GUÍA DE TRABAJO - PROGRAMACIÓN ORIENTADA A OBJETOS

1. INFORMACIÓN DE LA ASIGNATURA

Asignatura:	Lógica y Representación I			
Código:	2554208		**Tipo de Curso:**	Obligatorio
Plan de Estudios:	5		**Semestre:**	2
Créditos: 3	**TPS:** 6	**TIS:** 3	**TPT:** 64	**TIT:** 32

2. OBJETIVO

Unidad:	Introducción a la Programación Orientada a Objetos (P.O.O.)
Objetivo de la asignatura:	Desarrollar en el estudiante la habilidad para plantear soluciones lógicas a problemas computacionales que deriven en la implementación de programas en un lenguaje de programación
Resultados de aprendizaje abordados:	▪ Analizar un problema identificando correctamente las entradas, el proceso y la salida asociados a su solución ▪ Escribir algoritmos y programas que solucionen problemas computacionales
Contenido temático:	▪ Los paradigmas de programación y el paradigma de programación orientado a objetos ▪ Clases, atributos y métodos ▪ Modificadores de acceso ▪ Tipos de métodos: constructores, accesores, modificadores y analizadores

3. DESARROLLO TEÓRICO

3.1 LOS PARADIGMAS DE PROGRAMACIÓN

Los paradigmas de programación son enfoques que nos permiten solucionar problemas computacionales desde diferentes enfoques. Cada paradigma interpreta al mundo de una manera determinada y sobre esa visión establece un conjunto de reglas y guías para generar la solución de un problema. En nuestra área de desempeño profesional es común que los problemas se solucionen bien sea bajo la luz de un paradigma específico o combinando varios de ellos, todo de acuerdo con las específicas del problema.

Estos son algunos de los paradigmas de programación:

- ***Programación Estructurada***, la cual establece que la solución de un problema se puede definir a partir del uso de tres tipos de estructuras de control: secuencial, de selección y de iteración. Entre los lenguajes de programación que adoptan este paradigma están C, Pascal y Fortran.
- ***Programación Funcional***, la cual se enfoca en solucionar un problema a partir de la definición de funciones matemáticas puras. Este paradigma usa como conceptos clave la recursividad y la inmutabilidad, y su construcción se realiza sobre las bases del Cálculo Lambda, que es un modelo computacional definido por Alonzo Church en 1932. Entre los lenguajes de programación que adoptan este paradigma están LISP, Haskell, Clojure.

- ***Programación Lógica***, la cual usa como elementos clave las reglas de la lógica proposicional y la lógica de predicados. En éste, un programa consiste en una base de hechos (proposiciones y predicados) y reglas para inferir conclusiones a partir de esa base de hechos. Un lenguaje que implementa este paradigma es Prolog.
- ***Programación Reactiva***, la cual se centra en la propagación de cambios, es decir, en cómo los sistemas reaccionan a eventos o cambios en los datos de manera automática. Este paradigma es especialmente útil en aplicaciones asíncronas, sistemas concurrentes, interfaces de usuario y más. Entre los lenguajes que soportan este paradigma están Kotlin, Scala, Dart y Ruby.
- ***Programación Basada en Eventos***, la cual se centra en responder a eventos y acciones que suceden en la ejecución del programa, como clics de mouse o pulsaciones de teclas. Este paradigma utiliza un tipo de funciones llamadas *callbacks* para responder asíncronamente a los eventos. Un lenguaje de programación que implementa este paradigma es JavaScript.

En este curso, nos centramos en solucionar los problemas usando de base el paradigma de programación Orientada a Objetos, el cual se describe a continuación.

3.2 EL PARADIGMA ORIENTADO A OBJETOS

La Programación Orientada a Objetos (POO) es un paradigma de programación en el que la solución de un problema computacional se modela usando "**objetos**", en lugar de funciones y procedimientos como en el paradigma estructurado. Un **objeto** puede ser entendido como una entidad que combina datos y comportamiento. Los datos son representados mediante "atributos" (también llamados propiedades o campos), mientras que el comportamiento es representado mediante "métodos" (también llamados funciones o procedimientos).

En POO, el énfasis está en modelar el software como un conjunto de objetos que interactúan entre sí, similar a cómo interactúan los objetos en el mundo real. Esto facilita la reutilización del código, la modularidad, la escalabilidad, y el mantenimiento del software.

Existen cuatro elementos clave en la programación orientada a objetos:

- Una ***clase*** es un "molde" o "plantilla" que define un tipo particular de objeto. Esta especifica qué atributos y métodos tendrán los objetos que pertenezcan a esa clase. Por Ejemplo, una clase denominada Celular puede tener atributos como marca, modelo, color y cantidad_de_memoria, y métodos como hacer_llamada, tomar_foto y enviar_sms.
- Los ***objetos*** son las instancias concretas de una clase. Estos representan las entidades tangibles o los conceptos abstractos relacionados con un problema particular. Por ejemplo, cuando pensamos la clase Celular, un objeto (o ***instancia***) de esa clase es tu celular, si el celular que tienes en la mano.
- Los ***atributos*** son los datos que, de acuerdo con el problema, se deben almacenar del objeto. Así, los atributos son variables que almacenan la información del objeto. En la clase Celular los atributos son marca, modelo, color y cantidad_de_memoria. Note que los atributos son sustantivos.
- Los ***métodos*** son funciones que definen el comportamiento de los objetos de una clase. Estos permiten manipular los atributos y definen las interacciones del objeto. En el ejemplo, “hacer_llamada”, “tomar_foto” y “enviar_sms” son los métodos. Note que los métodos inician con un verbo conjugado en infinitivo.

Además de esos elementos, la POO está sustentada en cuatro pilares (o principios), los cuales son:

- La ***abstracción***, que es el proceso de identificar las características y el comportamiento esencial de un objeto del mundo del problema y suprimir los detalles irrelevantes. Por ejemplo, al modelar un “Gato”, podríamos abstraer comportamientos comunes a todos los gatos, como “maullar” y “dormir”, independientemente de su raza o color.
- El ***encapsulamiento***, el cual permite ocultar los detalles internos de un objeto y exponer solo lo que es necesario a través de una interfaz pública. Este principio está implementado a partir de la visibilidad de los atributos y métodos.
- El ***polimorfismo***, que es el principio que permite que objetos de diferentes clases puedan responder al mismo mensaje de manera única y específica.
- La ***herencia***, la cual es la propiedad que permite que una clase transfiera sus atributos y métodos a otra clase.

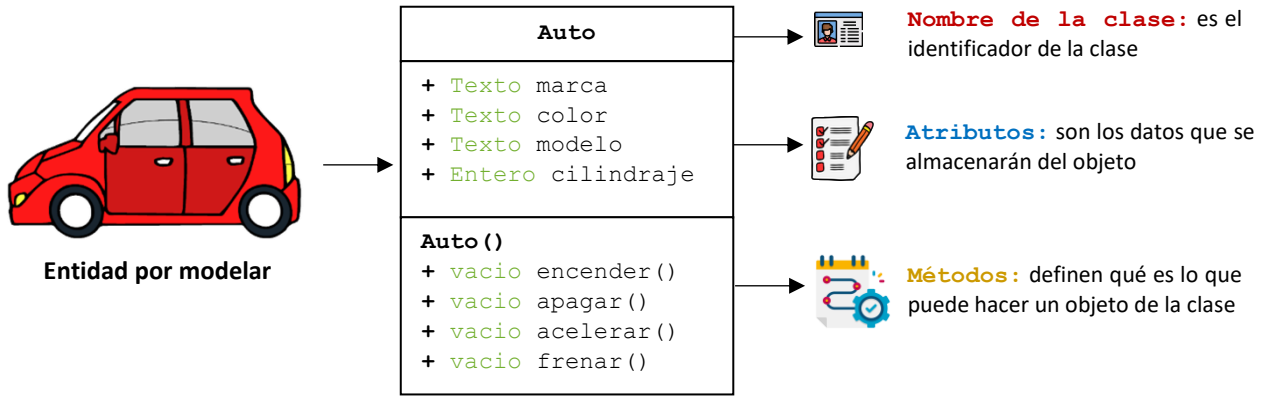
Estos principios otorgan a la POO la capacidad de solucionar problemas de manera modular, favorecen la reutilización del código y ayuda a la gestión eficiente de la complejidad en la que se pueden ver envueltos los proyectos de software.

3.3 Representación Gráfica de una Clase

Como se mencionó, una de las bases de la programación orientada a objetos son las clases. En términos simples, la ***clase*** puede verse como un plano que determina cómo serán los objetos de esa clase, cuáles serán sus atributos y cuáles sus comportamientos. Como se ha mencionado, una clase puede verse como un molde para fabricar objetos.

Imagínese un auto, cualquier tipo de auto. Cuando pensamos en un auto, todos partimos de un modelo mental genérico. En ese modelo todos los autos tienen entre sus características una ***marca***, un ***color***, un ***modelo*** y un ***cilindraje***, por ejemplo. Además, todos los autos tienen el mismo comportamiento: ***encender***, ***apagar***, ***acelerar*** y ***frenar***. Cuando abstraemos la generalidad de lo que representa un **Auto**, estamos diseñando una clase con sus atributos y sus métodos.

Para representar una clase de manera gráfica, usamos el lenguaje UML (*Unified Modeling Language*). En este, la representación de una clase tiene tres partes: el nombre de la clase, los atributos y los métodos. Un ejemplo de la representación gráfica para la clase **"Auto"** es la siguiente:

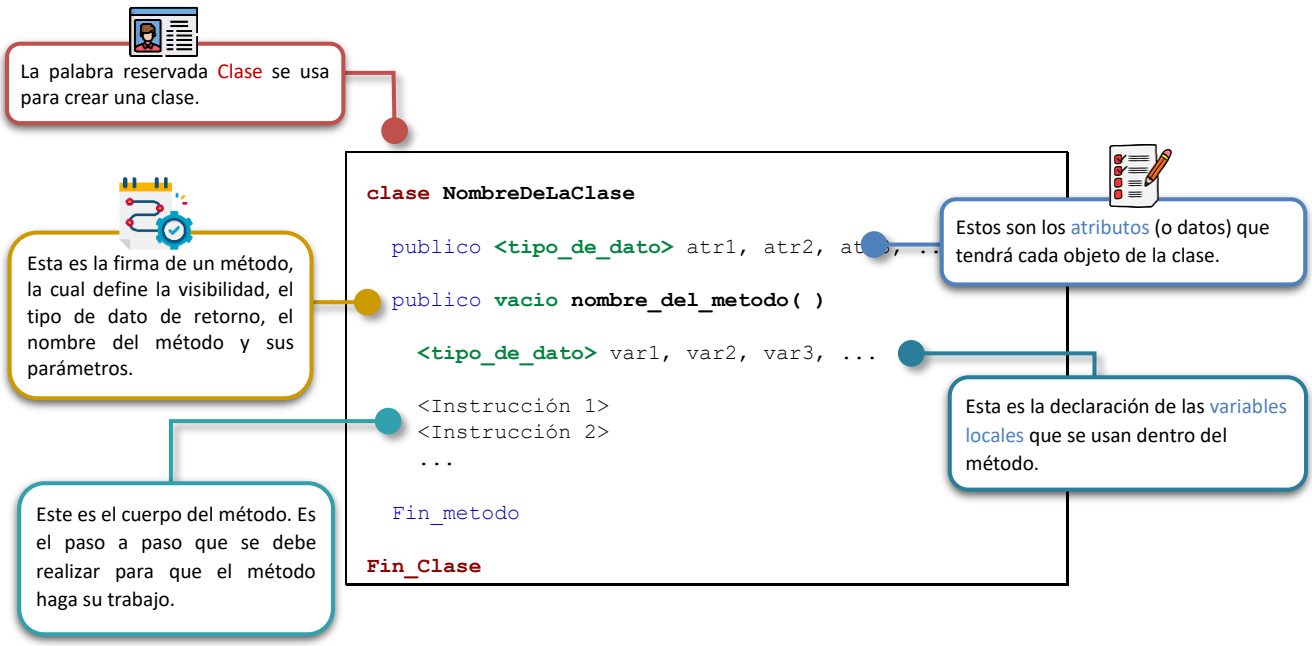


Elementos de la representación de una clase:

- El nombre de una clase debe seguir el estilo **`PascalCase`** en el que cada palabra que compone el nombre de la clase inicia en mayúscula y no se usa ningún separador entre las palabras. Por ejemplo, NombreDeLaClase.
- Cada atributo debe indicar su visibilidad (+ público, - privado o # protegido), después se define su <tipo_de_dato> (Entero, Real, Texto, Lógico) y posteriormente va el nombre del atributo. Aquí usaremos el estilo **`snake_case`** para nombrar los atributos. Por ejemplo, este_es_el_nombre_de_un_atributo.
- La lista de métodos indica la visibilidad del método, el <tipo_de_dato> de retorno del método, el nombre del método y la lista de parámetros del método en paréntesis. El nombre del método debe iniciar con un verbo conjugado en infinitivo y debe seguir la notación **`snake_case`**.

3.4 Estructura General de una Clase en Seudocódigo

Como en este curso está orientado a la programación orientada a objetos, las clases deben ser diseñadas en términos de un algoritmo. En la guía anterior se presentó un modelo general de algoritmo para una clase, ahora veamos cómo cambia ese modelo para una clase.



Siguiendo esta estructura, y con base en la descripción gráfica de la clase, el algoritmo para la clase **"Auto"** es el siguiente. Tenga presente que, en esta clase, la funcionalidad de cada método lo único que hace es mostrar un mensaje con la acción que está realizando el auto.

```
clase Auto

  # Estos son los atributos de la clase
  publico Texto marca
  publico Texto color
  publico Texto modelo
  publico Entero cilindraje

  # Este es el método CONSTRUCTOR
  # El método constructor no tiene ningún tipo de retorno
  Auto(Texto marca, Texto color, Entero modelo, Entero cilindraje)
    self.marca = marca
    self.color = color
    self.modelo = modelo
    self.cilindraje = cilindraje
  Fin_Metodo

  # Este método enciende el auto
  publico vacio encender()
    Escribir("El auto se ha encendido.")
  Fin_Metodo

  # Este método apaga el auto
  publico vacio apagar()
    Escribir("El auto se ha apago.")
  Fin_Metodo

  # Este método acelera el auto
  publico vacio acelerar()
    Escribir("El auto se ha acelerado.")
  Fin_Metodo

  # Este método frena el auto
  publico vacio frenar()
    Escribir("El auto se ha frenado.")
  Fin_Metodo

Fin_Clase
```

Una vez se ha definida la clase podemos instanciar objetos de esa clase. Recuerde que instanciar un objeto no es más que crear un objeto y almacenarlo en una variable. Cada instancia de la clase es independiente y tiene su propio conjunto de atributos, aunque todas las instancias de una clase comparten la misma estructura y los comportamientos definidos por esta.

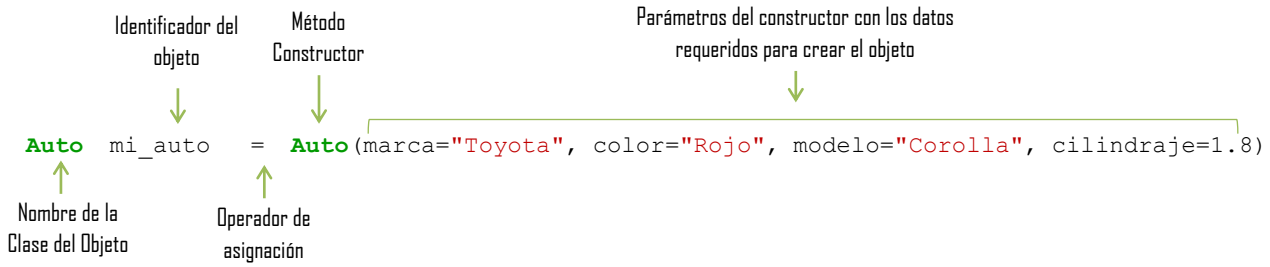
Vamos a ver, creemos un objeto (o instancia) de la clase **Auto**:

```
# Creamos un objeto de la clase
Auto mi_auto = Auto(marca="Toyota", color="Rojo", modelo="Corolla", cilindraje=1.8)
```

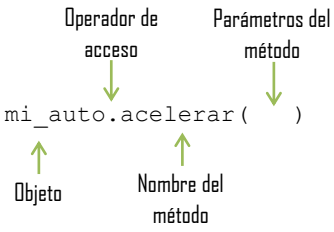
Otra forma de crear ese mismo objeto es como sigue. Note que en este caso no se usan los nombres de los parámetros del método:

```
# Creamos un objeto de la clase
Auto mi_auto = Auto("Toyota", "Rojo", "Corolla", 1.8)
```

En ambos fragmentos del seudocódigo, el objeto que se crea se almacena en la variable `mi_auto`; es decir, `mi_auto` es un objeto (o instancia) de la clase Auto. Tenga en cuenta entonces que cuando vamos a crear un objeto de una clase, al igual que como lo hacemos con una variable de otro tipo, debemos definir el tipo de la variable, que es la clase:



Una vez creado el objeto podemos invocar los métodos de ese objeto usando el operador de acceso, que es un punto (.). Esto lo hacemos así:



Con base en esto, si queremos que el auto arranque, acelere y luego frene, debemos indicárselo con las siguientes instrucciones:

```
mi_auto.arrancar()
mi_auto.acelerar()
mi_auto.frenar()
```

Así, al invocar el método acelerar, se mostrará en pantalla el mensaje `"El auto se ha encendido."` y después el mensaje `"El auto se ha acelerado."`; finalmente, se mostrará el mensaje `"El auto se ha frenado."`.

Importante: cuando se crean varios objetos de la misma clase, estos son independientes. Es decir, los valores en los atributos de cada objeto son independientes de los demás. Los objetos solo comparten la estructura de los atributos y el código que define su comportamiento.

3.5 MODIFICADORES DE ACCESO

En POO los modificadores de acceso permiten controlar la visibilidad y el acceso a los atributos y métodos de los objetos de una clase. En sí, los modificadores de acceso son una herramienta para implementar la propiedad de encapsulamiento. En general, la mayoría de los lenguajes de programación tienen tres modificadores de acceso:

- ***Público (+):*** el cual indica que el atributo o método puede ser usado desde cualquier parte del código, bien sea dentro de la clase o por fuera de esta.
- ***Privado (-)***: el cual indica que el atributo o método solo puede ser usado dentro de la propia clase. Así, un atributo método privado no puede ser usado por fuera de la clase, ni siquiera por las subclases.
- ***Protegido (#)***: un método o atributo marcado como protegido es accesible por la clase y por sus subclases. Sin embargo, éste no puede ser usado desde fuera de la clase o por fuera de su jerarquía de herencia.

En el seudocódigo utilizaremos las palabras `publico`, `privado` y `protegido` para definir el acceso a los métodos y atributos de la clase. A modo de ejemplo, considere la siguiente clase:

```
clase Misterio

  # Estos son los atributos de la clase
  publico Texto atr1
  protegido Texto atr2
  privado Texto atr3

  # Este método constructor inicializa los atributos
  Misterio()
    self.atr1 = "A"
    self.atr2 = "B"
    self.atr3 = "C"
  Fin_Metodo

  # Este método muestra el contenido de atr2
  publico vacio mostrar_atr2()
    # El atributo atr2 es protegido, se puede acceder desde su propia clase
    Escribir(self.atr2)
  Fin_Metodo

  # Este método muestra el contenido de atr3
  publico vacio mostrar_atr3()
    # El atributo atr3 es privado, sólo se puede acceder desde su propia clase
    Escribir(self.atr3)
  Fin_Metodo

Fin_Clase
```

Observe como usamos esta clase.

```
# Creamos un objeto de la clase
Misterio obj = Misterio()

# Modificamos el valor del atributo atr1, el cual es público
obj.atr1 = "AAA"  # VALIDO: atr1 es público, por tanto se puede acceder por fuera de la clase

# Mostramos el valor del atributo atr2, directamente desde fuera de la clase
Escribir(obj.atr2) # ERROR: atr2 es protegido, no se puede acceder por fuera la clase

# Mostramos el valor del atributo atr3, usando el método definido en la clase
Escribir(obj.mostrar_atr3()) # VALIDO: atr3 es privado, pero el método mostrar_atr3 es público
```

Es importante notar que la tercera instrucción es completamente válida porque en esa instrucción no se está usando directamente el atributo atr3, sino que se está usando el método público que está declarado en la misma clase de atr3 y, por tanto, ese método si puede acceder al atributo que es privado.

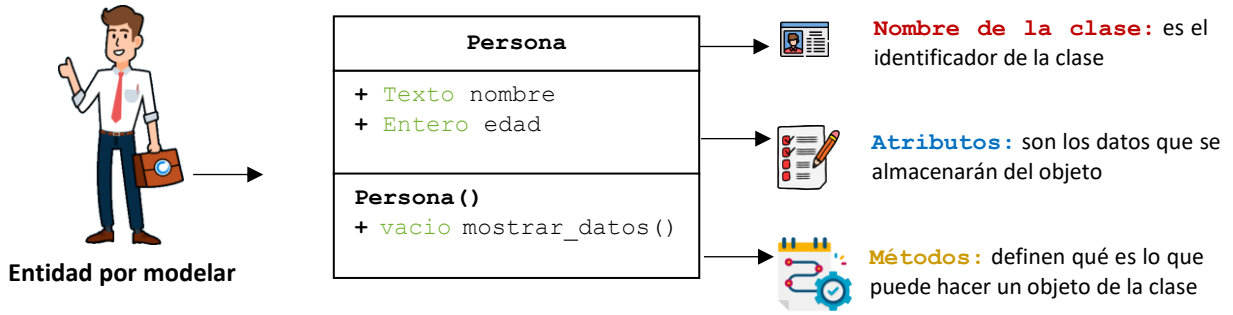
Nota: A medida que vayan ganando conocimiento en cuanto a los conceptos de diseño y modelado, aprenderán a definir la accesibilidad que realmente debe tener un atributo o un método en una clase.

3.6 Los Métodos de una Clase

En programación orientada a objetos los métodos se pueden pensar como funciones que definen el comportamiento de los objetos. Cada instancia (u objeto) de una clase puede invocar y ejecutar los métodos definidos por su clase. Así, los métodos encapsulan la lógica y definen el comportamiento y la forma de comunicarse de los objetos de la clase; además, los métodos proporcionan una forma para organizar y modularizar el código. Existen varios tipos de métodos, los cuales veremos a continuación.

3.6.1 El Método Constructor

Este es un método especial que se llama automáticamente cuando se crea un objeto de la clase. La función principal del método constructor es inicializar los atributos del objeto con el fin de crear un objeto que tenga un estado válido. Por ejemplo, considere un problema en el que se requiere almacenar y mostrar el nombre de una persona y su edad. Para la solución de este problema se puede modelar una clase como la que sigue.



Si observa la clase diseñada, hay dos atributos, uno para almacenar el nombre de la persona y otro para guardar su edad. Entre los métodos, hay un método que nos permitirá mostrar la información de la persona. En este diagrama, el método constructor es un método que tiene el mismo nombre de la clase y que no tiene ningún tipo de retorno. Como se describió, este método busca inicializar los atributos de un objeto cuando este se crea. Es por eso que una posible implementación de esta clase es la siguiente. Preste atención a lo que hace el constructor.

```
clase Persona

  # Estos son los atributos de la clase
  publico Texto nombre
  publico Entero edad

  # Este es el método CONSTRUCTOR
  Persona(Texto nombre, Entero edad)
    self.nombre = nombre
    self.edad = edad
  Fin_Metodo

  # Este método muestra la información de la persona
  publico vacio mostrar_datos()
    Escribir("Datos de la Persona: \n Nombre: ", self.nombre, "\n Edad: ", self.edad)
  Fin_Metodo

Fin_Clase
```

El método constructor indica al programador que en el momento de crear un objeto Persona, el programa ya debe conocer la información de esa persona para crearla. El constructor nos dice que sin el nombre y sin la edad no es posible crear la persona.

Como ejemplo, vamos a crear dos instancias de la clase Persona, una almacenará los datos de María, quien tiene 25 años, y la otra instancia debe almacenar los datos de Jorge, quien tiene 19 años. La porción de código para crear estos objetos es la siguiente:

```
# Creamos al objeto que almacena la información de María
Persona obj1 = Persona(nombre="Maria", edad=25)

# Creamos al objeto que almacena la información de Jorge
Persona obj2 = Persona(nombre="Jorge", edad=19)
```

Este es el llamado al método constructor

Ahora, considere que requerimos, por ejemplo, que los datos de la persona los pida un método de la propia clase denominado `pedir_datos()`. Esto implicaría que para crear una Persona no es necesario contar con los datos, puesto que estos se van a guardar, posterior a la creación del objeto. En ese caso, se debe hacer una modificación a ***la firma del método*** constructor. Analice el cambio que se hace en la clase:

```
clase Persona

  # Estos son los atributos de la clase
  publico Texto nombre
  publico Entero edad

  # ANALICE EL CAMBIO EN ESTE CONSTRUCTOR
  # Aquí los parámetros tienen valores iniciales. Esto permite crear un objeto
  # del que se desconocen sus datos
  Persona(Texto nombre="", Entero edad=0)
    self.nombre = nombre
    self.edad = edad
  Fin_Metodo

  # Este método muestra la información de la persona
  publico vacio mostrar_datos()
    Escribir("Datos de la Persona: \n Nombre: ", self.nombre, "\n Edad: ", self.edad)
  Fin_Metodo

  # Este método pide los datos de la persona al usuario de la app
  publico vacio pedir_datos()
    Escribir("Ingrese el nombre de la persona: ")
    Leer (self.nombre)
    Escribir("Ingrese la edad de la persona: ")
    Leer(self.edad)
  Fin_Metodo

Fin_Clase
```

Vamos nuevamente a crear dos instancias de la clase Persona, una almacenará los datos de Kevin, quien tiene 32 años y la otra instancia debe almacenar los datos de una persona que el usuario ingresará por teclado. Las instrucciones para hacer esto son las siguientes:

```
# Creamos al objeto que almacena la información de Kevin
Persona obj1 = Persona(nombre="Kevin", edad=32)

# Creamos otro objeto cuyos datos no se conocen
Persona obj2 = Persona()

# Pedimos al usuario los datos del objeto sin datos
obj2.pedir_datos()
```

Note que, cuando que cuando se crea `obj2`, como no se conocen el nombre y la edad, estos datos se omiten, y, por tanto, el método constructor le asigna una cadena vacía al nombre y el valor de 0 a la edad. Estos datos posteriormente se modifican cuando se invoca al método `pedir_datos()` de ese objeto.

3.6.2 Métodos Accesores (getters):

Los *getters* son métodos que se usan cuando se requiere dar acceso de lectura a un atributo declarado como privado en una clase. El propósito de estos métodos es proporcionar un acceso controlado a la información almacenada en un atributo privado, sin permitir un acceso directo desde fuera de la clase.

3.6.3 Métodos Modificadores (setters):

Los *setters* se crean para permitir modificar el valor de un atributo privado por fuera de la clase en la que está declaro. Su propósito es proporcionar un mecanismo controlado para actualizar el valor del atributo, permitiendo, por ejemplo, realizar validaciones o acciones adicionales antes de realizar la actualización.

En general, los métodos *getters* y *setters* proporcionan una capa de abstracción y control sobre los atributos de la clase, lo que facilita la gestión y modificación de los datos almacenados en los objetos de la clase. Además, permiten la implementación de lógica específica al obtener o establecer valores, lo que puede ser útil para realizar validaciones o aplicar ciertas reglas de negocio.

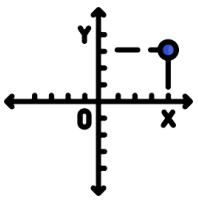
3.6.4 Métodos Analizadores

Los métodos analizadores son los que se usan para implementar las reglas de negocio o de transformación de los datos de un objeto. Es decir, estos métodos son los que implementan la lógica general del comportamiento de los objetos en la clase. Vamos a hacer un ejemplo.

Ejemplo – Clases, Atributos y Métodos

Su profesor de geometría le ha solicitado que desarrolle una aplicación que permita calcular la distancia entre dos puntos del plano cartesiano.

Al analizar el problema identificamos que su solución gira en torno a una entidad llamada **Punto**. Como esta entidad representar un punto del plano cartesiano, entonces los datos a almacenar de entidad son sus coordenadas (x e y) en el plano cartesiano. De acuerdo con el problema, un objeto de esta clase debe tener un comportamiento, que es el de calcular la distancia de sí mismo a otro punto. Esto nos lleva a que debemos diseñar una clase para esta entidad.



Punto
- Real x - Real y
Punto() + Real get_x() + Real get_y() + Real calcular_distancia(Punto p)

Observe que los atributos están definidos como privados y no hay forma de modificarlos, eso quiere decir, que una vez se crea el punto, no será posible modificar sus coordenadas. Este tipo de decisiones del diseño de la clase dependen, en muchos casos, del contexto del problema. Con base en el diseño de esta clase, procedemos a la construcción de su algoritmo en seudocódigo.

```
clase Punto

  # Estos son los atributos de la clase
  privado Real x
  privado Real y

  # Método constructor
  # Por defecto, se asume que las coordenadas x e y del de un punto son (0,0)
  Punto(Real x=0, Real y=0)
    self.x = x
    self.y = y
  Fin_Metodo

  # Retorna la coordenada x
  publico Real get_x()
    return self.x
  Fin_Metodo

  # Retorna la coordenada y
  publico Real get_y()
    return self.y
  Fin_Metodo

  # Este método calcula la distancia del punto a otro, el cual se recibe como parámetro
  publico Real calcular_distancia(Punto p)
    d = 0
    d = ((self.x - p.get_x())^2 + (self.y - p.get_y())^2)^(1/2)
    retornar d
  Fin_Metodo

Fin_Clase
```

Ahora, para usar esta clase debemos crear los objetos correspondientes, por ejemplo:

```
# Creamos el punto p1 con coordenadas (1,5)
Punto p1 = Punto(1, 5)

# Creamos otro punto, p2, con coordenadas (-7, 2)
Punto p2 = Punto(-7, 2)

# Calculamos y mostramos la distancia de p1 a p2
d = p1.calcular_distancia(p2)
Escribir("La distancia entre los puntos es: ",d)
```

En la primera instrucción, cuando se crea el punto `p1`, se invoca el método constructor con los valores `1` y `5`. Note que en este caso no se especifica el nombre del atributo al que se le está asignando cada valor por lo que la asignación se hace en el orden en el que están definidos en el constructor. En este caso también podíamos haber hecho lo que sigue, al igual que con el objeto p2.

```
# Creamos el punto p1 con coordenadas (1,5)
Punto p1 = Punto(x=1, y=5)
```

Esto nos indica que cuando invocamos un método se pueden omitir los identificadores de los parámetros; sin embargo, se debe tener cuenta el orden en el que estos están declarados en el constructor de la clase. En el ejemplo, el constructor está definido así: `Punto(Real x=0, Real y=0)`, eso nos indica que el primer valor en la invocación del método se almacenará en el parámetro `x` y el segundo valor se almacenará en el parámetro `y`, de ahí que usar el constructor como `Punto(1, 5)` sea lo mismo que usarlo como `Punto(x=1, y=5)`.

4. EJERCICIO PROPUESTO

Con el fin de poner en práctica la solución de problemas usando la P.O.O. considere el siguiente ejercicio.

Ejemplo – Algoritmos secuenciales con P.O.O

La tienda de barrio "***abarrotes y más***" le ha pedido que diseñe una aplicación en el permita almacenar la información de los productos que se venden en la tienda. De acuerdo con el dueño, de cada producto se debe almacenar el código, el nombre, la cantidad de disponibles, el valor de compra, el valor de venta sin IVA y el porcentaje que se debe cobrar por el IVA de cada producto. La tienda puede vender productos, lo que significa que debe cuando se venda un producto, la cantidad de disponibles debe disminuir en la cantidad de productos vendidos. Adicionalmente, el producto debe permitir consultar su valor, incluido el IVA.

Desarrolle una aplicación que permita al dueño almacenar la información de 4 de sus productos en el inventario. Además, la aplicación debe mostrar el valor de venta de cada producto y le debe indicar al dueño cuáles serán las ganancias si se vendieran la totalidad de los productos que registrará.

4.1 ANÁLISIS DEL PROBLEMA:

1. Identificamos el cliente y el usuario de la aplicación
   - Cliente: el dueño de la tienda
   - Usuario: el dueño de la tienda

2. Identificamos los requerimientos funcionales:
   - R1: Almacenar la información de los productos que se venden en la tienda
   - R2: Calcular el precio de venta de un producto, incluido el IVA
   - R3: Calcular el valor de ganancias del inventario de la tienda con los productos registrados

3. Identificamos las entidades del mundo del problema:
   - Producto, hace referencia al producto que venden en la tienda y del que se deben almacenar: el código, que es un entero mayor a cero; el nombre, que es una cadena de texto no vacía; la cantidad de disponibles, que es un entero mayor o igual a cero; el valor de compra y de venta sin IVA, ambos reales mayores a cero; el porcentaje de IVA que se cobra, el cuál es un real entre 0 y 100, ambos inclusive.

4. Descripción de cada requerimiento:

Identificador	**R1**
Nombre	**Almacenar la información de un producto**
Proceso	El sistema debe permitir almacenar el código, el nombre, la cantidad de disponibles, el valor de compra, el valor de venta sin IVA y el porcentaje que se debe cobrar por el IVA de producto, para ello: 1. Se muestra un mensaje pidiendo los datos del producto 2. Se almacenan los datos del producto en variables
Entradas	Código del producto Nombre del producto Cantidad de disponibles Valor de compra Valor de venta sin IVA Porcentaje del IVA
Salidas o Resultados	Se muestra en la pantalla un mensaje que indica que el producto se almacenó o un mensaje de error si se encuentra un error en el dominio de los datos

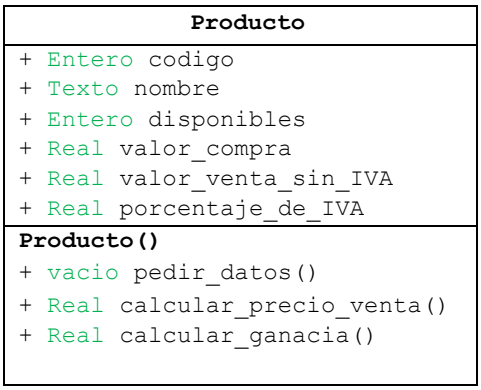
Identificador	**R2**
Nombre	**Calcular el precio de venta de un producto, incluido el IVA**

Proceso	El sistema debe permitir calcular el precio de venta de un producto. Para ello, 1. Se asume que se cuenta tanto con el valor de venta sin IVA y el porcentaje que se cobra de IVA 2. Se calcula el precio de venta como: precio_venta = valor_venta_sin_IVA * (1 + porcentaje_IVA/100)
Entradas	No se requieren entradas del usuario puesto que los datos necesarios los tiene el objeto
Salidas o Resultados	El precio de venta del producto, con el IVA incluido.

Identificador	**R3**
Nombre	**Calcular el valor de ganancias del inventario**
Proceso	El sistema debe permitir calcular el total de ganancias de los productos p_i en el inventario. 1. Se parte que se tienen almacenados n productos 2. El valor de la ganancia del inventario se calcula como: ganancia = $\sum_{i=1}^{n} ganacia\_de\_p_i * disponibles\_p_i$, donde, $ganacia\_de\_p_i = valor\_venta\_sin\_IVA\_p_i - valor\_compra\_p_i$
Entradas	No se requieren entradas del usuario puesto que los datos necesarios los tiene el objeto
Salidas o Resultados	El valor de las ganancias de los productos registrados.

4.2 Entidades del Mundo del Problema

En este problema sencillo, como vimos podemos usar una sola clase denominada `Producto`.



4.3 Seudocódigo de la Clase

El paso siguiente en el diseño es el desarrollo de los algoritmos de la clase. En nuestro caso el algoritmo es sencillo y se presenta a continuación:

```
clase Producto

  # Atributos de la clase
  publico Entero codigo
  publico Texto nombre
  publico Entero disponibles
  publico Real valor_compra
  publico Real valor_venta_sin_IVA
  publico Real porcentaje_de_IVA

  # Este es el constructor de la clase
  Producto(Entero codigo=0, Texto nombre="N.A.", Entero disponibles=0, Real valor_compra=0
           Real valor_venta_sin_IVA = 0, Real porcentaje_de_IVA=0)

    self.codigo = codigo
    self.nombre = nombre
    self.disponibles = disponibles
    self.valor_compra = valor_compra
    self.valor_venta_sin_IVA = valor_venta_sin_IVA
```

```
    self.porcentaje_de_IVA = porcentaje_de_IVA
  Fin_metodo

  # Este método pide los datos del producto al usuario
  publico vacio pedir_datos()
    Escribir("Ingrese el código del producto: ")
    Leer(self.codigo)
    Escribir("Ingrese el nombre del producto: ")
    Leer(self.nombre)
    Escribir("Ingrese la cantidad de ítems disponibles del producto: ")
    Leer(self.disponibles)
    Escribir("Ingrese el valor de compra del producto: ")
    Leer(self.valor_compra)
    Escribir("Ingrese el valor de venta, sin incluir el IVA: ")
    Leer(self.valor_venta_sin_IVA)
    Escribir("Ingrese el porcentaje, entre 1 y 100, que se cobra de IVA al producto: ")
    Leer(self.porcentaje_de_IVA)
  Fin_metodo

  # Este método calcula el precio de venta del producto
  publico Real calcular_precio_venta()
    precio_venta = self.valor_venta_sin_IVA * (1 + self.porcentaje_IVA/100)
    retornar precio_venta
  Fin_metodo

  # Este método calcula la gancia que deja el producto
  publico Real calcular_ganacia()
    ganacia = (self.valor_venta_sin_IVA - self.valor_compra)*disponibles
    retornar ganancia
  Fin_metodo

Fin_clase
```

Note que de momento no está resuelto el problema planteado porque esta clase por sí sola no da solución a los tres requerimientos planteados. Para solucionar el problema debemos crear la lógica de solución usando el diseño de la clase anterior.

Iniciamos con el primer requerimiento, el cual debe permitir almacenar la información de los productos de la tienda. En este caso, siguiendo las indicaciones del enunciado asumimos que son cuatro productos, de los cuales vamos a crear tres manualmente y solo se le pedirá la información al usuario del cuarto producto. Después de creados los productos, calculamos y mostramos el precio de cada producto, esto para dar respuesta al segundo requerimiento. Finalmente, calculamos la ganancia que nos dejarán los productos de la tienda, es decir la suma de las ganancias de los cuatro productos creados.

```
# Requerimiento R1: creamos los productos
Producto p1 = Producto(1, "Arroz", 50, 1500, 1800, 5)
Producto p2 = Producto(1, "Lentejas", 10, 1950, 2500, 19)
Producto p3 = Producto(1, "Huevos", 100, 300, 400, 0)
Producto p4 = Producto()

# Pedimos los datos al usuario del producto p4
p4.pedir_datos()

# Requerimiento R2: mostrar el precio de venta de los productos
Escribir("Precio de venta de los productos en la tienda: ")
Escribir("  - ", p1.nombre ," $ ", p1.calcular_precio_venta())
Escribir("  - ", p2.nombre ," $ ", p2.calcular_precio_venta())
Escribir("  - ", p3.nombre ," $ ", p3.calcular_precio_venta())
Escribir("  - ", p4.nombre ," $ ", p4.calcular_precio_venta())

# Requerimiento R3: calcular el valor de ganancia del inventario
ganancia_total = 0
ganancia_total = p1.calcular_ganacia() + p2.calcular_ganacia() + p3.calcular_ganacia()
                 + p4.calcular_ganacia()
Escribir("La ganancia que dejarán los productos del inventario es de  $", ganancia_total)
```

5. BIBLIOGRAFÍA

- Herrera, A., Ebratt, R., & Capacho, J. (2016). Diseño y construcción de algoritmos. Barranquilla: Universidad del Norte.
- Aguilar, L. (2020). Fundamentos de programación: algoritmos, estructuras de datos y objetos (5a.ed.). México: Mc Graw Hill.
- Thomas Mailund. Introduction to Computational Thinking: Problem Solving, Algorithms, Data Structures, and More, Apress, 2021.
- Steven S. Skiena. The Algorithm Design Manual. Third Edition, Springer, 2020.
- Samuel Tomi Klein. Basic Concepts in Algorithms, World Scientific Publishing, 2021.

Elaborado Por:	Carlos Andrés Mera Banguero
Versión:	2.0
Fecha	Agosto de 2024