



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA





UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

ASIGNATURA

PROGRAMACIÓN ORIENTADA A

OBJETOS



Transformamos el mundo desde la Amazonía

Ing. Edwin Gustavo Fernández Sánchez, Mgs.

DOCENTE - PERSONAL ACADÉMICO NO TITULAR OCASIONAL

DIRECTOR DE GESTIÓN DE TECNOLOGÍAS INFORMACIÓN Y COMUNICACIÓN

UNIVERSIDAD ESTATAL AMAZÓNICA





UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

SEMANA

6

DESARROLLO DE LA SEMANA 6: DEL LUN. 13 AL DOM. 19 DE ENERO/2025

Resultado de aprendizaje: Aplicar programas que hacen uso de relación de herencia, interfaces y clases abstractas con métodos y variables polimórficas

CONTENIDOS

UNIDAD II : Objetos, clases, Herencia, Polimorfismo.

- Tema 1: Elementos de programación
 - Subtema 1.2: Definición de Clase
 - Subtema 1.3: Definición de Objeto
 - Subtema 1.4 : Herencia, Encapsulación, Polimorfismo



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

Unidad 2

Objetos, clases, Herencia, Polimorfismo.

Tema 1.

Elementos de programación



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

Subtema 1.2: Definición de Clase

Subtema 1.3: Definición de Objeto

Subtema 1.4: Herencia, Encapsulación, Polimorfismo

**PROGRAMACIÓN ORIENTADA A
OBJETOS
(UEA-L-UFB-030)**



Introducción a la POO

La Programación Orientada a Objetos (POO) es un **paradigma** de programación que utiliza '**objetos**' y sus interacciones para diseñar aplicaciones y programas de software.

En POO, cada **objeto** puede representar una entidad del mundo real con características y comportamientos, lo que facilita la creación de programas más intuitivos y cercanos a cómo percibimos el mundo.



Componentes Clave de la POO

- **Clases:** Son los moldes o plantillas para crear objetos. Definen las características y comportamientos que tendrán los objetos creados a partir de ellas.
- **Objetos:** Son instancias de clases. Cada objeto tiene un estado definido por sus atributos y puede realizar acciones a través de sus métodos.

En POO, las clases y objetos son los pilares fundamentales. Nos permiten **abstraer** y encapsular la información y el comportamiento de las entidades que deseamos representar en nuestros programas.



Principios de la POO

Además de las clases y objetos, la POO se basa en tres **principios** clave: Herencia, Encapsulación y Polimorfismo.

Herencia

Permite que una clase herede características de otra, facilitando la reutilización de código.



Encapsulación

Oculto los detalles internos del funcionamiento de una clase y expone solo lo que es necesario para el exterior.



Polimorfismo

Permite que los objetos se comporten de manera diferente bajo el mismo interfaz.



Importancia de la POO

La POO es fundamental en el desarrollo de software moderno. Ofrece un enfoque modular, permitiendo la creación de software más organizado, reutilizable y fácil de mantener.

Este paradigma es ampliamente utilizado en muchas áreas de desarrollo, desde aplicaciones web hasta juegos y sistemas embebidos.

Entender la POO nos abre las puertas a un mundo de posibilidades en la programación. Nos ayuda a pensar en términos de objetos del mundo real, lo que facilita el diseño y la implementación de soluciones de software complejas.



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

Concepto Fundamental de Clase

En la Programación Orientada a Objetos, una '**Clase**' es una plantilla o modelo para crear objetos. Define las características y comportamientos comunes que compartirán los objetos creados a partir de ella.

Piensa en una clase como un 'molde' que dicta la estructura y capacidades de los objetos que se crearán.

Atributos y Métodos

Los '**Atributos**' son las características o propiedades de la clase.

Por ejemplo, en una clase Coche, los **atributos** pueden incluir marca, modelo y color.

Los '**Métodos**' son las acciones o funciones que puede realizar un objeto de esa clase.

En nuestra clase Coche, un método podría ser **arrancar()** o **detener()**."



Ejemplo de Clase en Python

Aquí hay un ejemplo simple de cómo definir una clase en Python

```
python Copy code

class Coche:
    def __init__(self, marca, modelo, color):
        self.marca = marca
        self.modelo = modelo
        self.color = color

    def arrancar(self):
        print(f"El {self.modelo} está arrancando.")
```

En este ejemplo, Coche es la clase con atributos marca, modelo, y color, y un método arrancar().

Creación de Objetos: A partir de esta clase, podemos crear múltiples objetos, cada uno representando un coche específico con sus propias características.



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

Fundamentos de la Estructura de Clase

Una clase en Python se define mediante la palabra clave class, seguida del nombre de la clase y dos puntos (:).

Dentro de la clase, definimos atributos y métodos. Los 'Atributos' son variables que pertenecen a la clase y representan las características de los objetos. Los 'Métodos' son funciones dentro de la clase que describen las acciones que los objetos pueden realizar.

Constructor `__init__`:

Casi todas las clases tienen un método especial llamado constructor, `__init__`, que se llama automáticamente cuando se crea un nuevo objeto de la clase.

El constructor `__init__` se utiliza para inicializar los atributos del objeto.


Por ejemplo, si tienes una clase Libro, podrías usar `__init__` para establecer el título, el autor y el número de páginas del libro.



Ejemplo de Clase en Python

Aquí hay un ejemplo simple de cómo definir una clase en Python

python

 Copy code

```
class Libro:
    def __init__(self, titulo, autor, paginas):
        self.titulo = titulo
        self.autor = autor
        self.paginas = paginas

    def informacion(self):
        return f"{self.titulo} escrito por {self.autor}, {self.paginas}"
```

En este ejemplo, la clase Libro tiene tres **atributos**: titulo, autor y paginas. Además, tiene un **método** informacion que devuelve una descripción del libro.

Importancia de la Estructura de Clase: La estructura de una clase nos ayuda a organizar y encapsular datos y comportamientos relacionados, haciendo nuestro código más modular y fácil de entender



Objetos: Instancias de Clases

En la Programación Orientada a Objetos, un '**Objeto**' es una instancia concreta de una **clase**. Mientras que una clase es la definición, el objeto es una realización específica de esa definición.

Cada objeto tiene su propio conjunto de atributos (datos) y métodos (funciones) que hereda de su clase, pero con valores únicos que los diferencian entre sí.



Proceso de Creación de un Objeto

Crear un objeto en Python implica llamar al nombre de la clase como si fuera una función. Este proceso se conoce como 'instanciación'.

```
python Copy code  
  
# Suponiendo que tenemos una clase 'Libro' definida previamente  
mi_libro = Libro("Cien Años de Soledad", "Gabriel García Márquez", 417)
```

Durante la instanciación, se invoca automáticamente el método constructor `__init__` de la clase, permitiendo inicializar los atributos del objeto.



Introducción a la Herencia

La herencia es un principio clave en la Programación Orientada a Objetos que permite a una clase (llamada clase hija) heredar atributos y métodos de otra clase (llamada clase padre o superclase).

Este mecanismo facilita la reutilización de código y la creación de jerarquías de clase, lo que contribuye a un diseño de software más eficiente y organizado



Cómo Funciona la Herencia

En la herencia, las clases hijas adquieren todas las características de la clase padre, pero también pueden tener características adicionales o modificar las heredadas.

Por ejemplo, si tenemos una clase general Vehículo con atributos como marca y modelo, una clase hija Coche podría heredar estos atributos y añadir otros específicos, como **numeroDePuertas**.

python

Copy code

```
class Vehiculo:
    def __init__(self, marca, modelo):
        self.marca = marca
        self.modelo = modelo

class Coche(Vehiculo): # Herencia de la clase Vehiculo
    def __init__(self, marca, modelo, numeroDePuertas):
        super().__init__(marca, modelo) # Llamada al constructor de la
        self.numeroDePuertas = numeroDePuertas

    # Método específico de la clase Coche
    def mostrar_informacion(self):
        return f"Coche {self.marca} {self.modelo}, con {self.numeroDePuertas} puertas"

# Creación de un objeto de la clase Coche
mi_coche = Coche("Toyota", "Corolla", 4)
print(mi_coche.mostrar_informacion())
```



Ventajas de la Herencia

La herencia permite crear **clases más especializadas** sin tener que reescribir el código de las clases de nivel superior, lo que reduce la redundancia y mejora la mantenibilidad.

Facilita la modificación y extensión de las funcionalidades existentes, ya que **los cambios en la clase padre pueden afectar automáticamente a las clases hijas**.

La herencia también permite el **polimorfismo**, donde las clases hijas pueden tener métodos con el mismo nombre que los de la clase padre, pero con implementaciones diferentes.



Encapsulación: Protegiendo los Datos

La **encapsulación** es un principio fundamental de la Programación Orientada a Objetos que se refiere a la práctica de ocultar los detalles internos de la implementación de una clase y exponer solo los componentes necesarios al exterior.

Este concepto ayuda a prevenir el acceso directo a los datos internos de un objeto, proporcionando una manera de **proteger** su estado y comportamiento de interferencias externas o mal uso.




Implementación de Encapsulación en Python

En Python, la encapsulación se implementa mediante el uso de **métodos y atributos privados o protegidos**. Por ejemplo, los atributos privados se pueden declarar con dos guiones bajos al inicio (`__atributo`).

Los métodos 'getter' y 'setter' se utilizan para acceder y modificar estos atributos privados, permitiendo un control más estricto sobre cómo y cuándo se modifican los datos del objeto.

python

 Copy code

```
class CuentaBancaria:
    def __init__(self, saldo_inicial):
        self.__saldo = saldo_inicial # Atributo privado

    def depositar(self, cantidad):
        if cantidad > 0:
            self.__saldo += cantidad

    def retirar(self, cantidad):
        if cantidad <= self.__saldo:
            self.__saldo -= cantidad

    def obtener_saldo(self): # Método 'getter' para el saldo
        return self.__saldo
```



Ejemplo de Encapsulación

En este ejemplo, la clase CuentaBancaria encapsula el `__saldo`. Los métodos depositar y retirar controlan cómo se modifica el saldo, mientras que obtener_saldo permite acceder al saldo.

Beneficios de la Encapsulación: La encapsulación mejora la seguridad y robustez del código, ya que previene cambios no autorizados o inesperados en el estado del objeto. Facilita la mantenibilidad y escalabilidad del software, ya que los cambios internos en una clase no afectan a otras partes del programa que la utilizan.



Definición de Polimorfismo

El polimorfismo, uno de los conceptos clave de la Programación Orientada a Objetos, se refiere a la capacidad de un método para realizar diferentes acciones en función del objeto que lo invoca.

Este principio permite que **objetos** de diferentes clases sean tratados como instancias de una misma clase, utilizando una interfaz común pero con comportamientos distintos.



Tipos de Polimorfismo

Existen dos tipos principales:

- **Polimorfismo de sobreescritura**, donde los métodos de la clase hija tienen el mismo nombre que los métodos de la clase padre pero comportamientos diferentes.
- **Polimorfismo de sobrecarga**, que permite tener varios métodos con el mismo nombre pero diferentes parámetros.



UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA

Ejemplo de polimorfismo en Python (Sobreescritura)


En este ejemplo, crearemos una clase base **DispositivoElectronico** y dos clases derivadas **Telefono** y **Computadora**, donde cada una sobrescribe un método de la clase base.

La clase **DispositivoElectronico** tiene un método **encender**, que proporciona un mensaje genérico de encendido.

Las clases Telefono y Computadora heredan de DispositivoElectronico y sobrescriben el método encender para proporcionar mensajes específicos para cada tipo de dispositivo.

Al crear instancias de Telefono y Computadora y llamar al método encender, se ejecutan las versiones sobrescritas de este método, demostrando polimorfismo.

python

 Copy code

```
class DispositivoElectronico:
    def encender(self):
        return "Dispositivo electrónico encendido"

class Telefono(DispositivoElectronico):
    def encender(self):
        return "Teléfono iniciado y listo para usar"

class Computadora(DispositivoElectronico):
    def encender(self):
        return "Computadora arrancando, bienvenido"

# Creando objetos de las clases derivadas
mi_telefono = Telefono()
mi_computadora = Computadora()

# Utilizando el método sobrescrito
print(mi_telefono.encender())
print(mi_computadora.encender())
```



Ejemplo de Polimorfismo en Python (Sobrecarga)

En Python, el polimorfismo de sobrecarga en el sentido tradicional (como se ve en lenguajes como Java o C++) **no se aplica** de la misma manera, ya que Python no soporta múltiples métodos con el mismo nombre pero con diferentes parámetros en la misma clase.

Sin embargo, se puede simular un comportamiento similar utilizando argumentos opcionales o múltiples en los métodos. Aquí te muestro cómo podríamos adaptar el concepto a nuestro ejemplo de `DispositivoElectronico`

python

Copy code

```
class DispositivoElectronico:
    def encender(self, modo='normal'):
        if modo == 'normal':
            return "Dispositivo electrónico encendido en modo normal"
        elif modo == 'eco':
            return "Dispositivo electrónico encendido en modo ecológico"
        else:
            return "Modo no reconocido, dispositivo electrónico encendi

# Creando un objeto de la clase DispositivoElectronico
mi_dispositivo = DispositivoElectronico()

# Utilizando el método con diferentes argumentos
print(mi_dispositivo.encender()) # Sin especificar modo
print(mi_dispositivo.encender('eco')) # Modo ecológico
print(mi_dispositivo.encender('rápido')) # Modo no reconocido
```




UEA
UNIVERSIDAD
ESTATAL AMAZÓNICA