

Tema 8.

PROGRAMACIÓN ORIENTADA A OBJETOS

I. CONCEPTOS BÁSICOS

1. Programación Orientada a Objeto (POO)

La programación Orientada a Objetos es una metodología que basa la estructura de los programas en torno a los objetos.

Los lenguajes de POO ofrecen medios y herramientas para describir los objetos manipulados por un programa. Más que describir cada objeto individualmente, estos lenguajes proveen una construcción (Clase) que describe a un conjunto de objetos que poseen las mismas propiedades.

2. Objeto

Es una entidad (tangible o intangible) que posee características y acciones que realiza por sí solo o interactuando con otros objetos.

Un objeto es una entidad caracterizada por sus atributos propios y cuyo comportamiento está determinado por las acciones o funciones que pueden modificarlo, así como también las acciones que requiere de otros objetos. Un objeto tiene identidad e inteligencia y constituye una unidad que oculta tanto datos como la descripción de su manipulación. Puede ser definido como una encapsulación y una abstracción: una encapsulación de atributos y servicios, y una abstracción del mundo real.

Para el contexto del Enfoque Orientado a Objetos (EOO) un objeto es una entidad que encapsula datos (atributos) y acciones o funciones que los manejan (métodos). También para el EOO un objeto se define como una instancia o particularización de una clase.

Los objetos de interés durante el desarrollo de software no sólo son tomados de la vida real (objetos visibles o tangibles), también pueden ser abstractos. En general son entidades que juegan un rol bien definido en el dominio del problema. Un libro, una persona, un carro, un polígono, son apenas algunos ejemplos de objeto.

Cada objeto puede ser considerado como un proveedor de servicios utilizados por otros objetos que son sus clientes. Cada objeto puede ser a la vez proveedor y cliente. De allí que un programa pueda ser visto como un conjunto de relaciones entre proveedores y clientes. Los servicios ofrecidos por los objetos son de dos tipos:

- 1.- Los datos, que llamamos atributos.
- 2.- Las acciones o funciones, que llamamos métodos.

Características Generales

- **Un objeto se identifica por un nombre o un identificador único que lo diferencia de los demás.** Ejemplo: el objeto Cuenta de Ahorros número 12345 es diferente al objeto Cuenta de Ahorros número 25789. En este caso el identificador que los hace únicos es el número de la cuenta.
- **Un objeto posee estados.** El estado de un objeto está determinado por los valores que poseen sus atributos en un momento dado.
- **Un objeto tiene un conjunto de métodos.** El comportamiento general de los objetos dentro de un sistema se describe o representa mediante sus operaciones o métodos. Los métodos se utilizarán para obtener o cambiar el estado de los objetos, así como para proporcionar un medio de comunicación entre objetos.
- **Un objeto tiene un conjunto de atributos.** Los atributos de un objeto contienen valores que determinan el estado del objeto durante su tiempo de vida. Se implementan con variables, constantes y estructuras de datos (similares a los campos de un registro).

- **Los objetos soportan encapsulamiento.** La estructura interna de un objeto normalmente está oculta a los usuarios del mismo. Los datos del objeto están disponibles solo para ser manipulados por los propios métodos del objeto. El único mecanismo que lo conecta con el mundo exterior es el paso de mensajes.
- **Un objeto tiene un tiempo de vida dentro del programa o sistema que lo crea y utiliza.** Para ser utilizado en un algoritmo el objeto debe ser creado con una instrucción particular (*New* ó *Nuevo*) y al finalizar su utilización es destruido con el uso de otra instrucción o de manera automática.

3. Clase

La clase es la unidad de modularidad en el EOO. La tendencia natural del individuo es la de clasificar los objetos según sus características comunes (clase). Por ejemplo, las personas que asisten a la universidad se pueden clasificar (haciendo abstracción) en estudiante, docente, empleado e investigador.

La clase puede definirse como la agrupación o colección de objetos que comparten una estructura común y un comportamiento común.

Es una plantilla que contiene la descripción general de una colección de objetos. Consta de atributos y métodos que resumen las características y el comportamiento comunes de un conjunto de objetos.

Todo objeto (también llamado instancia de una clase), pertenece a alguna clase. Mientras un objeto es una entidad concreta que existe en el tiempo y en el espacio, una clase representa solo una abstracción.

Todos aquellos objetos que pertenecen a la misma clase son descritos o comparten el mismo conjunto de atributos y métodos. Todos los objetos de una clase tienen el mismo formato y comportamiento, son diferentes únicamente en los valores que contienen sus atributos. Todos ellos responden a los mismos mensajes.

Su sintaxis algorítmica es:

Clase <Nombre de la Clase>

...

FClase <Nombre de la Clase>;

Características Generales

- **Una clase es un nivel de abstracción alto.** La clase permite describir un conjunto de características comunes para los objetos que representa. Ejemplo: La clase *Avión* se puede utilizar para definir los atributos (tipo de avión, distancia, altura, velocidad de crucero, capacidad, país de origen, etc.) y los métodos (calcular posición en el vuelo, calcular velocidad de vuelo, estimar tiempo de llegada, despegar, aterrizar, volar, etc.) de los objetos particulares *Avión* que representa.
- **Un objeto es una instancia de una clase.** Cada objeto concreto dentro de un sistema es miembro de una clase específica y tiene el conjunto de atributos y métodos especificados en la misma
- **Las clases se relacionan entre sí mediante una jerarquía.** Entre las clases se establecen diferentes tipos de relaciones de herencia, en las cuales la clase hija (subclase) hereda los atributos y métodos de la clase padre (superclase), además de incorporar sus propios atributos y métodos.
Ejemplos, Superclase: Clase *Avión*
Subclases de *Avión*: Clase *Avión Comercial*, *Avión de Combate*, *Avión de Transporte*
- Los nombres o identificadores de las clases deben colocarse en singular (clase *Animal*, clase *Carro*, clase *Alumno*).

4. Relación entre Clase y Objeto

Algorítmicamente, las clases son descripciones netamente estáticas o plantillas que describen objetos. Su rol es definir nuevos tipos conformados por atributos y operaciones.

Por el contrario, los objetos son instancias particulares de una clase. Las clases son una especie de molde de fábrica, en base al cual son construidos los objetos. Durante la ejecución de un programa sólo existen los objetos, no las clases.

La declaración de una variable de una clase NO crea el objeto.

La asociación siguiente: <Nombre_Clase> <Nombre_Variable>; (por ejemplo, Rectángulo R), no genera o no crea automáticamente un objeto Rectángulo. Sólo indica que R será una referencia o una variable de objeto de la clase Rectángulo.

La creación de un objeto, debe ser indicada explícitamente por el programador, de forma análoga a como inicializamos las variables con un valor dado, sólo que para los objetos se hace a través de un método Constructor (ver punto Métodos).

5. Atributo

Son los datos o variables que caracterizan al objeto y cuyos valores en un momento dado indican su estado.

Un atributo es una característica de un objeto. Mediante los atributos se define información oculta dentro de un objeto, la cual es manipulada solamente por los métodos definidos sobre dicho objeto

Un atributo consta de un nombre y un valor. Cada atributo está asociado a un tipo de dato, que puede ser simple (entero, real, lógico, carácter, *string*) o estructurado (arreglo, registro, archivo, lista, etc.)

Su sintaxis algorítmica es: <Modo de Acceso> <Tipo de dato> <Nombre del Atributo>;

Los modos de acceso son:

- **Público:** Atributos (o Métodos) que son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella. Este modo de acceso también se puede representar con el símbolo +
- **Privado:** Atributos (o Métodos) que sólo son accesibles dentro de la implementación de la clase. También se puede representar con el símbolo -
- **Protegido:** Atributos (o Métodos) que son accesibles para la propia clase y sus clases hijas (subclases). También se puede representar con el símbolo #

6. Método

Son las operaciones (acciones o funciones) que se aplican sobre los objetos y que permiten crearlos, cambiar su estado o consultar el valor de sus atributos.

Los métodos constituyen la secuencia de acciones que implementan las operaciones sobre los objetos. La implementación de los métodos no es visible fuera de objeto.

La *sintaxis algorítmica* de los métodos expresados como funciones y acciones es:

Para funciones se pueden usar cualquiera de estas dos sintaxis:

<Modo de Acceso> **Función** <Nombre> [(Lista Parámetros)]: <Descripción del Tipo de datos>

Para acciones:

<Modo de Acceso> **Acción** <Nombre> [(Lista Parámetros)] donde los parámetros son opcionales

Ejemplo: Un rectángulo es un objeto caracterizado por los atributos Largo y Ancho, y por varios métodos, entre otros Calcular su área y Calcular su perímetro.

Características Generales

- Cada método tiene un nombre, cero o más parámetros (por valor o por referencia) que recibe o devuelve y un algoritmo con el desarrollo del mismo.
- En particular se destaca el método constructor, que no es más que el método que se ejecuta cuando el objeto es creado. Este constructor suele tener el mismo nombre de la clase/ objeto, pero aunque es una práctica común, el método constructor no necesariamente tiene que llamarse igual a la clase (al menos, no en pseudocódigo). Es un método que recibe cero o más parámetros y lo usual es que inicialicen los valores de los atributos del objeto.
- En lenguajes como Java y C++ se puede definir más de un método constructor, que normalmente se diferencian entre sí por la cantidad de parámetros que reciben.
- Los métodos se ejecutan o activan cuando el objeto recibe un mensaje, enviado por un objeto o clase externo al que lo contiene, o por el mismo objeto de manera local.

Creación de Objetos y Métodos Constructores:

Cada objeto o instancia de una clase debe ser creada explícitamente a través de un método u operación especial denominado Constructor. Los atributos de un objeto toman valores iniciales dados por el constructor. Por convención el método constructor tiene el mismo nombre de la clase y no se le asocia un modo de acceso (es público).

Algunos lenguajes proveen un método constructor por defecto para cada clase y/o permiten la definición de más de un método constructor.

Método de Destructores de objetos:

Los objetos que ya no son utilizados en un programa, ocupan inútilmente espacio de memoria, que es conveniente recuperar en un momento dado. Según el lenguaje de programación utilizado esta tarea es dada al programador o es tratada automáticamente por el procesador o soporte de ejecución del lenguaje.

En la notación algorítmica NO tomaremos en cuenta ese problema de administración de memoria, por lo tanto no definiremos formas para destruir objetos. En cambio al utilizar lenguajes de programación si debemos conocer los métodos destructores suministrados por el lenguaje y utilizarlos a fin de eliminar objetos una vez no sean útiles.

7. Mensaje

Es la petición de un objeto a otro para solicitar la ejecución de alguno de sus métodos o para obtener el valor de un atributo público.

Estructuralmente, un mensaje consta de 3 partes:

- Identidad del receptor: Nombre del objeto que contiene el método a ejecutar.
- Nombre del método a ejecutar: Solo los métodos declarados públicos.
- Lista de Parámetros que recibe el método (cero o mas parámetros)

Su sintaxis algorítmica es: **<Variable_Objeto>.<Nombre_Método> ([<Lista de Parámetros>]);**

Cuando el objeto receptor recibe el mensaje, comienza la ejecución del algoritmo contenido dentro del método invocado, recibiendo y/o devolviendo los valores de los parámetros correspondientes, si los tiene ya que son opcionales: ([])

Ejemplo 1, Definición de la Clase Rectángulo

IMPLEMENTACIÓN

Clase Rectángulo;

// Atributos

Privado:

Real Largo, Ancho;

// Métodos

// método constructor

Acción Rectángulo(Real lar, anc);

Largo = lar;

Ancho = anc;

FAcción;

Público Función Área: Real

// Retorna el área o superficie ocupada por el rectángulo

retornar(Largo * Ancho);

FFunción Área;

Público Función Perímetro: Real

// Retorna el perímetro del rectángulo

retornar(2 * (Largo + Ancho));

FFunción Perímetro;

FClase Rectángulo;

// Uso de la clase rectángulo

Acción Principal

Rectángulo R; // se declara una variable de tipo objeto Rectángulo, a la cual llamaremos R

Real L, A; // se declaran las variables reales L y A para largo y ancho del rectángulo

Escribir("Suministre a continuación los valores para el largo y el ancho");

Leer(L); Leer(A);

R.Rectángulo(L, A);

Escribir("Resultados de los cálculos:");

Escribir("Área: " + R.Área + " - Perímetro " + R.Perímetro);

FAcción Principal;

Rectángulo
<u>Privado</u> <u>Real</u> Largo
<u>Privado</u> <u>Real</u> Ancho
<u>Constructor</u> <u>Acción</u> Rectángulo(lar, anc)
<u>Público</u> <u>Función</u> Área: <u>Real</u>
<u>Público</u> <u>Función</u> Perímetro: <u>Real</u>

II. REPRESENTACIÓN EN NOTACIÓN ALGORÍTMICA DE UNA CLASE

Las clases son el elemento principal dentro del enfoque orientado a objetos. En este lenguaje las declaraciones forman parte de su propio grupo, el grupo [Clases]. Cada una de las declaraciones de clase debe tener el siguiente formato:

```
Clase <Identificador> [Hereda de: <Clases>]  
  Atributos  
    Público:  
      [Constantes]; [Variables]; o [Estructuras];  
    Privado:  
      [Constantes]; [Variables]; o [Estructuras];  
    Protegido:  
      [Constantes]; [Variables]; o [Estructuras];  
  
  Operaciones  
    Público:  
      [Métodos]  
    Privado:  
      [Métodos]  
    Protegido:  
      [Métodos]  
FClase <Identificador>
```

En el caso de la especificación de los Atributos o de los Métodos los modos de acceso Público (+), Privado (-) o Protegido (#) pueden omitirse, solamente en el caso en el que los grupos [Constantes], [Estructuras] y [Variables] son vacíos, es decir, no se estén declarando Atributos ó Métodos.

NOTA: Otra forma de especificar los modos de acceso es colocándolo antes del nombre DE CADA UNO de los atributos o métodos

III. DIAGRAMAS DE CLASE

La representación gráfica de una o varias clases se hará mediante los denominados Diagramas de Clase. Para los diagramas de clase se utilizará la notación que provee el Lenguaje de Modelación Unificado (UML, ver www.omg.org), a saber:

- Las clases se denotan como rectángulos divididos en tres partes. La primera contiene el nombre de la clase, la segunda contiene los atributos y la tercera los métodos.
- Los modificadores de acceso a datos y operaciones, a saber: público, protegido y privado; se representan con los símbolos +, # y – respectivamente, al lado derecho del atributo. (+ público, # protegido, - privado).

En la figura se muestra el método “color” que no tiene ningún parámetro y retorna un valor entero y el método “modificar_tamaño” que tiene un real como parámetro y no retorna nada (es una acción).

Notación:	Ejemplos	
Nombre Clase	Ventana	Rectángulo
Atributos	+ <u>Real</u> área; # <u>Lógico</u> visible;	<u>Privado</u> <u>Real</u> Lado
Métodos	+ color: <u>Entero</u> + modificar_tamaño(<u>Real</u> porcentaje)	<u>Acción</u> Cuadrado(<u>Entero</u> lad) <u>Público</u> Función Área: <u>Real</u> <u>Público</u> Función Perímetro: <u>Real</u>

IV. RELACIONES ENTRE CLASES

Las clases no se construyen para que trabajen de manera aislada, la idea es que ellas se puedan relacionar entre sí, de manera que puedan compartir atributos y métodos sin necesidad de rescribirlos.

La posibilidad de establecer jerarquías entre las clases es una característica que diferencia esencialmente la programación orientada a objetos de la programación tradicional, ello debido fundamentalmente a que permite extender y reutilizar el código existente sin tener que rescribirlo cada vez que se necesite.

Los cuatro tipos de relaciones entre clases estudiados en este curso serán:

- Herencia (Generalización / Especialización o Es-un)
- Agregación (Todo / Parte o Forma-parte-de)
- Composición (Es parte elemental de)
- Asociación (entre otras, la relación Usa-a)

1. Relación de Herencia (Generalización / Especialización, Es un)

Es un tipo de jerarquía de clases, en la que cada subclase contiene los atributos y métodos de una (herencia simple) o más superclases (herencia múltiple).

Mediante la herencia las instancias de una clase hija (o subclase) pueden acceder tanto a los atributos como a los métodos públicos y protegidos de la clase padre (o superclase). Cada subclase o clase hija en la jerarquía es siempre una extensión (esto es, conjunto estrictamente más grande) de la(s) superclase(s) o clase(s) padre(s) y además incorporar atributos y métodos propios, que a su vez serán heredados por sus hijas.

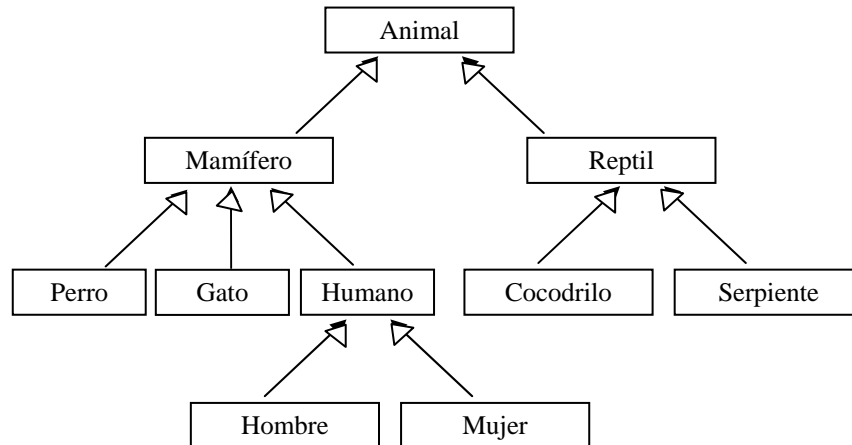
Representación:

- En la notación algorítmica: Se coloca el nombre de la clase padre después de la frase Hereda de del encabezado de la clase y se usan sus atributos y métodos públicos o protegidos. Ejemplo: Clase Punto3D Hereda de Punto2D

- En el diagrama de clases: La herencia se representa mediante una relación de generalización/especificación, que se denota de la siguiente forma:

clase hija —————> clase madre subclase —————> superclase

Ejemplo: El siguiente diagrama de clases muestra la relación de Herencia entre la Clase Animal y sus hijas





2. Relación de Agregación (Todo / Parte, Forma parte de)

Es una relación que representa a los objetos compuestos por otros objetos. Indica Objetos que a su vez están formados por otros. El objeto en el nivel superior de la jerarquía es el todo y los que están en los niveles inferiores son sus partes o componentes.

Representación en el Diagrama de Clase

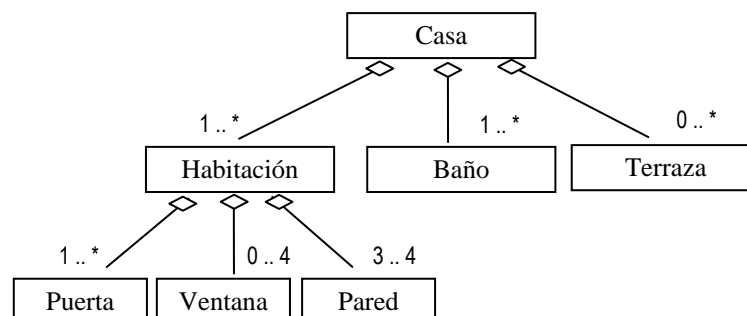
La relación forma parte de, no es más que una asociación, que se denota:

Nombre del objeto de nivel superior  multiplicidad Nombre del objeto que lo compone carro  4 .. * ruedas

Si motor forma parte de carro, la flecha apunta a la clase motor, y el diamante va pegado a carro.

La multiplicidad es el rango de cardinalidad permitido que puede asumir la asociación, se denota LI..LS. Se puede usar * en el límite superior para representar una cantidad ilimitada (ejemplo: 3..*).

Ejemplo: Objeto Casa descrito en términos de sus componentes

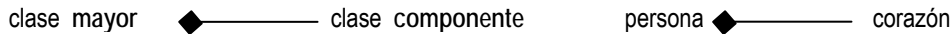


3. Relación de Composición

Un componente es parte esencial de un elemento. La relación es más fuerte que el caso de agregación, al punto que si el componente es eliminado o desaparece, la clase mayor deja de existir.

Representación en el Diagrama de Clases

La relación de *composición*, se denota de la siguiente forma:



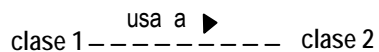
4. Relación de Asociación («uso», usa, cualquier otra relación)

Es una asociación que se establece cuando dos clases tienen una dependencia de utilización, es decir, una clase utiliza atributos y/o métodos de otra para funcionar. Estas dos clases no necesariamente están en jerarquía, es decir, no necesariamente una es clase padre de la otra, a diferencia de las otras relaciones de clases.

El ejemplo mas común de esta relación es de objetos que son utilizados por los humanos para alguna función, como Lápiz (se usa para escribir), tenedor (se usa para comer), silla (se usa para sentarse), etc. Otro ejemplo son los objetos Cajero y Cuenta. El Cajero “usa a” la cuenta para hacer las transacciones de consulta y retiro y verificar la información del usuario.

Representación en el Diagrama de Clases

La relación de *uso*, se denota con una dependencia estereotipada:



Ejemplos:



V. FUNDAMENTOS DEL ENFOQUE ORIENTADO A OBJETO

El Enfoque Orientado a Objeto se basa en cuatro principios que constituyen la base de todo desarrollo orientado a objetos. Estos principios son: la Abstracción, el Encapsulamiento, la Modularidad y la Herencia.

Otros elementos a destacar (aunque no fundamentales) en el EOO son: Polimorfismo, Enlace dinámico (o *binding*), Concurrencia y Persistencia.

Fundamento 1: Abstracción

Es el principio de ignorar aquellos aspectos de un fenómeno observado que no son relevantes, con el objetivo de concentrarse en aquellos que si lo son. Una abstracción denota las características esenciales de un objeto (datos y operaciones), que lo distingue de otras clases de objetos. Decidir el conjunto correcto de abstracciones de un determinado dominio, es el problema central del diseño orientado a objetos.

Los mecanismos de abstracción son usados en el EOO para extraer y definir del medio a modelar, sus características y su comportamiento. Dentro del EOO son muy usados mecanismos de abstracción: la Generalización, la Agregación y la clasificación.

- La generalización es el mecanismo de abstracción mediante el cual un conjunto de clases de objetos son agrupados en una clase de nivel superior (Superclase), donde las semejanzas de las clases constituyentes (Subclases) son enfatizadas, y las diferencias entre ellas son ignoradas. En consecuencia, a través de la generalización, la superclase almacena datos generales de las subclases, y las subclases almacenan sólo datos particulares. La especialización es lo contrario de la generalización. La clase Médico es una especialización de la clase Persona, y a su vez, la clase Pediatra es una especialización de la superclase Médico.
- La agregación es el mecanismo de abstracción por el cual una clase de objeto es definida a partir de sus partes (otras clases de objetos). Mediante agregación se puede definir por ejemplo un computador, por descomponerse en: la CPU, la ULA, la memoria y los dispositivos periféricos. El contrario de agregación es la descomposición.
- La clasificación consiste en la definición de una clase a partir de un conjunto de objetos que tienen un comportamiento similar. La ejemplificación es lo contrario a la clasificación, y corresponde a la instanciación de una clase, usando el ejemplo de un objeto en particular.

Fundamento 2: Encapsulamiento (Ocultamiento de Información)

Es la propiedad del EOO que permite ocultar al mundo exterior la representación interna del objeto. Esto quiere decir que el objeto puede ser utilizado, pero los datos esenciales del mismo no son conocidos fuera de él.

La idea central del encapsulamiento es esconder los detalles y mostrar lo relevante. Permite el ocultamiento de la información separando el aspecto correspondiente a la especificación de la implementación; de esta forma, distingue el "qué hacer" del "cómo hacer". La especificación es visible al usuario, mientras que la implementación se le oculta.

El encapsulamiento en un sistema orientado a objeto se representa en cada clase u objeto, definiendo sus atributos y métodos con los siguientes modos de acceso:

- Público (+) Atributos o Métodos que son accesibles fuera de la clase. Pueden ser llamados por cualquier clase, aun si no está relacionada con ella.
- Privado (-) Atributos o Métodos que solo son accesibles dentro de la implementación de la clase.
- Protegido (#): Atributos o Métodos que son accesibles para la propia clase y sus clases hijas (subclases).

Los atributos y los métodos que son públicos constituyen la interfaz de la clase, es decir, lo que el mundo exterior conoce de la misma.

Normalmente lo usual es que se oculten los atributos de la clase y solo sean visibles los métodos, incluyendo entonces algunos de consulta para ver los valores de los atributos. El método constructor (Nuevo, *New*) siempre es Público.

Fundamento 3: Modularidad

Es la propiedad que permite tener independencia entre las diferentes partes de un sistema. La modularidad consiste en dividir un programa en módulos o partes, que pueden ser compilados separadamente, pero que tienen conexiones con otros módulos. En un mismo módulo se suele colocar clases y objetos que guarden una estrecha relación. El sentido de modularidad está muy relacionado con el ocultamiento de información.

Fundamento 4: Herencia

Es el proceso mediante el cual un objeto de una clase adquiere propiedades definidas en otra clase que lo preceda en una jerarquía de clasificaciones. Permite la definición de un nuevo objeto a partir de otros, agregando las diferencias entre ellos (Programación Diferencial), evitando repetición de código y permitiendo la reusabilidad.

Las clases heredan los datos y métodos de la superclase. Un método heredado puede ser sustituido por uno propio si ambos tienen el mismo nombre.

La herencia puede ser simple (cada clase tiene sólo una superclase) o múltiple (cada clase puede tener asociada varias superclases). La clase Docente y la clase Estudiante heredan las propiedades de la clase Persona (superclase, herencia simple). La clase Preparador (subclase) hereda propiedades de la clase Docente y de la clase Estudiante (herencia múltiple).

Fundamento 5: Polimorfismo

Es una propiedad del EOO que permite que un método tenga múltiples implementaciones, que se seleccionan en base al tipo objeto indicado al solicitar la ejecución del método.

El polimorfismo operacional o Sobrecarga operacional permite aplicar operaciones con igual nombre a diferentes clases o están relacionados en términos de inclusión. En este tipo de polimorfismo, los métodos son interpretados en el contexto del objeto particular, ya que los métodos con nombres comunes son implementados de diferente manera dependiendo de cada clase.

Por ejemplo, el área de un cuadrado, rectángulo y círculo, son calculados de manera distinta; sin embargo, en sus clases respectivas puede existir la implementación del área bajo el nombre común Área. En la práctica y dependiendo del objeto que llame al método, se usará el código correspondiente.

Ejemplos, Superclase: Clase Animal

Subclases: Clases Mamífero, Ave, Pez

Se puede definir un método Comer en cada subclase, cuya implementación cambia de acuerdo a la clase invocada, sin embargo el nombre del método es el mismo.

Mamífero.Comer \neq Ave.Comer \neq Pez.Comer

Otro ejemplo de polimorfismo es el operador +. Este operador tiene dos funciones diferentes de acuerdo al tipo de dato de los operandos a los que se aplica. Si los dos elementos son numéricos, el operador + significa suma algebraica de los mismos, en cambio si por lo menos uno de los operandos es un String o Carácter, el operador es la concatenación de cadenas de caracteres.

Otro ejemplo de sobrecarga es cuando tenemos un método definido originalmente en la clase madre, que ha sido adaptado o modificado en la clase hija. Por ejemplo, un método Comer para la clase Animal y otro Comer que ha sido adaptado para la clase Ave, quien está heredando de la clase Animal.

Cuando se desea indicar que se está invocando (o llamando) a un método sobrecargado y que pertenece a otra clase (por ejemplo, a la clase padre) lo indicamos con la siguiente sintaxis:

Clase_Madre::nombre_método;

Para el ejemplo, la llamada en la clase hija Ave del método sobrecargado Comer de la clase madre Animal sería:

Animal::Comer

VI. EJEMPLOS

Ejemplo 1, Definición de la Clase Rectángulo

```
Clase Rectángulo;  
    // Atributos  
    Privado:  
        Real Largo, Ancho;  
  
    // Métodos  
    Constructor Rectángulo(Real lar, anc);  
        Largo = lar;  
        Ancho = anc;  
    FConstructor;  
  
    Público Función Área: Real  
        // Retorna el área o superficie ocupada por el rectángulo  
        retornar(Largo * Ancho);  
    FFunción Área;  
  
    Público Función Perímetro: Real  
        // Retorna el perímetro del rectángulo  
        retornar(2 * (Largo + Ancho));  
    FFunción Perímetro;  
FClase Rectángulo;
```

Rectángulo
- <u>Real</u> Largo - <u>Real</u> Ancho
+ <u>Acción</u> Rectángulo + <u>Función</u> Área: <u>Real</u> + <u>Función</u> Perímetro: <u>Real</u>

```
Clase Principal  
Acción Usa_Rectángulo  
    Rectángulo R; // se declara una variable de tipo objeto Rectángulo, a la cual llamaremos R  
    Real L, A; // se declaran las variables reales L y A para leer el largo y ancho dado por el usuario  
    Escribir("Suministre a continuación los valores para el largo y el ancho");  
    Leer(L, A);  
    R.Rectángulo(L, A);  
    Escribir("Resultados de los cálculos:");  
    Escribir("Área: " + R.Área + " - Perímetro " + R.Perímetro);  
  
    FAcción Usa_Rectángulo;  
FClase Principal;
```

Ejemplo 2, Clase que implementa un punto en el plano real

Clase Punto2D

```
//Atributos
Protegido Real X, Y; // representan las coordenadas X e Y de un punto

// Métodos u operaciones
Público:
    //constructores
    Acción Punto2D
        X = 0; Y = 0;
    FAcción;

    Acción Punto2D(Real CX, CY)
        X = CX; Y = CY;
    FAcción;

    // otros métodos de la clase
    Función CoordenadaX: Real
    // devuelve el valor de la coordenada X del punto que hace la llamada
        retornar(X);
    Ffunción;

    Función CoordenadaY: Real
    // devuelve el valor de la coordenada Y del punto que hace la llamada
        retornar(Y);
    Ffunción;

    Acción CambiarX(Real NX)
    // modifica el valor de la coordenada X del punto que hace la llamada
        X = NX;
    Ffunción;

    Acción CambiarY(Real NY)
    // modifica el valor de la coordenada Y del punto que hace la llamada
        Y = NY;
    Ffunción;

    Función Distancia(Real CX, CY): Real
    // devuelve la distancia entre el punto actual y otro con coordenadas CX y CY
        Real D; // variable donde se almacenará la distancia
        D = (RestaC(X, CX) + RestaC(Y, CY)) ^ (1/2);
        retornar(D);
    Ffunción;

    Función Distancia(Ref Punto2D P): Real;
    //devuelve la distancia entre el punto actual y otro punto P
        retornar(Distancia(P.X, P.Y));
    Ffunción Distancia;
```

```
Protegido Función RestaC (Real X,Y): Real;  
    // devuelve la resta entre X y Y elevada al cuadrado  
    retornar((X - Y) ^ 2));  
Ffunción Resta_C;  
FClase Punto2D;  
  
Clase Prueba_Punto  
  
    Acción Principal  
        Real CX, CY;    // valores para las coordenadas del punto a crear  
        Real D;        // contiene la distancia entre los puntos  
        Escribir ("Introduzca las coordenadas X e Y del punto a crear: ");  
        Leer(CX, CY);  
        Punto P = nuevo Punto2D(CX , CY);  
        Punto P1 = nuevo Punto2D;    // crea un punto en el origen, es decir, en el punto (0.0 , 0.0)  
        D = P.Distancia(P1.CoordenadaX , P1.CoordenadaY);  
        Escribir ("Distancia entre el punto P y el origen: " + D);  
        P1.CambiarX (15.0); P1.CambiarY(18.75); // se cambia a al punto (15.0 , 18.75)  
        Escribir ("Nuevo punto P1: (" + P1.CoordenadaX + " , "+ P1.CoordenadaY + ")");  
        D = P.Distancia(P1);  
        Escribir ("La nueva Distancia entre P y P1 es: "+ D);  
  
    fAcción Principal;  
  
FClase Prueba_Punto;
```

Ejemplo 3, Clase Punto3D que se deriva de la clase Punto2D

```
Clase Punto3D Hereda de Punto2D;  
    // Atributos  
    Protegido Real Z;    // coordenada Z del punto  
    // Métodos u operaciones  
    Público:  
        // métodos constructores Punto3D  
        Acción Punto3D  
            // Crea un punto con coordenadas (0,0,0) reutilizando el constructor de un punto de 2D  
            Punto2D;  
            Z = 0;  
        FAcción;  
  
        Acción Punto3D(Real CX, CY, CZ)  
            // Crea un punto de tres coordenadas reutilizando el constructor de un punto de 2D  
            Punto2D(CX, CY);  
            Z = CZ;  
        FAcción;
```

```
// otros métodos de la clase
Función CoordenadaZ: Real
// devuelve el valor de la coordenada Z del punto que hace la llamada
    retornar(Z);
Ffunción;
```

```
Acción CambiarZ(Real NZ)
// modifica el valor de la coordenada Z del punto que hace la llamada
    Z = NZ;
Ffunción;
```

```
Función Distancia3D(Real CX, CY, CZ): Real
// devuelve la distancia entre el punto actual y otro con coordenadas CX, CY y CZ
    Real D;          //contiene la distancia
    D = RestaC(X, CX) + RestaC(Y, CY) + RestaC(Z, CZ);
    D = D ^ (1/2);
    retornar(D);
Ffunción;
```

```
Función Distancia3D(Ref Punto3D P): Real
// devuelve la distancia entre el punto actual y otro punto P
    retornar(Distancia3D(P.X, P.Y, P.Z));
Ffunción;
```

FClase Punto3D



Plantea tus reflexiones y respuestas a:

1. ¿Qué atributos y métodos hereda la clase Punto3D de la clase Punto2D?
2. ¿Qué atributos y métodos hereda la clase Punto2D de la clase Punto3D?
3. ¿Quiénes pueden usar los atributos y métodos de Punto2D?
4. Plantea un algoritmo principal que utilice las clases Punto2D y Punto3D

Ejemplo 4, Clase Cuenta Bancaria

Clase CuentaBancaria

```
// atributos
Privado Entero Saldo;
Privado String NroCuenta;
Privado String Titular;

// métodos
Acción CuentaBancaria(Entero montoInicial; String num, nombre)
    // asigna a los atributo de la clase sus valores iniciales
    Saldo = montoInicial;
    NroCuenta = num;
    Titular = nombre;
Facción;

Público Acción depositar(Entero cantidad)
    // incrementa el saldo de la cuenta
    Saldo = Saldo + cantidad;
Facción;

Público Acción retirar(Entero cantidad)
    // disminuye el saldo de la cuenta
    Saldo = Saldo - cantidad;
Facción;

Público Función obtenerSaldo: Entero
    // permite conocer el monto disponible o saldo de la cuenta
    Retornar(saldo);
FFunción;
```

FinClase CuentaBancaria



Reflexiona y plantea propuestas para:

5. Crear un algoritmo principal que utilice la clase CuentaBancaria, creando varios objetos cuenta y actualizando sus saldos. Recuerda agregar en tu algoritmo principal las solicitudes de datos al usuario que consideres necesarias y las validaciones de estos datos de entrada
6. ¿Desde tu algoritmo principal podrías actualizar directamente el valor del atributo saldo? ¿Cómo lo puedes hacer?
7. Utiliza los métodos de la clase CuentaBancaria para verificar, en el algoritmo principal, que no se retire un monto que el saldo no puede cubrir

Ejemplo 5, Clase Raíces Ecuación de 2do Grado

Desarrollar un algoritmo bajo el enfoque orientado a objetos, que permita calcular las raíces (reales e imaginarias) de una ecuación de segundo grado ax^2+bx+c , dados los coeficientes a, b y c, con $a>0$.

Análisis:

El objeto principal a ser tratado en este problema es la ecuación y se implementará a través de la clase Ecuación2doGrado.

Se considerará que cada ecuación tiene asociadas como atributos sus raíces, las cuales deben ser calculadas por el método constructor a través de una acción llamada Resolver. El método constructor tendrá tres parámetros, que son los coeficientes de la ecuación (a, b y c).

La clase Ecuación2doGrado tendrá un método público para escribir las raíces llamado Imprimir.

Ecuación2doGrado

Privado Real RaízReal1

Privado Real RaízImaginaria1

Privado Real RaízReal2

Privado Real RaízImaginaria2

Constructor Ecuación2doGrado(Real a, b, c)

// Crea un Objeto Ecuacion2doGrado

Privado Acción Resolver(Real a, b, c)

// Calcula las raíces de la ecuación

Público Acción Imprimir

// Muestra las raíces de la ecuación

Clase Ecuación2doGrado;

Privado Real r1, r2, i1, i2;

Constructor Ecuación2doGrado(Real a, b, c);

Resolver(a,b,c);

FConstructor;

Privado Acción Resolver(Real a,b,c);

// Calcula las raíces de la ecuación ax^2+bx+c

Real delta;

delta = b * b - (4*a*c);

Si delta ≥ 0 entonces // Cálculo de raíces reales

Si b > 0 entonces r1 = - (b + delta^(1/2)) / (2*a);

sino r1 = - (delta^(1/2) - b) / (2*a);

Fsi;

r2 = c / (r1*a); i1 = 0; i2 = 0;

sino // Cálculo de raíces complejas

r1 = -b / (2*a); r2 = -b / (2*a);

i1 = (-delta)^(1/2) / (2*a);

i2 = -i1;

Fsi;

FAcción Resolver;

Público Acción Imprimir;

Muestra las raíces de la ecuación

Escribir(r1, i1);

Escribir(r2, i2);

FAcción Imprimir;

FClase Ecuación2doGrado;

// Usamos la clase Ecuacion2doGrado

Acción CalcularEcuaciones

Real a,b,c;

Ecuación2doGrado E, F;

Escribir("suministre coeficientes ecuación 1");

Leer(a, b, c);

E.Ecuación2doGrado(a, b, c);

E.Imprimir;

Escribir("suministre coeficientes ecuación 2");

Leer(a, b, c);

F.Ecuación2doGrado(a, b, c);

F.Imprimir;

FAcción;

CONCLUSIONES

En la Programación Orientada a Objetos los programas son representados por un conjunto de objetos que interactúan. Un objeto engloba datos y operaciones sobre estos datos.

La Programación Orientada a Objetos constituye una buena opción a la hora de resolver un problema, sobre todo cuando éste es muy extenso. En este enfoque de programación, se facilita evitar la repetición de código, no sólo a través de la creación de clases que hereden propiedades y métodos de otras, sino además que el código es reutilizable por sistemas posteriores que tengan alguna similitud con los ya creados.

BIBLIOGRAFÍA

- BOOCH, Grady. "Object-oriented Analysis and Design with Applications". 2da. edición, Benjamín/Cummings Publishing, 1993.
- CARMONA, Rhadamés. "El Enfoque Orientado a Objetos". Guía para estudiantes de Algoritmos y Programación. UCV. 2004.
- COTO, Ernesto. Notación Algorítmica para estudiantes de Algoritmos y Programación. UCV.
- DEITEL & DEITEL. "Cómo Programar en Java". Pearson Prentice Hall, 2004
- JOYANES AGUILAR, Luis; RODRÍGUEZ BAENA, Luis; FERNÁNDEZ, Matilde. "Texto: Fundamentos de Programación, Algoritmos, Estructuras de Datos y Objetos". McGraw-Hill, 2003
- JOYANES AGUILAR, Luis; RODRÍGUEZ BAENA, Luis; FERNÁNDEZ, Matilde. "Libro de problemas: Fundamentos de Programación, Algoritmos, Estructuras de Datos y Objetos". McGraw-Hill, 2003
- WU, Thomas C. "Introducción a la Programación Orientada a Objetos en Java". McGraw-Hill. 2001