ILERNA Online

Módulo 3A

Programación

UF1: PROGRAMACIÓN ESTRUCTURADA

1. Introducción a la programación

Un programa es un conjunto de instrucciones dadas al ordenador en un lenguaje que solo es entendible por él, para comunicarle lo que queremos que haga. Un algoritmo es una secuencia finita de operaciones que resuelven un problema expuesto.

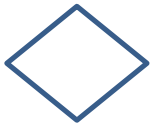
Un algoritmo es más parecido a una idea, una forma de resolver un problema, mientras que un programa está más relacionado con la realización de una o más tareas por parte de un ordenador.

Un programa debe cumplir una serie de **características**:

- Deber ser **finito**: formado por un conjunto limitado de líneas.
- Debe ser **legible**: es importante crear códigos "limpios" y fáciles de leer con tabulaciones y espacios que diferencien las partes del programa.
- Debe ser **modificable**: debe ser sencillo el proceso de actualización o modificación ante nuevas necesidades.
- Debe ser **eficiente**: debemos crear programas que ocupen poco espacio en memoria y se ejecuten rápidamente.
- Debe ser **modulable**: debemos realizar algoritmos que se dividan a su vez en subalgoritmos de forma que se disponga de un grupo principal desde el que llamaremos al resto. Así, incitamos a la reutilización de código.
- Debe ser **estructurado**: engloba a las características anteriores, ya que

Un diagrama de flujo es una representación gráfica de un proceso. Cada paso del proceso se representa con un símbolo diferente que contiene una breve descripción de la etapa de proceso.

Debemos usar una serie de **símbolos** estándar:

	Indican inicio o fin de programa.
	Representan una instrucción, un paso a dar o un proceso. Por ejemplo: cont=cont+1.
	Operaciones de entrada/salida de datos. Por ejemplo: visualiza en pantalla suma.
	Usaremos este símbolo si nos encontramos en un punto en el que se realizará una u otra acción en función de la decisión que el usuario tome.
	Conector. Permite unir diferentes zonas del diagrama de forma que se redefine el flujo de ejecución hacia otra parte del diagrama.
	Representa una función o subprograma.

Características de los diagramas de flujo:

- Deben escribirse de arriba abajo o de izquierda a derecha.
- Debemos evitar el cruce de líneas, para eso se define la forma de conector. El uso de conectores debe producirse cuando no exista otra opción.
- Todas las líneas de flujo deben estar conectadas a algún objeto.
- El texto que se escribe en las formas debe ser escueto y legible.
- Todos los símbolos de decisión deben tener más de una línea de salida, es decir, deben indicar qué camino seguir en función de la decisión tomada.

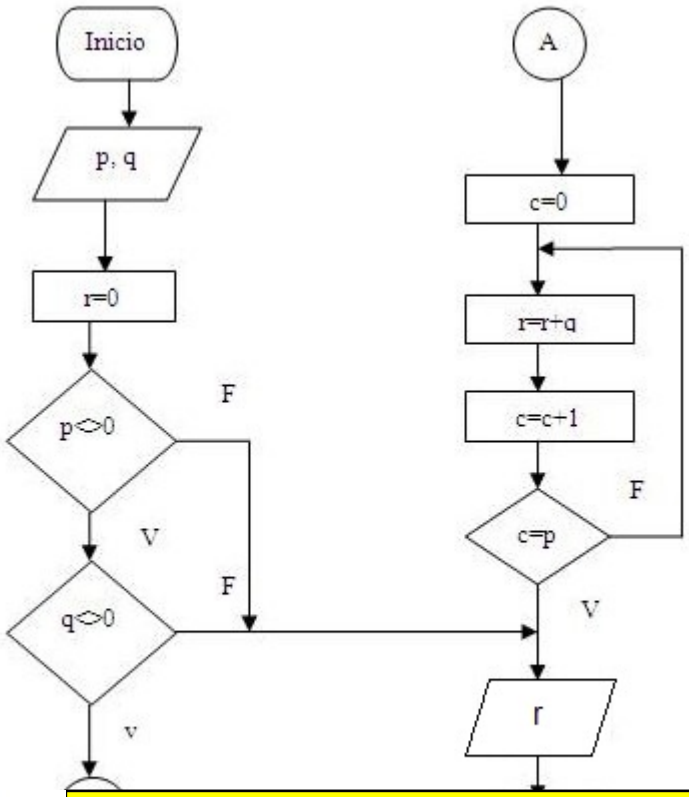
Veamos un ejemplo de cómo trabajar con estos diagramas. A continuación, mostramos un algoritmo:

```
Inicio
leer p, q
r=0

si (p<>0 entonces)
   si q<>0
        c=0
        repetir
          r=r+q
          c=c+1
        hasta c=p
   fin_si
```

Deberemos analizar cada una de las líneas para ver a qué tipo de representación corresponden:

- Inicio y fin serán representadas con un óvalo.
- La lectura y escritura será representada con un paralelogramo.
- Las tomas de decisiones (tanto las condiciones "si" como el "repetir") será representada con un rombo.
- Por último, las sentencias de instrucciones como "r=r+q" serán representadas con un rectángulo.

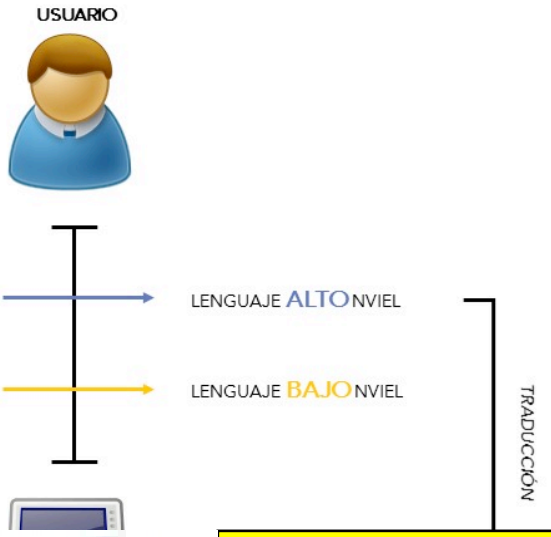
Veamos, pues, cómo quedaría nuestro diagrama:



El **proceso de programación** comunica a un usuario con una máquina. Para que se pueda realizar esta comunicación, debemos tener:

- Los dos agentes principales, **receptor** y **emisor** (usuario y máquina).
- **Canal**: para continuar con 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 tanto el receptor como el emisor hablan un lenguaje completamente diferente. Para que la comunicación sea fluida, debemos acercar los lenguajes tanto de la máquina como del usuario para lograr el entendimiento.

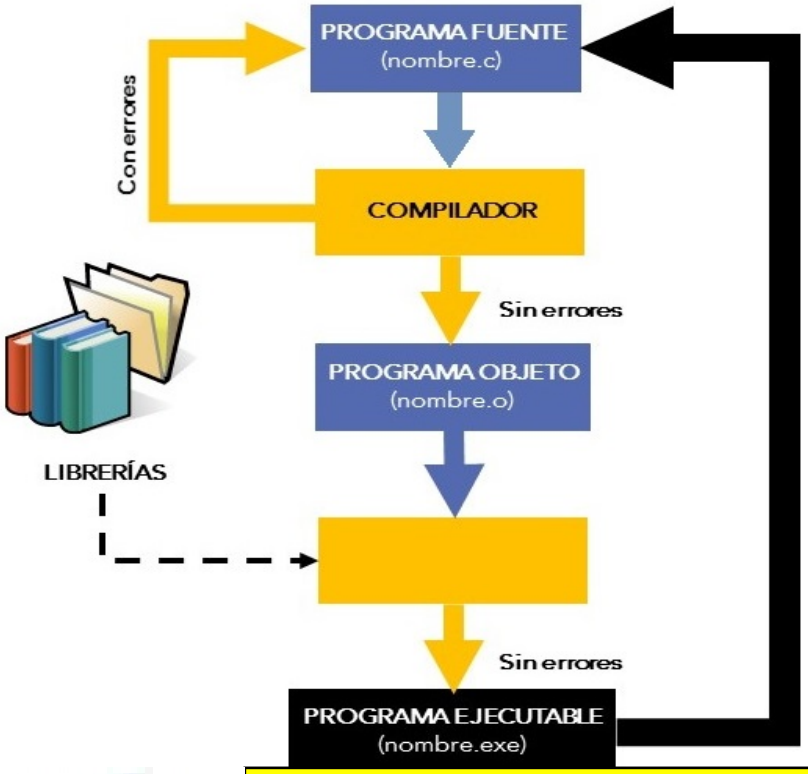
Para solventar el problema de la comunicación, aparecen lenguajes de programación de dos tipos: **alto nivel** y **bajo nivel**. Los lenguajes de alto nivel están más cerca del lenguaje que habla el usuario, mientras que los lenguajes de bajo nivel están más cerca de las estructuras del lenguaje de la 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.

Características del lenguaje de alto nivel:

- Es totalmente independiente de la máquina y, por tanto, muy portable.
- Muy utilizado en el mercado laboral informático.
- 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.



las constantes y los distintos operadores que podemos usar a la hora de implementar un código fuente.

Lenguajes compilados e interpretados

Aquellos lenguajes de alto nivel que utilizan un compilador para poder traducirlo al lenguaje máquina se denominan **lenguajes compilados**.

En nuestro caso, un programa escrito en C# necesita un compilador de C#. En este supuesto, el lenguaje máquina no corresponde al del ordenador, sino al de una máquina ficticia llamada **máquina virtual**. En C#, la máquina virtual se denomina CLR *(Common Language Runtime)*.

Esta máquina no existe físicamente, sino que es simulada por un ordenador. Podemos instalarla en nuestro ordenador copiando ese programa en nuestro disco duro.

Gracias a esa máquina, podemos ejecutar nuestro programa en cualquier ordenador del mundo.

Un **lenguaje interpretado** no genera un programa escrito en una máquina virtual. Efectúa directamente la traducción y ejecución simultáneamente para cada una de las sentencias.

Un intérprete verifica cada línea del programa cuando se escribe. La ejecución es más lenta debido a esa traducción simultánea.

Programación estructurada

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.

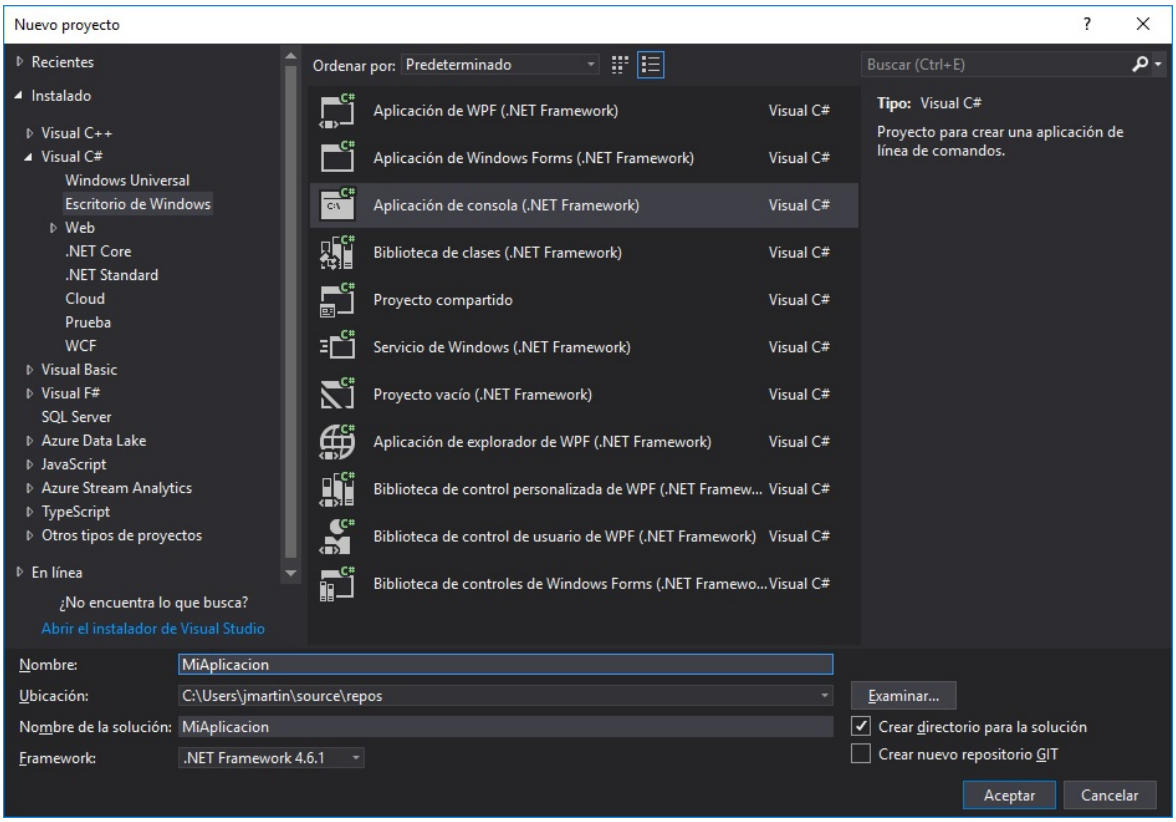
- Es recomendable a la hora de crear instrucciones para cualquier ámbito.
- Es un lenguaje orientado a objetos.
- Utiliza el recolector de basura.
- Permite la unificación de tipos.

2. Estructura de un programa informático

En el lenguaje de programación C# podemos tener una aplicación formada por uno o varios ficheros con su código fuente correspondiente, teniendo en cuenta que solo uno de ellos va a ser el principal.

Para todas las pruebas que realizaremos vamos a utilizar el **programa Visual Studio**. Con este programa vamos a ir generando los diferentes proyectos. Cada vez que creemos un proyecto nuevo, el programa va a generar solo 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**, ya que es la extensión utilizada por los ficheros en C#.



2.1. Bloques de un programa informático

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.

Si implementamos un programa en C#, debe tener, como mínimo, la parte del programa principal:

CÓDIGO:

```
class Program
   {
      static void Main(string[] args)
      {
      }
   }
```

El método ***Main*** es el punto de entrada de una aplicación de C#.

El método *Main* se puede declarar con o sin un parámetro *string*, que contiene los argumentos de línea de comandos.

Es conveniente ir añadiendo comentarios a nuestro programa sobre los pasos que se van realizando. Los **comentarios** van con "//" si el comentario ocupa una sola línea, y "/*" al inicio de la primera línea y "*/" al final de la última 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
{
   static void Main()
   {
      Console.WriteLine("Buenas Tardes");
   }
}
```

La primera parte, con los símbolos "/*" y "*/" es un comentario donde se explica lo que hay que hacer en el ejercicio. A continuación, se declara la **clase** "Hello1".

Una **clase** consta de métodos (funciones) y datos de características comunes.

Dentro de la función ***Main***, tenemos el operador ***Console.WriteLine***, que muestra por pantalla el mensaje "Buenas Tardes".

Las **operaciones de E/S (entrada/salida)** se realizan de forma diferente según el lenguaje de programación que se utilice. En C necesita funciones declaradas en *stdio.h*, mientras que en C++ utiliza *stream,* que es el que se refiere al flujo de la información mediante E/S.

2.2. Variables. Usos y tipos

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.

Cada variable va a almacenar un **tipo de dato** como:

Tipo simple	Descripción
Sbyte, short, int, long	Enteros con signo
Byte, ushort, uint, ulong	Enteros sin signo
Float, **double**	Punto flotante
Char	Uno o varios caracteres
Decimal	Decimal de alta precisión
Bool	Booleano
String	Cadena de caracteres

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 este nombre tenga relación con el ejercicio que estemos desarrollando.

Para **identificar a una variable**, y que tenga un identificador válido por el compilador, debemos seguir una serie de normas:

- Debe comenzar por un carácter (letra o "_").
- Debe estar formada por caracteres del alfabeto (no podemos usar "ñ"), de dígitos (del 0 al 9) y se puede utilizar subrayado ("_").
- No debe comenzar con un dígito.
- Distingue entre mayúsculas y minúsculas.
- No puede llevar espacios en blanco.

Debe **cumplir** la siguiente sentencia:

CÓDIGO:

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

Y en **C#** lo pondríamos de la siguiente forma:

CÓDIGO:

```
int num;
```

En este caso, estamos definiendo una variable de tipo entero denominada "num".

A las variables también se les puede asignar un **valor**:

CÓDIGO:

```
int num=5;
```

Definimos una variable 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.

El problema de las variables globales es que crean dependencias ocultas. Cuando se trata de una aplicación grande, el programador no conoce los objetos que tiene ni sus relaciones.

2.3. Constantes. Tipos y utilidades

En el apartado anterior hemos hablado de las variables como espacio de memoria donde se almacenan datos que pueden variar a lo largo del programa. Ahora trabajaremos con los espacios de memoria **cuyo contenido no se puede alterar a lo largo del programa: las constantes**.

Son muy parecidas a las variables, pero añadiendo la palabra "**const**".

Se **definen** de la siguiente forma:

CÓDIGO:

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

Por **ejemplo**:

CÓDIGO:

```
const int num;
```

Declaramos una constante de tipo entero que denominamos "num".

A continuación, declararemos una constante llamada "días semana", con el valor 7, ya que todas las semanas tienen 7 días:

CÓDIGO:

2.4. Operadores del lenguaje de programación

Para poder operar con las diferentes variables, utilizaremos una serie de **operadores** que detallamos en las siguientes tablas:

Operadores aritméticos	
+	Suma aritmética de dos valores.
++	Incremento en 1 del operando.
-	Resta aritmética de dos valores.
--	Decremento en 1 del operando.
*	Multiplicación aritmética de dos valores.
/	División aritmética de dos valores.
%	Obtiene el resto de la división entera.

Operadores de comparación	
>	Mayor.
<	Menor.
>=	Mayor o igual.
<=	Menor o igual.
==	Igual.
!=	desigualdad.
=	Asignación.

||	OR u O lógico.
!	NOT o No lógico.

2.5. Conversiones de tipos de clase

Como hemos visto anteriormente, un programa consta de numerosas variables de diferentes tipos, que suelen declararse al comienzo de este. Sin embargo, conforme vamos avanzando en el programa y 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 implícita recibe 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**". Les 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

ponemos entre paréntesis el tipo de datos al que queremos convertir el valor, para que tenga el mismo tipo que la variable destino. En este caso, es entero.

Con esto estamos diciendo que el valor de "resultado" va a ser un número entero, pero nuestra variable "resultado" seguirá siendo de tipo *double*.

Conversión explícita

Las conversiones explícitas requieren un operador de conversión. Se utilizan cuando es posible la perdida de información.

Cada tipo de dato tiene una clase que posee un método llamado *parse* que convierte al tipo de dato desde un *string*.

CÓDIGO:

```
String cadena =“5”;
float variable_flotante = float.Parse(cadena);
int numero_entero = int.Parse(cadena);
char var_car = char.Parse(cadena);
```

2.6. Ejercicio propuesto

¿Qué resultados se obtienen al realizar las operaciones siguientes? Si hay errores en la compilación, corríjalos y dé una explicación de por qué suceden:

CÓDIGO:

```
x=a/b;

c=a<b && c;

d= a + b++;

e= ++a - b;

y= (float)a / b;
```

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, condicional e iterativa**.

A finales de los **años setenta** surgió una nueva forma de desarrollar los programas que, además de conseguir el correcto funcionamiento de estos, también facilitó su comprensión gracias a la forma en la que estaban escritos. Estos programas resultaron ser más fiables y eficientes.

En esta época fue cuando **Edsger Dijkstra** comprobó que todo programa se puede escribir utilizando únicamente tres instrucciones de control:

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

Utilizando estas tres estructuras, 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 general de a qué nos referimos. En este caso, podemos definir el concepto de programar como **decirle a una máquina 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

Por su parte, 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 es una etapa en todo el proceso de desarrollo que existe a la hora de resolver un problema**.

3.2. Introducción a la algoritmia

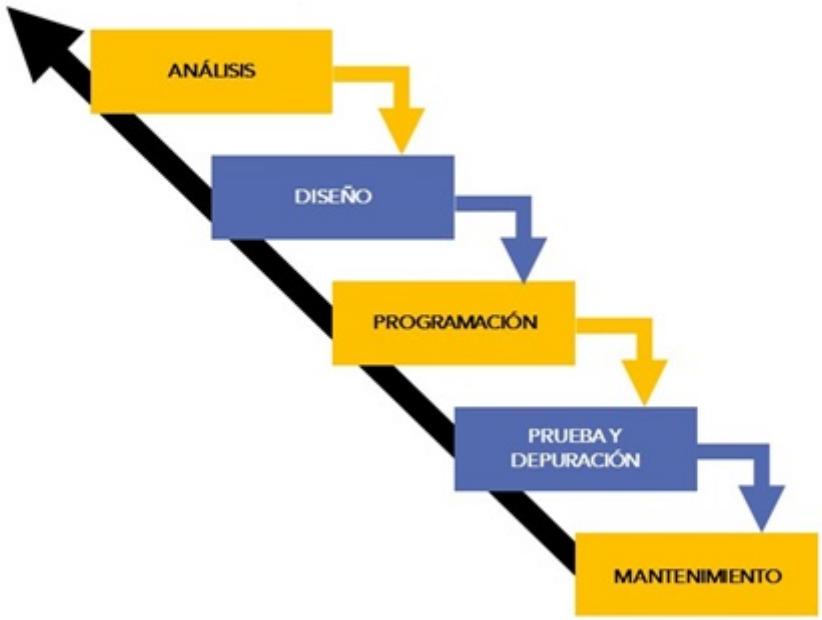
Las personas estamos acostumbradas a obedecer una serie de **órdenes secuenciales y lógicas**. Lo mismo pasa con los ordenadores, que realizan diferentes tareas siguiendo una serie de pasos lógicos, ya que están programados (mediante algoritmos) para solucionar distintos problemas en un lenguaje de programación determinado.

La **algoritmia** es un conjunto ordenado y finito de operaciones que permite encontrar la solución a un problema cualquiera.

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 óptima para un problema determinado**. Todo esto se consigue mediante el diseño, creación e implementación de un algoritmo.

3.3. Ciclo de vida

La imagen representa el **ciclo de vida de un programa informático**, de manera que las flechas indican el orden de realización de cada etapa.



- **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. El contenido de comunicación de esta etapa es muy intenso, ya que el objetivo es eliminar la ambigüedad en la medida de lo posible.
- **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

funcionar la aplicación completa, comprobamos que cumple lo establecido en el diseño.

- **Pruebas de validación**: el último paso de la integración se basa en realizar nuevas pruebas de la aplicación en su conjunto. El objetivo es cerciorarse de que se cumple lo establecido en el documento de requisitos y que cubre las necesidades de los usuarios que ya habíamos previsto.
- **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.

Se prueba un programa para demostrar la existencia de un **defecto**: algorítmico, de sintaxis, de precisión, de documentación, de sobrecarga, de capacidad, de rendimiento, de sincronización, de recuperación, de *hardware* y *software* y de estándares.

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 anomalía en su ejecución.

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 considerar el rango de valores que puede tomar, las operaciones a realizar y el espacio necesario para almacenar datos.

Debemos tener en cuenta que el tipo de datos simple no está compuesto por otros tipos, y que contiene 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**, son los siguientes:

- **Natural**: números naturales (N): *byte, uint, ushort, ulong.*
- **Entero**: números enteros (Z): *sbyte, int, long, short.*
- **Real**: números reales (R): *decimal, float, double.*
- **Carácter**: caracteres (C): *char.*
- **Lógico**: *booleanos* (B): *bool.*

Tipos simples definidos por el usuario

CÓDIGO:

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

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 o estructurados**

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];

int[] array1 = new int[] { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
```

Con el anterior código, estamos declarando 10 números 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 número entero del vector.

v[1] → Segundo número entero del vector.

v[2] → Tercer número entero del vector.

v[3] → Cuarto número entero del vector.

…

tanto, esos datos nos indican 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, 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.

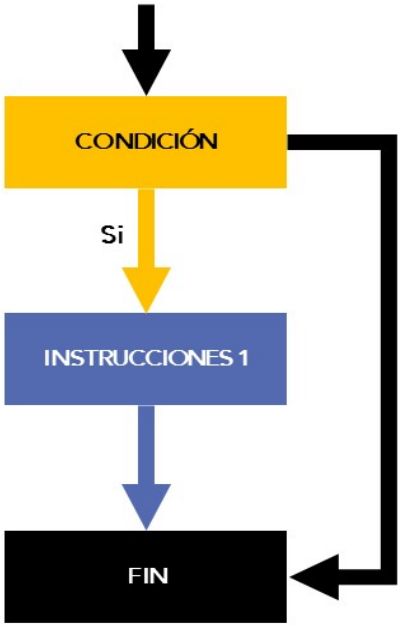
3.6. Estructuras de selección (instrucciones condicionales)

Las estructuras de selección son aquellas que permiten ejecutar una parte del código dependiendo de si cumple o no una determinada condición.

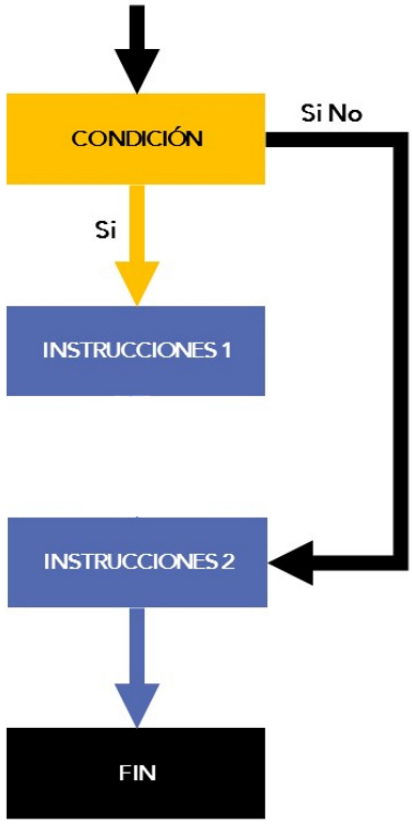
El mínimo bloque que debe tener esta estructura sería de la siguiente forma:

- **Bloque mínimo**

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



- **SI/SI NO**



SI CONDICIÓN ENTONCES:

INSTRUCCIONES_1;

SI NO:

INSTRUCCIONES_2;

FIN SI;

En este caso, si CONDICIÓN ES CIERTA, ejecutaremos lo que hay en INSTRUCCIONES_1 y saltaremos hasta el FIN SI.

Si CONDICIÓN ES FALSA, ejecutaremos lo que hay en INSTRUCCIONES_2.

Debe haber un SI NO por cada SI en estas estructuras

- **SI/SI NO, SI**

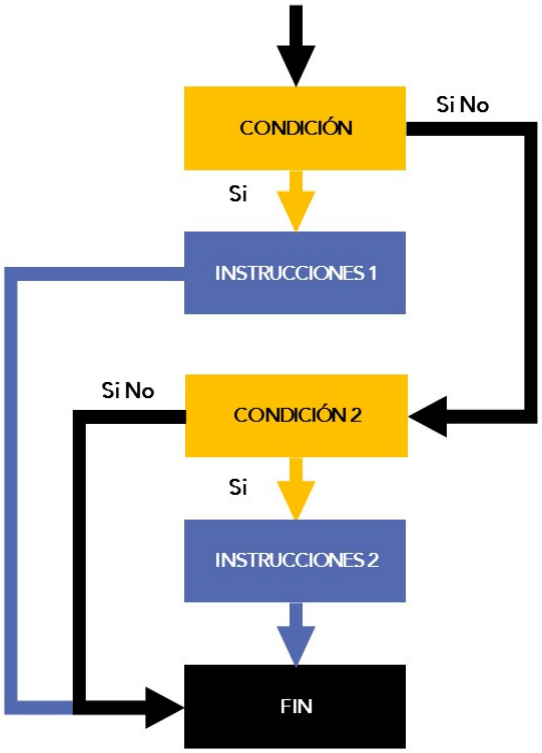
SI CONDICIÓN ENTONCES:

INSTRUCCIONES_1;

SI NO, SI CONDICIÓN_2:

INSTRUCCIONES_2;

FIN SI;



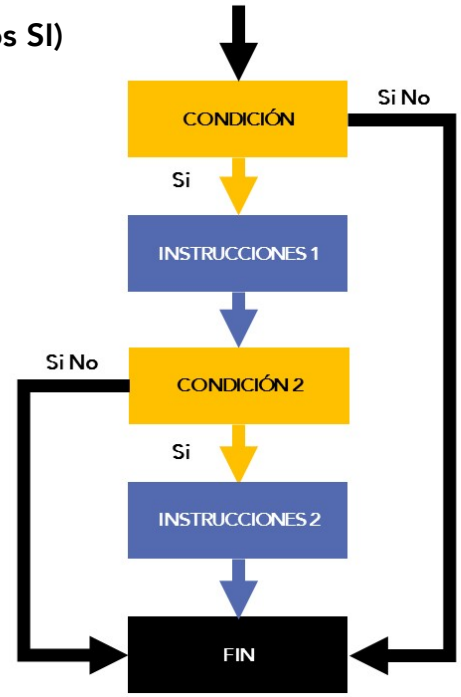
En este caso, si CONDICIÓN ES CIERTA, ejecutaremos lo que hay en INSTRUCCIONES 1 y saltaremos hasta el FIN SI, por tanto, NO MIRAREMOS SI CONDICIÓN_2 ES CIERTA O NO.

Si CONDICIÓN ES FALSA, miraremos si CONDICIÓN_2 ES CIERTA, en cuyo caso ejecutaremos lo que hay en INSTRUCCIONES_2 (fijaos en que solo lo miramos en caso de que CONDICIÓN sea FALSA), y saltaremos hasta el FIN SI.

En caso de que CONDICIÓN_2 sea FALSA, terminaremos la ejecución, saliendo del bloque SI.

- **SI anidado (pueden anidarse varios SI)**

```
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.

- **Operadores lógicos**

&&	AND: ambas condiciones deben cumplirse para entrar en el bloque SI.
||	OR: es suficiente con que una de las condiciones se cumpla 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;
```

Sentencia *Switch*

Es una instrucción de selección que elige una sola acción de una lista de opciones en función de una coincidencia de distintos patrones con la expresión de coincidencia.

CÓDIGO:

```
{
  public static void Main()
  {
    int var = 1;


    switch (var)
    {
        case 1:
            Console.WriteLine("Caso 1");
            break;
        case 2:
            Console.WriteLine("Caso 2");
            break;
        default:
            Console.WriteLine("Otro caso");
            break;
    }
  }
}
```

3.7. Estructuras de repetición

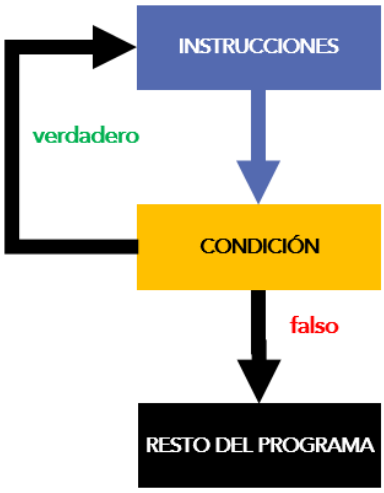
- **Mientras *(While)***

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 ejecutarse nunca.


```
Mientras CONDICIÓN Hacer
    Instrucción_1;
    Instrucción_2;
    …;
    Instrucción_N;
    ModificarCondición;
FinMientras;
```

- **Hacer… mientras *(Do… while)***

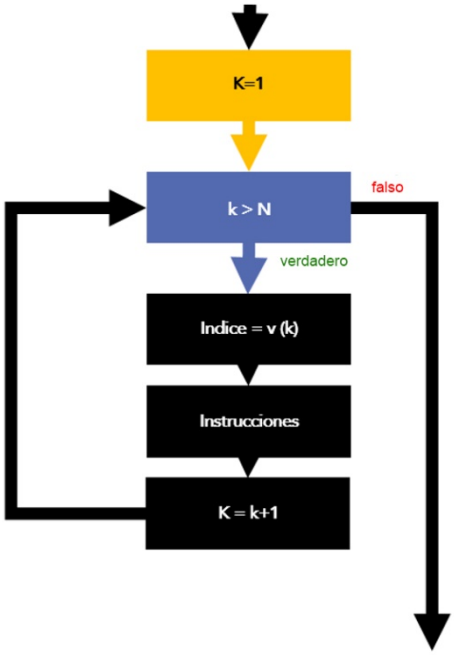
Mientras que se cumpla una condición, el código incluido dentro del bucle se repite. La condición se evalúa al final, por lo que, como mínimo, se va a ejecutar una vez.




```
Hacer
    Instrucción_1;
    Instrucción_2;
```

- **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 ejecutarse nunca.




```
Para CONDICIÓN = v
        Instrucción1;
        Instrucción2;
        …
        InstrucciónN;
        ModificarCondición;
```

3.8. Estructuras de salto

Las **estructuras de salto** son todas aquellas que detienen (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 solo se ejecuten los

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 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 creada previamente.

CÓDIGO:

```
char []letras = {'h', 'o', 'l', 'a'};
```

new

El operador *new* crea una nueva instancia de un tipo.

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 el número de caracteres.
ToCharArray()	Convierte un *string* en *array* de caracteres.
SubString()	Extrae parte de una cadena. Método que puede ser sobrecargado indicando su inicio y su fin: *SubString* (*int* inicio) o *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. Devuelve un entero que indica si la posición de esta instancia es anterior, posterior o igual que la posición del *string*.
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

Trim()	Quita todos los espacios en blanco del principio y el final 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**.

Podemos encontrar tres **tipos de errores** diferentes:

- De **compilación o sintaxis**: errores en el código.
- De **tiempo de ejecución**: los que producen un fallo a la hora de ejecutar el programa. Se trata de fragmentos de código que parecen estar correctos y que no tienen ningún error de sintaxis, pero que no se

- **Lógicos**: son aquellos que aparecen cuando la aplicación está en uso. Son la mayoría de los resultados erróneos, diferentes a los esperados o no deseados, a menudo en respuesta a las acciones del usuario. Son los más difíciles de corregir, ya que no siempre está claro dónde se originan.

Cuando queramos **depurar errores**, tenemos la opción, en cualquier momento de nuestro programa, de poner un punto de interrupción en una determinada línea de código. Para ello, presionamos *F9*.

Si queremos ejecutar la aplicación en el depurador de 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 estas, o podemos comprobar cuándo finalizan los bucles, entre otras cosas. Podemos ir depurando el código 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 de 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 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 problema, pero, en el caso de que se compile, devuelve algo que no era la solución que se esperaba. El programa no da error cuando se ejecuta en el inicio, por

3.11. Documentación de programas

Una vez que finaliza nuestro proceso de compilación y ejecución, debemos elaborar 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.

summary

La etiqueta *<summary>* debe usarse para describir un tipo y para agregar información adicional dentro de nuestro código. Podemos utilizar el atributo *cref* para permitir que herramientas de documentación como GhostDoc y Sandcastle creen hipervínculos internos a las páginas de documentación de los elementos de código.

CÓDIGO: XML

```
<summary>description</summary>
```

CÓDIGO: XML

```
namespace ConsoleApp33

{ // Mi clase. Para ello, implemento los métodos que vamos a utilizar
```

```
        /// Main es el método principal de nuestra clase
        ///
        /// </summary>

        static void Main(string [] args)
        {
            // Voy a realizar un programa y a documentarlo
            char []letras = { 'h', 'o', 'l', 'a'};

            string frase = new string(letras);
            char caracter = frase [0];
            //salida por pantalla
            Console.WriteLine(caracter);

            Console.ReadLine();
        }
    }
}
```

CÓDIGO: XML

```
<?xml version="1.0"?>
<doc>
    <assembly>
```

```
        <summary>


        Main es el método principal de nuestra clase.


        </summary>
      </member>
    </members>
</doc>
```

3.12. Entornos de desarrollo de programas

Una vez que diseñamos un programa, podemos denominar **entorno de desarrollo integrado** (IDE o *integrated development environment*) al entorno de programación que hemos utilizado para su realización: editor de código, compilador, depurador e interfaz gráfica (GUI o *graphical user interface*).

Los IDE son utilizados por distintos lenguajes de programación como C++, PHP, Python, Java, C#, Visual Basic, etcétera. 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, Visual Code o Visual Studio, entre otros.

Los IDE deben cumplir con una serie de **características** para su correcto funcionamiento, como, por ejemplo:

- Son multiplataforma.

- Permiten importar y exportar proyectos.
- Manejan diferentes idiomas.
- Facilitan un manual de ayuda para el usuario.

Las diferentes **ventajas** que ofrecen los IDE son:

- Presentan una baja curva de aprendizaje.
- Son de uso óptimo para usuarios que no son expertos.
- Formatean el código.
- Usan funciones para renombrar funciones y variables.
- Permiten crear proyectos.
- Facilitan herramientas para extraer partes de código.
- Muestran en pantalla errores y *warnings*.

UF2: DISEÑO MODULAR

1. Programación modular

Hasta ahora hemos estudiado el desarrollo de un programa como una unidad compuesta 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.

Un **subproblema** se denomina **módulo**, y es una parte del problema que se puede resolver de manera **independiente**.

1.2. Ventajas e inconvenientes

Enumeramos las ventajas e inconvenientes más relevantes en el trabajo de diseño modular:

- **Ventajas**:
  - Facilita el mantenimiento, la modificación y la documentación

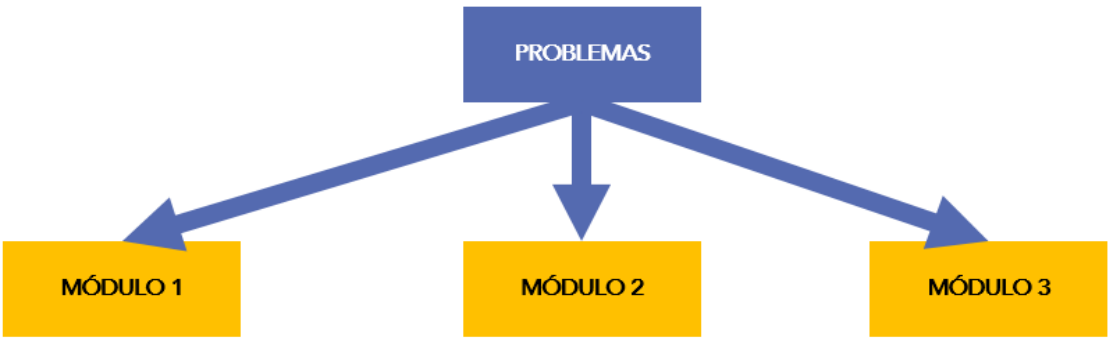
- **Inconvenientes**:
    - Separación de módulos.

  Aumenta el uso de memoria y el 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.

Este diseño se basa en el principio **"divide y vencerás"**.



1.4. Modulación de programas. Subprogramas

Podemos denominar los **módulos** como subprogramas, y estos como las diferentes partes del problema que puede resolverse de forma

Cada módulo codificado dentro del programa como subprograma se puede definir como una parte del código independiente, que realiza una tarea asignada.

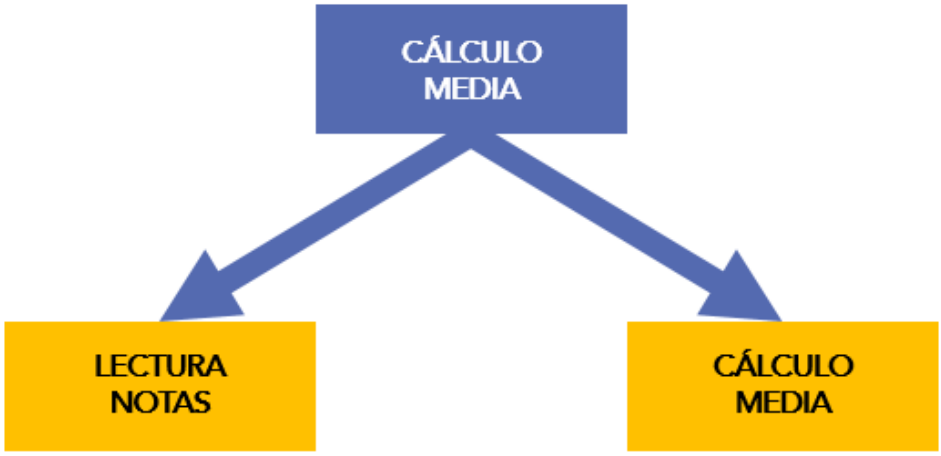
A continuación, vamos a ver a modo de ejemplo un programa que devuelva la nota media de un alumno:

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 interior 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 acumulando las notas
*/
for ( int i = 0 ; i < notas . Length ; i ++)
```

```
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***

CÓDIGO:

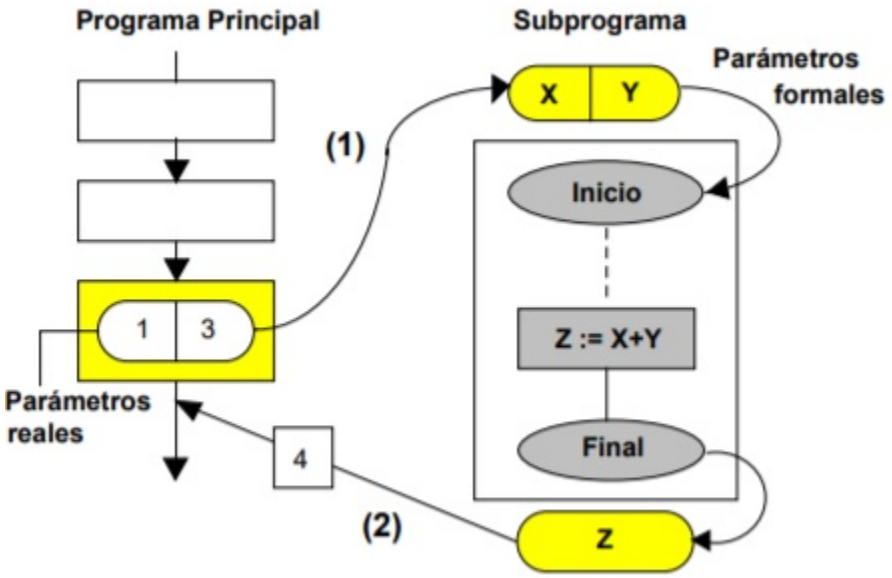
```
static void Main ( string [] args)
{
int [] notas = new int [ 5 ];
for ( int i = 0 ; i < notas . Length ; i ++) {
Console . Write ( "Introduce la nota en la posición " + ( i + 1 )+ ": " );
notas [ i ]= Convert . ToInt32 ( Console . ReadLine ());
}
Console . WriteLine ( "Su nota media es " + calcularMedia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

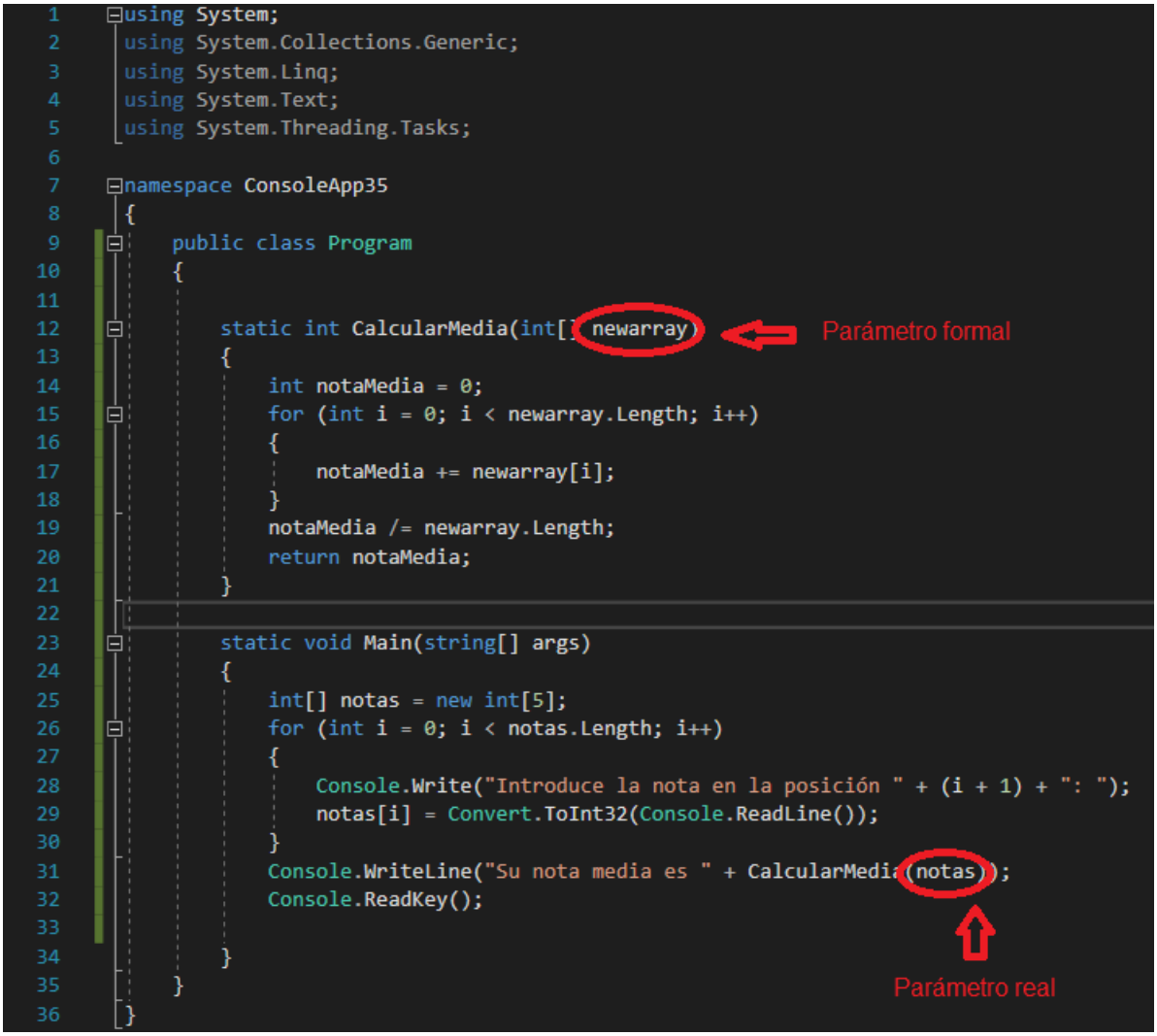
- **Función *calcularMedia()***.

Creamos esta función en la división de nuestro problema. 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;
```






```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp35
{
    public class Program
    {

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

        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();

        }
    }
}
```

Podemos comprobar que en C# los programas se dividen en subprogramas, denominados **funciones**. Estas funcionan de forma similar a una caja negra, es decir, el programa principal solo debe conocerlas para llamarlas por su nombre, los parámetros que reciben (las variables que debemos enviar) y lo que devuelven 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 tipos de métodos: funciones y procedimientos.

Las **funciones** se pueden definir como un **conjunto de instrucciones (delimitadas por llaves) que tienen un nombre y son de un tipo específico**.

Sintaxis:

CÓDIGO:

```
Modificadores Tipo NombreFuncion (Parámetros de entrada ){

Código de la función

Return expresión;

}
```

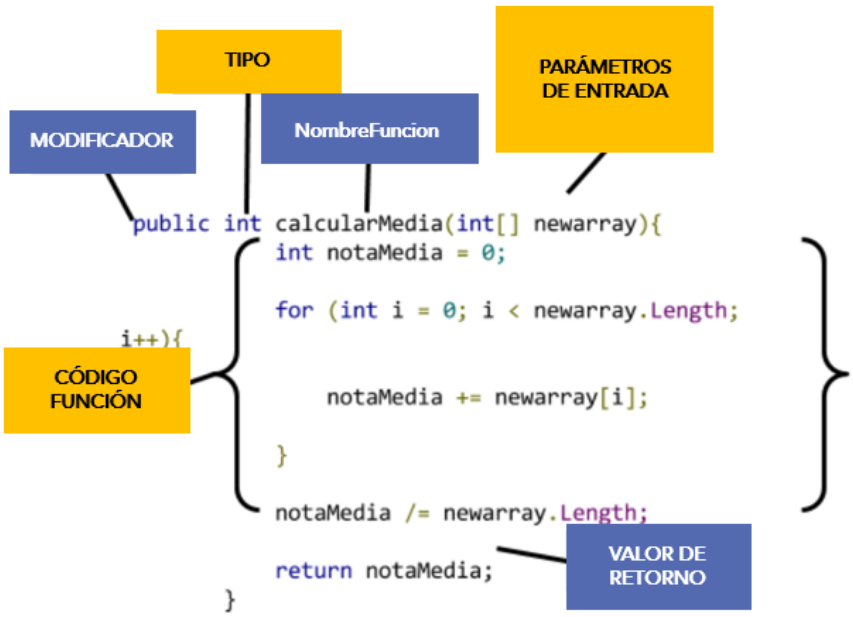
Veamos paso a paso el **significado de cada parte** de la función:

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

  1. ***Public***: el acceso no está restringido.
  2. ***Private***: el acceso está limitado al tipo contenedor.
  3. ***Protected***: el acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora.

- **Nombre de la función**: 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 requiera parámetros; en ese caso, no escribiremos nada aquí.
- ***Return***: Es obligatorio que devuelvan un parámetro del tipo adecuado. Utiliza la palabra reservada *return*.

Veamos cómo quedaría utilizando el mismo ejemplo que propusimos en el apartado anterior:



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

Procedimientos en C#

Los procedimientos son fragmentos que realizan una tarea específica y que pueden recibir parámetros, pero no necesitan devolver un valor como es el caso de funciones.

CÓDIGO:

```
Modificadores Tipo NombreProcedimiento (Parámetros de entrada ){
//Código del procedimiento
}
```

CÓDIGO EJEMPLO:

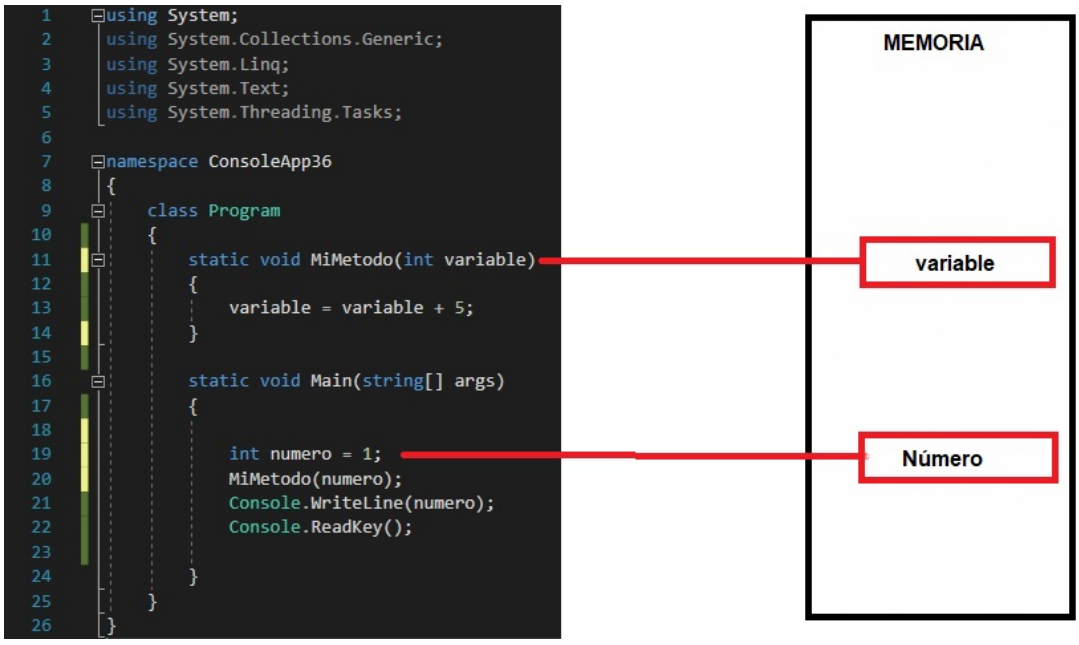
```
static void Par(int b)
{
int x;
x = (b % 2);
if (x == 0)
{
Console.WriteLine("\nEs par");
}else
{
Console.WriteLine("\nEs impar");
}
```

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

- Por valor o copia.
- Por referencia.

Paso de parámetros por valor

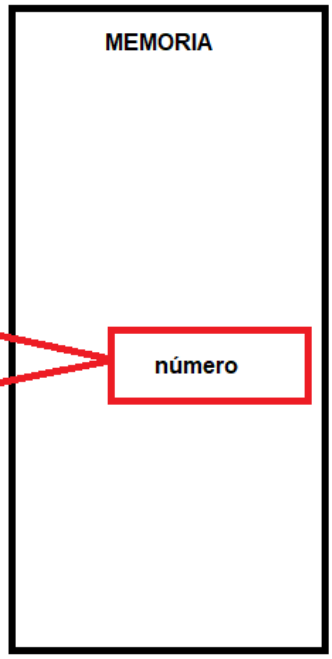
Cuando ejecutamos una función que tiene parámetros pasados por valor, se realiza una copia del parámetro que se ha pasado, es decir, que todas las modificaciones y/o cambios que se realicen se están haciendo en esta copia que se ha creado. El original no se modifica, de manera que no se altera su valor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



Paso de parámetros por referencia

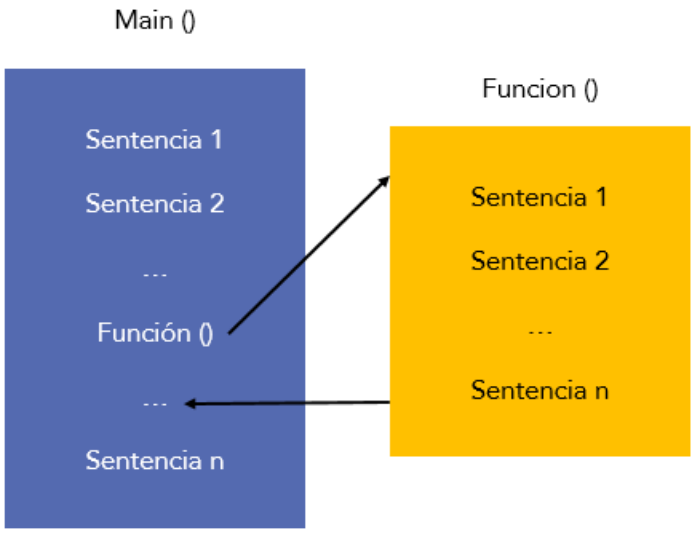
Sin embargo, cuando ejecutamos una función que tiene parámetros pasados por referencia, todas aquellas modificaciones que se realicen en la función van a afectar a sus parámetros, ya que se trabaja con los originales.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp36
{
    class Program
    {
        static void MiMetodo(ref int variable)
        {
            variable = variable + 5;
        }

        static void Main(string[] args)
        {

            int numero = 1;
            MiMetodo(ref numero);
            Console.WriteLine(numero);
            Console.ReadKey();

        }
    }
}
```



1.6. Ámbito de las llamadas a funciones

Una **función puede ser llamada** por 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 sola 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, iremos llamándolas en nuestra función principal *(main)* según sus prioridades de ejecución.

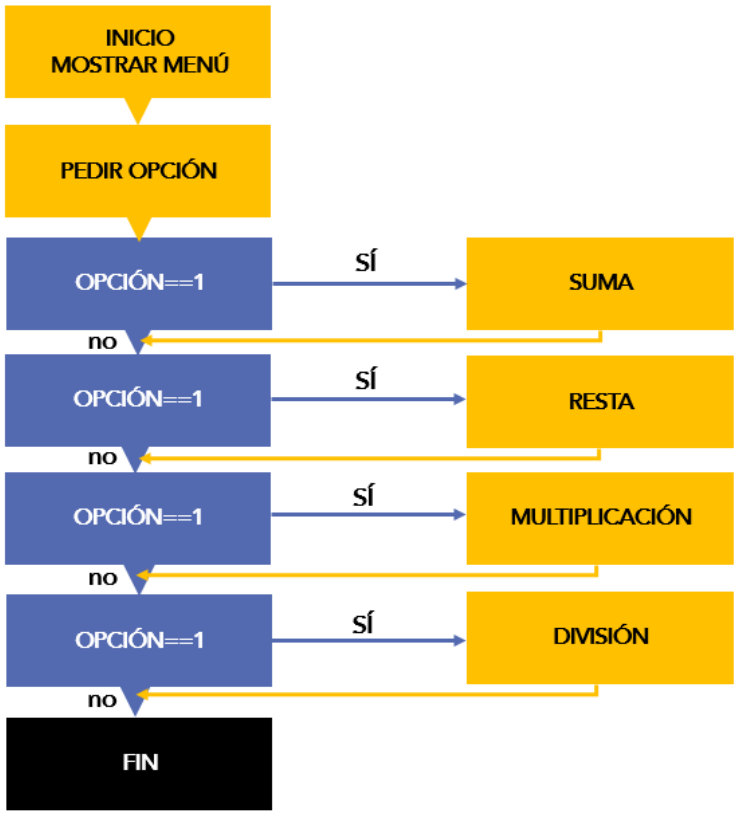
Podemos **dividir las funciones** en cuatro tipos básicos:

- Las que solo ejecutan código.
- Las que reciben parámetros.

Las funciones se utilizan para **no repetir tanto código** dentro de un mismo programa. 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 únicamente ejecutan código**

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



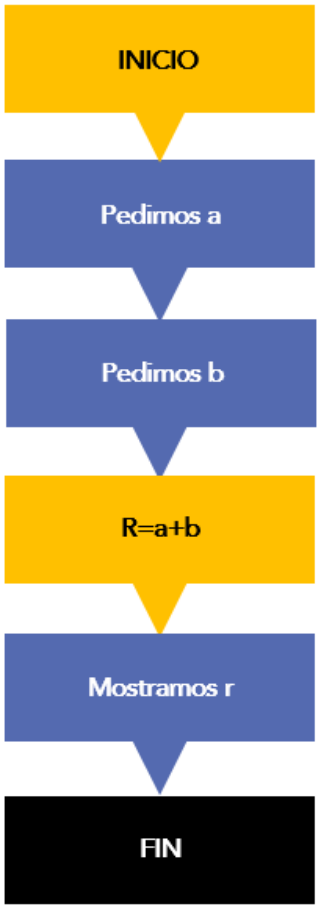
CÓDIGO:

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


{
class Program
{
// Esta es la función principal del programa
// Aquí inicia la aplicación
static void Main(string[] args)
{
// Variables necesarias
int opcion = 0;
string valor = "";
// Mostramos el menú
Console.WriteLine("1-Suma");
Console.WriteLine("2-Resta");
Console.WriteLine("3-Multiplicacion");
Console.WriteLine("4-Dividir");
// Pedimos la opción
Console.WriteLine("Cual es tu opción:");
```

```
{

}

// Implementamos la resta

if (opcion == 2)

{

}

// Implementamos la multiplicación

if (opcion == 3)

{

}

// Realizamos la división


if (opcion == 4)

{

}

}

}

}
```

La función ***main()*** contiene la lógica principal del programa en cuestión. Los bloques de código están vacíos de momento, y los 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



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;
```

```
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);


// Devolvemos el resultado

return res;


}
```

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

```
static void suma()
```

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 realizar un seguimiento de nuestro programa.

Una vez que la función ya está programada, no ocurre ningún cambio en nuestro programa, y no es porque no esté bien realizada. Es simplemente 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 *suma()*.

CÓDIGO:

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

- **Las que devuelven valores**

Cuando se llama a una función, esta debe 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 se utilizará la palabra reservada *return*, como, por ejemplo:

CÓDIGO:

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

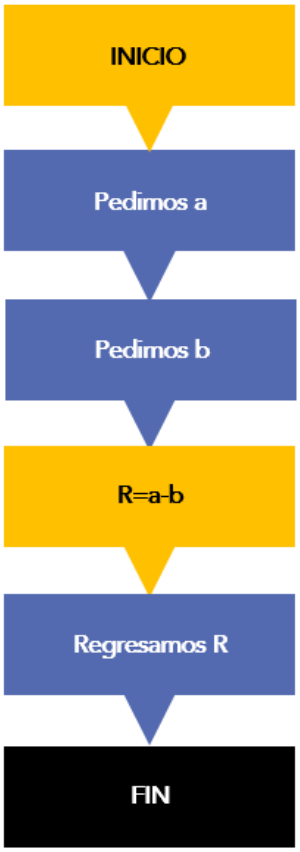
Cuando el programa encuentra la palabra *return*, la función ha concluido, es decir, ha finalizado.

A la hora de llamar a nuestra función, debemos almacenarla en una variable del mismo tipo que el valor que devuelva. La llamada a la función se escribiría así:

CÓDIGO:

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()
```

```
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*

CÓDIGO:

```
// Código para hacer la resta
if (opcion==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);
}
```

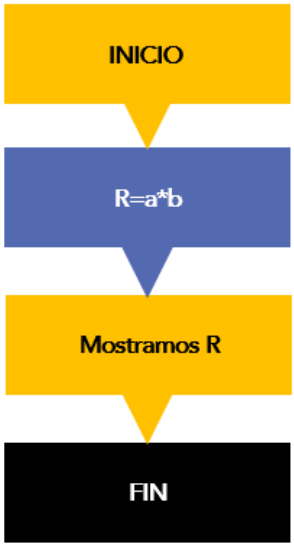
- **Las que reciben valores**

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 lugar de pedirlos. Estos parámetros pueden ser *int*, *float*, *string* 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 tienen (van a trabajar

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 de multiplicación sería de la siguiente forma:



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

CÓDIGO:

```
static void Multiplicacion(float num1, float num2)
{
        // Variables necesarias
        float res;

        // Calculamos el valor
        res=num1*num2;
```

Podemos comprobar que la función tiene dos parámetros tipo *float* que reciben el nombre de **"num1"** y **"num2"**. Se escriben separados mediante una coma, y su contenido va a ser el que se les pase 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 (opcion==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
```

Dentro del ***if*** creamos dos variables tipo *float*, y pedimos al usuario los valores con los que se va a trabajar. Estos 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 la función ya puede llevarse a cabo.

También podemos asignar el valor directamente en la llamada, como, por ejemplo:

CÓDIGO:

```
// Llamada a la función asignándole un valor

Multiplicacion(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 de valor 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, vamos a implementar la función de división, que va a recibir dos valores *float*. 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 solo puede hacer un *return*, guardamos el valor del resultado en *res* tanto si entramos en la primera condición como si vamos a utilizar la segunda. Almacenamos el valor correspondiente en la variable *res* y

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(opcion==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
```

```
}
```

Al igual que en los ejemplos anteriores, primero declaramos las variables que vamos a necesitar para nuestro programa, luego 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 lo mostramos en el *main()*.

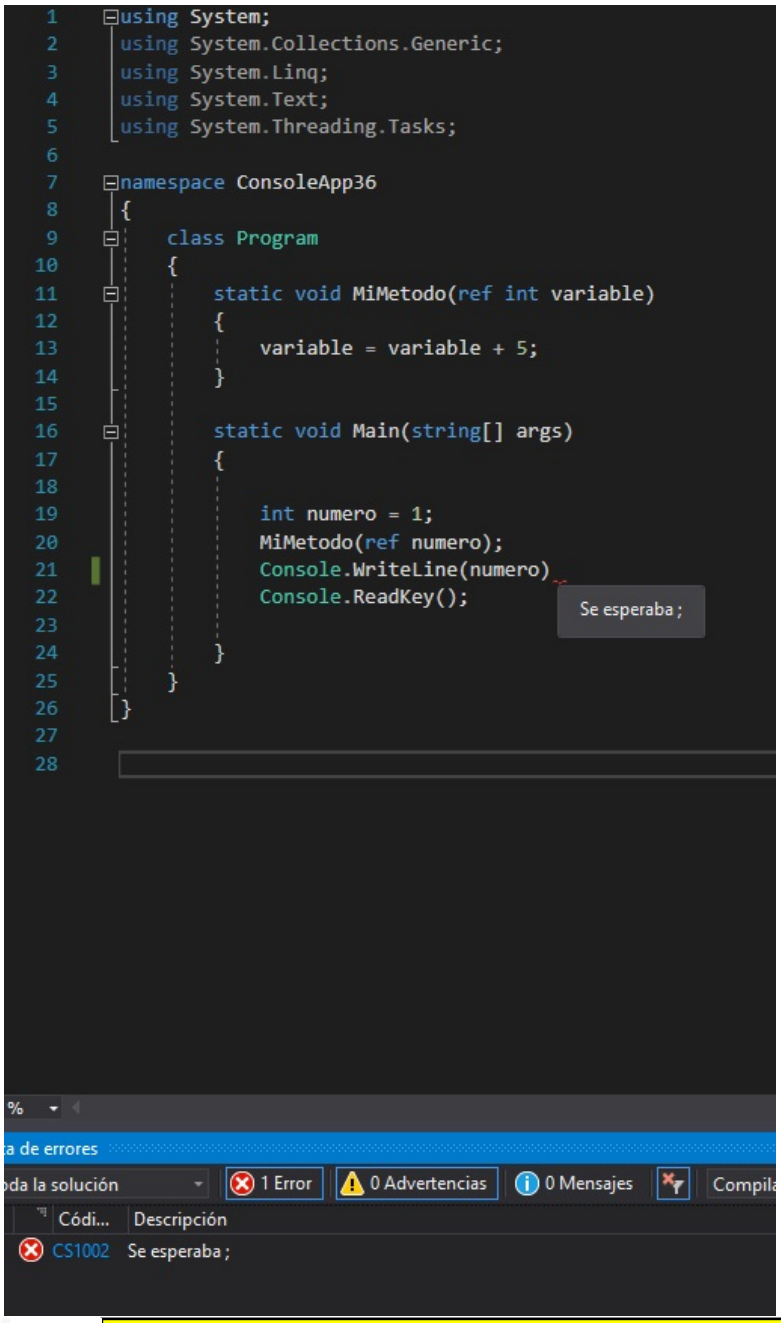
El **conjunto de todas estas operaciones** formaría nuestro programa.

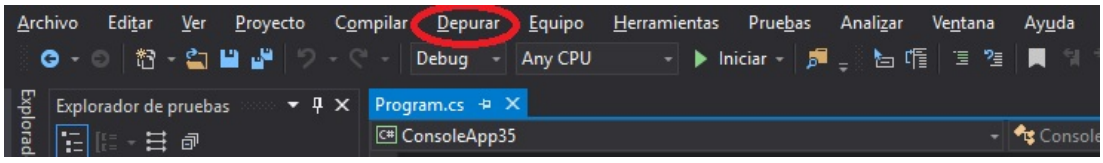
1.7. Prueba, depuración y comentarios de programas

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 no realiza la tarea que debería cuando se ejecuta.

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 con él.

Podemos encontrarnos **errores** en nuestro código, tanto sintácticos como semánticos, que pueden ir apareciendo 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.





Cuando ponemos una aplicación en modo depuración, Visual Studio tiene la posibilidad de visualizar, línea a línea *(F11)*, nuestro código para analizarlo de forma más detallada.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace ConsoleApp35
{
    public class Program
    {

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

        static void Main(string[] args)
        {
            int[] notas = new int[5];
            for (int i = 0; i < notas.Length; i++)  ≤1 ms transcurridos
            {
                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();

        }
    }
}
```

108 %

Automático

Nombre	Valor	Tipo
notas	{int[5]}	int[]
[0]	1	int
[1]	3	int

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. Estos 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, etcétera.

Una vez que estos errores se corrigen, podríamos decir que el programa se va a compilar y va a pasar a ejecutarse, aunque no sabemos todavía si va a realizar la función exacta para la que ha sido diseñado.

vinculan cuando el programa comienza 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**

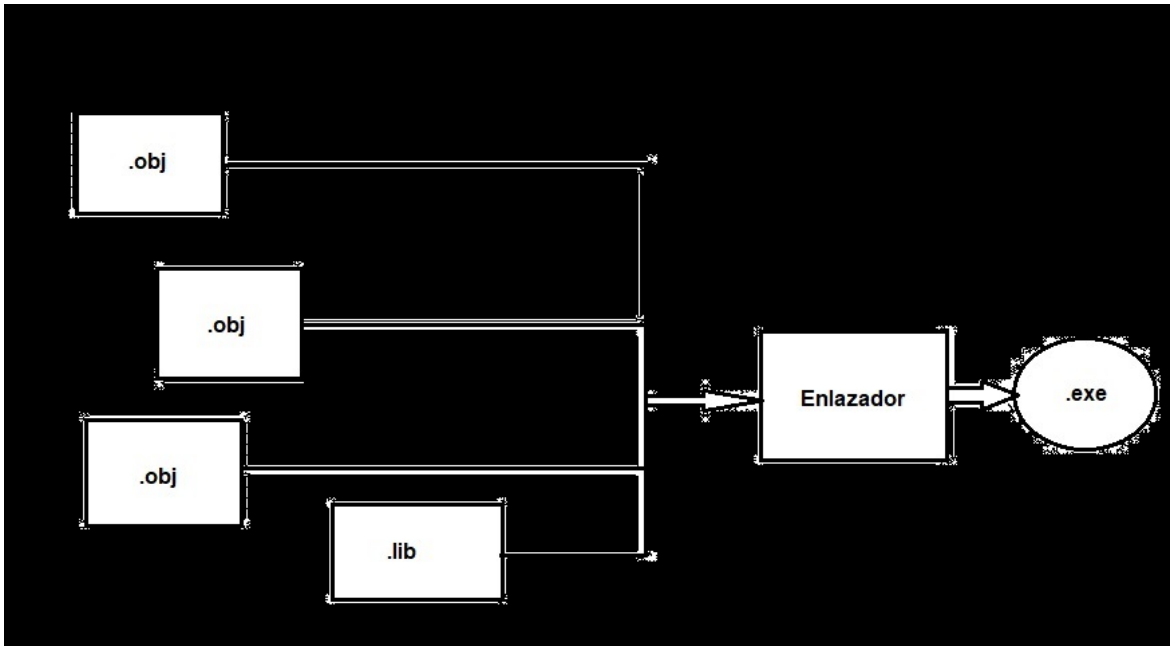
Si hemos conseguido llegar a la fase de ejecución, significa que nuestro programa ya no tiene errores, pero todavía no sabemos si va a funcionar o no. Algunos errores se detectan de forma automática, pero otros son más complicados de detectar. Estos pueden producir cambios inesperados, por lo que debemos 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 archivos que nos permiten llevar a cabo diferentes acciones y tareas sin necesidad de que el programador se preocupe de cómo están desarrolladas, solo debe entender cómo utilizarlas.

Las librerías en C# permiten hacer nuestros programas más modulares y reutilizables, facilitando además crear programas con funcionalidades bastante complejas.

Cuando compilamos nuestro programa, solamente se comprueba que se ha llamado de manera correcta a las funciones que pertenecen a las diferentes librerías que nos ofrece el compilador, aunque el código de estas funciones



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 es, a veces, una tarea complicada, por lo que los IDE nos proporcionan una herramienta denominada "Depurador" que nos facilita la tarea.

Uno de los motivos por lo que es recomendable utilizar el lenguaje de programación C# es que permite la **interoperación con otros lenguajes**, siempre que estos tengan:

- Acceso a las diferentes librerías a través de COM+ y servicios .NET.
- Soporte XML.
- Simplificación en administración y componentes gracias a un

1.9. Uso de librerías

Una **librería** es un conjunto de métodos relacionados con el mismo objetivo para poder ser reutilizado cada vez que el programador lo desee. Para la realización de este módulo, en los ejercicios prácticos, vamos a utilizar 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 mirar cuáles son los métodos que están implementados en la librería *Math* y podemos comprobar que existe un método llamado *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()
   {
```

```
    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 intervalo de 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();
    }
  }
}
```

```
//       -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_funcion(tipo par1, tipo par2, …, tipo parn)
{
```

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 esta la que llama a la primera.

La recursividad tiene la siguiente estructura:

- Un **caso base** que permita la finalización del programa.
- **Casos recursivos**, que son los que se van a encargar 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

El factorial de un número n se calcula:

$$n! = n*(n-1)*(n-2)*(n-3)*\ldots*1$$

Debemos saber qué es el factorial de un número. Por ejemplo:

3!=3*2*1

4!=4*3*2*1

CÓDIGO:

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

¿Cuál podría ser el caso base?

Debemos encontrar uno que finalice el programa. En este supuesto, siempre tenemos que buscar el caso más pequeño que existe. En nuestro ejemplo, 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!

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

De modo 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 llamada "factorial" (entero n) retorna entero:

CÓDIGO:

```
{
        si n>1 entonces
                // Caso recursivo
                devuelve n*factorial(n-1);
        Si no
                // Caso genérico
                devuelve 1;
        Fin Si;
}
```

Si lo traducimos al lenguaje de programación C#:

CÓDIGO:

```
static int factorial(int n)
{
        if (n>1)
```

UF3: FUNDAMENTOS DE GESTIÓN DE FICHEROS

1. Gestión de ficheros

Hasta aquí, hemos visto el uso de variables, las estructuras de datos y hemos manipulado la información de la que disponíamos. Esta información, una vez que finaliza la ejecución del programa, desaparece de la memoria, ya que ha estado almacenada en la memoria principal el tiempo que dura la ejecución del *software*.

La solución para que los datos persistan después de la ejecución es almacenarlos en un fichero. Entonces, cada vez que se ejecute la aplicación que trabaja con esos datos, podrá leer del fichero los que necesite y manipularlos.

1.1. Concepto y tipos de ficheros

Un **fichero** es 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

CÓDIGO:

<nombre_fichero>.<extensión>

<nombre_fichero> nombre del fichero.

<extensión> tipo de 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, bytes, líneas o registros 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 permite 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.

Por tanto, para manipular un fichero, que lo identificamos por un nombre, realizaremos tres operaciones:

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

- La información es binaria (conjunto de ceros y unos).
- 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 fácilmente el sistema de archivos.
- Evitan conflictos con sus nombres, ya que cada programa instala sus ficheros en directorios diferentes. Por tanto, en una misma máquina pueden existir dos ficheros identificados por el mismo nombre, ya que, como van a tener distinta ruta, los vamos a poder diferenciar.
- 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 inequívoca de este fichero, habrá que nombrar el camino que

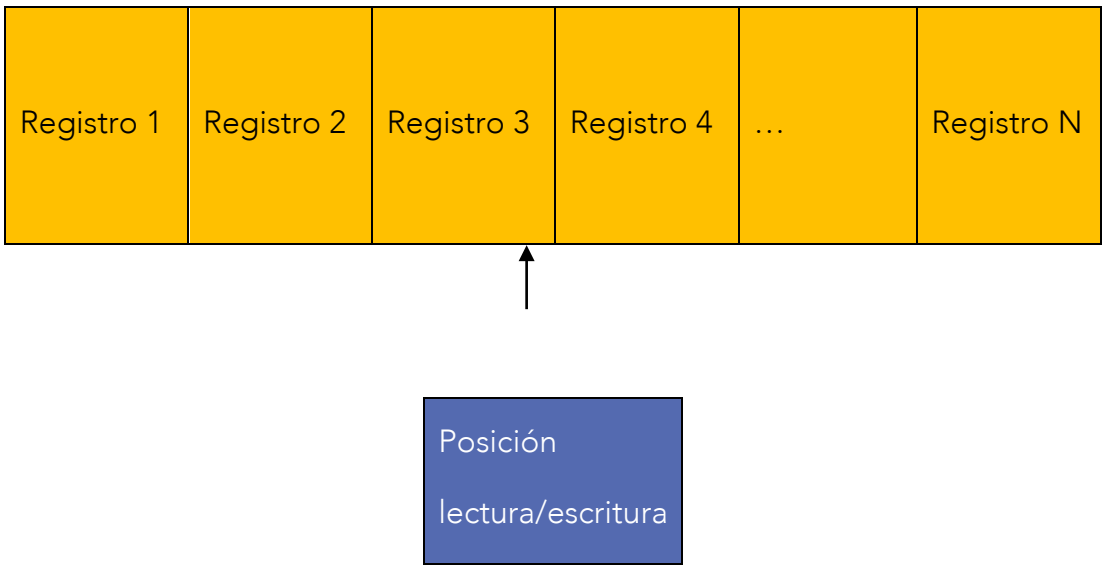
de la unidad en cuestión. En el sistema operativo Linux, empieza por "/".

- **Ruta relativa**: se le indica el camino del directorio desde la posición actual. Por tanto, la ruta no empezaría con la letra de la unidad.

Tipos de ficheros

- **Según su acceso**: según la forma de organizar la información, podemos distinguir 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.

    Solo 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 indicando una posición perteneciente a un conjunto de posiciones posibles. Debido a que los registros están organizados, estos 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 este.



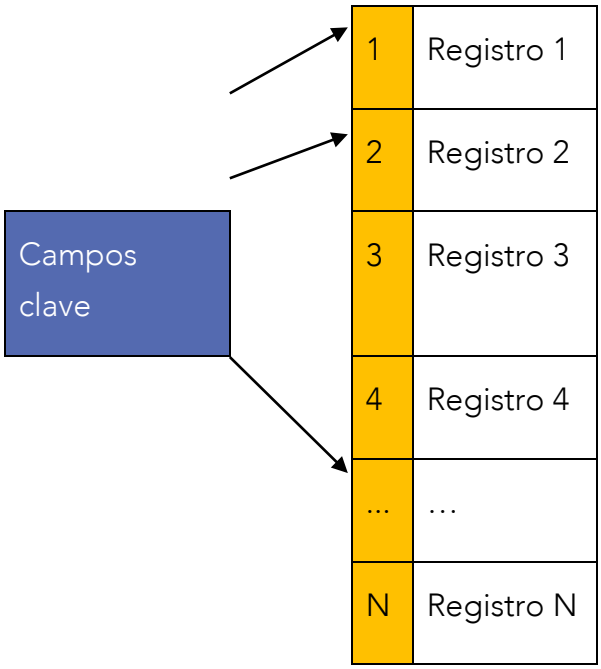
Para leer el "Registro 4", colocamos el puntero de lectura/escritura justo antes de su posición.

- **Secuencial indexado**: los ficheros indexados poseen un campo clave para ser identificados. Permiten el acceso secuencial y aleatorio 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 solo 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.



- **Según su estructura**:
  - **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 ceros y unos, de manera que se hace mucho más eficiente su almacenamiento.

1.2. Diseño y modulación de las operaciones sobre ficheros

1.2.1. Fundamentos de los flujos

Los **flujos** (también llamados *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 cualquier lenguaje de programación.

Dicho acceso se hace como una secuencia o sucesión de elementos que pueden entrar o salir del programa.

Los métodos que nos proporciona esta clase son:

FileStream (*string* nombre, *FileMode* modo)

FileStream (*string* nombre, *FileMode* modo, *FileAccess* acceso)

El primer método abre un flujo de entrada y salida que se vincula al fichero especificado en el nombre, mientras que el segundo método realiza lo mismo y añade el tipo de acceso: lectura, escritura o lectura/escritura.

El parámetro *nombre* es una cadena de caracteres que indica la ruta donde

@c:\users\AppData\file.txt (La ventaja de utilizar la @ es que las secuencias de escape no se procesan)

El parámetro *modo* puede tomar los siguientes valores:

CreateNew	Crea un nuevo fichero. Si existe, lanzará un error.
Truncate	Abre un fichero existente. El fichero será truncado a cero bytes de longitud.
Create	Crea un nuevo fichero. Si el fichero existe, será sobrescrito.
Open	Abre un fichero existente. Si el fichero no existe, lanzará un error.
OpenOrCreate	Abre un fichero si existe; si no, se crea un fichero nuevo.
Append	Abre un fichero para añadir datos al final del mismo si existe, o crea un fichero nuevo si no existe.

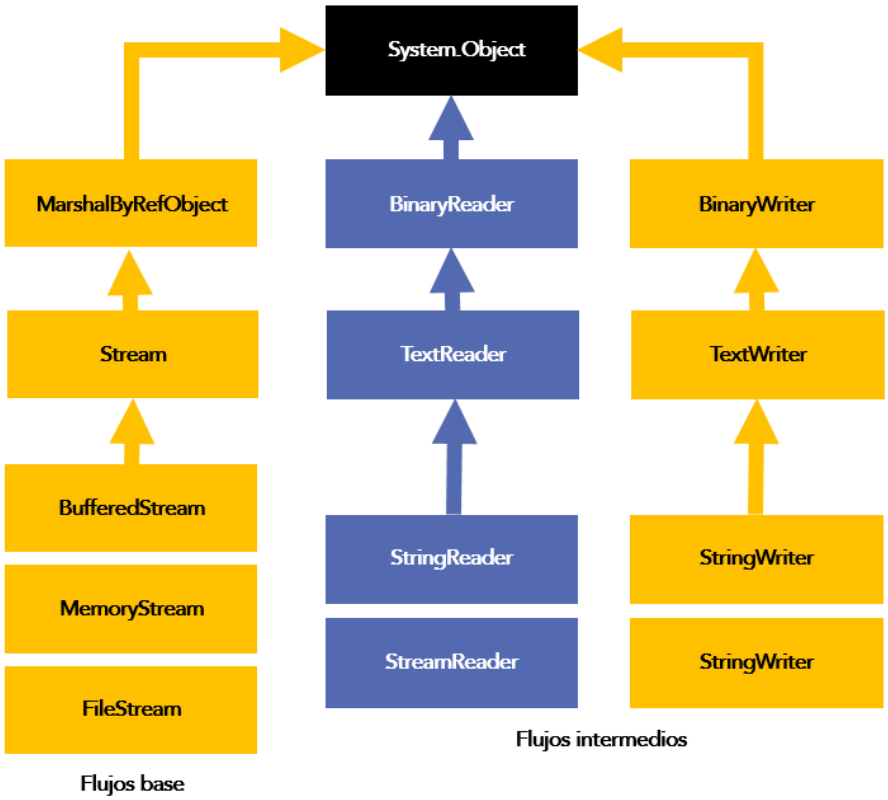
El parámetro *acceso* puede tomar los siguientes valores:

Read	Permite acceder al fichero para realizar operaciones de lectura.
ReadWrite	Permite acceder al fichero para realizar operaciones de lectura y escritura.
Write	Permite acceder al fichero para realizar operaciones de escritura.

El flujo de salida (escritura) también es en sentido unidireccional, ya que solo 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. El tipo de *stream* lo vamos a definir al iniciar el trabajo con ficheros, y no se podrá cambiar una vez abierto.

1.2.2. Clases de flujos

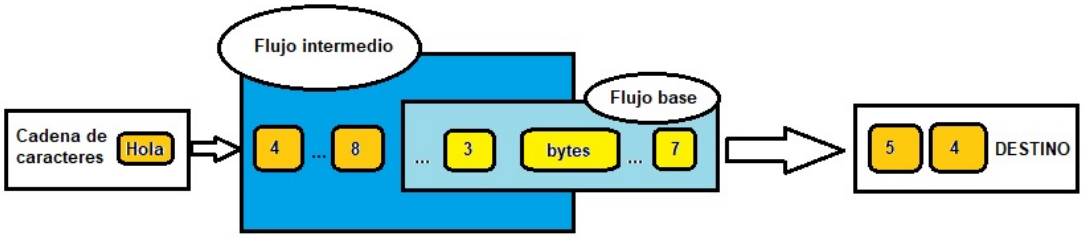


En este apartado utilizaremos las clases de C# pertenecientes a los dos tipos de ficheros: **binarios** y de **textos**.

Se pueden distinguir **dos tipos de flujos**:

pueda verse beneficiado por todas las funcionalidades que ofrezca el flujo base.

Podríamos decir que el flujo intermedio procesa la información, mientras que el base realiza la función de envío de bytes de un lugar a otro.



1.3. Operaciones sobre ficheros secuenciales

Las **tres operaciones básicas** que tenemos cuando trabajamos con ficheros son:

1. 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:

- **Añadir**: parecido al de **escritura**, nos 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.

2. Lectura/escritura

Tenemos que leer o escribir la información del fichero prestando atención tanto al fin de fichero en ficheros secuenciales como a la posición del puntero en ficheros aleatorios.

Las operaciones de lectura/escritura se pueden hacer de dos formas diferentes: para ficheros secuenciales o para ficheros aleatorios. Vamos a verlo detenidamente con el siguiente ejemplo:

- **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;
```

- **Escritura secuencial**

CÓDIGO:

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

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

Mientras existan datos a escribir
        Configurar registros a partir de los datos
f1.escribir(registro);
FinMientras

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

- **Lectura aleatoria**

CÓDIGO:

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

```
Mientras condición del programa Hacer
        Situar el puntero del fichero justo antes del registro requerido
        f1.leer(registro);
        operaciones con registro leído;
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
```

3. Cierre

Cuando cerramos el fichero, este queda liberado y termina el proceso de almacenamiento de información.

1.3.1. Clase FileStream

En el apartado 1.2.1 vimos los tipos de métodos que nos proporciona esta clase:

FileStream (*string* nombre, *FileMode* modo)

FileStream (*string* nombre, *FileMode* modo, *FileAccess* acceso)

y los diferentes valores que pueden tomar el parámetro *modo* y *acceso.*

Vamos a trabajar con estos métodos para ficheros de texto y ficheros binarios.

Ficheros de texto

Los datos pueden ser escritos o leídos de un fichero carácter a carácter en un formato portable (UTF-8: *UCS Transformation Format,* formato de 8 bits en el que cada carácter Unicode emplea uno o más bytes) utilizando los flujos de las clases ***StreamWriter*** y ***StreamReader***.

CÓDIGO:

```
using System.IO; // Importación de la clase System.IO
public class Escribir
{
Public static void Main(string [ ] args)
{
```

```
fs = new FileStream (“doc.txt”, FileMode.Append, FileAccess.Write);

sw= new StreamWriter(fs);


Console.WriteLine (“Escriba las líneas que quiere almacenar en el fichero” \n” +

                    “Finalice cada línea pulsando <enter>\n” +

                    “Para finalizar, pulsar la tecla <enter>\n”);

// Leer una línea de la entrada estándar

str = Console.ReadLine();

While (str.length != 0)

{

     // Escribir la línea leída en el fichero

    sw.WriteLine(str);

    // Leer la línea siguiente

    str = Console.ReadLine();

}

Sw.close();

Fs.Close();

}

}
```

CÓDIGO:

```
using System.IO; // Importación de la clase System.IO
```

```
StreamReader sr = null;
String str;

sr= new StreamReader ("doc.txt");

// Leer del fichero una línea de texto

Str = sr.ReadLine();

While (str !=null)
{
      // Mostrar la línea leída
    Console.WriteLine(str);
     // Leer la línea siguiente
     str = sr.ReadLine();
}
sr.Close();

}
}
```

Para estos casos, el espacio de nombres System.IO proporciona las clases *BinaryReader* y *BinaryWriter*, las cuales permiten leer y escribir, respectivamente, datos de cualquier tipo primitivo en formato binario.

CÓDIGO:

```
BinaryReader(Stream flujo)

BinaryWriter(Stream flujo)
```

Podemos poner un ejemplo de escritura en un fichero binario llamado "datos.dat" usando UTF-8 al codificar los caracteres:

CÓDIGO:

```
FileStream f = new FileStream("datos.dat", FileMode.Create, FileAccess.Write);

BinaryWriter f_Binario = new BinaryWriter(f);

f_Binario.Write("Este mensaje se escribirá en UTF-8 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, FileAccess.Read);

BinaryReader f_Binario = new BinaryReader(f);

Console.WriteLine("Leído de datos.dat: {0}", f_Binario.ReadString());
```

Ejemplo de escritura en ficheros binarios

CÓDIGO:

```
FileStream f = new FileStream("datos.dat", FileMode.Create, FileAccess.Write);
BinaryWriter f_binario = new BinaryWriter(f);

// Almacenamos el nombre, la dirección y el teléfono en el fichero
f_binario.Write("MiNombre");
f_binario.Write("MiDireccion");
f_binario.Write("12345678-A");

f_binario.Close();
f.Close();
```

Los métodos más utilizados de esta clase son:

Write(byte)	Escribe un valor de tipo *byte*.
Write(char)	Escribe un valor de tipo *char*.
Write(short)	Escribe un valor de tipo *short*.
Write(int)	Escribe un valor de tipo entero de 32 bits.
Write(long)	Escribe un valor de tipo entero de 64 bits.

Close	Cierra el flujo y libera los recursos adquiridos.

Ejemplo de lectura en ficheros binarios

CÓDIGO:

```
FileStream f = new FileStream("datos.dat", FileMode.Create, FileAccess.Read);

BinaryReader f_binario = new BinaryReader(f);


// Leemos el nombre, la dirección y un teléfono desde el fichero

Nombre = f_binario.ReadSting();

Dirección = f_binario.ReadString();

Teléfono = f_binario.ReadInt64();


f_binario.Close();

f.Close();
```

Los métodos más utilizados de esta clase son:

ReadByte	Devuelve un valor de tipo *byte*.
ReadChar	Devuelve un valor de tipo *char*.

ReadInt64	Devuelve un valor de tipo *long*.
ReadDouble	Devuelve un valor de tipo *double*.
ReadString	Devuelve una cadena de caracteres en formato UTF-8.
Close	Cierra el flujo y libera los recursos adquiridos.

Bibliografía

Joyanes, L. (2000). *Programación en C++: algoritmos, estructuras de datos y objetos*. McGraw Hill.

Cerezo, Y.; Peñalba, O. y Caballero, R. (2007). *Iniciación a la programación en C#. Un enfoque práctico*. Delta publicaciones.

Bell, D. y Parr, M. (2010). *C# para estudiantes*. México: Editorial Pearson.

Ceballos, F. J. (2011). *Microsoft C#. Curso de programación*. 2ªEdición. RA-MA Editorial.

Webgrafía

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