Tema 4

Descargar estos apuntes

Índice

Introducción

Concepción de los primeros programas

Antiguamente **los programas eran monolíticos** con instrucciones de ruptura y salto tanto condicional como incondicional del tipo `GoTo` o `GoSub`.

> **Problemáticas**

- Código "*spaghetti*".
- Tiempo elevado de corrección de errores.
- Baja reusabilidad del código.
- Difícil documentación.
- Mantenimiento excesivamente costoso.

> **Lo que buscamos**

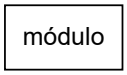
- Corrección.
- Legibilidad y portabilidad.
- Fácil mantenimiento, código no duplicado o repetido.
- Fácil depuración de errores.
- Código reutilizable (genericidad) y sencillez.

Programación Estructurada

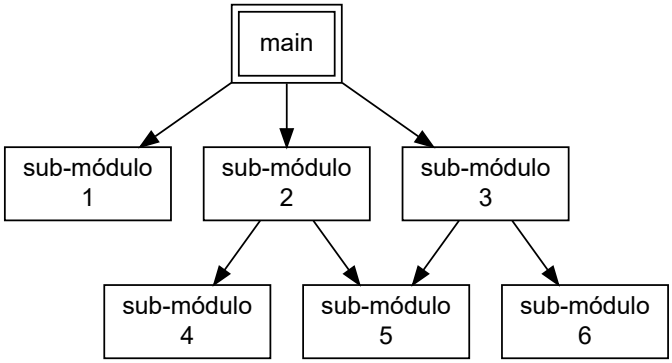
Durante la **década de los 70** diferentes autores como Stevens, Myers y Constantine, 1974 y Yourdon y Constantine, 1979 definieron las bases de la **PE** (*programación estructurada)*. Esta, definió las bases de cómo modularizar el código, las cuales se usaron hasta **principios de los 90** donde se empezó a popularizar la **POO** (*Programación Orientada a Objetos*). **No obstante, estos principios estarán también vigentes en la POO y por tanto deberemos de conocerlos.**

Definiciones formales a la hora de modularizar en PE

- **Módulo** o (*Subprograma*, *Subrutina*): Parte de un programa que realiza una tarea concreta mediante una serie de instrucciones.
  Lo simbolizaremos mediante una caja.

- **Modularizar**: Dividir una tarea o módulo en otros sub-módulos.
- **Transferencia de control:** Cuando un módulo llama a un sub-módulo para realizar parte de su tarea.
- **Módulo principal** o (*main*): Módulo que controla al resto.
  En el siguiente diagrama estamos simbolizando una **modularización**, donde el **módulo principal** será el de más arriba y lo simbolizaremos por una línea doble. Para realizar su tarea hace una **transferencia de control** a los 3 sub-módulos inferiores en orden de izquierda a derecha. A su vez el `sub-módulo 2` llama transfiere el control a los `sub-módulos 4 y 5` respectivamente para realizar su tarea y tiene el `sub-módulo 5` en común con el `sub-módulo 3`. Generándose así una estructura **jerárquica**.



- **Grado de entrada** o (*fan-in*): El número de módulos que usan otro módulo.
  En el diagrama de ejemplo serían las flechas que llegan. Así pues, el `sub-módulo 5` tiene `fan-in = 2`
- **Grado de salida** o (*fan-out*): El número de módulos subordinados a otro. Debería ser el mínimo necesario para realizar la tarea.
  En el diagrama de ejemplo serían las líneas que salen. Así pues, el `main` tiene `fan-out = 3`, generándose así una **dependencia** de sus 3 sub-módulos.
- **Visibilidad** o (*Scope*): El conjunto de módulos o componentes que pueden ser usados por otro.
- **Cohesión:**
  - Debe ser lo más **alta** posible.
  - Se da cuando los módulos en el último nivel, realizarán tareas **atómicas** y no tendrá diferentes funcionalidades según la entrada aprovechando un código común.
  - Un modulo solo hace una cosa y las instrucciones para hacerlo guardan un orden y relación lógica.
- **Acoplamiento:**
  - Debe ser el más **bajo** posible.
  - El acoplamiento **mide la interacción** entre módulos.
  - Por tanto trataremos de ocultar la información contenida en los mismos de tal manera que no pueda ser accedida desde otros módulos.
  - **Para tener bajo acoplamiento:**
    - Todo lo que comparten dos rutinas se especifica en la lista de parámetros del la subrutina llamada (Solo los parámetros indispensables).
    - No debemos conocer la implementación de un módulo para poder utilizarlo.
    - Evitar siempre que dos módulo utilicen algún tipo de variable global común.
  - **Nos proporciona:**
    - Nos proporciona independencia entre módulos.
    - Evita la propagación de errores.

Tipos de sub-módulos según su valor de retorno.

- **Procedimientos**: No devuelven ningún valor asociado a su nombre y son el resultado de la división de un módulo más grande que agrupa sentencias relacionadas en una subrutina.
- **Funciones**: Retornan al módulo llamador un valor asociado a su nombre.

🤚 Resumen de criterios para una correcta modularización

> - Definiremos una **interfaz clara**, de tal manera que el nombre y los parámetros me indique, sin redundancia, lo que hace un módulo **sin necesidad de conocer su implementación**.
> - Buscaremos módulos con **alta cohesion**. Esto es, pequeños, de poca complejidad y que solo hagan una cosa.
> - Buscaremos módulos con **bajo acoplamiento**. Esto es, evitaremos usar en ellos variables globales o externas al ámbito del mismo. Un sub-modulo se dedicará a hacer su tarea en función de sus parámetros de entrada, sin saber por quien es llamado ni para quien la hace. Esto implica que si producimos una salida, será siempre en función de los parámetros de entrada.

Diseño modular Orientado a Objetos

Método

- Aunque en posteriores temas entraremos formalmente el la POO. Podemos decir ya que en POO los módulos siempre se definirán dentro de una clase y dentro de este ámbito se denominarán **métodos**.
- Como C# es un lenguaje OO puro, los módulos siempre estarán definidos dentro de una y por tanto tendrán la nomenclatura de método independientemente de si son funciones o procedimientos.

Signatura

- En POO al conjunto formado por: el identificador o nombre de un método, y el número y tipo de sus parámetros formales, se le denomina **signatura**.
- Dos métodos pueden tener el mismo nombre, si sus parámetros son diferentes. (Cambia su signatura). A esto ya veremos más adelante que se le denomina Sobrecarga o Polimorfismo funcional en POO.

Tipos de métodos

1. **Métodos de objeto o también (de *instancia*)**: Son llamados a través de un objeto o una estructura instanciada en memoria.
   (Profundizaremos en ellos al ver POO)
2. **Métodos de acceso y actualización**: Me permiten acceder y modificar el estado de un objeto.
   (Profundizaremos en ellos al ver POO)
3. **Métodos estáticos o también (de tipo, de *clase*)**:
   - No necesitan ningún tipo de instancia en memoria del objeto o la estructura a la que pertenecen para ser llamados.
   - Son los más parecidos conceptualmente a las funciones y procedimientos en la PE.

Definición de interfaces en métodos

Conceptos previos

Definiciones conceptuales de parámetros

- **Formales**
  La forma en que se comunican el subprograma y el módulo que lo llama.

```
static double Pow(double x, double y) // Parámetro Formal
{
    ...
}
```

- **Reales o Actuales**
  Podemos decir que es un valor concreto en un momento de la ejecución de los parámetros formales.

```
Pow(2d, 10d); // Parámetro Real o Actual
```

Tipo de parámetros al flujo de datos

- **De entrada**: Los necesitará el método para realizar su ejecución y no deben modificar su valor dentro del mismo.
- **De salida**: Si el método es una '*función*', contendrán el valor o los valores que devuelve la misma después de su ejecución.
- **De entrada/salida**: Pasan un dato de entrada que puede quedar modificado tras la ejecución del método. (💀 **Hay que evitarlos**)

Sintaxis en C#

Declaración de método estático con retorno

```
class Tipo
{
  public static tipoDevuelto IdentificadorMétodo(parámetrosFormales)
  {
    // Cuerpo del método…
    return variableDelTipoDevuelto; // Devuelve una copia
  }
}
```

Al ser público el identificador del método irá en **PascalCasing** y para los parámetros formales usaremos **camelCasing**.

Llamada a método estático con retorno

- Si se hace **fuera** de la clase donde se define:
  `tipoDevuelto idVariable = Tipo.IdentificadorMétodo(parámetrosReales);`
  (Profundizaremos en ellos al ver POO)

```
    class Fecha
    {
        public static bool AñoBisiesto(ushort año)
        {
            return (bool)(año % 4 == 0 && (año % 100 != 0 || año % 400 == 0));
        }
    }

    class Principal
    {
        static void Main()
        {
13          string salida = (Fecha.AñoBisiesto(2008) ? "Es" : "No es") + " bisiesto";
            Console.WriteLine(salida);
        }
    }
```

- Si se hace **dentro** de la clase donde se define:
  `tipoDevuelto idVariable = IdentificadorMétodo(parámetrosReales);`
  (Son los que usaremos para aprender a modularizar en los primeros programas)

```
    class Principal
    {
        static bool AñoBisiesto(ushort año)
        {
            return (bool)(año % 4 == 0 && (año % 100 != 0 || año % 400 == 0));
        }

        static void Main()
        {
10          string salida = (AñoBisiesto(2008) ? "Es" : "No es") + " bisiesto";
            Console.WriteLine(salida);
        }
    }
```

Declaración de método estático sin retorno

- Usaremos para ello el tipo `void` (vacío) como tipo de retorno.
- Su sintaxis suponiendo que es un método estático dentro de una misma clase será:

```
static void IdProcedimiento(parámetros_formales)
{
    // Cuerpo del método
    return; // Podemos obviarlo
}
```

- Ejemplo ...

```
    class Datos
    {
3       public static void Muestra(
            string nombre,
5           string apellido,
            ushort edad)
        {
            Console.WriteLine($"   Nombre: {nombre}\n" +
                              $" Apellido: {apellido}\n" +
                              $"     Edad: {edad}");
        }

        public static void MétodoSinRetorno()
        {
            // Llamada desde un método de la misma clase.
            Muestra("Xusa", "García", 15);
        }
    }
```

Llamada de método estático sin retorno

- Haremos la llamada sin asignar el retorno a una variable o sin usarlo en una expresión.

```
IdProcedimiento(parámetros_reales);
```

- Ejemplo ...

```
    class Principal
    {
        static void Main()
        {
5           Datos.Muestra("Juanjo", "Guarinos", 18);
6           Datos.MétodoSinRetorno();
        }
    }
```

'Tips' sobre la alineación de parámetros

- Visual Studio y **VS Code** tienen herramientas para alineación y sangria de parámetros en métodos.
- Cuando tenemos pocos parámetros pueden ir en la misma línea de la definición.

```
public static void Muestra(string nombre, string apellido, ushort edad, string dirección)
```

  Si no están así, esto se consigue en el editor, colocándose sobre un parámetro y pulsando `Ctrl+.` A continuación nos ofrecerá la opción `'Ajustar todos los parámetros → Desajustar todos los parámetros'`
- Si tenemos 4 o más parámetros, nuestra recomendación es que se haga una sangria de parámetros.
  Para ello: `Ctrl+.` sobre los parámetros y a continuación seleccionar `'Ajustar todos los parámetros → Alinear parámetros ajustados'` obteniendo así...

```
public static void Muestra(string nombre,
                           string apellido,
                           ushort edad,
                           string dirección)
```

  Posteriormente otra vez `Ctrl+.` y a continuación seleccionar `'Ajustar todos los parámetros → Aplicar sangría a todos los parámetros'` obteniendo así...

```
public static void Muestra(
    string nombre,
    string apellido,
    ushort edad,
    string dirección)
```

De esta forma será más fácil...

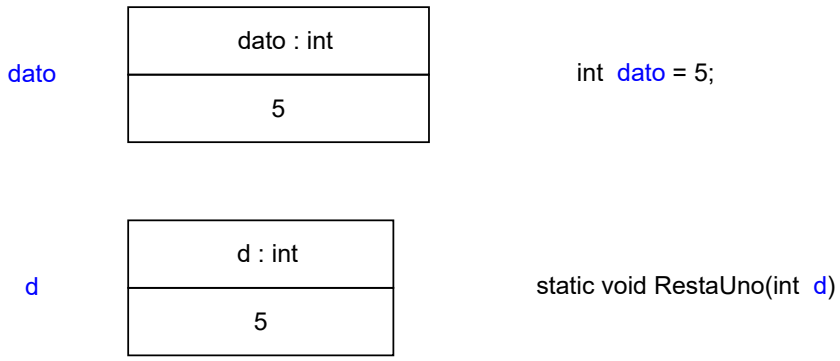
1. Ver y saber todos los parámetros de una función. Si que se pierdan por el lado derecho del editor.
2. Añadir parámetros nuevos.
3. Eliminar un parámetro en la declaración.
4. Cambiar el orden de los mismos con `Alt+↑` o `Alt+↓` sobre el parámetro.
5. Además, podremos cambiar fácilmente la indentación de los mismos, seleccionándolos y luego aplicando `Tab` o `Shift+Tab`

Definir parámetros de entrada

Paso por valor

- Si no ponemos ningún modificador o cláusula al parámetro formal. El paso será **por valor**.
- Por tanto, si es un tipo valor el objeto se duplica en memoria y si pasamos un tipo referencia, se duplicará la referencia al objeto.

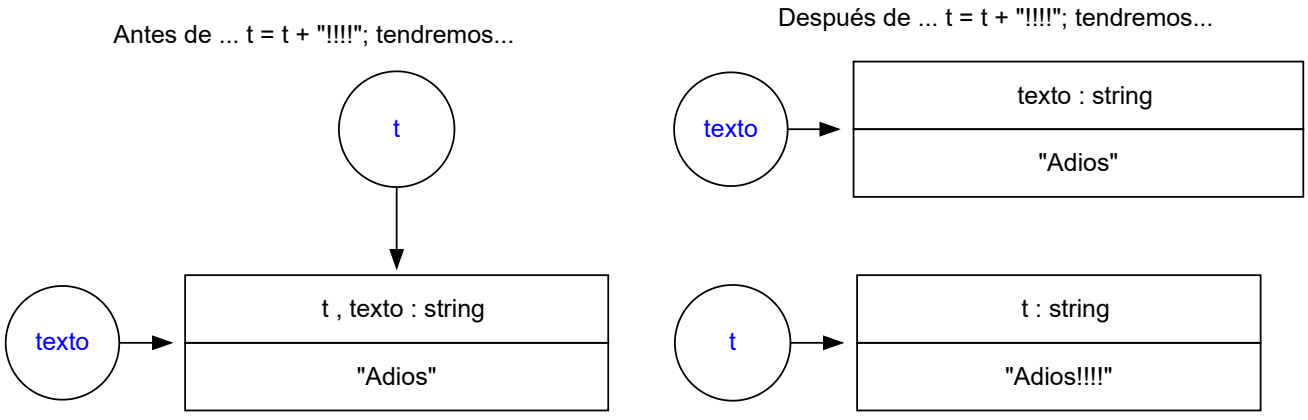
```
class Ejemplo
{
    static void RestaUno(int d)
    {
        // El parámetro formal con id. 'd' solo existe en este
        // ámbito y es una copia en el Heap del tipo valor dato
        // se pasó como parámetro real.

        --d;
        Console.Write(d); // Muestra 4

    } // el parámetro de entrada 'd' deja de existir al finalizar el método.

    static void Main()
    {
        int dato = 5;

        // Se pasa una copia de dato por que es un Tipo-valor.
        RestaUno(dato);
        // Después de la llamada dato sigue valiendo 5
        // por que lo que se modifica es la copia.
        Console.Write(dato); // Muestra 5
    }
}
```



```
class Ejemplo
{
3       static int RestaUno(int d)
        {
            --d;
            Console.Write(d); // Muestra 4

            // Devolvemos una copia de lo que vale d en el Heap
            // antes de que se 'destruya'.
            return d;
        }

        static void Main()
        {
            int dato = 5;

            // Sobrescribimos el Tipo-valor dato con lo que devuelve la función.
            dato = RestaUno(dato);
            Console.Write(dato); // Muestra 4
        }
}
```

- **¿Qué sucede si pasamos por valor un Tipo-referencia en lugar de un Tipo-Valor?**
  En el siguiente ejemplo vamos a ver que pasa usando el tipo referencia básico cadena.
  (*Este tipo lo veremos en más profundidad más adelante.*)

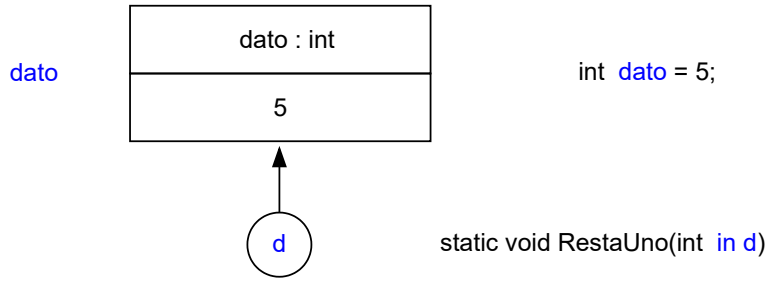
```
class Ejemplo
{
3       static void AñadeExclamación(string t)
        {
            // t es una copia de la referencia texto.
            t = t + "!!!!";
            // t referenciará ahora a una cadena nueva que será destruida
            // al finalizar este ámbito de ejecución.
        }

        static void Main()
        {
            // texto será una referencia a un objeto cadena.
            string texto = "Adios";

            // Al pasar por valor hacemos una copia de la referencia.
            AñadeExclamación(texto);
            Console.Write(texto); // Muestra "Adios" sin la !
        }
}
```



Paso por referencia con in

- Al poner el modificador o cláusula `in` (*línea 3*) al identificador del parámetro formal. El paso será **por referencia**, pero **no lo podremos modificar dentro del método**.
- Por tanto, si es un tipo valor será una referencia al mismo objeto en memoria y si pasamos un tipo referencia, se tratará de una referencia a la referencia.
- En la llamada (**línea 20**) deberemos anteponer al parámetro real la cláusula `in`.

```
class Ejemplo
{
3       static void RestaUno(in int d)
        {
            // El parámetro formal con id. 'd' es una referencia
            // a dato que se pasó como parámetro real.

            // Ya no podremos hacer --d; pues dará error.
            Console.Write(d); // Muestra 5

        } // el parámetro de entrada 'd' deja de existir y por tanto la referencia a dato.
          // pero dato sigue existiendo.

        static void Main()
        {
            int dato = 5;

            // Se pasa una referencia a la zona de memoria donde está dato.
            // Sabemos que nunca se modificará el valor internamente.
20          RestaUno(in dato);
            Console.Write(dato); // Muestra 5
        }
    }
```



💀 Definir parámetros de entrada/salida

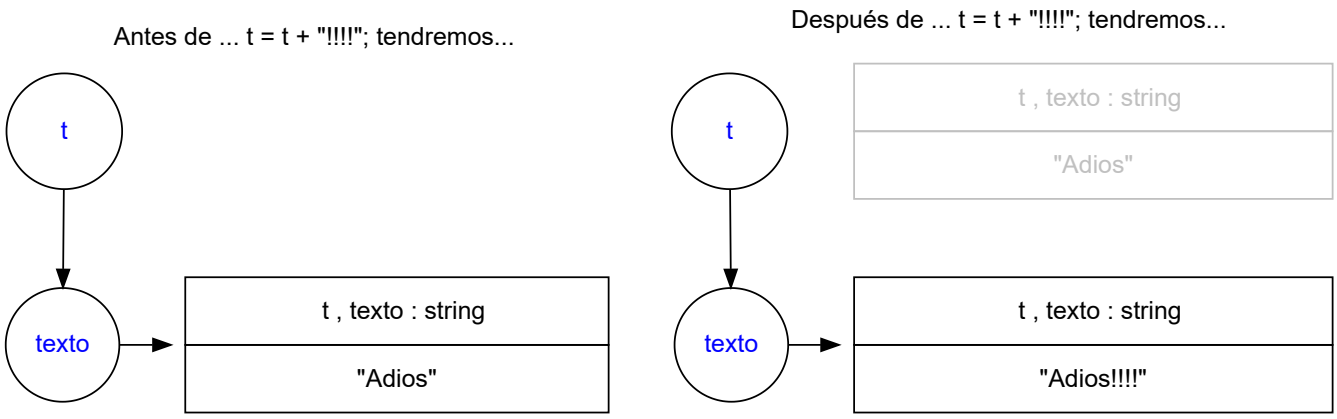
Utilizaremos el modificador `ref` (*línea 3*) antes del tipo para indicar un paso por referencia.

- Igual que `in` pero **podremos modificar el valor del parámetro formal** por eso consideramos que es un parámetro que '*entra*' con un valor y '*sale*' con otro.
- Deberemos evitarlos en la medida de lo posible, pues **dan lugar a efectos laterales** y **acoplamiento**.
- En la llamada (**línea 20**) deberemos anteponer al parámetro real la cláusula `ref`.

```
class Ejemplo
{
3       static void RestaUno(ref int d)
        {
            // El parámetro formal con id. 'd' es una referencia
            // a dato que se pasó como parámetro real.

            --d; // Ahora podemos modificar sin problema el valor del parámetro.
            Console.Write(d); // Muestra 4

        } // el parámetro de entrada 'd' deja de existir y por tanto la referencia a dato.
          // pero dato sigue existiendo.

        static void Main()
        {
            int dato = 5;

            // Se pasa una referencia a la zona de memoria donde está dato.
            // El valor podrá modificarse internamente.
20          RestaUno(ref dato);
            Console.Write(dato); // Muestra 4
        }
    }
```

- **¿Qué sucede si pasamos por referencia un Tipo-referencia en lugar de un Tipo-Valor?**
  Veamos que pasa con el ejemplo que usamos en el paso por valor.

```
class Ejemplo
{
    static void AñadeExclamación(ref string t)
    {
        // t es una referencia a la referencia texto.
        t = t + "!!!!";
        // al cambiar ahora t estaremos cambiando el valor de la referencia texto.
    }

    static void Main()
    {
        // texto será una referencia a un objeto cadena.
        string texto = "Adios";

        // Al pasar por valor hacemos pasaremos una referencia a la referencia.
        AñadeExclamación(ref texto);
        Console.Write(texto); // Muestra "Adios!"
    }
}
```



Definir parámetros de salida

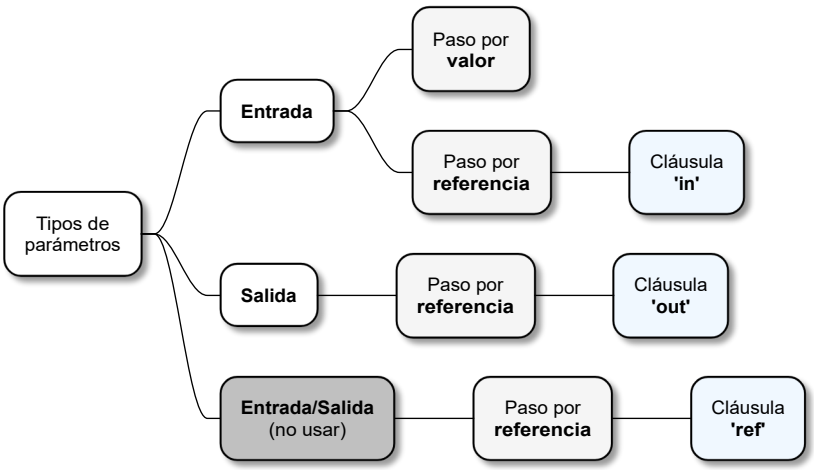
Utilizaremos la palabra reservada `out` (*líneas 6 y 7*) antes del tipo para indicar un parámetro de salida.

- Equivale al **paso por referencia**, pero **solo de salida**, es decir los parámetros reales **no tienen que estar inicializados**.
- Su uso muy aclaratorio para el programador, cuando un método **tiene más de un parámetro de salida**, además la devolución por referencia es menos costosa.
- En la llamada (*línea 19*) deberemos anteponer al parámetro real la cláusula `out` . Además, `out` nos permite **declarar el tipo** del parámetro en la misma llamada.

> **Nota**: La sintaxis del lenguaje a mejorado con el tiempo, y actualmente disponemos de otros mecanismos para devolver más de un parámetro, como son las **Tuplas** y los **Objetos Anónimos** que veremos más adelante.

```
class Ejemplo
{
    // Es conveniente colocarlos al final del interfaz.
    public static void Direccion(
        double anguloGr,
        out double sen,
        out double cos)
    {
        double anguloRad = anguloGr * Math.PI / 180;
        // Como están marcados de salida si se quedan sin asignar
        // el compilador me dará un error.
        sen = Math.Sin(anguloRad);
        cos = Math.Cos(anguloRad);
    }
    static void Main()
    {
        double anguloGr = 75;
        // Se pueden declarar en la misma llamada a la función.
        Direccion(anguloGr, out double sen, out double cos);
        Console.WriteLine($"Sen = {sen:F3}");
        Console.WriteLine($"Cos = {cos:F3}");
    }
}
```

Cuadro resumen paso de parámetros



'*Tips*' para definir un interfaz

Determinar el nombre o identificador

1. Usaremos **PascalCasing**. Ej: double CalculaDistancia(…)
2. Usaremos **sustantivos** o verbos conjugados en el caso de procedimientos.
3. Si es una función que retorna algo me indicará lo que devuelve o calcula.
4. No debe ser ambiguo, tiene que ser suficientemente descriptivo, así **auto-documentamos el código**. Solo con ver el nombre más los parámetros **sabremos lo que hace el método sin conocer su implementación**.
5. Recuerda un método sólo hace una cosa. Por tanto, si al decidir el nombre usamos alguna conjunción como 'y' u 'o' el método será incorrecto.
   - `float PorcentajeDeDescuento(double tarifa, double precio) {...}` 👍
   - `float Porcentaje(double tarifa, double precio) {...}` **Es ambiguo** 👎
6. Recuerda que **identificador + parámetros** definen la **signatura**. Por tanto, evitaremos redundancias en el nombre, en relación con los parámetros. Ya que estos también me proporcionan información de lo que hace el método.
   Esto último requiere de una reflexión tras elegir un nombre.
   - `public static double Distancia(Punto2D p1, Punto2D p2) {...}` 👍
   - `public static double CalculaDistanciaEntrePuntos(Punto2D p1, Punto2D p2) {...}` **Redundante** 👎

Determinar parámetros de entrada

1. Usaremos **nombres descriptivos** para los parámetros formales.
2. Aunque, comentamos que no debíamos utilizar contracciones. En determinados contextos podremos usar una inicial para el nombre del parámetro formal, pues no hay posibilidad de confusión, ni ambigüedad.
   - `double Distancia(Punto2D i, Punto2D j) {...}` **i, j son ids. usados para índices** 👎
   - `double Distancia(Punto2D punto1, Punto2D punto2) {...}` 👍
   - `double Distancia(Punto2D p1, Punto2D p2) {...}` 👍👍
3. El pasar los parámetros **sin agrupar** en los módulos o funciones básicas o atómicas nos proporcionará mayor reusabilidad. Ya que evitaremos acoplamientos y dependencias de tipos complejos.

   > **Nota:** Esto es difícil de valorar y objeto de discusión, porque lo normal es hacer lo contrário.

   En el siguiente ejemplo. Es más posible que pueda usar el método en más sitios, si utilizo coordenadas puesto que no necesito conocer la clase `Punto2D`.
   - `double Distancia(Punto2D p1, Punto2D p2){...}` 👍
   - `double Distancia(double x1, double y1, double x2, double y2){...}` 👍👍

Determinar parámetros de salida

1. A la hora de asignarles un nombre, si fuera necesario, usaremos la mismas reglas que para los parámetros de entrada.
2. Si solo tenemos **un parámetro de salida** lo podremos devolver con la misma función.
   - `double AnguloEnRadianes(double x1, double y1, double x2, double y2)` 👍
3. Si tenemos **más de un parámetro de salida**, tendremos **2 posibilidades**:
   1. Están relacionados entre sí, pero aparentemente uno está subordinado al otro.
      Un ejemplo puede ser el método estático TryParse del tipo `int`
      `bool TryParse (string s, out int result) {...}`
      Devuelve si ha podido o nó transformar la cadena `s` a entero, y si lo ha podido hacer (*subordinación*) me devolverá en `result` el valor entero resultado de la transformación.
   2. Si los parámetros están relacionados sin subordinación o jerarquía, los podremos devolver por referencia agrupándolos al final **después de los parámetros de entrada**.
      `void Direccion(double anguloGr, out double sen, out double cos) {...}` 👍
      A partir de **C#7** podemos hacer la siguiente sintaxis mucho más recomendable..
      `static (double sen, double cos) Direccion(double anguloGr) {...}` 👍👍

```
class Ejemplo
{
    static (double sen, double cos) Direccion(double anguloGr)
    {
        double anguloRad = anguloGr * Math.PI / 180;
        return (Math.Sin(anguloRad), Math.Cos(anguloRad));
    }
    static void Main()
    {
        double anguloGr = 75;
        (double sen, double cos) = Direccion(anguloGr);
        Console.WriteLine($"Sen = {sen:F3}");
        Console.WriteLine($"Cos = {cos:F3}");
    }
}
```

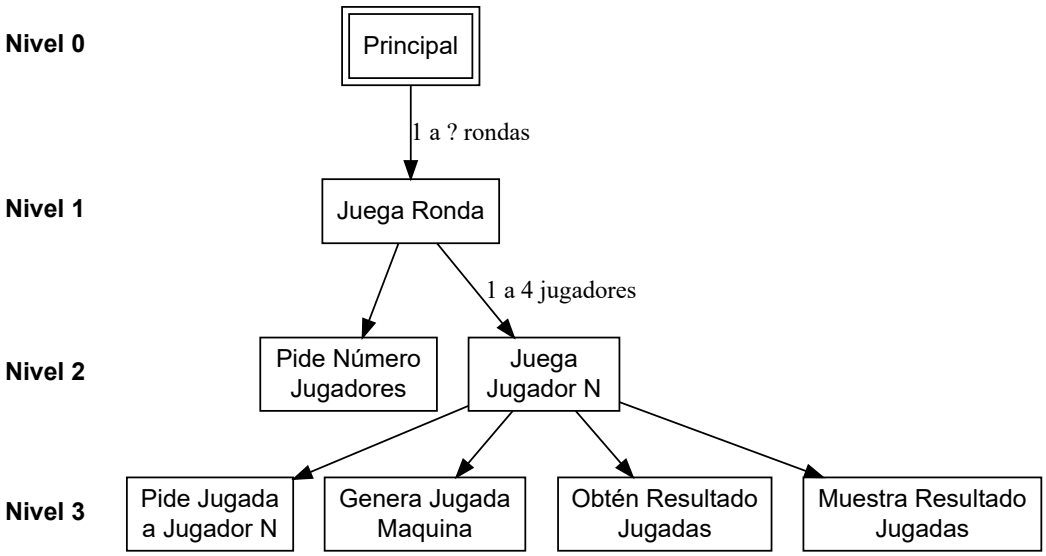
Otras consideraciones del diseño

- Los métodos deben ser cortos. Por tanto, intentaremos que su código no exceda de una pantalla. Si esto sucede, intentaremos modularizarlo o subdividirlo aún más.
- Deberemos agrupar bloques de código relacionado en procedimientos que además de acortar la función nos auto-documentará el código.
- Si el método tiene excesiva carga lógica, salvo en casos muy simples, **evitaremos usar retornos en medio de la lógica de la función**.
  Esta práctica además de liar puede llevar a errores por olvido cerrar flujos de datos, liberación de recursos en lenguajes no gestionados, etc…

Ejemplo práctico de modularización

> Se propone hacer un una versión extendida del programa de **piedra-papel-tijera**.
> En este versión se jugarán **rondas** contra la máquina de **1 a 4** jugadores. De tal manera que el programa nos pedirá el número de jugadores para la ronda de juego y posteriormente para cada jugador, indicará que jugador juega y le pedirá a este una jugada.
> Acto seguido, la máquina hará una jugada aleatoria y nos mostrará el resultado del juego de ese jugador en esa ronda.
> Cuando hayan jugado todos los jugadores de la ronda, la máquina nos pedirá que pulsemos una tecla para jugar otra ronda o ESC para salir.
>
> **Nota:** Para repasar algunas de las estructuras vistas en el anterior tema. Vamos ha hacer que el ordenador filtre las entradas da datos del usuario, nos deberá indicar si es incorrecta, y la volverá a pedir en tal caso.

Una posible descomposición del problema en módulos sería la siguiente...
Fíjate que cada módulo **solo hace una cosa**.
Nota: Aquellos módulos a los que se les pasa el control más de una vez en un bucle, lo indicaremos en la flecha con una etiqueta.



Vamos seguir la modularización propuesta teniendo en cuenta que:

- Empezaremos **con los módulos del nivel más bajo** y nos centraremos en lo que tienen que hacer, que parámetros necesitan y que devuelven. Sin preocuparnos de quien los llama.
- No todos los módulos puede que se sustáncien en un método, ya que puede que no tengan la entidad suficiente para hacerlo o ya estén implementados en el propio lenguaje.
- En este código hemos comentado lo que hacemos por ser un ejemplo aunque en la realidad **no lo haremos**.

1. Partiremos del siguiente esqueleto del programa. Fíjate que hemos definido 3 constantes de texto que nos serán de utilidad durante la implementación del programa, al tener el modificador `cosnt` podrán ser usadas por cualquier método estático de la clase.
   Nota: Todos nuestros módulos irán dentro de la clase `EjemploModularizacion` junto con el Main().

```
using System;

namespace Ejemplo
{
    class EjemploModularizacion
    {
        const string PIEDRA = "PIEDRA";
        const string PAPEL = "PAPEL";
        const string TIJERA = "TIJERA";

        static void Main()
        {

        }
    }
}
```

---

2. Como hemos comentado empezamos por los **módulos de Nivel 3** definiendo su interfaz.
   `[Pide Jugada a Jugador N]` → `static string PideJugada(string jugador)`
   Notemos que la signatura completa, incluido el identificador, me dice lo que va ha hacer:
   - **Identificador**: `PideJugada`
   - **Entrada**: `string jugador` Nombre del jugador al que le voy a pedir la jugada.
   - **Salida**: `string` Texto con la jugada de ese jugador "*PIEDRA*", "*PAPEL*" o "*TIJERA*".

```
static string PideJugada(string jugador)
{
    // Jugada a retornar por el jugador.
    string jugada;

    // Flag que me indicará si el Jugador N ha realizado una jugada correcta.
    bool jugadaCorrecta;

    // Establezco el texto de las jugadas para no tener que repetirlo.
    string opciones = $"{PIEDRA}, {PAPEL}, {TIJERA}";

    // Bucle que me irá pidiendo una jugada mientras no sea correcta.
    do
    {
        // Indico el jugador que tiene que jugar y que me llega como parámetro.
        Console.WriteLine($"Jugando {jugador} ...");
        Console.Write($"\tIntroduce tú jugada ({opciones}): ");
        jugada = Console.ReadLine().ToUpper();
        jugadaCorrecta = jugada == PIEDRA || jugada == PAPEL || jugada == TIJERA;

        // Si voy a volver a pedir la entrada le indico al jugador su error.
        if (!jugadaCorrecta)
            Console.WriteLine($"\t{jugada} no es una jugada correcta. Debe ser {opciones}");
    } while (!jugadaCorrecta);

    return jugada;
}
```

`[Genera Jugada Máquina]` → **`static string GeneraJugadaMaquina()`**

Notemos que la signatura completa, incluido el identificador, me dice lo que va ha hacer:

- **Identificador**: **`GeneraJugadaMaquina`**
- **Entrada**: Nada. Aunque en la mayoría de ocasiones **sería conveniente pasar la semilla con la que generar el número aleatorio**. De esta manera, evitaremos valores repetidos.
- **Salida**: **`string`** Texto con la jugada de la máquina "*PIEDRA*", "*PAPEL*" o "*TIJERA*".

```
static string GeneraJugadaMaquina()
{
    // Habrá muchas formas correctas de implementarlo. Pero por usar la nueva sintaxis de C#8
    // Podemos retornar el resultado de evaluar una expresión switch.
    return new Random().Next(0, 3) switch
    {
        0 => PIEDRA,
        1 => PAPEL,
        2 => TIJERA,
        _ => "Jugada no válida" // Este caso no se podrá dar, aquí deberíamos generar un error.
    };
}
```

`[Obtén Resultado Jugadas]` → **`static string Resultado(string jugadaUsuario, string jugadaMaquina)`**

Notemos que la signatura completa, incluido el identificador, me dice lo que va ha hacer:

- **Identificador**: **`Resultado`** ~~Obten~~Resultado~~Jugadas~~ sería redundante.
- **Entrada**:
  - **`string jugadaUsuario`** Texto con la jugada del usuario.
  - **`string jugadaMaquina`** Texto con la jugada de la máquina.
    Ambos serán "*PIEDRA*", "*PAPEL*" o "*TIJERA*"
- **Salida**: **`string`** Texto el resultado de la jugada a mostrar al usuario "*Empate*", "*He ganado*" o "*He perdido*".

```
static string Resultado(string jugadaUsuario, string jugadaMaquina)
{
    string resultado;
    if (jugadaMaquina == jugadaUsuario)
    {
        resultado = "Empate";
    }
    else switch (jugadaMaquina)
    {
        case PIEDRA when jugadaUsuario == TIJERA:
        case PAPEL when jugadaUsuario == PIEDRA:
        case TIJERA when jugadaUsuario == PAPEL:
            resultado = "He ganado";
            break;
        default:
            resultado = "He perdido";
            break;
    }

    return resultado;
}
```

`[Muestra Resultado Jugadas]` → **`static void WriteLine(...)`**

No se sustanciará en un método pues mostrar por consola ya está definido en **`System.Console`**

3. Como hemos comentado empezamos por el **módulos de Nivel 2** definiendo su interfaz.

`[Pide Número Jugadores]` → **`static int PideNumeroJugadores()`**

Notemos que la signatura completa, incluido el identificador, me dice lo que va ha hacer:

- **Identificador**: `PideNumeroJugadores`
- **Entrada**: Nada.
- **Salida**: `int` Número de jugadores que disputan esa ronda introducida por el usuario.

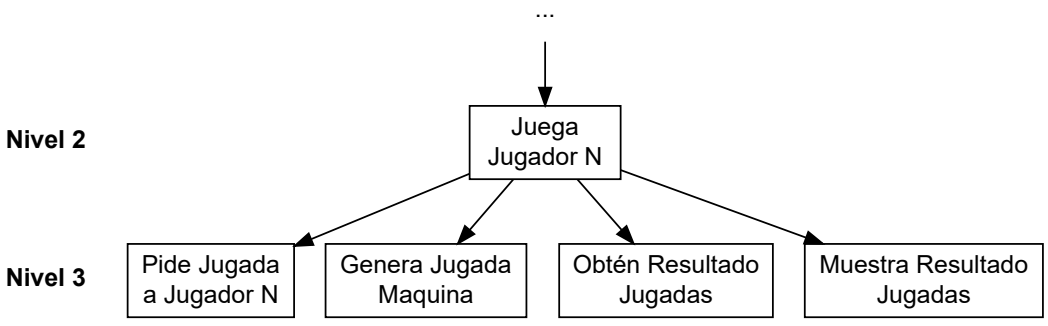
```
// El esquema algorítmico del método es análogo al de PideJugada
static int PideNumeroJugadores()
{
    bool numeroCorrecto;
    int jugadores;
    do
    {
        Console.Write("Introduce cuantos jugadores van a participar (1 a 4): ");
        string entrada = Console.ReadLine();
        numeroCorrecto = int.TryParse(entrada, out jugadores);
        numeroCorrecto = numeroCorrecto && jugadores >= 1 && jugadores <= 4;
        if (!numeroCorrecto)
            Console.WriteLine($"{entrada} no es correcto. Debe ser un valor entre 1 y 4.");
    } while (!numeroCorrecto);
    return jugadores;
}
```

`[Juega Jugador N]` → **`static void Juega(string jugador)`**

Notemos que la signatura completa, incluido el identificador, me dice lo que va ha hacer:

- **Identificador**: `Juega`
- **Entrada**: `string jugador` Nombre del jugador que va a jugar.
- **Salida**: Nada.

```
    static void Juega(string jugador)
    {
        // Juega transfiere el control a los 4 módulos en los que lo hemos subdividido
        // en el orden correcto (Izquierda a Derecha)
        // 1.- Pide Jugada a jugador N
        // 2.- Renera Jugada Máquina
        // 3.- Obtén Resultado Jugadas
        // 4.- Muestra el resultado.

        // Al modularizar el módulo queda legible, autodocumentado y ocupa menos de 10 líneas.
11      string jugadaUsuario = PideJugada(jugador);
        string jugadaMaquina = GeneraJugadaMaquina();
        string resultado = Resultado(jugadaUsuario, jugadaMaquina);
14      Console.WriteLine($"\tYo he jugado {jugadaMaquina}\n"
                        + $"\t{jugador} ha jugado {jugadaUsuario}\n"
                        + $"\t{resultado}\n");
    }
```



4. Por último definimos los módulos del **Nivel 1** y el **Nivel 0** donde estará el módulo principal o Main.

   `[Juega Ronda]` → **`static void JuegaRonda()`**

   Notemos que la signatura completa, incluido el identificador, me dice lo que va ha hacer:

   - **Identificador**: `JuegaRonda`
   - **Entrada**: Nada.
   - **Salida**: Nada.

```
static void JuegaRonda()
{
    // JuegaRonda transfiere el control a los 2 módulos en los que lo hemos subdividido...
    // 1.- Pide Numero Jugadores
    // 2.- Juega Jugador N

    int jugadores = PideNumeroJugadores();
    for (int i = 0; i < jugadores; i++)
        Juega($"Jugador_{i}");
}
```

   `[Principal]` → **`static void Main()`**

   - **Identificador**: `Main` Por la especificación del lenguaje C#
   - **Entrada**: Nada.
   - **Salida**: Nada.

```
static void Main()
{
    do
    {
        Console.Clear();

        // Podríamos pensar que si incluimos el código de JuegaRonda aquí dentro tampoco
        // quedaría un método muy complejo.
        // Pero tendríamos un bucle dentro de un bucle y eso nos está indicando que ese
        // segundo bucle está haciendo un proceso que a su ves se puede encapsular en
        // un módulo.
        JuegaRonda();

        Console.WriteLine("¡¡¡ FIN PARTIDA !!!.");
        Console.WriteLine("Pulsa una tecla para jugar otra ronda. ESC para salir.");
    } while (Console.ReadKey().Key != ConsoleKey.Escape);
}
```