ikastaroak.ulhi.net

PROG06.- Desarrollo de clases.

114-146 minutos

Tras su primer contacto con la **Programación Orientada a Objetos**, **María** está mucho más convencida de los beneficios que se pueden obtener utilizando este tipo de metodología. Las explicaciones de **Juan** y su breve experiencia con los **objetos** le hacen ahora ver el mundo de la programación desde otro punto de vista. Este nuevo método de trabajo les va a permitir distribuir las tareas de desarrollo de una manera más eficiente y poder adaptarse a cambios, correcciones y mejoras con mucho menos esfuerzo a la hora de modificar código.

María está convencida de que es el momento de empezar a escribir el código de sus propias **clases** para, a continuación, pasarlas a otros compañeros que las puedan usarlas. Una vez que otras personas reciban esas clases, podrán desentenderse de cómo están hechas por dentro (principios de **encapsulamiento** y **ocultación** de información) y limitarse únicamente a utilizarlas a través de sus **métodos**.

Para llevar a cabo esta labor **Juan** va a proseguir con las explicaciones con las que empezó cuando comenzaron a trabajar con objetos e introdujo el concepto de clase:

- Ha llegado el momento de empezar a desarrollar nuestras propias clases -le dice **Juan**.

-Genial, vamos a comenzar a desarrollar la **primera biblioteca** de clases para **BK Programación**, ¿no es así?

-Bueno, ésa es la idea. A ver si tenemos nuestro primer paquete de clases, que podríamos llamar `com.bkprogramacion`.

-Pues venga... ¡vamos allá!

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 **métodos** 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.

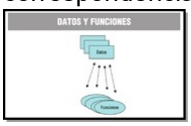
La Programación Orientada a Objetos es un sistema o conjunto de reglas que nos ayudan a descomponer la aplicación en objetos. A menudo se trata de representar las entidades y objetos que nos encontramos en el mundo real mediante componentes de una aplicación. Es decir, debemos establecer una correspondencia 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.

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

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

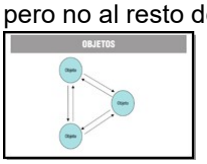
En la Programación Estructurada, el programa estaba compuesto por un conjunto de **datos y métodos "globales".** El término global significaba que eran accesibles por todo el programa, pudiendo ser llamados en cualquier ubicación de la aplicación. Dentro de los métodos se situaban las instrucciones del programa que manipulaban los datos. **Métodos 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 los métodos, éstos eran visibles en toda la aplicación. Ello ocasionaba que cualquier modificación en los datos podía requerir la modificación en todos los métodos 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 los métodos 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 los métodos definidos 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**.

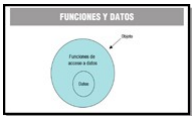
Según lo que hemos visto hasta ahora, **un objeto es cualquier entidad que podemos ver o apreciar.** El concepto fundamental de la Programación Orientada a Objetos son, precisamente, los objetos. Pero ¿qué beneficios aporta la utilización de objetos? Fundamentalmente la posibilidad de representar el 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.

> Primero resuelve el problema. Entonces, escribe el código.
>
> ***John Johnson***

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.

Una panorámica de la evolución de los lenguajes de programación orientados a objetos hasta llegar a los utilizados actualmente es la siguiente:

- **Simula (1962)**. El primer lenguaje con objetos fue B1000 en 1961, seguido por Sketchpad en 1962, el cual contenía clones o copias de objetos. Sin embargo, fue Simula el primer lenguaje que introdujo el concepto de clase, como elemento que incorpora datos y las operaciones sobre esos datos. En 1967 surgió Simula 67 que incorporaba un mayor número de tipos de datos, además del apoyo a objetos.
- **SmallTalk (1972)**. Basado en Simula 67, la primera versión fue Smalltalk 72, a la que siguió Smalltalk 76, versión totalmente orientada a objetos. Se caracteriza por soportar las principales propiedades de la Programación Orientada a Objetos y por poseer un entorno que facilita el rápido desarrollo de aplicaciones. El Modelo-Vista-Controlador () fue una importante contribución de este lenguaje al mundo de la programación. El lenguaje Smalltalk ha influido sobre otros muchos lenguajes como C++ y Java.
- **C++ (1985).** C++ fue diseñado por Bjarne Stoustrup en los laboratorios donde trabajaba, entre 1982 y 1985. Lenguaje que deriva del C, al que añade una serie de mecanismos que le convierten en un lenguaje orientado a objetos. No tiene recolector de basura automática, lo que obliga a utilizar un destructor de objetos no utilizados. En este lenguaje es donde aparece el concepto de clase tal y como lo conocemos actualmente, como un conjunto de datos y funciones que los manipulan.
- **Eiffel (1986)**. Creado en 1985 por Bertrand Meyer, recibe su nombre en honor a la famosa torre de París. Tiene una sintaxis similar a C. Soporta todas las propiedades fundamentales de los objetos, utilizado sobre todo en ambientes universitarios y de investigación. Entre sus características destaca la posibilidad de traducción de código Eiffel a Lenguaje C. Aunque es un lenguaje bastante potente, no logró la aceptación de C++ y Java.
- **Java (1995)**. Diseñado por Gosling de Sun Microsystems a finales de 1995. Es un lenguaje orientado a objetos diseñado desde cero, que recibe muchas influencias de C++. Como sabemos, se caracteriza porque produce un bytecode que posteriormente es interpretado por la máquina virtual. La revolución de Internet ha influido mucho en el auge de Java.
- **C# (2000)**. El lenguaje C#, también es conocido como Sharp. Fue creado por Microsoft, como una ampliación de C con orientación a objetos. Está basado en C++ y en Java. Una de sus principales ventajas que evita muchos de los

problemas de diseño de C++.

Relaciona los lenguajes de programación indicados con la característica correspondiente, escribiendo el número asociado a la característica en el hueco correspondiente.

El equipo de trabajo que han formado **María** y **Juan** parece funcionar bastante bien. El proyecto que tienen entre manos para la aplicación informática de la clínica veterinaria empieza a tomar forma respecto a los requisitos que el cliente necesita. Es el momento de que empiecen a pensar en los distintos objetos con los que pueden modelar la información cuyo tratamiento desean automatizar. Muchos de estos objetos no van a ser proporcionados por las bibliotecas de Java, de manera que habrá que "fabricarlos", es decir, van a tener que diseñar e implementar las clases (los "moldes" o "plantillas") que les permitirán crear los objetos que van a necesitar. Ha llegado el momento de enfrentarse al **concepto de clase**.

Como ya has visto en anteriores unidades, las **clases** representan un **tipo de dato complejo** y están compuestas por **atributos** y **métodos**. A diferencia de los arrays, las clases agrupan datos de diferentes tipos que se denominan atributos y también métodos que nos permiten trabajar con esos atributos.

Una **clase** por tanto, especifica las características comunes de un conjunto de **objetos**. Sin embargo, cuando queramos utilizar ese tipo de dato en nuestros programas tendremos que crear un objeto. De esta forma los programas que escribas estarán formados por un conjunto de **clases** a partir de las cuales irás creando **objetos** que se interrelacionarán unos con otros. En muchos casos también se habla de las clases como de las plantillas o planos a partir de los cuales se crean los objetos.

Además de ellos, veremos que las clases nos van a permitir organizar nuestros programas de otra manera. Es a lo que vamos a llamar programación orientada a objetos (POO). Vamos a utilizar los mismos elementos que hemos utilizado hasta ahora pero organizados en base a clases.

Desde el comienzo del módulo llevas utilizando el concepto de **objeto** para desarrollar tus programas de ejemplo. En las unidades anteriores se ha descrito un objeto como una entidad que contiene **información** y que es capaz de realizar ciertas **operaciones** con esa información. Según los valores que tenga esa información el objeto tendrá un **estado** determinado y según las operaciones que pueda llevar a cabo con esos datos será responsable de un **comportamiento** concreto.

Recuerda que entre las características fundamentales de un objeto se encontraban la **identidad** (los objetos son únicos y por tanto distinguibles entre sí, aunque pueda haber objetos exactamente iguales), un **estado** (los atributos que describen al objeto y los valores que tienen en cada momento) y un determinado **comportamiento** (acciones que se pueden realizar sobre el

objeto).

Algunos ejemplos de objetos que podríamos imaginar podrían ser:

-  Un coche de color rojo, marca SEAT, modelo Toledo, del año 2003. En este ejemplo tenemos una serie de atributos, como el color (en este caso rojo), la marca, el modelo, el año, etc. Así mismo también podríamos imaginar determinadas características como la cantidad de combustible que le queda, o el número de kilómetros recorridos hasta el momento.
- Un coche de color amarillo, marca Opel, modelo Astra, del año 2002.
- Otro coche de color amarillo, marca Opel, modelo Astra y también del año 2002. Se trataría de otro objeto con las mismas propiedades que el anterior, pero sería un segundo objeto.
- Un cocodrilo de cuatro metros de longitud y de veinte años de edad.
- Un círculo de radio 2 centímetros, con centro en las coordenadas (0,0) y relleno de color amarillo.
- Un círculo de radio 3 centímetros, con centro en las coordenadas (1,2) y relleno de color verde.

Si observas los ejemplos anteriores podrás distinguir sin demasiada dificultad al menos tres familias de objetos diferentes, que no tienen nada que ver una con otra:

- Los coches.
- Los círculos.
- Los cocodrilos.

 Es de suponer entonces que cada objeto tendrá determinadas posibilidades de **comportamiento** (**acciones**) dependiendo de la familia a la que pertenezcan. Por ejemplo, en el caso de los **coches** podríamos imaginar acciones como: arrancar, frenar, acelerar, cambiar de marcha, etc. En el caso de los **cocodrilos** podrías imaginar otras acciones como: desplazarse, comer, dormir, cazar, etc. Para el caso del **círculo** se podrían plantear acciones como: cálculo de la superficie del círculo, cálculo de la longitud de la circunferencia que lo rodea, etc.

Por otro lado, también podrías imaginar algunos **atributos** cuyos valores podrían ir cambiando en función de las acciones que se realizaran sobre el objeto: ubicación del coche (coordenadas), velocidad instantánea, kilómetros recorridos, velocidad media, cantidad de combustible en el depósito, etc. En el caso de los cocodrilos podrías imaginar otros atributos como: peso actual, el número de dientes actuales (irá perdiendo algunos a lo largo de su vida), el número de presas que ha cazado hasta el momento, etc.

Como puedes ver, un **objeto** puede ser cualquier cosa que puedas describir en términos de **atributos** y **acciones**.

Un objeto no es más que la representación de cualquier entidad concreta o abstracta que puedas percibir o imaginar y que pueda resultar de utilidad para modelar los elementos el entorno del problema que deseas resolver.

Pregunta

Tenemos un objeto bombilla, de marca ACME, que se puede encender o apagar, que tiene una potencia de 50 vatios y ha costado 3 euros. La bombilla se encuentra en este momento apagada. A partir de esta información, ¿sabrías decir qué atributos y qué acciones (comportamiento) podríamos relacionar con ese objeto bombilla?

Respuestas

Opción 1

Objeto bombilla con atributos potencia (50 vatios), precio (3 euros), marca (ACME) y estado (apagada). Las acciones que se podrían ejercer sobre el objeto serían encender y apagar.

Opción 2

Objeto bombilla con atributos precio (3 euros), marca (ACME) y apagado. Las acciones que se podrían ejercer sobre el objeto serían encender y apagar.

Opción 3

Objeto bombilla con atributos precio (3 euros), marca (ACME), potencia (50 vatios) y estado (apagada). No se puede ejercer ninguna acción sobre el objeto.

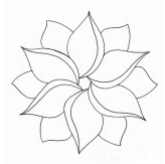
Opción 4

Se trata de un objeto bombilla cuyas posibles acciones son encender, apagar y arreglar. Sus atributos serían los mismos que en el caso a).

Está claro que dentro de un mismo programa tendrás la oportunidad de encontrar decenas, cientos o incluso miles de objetos. En algunos casos no se parecerán en nada unos a otros, pero también podrás observar que habrá muchos que tengan un gran parecido, compartiendo un mismo comportamiento y unos mismos atributos. Habrá muchos objetos que sólo se diferenciaran por los valores que toman algunos de esos atributos.

Es aquí donde entra en escena el concepto de **clase**. Está claro que no podemos definir la estructura y el comportamiento de cada objeto cada vez que va a ser utilizado dentro de un programa, pues la escritura del código sería una tarea interminable y redundante. La idea es poder disponer de una **plantilla** o **modelo** para cada conjunto de objetos que sean del mismo tipo, es decir, que tengan los mismos atributos y un comportamiento similar.

Una clase consiste en la definición de un tipo de objeto. Se trata de una descripción detallada de cómo van a ser los objetos que pertenezcan a esa clase indicando qué tipo de información contendrán (atributos) y cómo se podrá interactuar con ellos (comportamiento).



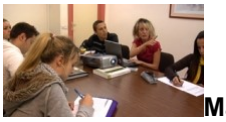
Como ya has visto en unidades anteriores, una clase consiste en un plantilla en la que se especifican:

- Los **atributos** que van a ser comunes a todos los objetos que pertenezcan a esa clase (información).
- Los **métodos** que permiten interactuar con esos objetos (comportamiento).

A partir de este momento podrás hablar ya sin confusión de objetos y de clases, sabiendo que los primeros son instancias concretas de las segundas, que no son más que una abstracción o definición.

Si nos volvemos a fijar en los ejemplos de objetos del apartado anterior podríamos observar que las clases serían lo que clasificamos como "familias" de objetos (coches, cocodrilos y círculos).

En el lenguaje cotidiano de muchos programadores puede ser habitual la confusión entre los términos clase y objeto. Aunque normalmente el contexto nos permite distinguir si nos estamos refiriendo realmente a una clase (definición abstracta) o a un objeto (instancia concreta), hay que tener cuidado con su uso para no dar lugar a interpretaciones erróneas, especialmente durante el proceso de aprendizaje.

 **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 caracterizarán a los objetos que se crean a partir de la clase y donde se define e implementa el comportamiento de dichos objetos; es decir, donde se declaran e implementan los **métodos** para manipular esos objetos.

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

```
[modificadores] class  <NombreClase> [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. En concreto, contendrá las declaraciones de atributos para contener el estado del objeto y los métodos que permitan manipular los objetos creados a partir de esa clase.

Si te fijas en los distintos programas que hemos desarrollado hasta ahora, 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 iremos aprendiendo poco a poco.

A la hora de implementar una clase Java 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.
- 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").
- El archivo debe tener el mismo nombre que la clase si queremos poder utilizarla desde otras clases que se encuentren fuera de ese archivo.

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.

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").

A continuación, vendrá el cuerpo de la clase, encerrado entre llaves {}.

La sintaxis completa de una cabecera queda de la siguiente forma:

```
[modificadores]
class <NombreClase> [extends <NombreSuperClase>]
[implements
<NombreInterface1>] [[implements <NombreInterface2>] ...]
{
```

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

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 veremos 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 y 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 y 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`.

Pregunta

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

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).

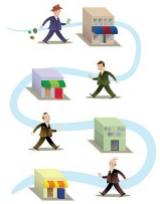
En el ejemplo anterior donde se definía una clase `Punto`, tendríamos los siguientes atributos:

- 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`:

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;
}
```

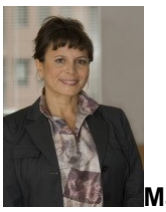
Cada uno de esos métodos puede ser llamado desde cualquier objeto que sea una instancia de la clase `Punto`. Se trata de operaciones que permiten manipular los datos (atributos) contenidos en el objeto bien para calcular otros datos o bien para modificar los propios atributos.

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.

Si te fijas en las clases que hemos utilizado hasta ahora, tanto Math como Array nos permitían utilizar métodos sin crear ningún objeto antes. Todos esos métodos que hemos usado eran métodos estáticos.

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**.

 **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 aparado anterior podríamos 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.

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

```
[modificadores] <tipo> <nombreAtributo>;
```

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 hemos 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 podemos 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 (clase, 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, puertaIzquierda, 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.

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] <tipo> <nombreAtributo>;
```

Vamos a estudiar con detalle cada uno de ellos.

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. No es necesario escribir ninguna palabra reservada. Si no se pone nada se supone que 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	Subclase	Mismo	Otro

	clase		paquete	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.

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.

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`).

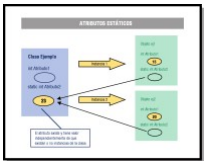
Imagina que estás diseñando un conjunto de clases para trabajar con expresiones geométricas (figuras, superficies, volúmenes, etc.) y necesitas utilizar muy a menudo la constante pi con abundantes cifras significativas, por ejemplo, 3.14159265. Utilizar esa constante literal muy a menudo puede resultar 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;
...
```

Pregunta

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



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.

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.

 **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 como 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).

Pregunta

¿Qué elementos forman la interfaz de un objeto?

Respuestas

Opción 2

Las variables locales de los métodos del objeto.

Opción 4

Los atributos estáticos de la clase.

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 encerado 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.

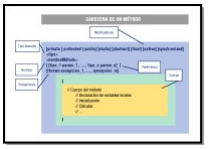
Ahora bien, la declaración de un método puede incluir algunos elementos más. Vamos a estudiar con detalle cada uno de ellos.

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]
   <tipo> <nombreMétodo> ( [<lista_parametros>] )
   [throws <lista_excepciones>]
   ```



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 ()
```

Pregunta

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

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 funcionaría 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

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

```
<tipo> nombreMetodo ( <tipo_1> <nombreParametro_1>,
<tipo_2> <nombreParametro_2>, ..., <tipo_n>
<nombreParametro_n> )
```

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

```
<tipo> <nombreMetodo> ( )
```

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 o arrays) 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.

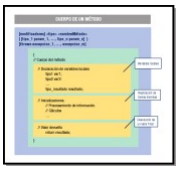
  Es posible utilizar una construcción especial llamada `varargs` (argumentos variables) que permite que un método pueda tener un número variable de parámetros. Para utilizar este mecanismo se colocan unos puntos suspensivos (tres puntos: "...") después del tipo del cual se puede tener una lista variable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

  ```
  <tipo> <nombreMetodo> (<tipo>... <nombre>)
  ```

  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.

  ```
  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 `establecerX`). 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.

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 **calcularSuperfice**, 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**.

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).`

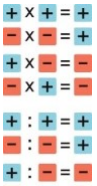
En este caso el compilador no va a generar ningún error pues se cumplen las normas ya que unos métodos son perfectamente distinguibles de otros (a pesar de tener el mismo nombre) gracias a que tienen listas de parámetros diferentes.

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.



Del mismo modo que hemos visto la posibilidad de sobrecargar métodos (disponer de varias versiones de un método con el mismo nombre cambiando su lista de parámetros), podría plantearse también la opción de sobrecargar operadores del lenguaje tales como +, -, *, ( ), <, >, etc. para darles otro significado dependiendo del tipo de objetos con los que vaya a operar.

En algunos casos puede resultar útil para ayudar a mejorar la legibilidad del código, pues esos operadores resultan muy intuitivos y pueden dar una idea rápida de cuál es su funcionamiento.

Un típico ejemplo podría ser el de la sobrecarga de operadores aritméticos como la suma (+) o el producto (*) para operar con fracciones. Si se definen objetos de una clase **Fracción** (que contendrá los atributos numerador y denominador) podrían sobrecargarse los operadores aritméticos (habría que redefinir el operador suma (+) para la suma, el operador asterisco (*) para el producto, etc.) para esta clase y así podrían utilizarse para sumar o multiplicar objetos de tipo **Fracción** mediante el algoritmo específico de suma o de producto del objeto **Fracción** (pues esos operadores no están preparados en el lenguaje para operar con esos objetos).

En algunos lenguajes de programación como por ejemplo C++ o C# se permite la sobrecarga, pero no es algo soportado en todos los lenguajes. ¿Qué sucede en el caso concreto de Java?

El lenguaje Java no soporta la sobrecarga de operadores.

En el ejemplo anterior de los objetos de tipo **Fracción**, habrá que declarar métodos en la clase **Fracción** que se encarguen de realizar esas operaciones, pero no lo podremos hacer sobrecargando los operadores del lenguaje (los símbolos de la suma, resta, producto, etc.). Por ejemplo:

```
public Fraccion sumar (Fraccion sumando)
```

```
public Fraccion multiplicar (Fraccion multiplicando)
```

Y así sucesivamente...

Dado que en este módulo se está utilizando el lenguaje Java para aprender a programar, no podremos hacer uso de esta funcionalidad. Más adelante, cuando aprendas a programar en otros lenguajes, es posible que sí tengas la posibilidad de utilizar este recurso.

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 ocultados 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.

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

Using the this Keyword.

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

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 como Math o Arrays. Iremos viendo más a lo largo del curso.

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`)

Pregunta

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?

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).

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).

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).

 **Juan** ya ha terminado de escribir algunas de las clases de prueba para la aplicación que está comenzando a desarrollar junto con **María**. Es el momento de crear instancias de esas clases (es decir, objetos) para probar si están correctamente implementadas.

La idea de **Juan** es pasar las clases a **María** junto con cierta documentación sobre su interfaz para que ella no tenga que examinar los detalles de implementación de las clases. De esta manera ella escribirá código en el que creará objetos a partir de las clases de **Juan** y a continuación comenzará a utilizar sus miembros públicos. Si todo ha ido bien, **María** habrá hecho uso de las clases de **Juan** sin tener que haber participado directamente en su desarrollo. Si se producen problemas de ejecución (de compilación no deberían producirse porque ya los habría resuelto **Juan**), **María** podrá informar de cuáles han sido esos errores para que **Juan** pueda intentar corregirlos, ya que él es quien sabrá en qué parte del código habrá que tocar.

Una vez que ya tienes implementada una clase con todos sus atributos y métodos, ha llegado el momento de utilizarla, es decir, de instanciar objetos de

esa clase e interaccionar con ellos. En unidades anteriores ya has visto cómo declarar un objeto de una clase determinada, instanciarlo con el operador `new` y utilizar sus métodos y atributos. 

Como ya has visto en unidades anteriores, la declaración de un objeto se realiza exactamente igual que la declaración de una variable de cualquier tipo:

En este caso el tipo será alguna clase que ya hayas implementado o bien alguna de las proporcionadas por la biblioteca de Java o por alguna otra biblioteca escrita por terceros.

Por ejemplo:

```
Punto p1;

Rectangulo r1, r2;

Coche cocheAntonio;

String palabra;
```

Esas variables (p1, r1, r2, cocheAntonio, palabra) en realidad son referencias (también conocidas como punteros o direcciones de memoria) que apuntan (hacen "referencia") a un objeto (una zona de memoria) de la clase indicada en la declaración.

Como ya estudiaste en la unidad dedicada a los objetos, un objeto recién declarado (referencia recién creada) no apunta a nada. Se dice que la referencia está vacía o que es una referencia nula (la variable objeto contiene el valor `null`). Es decir, la variable existe y está preparada para guardar una dirección de memoria que será la zona donde se encuentre el objeto al que hará referencia, pero el objeto aún no existe (no ha sido creado o instanciado). Por tanto se dice que apunta a un objeto nulo o inexistente.

Para que esa variable (referencia) apunte realmente a un objeto (contenga una referencia o dirección de memoria que apunte a una zona de memoria en la que se ha reservado espacio para un objeto) es necesario crear o instanciar el objeto. Para ello se utiliza el operador `new`.

Utilizando la clase **Rectangulo** implementada en ejercicios anteriores, indica como declararías tres objetos (variables) de esa clase llamados **r1, r2, r3** .

Para poder crear un objeto (instancia de una clase) es necesario utilizar el operador `new`, el cual tiene la siguiente sintaxis:

```
nombreObjeto = new <ConstructorClase> ([listaParametros]);
```

 El constructor de una clase (**ConstructorClase**) es un método especial que tiene toda clase y cuyo nombre coincide con el de la clase. Es quien se encarga de crear o construir el objeto, solicitando la reserva de

memoria necesaria para los atributos e inicializándolos a algún valor si fuera necesario. Dado que el constructor es un método más de la clase, podrá tener también su lista de parámetros como tienen todos los métodos.

De la tarea de reservar memoria para la estructura del objeto (sus atributos más alguna otra información de carácter interno para el entorno de ejecución) se encarga el propio entorno de ejecución de Java. Es decir, que por el hecho de ejecutar un método constructor, el entorno sabrá que tiene que realizar una serie de tareas (solicitud de una zona de memoria disponible, reserva de memoria para los atributos, enlace de la variable objeto a esa zona, etc.) y se pondrá rápidamente a desempeñarlas.

Cuando escribas el código de una clase no es necesario que implementes el método constructor si no quieres hacerlo. Java se encarga de dotar de un constructor por omisión (también conocido como **constructor por defecto**) a toda clase. Ese constructor por omisión se ocupará exclusivamente de las tareas de reserva de memoria. Si deseas que el constructor realice otras tareas adicionales, tendrás que escribirlo tú. El constructor por omisión no tiene parámetros.

El constructor por defecto no se ve en el código de una clase. Lo incluirá el compilador de Java al compilar la clase si descubre que no se ha creado ningún método constructor para esa clase.

Algunos ejemplos de instanciación o creación de objetos podrían ser:

```
p1 = new Punto();
r1 = new Rectangulo();
r2 = new Rectangulo;
cocheAntonio = new Coche();
palabra = new String;
```

En el caso de los constructores, si éstos no tienen parámetros, pueden omitirse los paréntesis vacíos.

Un objeto puede ser declarado e instanciado en la misma línea. Por ejemplo:

Ampliar el ejercicio anterior instanciando los objetos **r1**, **r2**, **r3** mediante el constructor por defecto.

Una vez que un objeto ha sido declarado y creado (clase instanciada) ya sí se puede decir que el objeto existe en el entorno de ejecución, y por tanto que puede ser manipulado como un objeto más en el programa, haciéndose uso de sus atributos y sus métodos.

Para acceder a un miembro de un objeto se utiliza el operador **punto** (.) del siguiente modo:

```
<nombreObjeto>.<nombreMiembro>
```

Donde `<nombreMiembro>` será el nombre de algún miembro del objeto (atributo o método) al cual se tenga acceso.

Por ejemplo, en el caso de los objetos de tipo `Punto` que hemos declarado e instanciado en los apartados anteriores, podríamos acceder a sus miembros de la siguiente manera siempre que los hayamos declarado como públicos:

```
Punto p1, p2, p3;
p1 = new Punto();
p1.x = 5;
p1.y = 6;
```

```
System.out.printf("p1.x: %d\np1.y: %d\n", p1.x, p1.y);
System.out.printf("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());
p1.establecerX(25);
p1.establecerX(30);
System.out.printf("p1.x: %d\np1.y: %d\n", p1.obtenerX(), p1.obtenerY());
```

Es decir, colocando el operador **punto** (.) a continuación del nombre del objeto

y seguido del nombre del miembro al que se desea acceder.

Utilizar el ejemplo de los rectángulos para crear un rectángulo **r1**, asignarle los valores x1 = 0, y1 = 0, x2 = 10, y2 = 10, calcular su área y su perímetro y mostrarlos en pantalla.

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.

Pregunta

¿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?

Respuestas

Opción 1

Método constructor. Su nombre dentro de la clase será `constructor`.

Opción 2

Método inicializador. Su nombre dentro de la clase será el mismo nombre que tenga la clase.

Opción 3

Método constructor. Su nombre dentro de la clase será el mismo nombre que tenga la clase.

Opción 4

Método constructor. Su nombre dentro de la clase será `new` .

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).

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**.

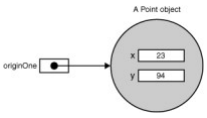
Si defines constructores personalizados para una clase, el constructor por defecto (sin parámetros) para esa clase deja de ser generado por el compilador, de manera que tendrás que crearlo tú si quieres poder utilizarlo.

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`.



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.

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.

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`.

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

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`.

Pregunta

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?

Respuestas

Opción 1

Sí es posible. El nombre del método sería `finalize()`.

Opción 2

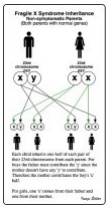
No es posible disponer de un método destructor en una clase Java.

Opción 3

Sí es posible. El nombre del método sería `destructor ()`.

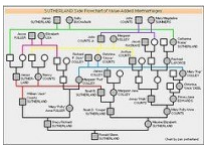
Opción 4

Sí es posible. El nombre del método sería ~nombreClase , como en el lenguaje C++.



María y **Juan** ya han implementado algunos objetos con atributos y métodos tanto de objeto como de clase (estáticos), les han incluido constructores y los han utilizado desde otros objetos. Se consideran relativamente competentes para empezar a desarrollar pequeñas aplicaciones basadas en la metodología de la **Programación Orientada a Objetos**. **Ada** sigue muy satisfecha con estos avances y los ha felicitado por su esfuerzo. Pero también les ha dicho lo siguiente:

-Perfecto, ya sabéis trabajar con clases y con objetos, y cómo utilizar sus métodos, sus atributos y sus constructores. Sois capaces de trabajar con la sobrecarga y conocéis conceptos tales como encapsulación, visibilidad u ocultación de información. Aún nos quedan por ver algunos conceptos más dentro de la Programación Orientada a Objetos. Los iremos viendo poco a poco... Por ejemplo, ¿habéis oído hablar alguna vez sobre la herencia...?



La **herencia** es uno de los conceptos fundamentales que introduce la programación orientada a objetos. La idea fundamental es permitir crear nuevas clases aprovechando las características (atributos y métodos) de otras clases ya creadas evitando así tener que volver a definir esas características (**reutilización**).

A una clase que hereda de otra se le llama **subclase** o **clase hija** y aquella de la que se hereda es conocida como **superclase** o **clase padre**. También se puede hablar en general de **clases descendientes** o **clases ascendientes**. Al heredar, la subclase adquiere todas las características (atributos y métodos) de su superclase, aunque algunas de ellas pueden ser sobrescritas o modificadas dentro de la subclase (a eso se le suele llamar **especialización**).

Una clase puede heredar de otra que a su vez ha podido heredar de una tercera y así sucesivamente. Esto significa que las clases van tomando todas las características de sus clases ascendientes (no sólo de su superclase o clase padre inmediata) a lo largo de toda la rama del árbol de la jerarquía de clases en la que se encuentre.

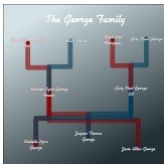
Imagina que quieres modelar el funcionamiento de algunos vehículos para

trabajar con ellos en un programa de simulación. Lo primero que haces es pensar en una clase **Vehículo** que tendrá un conjunto de atributos (por ejemplo: posición actual, velocidad actual y velocidad máxima que puede alcanzar el vehículo) y de métodos (por ejemplo: detener, acelerar, frenar, establecerDirección, establecer sentido).

Dado que vas a trabajar con muchos tipos de vehículos, no tendrás suficiente con esas características, así que seguramente vas a necesitar nuevas clases que las incorporen. Pero las características básicas que has definido en la clase **Vehículo** van a ser compartidas por cualquier nuevo vehículo que vayas a modelar. Esto significa que si creas otra clase podrías heredar de **Vehículo** todas esos atributos y propiedades y tan solo tendrías que añadir las nuevas.

Si vas a trabajar con vehículos que se desplazan por tierra, agua y aire, tendrás que idear nuevas clases con características adicionales. Por ejemplo, podrías crear una clase **VehiculoTerrestre**, que herede las características de **Vehículo**, pero que también incorpore atributos como el número de ruedas o la altura de los bajos). A su vez, podría idearse una nueva clase que herede de **VehiculoTerrestre** y que incorpore nuevos atributos y métodos como, por ejemplo, una clase **Coche**. Y así sucesivamente con toda la jerarquía de clases heredadas que consideres oportunas para representar lo mejor posible el entorno y la información sobre la que van a trabajar tus programas.

¿Cómo se indica en Java que una clase hereda de otra? Para indicar que una clase hereda de otra es necesario utilizar la palabra reservada `extends` junto con el nombre de la clase de la que se quieren heredar sus características:




```
class <NombreClase> extends <nombreSuperClase> {
...
}
```

En el ejemplo anterior de los vehículos, la clase **VehiculoTerrestre** podría quedar así al ser declarada:


```
class VehiculoTerrestre extends Vehículo {
...
}
```

Y en el caso de la clase **Coche**:

```
class Coche extends VehículoTerrestre {
...
}
```

En unidades posteriores estudiarás detalladamente cómo crear una **jerarquía de clases** y qué relación existe entre la herencia y los distintos modificadores de clases, atributos y métodos. Por ahora es suficiente con que entiendas el concepto de herencia y sepas reconocer cuándo una clase hereda de otra

(uso de la palabra reservada `extends`).

Puedes comprobar que en las bibliotecas proporcionadas por Java aparecen jerarquías bastante complejas de clases heredadas en las cuales se han ido aprovechando cada uno de los miembros de una clase base para ir construyendo las distintas clases derivadas añadiendo (y a veces modificando) poco a poco nueva funcionalidad. Eso suele suceder en cualquier proyecto de software conforme se van a analizando, descomponiendo y modelando los datos con los que hay que trabajar. La idea es poder representar de una manera eficiente toda la información que es manipulada por el sistema que se desea automatizar. Una jerarquía de clases suele ser una buena forma de hacerlo.

En el caso de Java, cualquier clase con la que trabajes tendrá un ascendiente. Si en la declaración de clase no indicas la clase de la que se hereda (no se incluye un `extends`), el compilador considerará automáticamente que se hereda de la clase `Object`, que es la clase que se encuentra en el nivel superior de toda la jerarquía de clases en Java (y que es la única que no hereda de nadie).

También irás viendo al estudiar distintos componentes de las bibliotecas de Java (por ejemplo en el caso de las interfaces gráficas) que para poder crear objetos basados en las clases proporcionadas por esas bibliotecas tendrás que crear tus propias clases que hereden de algunas de esas clases. Para ellos tendrás que hacer uso de la palabra reservada `extends`.

En Java todas las clases son descendientes (de manera explícita o implícita) de la clase `Object`.

Pregunta

¿De qué objeto hereda cualquier clase en Java?


María y **Juan** ya han terminado por ahora de escribir todas las clases que habían diseñado. Es el momento de organizar adecuadamente todo el código que tienen implementado a lo largo de todas esas clases. **María** recuerda que hace algunos días Juan ya le habló de la posibilidad de organizar las clases en paquetes:

Este sería un buen momento para poner en práctica todo aquello que me enseñaste sobre los paquetes, ¿no?

-La verdad es que tienes razón. Esto hay que organizarlo un poco...-Le contesta **Juan**.


La **encapsulación** de la información dentro de las clases ha permitido llevar a cabo el proceso de ocultación, que es fundamental para el trabajo con clases y objetos. Es posible que conforme vaya aumentando la complejidad de tus aplicaciones necesites que algunas de tus clases puedan tener acceso a parte de la implementación de otras debido a las relaciones

que se establezcan entre ellas a la hora de diseñar tu modelo de datos. En estos casos se puede hablar de un nivel superior de encapsulamiento y ocultación conocido como **empaquetado**.

Un **paquete** consiste en un conjunto de clases relacionadas entre sí y agrupadas bajo un mismo nombre. Normalmente se encuentran en un mismo paquete todas aquellas clases que forman una biblioteca o que reúnen algún tipo de característica en común. Esto la organización de las clases para luego localizar fácilmente aquellas que vayas necesitando.

Los paquetes en Java pueden organizarse jerárquicamente de manera similar a lo que puedes encontrar en la estructura de carpetas en un dispositivo de almacenamiento, donde:

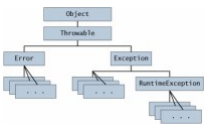
- Las clases serían como los archivos.
- Cada paquete sería como una carpeta que contiene archivos (clases).
- Cada paquete puede además contener otros paquetes (como las carpetas que contienen carpetas).
- Para poder hacer referencia a una clase dentro de una estructura de paquetes, habrá que indicar la trayectoria completa desde el paquete raíz de la jerarquía hasta el paquete en el que se encuentra la clase, indicando por último el nombre de la clase (como el path absoluto de un archivo).

  La estructura de paquetes en Java permite organizar y clasificar las clases, evitando conflictos de nombres y facilitando la ubicación de una clase dentro de una estructura jerárquica.

  Por otro lado, la organización en paquetes permite también el control de acceso a miembros de las clases desde otras clases que estén en el mismo paquete gracias a los modificadores de acceso (recuerda que uno de los modificadores que viste era precisamente el de paquete).

  Las clases que forman parte de la jerarquía de clases de Java se encuentran organizadas en diversos paquetes.

  Todas las clases proporcionadas por Java en sus bibliotecas son miembros de distintos paquetes y se encuentran organizadas jerárquicamente. Dentro de cada paquete habrá un conjunto de clases con algún tipo de relación entre ellas. Se dice que todo ese conjunto de paquetes forman la **API** de Java. Por ejemplo las clases básicas del lenguaje se encuentran en el paquete `java.lang`, las clases de entrada/salida las podrás encontrar en el paquete `java.io` y en el paquete `java.math` podrás observar algunas clases para trabajar con números grandes y de gran precisión.



Es posible acceder a cualquier clase de cualquier paquete (siempre que ese paquete esté disponible en nuestro sistema, obviamente) mediante la calificación completa de la clase dentro de la estructura jerárquica de paquete. Es decir indicando la trayectoria completa de paquetes desde el paquete raíz hasta la propia clase. Eso se puede hacer utilizando el operador **punto**.) para especificar cada subpaquete:

```
paquete_raiz.subpaquete1.subpaquete2. ...
.subpaquete_n.NombreClase
```

Por ejemplo:

En este caso se está haciendo referencia a la clase `String` que se encuentra dentro del paquete `java.lang`. Este paquete contiene las clases elementales para poder desarrollar una aplicación Java.

Otro ejemplo podría ser:

En este otro caso se hace referencia a la clase `Patern` ubicada en el paquete `java.util.regex`, que contiene clases para trabajar con expresiones regulares.

Dado que puede resultar bastante tedioso tener que escribir la trayectoria completa de una clase cada vez que se quiera utilizar, existe la posibilidad de indicar que se desea trabajar con las clases de uno o varios paquetes. De esa manera cuando se vaya a utilizar una clase que pertenezca a uno de esos paquetes no será necesario indicar toda su trayectoria. Para ello se utiliza la sentencia `import` (importar):

```
import paquete_raiz.subpaquete1.subpaquete2. ...
.subpaquete_n.NombreClase;
```

De esta manera a partir de ese momento podrá utilizarse directamente `NombreClase` en lugar de toda su trayectoria completa.

Los ejemplos anteriores quedarían entonces:

```
import java.lang.String;
```

```
import java.util.regex.Patern;
```

Si suponemos que vamos a utilizar varias clases de un mismo paquete, en lugar de hacer un `import` de cada una de ellas, podemos utilizar el **comodín** (símbolo **asterisco**: "`*`") para indicar que queremos importar todas las clases de ese paquete y no sólo una determinada:

```
import java.lang.*;
```

```
import java.util.regex.*;
```

Si un paquete contiene subpaquetes, el comodín no importará las clases de los subpaquetes, tan solo las que haya en el paquete. La importación de las clases contenidas en los subpaquetes habrá que indicarla explícitamente. Por ejemplo:

```
import java.util.*;
```

```
import java.util.regex.*;
```

En este caso se importarán todas las clases del paquete `java.util` (clases `Date, Calendar, Timer`, etc.) y de su subpaquete `java.util.regex` (`Matcher` y `Pattern`), pero las de otros subpaquetes como `java.util.concurrent` o `java.util.jar`.

Por último tan solo indicar que en el caso del paquete `java.lang`, no es necesario realizar importación. El compilador, dada la importancia de este paquete, permite el uso de sus clases sin necesidad de indicar su trayectoria (es como si todo archivo Java incluyera en su primera línea la sentencia `import java.lang.*`).


Al principio de cada archivo .java se puede indicar a qué paquete pertenece mediante la palabra reservada `package` seguida del nombre del paquete:

Por ejemplo:

```
package paqueteEjemplo;
class claseEjemplo {
...
}
```

La sentencia `package` debe ser incluida en cada archivo fuente de cada clase que quieras incluir ese paquete. Si en un archivo fuente hay definidas más de una clase, todas esas clases formarán parte del paquete indicado en la sentencia `package`.

Si al comienzo de un archivo Java no se incluyen ninguna sentencia `package`, el compilador considerará que las clases de ese archivo formarán parte del paquete por omisión (un paquete sin nombre asociado al proyecto).

Para evitar la ambigüedad, dentro de un mismo paquete no puede haber dos clases con el mismo nombre, aunque sí pueden existir clases con el mismo nombre si están en paquetes diferentes. El compilador será capaz de distinguir una clase de otra gracias a que pertenecen a paquetes distintos.

Como ya has visto en unidades anteriores, el nombre de un archivo fuente en Java se construye utilizando el nombre de la clase pública que contiene junto con la extensión .java, pudiendo haber únicamente una clase pública por cada archivo fuente. El nombre de la clase debía coincidir (en mayúsculas y minúsculas) exactamente con el nombre del archivo en el que se encontraba definida. Así, si por ejemplo tenías una clase `Punto` dentro de un archivo `Punto.java`, la compilación daría lugar a un archivo `Punto.class`.

En el caso de los paquetes, la correspondencia es a nivel de directorios o carpetas. Es decir, si la clase `Punto` se encuentra dentro del paquete `prog.figuras`, el archivo `Punto.java` debería encontrarse en la carpeta `prog\figuras`. Para que esto funcione correctamente el compilador ha de ser capaz de localizar todos los paquetes (tanto los estándar de Java como los definidos por otros programadores). Es decir, que el compilador debe tener conocimiento de dónde comienza la estructura de carpetas definida por los paquetes y en la cual se encuentran las clases. Para ello se utiliza el `ClassPath` cuyo funcionamiento habrás estudiado en las primeras unidades de este módulo. Se trata de una variable de entorno que contiene todas las rutas en las que comienzan las estructuras de directorios (distintas jerarquías posibles de paquetes) en las que están contenidas las clases.


Para crear un paquete en Java te recomendamos seguir los siguientes pasos:

1. **Poner un nombre al paquete**. Suele ser habitual utilizar el dominio de Internet de la empresa que ha creado el paquete. Por ejemplo, para el caso de **miempresa.com**, podría utilizarse un nombre de paquete **com.miempresa**.

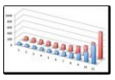
2. **Crear una estructura jerárquica de carpetas equivalente a la estructura de subpaquetes**. La ruta de la raíz de esa estructura jerárquica deberá estar especificada en el **ClassPath** de Java.
3. **Especificar a qué paquete pertenecen la clase** (o clases) de el archivo .java mediante el uso de la sentencia **package** tal y como has visto en el apartado anterior.

   Este proceso ya lo has debido de llevar a cabo en unidades anteriores al compilar y ejecutar clases con paquetes. Estos pasos simplemente son para que te sirvan como recordatorio del procedimiento que debes seguir a ala hora de clasificar, jerarquizar y utilizar tus propias clases.

   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 de 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)`.

Licencias de recursos utilizados en la Unidad de Trabajo.

Recurso (1)	Datos del recurso (1)	Recurso (2)	Datos del recurso (2)
	Autoría: f.e.weaver. Licencia: CC-by. Procedencia: http://www.flickr.com/photos/frankweaver/5602423401/		Autoría: JoshuaDavisPhotografy. Licencia: CC by-sa. Procedencia: http://www.flickr.com/photos/articnomad/418813105/in/photostream/

Autoría: PetsitUSA Pet Sitter Directory.

Licencia: CC-by-sa.

Procedencia: http://www.flickr.com/photos/petsitusa/3579542904/

Autoría: helena.40proof.

Licencia: CC by-sa.

Procedencia: http://www.flickr.com/photos/76099968@N00/557449861/

Autoría: Priority Pet Hospital

Licencia: CC BY-ND 2.0.

Procedencia: http://www.flickr.com/photos/priority_pet_hospital/5735593176/

Autoría: aaipodpics.

Licencia: CC BY-NC-SA 2.0.

Procedencia: http://www.flickr.com/photos/aaipodpics/4046279755/

Autoría: watz.

Licencia: CC BY-NC-SA 2.0.

Procedencia: http://www.flickr.com/photos/watz/5495949974/in/photostream/

Autoría: Sleepy Valley.

Licencia: CC BY-NC-ND 2.0.

Procedencia: http://www.flickr.com/photos/sleepyvalley/5710305175/in/photostream

Autoría: Sleepy Valley.

Licencia: CC BY-NC-ND

Autoría: bellydraft.

Licencia: CC by.

Procedencia: http://www.flickr.com/photos/bellydraft/348058918/

Autoría: Quasar.

Licencia: CC by-sa.

Procedencia: http://commons.wikimedia.org/wiki/File:Cimage.jpg

Autoría: pablosanz.

Licencia: CC BY-NC-SA 2.0

Procedencia: http://www.flickr.com/photos/pablosanz/2229169284/

Autoría: Ministerio de Fomento.

Licencia: CC BY-NC-ND 2.0.

Procedencia: http://www.flickr.com/photos/fomentogob/6099717231/

Autoría: Jo in NZ.

Licencia: CC BY-NC-SA 2.0.

Procedencia: http://www.flickr.com/photos/jo_in_nz/5055562942/

Autoría: Sleepy Valley.

Licencia: CC BY-NC-ND 2.0.

Procedencia: http://www.flickr.com/photos/sleepyvalley/36506974

Autoría: DailyPic.

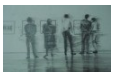
Licencia: CC BY-NC 2.0

2.0.

Procedencia: http://www.flickr.com/photos/sleepyvalley/3800057683/in/photostream/

Autoría: DailyPic.

Licencia: CC BY-NC 2.0



Procedencia: http://www.flickr.com/photos/dailypic/1468681275/in/photostream

Autoría: DailyPic.

Licencia: CC BY-NC 2.0

Procedencia: http://www.flickr.com/photos/dailypic/114312238/in/photostream/

Autoría: Collin Key.

Licencia: CC BY-NC-SA 2.0

Procedencia: http://www.flickr.com/photos/collin_key/3003658750/

Autoría: Mr. Muskrat.

Licencia: CC BY-NC-ND 2.0

Procedencia: http://www.flickr.com/photos/mrmuskrat/3722758392/

Autoría: KoppCorentin.

Licencia: CC BY-NC-ND 2.0

Procedencia: http://www.flickr.com/photos/corentinkopp/6328147027/

Autoría: redteam.

Licencia: CC BY-NC-ND 2.0

Procedencia: http://www.flickr.com/photos/redteam

Procedencia: http://www.flickr.com/photos/dailypic/1920527363/in/photostream/

Autoría: jorel314.

Licencia: CC BY 2.0

Procedencia: http://www.flickr.com/photos/jorel314/3352784321/

Autoría: Josell7

Licencia: CC BY-SA 3.0

Procedencia: Montaje sobre http://es.wikipedia.org/wiki/Archivo:Subprograma.svg

Autoría: Svacher.

Licencia: CC BY-NC-ND 2.0

Procedencia: http://www.flickr.com/photos/trufflepig/2313005883/

Autoría: SHerrington.

Licencia: CC BY-NC-SA 2.0

Procedencia: http://www.flickr.com/photos/s-herrington/222468516/

Autoría: geese.

Licencia: CC BY 2.0

Procedencia: http://www.flickr.com/photos/skhan/233168879/

Autoría: KoppCorentin.

Licencia: CC BY-NC-ND 2.0

Procedencia: http://www.flickr.com/photos/corentinkopp/6328147489/in/set-72157627964248471/

/102847847/

Autoría: ethorson.

Licencia: CC BY-SA 2.0

Procedencia: http://www.flickr.com/photos/ethorson/122310358/sizes/t/in/photostream/

Autoría: eric.delcroix.

Licencia: CC BY-NC-SA 2.0

Procedencia: http://www.flickr.com/photos/eric-delcroix/2674794422/in/photostream/

Autoría: DailyPic.

Licencia: CC BY-NC 2.0

Procedencia: http://www.flickr.com/photos/dailypic/11232715/in/photostream

Autoría: zen.

Licencia: CC BY-NC-SA 2.0

Procedencia: http://www.flickr.com/photos/zen/9055855/

Autoría: ponchoペpenguin.

Licencia: CC BY-NC-SA 2.0

Procedencia: http://www.flickr.com/photos/ponchopenguin/4176263081/

Autoría: lemonhalf

Licencia: CC BY-SA 2.0

Procedencia: http://www.flickr.com/photos/halfbisqued/2353845688/

Autoría: Chris_Carter_

Licencia: CC BY-NC-ND 2.0

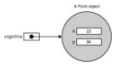
Autoría: blucarbnpinwheel.

Licencia: CC BY-NC-SA 2.0

Procedencia: http://www.flickr.com/photos/blucarbnpinwheel/387267526/

Autoría: Oracle.

Licencia: Copyright (cita).



Procedencia: http://docs.oracle.com/javase/tutorial/java/javaOO/objectcreation.html

Autoría: Image Editor.

Licencia: CC BY 2.0

Procedencia: http://www.flickr.com/photos/11304375@N07/3197825319/

Autoría: Family Art Studio.

Licencia: CC BY 2.0

Procedencia: http://www.flickr.com/photos/familyartstudio/5614659945/

Autoría: ChrisCampbell-CM

Licencia: CC BY-NC-ND 2.0

Procedencia: http://www.flickr.com/photos/footix2/3471012573/

Autoría: Oracle.

Licencia: Copyright (cita).

Procedencia: http://docs.oracle.com/javase/tutorial/essential/exceptions/throwing.html

Autoría: Marcus Mo.

Licencia: CC BY-NC-ND 2.0.

Procedencia:

Procedencia:
http://www.flickr.com
/photos/chris_carter_
/4112202756
/in/photostream/

http://www.flickr.com/photos
/marcusmo/3296047622/