

Utilización de objetos.

Caso práctico

Ada y **Juan** se han reunido para discutir sobre distintos proyectos de **BK Programación**. **Ada** le comenta a **Juan** que están teniendo algunos problemas con determinados proyectos. A menudo surgen modificaciones o mejoras en el software en el ámbito de los contratos de mantenimiento que tienen suscritos con los clientes, y realizar las modificaciones en los programas está suponiendo en muchos casos modificar el programa casi en su totalidad.

A eso se ha de sumar que las tareas de modificación son encargadas a las personas más adecuadas en ese momento, según la carga de trabajo que haya; que no tienen por qué coincidir con las personas que desarrollaron el programa. Las modificaciones en los proyectos se están retrasando, y hay algunas que deben estar listas antes de que surja el nuevo cambio de versión.

En reuniones anteriores se ha comentado la posibilidad de aumentar el precio del contrato de mantenimiento de los clientes. Se ha consultado con el equipo de comerciales y a regañadientes han aceptado un aumento que aún está por decidir, pero aún así quizás no sea suficiente. La empresa necesita mejorar el método de trabajo para reducir costes de mantenimiento del software y alcanzar la rentabilidad deseada.

1.- Introducción.



Si nos paramos a observar el mundo que nos rodea, podemos apreciar que casi todo está formado por **objetos**. Existen coches, edificios, sillas, mesas, semáforos, ascensores e incluso personas o animales. **Todos ellos pueden ser considerados objetos, con una serie de características y comportamientos.** Por ejemplo, existen coches de diferentes marcas, colores, etc. y pueden acelerar, frenar, girar, etc., o las personas tenemos diferente color de pelo, ojos, altura y peso y podemos nacer, crecer, comer, dormir, etc.

Los programas son el resultado de la búsqueda y obtención de una solución para un problema del mundo real. Pero ¿en qué medida los programas están organizados de la misma manera que el problema que tratan de solucionar? La respuesta es que muchas veces los programas se ajustan más a los términos del sistema en el que se ejecutarán que a los del propio problema.

Si redactamos los programas utilizando los mismos términos de nuestro mundo real, es decir, utilizando objetos, y no los términos del sistema o computadora donde se vaya a ejecutar, conseguiremos que éstos sean más legibles y, por tanto, más fáciles de modificar.

Esto es precisamente lo que pretende la **Programación Orientada a Objetos (POO)**, en inglés **OOP (Object Oriented Programming)**, establecer una serie de técnicas que permitan trasladar los problemas del mundo real a nuestro sistema informático. Ahora que ya conocemos la sintaxis básica de Java, es el momento de comenzar a utilizar las características orientadas a objetos de este lenguaje, y estudiar los conceptos fundamentales de este modelo de programación.

2.- Fundamentos de la Programación Orientada a Objetos.

Caso práctico



Juan cuenta con la ayuda de **María** para desarrollar la aplicación para la Clínica Veterinaria. Lo normal es pensar en tener una aplicación de escritorio para las altas y bajas de clientes y la gestión de mascotas, y una parte web para que la clínica pueda estar presente en Internet e, incluso, realizar la venta on-line de sus productos. María tiene bastante experiencia en administración de páginas web, pero para estar capacitada en el desarrollo de aplicaciones en Java, necesita adquirir conocimientos adicionales.

Juan le explica que tienen que utilizar un método de programación que les ayude a organizar los programas, a trabajar en equipo de forma que si uno de ellos tiene que dejar una parte para que se encargue el otro, que éste lo pueda retomar con el mínimo esfuerzo. Además, interesa poder reutilizar todo el código que vayan creando, para ir más rápido a la hora de programar. **Juan** le explica que si consiguen adoptar ese método de trabajo, no sólo redundará en una mejor organización para ellos, sino que ayudará a que las modificaciones en los programas sean más llevaderas de lo que lo están siendo ahora.

María asiente ante las explicaciones de **Juan**, e intuye que todo lo entenderá mejor conforme vaya conociendo los conceptos de Programación Orientada a Objetos.

De lo que realmente se trata es de que **BK Programación** invierta el menor tiempo posible en los proyectos que realice, aprovechando material elaborado con el esfuerzo ya realizado en otras aplicaciones.

Dentro de las distintas formas de hacer las cosas en programación, distinguimos dos paradigmas fundamentales:

Programación Estructurada, se crean **funciones y procedimientos** que definen las acciones a realizar, y que posteriormente forman los programas.

Programación Orientada a Objetos, considera los programas en términos de **objetos** y todo gira alrededor de ellos.

Pero ¿en qué consisten realmente estos paradigmas? Veamos estos dos modelos de programación con más detenimiento. Inicialmente se programaba aplicando las técnicas de programación tradicional, también conocidas como **Programación Estructurada**. El problema se descomponía en unidades más pequeñas hasta llegar a acciones o verbos muy simples y fáciles de codificar. Por ejemplo, en la resolución de una ecuación de primer grado, lo que hacemos es descomponer el problema en acciones más pequeñas o pasos diferenciados:

Pedir valor de los coeficientes.
Calcular el valor de la incógnita.
Mostrar el resultado.

Si nos damos cuenta, esta serie de acciones o pasos diferenciados no son otra cosa que verbos; por ejemplo el verbo pedir, calcular, mostrar, etc.

Sin embargo, la Programación Orientada a Objetos aplica de otra forma diferente la **técnica de programación "divide y vencerás"**. Este paradigma surge en un intento de salvar las dificultades que, de forma innata, posee el software. Para ello lo que hace es descomponer, en lugar de acciones, en objetos. El principal objetivo sigue siendo descomponer el problema en problemas más pequeños, que sean fáciles de manejar y mantener, fijándonos en cuál es el escenario del problema e intentando reflejarlo en nuestro programa. O sea, se trata de trasladar la visión del mundo real a nuestros programas. Por este motivo se dice que **la Programación Orientada a Objetos aborda los problemas de una forma más natural**, entendiendo como natural que está más en contacto con el mundo que nos rodea.

La Programación Estructurada se centra en el conjunto de acciones a realizar en un programa, haciendo una división de procesos y datos. La Programación Orientada a Objetos se centra en la relación que existe entre los datos y las acciones a realizar con ellos, y los encierra dentro del concepto de objeto, tratando de realizar una abstracción lo más cercana al mundo real.

La Programación Orientada a Objetos es un sistema o conjunto de reglas que nos ayudan a descomponer la aplicación en objetos. A menudo se trata de representar las entidades y objetos que nos encontramos en el mundo real mediante componentes de una aplicación. Es decir, debemos establecer una correspondencia directa entre el espacio del problema y el espacio de la solución. ¿Pero en la práctica esto qué quiere decir? Pues que a la hora de escribir un programa, nos fijaremos en los objetos involucrados, sus características comunes y las acciones que pueden realizar. Una vez localizados los objetos que intervienen en el problema real (espacio del problema), los tendremos que trasladar al programa informático (espacio de la solución). Con este planteamiento, la solución a un problema dado se convierte en una tarea sencilla y bien organizada.

AUTOEVALUACIÓN

Relaciona el término con su definición, escribiendo el número asociado a la definición en el hueco correspondiente.

Ejercicio de relacionar

Paradigma	Relación	Definición
-----------	----------	------------

Paradigma	Relación	Definición
Programación Orientada a Objetos.	□	1. Maneja funciones y procedimientos que definen las acciones a realizar.
Programación Estructurada.	□	2. Representa las entidades del mundo real mediante componentes de la aplicación.

2.1.- Conceptos.

Para entender mejor la filosofía de orientación a objetos veamos algunas características que la diferencian de las técnicas de programación tradicional.

En la Programación Estructurada, el programa estaba compuesto por un conjunto de **datos y funciones "globales"**. El término global significaba que eran accesibles por todo el programa, pudiendo ser llamados en cualquier ubicación de la aplicación. Dentro de las funciones se situaban las instrucciones del programa que manipulaban los datos. **Funciones y datos se encontraban separados y totalmente independientes**. Esto ocasionaba dos problemas principales:

Los programas se creaban y estructuraban de acuerdo con la arquitectura de la computadora donde se tenían que ejecutar.

Al estar separados los datos de las funciones, éstos eran visibles en toda la aplicación. Ello ocasionaba que cualquier modificación en los datos podía requerir la modificación en todas las funciones del programa, en correspondencia con los cambios en los datos.



En la **Programación Orientada a Objetos la situación es diferente**. La utilización de **objetos** permite un mayor nivel de **abstracción** que con la Programación Estructurada, y ofrece las siguientes diferencias con respecto a ésta:

El programador organiza su programa en **objetos**, que son **representaciones del mundo real** que están más cercanas a la forma de pensar de la gente.

Los datos, junto con las funciones que los manipulan, son parte interna de los objetos y no están accesibles al resto de los objetos. Por tanto, los cambios en los datos de un objeto sólo afectan a las funciones definidas para ese objeto, pero no al resto de la aplicación.

Todos los programas escritos bajo el paradigma orientado a Objetos se pueden escribir igualmente mediante la Programación Estructurada. Sin embargo, la Programación Orientada a Objetos es la que mayor facilidad presenta para el desarrollo de programas basados en [interfaces gráficas de usuario](#).

2.2.- Beneficios.

Según lo que hemos visto hasta ahora, **un objeto es cualquier entidad que podemos ver o apreciar**. El concepto fundamental de la Programación Orientada a Objetos son, precisamente, los objetos. Pero ¿qué beneficios aporta la utilización de objetos? Fundamentalmente la posibilidad de representar el problema en términos del mundo real, que como hemos dicho están más cercanos a nuestra forma de pensar, pero existen otra serie de ventajas como las siguientes:

Comprensión. Los conceptos del espacio del problema se hayan reflejados en el código del programa, por lo que la mera lectura del código nos describe la solución del problema en el mundo real.

Modularidad. Facilita la modularidad del código, al estar las definiciones de objetos en módulos o archivos independientes, hace que las aplicaciones estén mejor organizadas y sean más fáciles de entender.

Fácil mantenimiento. Cualquier modificación en las acciones queda automáticamente reflejada en los datos, ya que ambos están estrechamente relacionados. Esto hace que el mantenimiento de las aplicaciones, así como su corrección y modificación sea mucho más fácil. Por ejemplo, podemos querer utilizar un algoritmo más rápido, sin tener que cambiar el programa principal. Por otra parte, al estar las aplicaciones mejor organizadas, es más fácil localizar cualquier elemento que se quiera modificar y/o corregir. Esto es importante ya que se estima que los mayores costes de software no están en el proceso de desarrollo en sí, sino en el mantenimiento posterior de ese software a lo largo de su vida útil.

Seguridad. La probabilidad de cometer errores se ve reducida, ya que no podemos modificar los datos de un objeto directamente, sino que debemos hacerlo mediante las acciones definidas para ese objeto. Imaginemos un objeto lavadora. Se compone de un motor, tambor, cables, tubos, etc. Para usar una lavadora no se nos ocurre abrirla y empezar a manipular esos elementos, ya que lo más probable es que se estropee. En lugar de eso utilizamos los programas de lavado establecidos. Pues algo parecido con los objetos, no podemos manipularlos internamente, sólo utilizar las acciones que para ellos hay definidas.

Reusabilidad. Los objetos se definen como entidades reutilizables, es decir, que los programas que trabajan con las mismas estructuras de información, pueden reutilizar las definiciones de objetos empleadas en otros programas, e incluso las acciones definidas sobre ellos. Por ejemplo, podemos crear la definición de un objeto de tipo persona para una aplicación de negocios y deseamos construir a continuación otra aplicación, digamos de educación, en donde utilizamos también personas, no es necesario crear de nuevo el objeto, sino que por medio de la reusabilidad podemos utilizar el tipo de objeto persona previamente definido.

Citas para pensar

John Johnson: Primero resuelve el problema. Entonces, escribe el código.

2.3.- Características.

Cuando hablamos de Programación Orientada a Objetos, existen una serie de características que se deben cumplir. Cualquier lenguaje de programación orientado a objetos las debe contemplar. Las características más importantes del paradigma de la programación orientada a objetos son:

Abstracción. Es el proceso por el cual definimos las características más importantes de un objeto, sin preocuparnos de cómo se escribirán en el código del programa, simplemente lo definimos de forma general. En la Programación Orientada a Objetos la herramienta más importante para soportar la abstracción es la **clase**. Básicamente, una clase es un tipo de dato que agrupa las características comunes de un conjunto de objetos. Poder ver los objetos del mundo real que deseamos trasladar a nuestros programas, en términos abstractos, resulta de gran utilidad para un buen diseño del software, ya que nos ayuda a comprender mejor el problema y a tener una visión global de todo el conjunto. Por ejemplo, si pensamos en una clase **Vehículo** que agrupa las características comunes de todos ellos, a partir de dicha clase podríamos crear objetos como **Coche** y **Camión**. Entonces se dice que **Vehículo** es una abstracción de **Coche** y de **Camión**.

Modularidad. Una vez que hemos representado el escenario del problema en nuestra aplicación, tenemos como resultado un conjunto de objetos software a utilizar. Este conjunto de objetos se crean a partir de una o varias clases. Cada clase se encuentra en un archivo diferente, por lo que la modularidad nos permite modificar las características de la clase que define un objeto, sin que esto afecte al resto de clases de la aplicación.

Encapsulación. También llamada "**ocultamiento de la información**". La **encapsulación** o **encapsulamiento** es el mecanismo básico para ocultar la información de las partes internas de un objeto a los demás objetos de la aplicación. Con la encapsulación un objeto puede ocultar la información que contiene al mundo exterior, o bien restringir el acceso a la misma para evitar ser manipulado de forma inadecuada. Por ejemplo, pensemos en un programa con dos objetos, un objeto **Persona** y otro **Coche**. **Persona** se comunica con el objeto **Coche** para llegar a su destino, utilizando para ello las acciones que Coche tenga definidas como por ejemplo conducir. Es decir, **Persona** utiliza **Coche** pero no sabe cómo funciona internamente, sólo sabe utilizar sus métodos o acciones.

Jerarquía. Mediante esta propiedad podemos definir relaciones de jerarquías entre clases y objetos. Las dos jerarquías más importantes son la jerarquía "**es un**" llamada **generalización** o **especialización** y la jerarquía "**es parte de**", llamada **agregación**. Conviene detallar algunos aspectos:

La generalización o especialización, también conocida como **herencia**, permite crear una clase nueva en términos de una clase ya existente (herencia simple) o de varias clases ya existentes (herencia múltiple). Por ejemplo, podemos crear la clase **CochedeCarreras** a partir de la clase **Coche**, y así sólo tendremos que definir las nuevas características que tenga.

La agregación, también conocida como **inclusión**, permite agrupar objetos relacionados entre sí dentro de una clase. Así, un **Coche** está formado por **Motor**, **Ruedas**, **Frenos** y **Ventanas**. Se dice que **Coche** es una agregación y **Motor**, **Ruedas**, **Frenos** y **Ventanas** son agregados de **Coche**.

Polimorfismo. Esta propiedad indica la capacidad de que varias clases creadas a partir de una antecesora realicen una misma acción de forma diferente. Por ejemplo, pensemos en la clase **Animal** y la acción de expresarse. Nos encontramos que cada tipo de **Animal** puede hacerlo de manera distinta, los **Perros** ladran, los **Gatos** maullan, las **Personas** hablamos, etc. Dicho de otra manera, el polimorfismo indica la posibilidad de tomar un objeto (de tipo **Animal**, por ejemplo), e indicarle que realice la acción de expresarse, esta acción será diferente según el tipo de mamífero del que se trate.

3.- Clases y Objetos. Características de los objetos.

Caso práctico



María ha hecho un descanso de cinco minutos. Se está tomando un café y está repasando los conceptos de Programación Orientada a Objetos. Piensa que este paradigma supone un cambio de enfoque con respecto a las técnicas tradicionales. Ahora lo que necesita es ahondar en el concepto de objeto, que parece ser el eje central de este modelo de programación.

Al principio de la unidad veíamos que el mundo real está compuesto de objetos, y podemos considerar objetos casi cualquier cosa que podemos ver y sentir. Cuando escribimos un programa en un lenguaje orientado a objetos, debemos identificar cada una de las partes del problema con objetos presentes en el mundo real, para luego trasladarlos al modelo computacional que estamos creando.

En este contexto, un objeto de software es una representación de un objeto del mundo real, compuesto de una serie de características y un comportamiento específico. Pero ¿qué es más concretamente un objeto en Programación Orientada a Objetos? Veámoslo.

Un **objeto es un conjunto de datos con las operaciones definidas para ellos**. Los objetos tienen un **estado** y un **comportamiento**.

Por tanto, estudiando los objetos que están presentes en un problema podemos dar con la solución a dicho problema. Los objetos tienen unas características fundamentales que los distinguen:

Identidad. Es la característica que permite diferenciar un objeto de otro. De esta manera, aunque dos objetos sean exactamente iguales en sus atributos, son distintos entre sí. Puede ser una dirección de memoria, el nombre del objeto o cualquier otro elemento que utilice el lenguaje para distinguirlos. Por ejemplo, dos vehículos que hayan salido de la misma cadena de fabricación y sean iguales aparentemente, son distintos porque tienen un código que los identifica.

Estado. El estado de un objeto viene determinado por una serie de parámetros o atributos que lo describen, y los valores de éstos. Por ejemplo, si tenemos un objeto Coche, el estado estaría definido por atributos como Marca, Modelo, Color, Cilindrada, etc.

Comportamiento. Son las acciones que se pueden realizar sobre el objeto. En otras palabras, son los métodos o procedimientos que realiza el objeto. Siguiendo con el ejemplo del objeto

Coche, el el comportamiento serían acciones como: `arrancar()`, `parar()`, `acelerar()`, `frenar()`, etc.

3.1.- Propiedades y métodos de los objetos.

Como acabamos de ver todo objeto tiene un estado y un comportamiento. Concretando un poco más, las partes de un objeto son:

Campos, Atributos o Propiedades: Parte del objeto que almacena los datos. También se les denomina **Variables Miembro**. Estos datos pueden ser de cualquier tipo primitivo (boolean, char, int, double, etc) o ser su vez ser otro objeto. Por ejemplo, un objeto de la clase Coche puede tener un objeto de la clase Ruedas.

Métodos o Funciones Miembro: Parte del objeto que lleva a cabo las operaciones sobre los atributos definidos para ese objeto.

La idea principal es que el objeto reúne en una sola entidad los datos y las operaciones, y para acceder a los datos privados del objeto debemos utilizar los métodos que hay definidos para ese objeto.

La única forma de manipular la información del objeto es a través de sus métodos. Es decir, si queremos saber el valor de algún atributo, tenemos que utilizar el método que nos muestre el valor de ese atributo. De esta forma, evitamos que métodos externos puedan alterar los datos del objeto de manera inadecuada. **Se dice que los datos y los métodos están encapsulados dentro del objeto.**

3.2.- Interacción entre objetos.

Dentro de un programa los objetos se comunican llamando unos a otros a sus métodos. Los métodos están dentro de los objetos y describen el comportamiento de un objeto cuando recibe una llamada a uno de sus métodos. En otras palabras, cuando un objeto, `objeto1`, quiere actuar sobre otro, `objeto2`, tiene que ejecutar uno de sus métodos. Entonces se dice que el `objeto2` recibe un mensaje del `objeto1`.

Un **mensaje** es la acción que realiza un objeto. Un **método** es la función o procedimiento al que se llama para actuar sobre un objeto.

Los distintos mensajes que puede recibir un objeto o a los que puede responder reciben el nombre de **protocolo** de ese objeto.

El proceso de interacción entre objetos se suele resumir diciendo que se ha "enviado un mensaje" (hecho una petición) a un objeto, y el objeto determina "qué hacer con el mensaje" (ejecuta el código del método). Cuando se ejecuta un programa se producen las siguientes acciones:

Creación de los objetos a medida que se necesitan.

Comunicación entre los objetos mediante el envío de mensajes unos a otros, o el usuario a los objetos.

Eliminación de los objetos cuando no son necesarios para dejar espacio libre en la memoria del computador.

Los objetos se pueden comunicar entre ellos invocando a los métodos de los otros objetos.

AUTOEVALUACIÓN

Cuando un objeto, `objeto1`, ejecuta un método de otro, `objeto2`, se dice que el `objeto2` le ha mandado un mensaje al `objeto1`.

Verdadero. ☐ Falso. ☐

3.3.- Clases.

Hasta ahora hemos visto lo que son los objetos. Un programa informático se compone de muchos objetos, algunos de los cuales comparten la misma estructura y comportamiento. Si tuviéramos que definir su estructura y comportamiento objeto cada vez que queremos crear un objeto, estaríamos utilizando mucho código redundante. Por ello lo que se hace es crear una **clase**, que es una descripción de un conjunto de objetos que comparten una estructura y un comportamiento común. Y a partir de la clase, se crean tantas "copias" o "instancias" como necesitemos. Esas copias son los objetos de la clase.

Las clases constan de datos y métodos que resumen las características comunes de un conjunto de objetos. Un programa informático está compuesto por un conjunto de clases, a partir de las cuales se crean objetos que interactúan entre sí.

Si recuerdas, cuando utilizábamos los tipos de datos enumerados, los definíamos con la palabra reservada `enum` y la lista de valores entre llaves, y decíamos que un tipo de datos `enum` no es otra cosa que una especie de clase en Java. Efectivamente, todas las clases llevan su contenido entre llaves. Y una clase tiene la misma estructura que un tipo de dato enumerado, añadiéndole una serie de métodos y variables.

En otras palabras, **una clase es una plantilla o prototipo donde se especifican:**

Los **atributos** comunes a todos los objetos de la clase.

Los **métodos** que pueden utilizarse para manejar esos objetos.

Para declarar una clase en Java se utiliza la palabra reservada `class`. La declaración de una clase está compuesta por:

Cabecera de la clase. La cabecera es un poco más compleja que como aquí definimos, pero por ahora sólo nos interesa saber que está compuesta por una serie de modificadores, en este caso hemos puesto `public` que indica que es una clase pública a la que pueden acceder otras clases del programa, la palabra reservada `class` y el nombre de la clase.

Cuerpo de la clase. En él se especifican encerrados entre llaves los atributos y los métodos que va a tener la clase.

En la unidad anterior ya hemos utilizado clases, aunque aún no sabíamos su significado exacto. Por ejemplo, en los ejemplos de la unidad o en la tarea, estábamos utilizando clases, todas ellas eran clases principales, no tenían ningún atributo y el único método del que disponían era el método `main()`.

El método `main()` se utiliza para indicar que se trata de una clase principal, a partir de la cual va a empezar la ejecución del programa. Este método no aparece si la clase que estamos creando no va a ser la clase principal del programa.

4.- Utilización de objetos.

Caso práctico

María sigue fuera de la oficina. Esta noche en casa quiere repasar conceptos sobre Programación Orientada a objetos, así que aprovecha un momento para llamar a **Juan** y le comenta:

-Ya sé todo sobre objetos -le dice- sólo que...

-¿Solo qué? -añade **Juan**.

-Solo me falta saber... ¿cómo se crea un objeto?

Juan sonríe ante la pregunta de **María**, y le explica que los objetos se crean como si fuera declarando una variable más, tan sólo que el tipo de datos de dicho objeto será una clase. Tras declararlos hay que instanciarlos con la orden `new` para reservar memoria para ellos, y después ya podremos utilizarlos, refiriéndonos a su contenido con el operador punto.

-Te mando un documento por email que lo explica todo muy bien.

-¡Ah, gracias! Esta noche le echo un vistazo -añade **María**.

Una vez que hemos creado una clase, podemos crear objetos en nuestro programa a partir de esas clases.

Cuando creamos un objeto a partir de una clase se dice que hemos creado una **"instancia de la clase"**. A efectos prácticos, "objeto" e "instancia de clase" son términos similares. Es decir, nos referimos a objetos como instancias cuando queremos hacer hincapié que son de una clase particular.

Los objetos se crean a partir de las clases, y representan casos individuales de éstas.

Para entender mejor el concepto entre un objeto y su clase, piensa en un **molde de galletas y las galletas**. El molde sería la clase, que define las características del objeto, por ejemplo su forma y tamaño. Las galletas creadas a partir de ese molde son los objetos o instancias.



Otro ejemplo, imagina una clase `Persona` que reúna las características comunes de las personas (color de pelo, ojos, peso, altura, etc.) y las acciones que pueden realizar (crecer, dormir, comer, etc.). Posteriormente dentro del programa podremos crear un objeto `Trabajador` que esté basado en esa clase `Persona`. Entonces se dice que el objeto `Trabajador` es una instancia de la clase `Persona`, o que la clase `Persona` es una abstracción del objeto `Trabajador`.

Cualquier objeto instanciado de una clase contiene una copia de todos los atributos definidos en la clase. En otras palabras, lo que estamos haciendo es reservar un espacio en la memoria del ordenador para guardar sus atributos y métodos. Por tanto, cada objeto tiene una **zona de almacenamiento propia** donde se guarda toda su información, que será distinta a la de cualquier otro objeto. A las variables miembro instanciadas también se les llama **variables instancia**. De igual forma, a los métodos que manipulan esas variables se les llama **métodos instancia**.

En el ejemplo del objeto `Trabajador`, las variables instancia serían `color_de_pelo`, `peso`, `altura`, etc. Y los métodos instancia serían `crecer()`, `dormir()`, `comer()`, etc.

Autoevaluación

Las variables instancia son un tipo de variables miembro.

Verdadero. ☐ Falso. ☐

4.1.- Ciclo de vida de los objetos.

Todo programa en Java parte de una única clase, que como hemos comentado se trata de la clase principal. Esta clase ejecutará el contenido de su método **main()**, el cual será el que utilice las demás clases del programa, cree objetos y lance mensajes a otros objetos.

Las instancias u objetos tienen un tiempo de vida determinado. Cuando un objeto no se va a utilizar más en el programa, es destruido por el recolector de basura para liberar recursos que pueden ser reutilizados por otros objetos.

A la vista de lo anterior, podemos concluir que los objetos tienen un ciclo de vida, en el cual podemos distinguir las siguientes fases:

Creación, donde se hace la reserva de memoria e inicialización de atributos.

Manipulación, que se lleva a cabo cuando se hace uso de los atributos y métodos del objeto.

Destrucción, eliminación del objeto y liberación de recursos.

4.2.- Declaración.

Para la creación de un objeto hay que seguir los siguientes pasos:

Declaración: Definir el tipo de objeto.

Instanciación: Creación del objeto utilizando el operador `new`.

Pero ¿en qué consisten estos pasos a nivel de programación en Java? Veamos primero cómo declarar un objeto. Para la definición del tipo de objeto debemos emplear la siguiente instrucción:

```
nombre_objeto;
```

Donde:

`tipo` es la clase a partir de la cual se va a crear el objeto, y `nombre_objeto` es el nombre de la variable referencia con la cual nos referiremos al objeto.

Los tipos referenciados o referencias se utilizan para guardar la dirección de los datos en la memoria del ordenador.

Nada más crear una referencia, ésta se encuentra vacía. Cuando una referencia a un objeto no contiene ninguna instancia se dice que es una referencia nula, es decir, que contiene el valor `null`. Esto quiere 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".

Para entender mejor la declaración de objetos veamos un ejemplo. Cuando veíamos los tipos de datos en la Unidad 2, decíamos que Java proporciona un tipo de dato especial para los textos o cadenas de caracteres que era el tipo de dato `String`. Veíamos que realmente este tipo de dato es un **tipo referenciado** y creábamos una variable `mensaje` de ese tipo de dato de la siguiente forma:

```
String mensaje;
```

Los nombres de la clase empiezan con mayúscula, como `String`, y los nombres de los objetos con minúscula, como `mensaje`, así sabemos qué tipo de elemento utilizando.

Pues bien, `String` es realmente la clase a partir de la cual creamos nuestro objeto llamado `mensaje`.

Si observas poco se diferencia esta declaración de las declaraciones de variables que hacíamos para los tipos primitivos. Antes decíamos que `mensaje` era una variable del tipo de dato `String`. Ahora realmente vemos que `mensaje` es un objeto de la clase `String`. Pero `mensaje` aún no contiene el objeto porque no ha sido instanciado, veamos cómo hacerlo.

Por tanto, cuando creamos un objeto estamos haciendo uso de una variable que almacena la dirección de ese objeto en memoria. Esa variable es una **referencia** o un **tipo de datos referenciado**, porque no contiene el dato si no la posición del dato en la memoria del ordenador.

```
String saludo = new String ("Bienvenido a Java");  
String s; //s vale null  
s = saludo; //asignación de referencias
```

En las instrucciones anteriores, las variables `s` y `saludo` apuntan al mismo objeto de la clase `String`. Esto implica que cualquier modificación en el objeto `saludo` modifica también el objeto al que hace referencia la variable `s`, ya que realmente son el mismo.

4.3.- Instanciación.

Una vez creada la referencia al objeto, debemos crear la instancia u objeto que se va a guardar en esa referencia. Para ello utilizamos la orden **new** con la siguiente sintaxis:

```
nombre_objeto = new ([, , ..., ]);
```

Donde:

`nombre_objeto` es el nombre de la variable referencia con la cual nos referiremos al objeto,
`new` es el operador para crear el objeto,
`Constructor_de_la_Clase` es un método especial de la clase, que se llama igual que ella, y se encarga de inicializar el objeto, es decir, de dar unos valores iniciales a sus atributos, y
`par1-parN`, son parámetros que puede o no necesitar el constructor para dar los valores iniciales a los atributos del objeto.

Durante la instanciación del objeto, se reserva memoria suficiente para el objeto. De esta tarea se encarga Java y juega un papel muy importante el **recolector de basura**, que se encarga de eliminar de la memoria los objetos no utilizados para que ésta pueda volver a ser utilizada.

De este modo, para instanciar un objeto `String`, haríamos lo siguiente:

```
mensaje = new String;
```

Así estaríamos instanciando el objeto `mensaje`. Para ello utilizaríamos el operador `new` y el constructor de la clase `String` a la que pertenece el objeto según la declaración que hemos hecho en el apartado anterior. A continuación utilizamos el constructor, que se llama igual que la clase, `String`.

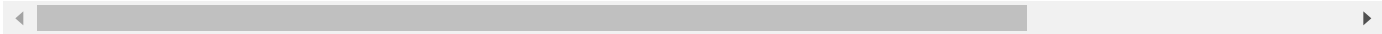
En el ejemplo anterior el objeto se crearía con la cadena vacía (`""`), si queremos que tenga un contenido debemos utilizar parámetros en el constructor, así:

```
mensaje = new String ("El primer programa");
```

Java permite utilizar la clase `String` como si de un tipo de dato primitivo se tratara, por eso no hace falta utilizar el operador `new` para instanciar un objeto de la clase `String`.

La declaración e instanciación de un objeto puede realizarse en la misma instrucción, así:

```
String mensaje = new String ("El primer programa");
```



4.4.- Manipulación.

Una vez creado e instanciado el objeto ¿cómo accedemos a su contenido? Para acceder a los atributos y métodos del objeto utilizaremos el nombre del objeto seguido del **operador punto (.)** y el nombre del atributo o método que queremos utilizar. Cuando utilizamos el operador punto se dice que estamos enviando un mensaje al objeto. La forma general de enviar un mensaje a un objeto es:

```
nombre_objeto.mensaje
```

Por ejemplo, para acceder a las variables instancia o atributos se utiliza la siguiente sintaxis:

```
nombre_objeto.atributo
```

Y para acceder a los métodos o funciones miembro del objeto se utiliza la sintaxis es:

```
nombre_objeto.método( [par1, par2, ..., parN] )
```

En la sentencia anterior `par1`, `par2`, etc. son los parámetros que utiliza el método. Aparece entre corchetes para indicar son opcionales.

Para entender mejor cómo se manipulan objetos vamos a utilizar un ejemplo. Para ello necesitamos la Biblioteca de Clases Java o API (Application Programming Interface - Interfaz de programación de aplicaciones). Uno de los paquetes de librerías o bibliotecas es `java.awt`. Este paquete contiene clases destinadas a la creación de objetos gráficos e imágenes. Vemos por ejemplo cómo crear un rectángulo.

En primer lugar instanciamos el objeto utilizando el método constructor, que se llama igual que el objeto, e indicando los parámetros correspondientes a la posición y a las dimensiones del rectángulo:

```
Rectangle rect = new Rectangle(50, 50, 150, 150);
```

Una vez instanciado el objeto rectángulo si queremos cambiar el valor de los atributos utilizamos el operador punto. Por ejemplo, para cambiar la dimensión del rectángulo:

```
rect.height=100;  
rect.width=100;
```

O bien podemos utilizar un método para hacer lo anterior:


```
rect.setSize(200, 200);
```

A continuación puedes acceder al código del ejemplo:

[Manipular objetos.](#)

4.5.- Destrucción de objetos y liberación de memoria.

Cuando un objeto deja de ser utilizado, es necesario liberar el espacio de memoria y otros recursos que poseía para poder ser reutilizados por el programa. A esta acción se le denomina **destrucción del objeto**.

En Java la destrucción de objetos corre a cargo del **recolector de basura (garbage collector)**. Es un sistema de destrucción automática de objetos que ya no son utilizados. Lo que se hace es liberar una zona de memoria que había sido reservada previamente mediante el operador `new`. Esto evita que los programadores tengan que preocuparse de realizar la liberación de memoria.

El recolector de basura se ejecuta en modo segundo plano y de manera muy eficiente para no afectar a la velocidad del programa que se está ejecutando. Lo que hace es que periódicamente va buscando objetos que ya no son referenciados, y cuando encuentra alguno lo marca para ser eliminado. Después los elimina en el momento que considera oportuno.

Justo antes de que un objeto sea eliminado por el recolector de basura, se ejecuta su método `finalize()`. Si queremos forzar que se ejecute el proceso de finalización de todos los objetos del programa podemos utilizar el método `runFinalization()` de la clase `System`. La clase `System` forma parte de la Biblioteca de Clases de Java y contiene diversas clases para la entrada y salida de información, acceso a variables de entorno del programa y otros métodos de diversa utilidad. Para forzar el proceso de finalización ejecutaríamos:

```
System.runFinalization();
```

AUTOEVALUACIÓN

Las fases del ciclo de vida de un objeto son: Creación, Manipulación y Destrucción.

Verdadero. ☐ Falso. ☐

5.- Utilización de métodos.

Caso práctico

María está contenta con lo que está aprendiendo sobre Java, básicamente se trata de ampliar sus conocimientos y con la experiencia que ella tiene no le resultará difícil ponerse a programar en poco tiempo.

Juan ha comenzado el proyecto de la Clínica Veterinaria y quiere implicarse ella también lo antes posible. Además, están los dos becarios **Ana** y **Carlos**, que se han incorporado al proyecto hace poco y hay que comenzar a pensar en tareas para ellos.

Por lo pronto, **María** continúa con el documento facilitado por **Juan**, ahora tiene que ver cómo se utilizan los métodos, aunque intuye que no va a ser algo muy diferente a las funciones y procedimientos de cualquier otro lenguaje.

Los métodos, junto con los atributos, forman parte de la estructura interna de un objeto. Los métodos contienen la declaración de variables locales y las operaciones que se pueden realizar para el objeto, y que son ejecutadas cuando el método es invocado. Se definen en el cuerpo de la clase y posteriormente son instanciados

para convertirse en **métodos instancia** de un objeto.

Para utilizar los métodos adecuadamente es conveniente conocer la estructura básica de que disponen.

Al igual que las clases, los métodos están compuestos por una **cabecera** y un **cuerpo**. La cabecera también tiene modificadores, en este caso hemos utilizado `public` para indicar que el método es público, lo cual quiere decir que le pueden enviar mensajes no sólo los métodos del objeto sino los métodos de cualquier otro objeto externo.

Dentro de un método nos encontramos el cuerpo del método que contiene el código de la acción a realizar. Las acciones que un método puede realizar son:

Inicializar los atributos del objeto

Consultar los valores de los atributos

Modificar los valores de los atributos

Llamar a otros métodos, del mismo del objeto o de objetos externos

Citas para pensar

Franklin Delano Roosevelt: En cuestión de sentido común tomar un método y probarlo. Pero lo importante es probar algo.

5.1.- Parámetros y valores devueltos.

Los métodos se pueden utilizar tanto para consultar información sobre el objeto como para modificar su estado. La información consultada del objeto se devuelve a través de lo que se conoce como **valor de retorno**, y la modificación del estado del objeto, o sea, de sus atributos, se hace mediante la **lista de parámetros**.

En general, la lista de parámetros de un método se puede declarar de dos formas diferentes:

Por valor. El valor de los parámetros no se devuelve al finalizar el método, es decir, cualquier modificación que se haga en los parámetros no tendrá efecto una vez se salga del método. Esto es así porque cuando se llama al método desde cualquier parte del programa, dicho método recibe una copia de los argumentos, por tanto cualquier modificación que haga será sobre la copia, no sobre las variables originales.

Por referencia. La modificación en los valores de los parámetros sí tienen efecto tras la finalización del método. Cuando pasamos una variable a un método por referencia lo que estamos haciendo es pasar la dirección del dato en memoria, por tanto cualquier cambio en el dato seguirá modificado una vez que salgamos del método.

En el lenguaje Java, todas las variables se pasan por valor, excepto los objetos que se pasan por referencia. En Java, la declaración de un método tiene dos restricciones:

Un método siempre tiene que devolver un valor (no hay valor por defecto). Este **valor de retorno** es el valor que devuelve el método cuando termina de ejecutarse, al método o programa que lo llamó. Puede ser un tipo primitivo, un tipo referenciado o bien el tipo void, que indica que el método no devuelve ningún valor.

Un método tiene un número fijo de argumentos. Los argumentos son variables a través de las cuales se pasa información al método desde el lugar del que se llame, para que éste pueda utilizar dichos valores durante su ejecución. Los argumentos reciben el nombre de **parámetros** cuando aparecen en la declaración del método.

El **valor de retorno** es la información que devuelve un método tras su ejecución.

Según hemos visto en el apartado anterior, la cabecera de un método se declara como sigue:

```
public tipo_de_dato_devuelto nombre_metodo (lista_de_parametros);
```

Como vemos, el tipo de dato devuelto aparece después del modificador public y se corresponde con el valor de retorno.

La lista de parámetros aparece al final de la cabecera del método, justo después del nombre, encerrados entre signos de paréntesis y separados por comas. Se debe indicar el tipo de dato de cada parámetro así:

(tipo_parámetro1 nombre_parámetro1, ..., tipo_parámetroN nombre_parámetroN)

Cuando se llame al método, se deberá utilizar el nombre del método, seguido de los argumentos que deben coincidir con la lista de parámetros.

La **lista de argumentos** en la llamada a un método debe coincidir en número, tipo y orden con los **parámetros** del método, ya que de lo contrario se produciría un error de sintaxis.



5.2.- Constructores.

¿Recuerdas cuando hablábamos de la creación e instanciación de un objeto? Decíamos que utilizábamos el operador **new** seguido del nombre de la clase y una pareja de abrir-cerrar paréntesis. Además, el nombre de la clase era realmente el constructor de la misma, y lo definíamos como un método especial que sirve para inicializar valores. En este apartado vamos a ver un poco más sobre los constructores.

Un constructor es un método especial con el mismo nombre de la clase y que no devuelve ningún valor tras su ejecución.

Cuando creamos un objeto debemos instanciarlo utilizando el constructor de la clase. Veamos la clase `Date` proporcionada por la Biblioteca de Clases de Java. Si queremos instanciar un objeto a partir de la clase `Date` tan sólo tendremos que utilizar el constructor seguido de una pareja de abrir-cerrar paréntesis:

```
Date fecha = new Date();
```

Con la anterior instrucción estamos creando un objeto `fecha` de tipo `Date`, que contendrá la fecha y hora actual del sistema.

La estructura de los constructores es similar a la de cualquier método, salvo que no tiene tipo de dato devuelto porque no devuelve ningún valor. Está formada por una cabecera y un cuerpo, que contiene la inicialización de atributos y resto de instrucciones del constructor.

El método constructor tiene las siguientes particularidades:

El constructor es invocado automáticamente en la creación de un objeto, y sólo esa vez.

Los constructores no empiezan con minúscula, como el resto de los métodos, ya que se llaman igual que la clase y las clases empiezan con letra mayúscula.

Puede haber varios constructores para una clase.

Como cualquier método, el constructor puede tener **parámetros** para definir qué valores dar a los atributos del objeto.

El **constructor por defecto** es aquél que no tiene argumentos o parámetros. Cuando creamos un objeto llamando al nombre de la clase sin argumentos, estamos utilizando el constructor por defecto.

Es necesario que toda clase tenga al menos un constructor. Si no definimos constructores para una clase, y sólo en ese caso, el compilador crea un constructor por defecto vacío, que inicializa los atributos a sus valores por defecto, según del tipo que sean: `0` para los tipos numéricos, `false` para los boolean y `null` para los tipo carácter y las referencias. Dicho constructor lo que hace es llamar al constructor sin argumentos de la superclase (clase de la cual hereda); si la superclase no tiene constructor sin argumentos se produce un error de compilación.

Cuando definimos constructores personalizados, el constructor por defecto deja de existir, y si no definimos nosotros un constructor sin argumentos cuando intentemos utilizar el constructor por defecto nos dará un error de compilación.

5.3.- El operador this.

Los constructores y métodos de un objeto suelen utilizar el operador `this`. Este operador sirve para referirse a los atributos de un objeto cuando estamos dentro de él. Sobre todo se utiliza cuando existe ambigüedad entre el nombre de un parámetro y el nombre de un atributo, entonces en lugar del nombre del atributo solamente escribiremos `this.nombre_atributo`, y así no habrá duda de a qué elemento nos estamos refiriendo.

Vamos a ilustrar mediante un ejemplo la utilización de objetos y métodos, así como el uso de parámetros y el operador `this`. Aunque la creación de clases la veremos en las siguientes unidades, en este ejercicio creamos una pequeña clase para que podamos instanciar el objeto con el que vamos a trabajar.

Las clases se suelen representar como un rectángulo, y dentro de él se sitúan los atributos y los métodos de dicha clase.

En la imagen, la clase `Pajaro` está compuesta por tres atributos, uno de ellos el nombre y otros dos que indican la posición del ave, `posX` y `posY`. Tiene dos métodos constructores y un método `volar()`. Como sabemos, los métodos constructores reciben el mismo nombre de la clase, y puede haber varios para una misma clase, dentro de ella se diferencian unos de otros por los parámetros que utilizan.

Ejercicio resuelto

Dada una clase principal llamada `Pajaro`, se definen los atributos y métodos que aparecen en la imagen. Los métodos realizan las siguientes acciones:

`pajaro()`. Constructor por defecto. En este caso, el constructor por defecto no contiene ninguna instrucción, ya que Java inicializa de forma automática las variables miembro, si no le damos ningún valor.

`pajaro(String nombre, int posX, int posY)`. Constructor que recibe como argumentos una cadena de texto y dos enteros para inicializar el valor de los atributos.

`volar(int posX, int posY)`. Método que recibe como argumentos dos enteros: `posX` y `posY`, y devuelve un valor de tipo `double` como resultado, usando la palabra clave `return`. El valor devuelto es el resultado de aplicar un desplazamiento de acuerdo con la siguiente fórmula:

Diseña un programa que utilice la clase `Pajaro`, cree una instancia de dicha clase y ejecute sus métodos.

5.4.- Métodos estáticos.

Cuando trabajábamos con cadenas de caracteres utilizando la clase `String`, veíamos las operaciones que podíamos hacer con ellas: obtener longitud, comparar dos cadenas de caracteres, cambiar a mayúsculas o minúsculas, etc. Pues bien, sin saberlo estábamos utilizando métodos estáticos definidos por Java para la clase `String`. Pero ¿qué son los métodos estáticos? Veámoslo.

Los **métodos estáticos** son aquellos métodos definidos para una clase que se pueden usar directamente, sin necesidad de crear un objeto de dicha clase. También se llaman **métodos de clase**.

Para llamar a un método estático utilizaremos:

El nombre del método, si lo llamamos desde la misma clase en la que se encuentra definido.
El nombre de la clase, seguido por el operador punto (.) más el nombre del método estático, si lo llamamos desde una clase distinta a la que se encuentra definido:

```
nombre_clase.nombre_metodo_estatico
```

El nombre del objeto, seguido por el operador punto (.) más el nombre del método estático.
Utilizaremos esta forma cuando tengamos un objeto instanciado de la clase en la que se encuentra definido el método estático, y no podamos utilizar la anterior:

```
nombre_objeto.nombre_metodo_estatico
```

Los métodos estáticos no afectan al estado de los objetos instanciados de la clase (variables instancia), y **suelen utilizarse para realizar operaciones comunes a todos los objetos de la clase**. Por ejemplo, si necesitamos contar el número de objetos instanciados de una clase, podríamos utilizar un método estático que fuera incrementando el valor de una variable entera de la clase conforme se van creando los objetos.

En la Biblioteca de Clases de Java existen muchas clases que contienen métodos estáticos. Pensemos en las clases que ya hemos utilizado en unidades anteriores, como hemos comentado la clase `String` con todas las operaciones que podíamos hacer con ella y con los objetos instanciados a partir de ella. O bien la clase `Math`, para la conversión de tipos de datos. Todos ellos son métodos estáticos que la API de Java define para esas clases. Lo importante es que tengamos en cuenta que al tratarse de métodos estáticos, para utilizarlos no necesitamos crear un objeto de dichas clases.

AUTOEVALUACIÓN





Los métodos estáticos, también llamados métodos instancia, suelen utilizarse para realizar operaciones comunes a todos los objetos de la clase.

Verdadero. ☐ Falso. ☐

6. Desarrollo de clases.

Llegados a este punto de la unidad, es momento de profundizar en las diferentes estructuras de almacenamiento que permiten el desarrollo de clases.

Concretamente en este apartado del tema veremos:

- Estructura y miembros de una clase.

- Atributos

- Métodos.

- Encapsulación, control de acceso y visibilidad.

- Constructores.

Partes de la temática que se detallan en subapartados a continuación.

6.1.- Estructura y miembros de una clase.

Caso práctico

María empieza a tener bastante más claro en qué consiste una clase y cuál es la diferencia respecto a un objeto. Es el momento de empezar a crear en Java algunas de las clases que han estado pensando que podrían ser útiles para su aplicación. Para ello es necesario saber cómo se declara una clase en un lenguaje de programación determinado. Los becarios **Ana** y **Carlos** intuyen que van a comenzar a ver cómo está hecha una clase "por dentro". Parece ser que es el momento de empezar a tomar notas:

-De acuerdo, ya hemos diseñado algunas de las clases que queremos para nuestra aplicación. Pero, ¿cómo escribimos eso en Java? ¿Cómo declaramos o definimos una clase en Java? ¿Qué palabras reservadas hay que utilizar? ¿Qué partes tiene esa definición? -pregunta **María** con interés.


-Bien, es el momento de ver cómo es la estructura de una clase y cómo podemos escribirla en Java para luego poder fabricar objetos que pertenezcan a esa clase -le responde **Juan**.

En unidades anteriores ya se indicó que para declarar una clase en Java se usa la palabra reservada `class`. En la declaración de una clase vas a encontrar:

Cabecera de la clase. Compuesta por una serie de modificadores de acceso, la palabra reservada `class` y el nombre de la clase.

Cuerpo de la clase. En él se especifican los distintos miembros de la clase: **atributos** y **métodos**. Es decir, el contenido de la clase.

Como puedes observar, 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**.



Toda definición de una clase consta de cabecera y cuerpo. En la cabecera se definen los atributos de los objetos que se crearán a partir de esa clase y en el cuerpo estarán definidos los distintos métodos disponibles para manipular esos objetos. ¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

6.1.1.- Declaración de una clase.

La declaración de una clase en Java tiene la siguiente estructura general:

```
[modificadores] class [herencia] [interfaces] { // Cabecera de la clase
    // Cuerpo de la clase
    Declaración de los atributos
    Declaración de los métodos
}
```

Un ejemplo básico pero completo podría ser:

[Código de la clase Punto.](#) (1 KB)

En este caso se trata de una clase muy sencilla en la que el cuerpo de la clase (el área entre las llaves) contiene el código y las declaraciones necesarias para que los objetos que se construyan (basándose en esta clase) puedan funcionar apropiadamente en un programa (declaraciones de atributos para contener el estado del objeto y métodos que implementen el comportamiento de la clase y los objetos creados a partir de ella).

Si te fijas en los distintos programas que se han desarrollado en los ejemplos de las unidades anteriores, podrás observar que cada uno de esos programas era en sí mismo una clase Java: se declaraban con la palabra reservada `class` y contenían algunos atributos (variables) así como algunos métodos (como mínimo el método `main`).

En el ejemplo anterior hemos visto lo mínimo que se tiene que indicar en la **cabecera de una clase** (el nombre de la clase y la palabra reservada `class`). Se puede proporcionar bastante más información mediante modificadores y otros indicadores como por ejemplo el nombre de su **superclase** (si es que esa clase hereda de otra), si implementa algún **interfaz** y algunas cosas más que irás aprendiendo poco a poco.

A la hora de implementar una clase Java (escribirla en un archivo con un editor de textos o con alguna herramienta integrada como por ejemplo **Netbeans** o **Eclipse**) debes tener en cuenta:

Por convenio, se ha decidido que en lenguaje Java los nombres de las clases deben de **empezar por una letra mayúscula**. Así, cada vez que observes en el código una palabra con la primera letra en mayúscula sabrás que se trata de una clase sin necesidad de tener que buscar su declaración. Además, **si el nombre de la clase está formado por varias palabras, cada una de ellas también tendrá su primera letra en mayúscula**. Siguiendo esta recomendación,

algunos ejemplos de nombres de clases podrían ser: Recta, Circulo, Coche, CocheDeportivo, Jugador, JugadorFutbol, AnimalMarino, AnimalAcuatico, etc.

El archivo en el que se encuentra una clase Java debe tener el mismo nombre que esa clase si queremos poder utilizarla desde otras clases que se encuentren fuera de ese archivo (**clase principal del archivo**).

Tanto la definición como la implementación de una clase se incluye en el mismo archivo (archivo ".java"). En otros lenguajes como por ejemplo C++, definición e implementación podrían ir en archivos separados (por ejemplo en C++, serían sendos archivos con extensiones ".h" y ".cpp").

Para saber más

Si quieres ampliar un poco más sobre este tema puedes echar un vistazo a los tutoriales de iniciación de Java en el sitio web de Oracle (en inglés):

[Java Classes.](#)

6.1.2.- Cabecera de una clase.

En general, la declaración de una clase puede incluir los siguientes elementos y en el siguiente orden:

Modificadores tales como `public`, `abstract` o `final`.

El nombre de la clase (con la primera letra de cada palabra en mayúsculas, por convenio).

El nombre de su **clase padre (superclase)**, si es que se especifica, precedido por la palabra reservada `extends` ("extiende" o "hereda de").

Una lista separada por comas de **interfaces** que son implementadas por la clase, precedida por la palabra reservada `implements` ("implementa").

El cuerpo de la clase, encerrado entre llaves `{}`.

La sintaxis completa de una cabecera (los cuatro primeros puntos) queda de la forma:

```
[modificadores]
class [extends ][implements
] [[implements ] ...] {
```

En el ejemplo anterior de la clase `Punto` teníamos la siguiente cabecera:

```
class Punto {
```

En este caso no hay **modificadores**, ni indicadores de **herencia**, ni implementación de **interfaces**. Tan solo la palabra reservada `class` y el nombre de la clase. Es lo mínimo que puede haber en la cabecera de una clase.

La **herencia** y las **interfaces** las verás más adelante. Vamos a ver ahora cuáles son los **modificadores** que se pueden indicar al crear la clase y qué efectos tienen. Los **modificadores de clase** son:

```
[public] [final | abstract]
```

Veamos qué significado tiene cada uno de ellos:

Modificador `public`. Indica que la clase es visible (se pueden crear objetos de esa clase) desde cualquier otra clase. Es decir, desde cualquier otra parte del programa. Si no se especifica este modificador, la clase sólo podrá ser utilizada desde clases que estén en el mismo **paquete**. El

concepto de paquete lo veremos más adelante. Sólo puede haber una clase `public` (clase principal) en un archivo `.java`. El resto de clases que se definan en ese archivo no serán públicas. Modificador `abstract`. Indica que la clase es **abstracta**. Una clase abstracta no es instanciable. Es decir, no es posible crear objetos de esa clase (habrá que utilizar clases que hereden de ella). En este momento es posible que te parezca que no tenga sentido que esto pueda suceder (si no puedes crear objetos de esa clase, ¿para qué la quieres?), pero puede resultar útil a la hora de crear una jerarquía de clases. Esto lo verás también más adelante al estudiar el concepto de **herencia**.

Modificador `final`. Indica que no podrás crear clases que hereden de ella. También volverás a este modificador cuando estudies el concepto de **herencia**. Los modificadores `final` y `abstract` son excluyentes (sólo se puede utilizar uno de ellos).

Todos estos modificadores y palabras reservadas las iremos viendo poco a poco, así que no te preocupes demasiado por intentar entender todas ellas en este momento.

En el ejemplo anterior de la clase `Punto` tendríamos una clase que sería sólo visible (utilizable) desde el mismo paquete en el que se encuentra la clase (modificador de acceso por omisión o de paquete, o `package`). Desde fuera de ese paquete no sería visible o accesible. Para poder utilizarla desde cualquier parte del código del programa bastaría con añadir el atributo `public`: `public class Punto`.



Autoevaluación

Si queremos poder instanciar objetos de una clase desde cualquier parte de un programa, ¿qué modificador o modificadores habrá que utilizar en su declaración?

- ☐ `private`.
- ☐ `public`.
- ☐ `abstract`.
- ☐ Ninguno de los anteriores.

6.1.3.- Cuerpo de una clase.

Como ya has visto anteriormente, el cuerpo de una clase se encuentra encerrado entre llaves y contiene la declaración e implementación de sus miembros. Los miembros de una clase pueden ser:

Atributos, que especifican los datos que podrá contener un objeto de la clase.

Métodos, que implementan las acciones que se podrán realizar con un objeto de la clase.

Una clase puede no contener en su declaración atributos o métodos, pero debe de contener al menos uno de los dos (la clase no puede ser vacía).

En el ejemplo anterior donde se definía una clase `Punto`, tendríamos los siguientes atributos:

Atributo `x`, de tipo `int`.

Atributo `y`, de tipo `int`.

Es decir, dos valores de tipo entero. Cualquier objeto de la clase `Punto` que sea creado almacenará en su interior dos números enteros (`x` e `y`). Cada objeto diferente de la clase `Punto` contendrá sendos valores `x` e `y`, que podrán coincidir o no con el contenido de otros objetos de esa misma clase `Punto`.

Por ejemplo, si se han declarado varios objetos de tipo `Punto`:

```
Punto p1, p2, p3;
```

Sabremos que cada uno de esos objetos `p1`, `p2` y `p3` contendrán un par de coordenadas (`x`, `y`) que definen el estado de ese objeto. Puede que esos valores coincidan con los de otros objetos de tipo `Punto`, o puede que no, pero en cualquier caso serán objetos diferentes creados a partir del mismo molde (de la misma clase).

Por otro lado, la clase `Punto` también definía una serie de métodos:

```
* int obtenerX () { return x; }  
* int obtenerY() { return y;}  
* void establecerX (int vx) { x= vx; };  
* void establecerY (int vy) { y= vy; };
```

Cada uno de esos métodos puede ser llamado desde cualquier objeto que sea una instancia de la clase `Punto`. Se trata de operaciones que permiten manipular los datos (atributos) contenidos en el objeto bien para calcular otros datos o bien para modificar los propios atributos.

AUTOEVALUACIÓN

Si disponemos de varios objetos que han sido creados a partir de la misma definición de clase, en realidad tendremos un único objeto, pues hacen referencia a un mismo tipo de clase (plantilla). ¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

6.1.4.- Miembros estáticos o de clase.

Cada vez que se produce una instancia de una clase (es decir, se crea un objeto de esa clase), se desencadenan una serie de procesos (construcción del objeto) que dan lugar a la creación en memoria de un espacio físico que constituirá el objeto creado. De esta manera cada objeto tendrá sus propios miembros a imagen y semejanza de la plantilla propuesta por la clase.

Por otro lado, podrás encontrarte con ocasiones en las que determinados miembros de la clase (atributos o métodos) no tienen demasiado sentido como partes del objeto, sino más bien como partes de la clase en sí (partes de la plantilla, pero no de cada instancia de esa plantilla). Por ejemplo, si creamos una clase Coche y quisiéramos disponer de un atributo con el nombre de la clase (un atributo de tipo `String` con la cadena "Coche"), no tiene mucho sentido replicar ese atributo para todos los objetos de la clase Coche, pues para todos va a tener siempre el mismo valor (la cadena "Coche"). Es más, ese atributo puede tener sentido y existencia al margen de la existencia de cualquier objeto de tipo Coche. Podría no haberse creado ningún objeto de la clase Coche y sin embargo seguiría teniendo sentido poder acceder a ese atributo de nombre de la clase, pues se trata en efecto de un atributo de la propia clase más que de un atributo de cada objeto instancia de la clase.

Para poder definir miembros estáticos en Java se utiliza el modificador `static`. Los miembros (tanto atributos como métodos) declarados utilizando este modificador son conocidos como miembros estáticos o miembros de clase. A continuación vas a estudiar la creación y utilización de atributos y métodos. En cada caso verás cómo declarar y usar atributos estáticos y métodos estáticos.

6.2.- Atributos.

Caso práctico

María está entusiasmada con las posibilidades que le ofrece el concepto de clase para poder crear cualquier tipo de objeto que a ella se le ocurra. Ya ha aprendido cómo declarar la cabecera de una clase en Java y ha estado probando con algunos de los ejemplos que le ha proporcionado **Juan**. Entiende más o menos cómo funcionan algunos de los modificadores de la clase, pero ha visto que el cuerpo es algo más complejo: ya no se trata de una simple línea de código como en el caso de la cabecera. Ahora tiene un conjunto de líneas de código que parecen hasta cierto punto un programa en pequeño. Puede encontrarse con declaraciones de variables, estructuras de control, realización de cálculos, etc.

Lo primero que **María** ha observado es que al principio suele haber algunas declaraciones de variables:

-¿Estas declaraciones de variables son lo que hemos llamado atributos?-le pregunta a **Juan**.

-Así es -contesta **Ada**, que en ese momento acaba de entrar por la puerta con **Ana** y con **Carlos**.

-Ahora que estáis todos juntos, creo que ha llegado el momento que os explique algunas cosas acerca de los miembros de una clase. Vamos a empezar por los atributos.

Los **atributos** constituyen la estructura interna de los objetos de una clase. Se trata del conjunto de datos que los objetos de una determinada clase almacenan cuando son creados. Es decir es como si fueran variables cuyo ámbito de existencia es el objeto dentro del cual han sido creadas. Fuera del objeto esas variables no tienen sentido y si el objeto deja de existir, esas variables también deberían hacerlo (proceso de destrucción del objeto). Los atributos a veces también son conocidos con el nombre de **variables miembro** o **variables de objeto**.

Los atributos pueden ser de cualquier tipo de los que pueda ser cualquier otra variable en un programa en Java: desde tipos elementales como `int`, `boolean` o `float` hasta tipos referenciados como `arrays`, `Strings` u objetos.

Además del tipo y del nombre, la declaración de un atributo puede contener también algunos modificadores (como por ejemplo `public`, `private`, `protected` o `static`). Por ejemplo, en el caso de la clase `Punto` que habíamos definido en el apartado anterior podrías haber declarado sus atributos como:

```
public int x;  
public int y;
```

De esta manera estarías indicando que ambos atributos son públicos, es decir, accesibles por cualquier parte del código programa que tenga acceso a un objeto de esa clase.

Como ya verás más adelante al estudiar el concepto de **encapsulación**, lo normal es declarar todos los atributos (o al menos la mayoría) como privados (`private`) de manera que si se desea acceder o manipular algún atributo se tenga que hacer a través de los métodos proporcionados por la clase.

AUTOEVALUACIÓN

Dado que normalmente se pretende encapsular el contenido de un objeto en su interior y permitir el acceso a sus atributos únicamente a través de los métodos, los atributos de una clase suelen declararse con el modificador `public`. ¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

6.2.1.- Declaración de atributos.

La sintaxis general para la declaración de un atributo en el interior de una clase es:

```
[modificadores] ;
```

Ejemplos:

```
int x;  
public int elementoX, elementoY;  
private int x, y, z;  
static double descuentoGeneral;  
final bool casado;
```

Te suena bastante, ¿verdad? La declaración de los atributos en una clase es exactamente igual a la declaración de cualquier variable tal y como has estudiado en las unidades anteriores y similar a como se hace en cualquier lenguaje de programación. Es decir mediante la indicación del tipo y a continuación el nombre del atributo, pudiéndose declarar varios atributos del mismo tipo mediante una lista de nombres de atributos separada por comas (exactamente como ya has estudiado al declarar variables).

La declaración de un **atributo** (o **variable miembro** o **variable de objeto**) consiste en la declaración de una variable que únicamente existe en el interior del objeto y por tanto su vida comenzará cuando el objeto comience a existir (el objeto sea creado). Esto significa que cada vez que se cree un objeto se crearán tantas variables como atributos contenga ese objeto en su interior (definidas en la clase, que es la plantilla o "molde" del objeto). Todas esas variables estarán encapsuladas dentro del objeto y sólo tendrán sentido dentro de él.

En el ejemplo que estamos utilizando de objetos de tipo **Punto** (instancias de la clase **Punto**), cada vez que se cree un nuevo **Punto p1**, se crearán sendos atributos **x**, **y** de tipo **int** que estarán en el interior de ese punto **p1**. Si a continuación se crea un nuevo objeto **Punto p2**, se crearán otros dos nuevos atributos **x**, **y** de tipo **int** que estarán esta vez alojados en el interior de **p2**. Y así sucesivamente...

Dentro de la declaración de un atributo puedes encontrar tres partes:

Modificadores. Son palabras reservadas que permiten modificar la utilización del atributo (indicar el control de acceso, si el atributo es constante, si se trata de un atributo de clase, etc.). Los iremos viendo uno a uno.

Tipo. Indica el tipo del atributo. Puede tratarse de un tipo primitivo (**int**, **char**, **bool**, **double**, etc) o bien de uno referenciado (objeto, array, etc.).

Nombre. Identificador único para el nombre del atributo. Por convenio se **suelen utilizar las minúsculas**. En caso de que se trate de un identificador que contenga varias palabras, **a partir de la segunda palabra se suele poner la letra de cada palabra en mayúsculas**. Por ejemplo:

primerValor, valor, puertalZquierda, cuartoTrasero, equipoVecendor, sumaTotal, nombreCandidatoFinal, etc. Cualquier identificador válido de Java será admitido como nombre de atributo válido, pero es importante seguir este convenio para facilitar la legibilidad del código (todos los programadores de Java lo utilizan).

Como puedes observar, los atributos de una clase también pueden contener modificadores en su declaración (como sucedía al declarar la propia clase). Estos modificadores permiten indicar cierto comportamiento de un atributo a la hora de utilizarlo. Entre los modificadores de un atributo podemos distinguir:

Modificadores de acceso. Indican la forma de acceso al atributo desde otra clase. Son modificadores excluyentes entre sí. Sólo se puede poner uno.

Modificadores de contenido. No son excluyentes. Pueden aparecer varios a la vez.

Otros modificadores: `transient` y `volatile`. El primero se utiliza para indicar que un atributo es transitorio (no persistente) y el segundo es para indicar al compilador que no debe realizar optimizaciones sobre esa variable. Es más que probable que no necesites utilizarlos en este módulo.

Aquí tienes la sintaxis completa de la declaración de un atributo teniendo en cuenta la lista de todos los modificadores e indicando cuáles son incompatibles unos con otros:

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

Vamos a estudiar con detalle cada uno de ellos.



6.2.2.- Modificadores de acceso.

Los modificadores de acceso disponibles en Java para un atributo son:

Modificador de acceso **por omisión** (o **de paquete**). Si no se indica ningún modificador de acceso en la declaración del atributo, se utilizará este tipo de acceso. Se permitirá el acceso a este atributo desde todas las clases que estén dentro del **mismo paquete** (**package**) que esta clase (la que contiene el atributo que se está declarando). No es necesario escribir ninguna

palabra reservada. Si no se pone nada se supone se desea indicar este modo de acceso.

Modificador de acceso **public**. Indica que **cualquier clase** (por muy ajena o lejana que sea) tiene acceso a ese atributo. No es muy habitual declarar atributos públicos (**public**).

Modificador de acceso **private**. Indica que sólo se puede acceder al atributo desde **dentro de la propia clase**. El atributo estará "oculto" para cualquier otra zona de código fuera de la clase en la que está declarado el atributo. Es lo opuesto a lo que permite **public**.

Modificador de acceso **protected**. En este caso se permitirá acceder al atributo desde cualquier **subclase** (lo verás más adelante al estudiar la **herencia**) de la clase en la que se encuentre declarado el atributo, y también desde las clases del **mismo paquete**.

A continuación puedes observar un resumen de los distintos niveles accesibilidad que permite cada modificador:

Cuadro de niveles accesibilidad a los atributos de una clase.

	Misma clase	Subclase	Mismo paquete	Otro paquete
Sin modificador (paquete)	Sí		Sí	
public	Sí	Sí	Sí	Sí
private	Sí			
protected	Sí	Sí	Sí	

¡Recuerda que los modificadores de acceso son excluyentes! Sólo se puede utilizar uno de ellos en la declaración de un atributo.

Ejercicio resuelto

Imagina que quieres escribir una clase que represente un **rectángulo** en el plano. Para ello has pensado en los siguientes atributos:

Atributos **x1**, **y1**, que representan la coordenadas del vértice inferior izquierdo del rectángulo. Ambos de tipo `double` (números reales).

Atributos **x2**, **y2**, que representan las coordenadas del vértice superior derecho del rectángulo. También de tipo `double` (números reales).

Con estos dos puntos (x1, y1) y (x2, y2) se puede definir perfectamente la ubicación de un rectángulo en el plano.

Escribe una clase que contenga todos esos atributos teniendo en cuenta que queremos que sea una clase visible desde cualquier parte del programa y que sus atributos sean también accesibles desde cualquier parte del código.

6.2.3.- Modificadores de contenido.

Los modificadores de contenido **no son excluyentes** (pueden aparecer varios para un mismo atributo). Son los siguientes:

Modificador `static`. Hace que el atributo sea común para todos los objetos de una misma clase. Es decir, todas las clases compartirán ese mismo atributo con el mismo valor. Es un caso de miembro estático o miembro de clase: un **atributo estático o atributo de clase o variable de clase**.

Modificador `final`. Indica que el atributo es una **constante**. Su valor no podrá ser modificado a lo largo de la vida del objeto. Por convenio, el nombre de los **atributos constantes** (`final`) se escribe con **todas las letras en mayúsculas**.

En el siguiente apartado sobre atributos estáticos verás un ejemplo completo de un atributo estático (`static`). Veamos ahora un ejemplo de atributo constante (`final`).

Imagina que estás diseñando un conjunto de clases para trabajar con expresiones geométricas (figuras, superficies, volúmenes, etc.) y necesitas utilizar muy a menudo la constante pi con abundantes cifras significativas, por ejemplo, 3.14159265. Utilizar esa constante literal muy a menudo puede resultar tedioso además de poco operativo (imagina que el futuro hubiera que cambiar la cantidad de cifras significativas). La idea es declararla una sola vez, asociarle un nombre simbólico (un identificador) y utilizar ese identificador cada vez que se necesite la constante. En tal caso puede resultar muy útil declarar un atributo `final` con el valor 3.14159265 dentro de la clase en la que se considere oportuno utilizarla. El mejor identificador que podrías utilizar para ella será probablemente el propio nombre de la constante (y en mayúsculas, para seguir el convenio de nombres), es decir, `PI`.

Así podría quedar la declaración del atributo:

```
class claseGeometria {  
    // Declaración de constantes  
    public final float PI= 3.14159265;  
    ...  
}
```

AUTOEVALUACIÓN

¿Con qué modificador puede indicarse en Java que un atributo es constante?

- ☐ Con el modificador `constant`.
- ☐ Con el modificador `starter`.
- ☐ Con el modificador `final`.
- ☐



Con el modificador `static`.

6.2.4.- Atributos estáticos.

Como ya has visto, el modificador `static` hace que el atributo sea común (el mismo) para todos los objetos de una misma clase. En este caso sí podría decirse que la existencia del atributo no depende de la existencia del objeto, sino de la propia clase y por tanto sólo habrá uno, independientemente del número de objetos que se creen. El atributo será siempre el mismo para todos los objetos y tendrá un valor único independientemente de cada objeto. Es más, aunque no exista ningún objeto de esa clase, el atributo sí existirá y podrá contener un valor (pues se trata de un

atributo de la clase más que del objeto).

Uno de los ejemplos más habituales (y sencillos) de atributos estáticos o de clase es el de un **contador** que indica el número de objetos de esa clase que se han ido creando. Por ejemplo, en la clase de ejemplo `Punto` podrías incluir un atributo que fuera ese contador para llevar un registro del número de objetos de la clase `Punto` que se van construyendo durante la ejecución del programa.

Otro ejemplo de atributo estático (y en este caso también constante) que también se ha mencionado anteriormente al hablar de miembros estáticos era disponer de un atributo **nombre**, que contuviera un `String` con el nombre de la clase. Nuevamente ese atributo sólo tiene sentido para la clase, pues habrá de ser compartido por todos los objetos que sean de esa clase (es el nombre de la clase a la que pertenecen los objetos y por tanto siempre será la misma e igual para todos, no tiene sentido que cada objeto de tipo `Punto` almacene en su interior el nombre de la clase, eso lo debería hacer la propia clase).

```
class Punto {  
    // Coordenadas del punto  
    private int x, y;  
    // Atributos de clase: cantidad de puntos creados hasta el momento  
    public static cantidadPuntos;  
    public static final nombre;
```

Obviamente, para que esto funcione como estás pensando, también habrá que escribir el código necesario para que cada vez que se cree un objeto de la clase `Punto` se incremente el valor del atributo `cantidadPuntos`. Volverás a este ejemplo para implementar esa otra parte cuando estudies los constructores.

Ejercicio resuelto

Ampliar el ejercicio anterior del rectángulo incluyendo los siguientes atributos:

Atributo **numRectangulos**, que almacena el número de objetos de tipo rectángulo creados hasta el momento.

Atributo **nombre**, que almacena el nombre que se le quiera dar a cada rectángulo.

Atributo **nombreFigura**, que almacena el nombre de la clase, es decir, "Rectángulo".

Atributo **PI**, que contiene el nombre de la constante PI con una precisión de cuatro cifras decimales.

No se desea que los atributos **nombre** y **numRectangulos** puedan ser visibles desde fuera de la clase. Y además se desea que la clase sea accesible solamente desde su propio paquete.

6.3.- Métodos.

Caso práctico

María ya ha estado utilizando **métodos** para poder manipular algunos de los objetos que han creado en programas básicos de prueba. En el proyecto de la **Clínica Veterinaria** en el que está trabajando junto con **Juan** van a tener que crear bastantes tipos de objetos (clases) que representen el sistema de información que quieren modelar y automatizar. Ya han pensando y definido muchos de los atributos que van a tener esas clases.

Ahora necesitan empezar a definir qué tipos de acciones se van a poder realizar sobre la información que contenga cada clase o familia de objetos:

-Ya tengo pensadas algunas de las acciones que van a ser necesarias para manipular algunas de las clases que hemos planteado -Le dice **María** a **Juan**.

-Muy bien. Entonces es el momento de empezar a definir métodos.

-Perfecto. ¿Y cómo lo hacemos? Cuando hemos utilizado objetos de clases ya incorporadas en el lenguaje, simplemente he utilizado sus métodos, pero aún no he declarado ninguno.

-No te preocupes, vamos a ver algunos ejemplos de declaración, implementación y utilización de métodos de una clase. Verás cómo es mucho más sencillo de lo que piensas.

Como ya has visto anteriormente, los **métodos** son las herramientas que nos sirven para definir el comportamiento de un objeto en sus interacciones con otros objetos. Forman parte de la estructura interna del objeto junto con los atributos.

En el proceso de declaración de una clase que estás estudiando ya has visto cómo escribir la cabecera de la clase y cómo especificar sus atributos dentro del cuerpo de la clase. Tan solo falta ya declarar los métodos, que estarán también en el interior del cuerpo de la clase junto con los atributos.

Los métodos suelen declararse después de los atributos. Aunque atributos y métodos pueden aparecer mezclados por todo el interior del cuerpo de la clase es aconsejable no hacerlo para mejorar la **claridad** y la **legibilidad** del código. De ese modo, cuando echemos un vistazo rápido al contenido de una clase, podremos ver rápidamente los atributos al principio (normalmente ocuparán menos líneas de código y serán fáciles de reconocer) y cada uno de los métodos inmediatamente después. Cada método puede ocupar un número de líneas de código más o menos grande en función de la complejidad del proceso que pretenda implementar.

Los métodos representan la **interfaz** de una clase. Son la forma que tienen otros objetos de comunicarse con un objeto determinado solicitándole cierta información o pidiéndole que lleve a cabo

una determinada acción. Este modo de programar, como ya has visto en unidades anteriores, facilita mucho la tarea al desarrollador de aplicaciones, pues le permite abstraerse del contenido de las clases haciendo uso únicamente del interfaz (métodos).

AUTOEVALUACIÓN

¿Qué elementos forman la interfaz de un objeto?

- ☐ Los atributos del objeto.
- ☐ Las variables locales de los métodos del objeto.
- ☐ Los métodos.
- ☐ Los atributos estáticos de la clase.

6.3.1.- Declaración de un método.

La definición de un método se compone de dos partes:

Cabecera del método, que contiene el nombre del método junto con el tipo devuelto, un conjunto de posibles modificadores y una lista de parámetros.

Cuerpo del método, que contiene las sentencias que implementan el comportamiento del método (incluidas posibles sentencias de declaración de variables locales).

Los **elementos mínimos** que deben aparecer en la declaración de un método son:

El tipo devuelto por el método.

El nombre del método.

Los paréntesis.

El cuerpo del método entre llaves: { }.

Por ejemplo, en la clase `Punto` que se ha estado utilizando en los apartados anteriores podrías encontrar el siguiente método:

```
int obtenerX ()
{
    // Cuerpo del método
    ...
}
```

Donde:

El tipo devuelto por el método es `int`.

El nombre del método es `obtenerX`.

No recibe ningún parámetro: aparece una lista vacía entre paréntesis: `()`.

El cuerpo del método es todo el código que habría encerrado entre llaves: { }.

Dentro del cuerpo del método podrás encontrar declaraciones de variables, sentencias y todo tipo de estructuras de control (bucles, condiciones, etc.) que has estudiado en los apartados anteriores.

Ahora bien, la declaración de un método puede incluir algunos elementos más. Vamos a estudiar con detalle cada uno de ellos.

6.3.2.- Cabecera de método.

La declaración de un método puede incluir los siguientes elementos:

1. **Modificadores** (como por ejemplo los ya vistos `public` o `private`, más algunos otros que irás conociendo poco a poco). No es obligatorio incluir modificadores en la declaración.
2. El **tipo devuelto** (o tipo de retorno), que consiste en el tipo de dato (primitivo o referencia) que el método devuelve tras ser ejecutado. Si eliges `void` como tipo devuelto, el método no devolverá ningún valor.
3. El **nombre del método**, aplicándose para los nombres el mismo convenio que para los atributos.
4. Una **lista de parámetros** separados por comas y entre paréntesis donde cada parámetro debe ir precedido por su tipo. Si el método no tiene parámetros la lista estará vacía y únicamente aparecerán los paréntesis.
5. Una **lista de excepciones** que el método puede lanzar. Se utiliza la palabra reservada `throws` seguida de una lista de nombres de excepciones separadas por comas. No es obligatorio que un método incluya una lista de excepciones, aunque muchas veces será conveniente. En unidades anteriores ya has trabajado con el concepto de excepción y más adelante volverás a hacer uso de ellas.
6. El **cuerpo del método**, encerrado entre llaves. El cuerpo contendrá el código del método (una lista sentencias y estructuras de control en lenguaje Java) así como la posible declaración de variables locales.

La sintaxis general de la cabecera de un método podría entonces quedar así:

```
[private | protected | public] [static] [abstract] [final] [native] [synchronized]
( [ ] )
[throws ]
```

Como sucede con todos los identificadores en Java (variables, clases, objetos, métodos, etc.), puede usarse cualquier identificador que cumpla las normas. Ahora bien, para mejorar la legibilidad del código, se ha establecido el siguiente convenio para nombrar los métodos: **utilizar un verbo en minúscula o bien un nombre formado por varias palabras que comience por un verbo en minúscula, seguido por adjetivos, nombres, etc. los cuales sí aparecerán en mayúsculas.**

Algunos ejemplos de métodos que siguen este convenio podrían ser: `ejecutar`, `romper`, `mover`, `subir`, `responder`, `obtenerX`, `establecerValor`, `estaVacio`, `estaLleno`, `moverFicha`, `subirPalanca`, `responderRapido`, `girarRuedaIzquierda`, `abrirPuertaDelantera`, `CambiarMarcha`, etc.

En el ejemplo de la clase `Punto`, puedes observar cómo los métodos `obtenerX` y `obtenerY` siguen el convenio de nombres para los métodos, devuelven en ambos casos un tipo `int`, su lista de parámetros es vacía (no tienen parámetros) y no lanzan ningún tipo de excepción:

```
* int obtenerX ()
```

```
* int obtenerY ()
```



Autoevaluación

¿Con cuál de los siguientes modificadores no puede ser declarado un método en Java?

- ☐ private.
- ☐ extern.
- ☐ static.
- ☐ public.

6.3.3.- Modificadores en la declaración de un método.

En la declaración de un método también pueden aparecer modificadores (como en la declaración de la clase o de los atributos). Un método puede tener los siguientes tipos de modificadores:

Modificadores de acceso. Son los mismos que en el caso de los atributos (por omisión o de paquete, `public`, `private` y `protected`) y tienen el mismo cometido (acceso al método sólo por parte de clases del mismo paquete, o por cualquier parte del programa, o sólo para la propia clase, o también para las subclases).

Modificadores de contenido. Son también los mismos que en el caso de los atributos (`static` y `final`) junto con, aunque su significado no es el mismo.

Otros modificadores (no son aplicables a los atributos, sólo a los métodos): `abstract`, `native`, `synchronized`.

Un método **static** es un método cuya implementación es igual para todos los objetos de la clase y sólo tendrá acceso a los atributos estáticos de la clase (dado que se trata de un método de clase y no de objeto, sólo podrá acceder a la información de clase y no la de un objeto en particular). Este tipo de métodos pueden ser llamados sin necesidad de tener un objeto de la clase instanciado.

En Java un ejemplo típico de métodos estáticos se encuentra en la clase `Math`, cuyos métodos son todos estáticos (`Math.abs`, `Math.sin`, `Math.cos`, etc.). Como habrás podido comprobar en este ejemplo, la llamada a métodos estáticos se hace normalmente usando el nombre de la propia clase y no el de una instancia (objeto), pues se trata realmente de un método de clase. En cualquier caso, los objetos también admiten la invocación de los métodos estáticos de su clase y funcionarían correctamente.

Un método **final** es un método que no permite ser sobrescrito por las clases descendientes de la clase a la que pertenece el método. Volverás a ver este modificador cuando estudies en detalle la **herencia**.

El modificador **native** es utilizado para señalar que un método ha sido implementado en código nativo (en un lenguaje que ha sido compilado a lenguaje máquina, como por ejemplo **C** o **C++**). En estos casos simplemente se indica la cabecera del método, pues no tiene cuerpo escrito en Java.

Un método **abstract** (**método abstracto**) es un método que no tiene implementación (el cuerpo está vacío). La implementación será realizada en las clases descendientes. Un método sólo puede ser declarado como **abstract** si se encuentra dentro de una clase **abstract**. También volverás a este modificador en unidades posteriores cuando trabajes con la **herencia**.

Por último, si un método ha sido declarado como **synchronized**, el entorno de ejecución obligará a que cuando un proceso esté ejecutando ese método, el resto de procesos que tengan que llamar a ese mismo método deberán esperar a que el otro proceso termine. Puede resultar útil si sabes que un determinado método va a poder ser llamado concurrentemente por varios procesos a la vez. Por ahora no lo vas a necesitar.

Dada la cantidad de modificadores que has visto hasta el momento y su posible aplicación en la declaración de clases, atributos o métodos, veamos un resumen de todos los que has visto y en qué casos pueden aplicarse:

Cuadro de aplicabilidad de los modificadores.

	Clase	Atributo	Método
Sin modificador (paquete)	Sí	Sí	Sí
public	Sí	Sí	Sí
private		Sí	Sí
protected	Sí	Sí	Sí
static		Sí	Sí
final	Sí	Sí	Sí
synchronized			Sí
native			Sí
abstract	Sí		Sí

6.3.4.- Parámetros en un método.

La lista de parámetros de un método se coloca tras el nombre del método. Esta lista estará constituida por pares de la forma " ". Cada uno de esos pares estará separado por comas y la lista completa estará encerrada entre paréntesis:

```
nombreMetodo ( , , ..., )
```

Si la lista de parámetros es vacía, tan solo aparecerán los paréntesis:

```
( )
```

A la hora de declarar un método, debes tener en cuenta:

Puedes incluir cualquier cantidad de parámetros. Se trata de una decisión del programador, pudiendo ser incluso una lista vacía.

Los parámetros podrán ser de cualquier tipo (tipos primitivos, referencias, objetos, arrays, etc.).

No está permitido que el nombre de una variable local del método coincida con el nombre de un parámetro.

No puede haber dos parámetros con el mismo nombre. Se produciría ambigüedad.

Si el nombre de algún parámetro coincide con el nombre de un atributo de la clase, éste será ocultado por el parámetro. Es decir, al indicar ese nombre en el código del método estarás haciendo referencia al parámetro y no al atributo. Para poder acceder al atributo tendrás que hacer uso del operador de autorreferencia `this`, que verás un poco más adelante.

En Java el paso de parámetros es siempre por valor, excepto en el caso de los tipos referenciados (por ejemplo los objetos) en cuyo caso se está pasando efectivamente una referencia. La referencia (el objeto en sí mismo) no podrá ser cambiada pero sí elementos de su interior (atributos) a través de sus métodos o por acceso directo si se trata de un miembro público.

Es posible utilizar una construcción especial llamada `varargs` (argumentos variables) que permite que un método pueda tener un número variable de parámetros. Para utilizar este mecanismo se colocan unos puntos suspensivos (tres puntos: "...") después del tipo del cual se puede tener una lista variable de argumentos, un espacio en blanco y a continuación el nombre del parámetro que aglutinará la lista de argumentos variables.

```
(...
```

Es posible además mezclar el uso de `varargs` con parámetros fijos. En tal caso, la lista de parámetros variables debe aparecer al final (y sólo puede aparecer una).

En realidad se trata una manera transparente de pasar un array con un número variable de elementos para no tener que hacerlo manualmente. Dentro del método habrá que ir recorriendo el array para ir obteniendo cada uno de los elementos de la lista de argumentos variables.

Para saber más

Si quieres ver algunos ejemplos de cómo utilizar el mecanismo de argumentos variables, puedes echar un vistazo al siguiente enlace (en inglés):

[Vargars.](#)

También puedes echar un vistazo al artículo general sobre paso de parámetros a métodos en los manuales de Oracle sobre Java:

[Passing Information to a Method or a Constructor.](#)

6.3.5.- Cuerpo de un método.

```
int obtenerX ()
{
    return x;
}
```

El interior de un método (cuerpo) está compuesto por una serie de sentencias en lenguaje Java:

Sentencias de **declaración de variables locales** al método.

Sentencias que implementan la **lógica del método** (estructuras de control como bucles o condiciones; utilización de métodos de otros objetos; cálculo de expresiones matemáticas, lógicas o de cadenas; creación de nuevos objetos, etc.). Es decir, todo lo que has visto en las unidades anteriores.

Sentencia de **devolución del valor de retorno** (return). Aparecerá al final del método y es la que permite devolver la información que se le ha pedido al método. Es la última parte del proceso y la forma de comunicarse con la parte de código que llamó al método (paso de mensaje de vuelta). Esta sentencia de devolución siempre tiene que aparecer al final del método. Tan solo si el tipo devuelto por el método es void (vacío) no debe aparecer (pues no hay que devolver nada al código llamante).

En el ejemplo de la clase `Punto`, tenías los métodos `obtenerX` y `obtenerY`. Veamos uno de ellos:

En ambos casos lo único que hace el método es precisamente devolver un valor (utilización de la sentencia `return`). No recibe parámetros (mensajes o información de entrada) ni hace cálculos, ni obtiene resultados intermedios o finales. Tan solo devuelve el contenido de un atributo. Se trata de uno de los métodos más sencillos que se pueden implementar: un método que devuelve el valor de un atributo. En inglés se les suele llamar métodos de tipo **get**, que en inglés significa **obtener**.

Además de esos dos métodos, la clase también disponía de otros dos que sirven para la función opuesta (`establecerX` y `establecerY`). Veamos uno de ellos:

```
void establecerX (int vx)
{
    x= vx;
}
```

En este caso se trata de pasar un valor al método (parámetro `vx` de tipo `int`) el cual será utilizado para modificar el contenido del atributo `x` del objeto. Como habrás podido comprobar, ahora no se devuelve ningún valor (el tipo devuelto es `void` y no hay sentencia `return`). En inglés se suele hablar de métodos de tipo **set**, que en inglés significa poner o fijar (**establecer** un valor). El método **establecerY** es prácticamente igual pero para establecer el valor del atributo `y`.

Normalmente el código en el interior de un método será algo más complejo y estará formado un conjunto de sentencias en las que se realizarán cálculos, se tomarán decisiones, se repetirán acciones, etc. Puedes ver un ejemplo más completo en el siguiente ejercicio.

Ejercicio resuelto

Vamos a seguir ampliando la clase en la que se representa un rectángulo en el plano (clase **Rectangulo**). Para ello has pensado en los siguientes métodos **públicos**:

Métodos **obtenerNombre** y **establecerNombre**, que permiten el acceso y modificación del atributo **nombre** del rectángulo.

Método **calcularSuperficie**, que calcula el área encerrada por el rectángulo.

Método **calcularPerímetro**, que calcula la longitud del perímetro del rectángulo.

Método **desplazar**, que mueve la ubicación del rectángulo en el plano en una cantidad `X` (para el eje `X`) y otra cantidad `Y` (para el eje `Y`). Se trata simplemente de sumar el desplazamiento `X` a las coordenadas `x1` y `x2`, y el desplazamiento `Y` a las coordenadas `y1` e `y2`. Los **parámetros** de entrada de este método serán por tanto `X` e `Y`, de tipo `double`.

Método **obtenerNumRectangulos**, que devuelve el número de rectángulos creados hasta el momento.

Incluye la implementación de cada uno de esos métodos en la clase **Rectangulo**.

6.3.6.- Sobrecarga de métodos.

En principio podrías pensar que un método puede aparecer una sola vez en la declaración de una clase (no se debería repetir el mismo nombre para varios métodos). Pero no tiene porqué siempre suceder así. Es posible tener varias versiones de un mismo método (varios métodos con el mismo nombre) gracias a la **sobrecarga de métodos**.

El lenguaje Java soporta la característica conocida como **sobrecarga de métodos**. Ésta permite declarar en una misma clase varias versiones del mismo método con el mismo nombre. La forma que tendrá el compilador de distinguir entre varios métodos que tengan el mismo nombre será mediante la lista de parámetros del método: si el método tiene una lista de parámetros diferente, será considerado como un método diferente (aunque tenga el mismo nombre) y el analizador léxico no producirá un error de compilación al encontrar dos nombres de método iguales en la misma clase.

Imagínate que estás desarrollando una clase para escribir sobre un lienzo que permite utilizar diferentes tipografías en función del tipo de información que se va a escribir. Es probable que necesitemos un método diferente según se vaya a pintar un número entero (`int`), un número real (`double`) o una cadena de caracteres (`String`). Una primera opción podría ser definir un nombre de método diferente dependiendo de lo que se vaya a escribir en el lienzo. Por ejemplo:

```
Método pintarEntero (int entero).  
Método pintarReal (double real).  
Método pintarCadena (double String).  
Método pintarEnteroCadena (int entero, String cadena).
```

Y así sucesivamente para todos los casos que desees contemplar...

La posibilidad que te ofrece la sobrecarga es utilizar un mismo nombre para todos esos métodos (dado que en el fondo hacen lo mismo: pintar). Pero para poder distinguir unos de otros será necesario que siempre exista alguna diferencia entre ellos en las listas de parámetros (bien en el número de parámetros, bien en el tipo de los parámetros). Volviendo al ejemplo anterior, podríamos utilizar un mismo nombre, por ejemplo `pintar`, para todos los métodos anteriores:

```
Método pintar (int entero).  
Método pintar (double real).  
Método pintar (double String).  
Método pintar (int entero, String cadena).
```

En este caso el compilador no va a generar ningún error pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes.

Lo que sí habría producido un error de compilación habría sido por ejemplo incluir otro método `pintar (int entero)`, pues es imposible distinguirlo de otro método con el mismo nombre y con la misma lista de parámetros (ya existe un método `pintar` con un único parámetro de tipo `int`).

También debes tener en cuenta que el **tipo devuelto** por el método no es considerado a la hora de identificar un método, así que un tipo devuelto diferente no es suficiente para distinguir un método de otro. Es decir, no podrías definir dos métodos exactamente iguales en nombre y lista de parámetros e intentar distinguirlos indicando un tipo devuelto diferente. El compilador producirá un error de duplicidad en el nombre del método y no te lo permitirá.

Es conveniente no abusar de sobrecarga de métodos y utilizarla con cierta moderación (cuando realmente puede beneficiar su uso), dado que podría hacer el código menos legible.

AUTOEVALUACIÓN

En una clase Java puedes definir tantos métodos con el mismo nombre como desees y sin ningún tipo de restricción pues el lenguaje soporta la sobrecarga de métodos y el compilador sabrá distinguir unos métodos de otros. ¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

6.3.7.- La referencia `this`.

La palabra reservada `this` consiste en una referencia al objeto actual. El uso de este operador puede resultar muy útil a la hora de evitar la ambigüedad que puede producirse entre el nombre de un parámetro de un método y el nombre de un atributo cuando ambos tienen el mismo identificador (mismo nombre). En tales casos el parámetro "oculta" al atributo y no tendríamos acceso directo a él (al escribir el identificador estaríamos haciendo referencia al parámetro y no al atributo). En estos casos la referencia `this` nos permite acceder a estos atributos ocultos por los parámetros.

Dado que `this` es una referencia a la propia clase en la que te encuentras en ese momento, puedes acceder a sus atributos mediante el operador punto (.) como sucede con cualquier otra clase u objeto. Por tanto, en lugar de poner el nombre del atributo (que estos casos haría referencia al parámetro), podrías escribir `this.nombreAtributo`, de manera que el compilador sabrá que te estás refiriendo al atributo y se eliminará la ambigüedad.

En el ejemplo de la clase `Punto`, podríamos utilizar la referencia `this` si el nombre del parámetro del método coincidiera con el del atributo que se desea modificar. Por ejemplo

```
void establecerX (int x)
{
    this.x= x;
}
```

En este caso ha sido indispensable el uso de `this`, pues si no sería imposible saber en qué casos te estás refiriendo al parámetro `x` y en cuáles al atributo `x`. Para el compilador el identificador `x` será siempre el parámetro, pues ha "ocultado" al atributo.

En algunos casos puede resultar útil hacer uso de la referencia `this` aunque no sea necesario, pues puede ayudar a mejorar la legibilidad del código.

Para saber más

Puedes echar un vistazo al artículo general sobre la referencia `this` en los manuales de Oracle (en inglés):

[Using the this Keyword.](#)

AUTOEVALUACIÓN

La referencia `this` en Java resulta muy útil cuando se quieren utilizar en un método nombres de parámetros que coinciden con los nombres de variables locales del método. ¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

Ejercicio resuelto

Modificar el método `obtenerNombre` de la clase `Rectangulo` de ejercicios anteriores utilizando la referencia `this`.

6.3.8.- Métodos estáticos.

Como ya has visto en ocasiones anteriores, un **método estático** es un método que puede ser usado directamente desde la clase, sin necesidad de tener que crear una instancia para poder utilizar al método. También son conocidos como **métodos de clase** (como sucedía con los **atributos de clase**), frente a los **métodos de objeto** (es necesario un objeto para poder disponer de ellos).

Los métodos estáticos no pueden manipular atributos de instancias (objetos) sino atributos estáticos (de clase) y suelen ser utilizados para realizar operaciones comunes a todos los objetos de la clase, más que para una instancia concreta.

Algunos ejemplos de operaciones que suelen realizarse desde métodos estáticos:

Acceso a atributos específicos de clase: incremento o decremento de contadores internos de la clase (no de instancias), acceso a un posible atributo de nombre de la clase, etc.

Operaciones genéricas relacionadas con la clase pero que no utilizan atributos de instancia. Por ejemplo una clase **NIF** (o **DNI**) que permite trabajar con el DNI y la letra del NIF y que proporciona funciones adicionales para calcular la letra NIF de un número de DNI que se le pase como parámetro. Ese método puede ser interesante para ser usado desde fuera de la clase de manera independiente a la existencia de objetos de tipo NIF.

En la biblioteca de Java es muy habitual encontrarse con clases que proporcionan métodos estáticos que pueden resultar muy útiles para cálculos auxiliares, conversiones de tipos, etc. Por ejemplo, la mayoría de las clases del paquete `java.lang` que representan tipos (`Integer`, `String`, `Float`, `Double`, `Boolean`, etc.) ofrecen métodos estáticos para hacer conversiones. Aquí tienes algunos ejemplos:

`static String valueOf (int i).` Devuelve la representación en formato `String` (cadena) de un valor `int`. Se trata de un método que no tiene que ver nada en absoluto con instancias de concretas de `String`, sino de un método auxiliar que puede servir como herramienta para ser usada desde otras clases. Se utilizaría directamente con el nombre de la clase. Por ejemplo:

```
String enteroCadena= String.valueOf (23).
```

`static String valueOf (float f).` Algo similar para un valor de tipo `float`. Ejemplo de uso:

```
String floatCadena= String.valueOf (24.341).
```

`static int parseInt (String s).` En este caso se trata de un método estático de la clase `Integer`. Analiza la cadena pasada como parámetro y la transforma en un `int`. Ejemplo de uso:

```
int cadenaEntero= Integer.parseInt ("-12").
```

Todos los ejemplos anteriores son casos en los que se utiliza directamente la clase como una especie de **caja de herramientas** que contiene métodos que pueden ser utilizados desde cualquier parte, por eso suelen ser métodos públicos.

Para saber más

Puedes echar un vistazo a algunas clases del paquete `java.lang` (por ejemplo `Integer`, `String`, `Float`, `Double`, `Boolean` y `Math`) y observar la gran cantidad de métodos estáticos que ofrecen para ser utilizados sin necesidad de tener que crear objetos de esas clases:

`Package java.lang.`

6.4.- Encapsulación, control de acceso y visibilidad.

Caso práctico

Juan está desarrollando algunas de las clases que van a necesitar para el proyecto de la **Clínica Veterinaria** y empiezan a asaltarle dudas acerca de cuándo deben ser visibles unos u otros miembros. Recuerda ha estado viendo con **María** los distintos modificadores de acceso aplicables a las clases, atributos y métodos.

Está claro que hasta que no empiece a utilizarlos para casos concretos y con aplicación práctica real no terminará de comprender exactamente su mecánica de funcionamiento: cuándo interesa ocultar un determinado miembro, cuándo interesa que otro miembro sea visible, en qué casos vale la pena crear un método para acceder al valor de un atributo, etc.

Dentro de la Programación **Orientada a Objetos** ya has visto que es muy importante el concepto de **ocultación**, la cual ha sido lograda gracias a la **encapsulación** de la información dentro de las clases. De esta manera una clase puede ocultar parte de su contenido o restringir el acceso a él para evitar que sea manipulado de manera inadecuada. Los **modificadores de acceso** en Java permiten especificar el **ámbito de visibilidad** de los miembros de

una clase, proporcionando así un mecanismo de accesibilidad a varios niveles.

Acabas de estudiar que cuando se definen los miembros de una clase (atributos o métodos), e incluso la propia clase, se indica (aunque sea por omisión) un modificador de acceso. En función de la visibilidad que se desee que tengan los objetos o los miembros de esos objetos se elegirá alguno de los modificadores de acceso que has estudiado. Ahora que ya sabes cómo escribir una clase completa (declaración de la clase, declaración de sus atributos y declaración de sus métodos), vamos a hacer un repaso general de las opciones de **visibilidad (control de acceso)** que has estudiado.

Los modificadores de acceso determinan si una clase puede utilizar determinados miembros (acceder a atributos o invocar miembros) de otra clase. Existen dos niveles de control de acceso:

1. **A nivel general (nivel de clase):** visibilidad de la propia clase.
2. **A nivel de miembros:** especificación, miembro por miembro, de su nivel de visibilidad.

En el caso de la clase, ya estudiaste que los niveles de visibilidad podían ser:

Público (modificador `public`), en cuyo caso la clase era visible a cualquier otra clase (cualquier otro fragmento de código del programa).

Privada al paquete (sin modificador o modificador "por omisión"). En este caso, la clase sólo será visible a las demás clases del mismo paquete, pero no al resto del código del programa (otros paquetes).

En el caso de los miembros, disponías de otras dos posibilidades más de niveles de accesibilidad, teniendo un total de cuatro opciones a la hora de definir el control de acceso al miembro:

Público (modificador `public`), igual que en el caso global de la clase y con el mismo significado (miembro visible desde cualquier parte del código).

Privado al paquete (sin modificador), también con el mismo significado que en el caso de la clase (miembro visible sólo desde clases del mismo paquete, ni siquiera será visible desde una subclase salvo si ésta está en el mismo paquete).

Privado (modificador `private`), donde sólo la propia clase tiene acceso al miembro.

Protegido (modificador `protected`)

Para saber más

Puedes echar un vistazo al artículo sobre el control de acceso a los miembros de una clase Java en los manuales de Oracle (en inglés):

[Controlling Access to Members of a Class.](#)

AUTOEVALUACIÓN

Si queremos que un atributo de una clase sea accesible solamente desde el código de la propia clase o de aquellas clases que hereden de ella, ¿qué modificador de acceso deberíamos utilizar?

- ☐ `private`.
- ☐ `protected`.
- ☐ `public`.
- ☐ Ninguno de los anteriores.

6.4.1.- Ocultación de atributos. Métodos de acceso.

Los atributos de una clase suelen ser declarados como privados a la clase o, como mucho, `protected` (accesibles también por clases heredadas), pero no como `public`. De esta manera puedes evitar que sean manipulados inadecuadamente (por ejemplos modificarlos sin ningún tipo de control) desde el exterior del objeto.

En estos casos lo que se suele hacer es declarar esos atributos como privados o protegidos y crear métodos públicos que permitan acceder a esos atributos. Si se trata de un atributo cuyo contenido puede ser observado pero no modificado directamente, puede implementarse un método de "obtención" del atributo (en inglés se les suele llamar método de tipo `get`) y si el atributo puede ser modificado, puedes también implementar otro método para la modificación o "establecimiento" del valor del atributo (en inglés se le suele llamar método de tipo `set`). Esto ya lo has visto en apartados anteriores.

Si recuerdas la clase `Punto` que hemos utilizado como ejemplo, ya hiciste algo así con los métodos de obtención y establecimiento de las coordenadas:

```
private int x, y;
// Métodos get
public int obtenerX () { return x; }
public int obtenerY () { return y; }
// Métodos set
public void establecerX (int x) { this.x= x; }
public void establecerY (int y) { this.y= y; }
```

Así, para poder obtener el valor del atributo `x` de un objeto de tipo `Punto` será necesario utilizar el método `obtenerX()` y no se podrá acceder directamente al atributo `x` del objeto.

En algunos casos los programadores directamente utilizan nombres en inglés para nombrar a estos métodos: `getX`, `getY` (), `setX`, `setY`, `getNombre`, `setNombre`, `getColor`, etc.

También pueden darse casos en los que no interesa que pueda observarse directamente el valor de un atributo, sino un determinado procesamiento o cálculo que se haga con el atributo (pero no el valor original). Por ejemplo podrías tener un atributo `DNI` que almacene los 8 dígitos del DNI pero no la letra del `NIF` (pues se puede calcular a partir de los dígitos). El método de acceso para el DNI (método `getDNI`) podría proporcionar el DNI completo (es decir, el NIF, incluyendo la letra), mientras que la letra no es almacenada realmente en el atributo del objeto. Algo similar podría suceder con el **dígito de control de una cuenta bancaria**, que puede

no ser almacenado en el objeto, pero sí calculado y devuelto cuando se nos pide el número de cuenta completo.

En otros casos puede interesar disponer de métodos de modificación de un atributo pero a través de un determinado procesamiento previo para por ejemplo poder controlar errores o valores inadecuados. Volviendo al ejemplo del NIF, un método para modificar un DNI (método `setDNI`) podría incluir la letra (NIF completo), de manera que así podría comprobarse si el número de DNI y la letra coinciden (es un NIF válido). En tal caso se almacenará el DNI y en caso contrario se producirá un error de validación (por ejemplo lanzando una excepción). En cualquier caso, el DNI que se almacenara sería solamente el número y no la letra (pues la letra es calculable a partir del número de DNI).

Autoevaluación

Los atributos de una clase suelen ser declarados como `public` para facilitar el acceso y la visibilidad de los miembros de la clase.

¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

6.4.2.- Ocultación de métodos.

Normalmente los métodos de una clase pertenecen a su interfaz y por tanto parece lógico que sean declarados como públicos. Pero también es cierto que pueden darse casos en los que exista la necesidad de disponer de algunos métodos privados a la clase. Se trata de métodos que realizan operaciones intermedias o auxiliares y que son utilizados por los métodos que sí forman parte de la interfaz. Ese tipo de métodos (de comprobación, de adaptación de formatos, de cálculos intermedios, etc.) suelen declararse como privados pues no son de interés (o no es apropiado que sean visibles) fuera del contexto del interior del objeto.

En el ejemplo anterior de objetos que contienen un DNI, será necesario calcular la letra correspondiente a un determinado número de DNI o comprobar si una determinada combinación de número y letra forman un DNI válido. Este tipo de cálculos y comprobaciones podrían ser implementados en métodos privados de la clase (o al menos como métodos protegidos).

AUTOEVALUACIÓN

Dado que los métodos de una clase forman la interfaz de comunicación de esa clase con otras clases, todos los elementos de una clase deben ser siempre declarados como públicos. ¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

Ejercicio resuelto

Vamos a intentar implementar una clase que incluya todo lo que has visto hasta ahora. Se desea crear una clase que represente un **DNI español** y que tenga las siguientes características:

La clase almacenará el número de DNI en un `int`, sin guardar la letra, pues se puede calcular a partir del número. Este atributo será privado a la clase. Formato del atributo: `private int numDNI`. Para acceder al DNI se dispondrá de dos métodos **obtener** (`get`), uno que proporcionará el número de DNI (sólo las cifras numéricas) y otro que devolverá el **NIF** completo (incluida la letra). El formato del método será:

```
* public int obtenerDNI ().  
* public String obtenerNIF ().
```

Para modificar el DNI se dispondrá de dos métodos establecer (set), que permitirán modificar el DNI. Uno en el que habrá que proporcionar el NIF completo (número y letra). Y otro en el que únicamente será necesario proporcionar el DNI (las siete u ocho cifras). Si el DNI/NIF es incorrecto se debería lanzar algún tipo de **excepción**. El formato de los métodos (**sobrecargados**) será:

```
* public void establecer (String nif) throws ...  
* public void establecer (int dni) throws ...
```

La clase dispondrá de algunos métodos internos privados para calcular la letra de un número de DNI cualquiera, para comprobar si un DNI con su letra es válido, para extraer la letra de un NIF, etc. Aquellos métodos que no utilicen ninguna variable de objeto podrían declararse como estáticos (pertenecientes a la clase). Formato de los métodos:

```
* private static char calcularLetraNIF (int dni).  
* private boolean validarNIF (String nif).  
* private static char extraerLetraNIF (String nif).  
* private static int extraerNumeroNIF (String nif).
```

Para calcular la letra NIF correspondiente a un número de DNI puedes consultar el artículo sobre el NIF de la Wikipedia:

[Artículo en la Wikipedia sobre el Número de Identificación Fiscal \(NIF\).](#)



6.5.- Constructores.

Caso práctico

María y Juan ya han creado y utilizado objetos y cuentan con algunos pequeños programas de ejemplo compuestos por varias clases además de la clase principal (la que contiene el método `main`). **Ada** ha estado revisando su trabajo y ha quedado muy satisfecha, aunque al observar la estructura de las clases les ha comentado algo que los ha dejado un poco despistados:

-Estas clases tienen muy buena pinta, aunque faltaría añadirles algunos constructores para poder mejorar su flexibilidad a la hora de instanciar objetos, ¿no creéis?

Ambos han asentido porque eran conscientes de que hasta el momento no habían estado incluyendo constructores en sus clases, estaban aprovechando el constructor por defecto que añadía el compilador.

-Parece que ha llegado el momento de añadir nuestros propios constructores -le dice **María** a **Juan**.

Como ya has estudiado en unidades anteriores, en el ciclo de vida de un objeto se pueden distinguir las fases de:

Construcción del objeto.

Manipulación y utilización del objeto accediendo a sus miembros.

Destrucción del objeto.

Como has visto en el apartado anterior, durante la fase de construcción o instanciación de un objeto es cuando se reserva espacio en memoria para sus atributos y se inicializan algunos de ellos. Un **constructor** es un método especial con el **mismo nombre de la clase** y que se encarga de realizar este proceso.

El proceso de declaración y creación de un objeto mediante el operador **new** ya ha sido estudiado en apartados anteriores. Sin embargo las clases que hasta ahora has creado no tenían constructor. Has estado utilizando los constructores por defecto que proporciona Java al compilar la clase. Ha llegado el momento de que empieces a implementar tus propios constructores.

Los métodos constructores se encargan de llevar a cabo el proceso de creación o construcción de un objeto.

AUTOEVALUACIÓN

¿Con qué nombre es conocido el método especial de una clase que se encarga de reservar espacio e inicializar atributos cuando se crea un objeto nuevo? ¿Qué nombre tendrá ese método en la clase?

- ☐ Método constructor. Su nombre dentro de la clase será constructor.
- ☐ Método inicializador. Su nombre dentro de la clase será el mismo nombre que tenga la clase.
- ☐ Método constructor. Su nombre dentro de la clase será el mismo nombre que tenga la clase.
- ☐ Método constructor. Su nombre dentro de la clase será new.

6.5.1.- Concepto de constructor.

Un **constructor** es un método que tiene el mismo nombre que la clase a la que pertenece y que no devuelve ningún valor tras su ejecución. Su función es la de proporcionar el mecanismo de creación de instancias (objetos) de la clase.

Cuando un objeto es declarado, en realidad aún no existe. Tan solo se trata de un nombre simbólico (una variable) que en el futuro hará referencia a una zona de memoria que contendrá la información que representa realmente a un objeto. Para que esa variable de objeto aún "vacía" (se suele decir que es una referencia nula o vacía) apunte, o haga referencia a una zona de memoria que represente a una instancia de clase (objeto) existente, es necesario "**construir**" el objeto. Ese proceso se realizará a través del método **constructor** de la clase.

Por tanto para crear un nuevo objeto es necesario realizar una llamada a un método constructor de la clase a la que pertenece ese objeto. Ese proceso se realiza mediante la utilización del operador `new`.

Hasta el momento ya has utilizado en numerosas ocasiones el operador `new` para instanciar o crear objetos. En realidad lo que estabas haciendo era una llamada al constructor de la clase para que reservara memoria para ese objeto y por tanto "crear" físicamente el objeto en la memoria (dotarlo de existencia física dentro de la memoria del ordenador). Dado que en esta unidad estás ya definiendo tus propias clases, parece que ha llegado el momento de que empieces a escribir también los constructores de tus clases.

Por otro lado, si un constructor es al fin y al cabo una especie de método (aunque algo especial) y Java soporta la sobrecarga de métodos, podrías plantearte la siguiente pregunta: ¿podrá una clase disponer de más de constructor? En otras palabras, ¿será posible la sobrecarga de constructores? La respuesta es afirmativa.

Una misma clase puede disponer de varios constructores. Los constructores soportan la sobrecarga.

Es necesario que toda clase tenga al menos un constructor. Si no se define ningún constructor en una clase, el compilador creará por nosotros un constructor por defecto vacío que se encarga de inicializar todos los atributos a sus valores por defecto (0 para los numéricos, `null` para las referencias, `false` para los boolean, etc.).

Algunas analogías que podrías imaginar para representar el constructor de una clase podrían ser:

Los moldes de cocina para flanes, galletas, pastas, etc.

Un cubo de playa para crear castillos de arena.

Un molde de un lingote de oro.

Una bolsa para hacer cubitos de hielo.

Una vez que incluyas un constructor personalizado a una clase, el compilador ya no incluirá el constructor por defecto (sin parámetros) y por tanto si intentas usarlo se produciría un error de compilación. Si quieres que tu clase tenga también un constructor sin parámetros tendrás que escribir su código (ya no lo hará por ti el compilador).

6.5.2.- Creación de constructores.

Cuando se escribe el código de una clase normalmente se pretende que los objetos de esa clase se creen de una determinada manera. Para ello se definen uno o más constructores en la clase. En la definición de un constructor se indican:

- El tipo de acceso.

- El nombre de la clase (el nombre de un método constructor es siempre el nombre de la propia clase).

- La lista de parámetros que puede aceptar.

- Si lanza o no excepciones.

- El cuerpo del constructor (un bloque de código como el de cualquier método).

Como puedes observar, la estructura de los constructores es similar a la de cualquier método, con las excepciones de que **no tiene tipo de dato devuelto** (no devuelve ningún valor) y que **el nombre del método constructor debe ser obligatoriamente el nombre de la clase**.

Reflexiona

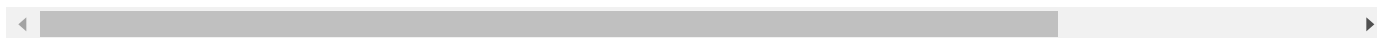
Si defines constructores personalizados para una clase, el constructor por defecto (sin parámetros) para esa clase deja de ser generado por el compilador, de manera que tendrás que crearlo tú si quieres poder utilizarlo.

Si se ha creado un constructor con parámetros y no se ha implementado el constructor por defecto, el intento de utilización del constructor por defecto producirá un error de compilación (el compilador no lo hará por nosotros).

Un ejemplo de constructor para la clase `Punto` podría ser:

```
public Punto (int x, int y)
{
    this.x= x;
    this.y= y;
    cantidadPuntos++; // Suponiendo que tengamos un atributo estático cantidadPuntos
}
```

En este caso el constructor recibe dos parámetros. Además de reservar espacio para los atributos (de lo cual se encarga automáticamente Java), también asigna sendos valores iniciales a los atributos `x` e `y`. Por último incrementa un atributo (probablemente estático) llamado `cantidadPuntos`.



AUTOEVALUACIÓN

El constructor por defecto (sin parámetros) está siempre disponible para usarlo en cualquier clase. ¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

6.5.3.- Utilización de constructores.

Una vez que dispongas de tus propios constructores personalizados, la forma de utilizarlos es igual que con el constructor por defecto (mediante la utilización de la palabra reservada `new`) pero teniendo en cuenta que si has declarado parámetros en tu método constructor, tendrás que llamar al constructor con algún valor para esos parámetros.

Un ejemplo de utilización del constructor que has creado para la clase `Punto` en el apartado anterior podría ser:

```
Punto p1;  
p1= new Punto (10, 7);
```

En este caso no se estaría utilizando el constructor por defecto sino el constructor que acabas de implementar en el cual además de reservar memoria se asigna un valor a algunos de los atributos.

Para saber más

Puedes echar un vistazo al artículo sobre constructores de una clase Java en los manuales de Oracle (en inglés):

[Providing Constructors for Your Classes.](#)

También puedes echar un vistazo a estos vídeos que ya se te han recomendado en unidades anteriores. Ahora probablemente comprenderás mucho mejor el proceso que se muestra pues es más o menos lo que has tenido que hacer tú:

[Resumen textual alternativo](#)

[Resumen textual alternativo](#)

Ejercicio resuelto

Ampliar el ejercicio de la clase **Rectangulo** añadiéndole tres constructores:

1. Un constructor sin parámetros (para sustituir al constructor por defecto) que haga que los valores iniciales de las esquinas del rectángulo sean $(0,0)$ y $(1,1)$;
2. Un constructor con cuatro parámetros, **x1**, **y1**, **x2**, **y2**, que rellene los valores iniciales de los atributos del rectángulo con los valores proporcionados a través de los parámetros.
3. Un constructor con dos parámetros, **base** y **altura**, que cree un rectángulo donde el vértice inferior derecho esté ubicado en la posición $(0,0)$ y que tenga una base y una altura tal y como indican los dos parámetros proporcionados.

6.5.4.- Constructores de copia.

Una forma de iniciar un objeto podría ser mediante la copia de los valores de los atributos de otro objeto ya existente. Imagina que necesitas varios objetos iguales (con los mismos valores en sus atributos) y que ya tienes uno de ellos perfectamente configurado (sus atributos contienen los valores que tú necesitas). Estaría bien disponer de un constructor que hiciera copias idénticas de ese objeto.

Durante el proceso de creación de un objeto puedes generar objetos exactamente iguales (basados en la misma clase) que se distinguirán posteriormente porque podrán tener estados distintos (valores diferentes en los atributos). La idea es poder decirle a la clase que además de generar un objeto nuevo, que lo haga con los mismos valores que tenga otro objeto ya existente. Es decir, algo así como si pudieras **clonar** el objeto tantas veces como te haga falta. A este tipo de mecanismo se le suele llamar **constructor copia** o **constructor de copia**.

Un constructor copia es un método constructor como los que ya has utilizado pero con la particularidad de que recibe como parámetro una referencia al objeto cuyo contenido se desea copiar. Este método revisa cada uno de los atributos del objeto recibido como parámetro y se copian todos sus valores en los atributos del objeto que se está creando en ese momento en el método constructor.

Un ejemplo de constructor copia para la clase `Punto` podría ser:

```
public Punto (Punto p)
{
    this.x= p.obtenerX();
    this.y= p.obtenerY();
}
```

En este caso el constructor recibe como parámetro un objeto del mismo tipo que el que va a ser creado (clase `Punto`), inspecciona el valor de sus atributos (atributos `x` e `y`), y los reproduce en los atributos del objeto en proceso de construcción (`this`).

Un ejemplo de utilización de ese constructor podría ser:

```
Punto p1, p2;
p1= new Punto (10, 7);
p2= new Punto (p1);
```

En este caso el objeto `p2` se crea a partir de los valores del objeto `p1`.

AUTOEVALUACIÓN

Toda clase debe incluir un constructor copia en su implementación.
¿Verdadero o falso?

Verdadero. ☐ Falso. ☐

Ejercicio resuelto

Ampliar el ejercicio de la clase Rectangulo añadiéndole un constructor copia.

6.5.5.- Destrucción de objetos.

Como ya has estudiado en unidades anteriores, cuando un objeto deja de ser utilizado, los recursos usados por él (memoria, acceso a archivos, conexiones con bases de datos, etc.) deberían de ser liberados para que puedan volver a ser utilizados por otros procesos (mecanismo de **destrucción del objeto**).

Mientras que de la construcción de los objetos se encargan los métodos constructores, de la destrucción se encarga un proceso del entorno de ejecución conocido como **recolector de basura (garbage collector)**. Este proceso va buscando periódicamente objetos que ya no son referenciados (no hay ninguna variable que haga referencia a ellos) y los marca para ser eliminados. Posteriormente los irá eliminando de la memoria cuando lo considere oportuno (en función de la carga del sistema, los recursos disponibles, etc.).

Normalmente se suele decir que en Java no hay método destructor y que en otros lenguajes orientados a objetos como **C++**, sí se implementa explícitamente el destructor de una clase de la misma manera que se define el constructor. En realidad en Java también es posible implementar el método destructor de una clase, se trata del método `finalize()`.

Este método `finalize` es llamado por el recolector de basura cuando va a destruir el objeto (lo cual nunca se sabe cuándo va a suceder exactamente, pues una cosa es que el objeto sea marcado para ser borrado y otra que sea borrado efectivamente). Si ese método no existe, se ejecutará un destructor por defecto (el método `finalize` que contiene la clase `Object`, de la cual heredan todas las clases en Java) que liberará la memoria ocupada por el objeto. Se recomienda por tanto que si un objeto utiliza determinados recursos de los cuales no tienes garantía que el entorno de ejecución los vaya a liberar (cerrar archivos, cerrar conexiones de red, cerrar conexiones con bases de datos, etc.), implementes explícitamente un método `finalize` en tus clases. Si el único recurso que utiliza tu clase es la memoria necesaria para albergar sus atributos, eso sí será liberado sin problemas. Pero si se trata de algo más complejo, será mejor que te encargues tú mismo de hacerlo implementando tu destructor personalizado (`finalize`).

Por otro lado, esta forma de funcionar del entorno de ejecución de Java (destrucción de objetos no referenciados mediante el recolector de basura) implica que no puedas saber exactamente cuándo un objeto va a ser definitivamente destruido, pues si una variable deja de ser referenciada (se cierra el ámbito de ejecución donde fue creada) no implica necesariamente que sea inmediatamente borrada, sino que simplemente es marcada para que el recolector la borre cuando pueda hacerlo.

Si en un momento dado fuera necesario garantizar que el proceso de finalización (método `finalize`) sea invocado, puedes recurrir al método `runFinalization()` de la clase `System`

para forzarlo:

```
System.runFinalization ();
```

Este método se encarga de llamar a todos los métodos `finalize` de todos los objetos marcados por el recolector de basura para ser destruidos.

Si necesitas implementar un destructor (normalmente no será necesario), debes tener en cuenta que:

El nombre del método destructor debe ser `finalize ()`.

No puede recibir parámetros.

Sólo puede haber un destructor en una clase. No es posible la sobrecarga dado que no tiene parámetros.

No puede devolver ningún valor. Debe ser de tipo `void`.

AUTOEVALUACIÓN

Cuando se abandona el ámbito de un objeto en Java éste es marcado por el recolector de basura para ser destruido. En muchas ocasiones una clase Java no tiene un método destructor, pero si fuera necesario hacerlo ¿podrías implementar un método destructor en una clase Java? ¿Qué nombre habría que ponerle?

- ☐ Sí es posible. El nombre del método sería `finalize()`.
- ☐ No es posible disponer de un método destructor en una clase Java.
- ☐ Sí es posible. El nombre del método sería `destructor ()`.
- ☐ Sí es posible. El nombre del método sería `~nombreClase`, como en el lenguaje C++.

7.- Librerías de objetos (paquetes).

Caso práctico

-¡Vaya! -exclama **María**- No consigo encontrar la clase **Persona** dentro del conjunto de clases que hasta ahora he creado.

-¿Por qué no pruebas a dividir las clases en paquetes? -pregunta **Juan**- En un paquete agrupas las que estén relacionadas, y así te será más fácil encontrar una clase la próxima vez. Además puedes crear paquetes dentro de otros, como si fuera una estructura de directorios.

-¿Ah sí? Pues es justo lo que necesito para resolver este desorden. Voy a ponerme con ello -añade **María**.

Conforme nuestros programas se van haciendo más grandes, el número de clases va creciendo. Meter todas las clases en único directorio no ayuda a que estén bien organizadas, lo mejor es hacer **grupos de clases**, de forma que todas las clases que estén relacionadas o traten sobre un mismo tema estén en el mismo grupo.

Un **paquete** de clases es una agrupación de clases que consideramos que están relacionadas entre sí o tratan de un tema común.

Las clases de un mismo paquete tienen un acceso privilegiado a los atributos y métodos de otras clases de dicho paquete. Es por ello por lo que se considera que los paquetes son también, en cierto modo, unidades de encapsulación y ocultación de información.

Java nos ayuda a organizar las clases en **paquetes**. En cada fichero `.java` que hagamos, al principio, podemos indicar a qué **paquete** pertenece la clase que hagamos en ese fichero.

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;
```

Por ejemplo, si decidimos agrupar en un paquete "ejemplos" un programa llamado "Bienvenida", pondríamos en nuestro fichero `Bienvenida.java` lo siguiente:

El código es exactamente igual que como hemos venido haciendo hasta ahora, solamente hemos añadido la línea `"package ejemplos;"` al principio. En la imagen se muestra cómo aparecen los paquetes en el entorno integrado de Netbeans.

Debes conocer

En la siguiente presentación puedes ver una demostración de cómo hemos creado el proyecto Bienvenida:

Creación de proyecto con paquetes

[Resumen textual alternativo](#)

7.1.- Sentencia import.

Cuando queremos utilizar una clase que está en un paquete distinto a la clase que estamos utilizando, se suele utilizar la sentencia `import`. Por ejemplo, si queremos utilizar la clase `Scanner` que está en el paquete `java.util` de la Biblioteca de Clases de Java, tendremos que utilizar esta sentencia:

```
import java.util.Scanner;
```

Se pueden importar todas las clases de un paquete, así:

```
import java.awt.*;
```

Esta sentencia debe aparecer al principio de la clase, justo después de la sentencia `package`, si ésta existiese.

También podemos utilizar la clase sin sentencia `import`, en cuyo caso cada vez que queramos usarla debemos indicar su ruta completa:

```
java.util.Scanner teclado = new java.util.Scanner (System.in);
```

Hasta aquí todo correcto. Sin embargo, al trabajar con paquetes, Java nos obliga a organizar los directorios, compilar y ejecutar de cierta forma para que todo funcione adecuadamente.

Para saber más

Te proponemos el siguiente enlace para repasar conceptos que hemos visto a lo largo de la unidad sobre programación orientada a objetos y utilización de clases:

[Tutorial de Java en español- Capítulo 13 - Clases y Objetos.](#)

[Resumen textual alternativo](#)

La siguiente parte del vídeo habla sobre la utilización de objetos y clases en Java:

Tutorial de Java en español- Capítulo 14 - Clases y Objetos.

Resumen textual alternativo

7.2.- Compilar y ejecutar clases con paquetes.

Si hacemos que `Bienvenida.java` pertenezca al paquete `ejemplos`, debemos crear un subdirectorio `"ejemplos"` y meter dentro el archivo `Bienvenida.java`.

Por ejemplo, en Linux tendríamos esta estructura de directorios:

```
//Proyecto_Bienvenida/ejemplos/Bienvenida.java
```

Debemos tener cuidado con las mayúsculas y las minúsculas, para evitar problemas, tenemos que poner el nombre en `"package"` exactamente igual que el nombre del subdirectorio.

Para **compilar** la clase `Bienvenida.java` que está en el paquete `ejemplos` debemos situarnos en el directorio padre del paquete y compilar desde ahí:

```
$ cd //Proyecto_Bienvenida
$ javac ejemplos/Bienvenida.java
```

Si todo va bien, en el directorio `ejemplos` nos aparecerá la clase compilada `Bienvenida.class`.

Para ejecutar la clase compilada `Bienvenida.class` que está en el directorio `ejemplos`, debemos seguir situados en el directorio padre del paquete. El nombre completo de la clase es `"paquete.clase"`, es decir `"ejemplos.Bienvenida"`. Los pasos serían los siguientes:

```
$ cd //Proyecto_Bienvenida
$ java ejemplos/Bienvenida
Bienvenido a Java
```

Si todo es correcto, debe salir el mensaje `"Bienvenido a Java"` por la pantalla.

Si el proyecto lo hemos creado desde un entorno integrado como Netbeans, la compilación y ejecución se realizará al ejecutar la opción **RunFile** (Ejecutar archivo) o hacer clic sobre el botón **Ejecutar**.



7.3.- Jerarquía de paquetes.

Para organizar mejor las cosas, un **paquete**, en vez de clases, también puede contener otros paquetes. Es decir, podemos hacer **subpaquetes** de los paquetes y subpaquetes de los subpaquetes y así sucesivamente. Esto permite agrupar paquetes relacionados en un paquete más grande. Por ejemplo, si quiero dividir mis clases de ejemplos en ejemplos básicos y ejemplos avanzados, puedo poner más niveles de paquetes separando por puntos:

```
package ejemplos.basicos;  
package ejemplos.avanzados;
```

A nivel de sistema operativo, tendríamos que crear los subdirectorios `basicos` y `avanzados` dentro del directorio `ejemplos`, y meter ahí las clases que correspondan.

Para compilar, en el directorio del proyecto habría que compilar poniendo todo el path hasta llegar a la clase. Es decir, el nombre de la clase va con todos los paquetes separados por puntos, por ejemplo `ejemplos.basicos.Bienvenida`.

La estructura de directorios en el sistema operativo cuando usamos subpaquetes sería:

```
//Proyecto_Bienvenida/ejemplos/basicos/HolaMundo.java
```

Y la compilación y ejecución sería:

```
$ cd //Proyecto_Bienvenida  
$ javac ejemplos/basicos/Bienvenida.java  
$ java ejemplos/basicos/Bienvenida  
Hola Mundo
```

La Biblioteca de Clases de Java se organiza haciendo uso de esta jerarquía de paquetes. Así por ejemplo, si quiero acceder a la clase `Date`, tendré que importarla indicando su ruta completa, o sea, `java.util.Date`, así:

```
import java.util.Date;
```

Citas para pensar

Tan bueno como es heredar una biblioteca, es mejor coleccionar una.

Augustine Birrel.

7.4.- Librerías Java.

Cuando descargamos el entorno de compilación y ejecución de Java, obtenemos la API de Java. Como ya sabemos, se trata de un conjunto de bibliotecas que nos proporciona paquetes de clases útiles para nuestros programas.

Utilizar las clases y métodos de la Biblioteca de Java nos va a ayudar a reducir el tiempo de desarrollo considerablemente, por lo que es importante que aprendamos a consultarla y conozcamos las clases más utilizadas.

Los paquetes más importantes que ofrece el lenguaje Java son:

`java.io`. Contiene las clases que gestionan la entrada y salida, ya sea para manipular ficheros, leer o escribir en pantalla, en memoria, etc. Este paquete contiene por ejemplo la clase `BufferedReader` que se utiliza para la entrada por teclado.

`java.lang`. Contiene las clases básicas del lenguaje. Este paquete no es necesario importarlo, ya que es importado automáticamente por el entorno de ejecución. En este paquete se encuentra la clase `Object`, que sirve como raíz para la jerarquía de clases de Java, o la clase `System` que ya hemos utilizado en algunos ejemplos y que representa al sistema en el que se está ejecutando la aplicación. También podemos encontrar en este paquete las clases que "envuelven" los tipos primitivos de datos. Lo que proporciona una serie de métodos para cada tipo de dato de utilidad, como por ejemplo las conversiones de datos.

`java.util`. Biblioteca de clases de utilidad general para el programador. Este paquete contiene por ejemplo la clase `Scanner` utilizada para la entrada por teclado de diferentes tipos de datos, la clase `Date`, para el tratamiento de fechas, etc.

`java.math`. Contiene herramientas para manipulaciones matemáticas.

`java.awt`. Incluye las clases relacionadas con la construcción de interfaces de usuario, es decir, las que nos permiten construir ventanas, cajas de texto, botones, etc. Algunas de las clases que podemos encontrar en este paquete son `Button`, `TextField`, `Frame`, `Label`, etc.

`java.swing`. Contiene otro conjunto de clases para la construcción de interfaces avanzadas de usuario. Los componentes que se engloban dentro de este paquete se denominan componentes `Swing`, y suponen una alternativa mucho más potente que `AWT` para construir interfaces de usuario.

`java.net`. Conjunto de clases para la programación en la red local e Internet.

`java.sql`. Contiene las clases necesarias para programar en Java el acceso a las bases de datos.

`java.security`. Biblioteca de clases para implementar mecanismos de seguridad.

Como se puede comprobar Java ofrece una completa jerarquía de clases organizadas a través de paquetes.

Para saber más

En el siguiente enlace puedes acceder a la información oficial sobre la Biblioteca de Clases de Java (está en Inglés).

[Información oficial sobre la Biblioteca de Clases de Java.](#)

En el siguiente vídeo se explica de manera interesante el paquete básico `java.lang`:

[LINK VIDEO](#)

8.- Programación de la consola: entrada y salida de la información.

Caso práctico

Juan va a realizar algunas pruebas con el proyecto de la Clínica Veterinaria. Le comenta a **María** que lo que tienen ahora mismo es la estructura básica del proyecto, que básicamente se trata de la definición de algunas clases y objetos que se van a utilizar. Van a necesitar introducir datos por pantalla para ver cómo se comportan esos objetos, y mostrar por pantalla el resultado de manipularlos. Para ello utilizarán las clases `System` y `Scanner`.

—Estas clases ya las hemos utilizado anteriormente, están en los paquetes `java.lang` y `java.util`, respectivamente, observa en el siguiente código cómo las utilizo —dice **Juan**.

—De acuerdo, enséñame ese código —comenta **María**.

Los programas a veces necesitan acceder a los recursos del sistema, como por ejemplo los dispositivos de entrada/salida estándar, para recoger datos de teclado o mostrar datos por pantalla.

En Java, la entrada por teclado y la salida de información por pantalla se hace mediante la clase `System` del paquete `java.lang` de la Biblioteca de Clases de Java.

Como cualquier otra clase, está compuesta de métodos y atributos. Los atributos de la clase `System` son tres objetos que se utilizan para la entrada y salida estándar. Estos objetos son los siguientes:

`System.in`. Entrada estándar: teclado.

`System.out`. Salida estándar: pantalla.

`System.err`. Salida de error estándar, se produce también por pantalla, pero se implementa como un fichero distinto al anterior para distinguir la salida normal del programa de los mensajes de error. Se utiliza para mostrar mensajes de error.

No se pueden crear objetos a partir de la clase `System`, sino que se utiliza directamente llamando a cualquiera de sus métodos con el operador de manipulación de objetos, es decir, el operador punto (`.`):

```
System.out.println("Bienvenido a Java");
```

Para saber más

En el siguiente enlace puedes consultar los atributos y métodos de la clase `System` del paquete `java.lang` perteneciente a la Biblioteca de Clases de Java:

[LINK](#)

Autoevaluación

Texto de la pregunta tipo verdadero o falso:

La clase `System` del paquete `java.io`, como cualquier clase, está formada por métodos y atributos, y además es una clase que no se puede instanciar, sino que se utiliza directamente.

Verdadero. ☐ Falso. ☐

8.1.- Conceptos sobre la clase System.

La lectura por teclado es muy importante cuando empezamos a hacer nuestros primeros programas. Para entender mejor en qué consiste la clase `System`, y en particular el objeto `System.in` vamos a describirlo más detenidamente.

En el apartado anterior hemos dicho que `System.in` es un atributo de la clase `System`, que está dentro del paquete `java.lang`. Pero además, si consultamos la Biblioteca de Clases de Java, nos damos cuenta que es un objeto, y como todos los objetos debe ser instanciado. En efecto, volviendo a consultar la biblioteca de clases nos damos cuenta que `System.in` es una instancia de una clase de java que se llama `InputStream`.

En Java, `InputStream` nos permite leer en bytes, desde teclado, un archivo o cualquier otro dispositivo de entrada. Con esta clase podemos utilizar por ejemplo el método `read()` que permite leer un byte de la entrada o `skip(long n)`, que salta `n` bytes de la entrada. Pero lo que realmente nos interesa es poder leer texto o números, no bytes, para hacernos más cómoda la entrada de datos. Para ello se utilizan las clases:

`InputStreamReader`. Convierte los bytes leídos en caracteres. Particularmente, nos va a servir para convertir el objeto `System.in` en otro tipo de objeto que nos permita leer caracteres.

`BufferedReader`. Lee hasta un fin de línea. Esta es la clase que nos interesa utilizar, pues tiene un método `readLine()` que nos va a permitir leer caracteres hasta el final de línea.

La forma de instanciar estas clases para usarlas con `System.in` es la siguiente:

```
InputStreamReader isr = new InputStreamReader(System.in);
BufferedReader br = new BufferedReader (isr);
```

En el código anterior hemos creado un `InputStreamReader` a partir de `System.in` y pasamos dicho `InputStreamReader` al constructor de `BufferedReader`. El resultado es que las lecturas que hagamos con el objeto `br` son en realidad realizadas sobre `System.in`, pero con la ventaja de que podemos leer una línea completa. Así, por ejemplo, si escribimos una `A`, con:

```
String cadena = br.readLine();
```

Obtendremos en cadena una "A".

Sin embargo, seguimos necesitando hacer la conversión si queremos leer números. Por ejemplo, si escribimos un entero 32, en cadena obtendremos "32". Si recordamos, para convertir cadenas de texto a enteros se utiliza el método estático `parseInt()` de la clase `Integer`, con lo cual la lectura la haríamos así:

```
int numero = Integer.parseInt (br.readLine());
```


8.2.- Entrada por teclado. Clase System.

A continuación vamos a ver un ejemplo de cómo utilizar la clase `System` para la entrada de datos por teclado en Java.

Como ya hemos visto en unidades anteriores, para compilar y ejecutar el ejemplo puedes utilizar las órdenes `javac` y `java`, o bien crear un nuevo proyecto en Netbeans y copiar el código que se proporciona en el archivo anterior.

Código de entrada por teclado con la clase `System`.

Observa que hemos metido el código entre excepciones `try-catch`. Cuando en nuestro programa falla algo, por ejemplo la conversión de un `String` a `int`, Java nos avisa lanzando excepciones. Si "capturamos" esa excepción en nuestro programa, podemos avisar al usuario de qué ha pasado. Esto es conveniente porque si no tratamos la `System` excepción seguramente el programa se pare y no siga ejecutándose. El control de excepciones lo veremos en unidades posteriores, ahora sólo nos basta saber que en las llaves del `try` colocamos el código que puede fallar y en las llaves del `catch` el tratamiento de la excepción.

Para saber más

Te proponemos el siguiente enlace con un descriptivo vídeo sobre cómo capturar datos desde el teclado:

[LINK VIDEO](#)

8.3.- Entrada por teclado. Clase Scanner.

La entrada por teclado que hemos visto en el apartado anterior tiene el inconveniente de que sólo podemos leer de manera fácil tipos de datos `String`. Si queremos leer otros tipos de datos deberemos convertir la cadena de texto leída en esos tipos de datos.

El kit de Desarrollo de Java, a partir de su versión 1.5, incorpora la clase `java.util.Scanner`, la cual permite leer tipos de datos `String`, `int`, `long`, etc., a través de la consola de la aplicación. Por ejemplo para leer un tipo de datos entero por teclado sería:

```
Scanner teclado = new Scanner (System.in);  
int i = teclado.nextInt ();
```

O bien esta otra instrucción para leer una línea completa, incluido texto, números o lo que sea:

```
String cadena = teclado.nextLine();
```

En las instrucciones anteriores hemos creado un objeto de la clase `Scanner` llamado `teclado` utilizando el constructor de la clase, al cual le hemos pasado como parámetro la entrada básica del sistema `System.in` que por defecto está asociada al teclado.

Para conocer cómo funciona un objeto de la clase `Scanner` te proporcionamos el siguiente ejemplo:

[Código de entrada por teclado con la clase Scanner.](#)

Para saber más

Si quieres conocer algo más sobre la clase `Scanner` puedes consultar el siguiente enlace:

[Capturar datos desde teclado con Scanner.](#)

8.4.- Salida por pantalla.

La salida por pantalla en Java se hace con el objeto `System.out`. Este objeto es una instancia de la clase `PrintStream` del paquete `java.lang`. Si miramos la API de `PrintStream` obtendremos la variedad de métodos para mostrar datos por pantalla, algunos de estos son:

```
void print(String s): Escribe una cadena de
texto.
void println(String x): Escribe una cadena de texto y termina la línea.
void printf(String format, Object... args): Escribe una cadena de texto
utilizando formato.
```

En la orden `print` y `println`, cuando queramos escribir un mensaje y el valor de una variable debemos utilizar el operador de concatenación de cadenas (+), por ejemplo:

```
System.out.println("Bienvenido, " + nombre);
```

Escribe el mensaje de "Bienvenido, Carlos", si el valor de la variable `nombre` es Carlos.

Las órdenes `print` y `println` todas las variables que escriben las consideran como cadenas de texto sin formato, por ejemplo, no sería posible indicar que escriba un número decimal con dos cifras decimales o redondear las cifras, o escribir los puntos de los miles, por ejemplo. Para ello se utiliza la orden `printf()`.

La orden `printf()` utiliza unos códigos de conversión para indicar si el contenido a mostrar de qué tipo es. Estos códigos se caracterizan porque llevan delante el símbolo %, algunos de ellos son:

```
%c: Escribe un carácter.
%s: Escribe una cadena de texto.
%d: Escribe un entero.
%f: Escribe un número en punto flotante.
%e: Escribe un número en punto flotante en notación científica.
```

Por ejemplo, si queremos escribir el número `float` 12345.1684 con el punto de los miles y sólo dos cifras decimales la orden sería:

```
System.out.printf("% ,.2f\n", 12345.1684);
```

Esta orden mostraría el número 12.345,17 por pantalla.

Estas órdenes pueden utilizar las secuencias de escape que vimos en unidades anteriores, como `"\n"` para crear un salto de línea, `"\t"` para introducir un salto de tabulación en el texto, etc.

Para saber más

Si quieres conocer algo más sobre la orden `printf()` en el siguiente enlace tienes varios ejemplos de utilización:

- Salida de datos con la instrucción `printf()`: [LINK](#)
- Más ejemplos de formateo con la función `printf()`: [LINK](#)

8.5.- Salida de error.

La salida de error está representada por el objeto `System.err`. Este objeto es también una instancia de la clase `PrintStream`, por lo que podemos utilizar los mismos métodos vistos anteriormente.

No parece muy útil utilizar `out` y `err` si su destino es la misma pantalla, o al menos en el caso de la consola del sistema donde las dos salidas son representadas con el mismo color y no notamos diferencia alguna. En cambio en la consola de varios entornos integrados de desarrollo como NetBeans o Eclipse la salida de `err` se ve en un color diferente. Teniendo el siguiente código:

La salida de este ejemplo en Netbeans es:

Como vemos en un entorno como Netbeans, utilizar las dos salidas nos puede ayudar a una mejor depuración del código.

AUTOEVALUACIÓN

Relaciona cada clase con su función, escribiendo el número asociado a la función en el hueco correspondiente.

Ejercicio de relacionar

Clase.	Relación.	Función.
Scanner	<input type="text"/>	1. Convierte los bytes leídos en caracteres.
PrintStream	<input type="text"/>	2. Lee hasta un fin de línea.
InputStreamReader	<input type="text"/>	3. Lee diferentes tipos de datos desde la consola de la aplicación.
BufferedReader	<input type="text"/>	4. Contiene varios métodos para mostrar datos por pantalla.



Enviar

Anexo.- Licencias de recursos.

Licencias de recursos utilizados en la Unidad de Tr

Recurso (1)	Datos del recurso (1)	Recurso (2)	
	Autoría: f.e.weaver. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/frankweaver/5602423401/		Autoría Licenci Proced http://w 418813
	Autoría: PetsitUSA Pet Sitter Directory. Licencia: CC-by-sa. Procedencia: http://www.flickr.com/photos/petsitusa/3579542904/		Autoría Licenci Proced http://w 348058
	Autoría: helena.40proof. Licencia: CC by-sa. Procedencia: http://www.flickr.com/photos/76099968@N00/557449861/		Autoría Licenci Proced http://c File:Cir
	Autoría: in pulverem reverteris. Licencia: CC-by-sa. Procedencia: http://www.flickr.com/photos/aepedraza/4192197633/		Autoría Licenci Proced http://c _Srinat
	Autoría: Ed Yourdon. Licencia: CC-by-sa. Procedencia: http://www.flickr.com/photos/yourdon/3564741632/		Autoría Licenci Proced http://w 561052
	Autoría: rameshng. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/rameshng/5930493923/		Autoría Licenci Proced http://w 201966
	Autoría: eneas. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/eneas/2767689552/		Autoría Licenci Proced http://w 549854
	Autoría: Seudonimoanonimo. Licencia: CC-by.		Autoría Licenci

	<p>Procedencia: http://www.flickr.com/photos/22890631@N03/3344697018/</p>		<p>Proced http://w 270165</p>
	<p>Autoría: jmereleo. Licencia: CC by-sa. Procedencia: http://www.flickr.com/photos/atalaya/1015839107/</p>		<p>Autoría Licenci Proced http://w 276854</p>
	<p>Autoría: npslibrarian. Licencia: CC by-sa. Procedencia: http://www.flickr.com/photos/npslibrarian/2105037982/</p>		<p>Autoría Licenci Proced http://w 343167</p>
	<p>Autoría: Mª Flor Moncada Añón. Licencia: GNU. Procedencia: Montaje sobre Captura de pantalla de Ubuntu.</p>		<p>Autoría Licenci Proced http://w 453660</p>
	<p>Autoría: Mª Flor Moncada Añón. Licencia: GNU. Procedencia: Montaje sobre Captura de pantalla de Ubuntu.</p>		<p>Autoría Licenci Proced http://w 326764</p>
	<p>Autoría: mpeterke. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/mpeterke/3546334679/</p>		<p>Autoría Licenci Proced Oracle.</p>
	<p>Autoría: Oracle. Licencia: Copyright (cita). Procedencia: Captura de pantalla de la web de Oracle.</p>		<p>Autoría Licenci Proced Oracle.</p>
	<p>Autoría: Ameya arsekar. Licencia: CC by-sa. Procedencia: http://commons.wikimedia.org/wiki/File:Error.gif</p>		<p>Autoría Licenci Proced aplicac propiec GNU G</p>
	<p>Autoría: Sun Microsystems. Licencia: Uso educativo-nc.</p>		

Procedencia: Captura de pantalla de la aplicación NetBeans, propiedad Sun Microsystems, bajo licencia GNU GPL v2.



