

# Módulo 2

## OOP o el lenguaje C# en serio

Tecnara – Cursos de formación



## Índice de contenidos

### 1. Parada para reflexionar

#### 1. Cambio de tercio

### 2. Programación orientada a objetos

1. Conceptos generales: ¿Cómo organizar el universo?
2. Clases y objetos
3. Métodos
4. Modificador static, namespace



01

# Parada para reflexionar

# Parada para reflexionar

## Programa 13 versión 1: Gestión de cuenta bancaria para un solo usuario

- El programa mostrará un menú con las siguientes opciones:
  - 1- Ingreso de efectivo
  - 2- Retirada de efectivo
  - 3- Lista de todos los movimientos
  - 4- Lista de todos los ingresos de efectivo
  - 5- Lista de todas las retiradas de efectivo
  - 6- Mostrar saldo actual
  - 7- Salir.
- Existirá una variable que guardará el saldo de un único cliente y sobre ese saldo se realizarán los ingresos (sumas de dinero al saldo actual) o retiradas (restas de dinero al saldo actual).
- Una vez finalizada cualquier operación, el programa preguntará si queremos realizar alguna otra operación:
  - Si usuario responde que sí, se volverá a mostrar el menú, esperando que el usuario elija otra opción
  - en caso contrario mostrará el valor actual de saldo y finalizará el programa.

# Parada para reflexionar

Programa 13 versión 1: Gestión de cuenta bancaria para un solo usuario

- Hemos conseguido que funcione correctamente, pero no estamos contentos con algunas cosas:
  - Estructura monolítica (todo el código junto). Nos gustaría separar el código en distintos trozos para:
    - Facilitar su comprensión
    - Permitir reutilizar trozos de código que ahora está repetido



# Parada para reflexionar

## Programa 13 versión 2: Gestión de cuenta bancaria multiusuario

- Cuando nos planteamos ampliar el enunciado para que no se limite a operar con el saldo de un único cliente, sino que permita realizar operaciones a distintos clientes (cada operación aplicada solo al saldo del cliente correspondiente, claro está), la cosa se complica:

`List<decimal> SaldosPorCliente`

`List<List<decimal>> ListasIngresosPorCliente // si se ha hecho con listas separadas`

`List<List<decimal>> ListasRetiradasPorCliente // si se ha hecho con listas separadas`

`List<List<decimal>> ListasMovimientosPorCliente`

- Y además, hay que identificar a cada cliente antes de hacer cualquier operación:

`List<string> NumerosDeCuentaPorCliente // string porque como entero es demasiado grande`

`List<int> NumerosPinPorCliente // solo el pin no vale, porque puede haberlos repetidos`

# Parada para reflexionar

## Programa 13 versión 2: Gestión de cuenta bancaria multiusuario

- Con las listas anteriores (algunas, listas de listas) se puede hacer, pero resulta contra-intuitivo:
  - No parece natural tener que consultar listas diferentes para obtener valores de elementos asociados a un mismo cliente
  - Lo natural sería:
    - Cada cliente distinto tendría que tener asociada una variable distinta, pero una sola
    - Todo lo que corresponde a un mismo cliente debería estar accesible desde esa única variable asociada

# Parada para reflexionar

Conclusión a la que pueden llegar algunos en este punto:

“Esto de la programación es una  ”

- Hace muchos años, era la única forma de programar
- Pero en algún momento, todo cambió... a mejor (para el que lo entiende...)
  - En este tema, veremos cómo solucionar aquello con lo que ahora no estamos a gusto: código monolítico, información de un mismo elemento del mundo en variables distintas
  - ... y alguna mejora más que nos servirá para otros programas...



# Cambio de tercio

Lo que hemos visto hasta ahora ha sido solo la pantalla de aprendizaje del videojuego...

- Solo se ha hablado del código del interior de Main (y como mucho del using...)
- Todo el programa se ha escrito en Main, y es largo y tedioso de leer y entender

# Cambio de tercio

Lo anterior ha pretendido ser un campo de juego simplificado (sandbox) donde:

- Poder aprender algunos elementos básicos del lenguaje
  - Entrada/Salida por consola (petición de datos al usuario, escritura en pantalla)
  - Declaración, inicialización, lectura y asignación (escritura) de variables
  - Operaciones con variables de tipos simples, y dos tipos indexados: Array y List
  - Control de flujo condicional (ejecutar solo si...) e iterativo (ejecutar hasta que...)

# Cambio de tercio

Lo anterior ha pretendido ser un campo de juego simplificado (sandbox) donde:

- Poder jugar con esos elementos creando programas sencillos para
  - Familiarizarse con la sintaxis y la detección de errores de compilación y de ejecución
  - Comprobar que lo que se escribe y lo que se ejecuta coinciden (programar tiene sentido)
  - Aprender a usar los breakpoints para 'sentir' que tenemos el control de lo que se va a ejecutar, y que podemos conocer el valor de las variables en cualquier punto de nuestro programa si lo necesitamos

# Cambio de tercio

Ahora vamos a pasar de pantalla...





02

# Programación orientada a objetos

# Conceptos generales

Paradigma de Programación Orientada a Objetos (Object-Oriented Programming, OOP)

C# es un lenguaje orientado a objetos. ¿Qué significa esto?

- Todo lo que se escribe en C#:
  - O es una clase
  - O es una propiedad definida dentro de una clase
  - O es un método definido dentro de una clase (lo único que hemos escrito hasta ahora)

# Conceptos generales

- Bueno... casi todo. Vaaale, se pueden escribir cosas fuera de claaases:
  - Declaración de namespaces (sea lo que sea)
  - Sentencias 'using' (sea lo que sea)
  - Declaración de tipos enum (sea lo que sea)
  - Declaración de interfaces (sea lo que sea)
  - Declaración de delegados (sea lo que sea)
- Pero todo esto lo veremos más adelante. De momento, pensemos que todo son clases...



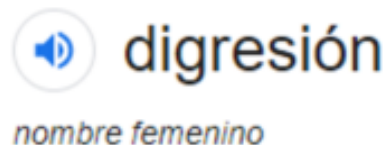
# Conceptos generales

Paradigma de Programación Orientada a Objetos (OOP)

Maravilloso, todo son clases, lo captamos... ¿Pero se puede saber QUÉ ES UNA CLASE?

Bien, para entender este concepto, va a dar comienzo la explicación más alejada del mundo de la programación que veremos en todo el curso.

Permitidme una pequeña digresión filosófica (o sea, dejadme que me vaya un rato por los cerros de Úbeda)...



1. Hecho de apartarse en un relato, discurso o exposición del asunto principal para tratar de algo que surge relacionado con él.  
"el cuento, a diferencia de la novela, exige una mayor precisión y no admite digresiones"
2. Parte de un discurso, exposición, etc., que no tiene relación directa con el asunto principal que se está tratando.  
"la digresión de lo que se entiende por estilo ocupa veinte páginas"



# Conceptos generales: ciencia

Las ciencias naturales (física, química, geología, biología, etc) se dedican a describir cómo es la naturaleza

- La observan para descubrir patrones (comportamientos que se repiten siempre bajo una mismas condiciones iniciales)
- analizan esos patrones para tratar de modelarlos (representarlos o describirlos mediante recursos matemáticos):
  - fórmulas que describen comportamientos cambiantes
  - topologías que describen una distribución en el espacio

# Conceptos generales: ciencia

Las ciencias naturales (física, química, geología, biología, etc) se dedican a describir cómo es la naturaleza

- Por difícil e importante que sea ese trabajo para el progreso de la Humanidad, 'lo único' que hacen es tratar de descubrir, modelar con lenguaje matemático y describir en detalle los comportamientos que tienen las cosas que existen
- Pero en ningún momento se plantean los científicos preguntas como
  - ¿Por qué las cosas se comportan así y no de otro modo?
  - ¿Por qué podemos conocer y describir los comportamientos de las cosas?
  - ¿Qué son exactamente las cosas que sabemos que existen en el mundo?
  - ¿Qué significa que sabemos que existen cosas en el mundo?

# Conceptos generales: Física

- En la época de Aristóteles, filósofo griego, no existía el concepto de ciencia, pero sí la ocupación de observar la naturaleza y buscar leyes matemáticas que la describan
- Esta ocupación o disciplina dedicada a estudiar y describir la naturaleza se llamaba Física (de Physis, 'Naturaleza' en griego).
- Aristóteles distinguió entre la descripción de la naturaleza (Física) y el planteamiento de preguntas sobre:
  - por qué existimos nosotros
  - por qué existe la naturaleza
  - por qué estamos nosotros en la naturaleza
  - por qué podemos conocer y utilizar la naturaleza en nuestro beneficio

# Conceptos generales: Metafísica

A esta nueva disciplina de pensamiento, que no se contenta con entender la naturaleza tal y como es, sino que busca ir más allá y preguntarse por qué es así, Aristóteles la llamó Metafísica

→ meta = más allá, después de; física = descripción de la naturaleza

Esta disciplina, que trata de entender el origen de todo lo que consideramos REAL, abarca muchos temas variados, como la existencia y no existencia, la identidad y el cambio, el espacio y el tiempo, la causalidad, o la necesidad (esto debe ser obligatoriamente así) y la posibilidad (esto podría perfectamente ser de otra manera).

Dejando ya de dar rodeos, nos centraremos en la existencia/no existencia, tema estudiado por una rama de la metafísica conocida como Ontología

→ ontos = ente, ser, existencia; logos = conocimiento



# Conceptos generales: Ontología

Ontología: rama de la metafísica dedicada a tratar de entender:

- Por qué tenemos conciencia de nosotros mismos y conciencia de otros seres distintos a nosotros
- Por qué se producen cambios en nosotros mismos y en los demás seres, y aún así, después del cambio, tenemos la misma conciencia sobre nosotros mismos y sobre los demás seres, como si no hubiese habido cambio alguno
- ...
- Cómo organizar a todos los seres en categorías, y relacionar unas categorías con otras.

La Ontología trata de distinguir qué seres o entidades existen y cómo dichas entidades pueden ser agrupadas, jerarquizadas y subdivididas atendiendo a similitudes y diferencias entre ellas

# Conceptos generales: Ontología

Salgamos mentalmente de esta clase y hagamos un pequeño viaje por el mundo...  
¿Cómo sabemos que existen cosas (seres, entidades) diferentes en el mundo?

# Conceptos generales: Ontología

Percibimos el mundo a través de SENSACIONES que nos aportan nuestros sentidos: vista, oído, olfato, gusto y tacto

- Sensaciones visuales: color, tamaño, forma, posición...
  - Sensaciones auditivas: intensidad, altura, duración, timbre...
  - Sensaciones olfativas: intensidad, composición química...
  - Sensaciones gustativas: sabor (dulce, salado, ácido, amargo...)
  - Sensaciones hápticas: peso, dureza, textura, temperatura, humedad, conductividad eléctrica (y en caso positivo, intensidad de la corriente), capacidad corrosiva (se percibe cuando se están disolviendo los tejidos corporales en contacto)...
- Esas sensaciones pueden ir cambiando con el tiempo (percepción de movimiento)



# Conceptos generales: Ontología

Veamos un ejemplo simplificado de percepción del mundo:

- solo podemos percibir el mundo a través de la vista
- solo tenemos sensaciones visuales de posición y de color
- el mundo que podemos ver se reduce a lo que muestra esta imagen de una vidriera





# Conceptos generales: Ontología

- Si percibimos con la vista que existen cosas en diferentes posiciones del mundo:
  - Si no existe aún, creamos un nombre para definir esa sensación que permite distinguir unas cosas del mundo de otras  
→ nuevo nombre de SENSACIÓN: posición
  - Si no existen aún, creamos tantos símbolos como valores diferentes de posición logremos distinguir entre todas las cosas del mundo que podemos ver  
→ nuevos símbolos válidos para ASIGNAR A LA SENSACIÓN posición: (círculos rojos en la imagen)

# Conceptos generales: Ontología

- posición → valores posibles: círculos rojos en imagen



# Conceptos generales: Ontología

- Si percibimos con la vista que existen cosas de diferentes colores en el mundo:
  - Si no existe aún, creamos un nombre para definir esa sensación que permite distinguir unas cosas del mundo de otras  
→ nuevo nombre de SENSACIÓN: color
  - Si no existen aún, creamos tantos símbolos como valores diferentes de color logremos distinguir entre todas las cosas del mundo que podemos ver  
→ nuevos símbolos válidos para ASIGNAR A LA SENSACIÓN color: azul, amarillo, rojo, ...

# Conceptos generales: Ontología

- posición → valores posibles: círculos rojos en imagen
- color → valores posibles: azul, amarillo, rojo, ...



# Conceptos generales: Ontología

- Si vemos dos trozos de vidriera (distinta posición) de distinto color (uno amarillo y otro azul):
  - Pensamos en ellos como dos cosas (seres) diferentes, porque, o su posición, o su color, o ambos, tienen valores diferentes para uno y para otro. Por tanto, si no existen aún, creamos dos nombres para definir cada nueva cosa descubierta  
→ nuevos nombres de COSAS: vidrio1 y vidrio2
  - A cada cosa le ASOCIAMOS sus valores correspondientes de posición y color  
→ vidrio1 -> posición: (ver imagen); color: amarillo  
→ vidrio2 -> posición: (ver imagen); color: azul



# Conceptos generales: Ontología

- posición → valores posibles: círculos rojos en imagen
- color → valores posibles: azul, amarillo, rojo, ...

- vidrio1 {posición: ver imagen; color: amarillo}
- vidrio2 {posición: ver imagen; color: azul}



# Conceptos generales: Ontología

- Si luego vemos un tercer trozo de vidriera (distinta posición) de color azul:
  - Pensamos en él como una cosa diferente, porque al menos su posición tiene valor diferente al valor de posición de las demás cosas. Por tanto, si no existe aún, creamos un nombre para definir la nueva cosa descubierta  
→ nuevo nombre de COSA: vidrio3
  - A la nueva cosa le ASOCIAMOS sus valores correspondientes de posición y color  
→ vidrio3 -> posición: (ver imagen); color: azul

# Conceptos generales: Ontología

- posición → valores posibles: círculos rojos en imagen
- color → valores posibles: azul, amarillo, rojo, ...

- vidrio1 {posición: ver imagen; color: amarillo}
- vidrio2 {posición: ver imagen; color: azul}
- vidrio3 {posición: ver imagen; color: azul}



# Conceptos generales: Ontología

- Supongamos que tenemos la sensación de que vidrio3 es de un color azul más claro que vidrio2:
  - Como ambos azules producen sensaciones diferentes para el mismo color, podemos crear una sensación llamada 'tono' con valores 'claro' y 'oscuro' que permitan diferenciar vidrio2 y vidrio3 no solo por su posición, sino también por su tono de color, aunque su color tenga para ambas el mismo valor.
    - nuevo nombre de SENSACIÓN: tono (aplicada SOLO para distintos tonos de AZUL)
    - nuevos símbolos válidos para ASIGNAR A LA SENSACIÓN tono: claro, oscuro
  - Al conocer varias cosas del mundo de color azul, distintas entre sí gracias a la sensación tono que acabamos de crear, podemos pensar en crear un nombre para definir el conjunto de cosas del mundo de color azul, tengan el tono de azul que tengan. Serán tres las sensaciones que tendrán valor asignable en todas las cosas de color azul: posición, color y tono de azul
    - nuevo nombre de AGRUPACIÓN DE COSAS: VidriosAzules



# Conceptos generales: Ontología

- posición → valores posibles: círculos rojos en imagen
- color → valores posibles: azul, amarillo, rojo, ...
- tono: claro, oscuro

- vidrio1 {posición: ver imagen; color: amarillo}
- vidrio2 {posición: ver imagen; color: azul; tono: oscuro}
- vidrio3 {posición: ver imagen; color: azul; tono: claro}



VidriosAzules



# Conceptos generales: Ontología

- Si nos acabamos fijando en el fondo de la imagen (en negro):
  - Pensamos en él como una cosa diferente, porque al menos su color tiene valor diferente al valor de color de las demás cosas. Por tanto, si no existe aún, creamos un nombre para definir la nueva cosa descubierta  
→ nuevo nombre de COSA: fondo
  - A la nueva cosa le ASOCIAMOS su color con valor negro, pero no podemos asociarle su posición porque al ocupar muchos trozos de la imagen, no tiene una posición definida  
→ fondo -> color: negro

# Conceptos generales: Ontología

- posición → valores posibles: círculos rojos en imagen
- color → valores posibles: azul, amarillo, rojo, ...
- tono: claro, oscuro
- vidrio1 {posición: ver imagen; color: amarillo}
- vidrio2 {posición: ver imagen; color: azul; tono: oscuro}
- vidrio3 {posición: ver imagen; color: azul; tono: claro}
- fondo {color: negro}



VidriosAzules

# Conceptos generales: Ontología

- Ahora que existe el fondo, solo con color, y todas las demás cosas percibidas en la imagen tienen color y posición:
  - Al conocer varias cosas del mundo que tienen color y posición, distintas entre sí, y distintas al fondo, que no tiene posición, podemos pensar en crear un nombre para definir el conjunto de cosas del mundo que tienen color y posición, sean cuales sean.  
→ nuevo nombre de AGRUPACIÓN DE COSAS: Vidrios
  - También podemos pensar en que todas las cosas que vemos en la vidriera tienen color, y crear un nombre para definir el conjunto de cosas del mundo que tienen color, ya que hasta el momento no hay ningún nombre que defina esa agrupación  
→ nuevo nombre de AGRUPACIÓN DE COSAS: CosasConColor

# Conceptos generales: Ontología

- posición → valores posibles: círculos rojos en imagen
- color → valores posibles: azul, amarillo, rojo, ...
- tono: claro, oscuro

- vidrio1 {posición: ver imagen; color: amarillo}
- vidrio2 {posición: ver imagen; color: azul; tono: oscuro}
- vidrio3 {posición: ver imagen; color: azul; tono: claro}
- fondo {color: negro}



CosasConColor

Vidrios

VidriosAzules

# Conceptos generales: Ontología

Después de esta experiencia, hemos obtenido la siguiente información del mundo de la vidriera:

- Hay sensaciones que nos permiten comparar cosas. Hemos usado dos, la posición y el color, y hemos creado una tercera: el tono de azul
- Los distintos valores de posición, color y tono que hemos podido distinguir en las cosas también los hemos nombrado (caso de colores y tonos) o simbolizado (caso de posiciones)
- Usando la posición para comparar hemos visto cosas diferentes. Hemos dado nombres diferentes a cada cosa que estaba en una posición diferente, y hemos asignado a cada cosa, ya con un nombre único, su posición y color correspondientes



# Conceptos generales: Ontología

Después de esta experiencia, hemos obtenido la siguiente información del mundo:

- Usando el color para comparar hemos visto cosas iguales (del mismo color: azul). Sin embargo, aunque el color era azul, la sensación era diferente. Eso ha llevado a crear una nueva sensación (tono) que compartían todas las cosas de color azul. Desde ese momento, existía en la vidriera un conjunto de cosas que tenían en común tres sensaciones asociadas: tamaño, color y tono. Hemos dado al conjunto de cosas que compartían las tres sensaciones el nombre VidriosAzules

# Conceptos generales: Ontología

Después de esta experiencia, hemos obtenido la siguiente información del mundo:

- Hemos encontrado el fondo, que era diferente a todas las demás cosas de la vidriera porque no podíamos otorgarle un valor de posición de la misma forma que al resto de los elementos ya encontrados. Desde ese momento, existía en la vidriera un conjunto de cosas, distintas todas del fondo, que tenían en común dos sensaciones asociadas: posición y color. Hemos dado al conjunto de cosas que compartían las dos sensaciones el nombre Vidrios

# Conceptos generales: Ontología

Después de esta experiencia, hemos obtenido la siguiente información del mundo:

- Finalmente, hemos repasado las sensaciones asociadas a las distintas cosas vistas en la vidriera, y hemos descubierto que todas las cosas de la vidriera tenían en común una sensación asociada: color. Hemos dado al conjunto de cosas que compartían esa sensación el nombre CosasConColor

# Conceptos generales: Ontología

Después de esta experiencia, hemos obtenido la siguiente información del mundo:

- Así pues, hemos dado nombre a las siguientes tres agrupaciones de cosas de la vidriera:
  - CosasConColor → cosas con sensación color, como mínimo
  - Vidrios → cosas con sensaciones tamaño y color, como mínimo
  - VidriosAzules → cosas con sensaciones tamaño, color y tono, como mínimo
- Si nos fijamos bien, existe una relación jerárquica de inclusión entre las tres clases: TODAS las cosas incluidas en VidriosAzules están incluidas en Vidrios, y todas las cosas incluidas en Vidrios, sean VidriosAzules o no, están incluidas en CosasConColor

Hemos REPRESENTADO conjuntos de cosas y cosas del mundo con palabras, propiedades y relaciones. Eso es Ontología.

# Conceptos generales: clase y objeto

Con el ejemplo visto, ya tenemos todos los elementos que necesitamos definir para continuar. Ahora vamos a darles sus nombres adecuados:

- objeto (llamado cosa en el ejemplo): elemento del mundo
  - propiedad (usadas tres en el ejemplo: posición, color y tono): característica perceptible que posee un objeto o conjunto de objetos. Cada objeto que la posee, tiene asociado a la propiedad un valor que se puede conocer
- cada elemento del mundo posee unas propiedades con unos valores tales que ningún otro elemento del mundo posee exactamente las mismas propiedades con exactamente los mismos valores. Esto permite que percibamos cada objeto como individual, como único en el mundo.



# Conceptos generales: clase y objeto

- clase (llamado agrupación o conjunto de cosas en el ejemplo): conjunto mínimo de propiedades que uno o varios objetos comparten entre sí  
NOTA → para cada objeto individual los valores de esas propiedades pueden ser diferentes
- herencia de clase (llamada relación jerárquica de inclusión en el ejemplo): relación entre clases que se da cuando todo el conjunto de propiedades de una clase está incluido dentro del conjunto de propiedades de otra clase. De las dos, la clase que tiene menos propiedades se denomina clase padre, clase base o clase heredada, y la clase que tiene todas las propiedades de la clase padre y aparte añade alguna más, se llama clase hija, clase derivada o clase heredera

# Conceptos generales: clase y objeto

... volvemos al mundo de la programación (Uff... ¡Gracias!)

¿Qué tiene que ver toda la chapa filosófica anterior con escribir y ejecutar programas?

Pues bastante más de lo que parece, si queremos solucionar aquello con lo que ahora no estamos a gusto sobre el programa 13...

Recordamos:

- Código monolítico (todo en una única función)
- Información de un mismo elemento del mundo (cliente) repartida en variables distintas (listas)

# Conceptos generales: clase y objeto

- Los programas suelen trabajar con elementos del mundo real (o de un mundo imaginado)
- Como los elementos del mundo no se pueden meter físicamente en un programa de ordenador, la Programación Orientada a Objetos (OOP) plantea REPRESENTAR cada elemento del mundo como un conjunto de valores que corresponden a diferentes propiedades
- Este conjunto de propiedades está definido en una clase, que se comporta como un tipo de dato que permite declarar distintas variables de esa clase, una por elemento del mundo a representar
- Es decir, OOP plantea representar los elementos del mundo con los que queremos trabajar en nuestros programas como objetos pertenecientes a una clase, en cuya declaración se definen las propiedades que compartirán todos los objetos de esa clase

# Conceptos generales: clase y objeto

- Pero los programas no se limitarán a consultar las propiedades de un objeto estático, invariable. Normalmente realizarán operaciones que modificarán los valores de esas propiedades
- Una ontología da un nombre a cada objeto del mundo, y asocia a cada uno de esos nombres distintos unos valores para las propiedades definidas en la clase a la que pertenezca.
- OOP da un paso más y asocia también, al nombre de cada objeto, operaciones o funciones que se pueden aplicar sobre las propiedades que tenga asociadas para modificar sus valores
- Dicho de otro modo, las clases en OOP no son como las clases de una ontología: no definen solo propiedades, sino también funciones (llamadas **métodos** en OOP)

# Conceptos generales: clase y objeto

En términos de programación:

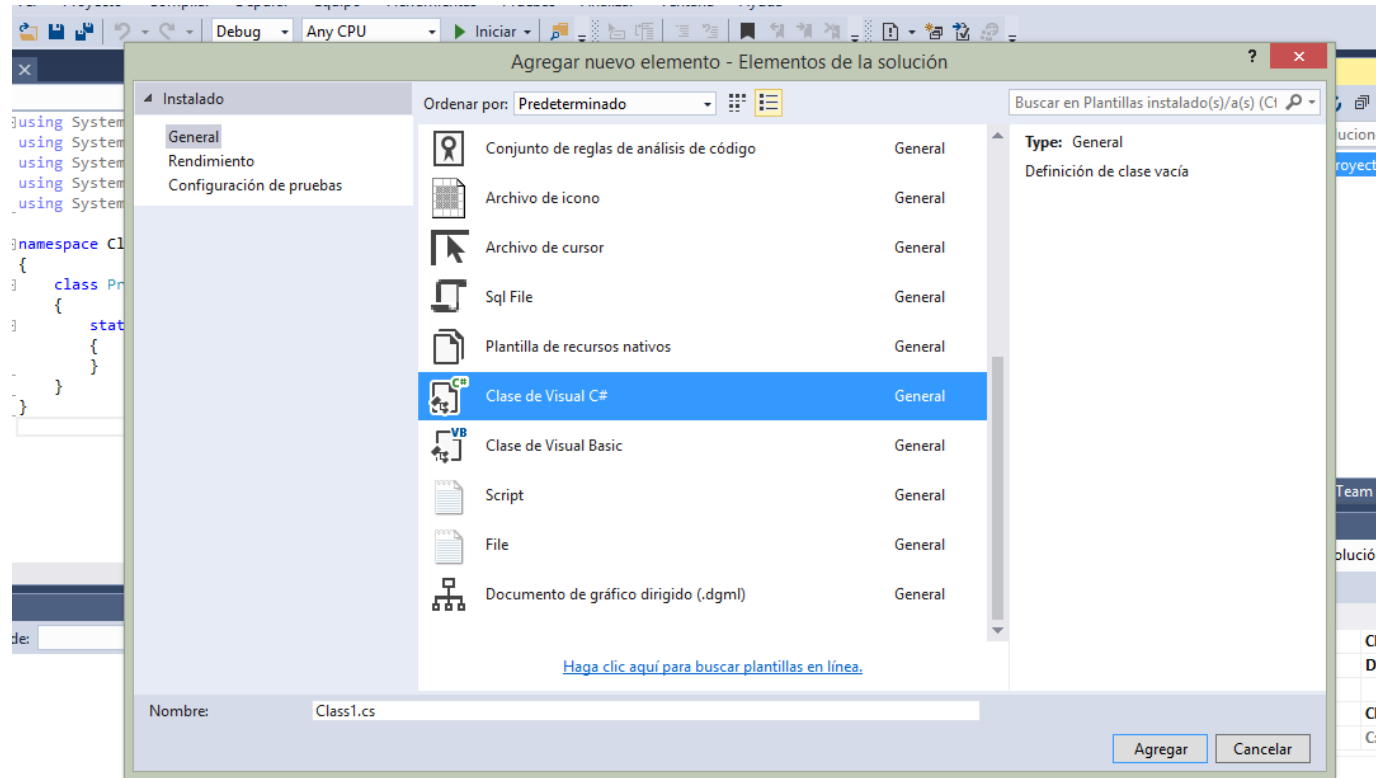
- una **clase** es una definición de un tipo de dato complejo que puede contener en su interior variables de distinto tipo (simple o complejo) y funciones que trabajen con esas variables
- Las variables definidas en una clase se llaman **propiedades** (las vamos a llamar así de momento...)
- Las funciones definidas en una clase se llaman **métodos**
- Un **objeto** es una variable de un tipo que está definido como clase. También se le llama **instancia** de clase (del inglés instance = ejemplo, ocurrencia única)



# Conceptos generales: crear clases

Para crear una nueva clase con VS:

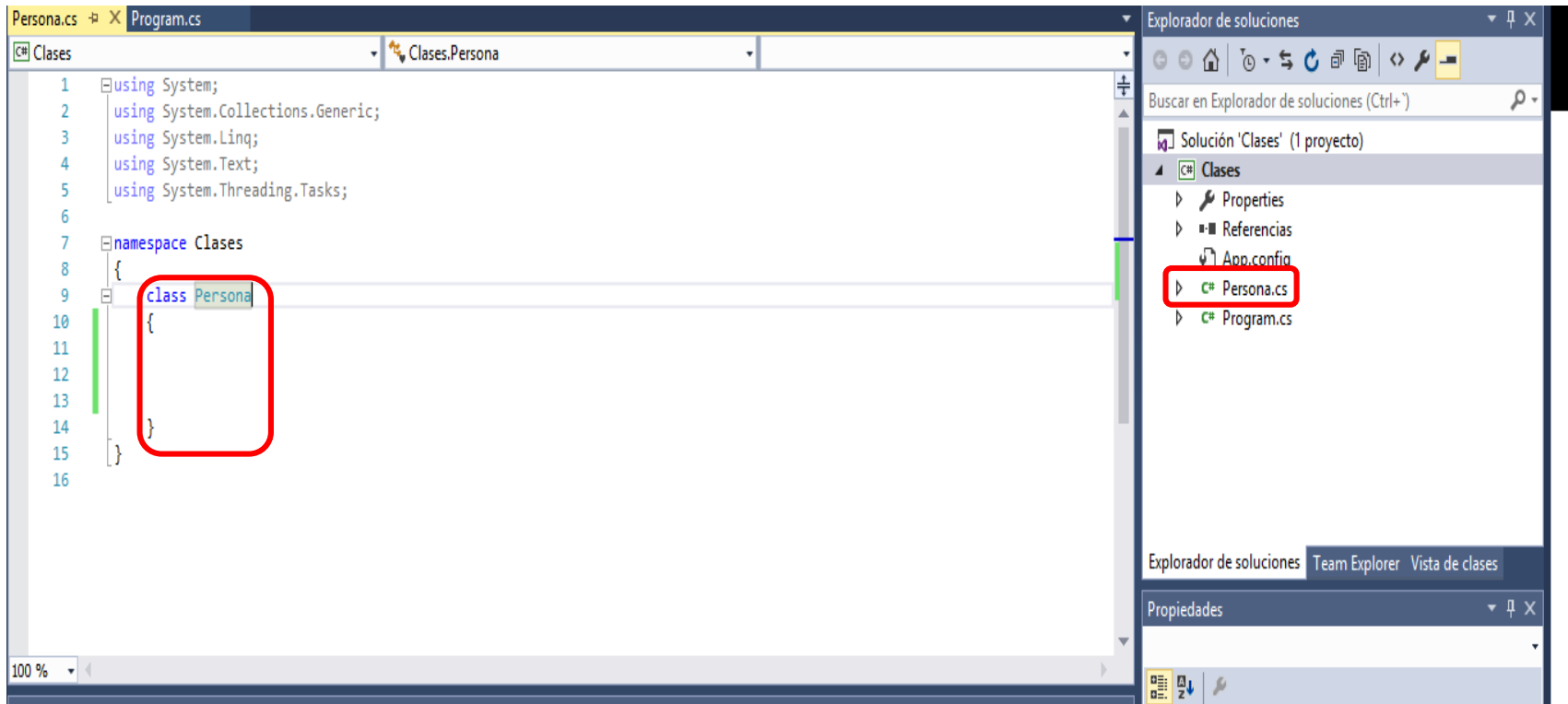
botón derecho sobre el proyecto en el Explorador de soluciones → Agregar → Clase



# Conceptos generales: crear clases

Para crear una nueva clase con VS:

Se genera un fichero con la siguiente estructura (en rojo lo relacionado con la clase creada)



# Conceptos generales: definir clases

Ejemplo de definición de clase:

```
public class Persona
{
    2 referencias
    public string Nombre { get; set; }
    2 referencias
    public string Apellidos { get; set; }
    0 referencias
    public int Edad { get; set; }

    0 referencias
    public Persona()
    {
        Nombre = "";
        Apellidos = "";
    }

    0 referencias
    public string NombreCompleto()
    {
        return Nombre + " " + Apellidos;
    }
}
```

# Conceptos generales: declarar clases

Ejemplo de definición de clase:

- Las clases, sus propiedades y sus métodos, de momento, se van a declarar como públicos
- Para poder obtener el valor de una propiedad, a ésta se le declara un método get;
- Para poder modificar el valor de una propiedad, a ésta se le declara un método set;

```
public class Persona
{
    2 referencias
    public string Nombre { get; set; }
    2 referencias
    public string Apellidos { get; set; }
    0 referencias
    public int Edad { get; set; }
    0 referencias
    public Persona()
    {
        Nombre = "";
        Apellidos = "";
    }
    0 referencias
    public string NombreCompleto()
    {
        return Nombre + " " + Apellidos;
    }
}
```

# Conceptos generales: declarar clases

Se puede declarar una variable (objeto) de tipo (clase) Persona, y leer o modificar las propiedades Nombre, Apellidos y Edad de ese objeto, o ejecutar el método NombreCompleto() de ese objeto con esta sintaxis:

```
Persona yo = new Persona();
```

```
yo.Nombre = "Sergio";  
yo.Apellidos = "Milla Pineda";
```

```
Console.WriteLine("Me llamo " + yo.NombreCompleto());
```

Para declarar e inicializar un objeto (asignar valor inicial a las propiedades deseadas) existe la siguiente sintaxis:

```
Persona yo = new Persona() { Nombre = "Sergio", Apellidos = "Milla Pineda" };
```

```
public class Persona  
{  
    2 referencias  
    public string Nombre { get; set; }  
    2 referencias  
    public string Apellidos { get; set; }  
    0 referencias  
    public int Edad { get; set; }  
  
    0 referencias  
    public Persona()  
    {  
        Nombre = "";  
        Apellidos = "";  
    }  
  
    0 referencias  
    public string NombreCompleto()  
    {  
        return Nombre + " " + Apellidos;  
    }  
}
```



# Conceptos generales: declarar clases

```
Persona yo = new Persona() { Nombre = "Sergio", Apellidos = "Milla Pineda" };  
Persona otro = new Persona() { Nombre = "John", Apellidos = "Smith" };
```

Cada vez que se declara una variable distinta de clase Persona con 'new Persona()', se está creando (instanciando) un objeto de la clase Persona.

Un objeto es una copia única de las propiedades y métodos que define la clase, de forma que los valores de las propiedades de un objeto (yo.Nombre y yo.Apellidos) serán distintos de los de cualquier otro objeto (otro.Nombre y otro.Apellidos), y los métodos de un objeto podrán acceder a los valores que tengan las propiedades de ese mismo objeto. (yo.NombreCompleto() devolverá "Sergio Milla Pineda" y otro.NombreCompleto() devolverá "John Smith")

```
public class Persona  
{  
    2 referencias  
    public string Nombre { get; set; }  
    2 referencias  
    public string Apellidos { get; set; }  
    0 referencias  
    public int Edad { get; set; }  
  
    0 referencias  
    public Persona()  
    {  
        Nombre = "";  
        Apellidos = "";  
    }  
  
    0 referencias  
    public string NombreCompleto()  
    {  
        return Nombre + " " + Apellidos;  
    }  
}
```

# Conceptos generales: declarar clases

- Con la posibilidad de definir clases tenemos resuelta una parte de aquello que no nos gustaba en el ejercicio del cajero multiusuario: la necesidad de crear distintas listas, y de tener que acceder a cada una de ellas para obtener los distintos valores asociados a un determinado cliente del cajero
- Ahora, podemos tener una única lista de elementos de clase Cliente donde, en su definición, esa clase Cliente tiene declaradas como propiedades las distintas informaciones asociadas a cada cliente: NumeroCuenta, Pin, Saldo, ListaIngresos, ListaRetiradas, ListaMovimientos, ...
- Más intuitivo de programar (más fácil, eso ya no lo sé... Depende de cada uno...)

# Conceptos generales: estructura de programa

- Ahora nos falta hacer algo con el código monolítico...
- Con lo que sabemos ahora, podemos entender que nuestro programa está ejecutando actualmente el código del método Main declarado en la clase Program
- Si os digo que una clase puede tener varios métodos definidos, y que desde el código de uno de ellos se puede ejecutar el código de cualquier otro... podemos empezar a pensar cómo trocear nuestro código monolítico para separarlo en distintos métodos declarados dentro de la clase Program...
  - Pero hay un problema: La clase Program es estática... Y no es tan fácil usar una clase estática...

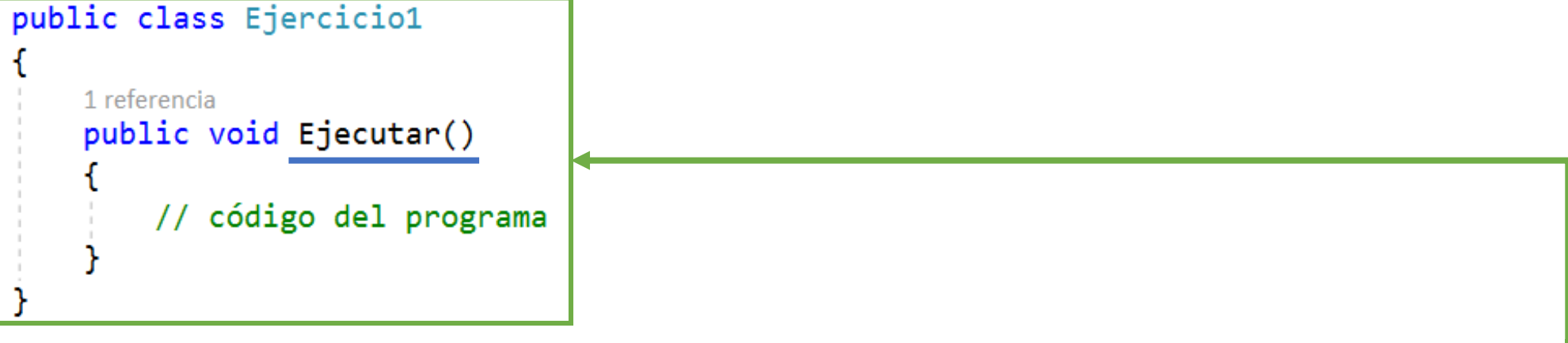
# Conceptos generales: estructura de programa

Veremos más adelante lo que es una clase estática, y cómo trabajar con ella, pero para facilitar lo que necesitáis saber al empezar, vamos a definir una estructura especial para todos los programas de consola que vayamos a escribir:

- Nuestro código NO VA A ESTAR en la clase Program, sino en una clase nueva que crearemos para cada ejercicio.
- Como el método Main() de la clase Program seguirá siendo el punto de entrada de la ejecución de los programas de consola, habrá que escribir código en él, pero el único código que contendrá será:
  - La declaración de un objeto de una clase que contenga el código del programa que queramos ejecutar
  - La ejecución, A TRAVÉS DEL OBJETO CREADO, del método que contenga el código inicial de nuestro programa

# Conceptos generales: estructura de programa

```
public class Ejercicio1
{
    1 referencia
    public void Ejecutar()
    {
        // código del programa
    }
}
```



```
class Program
{
    0 referencias
    static void Main(string[] args)
    {
        Ejercicio1 ej1 = new Ejercicio1();
        ej1.Ejecutar();
    }
}
```

- La **declaración de un objeto** de una clase que contenga el código del programa que queremos ejecutar
- La ejecución, **A TRAVÉS DEL OBJETO CREADO**, del **método** que contenga el código inicial de nuestro programa
- (En breve veremos eso de void qué significa...)



# Conceptos generales: estructura de programa

Para quien no quiera escribirlo en dos líneas, hay una sintaxis más corta:

```
class Program
{
    0 referencias
    static void Main(string[] args)
    {
        new Ejercicio1().Ejecutar();
    }
}
```

En este caso se puede aplicar esta sintaxis, donde no se guarda el objeto en ninguna variable, porque no necesitamos leer ni modificar el objeto creado después de ejecutar el método que contiene el código del programa

# Conceptos generales: estructura de programa

En todo caso, se recomienda acabar todos los programas de consola con las líneas:

```
class Program
{
    0 referencias
    static void Main(string[] args)
    {
        new Ejercicio1().Ejecutar();

        Console.WriteLine("(Fin del programa: Pulse cualquier tecla para terminar...)");
        Console.ReadKey();
    }
}
```

Las dos líneas finales se podrían incluir al final del método Ejecutar() de la clase Ejercicio1. Eso tendría sentido si nuestro proyecto de consola solo pudiera ejecutar un programa. Pero aquí haremos que nuestro proyecto de consola pueda ejecutar los 13 programas escritos en el módulo 1, y para no tener que escribir al final de todos ellos las mismas dos líneas, las escribimos solo una vez al final del método Main() que permitirá ejecutar un programa u otro

# Conceptos generales: estructura de programa

Así pues, se creará una clase distinta por programa, y para cambiar de un programa a otro:

```
static void Main(string[] args)
{
    new Ejercicio1().Ejecutar();

    Console.WriteLine("(Fin del programa: Pulse cualquier tecla para terminar...)");
    Console.ReadKey();
}
```



```
static void Main(string[] args)
{
    new Ejercicio2().Ejecutar();

    Console.WriteLine("(Fin del programa: Pulse cualquier tecla para terminar...)");
    Console.ReadKey();
}
```

# Conceptos generales: método

- Y bueno, ahora que hemos visto que tendremos que ejecutar métodos... parece un buen momento para hacerse una pregunta:

¿qué son exactamente los métodos?



# Conceptos generales: método

- Un método es un bloque de código que contiene una serie de instrucciones. Se puede entender como un miniprograma (datos de entrada, operaciones, resultado)
- Su bloque de código se ejecuta cuando desde otra parte del programa se invoca al método: nombreMétodo + '(' + parámetros (si los tiene, separados por comas) + ')'.

```
public override double GetPorcentajeIVA()
{
    return 0.21;
}

static void Main()
{
    double precio = 100;
    double IVA = GetPorcentajeIVA() * precio;
    Console.WriteLine("The price is {0}", IVA);
}
```



# Conceptos generales: método

- Hasta ahora, hemos ejecutado un solo método (Main) con todo el código del programa
- En su interior, sin decirlo, hemos invocado ya a distintos métodos de otras clases:
  - `Console.WriteLine()`
  - `int.TryParse()`
  - `{nombre variable tipo List<decimal>}.Add()`
  - ...

# Conceptos generales: método

- La sintaxis en C# para definir un método es la siguiente:

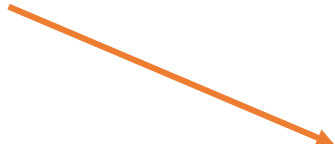
```
[NivelDeAccesibilidad] [tipo] [nombre del método] ([parámetros])  
{  
    return [variabletipo];  
}
```

- Ejemplo:

```
public int AddTwoNumbers (int number1, int number2)  
{  
    int result = number1 + number2;  
    return result;  
}
```

# Conceptos generales: método

- La definición del tipo devuelto, el nombre del método y su lista de parámetros se conoce como SIGNATURA del método:



```
public int AddTwoNumbers (int number1, int number2)
{
    int result = number1 + number2;
    return result;
}
```

# Conceptos generales: método

- Por el momento, vamos a pensar que los métodos tienen que tener su nivel de accesibilidad como public, aunque no se explique aún lo que significa
- Más adelante, cuando se explique el principio de abstracción en OOP, se podrán dar otros niveles de accesibilidad a los métodos, ya sabiendo lo que se hace...

# Conceptos generales: paso de parámetros

Para hablar de las distintas formas de paso de parámetros, partimos de una invocación a un método sencillo, que recibe dos enteros como parámetros, y devuelve su suma:

```
int suma = AddTwoNumbers(2, 3);
```

-----

```
public int AddTwoNumbers (int number1, int number2)
{
    int result = number1 + number2;
    return result;
}
```

# Conceptos generales: paso de parámetros

## Paso de parámetros: por valor (in)

```
int parametroIn = 0;
Console.WriteLine($"antes de AddTwoNumbersIn: parametroIn={parametroIn}");
int res = AddTwoNumbersIn(2, 5, parametroIn);
Console.WriteLine($"después de AddTwoNumbersIn: res={res}, parametroIn={parametroIn}");
Console.WriteLine(res);
```

```
-----

public int AddTwoNumbersIn(int n1, int n2, int paramIn)
{
    Console.WriteLine($" - empieza AddTwoNumbersIn: n1={n1}, n2={n2}, paramIn={paramIn}");
    int result = n1 + n2;
    paramIn = result+1;
    Console.WriteLine($" - termina AddTwoNumbersIn: {n1}+{n2}={result}, paramIn={paramIn}");
    return result;
}
```



# Conceptos generales: paso de parámetros

## Paso de parámetros: por valor (in)

```
antes de AddTwoNumbersIn: parametroIn=0
- empieza AddTwoNumbersIn: n1=2, n2=5, paramIn=0
- termina AddTwoNumbersIn: 2+5=7, paramIn=8
después de AddTwoNumbersIn: res=7, parametroIn=0
7
```

Tipo	Descripción
Por valor	<ul style="list-style-type: none"><li>- Programa invoca al método pasándole una variable (parametroIn) con valor 0.</li><li>- C# Sí asigna ese valor 0 al parámetro paramIn antes de empezar la ejecución del método</li><li>- Se ejecuta el método, y justo antes de acabar, el valor del parámetro paramIn es 8</li><li>- Al acabar el método, NO se asigna el valor del parámetro paramIn a la variable parametroIn</li><li>- Se sigue ejecutando el programa que ha invocado al método, y se comprueba que el valor de la variable parametroIn sigue con el valor que tenía antes de invocar al método</li></ul>

# Conceptos generales: paso de parámetros

## Paso de parámetros: por referencia sin inicializar (out)

```
int parametroOut = 0;
Console.WriteLine($"antes de AddTwoNumbersOut: parametroOut={parametroOut}");
int res = AddTwoNumbersOut(2, 5, out parametroOut);
Console.WriteLine($"después de AddTwoNumbersOut: res={res}, parametroOut={parametroOut}");
Console.WriteLine(res);
```

```
-----
public int AddTwoNumbersOut(int n1, int n2, out int paramOut)
{
    Console.WriteLine($"empieza AddTwoNumbersOut: n1={n1}, n2={n2}, paramOut=(Uso del parámetro out sin asignar 'paramOut')");
    int result = n1 + n2;
    paramOut = result + 1;
    Console.WriteLine($"termina AddTwoNumbersOut: {n1}+{n2}={result}, paramOut={paramOut}");
    return result;
}
```

# Conceptos generales: paso de parámetros

## Paso de parámetros: por referencia sin inicializar (out)

```
antes de AddTwoNumbersOut: parametroOut=0
empieza AddTwoNumbersOut: n1=2, n2=5, paramOut=(Uso del parámetro out sin asignar 'paramOut')
termina AddTwoNumbersOut: 2+5=7, paramOut=8
después de AddTwoNumbersOut: res=7, parametroOut=8
7
```

Tipo	Descripción
Por referencia sin inicializar	<ul style="list-style-type: none"><li>- Programa invoca al método pasándole una variable (parametroOut) con valor 0.</li><li>- C# NO asigna ese valor 0 al parámetro paramOut antes de empezar la ejecución del método</li><li>- Es OBLIGATORIO que el método le asigne un valor durante su ejecución, sino NO COMPILARÁ</li><li>- Se ejecuta el método, y justo antes de acabar, el valor del parámetro paramOut es 8</li><li>- Al acabar el método, Sí se asigna el valor del parámetro paramOut a la variable parametroOut</li><li>- Se sigue ejecutando el programa que lo ha invocado, y se comprueba que el valor de la variable parametroOut tiene el valor que ha acabado teniendo el parámetro paramOut</li></ul>

# Conceptos generales: paso de parámetros

Paso de parámetros: por referencia con valor inicial (ref)

```
int parametroRef = 0;
Console.WriteLine($"antes de AddTwoNumbersRef: parametroOut={parametroRef}");
int res = AddTwoNumbersRef(2, 5, ref parametroRef);
Console.WriteLine($"después de AddTwoNumbersRef: res={res}, parametroRef={parametroRef}");
Console.WriteLine(res);
```

```
-----
public int AddTwoNumbersRef(int n1, int n2, ref int paramRef)
{
    Console.WriteLine($"empieza AddTwoNumbersRef: n1={n1}, n2={n2}, paramRef={paramRef}");
    int result = n1 + n2;
    paramRef = result + 1;
    Console.WriteLine($"termina AddTwoNumbersRef: {n1}+{n2}={paramRef}, paramRef={paramRef}");
    return result;
}
```

# Conceptos generales: paso de parámetros

Paso de parámetros: por referencia con valor inicial (ref)

```
antes de AddTwoNumbersRef: parametroRef=0
empieza AddTwoNumbersRef: n1=2, n2=5, paramRef=0
termina AddTwoNumbersRef: 2+5=8, paramRef=8
después de AddTwoNumbersRef: res=7, parametroRef=8
7
```

Tipo	Descripción
Por referencia con valor inicial	<ul style="list-style-type: none"><li>- Programa invoca al método pasándole una variable (parametroRef) con valor 0.</li><li>- C# Sí asigna ese valor 0 al parámetro paramRef antes de empezar la ejecución del método</li><li>- Se ejecuta el método, y justo antes de acabar, el valor del parámetro paramRef es 8</li><li>- Al acabar el método, Sí se asigna el valor del parámetro paramRef a la variable parametroRef</li><li>- Se sigue ejecutando el programa que lo ha invocado, y se comprueba que el valor de la variable parametroRef tiene el valor que ha acabado teniendo el parámetro paramRef</li></ul>

# Conceptos generales: paso de parámetros

En resumen:

Tipo	¿el valor de la variable pasada en la invocación se transmite al parámetro correspondiente antes de que el método empiece su ejecución?	¿el valor del parámetro correspondiente se transmite a la variable pasada en la invocación después de que el método termine su ejecución?
Por valor (in)	SÍ	NO
Por referencia sin inicializar (out)	NO	SÍ
Por referencia con valor inicial (ref)	SI	SÍ



# Conceptos generales: paso de parámetros

En resumen:

- Parámetro por valor: cuando se desea comunicar un valor a un método, pero no que el método pueda cambiar ese valor
- Parámetro por referencia sin inicializar: cuando se desea que un método devuelva varios resultados (uno lo devuelve en el return, los demás en parámetros out)
- Parámetro por referencia con valor inicial: cuando se desea comunicar un valor a un método y que el método pueda cambiar ese valor

# Conceptos generales: paso de parámetros

- Las variables de tipos primitivos (int, decimal, bool, char), por defecto se pasan por valor. Si se desea que pasen por referencia es necesario añadirles out o ref tanto en la invocación como en la declaración del método
- Las variables de tipos correspondientes a clases, por defecto se pasan por referencia con valor inicial, y no se pueden pasar por valor. Si se desea que pasen por referencia sin inicializar es necesario añadirles out tanto en la invocación como en la declaración del método
- El tipo string es especial, porque aunque String es una clase (y como tal deberían pasar siempre por referencia), String es a la vez un tipo inmutable, por lo que cada vez que estamos modificando una variable string, realmente estamos creando otro objeto con el nuevo valor. Eso hace que por defecto se pasen por valor, como el resto de los tipos primitivos.
  - Si se desean usar strings mutables, existe la clase System.Text.StringBuilder

# Conceptos generales: paso de parámetros

Paso de parámetros: parámetros opcionales que tienen un valor por defecto

- En la invocación no es necesario pasar el parámetro
- Los parámetros opcionales solo pueden ir al final de la lista de parámetros

```
int resultado = AddTwoNumbers(2);    // resultado tendrá valor 2+0 = 2  
int resultado2 = AddTwoNumbers(2, 3); // resultado2 tendrá valor 2+3 = 5
```

```
public int AddTwoNumbers (int number1, int number2 = 0)  
{  
    int resul = number1 + number2;  
    return resul;  
}
```

# Conceptos generales: paso de parámetros

Paso de parámetros: parámetros pasados por nombre

- Permite paso de parámetros desordenado, SOLO si todos los parámetros se pasan así

```
int res = AddTwoNumbersRef(paramRef: ref parametroRef, n1: 2, n2: 5);
```

```
-----  
public int AddTwoNumbersRef(int n1, int n2, ref int paramRef)  
{  
    Console.WriteLine($"empieza AddTwoNumbersRef: n1={n1}, n2={n2}, paramRef={paramRef}");  
    int result = n1 + n2;  
    paramRef = result + 1;  
    Console.WriteLine($"termina AddTwoNumbersRef: {n1}+{n2}={paramRef}, paramRef={paramRef}");  
    return result;  
}
```

# Conceptos generales: tipo devuelto

Podemos querer un método que no devuelva ningún resultado. Para ello, el tipo devuelto indicado en la declaración del método será 'void' (vacío), y además dentro del bloque de código del método NO habrá ninguna instrucción 'return {expr};'.

- Puede haber instrucción 'return;', sin nada entre 'return' y el punto y coma, para terminar.
- Si no hay ninguna instrucción 'return;', el método terminará tras ejecutarse su última línea

- Ejemplo:

```
public void WriteErrorMessage (string msg)
{
    System.Console.WriteLine("ERROR: " + msg);
}
```

# Conceptos generales: constructor

Un constructor es un método especial dentro de una clase:

- Es un método cuyo código se ejecuta SOLO cuando se crea una nueva instancia (u objeto) de la clase. Su nivel de accesibilidad debe ser public porque la idea es que se invoque desde otra clase diferente (aunque se puede invocar desde la propia clase)
- Es un método que tiene el mismo nombre que la clase, y se utiliza para dar valores iniciales a los atributos de la clase, y/o ejecutar métodos que queramos ejecutar siempre que se cree un nuevo objeto de la clase
- Una clase puede tener diferentes constructores, con distintos parámetros cada uno
- Aunque el constructor no esté escrito explícitamente en una clase, existe igualmente
  - En ese caso, lo único que hace es crear los atributos de la clase con los valores por defecto para cada tipo
  - El constructor creado por defecto no tiene parámetros, por eso podemos escribir `Persona yo = new Persona();` // invocación al constructor por defecto si no hay otro



# Conceptos generales: modularización

- Ahora sabemos cómo crear clases con métodos, y hemos decidido crear una clase distinta para cada programa que hemos ido desarrollando hasta ahora en el curso
- En cada clase creada correspondiente a cada programa, se crea un método, por ejemplo de nombre Ejecutar(), que contiene el código (DESCOMENTADO si estaba comentado desde el módulo 1) del programa correspondiente
- Como hemos comentado, NO SE RECOMIENDA pasar el código de cada ejercicio a un método definido en la misma clase Program, porque el método Main es static, y aunque aún no se ha visto lo que es, os adelanto que desde un método static solo se pueden invocar a otros métodos static y solo se pueden leer y escribir propiedades static, y eso será un problema si más adelante queréis subdividir esos métodos y usar propiedades no static para pasar información entre métodos

# Conceptos generales: modularización

- Ejemplo: partimos de una clase con un método Ejecutar que tiene el siguiente código:

```
public class Ejercicio1
{
    public void Ejecutar()
    {
        Console.Write("Introduzca un número entero: ");
        int num1 = (int.TryParse(Console.ReadLine(), out int n1) == true ? n1 : 0);
        Console.Write("Introduzca otro número entero: ");
        int num2 = (int.TryParse(Console.ReadLine(), out int n2) == true ? n2 : 0);
        int res = num1 + num2;
        Console.WriteLine($"{num1} + {num2} = {res}");
    }
}
```

# Conceptos generales: modularización

- Podemos pensar en DIVIDIR, SEPARAR el código del programa en distintos métodos, pero si varios métodos hacen uso de una misma variable que solo esté declarada en uno de ellos, allí donde se use sin estar declarada habrá un error de compilación

```
public void PedirDato1PorPantalla()
{
    Console.Write("Introduzca un número entero: ");
    int num1 = (int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0);
}

public void PedirDato2PorPantalla()
{
    Console.Write("Introduzca otro número entero: ");
    int num2 = (int.TryParse(Console.ReadLine(), out int n2) ? n2 : 0);
}

public void ObtenerResultado()
{
    int res = num1 + num2; ¡ERROR!
}
```

```
public void Ejecutar()
{
    PedirDato1PorPantalla();
    PedirDato2PorPantalla();
    ObtenerResultado();
    Console.WriteLine($"{num1} + {num2} = {res}");
}

¡ERROR!
```

# Conceptos generales: modularización

- Para solucionar el problema de las variables usadas en distintos métodos, se puede:
  - Transmitir el valor de la variable entre métodos por paso de parámetros, a lo largo de la ejecución del método principal

```
public int PedirDato1PorPantalla()
{
    Console.Write("Introduzca un número entero: ");
    return (int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0);
}

public int PedirDato2PorPantalla()
{
    Console.Write("Introduzca otro número entero: ");
    return (int.TryParse(Console.ReadLine(), out int n2) ? n2 : 0);
}

public int ObtenerResultado(int num1, int num2)
{
    return num1 + num2;
}
```

```
public void Ejecutar()
{
    int num1 = PedirDato1PorPantalla();
    int num2 = PedirDato2PorPantalla();
    int res = ObtenerResultado(num1, num2);
    Console.WriteLine($"{num1} + {num2} = {res}");
}
```

# Conceptos generales: modularización

- Para solucionar el problema de las variables usadas en distintos métodos, se puede:
  - No declarar la variable en ningún método, sino como propiedad de la clase, y acceder a leer/escribir su valor desde cualquier método (inicializándola antes donde corresponda)

```
public void PedirDato1PorPantalla()
{
    Console.Write("Introduzca un número entero: ");
    num1 = (int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0);
}

public void PedirDato2PorPantalla()
{
    Console.Write("Introduzca otro número entero: ");
    num2 = (int.TryParse(Console.ReadLine(), out int n2) ? n2 : 0);
}

public void ObtenerResultado()
{
    res = num1 + num2;
}
```

```
public class Ejercicio1
{
    public int num1 { get; set; }
    public int num2 { get; set; }
    public int res { get; set; }

    public void Ejecutar()
    {
        PedirDato1PorPantalla();
        PedirDato2PorPantalla();
        ObtenerResultado();
        Console.WriteLine($"{num1} + {num2} = {res}");
    }
}
```

# Conceptos generales: modularización

- Si se hace de la 2ª forma (la forma recomendada), tener especial cuidado de inicializar propiedades de tipos que sean una clase, como List o alguna clase propia como Persona o Cliente. Si no, no habrá errores de compilación, pero sí de ejecución:

```
public void Ejecutar()
{
    List<int> lista = new List<int>();
    Console.Write("Introduzca un número entero: ");
    int num1 = (int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0);
    lista.Add(num1);
    Console.WriteLine($"El 1º elemento de la lista es {lista[0]}");
}
```



```
public List<int> lista { get; set; }
public int num1 { get; set; }

public void Ejecutar()
{
    PedirDatoPorPantalla();
    lista.Add(num1);
    Console.WriteLine($"El 1º elemento de la lista es {lista[0]}");
}
```

```
public void Ejecutar()
{
    PedirDatoPorPantalla();
    lista.Add(num1);
    Console.WriteLine($"El 1º elemento de la lista es {lista[0]}");
}
```

```
public void PedirDatoPorPantalla()
{
    Console.Write("Introduzca un número entero: ");
    num1 = (int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0);
}
```

Excepción producida

**System.NullReferenceException:** 'Referencia a objeto no establecida como instancia de un objeto.'

GestionTrabajadores.Services.Ejercicio1.lista.get devolvió null.



¡MAAAAAAAL!



# Conceptos generales: modularización

- Si se hace de la 2ª forma (la forma recomendada), tener especial cuidado de inicializar propiedades de tipos que sean una clase, como List o alguna clase propia como Persona o Cliente. Si no, no habrá errores de compilación, pero sí de ejecución:

```
public void Ejecutar()  
{  
    List<int> lista = new List<int>();  
    Console.Write("Introduzca un número entero: ");  
    int num1 = (int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0);  
    lista.Add(num1);  
    Console.WriteLine($"El 1º elemento de la lista es {lista[0]}");  
}
```



```
public List<int> lista { get; set; }  
public int num1 { get; set; }  
  
public void Ejecutar()  
{  
    lista = new List<int>();  
    PedirDatoPorPantalla();  
    lista.Add(num1);  
    Console.WriteLine($"El 1º elemento de la lista es {lista[0]}");  
}  
  
public void PedirDatoPorPantalla()  
{  
    Console.Write("Introduzca un número entero: ");  
    num1 = (int.TryParse(Console.ReadLine(), out int n1) ? n1 : 0);  
}
```

```
Introduzca un número entero: 23  
El 1º elemento de la lista es 23
```



¡BIEEEEEEN!

# Conceptos generales: modularización

- Con la posibilidad de mover cada código de programa a una clase distinta, y además separar el código dentro de la clase en distintos métodos, tenemos completamente resuelto el problema de la estructura monolítica del código, que:
  - Obligaba a dejar comentado el código que no queríamos ejecutar para no perderlo, y tenerlo disponible solo descomentándolo cuando lo quisiéramos volver a ejecutar
  - No permitía de ningún modo la reutilización de código

# Conceptos generales: modularización

Así pues, conviene:

- Separar en distintas clases todo el código y todas las variables que no tengan relación entre sí
- Juntar dentro de una misma clase todo el código y todas las variables que tengan relación entre sí, pero separándola en lo posible en distintos métodos
- Si una variable se usa en varios métodos, conviene crearla como propiedad de la clase e inicializarla en el primer método donde se utilice.

A este proceso se le llama MODULARIZACIÓN del código. Beneficios:

- Código separado (frente a código monolítico). Más fácil de leer y corregir
- Permite reutilización del código repetido, o que se desee repetir en otro programa

# Ejercicio práctico

## Programa 14: Agenda

Realizar una aplicación, la cual al entrar nos indique las siguientes opciones:

- 1-Dar de alta nuevo contacto; 2-Borrar contacto; 3-Buscar teléfono de un contacto; 4-Mostrar todos los contactos y devolver el número de contactos dados de alta; 5-Salir.
- Opc 1 - Por cada contacto, el programa pedirá Nombre, Apellidos y Nº teléfono. Los contactos se almacenarán en memoria.
- Opc 2 – El programa pedirá nombre y apellidos del contacto que queremos eliminar y buscará si existe en la agenda. Si existe, antes de eliminarlo pedirá confirmación por pantalla y solo lo eliminará si el usuario confirma su deseo de eliminarlo.
- Opc 3 – El programa pedirá nombre y apellidos de un contacto existente y buscará si existe en la agenda. Si existe, devolveremos su número de teléfono.
- Opc 4 – El programa mostrará todos los contactos ordenados alfabéticamente, y al final mostrará el nº de contactos encontrados.
- Opc 5 – El programa se cerrará.

# Ejercicio práctico

## Programa 14: Agenda

- El programa declarará una variable de tipo `List<Contacto>`
- Contacto será una clase pública con tres propiedades públicas de tipo string: Nombre, Apellidos y Telefono

# Conceptos avanzados: modificador static

Podemos necesitar propiedades que no guarden valores diferentes para cada nueva instancia de su clase, sino que guarden un valor compartido para todas las instancias.

Estas propiedades se declaran con la palabra clave **static**

```
public class Persona {  
    public static int Incrementer { get; set; }  
    public static List<Persona> PersonasList { get; set; }  
    ...  
}
```

Se accede a ellas desde fuera de su clase con  
{nombre\_clase}.{nombre\_propiedad} -> Ejemplos:

Persona.Incrementer, Persona.PersonasList



# Conceptos avanzados: modificador static

Podemos necesitar métodos que lean o escriban valores en propiedades estáticas (y solo en propiedades estáticas). Estos métodos se declaran con la palabra clave **static**

```
public class Persona {  
    public static List<Persona> PersonasList { get; set; }  
  
    ...  
  
    public static void MostrarLista() { (código antes) PersonasList (código después) }  
}
```

Se accede a ellos desde fuera de su clase con  
{nombre\_clase}.{nombre\_método}({parámetros}) -> Ejemplo: Persona.MostrarLista()

# Conceptos avanzados: modificador static

Podemos necesitar métodos que lean o escriban valores en propiedades estáticas (y solo en propiedades estáticas). Estos métodos se declaran con la palabra clave **static**

- Los métodos estáticos sólo pueden invocar a otros métodos estáticos
- Los métodos estáticos solo pueden leer o escribir sobre propiedades estáticas (o variables definidas en el propio método)
- Estas dos particularidades son las que nos han llevado a preferir la estructura de programa donde el código del programa está en una clase externa NO ESTÁTICA, en lugar de tener que estar acordándonos de definir como estáticos en la clase Program tanto los métodos como las propiedades que fuésemos a necesitar...

# Conceptos avanzados: modificador static

Podemos necesitar clases que contengan propiedades con valores constantes o métodos que ejecuten operaciones tan generales que nos interese definirlos como métodos de acceso global, sin necesidad de crear un objeto de su clase para poder invocarlos. Estos métodos también se declaran con la palabra clave **static**, y se definen dentro de una clase **static**. NO SE PUEDEN CREAR INSTANCIAS de una clase static

```
public static class Math {  
    public const double PI = 3.1415926535897931;  
  
    ...  
  
    public static decimal Max(decimal val1, decimal val2);  
    public static decimal Min(decimal val1, decimal val2);  
}
```

Se accede a ellos desde fuera de su clase con  
Math.PI, Math.Max(1,2), Math.Min(variable1, 10)

# Conceptos avanzados: modificador static

Ejemplos de utilidad:

```
public class Persona {  
    public static int Incrementer { get; set; }  
    public int Id { get; set; }  
  
    public Persona() { Incrementer++; Id = Incrementer; }  
}
```

El código anterior ASEGURA que cada nueva instancia de la clase Persona tenga un atributo Id con un valor distinto al de todas las demás instancias, es decir, permite usar el atributo Id como un IDENTIFICADOR ÚNICO para cada objeto de clase Persona creado durante la ejecución del programa.

# Conceptos avanzados: modificador static

Ejemplos de utilidad:

```
public class ListaPersonas {  
    public static List<Persona> Lista { get; set; }  
  
    public static void Mostrar() { (código antes) Lista (código después) }  
}
```

El código anterior permite gestionar una lista de personas QUE SEA ACCESIBLE desde cualquier otra clase, por ejemplo la clase principal del programa, y además poder invocar un método, también desde cualquier clase, que muestre el contenido de la lista anterior por pantalla.

Acceder a la lista: 'ListaPersonas.Lista', Invocar al método: 'ListaPersonas.Mostrar()'

# Ejercicio práctico

## Programa 15: Gestión de una biblioteca

Escribir un programa desde el cual se pueda gestionar una biblioteca. Al entrar nos mostrará las siguientes opciones:

- 1 - Añadir nuevo libro
- 2 - Alquilar libro
- 3 - Buscar libro
- 4 - Mostrar listado de todos los libros de un determinado género
- 5 - Mostrar listado de libros alquilados
- 6 - Mostrar listado de libros disponibles (no alquilados)
- 7 - Devolver un libro
- 8 - Salir

# Ejercicio práctico

## Programa 15: Gestión de una biblioteca. Explicación de cada opción

Opc 1 - Por cada libro nos pedirá: título, autor, año y género. Se almacenará en memoria. Cada libro tendrá asociado un código numérico, que se calculará de manera transparente al usuario, y un bool que indicará si el libro se encuentra alquilado o no (y que se inicializará a false al añadir cada nuevo libro)

Opc 2 - Nos pedirá el código, habrá que comprobar que el código exista, así como si ha sido alquilado o no. En caso de estar alquilado no permitirá alquilarlo, lo mismo en caso de que no exista.

Opc 3 - Nos solicitará el título y devolveremos los datos del libro: título, autor, año, género, y si se encuentra alquilado o no, siempre y cuando el libro exista.

Opc 4 - Nos solicitará un género y devolveremos todos los datos de los libros correspondientes al género introducido.

Opc 5 - Devolverá una lista de los libros alquilados.

Opc 6 - Devolverá una lista de los libros no alquilados.

Opc 7 - Nos solicitará el código del libro e indicaremos que el libro ya no está alquilado.



# Conceptos avanzados: namespace

- Todas las clases de C# se definen DENTRO de un namespace (espacio de nombres) con un nombre determinado. Este namespace se declara al principio del archivo donde se define la clase.

```
using System;

namespace Program
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("¡Hola mundo!");
        }
    }
}
```

# Conceptos avanzados: namespace

Existen varias formas de poder acceder a la clase Console del namespace System desde el método Main de la clase Program:

- Escribiendo System.Console cada vez que se quiera utilizar esa clase para algo

```
namespace Program
{
    class Program
    {
        static void Main(string[] args)
        {
            System.Console.WriteLine("¡Hola mundo!");
        }
    }
}
```

# Conceptos avanzados: namespace

- Añadiendo al principio del archivo donde se define la clase Program una sentencia 'using' que contenga el namespace System al que pertenece la clase Console, y escribiendo solo Console cada vez que se quiera utilizar esa clase para algo. Esta es la forma recomendada.

```
using System;

namespace Program
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("¡Hola mundo!");
        }
    }
}
```

# Conceptos avanzados: namespace

Estas mismas dos formas diferentes se pueden aplicar para:

- declarar una propiedad de clase Clase1 en otra clase Clase2

```
public Clase2 { ... public Clase1 Atributo1 {get; set;} ... }
```

- declarar una variable de clase Clase1 en algún método de otra clase Clase2

```
public Clase2 { ... public metodo1(){ ... Clase1 c1 = new Clase1(); ...} ... }
```

- invocar a un método estático de la clase Clase1 en algún método de otra clase Clase2

```
public Clase2 { ... public metodo1(){ ... Clase1.MetodoX(); ...} ... }
```

# Conceptos avanzados: namespace

- Hay una tercera forma, consistente en añadir una sentencia using estática, no del namespace que contiene la clase Console, sino de la propia clase Console:

```
using static System.Console;
```

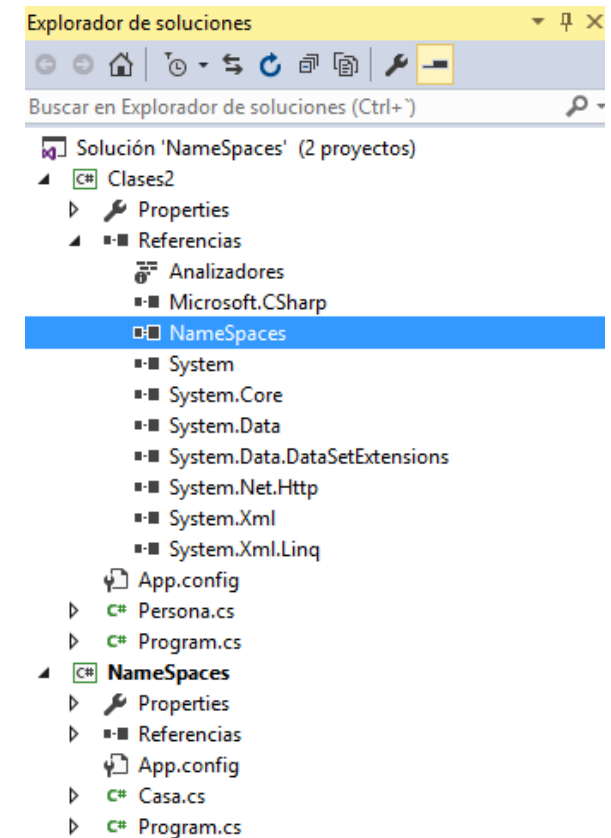
- Con esto se evita tener que escribir 'Console.' delante de cada invocación a cualquier método estático de la clase Console, pasando de 'Console.WriteLine();' a 'WriteLine()'
- Aunque cada programador es libre de usar o no esta tercera forma, no se recomienda en absoluto para programas de consola que trabajan a la vez con entrada/salida por pantalla y con entrada/salida desde ficheros de texto, ya que la clase File (como se verá más adelante) tiene un método WriteLine() que puede generar ambigüedad con el de consola si se emplea esta sentencia using estática con la clase Console

# Conceptos avanzados: namespace

En caso de tener más de un proyecto en una misma solución y querer acceder desde un proyecto a clases definidas en otro, hay que hacer lo siguiente:

- En el proyecto que quiere acceder a la clase externa, agregamos nueva referencia al proyecto donde está definida esa clase

En el ejemplo, desde el proyecto Clases2 se ha añadido → una referencia al proyecto NameSpaces. Tras esto, y nunca antes de esto, será posible que la clase Program del proyecto Clases2 pueda declarar variables de tipo Casa, que es una clase pública definida en el proyecto NameSpaces.





# MUCHAS GRACIAS



Pasión por la innovación

[www.integratecnologia.es](http://www.integratecnologia.es)