

```
FileError: resource 'https://fonts.googleapis.com/css?family=Catamaran' gave this Err  
Error: getaddrinfo ENOTFOUND fonts.googleapis.com  
in input on line 13, column 1:  
12  
13 @import url(https://fonts.googleapis.com/css?family=Catamaran);  
14
```

Ejercicios Redefinición de Operadores e Indizadores

[Descargar estos ejercicios](#)

Índice


1. [Ejercicio 1](#)
2. [Ejercicio 2](#)
3.  [Ejercicio 3](#)
4. [Ejercicio 4](#)
5.  [Ejercicio 5](#)
6.  [Ejercicio 6](#)

Ejercicio 1

Para comenzar con la redefinición de operadores, vamos a construir dos clases: la clase **Euro** y la clase **Peseta** (la peseta era la antigua moneda oficial de España antes de ser reemplazada por el Euro). Ambas clases tendrán el atributo necesario para guardar el valor de la moneda, constructor y anulación del ToString para hacerla funcional.

Por otro lado deberás **redefinir los operadores** necesarios para que los objetos de estas clases se puedan **sumar**, **restar**, **comparar** (operadores `==`, `!=` e invalidación de `Equals` de la clase `object`), **incrementar** con `++` y **decrementar** con `--` como si fueran tipos numéricos, teniendo presente que **1 Euro + 166.386 pesetas = 2 euros**.

Además deberás redefinir el operador de **cast implícito** para que ambos tipos sean compatibles entre sí y también con el tipo `double`.

 **Nota:** este ejercicio es casi idéntico al caso de estudio que hay en los apuntes, ante cualquier duda podéis recurrir a el para ayudaros.

Ejercicio 2

A partir del ejercicio de **Cuenta Bancaria que hicimos en temas anteriores**, vas a redefinir como mínimo todos los operadores de comparación.

📌 **Nota:** Para ver si dos cuentas son iguales tendrás que ver además del titular y saldo, si los números de cuenta son iguales (por lo que tendrás que redefinir, también, en numero de cuenta los operadores de comparación).

✓ Ejercicio 3

Para cierta implementación que no viene al caso, el departamento de diseño ha detectado la necesidad de crear un nuevo tipo de números a los que ha denominado "números curiosos". Un número curioso se caracteriza por tres coordenadas reales

$$(a, b, c) \text{ que verifican } a^2 + b^2 + c^2 = 1,$$

salvo en el caso del número "cero" cuyas coordenadas son (0, 0, 0)

Sobre los números curiosos interesa realizar las siguientes operaciones:

- **Suma:**

$$(a_1, b_1, c_1) + (a_2, b_2, c_2) = \left(\frac{a_1 + a_2}{\sqrt{(b_1 + b_2)^2 + (c_1 + c_2)^2 + (a_1 + a_2)^2}}, \right. \\ \left(\frac{b_1 + b_2}{\sqrt{(b_1 + b_2)^2 + (c_1 + c_2)^2 + (a_1 + a_2)^2}}, \right. \\ \left. \left(\frac{c_1 + c_2}{\sqrt{(b_1 + b_2)^2 + (c_1 + c_2)^2 + (a_1 + a_2)^2}} \right) \right)$$

cuando $(a_1 + a_2)^2 + (b_1 + b_2)^2 + (c_1 + c_2)^2 \neq 0$ y (0, 0, 0) en caso contrario

- **Resta:**

$$(a_1, b_1, c_1) - (a_2, b_2, c_2) = \left(\frac{a_1 - a_2}{\sqrt{(b_1 - b_2)^2 + (c_1 - c_2)^2 + (a_1 - a_2)^2}}, \right. \\ \left(\frac{b_1 - b_2}{\sqrt{(b_1 - b_2)^2 + (c_1 - c_2)^2 + (a_1 - a_2)^2}}, \right. \\ \left. \left(\frac{c_1 - c_2}{\sqrt{(b_1 - b_2)^2 + (c_1 - c_2)^2 + (a_1 - a_2)^2}} \right) \right)$$

cuando $(a_1 - a_2)^2 + (b_1 - b_2)^2 + (c_1 - c_2)^2 \neq 0$ y (0, 0, 0) en caso contrario

Crea la clase **NumeroCurioso** con los atributos, propiedades y métodos que creas necesarios para su correcta implementación y prueba.

Redefine los operadores necesarios para poder realizar la **suma y resta** de dos números curiosos.

Ejercicio 4

Crea un sencillo programa con una clase **DiaLaborable** que tendrá un tipo enumerado **DiaSemana** (lunes, martes, ..., domingo). Además deberemos implementar un indizador indexado por un entero, que devuelva el **DiaSemana** correspondiente al índice (el lunes será el 1, el martes el 2, etc). El indizador también deberá controlar si el índice introducido no está entre 1 y 7 o si es 6 ó 7 (estos se consideran no laborables), en esos casos deberá lanzar una excepción del tipo **DiaNoValidoException** indicando cual es el problema. Las siguientes líneas serán el código de la main, para que pruebes con el tu código.

```
static void Main()
{
    DiaLaborable dia = new DiaLaborable();
    try
    {
        Console.WriteLine(dia[1]);
        Console.WriteLine(dia[4]);
        Console.WriteLine(dia[9]);
        Console.WriteLine(dia[7]);
    }
    catch (DiaLaborable.DiaNoValidoException e)
    {
        Console.WriteLine(e.Message);
    }
}
```

✓ Ejercicio 5

Necesitamos una clase para almacenar los datos de una factura. Para ello deberemos implementar la clase **Factura**:

- Cada factura tendrá un **Cliente** con nombre, teléfono, dirección, población, provincia, código postal y DNI.
- La factura puede tener una o varias **líneas de Detalle**, cada una de ellas estará constituida por la siguiente información: cantidad (número de artículos vendidos del mismo tipo), descripción (nombre del artículo), precio unitario y por último importe total que será una propiedad calculada (cantidad * precio).
- Además la factura también deberá ofrecer las **propiedades públicas y calculadas** que devuelvan la base imponible, la cuota de IVA y el total a pagar, además de la propiedad privada y de solo lectura porcentaje de IVA.
- La Factura poseerá un **indizador** para gestionar la asignación de los elementos al array, para su posterior uso en la redefinición del operador **+**. Este indizador se indizará mediante un entero y será de tipo Detalle. El indizador se encargará de dar acceso lectura/escritura al elemento del array detalle indicado en la indización, controlando que la posición no pueda ser menor que 0 ni mayor que la cantidad de líneas que lleve la factura en el momento, lanzando excepción en caso de ser necesario.
- Por último se redefinirá el operador **+**, para sumar una Factura con un Detalle **usando el indizador** ver **Nota**. Como puedes imaginar, se tendrá que redimensionar cada vez que se añada un nuevo detalle.

Para terminar escribe el código necesario para testear las características de una factura completa. No olvides anular el ToString en las clases para que proporcionen la salida lo más parecida a una factura.

📌 **Nota:** Supongo que ya habrás deducido que para que la clase Factura cumpla los requisitos que se piden, tendrás que construir también una clase Detalle. Pues bien, también debes sobrecargar el **operador +** para que se puedan sumar objetos de la clase Detalle a objetos de la clase Factura (esta será la manera de añadir líneas de compra al objeto factura y **deberá usar el indizador para ello**).

Ojo 👁, en este caso solamente queremos hacer posible la suma **Factura + Detalle** nada más.

✓ Ejercicio 6

Vamos a encapsular en una clase denominada **Presupuestos** una matriz cruzada de presupuestos donde las filas serán los conceptos: **Comida**, **Casa** y **Entretenimiento** (enumeración). Y las columnas se corresponderán con los 12 meses del año (enumeración). De tal manera que el contenido de una celda me indicará el presupuesto de un concepto para un determinado mes.

El constructor de la clase inicializará la matriz de manera que para comida tendremos un presupuesto de 300 €, para la casa de **500 €** y para entretenimiento de **200 €**, siendo igual para todos los meses del año (usaremos for para la inicialización).

	1	2	3	4	5	6	7	8	9	10	11	12
Comida	300	300	300	300	300	300	300	300	300	300	300	300
Casa	500	500	500	500	500	500	500	500	500	500	500	500
Entretenimiento	200	200	200	200	200	200	200	200	200	200	200	200


Tendremos que crear tres indizadores distintos para acceder al presupuesto correspondiente a la celda, estos serán:

- Indexación por dos enteros.
- Indexación por dos string (concepto y mes).
- Indexación por dos enumerados (Concepto y Mes)

También habrá un método **PresupuestoPor** al que le llegará un Concepto y devolverá una cadena con el presupuesto de todos los meses, asociado a ese concepto. Por ejemplo para el concepto *Casa* devolverá la siguiente cadena.

	1	2	3	4	5	6	7	8	9	10	11	12
Casa	500	500	500	500	500	500	500	500	500	500	500	500

Además, la clase presupuesto redefinirá el método **ToString()** para devolver una cadena que muestre los presupuestos por meses en un primer nivel, y en un segundo nivel por conceptos. En la clase program deberemos crear los métodos necesarios para: mostrar el presupuesto según Concepto, mostrar todos los presupuestos, modificar presupuesto al realizar un gasto (pidiendo al usuario mes y concepto). Controlar mediante excepciones personalizadas, que el gasto no sobrepasa el presupuesto, y que los meses y conceptos pertenecen a la enumeración.

 **Nota:** Deber usar la implementación de estos indizadores para la inicialización en el Constructor (utilizando `this`) y para el recorrido en el método `ToString()` , además de para el acceso desde la clase `Program` .