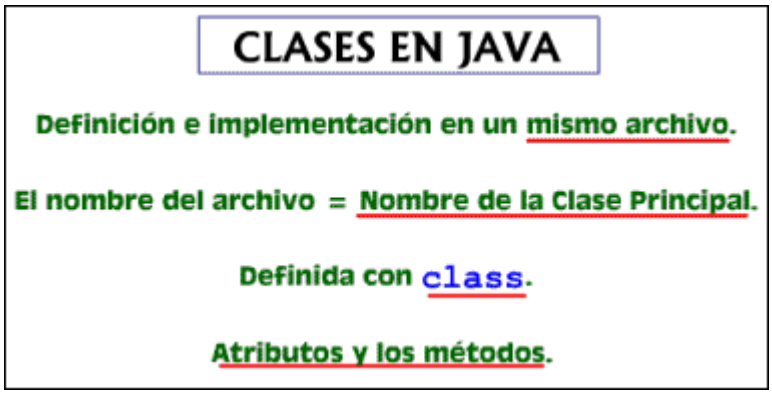
*Una vez que **Víctor** sabe qué son y para que se utilizan las clases, **Carmen** le explica que es preciso entender cómo utilizarlas y cuál es el código en Java que le va a permitir definirlas correctamente. Para ello es necesario distinguir claramente la cabecera y el cuerpo de la clase, los atributos que la caracterizan y los métodos que determinan su comportamiento.*

***Víctor** comienza a entender el proceso al ver los ejemplos de la clase Trabajador que **Carmen** está utilizando en la aplicación de Gestión de Personal. En principio no le parece muy complicado identificar la cabecera y el cuerpo de la clase, además tampoco son muchas las palabras clave utilizadas, de modo que lo que en principio le parecía tan complejo, va tomando forma y poco a poco comprende este modo de trabajo.*

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. Esto, claro está, desde el punto de vista del lenguaje de programación Java, pues no podemos afirmar que fuesen clases en sí mismas después de lo que hemos descrito como clase en este tema. Asimismo, también en temas anteriores, hemos visto que Java nos ofrece muchas clases ya definidas, las cuales hemos utilizado en nuestros propios programas y nos han facilitado la resolución de los problemas planteados. 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.

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.
- El archivo de una clase Java debe tener el mismo nombre que la clase principal que contenga 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.



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 del siguiente. 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.

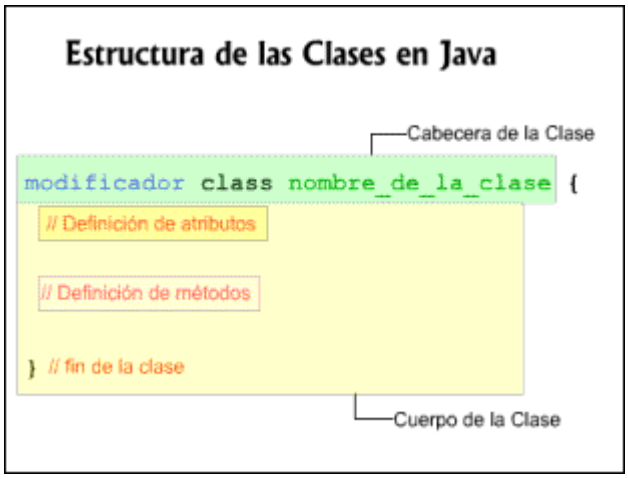
DEMO: Mira las distintas partes de una clase

Autoevaluación

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

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.



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

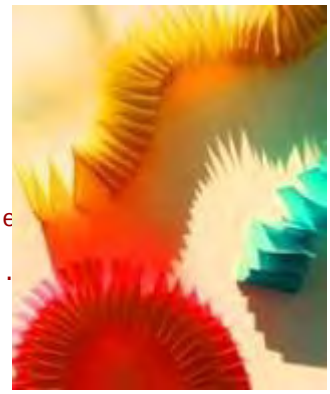
- **La palabra clave `public`**. Indica un nivel de acceso a la clase de tipo público. **Cuando una clase ha sido definida como pública, entonces cualquier clase puede hacer uso de ella o relacionarse con ella**, bien sea mediante clientela, es decir, definiendo variables u objetos que sean de esa clase, o por herencia, tal y como veremos en el tema siguiente. 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**. Cuando no aparece la palabra clave **`public`**, entonces se dice que la clase es no pública y su nivel de **acceso es de tipo package (de tipo 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. Por último, 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.
- **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**.
- **La palabra clave `abstract`**. Como veremos en profundidad en el tema siguiente, **hay algunas clases que pueden tener algún método que esté declarado pero para el que no se dé su implementación, 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**. Reflexiona un momento sobre la siguiente pregunta: ¿tiene sentido una clase que sea al mismo tiempo final y abstracta? Evidentemente, no, pues una clase abstracta no tiene sentido sin otras clases que hereden de ella y que aporten la implementación de los métodos que ésta dejó pendientes. Por su parte, las clases definidas como finales no pueden tener clases que hereden de ellas. Por lo tanto, **los conceptos de clase final y clase abstracta son excluyentes y las palabras clave final y abstract no pueden aparecer juntas como modificadores de una clase**.

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  [exte

[implements Interface_1 [, Interface_2] [, Interface_3] ...
```

Con la siguiente simulación puedes ver las posibles formas de definir clases, según lo que se indica en la notación anterior:

DEMO: Mira como se pueden definir las clases

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": 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, éste es el aspecto base de cada atributo de la clase:

```
[private | protected | public] [static] [final] [transient] [volatile]

          tipo_atributo Nombre_atributo;
```

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`** y **`volatile`** 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:

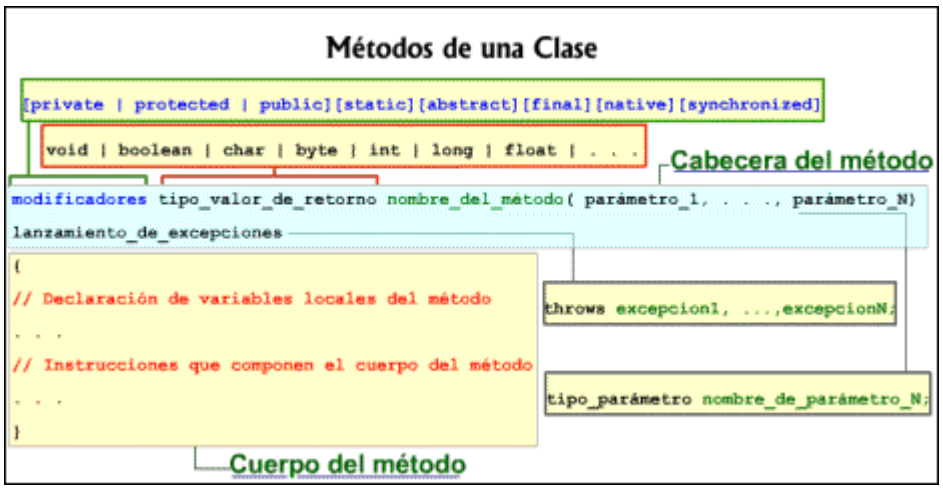
```
modificador_de_atributo tipo_atributo Nombre_atributo1, . . . ,        Nombre_atribu
```

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

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:

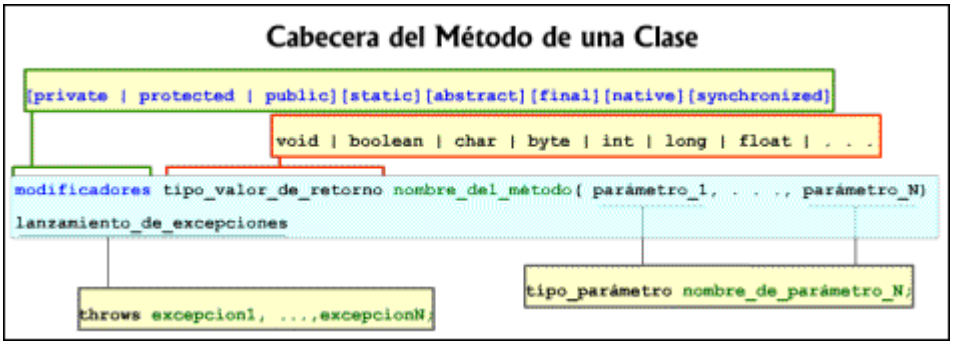


La **cabecera del método** está formada básicamente por:

- **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`** y **`synchronized`** 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, respectivamente.



- **Tipo de datos del valor devuelto por el método**. Los métodos en Java pueden devolver un valor 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_parámetroN   nombre_parámetroN )
```

¡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 juntar la 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:

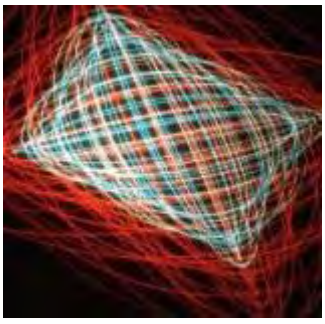
```
(tipo_parámetro  nombre_parámetro1,nombre_parametro2,..., tipo_parámetroN  nombre_pa
```

Como ya mencionamos en una unidad anterior cuando introdujimos 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.

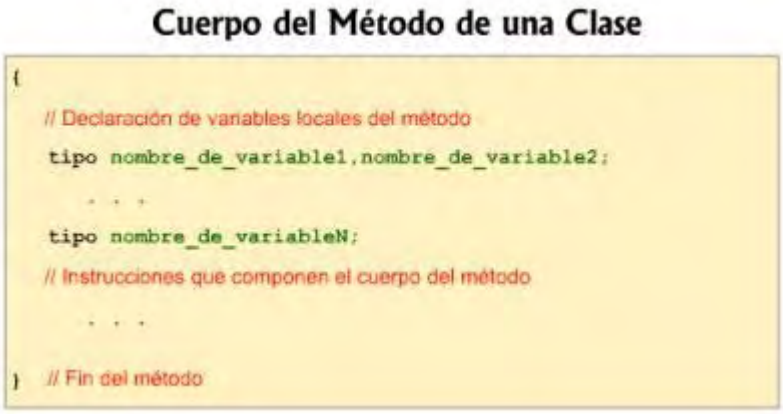
Así mismo, debemos recordar lo que ya comentamos en unidades anteriores, y es que estos parámetros son sólo de entrada o, dicho de otro modo, se pasan por valor. Es decir, no se devuelven al terminar el método, de tal manera que actúan casi como si fuesen variables locales dentro del método. De hecho, 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, donde recordemos que una excepción es un error en tiempo de ejecución que puede ser capturado y tratado para evitar que el programa acabe de forma abrupta. Tal y como vimos en la unidad 12, 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
```



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 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.

En resumen, la definición de una clase tiene el siguiente aspecto genérico:

DEMO: Visualiza un resumen de la estructura de una clase

Esto lo puedes observar claramente en el código Java de la clase Trabajador que se incluye a continuación y que hemos presentado ya en la unidad y que iremos construyendo a lo largo de la misma. De momento, échale un vistazo al código de la clase Trabajador e intenta señalar tanto la cabecera como el cuerpo de la clase, así como sus atributos y métodos, identificando los modificadores y la sintaxis de declaración de cada uno de ellos.

A partir de este momento, conforme vayamos construyendo la clase, iremos haciendo referencias a partes de este código. Una vez más, te recomendamos que leas con atención el código del ejemplo, junto a los comentarios explicativos que en él se incluyen.

Descarga el archivo Trabajador.java

Al igual que en unidades anteriores, te proporcionamos a continuación un enlace a la carpeta que contiene todo el proyecto, para que puedas seleccionar en el entorno NetBeans la opción abrir proyecto, y tener directamente cargadas todas las clases del ejemplo, listas para ejecutarlas sin nada más.

Descarga el proyecto Trabajador (Carpeta completa)

PARA SABER MÁS...
En este enlace encontrarás una buena aproximación a las clases de Java
Las Clases en Java [Versión en caché]

Introducción a la modularidad, el encapsulamiento y la ocultación de la implementación

*Llegados a este punto, **Víctor** necesita conocer algunos conceptos*

*básicos en la Programación Orientada a Objetos, con el fin de seguir el método de trabajo de la empresa y facilitar la reutilización del software. **Carmen** conoce bien la metodología de programación empleada en **SI Andalucía**, porque aunque se incorporó a la misma al tiempo que **Víctor**, ella contaba con una base de programación que adquirió durante sus estudios de Ciclo Formativo de Grado Superior, rápidamente se ha integrado en la empresa.*
***Carmen** piensa que para que su compañero entienda estos conceptos, lo mejor va a ser demostrarle con la aplicación de Gestión de Personal y la clase Trabajador, las ventajas que supone el uso de estas técnicas. En primer lugar debe comprender el concepto de Interfaz de la clase y la importancia de su utilización. Después debe acceder a las variables privadas de una clase y finalmente a utilizar Paquetes como parte de un proyecto.*

Llegados a este punto, seguro que tenemos muy clara una cosa: una clase es una plantilla que define la forma de un tipo de objetos, aquéllos que son instancias de dicha clase y, por lo tanto, define qué atributos y comportamiento van a tener los mismos.

Cuando comenzamos a desarrollar el programa para nuestra empresa identificamos la necesidad de manipular información sobre los trabajadores de la misma y analizamos qué atributos definen a un trabajador en el mundo real de nuestra empresa. Por otra parte, como sabemos que la programación orientada a objetos establece que todo lo que tenga que ver con la misma entidad conceptual debe de estar junto, encerramos dichos atributos o datos dentro de una clase de nuestro programa, llamada "Trabajador", conjuntamente con todas las operaciones que actúen sobre esos datos. De esta manera, una vez definida la clase, podemos crear tantos objetos Trabajador como necesitemos.

Como programadores, lo que estamos haciendo al definir una clase, no es más que encerrar o encapsular en una misma entidad conceptual o semántica, la clase, todos los datos que describen a una entidad genérica del mundo real, junto con todas las operaciones que se pueden realizar sobre esos datos. Pero para que el encapsulamiento sea completo, éste no tiene que ser sólo semántico, sino que debería quedar reflejado también a nivel sintáctico en el propio programa, en su código. Es decir, el lenguaje debe permitir, e incluso obligar, a que cada unidad semántica o clase esté aislada en un único módulo sintáctico del programa.

DEMO: Mira cómo se pasa de una entidad real a un módulo o fichero del programa

Como vimos en el apartado anterior, en el lenguaje Java:

- Una clase, ya sea ésta pública o no, está definida en un único archivo o módulo del programa.
- En un mismo archivo o módulo del programa 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.

Esta **propiedad de encerrar en una misma entidad sintáctica o módulo la totalidad de una entidad semántica, clase o tipo de dato**, recibe el nombre de **encapsulamiento** y es una de las características fundamentales de la programación orientada a objetos. Además, **al estar cada clase en un módulo del programa y al contener cada módulo del programa una única clase, estamos consiguiendo la descomposición modular o modularidad** que tanto alabamos en la unidad temática 6 y que señalamos en este mismo tema como uno de los factores de calidad internos del software que nos van a ayudar a

conseguir la calidad externa en nuestros programas.

Con esta filosofía de programación de identificar clase y módulo, aparte de poder disfrutar de todas las ventajas que tiene la programación modular, tal y como vimos en la unidad temática 6, conseguimos soporte para poder hacer uso de la que probablemente sea la **piedra filosofal de la programación orientada a objetos: la ocultación de la implementación** o programación orientada a la interfaz. Veamos detenidamente en qué consiste.

Ocultación de la implementación

¿Sabes conducir? Si es así, para poder hacerlo, ¿necesitas conocer qué sucede exactamente en las entrañas del coche cuando pisas el freno o cuando giras el volante? ¿Verdad que no? Simplemente te vale con saber cómo se pisa el freno y qué efectos tiene el pisar el freno: se reduce la velocidad. De igual manera, lo único que interesa al conductor es saber cómo se gira el volante y qué efectos tiene el girar el volante: se cambia la dirección en la que va el coche. Podemos decir que lo imprescindible para poder conducir y manejar el coche es conocer su cuadro de mandos y para qué sirve cada uno de los pedales, botones, palancas y resto de dispositivos del mismo. Es decir, conocer su **interfaz**, **aquello que el usuario ve externamente** del coche **y que es aquello con lo que interactúa** para poder conducirlo. Por el contrario, poco interés tiene para el conductor del coche su **implementación**, o lo que es lo mismo, **cómo funciona internamente éste**, con todo ese conjunto de válvulas, tornillos, bujías, motores, líquidos, dispositivos eléctricos, etcétera, o **cómo dichos dispositivos internos colaboran e interactúan para llevar a cabo cada una de las acciones asociadas a los distintos artefactos de la interfaz**

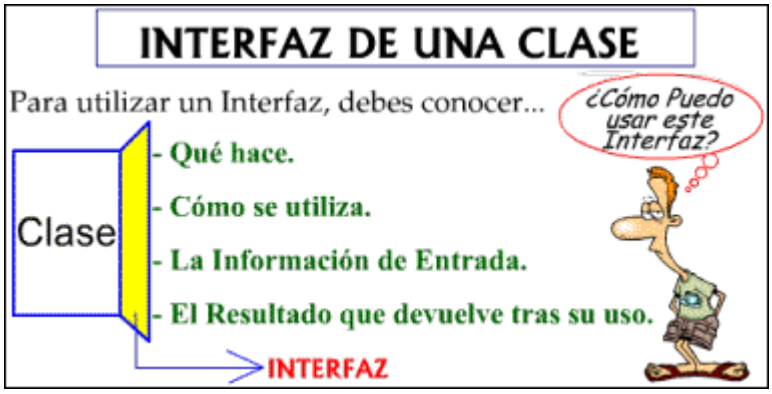
El cliente de una clase, al igual que el conductor del coche del ejemplo anterior, no necesita conocer el código fuente de dicha clase para poder utilizarla; es decir, **no necesita conocer su implementación**, no necesita conocer cómo ésta hace las cosas internamente. **Lo que realmente le interesa conocer es su interfaz**; es decir, qué hace la clase y cómo hay que interaccionar con ella para obtener los servicios que le ofrece. Así, **al cliente de una clase**:

- **Le interesa conocer qué información almacena la clase de la que es cliente y, por tanto, qué información puede proporcionarle ésta, pero no le interesa conocer en absoluto cómo dicha información está representada internamente en la clase**. Observa el código de la clase Trabajador que se te proporcionó en el apartado anterior. A una clase que sea cliente suya le interesa saber que dicha clase almacena, entre otras cosas, la edad del trabajador y que puede obtener ésta de alguna manera interactuando con la interfaz de la clase, en nuestro caso a través del método `getEdad()`. Sin embargo, poco o nada le interesa saber si dicha edad internamente en la clase Trabajador está almacenada en un atributo de tipo entero que contenga la edad en años del trabajador, o si lo que se almacena realmente en la clase es la fecha de nacimiento del trabajador, calculándose la edad en años dinámicamente a partir de la fecha de nacimiento almacenada y de la fecha actual en la que estemos.
- **Le interesa conocer qué servicios le ofrece la clase de la que es cliente, qué hacen los distintos métodos que proporciona, pero no le interesa conocer cómo están implementados dichos métodos, ni su código ni los algoritmos que utilizan los métodos para llevar a cabo sus tareas**. Observa el código de la clase "Trabajador" que se te proporcionó en el apartado anterior. A una clase que sea cliente suyo le interesa saber que dicha clase tiene un método que le proporcionará el sueldo en euros del trabajador, el método `getSueldo()`. Sin embargo, poco o nada le interesa saber cuál es el algoritmo o fórmula que se utiliza para obtener dicho sueldo; es decir, la implementación del mismo. Además, si posteriormente decidiese cambiar dicha fórmula, estos cambios sólo afectarán a este método de la clase Trabajador, pero en absoluto afectarán a los clientes de la clase, pues no existe vinculación alguna entre estos y la implementación del método.

Es decir, **en la programación orientada a objetos prima el "qué" sobre el "cómo"**, la interfaz sobre la implementación, donde **la interfaz de una clase especificará cómo se utilizan los servicios o funcionalidad ofrecida por la clase**, indicando para cada servicio:

- cómo se invoca o solicita,
- qué hace,
- qué información de entrada necesita, si es que necesita alguna, y
- qué información devuelve o proporciona como salida, si es que devuelve alguna.

Por su parte, **la implementación de una clase se oculta a sus clientes, que pueden utilizarla simplemente conociendo su interfaz**. De hecho, es lo que hemos estado haciendo hasta ahora en este curso cuando hemos usado las clases de la API de Java, las cuales hemos podido utilizar en nuestros programas sin haber visto nada absolutamente del código interno de las clases, simplemente haciendo uso de la interfaz descrita en la documentación de las mismas. Esto es lo que se conoce en el mundo de la orientación a objetos como **ocultación de la implementación**, y es una filosofía de programación que proporciona, principalmente, las siguientes **ventajas**:



- **Simplifica la percepción del cliente con respecto a una clase** pues sólo ve aquello que le interesa de la misma, su parte externa o interfaz. Imagina que para poder conducir un coche tuviésemos que accionar manualmente todos los elementos internos del coche. ¡Tendríamos que ser expertos en mecánica para poder conducir! Tener que conocer sólo la interfaz del coche para poder conducirlo nos hace la vida mucho más sencilla. De igual forma sucede con las clases, de las cuales sólo necesitamos conocer cómo funciona su interfaz para poder utilizarla.
- **Hace mucho más sencillo el mantenimiento de los programas, al hacer mucho menos traumática la modificación de los mismos**. Si sabemos manipular perfectamente los frenos y el volante de nuestro coche, los cuales pertenecen a la "interfaz" del mismo, ¿qué pasaría si decidiese llevar el coche al mecánico para que me cambiase el sistema de frenos normales por unos frenos con tecnología ABS o si le cambiase la dirección normal por una dirección asistida? ¿Dejaría de saber accionar los frenos del coche o de saber girar el volante del mismo? ¿Dejaría de saber conducir? Evidentemente no, pues lo que se ha modificado es la implementación de algunos aspectos internos del coche, pero se ha mantenido intacta su interfaz. Como yo, como conductor, sé manejar la interfaz, mientras no me cambien ésta no tengo por qué cambiar mi manera de conducir. Igualmente sucede con las clases, donde **cambiar la implementación de alguna parte de una clase**, por ejemplo, para mejorar su rendimiento mediante el uso de estructuras de almacenamiento o de algoritmos más eficientes, **no afecta a las clases que son sus clientes**,

que seguirán interactuando con ellas de la misma manera a como lo hacían antes de las modificaciones. Como puedes deducir, esta filosofía de programación orientada a la interfaz y con ocultación de la implementación favorece el desarrollo de software más fácil de modificar, ya que los cambios estarán localizados y no afectarán a otras partes del programa. Esto, evidentemente, siempre que los cambios no afecten a la interfaz de la clase, en cuyo caso sí que se verán afectados los clientes de la misma, de igual manera que situar los frenos del coche como una palanca en el volante en vez de como un pedal en el suelo, hará que tenga que modificar mi manera de conducir.

PARA SABER MÁS...
Si deseas profundizar sobre las API, en este enlace encontrarás una introducción a la API Reflection de Java.
Introducción al API Reflection (Reflexión) de Java [Versión en caché]

Paquetes en Java

Acabamos de hablar de la importancia de la **ocultación** de la implementación en la programación orientada a objetos y hemos establecido que la clase será la unidad mínima de encapsulación para llevar esta filosofía a cabo. Pero, ¿existen unidades mayores que la clase para la encapsulación y la ocultación de la información? En Java sí: los **paquetes**. Veamos qué son.

Un paquete no es más que un conjunto de clases que tienen alguna relación entre sí y que, por ello, se decide que confíen unas en otras**, donde esta relación de confianza implica que se va a permitir que puedan conocer unas las "interioridades" de las otras. Es decir, que** una clase va a poder tener acceso a la implementación del resto de clases de su mismo paquete, si no completamente, sí por lo menos en mayor medida que si no perteneciese al mismo**. Es por ello por lo que se considera que los paquetes son también, en cierto modo, unidades de encapsulación y ocultación de información.**

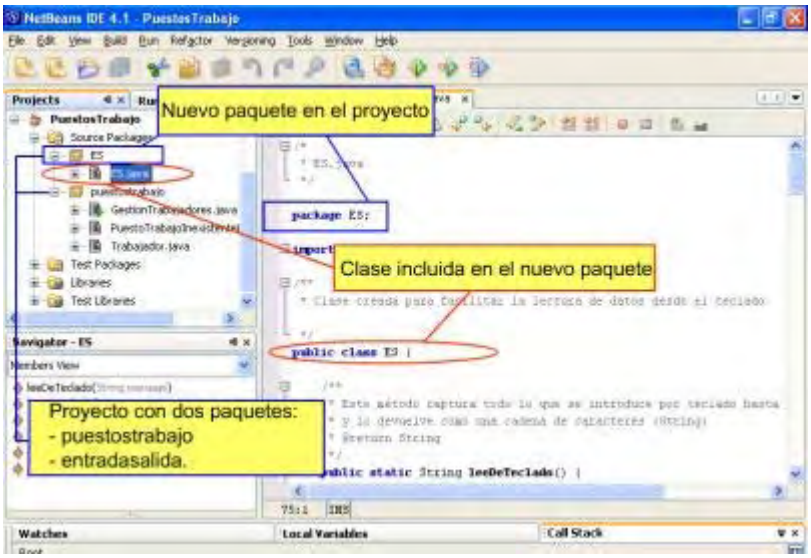
Seguro que la pregunta que te estás planteando en este instante es la siguiente: ¿qué clases deben pertenecer al mismo paquete? Lo cierto es que la decisión de qué clases pertenecerán al mismo paquete es importante, pues para una clase esto implicará la exposición de parte de su implementación al resto de clases del paquete. Generalmente, se suelen introducir en el mismo paquete todas las clases de un mismo proyecto o clases que vayan a pertenecer a una misma biblioteca.

Los paquetes se declaran utilizando la palabra clave **`package`** seguida del nombre del paquete. Para establecer el paquete al que pertenece una clase hay que poner una sentencia de declaración como la siguiente al principio de la clase:

```
package  Nombre_de_Paquete;
```

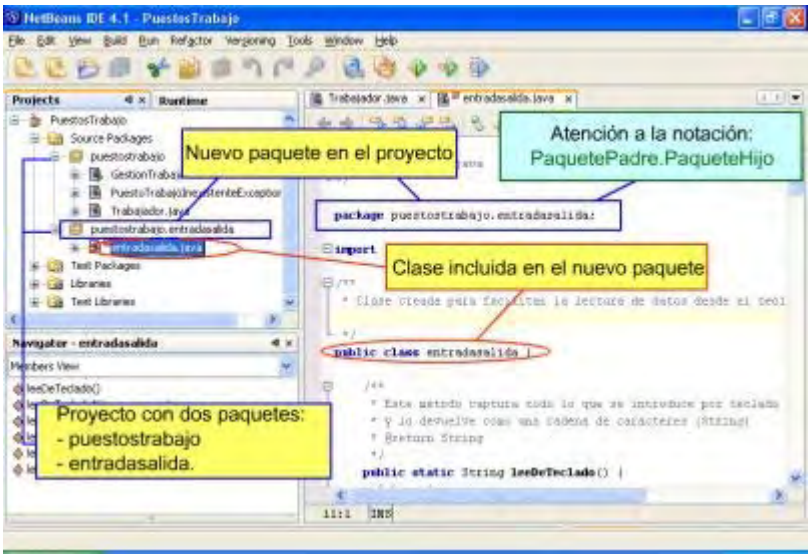
A la hora de utilizar una sentencia de este tipo debemos tener muy presente lo siguiente:

- La sentencia **`package`** no tiene por qué aparecer. Es decir, **una clase no tiene por qué pertenecer a un paquete**.
- **De aparecer la sentencia de definición, ésta debe aparecer al principio del archivo**. Si cualquier declaración o instrucción aparece en un archivo de código fuente en Java antes de la declaración **`package`**, entonces se producirá un error de sintaxis en tiempo de compilación.
- **Una clase puede pertenecer a un único paquete**, por lo que no puede haber dos sentencias de declaración **`package`** en una misma clase.
- Pertenecerán a un paquete todas las clases que comiencen con una sentencia de declaración de paquete y cuyo nombre de paquete en dicha sentencia coincida.



Java también soporta el concepto de jerarquía de paquetes. Esto es parecido a la jerarquía de directorios de la mayoría de los sistemas de ficheros de un sistema operativo, donde:

- Las clases serían el equivalente a los ficheros de un sistema de ficheros.
- Un paquete sería una carpeta que contiene clases, de igual modo que un directorio contiene ficheros en un sistema de ficheros. De hecho, realmente, **un paquete que manejamos a nivel de programación, al final deriva en una carpeta del sistema de ficheros del sistema operativo**.
- Un paquete, en vez de clases, puede contener otros paquetes, de igual modo que un directorio puede contener subdirectorios en un sistema de ficheros. Esto permite agrupar paquetes relacionados en un paquete más grande.
- Al igual que para referirnos a un fichero dentro de un sistema de ficheros tenemos que dar su ruta desde la raíz del árbol de directorios del sistema de ficheros, separando cada uno de los directorios de dicha ruta con una barra, para localizar una clase en la jerarquía de paquetes daremos la ruta desde el paquete raíz de la jerarquía, separando cada uno de los paquetes de dicha ruta con un punto.



En esta imagen puedes ver una demostración del uso de paquetes.

Así, por ejemplo, la ruta **`java.util.Date`** está señalando a la clase Date, que forma parte del paquete **`util`**, que al mismo tiempo es un subpaquete del paquete raíz **`java`**. Además, si miras en el fichero "src.zip" que encontrarás en el directorio en el que tengas instalado el JDK de Java, verás que se

corresponde con el fichero "Date.java" que se encuentra en la carpeta "util", que a su vez está dentro de la carpeta "java".

El mayor beneficio que se obtiene de esta manera de organización de las clases en paquetes es que proporciona una convención para tener nombres de clases que sean únicos. Con los cientos de miles de programadores en Java en todo el mundo, hay una muy buena probabilidad de que los nombres que elijas para tus clases tengan conflictos con los nombres que otros programadores elijan para las suyas. Sin embargo, el verdadero nombre de la clase está compuesto por toda su ruta por la jerarquía de paquetes, con lo que es más improbable que haya dos clases con el mismo nombre.

Hemos dicho en repetidas ocasiones ya, que uno de los principales beneficios que vamos a obtener con la filosofía de programación orientada a objetos es que va a ser muy fácil la **reutilización** del trabajo ya hecho. En nuestro caso, la reutilización de clases ya hechas y que podemos volver a usar en otros programas. Para desde una clase poder hacer uso de otra clase ajena a su paquete, habrá que anteponer al nombre de la clase la ruta completa por la jerarquía de paquetes.

Así, por ejemplo, suponiendo que la clase "Trabajador" perteneciese al paquete "paquete1.subpaquete2", para acceder al método **`getSueldo()`** de la clase "Trabajador" desde una clase ajena al paquete al que pertenece dicha clase, habrá que hacerlo mediante la sentencia:

```
paquete1.subpaquete2.Trabajador.getSueldo();
```

Para no tener que anteponer toda la ruta, lo cual puede ser una tarea tediosa si tenemos muchas referencias a la clase, podemos importar el paquete o la clase haciendo uso de la sentencia **`import`**, de la siguiente manera:

```
import Nombre_de_Paquete;
```

A la hora de utilizar una sentencia de este tipo debemos tener muy presente lo siguiente:

- Esta sentencia debe aparecer al principio de la clase, justo después de la sentencia **`package`**, si ésta existiese.
- Para importar la clase Trabajador desde una clase ajena a su paquete, se utilizaría una sentencia como la siguiente:
  `import paquete1.subpaquete2.Trabajador;`
- Si queremos importar de una vez todas las clases de un paquete, por ejemplo, del paquete subpaquete2 anterior, se utilizaría una sentencia como la siguiente:
  `import paquete1.subpaquete2.*;`.
  Ésta es una forma sencilla de tener acceso a todas las clases de un determinado paquete, aunque el uso del asterisco debe hacerse con cautela, pues obligará al compilador a cargar todas las clases del paquete, lo que hará más lenta la compilación. Por lo tanto, sólo es recomendable utilizar la notación asterisco cuando se vayan a usar varias clases de un mismo paquete y nunca cuando sólo se va a usar una. En ese caso, es preferible importar dicha clase únicamente. No obstante, el asterisco no tiene impacto alguno a la hora de la ejecución, sólo en tiempo de compilación.
- Deberá aparecer una sentencia **`import`** por cada clase o paquete importado.

La gestión de paquetes puede haberte parecido un tanto compleja. Afortunadamente, **el entorno de programación que estamos utilizando, NetBeans, facilita en extremo todo lo relativo a la gestión de paquetes**: creación, uso y la propia organización de las clases en paquetes.

DEMO: Visualiza cómo añadir un nuevo paquete

Hasta aquí hemos visto lo que son los paquetes y cómo se utilizan. En el apartado siguiente veremos cómo Java usa los paquetes y las clases como parte de la estrategia de control de acceso y ocultación de

la información. ¡Vamos allá! Veamos el control de acceso en Java.

Control de acceso en Java

Ya hemos hablado de la importancia de la **ocultación** de la implementación en la programación orientada a objetos y de los dos niveles que Java proporciona para ello: las clases y los paquetes. Veamos ahora qué mecanismos concretos me ofrece Java para poder llevarla a cabo.

Como ya sabemos, cada atributo y cada método de una clase tienen en su **cabecera** de definición unos elementos llamados modificadores que sirven para expresar alguna característica del atributo o del método. Algunos de esos modificadores son comunes a métodos y atributos y sirven para expresar el nivel de ocultación de los mismos, más comúnmente llamado nivel de acceso o control de acceso. Estos modificadores son, ordenados de más restrictivo a menos restrictivo, los siguientes:

- **La palabra clave `private`**. Es el nivel de acceso más restringido e indica que **al atributo o método que vaya precedido por este modificador sólo se podrá acceder desde la propia clase**. Es decir, si se trata de un atributo privado, como el atributo **`nif`** de la clase Trabajador, su valor sólo podrá ser leído o escrito desde los métodos de la propia clase, como por ejemplo los métodos **`getNif()`** y **`setNif()`** de la clase Trabajador, y nunca desde ningún método de ninguna otra clase, ya sea ésta cliente suya o su subclase. Por su parte, si se trata de un método privado, como el método **`comprobarNif()`** de la clase Trabajador, sólo podrá ser invocado para su ejecución desde los métodos de la propia clase y nunca desde ningún método de ninguna otra clase, ya sea ésta cliente suya o su subclase. De hecho, si un método que no sea miembro de una clase específica trata de acceder a uno de los miembros privados de esa clase, se producirá un error de sintaxis detectable en tiempo de compilación.
- **La palabra clave `package`**. Este nivel de acceso es el que **se usa por defecto si no se especifica nada** e indica que **el atributo o método que vaya precedido por este modificador podrá ser accedido desde cualquier clase del mismo paquete, pero no desde una clase perteneciente a otro paquete**.
- **La palabra clave `protected`**. Es un nivel de acceso menos restrictivo que el privado e indica que **al atributo o método que vaya precedido por este modificador sólo se podrá acceder desde la propia clase, desde sus subclases y desde las clases que pertenezcan a su mismo paquete.** Es decir, si se trata de un atributo protegido, su valor sólo podrá ser leído o escrito desde los métodos de la propia clase, desde los métodos de alguna de sus subclases o desde los métodos de alguna clase de su mismo paquete, y nunca desde ningún método de ninguna otra clase. Por su parte, si se trata de un método protegido, sólo podrá ser invocado para su ejecución desde los métodos de la propia clase, desde los métodos de alguna de sus subclases o desde los métodos de alguna clase de su mismo paquete, y nunca desde ningún otro método de ninguna otra clase.
- **La palabra clave `public`**. Es el nivel de acceso menos restrictivo e indica que **al atributo o método que vaya precedido por este modificador se podrá acceder desde cualquier otra clase**. Es decir, si se trata de un atributo público, como el método **`getSueldo()`** de la clase Trabajador, cualquier método de cualquier clase podrá leer o escribir su contenido, y si se trata de un método, público cualquier método de cualquier clase podrá invocarlo.

Estas relaciones de visibilidad quedan perfectamente reflejadas en la siguiente tabla, donde en las filas tenemos los distintos modificadores y marcamos con una equis en las columnas de las clases que tendrían acceso a un miembro con dicho modificador: la propia clase, las subclases, las clases del

paquete y todas las clases, respectivamente.

Niveles de Acceso	OBJETOS: Clase	Subclase	Paquete	Todos
private	✓	X	X	X
package	✓	X	✓	X
protected	✓	✓	✓	X
public	✓	✓	✓	✓

O expresado de otra manera:

OBJETOS	Niveles de Acceso: private	package	protected	public
Misma Clase.	✓	✓	✓	✓
Subclase del mismo Paquete.	X	✓	✓	✓
No-Subclase del mismo Paquete.	X	✓	✓	✓
Subclase de diferente Paquete.	X	X	✓	✓
No-Subclase de diferente Paquete.	X	X	X	✓

Es importante remarcar que en Java el control de acceso se aplica siempre a nivel de clase, no a nivel de objeto. Es decir, **los métodos de un objeto de una clase determinada tienen acceso directo a los miembros privados de cualquier otro objeto de la misma clase**.

PARA SABER MÁS...
En este enlace podrás encontrar ejemplos del acceso a los miembros de clase en Java
Controlar el Acceso a los Miembros de la Clase [Versión en caché]

Métodos públicos de acceso a atributos privados (I)

En el apartado anterior vimos una serie de **modificadores** que servían para controlar el acceso a los distintos atributos y métodos de una clase: `private`, `protected`, `public` y `package`. En principio, vamos a considerar sólo los dos extremos, los modificadores `private` y `public`, y vamos a olvidarnos del resto, que no son más que variaciones intermedias entre ellos y serán tratados en el tema siguiente. Si recordamos:

- El modificador `private` era el más restrictivo de todos, permitiendo sólamente el acceso al atributo o al método declarado como privado desde dentro de los métodos de su propia clase.
- Por el contrario, el modificador `public` era el menos restrictivo de todos, permitiendo el acceso al atributo o al método declarado como público desde cualquier método de cualquier otra clase.

Ahora, la pregunta que queda en el aire es: ¿cuándo debo utilizar uno y cuándo debo utilizar otro? **¿Cuándo declaro un atributo o un método como privado y cuándo lo hago cómo público?**

A la hora de tomar esta importante decisión en el diseño de nuestras clases debemos tener siempre presente el principio de ocultación de la implementación, que establecía que sólo debe darse a conocer la interfaz de la clase, quedando oculta su implementación. Por lo tanto:

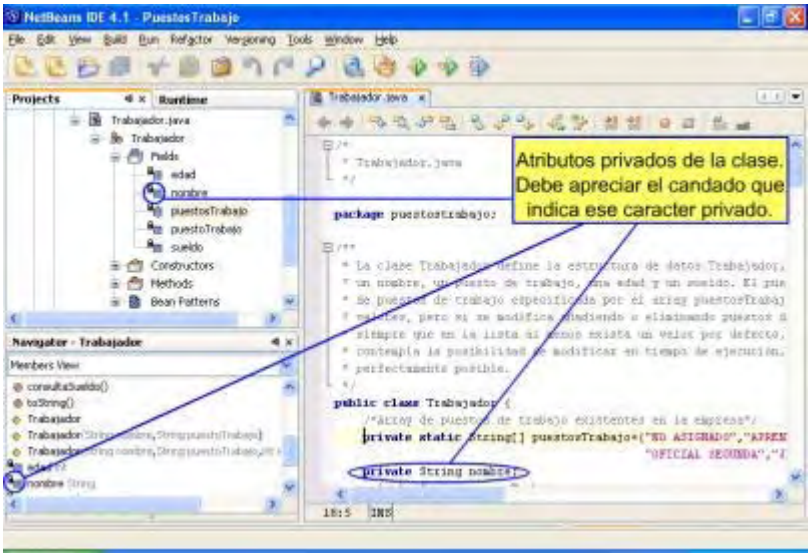
- Deberíamos utilizar el modificador `public` para todos aquellos atributos o métodos de la clase que

vayan a formar parte de la interfaz de la clase; es decir, para presentar a los clientes de la clase una vista de los servicios que la clase proporciona.

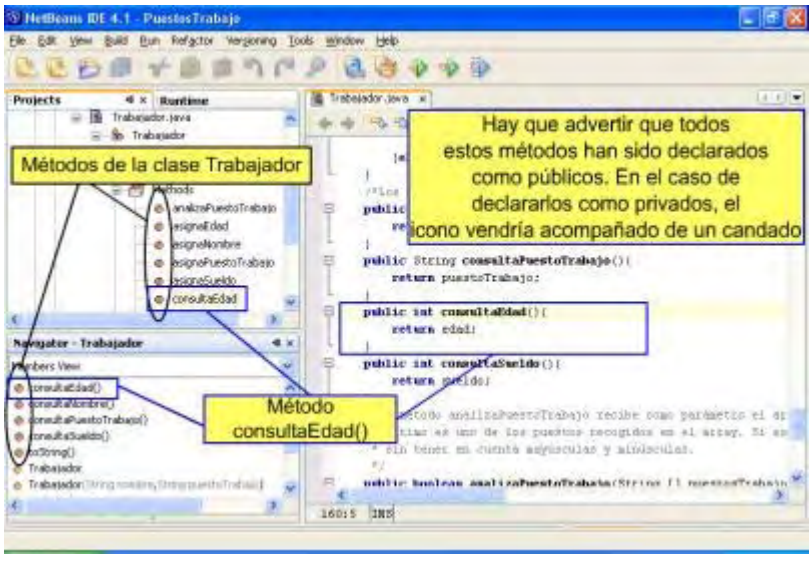
- Deberíamos utilizar el modificador **`private`** para el resto; es decir, para ocultar toda aquella información (atributos y métodos) de la clase que no pertenece a la interfaz de la misma y, por lo tanto, no debe ser conocida por las otras clases.

La mejor forma para tomar una decisión sobre qué miembros de la clase deben ser públicos y cuáles privados es analizar y definir cuál y cómo va a ser la interfaz de la clase, lo cual es, sin duda, el punto crítico del diseño de toda aplicación orientada a objetos pues, como sabemos, la programación orientada a objetos es una programación orientada a la interfaz. Una vez localizada la interfaz de la clase, se marcan todos los miembros de la misma como públicos y el resto como privados. Los expertos en ingeniería del software dan un consejo básico a la hora de buscar dicha interfaz: "**haga que el miembro de una clase sea privado si no hay razón para que se utilice fuera de la propia clase**". Además, más concretamente establecen que:

- **Los atributos de una clase nunca deben pertenecer a la interfaz de la misma y, por lo tanto, deberán ser declarados como privados**. Observa el código de la clase Trabajador que se te proporcionó en un apartado anterior de este mismo tema y verás cómo todos los atributos han sido declarados como privados.

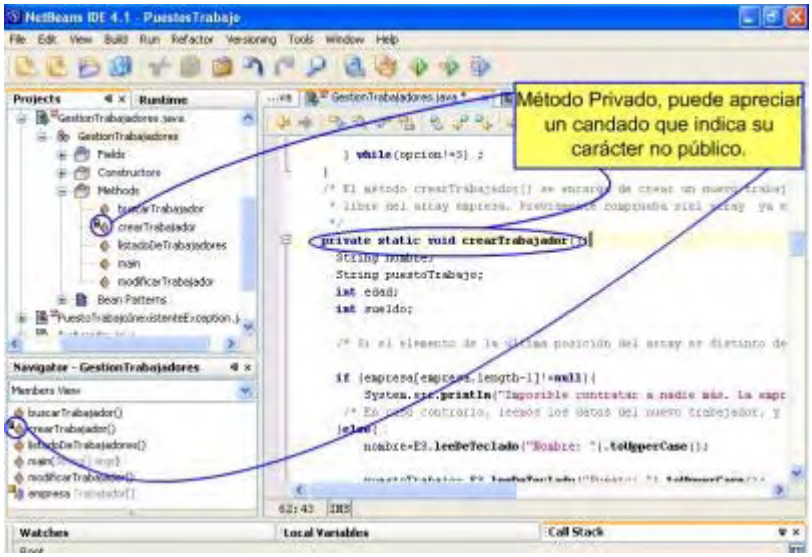


- Por su parte, **generalmente los métodos de la clase pertenecerán a la interfaz de la misma y, por lo tanto, deberán ser declarados como públicos**. Observa el código de la clase Trabajador y sabrás cuál es la interfaz de la clase viendo cuáles de los métodos de la misma son públicos.



- Sin embargo, **es posible que tengamos la necesidad de tener métodos privados dentro de una clase**. Estos métodos reciben el nombre de **métodos utilitarios o métodos ayudantes**, ya

que **pueden ser llamados sólo por otros métodos de esa clase** y su misión es la de ayudarles a realizar sus tareas internas. Observa en el código de la clase Trabajador el método **`comprobarFecha()`**. Este método se encarga de comprobar, dada una fecha, que ésta es correcta; es decir, por ejemplo, que si es una fecha del mes de abril, el día no sea mayor que 30. Este método es privado y proporciona este servicio de comprobación a varios de los métodos públicos de la clase, entre ellos el método **`setFechaNacimiento()`** y el método **`setFechaAlta()`**, de tal manera que les ayuda a realizar sus tareas y nos evita tener que repetir el mismo código de comprobación de fechas en ambos métodos. Sin embargo, el método **`comprobarFecha()`** es privado, lo que quiere decir que no pertenece a la interfaz de la clase y no podrá ser invocado desde una clase cliente de la clase Trabajador, pues éstas ni siquiera sabrán de su existencia. En la siguiente imagen se puede apreciar cómo hemos hecho una pequeña modificación en nuestra aplicación declarando el método **`crearTrabajador()`** como Privado, con el fin que mostrar su representación en NetBeans



.

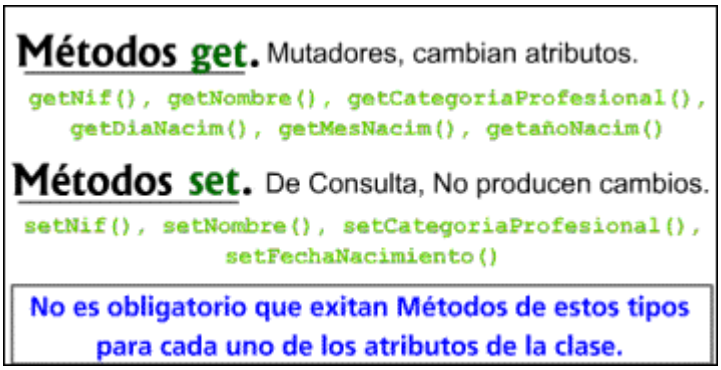
Métodos públicos de acceso a atributos privados (II)

Declarar todos los atributos como privados tiene una gran implicación: **la única manera de acceder a los valores de los atributos o de modificarlos desde el exterior de la clase será a través de los métodos públicos de la propia clase**, pues desde el mundo externo a la clase nadie conoce la existencia de dichos atributos, es una información que se les ha ocultado.

Las clases proporcionan a menudo métodos públicos para permitir a los clientes de la clase establecer (escribir) u obtener (leer) directamente los valores de los atributos privados:

- Los **métodos "establecer"**, o **métodos "set"** en inglés, también se conocen comúnmente como **métodos mutadores**, porque cambian el valor de un atributo.
- Por su parte, los **métodos "obtener"**, o **métodos "get"** en inglés, también se conocen comúnmente como **métodos de acceso o métodos de consulta**.
- Por convención dichos métodos **se nombran con la palabra set o get**, o sus correspondientes en castellano, **junto al nombre del atributo que modifican o consultan**. No es obligatorio llamar a estos métodos de esta forma, aunque sí aconsejable por respetar lo mundialmente establecido por la comunidad de programadores. Así, por ejemplo, en nuestra clase Trabajador encontramos los siguientes métodos "get":**`getNif()`**, **`getNombre()`**, **`getDiaNacimiento()`**, **`getMesNacimiento()`**, **`getAñoNacimiento()`**, **`getCategoriaProfesional()`**, **`getDiaAlta()`**, **`getMesAlta()`** y **`getAñoAlta()`**. Por su parte, también encontramos los siguientes métodos "set": **`setNif()`**, **`setNombre()`** y **`setCategoriaProfesional()`**.
- **No es obligatorio que exista un método "set" y un método "get" para cada atributo privado de la clase**, éstos sólo deberán proporcionarse cuando tengan sentido. Así, por ejemplo, si por el

diseño de la propia clase hay un atributo privado de la misma que nunca va a ser establecido directamente desde fuera, no será necesario que dicho atributo tenga un método "set" asociado. En nuestra clase Trabajador, por ejemplo, no existe un método "set" para los atributos **`diaNacimiento`**, **`mesNacimiento`** y **`añoNacimiento`**. Sin embargo, existe un único método, llamado **`setFechaNacimiento()`**, que establece el valor de los tres atributos simultáneamente.



Podríamos pensar que proporcionar métodos públicos para establecer y obtener los valores de atributos privados es exactamente lo mismo que hacer que dichos atributos fuesen directamente públicos, pero no es así. Veamos porqué:

- **Si una clase tiene algún atributo declarado como público, cualquier método de cualquier clase podrá leer o modificar su contenido**. Esto supone una violación del principio de ocultación de la información y de la programación orientada a la interfaz, pues dichas clases conocerán perfectamente cómo se almacena la información en el interior de la clase y **quedarán ligadas a dicha implementación**.
- Por el contrario, si el atributo es declarado como privado y tenemos un método mutador y un método de acceso públicos, es cierto que se está permitiendo a cualquier método de cualquier clase leer o modificar su contenido, pero tendrán que hacerlo a través del método público establecido para ello. Por lo tanto, **dichos métodos servirán para ocultarles la verdadera implementación de esa información en la clase, que "verán" el atributo única y exclusivamente a través de estos métodos**.
- Además, **dichos métodos servirán para controlar la manera en la que los clientes pueden tener acceso al atributo, impidiendo que algún agente externo pueda introducir en los atributos valores no válidos para los mismos**.

Veámoslo con un **ejemplo** sobre la clase Trabajador que estamos construyendo. A una clase que sea cliente de la clase Trabajador le interesa saber que dicha clase almacena, entre otras cosas y de alguna manera, la edad del trabajador. Supongamos que la clase Trabajador tuviese definido un atributo de tipo entero, llamado **edad**, que va a almacenar los años del trabajador:

- Si dicho atributo fuese público y las clases tuviesen que acceder directamente a él, quedarían ligadas a la implementación del mismo, que es un número de tipo entero. Si posteriormente decidiésemos que en vez de la edad como tal, queremos almacenar la fecha de nacimiento del trabajador y obtener su edad a partir de la misma; es decir, eliminar el atributo **`edad`** y tener en su lugar tres atributos, **`diaNacimiento`**, **`mesNacimiento`** y **`añoNacimiento`**, entonces tendríamos que modificar todas aquellas clases que fuesen cliente de la clase Trabajador, pues estaríamos modificando la interfaz de la clase.
- Sin embargo, si hacemos que el atributo **`edad`** sea **privado** y tenemos un método **`setEdad()`** y otro **`getEdad()`** públicos para acceder a él, las clases clientes desconocen totalmente la implementación real de dicho atributo, que podría ser tal y como hemos descrito, o podría ser la terna de atributos **`diaNacimiento`**, **`mesNacimiento`** y **`añoNacimiento`**. En el primer caso, el método **`getEdad()`** devolvería directamente el valor del atributo **`edad`**, mientras que en el segundo caso el método **`getEdad()`** realizaría los cálculos necesarios para hallar y devolver la edad en años del trabajador a partir de la fecha de nacimiento del mismo y la fecha actual. En cualquier caso, las clases clientes ignoran este detalle de implementación, pues ellas sólo "ven" dicha información de la clase a través de los métodos públicos "set" y "get", que son los que se

tendrán que encargar de realizar las transformaciones pertinentes entre la visión que las clases clientes tienen del atributo y la implementación real del mismo. Por lo tanto, cambiar los atributos privados de la clase o cambiar la implementación del método `getEdad()` no afecta a las clases clientes al no cambiar la interfaz de la clase: el método público `getEdad()` sigue siendo un método sin parámetros que devuelve un número entero.

- Además, estos métodos "set" y "get" se pueden ocupar de impedir que, por ejemplo, alguien intente introducir una edad negativa, cosa que sería imposible de controlar si las clases clientes accediesen directamente al atributo de la clase para modificar su valor. Coge el código de la clase Trabajador que se te ha proporcionado, y observa cómo los métodos `setFechaNacimiento()`, `setFechaAlta()`, `setNif()` y `setCategoriaProfesional()` realizan estas tareas de control para que no se pueda introducir un NIF, una fecha o una categoría profesional incorrecta en los atributos correspondientes de la clase.

Como acabamos de ver, y como ya hemos dicho en repetidas ocasiones, **el ocultamiento de la información fomenta la capacidad de modificar los programas y simplifica la percepción que tiene el cliente acerca de una clase**.

Objetos o instancias de una clase

*Antes de comenzar la jornada laboral, **Víctor** coincide durante el desayuno con **José**, en una cafetería cercana a las oficinas de **SI Andalucía**. Mientras toman sus respectivos desayunos, **José** se interesa por sus avances en Programación Orientada a Objetos. **Víctor** piensa que va bastante bien, aunque reconoce que necesitaría más tiempo para asentar las ideas y practicar un poco con programas reales, antes de poder asegurar que domina estas técnicas.*

*Ya en la oficina, cuando **Carmen** continúa con su explicación sobre objetos, **Víctor** se da cuenta de que esto no ha hecho más que empezar y que a partir de este momento debe comenzar a crear objetos y trabajar con ellos. Algo que le ha costado entender es que una **Instancia** de una clase, es un objeto definido con esa clase. Como **Carmen** le explicó, con la clase Trabajador podemos construir todos los trabajadores (objetos) que vamos a necesitar en la aplicación de Gestión de Personal, cada uno de estos trabajadores será una Instancia de la clase Trabajador con un identificador único.*

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, si bien antes recordaremos los aspectos más importantes ya estudiados acerca de los objetos:

- Para obtener un objeto debe existir una clase a partir de la cual instanciarlo, donde la clase no es más que una plantilla que define la forma (atributos y comportamiento) que tendrán los objetos que se construyan a partir de dicha plantilla.
- Instanciar un objeto no es más que crear un objeto **concreto** a partir de una clase, donde dicho objeto seguirá la estructura (atributos y comportamiento) que venía definida en la clase de la cual procede. Es decir, al instanciar un objeto se le asigna espacio en memoria para sus características; es decir, tanto para almacenar los valores de sus atributos como para los métodos que implementan el comportamiento del objeto.
- Todo objeto es instancia de una única clase y toda clase que forma parte del programa tiene, en un instante dado de la ejecución del mismo, cero o más objetos que son instancia de ella.
- 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, junto a 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.
- Los objetos no existen hasta que el programa no empieza a ejecutarse. A partir de ese momento los objetos empiezan a crearse, a interactuar y a desaparecer según indique el propio programa.

Resumiendo, podemos afirmar que **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;
```

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:

```
nombre_objeto = new constructor_de_la_clase(parámetros_del_constructor);
```

Como vemos, para asignar un objeto a una referencia, se hace una llamada al constructor de la clase por medio del operador **`new`**, donde:

- **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 recordemos 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. 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_clase nombre_objeto = new constructor_de_la_clase(parámetros_del_constructor
```

Y que 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, para distinguir entre los objetos, el lenguaje lo que hace es asignar a cada objeto un identificador único, llamado oid o identificador de objeto**. Por lo tanto, podemos afirmar que:

- Dos objetos con diferentes oids pueden tener los mismos valores en sus campos.
- Los valores de los campos de un objeto pueden cambiar, pero su oid es inmutable.

Miembros de objeto y 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**. En nuestro ejemplo de la clase Trabajador:

- Son variables de instancia las variables: **`nif`**, **`nombre`**, **`diaNacimiento`**, **`mesNacimiento`**, **`añoNacimiento`**, **`categoriaProfesional`**, **`diaAlta`**, **`mesAlta`** y **`añoAlta`**.
- Por su parte, son métodos de instancia los métodos: **`setNif()`**, **`getNif()`**, **`setNombre()`**, **`getNombre()`**, **`setFechaNacimiento()`**, **`getDiaNacimiento()`**, **`getMesNacimiento()`**, **`getAñoNacimiento()`**, **`setCategoriaProfesional()`**, **`getCategoriaProfesional()`**, **`setFechaAlta()`**, **`getDiaAlta()`**, **`getMesAlta()`**, **`getAñoAlta()`**, **`comprobarNif()`**, **`comprobarFecha()`**, **`comprobarCategoria()`**, **`getEdad()`** y **`getSueldo()`**.

Los miembros de objeto, las variables y los métodos de instancia, deberían sernos extremadamente familiares, pues es de lo que hemos estado hablando, sin mencionarlo expresamente, durante todo el tema al hablar de los atributos y los métodos de los objetos. 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 por 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 o mensajes.

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.

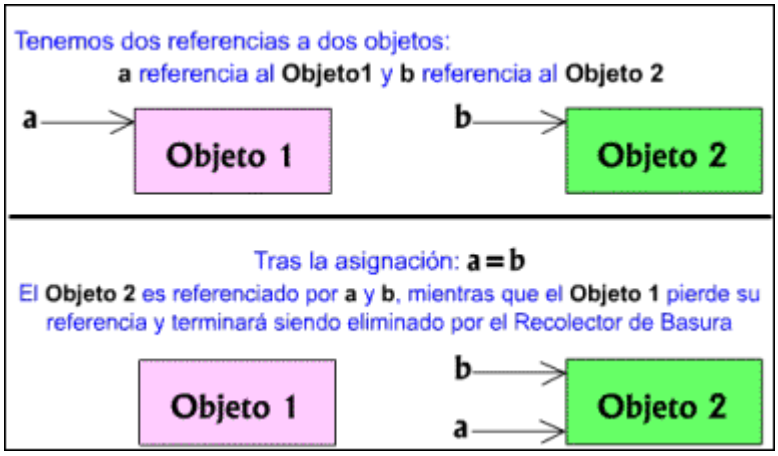
Cuestiones importantes sobre las referencias a objetos

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:

- **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.
- 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, aún 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**. 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.

Autoevaluación

Métodos o Mensajes

*Durante el fin de semana **Víctor** repasa todos los conceptos aprendidos sobre objetos y su manejo. Por sí mismo ha llegado a la conclusión de que este tipo de programación requiere dos tareas, una primera de preparación de los objetos (en la que define sus propiedades y comportamiento) y segunda que sería la programación en sí utilizando esos objetos según la definición anterior. Pero lo que en cualquier caso ha podido constatar es que los programas están formados por métodos de código en Java.*

*Tras practicar con varios ejemplos que su compañera **Carmen** le había propuesto, **Víctor** considera que ya ha adquirido la capacidad mínima necesaria para programar en Java y el lunes llegará dispuesto a demostrarlo.*

Llegados a este punto, momento en el que pretendemos aprender lo que son los métodos, te darás cuenta de que ya hemos visto bastante sobre ellos, no sólo en este tema, sino en temas anteriores, y de que conocemos casi todo lo que hay que conocer sobre los mismos. Básicamente, si hacemos repaso a cuestiones conceptuales ya estudiadas sobre los métodos, tenemos que:

- Los **métodos** son el conjunto de funcionalidades que describen la naturaleza y el comportamiento que tendrán los objetos de una clase; es decir, las operaciones aplicables a dichos objetos.
- Un método bien construido debería ejecutar una única tarea.
- Si hemos diseñado bien nuestra clase, respetando los principios de la ocultación de la información y la implementación, la única manera de acceder a los atributos debe ser a través de los métodos. Así, los atributos de la clase se habrán declarado como privados y los métodos, en su gran mayoría, como públicos. Además se habrán proporcionado métodos "get" y "set" para acceder a los atributos a los que sea necesario acceder, ya sea para consulta o para modificación de los mismos.
- En la **definición de una clase** podemos distinguir entre dos tipos de métodos: los métodos de instancia y los métodos de clase, donde los primeros se usarán para acceder a las variables propias de cada objeto instanciado o variables de instancia, y los segundos se usarán para acceder a las variables compartidas por todos los objetos de la clase o variables de clase.
- Los **parámetros** que se pasan a un método para proporcionarle cierta información que necesita para realizar su tarea, se pasan por valor; es decir, es como si fuesen variables locales dentro del método, de tal manera que cualquier modificación que se haga sobre ellas no tiene efectos fuera de lo que es el propio método. Sin embargo, debemos recordar que esto no sucede así cuando los parámetros no son de tipo primitivo sino objetos. En este caso, como lo que se pasa es una referencia al objeto, cualquier cambio que se haga en el mismo estará vigente incluso al salir del método.
- Los métodos pueden devolver algún valor, incluso la referencia a un objeto. Por lo tanto, hay que indicar cuál es el tipo de dato de aquello que se devuelve. Pero también pueden no devolver nada, en cuyo caso se usará la palabra clave `void` como tipo devuelto por el método.

Como programadores, una vez definida una clase y **una vez instanciados los objetos** de dicha clase, **para interactuar con**

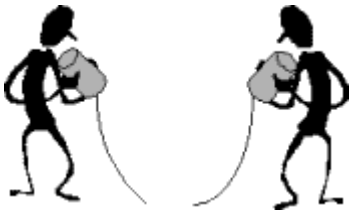
esos objetos y modificar su estado (modificar los valores de sus variables de instancia), **debemos provocar la ejecución de los métodos públicos que dichos objetos ofrecen**:

- Al hecho de **provocar la ejecución de un método de un objeto se le llama** "**realizar una petición**", "**solicitar un servicio**" o "**enviar un mensaje**" al objeto.
- Por su parte, desde el punto de vista del objeto, **recibir la solicitud para que ejecute uno de sus métodos recibe el nombre de** "**recibir una petición**", "**recibir una solicitud de servicio**" o "**recibir un mensaje**".
- **El emisor de un mensaje será siempre un objeto e, igualmente, el receptor de un mensaje será otro objeto. Por lo tanto, un mensaje no es más que una forma de comunicación o de colaboración entre los objetos**.
- Además, debemos tener en cuenta que **un objeto realiza una operación única y exclusivamente cuando recibe una petición o mensaje de un objeto cliente**. De hecho, los mensajes son el único modo de lograr que un objeto ejecute una operación o método y las operaciones o métodos son la única forma de cambiar los datos internos de un objeto.

Por lo tanto, con todos estos datos, podemos concluir que **un programa orientado a objetos no es más que una colección de objetos que intercambian mensajes unos con otros con el objetivo de llevar a buen puerto el cometido para el que el programa está hecho**.

Entonces, 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:

Primeramente, 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. Recordemos que 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 éste 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:

```
unTrabajador.getNif();

unTrabajador.getSueldo();

unTrabajador.getEdad();
```

Sobrecarga de métodos

Observa el código de la clase Trabajador que se te ha proporcionado y fíjate en el 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

   }
```

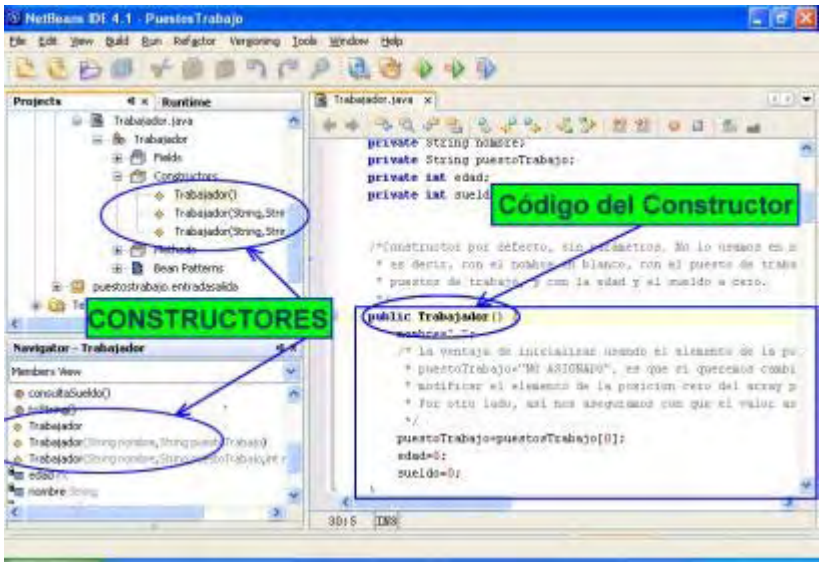
- De lo anterior se deriva que **no puede haber en una misma clase dos métodos que se llamen igual y que además tengan el mismo número y tipo para sus parámetros**. Declarar métodos con el mismo nombre y con listas idénticas de parámetros es un error de sintaxis y daría un error en tiempo de compilación.
- 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.

Constructores (I)

*Al llegar a la empresa tras el fin de semana, **Víctor** está dispuesto a demostrarle a **Carmen** su capacidad de programación de métodos, cuando ella le comenta la existencia de unos métodos especiales que se invocan para crear instancias de clase. Son los métodos Constructores, que pueden ser declarados de diferentes modos, es lo que se llama sobrecarga del método.*

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.



Como ya dijimos en su momento, 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 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 **`nombre_objeto`** es la referencia que hemos creado para conectarla al objeto, 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.

Constructores (II)

Podemos afirmar que **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 boolean, 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.
- Un constructor puede llamarse **sin argumentos** sólamente 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.
- 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.

Autoevaluación

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. 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 como 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.

```
      // Primer constructor de la clase que inicializa el NIF y el nombre del trabajador
public Trabajador(String nif, String nombre) {
        if (this.comprobarNif(nif)) {
            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, pa
   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.
        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 valo
                this.diaAlta=0;
                this.mesAlta=0;
                this.añoAlta=0;
            }
        }
```

Autoevaluación

Uso del operador this y del método this()

*Después de trabajar con los constructores, **Carmen** le presenta el código del programa sobre el que está trabajando actualmente para que entienda su forma de programar y explicarle algunas cosas. Al leer el código **Víctor** advierte algunas cosas que desconoce, concretamente la palabra this.*

***Carmen** le explica que ella lo utiliza como un operador que le permite en algunos casos evitar ambigüedad cuando hay atributos de diferentes objetos con el mismo nombre, y de ese modo indicar que se refiere al objeto definido en ese archivo.*

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 (comprobarFecha(dia, mes, año)){
           // Si se trata de una fecha válida procedemos a cambiar los valores de las var
           // 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 asignad
        }
    }
```

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

```
comprobarFecha(dia, mes, año)
```

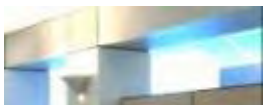
Observa también cómo se accede a las variables de instancia **diaNacimiento**, **mesNacimiento** y **añoNacimiento** para establecerles un valor:

```
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**. 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 var
            // 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 asigna
        }
    }
```

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:

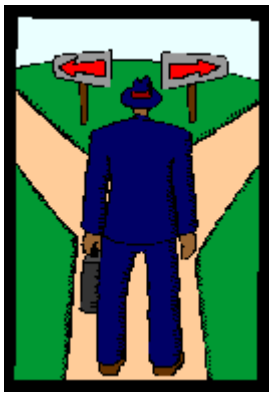
```
public void setNombre(String nombre){

        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 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 aún 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.

Pero 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**.

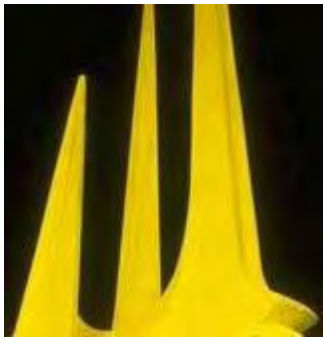
Observa el código del segundo de los constructores de la clase Trabajador. Como primera instrucción aparece una llamada al método **`this()`**. De hecho, y esto es importante, **de usarse el método `this()`, forzosamente tiene que ser la primera línea de código del constructor**.

```
        // Primer constructor de la clase que inicializa el NIF y el nombre del trabajador
    public Trabajador(String nif, String nombre) {
            if (this.comprobarNif(nif)) {
                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 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
            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 valo
                    this.diaAlta=0;
                    this.mesAlta=0;
```

```
            this.añoAlta=0;
        }
    }
```

Introducción al concepto de Interface

*Ha llegado para **Víctor** el momento de programar sus primeros métodos siguiendo la filosofía de la Programación Orientada a Objetos. Él lo ha resumido todo en la definición de la clase (mediante sus atributos y sus métodos) para su posterior utilización como parte del código en Java. Dice que una vez que ha desarrollado un par de programas sencillos, todo se ve más claro, resulta relativamente sencillo definir las clases y una vez hecho esto, su uso es casi intuitivo.*



Ya estamos llegando al final de esta unidad en la que hemos introducido lo que es la programación orientada a objetos y los conceptos más importantes que se manejan en la misma. De todo lo aprendido, si tuviésemos que quedarnos con una única frase que resumiese la filosofía de este paradigma de programación, sería la siguiente: **la programación orientada a objetos es una programación orientada a la interfaz que pretende ocultar la implementación; es decir, es una filosofía de programación que se centra en el qué se hace en vez de en el cómo se hacen las cosas**.

A lo largo del tema hemos visto que una clase define una parte pública, que es la que da a conocer al resto del mundo los servicios que ofrecen los objetos de dicha clase. **Esta parte pública de la clase configura lo que llamábamos interfaz de la clase** y es la parte con la que interactuará cualquier entidad que quiera utilizar algún objeto de dicha clase.

Pero esta idea de lo que es una **interfaz** podemos llevarla aún más lejos. Imagina cualquier dispositivo en el que puedas almacenar información, desde una libreta hasta un CD-ROM. ¿Qué tienen en común todos ellos? En principio podríamos decir que más bien poco ¿verdad? Sin embargo, si no nos quedamos simplemente en la superficie del aspecto y profundizamos un poco en lo que es la utilidad que tienen, podremos vislumbrar que todos los artefactos o dispositivos que hayas podido pensar, desde el más moderno al más arcaico, tienen en común lo siguiente:

- Permiten escribir información en el dispositivo,
- almacenan internamente, de algún modo, dicha información
- y permiten leer o recuperar la información almacenada.

Además, cualquier artefacto nuevo que inventemos y que cumpla dichas características, podremos considerarlo también un dispositivo de almacenamiento. Es cierto que cada uno de estos artefactos almacena la información de un modo distinto y tiene mecanismos distintos para leer y escribir la información; pero también es cierto que todos ellos ofrecen, de un modo u otro, la misma funcionalidad. Podríamos entonces considerar estas características que definen la funcionalidad de "ser dispositivo de almacenamiento" como una interfaz con entidad independiente, llamémosle **interface**, la cual **especifica el comportamiento que debe tener todo artefacto que quiera ser considerado "dispositivo de almacenamiento"**. Por su parte, cualquier artefacto, sea éste del tipo que sea, lo único que tiene que hacer para ser considerado un "dispositivo de almacenamiento" es ofrecer la funcionalidad o servicios especificados en la interface correspondiente, especificando en su código cómo va a llevar a cabo dichos servicios; es decir, proporcionando una implementación para la interface.

En el tema siguiente hablaremos más en profundidad del concepto de interface en Java y veremos otros

conceptos avanzados de la orientación a objetos.