POO -> NIVELES DE ACCESO

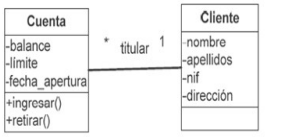
PUBLIC: desde cualquier clase.
PRIVATE: solo desde la misma clase
PROTECTED: desde la misma clase y las clases que heredan de esta.
INTERNAL: acceso desde el mismo Proyecto o componente .
ABSTRACT: solo permite herencia, pero no instancias . (EN UML SE VE CON TIPOGRAFIA CURSIVA. SI LOS METODOS SON ABSTRACT, SE DEBEN SOBREESCRIBIR CON OVERRIDE.)
STATIC: no permite herencia ni instancias.(LAS PROPIEDADES SUBRAYADAS SON ESTATICAS.)
`VIRTUAL:` sirve para sobrescribir métodos si fuese necesario..

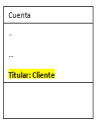
DIAGRAMAS CLASES UML: lenguaje unificado de modelado de clases.

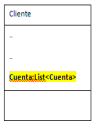
GENERALIZACION / ESPECIALIZACION (herencia)
clase padre (clase generalizaza)
clases hijo (clase especializada)

ASOCIACION: Relación entre objetos o relación de contención.

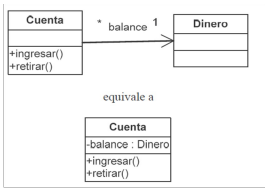
asociación simple







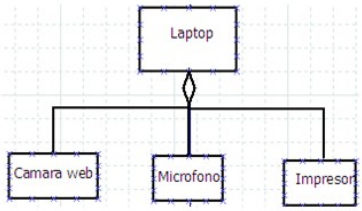
asociacion con dirección: (flecha) (la propiedad relacionada al objeto aparece solo de un lado en este caso en cuenta aparece dinero.)



cardinalidad: 1 a muchos, cero a uno, muchos a muchos etc.

TIPOS DE RELACIONES

AGREGACION: Por ejemplo sabemos que a una computadora portátil se le pueden agregar elementos como micrófono, cámara web e impresora; sin embargo la ausencia de estos elementos no repercute en el funcionamiento básico de dicha portátil. El símbolo de agregación es un diamante vacio colocado en el extremo de la clase que contiene las clases agregadas. En este tipo de asociación, la clase hija puede sobrevivir sin su clase padre(ciclo de vida independiente). Agregación es una relación débil.
(se representa con rombo/diamante sin pintar.)



Ejemplo c# (agregación):

```
public class Address
{
        ...
}

public class Person
   private Address address;
   public Person(Address address)
   {
      this.address = address;
   }

}
```

Person se usaría entonces de la siguiente manera:
Address address = new Address(); address.City="La Plata"; Person person = new Person(address)

COMPOSICIÓN: a diferencia de la agregación representa una clase que está compuesta por otras clases que son indispensables para que esta funcione.(se representa con ROMBO NEGRO en la imagen de abajo)
Composición es una relación fuerte.

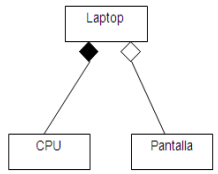


Figura. Representación de composición

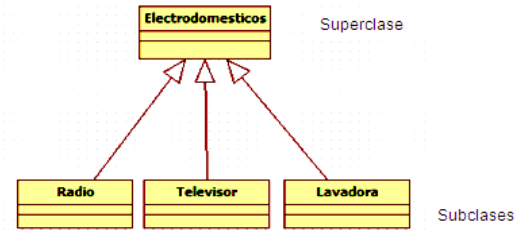
Ejemplo c# (composición)

```
public class Engine
{
   ...
}
public class Car
{
    Engine e = new Engine();
}
```

DEPENDENCIA: Es una relación semántica entre dos elementos en la cual un cambio a un elemento (el elemento independiente) puede afectar a la semántica del otro elemento (elemento dependiente).
Declara que un cambio en la especificación de una clase puede afectar a otra que la utiliza.
Se representa como una línea discontinua que a veces incluye una etiqueta.Suponga que diseñará un sistema que muestre formularios corporativos en pantalla para que los empleados los llenen. El empleado utiliza un menú para seleccionar el formulario por llenar. En su diseño, tiene una clase Sistema y una clase Formulario. Entre sus muchas operaciones, la clase sistema tiene mostrarFormulario(f:Form). El formulario que el sistema desplegará, dependerá, obviamente, del que elija el usuario. La notación del UML para ello es una línea discontinua con una punta de flecha en forma de triangulo sin relleno que apunta a la clase de la que depende, como se muestra en la siguiente

HERENCIA (POO) O GENERALIZACIÓN (UML)
Uno de las características principales de la orientación a objetos es poder asociar a los objetos (clases en UML) con características y operaciones que puedan heredarse entre ellos, siempre y cuando estos estén ligados a un entorno en común,

por ejemplo se no es fácil determinar que un radio, televisor y lavadora tienen atributos y operaciones en común como poseer un interruptor, un cable de energía eléctrica, operaciones de encendido y apagado que los identifican como parte de la superclase Electrodomésticos. La orientación a objetos se refiere a esto como herencia y en UML se denomina generalización la cual está representada por una línea continua que va desde la clase subordinada hasta la clase principal o clase "padre" conectada por un rectángulo vacio, como puede observarse en la figura.



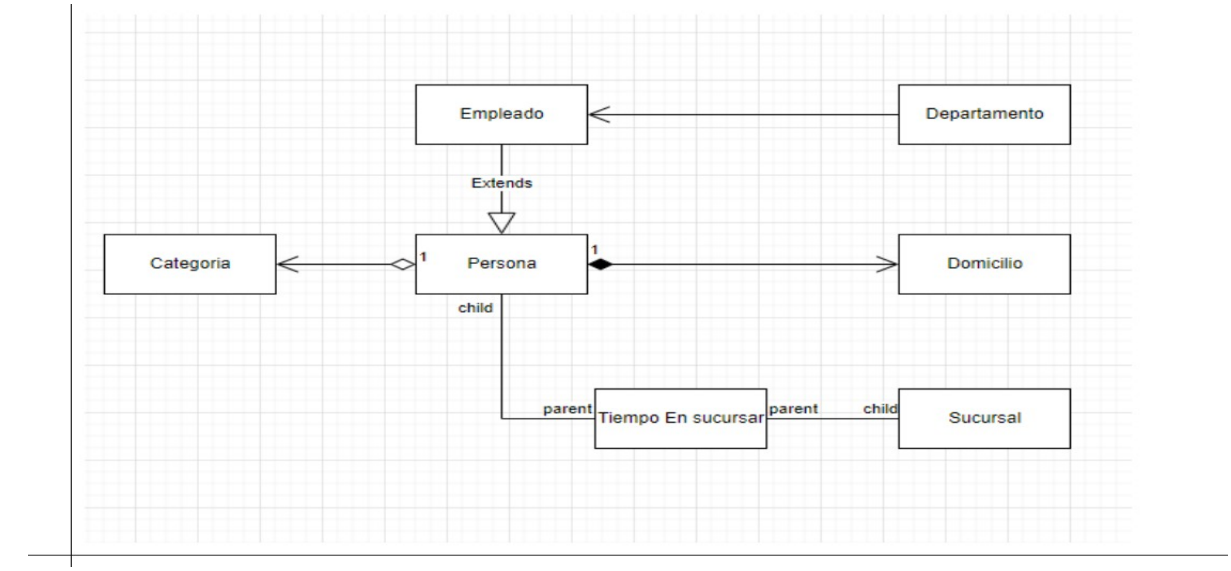
EJEMPLOS pasando C# a UML

```
class Persona
{

    Domicilio dom = new Domicilio();


    Categoria cat;
    public Persona(Categoria c)
    {
        cat = c;
    }

    //Atributos de la clase
    public string Apellido { get; set; }
    public string Nombre { get; set; }
    public int Id { get; set; }


 }
```

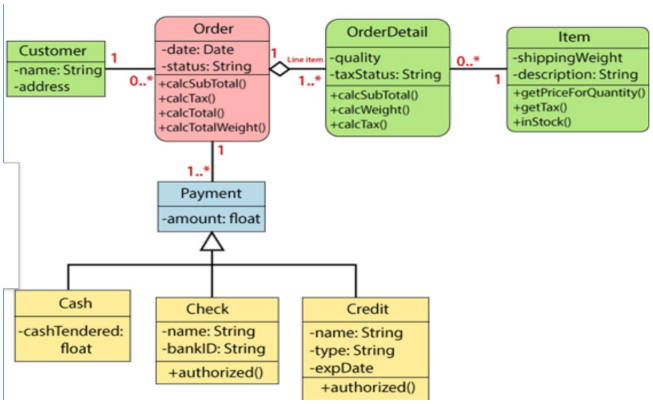
```
public class Empleado:Persona
{
....
 public string Legajo{get; set;}
 public Departamento Departamento {get; set;}
}


 public class Departamento
 {
 ...
 public List<Empleado> Empleado{get; set;}


 }
```

```
class Categoria
{
    public int Id { get; set; }
    public string IdCategoria { get; set; }
}
class Domicilio
{
    public int PersonaId { get; set; }
    public string Calle { get; set; }
    public int Numero { get; set; }
}
```

```
class Sucursal
{
    public int Id { get; set; }
    public string NombreSucursal { get; set; }
}
class TiempoEnSucursal
{
    //Atributos relacionales
    public Sucursal Sucursal { get; set; }
    public Persona Persona { get; set; }

    //Atributos de la clase
    public DateTime FechaInico { get; set; }
    public DateTime FechaFin { get; set; }
}
```



EJERCICIOS RESUELTOS POR PROFE C# (estudiar esto)




```
class Customer{
      public List<Order>Orders{get; set;}
 }

class Order{
      public Order(List<OrderDetail> OrderDetails){
            ...
      }
      public Customer Customer{get; set;}
      public List<Payment>Payments{get; set;}
}

class OrdeDetail {
     public Item Item {get; set;}
}

class Item{
      public List<OrdeDetail>OrdeDetails{get; set;}
}

class Payment{
      public Order Order{get; set;}
 }


 //Generalización
class Cash:Payment {
       …
}
```

DIAGRAMA DE COLABORACION O COMUNICACION: (CAPAS)
BOUNDARY -> CONTROL -> ENTIDAD
boundary: ui interfaz de usuario.
usuario: actor.
control: logica de negocio. (controllers)
entity: base de datos. (modelos)
los boundary, control, bases datos (se escriben todo junto sin dejar espacios)
entity (en singular siempre)
EJEMPLO:

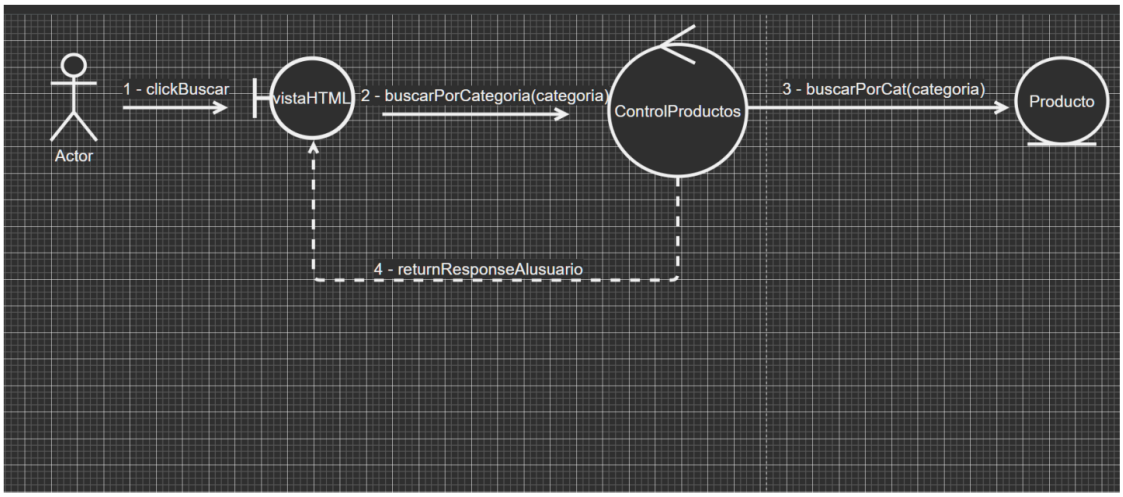
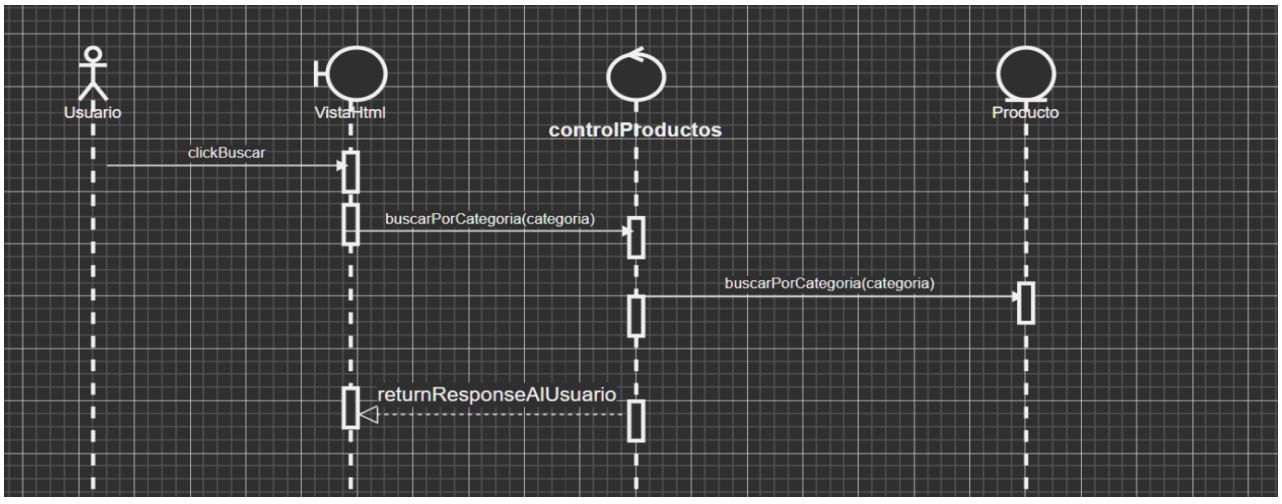
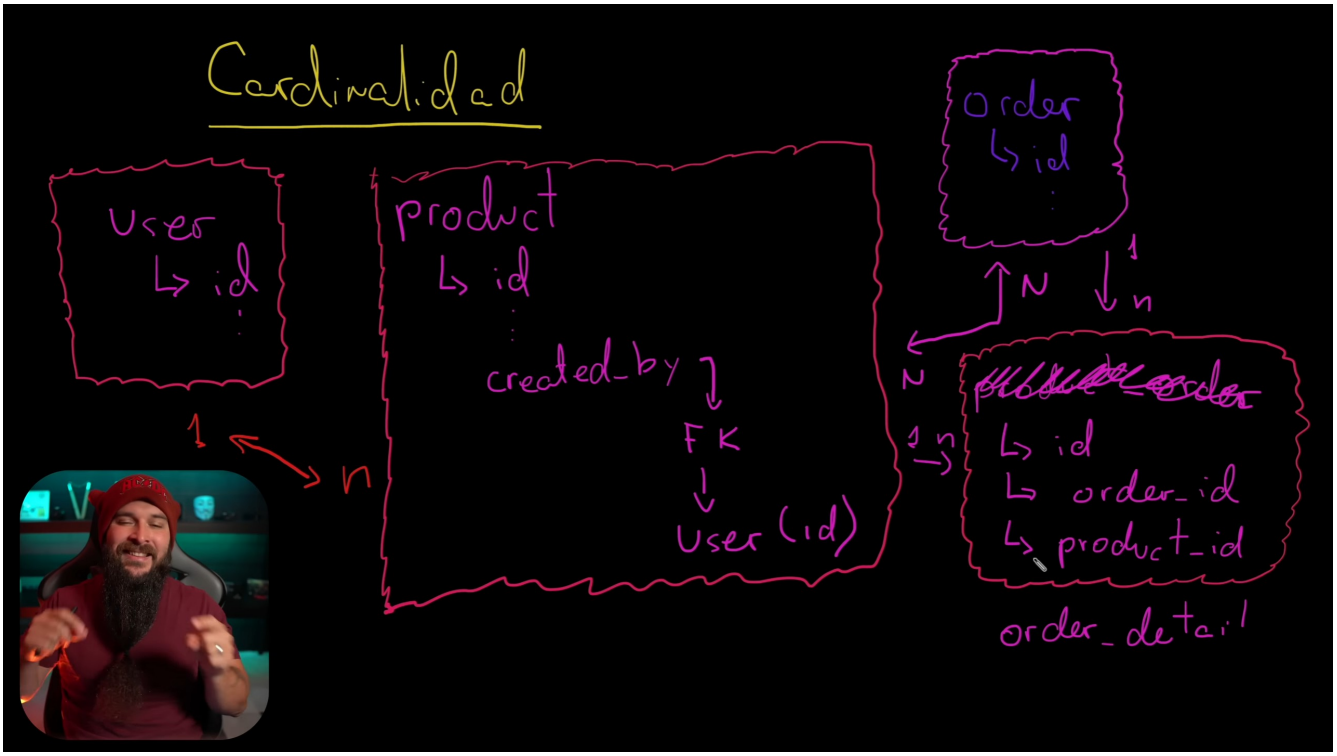
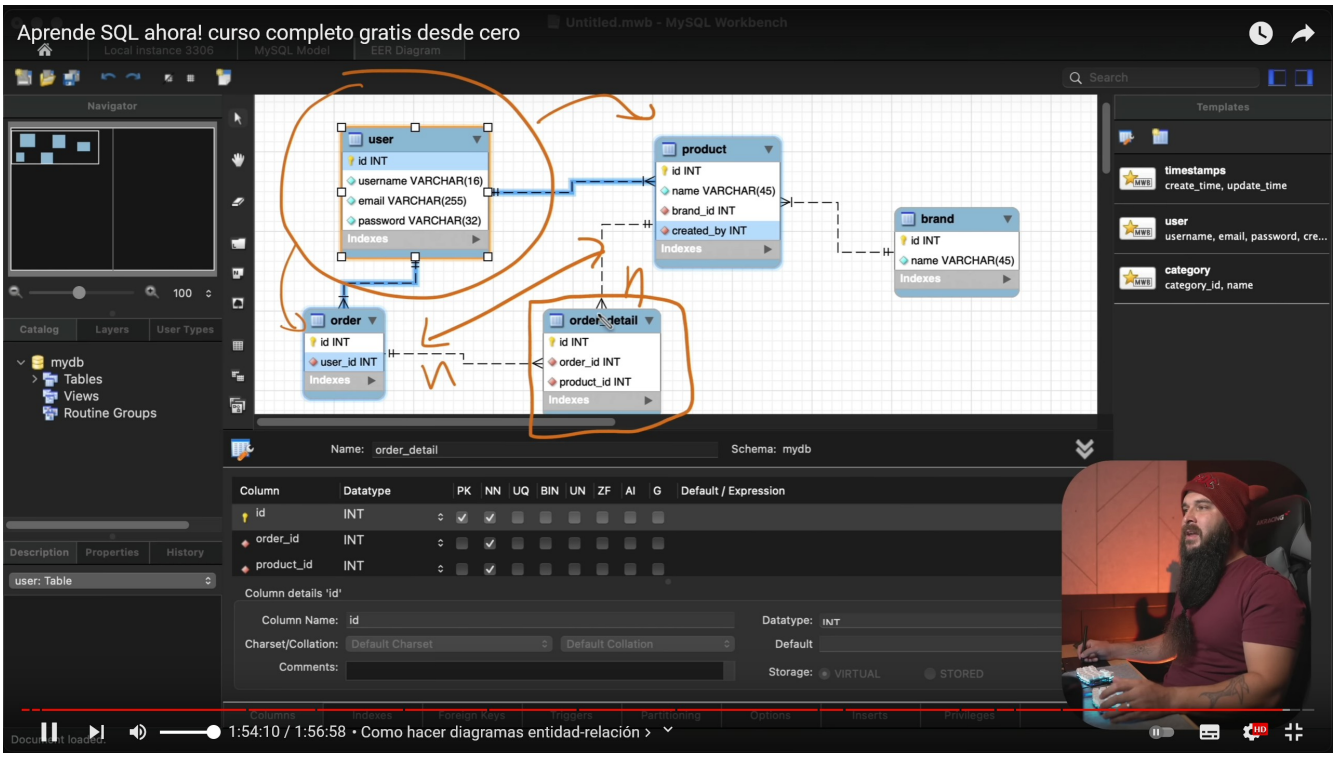


DIAGRAMA DE SECUENCIA:
mensajes: sincronicos o asincrónicos.
EJEMPLO:



NICO SCHURMANN ESXPLICANDO CARDINALIDAD Y DIAGRAMA ENTIDAD RELACION





CLASES VS INTERFACES EN C#

1 – El modificador abstract

Utilizamos el modificador abstract para definir clases o miembros de clases (métodos, propiedades, events, o indexers) para indicar que esos miembros deben ser implementados en las clases que derivan de ellas.

2 – Clases y miembros Abstractos

Cuando declaramos una clase como `abstract` estamos indicado que esa clase va a ser utilizada como clase base de otras clases, ya que ella misma no se puede instanciar.

Una clase abstracta puede contener miembros abstractos como no abstractos, y todos los miembros deben ser implementados en la clase que la implementa.

Para seguir con el ejemplo que vimos en las interfaces en POO utilizaremos el ejemplo de las piezas.

```
public abstract class Pieza
{
    public abstract decimal Area();
    protected abstract decimal Perimetro();
}
```

Como vemos a diferencia de las interfaces en las clases abstractas podemos incluir modificadores de acceso.

Y como hacíamos en las interfaces, en la clase que va a implementar la clase, indicamos dos puntos y la clase.

A diferencia de las interfaces, en las clases que implementan una clase abstracta debemos sobrescribir cada uno de los métodos. Para ello, como vimos en el post de polimorfismo utilizaremos la palabra clave `override` como vemos en el ejemplo, en las interfaces, simplemente implementamos el método, no lo sobreescribimos.

```
public class Cuadrado : Pieza
{
    readonly decimal Lado;
    public Cuadrado(decimal lado)
    {
        Lado = lado;
    }
    public override decimal Area()
    {
        return Lado * Lado;
    }

    public override decimal Perimetro()
    {
        return Lado * 4;
    }
}
```

Y como podemos observar, instanciamos el objeto sin ningún problema:

```
Cuadrado cuadrado = new Cuadrado(3);
```

Y finalmente observamos como podemos incluir miembros no abstractos dentro de la clase abstracta, lo cual definiremos como implementación por defecto:

```
public abstract class Pieza
{
    public abstract decimal Area();
    public abstract decimal Perimetro();
    public bool EjemploMetodo()
    {
        return false;
    }

    public int ValorNatural = 1;
}
```

Estos métodos, no estamos obligados a implementarlos en las clases que implementan la clase abstracta, pero podemos marcarlo como `virtual` para así poder sobrescribir el método.

Como nota especial diremos que el compilador no nos dejará incluir el modificador sealed en las clases abstractas ya que, al no poder heredar de ella no podríamos implementarla y dejaría de tener sentido.

3 - Uso de una clase abstracta en el mundo real

Como hemos explicado las clases abstractas se utilizan como clase base y es ahí donde radica todo su potencial. Ya que nos da la posibilidad de detectar el código común y extraerlo en ella.

Un ejemplo muy claro seria, por ejemplo, si tenemos eventos, distintos tipos de eventos, en los que debemos procesarlos, por ejemplo.

Alquiler de coches y alquiler de motos. Cuando vamos a procesar dichos eventos debemos estar seguros de que, por ejemplo

El coche no está alquilado, así como la moto no esta alquilada, para ello debemos comprobar en sus respectivas bases de datos que no están alquilados. Combinando generics (que veremos lo que son más adelante), junto con las clases abstractas nos queda un código muy potente, pero por ahora veremos un ejemplo más sencillo.

Como en el ejemplo anterior trabajaremos con figuras geométricas ya que es un ejemplo muy fácil de entender.

Como sabemos, podemos calcular el área de un triángulo utilizando su hipotenusa y uno de sus lados. Y esto es así con todos los diferentes tipos de triángulo. Por lo que ese método es común y deberemos ponerlo dentro de la clase abstracta.

```
public abstract class TrianguloBase
{
    public abstract decimal Perimetro();

    public double CalcularAreaConHipotenusa(int lado, int hipotenusa)
    {
        double ladob = Math.Sqrt(Math.Pow(hipotenusa, 2) - Math.Pow(lado, 2));
        return lado * ladob / 2;
    }
}
```

Y cuando definimos cualquier otro triángulo no vamos a definir ese método en concreto.

```
public class Escaleno : TrianguloBase
{
    public override decimal Perimetro()
    {
        throw new NotImplementedException();
    }
}
public class Acutangulo : TrianguloBase
{
    public override decimal Perimetro()
    {
        throw new NotImplementedException();
    }
}
```

En cambio podemos acceder a el sin problema cuando hemos inicializado las variables

```
Escaleno escalenno = new Escaleno();
Acutangulo acutangulo = new Acutangulo();

escalenno.CalcularAreaConHipotenusa(1, 5);
acutangulo.CalcularAreaConHipotenusa(1, 7);
```

4 - Clase abstracta vs interfaz en C#

Como has podido observar el uso de las clases abstractas es muy similar al de las interfaces pese a ello utilizamos unas u otras dependiendo del contexto, ya que su uso se puede solapar.

1. La principal diferencia que podemos observar es que en las clases abstractas podemos indicar el modificador de acceso, mientras que en las interfaces no podemos modificarlo.
2. Cuando utilizamos interfaces, podemos implementar múltiples interfaces. Mientras que solo podemos utilizar una clase abstracta como base.
3. En una clase abstracta, al poder incluir miembros no abstractos podemos incluir una implementación por defecto. Desde C#8 podemos crear elementos por defecto.
4. En una clase abstracta podemos incluir un constructor, mientras que en una interfaz no.

1 - Qué es una interfaz en programación orientada a objetos

Una interfaz es un contrato entre dos entidades, esto quiere decir que una interfaz provee un servicio a una clase consumidora. Por ende, la interfaz solo nos muestra la declaración de los métodos que esta posee, no su implementación, permitiendo así su encapsulamiento.

Aunque esta regla de utilizar únicamente la cabecera del método en la interfaz puede verse afectada en C#8, como ya vimos en el avance, ya que va a permitir añadir cuerpo en la declaración en las interfaces, pero por ahora, no se puede

Una interfaz se define utilizando la palabra reservada "interfaz". Y por norma general, indicamos en el nombre que es una interfaz haciendo empezar el nombre de la misma por la letra I mayúscula.

```
public interface IPieza
{
}
```

Las interfaces pueden contener los siguientes miembros:

- Métodos
- Propiedades
- Indexers
- Eventos

```
public interface IPieza
{
    decimal Area();
    decimal Perimetro();
}
```

Como vemos en el ejemplo, hemos declarado dos métodos.

2 – Implementación de una interfaz

Para implementar una interfaz, debemos declarar una clase o una strutura(struct) que herede de la interfaz, y entonces, implementar todos sus miembros.

Por ejemplo, creamos la clase cuadrado y heredamos de una interfaz, por lo que debemos implementar sus métodos. Por defecto lo podemos dejar en NotImplementedException()

```
public class Cuadrado : IPieza
{
    public decimal Lado { get; private set; }

    public Cuadrado(decimal lado)
    {
        Lado = lado;
    }

    public decimal Area()
    {
        return Lado * Lado;
    }

    public decimal Perimetro()
    {
        return Lado * 4;
    }
}
```

Cuando implementamos una interfaz debemos de asegurar un par de puntos:

- Los métodos y tipos que devuelven deben coincidir tanto en la interfaz como en la clase.
- Deben ser los mismos parámetros
- Los métodos de la interfaz deben ser públicos
- La utilización de interfaces mejora el código y el rendimiento de la aplicación.

3 – Múltiples clases heredan de una interfaz

Uno de los casos más comunes en programación orientada a objetos es que tenemos varias clases que heredan de una interfaz, lo que implica que en ambas debemos implementar el código. Para ello creamos en el código la clase triangulo (rectángulo) conjunto a la anterior de cuadrado, ambas son piezas. Por lo que ambas heredan de IPieza.

```
public class TrianguloRectangulo : IPieza
{
    public decimal LadoA { get; set; }
    public decimal LadoB { get; set; }
    public decimal Hipotenusa { get; set; }
    public TrianguloRectangulo(decimal ladoa, decimal ladob)
    {
        LadoA = ladoa;
```

```
        LadoB = ladob;
        Hipotenusa = CalculateHipotenusa(ladoa, ladob);
    }

    private decimal CalculateHipotenusa(decimal ladoa, decimal ladob)
    {
        return  Convert.ToDecimal(Math.Sqrt((double)(ladoa * ladoa + ladob * ladob)));

    }

    public decimal Area()
    {
        return LadoA * LadoB / 2;
    }

    public decimal Perimetro()
    {
        return LadoA + LadoB + Hipotenusa;
    }
}
```

Como podemos observar en ambos casos tenemos el área y el perímetro.

Si recordamos de la herencia, las clases son también del tipo del que heredan, lo que quiere decir que ambas clases, tanto Cuadrado, como TrianguloRectangulo, las podemos convertir en IPieza:

```
IPieza cuadrado = new Cuadrado(5);
IPieza trianguloRectangulo = new TrianguloRectangulo(5, 3);

Console.WriteLine($"El área del cuadrado es {cuadrado.Area()}");
Console.WriteLine($"El perímetro del cuadrado es {cuadrado.Perimetro()}");

Console.WriteLine($"El área del triángulo es {trianguloRectangulo.Area()}");
Console.WriteLine($"El perímetro del triángulo es {trianguloRectangulo.Perimetro()}");


//Resultado :
El área del cuadrado es 25
El perímetro del cuadrado es 20
El área del triángulo es 7.5
El perímetro del triángulo es 13.8309518948453
```

Este ejemplo es muy sencillito, pero imaginarios una lista con más de un millón de piezas, y cada una con diferentes formas, al heredar todas de IPieza, nos permite tener un bucle con únicamente una línea, un bucle foreach que ejecute el método Area() si lo que queremos es mostrar el área.

El uso de interfaces facilita enormemente programar cuando pasamos entidades reales a código.

Por supuesto se pueden tener métodos adicionales, normalmente privados, dentro de la clase, pero ellos no serán accesibles a través de la interfaz.

4 – Múltiples interfaces para en una sola clase

Puede darse el caso de que necesitemos implementar mas de una interfaz en nuestra clase, para ello no hay ningún problema simplemente las separamos con comas.

Pero esas interfaces pueden contener un método con el mismo nombre el mismo valor de retorno y los mismos parámetros.

La forma que tendremos entonces para diferenciar uno del otro es, en la implementación dentro de la clase

```
public interface interfaz1
{
    void MetodoRepetido();
}
public interface interfaz2
{
    void MetodoRepetido();
}

public class EjemploRepeticionMetodo : interfaz1, interfaz2
{
    public EjemploRepeticionMetodo()
    {
    }

    void interfaz1.MetodoRepetido()
    {
        throw new NotImplementedException();
    }

    void interfaz2.MetodoRepetido()
    {
        throw new NotImplementedException();
```

```
        }
}
```

Como podemos observar no utilizamos un modificador de acceso en la implementación del método.