UNIDAD VI: PROGRAMACIÓN ORIENTADA A OBJETOS.

Programación Avanzada [IM401]– Ingeniería Mecatrónica

Ing. Linder, German, Ing. Zarratea Diego, Ing. Kelm Marcelo

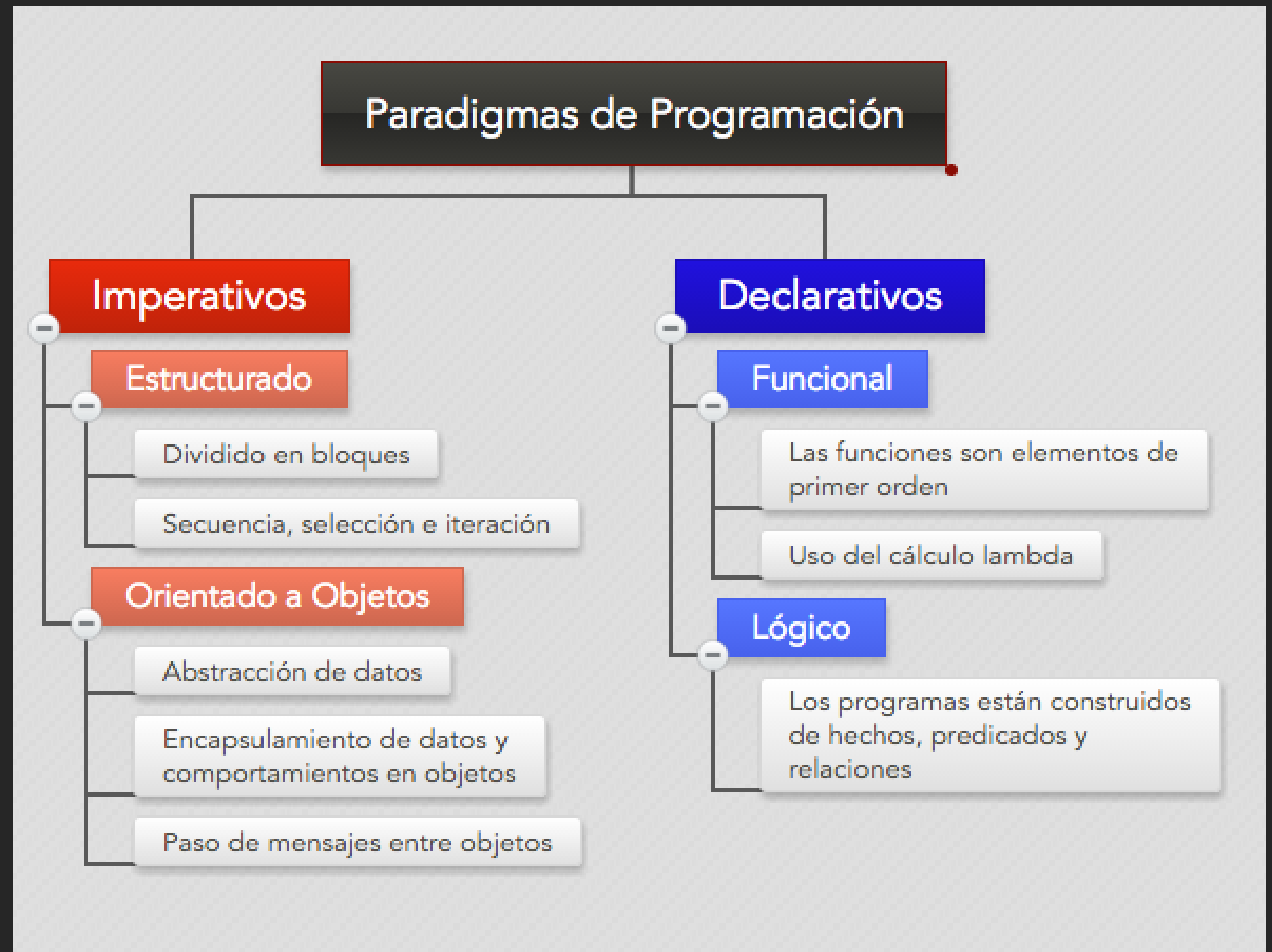
2025

PARADIGMA DE PROGRAMACIÓN

Un paradigma define un modelo o patrón con reglas y pautas que todo desarrollador debe seguir para resolver una situación problemática.

Implica prohibir o limitar ciertas acciones, definir un marco de referencia común.

No todos los lenguajes te permiten programar con cualquier paradigma.



PARADIGMAS IMPERATIVOS (O PROCEDURALES)

El programa indica cómo se debe hacer algo (paso a paso).

Paradigma	Descripción breve
Programación estructurada	Usa estructuras de control (if, for, while) para organizar el código.
Programación modular	Divide el código en módulos o funciones reutilizables.
Programación orientada a objetos (POO)	Modela el sistema con objetos que tienen atributos (estado) y métodos (comportamiento).
Programación concurrente / paralela	Permite ejecutar varias tareas o hilos al mismo tiempo.

Lenguajes típicos: C, C++, Java, Python, Rust, Go, Ada.

PARADIGMAS DECLARATIVOS

El programa indica qué se quiere lograr, sin especificar cómo hacerlo.

Paradigma	Descripción breve
Programación funcional	Usa funciones puras, evita el estado global y efectos secundarios.
Programación lógica	Usa reglas y hechos para deducir información (resolución automática).
Programación reactiva	Basado en flujos de datos y propagación de cambios (ideal para sistemas en tiempo real).
Programación basada en restricciones	Se definen restricciones que deben cumplirse y el sistema encuentra las soluciones.

Lenguajes típicos: Haskell, Prolog, SQL, Scala, Elixir, OCaml.

PARADIGMAS DE PROGRAMACIÓN

Tipo de Programación	Descripción breve	Ejemplo típico	Aplicaciones comunes
Estructurada	Organiza el código en bloques secuenciales, condicionales y repetitivos.	C, Pascal	Programas embebidos, controladores simples
Modular	Divide el programa en módulos o funciones independientes.	C con archivos .h y .c	Sistemas embebidos, firmware organizado
Orientada a Objetos	Modela el sistema como objetos que tienen atributos y comportamientos.	C++, Python, Java	Sistemas complejos, simuladores, RTOS, automatización
Funcional	Usa funciones puras y evita estados mutables.	Haskell, Lisp, Python	Inteligencia artificial, procesamiento de datos
Orientada a Eventos	El flujo del programa se basa en eventos (botones, interrupciones, señales).	JavaScript, C# (GUI), Arduino	Interfaces gráficas, RTOS, sistemas interactivos
Concurrente / Paralela	Permite ejecutar múltiples tareas al mismo tiempo.	FreeRTOS (C/C++), Python threads	Sistemas embebidos, control en tiempo real
Reactiva	Reacciona a flujos de datos en tiempo real, ideal para sistemas de control.	RxJS, LabVIEW, Stateflow	Sistemas de control, automatización industrial
Declarativa	Describe qué hacer, no cómo.	SQL, Prolog	Bases de datos, lógica de control

COMPARATIVA

Estructurado (C)

```c
#include <stdio.h>

int main() {
    int temperatura = 35;
    if (temperatura > 30) {
        printf("Motor encendido\n");
    }
    return 0;
}
```

Orientado a Objetos (Python)

```python
class Motor:
    def encender(self):
        print("Motor encendido")

motor = Motor()
temperatura = 35

if temperatura > 30:
    motor.encender()
```

Modular (C con funciones)

```c
#include <stdio.h>

void encender_motor() {
    printf("Motor encendido\n");
}

int main() {
    int temperatura = 35;
    if (temperatura > 30) {
        encender_motor();
    }
    return 0;
}
```

Programación por Eventos

```c
// Pseudocódigo en C estilo embebido
ISR(TEMP_SENSOR_INTERRUPT) {
    if (read_temp() > 30) {
        motor_on();
    }
}
```

Orientado a Objetos (C++)

```cpp
#include <iostream>
using namespace std;

// Definición de la clase Motor
class Motor {
public:
    void encender() {
        cout << "Motor encendido" << endl;
    }
};
int main() {
    Motor motor;                // Crear objeto motor
    int temperatura = 35;       // Temperatura simulada

    if (temperatura > 30) {
        motor.encender();       // Llamada al método del objeto
    }

    return 0;
}
```

COMPARATIVA

Programación por Eventos (C++)

```cpp
#include <iostream>
#include <functional>
using namespace std;

// Definimos el manejador de eventos (callback)
void alDetectarTemperaturaAlta() {
    cout << "Evento: Temperatura alta -> Motor encendido" << endl;
// Función que simula la lectura de temperatura y lanza eventos
void sensorTemperatura(int temperatura, function<void()> callback) {
    if (temperatura > 30) {
        // Se dispara el evento
        callback();
    } else {
        cout << "Temperatura normal -> Motor apagado" << endl;
    }
}
int main() {
    int temperatura = 35;
    sensorTemperatura(temperatura, alDetectarTemperaturaAlta);
    return 0;
}
```

Orientado a Objetos (C++)

```cpp
#include <iostream>
using namespace std;

// Definición de la clase Motor
class Motor {
public:
    void encender() {
        cout << "Motor encendido" << endl;
    }
};
int main() {
    Motor motor;              // Crear objeto motor
    int temperatura = 35;     // Temperatura simulada

    if (temperatura > 30) {
        motor.encender();     // Llamada al método del objeto
    }

    return 0;
}
```

COMPARATIVA

Concurrente (FreeRTOS - C)

```c
void tareaMotor(void *pvParameters) {
    while (1) {
        if (temperatura > 30) {
            encender_motor();
        }
        vTaskDelay(pdMS_TO_TICKS(1000));
    }
}
int main() {
    xTaskCreate(tareaMotor, "Motor", 1000, NULL, 1, NULL);
    vTaskStartScheduler();
}
```

Lógica (Prolog)

```prolog
encender_motor :- temperatura(T), T > 30, write('Motor encendido').

temperatura(35).
```

Consulta: ?- encender_motor.

Reactiva (RxJS - JavaScript)

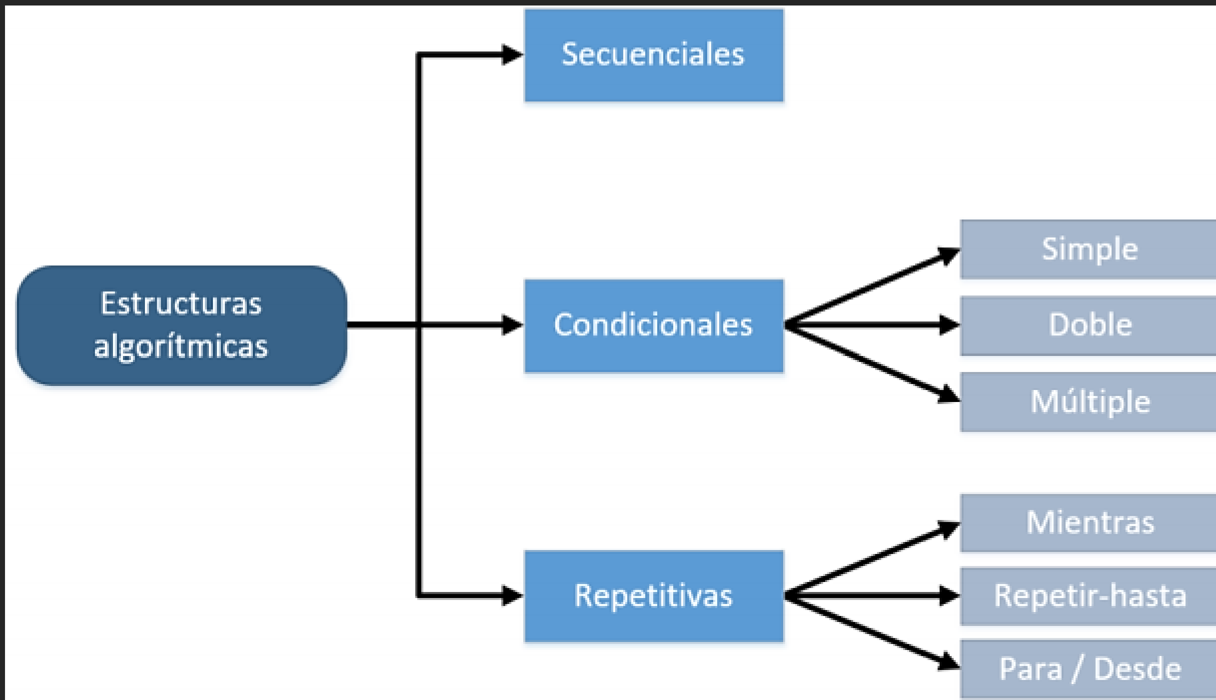
```javascript
import { fromEvent, map, filter } from 'rxjs';

const temperaturas = [25, 28, 32, 29, 35];

fromEvent(document, 'click')
  .pipe(
    map(() => temperaturas[Math.floor(Math.random() * temperaturas.length)]),
    filter(temp => temp > 30)
  )
  .subscribe(temp => console.log(`Motor encendido a ${temp} °C`));
```

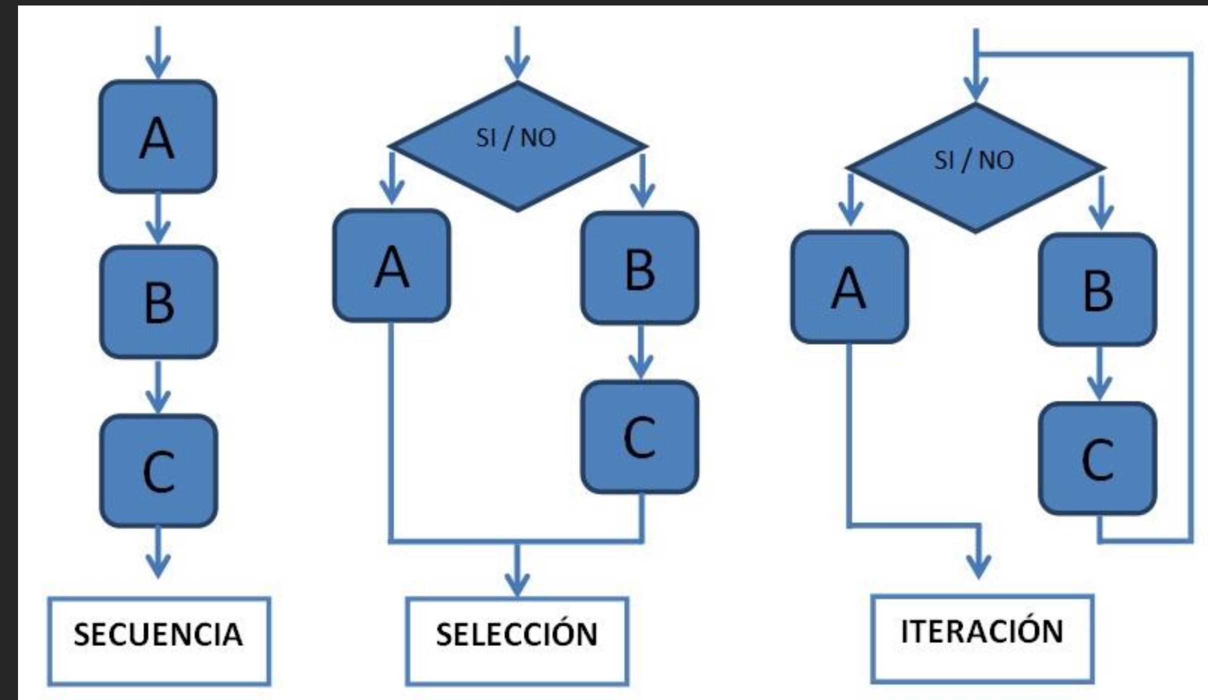
Funcional (Python)

```python
encender = lambda t: "Motor encendido" if t > 30 else "Motor apagado"
print(encender(35))
```

*Una **función lambda** es una **función anónima** que se define en una sola línea y se utiliza para expresar operaciones simples de forma concisa.

PROGRAMACIÓN ESTRUCTURADA(REPASO)





Estructura Secuencial C/C++

```c
int a = 5;
int b = 3;
int suma = a + b;
printf("La suma es: %d\n", suma);
```

Estructura Secuencial Python

```python
a = 5
b = 3
suma = a + b
print("La suma es:", suma)
```

PROGRAMACIÓN ESTRUCTURADA(REPASO)

Estructuras Condicionales

Simple en C/C++

```c
int temp = 32;
if (temp > 30) {
    printf("Hace calor\n");
}
```

Doble en C/C++

```c
int temp = 28;
if (temp > 30) {
    printf("Hace calor\n");
} else {
    printf("Hace fresco\n");
}
```

Múltiple en C/C++

```c
int opcion = 2;
switch(opcion) {
    case 1:
        printf("Elegiste uno\n");
        break;
    case 2:
        printf("Elegiste dos\n");
        break;
    default:
        printf("Opción no válida\n");
}
```

Simple en Python

```python
temp = 32
if temp > 30:
    print("Hace calor")
```

Doble en Python

```python
temp = 28
if temp > 30:
    print("Hace calor")
else:
    print("Hace fresco")
```

Múltiple en Python

```python
opcion = 2
if opcion == 1:
    print("Elegiste uno")
elif opcion == 2:
    print("Elegiste dos")
else:
    print("Opción no válida")
```

PROGRAMACIÓN ESTRUCTURADA(REPASO)

Estructuras Repetitivas

Para / Desde en C/C++

```c
for (int i = 0; i < 5; i++) {
    printf("i = %d\n", i);
}
```

Mientras en C/C++

```c
int i = 0;
while (i < 5) {
    printf("i = %d\n", i);
    i++;
}
```

Repetir-hasta en C/C++

```c
int i = 0;
do {
    printf("i = %d\n", i);
    i++;
} while (i < 5);
```

Para / Desde en Python

```python
for i in range(5):
    print("i =", i)
```

Simple en Python

```python
i = 0
while i < 5:
    print("i =", i)
    i += 1
```

Repetir-hasta
no existe en Python

```python
i = 0
while True:
    print("i =", i)
    i += 1
    if i >= 5:
        break
```

PROGRAMACIÓN ESTRUCTURADA(REPASO)

Funciones

Función en C/C++

```
tipo nombre_funcion(parámetros) {
    // Cuerpo de la función
    return valor;
}
```

Ejemplo

```
#include <stdio.h>

// Definición de función
int sumar(int a, int b) {
    return a + b;
}

int main() {
    int resultado = sumar(5, 3);
    printf("La suma es: %d\n", resultado);
    return 0;
}
```

Función en Python

```
def nombre_funcion(parámetros):
    # Cuerpo de la función
    return valor
```

Ejemplo

```
# Definición de función
def sumar(a, b):
    return a + b

# Llamado a la función
resultado = sumar(5, 3)
print("La suma es:", resultado)
```

PROGRAMACIÓN ESTRUCTURADA(REPASO)

Estracturas de datos Python

Listas

Las listas son colecciones ordenadas y modificables.

```
1  mi_lista = [1, 2, 3, 4, 5]
2  print(mi_lista[0])  # Acceder al primer elemento
3  mi_lista.append(6)  # Añadir un elemento
4  mi_lista.remove(3)  # Eliminar un elemento
```

Tuplas

Las tuplas son inmutables y ordenadas.

```
1  mi_tupla = (1, 2, 3)
```

Diccionarios

Diccionarios almacenan pares clave-valor.

```
1  mi_diccionario = {"nombre": "Python", "año": 1991}
2  print(mi_diccionario["nombre"])  # Acceder al valor de una clave
3  mi_diccionario["año"] = 2023  # Modificar un valor
```

Conjuntos (sets)

Un conjunto no admite duplicados.

```
1  mi_conjunto = {1, 2, 3, 4, 4}
2  mi_conjunto.add(5)  # Añadir un elemento
3  mi_conjunto.remove(3)  # Eliminar un elemento
```

PROGRAMACIÓN ESTRUCTURADA(REPASO)

Estracturas de datos en C/C++

```c
// Definimos una estructura para representar un sensor
struct Sensor {
    int id;
    float temperatura;
    char unidad;
};
```

```c
int main() {
    struct Sensor s1;

    s1.id = 101;
    s1.temperatura = 36.5;
    s1.unidad = 'C';
```

```c
struct Sensor *ptr = &s1;
printf("%.1f\n", ptr->temperatura);
```

Arrays

Declaración y acceso a arrays

```cpp
1 int arr[5] = {1, 2, 3, 4, 5};
2 cout << arr[0]; // Imprime 1
```

Punteros

Un puntero almacena la dirección de una variable.

```cpp
1 int x = 10;
2 int* p = &x; // p apunta a la dirección de x
3 cout << *p;  // Imprime el valor de x
```

`new` y `delete`

Se utilizan para reservar y liberar memoria en el heap.

```cpp
1 int* p = new int;   // Reserva memoria
2 *p = 10;
3 delete p;          // Libera memoria
```

Arrays dinámicos

```cpp
1 int* arr = new int[5];  // Reserva un array
2 delete[] arr;           // Libera el array
```

POO

La **POO** es un paradigma de programación basado en el uso de **objetos**, que combinan **variables (atributos)** y procedimientos y funciones (métodos).

Clase en Python

Declaración de clases

```python
class Persona:
    def __init__(self, nombre, edad):
        self.nombre = nombre
        self.edad = edad

    def presentarse(self):
        print("Hola, soy", self.nombre, "y tengo", self.edad, "años.")
```

Creación de objetos

```python
persona1 = Persona("Luis", 30)
persona1.presentarse()
```

* Una clase es una plantilla mediante la cual se crean los diferentes objetos requeridos para la solución del problema.
* Los objetos son instancias de las clases.

Clase en C/C++

Definición de clase

Una clase es una plantilla para crear objetos.

```cpp
class Persona {
  public:
    string nombre;
    int edad;
    void saludar() {
        cout << "Hola, soy " << nombre << endl;
    }
};
```

Creación de objetos

```cpp
Persona p;
p.nombre = "Luis";
p.edad = 25;
p.saludar();
```

POO: CARACTERÍSTICAS

- Facilita la reutilización de código a través de conceptos como la herencia y la composición. Esto permite aprovechar clases existentes para crear nuevas clases, lo que ahorra tiempo y esfuerzo.
- Proporciona una estructura organizada para el código al dividirlo en clases y objetos. Esto mejora la modularidad y facilita la comprensión y mantenimiento del software.
- Permite la abstracción de detalles complejos, centrándose en la representación de objetos del mundo real. Esto facilita la comprensión del sistema y la resolución de problemas.
- La modularidad y la estructura organizada de la POO facilitan la identificación y corrección de errores. Los cambios y actualizaciones se pueden realizar de manera más eficiente sin afectar otras partes del sistema.
- Facilita la escalabilidad del software al permitir la adición de nuevas funcionalidades mediante la creación de nuevas clases o la extensión de clases existentes.
- La POO puede introducir cierta sobrecarga (ralentización), pero el beneficio en mantenibilidad y escalabilidad suele ser mucho mayor.

ABSTRACCIÓN

Es la capacidad de representar entidades del mundo real mediante clases, capturando solo las características esenciales. Es decir, permite trabajar con modelos simplificados sin conocer todos los detalles internos.

Ejemplo: Una clase Motor de forma genérica abstrae propiedades como potencia, velocidad, y comportamientos como arrancar(), detener() o setVel(100).

```cpp
class SensorTemperatura {
public:
    std::string marca = "SensTech";
    std::string unidad = "°C";
    float valor_actual;

    void leer() {
        valor_actual = 20 + rand() % 80;
    }

    void mostrar_datos() {
        std::cout << "Marca: " << marca << ", Valor: " << valor_actual << " "
            << unidad << std::endl;
    }
};

int main() {
    SensorTemperatura sensor;
    sensor.leer();
    sensor.mostrar_datos();
    return 0;
}
```

```python
class SensorTemperatura:
    def __init__(self):
        self.marca = "SensTech"
        self.unidad = "°C"
        self.valor_actual = 0

    def leer(self):
        self.valor_actual = random.randint(20, 100)

    def mostrar_datos(self):
        print(f"Marca: {self.marca}, Valor: {self.valor_actual} {self.unidad}")

sensor = SensorTemperatura()
sensor.leer()
sensor.mostrar_datos()
```

Ventajas:
Permite simplificar sistemas complejos para trabajar a alto nivel.
Posibilita la separación hardware / software con un diseño modular y reutilizable.

ENCAPSULAMIENTO

Consiste en ocultar los detalles internos del funcionamiento de un objeto y exponer solo lo necesario mediante interfaces (métodos públicos).
Ejemplo: Los atributos de un sensor de temperatura están protegidos y se accede a ellos mediante funciones.

```cpp
// Definimos una clase llamada Sensor
class Sensor {
private:
    float temperatura; // Atributo privado: solo accesible dentro de la clase

public:
    // Método público para modificar la temperatura
    void setTemperatura(float t) {
        temperatura = t;
    }

    // Método público para obtener la temperatura
    float getTemperatura() {
        return temperatura;
    }
};

int main() {
    Sensor s;                      // Instancia de la clase
    s.setTemperatura(36.5);       // Asignamos una temperatura usando el método público
    cout << "Temperatura: "
         << s.getTemperatura()
         << "°C\n";               // Mostramos el valor encapsulado
    return 0;
}
```

```python
class Sensor:
    def __init__(self):
        self.__temperatura = 0.0  # Atributo "privado" con doble guión bajo

    def set_temperatura(self, t):
        self.__temperatura = t   # Método público para modificar

    def get_temperatura(self):
        return self.__temperatura  # Método público para acceder

s = Sensor()
s.set_temperatura(36.5)
print("Temperatura:", s.get_temperatura(), "°C")
```

Idea clave:
Encapsular es proteger los datos y controlar cómo se modifican. Por ejemplo, si un equipo solo puede operar de forma segura en un rango de 0 a 500rpm, la variable de velocidad solo debe ser accesible desde una función que impida ingresos erróneos.

HERENCIA

Permite que una clase herede atributos y métodos de otra, facilitando la reutilización de código y el modelado de familias de dispositivos.

```cpp
class Motor {
public:
    void encender() { std::cout << "Motor encendido" << std::endl; }
};

class MotorElectrico : public Motor {
public:
    int voltaje = 220;
};

class MotorCombustion : public Motor {
public:
    std::string tipo_combustible = "Diesel";
};

int main() {
    MotorElectrico me;
    me.encender();

    MotorCombustion mc;
    mc.encender();

    return 0;
}
```

```python
class Motor:
    def encender(self):
        print("Motor encendido")

class MotorElectrico(Motor):
    def __init__(self):
        self.voltaje = 220

class MotorCombustion(Motor):
    def __init__(self):
        self.tipo_combustible = "Diesel"

me = MotorElectrico()
me.encender()

mc = MotorCombustion()
mc.encender()
```

Ejemplo práctico: a partir de la clase base "motor()" puedo tener clases "MotorElectrico() y MotorCombustion()" que comparten métodos arrancar() / detener(), pero cada una los implementa internamente de forma distinta.

POLIMORFISMO

Permite que métodos con el mismo nombre se comporten de manera diferente según el objeto o contexto.

Ejemplo: El método encender() puede tener distintas implementaciones para un MotorCombustion o un MotorElectrico.

```cpp
class Motor {
public:
    virtual void encender() {
        std::cout << "Encendiendo motor genérico" << std::endl;
    }
};

class MotorElectrico : public Motor {
public:
    void encender() override {
        std::cout << "Encendiendo motor eléctrico" << std::endl;
    }
};

class MotorCombustion : public Motor {
public:
    void encender() override {
        std::cout << "Encendiendo motor a combustión" << std::endl;
    }
};

int main() {
    std::vector<Motor*> motores = {new MotorElectrico(), new MotorCombustion()};
    for (Motor* m : motores) {
        m->encender();
        delete m;
    }
    return 0;
}
```

```python
class Motor:
    def encender(self):
        print("Encendiendo motor genérico")

class MotorElectrico(Motor):
    def encender(self):
        print("Encendiendo motor eléctrico")

class MotorCombustion(Motor):
    def encender(self):
        print("Encendiendo motor a combustión")

motores = [MotorElectrico(), MotorCombustion()]
for m in motores:
    m.encender()
```

EJEMPLO

Clase Padre

```cpp
class Sensor {
private:
    float valor;

protected:
    std::string nombre;

public:
    virtual float leer() = 0;

    void setValor(float v){
        valor = v;
    }

    float getValor(){
        return valor;
    }
};
```

Atributo heredable, pero solo accesible desde la propia clase. (Analogía: electrónica interna sellada).

Atributo heredable y accesible dentro de clases padres como hijas. (Analogía: accesible para técnicos).

Atributo heredable y accesible públicamente. (Analogía: botones externos).

```cpp
int main(){

    Sensor* sensor;

    sensor = new SensorTemperatura();

    std::cout << sensor->leer() << std::endl;

    sensor = new SensorPresion();

    std::cout << sensor->leer() << std::endl;

}
```

✗
```cpp
sensor.valor = 99999;   // MAL
```

✓
```cpp
sensor.setValor(20);
```

Clases hijas

```cpp
class SensorTemperatura : public Sensor {
public:
    float leer() override {

        nombre = "Temperatura";

        setValor(25);

        return getValor();
    }
};
```

```cpp
class SensorPresion : public Sensor {
public:
    float leer() override {

        nombre = "Presion";

        setValor(1.5);

        return getValor();
    }
};
```

CONSTRUCTOR

Son funciones miembro especiales que sirven para inicializar un objeto de una determinada clase al mismo tiempo que se declara. Los constructores son especiales por varios motivos:

- Tienen el mismo nombre que la clase a la que pertenecen.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- No pueden ser heredados.
- Por último, deben ser públicos, no tendría ningún sentido declarar un constructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.

```cpp
class Rectangulo {
public:
    int ancho, alto;

    Rectangulo(int ancho, int alto) { // Constructor
        this->ancho = ancho;
        this->alto = alto;
        cout << "Constructor llamado" << endl;
    }
};

int main() {
    Rectangulo rect(10, 5); // Se llama el constructor
    return 0;
}
```

DESTRUCTOR

Son funciones miembro especiales que sirven para eliminar un objeto de una determinada clase. Al igual que los constructores, los destructores también tienen algunas características especiales:

- Tienen el mismo nombre que la clase a la que pertenecen.
- No tienen tipo de retorno, y por lo tanto no retornan ningún valor.
- No tienen parámetros.
- No pueden ser heredados.
- Deben ser públicos, no tendría ningún sentido declarar un destructor como privado, ya que siempre se usan desde el exterior de la clase, ni tampoco como protegido, ya que no puede ser heredado.
- No pueden ser sobrecargados, lo cual es lógico, puesto que no tienen valor de retorno ni parámetros, no hay posibilidad de sobrecarga.

```
class Rectangulo {
public:
    int ancho, alto;

    Rectangulo(int ancho, int alto) { // Constructor
        this->ancho = ancho;
        this->alto = alto;
        cout << "Constructor llamado" << endl;
    }

    ~Rectangulo() { // Destructor
        cout << "Destructor llamado" << endl;
    }
};

int main() {
    Rectangulo rect(10, 5); // Se llama el constructor
    // Cuando rect sale de la función main, se llama el destructor
    return 0;
}
```

EJEMPLO CON MEMORIA DINÁMICA

```cpp
class String {
public:
    char* cadena;
    int longitud;

    String(const char* str) {
        longitud = strlen(str);
        cadena = new char[longitud + 1];
        strcpy(cadena, str);
        cout << "Constructor llamado" << endl;
    }

    ~String() {
        delete[] cadena; // Liberar la memoria asignada dinámicamente
        cout << "Destructor llamado" << endl;
    }
};

int main() {
    String str("Hola, mundo!"); // Se llama el constructor
    // Cuando str sale de main, se llama el destructor
    return 0;
}
```

En este ejemplo, el constructor String(const char* str) asigna memoria dinámicamente para almacenar la cadena.

El destructor ~String() libera esta memoria antes de que el objeto String deje de existir, evitando así fugas de memoria.

Y EN PYTHON?

En Python, el constructor se define con el método __init__ y el destructor con __del__. El constructor se llama al crear un objeto de una clase, y el destructor se llama cuando el objeto se destruye

```python
class Perro:
    def __init__(self, nombre, raza):
        self.nombre = nombre  # Atributo de instancia
        self.raza = raza      # Atributo de instancia

    def ladrar(self):
        print(f"{self.nombre} ladra")
```

Explicación:

- __init__ es un método especial que se llama cuando se crea un objeto de la clase Perro.
- self es una referencia a la instancia de la clase.
- nombre y raza son argumentos que se pasan al constructor para inicializar los atributos self.nombre y self.raza

```python
class Archivo:
    def __init__(self, nombre):
        self.nombre = nombre
        self.archivo = None

    def abrir(self):
        self.archivo = open(self.nombre, "w")

    def cerrar(self):
        if self.archivo:
            self.archivo.close()

    def __del__(self):
        self.cerrar() #Limpia el recurso antes de destruir el objeto
        print(f"El archivo {self.nombre} ha sido destruido")
```

Explicación:

- __del__ es un método especial que se llama cuando el objeto Archivo es destruido.
- Dentro de __del__, se cierra el archivo si estaba abierto para liberar el recurso.
- Se muestra un mensaje indicando que el archivo ha sido destruido