

La Visibilidad

La visibilidad o accesibilidad determina el alcance que tiene un atributo u operación. El alcance (scope) de un miembro representa qué partes pueden acceder a dicho miembro. El alcance está determinado por los modificadores de visibilidad, presentados como "público" (su simbología es el signo +) y como "privado" (su simbología es el signo -)

Que un atributo u operación sea privado significa que aquel atributo u operación solo puede ser accedido desde dentro del objeto que lo contiene. En cambio si este atributo u operación fuera público podría ser accedido por cualquier objeto. En el siguiente ejemplo podemos ver los miembros públicos y privados de la clase "Auto" diferenciados por los signos "-" y "+":

Auto	
-	velocidad: int
+	acelerar(): void
+	frenar(): void

Esto mismo se puede aplicar con la clase "CuentaBancaria":

CuentaBancaria	
-	saldo: int
+	depositar(int): void
+	extraer(int): void

Una de las dudas que surgen a la hora de entender los modificadores de acceso es "¿por qué debería de tener un miembro privado?". Hacer que un atributo sea privado se utiliza para tratar con datos o valores los cuales no queremos que sean accedidos más que por el objeto contenedor. Las operaciones privadas cobran sentido como procesamientos internos del objeto que no queremos sean accedidas por otros objetos (tanto porque no tiene sentido que la accedan otros objetos como porque podría resultar peligroso para la integridad del objeto contenedor).

El porqué de utilizar miembros privados se aclarará a continuación junto con el tema "Encapsulamiento".

En resumen, la visibilidad determinan el alcance (scope) del atributo u operación en cuestión

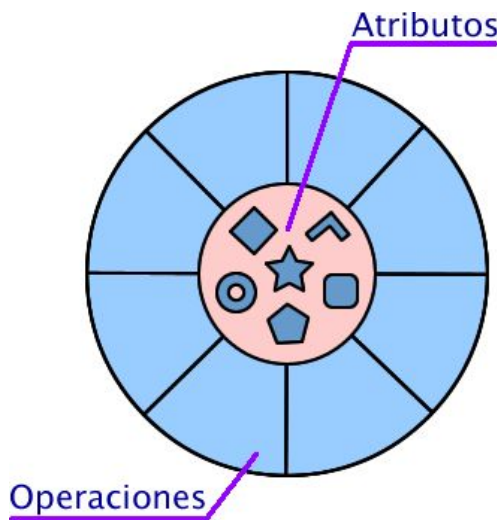
El Encapsulamiento

El encapsulamiento es una de las características representativas de la POO. Este concepto tiene que ver con la forma correcta de pensar en clases y objetos. Significa aislar o independizar las clases de nuestro programa de tal manera que su funcionamiento sea independiente del de las otras clases.

Automovil
- velocidad: int
+ asignarVelocidad(int): void

La clase "Automóvil" posee un atributo privado que representa la velocidad y posee una operación pública que le permite interactuar con otros objetos. Esta operación recibe como parámetro un valor que representa la velocidad que se le quiere asignar a dicho automovil. Si esta operación recibiera un valor negativo, podría pasarlo a un número positivo, mantener la velocidad anterior o hacer cualquier otra cosa evitando que aparezcan errores en el objeto por un dato inválido (velocidad negativa). Si en vez de utilizar un atributo privado y una operación pública, utilizáramos directamente un atributo público, este tipo de casos no podrían ser manejados y se atentaría contra la integridad del objeto.

La ocultación puede visualizarse mejor en el siguiente gráfico:



El gráfico representa que los atributos están encerrados por operaciones, es decir que los atributos son privados y las operaciones son públicas. Esto significa que para poder acceder a un atributo es necesario utilizar una operación, es decir que no hay posible acceso al atributo sino no se utiliza una operación.

A continuación se presenta el encapsulamiento con la clase "CuentaBancaria":

CuentaBancaria
- saldo: int
+ depositar(int): void
+ extraer(int): void

Para poder modificar el atributo saldo, es necesario hacerlo a través de las operaciones depositar o extraer.

En resumen, el encapsulamiento es la capacidad que tienen las clases de permitir que ciertos miembros sean accesibles y ciertos miembros no lo sean, y así lograr ocultar el estado interno de sus instancias.

Las principales ventajas que ello aporta son:

- Se facilita a los programadores que vayan a usar el tipo de dato el aprendizaje de cómo trabajar con él, pues se le pueden ocultar todos los detalles relativos a su implementación interna y sólo dejarle visibles aquellos que puedan usar con seguridad. Además, así se les evita que cometan errores por manipular inadecuadamente miembros que no deberían tocar.
- Se facilita al creador del tipo la posterior modificación del mismo, pues si los programadores no pueden acceder a los miembros no visibles, sus aplicaciones no se verán afectadas si éstos cambian o se eliminan. Gracias a esto es posible crear inicialmente tipos de datos con un diseño sencillo aunque poco eficiente, y si posteriormente es necesario modificarlos para aumentar su eficiencia, ello puede hacerse sin afectar al código escrito.
- La encapsulación se consigue añadiendo modificadores de acceso en las definiciones de miembros y tipos de datos.

Los Constructores

Toda clase tiene un constructor. Los constructores son operaciones que son invocadas al instanciar una clase. Los constructores se encargan de inicializar atributos y configurar todo lo que sea necesario antes de que comience a existir el objeto.

Al tratarse de operaciones, los constructores pueden contener cualquier comportamiento, aunque estas no tengan que ver exactamente con “construir” el objeto. Es posible tener más de un constructor en una misma clase; en este caso, cada constructor debe aceptar parámetros distintos para poder así distinguirlos.

Los constructores suelen llamarse con el mismo nombre de las clases, como se muestra a continuación:

Automovil	CuentaBancaria
- color: int - modelo: string - precio: int - usado: boolean	- saldo: int
+ acelerar(int, int): void + Automovil(): void + frenar(int, int): void	+ CuentaBancaria(): void + depositar(int): void + extraer(int): void

Por lo tanto la clase “Automóvil” tendrá un constructor llamado “Automovil()”, y la clase “CuentaBancaria” tendrá un constructor llamado “CuentaBancaria()”. En los ejemplos anteriores los constructores no tienen parámetros.

En resumen, un constructor es una operación que es invocada al instanciar una clase.

Los Destructores

Los destructores son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase. El destructor realizará procesos necesarios cuando un objeto termine su ámbito temporal, por ejemplo liberando la memoria

dinámica utilizada por dicho objeto o liberando recursos usados, como archivos, dispositivos, etc.

Los destructores tienen ciertas características:

- También tienen el mismo nombre que la clase a la que pertenecen, pero tienen el símbolo ~ delante.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- No tienen parámetros.
- No pueden ser heredados.
- Deben ser públicos, no tendría ningún sentido declarar un destructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.
- No pueden ser sobrecargados, lo cual es lógico, puesto que no tienen valor de retorno ni parámetros, no hay posibilidad de sobrecarga.
- Una clase no puede tener más de un destructor. Además, si omites el destructor, el compilador lo crea automáticamente.
- El sistema en tiempo de ejecución llama automáticamente a un destructor de la clase cuando la instancia de esa clase cae fuera de alcance o cuando la instancia se borra explícitamente.

La Sobrecarga de Operaciones

Así como es posible tener más de un constructor (que es una operación), es posible tener más de una operación con el mismo nombre dentro de la misma clase. Para poder hacer esto es necesario que se cumplan las mismas condiciones que para los constructores, es decir, que cada sobrecarga de operación tenga distintos tipos de parámetros.

¿Para qué podría servir tener varias operaciones con el mismo nombre pero con distintos parámetros? Por ejemplo, si quisiéramos crear una operación llamada "extraer()" que extraiga un determinado saldo de una cuenta, podríamos querer utilizarla sin determinar un tipo de moneda (utilizando la moneda del país como "por defecto"), o determinando un tipo de moneda para hacer la extracción de ese tipo de moneda (euro, dólar, etc).

La sobrecarga de operaciones nos permite utilizar el mismo nombre de operación para dos operaciones distintas, es decir, que aceptan distintos parámetros y tienen distinta funcionalidad. Sin la existencia de este mecanismo, se debería crear una operación distinta por cada "versión" de nuestra operación. Sin la sobrecarga de operaciones, las operaciones están limitadas a los parámetros de una sola definición de operación.

CuentaCorriente
- saldo: int
+ extraer(int) : int
+ extraer(int, string) : int

Si quisiéramos aplicar la sobrecarga de operaciones en una clase "Automóvil":

Automovil
- velocidad: int
+ acelerar(int) : void + acelerar(int, boolean) : void

La operación “acelerar()” tiene dos definiciones, en una acepta como argumento la cantidad de tiempo que debe acelerar, y en la otra la cantidad de tiempo que debe acelerar y si el auto utiliza turbo o no, representado por el segundo parámetro de tipo booleano

En resumen, la sobrecarga de operaciones es la capacidad de poder crear comportamientos distintos para la misma operación aceptando distintos tipos de parámetros.