LENGUAJES DE PROGRAMACIÓN

(Sesión 3)

2. PROGRAMACIÓN ORIENTADA A OBJETOS

2.1. Ocultación de información

2.2. Mensajes y métodos

Objetivo: Conocer aspectos de la programación orientada a objetos relativos al manejo de datos y entrega de resultados

La programación orientada a objetos o POO (OOP según sus siglas en inglés) es un paradigma de programación que usa objetos y sus interacciones para diseñar aplicaciones y programas de computadora. Está basado en varias técnicas, incluyendo herencia, modularidad, polimorfismo y encapsulamiento. Su uso se popularizó a principios de la década de 1990. Actualmente son muchos los lenguajes de programación que soportan la orientación a objetos.

Introducción

Los objetos son entidades que combinan *estado*, *comportamiento* e *identidad*:

- El *estado* está compuesto de datos, será uno o varios atributos a los que se habrán asignado unos valores concretos(datos).
- El *comportamiento* está definido por los procedimientos o Método (informática) con que puede operar dicho objeto, es decir, qué operaciones se pueden realizar con él.
- La *identidad* es una propiedad de un objeto que lo diferencia del resto, dicho con otras palabras, es su identificador (concepto análogo al de identificador de una variable o una constante).

La programación orientada a objetos expresa un programa como un conjunto de estos objetos, que colaboran entre ellos para realizar tareas. Esto permite hacer los programas y módulos más fáciles de escribir, mantener, reutilizar y volver a utilizar.

De aquella forma, un objeto contiene toda la información que permite definirlo e identificarlo frente a otros objetos pertenecientes a otras clases e incluso frente a objetos de una misma clase, al poder tener valores bien diferenciados en sus atributos. A su vez, los objetos disponen de mecanismos de interacción llamados métodos que favorecen la comunicación entre ellos. Esta comunicación favorece a su vez el cambio de estado en los propios objetos. Esta característica lleva a tratarlos como unidades indivisibles, en las que no se separan ni deben separarse el estado y el comportamiento.

Los métodos (comportamiento) y atributos (estado) están estrechamente relacionados por la propiedad de conjunto. Esta propiedad destaca que una clase requiere de métodos para poder tratar los atributos con los que cuenta. El programador debe pensar indistintamente en

ambos conceptos, sin separar ni darle mayor importancia a ninguno de ellos. Hacerlo podría producir el hábito erróneo de crear clases contenedoras de información por un lado y clases con métodos que manejen a las primeras por el otro. De esta manera se estaría realizando una programación estructurada camuflada en un lenguaje de programación orientado a objetos.

Esto difiere de la programación estructurada tradicional, en la que los datos y los procedimientos están separados y sin relación, ya que lo único que se busca es el procesamiento de unos datos de entrada para obtener otros de salida. La programación estructurada anima al programador a pensar sobre todo en términos de procedimientos o funciones, y en segundo lugar en las estructuras de datos que esos procedimientos manejan. En la programación estructurada sólo se escriben funciones que procesan datos.

Los programadores que emplean éste nuevo paradigma, en cambio, primero definen objetos para luego enviarles mensajes solicitándoles que realicen sus métodos por sí mismos.

Origen

Los conceptos de la programación orientada a objetos tienen origen en Simula 67, un lenguaje diseñado para hacer simulaciones, creado por Ole-Johan Dahl y Kristen Nygaard del Centro de Cómputo Noruego en Oslo. Al parecer, en este centro, trabajaban en simulaciones de naves, y fueron confundidos por la explosión combinatoria de cómo las diversas cualidades de diversas naves podían afectar unas a las otras. La idea ocurrió para agrupar los diversos tipos de naves en diversas clases de objetos, siendo responsable cada clase de objetos de definir sus *propios* datos y comportamiento. Fueron refinados más tarde en Smalltalk, que fue desarrollado en Simula en Xerox PARC (y cuya primera versión fue escrita sobre Basic) pero diseñado para ser un sistema completamente dinámico en el cual los objetos se podrían crear y modificar "en marcha" en lugar de tener un sistema basado en programas estáticos.

La programación orientada a objetos tomó posición como el estilo de programación dominante a mediados de los años ochenta, en gran parte debido a la influencia de C++, una extensión del lenguaje de programación C. Su dominación fue consolidada gracias al auge de las Interfaces gráficas de usuario, para las cuales la programación orientada a objetos está particularmente bien adaptada. En este caso, se habla también de programación dirigida por eventos.

Las características de orientación a objetos fueron agregadas a muchos lenguajes existentes durante ese tiempo, incluyendo Ada, BASIC, Lisp, Pascal, entre otros. La adición de estas características a los lenguajes que no fueron diseñados inicialmente para ellas condujo a menudo a problemas de compatibilidad y en la capacidad de mantenimiento del código. Los lenguajes orientados a objetos "puros", por otra parte, carecían de las características de las cuales muchos programadores habían venido a depender. Para saltar este obstáculo, se hicieron muchas tentativas para crear nuevos lenguajes basados en métodos orientados a objetos, pero permitiendo algunas características imperativas de maneras "seguras". El Eiffel de Bertrand

Meyer fue un temprano y moderadamente acertado lenguaje con esos objetivos pero ahora ha sido esencialmente reemplazado por Java, en gran parte debido a la aparición de Internet, y a la implementación de la máquina virtual de Java en la mayoría de navegadores. PHP en su versión 5 se ha ido modificando y soporta una orientación completa a objetos, cumpliendo todas las características propias de la orientación a objetos.

Conceptos fundamentales

La programación orientada a objetos es una forma de programar que trata de encontrar una solución a estos problemas. Introduce nuevos conceptos, que superan y amplían conceptos antiguos ya conocidos. Entre ellos destacan los siguientes:

- Clase: definiciones de las propiedades y comportamiento de un tipo de objeto concreto. La instanciación es la lectura de estas definiciones y la creación de un objeto a partir de ellas.
- Herencia: (por ejemplo, herencia de la clase D a la clase C) Es la facilidad mediante la cual la clase D hereda en ella cada uno de los atributos y operaciones de C, como si esos atributos y operaciones hubiesen sido definidos por la misma D. Por lo tanto, puede usar los mismos métodos y variables publicas declaradas en C. Los componentes registrados como "privados" (private) también se heredan, pero como no pertenecen a la clase, se mantienen escondidos al programador y sólo pueden ser accedidos a través de otros métodos públicos. Esto es así para mantener hegemónico el ideal de OOP.
- Objeto: entidad provista de un conjunto de propiedades o atributos (datos) y de comportamiento o funcionalidad (métodos) los mismos que consecuentemente reaccionan a eventos. Se corresponde con los objetos reales del mundo que nos rodea, o a objetos internos del sistema (del programa). Es una instancia a una clase.
- Método: Algoritmo asociado a un objeto (o a una clase de objetos), cuya ejecución se desencadena tras la recepción de un "mensaje". Desde el punto de vista del comportamiento, es lo que el objeto puede hacer. Un método puede producir un cambio en las propiedades del objeto, o la generación de un "evento" con un nuevo mensaje para otro objeto del sistema.
- Evento: Es un suceso en el sistema (tal como una interacción del usuario con la máquina, o un mensaje enviado por un objeto). El sistema maneja el evento enviando el mensaje adecuado al objeto pertinente. También se puede definir como evento, a la reacción que puede desencadenar un objeto, es decir la acción que genera.
- Mensaje: una comunicación dirigida a un objeto, que le ordena que ejecute uno de sus métodos con ciertos parámetros asociados al evento que lo generó.
- Propiedad o atributo: contenedor de un tipo de datos asociados a un objeto (o a una clase de objetos), que hace los datos visibles desde fuera del objeto y esto se define como sus características predeterminadas, y cuyo valor puede ser alterado por la ejecución de algún método.

- Estado interno: es una variable que se declara privada, que puede ser únicamente accedida y alterada por un método del objeto, y que se utiliza para indicar distintas situaciones posibles para el objeto (o clase de objetos). No es visible al programador que maneja una instancia de la clase.
- Componentes de un objeto: atributos, identidad, elaciones y métodos.
- Representación de un objeto: un objeto se representa por medio de una tabla o entidad que esté compuesta por sus atributos y funciones correspondientes.

En comparación con un lenguaje imperativo, una "variable", no es más que un contenedor interno del atributo del objeto o de un estado interno, así como la "función" es un procedimiento interno del método del objeto.

Características de la POO

Hay un cierto acuerdo sobre exactamente qué características de un método de programación o lenguaje le definen como "orientado a objetos", pero hay un consenso general en que las características siguientes son las más importantes:

•Abstracción: Denota las características esenciales de un objeto, donde se capturan sus comportamientos. Cada objeto en el sistema sirve como modelo de un "agente" abstracto que puede realizar trabajo, informar y cambiar su estado, y "comunicarse" con otros objetos en el sistema sin revelar *cómo* se implementan estas características. Los procesos, las funciones o los métodos pueden también ser abstraídos y cuando lo están, una variedad de técnicas son requeridas para ampliar una abstracción.

•Encapsulamiento: Significa reunir a todos los elementos que pueden considerarse pertenecientes a una misma entidad, al mismo nivel de abstracción. Esto permite aumentar la cohesión de los componentes del sistema. Algunos autores confunden este concepto con el principio de ocultación, principalmente porque se suelen emplear conjuntamente.

• Principio de ocultación: Cada objeto está aislado del exterior, es un módulo natural, y cada tipo de objeto expone una *interfaz* a otros objetos que especifica cómo pueden interactuar con los objetos de la clase. El aislamiento protege a las propiedades de un objeto contra su modificación por quien no tenga derecho a acceder a ellas, solamente los propios métodos internos del objeto pueden acceder a su estado. Esto asegura que otros objetos no pueden cambiar el estado interno de un objeto de maneras inesperadas, eliminando efectos secundarios e interacciones inesperadas. Algunos lenguajes relajan esto, permitiendo un acceso directo a los datos internos del objeto de una manera controlada y limitando el grado de abstracción. La aplicación entera se reduce a un agregado o rompecabezas de objetos.

•Polimorfismo: comportamientos diferentes, asociados a objetos distintos, pueden compartir el mismo nombre, al llamarlos por ese nombre se utilizará el comportamiento correspondiente al objeto que se esté usando. O dicho de otro modo, las referencias y las colecciones de objetos pueden contener objetos de diferentes tipos, y la invocación de un comportamiento en una referencia producirá el comportamiento correcto para el tipo real del objeto referenciado. Cuando esto ocurre en "tiempo de ejecución", esta última característica se llama *asignación tardía* o *asignación dinámica*. Algunos lenguajes proporcionan medios más estáticos (en "tiempo de compilación") de polimorfismo, tales como las plantillas y la sobrecarga de operadores de C++.

•Herencia: las clases no están aisladas, sino que se relacionan entre sí, formando una jerarquía de clasificación.

Los objetos heredan las propiedades y el comportamiento de todas las clases a las que pertenecen. La herencia organiza y facilita el polimorfismo y el encapsulamiento permitiendo a los objetos ser definidos y creados como tipos especializados de objetos preexistentes. Estos pueden compartir (y extender) su comportamiento sin tener que volver a implementarlo. Esto suele hacerse habitualmente agrupando los objetos en *clases* y estas en *árboles* o *enrejados* que reflejan un comportamiento común. Cuando un objeto hereda de más de una clase se dice que hay *herencia múltiple*.

•Recolección de basura: la Recolección de basura o Garbage Collection es la técnica por la cual el ambiente de Objetos se encarga de destruir automáticamente, y por tanto desasignar de la memoria, los Objetos que hayan quedado sin ninguna referencia a ellos.

Esto significa que el programador no debe preocuparse por la asignación o liberación de memoria, ya que el entorno la asignará al crear un nuevo Objeto y la liberará cuando nadie lo esté usando. En la mayoría de los lenguajes híbridos que se extendieron para soportar el Paradigma de Programación Orientada a Objetos como C++ u Object Pascal, esta característica no existe y la memoria debe desasignarse manualmente.

Es un paradigma que utiliza objetos como elementos fundamentales en la construcción de la solución. Surge en los años 70. Un objeto es una abstracción de algún hecho o cosa del mundo real que tiene atributos que representan sus características o propiedades y métodos que representan su comportamiento o acciones que realizan. Todas las propiedades y métodos comunes a los objetos se encapsulan o se agrupan en clases. ¨Una clase es una plantilla o un prototipo para crear objetos, por eso se dice que los objetos son instancias de clases. Lenguaje de programación: C++, Java, C#, VB.Net, etc

Un nuevo paso en la abstracción de paradigmas de programación es la Programación Orientada a Aspectos (POA). Aunque es todavía una metodología en estado de maduración, cada vez atrae a más investigadores e incluso proyectos comerciales en todo el mundo.

Abstracción

La abstracción consiste en aislar un elemento de su contexto o del resto de los elementos que lo acompañan. En programación, el término se refiere al énfasis en el "¿qué hace?" más que en el "¿cómo lo hace?" (característica de caja negra). El común denominador en la evolución de los lenguajes de programación, desde los clásicos o imperativos hasta los orientados a objetos, ha sido el nivel de abstracción del que cada uno de ellos hace uso.

Los lenguajes de programación son las herramientas mediante las cuales los diseñadores de lenguajes pueden implementar los modelos abstractos. La abstracción ofrecida por los lenguajes de programación se puede dividir en dos categorías: abstracción de datos (pertenecientes a los datos) y abstracción de control (perteneciente a las estructuras de control).

Los diferentes paradigmas de programación han aumentado su nivel de abstracción, comenzando desde los lenguajes de máquina, lo más próximo al ordenador y más lejano a la comprensión humana; pasando por los lenguajes de comandos, los imperativos, la orientación

a objetos (OO), la Programación Orientada a Aspectos (POA); u otros paradigmas como la programación declarativa, etc.

La abstracción encarada desde el punto de vista de la programación orientada a objetos expresa las características esenciales de un objeto, las cuales distinguen al objeto de los demás. Además de distinguir entre los objetos provee límites conceptuales.

Entonces se puede decir que la encapsulación separa las características esenciales de las no esenciales dentro de un objeto. Si un objeto tiene más características de las necesarias los mismos resultarán difíciles de usar, modificar, construir y comprender.

La misma genera una ilusión de simplicidad dado a que minimiza la cantidad de características que definen a un objeto.

Durante años, los programadores se han dedicado a construir aplicaciones muy parecidas que resolvían una y otra vez los mismos problemas. Para conseguir que los esfuerzos de los programadores puedan ser utilizados por otras personas se creó la POO. Que es una serie de normas de realizar las cosas de manera que otras personas puedan utilizarlas y adelantar su trabajo, de manera que consigamos que el código se pueda reutilizar.

La POO no es difícil, pero es una manera especial de pensar, a veces subjetiva de quien la programa, de manera que la forma de hacer las cosas puede ser diferente según el programador. Aunque podamos hacer los programas de formas distintas, no todas ellas son correctas, lo difícil no es programar orientado a objetos sino programar bien. Programar bien es importante porque así nos podemos aprovechar de todas las ventajas de la POO.

Ejemplo

Pensar en términos de objetos es muy parecido a cómo lo haríamos en la vida real. Por ejemplo vamos a pensar en un coche para tratar de modelizarlo en un esquema de POO. Diríamos que el coche es el elemento principal que tiene una serie de características, como podrían ser el color, el modelo o la marca. Además tiene una serie de funcionalidades asociadas, como pueden ser ponerse en marcha, parar o aparcar.

Pues en un esquema POO el coche sería el objeto, las propiedades serían las características como el color o el modelo y los métodos serían las funcionalidades asociadas como ponerse en marcha o parar.

Por poner otro ejemplo vamos a ver cómo modelizaríamos en un esquema POO una fracción, es decir, esa estructura matemática que tiene un numerador y un denominador que divide al numerador, por ejemplo 3/2. La fracción será el objeto y tendrá dos propiedades, el numerador

y el denominador. Luego podría tener varios métodos como simplificarse, sumarse con otra fracción o número, restarse con otra fracción, etc.

Estos objetos se podrán utilizar en los programas, por ejemplo en un programa de matemáticas harás uso de objetos fracción y en un programa que gestione un taller de coches utilizarás objetos coche.

Los programas orientados a objetos utilizan muchos objetos para realizar las acciones que se desean realizar y ellos mismos también son objetos.

Es decir, el taller de coches será un objeto que utilizará objetos coche, herramienta, mecánico, recambios, etc.

Encapsulamiento

En programación modular, y más específicamente en programación orientada a objetos, se denomina encapsulamiento al ocultamiento del estado, es decir, de los datos miembro, de un objeto de manera que sólo se puede cambiar mediante las operaciones definidas para ese objeto.

Cada objeto está aislado del exterior, es un módulo natural, y la aplicación entera se reduce a un agregado o rompecabezas de objetos. El aislamiento protege a los datos asociados a un objeto contra su modificación por quien no tenga derecho a acceder a ellos, eliminando efectos secundarios e interacciones.

De esta forma el usuario de la clase puede obviar la implementación de los métodos y propiedades para concentrarse sólo en cómo usarlos. Por otro lado se evita que el usuario pueda cambiar su estado de maneras imprevistas e incontroladas.

Encapsulamiento

Se dice que es el empaquetado de metodos y atributos dentro de un objeto, mediante una interfaz de mensajes. La clave está precisamente en el envoltorio del objeto.

Como se puede observar de los diagramas, las variables del objeto se localizan en el centro o núcleo del objeto. Los métodos rodean y esconden el núcleo del objeto de otros objetos en el programa. Al empaquetamiento de las variables de un objeto con la protección de sus métodos se le llama encapsulamiento. Típicamente, el encapsulamiento es utilizado para esconder detalles de la puesta en práctica no importantes de otros objetos. Entonces, los detalles de la puesta en práctica pueden cambiar en cualquier tiempo sin afectar otras partes del programa.

El encapsulamiento de variables y métodos en un componente de software ordenado es, todavía, una simple idea poderosa que provee dos principales beneficios a los desarrolladores de software: El encapsulamiento consiste en unir en la Clase las características y comportamientos, esto es, las variables y métodos. Es tener todo esto en una sola entidad. En los lenguajes estructurados esto era imposible. Es evidente que el encapsulamiento se logra gracias a la abstracción y el ocultamiento que veremos a continuación.

La utilidad del encapsulamiento va por la facilidad para manejar la complejidad, ya que tendremos a las Clases como cajas negras donde sólo se conoce el comportamiento pero no los detalles internos, y esto es conveniente porque lo que nos interesará será conocer qué hace la Clase pero no será necesario saber cómo lo hace.

La encapsulación da lugar a que las clases se dividan en dos partes:

1. Interface: captura la visión externa de una clase, abarcando la abstracción del comportamiento común a los ejemplos de esa clase.
2. Implementación: comprende la representación de la abstracción, así como los mecanismos que conducen al comportamiento deseado.

Formas de encapsular

1. Estándar (Predeterminado)
2. Abierto : Hace que el miembro de la clase pueda ser accedido desde el exterior de la Clase y cualquier parte del programa.
3. Protegido : Solo es accesible desde la Clase y las clases que heredan (a cualquier nivel).
4. Semi cerrado : Solo es accesible desde la clase heredada
5. Cerrado : Solo es accesible desde la Clase.

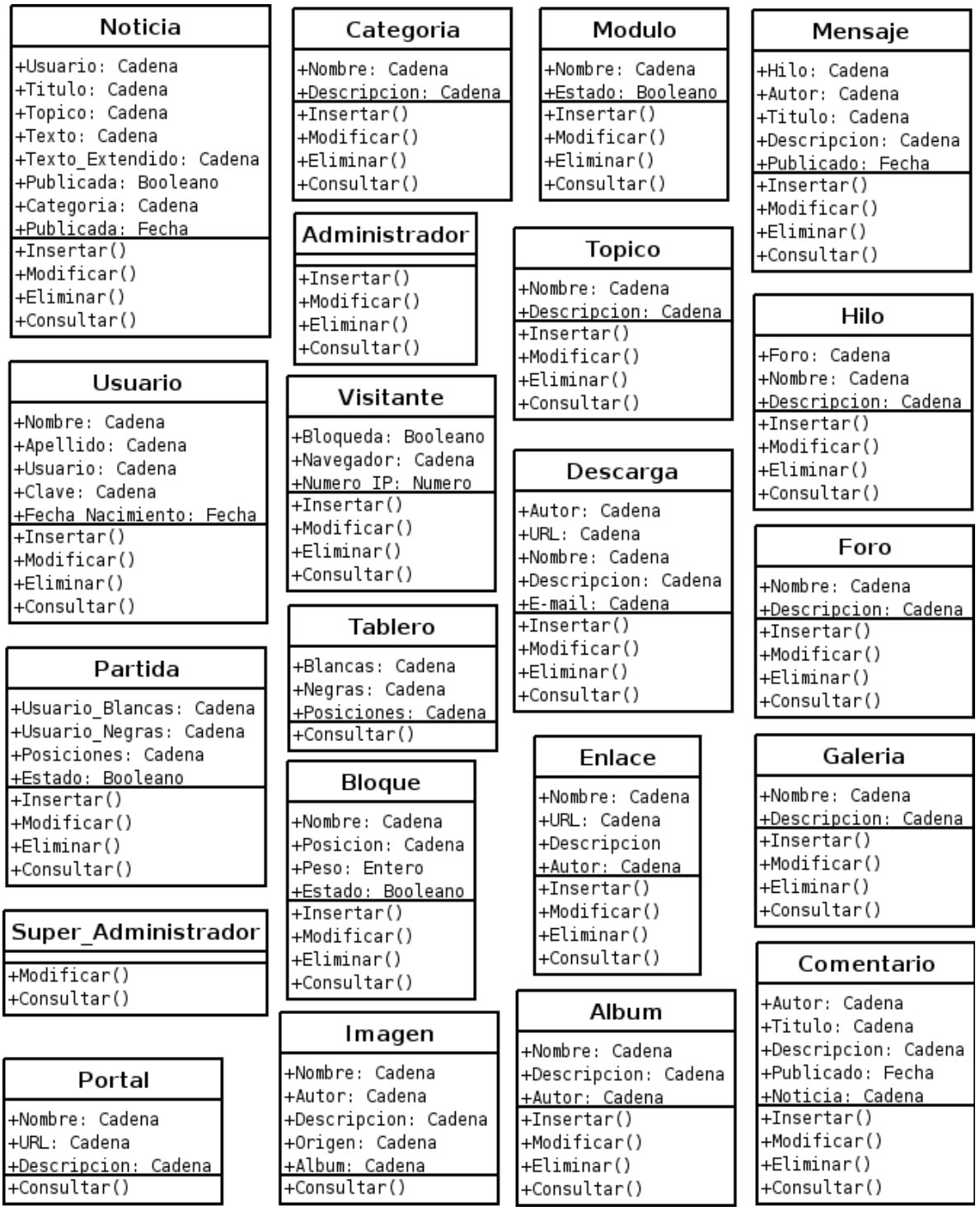
En el encapsulamiento hay analizadores que pueden ser semánticos y sintácticos.

Clase

Las clases son declaraciones o abstracciones de objetos, lo que significa, que una clase es la definición de un objeto. Cuando se programa un objeto y se definen sus características y funcionalidades, realmente se programa una clase.

Componentes

Una clase es un contenedor de uno o más datos (variables o propiedades miembro) junto a las operaciones de manipulación de dichos datos (funciones/métodos). Las clases pueden definirse como estructuras (struct), uniones (union) o clases (class) pudiendo existir diferencias entre cada una de las definiciones según el lenguaje. Además las clases son agrupaciones de objetos que describen su comportamiento


Noticia
+Usuario: Cadena
+Titulo: Cadena
+Topico: Cadena
+Texto: Cadena
+Texto_Extendido: Cadena
+Publicada: Booleano
+Categoria: Cadena
+Publicada: Fecha
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Categoria
+Nombre: Cadena
+Descripcion: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Modulo
+Nombre: Cadena
+Estado: Booleano
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Mensaje
+Hilo: Cadena
+Autor: Cadena
+Titulo: Cadena
+Descripcion: Cadena
+Publicado: Fecha
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Administrador
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Topico
+Nombre: Cadena
+Descripcion: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Hilo
+Foro: Cadena
+Nombre: Cadena
+Descripcion: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Usuario
+Nombre: Cadena
+Apellido: Cadena
+Usuario: Cadena
+Clave: Cadena
+Fecha_Nacimiento: Fecha
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Visitante
+Bloqueda: Booleano
+Navegador: Cadena
+Numero_IP: Numero
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Descarga
+Autor: Cadena
+URL: Cadena
+Nombre: Cadena
+Descripcion: Cadena
+E-mail: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Foro
+Nombre: Cadena
+Descripcion: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Tablero
+Blancas: Cadena
+Negras: Cadena
+Posiciones: Cadena
+Consultar()
Partida
+Usuario_Blancas: Cadena
+Usuario_Negras: Cadena
+Posiciones: Cadena
+Estado: Booleano
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Enlace
+Nombre: Cadena
+URL: Cadena
+Descripcion
+Autor: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Galeria
+Nombre: Cadena
+Descripcion: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Bloque
+Nombre: Cadena
+Posicion: Cadena
+Peso: Entero
+Estado: Booleano
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Super_Administrador
+Modificar()
+Consultar()
Imagen
+Nombre: Cadena
+Autor: Cadena
+Descripcion: Cadena
+Origen: Cadena
+Album: Cadena
+Consultar()
Album
+Nombre: Cadena
+Descripcion: Cadena
+Autor: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Comentario
+Autor: Cadena
+Titulo: Cadena
+Descripcion: Cadena
+Publicado: Fecha
+Noticia: Cadena
+Insertar()
+Modificar()
+Eliminar()
+Consultar()
Portal
+Nombre: Cadena
+URL: Cadena
+Descripcion: Cadena
+Consultar()

Ejemplos de clases

La sintaxis típica de una clase es:

```
class Nombre {
    // Variables miembro (habitualmente privadas)
    miembro_1;      //lista de
    miembros miembro_2;
    miembro_3;
```

```
    // Funciones o métodos (habitualmente
    públicas) funcion_miembro_1( ); // funciones
    miembro conocidas funcion_miembro_2 ( ); //
    funciones como métodos


    // Propiedades (habitualmente
    públicas)
    propiedad_
    1;
    propiedad_
    2;
    propiedad_
    3;
    propiedad_
    4;
}
```

Las clases habitualmente se denotan con nombres abstractos como Animal, Factura... aunque también pueden representar procesos o acciones como DarAlta

Objeto (programación)

En el paradigma de programación orientada a objetos (POO, o bien *OOP* en inglés), un objeto se define como la unidad que en tiempo de ejecución realiza las tareas de un programa. También a un nivel más básico se define como la instancia de una clase.

Estos objetos interactúan unos con otros, en contraposición a la visión tradicional en la cual un programa es una colección de subrutinas (funciones o procedimientos), o simplemente una lista de instrucciones para el computador. Cada objeto es capaz de recibir mensajes, procesar datos y enviar mensajes a otros objetos de manera similar a un servicio.

En el mundo de la programación orientada a objetos (POO), un objeto es el resultado de la instanciación de una clase. Una clase es el anteproyecto que ofrece la funcionalidad en ella

definida, pero ésta queda implementada sólo al *crear* una instancia de la clase, en la forma de un objeto.

Por ejemplo: dado un plano para construir sillas (una clase de nombre *clase_silla*), entonces una silla concreta, en la que podemos sentarnos, construida a partir de este plano, sería un objeto de *clase_silla*. Es posible crear (construir) múltiples objetos (sillas) utilizando la definición de la clase (plano) anterior.

Los conceptos de clase y objetos son análogos a los de tipo de datos y variable, es decir, definida una clase podemos crear objetos de esa clase, igual que disponiendo de un determinado tipo de dato (por ejemplo el tipo entero), podemos definir variables de dicho tipo:

```
int
a,b;
```

(*'int' es un tipo de dato y 'a' y 'b' son variables de tipo entero con las que podemos operar*)

Para utilizar la funcionalidad definida en una clase en particular (salvo en las clases abstractas), primeramente es necesario crear un objeto de esa clase. De la misma manera para una persona que desea sentarse, las especificaciones para construir una silla serán de poca utilidad; lo que se necesita es una silla real construida a partir de esas especificaciones. Siguiendo con la analogía anterior, también se puede decir que para hacer operaciones aritméticas, de nada sirve por sí solo el tipo entero (int); para ello necesitamos variables (o constantes) con las que operar.

Definición de Objeto

En filosofía un objeto es aquello que puede ser observado, estudiado y aprendido, en contraposición a la representación abstracta de ese objeto que se crea en la mente a través del proceso de generalización. Un objeto en POO representa alguna entidad de la vida real, es decir, alguno de los objetos que pertenecen al negocio con que estamos trabajando o al problema con el que nos estamos enfrentando, y con los que podemos interactuar.

A través del estudio de ellos se adquiere el conocimiento necesario para, mediante la abstracción y la generalización, agruparlos según sus características en conjuntos, estos conjuntos determinan las clases de objetos con las que estamos trabajando. Primero existen los objetos, luego aparecen las clases en función de la solución que estemos buscando. Ésta es la forma más común de adquirir conocimiento aunque no es la única.

En ocasiones cuando el observador es un experto del negocio (o del problema), el proceso puede ser a la inversa y comenzar el análisis en una base teórica abstracta, sustentada por el conocimiento previo que da lugar primeramente a clases de objetos que satisfagan las necesidades de la solución.

Estos conceptos son parte de la base teórica de la idea de objeto y clase utilizados en la POO. Los objetos tienen características fundamentales que nos permiten conocerlos mediante la observación, identificación y el estudio posterior de su comportamiento; estas características son:

- Identidad
- Comportamiento
- Estado

En las ramas de las ciencias de la computación más estrictamente matemáticas, el término *objeto* es usado en sentido puramente matemático para referirse a cualquier "cosa". Esta interpretación resulta útil para discutir sobre teorías abstractas, pero no es suficientemente concreta para servir como definición de un tipo primitivo en discusiones de ramas más específicas como en la programación, que está más cerca de cálculos reales y el procesamiento de información.

Identidad

La identidad es la propiedad que permite a un objeto diferenciarse de otros. Generalmente esta propiedad es tal, que da nombre al objeto. Tomemos por ejemplo el "verde" como un *objeto* concreto de una clase *color*; la propiedad que da identidad única a este objeto es precisamente su "color" verde. Tanto es así que para nosotros no tiene sentido usar otro nombre para el objeto que no sea el valor de la propiedad que lo identifica.

En programación la identidad de los objetos sirve para comparar si dos objetos son iguales o no. No es raro encontrar que en muchos lenguajes de programación la identidad de un objeto esté determinada por la dirección de memoria de la computadora en la que se encuentra el objeto, pero este comportamiento puede ser variado redefiniendo la identidad del objeto a otra propiedad.

Comportamiento

El comportamiento de un objeto está directamente relacionado con su funcionalidad y determina las operaciones que este puede realizar o a las que puede responder ante mensajes enviados por otros objetos. La funcionalidad de un objeto está determinada, primariamente, por su responsabilidad. Una de las ventajas fundamentales de la POO es la reusabilidad del código; un objeto es más fácil de reutilizarse en tanto su responsabilidad sea mejor definida y más concreta.

Una tarea fundamental a la hora de diseñar una aplicación informática es definir el comportamiento que tendrán los objetos de las clases involucradas en la aplicación, asociando la funcionalidad requerida por la aplicación a las clases adecuadas.

Estado

El estado de un objeto se refiere al conjunto de los valores de sus atributos en un instante de tiempo dado. El comportamiento de un objeto puede modificar el estado de este. Cuando una operación de un objeto modifica su estado se dice que esta tiene "efecto colateral".

Esto tiene especial importancia en aplicaciones que crean varios hilos de ejecución. Si un objeto es compartido por varios hilos y en el transcurso de sus operaciones estas modifican el estado del objeto, es posible que se deriven errores del hecho de que alguno de los hilos asuma que el estado del objeto no cambiará (Véase Condición de carrera).

Representación en las computadoras

Los objetos aunque son *entidades conceptuales*, dado el diseño de las computadoras, se corresponde directamente con bloques de memoria de tamaño y localización específicos. Esto ocurre porque los cálculos y el procesamiento de la información en última instancia requieren de una representación en la memoria de la computadora.

En este sentido, los objetos son primitivas fundamentales necesarias para definir de forma precisa conceptos como referencias, variables y vinculación de nombres. En ciencias de la computación se utiliza cotidianamente la interpretación más concreta de *objeto* en lugar de las más abstractas sin que esto sea considerado un error.

Es preciso hacer notar que aunque un bloque de memoria puede aparecer contiguo en un nivel de abstracción y no contiguo en otro, lo importante es que este aparece contiguo para el programa, quien lo trata como un objeto.

Por este motivo, los detalles de implementación privados de un modelo de objetos, no deben ser expuestos al cliente del objeto, y estos pueden ser cambiados sin que se requieran cambios al código cliente.

Los objetos en la computadora existen entonces, sólo dentro de contextos capaces de reconocerlos; un espacio de memoria sólo contiene un objeto si un programa lo trata como tal (por ejemplo, reservándolo para uso exclusivo de un procedimiento específico y/o asociándole un tipo de dato). Así, el tiempo de vida de un objeto es el tiempo durante el cual este es tratado como un objeto. Es por esto que los objetos son entidades conceptuales, a pesar de su presencia física en la memoria de la computadora.

En otras palabras, los conceptos abstractos que no ocupen espacio de memoria en tiempo de ejecución, no son, de acuerdo con esta definición, objetos. Ejemplos de estos conceptos son: patrones de diseño exhibidos por un conjunto de clases y tipos de datos en lenguajes de programación que utilizan tipos estáticos.

Se llama *objeto fantasma* a un objeto que no es referenciado en un programa, y que por tanto no sirve a ningún propósito. En un lenguaje que posea un recolector de basura, este marcará la memoria ocupada por el objeto como libre, aunque ésta todavía contendrá los datos del objeto hasta el momento que sea reescrita.

Objetos en la programación orientada a objetos

En programación orientada a objetos (POO), una instancia de programa (por ejemplo un programa ejecutándose en una computadora) es tratado como un conjunto dinámico de objetos interactuando entre sí. Los objetos en la POO extienden la noción más general descrita en secciones anteriores para modelar un tipo muy específico que está definido fundamentalmente por:

1. Atributos que representan los datos asociados al objeto, o lo que es lo mismo sus propiedades o características. Los atributos y sus valores en un momento dado, determinan el estado de un objeto.

2. métodos que acceden a los atributos de una manera predefinida e implementan el comportamiento del objeto.

Los atributos y métodos de un objetos están definidos por su clase, aunque (en un lenguaje dinámico como Python o Ruby) una instancia puede poseer atributos que no fueron definidos en su clase. Algo similar ocurre con los métodos, una instancia puede contener métodos que no estén definidos en su clase de la misma manera una clase puede declarar ciertos métodos como "métodos de clase", y estos (en dependencia del lenguaje) podrán estar o no presentes en la instancia.

En el caso de la mayoría de los objetos, los atributos solo pueden ser accedidos a través de los métodos, de esta manera es más fácil garantizar que los datos permanecerán siempre en un estado bien definido (Invariante de Clase).

En un lenguaje en el que cada objeto es creado a partir de una clase, un objeto es llamado una instancia de esa clase. Cada objeto pertenece a un tipo y dos objetos que pertenezcan a la misma clase tendrán el mismo tipo de dato. Crear una instancia de una clase es entonces referido como instanciar la clase.

En casi todos los lenguajes de programación orientados a objeto, el operador "punto" (.) es usado para referirse o "llamar" a un método particular de un objeto. Un ejemplo de lenguaje que no siempre usa este operador el C++ ya que para referirse a los métodos de un objeto a través de un puntero al objeto se utiliza el operador (->).

Consideremos como ejemplo una clase aritmética llamada Aritmética. Esta clase contiene métodos como "sumar", "restar", "multiplicar", "dividir", etc. que calculan el resultado de realizar estas operaciones sobre dos números.

Un objeto de esta clase puede ser utilizado para calcular el producto de dos números, pero primeramente sería necesario definir dicha clase y crear un objeto. En las secciones a continuación se muestra como hacer esto utilizando dos lenguajes de programación: C++ y Python.

Declaración de una clase

Esta clase podría ser definida de la siguiente manera en C++:

```
class Aritmetica
{
    public:
        inline int sumar (int a, int b) const
        {
            return a + b;
        }


        inline int restar (int a, int b) const
        {
            return a - b;
        }


        inline float multiplicar (int a, int b) const
        {
            return a * b;
        }


        inline float dividir (int a, int b) const
        {
            return a / b;
        }
};
```

o como sigue en Python:

```
class Aritmetica:
    def sumar(self, a, b):
```

```
    return a + b


def restar(self, a, b):
    return a - b


def multiplicar(self, a, b):
    return a * b


def dividir(self, a, b):
    return a / b
```

Instanciación de una clase en un objeto

Para crear un objeto de tipo 'Aritmetica' (*instanciar a Aritmetica*) en C++ se haría de la siguiente forma:

```
Aritmetica calculador = Aritmetica();
#Otra manera usando punteros
Aritmetica* calculador1 = new Aritmetica();
```

la misma operación usando python sería así:

```
calculador = Aritmetica()
```

Operando con un objeto

Una vez que tenemos un objeto de 'Aritmetica', podemos usarlo para realizar cálculos sobre dos números. En C++ contamos con dos objetos de ejemplo: "calculador" y "calculador1", en esta
última variable en realidad hemos almacenado la dirección de memoria del objeto creado. En este lenguaje esto sienta diferencias a la hora de utilizar el objeto.

Para calcular la suma entre 78 y 69 usando un objeto "calculador" necesitaríamos un código como el siguiente en C++:

```
int resultado = 0;
resultado = calculador.sumar(78, 69);
#Otra manera usando punteros
resultado = calculador1->sumar(78, 69);
```

ahora usando Python para sumar dos números con el objeto calculador:

```
resultado =
calculador.sumar(78, 69)
```

Otro ejemplo del mundo real de un objeto podría ser "mi perro", el cual es una instancia de un tipo (una clase) llamada "perro", la que es una subclase de la clase "animal". En el caso de un objeto polimórfico, algunos detalles de su tipo pueden ser ignorados, por ejemplo el objeto "mi perro" puede ser usado en un método que espera recibir un "animal". También podría usarse un objeto "gato", puesto que esta también pertenece a la clase "animal". Pero mientras es accedido como un "animal", algunos atributos de un "perro" o un "gato" permanecerán no disponibles, como la "cola", porque no todos los animales tienen colas.

Atributos dinámicos en objetos

Python y C++ son lenguajes con características muy diferentes. Python utiliza un sistema de tipos dinámico y C++ en cambio, uno estático o estricto. El sistema de tipos usado en Python permite al programador agregar atributos a una instancia que no han sido definidos en la clase que le dio origen, cosa que no es posible hacer en un lenguaje como C++, por ejemplo:

La clase siguiente en Python no define ningún atributo:

```
class Prueba(object):
    pass
```

pero es posible hacer lo siguiente:

```
1: p = Prueba()
2: p.unNumero = 3
3: print("Atributo unNumero de p = %s" % p.unNumero)
4: Atributo unNumero de p = 3
```

A la instancia de p creada en la línea 1, le es asignado en la línea 2 el valor "3", lo cual crea un atributo de nombre

unNumero en p de tipo "int" para almacenar el número 3.

Relaciones entre objetos

Como ya se ha dicho antes, un sistema orientado a objetos está caracterizado por objetos que interactúan entre si. Estas interacciones supone ciertos tipos de relaciones entre los objetos del sistema. La semántica que expresa un objeto en el sistema está determinado en primer lugar, por las relaciones que éste establece con otros objetos o conjunto de objetos. Tomemos como ejemplo un objeto fecha, del que sin establecer ningún tipo de relación, podría decirse que significa un día del año particular.

Pero si relacionamos ese objeto fecha con un objeto *Persona* de manera que represente la fecha en que esa persona nació, en ese contexto dado, el mismo objeto fecha adoptaría un significado diferente, el de un *cumpleaños*, aunque sigue siendo una fecha, ahora tiene otra idea asociada.

Las relaciones entre objetos no solo están definidas por los objetos que participan y la circunstancia que los relaciona, sino también por la cantidad de objetos (cardinalidad de la relación) y la dirección de la misma. Una relación puede tener cardinalidad:

• uno a uno, ejemplo: un auto tiene un motor

• uno a muchos, ejemplo: un auto tiene muchas ruedas

• muchos a muchos, ejemplo: un auto se puede servir en muchas gasolineras y una gasolinera puede servir a muchos autos.

y direccionalidad:

•unidireccional, ejemplo: un auto tiene cuatro ruedas.

•bidireccional

Las relaciones entre objetos más generales son las siguientes:

Composición

La composición (también conocida como relación asociativa) es un tipo de relación que se establece entre dos objetos que tienen comunicación persistente. Se utiliza para expresar que un par de objetos tienen una relación de dependencia para llevar a cabo su función, de modo que uno de los objetos involucrados *está compuesto por* el otro.

De manera práctica, es posible reconocer asociatividad entre dos objetos A y B si la proposición "A tiene un B" (o viceversa) es verdadera. Por ejemplo: "una computador tiene un disco duro" es verdadero, por tanto un objeto computador tiene una relación de composición con al menos un objeto disco duro.

Uso

Un objeto usa (conoce) a otro cuando puede enviarle mensajes, por ejemplo, para requerir de este algún servicio. La composición puede verse como un caso particular de esta relación.

Delegación

En ocasiones para lograr flexibilidad de diseño, un objeto es implementado de forma tal que este delegue parte de su funcionalidad en otro objeto. Esto es muy común en aplicaciones que hacen uso de interfaces gráficas de usuario, en las que los controles gráficos generales delegan la acción que se ejecutará ante determinado estímulo en otro objeto.

Objetos especializados

Algunos términos para tipos especializados de objetos son:

- Singleton: Un objeto del que solo puede existir una única instancia de su clase durante el tiempo de vida del programa.
- Functor : un objeto que puede ser utilizado como una función.
- Objeto inmutable: un objeto creado con un estado fijo y que no puede variar en el tiempo de vida del mismo.
- Objeto de primera clase: un objeto que puede ser utilizado sin restricciones.

- Contenedor: un objeto que contiene a otros objetos.
- Fábrica de objetos: un objeto cuyo propósito es crear otros objetos.
- Metaobjeto: un objeto a partir del cual se pueden crear otros objetos (comparable con una clase, la que no necesariamente es un objeto)
- Prototipo: un metaobjeto especializado a partir del cual se pueden crear otros objetos copiándolo.
- Objeto todopoderoso: un objeto que sabe *mucho* o hace *mucho*. Este es un ejemplo de antipatrón de diseño.
- Antiobjeto: una metáfora computacional útil para conceptualizar y solucionar problemas complejos, usualmente con aproximaciones paralelas.

Variables miembro

Las propiedades o atributos son características de los objetos. Cuando definimos una propiedad normalmente especificamos su nombre y su tipo. Nos podemos hacer a la idea de que las variables son algo así como el almacén de los datos de estado relacionados con los objetos.

Habitualmente, las variables miembro son privadas al objeto (siguiendo las directrices de diseño del Principio de ocultación) y su acceso se realiza mediante propiedades o métodos que realizan comprobaciones adicionales. Suelen denominarse con nombres.

Métodos en las clases

Implementan la funcionalidad asociada al objeto. Los métodos son el equivalente a las funciones en los lenguajes estructurados. Se diferencian de ellos en que es posible acceder a las variables de la clase de forma implícita.

Cuando se desea realizar una acción sobre un objeto, se dice que se le manda un mensaje invocando a un método que realizará la acción.

Habitualmente, los métodos suelen ser verbos. En Xbase++ los métodos se declaran como:

```
METHOD Metodo
```

Propiedades

Las propiedades son un tipo especial de métodos. Debido a que suele ser común que las variables miembro sean privadas para controlar el acceso y mantener la coherencia, surge la necesidad de permitir consultar o modificar su valor mediante pares de métodos: GetVariable y SetVariable.

Los lenguajes orientados a objetos más modernos (por ejemplo Java, C#) añaden la construcción de propiedad que es una sintaxis simplificada para dichos métodos:

```
tipo  Propiedad {
    get {
    }
    set {
    }
}
```

En Xbase++ las propiedades de asignación o acceso se declaran:

```
ACCESS ASSIGN METHOD Metodo VAR Propiedad

...

ACCESS ASSIGN METHOD Clase:Metodo( Propiedad )
    ::Propiedad:= Propiedad
RETURN::Propiedad
```

De esta forma es posible realizar operaciones sobre las propiedades como si fuesen variables normales, el compilador se encarga de crear el código apropiado que llame a la cláusula get o set según se necesite.
Las propiedades se denominan con nombres como las variables. Método (informática)

En la programación orientada a objetos, un método es una subrutina asociada exclusivamente a una clase (llamados métodos de clase o métodos estáticos) o a un objeto (llamados métodos de instancia). Análogamente a los procedimientos en los lenguajes imperativos, un método consiste generalmente de una serie de sentencias para llevar a cabo una acción, un juego de parámetros de entrada que regularán dicha acción y, posiblemente, un valor de salida (o valor de retorno) de algún tipo.

Algunos lenguajes de programación asumen que un método debe de mantener el invariante del objeto al que está asociado asumiendo también que éste es válido cuando el método es invocado. En lenguajes compilados dinámicamente, los métodos pueden ser objetos de primera clase, y en este caso se puede compilar un método sin asociarse a ninguna clase en particular, y luego asociar el vínculo o contrato entre el objeto y el método en tiempo de ejecución. En cambio en lenguajes no compilados dinámicamente o tipados estáticamente, se acude a precondiciones para regular los parámetros del método y postcondiciones para regular su salida (en caso de tenerla). Si alguna de las precondiciones o postcondiciones es falsa el método genera una excepción. Si el estado del objeto no satisface la invariante de su clase al comenzar o finalizar un método, se considera que el programa tiene un error de programación.

La diferencia entre un procedimiento (generalmente llamado "*función*" si devuelve un valor) y un método es que éste último, al estar asociado con un objeto o clase en particular, puede acceder y modificar los datos privados del objeto correspondiente de forma tal que sea consistente con el comportamiento deseado para el mismo. Así, es recomendable entender a un método no como una secuencia de instrucciones sino como la forma en que el objeto es útil (el método para hacer su trabajo). Por lo tanto, podemos considerar al método como el pedido a un objeto para que realice una

tarea determinada o como la vía para enviar un mensaje al objeto y que éste reaccione acorde a dicho mensaje.

Tipos de métodos

Como ya se mencionó, los métodos de instancia están relacionados con un objeto en particular, mientras que los métodos estáticos o de clase (también denominados métodos compartidos) están asociados a una clase en particular. En una implementación típica, a los métodos de instancia se les pasa una referencia oculta al objeto al que pertenecen, comúnmente denominada *this* o *self* (referencias a sí mismo por sus significados en inglés), para que puedan acceder a los datos asociados con el mismo. Un ejemplo típico de un método de clase sería uno que mantuviera la cuenta de la cantidad de objetos creados dentro de esa clase.

Los llamados *métodos obtener* y *métodos establecer* (en inglés *get* y *set*) proveen un mecanismo para leer y modificar (respectivamente) los datos *privados* que se encuentran almacenados en un objeto o clase.

Algunos lenguajes de programación requieren la definición de constructores, siendo estos métodos de instancia especiales llamados automáticamente cuando se crea una instancia de alguna clase. En Java y C++ se distinguen por tener el mismo nombre de la clases a la que están asociados. Lenguajes como Smalltalk no requieren constructores ni destructores.

Los *métodos de acceso* son un tipo de método normalmente pequeño y simple que se limita a proveer información acerca del estado de un objeto. Aunque introduce una nueva dependencia, la utilización de métodos es preferida a acceder directamente a la información para proveer de una nueva capa de abstracción (programación orientada a objetos). Por ejemplo, si una clase que modela una cuenta bancaria provee de un método de acceso *"obtenerBalance()"* en versiones posteriores de la clase se podría cambiar el código de dicho método substancialmente sin que el código dependiente de la clase tuviese que ser modificado (un cambio sería necesario siempre que el tipo de dato devuelto por el método cambie). Los métodos de acceso que pueden cambiar el estado de un objeto son llamados, frecuentemente, *métodos de actualización* ó *métodos de mutación*; a su vez, los objetos que proveen de dichos métodos son denominados *óbjetos mutables*.

Constructor (informática)

El objetivo de un constructor es el de inicializar un objeto cuando éste es creado. Asignaremos los valores iniciales así como los procesos que ésta clase deba realizar.

Se utiliza para crear tablas de métodos virtuales y poder así desarrollar el polimorfismo, una de las herramientas de la programación orientada a objetos (POO). Al utilizar un constructor, el

compilador determina cual de los objetos va a responder al mensaje (virtual) que hemos creado. Tiene un tipo de acceso, un nombre y un paréntesis.

Java

En java es un método especial dentro de una clase, que se llama automáticamente cada vez que se crea un objeto de esa clase.

Posee el mismo nombre de la clase a la cual pertenece y no puede regresar ningún valor (ni siquiera se puede especificar la palabra reservada `void`). Por ejemplo si añadiéramos a la clase SSuma un constructor, tendríamos que llamarlo también SSuma. Cuando en una clase no se escribe propiamente un constructor, java asume uno por defecto.

Constructor por defecto

Un constructor por defecto es un constructor sin parámetros que no hace nada. Sin embargo será invocado cada vez que se construya un objeto sin especificar ningún argumento, en cuyo caso el objeto será iniciado con los valores predeterminados por el sistema (los atributos numéricos a ceros, los alfanuméricos a nulos, y las referencias a objetos a `null`).

Ejemplo

Un constructor por defecto, para la clase SSuma quedaría así:

```
public SSuma() {}
```

Como se observa el constructor no posee ningún parámetro, ya que como no ha sido definido propiamente por el programador, Java lo hace por `default`.

Si se define un constructor con parámetros, (definido explícitamente) el constructor por `default` se reemplaza por éste.

Ahora podemos crear un constructor explícito para una clase simple, utilizando el nombre `Arychan` para la clase. `Arychan` es una clase que se refiere a una persona de cierta

edad, que posee un nombre y ciertos atributos, puede ser divertida, y hermosa, estos atributos serán del tipo cadena de caracteres (String) por lo que se agregará un atributo más llamado salario y será de tipo numérico.

Primero se declara la clase con sus atributos:

```
class Arychan
{
  //ATRIBUTOS
  private String nombre;
  private String descripción;
  private String formaDeSer;
  private double salario;


  //METODOS



  public Arychan() {} //CONSTRUCTOR SIN PARÁMETROS
  public Arychan(String nom, String des, String forma, double sal)
  {
      asignarNombre(nom); // nombre = nom;
      asignarDescripcion(des); // descripción = des; describir(forma);  //formaDeSer = forma; ingreso(sal); //salario = sal;
  }
  //...
}
```

El constructor sin parámetros es reemplazado por el constructor explícito. En este ejemplo se puede observar que los constructores preferentemente se declaran públicos para que puedan ser invocados desde cualquier parte.

Una línea como esta invocará al constructor sin parámetros:

```
Arychan ary01 = new Arychan();// invoca al constructor Arychan
```

El operador new crea un nuevo objeto, en este caso de la clase Arychan, y a continuación se invoca al constructor de la clase para realizar las operaciones de iniciación que estén programadas.

Ahora invocaremos al constructor con parámetros, recordemos que la clase Arychan es una persona con características como divertida, hermosa, mismas que pasaremos como argumentos. `Arychan ary02 = new Arychan("Ary", "hermosa", "divertida", 25000);`

Instancia

Una instancia de un programa es una copia de un version ejecutable del programa que ha sido escrito en la memoria del computador.

Una instancia de un programa es creado típicamente por el click de usuario en un icono de una interfaz Gráfica para usuarios GUI o por la entrada de un comando en una interfaz de linea de comandos CLI y presionando la tecla ENTER. Instancias de programas puede ser creado por otros programas.

Un programa es una secuencia de instrucciones que indica cuales operaciones se deben realizar sobre un conjunto de datos. Una version ejecutable de un programa, también llamado un *programa ejecutable*, es una version de un programa que es entendible para el CPU del computador y esta listo para funcionar tan pronto como se copia en memoria. Esto contrasta con la version de codigo fuente de un programa, el cual es la version originalmente escrita por lenguaje de alto nivel, y luego es traducido a lenguaje de maquinas por otro programa especializado llamado compilador.

Multitareas

Multitareas, permite a multiples programas aparentemente ejecutarse simultaneamente en el mismo computador, también permite que multiples instancias de un mismo programa se ejecuten simultaneamente, si el programa lo permite. Sin embargo, a veces se desea que exista un sola instancia del programa en el computador, y los lenguajes de programación proporcionan las técnicas que pueden utilizarse para hacer cumplir esto.

Programación Orientada a objetos
POO

En programación una instancia se produce con la creación de un objeto perteneciente a una clase (instanciar una clase), que hereda entonces sus atributos, propiedades y métodos para ser usados dentro de un programa, ya sea como contenedores de datos o como partes funcionales del programa al contener en su interior funcionalidades de tratamiento de datos y procesamiento de la información que ha sido programada con anterioridad en la clase a la que pertenece.

Herencia (informática)

Es una propiedad que permite que los objetos sean creados a partir de otros ya existentes, obteniendo características (métodos y atributos) similares a los ya existentes. Es la relación entre una clase general y otra clase más especifica. Es un mecanismo que nos permite crear

clases derivadas a partir de clase base, nos permite compartir automáticamente métodos y datos entre clases, subclases y objetos. Por ejemplo: Si declaramos una clase párrafo derivada de una clase texto, todos los métodos y variables asociadas con la clase texto, son automáticamente heredados por la subclase párrafo.

La herencia es uno de los mecanismos de la programación orientada a objetos, por medio del cual una clase se deriva de otra, llamada entonces *superclase*, de manera que extiende su funcionalidad. Una de sus funciones más importantes es la de proveer Polimorfismo y *late binding*.

La idea es la de partir de las situaciones más generales e ir *derivando* hacia las más particulares, creando categorías, de la misma forma en que piensa el ser humano.

Ejemplo en Java

```
public class Mamifero
{
private int patas;
private String
nombre; public void
imprimirPatas()
{
System.out.println(nombre + " tiene " + patas + " patas\n");
}

public Mamifero(String nombre, int patas)
{
this.nombre = nombre;
this.patas = patas;
}
}

public class Perro extends Mamifero
{
      public Perro(String nombre)
      {
      super(nombre, 4);
      }
}

public class Gato extends Mamifero
{
      public Gato(String nombre)
      {
      super(nombre, 4);
      }
}
```

```
public class
CreaPerro
{
public static void
main(String []args)
{
Perro bobi = new
Perro("Bobi");
bobi.imprimirPatas(); /*Está en la
clase mamífero*/
}
}
```

Se declaran las clases mamíferos, gato y perro, haciendo que gato y perro *sean unos* mamíferos (derivados de esta clase), y se ve como a través de ellos se nombra al animal pero así también se accede a patas dándole el valor por defecto para esa especie.

Clase Abstracta

La herencia permite que existan clases que nunca sean instanciadas directamente. En el ejemplo anterior, una clase "perro" heredaría los atributos y métodos de la clase "mamífero", así como también "gato", "delfín" o cualquier otra subclase; pero que ocurra que en el sistema no haya ningún objeto "mamífero" que no pertenezca a alguna de las subclases. En ese caso, a una clase así se la conocería como Clase Abstracta. La ausencia de instancias específicas es su única particularidad, para todo lo demás es como cualquier otra clase.

Redefinición

Si en una clase en particular se invoca a un método, y el método no está definido en la misma, es buscado automáticamente en las clases superiores. Sin embargo, si existieran dos métodos con el mismo nombre y distinto código, uno en la clase y otro en una superclase, se ejecutaría el de la clase, no el de la superclase.

Por lo general, siempre se puede acceder explícitamente al método de la clase superior mediante una sintaxis diferente, la cual dependerá del lenguaje de programación empleado.

Ventajas

- Ayuda a los programadores ahorrar código y tiempo, ya que si tiene una clase lista es solo de implementarla y listo todo el código de esta se resume a solo un llamado.
- Los objetos pueden ser construidos a partir de otros similares. Para ello es necesario que exista una clase base y una jerarquía (relacionamiento) de clases.
- La clase derivada puede heredar código y datos de la clase base, añadiendo código o modificando lo heredado.

- Las clases que heredan propiedades de otra clase pueden servir como clase base de otras.

Estereotipos de herencia

- Herencia simple: Un objeto puede extender las características de otro objeto y de ningún otro, es decir, que solo puede heredar o tomar atributos de un solo padre o de una sola clase.
- Herencia múltiple: Un objeto puede extender las características de uno o más objetos, es decir, puede tener varios padres. En este aspecto hay discrepancias entre los diseñadores de lenguajes. Algunos de ellos han preferido no admitir la herencia múltiple por las posibles coincidencias en nombres de métodos o datos miembros. Por ejemplo C++, Python permiten herencia múltiple, mientras que Java, Ada y C# sólo permiten herencia simple.

Polimorfismo

En programación orientada a objetos se denomina polimorfismo a la capacidad que tienen los objetos de una clase de responder al mismo mensaje o evento en función de los parámetros utilizados durante su invocación. Un objeto polimórfico es una entidad que puede contener valores de diferentes tipos durante la ejecución del programa.

Dicho de otra forma, el polimorfismo consiste en conseguir que un objeto de una clase se comporte como un objeto de cualquiera de sus subclases, dependiendo de la forma de llamar a los métodos de dicha clase o subclases. Una forma de conseguir objetos polimórficos es mediante el uso de punteros a la superclase. De esta forma podemos tener dentro de una misma estructura (arrays, listas, pilas, colas, ...) objetos de distintas subclases, haciendo que el tipo base de dichas estructuras sea un puntero a la superclase. Otros lenguajes nos dan la posibilidad de crear objetos polimórficos con el operador new. La forma de utilizarlo, por ejemplo en java, sería:

```
Superclase sup = new (Subclase);
```

En la práctica esto quiere decir que un puntero a un tipo puede contener varios tipos diferentes, no solo el creado. De esta forma podemos tener un puntero a un objeto de la clase Trabajador, pero este puntero puede estar apuntando a un objeto subclase de la anterior como podría ser Márketing, Ventas o Recepcionistas (todas ellas deberían ser subclase de Trabajador).

El concepto de polimorfismo se puede aplicar tanto a funciones como a tipos de datos. Así nacen los conceptos de *funciones polimórficas* y *tipos polimórficos*. Las primeras son aquellas funciones que pueden evaluarse o ser aplicadas a diferentes tipos de datos de forma indistinta; los *tipos polimórficos*, por su parte, son aquellos tipos de datos que contienen al menos un elemento cuyo tipo no está especificado.

Clasificación

Se puede clasificar el polimorfismo en dos grandes clases:

- Polimorfismo dinámico (o polimorfismo paramétrico) es aquél en el que el código no incluye ningún tipo de especificación sobre el tipo de datos sobre el que se trabaja. Así, puede ser utilizado a todo tipo de datos compatible.
- Polimorfismo estático (o polimorfismo *ad hoc*) es aquél en el que los tipos a los que se aplica el polimorfismo deben ser explicitados y declarados uno por uno antes de poder ser utilizados.

El polimorfismo dinámico unido a la herencia es lo que en ocasiones se conoce como programación genérica.

También se clasifica en herencia por redefinición de métodos abstractos y por método sobrecargado. El segundo hace referencia al mismo método con diferentes parámetros.

Otra clasificación agrupa los polimorfismo en dos tipos: Ad-Hoc que incluye a su vez sobrecarga de operadores y coerción, *Universal* (inclusión o controlado por la herencia, paramétrico o genericidad).

Ejemplo de polimorfismo

En este ejemplo haremos uso del lenguaje C++ para mostrar el polimorfismo. También se hará uso de las funciones virtuales puras de este lenguaje, aunque para que el polimorfismo funcione no es necesario que las funciones sean virtuales puras, es decir, perfectamente el código de la clase "superior" (en nuestro caso Empleado) podría tener código

```
class
Empleado {
```

```
ed:
    static const unsigned int SUELDO_BASE = 700; // Supuesto sueldo
    base para todos
```

```
c:
    /* OTROS MÉTODOS */
    virtual unsigned int sueldo() = 0;
};
```

```
class Director : public Empleado {
public:
    /* OTROS MÉTODOS */
    unsigned int sueldo() { return SUELDO_BASE*100; }
};

class Ventas : public Empleado {
private:
    unsigned int ventas_realizadas; // Contador de ventas realizadas
por el vendedor


public:
    /* OTROS MÉTODOS */
    unsigned int sueldo() { return SUELDO_BASE + ventas_realizadas*60;
}
};


class Mantenimiento : public Empleado {
public:
    /* OTROS MÉTODOS */
    unsigned int sueldo() { return SUELDO_BASE + 300; }
};


class Becario : public Empleado {
private:
```

```
    bool jornada_completa; // Indica si el becario trabaja a
    jornada completa


public:
    /* OTROS MÉTODOS */
    unsigned int sueldo() {
        if (jornada_completa) return SUELDO_BASE/2;
        else return SUELDO_BASE/4;
    }
};


/* AHORA HAREMOS USO DE LAS CLASES */
int main() {
    Empleado* e[4]; // Punteros a Empleado
    Director d;
    Ventas v; // Estas dos las declararemos como objetos
    normales en la pila


    e[0] = &d;//Asignamos a un puntero a Empleado la dirección
    de un objeto del tipo
    Director
    e[1] = &v; // Lo mismo con Ventas
    e[2] = new Mantenimiento();
    e[3] = new Becario();

    unsigned int sueldo = 0;
    for (int i = 0; i < 4; ++i) sueldo += e[i]->sueldo();
```

```
    cout << "Este mes vamos a gastar " << sueldo << " dinero en sueldos" << endl;
}
```

Bibliografía Recomendada:

- Khoshafian, S. Abnous, R. Object Orientation. John Wiley & Sons. 1995.
- Korson, T. McGregor, J. Understanding Object Oriented: a Unifying Paradigm. Communications of the ACM. 33(9). 9/1990.
- Eckel, B. Thinking in Java. Prentice-Hall. 1998.
- Gamma, E. Helm, R. Design Patterns. Addison-Wesley, 1995.
- The Java Tutorial. http://java.sun.com/
- Lalonde, W. Pugh, J. Inside Smalltalk, Vol. I y II. 05/1994.
- Oestereich, Bernd. “Developing Software with UML”. Addison-Wesley, 1997.

Bibliografía complementaria

- Atkinson, C. Object Oriented Reuse, Concurrency and Distribution. Addison-Wesley, 1991
- Carmichael, A. Object Development Methods, SIGS Books, 1994.
- Goldberg, A. Robson, D. Smalltalk 80: The Language, Addison-Wesley, 1989.
- Meyer, B. Object Oriented Software Construction. Prentice Hall, 1988.
- Shaw, M. Garlan,D. Software Architecture. Prentice-Hall, 1996.
- Stroustrup, B. The C++ Programming Language. Addison-Wesley, 1991.
- Cardelli, L. Wegner, P. On Understanding Types, Data Abstraction and Polymorphism. Computing Surveys, 17(4), 1985.

- Maes, P. Concepts and Experiments in Computational Reflection. Proceedings of OOPSLA 87.
- Meyer, B. Applying Design By Contract. IEEE Computer. 25(10), 10/1992.
- Snyder, A. The Essence of Objects: Concepts and Terms. IEEE Software, 10(1), 1/1993.
- Wegner, P. Dimensions of Object-Based Language Design. Proceedings of OOPSLA 87.
- Wegner, P. Dimensions of Object-Oriented Modeling. IEEE Computer, 25(10), 10/1992.

Sitios consultados

- http://djaramillo2dani.blogspot.mx/2011/04/guia-practica-uno-1-estructura-228106.html
- http://marcelo-trabajo.blogspot.mx/
- http://giomonografia.blogspot.mx/p/teorico.html
- http://diaolaya.weebly.com/1/post/2013/09/resumen-teora-de-compiladores.html
- http://lesbiamartinez.es.tl/teoria-sobre-el-lenguaje-de-programacion.htm
- https://es.scribd.com/doc/217164157/Tema-1
- http://cursos.aiu.edu/Lenguages%20de%20Programacion/PDF/Tema%201.pdf
- http://algoritmosylenguajes.blogspot.mx/2008/05/unidad-iii_31.html