Programación Orientada a Objetos

Dr. Freddy Alberto Paz Espinoza
fpaz@pucp.pe

¿Qué es un objeto?

Es la representación del **estado** y **comportamiento** de un objeto real o abstracto.

El **estado** esta representado por un conjunto de "datos".

El **comportamiento** esta 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.**

Pueden encapsular: → *ENCAPSULAMIENTO*

- 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 a las necesidades del usuario.

Constructor

```
public class Persona {
	private String nombre;
	private int edad;

	public Persona(){

	}

	public Persona(String nombre, int edad){
		nombre = nombre;
		edad = edad;
	}
}
```

Constructor

C#

```
public class Persona {
	private string nombre;
	private int edad;

	public Persona(){

	}

	public Persona(string nombre, int edad){
		nombre = nombre;
		edad = edad;
	}
}
```

Destructor

El destructor se utiliza para **destruir** una instancia de una clase y liberar memoria. En Java no hay destructores, ya que la liberación de memoria es llevada acabo por el **Garbage Collector** cuando las instancias de los objetos quedan no referenciadas.

Destructor

```
public class Persona {
	private String nombre;
	private int edad;

	public Persona(){

	}

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

Destructor

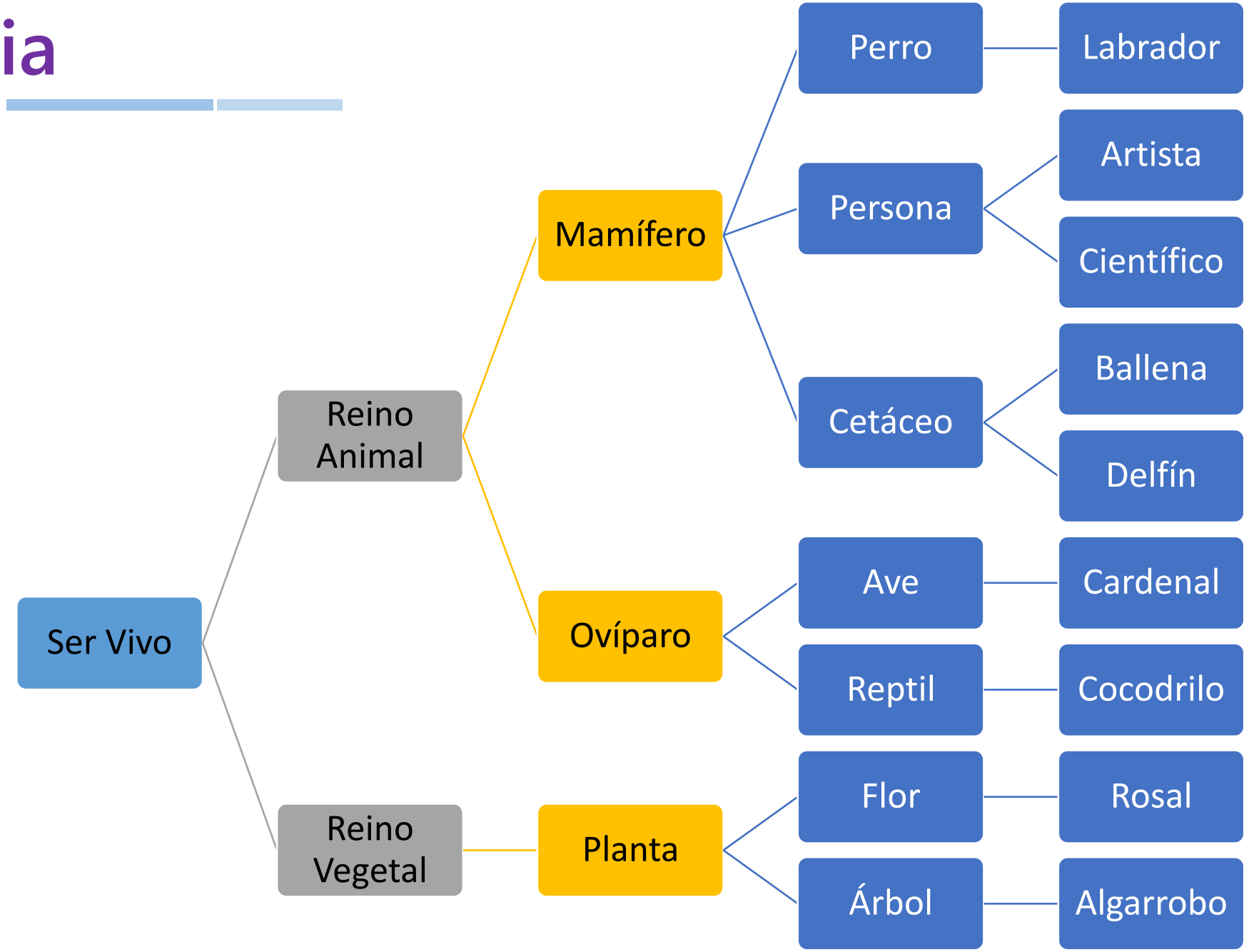
C#

```
public class Persona {
    private string nombre;
    private int edad;

    public Persona(){

    }

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

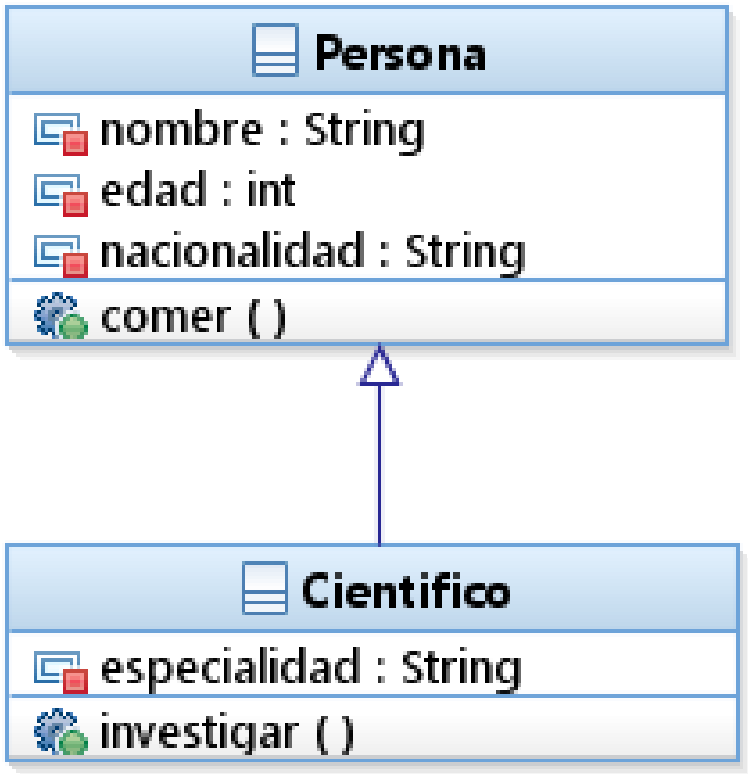
Herencia

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


Ser Vivo
Reino Animal
Reino Vegetal
Mamífero
Ovíparo
Planta
Perro
Persona
Cetáceo
Ave
Reptil
Flor
Árbol
Labrador
Artista
Científico
Ballena
Delfín
Cardenal
Cocodrilo
Rosal
Algarrobo

Herencia


Persona
nombre : String
edad : int
nacionalidad : String
comer ( )
Cientifico
especialidad : String
investigar ( )

Herencia

```
public class Cientifico extends Persona {

    public String especialidad;

    public void investigar () {…}

}
```

Herencia

C#

```
public class Cientifico : Persona {

    public string especialidad;

    public void investigar (){…}

}
```

Principios de POO

Encapsulamiento

Al restringir el acceso de otros objetos a sus datos. Acceso indirecto, por sus métodos.

Herencia

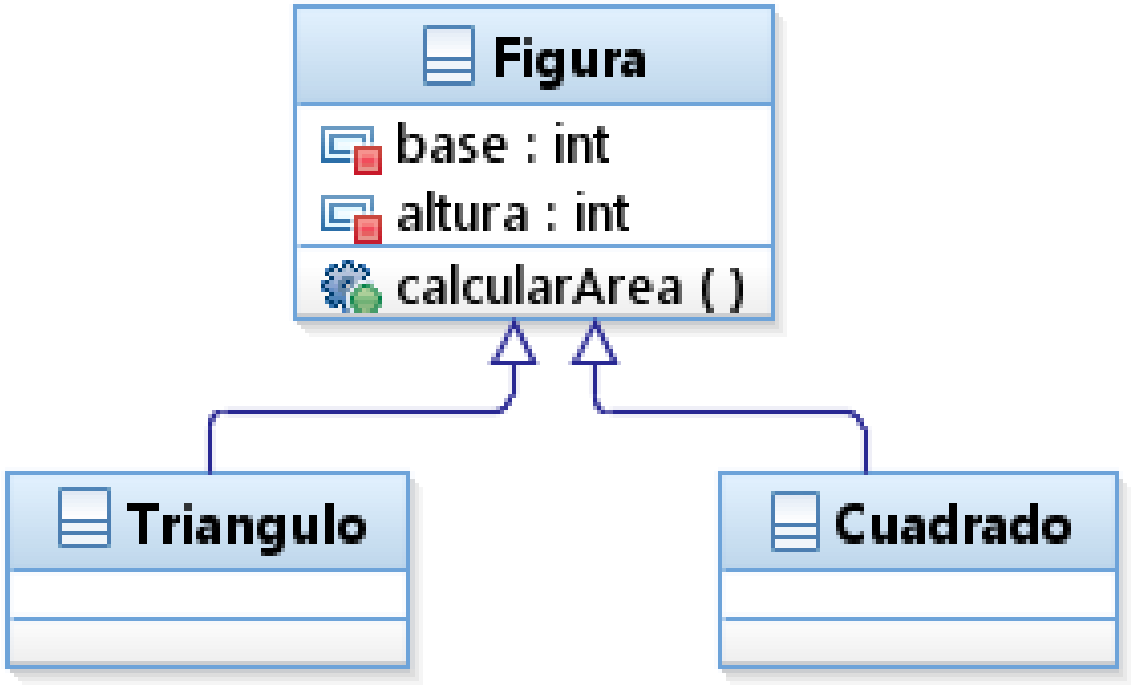
Polimorfismo

Es posible enviar el mismo 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.

Polimorfismo (Ejemplo)



Las clases **Cuadrado** y **Triangulo** implementan el método calcular área de forma específica.

Polimorfismo

```
public class Figura{
	public int base;
	public int altura;

	public Figura(int base, int altura){
		this.base = base;
		this.altura = altura;
	}

	public void calcularArea(){
		System.out.println("Procedimiento para calcular el area");
	}
}
```

Polimorfismo

```
public class Cuadrado extends Figura{
	public Cuadrado(int base, int altura){
		super(base,altura);
	}

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

Polimorfismo

```
public class Triangulo extends Figura{
	public Triangulo(int base, int altura){
		super(base,altura);
	}

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

Polimorfismo

```
public class Principal{
	public static void main(String[] args){
		Triangulo t1 = new Triangulo(10,20);
		Figura t2 = new Triangulo(10,20);
		Cuadrado c1 = new Cuadrado(10,20);
		Figura c2 = new Cuadrado(10,20);
		t1.calcularArea();
		t2.calcularArea();
		c1.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 que la definen.

Sobrecarga

C#

```
public class Persona
   {
      private string nombre;
      public Persona(string nombre){
               this.nombre = nombre;
          }
      public Persona(string nombre, string apellidos){
               this.nombre = nombre + " " + apellidos;
          }
   }
```

Sobrecarga

C#

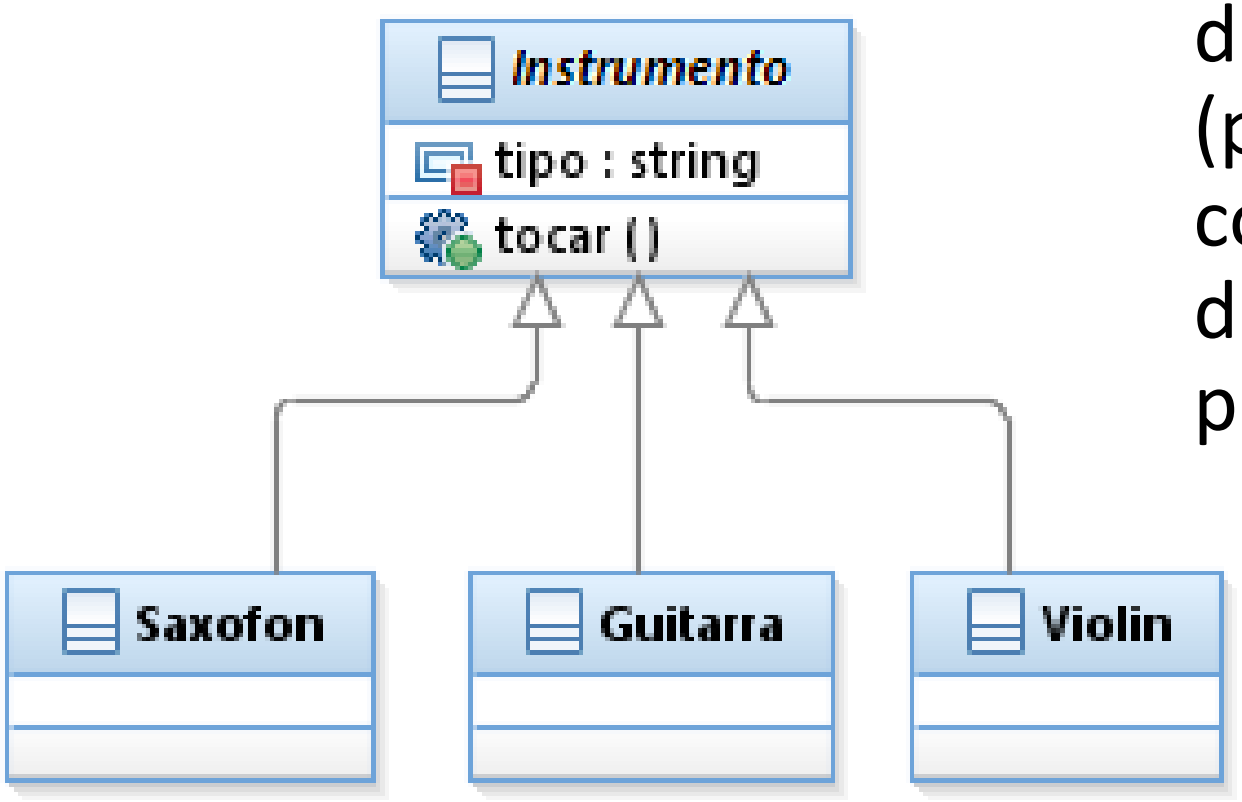
```
public class Persona
  {
    public void setNombre(string nombre){
            this.nombre = nombre;
    }
    public void setNombre(string nombre, string apellidos)
    {
            this.nombre = nombre + apellidos;
    }
}
```

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



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

C#

```
public abstract class SerVivo{

      public abstract void alimentarse();

      public void correr(){
            //Implementación de correr
      }
}
```

Interfaz

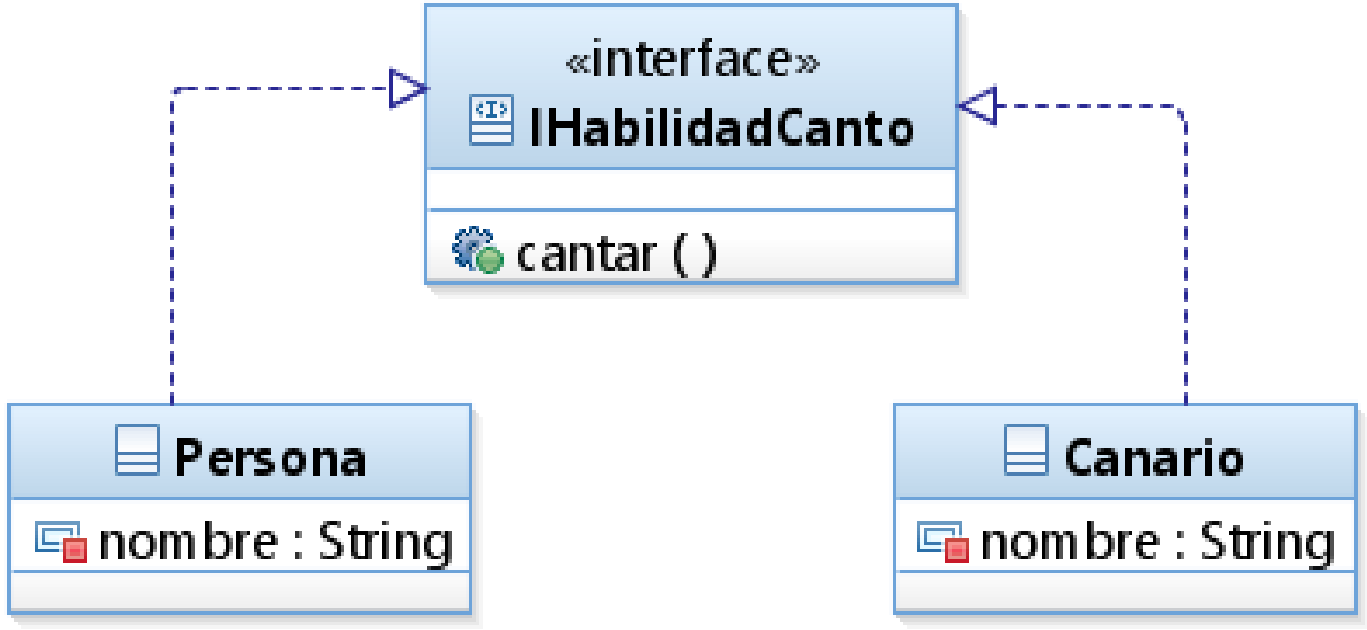
Define el comportamiento de una clase pero no la implementación.

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.

Interfaz

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



Interfaz

C#

```
interface IhabilidadCanto{
     void cantar();
}

public class Persona : IHabilidadCanto{
     public void cantar(){
     System.Console.WriteLine(" la la la la ");}
}
```

Interfaz

```
interface IhabilidadCanto{
     void cantar();
}

public class Canario implements IHabilidadCanto{
     public void cantar(){
     System.out.println(" pio pio pio ");}
}
```

Comparación Clase Abstracta - Interfaz

	Interfaz	Clase abstracta
Declarar métodos abstractos	✓	✓
Implementar Métodos	X	✓
Añadir datos miembros	X	✓
Crear objetos	X	X
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

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

Dias hoyDia = Dias.Lunes;
```

C#

Enumerados

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

Dias d = Dias.Domingo;
```

Arreglos

C#

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

Arreglos

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

Get y Set

Las propiedades deben manejarse de forma privada y se deben establecer métodos para modificar y leer estos valores.

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

Permite trabajar un objeto como si fuese un arreglo.

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]);
     }
}
```

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.