ILERNA Online

Módulo 3
Programación

UF1: Programación estructurada

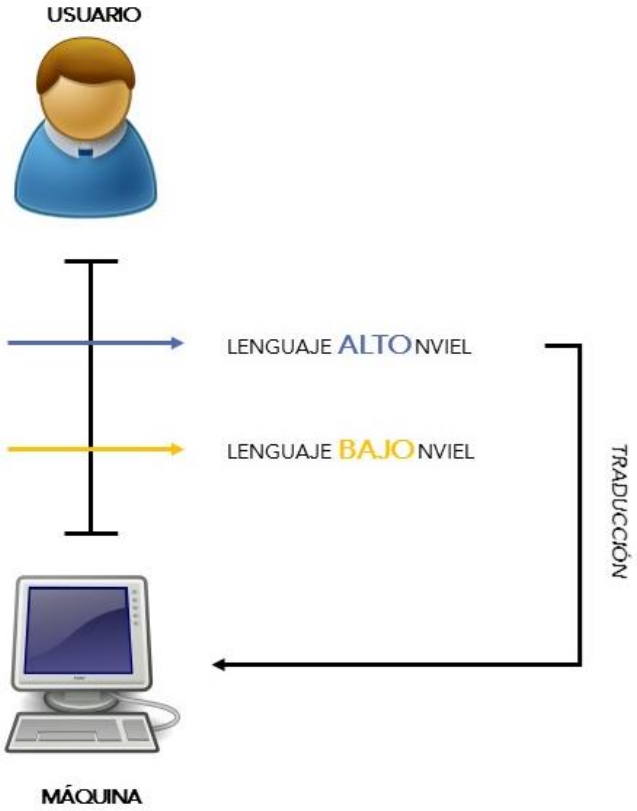
1. Introducción a la programación

Una definición técnica de **lenguaje de programación** sería la de herramienta para transformar un algoritmo en código fuente. Entendiendo algoritmo como una secuencia finita de operaciones que resuelven un problema expuesto.

El proceso de programación es realizar la comunicación entre un usuario y una máquina. Para que se pueda realizar esta comunicación debemos de tener:

- Los dos agentes principales, **receptor** y **emisor**. En este caso es: Usuario y Máquina.
- **Canal**: Para asemejar el ejemplo con el que estamos explicando el proceso de programación, podemos decir que nuestro canal es el teclado.
- **Lenguaje**. Aquí es donde viene la dificultad, ya que los agentes hablan un lenguaje completamente diferente. Para que la comunicación sea fluída debemos de acercar los lenguajes, y, tanto la máquina como el usuario, hacer un esfuerzo para el entendimiento.

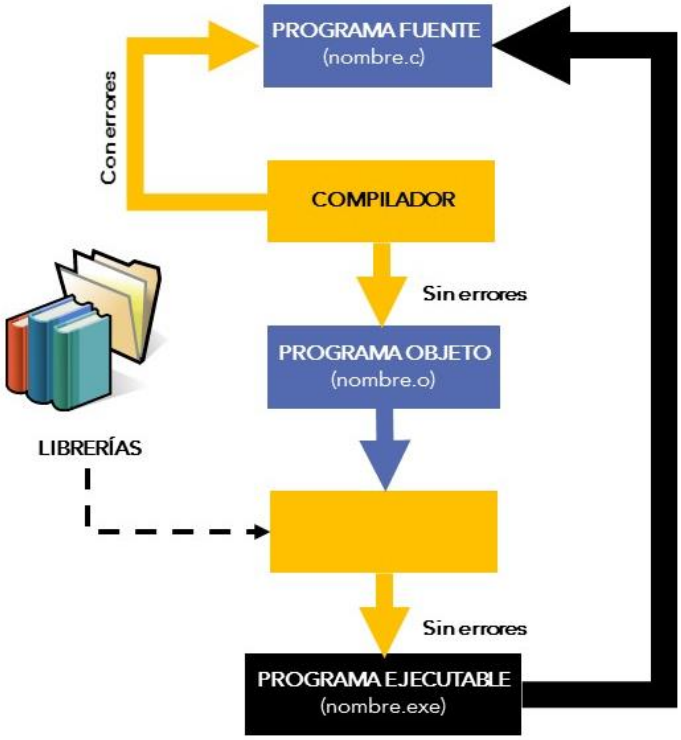
Para solventar el problema anterior, aparecen los lenguajes de programación de dos tipos: **Alto nivel** y **Bajo nivel**. Los lenguajes de alto nivel están más cercano al lenguaje que habla el usuario, mientras que los lenguajes de bajo nivel están más cercano a las estructuras del lenguaje máquina.



Para facilitar nuestro trabajo, implementaremos nuestro código con lenguajes de Alto nivel. Necesitando un proceso de traducción para convertir el programa escrito en lenguaje máquina.

Las caracteristicas del lenguaje de alto nivel son:

- Son totalmente independientes a la máquina, por tanto, muy portables.
- Muy utilizado en el mercado laboral informático.
- Ámplio juego de instrucciones.
- Tanto las modificaciones como las actualizaciones son muy fáciles de realizar.
- Para la tarea de traducción de código, necesitamos un compilador y un enlazador con librerías del propio lenguaje de programación elegido.



A lo largo de esta Unidad formativa hablaremos de la estructura de un programa informático, los elementos principales como son las variables, las constantes y los distintos operadores que podemos usar a la hora de implementar un código fuente.
Continuaremos hablando del control que podemos tener sobre un código a la hora de su proceso de compilación. Este control de ejecución es fundamental cuando construimos cualquier programa en un lenguaje de alto nivel. En una programación estructurada, las instrucciones deben ejecutarse una detrás de otra, dependiendo de una serie de condiciones que pueden cumplir o no. Estas instrucciones pueden repetirse diferentes veces hasta que lleguen a cumplir alguna condición especificada. En este capítulo nos vamos a centrar en el lenguaje de programación C# porque:

- Es un lenguaje sencillo y cómodo de utilizar
- Es recomendable a la hora de crear instrucciones para cualquier ámbito
- Es un lenguaje orientado a objetos
- Utiliza el recolector de basura
- Permite la unificación de tipos

2. Estructura de un programa informático.

En el lenguaje de progrmación C# podemos tener una aplicación formada por uno o varios ficheros con su código fuente correspondiente, teniendo en cuenta que sólo uno de ellos va a ser el principal.
Para todas las pruebas que vamos a ir realizando, vamos a utilizar el programa Visual Studio, y aquí es donde vamos a ir generando los diferentes proyectos.
Cada vez que creemos un proyecto nuevo, el programa va a generar sólo un fichero de código fuente, que es el que va a dar lugar al ejecutable que necesita la aplicación.
Este fichero que se genera debe tener la extensión "cs" que es la utilizada por los ficheros en C#.

2.1. Bloques de un programa informático.

A la hora de implementar un programa en c#, como mínimo, debe tener la parte del programa principal:

CÓDIGO:

```
void main(){
// Bloque de implementación

}
```

Estamos definiendo una función llamada *main* que no tiene ningún argumento y no devuelve parámetro. Las "{}" delimitan el bloque en C#. Este va a ser el cuerpo de nuestra función *main*. Todos los programas van a tener una función ***main()*** que se va a ejecutar al principio.
Podemos definir un programa como una secuencia de instrucciones separadas por punto y coma, que se van a ir agrupando en diferentes bloques mediante llaves.
Las directivas son instrucciones que se les van a pasar al compilador, por si tiene que realizar alguna operación antes de que compilemos el programa.
Las directivas empiezan por el símbolo **#** y no llevan punto y coma al final

Siempre es conveniente ir añadiendo comentarios a nuestro programa sobre los pasos que se van realizando. Los comentarios van con // si el comentario ocupa una sóla línea y /* */ si el comentario ocupa diferentes líneas.
A continuación, vamos a ver cómo sería un programa básico en C# que imprima una frase por pantalla.

CÓDIGO:

```
/*Programa que muestre por pantalla la frase Buenas tardes*/

// Nombre del archivo BuenasTardes.cs

public class Hello1
{
   public static void Main()
   {
      System.Console.WriteLine("Buenas Tardes");
   }
}
```

La primera parte, con los símbolos /* y*/ es un comentario en el que se explica lo que hay que hacer en el ejercicio.
A continuación, se declara la función principal int main(), que no devuelve nada.
La sentencia que está separada entre llaves, indica que se muestre por teclado la frase "Buenas tardes". Mediante el operador System.Console.WriteLine ("lo que se va a mostrar").

Las operaciones de E/S se relizan de forma diferente según el lenguaje de programación que se utilice. En C necesita funciones declaradas en "stdio.h", miestras que en C++ utiliza *stream*, que es el que se refiere al flujo de la información mediante E/S. En C# se utiliza la sentencia System.Console.WriteLine("Mensaje");

2.2. Proyectos de desarrollo de aplicaciones. Entornos integrados de desarrollo.

A la hora de desarrollar un programa, lo primero que heremos es crear un nuevo proyecto.
En la página principal, seleccionamos Nuevo Proyecto.Nos ofrece la posibilidad de crear el proyecto en Visual Basic, C++ y C#. En este caso, elegiremos C# y le ponemos un nombre, por ejemplo, proyecto1. Cuando tengamos todos los pasos, aceptamos y se nos creará nuestro "proyecto1".

2.3. Variables. Usos y tipos.

Todo programa informático suele tener como elemento principal las variables. Estas podemos definirlas como un espacio de memoria, identificada por un nombre, que pueden ir variando a lo largo de un programa, por lo que son inestables. Cuando hablamos de variables, nos referimos al símbolo que vamos a utilizar para identificar un elemento de un conjunto determinado.
A la hora de desarrollar un programa, las variables se utilizan para almacenar unos datos determinados. Se pueden nombrar a lo largo de todo el programa.
Las variables tienen como ámbito de trabajo el bloque donde han sido definidas. Se definen al comienzo del bloque y, al salir de él, se destruyen. Si queremos utilizarlas una vez que hayamos salido del bloque, podemos hacerlo utilizando la palabra *extern*, esta cláusula hace que la variable sea vista por todo el programa informático. A este tipo de varible se le llama **Variable global** y **no aconsejamos su uso**.

- Cada variable, va a almacenar un tipo de dato como:

Tipo	Espacio que ocupa en memoria (bytes)
Int Números enteros	2
Char Uno o varios caracteres	1
Float Números decimales	4
Double Cadenas numéricas	8

Para definir una variable, necesitamos conocer primero el tipo de datos que va a almacenar, y, a continuación, el nombre que le vamos a asignar (es recomendable que tenga relación con el ejercicio que estemos desarrollando)
Para identificar a una variable, y que tenga un identificador válido por el compilador, debemos de seguir una serie de normas:

- Debe de estar formados a partir de alfabeto (no podemos usar 'ñ' ni palabras acentuadas), dígitos (0..9) y el subrayado (_).
- No se puede comenzar con un dígito.
- Distingue entre mayúsculas y minúsculas.

Debe cumplir la siguiente sentencia:

CÓDIGO:

```
[cualificador] <tipo> <nombre_variable>;
```

En pseudocódigo sería:

CÓDIGO:

```
entero numero;
```

Y en C# lo pondríamos de la siguiente forma:

CÓDIGO:

```
int num;
```

Estamos definiendo una variable de tipo entero denominada num.

A las variables también se le puede asignar un valor:

CÓDIGO:

```
int num=5;
```

Definimos una varibale de tipo entero, denominada ***num*** y le asignamos el valor de 5.
Cuando definimos una variable, es importante que sepamos cuánto espacio nos va a ocupar para intentar seleccionar qué tipo de variable es la más adecuada.

Podemos encontrar dos tipos de variables diferentes: globales y locales.

- **Local**: si está definida dentro de un bloque.

- **Global**: si está definida fuera de un bloque, pero se puede acceder a ella.

2.4. Constantes. Tipos y utilidades.

En el apartado anterior hemos hablado de las variables como espacio de memoria, donde se almacenan datos que se pueden variar a lo largo del programa. Ahora trabajaremos con los espacios de memoria donde **NO** se pueden alterar su contenido a lo largo del programa, las **constantes**.
Son muy parecidas a las variables como hemos indicado anteriormente, aunque añadiendo la palabra ***const***. Se definen de la siguiente forma:

CÓDIGO:

```
const <tipo> <nombre_variable>;
```

Como, por ejemplo:

CÓDIGO:

```
const int num;
```

Declaramos una constante de tipo entero, que denominamos: *num*.
A continuación, vamos a declarar una constante llamada "días_semana", con el valor: 7, ya que todas las semanas tienen 7 días:

CÓDIGO:

```
class Calendar1
{
    public const int dias_semanas = 7;
}
```

2.5. Operadores del lenguaje de programación.

Para poder operar con las diferentes variables, podemos utilizar una serie de operaciones que detallamos en la siguiente tabla.

Operadores aritméticos	
+	Suma aritmética de dos valores.
-	Resta aritmética de dos valores
*	Multiplicación aritmética de dos valores
/	División aritmética de dos valores
%	Obtiene el resto de la división entera.
Potencia(a,b)	Potencia a eleva a b
Raíz(a,b)	Raíz a de b

Operadores incremento y decremento	
++	Incremento en 1 del operando
--	Decremento en 1 del operando

Operadores relacionales	
>	Mayor
<	Menor
>=	Mayor o igual
<=	Menor o igual
==	Igual
!=	Distinto
=	Asignación

2.6. Conversiones de tipos de clase.

Como hemos visto anteriormente, un programa consta de numerosas variables de diferentes tipos, que suelen declararse al comienzo de éste. Sin embargo, conforme vamos avanzando en el programa, realizando

operaciones o almacenando resultados, es posible que tengamos que ir adaptando las variables a los nuevos tipos de datos que estemos utilizando. Esta conversión recible el nombre de ***casting***.

CÓDIGO:

```
double resultado;
int numero1=3, numero2=9;
resultado = numero1;
numero2 = (int)resultado;
```

En este fragmento de código declaramos una variable denominada "**resultado**" de tipo "**double**" y otras dos de tipo entero: "**numero1**" y "**numero2**". Le asignamos respectivamente, los valores 1 y 4.
Al decir que "resultado=numero1" estamos realizando una operación de asignación de un valor entero a una variable double. No se produce error, porque el programa interpreta la variable "resultado1=3.0".
En la última línea, si no tuviéramos el paréntesis (*int*) asignaríamos un valor *double* a una variable entera, y, por tanto, el compilador no nos dejaría, ya que el tipo de datos *double* es decimal, y el tipo de datos *int* es entero, por tanto, desperdiciaríamos la parte decimal de la variable. Para solventar este problema, hacemos uso del "*casting*" es decir conversión de tipo, y, por tanto, ponemos entre paréntesis el tipo de datos al que queremos convertir el valor, para que tenga el mismo tipo que la varible destino. En este caso, entero. Con esto estamos diciendo que el valor de resultado va a ser un número entero pero NO estamos diciendo que nuestra variable resultado deje de ser un double.

2.7. Comentarios al código.

A lo largo de esta Unidad formativa, nos estamos introduciendo en la creación por parte de un programador de código fuente con una finalidad en concreto. No debemos pasar por alto el hecho de que, además de realizar programas, lo debemos de hacer de la forma más óptima y ordenada posible. En el ámbito de la organización entra en escena el concepto de los comentarios. Herramienta disponible en el compilador, para que el programador pueda hacer anotaciones en el propio código sin que sea procesado por el compilador.
Anotaciones que deben de esclarecer el propio código y ayudar a entender las sentencia del programa.

En el lenguaje de programación C#, está permitido hacer dos tipo de comentarios:

- **Comentarios de una línea**: Son frases cortas, y, por tanto, solo pueden ocupar una sola línea del código. Debemos de escribir // para comenzar con dichas anotaciones.
- **Comentarios *multiline* o mult-ilíneas**: Se utilizan para hacer una explicación mucho mas detallada del código en cuentión, o también podemos hacer uso de este tipo de comentarios cuando deseamos que una parte de código no sea procesada por el compilador. En este caso debemos de englobar el texto entre los caracteres /* texto */.

El uso de los comentarios en un código es muy útil, y, por tanto, aconsejamos a los programadores a que implementen programas con tantos comentarios como sea posible.

3. Programación estructurada.

La programación estructurada es la manera que tenemos de escribir o diseñar un programa de una forma clara y concisa. Vamos a hacerlo siempre siguiendo tres estructuras básicas: **secuencial, selectiva e iterativa**. No se permite utilizar instrucciones de trasferencia incondicional (*GOTO*), ya que no son necesarias.
A finales de los años setenta fue cuando surge una nueva forma de desarrollar los programas que, además de conseguir el correcto funcionamiento de éstos, también facilitaba su comprensión gracias a la forma en la que estaban escritos. Estos programas son más fiables y eficientes.
En esta época, fue cuando Alfred Dijkstra comprueba que todo programa se puede escribir utilizando sólo tres instrucciones de control:

- Secuencia de instrucciones
- Instrucción condicional
- Iteración o bucle de instrucciones

Utilizando estas tres estructuras (Teorema de Dijkstra) se pueden crear programas informáticos, a pesar de que los lenguajes ofrecen un repertorio de instrucciones bastante mayor.

3.1. Fundamentos de programación.

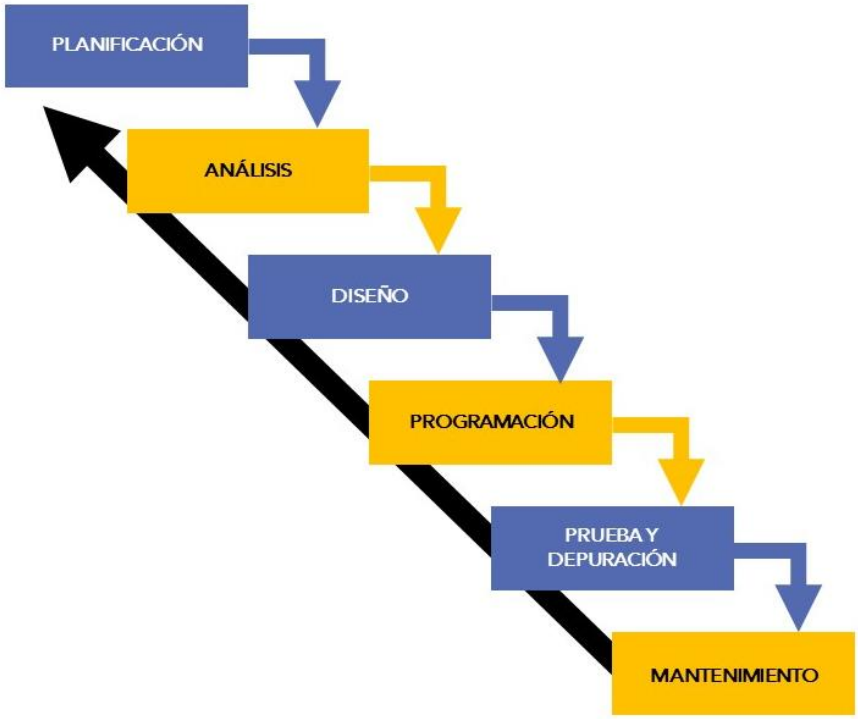
Cuando hablamos de programar debemos tener una idea previa general de a qué nos referimos. En este caso, podemos definir el concepto de programar como: **decirle a una máquina o qué debe realizar en el menor tiempo posible**. Ahora bien, para llegar a buen fin, es conveniente especificar la estructura y el comportamiento de un determinado programa, probar que realiza la tarea que le ha sido asignada de forma correcta y que ofrece un buen rendimiento. El programa se encarga de transformar le entrada en salida.

Mientras que, el algoritmo es la secuencia de los pasos y las distintas operaciones que debe realizar el programa para conseguir resolver un problema planteado.
Dicho programa debe implementar el algoritmo en un lenguaje de programación especificado previamente.
En resumen, podemos señalar que la programación sólo es una etapa más en todo el proceso de desarrollo que existe a la hora de resolver un problema.

3.2. Introducción a la algoritmia.

Las personas, estamos acostumbradas a obedecer una serie de órdenes secuenciales y lógicas. Como tenemos costumbre de hacerlas de forma automática, no nos damos cuenta de que realmente son órdenes enviadas a nuestro cerebro para, posteriormente, ejecutarlas. Lo mismo pasa con los ordenadores, que realizan diferentes tareas siguiendo una serie de pasos lógicos, ya que están programados (mediante algoritmos) para solucionar distintos problemas en un lenguaje de programación determinado.
Es fundamental que logremos desarrollar y utilizar nuestra mente en forma de algoritmo, ya que, como la vida misma, la programación consiste en buscar la solución más óptima para un problema determinado. Todo esto se consigue mediante al diseño, creación e implementación de un algoritmo.

3.3. Diseño de algoritmos.



La imagen representa el ciclo de vida de un programa informático, de manera que las flechas indican el orden de relización de cada etapa. Este modelo apenas se utiliza, pero sigue siendo de vital importancia a la hora de identificar las etapas principales y el orden en el que se realizan.

- Análisis de requisitos

A partir de las necesidades del usuario o del programa planteado, se decide qué es lo que hay que hacer para llegar a conseguir una solución óptima, y se genera un documento de requisitos.

- Diseño de la arquitectura

Se hace un estudio para ver los distintos componentes que van a formar parte de nuestro programa (módulos, subsistemas, etc). Y se genera un documento de diseño. Esta fase se va a revisar todas las veces que sea necesario hasta que estemos seguros de cuál va a ser la mejor solución.

- Etapa de implementación o codificación

En esta etapa vamos a pasar a codificar las aplicaciones que hemos elegido en la etapa anterior, empleando el lenguaje de programación en el que estemos trabajando. El resultado que vamos a obtener va a ser el código fuente.

- Pruebas de integración

Se hace funcionar la aplicación completa, combinando todos sus módulos. Se realizan ensayos para comprobar que el funcionamiento de conjunto cumple lo establecido en el documento de diseño.

- Pruebas de validación

Como paso final de la integración se realizan nuevas pruebas de la aplicación en su conjunto. En este caso, el objetivo es comprobar que el producto desarrollado cumple con lo establecido en el documento de requisitos, y si satisface, por tanto, las necesidades de los usuarios en la medida prevista.

- Fase de mantenimiento

Revisar todo el proceso anterior e ir actualizando o modificando los cambios oportunos en las etapas anteriores.

3.4. Prueba de programas.

Una vez implementado y compilado el código de nuestro algoritmo, debemos ponerlo en marcha y comenzar la etapa de "*testing*", plan de prueba o prueba de programa.
Este proceso de prueba de programa se lleva a cabo de manera automática o manual, persiguiendo los siguientes objetivos.

- Comprobación de los requisitos funcionales y no funcionales del programa.
- Probar todo tipo de casos para detectar algún tipo de anomalías en su ejecución.

El plan de prueba consta de muchas etapas, ya que, después de implementar el código, compilar, ejecutar y probar si existiera algún tipo de fallo en el programa, tendríamos que volver a empezar con el nuevo código modificado. Por tanto el tiempo destinado a esta etapa es bastante considerable

3.5. Tipos de datos simples y compuestos.

C# es un lenguaje de programación en el que cada variable, constante, atributo o valor que devuelve una función, se encuentra establecido en un rango de elementos ya definidos. Podemos diferenciar entre:

- TIPOS SIMPLES

A la hora de seleccionar un determinado tipo debemos tener en cuenta: el rango de valores que puede tomar, las operaciones a realizar, el espacio necesario para almacenar datos.
Debemos tener en cuenta que el tipo de datos simple no está compuesto por otros tipos, y que contienen un valor único.

- **Tipos simples predefinidos.** Entre sus propiedades más importantes podemos destacar que son indivisibles, tienen existencia propia, y permiten operadores relacionales.
  Se utilizan sin necesidad de ser definidos previamente. En pseudolenguaje, se corresponden de la siguiente forma:
  - **Natural.** Números naturales (N).
    unsigned int, insigned short, unsigned long.
  - **Entero.** Números enteros (Z)
    int, long, short.
  - **Real.** Números reales (R)
    float, double
  - **Carácter.** Caracteres (C)
    char
  - **Lógico.** Booleanos (B)
    bool

- **Tipos simples definidos por el usuario.**
  - **Tipos enumerados.** Ofrecen la posibilidad de definir nuevos tipos simples, aunque de cardinalidad reducida. Estos tipos se declaran en el apartado tipos de un programa de la siguiente forma:

CÓDIGO:

```
ENUM {Id1, Id2, Id3} IdTipoEnumerado

public class EnumTest
{
    enum                              Dias                              {
Domingo,Lunes,Martes,Miercoles,Jueves,Viernes,Sabado };

    static void Main()
    {
        int x = (int)Dias.Domingo;
        int y = (int)Dias.Viernes;
        Console.WriteLine("Domingo = {0}", x);
        Console.WriteLine("Viernes = {0}", y);
    }
}
/* Salida:
   Domingo = 0
   Viernes = 5
*/
```

- **TIPOS COMPUESTOS ó ESTRUCTURADOS**

Se crean mediante la unión de varios tipos (simples o compuestos).

- **Vectores.** Se utilizan para agrupar varias variables de un mismo tipo con un nombre único. También son llamado *array unidimensional*, es decir, de una dimensión. Esa anotación también nos dice la cantidad de indicadores de posición que necesitamos para acceder a un elemento de la tabla. En este caso solo necesitamos una posición.

Sintaxis:

CÓDIGO:

```
<tipo> [] <nombre> = new <tipo> [<tamaño>];
```

Por ejemplo:

CÓDIGO:

```
int [] v = new int [10];
```

Nos estamos declarando 10 enteros en un vector al que hemos llamado v

Para poder acceder a cada uno de ellos lo haremos de la siguiente forma, siempre recordando que la primera posición de todo *Array* es el 0.

v[0] → Primer entero del vector
v[1] → Segundo entero del vector
v[2] → Tercer entero del vector
v[3] → Cuarto entero del vector
…
v[9] → Último entero del vector

- **Matrices.** Unión de varios vectores de cualquier tipo simple (enteros, reales, …). También la podemos ver como un array bidimensional, por tanto ese datos nos indica que necesitamos dos indicadores de posición para acceder al elemento. La primera posición de un array es el 0.

Sintaxis:

CÓDIGO:

```
Tipo [,] Nombre= new Tipo  [filas, columnas];
```

Por ejemplo:

CÓDIGO:

```
int [,] matriz = new bool[2,3];
```

En este caso, no estamos declarando una matriz denominada matriz, de tipo entero, que consta de 2 filas y 3 columnas.

Accedemos a cada uno de sus elementos de la forma que se indica a continuación:
matriz[0,0] → Elemento correspondiente a la primera fila, primera columna.
Matriz[0,1] → Elemento correspondiente a la primera fila, segunda columna.
Matriz[0, 2] → Elemento correspondiente a la primera fila, tercera columna.
Matriz[1, 0] → Elemento correspondiente a la segunda fila, primera columna.
Matriz[1,1] → Elemento correspondiente a la segunda fila, segunda columna.
Matriz[1,2] → Elemento correspondiente a la segunda fila, tercera columna.

00	01	02
10	11	12

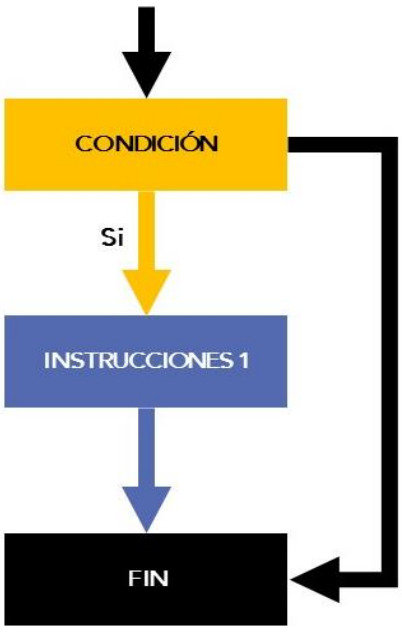
Matriz de 6 elementos (2 filas por 3 columnas)

3.6. Estructuras de selección.

Las estructuras de selección son aquellas que permiten ejecutar una parte del código dependiendo de si cumple una determinada condición. El mínimo bloque que debe tener esta estructura sería de la siguiente forma:

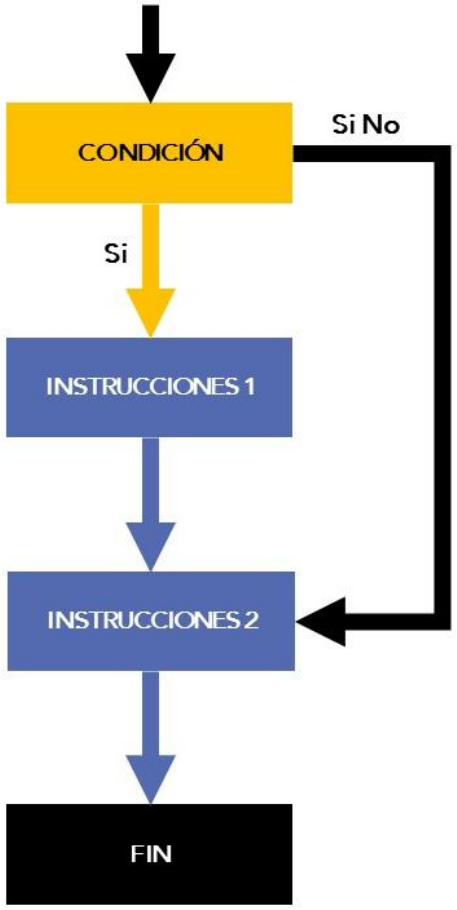
- **Bloque mínimo**

```
SI CONDICIÓN ENTONCES:
        INSTRUCCIONES_1
FIN SI;
```



- **Si/ Sino**

```
SI CONDICIÓN ENTONCES:
        INSTRUCCIONES_1;
SINO
        INSTRUCCIONES_2;
FIN SI;
```

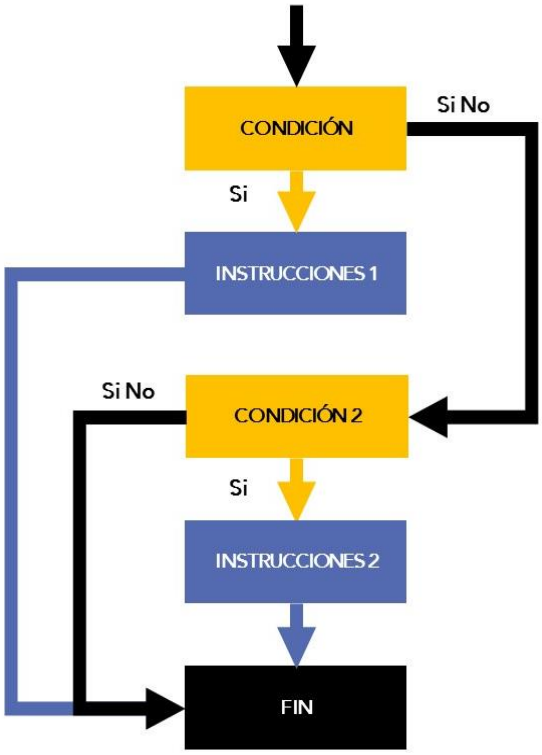


En este caso,
si CONDICION ES CIERTA, ejecutaremos lo que hay en INSTRUCCIONES1 y saltaremos hasta el FINSI.
Si CONDICION ES FALSA, ejecutaremos lo que hay en INSTRUCCIONES2.
En un BLOQUE SI.

Sólo puede haber un único **SINO** (o ninguno).

- Si/ Sino Si

```
SI CONDICIÓN ENTONCES:
        INSTRUCCIONES_1;
SINO SI CONDICIÓN_2:
        INSTRUCCIONES_2;
FIN SI;
```



En éste caso,
si CONDICION ES CIERTA, ejecutaremos lo que hay en INSTRUCCIONES 1 y saltaremos hasta el FINSI, por tanto, NO MIRAREMOS SI CONDICION2 ES CIERTA O NO.

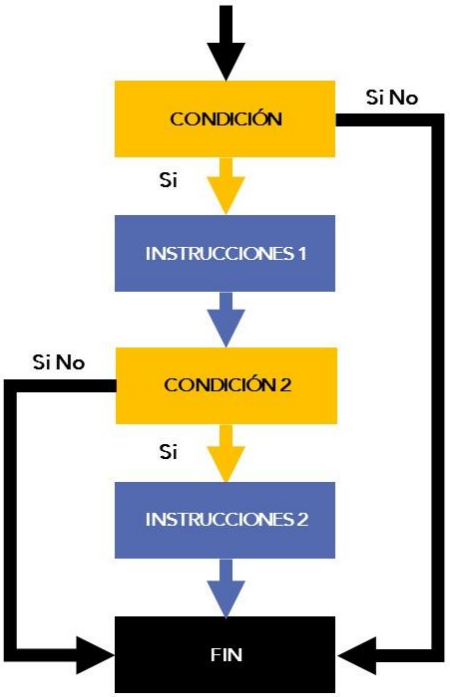
Si CONDICION ES FALSA, miraremos si CONDICION2 ES CIERTA, en cuyo caso ejecutaremos lo que hay en INSTRUCCIONES2 (Fijaros que sólo lo miramos en caso que CONDICION1 sea FALSA), y saltaremos hasta el FINSI.

En caso que CONDICION2 ES FALSA, entraremos en SINO, ejecutaremos INSTRUCCIONES3, y saldremos del BLOQUE SI.

En un bloque de este tipo, puede haber varios SINOSI, y un solo sino (o ninguno).

- Si (es) Anidados

```
SI CONDICIÓN ENTONCES:
        INSTRUCCIONES_1;
        SI CONDICIÓN_2:
                INSTRUCCIONES_2;
        FIN SI;
FIN SI;
```



Los distintos operadores que se pueden utilizar al realizar en estas estructuras serían:

- **Operadores de comparación**. Por ejemplo, cuando 1==1 → *True* y cuando 1!=1 → *False*.

<	Menor que (Estricto)
>	Mayor que (Estricto)
<=	Menor o igual que
>=	Mayor o igual que
==	Igual a
!=	Distinto a

- **Operadores lógicos.**

&&	AND: Ambas condiciones deben cumplirse para entrar en el bloque SI
||	OR: Es suficiente con que una de las condiciones se cumplan para poder entrar en el bloque SI

Ejemplo del "y" lógico:

```
SI CONDICIÓN_1 && CONDICIÓN_2 ENTONCES:
FIN SI;
```

Ejemplo del "o" condicional:

```
SI CONDICIÓN_1 || CONDICIÓN_2 ENTONCES:
FIN SI;
```

En todos los ejemplos, donde pone **INSTRUCCIONES**, puede haber:

1. Una o muchas instrucciones, como leer, escribir, hacer operaciones…
2. Otro Bloque SI /SI-SINO / SI-SINOSI entero, es decir, con su SI y su FINSI.

3.7. Estructuras de repetición.

Las estructuras en repetición, siempre se ejecutan mediante el uso de **bucles,** que va a ser el que permite que una o varias líneas de código se ejecuten de forma repetida. Se van a ir repitiendo miestras que se cumpla una determinada condición de salida.
Existen diferentes estructuras de repetición:

- **Mientras**

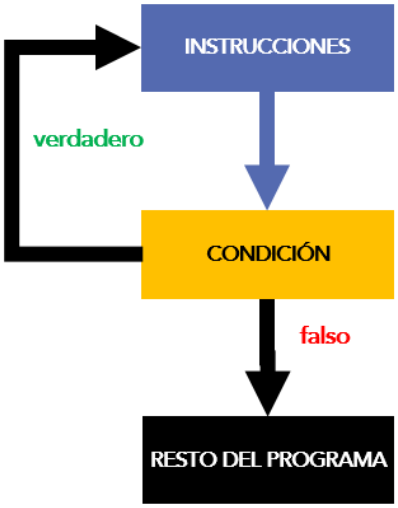
Mientras que se cumpla una condición, el código incluido dentro del bucle se repite. La condición se evalúa al principio, por lo que puede que no llegue a ajecutarse nunca.


```
Mientras CONDICIÓN Hacer
```

```
  Instrucción1;
  Instrucción2;
  …
  InstrucciónN;
  ModificarCondición;
FinMientras
```

- **Hacer…mientras**

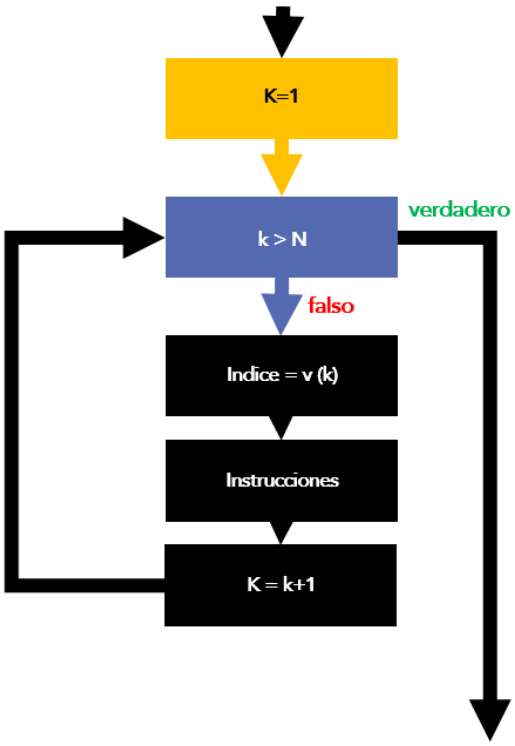
Mientras que se cumpla una condición, el código incluido dentro del bucle se repite. La condición se evalúa al final, por lo que, como mínimo, se va a ejecutar una vez.




```
Hacer
  Instrucción1;
  Instrucción2;
  …
  InstrucciónN;
  ModificarCondición;
Mientras CONDICIÓN
```

- **Para**

Mientras que se cumpla una condición, el código incluido dentro del bucle se repite tantas veces como indique el contador que se irá modificando en cada sentencia. La condición se evalúa al principio, por lo que puede que no llegue a ajecutarse nunca.




```
Para CONDICIÓN = v
  Instrucción1;

Instrucción2;
  …
  InstrucciónN;
  ModificarCondición;
```

3.8. Estructuras de salto.

Las estructuras de salto son todas aquellas que paran (de diferentes formas), la ejecución de alguna de las sentencias de control de nuestro programa.

Break	Se encuentra al final de la sentencia *switch*. Interrumpe el bucle indicando que debe continuar a partir de la siguiente sentencia después del bloque del ciclo. En una sentencia *switch* con varios casos, garantiza que sólo se ejecuten los que están relacionados
continue	Se utiliza sólo en sentencias repetitivas con bucles *for*, *while* y *do… while*. Transfiere el control a la siguiente iteración
Return	Esta estructura de salto obliga a que finalice la ejecución de una determinada función. Normalmente se utiliza para devolver el control a la función de llamada.

3.9. Tratamiento de cadenas.

Para la representación de cadenas de caracteres se utiliza el tipo de datos string. En programas como C y C++ no existe este tipo de datos, por lo que se suele representar mediante el uso de un array de caracteres.
En C#, la representación del tipo de datos string sería de la siguiente forma:

CÓDIGO:

```
string frase = "Buenos días";
Console.WriteLine("El carácter que ocupa la posición 5 es: {0}, frase[5]");
//Operación Permitida

frase[5] = 'S';
//Operación NO permitida
```

Siempre que queramos escribir en nuestro código combinaciones alfanuméricas, irán entre comillas dobles (“”), ya que el compilador las va a tomar como un tipo *string*. Estas variables *string* se pueden inicializar a partir de una tabla de caracteres creados previamente.

CÓDIGO:

```
char []letras = char[] {'B','u','e','n','o','s'};
string frase=new string(letras);
```

La palabra ***new*** crea un objeto tipo *String* al que pasamos el parámetro **letras** como una tabla de caracteres.

A continuación, vamos a ver los principales métodos de los que dispone string:

Length	Devuelve número de caracteres.
ToCharArray()	Devuelve array de carcteres. En cada posición, hay un carácter de la cadena almacenada. Convierte un *string* a *array* de caracteres.
SubString()	Extrae parte de una cadena. Método que puede ser sobrecargado indicando su inicio y su fin. *SubString*(int inicio) ó *SubString*(int inicio, int tamaño).
CopyTo()	Copia un número de caracteres especificados a una determinada posición del *string*.
CompareTo()	Compara la cadena que contiene el string con otra pasada por parámetro. Si son iguales, devuelve 0 y sino, valores positivos o negativos dependiendo de si están antes o después alfabéticamente.
Contains()	Si la cadena que se le pasa por parámetro forma parte del *string*, devuelve un booleano.
IndexOf()	Si aparece un carácter especificado en el *string*, devuelve el índice de la posición de la primera vez que aparece.
Insert()	Inserta una cadena de caracteres en el *string*.
Trim()	Elimina las apariciones de un carácter especificado en el inicio y fin de la cadena de caracteres.
Replace()	Sustituye un *string* o carácter por otro.
Remove()	Elimina una cantidad de veces un carácter determinado.
Split()	Ofrece la posibilidad de separar en varias partes una cadena de caracteres. Este método devuelve un array de string.
ToLower()	Devuelve la copia del string que hace la llamada en minúsculas.
ToUpper()	Devuelve la copia del string que hace la llamada en mayúsculas.

3.10. Depuración de errores.

Una vez que llegamos a la etapa de depuración de nuestro programa, nuestro objetivo será descubrir todos los errores que existan e intentar solucionarlos de la mejor forma posible. Tenemos tres tipos de errores diferentes que podemos encontrar:

- **Compilación**: errores en el código.
- **Tiempo de ejecución**: los que producen un fallo a la hora de ejecutar el programa.
- **Lógicos**: los que no dejan que el programa ni compile ni se ejecute. Pueden devolver resultados erróneos o diferentes a los esperados.

Cuando queramos depurar errores, tenemos la opción, en cualquier momento de nuestro programa, poner un punto de interrupción en una determinada línea de código. Para ello, presionamos F9. Si queremos ejecutar la aplicación en el depurador *Visual Studio*, podemos hacerlo presionando F5. La aplicación se detiene en la línea, y podremos examinar cuánto valen las variables para ir realizando un seguimiento de las mismas, podemos comprobar cuándo finalizan los bucles, entre otras cosas. Podemos añadir puntos de interrupción adicionales tecleando F10 (paso a paso).

- **Errores de compilación**. Estos errores impiden la ejecución de un programa. Mediante F5 ejecutamos un programa. Inicialmente se compila el programa, y, si el compilador Visual Basic encuentra cualquier cosa que no entiende, lanza un error de compilación.
  Casi todos los errores que ocasiona el compilador se producen mientras escribimos el código.
- **Errores en tiempo de ejecución**. Aparecen mientras se ejecuta el programa, normalmente cuando se pretende realizar una operación que no se lleva a ninguna solución. Como, por ejemplo, cuando se pretende dividir por cero.
- **Errores lógicos**. Impiden que se lleve a cabo lo que se había previsto. El código se puede compilar y ejecutar sin problemas, pero, en el caso de que compile, devuelve algo que no era la solución que se esperaba. Como inicialmente el programa cuando se ejecuta no da error, estos errores son los más difíciles de corregir, ya que debemos encontrar el fallo en un programa que funciona, aunque de una forma distinta a la que debería.

3.11. Documentación de programas.

Una vez que finaliza nuestro proceso de compilación y ejecución, debemos hacer una memoria para que quede registrado todo el desarrollo que hemos llevado a cabo, los fallos que ha presentado y cómo hemos conseguido solventarlos.
El diseño de los archivos que vamos a entregar puede venir condicionado por un gran número de factores que deben estar resueltos ya que lo que entreguemos, debe funcionar.
Visual Studio es un proyecto que debe estar asociado al principal. Puede existir más de un proyecto de desarrollo y cada proyecto puede ser utilizado para entregar más de una aplicación.
Permite configurar los archivos que se deben entregar y, además, ejecuta y genera un *wizard* de instalación.
También ofrece la posibilidad de configurar directorios de destino, el registro de Windows y el escritorio de usuario.

3.12. Entornos de desarrollo de programas.

Una vez que diseñamos un programa, podemos denominar **como entorno de desarrollo integrado** al entorno de programación que hemos utilizado para su realización: editor de código, compilador, depurador e interfaz gráfica (GUI).
Los IDE son utilizados por distintos lenguajes de programación como*: C++, PHP, Python, Java, C#, Visual Basic*…entre otros. En algunos casos, funcionan como sistemas en tiempo de ejecución, en los que se permite el uso de un lenguaje de programación de forma interactiva sin necesidad de utilizar archivos de texto.
A modo de ejemplo, podemos señalar algunos entornos integrados de desarrollo como: *Eclipse, Netbeans, JDeveloper de Oracle, Visual C++*, etcétera.
Los IDE deben cumplir con una serie de características para su correcto funcionamiento, como, por ejemplo:

- Multi-plataforma.
- Actúan como soporte para diferentes lenguajes de programación.
- Reconocen sintaxis.
- Integrados con sistemas de control de diferentes versiones.
- Depurador.
- Debe permitir importar-exportar proyectos.
- Diferentes idiomas.

- Manual de ayuda para el usuario.

Las diferentes ventajas que ofrecen los IDEs son:

- Tienen una baja curva de aprendizaje.
- Uso óptimo para usuarios que no son expertos.
- Formateo de código.
- Uso de funciones para renombrar funciones y variables.
- Creación de proyectos.
- Herramientas para extraer partes de código.
- Errores y *warnings*.

UF2: Diseño Modular

1. Programación modular.

Hasta ahora hemos estudiado el desarrollo de un programa como un ente único compuesto por líneas que se ejecutan de forma secuencial, sin embargo, un acercamiento más próximo a la realidad sería el paradigma conocido como programación modular.

1.1. Concepto.

La **programación modular** consiste en dividir el problema original en diversos subproblemas, que se pueden resolver por separado, para, después, recomponer los resultados y obtener la solución al problema.

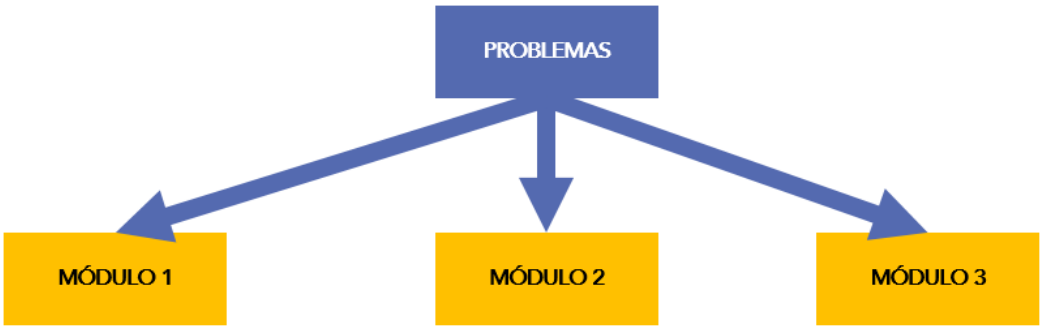
1.2. Ventajas e inconvenientes.

A continuación, vamos a desarrollar cuáles son las ventajas e inconvenientes más importantes a la hora de trabajar con diseño modular.

- **Ventajas**
  - Facilita el mantenimiento, la modificación y la documentación
  - Escritura y testing
  - Reutilización de módulos
  - Independencia de fallos
  - Sin embargo, también presenta una serie de inconvenientes entre los que se encuentra:
  - Separación de módulos.
  - Memoria y tiempo de ejecución

1.3. Análisis descendente (*top down*).

El diseño descendente es una técnica que permite diseñar la solución de un problema con base en la modularización o segmentación, dándole un enfoque de arriba hacia abajo (*Top Down Design*). Esta solución se divide en módulos que se estructuran e integran jerárquicamente.



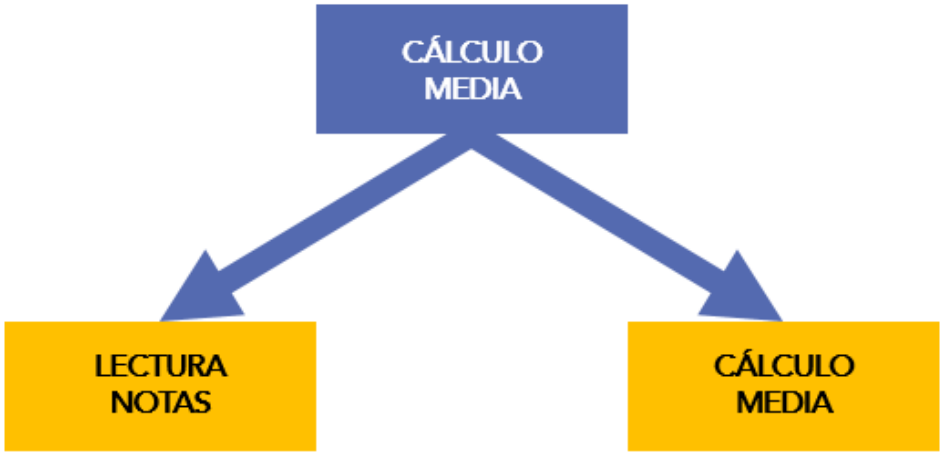
1.4. Modulación de programas. Subprogramas.

Podemos denominar los módulos como subprogramas, y éstos son las diferentes partes del problema que pueden resolverse de forma indepentiente. Los módulos, al ser independientes unos de otros, permiten que podamos centrarnos en una de sus partes (sin tener en cuenta el resto), y también ofrecen la posibilidad de que se puedan utilizar las soluciones obtenidas en otras partes del programa.
Cada módulo codificado dentro del programa como subprograma se puede definir como una parte del código independiente, que realiza una tarea asignada.
A continuación, vamos a ver a modo de ejemplo un programa que devuelva la nota media de un alumno:

CÓDIGO:

```
//Declaramos el array de enteros que almacenará las notas del alumno
int [] notas = new int [ 5 ];
/*Mediante un bucle for recorremos el array almacenando
* en su intenrior notas que le pedimos al usuario por teclado
*/
for ( int i = 0 ; i < notas . Length ; i ++) {
Console . Write ( "Introduce la nota en la posición " + ( i + 1 )+
": " );
notas [ i ]= Convert . ToInt32 ( Console . ReadLine ());
}
//Declaramos la variable en la que almacenaremos la media
int notaMedia = 0;
/* Mediante una nueva estructura for, vamos recorriendo el
* array y vamos acumulandolas notas
*/
for ( int i = 0 ; i < notas . Length ; i ++)
{
notaMedia += notas [ i ];
}
//Finalmente dividimos el sumatorio por el numero de notas
notaMedia /= notas . Length;
//Mostramos por pantalla la media obtenida.
Console . WriteLine ( "Su nota media es " + notaMedia );
Console . ReadKey ();
```

Si observamos bien el código, podemos comprobar que existen dos partes bien diferenciadas, por un lado, tenemos la lectura de las notas que el alumno debe introducir, y, por otro lado, el cálculo de la media.



Se puede comprobar perfectamente que nuestro programa podría estar formado por dos módulos independientes de la siguiente manera:

- **Función MAIN.**

Todos los programas desarrollados en C# tienen una función principal denominada **main()**, que se ejecuta al iniciar el programa. Esta función se encarga de gestionar el flujo de ejecución llamando a los diferentes módulos si es necesario.

CÓDIGO:

```
static void Main ( string [] args)
{
int [] notas = new int [ 5 ];
for ( int i = 0 ; i < notas . Length ; i ++) {
Console . Write ( "Introduce la nota en la posición " + ( i + 1 )+
": " );
notas [ i ]= Convert . ToInt32 ( Console . ReadLine ());
}
Console . WriteLine ( "Su nota media es " + calcularMedia ( notas
));
Console . ReadKey ();
}
```

- **Función calcularMedia().**

Esta función nos la creamos nosotros porque pensamos que sería conveniente desarrollar el programa mediante diseño modular. Esta función recibe las notas introducidas por los alumnos y devuelve la nota media de las mismas.

CÓDIGO:

```
public int calcularMedia ( int [] newarray ){
int notaMedia = 0;
for ( int i = 0 ; i < newarray . Length ; i ++){
notaMedia += newarray [ i ];
}
notaMedia /= newarray . Length;
return notaMedia;
}
```

Comprobar que en C# los programas se dividen en subprogramas denominadas "**funciones**". Éstas funcionan de forma similar a una caja negra, es decir, el programa principal sólo debe conocerla para llamarla por su nombre, los parámetros que recibe (las variables que debemos enviar) y lo

que devuelve como resultado. Por ejemplo, nuestra función **calcularMedia**, recibe un *array* de notas y devuelve un entero con la media.

1.5. Llamadas a funciones. Tipos y funcionamiento.

- **Funciones en C#**

En C#, podemos encontrar dos niveles diferentes a la hora de implementar la modularidad en funciones y clases.
En este caso, vamos a centrarnos en las funciones, ya que las clases las veremos más adelante. Las funciones se pueden definir como un conjunto de instrucciones (delimitadas por llaves) que tienen un nombre y son de un tipo específico.

Sintáxis:

CÓDIGO:

```
Modificadores Tipo NombreFuncion (Parámetros de entrada ){
Código de la función
Return expresión ; //En caso de que devuelva algo.
}
```

Vemos paso a paso el significado de cáda parte de la función:

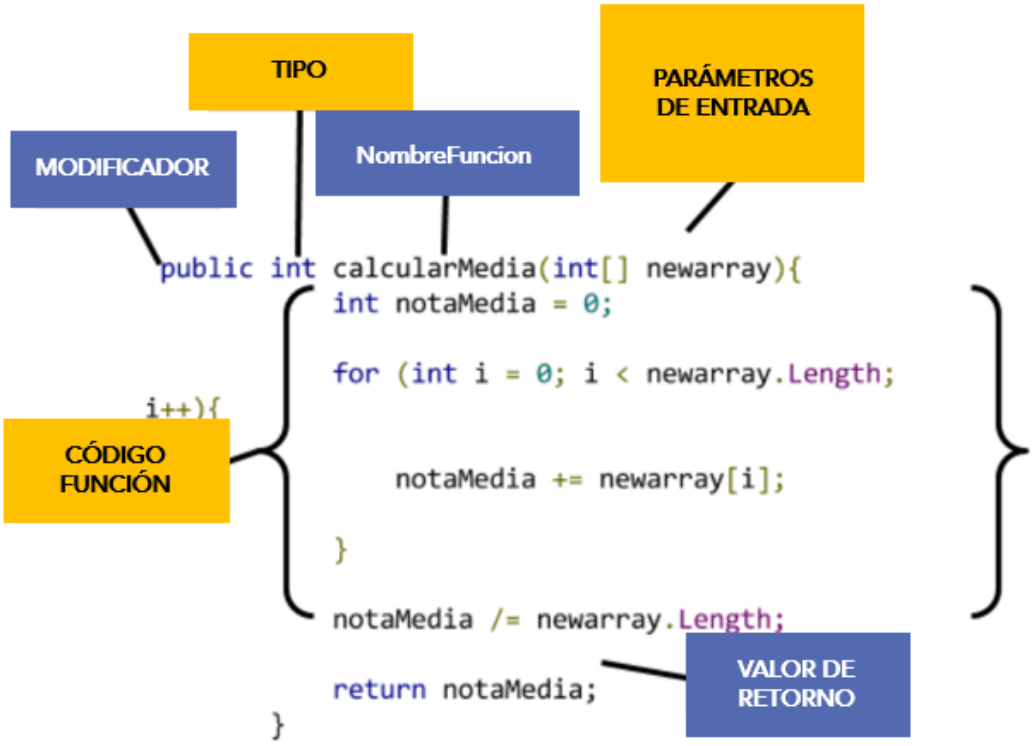
- **Modificadores**: conjunto de palabras reservadas que modifican o amplican propiedades sobre la función. Los más comunes con los modificadores de acceso: *Public, Private, Protected, Internal*.

En nuestro caso, debemos poner siempre la palabra reservada ***static*** antecediendo al modificador de acceso, no es el objetivo en este punto explicar en profundidad los modificadores, lo veremos más adelante.

- **Tipos y Funcionamiento**
  - **Tipo**: las funciones, al igual que ocurría con las variables, poseen un tipo, es decir, el dato que devuelven es de un tipo determinado. Se puede dar el caso en el que una función no devuelva nada. En este caso, decimos que la función es de tipo ***void*** (vacío).

- **NombreFunción**: es el identificador de la función, lo usaremos para referenciarla, del mismo modo que usábamos los nombres de las variables.
- **Parámetros de entrada**: una lista de las variables que recibirá la función en el momento de su llamada. Es posible que una función no requiere parámetros, en ese caso, no escribiremos nada aquí.
- **Return**: en caso de que la función sea de cualquier tipo diferente de *void*, es obligatorio que devuelvan un parámetro del tipo adecuado. Utiliza la palabra reservada return.

A modo de ejemplo, veamos cómo quedaría utilizando el mismo que vimos en el apartado anterior de calcular la nota media de un alumno.



Mediante el uso de parámetros se permite la comunicación de las diferentes funciones con el código.

En la mayoría de los lenguajes de programación, se distinguen dos tipos de parámetros diferentes:

- Por valor ó copia.
- Por referencia.

- **Paso de parámetros por valor**

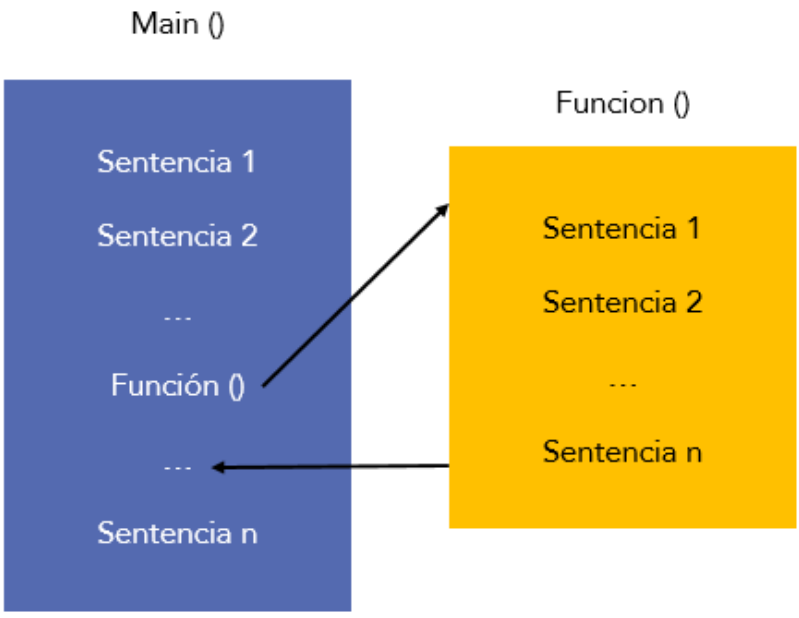
Cuando ejecutamos una función que tiene parámetros pasados por valor, se realiza una copia del parámetro que se ha pasado, es decir, que todas las modificaciones/cambios que se realicen, se están haciendo en esta copia que se ha creado. El original no se modifica de manera que no se altera su valor en la función.

- **Paso de parámetros por referencia**

Sin embargo, cuando ejecutamos una función que tiene parámetros pasados por referencia, todas aquellas modificaciones que se realicen en la función, van a afectar a sus parámetros, ya que se trabaja con los originales.

1.6. Ámbito de las llamadas a funciones.

Una función puede ser llamada en un programa cada vez que sea necesario. Al ser llamada, se ejecuta el código que se encuentra, delimitado entre llaves, en su interior. Cuando la función finaliza (termina de ejecutarse), el programa continúa en la siguiente sentencia en la que fue llamada.



A la hora de trabajar con funciones, es recomendable que utilicemos una sólo función para cada tarea específica, y, por supuesto, que funcione correctamente. Podremos ir creando nuevas funciones cada vez que las

vayamos necesitando, y, una vez que las tengamos implementadas, en nuestra función principal (*main*) iremos llamándolas según sus prioridades de ejecución.

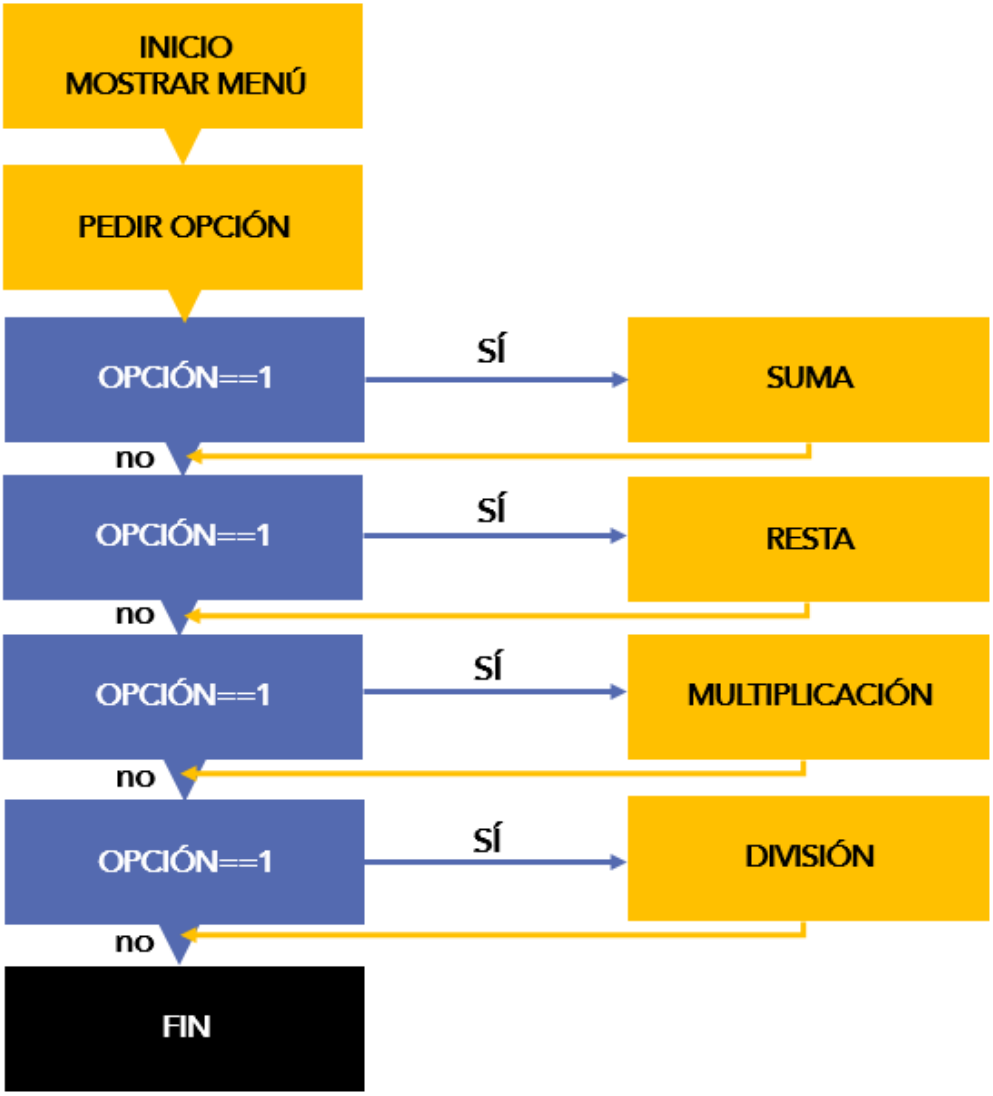
Podemos dividir las funciones en cuatro tipos básicos:

- Las que sólo ejecutan código.
- Las que reciben parámetros.
- Las que devuelven valores.
- Las que reciben parámetros y valores.

Para no repetir tanto código dentro de un mismo programa, se utilizan las funciones. Se escribe el código necesario para realizar algo, y, cada vez que sea necesario repetir esa parte de código, será suficiente con que llamemos a la función y así evitamos repetir varias veces lo mismo.

- **Las que sólo ejecutan código**

A continuación, vamos a ver el diagrama de flujo de las operaciones básicas que se pueden implementar en una función.



Cada una de las operaciones va a tener su propia función. Cuando trabajamos con funciones no es necesario que programemos todo a la vez, iremos implementando nuestro código poco a poco, así irá creciendo nuestro programa.

Comenzaremos nuestro programa desde el **main()** tal y como aparece en el diagrma de flujo.

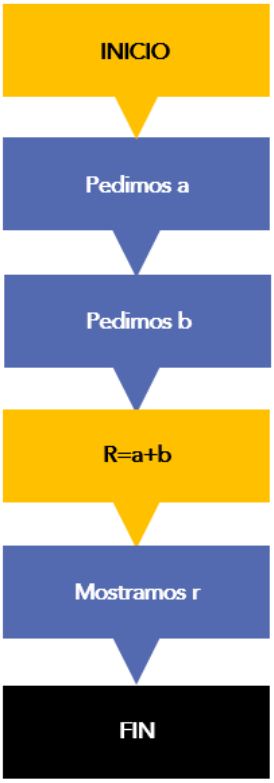
CÓDIGO:

```
using System;
using System.Collections.Generic;
using System.Text;
namespace AplicacionBase

{
class Program
{
// Esta es la funcion principal del programa
// Aqui inicia la aplicacion
static void Main(string[] args)
{
// Variables necesarias
int opcion = 0;
string valor = “”;
// Mostramos el menu
Console.WriteLine(“1-Suma”);
Console.WriteLine(“2-Resta”);
Console.WriteLine(“3-Multiplicacion”);
Console.WriteLine(“4-Dividir”);
// Pedimos la opcion
Console.WriteLine(“Cual es tu opcion:”);
valor = Console.ReadLine();
opcion = Convert.ToInt32(valor);
// Comprobamos la suma
if (opcion == 1)
{
}
// Implementamos la resta
if (opcion == 2)
{
}
// Implementamos la multiplicacion
if (opcion == 3)
{
}
// Realizamos la division

if (opcion == 4)
{
}
}
}
}
```

La función **main()** tiene la lógica principal del programa en cuestión. Los bloques de código están vacíos de momento, y lo iremos completando según avancemos.

Comenzaremos con la función suma. Esta función no va a recibir ningún parámetro, ni tampoco va a devolver ningún valor. Llevará en su interior todo el código por lo que pedirá los valores de los operandos, y mostrará el resultado de la operación.

Veamos primero su diagrama de flujo para pasar a implementarlo posteriormente.



Por lo que nuestra función va a quedar de la siguiente manera:

CÓDIGO:

```
static void suma()
{
      //variables que necesitamos
      float num1=0;
      float num2=0;
      float res=0;
      string numero="";

      //Pedimos valores para los datos por teclado
      Console. WriteLine ("Introduzca el valor del primer número");
      numero=Console.ReadLine();
      num1=Convert.ToSingle(numero);

      Console. WriteLine ("Introduzca el valor del segundo número");
      numero=Console.ReadLine();
      num2=Convert.ToSingle(numero);

      //Calculamos la suma de los dos números y lo almacenamos el
      resultado en la //variable  res

      res=num1+num2;


      //Mostramos el resultado
      Console.WriteLine("El resultado es {0}",res);

      //Delvolvemos el resultado
      return res;

}
```

Sobre la función podemos observar que se llama "suma", que como no devuelve ningún valor, es de tipo *void,* y que no recibe parámetros de entrada por eso los paréntesis están vacíos.

```
static void suma()
```

Recordemos también que las variables que se declaran dentro de una función son variables locales, y sólo podrán utilizarse dentro de la función en la que son declaradas.
Asignamos a la función el nombre de "suma" porque tiene relación con lo que se va a hacer, y, de esta forma, es más sencillo a la hora de hacer un seguimiento de nuestro programa.
Una vez que la función ya está realizada, no ocurre ningún cambio en nuestro programa y no es porque no esté bien realizada. Es simpemente porque el programa principal todavía no la ha llamado. Así que nos vamos a nuestro **main(),** y en la primera opción (que es la de "sumar") realizamos la llamada a la función.

CÓDIGO:

```
if (opción==1)
{
      suma();
}
```

- **Las que devuelven valores**

Cuando se llama a una función, éstas deben tener un código que realice alguna operación para que pueda devolver un valor como resultado. Puede ser llamada desde la función main o desde cualquier otra función.
Como la función tiene que devolver un valor, debemos indicar de qué tipo va a ser ese valor devuelto, y para hacerlo utilizará la palabra reservada *return*, como, por ejemplo:

CÓDIGO:

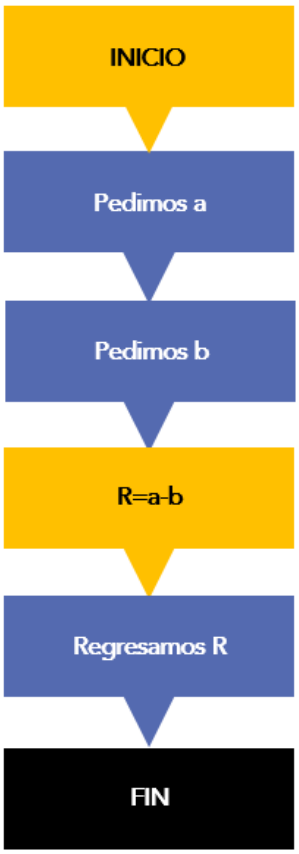
```
int función()
{
      int variable;
      …
      …
   return variable;
}
```

Cuando el programa encuentra la palabra return la función ha concluido, es decir, ha finalizado.
A la hora de llamar a nuestra función, debemos almacenar en una variable del mismo tipo que devuelva, la llamada a dicha función tal que así:

CÓDIGO:

```
int num;
num=función();
```

Sin embargo, en la función **main()** vamos a recibir un valor, pero, ¿qué hacemos con él?
El diagrama de flujo nos quedaría de la siguiente forma:



Por lo que la implementación de nuestra función sería:

CÓDIGO:

```
static float resta()
{
      //variables que necesitamos
      float num1=0;
      float num2=0;
      float res=0;
      string numero="";

      //Pedimos valores para los datos por teclado
      Console. WriteLine ("Introduzca el valor del primer número");
      numero=Console.ReadLine();
      num1=Convert.ToSingle(numero);

      Console. WriteLine ("Introduzca el valor del segundo número");
      numero=Console.ReadLine();
      num2=Convert.ToSingle(numero);

      //Calculamos la resta de los dos números y almacenamos el
      resultado en la //variable  res

      res=num1-num2;

      //Mostramos el resultado
      Console.WriteLine("El resultado es {0}",res);

      //Devolvemos el resultado
      return res;
}
```

La función **resta()** es de tipo *float*, por lo que al final hemos puesto un *return* que devuelve el resultado (variable *res*) de la resta que también es de tipo *float*. Además debemos añadir la parte de código que se corresponde con la segunda opción que es la resta.

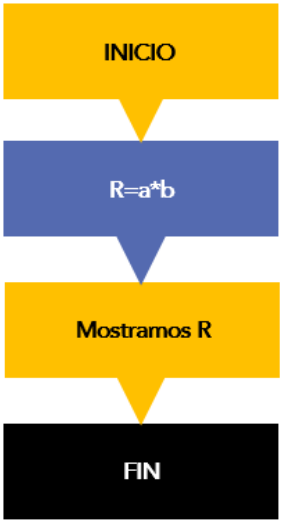
CÓDIGO:

```
//código para hacer la resta

if (opción==2)
{
      //variable donde vamos a almacenar el resultado
      float res=0;

      //llamamos a la función resta
      res=resta();

      //mostramos el resultado para ver si es correcto
      Console.WriteLine("El resultado de la resta es {0}",res);
}
```

- Las que reciben valores

En los apartados anteriores hemos trabajado pidiendo los valores directamente al usuario, mientras que, en este caso, se le pasan al programa los valores en vez de pedirlos. Estos parámetros pueden ser: entero, flotante, cadena, o, incluso, pueden ser tipos de datos definidos por la persona que realiza el programa.
Es importante señalar que los parámetros deben llevar su tipo y su nombre. Mediante el nombre podemos acceder a los datos que tiene (van a trabajar como variables locales a la función).
La llamada a la función es bastante sencilla, sólo se pone el nombre de la función, y entre paréntesis aquellos datos que se pasan como parámetros.
Para verlo de forma práctica, vamos a hacer el caso de la multiplicación de dos números. Esta función va a recibir dos parámetros desde la función **main()**, a continuación va a hacer el cálculo, y, por último, mostrará su resultado.

El diagrama de flujo correspondiente a la función multiplicación sería de la siguiente forma:



El código que le corresponde a este flujo sería el siguiente:

CÓDIGO:

```
static void Multiplicación(float num1, float num2)
{
      //variables necesarias
      float res;

      //calculamos el valor
      res=num1*num2;

      //mostramos el resultado
      Console.WriteLine("El resultado es {0}",res);
}
```

Podemos comprobar que la función tiene dos parámetros tipo flotante, que reciben el nombre de **num1** y **num2**. Se escriben separados mediante una coma, y su contenido va a ser los que se le pasen en la llamada.
Por tanto, en la función ***main()*** añadimos en otro ***if*** la opción de la multiplicación, quedando de la siguiente forma;

CÓDIGO:

```
if (opción==3)
{
      //declaramos las variables
      float num1=0;
      float num2=0;
      string numero="";

      //pedimos los valores
      Console.WriteLine("Introduzca valor del primer número");
      Numero=Console.ReadLine();
      num1=Convert.ToSingle(numero);

      Console.WriteLine("Introduzca valor del segundo número");
      Numero=Console.ReadLine();
      num2=Convert.ToSingle(numero);

      //Llamamos a la función

      Multiplicación(num1,num2);

}
```

Dentro del ***if*** creamos dos variables flotantes, y pedimos al usuario los valores con los que se va a trabajar. Éstos van a quedar guardados en las variables ***n1*** y ***n2***. A continuación, llamamos a la función que como se le pasan dos parámetros, en la llamada, también debe tener una copia del valor que contiene ***n1*** y otra de ***n2***. Y ya puede la función llevarse a cabo. También

podemos ponerle el valor asignándoselo directamente en la llamada, como, por ejemplo:

CÓDIGO:

```
//Llamada a la función asignándole un valor
Multiplicación(n1,4.0f);
```

- **Las que reciben parámetros y devuelven un valor**

En este caso, las funciones van a recibir parámetros y, además, también van a devolver un valor. Como hemos visto ya en las funciones anteriores, los parámetros se van a declarar dentro de los paréntesis, y van a ser utilizados como si fueran variables. Debemos indicar el tipo que van a recibir, seguido del nombre de la función, y los parámetros irán entre paréntesis.
Como la función devuelve un valor debemos indicar su tipo cuando la declaramos, y no olvidemos poner el ***return*** al final del método.
En este caso, vmos a implementar la función división, que va a recibir dos valores flotantes, va a comprobar que la división se puede realizar, es decir, que no divide por cero, y nos va a devolver el número que devuelve tras realizar la operación.

CÓDIGO:

```
static int Division(float num1, float num2)
{
      //variables necesarias
      float res=0;

      //nos aseguramos de que no se divida por cero
      if(num2==0
      {
            Console.WriteLine("No se puede dividir entre el valor
cero");
            res=0.0f;

      }
      else
      {
            res=num1/num2;
      }
      return res;
}
```

Como una función sólo puede hacer un *return*, guardamos el valor del resultado en *res* si entramos en la primera condición ó si vamos a utilizar la segunda. Almacenamos el valor correspondiente en la variable *res* y, al final, hacemos un solo *return* con el valor que devuelve.
Es importante que señalemos que es importante contemplar la división por cero. En caso de que se pretenda dividir este valor, devolveremos 0.0f porque es un tipo *float*.
Si son valores distintos de cero, realizamos la división normal de un número (*num1*) entre otro (*num2*).
En el método **main()** debemos añadir a nuestro código lo siguiente:

CÓDIGO:

```
//Si vamos a elegir la opción de dividir
if(opción==4)
{
      //variables necesarias
      float num1=0.0f;
      float num2=0.0f;
      float res=0.0f;
      string numero="";

      //pedimos los valores
      Console.WriteLine("Introduzca valor del primer número");
      numero=Console.ReadLine();
      num1=Convert.ToSingle(numero);

      Console.WriteLine("Introduzca valor del segundo número");
      numero=Console.ReadLine();
      num2=Convert.ToSingle(numero);

      //Llamamos a la función

      res=Division(num1,num2);

      //Mostramos el valor del resultado
      Console.WriteLine("El resultado es {0}",res);

}
```

Al igual que en los ejemplos anteriores, primero declaramos las variables que vamos a necesitar para nuestro programa, pedimos los valores, y, finalmente, hacemos la llamada de la función. Como en este caso la función devuelve un valor, lo asignamos a la variable *res*, que es la que vamos a utilizar para almacenar el resultado, y en el *main()* lo mostramos.
El conjunto de todas estas operaciones formaría nuestro programa.

1.7. Prueba, depuración y comentarios de programas.

Una vez que tenemos terminado nuestro programa con su código ya implementado, es el momento de probarlo para comprobar que no tiene fallos, y, sobre todo, que su funcionamiento es el correcto. En muchos casos no existen errores, pero el programa cuando se ejecuta no realiza la tarea que debería.
Por eso es muy conveniente que nuestro programa tenga comentarios al menos en todo lo importante que realiza. De esta manera, nos será más fácil manejarnos por él.
Podemos encontrarnos errores en nuestro código, tanto sintácticos como semánticos, que pueden ir apareciendo en nuestro código sin esperarlo. La mayoría de los lenguajes de programación disponen de una serie de herramientas que nos van a servir para tratar estos errores y conseguir solucionarlos. Por ejemplo, el programa *Visual Studio*, cuenta con un menú "**Depurar",** que permite acceder a las herramientas de las que dispone el depurador. En **C#**, además tiene unas técnicas propias para controlar diferentes situaciones en las que se produzcan errores en tiempo de ejecución.
Cuando ponemos una aplicación en "modo depuración", *Visual Studio* tiene la posibilidad de visualizar, línea a línea nuestro código para analizarlo de forma más detallada. En este modo podemos, entre otras funciones:

- Hacer consultas y realizar cualquier modificación de las variables.
- Parar la ejecución, editar código y continuar ejecutando.
- Detener la ejecución en una posición determinada.
- Ejecutar línea a línea.

Cuando ejecutamos un programa, podemos dividir el proceso en varias partes:

- Compilación
- Vinculación
- Ejecución

Veamos cómo funcionan cada una de estas etapas.

- **Compilación**

Mientras vamos escribiendo un determinado programa, el entorno de desarrollo de C# va haciendo comprobaciones exhaustivas para exponer los errores que podrían persistir en el código. Éstos errores aparecen subrayados conforme escribimos las distintas instrucciones, **y son denominados errores**

de compilación. Algunos ejemplos serían la falta de un punto y coma, una variable no declarada, etc.
Una vez que estos errores se corrigen, podríamos decir que el programa se va a compilar de una manera limpia, y va a pasar a ejecutarse, aunque no sabemos todavía si va a realizar la función exacta para la que ha sido diseñado.

- **Vinculación**

Debemos señalar que todos los programas hacen uso de bibliotecas, y, además, algunos utilizan clases diseñadas por el programador. Las clases se vinculan cuando el programa comineza a ejecutarse, aunque son los IDE los que comprueban el programa a medida que se va escribiendo para intentar garantizar que todos los métodos que se invoquen existan, y que coincidan con su tipo y parámetros.

- **Ejecución**

Una vez que conseguimos llegar a la fase de ejecución, significa que ya nuestro programa no tiene errores, pero todavía no sabemos si va a funcionar o no. Algunos errores se detectan de forma automática, pero otros son más complicados de detectar, y pueden producir cambios inesperados, por lo que debemos comenzar con un proceso de depuración en el que vamos a ir comprobando, paso a paso, cómo va a ir funcionando nuestro programa hasta que demos con el error.

1.8. Concepto de librerías.

Cuando hablamos de librerías nos referimos a un conjunto de funciones que están preparadas para ejecutarse, por lo que facilita el trabajo del encargado de desarrollar el programa.
Cuando compilamos nuestro programa sólo se hace la comprobación de que se han llamado de manera correcta las funciones que pertenecen a las diferentes librerías que nos ofrece el compilador, aunque el código de estas funciones todavía no se ha insertado en el programa. Va a ser el enlazador ó *linkador* el que realice esta inserción, y, por tanto, el encargado de completar el código máquina con la finalidad de obtener el programa ejecutable. Una vez que hemos obtenido el código fuente, debemos comprobar que no se producen errores en tiempo de ejecución. Si los hubiera, debemos depurar el programa. Encontrar cuál es la causa que produce un error en tiempo de ejecución, a veces, es una tares complicada, por lo que los EID (Entornos

Integrados de Desarrollo) nos proporcionan una herramienta denominada Depurador y nos facilita esta tarea.
Uno de los motivos por lo que es recomendable utilizar el lenguaje de programación C# es que éste permite la interoperación con otros lenguajes, siempre que éstos tengan:

- Acceso a las diferentes librerías a través de COM+ y servicios .NET.
- Soporte XML.
- Simplificación en administración y componetntes gracias a un mecanismo muy cuidado de versiones.

1.9. Uso de librerías.

El concepto de librería se podría dar como conjunto de métodos relacionados con el mismo objetivo, para poder ser reutilizado cada vez que cualquier programador lo desee. Para la realización de este módulo, en los ejercicios prácticos, vamos a ir utilizando las librerías "*Math*", para cualquier operación matemática, y la librería "*Random*".

Vamos a explicar algunos ejemplos
A.- En este primer ejemplo vamos a calcular la potencia de base 10 y exponente 2. Para ello debemos de mirar cuales son los métodos que están implementados en la librería *Math* y podemos comprobar que, existe un método llamada *POW* que realiza las operaciones exponenciales.

CÓDIGO:

```
Using System; // para la utilización de dicha librería debemos de
importar System
double calculo = Math.Pow(10, 2);
```

B.- Para el siguiente ejercicio vamos a poder comprobar el funcionamiento de varios ejemplos de *random* con distintos intervalos.

CÓDIGO:

```
using System;

public class Example
{
   public static void Main()
   {
      Random rnd = new Random();
 // Declaración e inicialización de la variable aleatoria

      Console.WriteLine("\n20 random integers from -100 to 100:");
      for (int ctr = 1; ctr <= 20; ctr++)
      {
         Console.Write("{0,6}", rnd.Next(-100, 101));
/* El método Next es el que calcula el numero aleatorio, indicando
el rango menor y el intervalo de números que queremos calcular en
este caso desde -100 con un intervalor 101 números, contando con el
número inicial. */

         if (ctr % 5 == 0) Console.WriteLine();
      }

      Console.WriteLine("\n20 random integers from 1000 to 10000:");
      for (int ctr = 1; ctr <= 20; ctr++)
      {
         Console.Write("{0,8}", rnd.Next(1000, 10001));
         if (ctr % 5 == 0) Console.WriteLine();
      }

      Console.WriteLine("\n20 random integers from 1 to 10:");
      for (int ctr = 1; ctr <= 20; ctr++)
      {
         Console.Write("{0,6}", rnd.Next(1, 11));
         if (ctr % 5 == 0) Console.WriteLine();
      }
   }
}
// The example displays output similar to the following:
//       20 random integers from -100 to 100:
//            5    -5  - 20    94   -86
//           -3   -16   -45   -19    47
//          -67   -93    40    82    68
//          -60   -55    67   -51   -11
//
//       20 random integers from 1000 to 10000:
//            4857    9897    4405    6606    1277
//            9238    9113    5151    8710    1187
//            2728    9746    1719    3837    3736
//            8191    6819    4923    2416    3028
//
//       20 random integers from 1 to 10:
//             4     4     5     9     9
//             5     1     2     3     5
//             5     4     5    10     5
//             7        7     7    10     5
```

1.10. Introducción al concepto de recursividad.

Cuando trabajamos con funciones, podemos definir la recursividad como la llamada de una función a sí misma hasta que cumpla una determinada condición de salida.

Sintaxis:

CÓDIGO:

```
modificador tipo_devuelto nombre_función(tipo par1, tipo par2, …, tipo parn)
{
      nombre_funcion(var1, …, var2)
      …
}
```

Podemos diferenciar entre dos tipos de recursividad

- **Directa.** Cuando la función hace la llamada a sí misma desde un punto específico de su código (entre las llaves que la delimitan).
- **Indirecta.** Cuando la función hace la llamada a otra función y es ésta quien llama a la primera.

Los problemas que presenta la recursividad suelen tener una estructura similiar, necesitan de:

- Un **caso base** que permita la finalización del programa.
- **Casos recursivos,** que son llos que se vana aencargar de que la función vuelva a ejecutarse, pero acercándose cada vez más al caso base.

Vamos a ver un ejemplo para poner en práctica todos estos conceptos.

Factorial de un número.

Debemos saber que es el factorial de un número. Por ejemplo:
3!=3*2*1
4!=4*3*2*1
5!=5*4*3*2*1

Si queremos realizar en C# una función que calcule el factorial de un número, podríamos definirla de la siguiente forma:

CÓDIGO:

```
int factorial(int n)
```

¿Cuál podría ser el caso base?
Debemos encontrar uno, que finalice el programa. En este caso siempre tenemos que buscar el caso más pequeño que existe. En este caso, el 1!=1.

factorial(1)=1;

Pensemos ahora en el caso recursivo.
Debemos encontrar uno que sirva para todos los números. Debe ir haciéndose cada vez más pequeño hasta que consiga llegar al caso base que es el factorial de 1.

Si nos fijamos en el ejemplo que hemos puesto:
3!=3*2*1
4!=4*3*2*1
5!=5*4*3*2*1

Podemos apreciar lo siguiente:
3!=3*2*1
4!=4*3!
5!=5*4!
….

Vemos que en todos los casos sucede lo mismo, se multiplica el número por el número -1 factorial. Por lo que ya tenemos el caso recursivo.

factorial(n)=n*factorial(n-1)

Por lo que nos quedaría de la siguiente forma:

Caso base → Si n=1, factorial(1) devuelve 1;
Caso genérico → Si n>1, factorial(n) = n*factorial(n-1);

Si hacemos una función:

función factorial (entero n) retorna entero

CÓDIGO:

```
{
	si n>1 entonces
		//caso recursivo
		devuelve n*factorial(n-1);
	Sino
		//caso genérico
		devuelve 1;
	FinSi;
}
```

Si lo traducimos al lenguaje de programación C#:

CÓDIGO:

```
static int factorial(int n)
{
	if (n>1)
		return n*factorial(n-1);
	else
		return 1;
}
```

UF3: Fundamentos de Gestión de ficheros

1. Gestión de ficheros

Todo lo que llevamos visto hasta aquí ha sido mediante variables, estructuras de datos, y hemos manipulado la información de la que disponíamos. Esta información, una vez que finaliza la ejecución del software, desaparece de memoria, ya que ha estado almacenada en la memoria principal el tiempo que dura la ejecución del software.
Una vez que llegamos al tema de los ficheros, cambia un poco la forma de trabajar que traíamos, de manera que vamos a utilizar nuevas estructuras o métodos, que van a permitir que la información no se volatilice, es decir, que no se borre cuando terminemos el proceso de ejecución.

1.1. Concepto y tipos de ficheros.

Podemos ver los ficheros como una parte de un dispositivo no volátil a la que se le asigna un nombre, y que puede contener una cantidad de datos que va a estar limitada, o por la cantidad de espacio del que disponga el dispositivo o por las características del Sistema Operativo. Entendemos por dispositivos no volátiles aquellos que no pierden la información que poseen cuando apagamos nuestro ordenador.
Debido a la importancia que tienen los ficheros al actuar de almacenes no volátiles de la información, la *BCL* reserva un espacio de nombres denominados Systm.IO, y están destinados a trabajar con ellos.
Cada sistema operativo puede tener un formato diferente de nombres de ficheros, nosotros vamos a centrarnos en la plataforma .NET, que consiste básicamente en nombrar los ficheros utilizando como máximo 259 caracteres diferentes según el siguiente formato:

CÓDIGO:

```
<nombre_fichero>.<extensión>

<nombre_fichero> nombre del fichero.
<extensión> tipo del fichero del que se trata
```

Los ficheros se almacenan en directorios o carpetas. Cada directorio puede contener, a su vez, otros directorios diferentes, y esto hace que el sistema de archivos vaya creando una estructura jerárquica en la que cada fichero o

directorio tiene como padre al directorio que lo contiene. Como es lógico, y para que esta estructura sea finita, debe existir un directorio raíz que va a ser el que contenga a todos los demás y no tenga dependencia de ningún otro.

- En resumen:

Los ficheros o archivos son una secuencia de bits (0 y 1) que se almacenan en un dispositivo de almacenamiento secundario, por lo que la información va a permanecer a pesar de que se cierre la aplicación que los utilice. Esto da más independencia sobre la información, ya que no necesitamos que el programa se esté ejecutando para que la información que contiene exista. Cuando se diseña un programa y se quiere guardar información en algún fichero, el programador es el encargado de indicar cómo se va a hacer. Los ficheros tienen la ventaja de poder almacenar gran cantidad de información

A la hora de trabajar con ficheros, debemos tener en cuenta:

- La información es un conjunto de 0 y 1.
- Al agrupar los bits se forman bytes o palabras.
- Los tipos de datos van a estar formados por un conjunto de bytes o palabras.
- Al agrupar los campos, se crean los registros de información.
- Un fichero es creado por un conjunto de ficheros de manera que todos tienen en común la misma estructura.
- Los directorios tienen la función de agrupar distintos ficheros siguiendo unas condiciones determinadas dadas por el sistema operativo o por el programador.

Utilidades de los ficheros:

- Permiten organizar más facílmente el sistema de archivos.
- Evitan conflictos con sus nombres, ya que cada programa instala sus ficheros en directorios diferentes. Por tanto puede haber en una misma máquina dos ficheros identificado por el mismo nombre, ya que como van a tener distinta ruta, lo vamos a poder diferenciar. Como se puede comprobar, la relación entre ficheros y directorio es muy cercana, en C# se establece entre tipos y espacio de nombres

- **Rutas de ficheros y directorios**

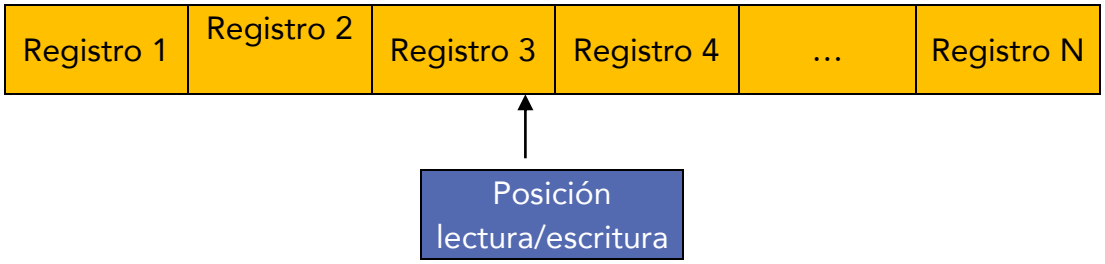
En el apartado anterior hemos visto la relación entre fichero y directorio. Para la identificación unequívoca de este fichero, habrá que nombrar el camino que nos lleva hasta él. A este camino lo denominamos **Ruta**.
Dependiendo de como empecemos la ruta de directorio para nombrar el archivo, podemos tener dos tipos de rutas bien diferenciadas:

- **Ruta absoluta o completa**: Se le indica el camino de directorío desde el cominezo. Si es en el sistema operativo Linux o Mac OS, empezará por el raíz (\), en caso contrario si el sistema operativo es Windows debe de empezar por el nombre de la unidad en cuestión.
- **Ruta relativa**: Se le indica el camino de directorio desde la posición actual. Por tanto, la ruta no empezaría con el carácter raíz (\) o la letra de la unidad.

- **Tipos de ficheros**
  - Según su acceso. Según la forma de organizar la información, podemos distringuir entre tres tipos diferentes de ficheros:
    - **Secuencial.** En los ficheros secuenciales, los registros se van almacenando en posiciones consecutivas de manera que cada vez que queramos acceder a ellos tendremos que empezar desde el primero e ir recorriéndolos de uno en uno.
      Sólo se puede realizar una operación de lectura o escritura a la vez. Por ejemplo, si estamos leyendo el fichero, no podemos realizar ninguna operación de escritura sobre él hasta que termine de ser leído y viceversa.

    - **Aleatorio o directo.** En los ficheros aleatorios o directos, podemos acceder a un registro concreto del mismo indicando una posición perteneciente a un conjunto de posiciones posibles. Debido a que los registros están organizados, éstos pueden ser leídos o escritos en cualquier orden, ya que se accede a cada uno a través de su posición. Cuando queremos realizar una operación,

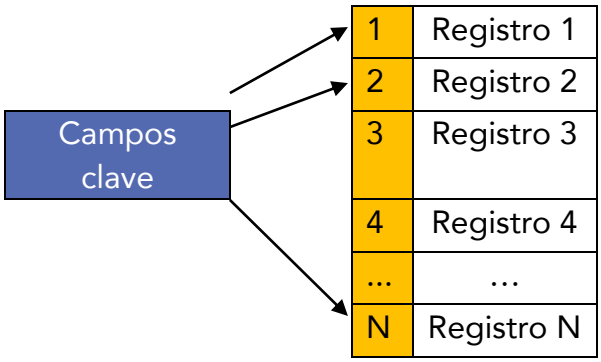
basta con colocar el puntero que maneja el fichero justo antes de éste.



Para leer el Registro 4 colocamos el puntero de lectura/ escritura justo antes de su posición.

- **Secuencial indexado.** Los ficheros indexados poseen un campo clave para ser identificados. Permiten el acceso secuencial y aleatorios a un fichero de la siguiente forma:
  1) Primero busca de forma secuencial el campo clave
  2) Una vez que lo encuentra, el acceso al fichero es directo, ya que sólo tenemos que acceder a la posición indicada por el campo clave.

Para que este proceso funcione de manera correcta, los índices se encuentran ordenados permitiendo de esta forma un acceso más rápido.



Estructura de un fichero indexado.

- **En cuanto al contenido del fichero:**
  - **Ficheros de texto.**

- **Ficheros binarios**: Los datos se almacenan de forma binaria (como su nombre indica), y, por tanto, de la misma forma que se guarda en memoria. El dato se encripta en 0 y 1, de esta manera se hace mucho más eficiente su almacenamiento. Se necesita un visualizar en concreto para poder tratar su contenido.

1.2. Operaciones sobre ficheros secuenciales.

Las tres operaciones básicas que tenemos cuando trabajamos con ficheros son

- **Apertura.**

Lo primero que debemos hacer siempre es abrir el fichero para la operación que vayamos a realizar sobre él. Cuando abrimos el fichero estamos relacionando un objeto de nuestro programa con un archivo.
Los diferentes modos en los que se puede abrir un fichero son los siguientes:

- **Lectura**. Sólo va a realizar operaciones de lectura en el fichero.
- **Escritura**. Realiza operaciones de escritura en el fichero. Si ya existía, lo sobreescribe.
- **Añadir**. Permite realizar la operación de escritura en un fichero que ya existía añadiéndole a lo anterior lo que se quiera escribir sin necesidad de eliminarlo.
- **Lectura/Escritura**. Permite operaciones de lectura/escritura sobre el fichero.

- **Lectura/ Escritura.**

Antes de realizar la operación de lectura o escritura, debemos asegurarnos de que el puntero se encuentra en la posición correcta para realizarlo. Éste indicará dónde se va a comenzar a leer. Si estuviéramos al final del fichero debe indicar que la operación no se puede realizar.
Las operaciones de lectura/escritura se pueden hacer de dos formas diferentes: para ficheros secuenciales o aleatorios. Vamos a verlo con un ejemplo más detenidamente.

- **Lectura Secuencial:**

CÓDIGO:

```
//Declaración de la variable del fichero
fichero f1;

//Abrimos el fichero para leerlo
f1.abrir(lectura);

Mientras no final de fichero hacer
      f1. leer(registro);
      operaciones con registro leído;
finMientras

//Cerramos el fichero
f1.cerrar();
```

- **Lectura Aleatoria:**

CÓDIGO:

```
//Declaración de la variable del fichero
fichero f1;

//Abrimos el fichero para leerlo
f1.abrir(lectura);

Mientras condición del programa hacer   //No es necesario estructura
repetitiva en
                                        //todos los casos
      f1. leer(registro);
      operaciones con registro leído;
finMientras

//Cerramos el fichero
f1.cerrar();
```

- **Escritura Secuencial:**

CÓDIGO:

```
//Declaración de la variable del fichero
fichero f1;

//Abrimos el fichero para leerlo
f1.abrir(escritura);

Mientras tengo información el fichro hacer
      Configurar registros a partir de unos datos
      f1. escribir(registro);

finMientras

//Cerramos el fichero
f1.cerrar();
```

- **Escritura Aleatoria:**

CÓDIGO:

```
//Declaración de la variable del fichero
fichero f1;

//Abrimos el fichero para leerlo
f1.abrir(escritura);

Mientras se deseen escribir datos hacer
      Posicionar el puntero del fichero en la posición deseada
      f1. escribir(registro);
      operaciones con registro leído;

finMientras

//Cerramos el fichero
f1.cerrar();
```

- **Cierre:** cuando cerramos el fichero, éste queda liberado y termina el proceso de almacenamiento de información.

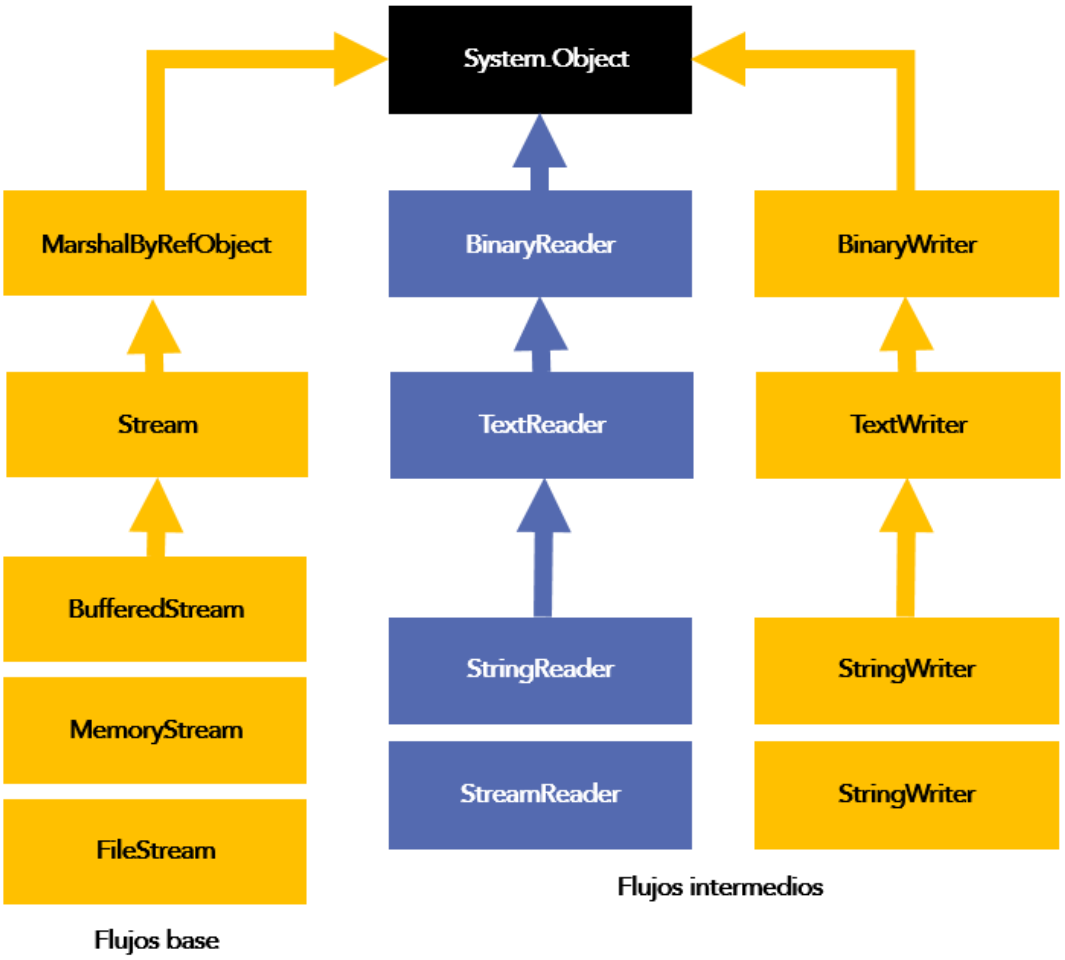
1.3. Diseño y Modulación de las operaciones sobre ficheros.

1.3.1. Fundamentos de los flujos

Los flujos (también llamado *Stream*) de datos son las estructuras o pasarelas que tenemos para acceder a los datos de un fichero, de una forma consistente y fiable, desde un código fuente en un cualquier lenguaje de programación. Dicho acceso se hace como un secuencia o sucesión de elementos que pueden entrar o salir del programa.
Dependiendo de la operación que vayamos a realizar con los datos, podemos tener varios tipos de flujos: De Entrada, de Salida o de Entrada / Salida.
En el flujo de Entrada el datos (Lectura) solo podemos realizar la operación de lectura de un fichero, es decir, existe una comunicación unilateral desde el fichero al programa.
En el flujo de Salida (Escritura) también es en sentido unidireccional, ya que sólo podemos realizar la operación de escritura en el fichero.
En el flujo o stream de Entrada / Salida es cuando podemos tanto leer, como escribir en el fichero.
Es tipo de *stream* lo vamos a definir al principio de trabajar con ficheros y no se podrá cambiar una vez abierto.
Se pueden distiguir dos tipos de flujos:

- **Flujos base**: son aquellos que operan más a nivel máquina, como, por ejemplo: porción de memoria, espacio de disco o conexión de red.
- **Flujos intermedios**: son aquellos flujos que trabajan por encima de los anteriores. Se puede combinar con el flujo bases de manera que éste pueda verse beneficiado por todas las funcionalidades que ofrezcan el flujo base. Algunos ejemplos podrían ser los flujos que proporcionan encriptación de datos o los buffers de almacenamiento intermedio.

1.3.2. Clases de flujos



En este apartado, utilizaremos las clases de C# pertenecientes a los dos tipos de ficheros: **Binarios** y de **Textos**.

- **Clase FileStream**

Es la clase del flujo bases de las operaciones de lectura y escritura de un fichero. Lo primero que debemos de detallar es el contructor *FileStream* para indicar los siguientes parámetros:

- El nombre del archivo que vamos a abrir.
- El modo en el vamos a abrir el fichero (*FileMode*). En esta ocasión se indica si debemos de crear el fichero o no. Se explica en el siguiente punto.
- El modo en el que accedemos al fichero (FileAccess).
    - **Read**: Acceso para leer el archivo.

- **Write**: Acceso de escritura al archivo.
- **ReadWrite**: Acceso de lectura y escritura al archivo.

- **Los Modos de Apertura (*FileMode*).**

Se puede abrir un fichero de varias formas posibles. Cada una conlleva a unas operaciones que el fichero realizar de antemano.

- **Append:** modo de añadir. Abre el fichero para añadir apartir de lo ya escrito o bien lo crea vacío si no existiera el fichero
- **Create:** dicho modo crea un fichero en blanco. Si ya existiera, borraría su contenido y lo dajaría en blanco.
- **Open:** sólo abre un fichero ya existente, en caso contrario se produce fallo en el programa.
- **OpenOrCreate:** abre un fichero en caso de la exista previamente en la carpeta o lo crea en caso contrario.

- **Ficheros binarios**

Para acceder de una forma más cómoda y sin ninguna limitación de tamaño de datos, dentro de ***System.IO*** podemos utiliza dos operaciones: **BinaryReader** y **BinaryWriter,** encapsuladas en **FileStreams,** que proporcionan unos cuantos de métodos para facilitar las operaciones de lectura y escritura de cualquier objeto en ficheros. De la siguiente forma utilizar dichas operaciones:

CÓDIGO:

```
BinaryReader(Stream flujo)
BinaryWriter(Stream flujo)
```

Podemos poner un ejemplo para escribir en un fichero binario llamado "datos.dat" usando UTF8 al codificar los caracteres:

CÓDIGO:

```
FileStream f = new FileStream("datos.dat", FileMode.Create);
BinaryWriter f_Binario = new BinaryWriter(f, Encoding.UTF8));
f_Binario.Write("Este mensaje se escribirá en UTF8 dentro de
datos.dat");
```

De forma análoga podemos leer un dato de un fichero binario llamado "datos.dat":

CÓDIGO:

```
   FileStream f = new FileStream("datos.dat", FileMode.Open);
   BinaryReader f_Binario = new BinaryReader(f, Encoding.UTF8));
   Console.WriteLine("Leido de datos.dat: {0}",
f_Binario.ReadString());
```

- **Ficheros de texto**

Para el tratamiento con los ficheros de texto vamos a utilizar la clase *StreamWriter*. Principalmente dos métodos para escribir y leer caracteres de texto en un fichero.

- **WriteLine**. Escribe una cadena de caracteres agregando el marcador de fin de línea.
  Un ejemplo podría ser el siguiente:

CÓDIGO:

```
using System.IO; // importación de la clase System.IO
private void escribir_fichero(object sender, EventArgse)
    {
// escribe varias líneas de texto en el archivo
StreamWriter flujoSalida = File.CreateText("datos.txt");
//Creación y apertura del flujo de salida para el fichero "datos.txt"
flujoSalida.WriteLine("Este archivo");
flujoSalida.WriteLine("contiene tres");
flujoSalida.WriteLine("líneas de texto.");
flujoSalida.Close();  // Cierre del fichero
     }
```

Bibliografía

Joyanes, L. Programación en C++: Algoritmos, Estructutas de Datos y Objetos. McGraw Hill. 2000.

Yolanda Cerezp, Olga Peñalba, Rafael Caballero. Iniciación a la programación rn C#. Un enfoque práctico. Delta publicaciones. 2007

DOUGLAS BELL & MIKE PARR c# para estudiantes. Editorial pearsonISBN: 978-607-32-0328-9México, 2010

Webgrafía

http://www-assig.fib.upc.edu/~prop/provaprogrames08.pdf

ILERNA Online