Abstracción y encapsulación

PID_00269657

David García Solórzano

Tiempo mínimo de dedicación recomendado: 4 horas

Universitat Oberta de Catalunya

David García Solórzano

Graduado Superior en Ingeniería en Multimedia e Ingeniero en Informática por la Universitat Ramon Llull desde 2007 y 2008, respectivamente. Es también Doctor por la Universitat Oberta de Catalunya desde 2013, donde realizó una tesis doctoral relacionada con el ámbito del e-learning. Desde 2008 es profesor de la Universitat Oberta de Catalunya en los Estudios de Informática, Multimedia y Telecomunicación.

El encargo y la creación de este recurso de aprendizaje UOC han sido coordinados por el profesor: David García Solórzano (2020)

Primera edición: febrero 2020

Av. Tibidabo, 39-43, 08035 Barcelona
Realización editorial: FUOC

Índice

Introducción

En este módulo veremos dos de los cuatro pilares del paradigma de la programación orientada a objetos: la abstracción y la encapsulación.

Gracias a ellas aprenderemos cómo pensar siguiendo un enfoque *bottom-up*. Como consecuencia, profundizaremos en qué es una clase y cuáles son sus miembros, qué es un objeto, qué caracteriza a un objeto (estado y comportamiento), cómo se instancia, qué son los niveles o modificadores de acceso, cuáles existen y qué implicaciones tienen.

Por último, introduciremos el lenguaje de modelado de sistemas de *software* UML (*Unified Modeling Language*). En este módulo nos centraremos exclusivamente en cómo representar una clase dentro de un diagrama de clases UML.

Salvo que se indique un lenguaje de programación concreto, los ejemplos de codificación están escritos con un lenguaje de programación inventado, es decir, un pseudocódigo. Si tuviéramos que decir a qué lenguaje de programación real se parece el pseudocódigo empleado, diríamos que es parecido a Java (pero sin elementos que dificultan el entendimiento de los ejemplos).

Así pues, deberemos consultar la documentación del lenguaje de programación que deseamos utilizar para ver cómo se codifican los conceptos explicados, así como los ejemplos proporcionados.

Objetivos

El objetivo principal de este módulo es asentar las bases del paradigma de la programación orientada a objetos, concretamente:

1. Ver el proceso de abstracción que hay que seguir a la hora de diseñar una clase y, por ende, para solucionar un problema mediante el paradigma de la programación orientada a objetos.

2. Saber diferenciar entre un objeto y una clase y, al mismo tiempo, comprender la relación que existe entre ellos.

3. Conocer y entender el proceso de encapsulación para agrupar los miembros de una clase, es decir, los atributos y los métodos. Asimismo, comprender los beneficios que la encapsulación aporta.

4. Entender los conceptos de *constructor* y *destructor* de una clase.

5. Conocer la ocultación como un mecanismo que define diferentes niveles de acceso a los miembros de una clase.

6. Saber los conceptos relacionados con un objeto: instancia, estado, comportamiento y mensaje.

7. Saber qué significa y qué implicaciones tiene que un atributo, un método, un constructor o una clase sean estáticos.

8. Tener un primer contacto con el lenguaje de modelado UML.

1. Del problema al diseño de una solución

Cuando nos enfrentamos a un problema, este está formado por objetos (algunos más tangibles y otros menos). Serán estos objetos y las relaciones entre ellos lo que nos permitan diseñar un programa que dé respuesta o solución al problema.

1.1. ¿Qué es un objeto?

Como ya te habrás imaginado, el **objeto** es el elemento principal de la programación orientada objetos y alrededor del cual gira este paradigma, de ahí el nombre.

Aunque acabas de empezar a leer estos materiales, detente un momento. Da igual dónde estés, aparta la mirada de esta página y mira a tu alrededor, ¡pero vuelve, que tienes que seguir leyendo! ¿Qué has visto? Quizás respondas: cosas. Bueno, no está mal, ¿pero puedes ser más concreto?

Ejemplo 1 (objeto) – Buscando objetos

Pongamos que estás en el despacho donde estudias o, mejor, en el sofá del comedor (el estudio no es incompatible con la comodidad). Quizás hayas visto una mesa, una silla, dos lámparas, cinco lápices, un ordenador, un perro, etc. ¿Podemos decir que lo que has visto a tu alrededor son objetos? ¿Sí? ¡Genial!

Un momento, ¿has visto ese coche blanco que acaba de pasar por delante de tu ventana? Efectivamente, es un objeto compuesto de otros muchos objetos: un volante, cuatro ruedas, etc. Correcto, un coche es un objeto tan complejo que está formado o compuesto por otros objetos.

¿Y qué me dices del escritorio que tienes en tu despacho? ¡Claro que sí, también es un objeto! Además está formado por tres cajones, cada uno de los cuales es un objeto.

¡Espera! ¿Acaba de pasar otro coche? Vaya, ¡sí que hay tráfico! Ya has visto dos coches, ya has visto dos objetos similares.

¿Qué es ese ruido? ¡Es tu *smartphone*! Por favor, silencia las notificaciones, ¡estás estudiando! Sí, sí, tienes razón, tu *smartphone* es un objeto que dentro tiene una lista de aplicaciones, y sí, cada aplicación es un objeto.

En este punto seguramente estarás diciéndote: «¡De acuerdo, sé lo que es un objeto en la vida real! Pero, ¿qué se entiende por objeto en el paradigma de la POO?». Para responderte lo haremos de manera formal e informal. La formal nos lleva a la siguiente definición:

Definición formal de objeto

Un objeto en POO representa alguna entidad de la vida real, es decir, alguno de los objetos únicos que pertenecen al problema con el que nos estamos enfrentando y con el que podemos interactuar.

Cada objeto, de igual modo que la entidad de la vida real a la que representa, tiene un estado (definido por unos atributos que tienen unos valores concretos) y un comportamiento (es decir, tiene funcionalidades o sabe hacer unas acciones concretas).

Entidad

A los objetos del mundo real se les suele llamar *entidades* para diferenciarlos de sus homólogos en el mundo de la programación orientada a objetos, llamados *objetos*.

De manera informal podemos decir:

Definición informal de objeto

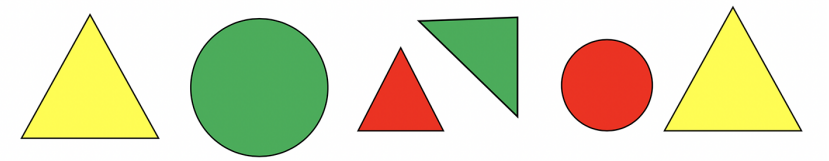
Un objeto en POO es cualquier elemento del mundo real con el que se puede interactuar.

Así pues, el primer coche que ha pasado por delante de tu ventana es un objeto, mientras que el segundo coche (aunque sea del mismo color, marca y modelo que el anterior) es otro objeto diferente.

Para terminar de entender qué es un objeto, veamos el siguiente ejemplo.

Ejemplo 2 (objeto) – Jugando con los objetos

A Marina su mamá Elena le ha puesto los siguientes objetos delante y le dice que diga cuántos objetos hay:



Marina los mira, empieza a contar «uno, dos, tres…» y acaba diciendo «seis». «¡Correcto!» –grita contenta Elena. Marina ve que, aunque a simple vista hay dos objetos idénticos (los de los extremos), estos son dos objetos diferentes.

1.2. ¿Qué caracteriza a un objeto?

Como hemos leído en la definición formal de objeto, todo objeto en la POO tiene un **estado** y un **comportamiento**. Esto es así porque los objetos (o entidades) de la vida real comparten estas dos características.

Definición de estado

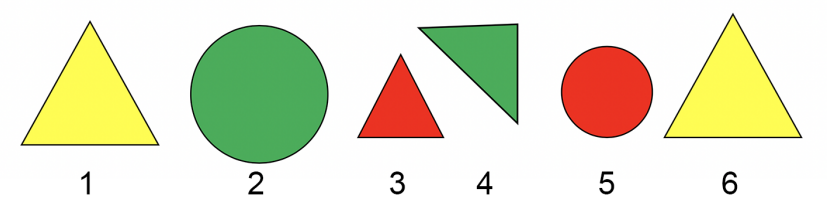
El estado de un objeto viene determinado por los valores que toman en un instante determinado los atributos que definen ese objeto.

Así pues, si un televisor concreto (el objeto) tiene el atributo `canal actual` igual a `5`, ese televisor está en un estado diferente a si tuviera el `canal actual` igual al número `6`.

Veamos el concepto de *estado* utilizando el juego de las formas con el que jugaban Marina y Elena.

Ejemplo 3 (estado de un objeto) – Estado en el juego de las formas

Fíjate en los seis objetos siguientes:



Todas las figuras tienen un atributo o propiedad común: el color. Eso sí, cada objeto tiene un valor asignado para esa propiedad. Por ejemplo, los objetos 1 y 6 son amarillos, mientras que los objetos 2 y 4 son verdes, y los objetos 3 y 5 son rojos. Asimismo, los objetos 1, 3, 4 y 6 tienen tres atributos, cada uno de los cuales representa un lado del triángulo y cuyo valor es igual a la longitud de ese lado. Estas longitudes pueden variar, como es el caso de los objetos 1 y 6 respecto a los objetos 3 y 4. Además, los objetos 1, 3, 4 y 6 tienen otro atributo que representa el ángulo de rotación, siendo su valor 0º para los objetos 1, 3 y 6, y un valor diferente para el objeto 4.

Así pues, las diferencias en los valores de los atributos de un objeto hacen que ese objeto esté en un estado diferente. Los objetos 1 y 6 están en el mismo estado (todos sus atributos tienen los mismos valores), pero los objetos 3 y 4 están en un estado diferente.

Ahora veamos qué se entiende por *comportamiento*:

Definición de comportamiento

Es el conjunto de funcionalidades que un objeto es capaz de realizar. Estas funcionalidades son determinadas por los métodos.

Por ejemplo, las acciones que puede hacer un televisor son: encenderse, apagarse, cambiar de canal, subir y bajar el volumen, etc.

Sigamos con el ejemplo en el que Elena y Marina juegan con objetos de distintas formas.

Ejemplo 4 (comportamiento de un objeto) – Comportamiento en el juego de las formas

Las seis figuras tienen un área, por tanto, son capaces de calcularla a partir de los valores de sus atributos. Si bien es cierto que las figuras 1, 3, 4 y 6 usarán la fórmula $b * h/2$,

las figuras 2 y 5 usarán πr^2. También las seis figuras podrán calcular su perímetro con sus respectivas fórmulas. Además, las figuras 1, 3, 4 y 6 podrán calcular las coordenadas de su baricentro. Así pues, el área y el perímetro son dos métodos que comparten todas las figuras, mientras que el baricentro es exclusivo de los objetos 1, 3, 4 y 6. Del mismo modo, los objetos 1, 3, 4 y 6 pueden rotar, mientras que los objetos 2 y 5, no.

1.3. ¿Qué es una clase?

El concepto *clase* está íntimamente relacionado con el concepto *objeto*.

Definición de clase

Podemos definir informalmente una clase como una plantilla (o esqueleto o plano) a partir de la cual se crean los objetos.

Veamos la relación que existe entre clase y objeto con un ejemplo.

Ejemplo 5 (relación entre clase y objeto) – El televisor de Jose

David va a casa de su amigo Jose y se da cuenta de que tiene el mismo televisor que él. ¿El mismo? El mismo no. Son dos objetos distintos, David tiene su televisor en su casa y Jose el suyo en su casa. ¿Acaso no puedes tocarlos y ver que son dos objetos diferentes? Eso sí, son dos objetos (dos televisores) que son de la misma marca y modelo, por lo tanto, parecen el mismo objeto, ya que cada uno de esos dos televisores ha sido montado a partir de un mismo plano, esqueleto o plantilla y, consecuentemente, ambos tienen los mismos componentes, conexiones y funcionalidades. Ese plano, esqueleto o plantilla es, en términos de programación orientada a objetos, una clase (la clase `Television`, que representa el concepto abstracto de televisor, es decir, las características y acciones comunes de los televisores), mientras que cada televisor es un objeto de esa clase.

Lo mismo ocurre con el siguiente ejemplo.

Ejemplo 6 (relación entre clase y objeto) – El plano de una casa

Elena es arquitecta y le han hecho un encargo. Le han pedido que diseñe una casa unifamiliar con la que urbanizar todo un barrio. En este escenario, el plano que Elena dibuje con todos los detalles de la casa sería en POO la clase, mientras que cada casa que se construya a partir de dicho plano, por mucho que se parezcan entre sí, sería en POO un objeto. De hecho, en el plano de Elena pone que el `suelo` de la estancia puede ser de `parquet` o de `gres`. Así pues, habrá casas cuyo valor para el atributo `suelo` será `gres` y otros cuyo valor será `parquet`. Sin embargo, a partir del mismo plano (i.e. clase) se han creado todas las casas (i.e. objetos).

Finalmente:

Ejemplo 7 (relación entre clase y objeto) – La reunión familiar

Si estamos en una reunión familiar, cada persona de la familia es única y se puede interactuar con ella. Es decir, cada persona de esa reunión es un objeto. Así pues, `Marina`, su madre `Elena` y su padre `David` son, cada uno de ellos, objetos. Es más, si el bisabuelo y el abuelo paternos de `Marina` se llaman ambos `Manuel`, cada uno de ellos es un objeto diferente, puesto que si bien se llaman igual, no son la misma persona (es decir, el mismo objeto). El bisabuelo y el abuelo, así como el resto de integrantes de la familia, son elementos con los que se puede interactuar de manera individual. Eso sí, todos los miembros de la familia son del mismo tipo `Person` (que es la clase a la que pertenecen). En el caso de esta familia, hay dos objetos que comparten el valor `Manuel` para el atributo `nombre`.

Así pues, en este punto debe quedar muy claro que *tu* perro, *tu* televisor y *tu* bolígrafo son objetos diferentes a *mi* perro, *mi* televisor y *mi* bolígrafo, aunque sean de la misma raza, modelo y color, respectivamente. Es más, si tienes dos perros de la misma raza, por muy iguales que sean, cada uno de ellos es único y, por consiguiente, son dos objetos distintos.

Vídeo de interés

Para acabar de entender la relación entre los conceptos *clase* y *objeto*, te recomendamos que veas el vídeo «Clase, objeto, atributo y método» que encontrarás en el aula de la asignatura.

1.4. Abstracción

Como dijimos en el módulo anterior, para diseñar un programa basado en el paradigma orientado a objetos debemos seguir un enfoque *bottom-up*. Esto significa ir de los objetos a las clases, es decir, a partir de los objetos del problema, tendremos que obtener las clases de dichos objetos. Para ello, tendremos que seguir un proceso de abstracción. Es por eso que:

> Una *clase* es la abstracción o definición de un conjunto de objetos similares de la vida real.

Por **abstracción** entendemos:

> **Definición de abstracción**
>
> El proceso mental de identificar las entidades (es decir, objetos) así como sus estados (atributos) y comportamientos (métodos) que son relevantes para el problema que estamos tratando. Así pues, también consiste en ignorar todos aquellos aspectos que son irrelevantes en el contexto en el que estamos trabajando.

Por consiguiente:

> Una *clase* describe las características y comportamientos comunes de un conjunto de objetos similares en un contexto o problema determinado.

Para abstraer las clases a partir de objetos concretos, hay que **entender muy bien el problema que queremos resolver** y **quedarse solo con lo estrictamente necesario**. Así pues, el contexto lo es todo, el contexto determinará qué clases hay y qué atributos y métodos definimos para cada una de ellas.

Ejemplo 8 (abstracción) – Abriendo una cuenta de ahorros

David ha ido al banco para abrir una cuenta de ahorros. La persona que atiende a David necesita pedirle cierta información personal para poder gestionar la creación de la nueva cuenta. De la siguiente información que listamos, ¿qué crees que el banco necesita saber?:

- Nombre
- Alergias alimenticias
- Apellido
- Dirección postal

- DNI
- Grupo de música favorito
- Teléfono de contacto
- Número de hijos

Efectivamente, David como objeto tiene mucha información, pero ¿realmente necesita el banco todos estos datos? La respuesta es «no». Para abrir una cuenta, el banco no necesita conocer las alergias alimenticias de David (si las tiene), ni su grupo de música favorito ni el número de hijos.

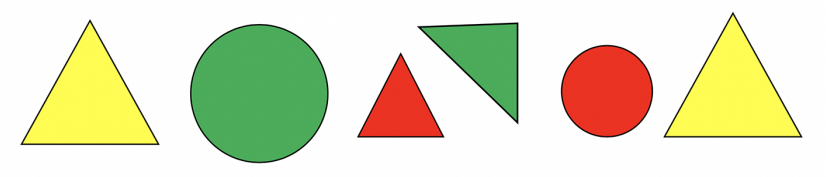
Ahora imagina que, en vez de abrir una cuenta de ahorros, David tiene que someterse a una operación quirúrgica. Entonces el hospital seguramente necesitará los mismos datos que en el caso del banco más alguno adicional como, por ejemplo, alergias alimenticias (ahora sí), grupo sanguíneo, etc.

En ambos contextos podríamos haber dicho que David era un objeto de la clase `Person`, pero los datos o atributos que necesitaríamos serían diferentes (y los métodos, también). Por lo tanto, en ambos contextos, la clase `Person`, aun llamándose igual, la hubiéramos definido de manera distinta, es decir, con distintos atributos y métodos.

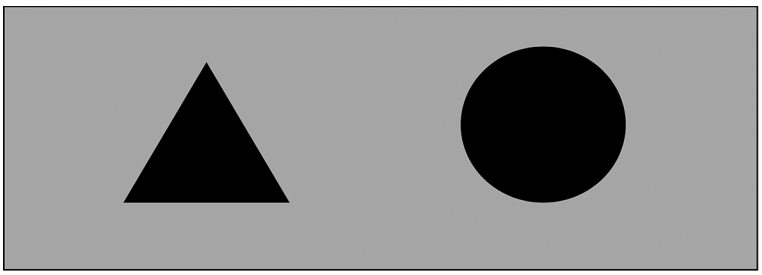
Veamos otro ejemplo a partir del juego de las formas al que jugaban Marina y Elena.

Ejemplo 9 (abstracción) – Encajando formas

A continuación Elena le pide a Marina que agrupe los siguientes objetos.



para introducirlos en la siguiente caja con dos orificios:



Después de un tiempo, la pequeña Marina los agrupa de la siguiente manera:

¿Qué criterio ha seguido? Ha usado la *forma* como criterio de agrupación. Mediante un proceso de abstracción, Marina se ha dado cuenta de que, por un lado, hay unos objetos que tienen tres lados y otros que no tienen ninguno. Se ha percatado de que esa característica permite resolver el problema al que se enfrenta mejor que, por ejemplo, el color de los objetos.

Ahora su mamá Elena señala a Marina los objetos de la izquierda y le pregunta:

–Marina, ¿qué son?

Marina se queda pensativa y contesta:

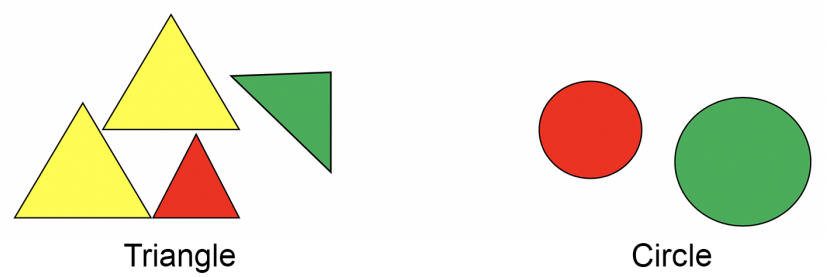
–¡Triángulos! –dice con lengua de trapo.

–¡Muy bien! –exclama Elena sorprendida. ¿Y estos otros? –pregunta Elena señalando al grupo de objetos de la derecha.

–Redondas –contesta Marina, tras dudar unos segundos.

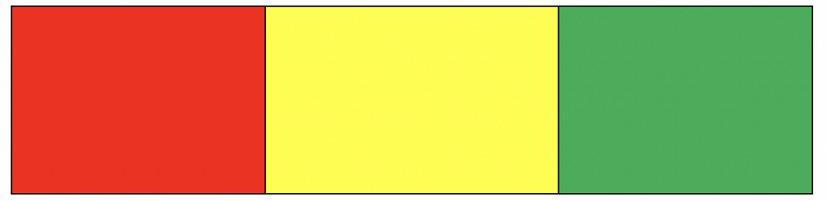
–¡Genial, Marina, son redondas, círculos! –dice Elena.

Marina ha dado un paso más en el proceso de abstracción y ha visto que diferentes objetos comparten un «algo» común. Este «algo» es una **clase**. Así, a los objetos de la izquierda les ha asignado la clase `Triangle` y a los de la derecha, la clase `Circle`.

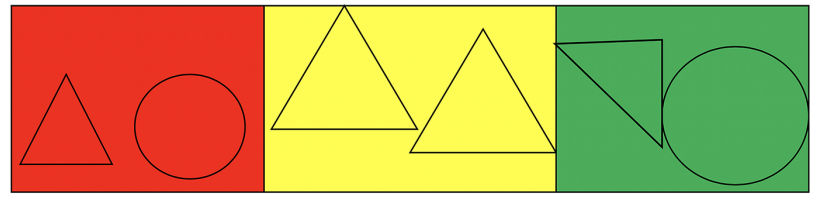


Por lo tanto, de la clase `Triangle` hay cuatro objetos y de la clase `Circle` hay dos. Ambas clases son diferentes. Como ya dijimos, los triángulos y los círculos pueden calcular su perímetro y área, pero los primeros son los únicos que pueden calcular su baricentro o que pueden ser rotados. Igualmente, hay atributos comunes como el color, pero también hay diferentes: longitud de los lados frente al radio.

Si el problema que se le hubiera presentado a Marina hubiera sido guardar los seis objetos en la siguiente caja:



entonces los hubiera agrupado de manera diferente, y las clases, junto con los atributos y métodos, seguramente hubieran sido otras (quizás `Red`, `Yellow` y `Green`).



Es muy importante darse cuenta de que los mismos objetos se pueden agrupar de manera diferente (es decir, abstraer en diferentes clases) dependiendo del problema que haya que resolver.

La **abstracción** es uno de los pilares en los que se fundamenta el paradigma de la programación orientada a objetos. La abstracción se realiza en la fase de diseño y consiste en centrarse solo en los datos y comportamientos de los objetos que son relevantes para solucionar el problema al que nos enfrentamos. Gracias a esto, simplificamos la complejidad del problema obteniendo una re-

presentación más simple y abstracta de la realidad y, a la vez, aumentamos la eficiencia. La abstracción se centra en qué hacen unos objetos, no en cómo lo hacen. Veamos un ejemplo:

Ejemplo 10 (abstracción) – Definiendo un coche en inglés

Como propósito de este año, David se ha apuntado a clases de inglés, su asignatura pendiente. En una de esas clases, la profesora le pregunta «¿qué es un coche?» (*what is a car?*). David balbucea y empieza a hablar con el reducido vocabulario que tiene: «Es un vehículo que se puede mover». De primeras, David describe el objeto en términos más abstractos diciendo que se mueve, pero sin decir cómo: usando neumáticos, volando, navegando, etc. Solo se mueve, no necesitamos más detalles para tener una primera idea del objeto. Después, David dice: «es terrestre y tiene cuatro ruedas» y «en general, tiene cinco asientos». Refina la abstracción del objeto *coche* para diferenciarlo de otros objetos como pueden ser una *moto* o un *camión*, pero sin entrar en demasiados detalles. Así pues, un coche es «un vehículo terrestre que tiene cuatro ruedas, por lo general, tiene cinco asientos y se puede mover». ¿Cómo son las ruedas y cómo están puestas?, ¿y los asientos?, ¿cómo se mueve?, ¿con gasolina o con electricidad?, ¿cómo pasa la fuerza del motor a las ruedas? No importa, eso es cómo lo hace, no qué hace.

Por último, veamos más ejemplos en los que abstraemos las características o atributos y acciones o métodos de diferentes entidades (u objetos) de la vida real. Nos daremos cuenta, una vez más, que cualquier entidad u objeto tiene un estado y un comportamiento:

- Los *perros* tienen un estado (atributos: nombre, raza, color, etc.) y un comportamiento (métodos: ladrar, mover la cola, enterrar huesos, etc.).

- Los *coches* también tienen un estado (velocidad actual, marcha actual, color, longitud, ancho, etc.) y un comportamiento (arrancar, subir marcha, bajar marcha, encender intermitente, etc.).

- De igual modo, los *televisores* tienen un estado (encendido o apagado, canal actual, volumen actual, etc.) y un comportamiento (encender, apagar, cambiar a un canal concreto, incrementar el número de canal, decrementar el número de canal, aumentar el volumen, disminuir el volumen, sintonizar, etc.).

- También las *facturas* tienen un estado (cobrada o no, importe total, paga y señal abonada, etc.) y un comportamiento (cambiar de no estar cobrada a estar cobrada y viceversa, modificar el valor del importe total, etc.).

- Incluso algo más abstracto o intangible como son los *contactos* del teléfono tienen un estado (nombre, apellido, teléfono, correo electrónico, etc.) y un comportamiento (introducir un atributo –es decir, nombre, apellido, teléfono, correo electrónico, etc.–, modificar el valor de un atributo y consultar el valor de un atributo).

- ¿Qué nos dices de los triángulos con los que jugaban Marina y Elena? Los *triángulos* tienen un estado (tres lados con una longitud cada uno –dependiendo de esto el triángulo será isósceles, escaleno, etc.–, un color, etc.) y

un comportamiento (está rotado, el cálculo de su área, cambiar el valor de la longitud de un lado, etc.).

Hasta ahora solo hemos abstraído, para cada conjunto de objetos similares (por ejemplo perros, coches, etc.), estados y comportamientos comunes (o dicho más simple, atributos y métodos). De momento no nos hemos preocupado de cómo vamos a implementar estos estados y comportamientos. El «cómo» lo veremos con la **encapsulación**.

Vídeo de interés

Para acabar de entender el concepto de *abstracción*, te recomendamos que veas el vídeo «Abstracción y paradigma bottom-up» que se encontrarás en el aula de la asignatura.

Finalmente, si nos fijamos en los ejemplos anteriores, nos daremos cuenta de que hay dos tipos de métodos:

1) Aquellos que hacen acciones que realiza la entidad real (por ejemplo, ladrar en el caso del perro, arrancar el motor por parte de un coche, la acción de encenderse de un televisor, etc.).

2) Aquellos que operan directamente sobre los atributos del objeto. Por un lado, están los métodos que modifican el valor de los atributos del objeto (por ejemplo, modificar el número de teléfono de un contacto o el canal actual de un televisor) y que, por consiguiente, cambian el estado del objeto. Por otro lado, están los métodos que consultan el valor de los atributos del objeto (por ejemplo, consultar el número de teléfono de un contacto o el canal actual de un televisor). Estos métodos de modificación y consulta son llamados, en el ámbito de la programación, ***setter*** y ***getter***, respectivamente.

2. Encapsulación: del diseño a la implementación

Si hemos dicho que la abstracción se centra en el diseño, la encapsulación se centra en la implementación.

Definición de encapsulación

Es el proceso de encerrar o empaquetar los atributos y métodos dentro de clases, que son entidades más grandes y más abstractas.

El principal objetivo de la encapsulación es hacer que un sistema complejo sea más sencillo de manejar por parte del usuario final. Gracias a la encapsulación se logra:

- Proteger los datos y el código ocultando los detalles internos de la implementación.
- Facilitar el uso de las clases y la reutilización de su código.
- Hacer transparente el mantenimiento de las clases (por ejemplo, realizar cambios internos) de cara al usuario final.

Antes de ver cómo logra los beneficios anteriores, vamos a aprender cómo se define una clase siguiendo el concepto de *encapsulación* y cómo se crean objetos a partir de la clase.

2.1. Miembros de una clase

Como hemos visto, un objeto está formado por atributos que establecen su estado y métodos que limitan su comportamiento. Hemos dicho que gracias a la encapsulación, los atributos y métodos de un conjunto de objetos similares los agrupamos en una **clase**. A su vez, al binomio formado por los **atributos** y los **métodos** se le denomina **miembros de una clase**. Así pues, tanto un atributo como un método son miembros de una clase.

2.1.1. Atributos

Los atributos, también llamados *campos* (*fields*), son como variables que codifican el estado de un objeto.

Clase sin atributos

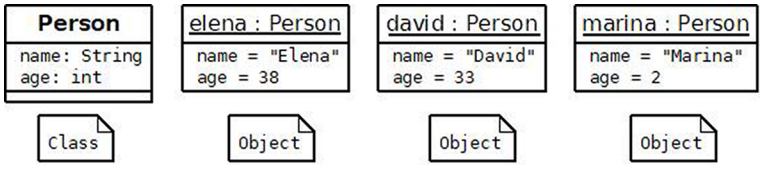
Una clase puede no tener atributos.

Si tenemos la clase `Person` con los atributos `name` y `age` –de tipo cadena de caracteres y entero, respectivamente–, cada objeto que se defina del tipo `Person` tendrá estos dos atributos.

Tipos de atributos

Un atributo puede ser de tipo básico o primitivo (`int`, `char`, etc.) o de un tipo de clase concreta, por ejemplo `Person`. El tipo de los atributos se define como cualquier otra variable.

El estado de cada objeto `Person` en un instante determinado dependerá de los valores que se les asigne a estos dos atributos, tal como se ve en la figura siguiente:



Aunque dos objetos compartan el mismo estado –es decir, mismos valores para todos sus atributos–, estos dos objetos son diferentes. Solo hay que pensar que puede haber dos personas llamadas `David` con `33` años de edad en el mundo y, obviamente, son personas (u objetos) diferentes.

2.1.2. Métodos

Los métodos implementan el comportamiento de un objeto o, dicho de otro modo, las funcionalidades que un objeto es capaz de realizar. Si hacemos una analogía con la programación estructurada, los métodos serían como las funciones (devuelvan algo o no). De ahí que un método, además de por el nombre, se caracterice por los parámetros que recibe y por el valor de retorno que devuelve. La descripción de estos elementos se conoce como **firma del método o signatura del método**. En pseudocódigo sería:

```
functionName(paramName1:type, …, paramNameN:type):returnType
```

En el caso de la clase `Person`, algunos métodos podrían ser:

```
talk(text:vector[30] of char):void
walk(speed:integer):void
```

El patrón que sigue la firma de los métodos depende de cada lenguaje de programación.

En este punto, cabe mencionar el concepto de **sobrecarga**.

La sobrecarga se produce cuando dos o más métodos tienen el mismo nombre, pero se diferencian en la lista de parámetros. Esta diferenciación debe darse en el número de parámetros, en el tipo de los parámetros o en el orden de los parámetros.

Un ejemplo de sobrecarga del método `talk` podría ser:

```
talk(text:vector[30] of char):void
talk(text:vector[30] of char, speed:integer):void
```

Sobrecarga en PHP

En PHP la sobrecarga de manera nativa no existe. Para «simular» una sobrecarga se debe usar la función mágica (*magic function*) llamada `__call()`.

El compilador decide qué método (cuál de los dos `talk`) invocar comparando los argumentos de la llamada con los parámetros de la firma.

Sin embargo, la siguiente sobrecarga daría error:

```
talk(text:vector[30] of char, speed:integer):void
talk(text:vector[30] of char, volume:integer):void
```

¿Por qué? Porque al llamar `talk("Elena", 50)`, el compilador no sabría a cuál de los dos métodos llamar: ¿el `50` es velocidad o volumen? ¿Tú lo sabes? El compilador tampoco.

Para solucionar este error, se podría cambiar el orden de los parámetros en la segunda firma, por ejemplo:

```
talk(volume:integer, text:vector[30] of char):void
```

2.2. Constructor y destructor

Las clases tienen dos tipos de métodos especiales llamados **constructor** y **destructor** que no se consideran miembros de una clase como tales. No se les considera miembros de una clase porque ni el constructor ni el destructor se heredan.

Ved también

El concepto de *herencia* se estudia en el módulo «Herencia (relaciones entre clases)» de esta asignatura.

La mayoría de los lenguajes de programación orientados a objetos implementan el método constructor, incluso algunos obligan a codificar explícitamente uno. No ocurre lo mismo con el destructor, cuya codificación se puede obviar en muchos lenguajes, por ejemplo, en Java.

2.2.1. Constructor

El constructor es aquel método especial que debemos llamar para crear un objeto. Con esta llamada, el objeto es situado en la memoria y se inicializan los atributos declarados en la clase. En la mayoría de lenguajes el constructor tiene las siguientes características:

1) Normalmente el nombre del constructor es el mismo que el de la clase.

Nombre del constructor diferente al de la clase

Desde la versión 5.3.3 de PHP, el constructor se define mediante un método llamado `__construct()`; en cambio, en Python, se debe usar el método especial `__init_()_`.

2) El constructor no tiene tipo de retorno, ni siquiera `void` (el tipo `void` quiere decir que no devuelve nada).

3) En la mayoría de lenguajes el constructor puede recibir argumentos con el fin de inicializar los atributos de la clase para el objeto que se está creando en ese momento.

4) En general suele ser público, pero algunos lenguajes permiten que sea protegido o privado. Veremos los conceptos *público*, *protegido* y *privado* en el apartado 2.5 de este módulo.

Atendiendo a las cuatro características anteriores, un ejemplo de constructor para la clase `Person` podría ser:

```
public Person(String nameParam){//Java
     name = nameParam;
     age = 33;
}
```

Hay lenguajes que permiten crear más de un constructor, por ejemplo C++, C# y Java, entre otros. En estos casos, al constructor sin parámetros se le suele llamar **constructor por defecto o predeterminado**, mientras que a aquellos que tienen parámetros se les llama **constructores parametrizados o con argumentos**. Como se puede apreciar, decir constructor «por defecto» y «parametrizado» es lo mismo que decir que se hace una sobrecarga del constructor. Debido a la sobrecarga, la única limitación cuando se quiere (y se puede) definir más de un constructor es que no pueden declararse varios constructores con el mismo número y el mismo tipo de parámetros (es decir, las mismas limitaciones que explicamos para la sobrecarga de métodos).

En los lenguajes en los que solo se puede codificar un constructor, por ejemplo, Python y PHP, a este se le llama simplemente constructor.

Hay que resaltar que en muchos lenguajes no es obligatorio que una clase tenga un constructor por defecto. Dependiendo del contexto, puede interesarnos que todos los constructores de una clase sean con argumentos.

Finalmente, decir que en muchos lenguajes no es obligatorio definir explícitamente un constructor para una clase. En estos lenguajes, si no se define ningún constructor para la clase, el propio compilador creará un constructor por defecto –es decir, sin argumentos– que no hará nada especial más allá de ubicar el objeto en memoria. Sin embargo, en el momento en que el programador implementa un constructor (sea por defecto o con argumentos), el compilador no añadirá automáticamente el constructor por defecto.

2.2.2. Destructor

El destructor es un método especial que es llamado durante la ejecución del programa cuando el objeto se está destruyendo, eliminando o liberando de la memoria. Podemos decir que es el último método al que se llama antes de que desaparezca el objeto de memoria. En muchos lenguajes este método se llama de manera automática cuando ocurren una serie de circunstancias. Asimismo hay que tener en cuenta que:

Llamada al destructor en C++

En C++, para eliminar un objeto, se debe utilizar el operador `delete` seguido del nombre del objeto, p.ej. `delete david`. El operador `delete` llamará al destructor de la clase `Person` a la que pertenece el objeto `david`.

1) No todos los lenguajes obligan a implementar un método destructor. En estos casos, si no se codifica uno explícitamente, el compilador crea uno por defecto.

2) Por norma general, una clase tiene solo un destructor.

3) En algunos lenguajes no tiene tipo de retorno, ni siquiera `void`. En otros, generalmente, tiene `void` como tipo de retorno.

4) No tiene parámetros.

5) En general suele ser público.

La manera en la que se declara el método destructor varía según el lenguaje. Por ejemplo, en C++ y C# el nombre del destructor es el mismo que el de la clase precedido por el símbolo `~`, por ejemplo `~Person()`. En otros lenguajes se usa un método especial que se comporta como un destructor, aunque no es un destructor propiamente dicho, por ejemplo en Java se utiliza el método especial `finalize()`.

Algunas tareas habituales que se suelen hacer dentro de un destructor y por las cuales tendremos que implementar explícitamente uno son:

- Cerrar ficheros, *sockets* o conexiones a bases de datos que el objeto ha abierto.
- Liberar recursos compartidos que el objeto tiene bloqueados (muy típico en programación distribuida y concurrente).
- Liberar memoria, por ejemplo, punteros que el objeto ha pedido.

2.3. Creando instancias de una clase

Como ya hemos comentado, los objetos son ejemplares de una clase y, por consiguiente, se crean a partir de la definición de su clase. A la hora de crear un objeto debemos seguir los siguientes pasos:

1) Declarar una variable para el objeto que queremos crear. Esta declaración, en muchos lenguajes, deberá tener un tipo (que será el nombre de una clase) y un identificador (es decir, un nombre). En otros lenguajes, no hace falta indicar el tipo (es decir, la clase).

```
ClassName instanceName; //Java, C#
instanceName; //Python, PHP (no se definen los tipos explícitamente)
```

Por ejemplo:

```
Person david; //Java, C#
david //Python
$david; //PHP, las variables van precedidas de $
```

2) Crear el objeto en memoria (también llamado **instanciar** el objeto). Para ello se llama a uno de los constructores que se haya definido dentro de la clase. Hay lenguajes que utilizan una palabra especial para llamar al constructor, por ejemplo, la palabra reservada `new`.

¿Instancia u objeto?

Debido a que a la acción de crear un objeto a partir de una clase se le llama *instanciar*, muchas veces a los objetos se les llama *instancia*. Por consiguiente, `david` es una «instancia» o un «objeto» de `Person`.

```
david = new Person("David"); //Java, C#
david = Person("David") //Python
$david = new Person("David"); //PHP
```

Los pasos 1 y 2 se pueden agrupar en un único paso (algunos lenguajes obligan a hacerlo así, por ejemplo C++):

```
Person david = new Person("David"); //Java, C#
Person david("David"); //C++
val david = new Person("David"); //Scala
david = Person("David") //Python
$david = new Person("David"); //PHP
```

Si nos quedáramos en el paso 1, es decir, solo en la declaración, y no hiciéramos la instanciación, entonces el objeto no sería construido en memoria y, por lo general, el compilador le asignaría el valor `null`.

Una vez se ha llamado al constructor, tenemos el objeto `david` de tipo `Person` creado en memoria con los atributos y métodos de la clase `Person` copiados. A partir de ese momento ya podemos acceder a sus atributos y métodos. A los

atributos y métodos de un objeto concreto, por ejemplo, `david`, se les llama **miembros de la instancia**. Esto es así porque ese objeto o instancia tiene su propia copia de los atributos y de los métodos, los cuales ocupan una zona de memoria diferente a los de otro objeto `Person`. Así, si instanciáramos un segundo objeto llamado `elena`, al modificar su atributo `name`, no estaríamos modificando el atributo `name` del objeto `david`.

2.4. Mensaje

Cuando los objetos quieren interactuar entre ellos utilizan **mensajes**. Un mensaje es la manera que existe de acceder a los atributos y métodos de un objeto (es decir, a los miembros de la instancia). La forma de un mensaje, en la mayoría de lenguajes, sigue la siguiente sintaxis:

```
instanceName.member
```

donde `member` puede ser un atributo o un método. Por ejemplo:

```
elena.name = "Elena";
david.talk("Hola!!");
elena.talk(david.name);
```

En la mayoría de los lenguajes para crear un mensaje (es decir, acceder a un atributo o método) se utiliza el operador punto (`.`). En otros lenguajes se utilizan otros símbolos. Por ejemplo, en PHP se utiliza `->`, es decir, `$david->talk()`.

2.5. Ocultación de información

Uno de los mecanismos íntimamente relacionados con la encapsulación es la **ocultación de información**. De hecho, muchas personas hacen de los términos *encapsulación* y *ocultación de información* sinónimos y, sin embargo, el primero incluye al segundo. O dicho de otro modo, la ocultación de información es una consecuencia directa de la encapsulación. Por *información* se entiende tanto los atributos como los métodos.

Terminología

En otras explicaciones verás el concepto de *ocultación* referenciado como «visibilidad de los miembros de una clase» o «restricción de acceso a atributos y métodos» o, especialmente en documentación centrada en la programación, «modificadores de acceso».

El objetivo que se persigue con la ocultación es:

Hacer que una clase se comporte como una caja negra que proporciona un conjunto de servicios al programador que la utiliza, ocultándole todos aquellos detalles que no interesan que sepa o tenga acceso. Es decir, que el programador usuario solo debe conocer y poder acceder directamente a los atributos y métodos estrictamente necesarios, nada más. De esta manera, se separa o diferencia lo que un objeto puede hacer de la implementación real de cómo el objeto lo hace.

Veamos dos ejemplos para entender el concepto de *ocultación*.

Ejemplo 11 (ocultación de información) – La pastilla

Cuando estamos enfermos y el médico nos receta una pastilla, las dos únicas cosas que nos interesan *a priori* como usuarios son:

1) qué cura esa pastilla,

2) cómo se administra (oral o rectal, frecuencia, cantidad, etc.).

Es decir, usamos la pastilla y ya está, no nos preocupa con qué ingredientes está elaborada. Sabemos que la pastilla azul es la de la tos y la roja la de la tensión; conocemos su aspecto (interfaz o exterior), su utilidad y su modo de administración, pero no sabemos nada acerca de su composición (es decir, de su interior).

Ejemplo 12 (ocultación de información) – Cápsulas de café

Lo mismo ocurre con las cápsulas de café que tan de moda se han puesto. Por un lado, sabemos que el color de la cápsula –que es la interfaz– nos indica el sabor e intensidad del café y, por otro, sabemos que la manera de utilizar la cápsula es poniéndola de una determinada manera dentro de la cafetera compatible con dichas cápsulas. En este punto, poco sabemos acerca de la composición exacta del café que hay en su interior, pero como usuarios tampoco nos interesa demasiado, sabemos que el café resultante está bueno y nos lo tomamos.

El hecho de encapsular la pastilla y la cápsula de café hace que su utilización por parte del usuario final sea sencilla. Veamos un ejemplo más extenso:

Ejemplo 13 (ocultación de información) – Coche

Si bien es cierto que cuando obtenemos el permiso de conducir de coche se supone que tenemos unas nociones de mecánica, la realidad es que la mayoría de conductores no tienen ni idea de cómo funciona su coche por dentro. Esto, sin embargo, no les limita a la hora de conducir su vehículo.

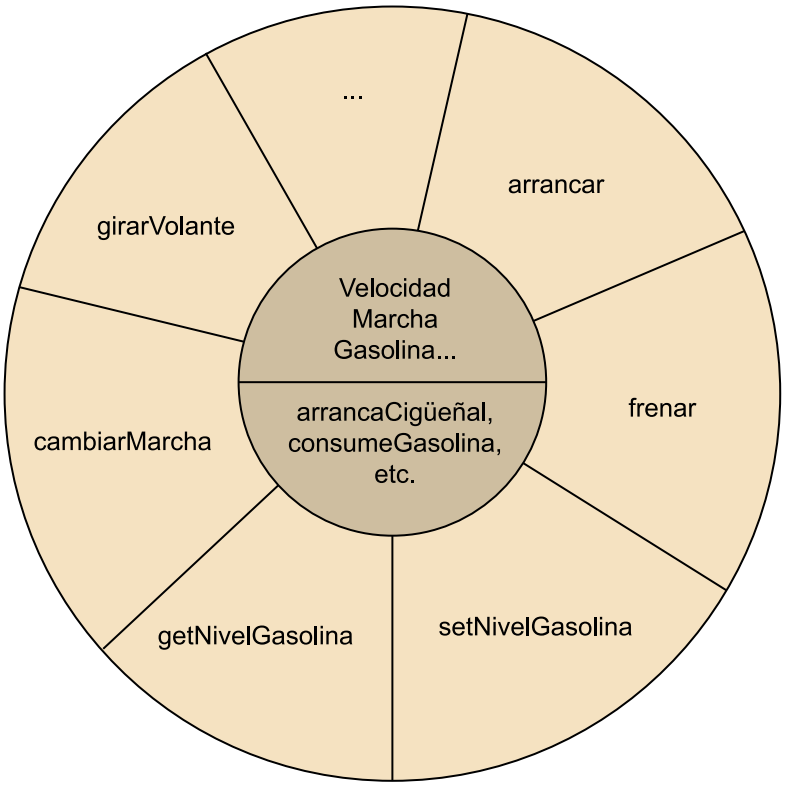
El conductor sabe cómo se utiliza el volante, cómo se cambia de marcha, para qué sirve cada pedal y cuándo debe usarlo, sabe dónde está la palanca que activa los intermitentes, las luces, etc. En definitiva, el conductor conoce la interfaz del vehículo, no su implementación; es decir, no sabe qué mecanismos se activan para que al dar al botón del parabrisas este se ponga en marcha. De igual modo, sabe que debe introducir la llave en el contacto y girarla para que el coche se encienda, pero desconoce qué acciones realiza la mecánica del coche para arrancar y ponerse en marcha.

Si viéramos el coche como una clase de la POO, podríamos decir que los métodos `arrancar()`, `frenar()`, `girarVolante(derechaOIzquierda:boolean, grados:float)`, etc. son la parte visible y, por consiguiente, conocida por el conductor (es decir, por el programador que usa la clase `Car`). Sin embargo, cuando el conductor llama al método `arrancar()`, por dentro, este método llama a muchos otros métodos ocultos que el conductor desconoce –por ejemplo, `arrancaCigüeñal`, `consumeGasolina`, etc.– y que solo conoce el mecánico (es decir, el programador que ha diseñado y codificado la clase `Car`).

Asimismo, la clase `Car` tiene atributos que como conductores podemos consultar y modificar mediante métodos *getter* y *setter*. Por ejemplo, el nivel de gasolina del depósito es un atributo oculto que consultamos mediante el método `getNivelGasolina()`, que

nos devuelve el valor en el panel de información del vehículo en forma de número o de aguja. También somos capaces de modificarlo cuando repostamos en una gasolinera.

De modo gráfico y resumido, un coche (clase `Car`) visto como una clase POO sería:



En la parte exterior estarían los métodos y atributos visibles, y en la interior los ocultos.

Hay diferentes niveles de ocultación (o «modificadores de acceso», *access modifiers* o *access level modifiers*). Podemos restringir el acceso a los atributos y métodos en diferente medida. En general, la mayoría de los lenguajes de programación orientados a objetos definen tres modificadores de acceso (fue el lenguaje C++ quien los introdujo) que establecen tres niveles de acceso u ocultación diferentes: público, privado y protegido. Antes de explicar en qué consisten estos tres modificadores de acceso, cabe enfatizar tres aspectos clave:

1) Cuando hablamos de visibilidad u ocultación de un atributo o método de una clase, la debemos entender en relación con el resto de las clases.

2) Una clase siempre puede acceder a todos los atributos y métodos definidos en ella.

3) La interacción entre dos clases se puede hacer, o bien mediante la interacción de dos objetos de esas dos clases (interactúan usando mensajes, como veremos más adelante), o bien mediante el mecanismo de herencia (lo veremos en el último módulo de estos materiales).

2.5.1. Público

Si un atributo o método es público en una clase `A`, este puede ser accedido desde fuera de la clase `A`. Dicho de otro modo, cualquier otra clase puede acceder al atributo o método usando en su código un objeto de tipo o clase `A`. Si queremos definir un elemento como público, en la mayoría de lenguajes usaremos la palabra reservada `public`.

```
class Person{
     public int age = 5;
     public int getAge(){
         return age;
     }
}
```

En el ejemplo anterior estamos definiendo un atributo llamado `age` que es de tipo `int` (entero), cuyo valor de inicialización es `5` y su nivel de acceso es público. Además, estamos implementando un método público llamado `getAge` que no recibe parámetros, pero sí devuelve un entero, concretamente, el valor del atributo `age`.

Ejemplo 14 (ocultación de información) – Public

```
class MainApp{
  public void main(){
   Person david = new Person("David");
   david.getAge(); //devuelve 5
   david.age = 36;
   david.getAge(); //devuelve 36
  }
}
```

En una clase que hace de programa principal llamada `MainApp` tenemos un método `main` que utiliza un objeto llamado `david` de la clase `Person`. Si imaginamos que hemos definido en la clase `Person` el atributo `age` y el método `getAge()` como públicos, entonces podemos acceder a ellos directamente desde cualquier sitio de nuestro código.

2.5.2. Privado

Cuando un atributo o método es privado en una clase, a este solo se puede acceder desde dentro de la propia clase que lo define (a la cual se le llama *clase contenedora*), es decir, que ninguna otra clase puede acceder a él directamente. Si queremos definir un elemento como privado, en la mayoría de lenguajes usaremos la palabra reservada `private`.

`Private` en C++

En C++ las clases declaradas como amigas (*friend classes*) también pueden acceder a los atributos o métodos declarados como privados en la clase de la que es amiga.

```
private int age = 5;
public int getAge(){
    return age;
}
```

Imagina que en la clase `Person` hemos hecho la implementación anterior, es decir, hemos definido solo el atributo `age` como privado y, mientras, hemos mantenido el método `getAge()` como público.

Ejemplo 15 (ocultación de información) – Private

```
class MainApp{
  public void main(){
   Person david = new Person("David");
   david.getAge(); //devuelve 5
   david.age = 36; //error, no se asigna
   david.getAge(); //si la ejecución
   //continuara, devolvería 5
   david.age; //error, no se puede acceder
  }
}
```

En este caso el método `getAge()` codificado en la clase `Person` puede seguir accediendo al atributo `age` porque dicho atributo, aunque privado, está definido en la misma clase que `getAge()`.

Sin embargo, desde otra clase, como `MainApp`, no podemos acceder directamente al atributo `age`. Si queremos saber su valor, deberemos usar el método `getAge()`, que es público.

`private` en Python

En Python realmente no existen los miembros «privados» y, por lo tanto, tampoco la palabra reservada `private`. No obstante, existe la convención de usar la técnica de *name mangling* (o *name decoration*), que consiste en preceder el nombre del atributo o método con doble *underscore* (_), por ejemplo `self.__name`. Esto hace que el acceso al atributo o método no sea tan directo y parezca privado.

Acceso a miembros `private` en clases anidadas

Los diferentes lenguajes de programación orientados a objetos definen diferentes comportamientos a la hora de poder acceder directamente o no a los miembros privados de una clase desde una clase anidada dentro esta.

2.5.3. Protegido

Cuando un atributo o método es protegido en una clase, a este solo se puede acceder desde dentro de la propia clase que lo define (clase contenedora), así como desde clases hijas o derivadas (o subclases) de la clase contenedora.

`protected` en Python

En Python no existe la palabra reservada `protected`. En su lugar se precede el atributo o método con un *underscore* (_), por ejemplo, `self._name`.

El concepto de *clase hija* o *derivada* (o *subclase*) se verá en el módulo «Herencia (relaciones entre clases)» cuando veamos el mecanismo de herencia.

2.5.4. ¿Existen más modificadores de acceso?

Sí, pero depende del lenguaje de programación. Por ejemplo:

- En C# existe el modificador de acceso `internal`, que limita el acceso al `assembly` (unidad física de composición, equivalente a un `.jar`, `.dll` o binario) actual. Además, permite combinar dos modificadores o niveles: `protected internal` y `private protected`.

- En Java existe el nivel de acceso `package-private`, que no tiene palabra clave (*keyword*), simplemente se asume al no escribir ningún nivel de acceso.

2.5.5. ¿Solo podemos asignar modificadores de acceso a los miembros de una clase?

La respuesta es «no». Hay diferentes elementos de un programa orientado a objetos, aparte de los atributos y métodos, a los que se les puede asignar un nivel de acceso. Esto depende del lenguaje. Normalmente estos elementos se dividen en dos dominios o niveles:

- **Nivel alto:** clases, interfaces, `namespace`, `enum`, etc.
- **Nivel de miembro:** atributos y métodos (incluyendo constructores y destructores).

Ved también

El concepto de *interfaz* se verá en el módulo «Herencia (relaciones entre clases)».

Asimismo, los modificadores de acceso que se pueden aplicar varían según el lenguaje y el dominio o nivel del elemento al que se lo vamos a aplicar. Normalmente los elementos de nivel alto (es decir, clases, interfaces, etc.) suelen tener menos opciones. Por ejemplo, en Java, una clase o interfaz cuando es un elemento de nivel alto no puede ser declarada `protected` ni `private`, solo puede ser `package-private` o `public`. Sin embargo, cuando la clase o interfaz está anidada dentro de una segunda clase, entonces podemos también usar `protected` o `private`, ya que la clase o interfaz ha dejado de ser un elemento de nivel alto y se convierte en un «miembro» más de otra clase, es decir, como si fuera un atributo o un método de esa otra clase.

Por su parte, en C#, los `namespaces` son implícitamente públicos y no se permite asignar ningún modificador de acceso, por lo que el nivel de acceso público no puede ser modificado. Asimismo, C# solo permite asignar los modificadores `internal` y `public` a los elementos de alto nivel si estos no están anidados, e `internal` es el acceso por defecto (i.e. si no se indica nada). En caso de estar anidados, entonces sí que permite asignarles otros modificadores de acceso, los cuales dependerán del tipo de elemento (clase, interfaz, `struct`, etc.) al que se le vaya a anidar.

2.5.6. ¿Y si no indicamos un modificador de acceso?

Si a un elemento no le asignamos un modificador de acceso, entonces el nivel que se aplica o asume depende del lenguaje de programación que estemos usando. Además depende de qué estamos delimitando: un atributo, un método, una clase, una interfaz, etc. Veamos algunos ejemplos:

- **Clases:** Java las considera `package-private`, mientras que C# las considera `internal`. C++, Python y TypeScript no contemplan otro nivel de acceso que no sea `public`. Para Scala la única manera de decir que algo es `public` es no indicando un modificador de acceso.

- **Atributos y métodos de una clase:** TypeScript, Python y Scala, por ejemplo, los consideran `public`, mientras que los lenguajes C++ y C# los considera `private`. Por su parte, Java los considera `package-private`.

2.5.7. ¿Qué modificador de acceso se asigna habitualmente a cada elemento?

A continuación indicamos cuáles son los niveles o modificadores de acceso más habituales para cada elemento, lo que no quiere decir que no se puedan aplicar otros dependiendo del problema que hay que resolver.

a) Clases e interfaces: por lo general se declaran como públicas, pero en algunos lenguajes también pueden ser privadas o protegidas (solo cuando están anidadas). En muchos lenguajes, el nivel de acceso `protected` para clases e interfaces no tiene sentido si la clase o interfaz no está anidada y suele no estar permitido. En cambio, en lenguajes como Scala, sí que puede haber clases `private` o `protected` a nivel alto (es decir, sin estar anidadas).

```
public class Person{}
```

b) Atributos: usualmente los programadores los declaran privados, aunque pueden ser públicos y protegidos. Hacerlos protegidos es interesante cuando existe herencia entre clases.

```
private int age;
```

c) Métodos: lo más habitual es declararlos, o bien públicos, o bien privados. Los declararemos públicos cuando queramos que se puedan usar desde fuera de la clase (por ejemplo, mediante un objeto de esa clase) y privados cuando sean de uso interno de la clase. No obstante, igual que los atributos, también pueden ser protegidos.

```
public int getAge(){}
private void doSomething(int param){}
```

d) Constructores: típicamente son declarados públicos. Sin embargo, también pueden ser privados y protegidos. Este último caso es interesante cuando la clase que tiene el constructor protegido es una superclase de otra clase.

```
public Person(){}
```

Ved también

El concepto de *superclase* se verá en el módulo «Herencia (relaciones entre clases)» cuando veamos el mecanismo de herencia.

2.6. Protección de los datos

Vamos a intentar ejemplificar la utilidad de restringir el acceso al código, concretamente a los atributos, y ver cómo de útil es usar los modificadores de acceso para lograr un comportamiento estable y deseado (seguro) de nuestros programas. También quedará de manifiesto la importancia de usar métodos *getter* y *setter* para acceder a los atributos.

Ejemplo 16 (protección de los datos) - ¿Por qué se comportan de manera diferente?

Imagina que a un amigo le han proporcionado ya hecha una clase `A` con un atributo público llamado `valueA` de tipo `int`. Además, esta clase tiene un constructor por defecto y dos métodos públicos, uno que es un *getter* `getValueA()` que devuelve el valor de `valueA` y un *setter,* llamado `setValueA(int val)`, que recibe un entero y se lo asigna al atributo `valueA`.

Él te dice que ha desarrollado una clase `B` que tiene dos objetos de tipo `A` y que en ella hace lo siguiente:

```
public class B{
     public A objectA1, objectA2;

     public B(){
          objectA1 = new A();
          objectA1.valueA = 100; //Asignamos directamente porque es público
          print(objectA1.getValueA()); //Imprime 100
          print(objectA1.valueA); //Imprime 100

          objectA2 = new A();
          objectA2.setValueA(100); //Asignamos a través del setter
          print(objectA2.getValueA()); //Imprime 300
          print(objectA2.valueA); //Imprime 300
     }
}
```

Te explica que no entiende por qué con el objeto `objectA1` el programa imprime `100` y con `objectA2` imprime `300`. ¿Sabrías decirle por qué puede ser?

La respuesta es que con `objectA1` está accediendo directamente al atributo y lo está modificando. Sin embargo, el método `setValueA(int val)` lo que hace es multiplicar por tres el valor que se le pasa como parámetro y luego asignarlo al atributo `valueA`. Evidentemente, cuando se hace la asignación directamente (como se hace con `objectA1`), esta multiplicación no se realiza.

Gracias al ejemplo anterior vemos que la encapsulación no solo oculta información o restringe el acceso a los elementos, sino que también asegura que los datos sean tratados correctamente forzando a que se usen los métodos que hemos programado. Con los datos/atributos encapsulados con un modificador de acceso `private`, cualquier cálculo o comprobación que queramos que se haga sobre los datos será realizado, ya que no se puede acceder a ellos directamente desde fuera y, obligatoriamente, hay que utilizar un método público proporcionado por la propia clase para modificarlos.

Sigamos con el ejemplo anterior para acabar de ver lo útil que es la encapsulación:

Ejemplo 17 (protección de los datos) – ¿Por qué de nuevo no se comporta igual?

Imagina que en vez de asignar 100, ahora asignamos -100.

```
objectA1.valueA = -100; //Se asigna sin problemas
objectA2.setValueA(-100); //Da un error
```

¿Por qué la segunda línea da un error? Pues porque seguramente dentro de `setValueA(int val)` se está comprobando que no se asigne un valor negativo, porque para los objetos de esta clase está prohibido. Imagina que `valueA` fuera `age`, es obvio que nadie tiene una edad negativa. Con la asignación directa no podríamos controlarlo mediante el código de la clase, sino que lo tendríamos que controlar, allá donde hagamos la asignación, de la siguiente manera (lo cual nos hace repetir trozos de código similares miles de veces en nuestro programa, y esto no es eficiente):

```
int val = -100;

if(val>=0){
      objectA1.valueA = val;
}else{
      print "ERROR";
}
```

2.7. Facilidad de uso y reutilización

Si partimos de la clase `A` del Ejemplo 16, veremos que podemos usar esta clase en cualquiera de nuestros programas. Así, si la clase `A` fuera en verdad la clase `Customer`, allá donde tuviéramos clientes, por ejemplo, un restaurante, hotel, tienda, etc., podríamos usar dicha clase para crear objetos de tipo `Customer`. Gracias a la encapsulación podemos ver una clase como un elemento autocontenido que es *plug and play*. Quizás para cada contexto haya que hacer ajustes (por ejemplo, añadir algún método o atributo), pero el grueso de lo que hace un cliente ya lo tendríamos implementado.

2.8. Transparencia a los cambios

Si en una clase hacemos pequeños cambios dentro de un método, estos cambios deberían ser transparentes para el usuario final (es decir, el programador que usa la clase). Así pues, el programador que usa la clase no debería modificar su código, ya que el método modificado seguirá funcionando y, por lo tanto, las llamadas a esta serán correctas.

Imagina que el método `setValueA(int val)` de la clase `A` de antes, además de comprobar que el valor pasado por parámetros es positivo y asignar al atributo `valueA` el triple del valor pasado, también incrementara un atributo privado que acabamos de crear para saber cuántas veces se ha llamado a `setValueA(int val)` para el objeto en cuestión. Este cambio es tan leve que no se deberá modificar el código del programa que utiliza la clase `A`.

Si hay cambios más grandes, entonces habrá que ver cómo afecta. Por ejemplo, eliminar un método es un cambio importante, por ello, lo que se suele hacer, antes de eliminar un método, es marcarlo como `deprecated` u `obsolete`, según sea el lenguaje, en la versión de la clase justo anterior a la versión en la que se elimina e indicar que se utilice el nuevo método que lo sustituye.

De esta manera, el programador sabe que de momento puede seguir usando el método, pero que en futuras versiones de la clase dicho método no existirá y su programa dejará de funcionar si lo utiliza.

Así pues, la encapsulación nos ayuda también a poder hacer cambios en las clases y que los cambios, si son leves, sean transparentes para los usuarios finales. Por lo tanto, podemos decir que otro objetivo de la encapsulación es:

Ocultar al mundo exterior el trabajo interno de los objetos para que pueda cambiarse más tarde sin afectar a los usuarios externos.

Vídeo de interés

Para entender mejor el concepto de encapsulación, así como los niveles básicos de ocultación (es decir, los modificadores de acceso), te recomendamos que veas el vídeo «Encapsulación» que encontrarás en el aula de la asignatura.

3. Elementos estáticos (*static*)

3.1. Atributos y métodos estáticos

Hemos visto que gracias a la abstracción y la encapsulación podemos agrupar en una clase los atributos y métodos que comparten diferentes objetos. Asimismo, hemos comentado que cuando instanciamos un objeto de una clase, ese objeto se ubica en memoria teniendo su propia copia de los atributos y métodos de la clase a la que pertenece. Es por eso que decimos que dichos atributos y métodos son miembros de la instancia. Cada nuevo objeto de la clase tendrá sus miembros de la instancia, independientes del resto de objetos de la misma clase.

La pregunta ahora es: ¿podemos tener un atributo o un método que sea compartido por todos los objetos de la clase? La respuesta es «sí». Muchos lenguajes de programación orientados a objetos –como C++, Java, C#, etc.– incluyen el modificador `static`, con el que podemos forzar que el atributo o método sea un **miembro de la clase** y no de la instancia. Esto implica que dicho atributo o método sea común a todos los objetos y, por lo tanto, ocupe una única zona de memoria y se evite que haya una copia de dicho atributo o método para cada objeto que se cree. El hecho de que un atributo o método sea estático sigue permitiendo que cada objeto de la clase pueda acceder al atributo o método, pero también nos permite acceder al atributo o método sin necesidad de crear un objeto de la clase.

Miembros de la clase en otros lenguajes

Hay lenguajes, como Python, que no tienen el modificador `static`, pero permiten declarar atributos o métodos como estáticos. En el caso de Python, por ejemplo, los atributos inicializados en su declaración dentro de la clase son considerados *static*, mientras que si son inicializados en un constructor o método, entonces son *miembros de la instancia*. En el caso de los métodos se debe utilizar el decorador `@staticmethod`.

Ejemplo 18 (*static*) – Atributo estático I

```
public class A{
     private int id;
     private static int counter  = 0;

     public A(){ //constructor
          id = counter;
          counter = counter + 1;
     }
}
```

El atributo `counter` es un miembro de la clase, no de la instancia. Así pues, es compartido por todos los objetos de la clase `A`. Si creamos `2` objetos de tipo `A`, el valor de `id` del primer objeto será `0` y el del segundo `1`; y, una vez creado el segundo objeto, el valor del atributo `counter` será `2` para los dos objetos, porque `counter` es de la clase.

Veamos otro ejemplo, pero ahora con dos clases:

Ejemplo 19 (*static*) – Atributo estático II

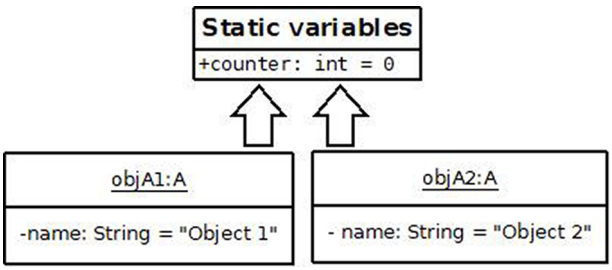
```
public class A{
     public static int counter  = 0;
     private String name;

     public A(String n){
          name = n;
     }
}

public class B{
     private A objA1;
     private A objA2;

     public B(){//constructor
          A.counter = A.counter+1;
          objA1 = new A("Object 1");
          print objA1.counter; //1
          print A.counter; //1
          objA1.counter = 5;
          print A.counter; //5
          objA2 = new A("Object 2");
          print objA2.counter; //5
     }
}
```

El atributo `counter` es un miembro de la clase `A`, no de la instancia. Así pues, es compartido por todo los objetos de la clase `A`.

En el constructor de la clase `B` accedemos al atributo `counter` de la clase `A` mediante el nombre de la clase `A` y mediante dos objetos de la clase `A`, `objA1` y `objA2`. Como el atributo `counter` solo existe una vez en memoria y es de la clase –lo que significa que es compartido por todos los objetos de la clase `A`–, da igual cómo accedamos, porque siempre estaremos modificando la misma «copia» del atributo `counter`.

En estos casos se recomienda acceder al atributo con el formato `className.staticField` para que quede claro que es un miembro de la clase y no de la instancia.

Visualmente sería así:



Los objetos `objA1` y `objA2` comparten la variable estática `counter`, pero ambos objetos tienen su atributo de instancia `name`, es decir, su propia copia del atributo `name`.

Ahora veamos un ejemplo de método estático. De igual modo que los atributos estáticos, los métodos estáticos también son compartidos por todos los objetos de la clase. Asimismo, se puede acceder mediante un objeto de la clase o mediante el nombre de la clase. Tal y como hemos dicho para los atributos estáticos, se recomienda acceder a los métodos estáticos con el nombre de la clase.

Ejemplo 20 (*static*) – Método estático

```
public class A{
     private static int counter  = 0;

     public static int getCounter(){
          return counter;
     }
     public static void setCounter(int c){
          counter = c;
     }
}

public class B{
     private A objA; //objeto de la instancia

     public B(){//constructor
          A.setCounter(5);
          objA = new A();
          print objA.getCounter(); //5
          print A.getCounter(); //5
          objA.counter = 5; //error
          print A.counter; //error
     }
}
```

En este ejemplo hemos declarado el atributo `counter` como `private`. Así pues, la única manera de acceder a él es mediante los métodos *getter* y *setter*.

En general, hay unas reglas que se suelen cumplir en los lenguajes de programación que permiten el uso del modificador `static` con atributos y métodos:

- Un método de la instancia (es decir, no estático) puede acceder directamente tanto a atributos y métodos estáticos como no estáticos.

- Un método de la clase (es decir, estático) no puede acceder directamente a un atributo o método de la instancia (es decir, no estático). Para poder acceder a estos necesita usar un objeto de la clase. Esto es obvio porque a un método estático se puede acceder sin necesidad de un objeto y, por lo tanto, en este caso no sabría, por ejemplo, qué valor tienen los atributos no estáticos que dependen de la instanciación de un objeto.

- Los métodos estáticos no pueden ser sobrescritos, pero pueden ser sobrecargados.

- Un método abstracto no puede ser declarado como estático.

Ved también

Los conceptos de *sobrescritura* y método *abstracto* se explicarán en el módulo «Herencia (relaciones entre clases)».

¿Cuándo es útil declarar un atributo o método como estático? En el caso de un atributo será útil declararlo estático cuando su finalidad sea la de contar el número de instancias u objetos que hemos declarado de una clase. También será útil cuando guarde un valor que es común a todos los objetos de la clase, por ejemplo, el valor del billete de los autobuses de una compañía (todos los

objetos `Bus` cobran el mismo precio por viaje). ¿Para qué guardar el mismo valor tantas veces como objetos haya? Esto sería una pérdida de memoria innecesaria.

Por otro lado, se debe tener cuidado con el uso de atributos `static`, puesto que podemos caer en el error de usarlos como variables globales –es decir, que estén disponibles en cualquier parte de nuestro programa–, con los problemas que ya sabemos que esto acarrea.

En el caso de los métodos, estos se declaran estáticos cuando la operación o acción que realiza es independiente de la creación de una instancia u objeto.

3.2. Constructor estático

En algunos lenguajes como C# también podemos hacer estático un constructor. En el caso de C#, si creamos un constructor estático, este se llamará automáticamente la primera vez que se crea un objeto de dicha clase. Este constructor no puede tener argumentos ni se le debe asignar un modificador de acceso. Un constructor estático suele usarse para inicializar cualquier atributo estático de la clase o realizar alguna acción que necesite realizarse solo una vez para los objetos de la clase. Además, el constructor estático no se hereda.

Ejemplo 21 – Constructor estático

```
public class A{
     static A(){ print "hola"; }
     public A(int a){ print a;}
     public void main(){
          //Imprimirá "hola" y luego 5;
          A obj1 = new A(5);
          //Imprimirá 6.
          A obj2 = new A(6);
     }
}
```

Dado que el primer objeto de la clase `A` que instanciamos es `obj1`, al crear `obj1` con el constructor no estático se llama primero al constructor estático y luego al constructor público. En cambio, para `obj2`, al tratarse ya del segundo objeto de la clase `A`, solo se llama al constructor no estático.

3.3. Clase estática

En algunos lenguajes –por ejemplo, Java, PHP, C++ y C#– es posible declarar una clase como estática usando el modificador `static`. El comportamiento de una clase estática depende del lenguaje de programación que estemos usando. Por ejemplo, veamos las diferencias entre Java y C#:

	Java	C#
¿Qué clases pueden ser estáticas?	Solo las clases que están dentro de otra (es decir, las clases anidadas, *nested classes*) pueden ser estáticas.	Cualquier clase puede ser declarada estática.
¿Puede ser instanciada?	Sí	No
Tipos de constructor	Solo constructor de instancia (i.e. el normal), ya que en Java no existe la posibilidad de declarar un constructor estático.	Solo constructor estático
¿Puede ser heredada?	Sí	No
¿Puede heredar de otra clase?	Sí	No, solo de `Object`.
Tipos de atributos o métodos que puede contener	Tanto estáticos como no estáticos	Solo estáticos

Así pues, vemos que en C# el modificador `static` aplicado a una clase fuerza a que todos sus miembros (i.e. atributos y métodos) sean estáticos, mientras que en Java significa que la clase (que es una *nested class*) es un miembro estático de la clase en la que está anidada como puede ser cualquier otro atributo o método de la clase contenedora. Veamos un ejemplo (el código no es Java, pero se parece):

Ejemplo 22 – Clase anidada estática (estilo Java)

```
public class A{
     private int a = 5;
     public static class B(){} //clase B anidada en la clase A
}

public class C{
     public C(){
         A.B objB = new A.B(); //podemos hacer esto.
     }
}
```

Otros lenguajes, como Scala y Kotlin, no tienen el modificador `static`, pero permiten crear un comportamiento similar mediante otras técnicas, por ejemplo, usando *companion objects*.

4. Representación de una clase y un objeto en UML

4.1. Clase

Para definir una clase (y en definitiva un programa) de manera formal y gráfica, se suele utilizar el lenguaje UML (*Unified Modeling Language*). Este lenguaje incluye muchos tipos de diagramas, por ejemplo: casos de uso, diagramas de secuencia, etc. Nosotros nos centraremos en el **diagrama de clases**.

Usar un diagrama basado en un lenguaje de modelización como UML permite leer y entender lo que representa una clase sin que sea necesario ver el código.

UML es un lenguaje conceptual, por lo que el **diagrama de clases** debería ser **lo más independiente posible al lenguaje de programación que se vaya a utilizar en la etapa de codificación**. De este modo, un mismo diagrama de clases sirve para cualquier lenguaje de programación.

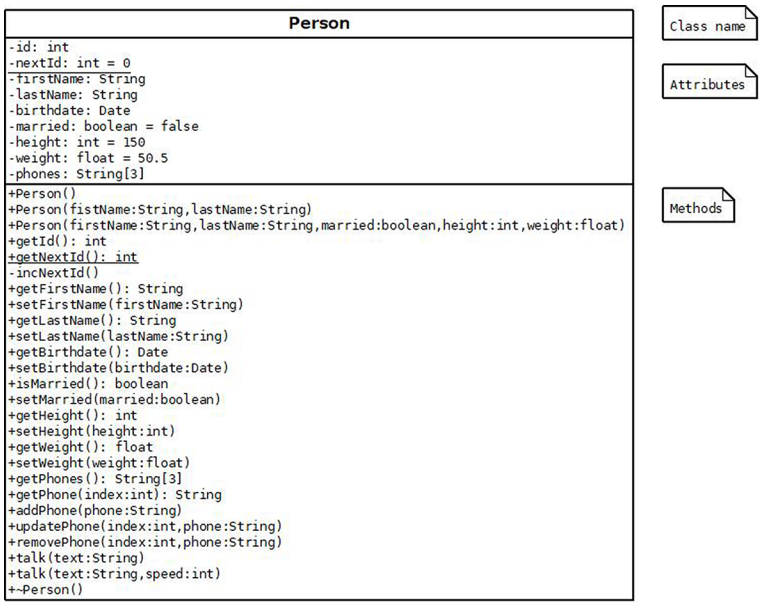
No obstante, es muy frecuente encontrar los diagramas de clases adaptados al lenguaje de programación que se va a utilizar en la etapa de codificación. Por ejemplo, si el programa lo vamos a desarrollar en Java, entonces a menudo no se escribe el destructor en el diagrama de clases, puesto que en este lenguaje no existe como tal. En Java, si se utiliza, aparecería `finalize()`.

También suele ocurrir que un diagrama de clases no pueda representarse exactamente en un lenguaje de programación. Por ejemplo, cuando veamos las clases asociativas, nos daremos cuenta de que la mayoría de lenguajes no nos permiten implementarlas. En tales casos, deberemos ser creativos e intentar traducir el diagrama de clases a código lo mejor que sepamos.

Ved también

El concepto de *clase asociativa* se estudia en el módulo «Asociaciones (relaciones entre objetos)» de esta asignatura.

Una vez introducido qué es UML, vamos a ver cómo se representa una clase en un diagrama de clases. En dicho tipo de diagramas, una clase se representa como una caja con tres compartimentos bien diferenciados:



1) Nombre de la clase: el compartimento superior contiene el nombre de la clase. Este se escribe en negrita y centrado. Por convenio, los nombres de las clases se escriben con la primera letra en mayúscula y en singular. Si el nombre de la clase está formada por varias palabras, cada una de las palabras debe empezar por mayúscula, por ejemplo, `BankAccount`, `PrintStream`, `VipMember`, `HttpProxyServer`, etc.

2) Atributos: el compartimento del medio contiene la definición de los atributos. Para cada atributo, como mínimo, se indica su visibilidad (o modificador de acceso), su nombre y el tipo al que pertenecen. Además es posible indicar su valor por defecto (de inicialización). En el diagrama anterior, el atributo `married` es de tipo `boolean` y su valor por defecto es `false`. Fíjate que los *arrays* se indican con el formato tipo `[longitud]`, por ejemplo, `phones: String[3]` para decir que es un *array* con tres `String` (cadenas de caracteres).

Por convenio, el nombre de los atributos es un sustantivo o sintagma nominal. Además, la nomenclatura habitual a la hora de escribir el nombre de los atributos es *lower camel case*, es decir, escribir en minúscula la primera palabra y, si el nombre del atributo está compuesto por dos o más palabras, entonces a partir de la segunda palabra la primera letra de cada una de ellas se escribe en mayúscula, por ejemplo, `lastName` (está compuesta de `last` + `name`), `height`, `resourceNumber`, `xMin`, `yTopLeft`, etc.

Como tipos de atributo suelen utilizarse los tipos primitivos habituales (es decir, `int`, `float`, `double`, `char`, etc.) así como `String` (es la manera usual de indicar una cadena de caracteres) o una clase proporcionada por el propio lenguaje de programación (no creada por nosotros), por ejemplo, `birthdate: Date` (este atributo es un objeto de la clase `Date`. Esta clase existe tanto en el estándard UML como en muchos lenguajes de programación, p.ej. Java).

3) Métodos: el último compartimento contiene los métodos de la clase, incluyendo los constructores y el destructor. En el caso del ejemplo, la clase `Person` tiene un constructor por defecto, dos constructores con argumentos y un destructor. Para poder diferenciar el constructor por defecto del destructor, a este último se le precede del símbolo `~`, por ejemplo, `~Person()`.

Dado que los métodos definen acciones, se ha establecido por convenio que el nombre de los métodos sea un verbo o sintagma verbal, a excepción de los constructores y el destructor, dado que se tienen que llamar exactamente igual que la clase. Igual que los atributos, el nombre de los métodos sigue un estilo *lower camel case*, es decir, la primera palabra del nombre se escribe en minúscula y las siguientes empiezan por mayúscula, por ejemplo, `getHeight()`, `getNextValue()`, etc.

Los parámetros de los métodos siguen la misma convención de nombre que los atributos y, además, se debe indicar su tipo. Asimismo, los métodos que devuelven algo deben indicar su tipo, por ejemplo, `getLastName():String`. En caso de no devolver nada, lo habitual es no escribir un tipo de retorno, por ejemplo `talk(text:String)`. No obstante, en algunos diagramas de clases se puede leer `void` como tipo de retorno, que quiere decir que el método no devuelve nada.

Nomenclatura para *getter* y *setter*

Por convenio, a los métodos de tipo *getter* se les suele llamar `get + nombre del atributo`, por ejemplo, `getHeight` para llamar al *getter* del atributo `height`. Excepcionalmente, los atributos de tipo `boolean` se llaman `is + nombre del atributo`, por ejemplo, `isMarried`. Por su parte, todos los *setter* se suelen llamar `set + nombre del atributo`, por ejemplo, `setHeight` y `setMarried`.

Por otro lado, nos falta explicar cómo se representan los diferentes niveles o modificadores de acceso o visibilidad en un diagrama de clases UML.

- El símbolo `+` indica que es público.
- El símbolo – indica que es privado.
- El símbolo `#` indica que es protegido.
- El símbolo `~` indica que su nivel de acceso es de tipo paquete (`package`). Su equivalente en Java sería el modificador `package-private`.

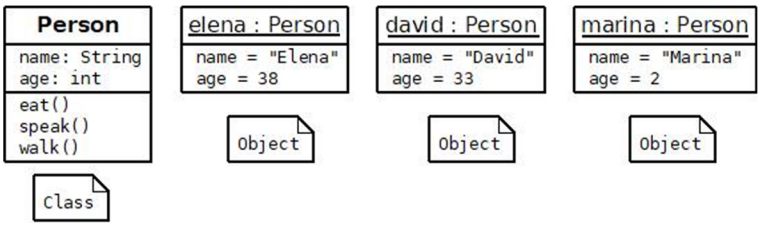
Finalmente, decir que para indicar que un atributo o un método es *static*, lo hacemos subrayando toda la definición del atributo o toda la firma del método. Mira el atributo `nextId` y el método `getNextId()` de la clase `Person` del diagrama de clases anterior.

En el caso de las clases estáticas, no hay un estándar, por lo que se da libertad al diseñador del diagrama de clases. Es frecuente indicar que una clase es estática anteponiendo el estereotipo `<<static>>` antes del nombre de la clase. Cuando la clase solo contiene atributos y métodos estáticos, entonces es habitual usar el estereotipo `<<utility>>`, en vez de `<<static>>`.

4.2. Objeto

Para mostrar un objeto en un diagrama de clases, este se representa como una caja con dos compartimentos. El primero de ellos está formado por un texto de una línea, todo él subrayado, que incluye el nombre de la instancia, a continuación dos puntos (:) y finalmente el nombre de la clase a la que pertenece.

El segundo compartimento es el estado del objeto, es decir, los atributos que son miembros de la instancia con sus correspondientes valores. Los métodos no se ponen porque ya se entiende que son los de la clase a la que pertenecen.



Resumen

Clase

- Para definir una clase (es decir, sus atributos y métodos) se debe hacer un ejercicio de abstracción siguiendo un proceso *bottom-up* (i.e. identificar primero los objetos y luego las clases).

- Una clase es un elemento *software* que representa a un conjunto de entidades del mundo real que son similares, p.ej., `Student`, `Ball`, etc.

- Por convenio, el nombre de una clase es un sustantivo o sintagma nominal en singular escrito en *upper camel case* (o *Pascal case*), p.ej., `Student`, `Ball`, `VipAuthor`.

- Una clase encapsula en un único elemento los datos (en forma de atributos) y comportamientos (en forma de métodos) comunes de las entidades que representa.

- Por convenio, el nombre de un atributo es un sustantivo o sintagma nominal, mientras que el nombre de un método es un verbo o sintagma verbal. Ambos casos se escriben en *lower camel case* (o *dromedary case*), por ejemplo, `numLifes`, `getParametersValues()`, etc.

- Gracias a la encapsulación las clases son elementos autocontenidos fáciles de reutilizar.

- La mayoría de lenguajes tienen, como mínimo, tres niveles de visibilidad (o modificadores de acceso): `public`, `protected` y `private`. Según el lenguaje utilizado, estos se pueden aplicar a diferentes elementos del programa: atributos, métodos, clases, etc.

- En un diagrama de clases UML, una clase se representa como una caja de tres compartimentos: nombre de la clase, atributos y métodos.

- Los modificadores de acceso en un diagrama de clases UML se representan con un `+ (public)`, `# (protected)`, `- (private)` y `~ (package)`.

- En general, cuando el modificador `static` se aplica a:
  - una clase, su comportamiento depende del lenguaje de programación;
  - un atributo o método, este se convierte en miembro de la clase (no de la instancia) y, por lo tanto, se puede acceder sin necesidad de

crear un objeto de la clase, solo con usar `className.fieldName` o `className.methodName()`.

Objeto

- Un objeto es un caso particular o concreto de una clase, por ejemplo, `pau`, `mateo` y `sofia` son entidades concretas de la clase `Student`. Así pues, `pau`, `mateo` y `sofia` son objetos de la clase `Student`.

- La acción de crear un objeto y alojarlo en la memoria se llama *instanciar*, de ahí que a los objetos a menudo se les llame también *instancias*.

- Para instanciar un objeto se debe llamar a uno de los constructores de la clase. La manera de hacer esta llamada varía según el lenguaje de programación utilizado.

- Una vez instanciado el objeto en la memoria, este tendrá sus propios atributos y métodos, a los que llamaremos miembros de la instancia.

- Los valores que les asignemos a los atributos de la instancia u objeto constituyen el estado del objeto en un momento determinado.

- Los métodos constituyen el comportamiento o acciones que puede tener o hacer un objeto. Estos utilizarán los valores que tengan los atributos de la instancia a la que pertenecen.

- Para llamar a un miembro de la instancia, debemos usar mensajes. En la mayoría de lenguajes, el patrón para crear un mensaje es: `instanceName.fieldName` o `instanceName.methodName()`.

- El destructor es un método especial que se llama justo antes de que el objeto quede eliminado de la memoria para siempre. Cómo se define este método, depende de cada lenguaje de programación. En la mayoría de lenguajes el destructor es llamado automáticamente cuando ocurren una serie de circunstancias.

Actividades

Ejercicio 1
Piensa en un cajero automático. Este deberá controlar el dinero del que dispone y cómo lo tiene distribuido. Por ejemplo, si tiene 100 €, un cliente no podrá sacar 200 € por mucho que los tenga en su cuenta corriente. Asimismo, los 100 € los puede tener en dos billetes de 50 €, en diez billetes de 10 € o mediante la combinación de un billete de 50 € y cinco billetes de 10 €. Toda esta gestión la haremos con la clase `Cash` (en español, 'Dispensador'). La clase `Cash` la definimos de la siguiente manera:

a) Tiene tres depósitos de billetes (*notes*), uno para los billetes de 10 €, otro para los de 20 € y otro para los de 50 €.

b) Tiene un método llamado `getBalance` que permite saber la cantidad de dinero en euros que tiene en cada momento el cajero.

c) Tiene un método público llamado `toString` que devuelve el siguiente texto (`String`):

```
Amount: A €
[Notes 10 €: N10, Notes 20 €: N20, Notes 50 €: N50]
```

Donde `A` es la cantidad de dinero que tiene el dispensador en aquel momento y `N10`, `N20` y `N50` son el número de billetes de 10, 20 y 50 €, respectivamente, que hay en aquel momento.

d) El constructor por defecto inicializa los atributos de la clase a cero (`=0`) si son numéricos, a `false` si son `boolean` y a `null` si son de otro tipo.

e) Tiene un constructor con argumentos que tiene tantos argumentos como atributos tiene la clase.

f) Incluye todos los métodos *getter* y *setter* de los atributos de la clase. Todos son públicos.

g) Tiene un método llamado `withdraw` que recibe por parámetro una cantidad sin decimales de dinero en euros, comprueba si el cajero tiene esa cantidad entre todos sus depósitos y devuelve un *array* de tres casillas (una por tipo de billete) con la cantidad de billetes de cada tipo que dispensa para la cantidad solicitada. Además, actualiza la cantidad de dinero que queda en cada dispensador después de realizar la operación. Por ejemplo, si se pidieran 60 € y el cajero tuviera cinco billetes de 10 €, cero de 20 € y uno de 50 €, entonces devolvería un *array* donde la casilla 0 (= 10 €) sería 1, la casilla 1 (= 20 €) sería 0 y la casilla 2 (= 50 €) sería 1. Después de la operación, quedarían 4, 0 y 0 billetes, respectivamente. En caso de no poder dar billetes, por la razón que sea, imprime el texto «`There are not enough notes`».

Dibuja el diagrama de clases UML de la clase `Cash` indicando el nivel de acceso o visibilidad de los atributos y métodos, así como los tipos de los atributos de la clase, de los parámetros de los métodos y de los valores de retorno de los métodos.

Ejercicio 2
Nos piden que creemos una clase para gestionar la información de las diferentes sucursales o locales que tiene una cadena de restaurantes llamada UocDonald's. A esta clase la llamaremos `Restaurant` y la definiremos de la siguiente manera:

a) Tiene un atributo que permite identificar numéricamente la sucursal.

b) Tiene otros siete atributos que son: nombre de la sucursal (`name`), una dirección, que será la calle más el número (`address`), el nombre de la ciudad en la que se encuentra (`city`), un código postal (`zip`), un teléfono (`phone`), un correo electrónico de contacto (`email`) y un *flag* que dice si la sucursal está abierta o no (`open`).

c) Tiene un método público llamado `toString` que devuelve el siguiente texto (`String`):

```
[ID: I, Address: A, Phone: P, Email: E, Open: O]
```

Donde `I` es el identificador de la sucursal, `A` es su dirección (i.e. calle y número) más la ciudad y el código postal, `P` es el número de teléfono, `E` es el correo electrónico y `O` es el valor que indica si la sucursal está abierta o no.

d) Tiene un constructor por defecto que inicializa los atributos de la clase a cero (`=0`) si son numéricos, a `false` si son de tipo `boolean` y a `null` si son de otro tipo.

e) Tiene un constructor con argumentos que tiene tantos argumentos como atributos tiene la clase.

f) Incluye todos los métodos *getter* y *setter* de los atributos de la clase. Todos son públicos.

Dibuja el diagrama de clases UML de la clase `Restaurant` indicando el nivel de acceso o visibilidad de los atributos y métodos, así como los tipos de los atributos de la clase y su valor de inicialización (si se indica en el enunciado), los tipos de los parámetros de los métodos y de los valores de retorno de los métodos.

Ejercicio 3
Nos piden que creemos la clase `Room`, la cual modelará la información y el comportamiento de las salas que forman parte del museo MuseUOC. La clase `Room` la definimos de la siguiente manera:

a) Tiene un atributo que permite identificar la sala. Este identificador está formado siempre por tres dígitos, donde el primero hace referencia a la planta donde se ubica la sala, por ejemplo, `0` para la planta baja, `1` para la primera planta, etc. Los otros dos dígitos hacen referencia al número de la sala en aquella planta. Así pues, una sala con identificador `102` significaría que es la sala número `2` de la primera planta. El museo no tiene plantas bajas (i.e. sótanos).

b) Tiene otros cuatro atributos que son: nombre de la sala (`name`), un *flag* que dice si la sala está abierta o no (`open`), los metros cuadrados (`m2`) y el aforo máximo (`capacity`).

c) Tiene un método público llamado `toString` que devuelve el siguiente texto (`String`):

```
[ID: I, Name: N, m2: M, Capacity: C, Open: O]
```

Donde `I` es el identificador de la sala, `N` es su nombre, `M` son los metros cuadrados, `C` es el aforo de la sala y `O` es el valor que indica si la sala está abierta o no.

d) Tiene un constructor por defecto que inicializa los atributos de la clase a cero (`=0`) si son numéricos, a `false` si son de tipo `boolean` y a `null` si son de otro tipo.

e) Tiene un constructor con argumentos que tiene tantos argumentos como atributos tiene la clase.

f) Incluye todos los métodos *getter* y *setter* de los atributos de la clase. Todos son públicos.

Dibuja el diagrama de clases UML de la clase `Room` indicando el nivel de acceso o visibilidad de los atributos y métodos, así como los tipos de los atributos de la clase y su valor de inicialización (si se indica en el enunciado), los tipos de los parámetros de los métodos y de los valores de retorno de los métodos.

Ejercicio 4
Para modelar la información y el comportamiento de un usuario de la biblioteca BibliUOC-teca, usaremos la clase `Member`, la cual definimos de la siguiente manera:

a) Tiene un atributo que permite identificar al usuario. Este identificador está formado siempre por diez dígitos. El primer usuario es el `0000000000`.

b) Tiene otros cuatro atributos que son: correo electrónico (`email`), dirección (`address`), un *flag* que dice si el usuario está bloqueado o no (`blocked`) y la fecha de nacimiento (`birthday`).

c) Tiene un método público llamado `toString` que devuelve el siguiente texto (`String`):

```
[ID: I, E-mail: E, Address, A: S, Birthday: D, Blocked: B]
```

Donde `I` es el identificador del miembro, `E` es su correo electrónico, `A` es su dirección, `D` es su fecha de nacimiento y `B` es el valor que indica si el usuario está bloqueado o no.

d) Tiene un constructor por defecto que inicializa los atributos de la clase a cero (`=0`) si son numéricos, a `false` si son de tipo `boolean` y a `null` si son de otro tipo (por ejemplo, `Date`, etc.).

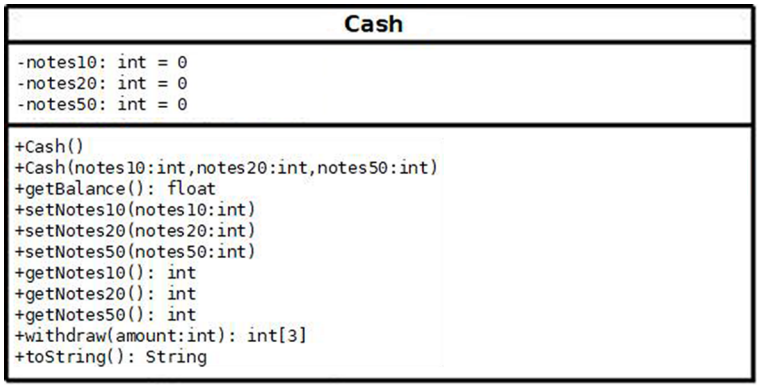
e) Tiene un constructor con argumentos que tiene tantos argumentos como atributos tiene la clase.

f) Incluye todos los métodos *getter* y *setter* de los atributos de la clase. Todos son públicos.

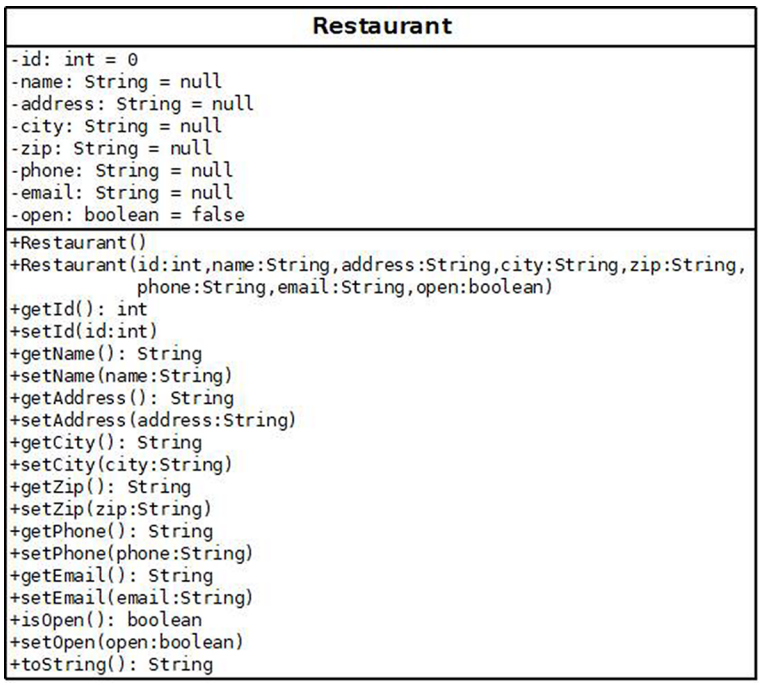
Dibuja el diagrama de clases UML de la clase `Member` indicando el nivel de acceso o visibilidad de los atributos y métodos, así como los tipos de los atributos de la clase y su valor de inicialización (si se indica en el enunciado), los tipos de los parámetros de los métodos y de los valores de retorno de los métodos.

Solucionario

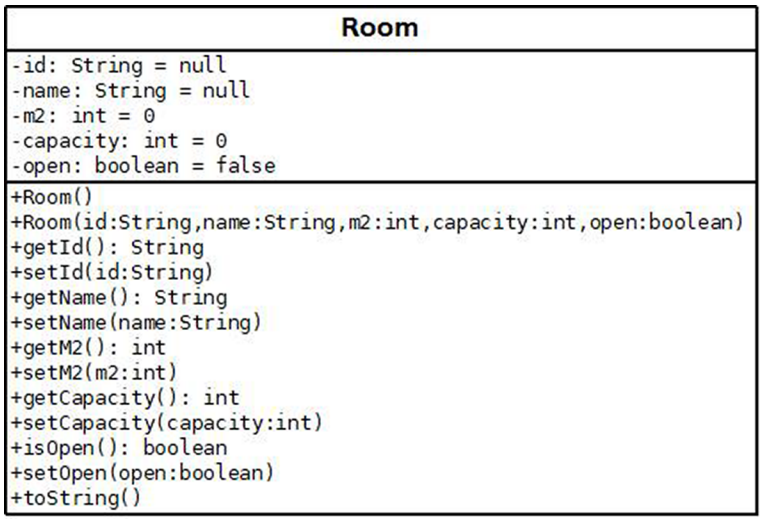
Solución ejercicio 1



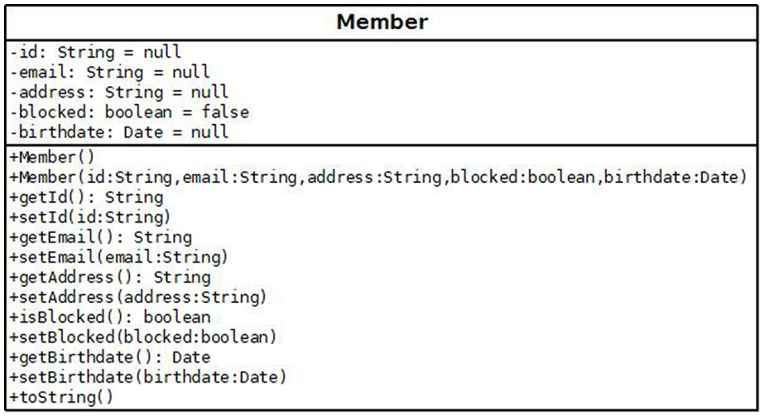
Solución ejercicio 2



Solución ejercicio 3



Solución ejercicio 4



Bibliografía

Booch, G.; Maksimchuk, R.; Engle, M.; Young, B. J.; Conallen, J.; Houston, K. (2007). *Object-Oriented Analysis and Design with Applications*. Addison-Wesley Professional. ISBN: 978-0201895513.

Griffiths, D.; Griffiths, D. (2019). *Head First Kotlin*. O'Reilly Media, Inc. ISBN: 978-1491996690.

Hunt, A.; Thomas, D. (2019). *The Pragmatic Programmer: your journey to mastery, 20th Anniversary Edition* (2.ª ed.). Addison-Wesley Professional. ISBN: 978-0135956977.

Microsoft (s. f.). «C# Guide» [en línea]. Microsoft. <https://docs.microsoft.com/en-us/dotnet/csharp/>

Phillips, D. (2018). *Python 3 Object-Oriented Programming* (3.ª ed.). Packt Publishing. ISBN: 978-1789615852.

Pollice, G.; West, D.; McLaughlin, B. (2006). *Head First Object-Oriented Analysis and Design*. O'Reilly Media, Inc. ISBN: 978-0596008673.

Robinson, S.; Nagel, C.; Watson, K.; Glynn, J.; Skinner, M.; Evjen, B. (2004). *Professional C#* (3.ª ed.). Wrox. ISBN: 978-0764557590.

Sharp, J. (2007). *Microsoft Visual C# 2008 Step by Step*. Microsoft Press. ISBN: 978-0735624306.

Weisfeld, M. (2019). *The Object-Oriented Thought Process* (5.ª ed.). Addison-Wesley Professional. ISBN: 978-0135182130.