PUCP

PROGRAMACIÓN 3 (1INF30)

Programación Orientada a Objetos

2025

Prof. Freddy Paz

AGENDA

1. ¿Qué es un objeto?
2. ¿Qué son las clases?
3. Constructor
4. Destructor
5. Herencia
6. Principios de POO
7. Polimorfismo (subtipo, sobrecarga y paramétrico)
8. Encapsulamiento
9. Clase Abstracta
10. Clase de Tipo Interface
11. Miembros Estáticos

AGENDA

12. Clase Anidada
13. Enumerados
14. Arreglos
15. Indizadores
16. Manejo de Fechas
17. Lectura por Consola
18. Manejo de Listas
19. instanceof / is
20. Programando relaciones entre Clases

¿Qué es un objeto?

- Es la representación del **estado** y **comportamiento** de un objeto real o abstracto.
- El **estado** está representado por un conjunto de "datos".
- El **comportamiento** está representado por un conjunto de "métodos".

¿Qué es un objeto?

- En la POO, un programa se conceptualiza como **un conjunto de objetos en interacción**.
- Los objetos interactúan enviándose **mensajes** unos a otros.
- Un objeto responde a un mensaje ejecutando un método.

¿Qué son las clases?

- Un conjunto de **objetos** que comparten el mismo comportamiento y tipos de sus datos (no los valores) se dicen que pertenecen a la misma **clase**.
- En POO, una **clase** es un tipo de dato, cuyas instancias son objetos.
- Una **clase** contiene la descripción de los datos y métodos de un conjunto de **objetos**.

¿Qué son las clases?

- Las clases deben ser implementadas de forma tal que los objetos que de ellas se creen **siempre** tengan un **estado consistente**.
- En una clase es posible abstraer:
  - Datos (datos miembros)
  - Métodos (funciones miembros)
  - Otros tipos de datos (tipos miembros)

Constructor

- Los constructores son métodos pertenecientes a la clase. Se utilizan para **construir** o **instanciar** una clase. Puede haber varios constructores, de acuerdo con las necesidades del usuario.

Constructor

```
public class Estudiante {
	private String nombre;
	private double CRAEST;
	public Estudiante(){

	}
	public Estudiante(String nombre, double CRAEST){
		this.nombre = nombre;
		this.CRAEST = CRAEST;
	}
}
```

Constructor

C#

```
public class Estudiante {
	private string nombre;
	private double CRAEST;
	public Estudiante(){

	}
	public Estudiante(string nombre, double CRAEST){
		this.nombre = nombre;
		this.CRAEST = CRAEST;
	}
}
```

Destructor

- El destructor se utiliza para **destruir** una instancia de una clase y liberar memoria. En Java y C#, no es necesario definir constructores, ya que la liberación de memoria es llevada a cabo por el **Garbage Collector** cuando las instancias de los objetos quedan no referenciadas.

Destructor

```
public class Estudiante {
	private String nombre;
	private double CRAEST;
	public Estudiante(){

	}
	public void finalize(){
		System.out.println("El objeto se esta destruyendo");
	}
}
```

Destructor

C#

```
public class Estudiante {
      private string nombre;
      private double CRAEST;
      public Estudiante(){

      }
      ~Estudiante(){
        System.Console.WriteLine("Se esta destruyendo un objeto");
    }
}
```

Herencia

- La herencia permite crear nuevas clases que reutilizan, extienden y modifican el comportamiento que se define en otras clases. La clase cuyos miembros se heredan se denomina ***clase base*** y la clase que hereda esos miembros se denomina ***clase derivada***. Una clase derivada **solo puede tener una clase base directa**. Sin embargo, la herencia es transitiva.

Herencia

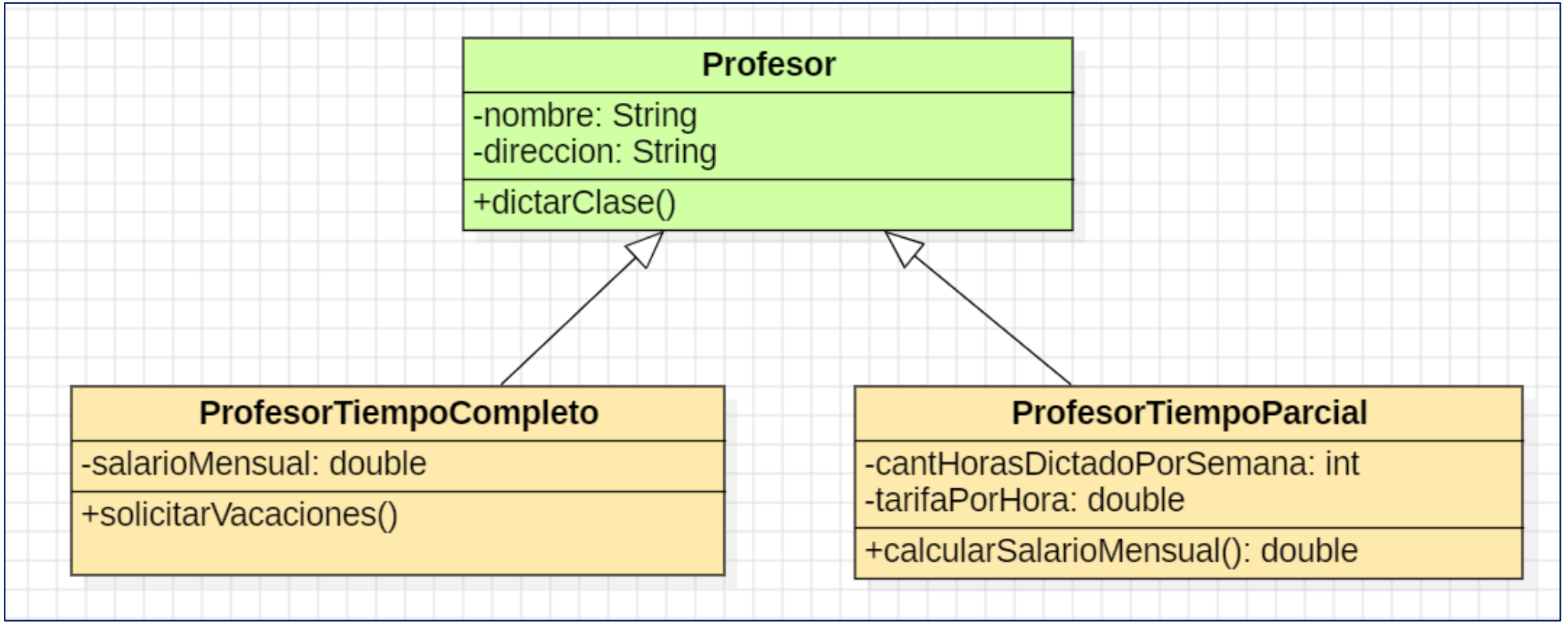

Fig. 01. Diagrama de clases con notación UML donde se evidencia un ejemplo la herencia

Herencia

```
public class ProfesorTiempoParcial extends Profesor {
	private int cantHorasDictadoPorSemana;
	private double tarifaPorHora;
	public ProfesorTiempoParcial(){

	}
	public double calcularSalarioMensual(){
		return cantHorasDictadoPorSemana * tarifaPorHora;
	}
}
```

Herencia

C#

```
public class ProfesorTiempoParcial : Profesor {
      private int cantHorasDictadoPorSemana;
      private double tarifaPorHora;
      public ProfesorTiempoParcial(){

      }
      public double calcularSalarioMensual(){
            return cantHorasDictadoPorSemana * tarifaPorHora;
      }
}
```

Principios de Programación Orientada a Objetos

- **Encapsulamiento**
  - Al restringir el acceso de otros objetos a sus datos. Acceso indirecto, por sus métodos.
- **Herencia**
- **Polimorfismo**
  - Es posible enviar el mismo mensaje, o mensajes iguales a objetos de distintos tipos.

Polimorfismo

- El polimorfismo se refiere a la propiedad por la que es posible **enviar mensajes sintácticamente iguales** a objetos de tipos distintos. El único requisito que deben cumplir los objetos que se utilizan de manera polimórfica es saber responder al mensaje que se les envía.
- Existen 3 tipos de polimorfismo: (1) *subtipo*, (2) *sobrecarga* y (3) *paramétrico*.

Polimorfismo de Subtipo

- Se refiere a la capacidad de un objeto de una clase base de comportarse como un objeto de cualquier clase derivada de esa clase base. Este tipo de polimorfismo se implementa mediante herencia y métodos virtuales o abstractos.

Polimorfismo de Subtipo

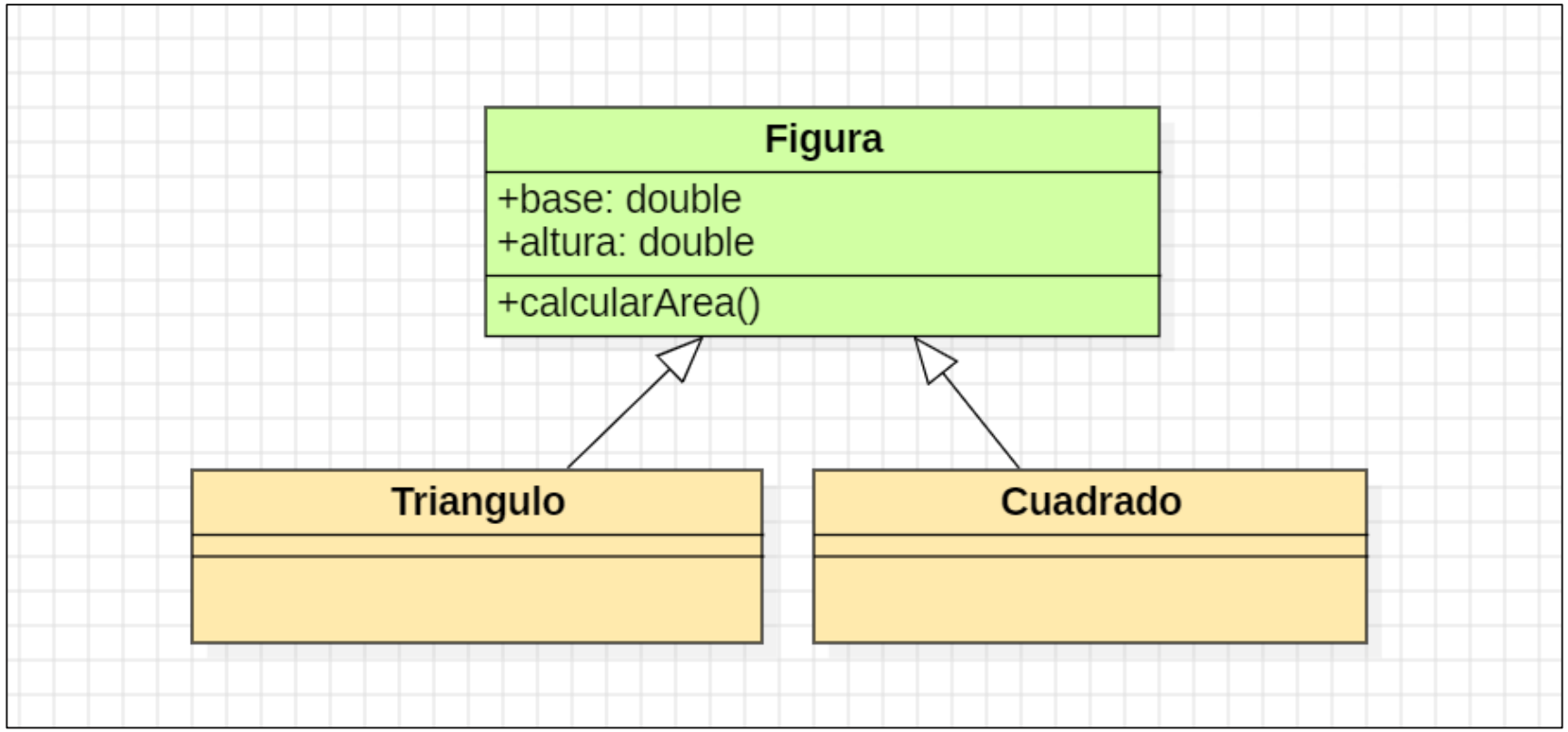

Fig. 02. Diagrama de clases con notación UML donde se evidencia un ejemplo de polimorfimo de subtipo

Polimorfismo de Subtipo

```
public class Figura {
	public double base;
	public double altura;
	public Figura(double base, double altura){
		this.base = base;
		this.altura = altura;
	}
	public void calcularArea(){
		System.out.println("Procedimiento para calcular el
area");
	}
}
```

Polimorfismo de Subtipo

```
public class Triangulo extends Figura{

      public Triangulo(double base, double altura){
            super(base,altura);
      }

      @Override
      public void calcularArea(){
            System.out.println(base * altura / 2);
      }
}
```

Polimorfismo de Subtipo

```
public class Cuadrado extends Figura{

      public Cuadrado(double base, double altura){
            super(base,altura);
      }

      @Override
      public void calcularArea(){
            System.out.println(base * altura);
      }
}
```

Polimorfismo de Subtipo

```
public class Principal {

      public static void main(String[] args){
            Figura t1 = new Triangulo(10,20);
            Figura c1 = new Cuadrado(10,20);

            t1.calcularArea(); c1.calcularArea();

            Triangulo t2 = new Triangulo(10,20);
            Cuadrado c2 = new Cuadrado(10,20);
            t2.calcularArea(); c2.calcularArea();
      }
}
```

Sobrecarga

- La sobrecarga (***overload***) es un tipo de polimorfismo, que se caracteriza de poder definir más de un método o constructor con el mismo nombre (***identificador***), siendo distinguidos entre sí por el número y la clase (***tipo***) de los argumentos.

Sobrecarga

C#

```
public class Persona {
	private string nombre;
	private string apellidoPaterno;
	//Sobrecarga del constructo
	public Persona(){}
	public Persona(string nombre){
		this.nombre = nombre;
	}
	public Persona(string nombre, string apellidoPaterno){
		this.nombre = nombre;
		this.apellidoPaterno = apellidoPaterno;
	}
}
```

Sobrecarga

C#

```
public class Operacion {
	//Sobrecarga del método sumar
	public int sumar(int a, int b){
		return a + b;
	}
	public int sumar(int a, int b, int c){
		return a + b + c;
	}
	public double sumar(double a, double b){
		return a + b;
	}
}
```

Polimorfismo Paramétrico

- Se refiere a la capacidad de escribir código que pueda trabajar con **cualquier tipo de datos**. En este contexto, "paramétrico" se refiere a los parámetros de tipo que se utilizan para definir comportamientos o estructuras de datos que pueden ser parametrizados por un tipo específico.

Polimorfismo Patramétrico

```
public class Contenedor<T> {
	public T dato;
	public Contenedor(T dato){
		this.dato = dato;
	}
	public void imprimirTipoDato(){
		System.out.println(dato.getClass().getSimpleName());
	}
}
```

Polimorfismo Patramétrico

```
public class Principal{
	public static void main(String[] args){
		Contenedor<Integer> contenedorEntero = new
Contenedor<>(10);
		Contenedor<String> contenedorString = new
Contenedor<>("Hola");

		contenedorEntero.imprimirTipoDato();
		contenedorString.imprimirTipoDato();
	}
}
```

Encapsulamiento

- Se refiere a ocultar los detalles internos de un objeto y solo exponer las operaciones o funcionalidades. Consiste en agrupar los datos y los métodos que operan sobre esos datos dentro de una clase y controlar el acceso a estos datos mediante modificadores de acceso ("**private**", "**public**", "**protected**", etc.). Esto significa que los datos son inaccesibles directamente desde fuera de la clase y solo pueden ser modificados o accedidos mediante métodos específicos proporcionados por la clase (conocidos como ***getters*** y ***setters***).

Encapsulamiento

```
public class Estudiante{

	private String nombre;

	public String getNombre(){
		return nombre;
	}
	public void setNombre(String nombre){
		this.nombre = nombre;
	}
}
```

Encapsulamiento

C#

```
public class Estudiante{
	private string nombre;
	public string Nombre{
		get{
			return nombre;
		}
		set{
			this.nombre = value;
		}
	}
}
```

Clase Abstracta

- Puede declarar una clase como abstracta si desea evitar la creación directa de instancias por medio de la palabra clave **new**. Si hace esto, la clase solo se puede utilizar si una nueva clase se deriva de ella.
- Basta con que un método sea abstracto para que la clase sea abstracta. A las clases que tienen todos sus métodos implementados se les llama "**clases concretas**". De manera similar un método declarado y no implementado se le dice "**método abstracto**", y uno implementado se le dice "**método concreto**"

Clase Abstracta

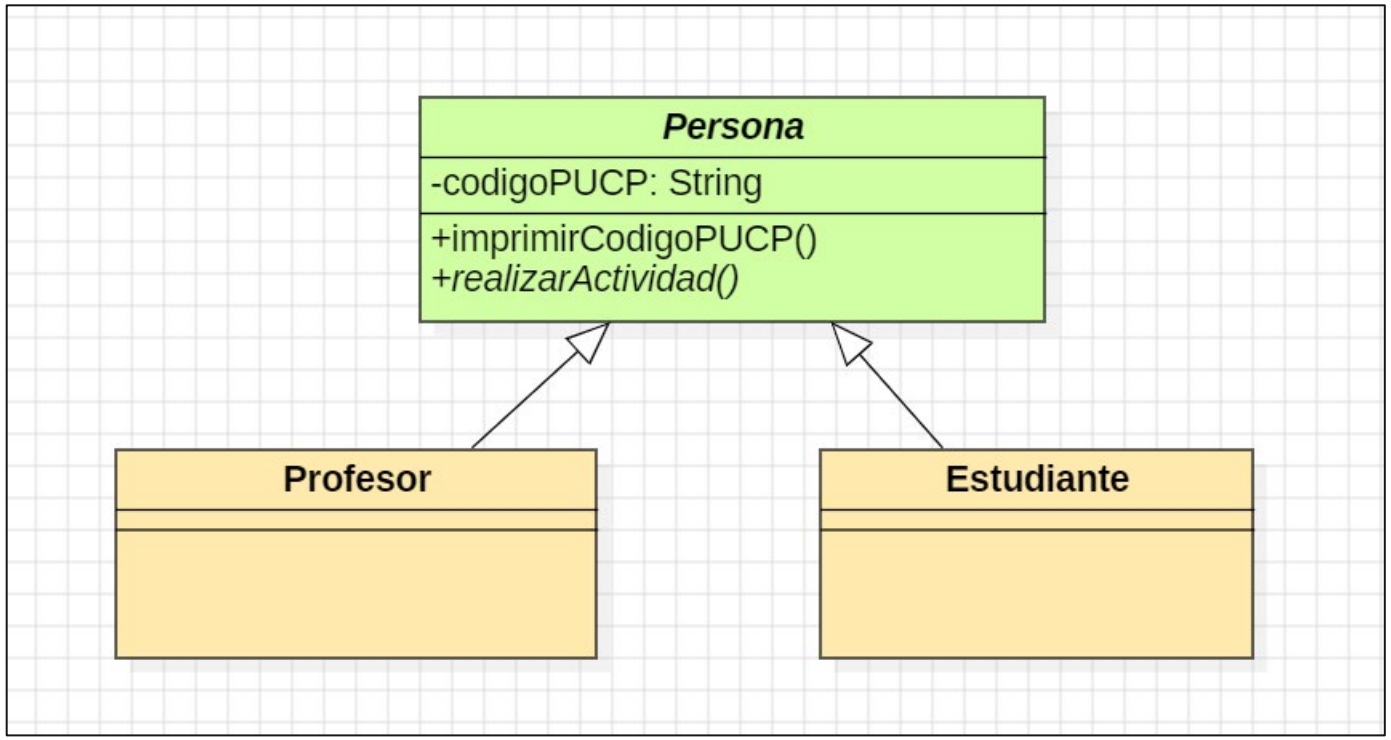

Fig. 03. Diagrama de clases con notación UML donde se evidencia un ejemplo de clase abstracta

- Utilizar cuando exista una clase de la cual es necesario heredar (pues agrupa características y comportamientos) pero que NO debe ser instanciada en nuestro programa.

Clase Abstracta

```
public abstract class Persona{
	private String codigoPUCP;
	public Persona(String codigoPUCP){
		this.codigoPUCP = codigoPUCP;
	}
	public void imprimirCodigoPUCP(){
		System.out.println(codigoPUCP);
	}
	public abstract void realizarActividad();
}
```

Clase de Tipo Interface

- Define el comportamiento de una clase, pero no la implementación.
- Las interfaces se utilizan para definir funciones específicas para las clases que no tienen necesariamente una relación de identidad.
- No se establece el modo de acceso de los métodos de una interfaz. Por defecto son públicos.

Clase de Tipo Interface

- Utilizar cuando notamos comportamientos similares que obligatoriamente deben ser implementados por algunas clases.

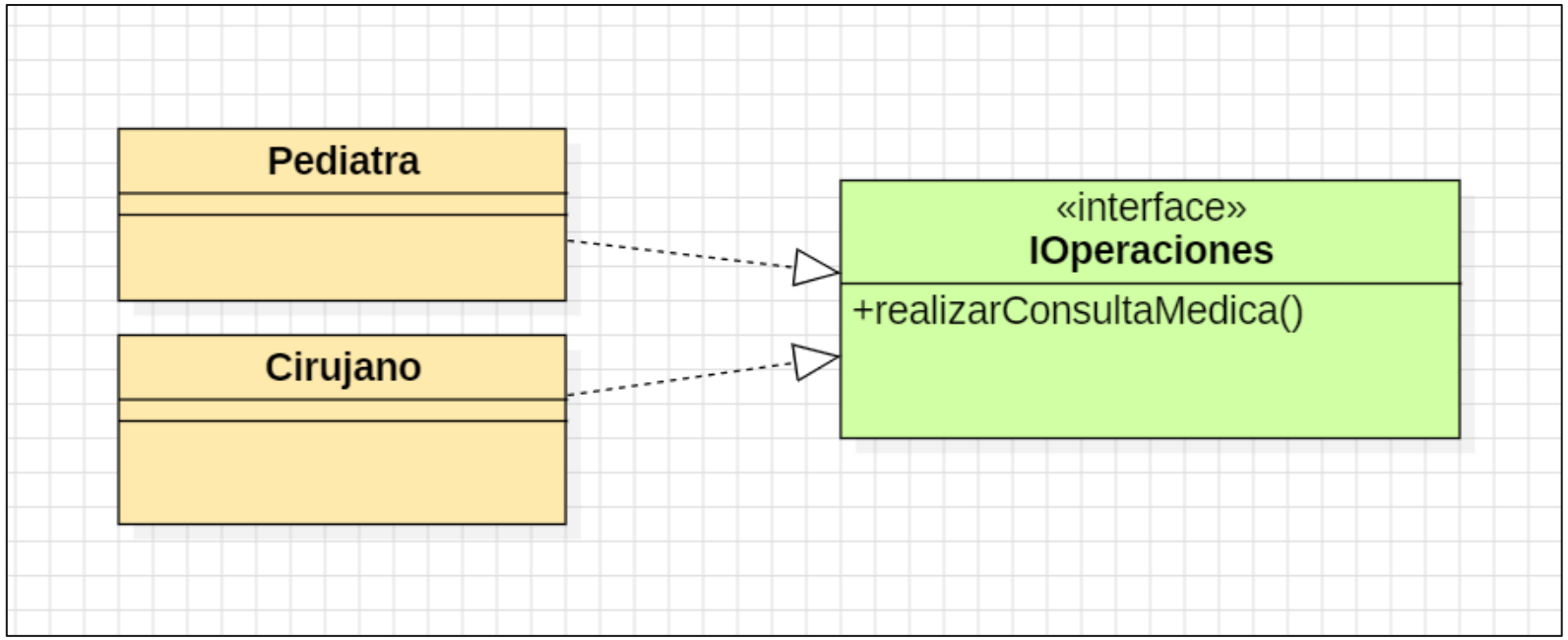

Fig. 04. Diagrama de clases con notación UML donde se evidencia un ejemplo de clase de tipo interface

Clase de Tipo Interface

```
interface IOperaciones{
	void realizarConsultaMedica();
}
public class Cirujano implements IOperaciones{
	@Override
	public void realizarConsultaMedica(){
		System.out.println("El cirujano esta realizando una cirugia");
	}
}
```

Clase de Tipo Interface

C#

```
interface IOperaciones{
      void realizarConsultaMedica();
}
public class Cirujano : IOperaciones{

      public void realizarConsultaMedica(){
            System.Console.WriteLine("El cirujano esta realizando
una cirugia");
      }
}
```

Comparación Clase Abstracta - Interface

	Clase abstracta	Clase interface
Declarar métodos abstractos	✓	✓
Implementar métodos	✓	✗
Añadir datos miembros	✓	✗
Crear objetos	✗	✗
Crear arreglos, referencias	✓	✓

Miembros Estáticos

- Constituyen datos, métodos y tipos que forman parte de un tipo de dato (por ejemplo, una clase) pero que no requieren una instancia de este para ser utilizados.

Clase Anidada

- Es una clase definida como miembro de otra clase.
- En general, un tipo de dato definido dentro de otro se le llama tipo de dato anidado.
- Se le conoce como clase ***inner***, y a la clase dentro de la que se definen, clase ***outer***.

```
public class A{
      public class B{
      }
}
```

Clase Anidada

```
public class Computador{
	public void imprimir(){
		System.out.println("Imprimir desde Computador");
	}
	public class Microprocesador{
		public void imprimir(){
			System.out.println("Imprimir desde
Microprocesador");
		}
	}
}
```

Clase Anidada

```
public class Principal{

      public static void main(String[] args){
            Computador c = new Computador();
            c.imprimir();
            Computador.Microprocesador m = c.new Microprocesador();
            m.imprimir();
      }
}
```

Clase Anidada

```
public class Computador{
      public void imprimir(){
            System.Console.WriteLine("Imprimir desde Computador");
      }
      public class Microprocesador{
            public void imprimir(){
                  System.Console.WriteLine("Imprimir desde
Microprocesador");
            }
      }
}
```

C#

Clase Anidada

```
public class Principal{

      public static void Main(string[] args){
            Computador c = new Computador();
            c.imprimir();
            Computador.Microprocesador m = new
Computador.Microprocesador();
            m.imprimir();
      }
}
```

C#

Enumerados

- La enumeración (también denominado ***enum***) proporciona una manera eficaz de definir un conjunto de constantes integrales con nombre que pueden asignarse a una variable.

Enumerados

```
enum Dias
{
      Domingo, Lunes, Martes, Miercoles, Jueves,
Viernes, Sabado
}

public class Principal{
      public static void main(String[] args){
            Dias d = Dias.Domingo;
      }
}
```

Enumerados

C#

```
enum Dias
{
      Domingo, Lunes, Martes, Miercoles, Jueves,
Viernes, Sabado
}

public class Principal{
      public static void Main(string[] args){
            Dias hoyDia = Dias.Lunes;
      }
}
```

Arreglos

C#

```
public class Prueba{
    public static void Main(){
        //Unidimensionales
        int[] a = {0,1,2,3,4,5,6,7,8,9};
        int[] b = new int[10]{0,1,2,3,4,5,6,7,8,9};
        //Multiples dimensiones
        int[,] aa = {{0,0},{1,1},{2,2}};
        int[,] bb = new int[2,3];
        //Matrices escalonadas
        int[][] aaa = new int[3][];//Cantidad de filas
        aaa[0] = new int[2]; //Cantidad de columnas en fila 0
        aaa[1] = new int[3]{1,2,3}; //Cantidad de columnas en fila 1
        aaa[2] = new int[4];//Cantidad de columnas en fila 2
    }
}
```

Arreglos

```
1  public class Arreglo{
2      public static void main(String[] args){
3          //Unidimensionales
4          int[] a = {0,1,2,3,4,5,6,7,8,9};
5          int[] b = new int[10];
6          //Matrices escalonadas
7          int[][] bb = new int[3][]; //Cantidad de filas
8          bb[0] = new int[2]; //Cantidad de columnas en la fila 1
9          bb[1] = new int[3]; //Cantidad de columnas en la fila 2
10         bb[2] = new int[4]; //Cantidad de columnas en la fila 3
11         int[][] aa = {{6,7,5,0,4},
12                       {3,8,4},
13                     {1,0,2,7},
14                       {9,5}};
15     }
16 }
```

Indizadores

- Permite trabajar un objeto como si fuese un arreglo.

C#

```
public class Departamento{
      private string[] empleados = string[10];
      public string this[int indice]
         {
              set{
                   empleados[indice] = value;
              }
              get{
                   return empleados[indice];
              }
         }
}
```

Indizadores

C#

```
public class Principal{
      public static void Main(){
            departamento d = new departamento();
            d[0] = "Juan";
            d[1] = "Marco";
            System.Console.WriteLine(d[0]);
            System.Console.WriteLine(d[1]);
      }
}
```

Manejo de Fechas – DateTime C#

C#

```
using System;
public class Principal{
      public static void Main(string[] args){
            DateTime fecha = DateTime.ParseExact("18-03-2024 15:00",
"dd-MM-yyyy HH:mm",
System.Globalization.CultureInfo.InvariantCulture);
            System.Console.WriteLine(fecha.ToString("dd-MM-yyyy
HH:mm"));
      }
}
```

Manejo de Fechas – Date JAVA

```
import java.util.Date;
import java.text.SimpleDateFormat;
public class Principal{
      public static void main(String[] args) throws Exception{
            SimpleDateFormat sdf = new SimpleDateFormat("dd-MM-yyyy
HH:mm");
            Date fecha = sdf.parse("18-03-2024 15:00");
            System.out.println(sdf.format(fecha));
      }
}
```

Lectura por Consola – C#

C#

```
public class Principal{

      public static void Main(string[] args){

            string nombre = System.Console.ReadLine();
            System.Console.WriteLine(nombre);

      }
}
```

Lectura por Consola – JAVA – Primera Forma

```java
import java.util.Scanner;
public class Principal{

	public static void main(String[] args){

		String nombre;
		Scanner teclado = new Scanner(System.in);
		nombre = teclado.nextLine();
		System.out.println(nombre);
	}
}
```

Lectura por Consola – JAVA – Segunda Forma

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
public class Principal{

      public static void main(String[] args) throws Exception{

            String nombre;
            BufferedReader teclado = new BufferedReader(new
InputStreamReader(System.in));
            nombre = teclado.readLine();
            System.out.println(nombre);
      }
}
```

Manejo de Listas – C# - List

C#

```
using System.Collections.Generic;
class Empleado{
	public string nombre;
	public Empleado(string nombre){ this.nombre = nombre; }
}
public class Principal{
	public static void Main(string[] args){
		Empleado emp1 = new Empleado("Juan");
		Empleado emp2 = new Empleado("Andrea");
		List<Empleado> empleados = new List<Empleado>();
		empleados.Add(emp1); empleados.Add(emp2);
		foreach(Empleado emp in empleados) {
			System.Console.WriteLine(emp.nombre);
		}
	}
}
```

Manejo de Listas – C# - BindingList

C#

```
using System.ComponentModel;
class Empleado{
	public string nombre;
	public Empleado(string nombre){ this.nombre = nombre; }
}
public class Principal{
	public static void Main(string[] args){
		Empleado emp1 = new Empleado("Juan");
		Empleado emp2 = new Empleado("Andrea");
		BindingList<Empleado> empleados = new BindingList<Empleado>();
		empleados.Add(emp1); empleados.Add(emp2);
		foreach(Empleado emp in empleados) {
			System.Console.WriteLine(emp.nombre);
		}
	}
}
```

Manejo de Listas – JAVA - ArrayList

```
import java.util.ArrayList;
class Empleado{
	public String nombre;
	public Empleado(String nombre){ this.nombre = nombre; }
}
public class Principal{
	public static void main(String[] args){
		Empleado emp1 = new Empleado("Juan");
		Empleado emp2 = new Empleado("Andrea");
		ArrayList<Empleado> empleados = new ArrayList<>();
		empleados.add(emp1); empleados.add(emp2);
		for(Empleado emp : empleados) {
			System.out.println(emp.nombre);
		}
	}
}
```

instanceof (JAVA) – is (C#)

- **instanceof** / **is** son operadores que se utilizan para comprobar si un objeto es una instancia de una clase específica, una instancia de una subclase o una instancia de una clase que implementa una determinada clase de tipo interface.
- Los operadores devuelven **true** si el objeto es una instancia de la clase especificada o de alguna de sus subclases, o si el objeto implementa la interface especificada. De lo contrario, devuelve **false**.

instanceof (JAVA)

```
class Auto extends Vehiculo{ }
class Moto extends Vehiculo{ }
class Vehiculo{ }
public class Principal{
      public static void main(String[] args){
            Vehiculo veh01 = new Auto();
            Vehiculo veh02 = new Moto();
            System.out.println(veh01 instanceof Vehiculo);
            System.out.println(veh01 instanceof Auto);
            System.out.println(veh01 instanceof Moto);
      }
}
```

instanceof (C#)

C#

```
class Auto : Vehiculo{ }
class Moto : Vehiculo{ }
class Vehiculo{ }
public class Principal{
      public static void Main(string[] args){
            Vehiculo veh01 = new Auto();
            Vehiculo veh02 = new Moto();
            System.Console.WriteLine(veh01 is Vehiculo);
            System.Console.WriteLine(veh01 is Auto);
            System.Console.WriteLine(veh01 is Moto);
      }
}
```

Programando relaciones entre Clases

- Suponiendo la siguiente lógica de negocio:
  - *Un **cliente** tiene **muchas** **órdenes de compra**.*
  - *Una **orden de compra** le pertenece a **un** **cliente**.*

Fig. 05. Diagrama de clases con notación UML donde se evidencia un ejemplo de relación de asociación de 1 a muchos.

Programando relaciones entre Clases

```
import java.util.ArrayList;
class Cliente{
        private ArrayList<OrdenCompra> ordenesCompra;
        public ArrayList<OrdenCompra> getOrdenesCompra(){
                return ordenesCompra;
        }
        public void setOrdenesCompra(ArrayList<OrdenCompra> ordenesCompra){
                this.ordenesCompra = ordenesCompra;
        }
}
class OrdenCompra{
        private Cliente cliente;
        public Cliente getCliente(){
                return cliente;
        }
        public void setCliente(Cliente cliente){
                this.cliente = cliente;
        }
}
```

Programando relaciones entre Clases

C#

```
using System.ComponentModel;
class Cliente{
        private BindingList<OrdenCompra> ordenesCompra;
        public BindingList<OrdenCompra> OrdenesCompra{
                get{ return ordenesCompra; }
                set{ this.ordenesCompra = value; }
        }
}
class OrdenCompra{
        private Cliente cliente;
        public Cliente Cliente{
                get{ return cliente; }
                set{ this.cliente = value; }
        }
}
```

Referencias

- D.J. Barnes y M. Kölling, Programación orientada a objetos con Java. Pearson Educación, 2007
- T. Budd, An introduction to Object-Oriented Programming (Third Edition). Pearson Education, 2001
- E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Paterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994
- B. Stroustrup, The C++ Programming Language (Third Edition) Addison-Wesley, 1997
- Agustín Froufe. Java 2. Manual de usuario y tutorial. Ed. Ra-Ma
- J. Sánchez, G. Huecas, B. Fernández y P. Moreno, Iniciación y referencia: Java 2. Osborne McGraw-Hill, 2001.
- B. Meyer, Object-Oriented Software Construction (Second Edition). Prentice Hall, 1997.

¡Gracias!