

An abstract graphic on the left side of the slide, consisting of a network of thin, light-blue lines and small circles, resembling a circuit board or a neural network. The lines are vertical and horizontal, with some diagonal connections, and the circles are small and white, scattered along the lines.

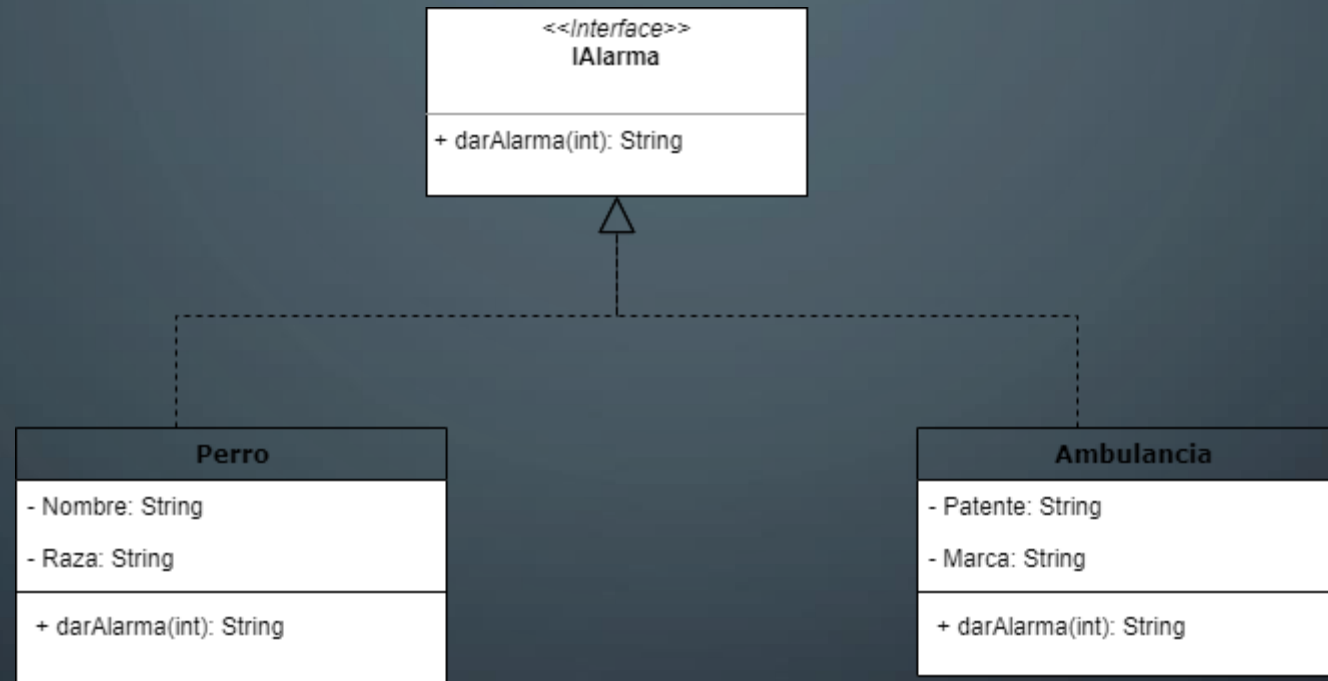
# INTERFACES

# CONCEPTO



- Son entidades similares a las clases (abstracciones)
- Tendrán solamente comportamientos definidos (ya sea solo la firma o la funcionalidad).
- En C# son importantes ya que al no aceptar herencia multiple, nos permiten tomar comportamientos de distintas clases

# VISTA DE DISEÑO



# VISTA DE DESARROLLO

```
interface IAlarma
{
    String darAlarma(int repeticiones);
}
```

```
class Perro : IAlarma
{
    String nombre;
    String raza;

    public Perro(string nombre, string raza)
    {
        this.nombre = nombre;
        this.raza = raza;
    }

    public string darAlarma(int repeticiones)
    {
        String rta = "";
        for (int i=0; i < repeticiones; i++)
        {
            rta += "guau";
        }
        return rta;
    }
}
```

```
class Ambulancia:IAlarma
{
    String patente;
    String marca;

    public Ambulancia(string patente, string marca)
    {
        this.patente = patente;
        this.marca = marca;
    }

    public string darAlarma(int repeticiones) //Definicion implicita
    {
        String rta = "";
        for (int i = 0; i < repeticiones; i++)
        {
            rta += "Ninu ninu";
        }
        return rta;
    }
}
```

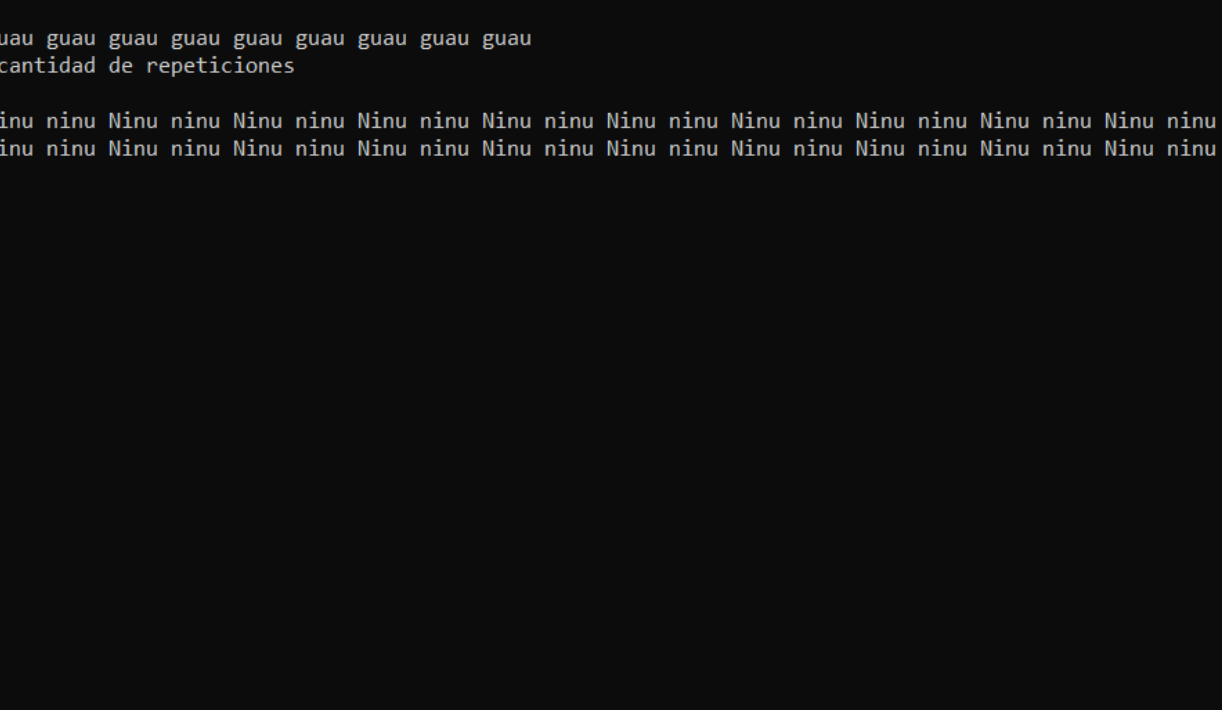
# VISTA DE DESARROLLO

```
class Program
{
    static void Main(string[] args)
    {
        Perro perro = new Perro("boby", "Caniche");
        Ambulancia ambulancia = new Ambulancia("ABC123", "Ford F100");
        int repeticiones = ObtenerRepeticiones();
        MostrarAlarma(perro, repeticiones);
        repeticiones = ObtenerRepeticiones();
        MostrarAlarma(ambulancia, repeticiones);
        Console.ReadKey();
    }

    static int ObtenerRepeticiones()
    {
        Console.WriteLine("Ingrese la cantidad de repeticiones");
        int repeticiones = int.Parse(Console.ReadLine());
        return repeticiones;
    }

    static void MostrarAlarma(IAlarma objeto, int repeticiones)
    {
        String alarma = objeto.darAlarma(repeticiones);
        Console.WriteLine(alarma);
    }
}
```

# EN EJECUCIÓN



C:\Users\Charles\Documents\utn\Programacion III\clases ppt\clase 9 - interfaces\ejemplos interfaces\ejemplos interfaces\bin\Debug\ejempl... [-] [Maximize] [X]

```
Ingrese la cantidad de repeticiones
11
guau guau guau guau guau guau guau guau guau guau
Ingrese la cantidad de repeticiones
23
Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu
Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu Ninu ninu
```

# MÉTODOS ESTÁTICOS EN UNA INTERFAZ

ES VÁLIDO PENSAR EN MÉTODOS  
ESTÁTICOS PARA UNA INTERFAZ?







# MÉTODOS ESTÁTICOS EN UNA INTERFAZ

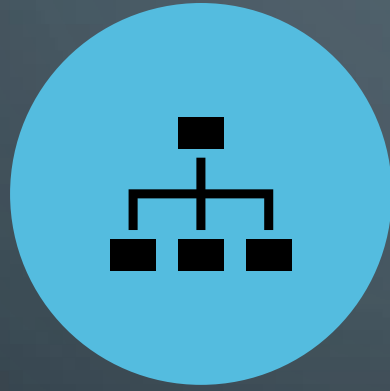
- Los atributos y métodos estáticos se definen a nivel clase para que no sea necesario crear un objeto para acceder a una funcionalidad o dato.
- Las interfaces definen un contrato que las clases deben cumplir.
- Al no haber objetos implicados un static no es aplicable acá.



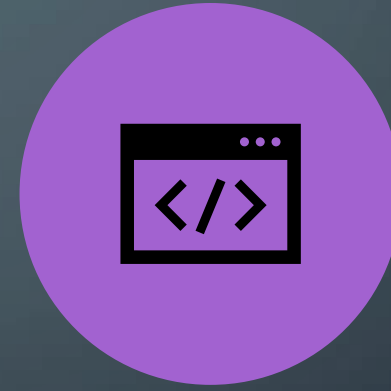
# CONCEPTO DE POLIMORFISMO

- El polimorfismo es la capacidad de invocar al mismo método en dos objetos distintos y que cada uno pueda actuar de forma distinta.
  - Podemos llegar al polimorfismo desde distintas formas.
- Una interfaz nos permite brindar polimorfismo desde clases que nada tienen que ver entre si. Son clases heterogéneas.

# TIPOS DE INTERFACES



INTERFACES STANDARD: SON  
AQUELLAS INTERFACES DEFINIDAS Y  
PROVISTAS POR EL FRAMEWORK .NET



INTERFACES PERSONALIZADAS O  
CONSTRUIDAS POR EL  
PROGRAMADOR.

# INTERFACES STANDARD

- Tenemos varias interfaces que podemos usar. Un ejemplo es la interfaz `Comparable`.
- Esta interfaz declara el método polimórfico `CompareTo`
  - `Int CompareTo(object obj)`
- Este método es el que usa en la clase `Array` para poder hacer el ordenamiento. (método `Sort`)

# USANDO COMPARETO PARA LA CLASE CUENTA

```
public int CompareTo(object obj)
{
    if(obj is Producto) //como aca podemos recibir cualquier cosa hay que asegurarse que sean del mismo tipo
    {
        Producto n = (Producto)obj;
        if (this.PrecioUnitario > n.PrecioUnitario)
        {
            return 1;
        }
        if (this.PrecioUnitario < n.PrecioUnitario)
        {
            return -1;
        }
        return 0;
    }
    else
    {
        return int.MaxValue;
    }
}
```

# EJEMPLO: HAGAMOS LA SIGUIENTE CLASE

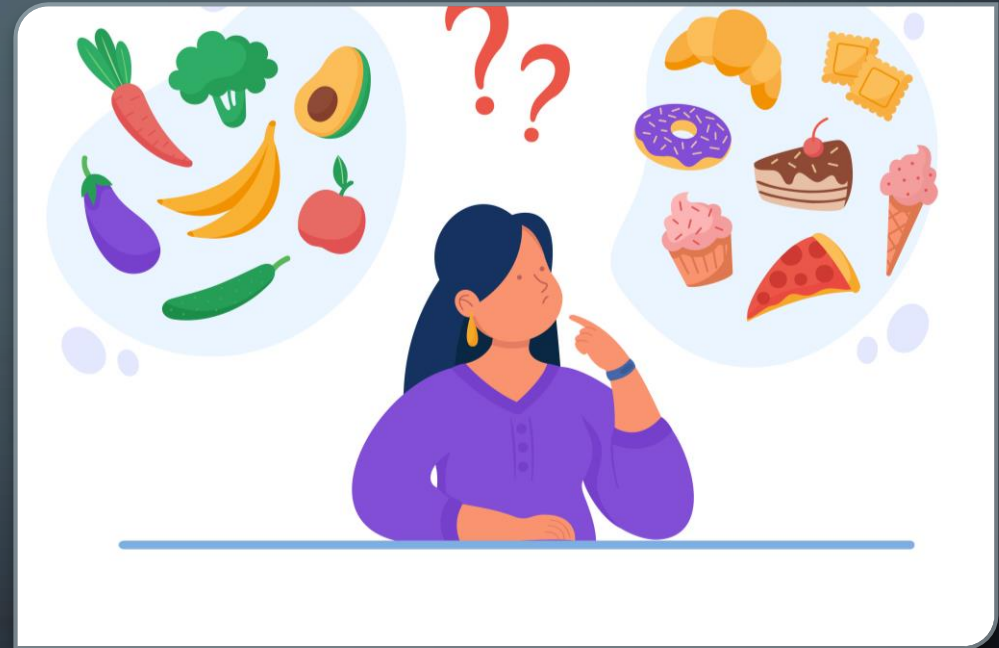
Atleta: IComparable, IEquatable
Numero
Nombre
Apellido
Mejor tiempo
CompareTo
MostrarDatos

# BUENAS PRÁCTICAS EN POO

- Cuando implementamos programación orientada a objetos (o cualquier otro modelo) debemos tener en cuenta la calidad del software.
- Un software de calidad nos permite su reutilización y que sea escalable fácilmente.
- Si cada vez que implementamos un cambio debemos cambiar todo el programa o rompemos algo nuevo no nos sirve

# BUENAS PRÁCTICAS EN POO

- Para lograr un software de calidad tenemos distintas buenas prácticas:
  - Patrones de diseño (como los patrones GoF).
  - Principios.

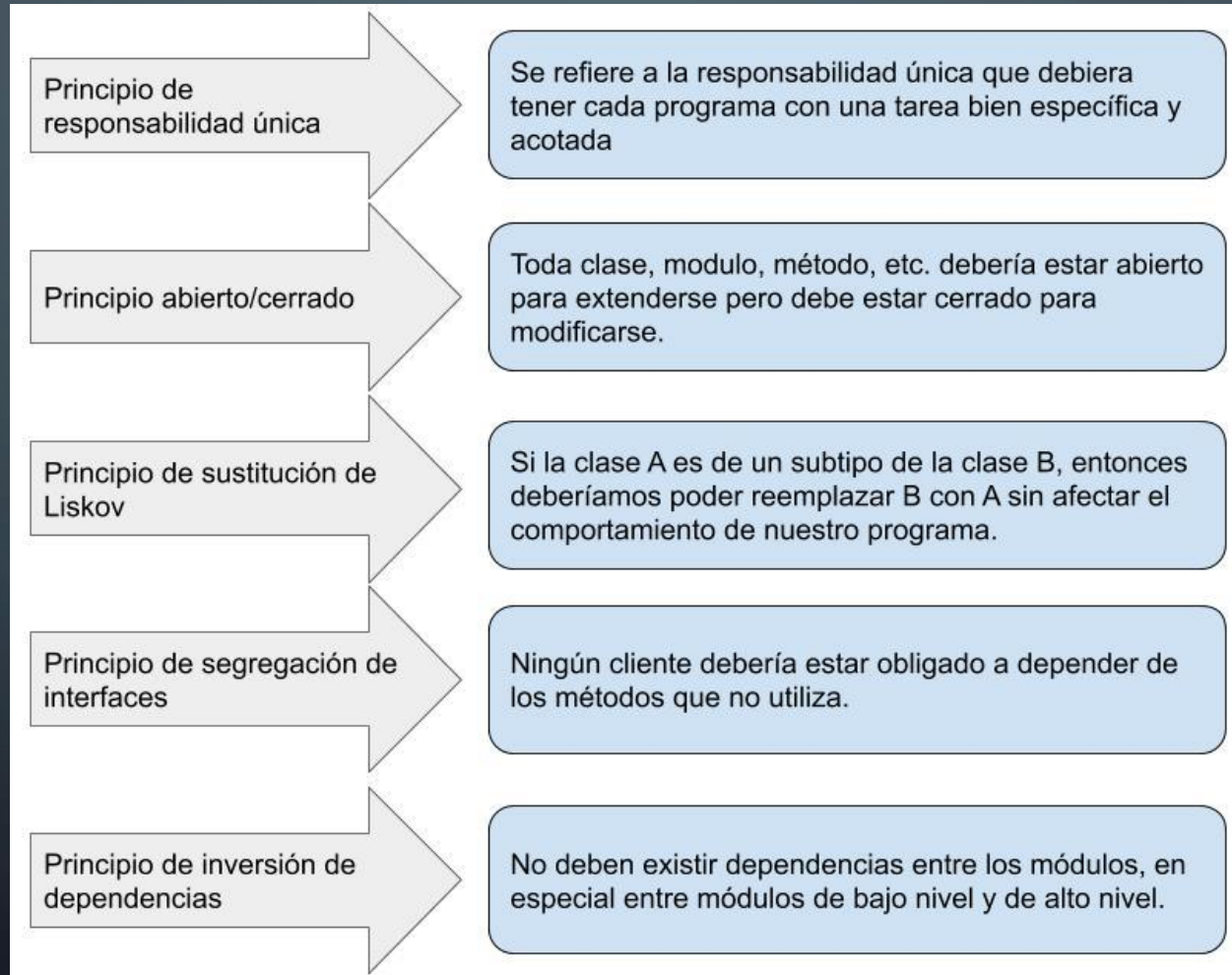




# PRINCIPIOS SOLID

- Son 5 principios que resumen las buenas practicas de diseño de sistemas orientados a objetos.
- Tiene como objetivos:
  - Crear un software eficaz, robusto y estable.
  - Escribir un código limpio y flexible ante los cambios.
  - Permitir la escalabilidad del sistema.

# PRINCIPIOS SOLID



# PRINCIPIO DE SEGREGACIÓN DE INTERFACES

- Una clase puede implementar varias interfaces a la vez.
- Entonces, debemos separar las funcionalidades de una interfaz lo más posible.
- Esto nos permite saber qué exactamente puede implementar cada clase y reutilizar mucho más nuestro código.
- Esta es una buena práctica que se encuentra dentro de los 5 principios SOLID.

# FORMULARIO CLASE 9

- <https://forms.gle/RiYuVgfP9EJRbr2a8>

