

PROGRAMACIÓN - A

ADMINISTRACIÓN DE SISTEMAS INFORMÁTICOS EN RED

DESARROLLO DE APLICACIONES MULTIPLATAFORMA

DESARROLLO DE APLICACIONES WEB

ILERNA

© Ilerna Online S. L., 2023

Maquetado e impreso por Ilerna Online S. L.

© Imágenes: Shutterstock

Impreso en España - Printed in Spain

Reservado todos los derechos. No se permite la reproducción total o parcial de esta obra, ni su incorporación a un sistema informático, ni su transmisión en cualquier forma o por cualquier medio (electrónico, mecánico, fotocopia, grabación u otros) sin autorización previa y por escrito de los titulares del copyright. La infracción de dichos derechos puede constituir un delito contra la propiedad intelectual.

Ilerna Online S. L. ha puesto todos los recursos necesarios para reconocer los derechos de terceros en esta obra y se excusa con antelación por posibles errores u omisiones y queda a disposición de corregirlos en posteriores ediciones.

2.^a edición: marzo 2023

ÍNDICE

Programación - A

1. Introducción a la programación.....	6
2. Estructura de un programa informático.....	16
2.1. Bloques de un programa informático	19
2.2. Variables. Usos y tipos	25
2.3. Constantes y literales. Tipos y utilidades.....	28
2.4. Operadores del lenguaje de programación	30
2.5. Conversiones de tipos de clase	34
2.6. Comentarios al código.....	37
3. Programación estructurada	40
3.1. Fundamentos de programación	42
3.2. Introducción a la algoritmia y herramientas de diseño.....	43
3.3. Ciclo de vida de un proyecto software	43
3.4. Prueba de programas.....	45
3.5. Tipos de datos: simples y compuestos	46
3.6. Estructuras de selección (instrucciones condicionales)	50
3.7. Estructuras de repetición	54
3.8. Estructuras de salto.....	57
3.9. Tratamiento de cadenas	58
3.10. Depuración de errores	61
3.11. Documentación de programas.....	63
3.12. Entornos de desarrollo de programas.....	65
4. Programación modular.....	66
4.1. Concepto	67
4.2. Ventajas e inconvenientes.....	67
4.3. Análisis descendente (<i>top down</i>).....	68
4.4. Modulación de programas. Subprogramas	69
4.5. Llamadas a funciones (métodos). Tipos y funcionamiento.....	71
4.6. Ámbito de las llamadas a funciones	79
4.7. Prueba, depuración y comentarios de programas	83
4.8. Concepto de librerías.....	86
4.9. Uso de librerías.....	89
4.10. Introducción al concepto de recursividad.....	91

5. Gestión de ficheros.....	96
5.1. Concepto y tipos de ficheros	97
5.2. Diseño y modulación de las operaciones sobre ficheros	103
5.3. Operaciones sobre ficheros secuenciales	106
5.4. Control de excepciones.....	115
Bibliografía / webgrafía	116
Solucionario	117



1

(1+x+y+2a)-(3a+3g+x)

```
selection at the end -ad  
    mirror_ob.select= 1  
    E=mc2  
    context.scene.objects.active  
    "Selected" + str(modifier)  
    mirror_ob.select = 0  
    bpy.context.selected_objects  
    E=mc2  
    Eta.objects[one.name].sel
```

1+x+y+2a
2+...+2a+2a
- OPERATOR CLASSES -----
45-4a-3

1+x+y+2a+21

45-4a-3

{x-12-y+n...}

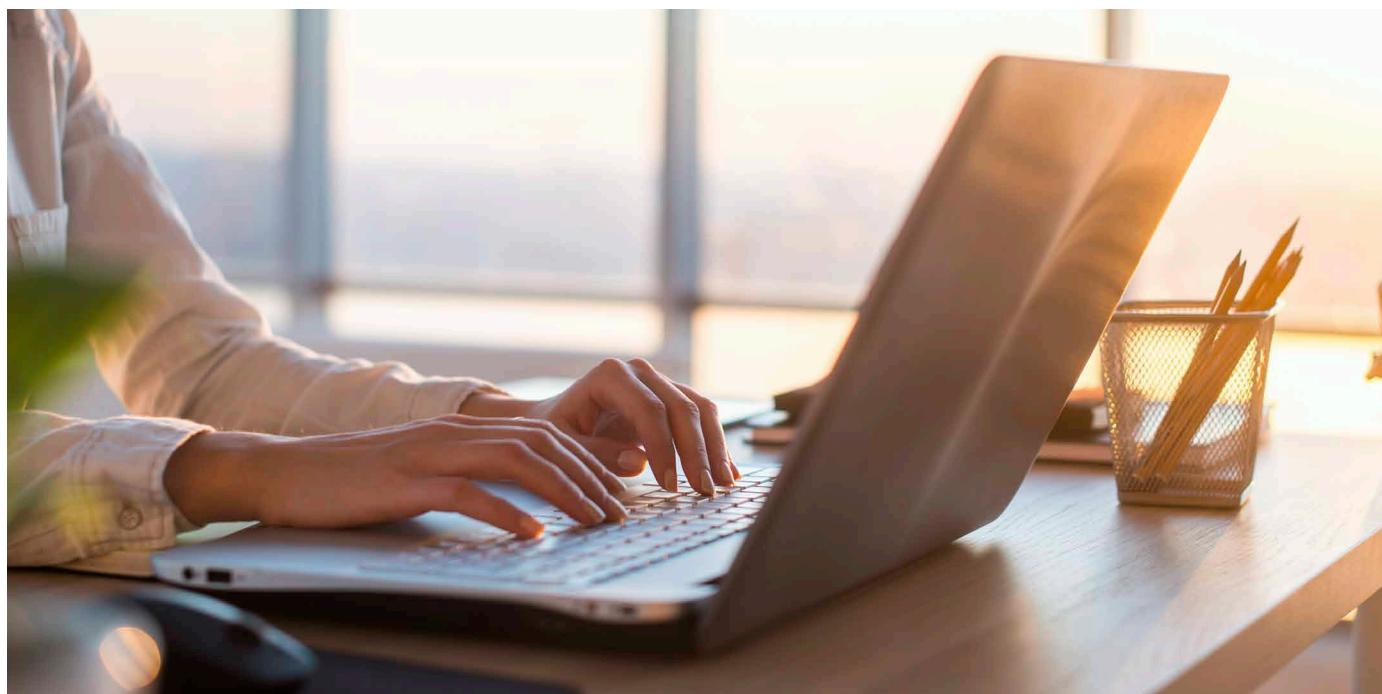
1lim h-->0

x=0 xn {x-12-y+n...}

types.Operator:
X mirror to the selected
object_mirror_mirror_x
or X" (1+x+y+2a)-(3a+3g+x)]
5+x+k+2a+21
2+...+2a+2a
context):
context.active_object is not

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 determinado problema.

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**:

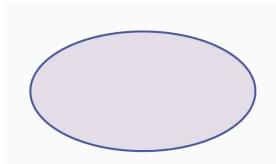
- Deber ser **finito**: formado por un conjunto limitado de sentencias.
- Debe ser **legible**: es importante crear códigos *limpios* y fáciles de leer con tabulaciones y espacios que diferencien las partes del programa.
- Debe ser **modificable**: debe ser sencillo el proceso de actualización o modificación 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 programa estructurado será fácil de leer y modificar, y estará compuesto de subprogramas que permitirán la reutilización de código.

Representaciones gráficas

Para representar un algoritmo, podemos utilizar dos herramientas gráficas: los **diagramas de flujo** y el **pseudocódigo**.

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.

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



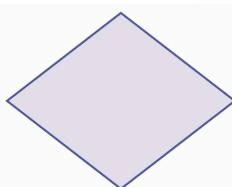
Indican inicio o fin de programa.



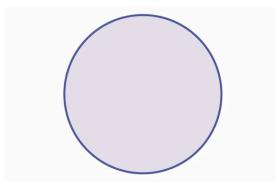
Representan una instrucción, un paso a dar o un proceso. Por ejemplo: `cont = cont + 1`.



Operaciones de entrada/salida de datos. Por ejemplo: visualiza en pantalla suma.



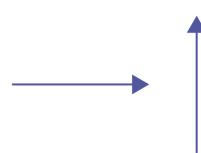
Usaremos este símbolo si nos encontramos en un punto en el que se realizará una u otra acción en función de una decisión. Por ejemplo, si `miVariable = 0`, hacemos la instrucción 1. Si no, hacemos la instrucción 2.



Conector. Permite unir diferentes zonas del diagrama de forma que se redefine el flujo de ejecución hacia otra parte del diagrama.



Representa un método o subprograma.



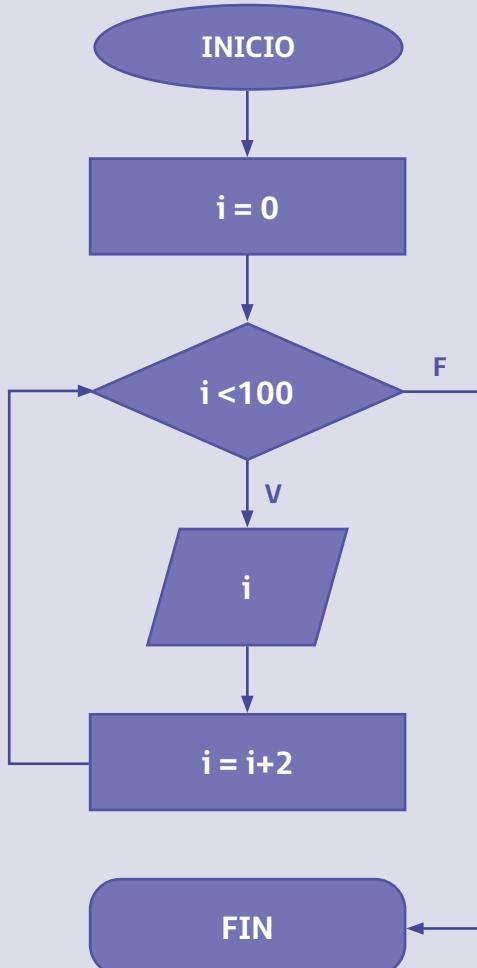
Flechas para indicar la dirección de flujo.

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.

EJEMPLO

Se representa el siguiente diagrama de flujo, que muestra todos los números pares del 0 al 98:



La otra herramienta para representar los algoritmos es el **pseudocódigo**.

Es la manera de representar las distintas sentencias que va a realizar nuestro algoritmo con un lenguaje cercano al natural. Por tanto, es un lenguaje que no se puede ejecutar en una máquina.

Características del pseudocódigo

- Comienza con la palabra *algoritmo* seguido de su nombre.
- A continuación, tenemos la secuencia de instrucciones de este algoritmo.
- Finaliza con la estructura **FinAlgoritmo**.
- No es necesaria la indentación (sangrado), aunque es recomendable.
- No se diferencia entre mayúsculas ni minúsculas.

Algoritmo triángulo

Definir área, altura, base de tipo números reales;

Escribir "Introduce la altura del triángulo:";

Leer altura;

Escribir "Introduce la base del triángulo:";

Leer base;

Área = (altura*base)/2;

Escribir "El área del triángulo es:", área;

FinAlgoritmo

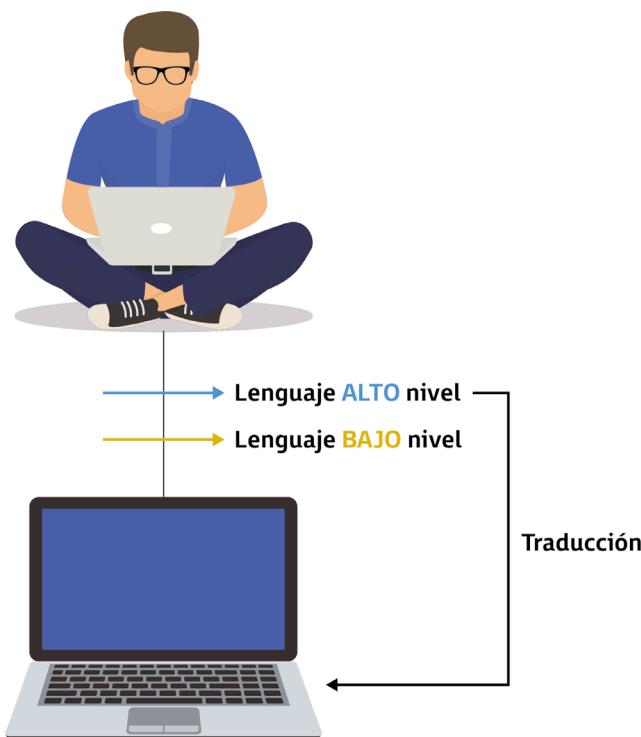
Comunicación de un sistema informático

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

- Los dos agentes principales, **usuario** y **máquina**.
- El **canal**. Para continuar con el ejemplo con el que estamos explicando el proceso de programación, podemos decir que el canal por el que se comunica nuestro usuario será el teclado.
- El **lenguaje**. 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, y así lograr el entendimiento.



Ahora bien, para solventar el problema de la comunicación, tenemos lenguajes de programación de tres tipos: **alto nivel**, **nivel medio** y **bajo nivel**. Los lenguajes de alto nivel están más cerca del lenguaje que habla el usuario, mientras que los lenguajes de bajo nivel se acercan más a las estructuras del lenguaje de la máquina. Por su parte, los lenguajes de nivel medio toman características de ambos, como por ejemplo el lenguaje C.



```

static function day_list() {
    $return = array();
    $result = mysql::query("SELECT * FROM image_date ORDER BY shot_date DESC");
    while($day = mysql::fetch($result)) {
        $tmp_studio_list = array();
        $shots_result = mysql::query("SELECT DISTINCT(studio) as studio, COUNT(*) as count FROM image WHERE day_id = '$id' AND enabled='1' GROUP BY studio");
        while($studio_list = mysql::fetch($shots_result)) {
            $day_info = metadata::day_info($day->shot_date, $studio_list->studio, "quick");
            $tmp_studio_list[] = array("studio" => $studio_list->studio, "count" => $day_info->count, "title" => $day_info->title);
        }
        $day->studio_list = $tmp_studio_list;
        $return[$day->shot_date] = $day;
    }
    return $return;
}

static function day_images_list($date, $studio) {
    global $global_studio_list;
    if(!array_key_exists($studio, $global_studio_list)) die("error_studio");
    if($global_studio_list[$studio]['count'] == 0) die("no_images");
}

```



Para facilitar el trabajo, implementaremos nuestro **código con lenguajes de alto nivel**, de modo que necesitaremos un proceso de traducción para convertir el programa escrito en lenguaje máquina.

Características del lenguaje de alto nivel

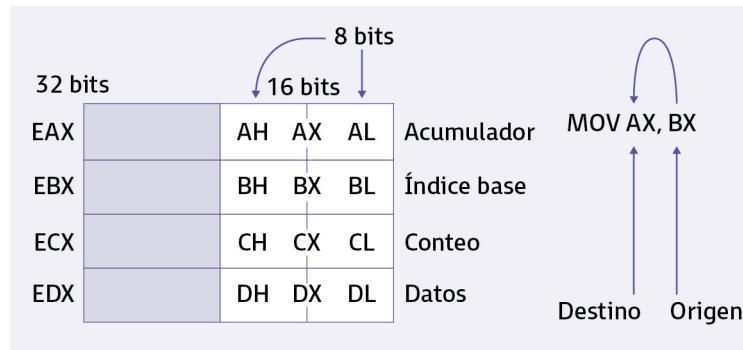
- Es totalmente independiente de la máquina y, por tanto, muy portable.
- Muy utilizado en el mercado laboral informático.
- Tanto las modificaciones como las actualizaciones son muy fáciles de realizar.
- Para la tarea de traducción de código necesitamos un compilador y un enlazador con librerías del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

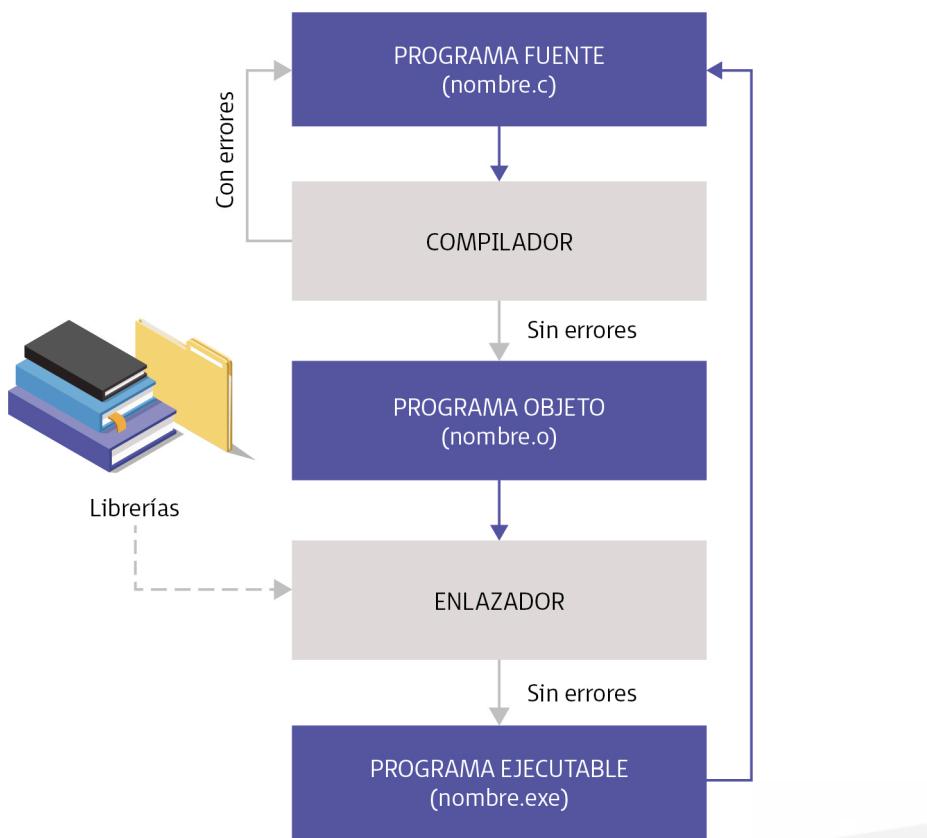
Algunos ejemplos de lenguajes de alto nivel son java, php, c#, VB.NET, Python...

Características del lenguaje de bajo nivel

- Está relacionado con las características del hardware.
- Es fácilmente traducible al lenguaje máquina.
- Es totalmente dependiente de la máquina.

Un ejemplo de lenguaje de bajo nivel es Ensamblador. Este lenguaje trabaja con registros a más bajo nivel.





A lo largo de esta unidad formativa hablaremos de la **estructura de un programa informático** y de sus elementos principales, como son las variables, 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 esta traducción simultánea.

En esta asignatura vamos a centrarnos en el lenguaje de **programación C#**. Otro lenguaje muy utilizado actualmente es el lenguaje Java. En la siguiente tabla mostramos las características de ambos:

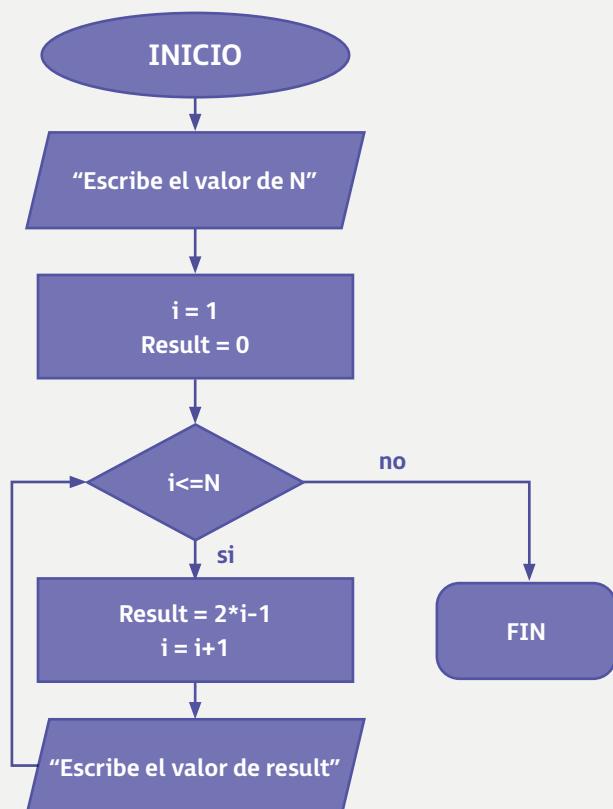
C#	Java
Es un lenguaje orientado a objetos.	Es un lenguaje orientado a objetos.
Contiene dos categorías generales de tipos de datos integrados: tipos de valor y tipos de referencia.	Los argumentos son siempre pasados por valor independientemente del tipo de variable.
La migración de C# a Java es sencilla porque mantiene una sintaxis muy similar.	La migración de Java a C# es sencilla porque mantiene una sintaxis muy similar.
C# está incluido en la plataforma .NET que tiene a su disposición el recolector de basura del CLR (Common Language Runtime o entorno en tiempo de ejecución).	Gracias al recolector de basura, se elimina la necesidad de la liberación explícita de memoria.
En C# todos los tipos derivan de una superclase común llamada <code>System.Object</code> . A diferencia de Java, esta característica también se aplica para los tipos básicos.	Java soporta sincronización de múltiples hilos de ejecución (<i>multithreading</i>) a nivel de lenguaje.
C# es un lenguaje compilado.	Java es compilado, en la medida en que su código fuente se transforma en un código máquina muy parecido al lenguaje ensamblador.
Se genera un código intermedio llamado MSIL (Microsoft Intermediate Language) y luego ese código se vuelve a compilar para obtener el formato nativo en la máquina en la que se va a ejecutar.	Java es interpretado, ya que ese código máquina se puede ejecutar directamente sobre cualquier máquina la cual tenga el intérprete y el sistema de ejecución en tiempo real.

ponte a prueba

¿Cómo debe ser un programa informático?

- a) Legible.
- b) Modificable.
- c) Eficiente.
- d) Todas las opciones anteriores son correctas.

¿Cuál de las siguientes opciones sobre el siguiente diagrama de flujo es correcta?



- a) Muestra los n primeros números impares.
- b) Realiza un producto de dos números.
- c) Muestra los n primeros múltiplos.
- d) Realiza la suma de dos números.

C# tiene dos categorías tipos de datos integrados: por valor y por referencia.

- a) Verdadero.
- b) Falso.



2

ESTRUCTURA DE UN PROGRAMA INFORMÁTICO



El lenguaje de programación C# es un lenguaje de alto nivel que pertenece al conjunto de lenguajes de .NET. C# es una evolución de C++ y mucha de la sintaxis está heredada de este lenguaje.

Con C#, podemos crear aplicaciones con una interfaz de texto, que denominamos aplicaciones de consola.

Una de las ventajas de aprender este lenguaje es que es independiente de la plataforma donde lo ejecutemos (sea Intel o AMD).

Para desarrollar nuestros programas en C#, utilizaremos un IDE. Un entorno de desarrollo integrado (IDE) es un software diseñado para el desarrollo de aplicaciones con un conjunto de herramientas integradas en una interfaz gráfica de usuario (GUI).

Sus características son:

- **Editor de código fuente:** ayuda al programador a escribir el código, resaltando la sintaxis de forma visual y rellenando de forma automática las distintas opciones que nos proporciona el lenguaje.
- **Depurador:** ayuda al programador a localizar sus errores de código.
- **Compilación del código fuente:** un código binario entendible por la máquina.

Para plataformas **Windows** y **Mac OSX**, podemos desarrollar con **IDE Visual Studio**. El proyecto que implementa el desarrollo de .NET en **GNU/Linux** es el proyecto **Mono**, concretamente el entorno de desarrollo **MonoDevelop**.

Con estos programas, vamos a ir generando los diferentes proyectos. Cada vez que creamos 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#.

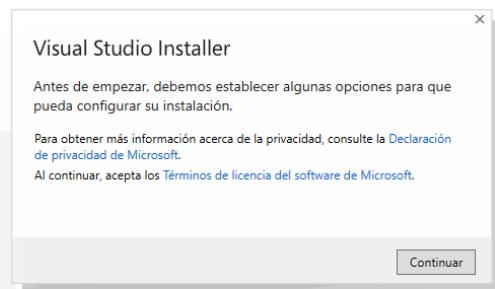
Instalación de Visual Studio

Para instalar nuestro Visual Studio, vamos a la página oficial de Microsoft, donde, en la parte de *downloads*, tendremos varias opciones.

Así, descargamos Visual Studio Community, que es la versión gratuita:



Una vez descargado, procedemos a la instalación.



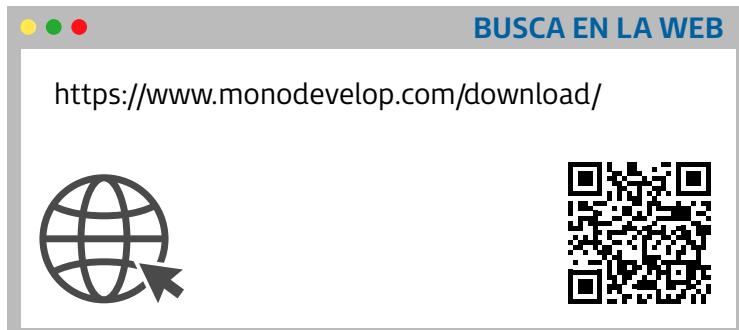
Durante la instalación, nos dará a elegir entre una serie de paquetes instalables.

Nosotros, para realizar nuestras aplicaciones de consola, añadiremos el siguiente paquete:

Cargas de trabajo	Componentes individuales	Paquetes de idioma	Ubicaciones de instalación
Web y nube (4)			
Desarrollo de ASP.NET y web Compila aplicaciones web con ASP.NET Core, ASP.NET, HTML/JavaScript y contenedores, e incluye compatibilidad...	Desarrollo de Azure Proyectos, herramientas y SDK de Azure para desarrollar aplicaciones en la nube y crear recursos mediante .NET...		
Desarrollo de Python Edición, depuración, desarrollo interactivo y control de código fuente de Python.	Desarrollo de Node.js Compile aplicaciones de red escalables con Node.js, un entorno de ejecución JavaScript controlado por eventos...		
Móviles y de escritorio (5)			
Desarrollo de escritorio de .NET Compila WPF, Windows Forms y aplicaciones de consola mediante C#, Visual Basic y F# con .NET Core y .NET...	Desarrollo para el escritorio con C++ Cree aplicaciones modernas de C++ para Windows con las herramientas que prefiera, como MSVC, Clang, CMake o...		
Desarrollo de la plataforma universal de Windows Cree aplicaciones para la Plataforma universal de Windows con C#, VB y, opcionalmente, C++.	Desarrollo para dispositivos móviles con .NET Compile aplicaciones multiplataforma para iOS, Android o Windows con Xamarin.		

Instalación de MonoDevelop

Para la instalación de este IDE en plataformas como Ubuntu o Debian, podemos seguir las indicaciones de la página oficial:

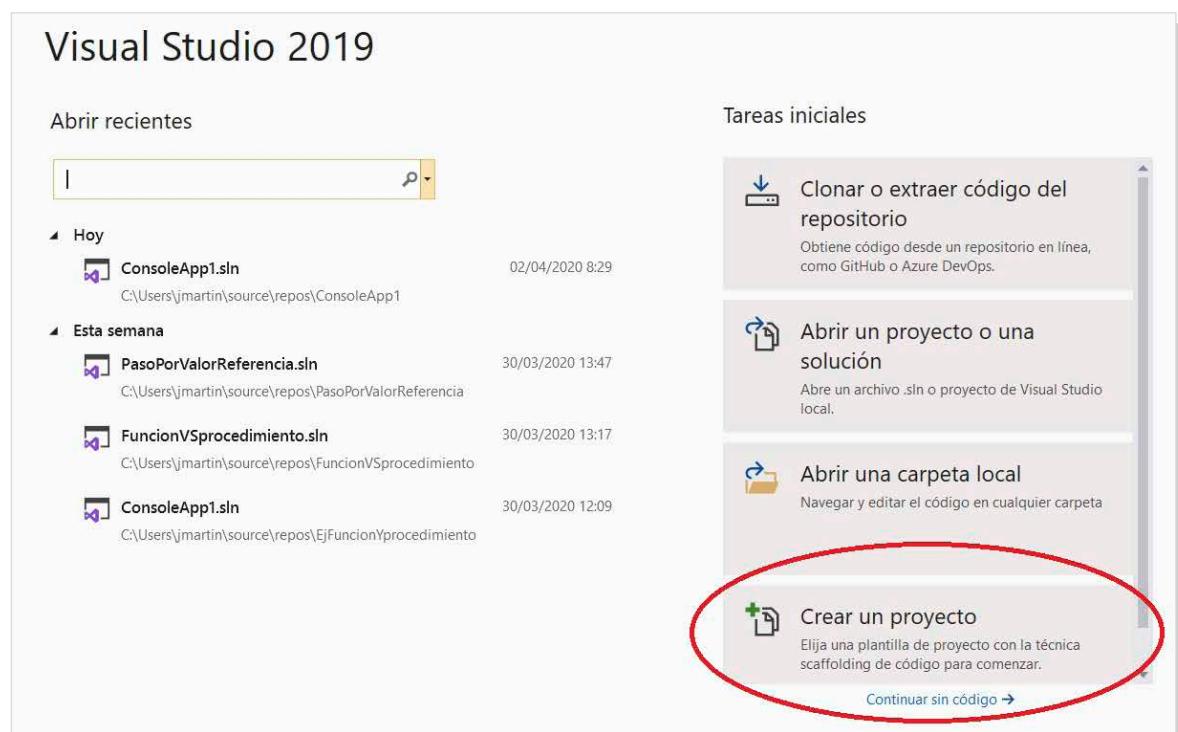


2.1. BLOQUES DE UN PROGRAMA INFORMÁTICO

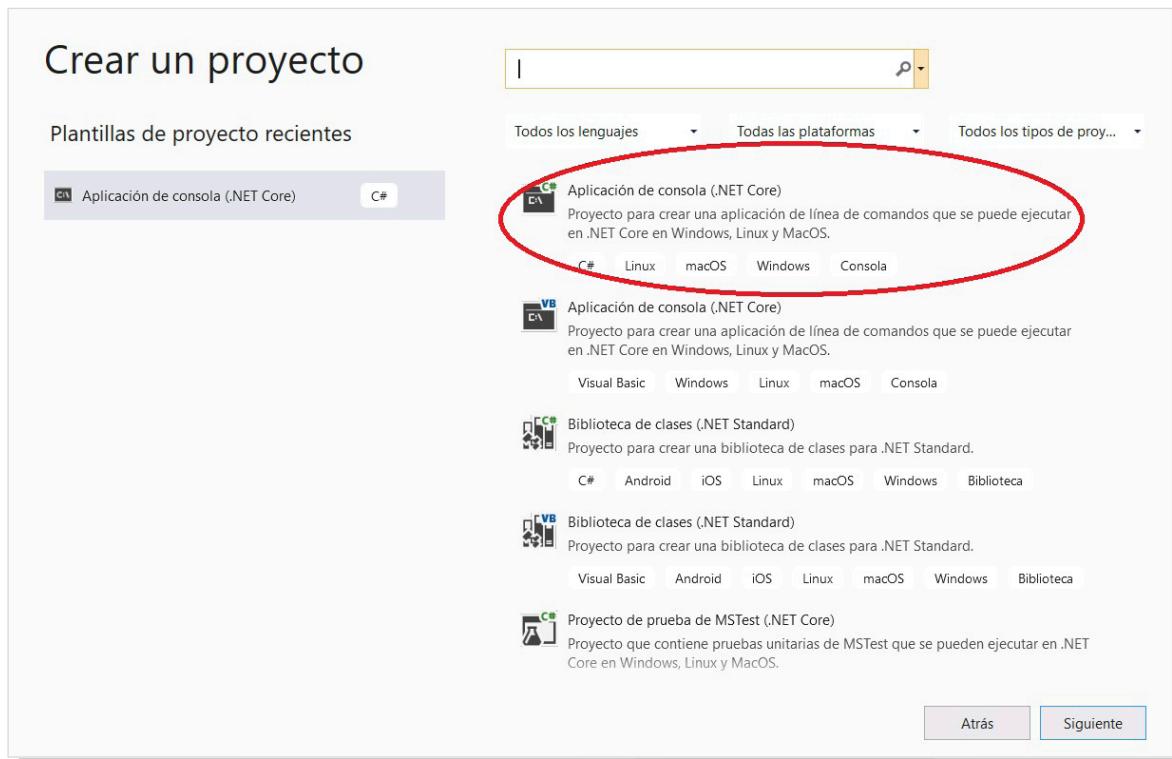
Estructura de un programa
youtu.be/MTJoNPqXT1E

Podemos definir un programa como una secuencia de **instrucciones separadas por un punto y coma** que se van a ir agrupando en diferentes bloques mediante llaves.

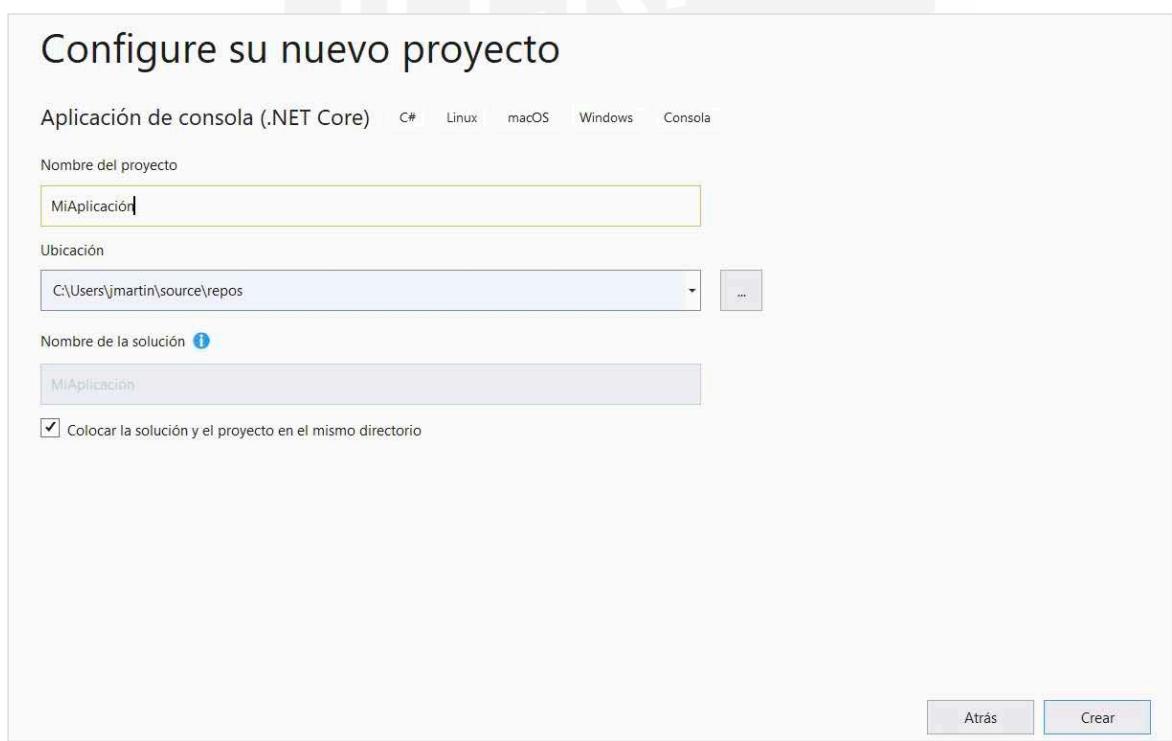
Para crear un proyecto, abrimos nuestro Visual Studio y seleccionamos la opción de *Crear un proyecto*.



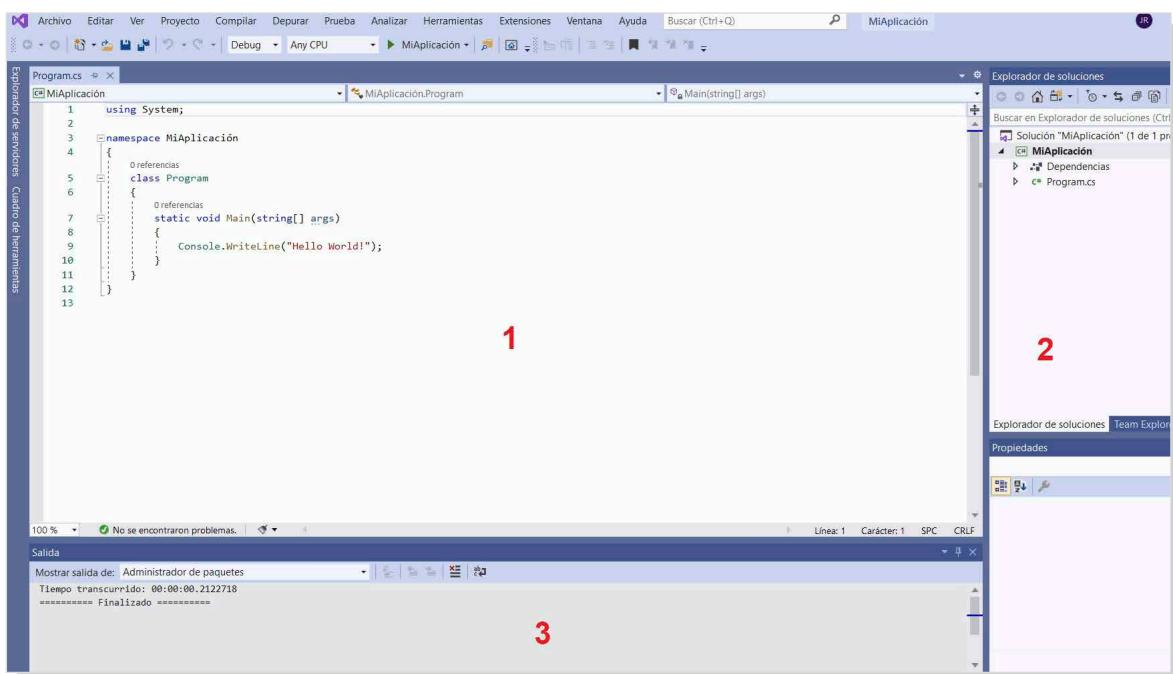
Posteriormente, elegiremos la opción de *Crear un aplicativo de consola*.



Nombramos nuestra aplicación con el nombre que queramos y seleccionamos la ruta donde lo guardaremos.



Cuando hayamos configurado estos parámetros, se nos abrirá la interfaz de nuestro IDE, que estará formado por tres partes principales:



- La primera parte será nuestro editor de código. El desarrollador implementará todas las instrucciones necesarias para completar su programa.
- La segunda es el explorador de soluciones. En esta parte de nuestro IDE, veremos los archivos generados de nuestro aplicativo.
- La tercera parte es la salida. Nos proporcionará el resultado de si la compilación ha sido correcta o, por el contrario, si ha habido un error, así como también el porqué de ese error.

Cuando creamos el proyecto, se crea la siguiente estructura:

```
using System;
namespace MiAplicación
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

La estructura que Visual Studio nos crea por defecto es la clase **program**. Una **clase** es una estructura de datos que está compuesta por atributos o variables (de las que hablaremos más adelante) y métodos o funciones. El método principal se denomina **Main**.



De hecho, siempre que ejecutemos un programa, el primer método que se ejecutará será el Main. Este método tiene una serie de **argumentos**. Estos argumentos son cadenas de caracteres que podemos pasar línea de comandos (CMD):

```
C:\ Símbolo del sistema  
C:\Users\jmartin\source\repos\MiAplicación\bin\Debug\netcoreapp3.1>MiAplicación.exe "hola", "pepe"  
hola,pepe  
C:\Users\jmartin\source\repos\MiAplicación\bin\Debug\netcoreapp3.1>
```

Nosotros trabajaremos con la *Consola* de nuestro entorno de desarrollo.

Entrada y salida por pantalla

Para establecer la comunicación entre el usuario y el programa, emplearemos la clase **Console**.

- Para la salida por pantalla, podemos utilizar los métodos de esta clase, **Write** y **Writeline**:
 - El método Write() escribe un valor o valores especificado(s) en el flujo de salida estándar, que generalmente será nuestra pantalla.
 - El método WriteLine() escribe un valor o valores especificado(s) en el flujo de salida estándar, que generalmente será nuestra pantalla, y se realiza un salto de línea en nuestra consola.

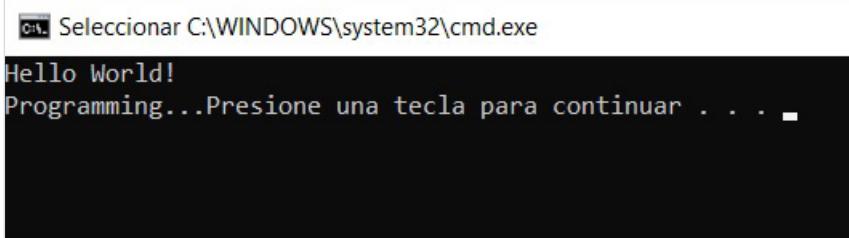
```

using System;

namespace MiAplicación
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
            Console.Write("Programming..."); 
        }
    }
}

```

La salida será:



```

[...] Seleccionar C:\WINDOWS\system32\cmd.exe
Hello World!
Programming...Presione una tecla para continuar . . .

```

Para la entrada de datos de usuario, utilizaremos el método **ReadLine()**. Este método leerá el dato introducido por el usuario y devolverá un dato de tipo cadena (string).

```

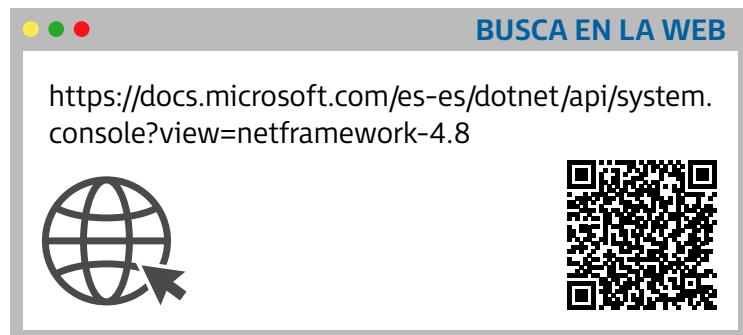
using System;
namespace MiAplicación
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Introduce un
dato");
            string dato = Console.ReadLine();
        }
    }
}

```

Otra forma para recoger los datos de entrada de usuario es el método **Read()**. Este método leerá el dato introducido por el usuario y devolverá un dato de tipo entero (int).

Asimismo, otro método utilizado por la clase Console es el método **.ReadKey()**. En este caso, se obtiene el carácter presionado por el usuario y se muestra esta tecla por la pantalla.

Se pueden consultar más métodos de esta clase en el siguiente enlace:



ponte a prueba

¿Qué es un IDE?

- a) El lenguaje C# es considerado como un IDE.
- b) El lenguaje java es considerado como un IDE.
- c) Es un software diseñado para el desarrollo de aplicaciones.
- d) Ninguna de las respuestas es correcta.

Siempre que ejecutemos un programa, el primer método que se ejecutará será el Main.

- a) Verdadero.
- b) Falso.

¿Qué realiza el siguiente código?

```
static void Main(string[] args)
{
    Console.WriteLine("Hello World!");
}
```

- a) Muestra por pantalla args.
- b) Muestra por pantalla "Hello World!".
- c) Muestra por pantalla Hello World!
- d) Realiza una lectura de datos de args.

2.2. VARIABLES. USOS Y TIPOS

Una **variable** es una estructura de datos que ocupan un espacio en memoria y pueden variar a lo largo de un programa.

Cuando **declaramos una variable**, establecemos el tipo de dato (entero, cadena, carácter, etc.) y su identificador.

```
class Program
{
    static void Main(string[] args)
    {
        int numero;
        numero = 10;
        char letra = 'a';
        string cadena = "Esta es una cadena";
    }
}
```

Tipos de datos en C#:

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

- **Tipo String:** son cadenas de caracteres. Por ejemplo:
string cadena = "Hola mundo";
- **Tipo de dato enumerado:** enum días_semana {lunes, martes, miercoles};
- **Tipos matriz:**

– **Unidimensionales:**

```
int[] array = new int[5];
int[] array2 = new int[] { 1, 3, 5, 7, 9 };
int[] array3 = { 1, 2, 3, 4, 5, 6 };
```

– **Multidimensionales:**

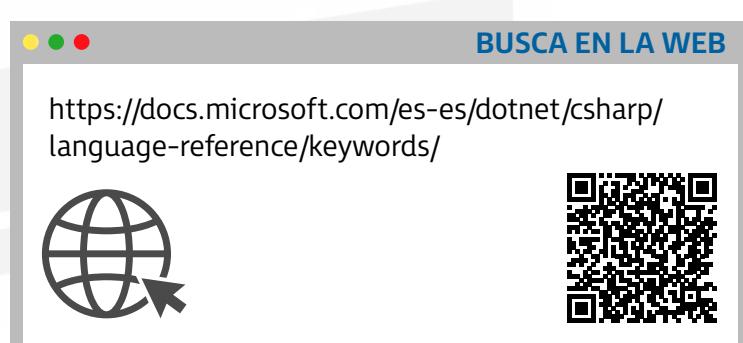
```
int[,] multiDimensionalArray1 = new int[2, 3];
int[,] multiDimensionalArray2 = { { 'a', 'b', 'c' }, { 'd', 'e', 'f' } };
```

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”) o por dígitos (del 0 al 9). Se puede utilizar el subrayado (“_”).
- No debe comenzar con un dígito.
- Distingue entre mayúsculas y minúsculas.
- No puede llevar espacios en blanco.
- No se pueden utilizar palabras reservadas. Por ejemplo, string es una palabra reservada como tipo de dato cadena.

A continuación, encontraréis una lista de las palabras claves (o identificadores) predefinidos por el compilador:



Para declarar una variable:

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

Y en C# lo pondríamos de la siguiente forma:

```
int num;
```

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

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

```
int num=5;
```

Definimos una variable de tipo entero, denominada **num** y le asignamos el valor de 5.

Debemos utilizar un tipo u otro de variable, dependiendo del tipo de dato que queremos tratar.

A grandes rasgos, podemos dividir las variables en dos grandes bloques, dependiendo del ámbito en el que se encuentren:

- **Variables locales:** aquellas declaradas dentro de un método o función.
- **Variables globales:** aquellas declaradas fuera de un método o función. Podemos acceder a ellas desde cualquier punto del programa.

Más adelante definiremos el concepto de *ámbito de variables*, cuando hablemos del concepto de programación modular.

No es buena práctica utilizar variables globales.

AVISO

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.





ponte a prueba

¿Cuál de las siguientes opciones es una característica de una variable?

- a) Ocupa espacio en memoria.
- b) Puede variar su contenido a lo largo del programa.
- c) Es de un tipo de dato.
- d) Todas las respuestas son correctas.

¿Qué tipo de dato es un char?

- a) Carácter.
- b) Entero.
- c) String.
- d) Decimal.

Las variables globales son aquellas que están declaradas dentro de un método o función.

- a) Verdadero.
- b) Falso.

2.3. CONSTANTES Y LITERALES. TIPOS Y UTILIDADES

Constantes

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**.

El valor se establece en tiempo de compilación y no se puede cambiar su contenido.

```
const <tipo> <nombree_variable>;
```

Por ejemplo:

```
const int num;
```

Después, declaramos una constante de tipo entero que denominamos *num*.

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

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

Las constantes se definen fuera del cuerpo de cualquier función, normalmente al principio del programa.

- **Literales**

Mediante un literal podemos representar de forma explícita todos los valores que pueden tomar los tipos básicos del lenguaje.

Seguidamente, vamos a explicar los diferentes tipos de literales que podemos encontrarnos en C#.

- **Literales enteros:**

- **Decimales:** sin ningún prefijo. Por ejemplo: int dLiteral = 10;
- **Hexadecimales:** con el prefijo de 0x o 0X. Por ejemplo: int hLiteral= 0x3F;

- **Literales reales:**

- El literal sin sufijo o con el sufijo d o D es del tipo **double**.
- Por ejemplo: double num = 3.4d;
- Double num2= 3.4;
- El literal con el sufijo f o F es del tipo **float**.
- Por ejemplo: float variable = 4.5F;
- El literal con el sufijo m o M es del tipo **decimal**.
- Por ejemplo: decimal numero = 3.2m;

- **Literales de lógicos:** representados por los valores lógicos *true* (verdadero) y *false* (falso).

- **Literales de carácter:** cualquier carácter en C# podemos representarlo con comillas simples. Por ejemplo, 'a' correspondería al carácter a.



ponte a prueba

¿Con qué palabra reservada se define una constante?

- a) Const
- b) Enum
- c) Array
- d) Ninguna de las opciones es correcta

¿Qué representa el literal \n?

- a) Una nueva línea
- b) Un carácter nulo
- c) Un salto de página
- d) Una tabulación vertical

Existen caracteres especiales que pueden representarse de la siguiente forma:

Carácter	Representación
Comilla simple	\'
Comilla doble	\\"
Carácter nulo	\0
Retroceso	\b
Salto de página	\f
Nueva línea	\n
Retorno de carro	\r
Tabulación horizontal	\t
Tabulación vertical	\v

2.4. OPERADORES DEL LENGUAJE DE PROGRAMACIÓN

Operadores aritméticos

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.

El **operador de incremento ++** puede ser aplicado como prefijo o sufijo de una variable.

El resultado de `num++` es el valor de `num` antes de la operación:

```
int num = 100;
Console.WriteLine(num);    // salida por pantalla: 100
Console.WriteLine(num++); // salida por pantalla: 100
Console.WriteLine(num);    // salida por pantalla: 101
```

El resultado de `++num` es el valor de `num` después de la operación:

```
double num = 10.4;
Console.WriteLine(num);    // salida por pantalla: 10.4
Console.WriteLine(++num); // salida por pantalla: 11.4
Console.WriteLine(num);    // salida por pantalla: 11.4
```

Asimismo, sucede lo mismo con el operador decrecimiento `--`.

La prioridad de los operadores (de mayor a menor prioridad) es:

1. Operadores de incremento `x++` y decremento `x--` sufijos.
2. Operadores de incremento `++x` y decremento `--x` prefijos.
3. Operadores de multiplicación, división y resta `*`, `/` y `%`.
4. Operadores de suma y resta `+` y `-`.

Operadores booleanos

Operadores booleanos	
<code>&</code>	AND lógico. Evalúa ambos operandos.
<code>&&</code>	AND lógico. Evalúa el operando derecho solo si es necesario.
<code> </code>	OR lógico. Evalúa ambos operandos.
<code> </code>	OR lógico. Evalúa el operando derecho solo si es necesario.
<code>^</code>	XOR lógico.
<code>!</code>	Negación lógica.

El operador **AND lógico &** calcula el AND lógico de sus operandos. El resultado de `a & b` es true si `a` y `b` se evalúan como true. Si no, el resultado es false.

El operador **AND lógico condicional &&** también evalúa el operador AND lógico de sus operandos, pero no evalúa el operando derecho si el izquierdo se evalúa como false.

El **operador |** calcula el operador OR lógico de sus operandos. El resultado de $a | b$ es true si a o b se evalúan como true. Si no, el resultado es false.

El operador **OR lógico condicional ||** también evalúa el operador OR lógico de sus operandos, pero no evalúa el operando derecho si el izquierdo se evalúa como true.

El operador \wedge es el operador OR exclusivo o **XOR**. La tabla de la verdad (tabla que muestra el resultado de una combinación de valores) es la siguiente:

a	b	$a \wedge b$	$a \vee b$	$a \& b$
T	T	F	T	T
T	F	T	T	F
F	T	T	T	F
F	F	F	F	F

El operador ! calcula la negación lógica de su operando, es decir, si el operando es true, devolverá el valor false. Si el operando es false, devolverá el valor true.

Operadores de comparación e igualdad

Operadores de comparación	
>	Mayor
<	Menor
\geq	Mayor o igual
\leq	Menor o igual
$=$	Igual
\neq	Desigualdad
$=$	Asignación

El operador $<$ devuelve *true* si el operando izquierdo es menor que el derecho.

El operador $>$ devuelve *true* si el operando izquierdo es mayor que el derecho.

El operador `<=` devuelve *true* si el operando izquierdo es menor o igual que el derecho.

El operador `>=` devuelve *true* si el operando izquierdo es mayor o igual que el derecho.

El operador de igualdad `==` devuelve *true* si sus operandos son iguales.

El operador de desigualdad `!=` devuelve *true* si sus operandos son distintos.



ponte a prueba

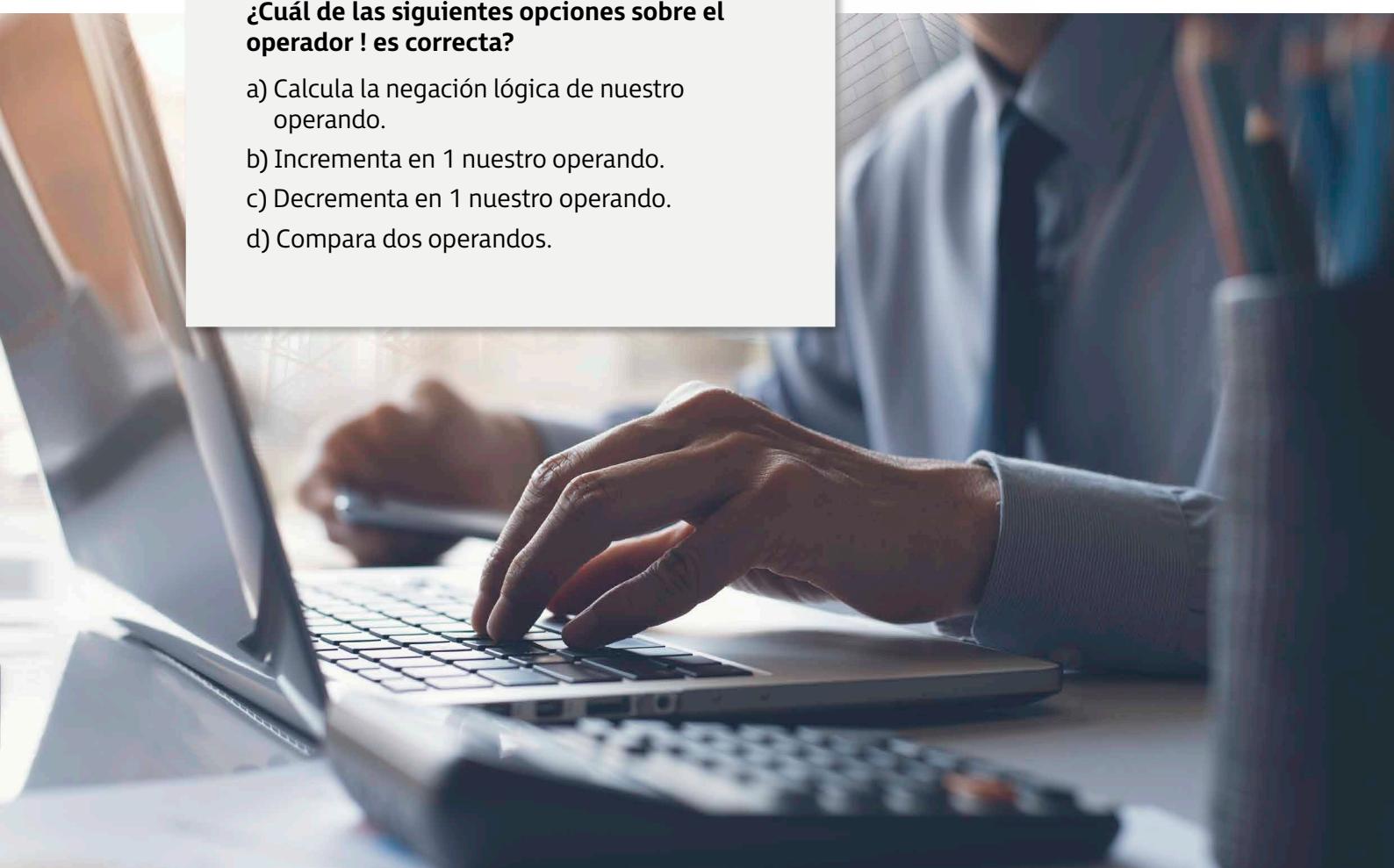
¿Cuál es el resultado del siguiente código?

```
int x = 1;  
num++;  
Console.WriteLine(num);
```

- a) 1.
- b) 2.
- c) 0.
- d) Ninguna de las repuestas es correcta.

¿Cuál de las siguientes opciones sobre el operador `!` es correcta?

- a) Calcula la negación lógica de nuestro operando.
- b) Incrementa en 1 nuestro operando.
- c) Decrementa en 1 nuestro operando.
- d) Compara dos operandos.



2.5. CONVERSIONES DE TIPOS DE CLASE

Como hemos visto anteriormente, en C#, cuando declaramos una variable, no se puede volver a declarar, ni asignar un valor de otro tipo.

Por tanto, si tenemos una variable de tipo entero, no podemos asignar un valor de tipo string.

Este fragmento de código daría un error:

```
int numero=10;
numero = "a"; // No se puede convertir
               implícitamente el tipo 'string' en 'int'
```

Sin embargo, en ocasiones, podemos convertir el valor de una variable de un tipo de dato a otro tipo. A estas conversiones se les denomina *conversiones de tipos*.

Podemos tener diferentes clases de conversiones:

- **Conversiones implícitas:** son aquellas que se producen cuando el valor que se va a almacenar se puede almacenar en la variable sin truncarse ni redondearse.

Por ejemplo, una variable de tipo long puede almacenar datos de hasta 64 bits. Por tanto, puede almacenar cualquier dato de tipo int (32 bits).

```
int a = 300;
long numero=a;
```

- **Conversiones explícitas:** cuando no hay posibilidad de realizar una conversión implícita es cuando debemos indicar al compilador que se pretende realizar una conversión que pueda producir una pérdida de datos. En este caso, se antepone a la variable el nuevo tipo de datos entre paréntesis.

```
double num = 100.85;
int b;
b = (int)num;
System.Console.WriteLine(b);

// La salida por pantalla será 100
```

Parse y TryParse

Con los métodos Parse y TryParse, podemos convertir una cadena en varios tipos numéricos: int, decimal, double...

El método Parse devuelve el número convertido. El método TryParse devuelve un valor booleano indicando si la

conversión se realizó correctamente o no, y devuelve el número convertido.

```

int numValido = Int32.Parse("-100");
Console.WriteLine(numValido);

// La salida por pantalla será -100

while (Int32.TryParse("50", out int miNumero))
    // El argumento miNumero, se pasa por referencia. Lo veremos más
    adelante en el apartado 4.5. Llamadas a funciones. Tipos y funcionamiento.

Console.WriteLine(miNumero);

// La salida por pantalla será 50

```

Convert

Es una clase que convierte un tipo de datos en otro. Algunos métodos son:

Método	Descripción
ToBoolean	Convierte un valor especificado en un valor booleano.
ToChar	Convierte un valor especificado en un carácter.
ToDecimal	Convierte un valor especificado en un número decimal.
ToDouble	Convierte un valor especificado en un número de punto flotante.
ToInt16	Convierte un valor especificado en un entero de 16 bits.
ToInt32	Convierte un valor especificado en un entero de 32 bits.
ToInt64	Convierte un valor especificado en un entero de 64 bits.
ToString	Convierte un valor especificado en su representación de cadena.

```

double numero = 30.5;
string cadena = Convert.ToString(numero);

// Mostraría por pantalla la cadena "30.5"

```



INTERNA

EJEMPLO

Ejemplo

Según el siguiente código:

```
int a=10, b=3, c=1, d, e;
```

```
Float x,y;
```

```
x=a/b;
```

```
c=a<b && c;
```

```
d= a + b++;
```

```
e= ++a - b;
```

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

// habría un error en la sentencia c=a<b && c; porque estamos evaluando una condición booleana, mientras que la variable c es una variable de tipo entero.



ponte a prueba

Las conversiones implícitas son aquellas que se producen cuando el valor que se va a almacenar se puede almacenar en la variable sin pérdida de información.

- a) Verdadero.
- b) Falso.

¿Qué convierte el método ToString?

- a) Un valor especificado en un entero de 64 bits.
- b) Un valor especificado en un carácter.
- c) Un valor especificado en un valor booleano.
- d) El valor especificado en su representación de cadena.

2.6. COMENTARIOS AL CÓDIGO

Hasta ahora nos hemos introducido en la creación por parte de un programador de código fuente con una finalidad concreta. No debemos pasar por alto que, además de realizar programas, lo debemos hacer de la forma más optimizada y ordenada posible.

En el ámbito de la organización entra en escena el concepto de los **comentarios**. Es una herramienta disponible en el compilador para que el programador pueda hacer anotaciones en el código sin que sea procesado por el compilador. Estas anotaciones deben de esclarecer el propio código y ayudar a entender las sentencias del programa. Así, futuros programadores podrán conocer su funcionamiento.

En el lenguaje de programación C# está permitido hacer **dos tipos de comentarios**:

- **Comentarios de una línea:** son frases cortas y, por tanto, solo pueden ocupar una línea del código. Debemos de escribir “//” antes de comenzar con dichas anotaciones.
- **Comentarios multiline o multilíneas:** son comentarios de varias líneas, que se utilizan para hacer una explicación mucho más detallada del código en cuestión. También podemos hacer uso de este tipo de comentarios cuando deseamos que una parte del código no sea procesada por el compilador. En este caso, debemos de englobar el texto entre los caracteres “/*” y “*/”.

El uso de los comentarios es muy útil y, por tanto, se aconseja a los programadores que implementen programas con tantos comentarios como sea posible para documentar el código.

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

// Está compuesto por la clase Hello1 y el método Main()

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

ponete a prueba

Según el siguiente código, ¿qué saldrá por pantalla?

```
static void Main(string[] args)
{
    /* Console.WriteLine ("HOLA MUNDO");
    Int i=10;
    Console.WriteLine (i);
    */
}
```

- a) HOLA MUNDO
- b) HOLA MUNDO10
- c) HOLA MUNDO
10
- d) Nada, porque el código está comentado

**Para hacer comentarios de una sola línea,
debemos utilizar el operado /.**

- a) Verdadero
- b) Falso

```
        onFail: failCount++  
    );  
},  
changeGroupState: function(state, btn) {  
    var doChangeState = function() {  
        if (cur.changingGroupState) return;  
        function participate(yes) {  
            if (yes) {  
                val('members_count', cur.count, btn);  
                replaceClass(btn, 'colorize');  
                val(btn, cur.unsubscribe, yes);  
                setStyle('aria-row');  
            } else {  
                val('members_group', cur.group, btn);  
                replaceClass(btn, 'colorize');  
                val(btn, cur.unsubscribe, yes);  
                setStyle('aria-row');  
            }  
        }  
        cur.changingGroupState = true;  
        if (yes) {  
            val('members_group', cur.group, btn);  
            replaceClass(btn, 'colorize');  
            val(btn, cur.unsubscribe, yes);  
            setStyle('aria-row');  
        } else {  
            val('members_count', cur.count, btn);  
            replaceClass(btn, 'colorize');  
            val(btn, cur.unsubscribe, yes);  
            setStyle('aria-row');  
        }  
    };  
    doChangeState();  
},  

```



3

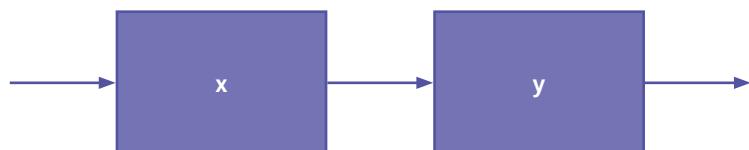
PROGRAMACIÓN ESTRUCTURADA

La programación estructurada es un procedimiento que consiste en diseñar e implementar programas de forma clara y sencilla para que puedan ser mantenidos con facilidad.

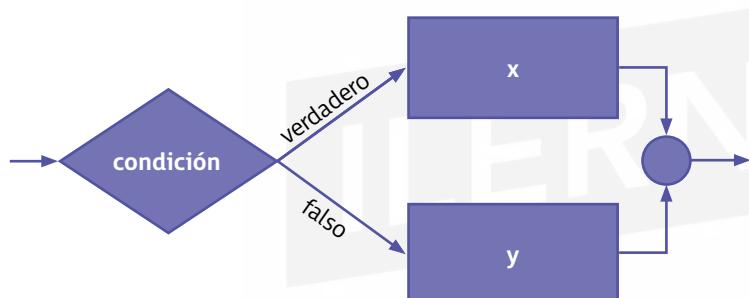
La idea de la programación estructurada fue expuesta por **E.W. Dijkstra** en 1965 y su base teórica se apoyó en los trabajos de **Böhm y Jacopini**.

El estudio concluyó que, con la combinación de tres tipos de instrucciones, podemos crear un programa estructurado:

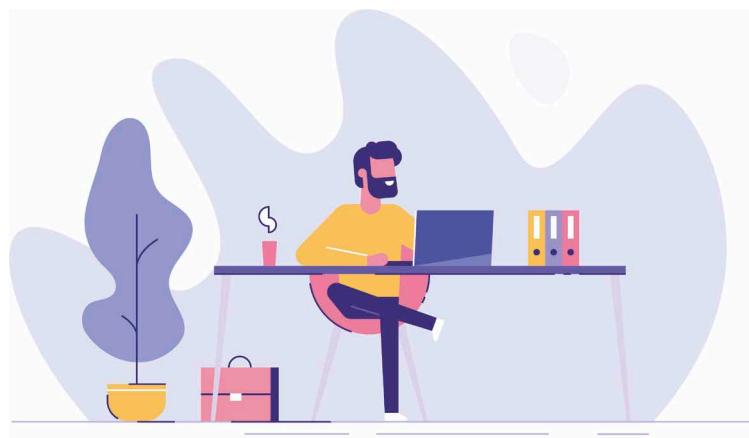
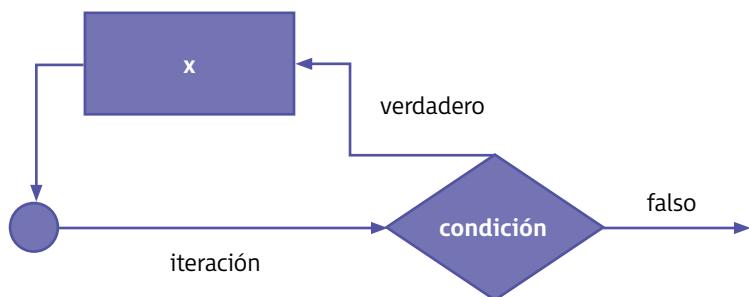
Secuencia de instrucciones



Selección de instrucciones



Iteración o bucle de instrucciones



BUSCA EN LA WEB

Si quieras conocer más sobre Edsger W. Dijkstra, visita esta web sobre el científico:

<https://dialnet.unirioja.es/descarga/articulo/3402333.pdf>

BUSCA EN LA WEB

Para saber más sobre los trabajos de Böhm y Jacopini, puedes visitar esta web:

<https://dl.acm.org/doi/pdf/10.1145/355592.365646>

3.1. FUNDAMENTOS DE PROGRAMACIÓN

Los fundamentos de la programación se basan en un conjunto de técnicas que persiguen desarrollar algoritmos que sean sencillos de escribir, leer, modificar y testear.

El orden en que se ejecutan las sentencias de un programa es **secuencial**. Este flujo significa que las sentencias se ejecutan en secuencia, una después de otra, y en el orden en que se encuentran dentro del programa.

Teorema de Böhm y Jacopini

Un programa debe de cumplir las siguientes características:

- Debe tener un único punto de entrada y uno de salida.
- Toda acción del algoritmo debe ser accesible. Tiene que haber al menos un camino que va desde el inicio hasta el fin del algoritmo.
- No deben existir, por tanto, bucles infinitos.

El programa se encarga de **transformar la entrada en salida**:



Por lo tanto, la programación es un proceso de desarrollo que nos ayuda a resolver un determinado problema.



ponte a prueba

Dijkstra concluyó que la combinación de varios tipos de instrucciones, podemos crear un programa estructurado. ¿Cuáles son?

- a) Secuencia de instrucciones.
- b) Selección de instrucciones.
- c) Iteración de instrucciones.
- d) Todas las opciones son correctas.

Según Böhm y Jacopini, ¿cuál de las siguientes opciones sobre un programa es cierta?

- a) Tiene un único punto de entrada y uno de salida.
- b) Toda acción del algoritmo debe ser accesible.
- c) No deben existir bucles infinitos.
- d) Todas las respuestas son correctas.

3.2. INTRODUCCIÓN A LA ALGORITMIA Y HERRAMIENTAS DE DISEÑO

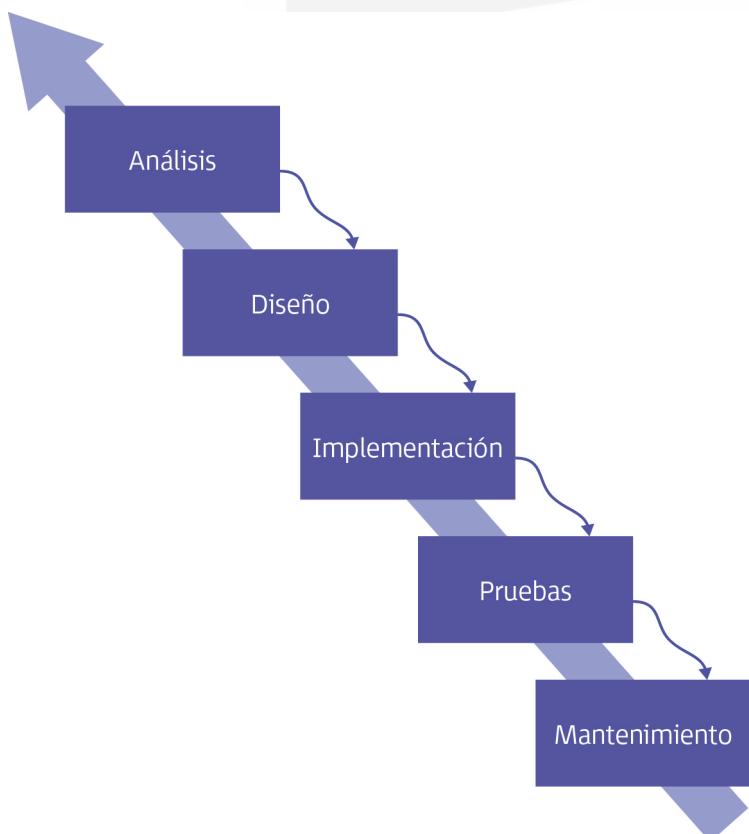
Un **algoritmo** es una secuencia de los pasos y las distintas operaciones que debe realizar el programa para conseguir resolver un problema planteado.

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

La algoritmia utiliza un conjunto de herramientas lógicas, matemáticas e informáticas con el objetivo de diseñar y presentar un algoritmo. Anteriormente, ya hemos hablado de dos herramientas para la construcción de algoritmos: el pseudocódigo y los diagramas de flujo. Este conjunto de herramientas es utilizado en las etapas de Ingeniería del Software que veremos a continuación.

3.3. CICLO DE VIDA DE UN PROYECTO SOFTWARE

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. Se trata de la etapa más importante del ciclo de vida.

Existen varias técnicas para obtener los requisitos del usuario:

- Entrevistas
- *Brainstroming*
- Prototipos
- Casos de uso
- JAD (entrevistas en grupos o talleres)

- **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.

- **Diseño estructurado:** tenemos varias notaciones como los diagramas de flujo o el pseudocódigo.
- **Diseño orientado a objetos:** trabajaremos con representaciones en UML (diagramas de clases, diagramas de secuencia, etc.).

- **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 con el que estemos trabajando. El resultado que vamos a obtener va a ser el código fuente.

- **Pruebas de integración:** hay que realizar ensayos del funcionamiento, combinando todos los módulos de la aplicación. Así, haciendo 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.



ponte a prueba

En la etapa de diseño, tomamos los requisitos de los clientes.

a) Verdadero
b) Falso

Después de la fase de pruebas, ¿qué etapa se lleva a cabo?

a) Análisis
b) Implementación
c) Mantenimiento
d) Ninguna de las opciones es correcta

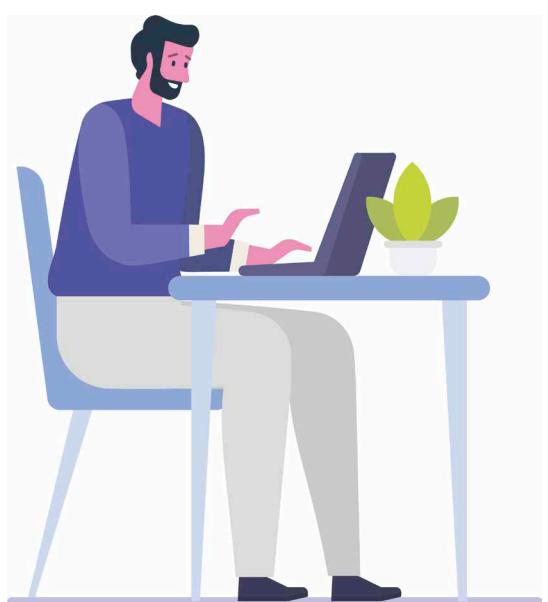
3.4. PRUEBA DE PROGRAMAS

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.

En ella, se prueba un programa para demostrar la existencia de **defectos**: algorítmicos, de sintaxis, de documentación, de sobrecarga, de capacidad, de rendimiento, de sincronización, de recuperación, de *hardware* y *software* y de estándares.

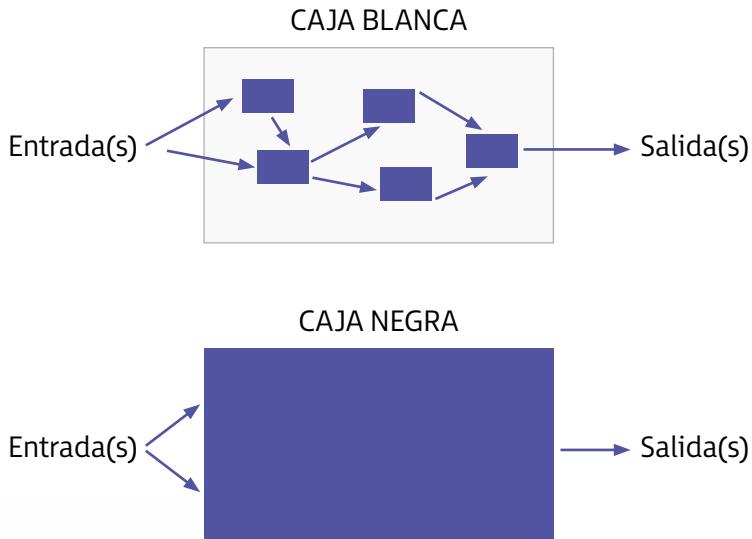
Este proceso de prueba del programa se lleva a cabo de manera automática o manual y persigue los siguientes **objetivos**:

- Comprobar los requisitos funcionales y no funcionales del programa.
- Probar todo tipo de casos para detectar alguna anomalía en su ejecución.



Tenemos dos tipos de técnicas de pruebas:

- **Pruebas de caja blanca:** donde se valida la estructura del sistema.
- **Pruebas de caja negra:** donde se analizan las entradas y salidas del sistema sin tener en cuenta el funcionamiento interno.



El **plan de prueba** consta de varias etapas, ya que, después de implementar el código, si existiera algún tipo de fallo en el programa, tendríamos que volver a empezar con el nuevo código modificado.

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:

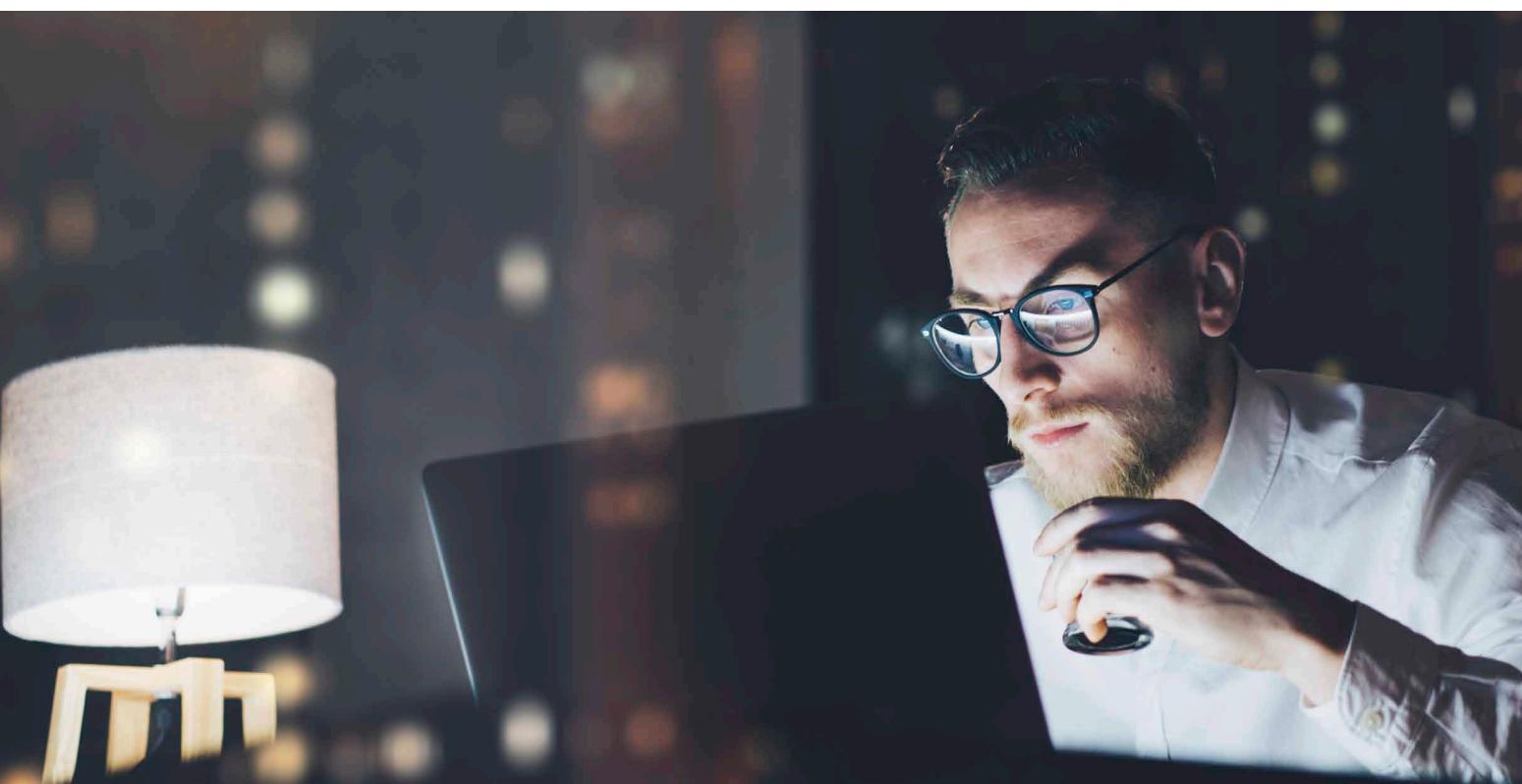
- **Natural**: números naturales (N): *byte*, *uint*, *ushort*, *ulong*.
- **Entero**: números enteros (Z): *sbyte*, *int*, *long*, *short*.
- **Real**: números reales (R): *decimal*, *float*, *double*.
- **Carácter**: caracteres (C): *char*.
- **Lógico**: booleanos (B): *bool*.

- **Tipos simples definidos por el usuario**

- **Tipos enumerados**: es un tipo de valor definido por el usuario, así como un conjunto de constantes. Para definir un tipo de enumeración, utilizamos la palabra clave *enum* y especificamos los nombres de componentes de la enumeración:

```
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

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

- **Vectores:** estructura donde almacenamos varias variables del mismo tipo. Se puede declarar una matriz si se especifica el tipo de sus elementos. También son llamados **array unidimensional**, es decir, de una dimensión. Con esto, nos damos cuenta de la cantidad de indicadores de posición que necesitamos para acceder a un elemento de la tabla.

Sintaxis:

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

Por ejemplo:

```
int [] v = new int [10];
int[] array1 = new int[] { 1, 3, 5, 7, 9, 11, 13, 15, 17, 19 };
int[] array2 = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };
```

Con el anterior código, estamos declarando 10 números enteros en un vector al que hemos llamado v.

A continuación, veremos la manera de acceder a cada uno de ellos. Cabe recordar 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.

...

v[9] → Último número entero del vector

Veamos el siguiente ejemplo:

```
char[] miArray = { 'a', 'b', 'c', 'd', 'e' };
for (int x=0; x<5; x++) {
miArray [1]='z';
}
Console.WriteLine(miArray[1]);
// Guardamos en la posición 1, el carácter z. Por lo que, por pantalla, mostrará ese carácter en esa posición. Ahora el vector estaría formado por { 'a', 'z', 'c', 'd', 'e' }
```

- **Matrices:** unión de varios vectores de cualquier tipo simple (enteros, reales, etcétera). También la podemos

ver como un *array* bidimensional, por tanto, esos datos nos indican que necesitamos dos indicadores de posición para acceder al elemento. La primera posición de una matriz es (0,0).

Sintaxis:

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

Por ejemplo:

```
int [,] matriz = new int[2,3];
int[,] array2D = new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
int[,] miMatriz = new int[4, 2] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
```

En este caso:

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

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.

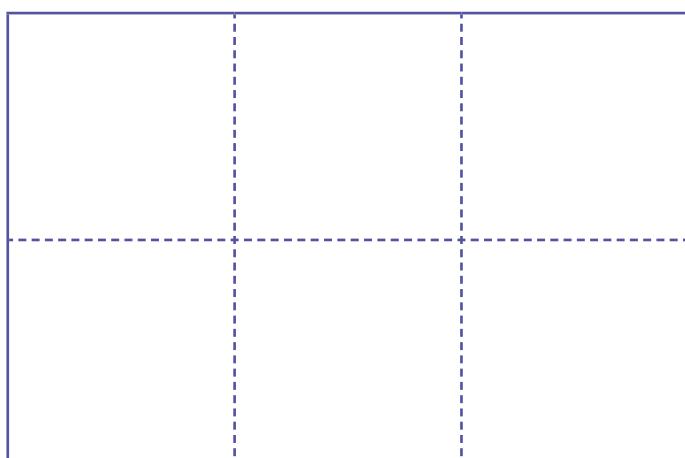
`matriz[0,2]` → Elemento correspondiente a la primera fila, tercera columna.

`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.

Matriz de 6 elementos (2 filas por 3 columnas):





ponte a prueba

¿Qué tipo de datos es el tipo enumerado?

- a) Compuesto
- b) Simple

¿Cómo puedo acceder al dato 2 si tenemos el siguiente array?

`int[] a = new int[] {1, 2, 3};`

- a) a[1]
- b) a[0]
- c) a[2]
- d) Ninguna de las opciones es correcta.

¿Cuántas columnas contendrá la siguiente matriz?

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

- a) 2

- b) 3

- c) 6

- d) Ninguna de las respuestas es correcta.

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.

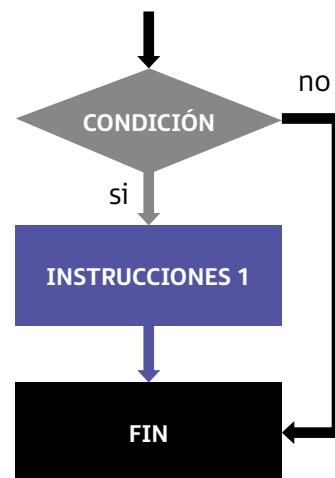
Instrucción simple (IF): su estructura sería de la siguiente forma:

Instrucción simple (IF)

SI CONDICIÓN ENTONCES:

 INSTRUCCIONES_1

 FIN SI;



Instrucción doble (IF-ELSE)

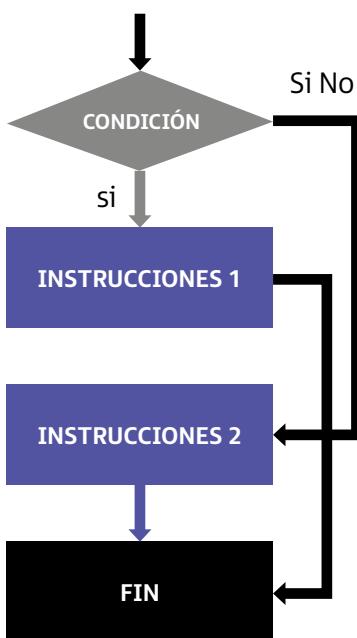
SI CONDICIÓN ENTONCES:

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.

Sin embargo, si CONDICIÓN ES FALSA, ejecutaremos lo que hay en INSTRUCCIONES_2.

CONCEPTO

En las estructuras dobles, por cada IF, debe haber un ELSE.

Operadores lógicos

&&	AND: ambas condiciones deben cumplirse para entrar en el bloque IF. No evalúa el operando derecho si el izquierdo es false.
	OR: es suficiente con que una de las condiciones se cumpla para entrar en el bloque IF. El resultado de $x \parallel y$ es true si x o y es true.
^	XOR: El resultado se evalúa como true si una condición se evalúa como false y otra como true. Si ambas condiciones se evalúan como false o como true, el resultado será false.



Ejemplo AND lógico condicional &&:

SI CONDICIÓN_1 && CONDICIÓN_2 ENTONCES:

FIN SI;

Si condición_1 y condición_2 son true, el resultado se evalúa como true. Si alguna condición es false, el resultado se evalúa como false.

Ejemplo OR lógico condicional ||:

SI CONDICIÓN_1 || CONDICIÓN_2 ENTONCES:

FIN SI;

Si condición_1 es true o condición_2 es true, el resultado se evalúa como true.

Ejemplo XOR exclusivo:

SI CONDICIÓN_1 ^ CONDICIÓN_2 ENTONCES:

FIN SI:

CONDICIÓN_1	CONDICIÓN_2	RESULTADO
false	false	False
false	true	True
True	false	true
true	true	false

Sentencia Switch (o de selección múltiple)

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.

Añadiremos una sentencia de salto (**break**) al final de cada bloque “case” para abandonar dicho bloque.

```

using System;

public class Example
{
    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;
        }
    }
}

// La salida mostrada será Caso 1 porque nuestra variable 'var'
// contiene el valor 1.

```

La cláusula default es optativa en esta estructura.



ponte a prueba

Según el siguiente código, ¿cuándo imprimimos la cadena “Hola a todos”?

```
If (a && b) {Console.WriteLine("Hola a todos");}
```

- a) Cuando la condición a se cumpla.
- b) Cuando la condición b se cumpla.
- c) Cuando se cumpla la condición a y b.
- d) Cuando se cumpla a, pero no b.

Según el siguiente código, ¿qué sucede?

```

int a=1;
if (a>0) {Console.WriteLine("Estamos en el if");}
else{
    Console.WriteLine("Estamos en el else");
}

```

- a) Escribimos por pantalla “Estamos en el if”.
- b) Escribimos por pantalla “Estamos en el else”.
- c) No se ejecuta ni el if ni el else.
- d) La variable a no está correctamente declarada.

3.7. ESTRUCTURAS DE REPETICIÓN

Las estructuras de repetición siempre se ejecutan mediante el uso de **bucles**. Estos son los que permiten que una o varias líneas de código se ejecuten de forma repetida. Se van a ir repitiendo mientras que se cumpla una determinada condición de salida.

Existen diferentes **estructuras de repetición**:

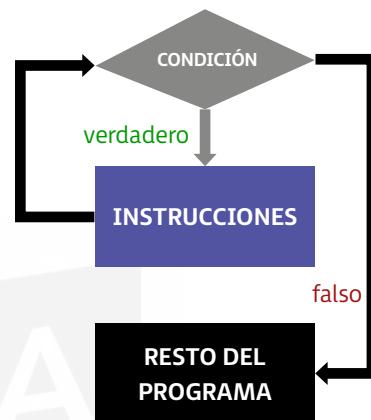
Mientras (While)

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;



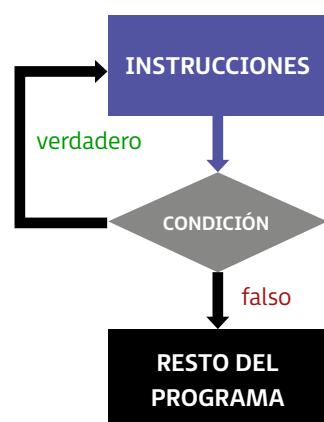
Hacer... mientras (Do... while)

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;  
...;  
Instrucción_N;  
ModificarCondición;
```

Mientras CONDICIÓN;

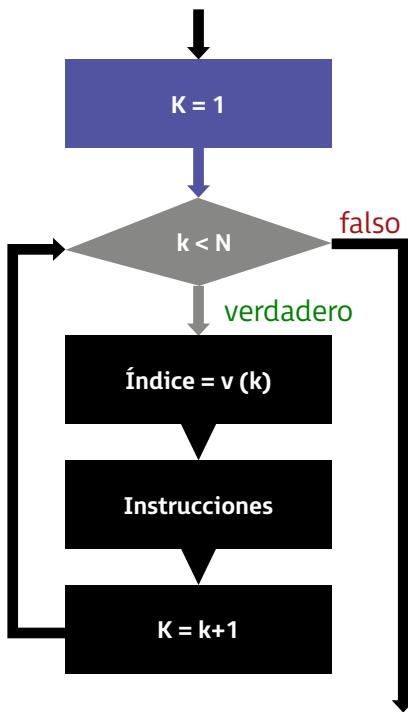


Para (For)

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

```
Instrucción1;  
Instrucción2;  
...  
InstrucciónN;  
ModificarCondición;
```



Foreach

Ejecuta una instrucción o bloque de instrucciones para cada elemento de ciertas estructuras de datos como arrays o listas.

EJEMPLO 1

```
int [] diasMes = {1, 21, 30};
foreach (int dias in diasMes) {
    Console.WriteLine("Dias del mes: {0}", dias);
    // creamos una variable días de tipo entero que nos servirá para recorrer cada elemento del
    // array diasMes
}
```

EJEMPLO 2

```
List<string> ListaColores = new List<string>(); //creamos una lista de tipo string
//añadimos un color a la lista
ListaColores.Add ("Azul");
ListaColores.Add ("Rojo");
ListaColores.Add ("Verde");
ListaColores.Add ("Amarillo");
ListaColores.Add ("Morado");

foreach (string color in ListaColores)
{
    // recorremos la lista mediante la variable string color
    Console.WriteLine ( color );
}
```



ponte a prueba

Según el siguiente código, ¿ejecutaríamos el código de la salida por pantalla?

```
int a=0; int b=0;
while (a>0 || b==0) {Console.WriteLine("Hola Ilerna"); b = b
- 1;}
```

- a) Sí, con que se cumpla una condición es suficiente.
- b) No, porque se deben cumplir ambas condiciones.
- c) No, porque no se cumple la condición a>0.
- d) Ninguna de las respuestas es correcta.

¿Cuántas veces ejecutaríamos un bucle do-while?

- a) Depende de la condición.
- b) Mínimo, una vez.
- c) Cero o más veces.
- d) Mínimo, dos veces.

¿Es correcto el siguiente código?

```
int [] array = {1,2,3};
for (i = 0; i < array.Length; i++) { Console.WriteLine("HOLA
MUNDO!!!!"); }
```

- a) Sí. Entra en el bucle for y muestra los datos 1, 2 y 3.
- b) No. La variable i no está declarada.
- c) Sí. Entra en el bucle y muestra por pantalla HOLA MUNDO!!!
- d) Ninguna de las respuestas es correcta.

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	<p>Se encuentra al final de la sentencia <i>switch</i>. Interrumpe el bucle indicando que debe continuar a partir de la siguiente sentencia después del bloque del ciclo.</p> <p>El control se devuelve a la instrucción que haya a continuación de la instrucción finalizada.</p>
continue	<p>Se utiliza en sentencias repetitivas con bucles <i>for</i>, <i>foreach</i>, <i>while</i> y <i>do...while</i>. Transfiere el control a la siguiente iteración.</p> <pre>for (int i = 1; i <= 10; i++) { if (i < 9) { continue; } Console.WriteLine(i); } Console.ReadKey(); // Mostraría por pantalla 9 y 10</pre>
return	<p>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.</p> <pre>static double CalculaArea(int radio) { double area = radio * radio * Math.PI; return area; } //Devuelve la variable local área como un valor de tipo entero</pre>



ponte a prueba

Según el siguiente código, ¿qué saldrá por pantalla?

```
int caseSwitch = 1;
switch (caseSwitch)
{
    case 1:
        Console.WriteLine("Case 1");
        break;
    case 2:
        Console.WriteLine("Case 2");
        break;
    default:
        Console.WriteLine("Default case");
        break;
}
```

- a) Case 1
- b) Case 2
- c) Default case
- d) caseSwitch

3.9. TRATAMIENTO DE CADENAS

Una cadena es un objeto de tipo *string*. Para la representación de cadenas de caracteres se utiliza la palabra reservada ***string***.

En C#, la **representación** del tipo de datos sería de la siguiente forma:

```
string frase = "buenos días";
Console.WriteLine("El carácter de la posición 5, es {0}", frase [5]);
// La salida por pantalla sería el carácter 's' que está en la posición 5
```

Siempre que queramos escribir combinaciones alfanuméricas, en nuestro código, 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.

```

char []letras = {'h', 'o', 'l', 'a'};
string frase = new string(letras);
char caracter = frase [0];
/* Creamos un array de caracteres llamado letras.
A continuación, vamos a crear un objeto de tipo string con la palabra reservada new.
Con esta palabra reservada, creamos una nueva instancia de tipo string (frase).
Por último, asignamos el primer valor de nuestro string a la variable carácter.
En este caso, el valor será 'h' */

```

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 <i>string</i> en <i>array</i> de caracteres.
SubString()	Extrae parte de una cadena. Método que puede ser sobrecargado indicando su inicio y su fin: <i>SubString (int inicio)</i> o <i>SubString (int inicio, int tamaño)</i> .
CopyTo()	Copia un número de caracteres especificados a una determinada posición del <i>string</i> .
CompareTo()	Compara la cadena que contiene el <i>string</i> 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 <i>string</i> .
Contains()	Si la cadena que se le pasa por parámetro forma parte del <i>string</i> , devuelve un booleano.
IndexOf()	Si aparece un carácter especificado en el <i>string</i> , devuelve el índice de la posición de la primera vez que aparece. Devuelve -1 si el carácter no se encuentra en la cadena.
Insert()	Inserta una cadena de caracteres en una posición concreta del <i>string</i> .
Trim()	Quita todos los espacios en blanco del principio y el final de la cadena de caracteres.

Replace()	Devuelve una cadena, donde se ha sustituido un carácter por otro.
Remove()	Devuelve una cadena donde se ha eliminado un número de caracteres especificado.
Split()	Ofrece la posibilidad de separar en varias partes una cadena de caracteres.
ToLower()	Devuelve una cadena convertida en minúsculas.
ToUpper()	Devuelve una cadena convertida en mayúsculas.

En el siguiente link, se puede consultar la información de los métodos *string*:

BUSCA EN LA WEB

<https://docs.microsoft.com/es-es/dotnet/api/system.string?view=netcore-3.1#methods>






ponte a prueba

¿Qué mostrará por pantalla el siguiente código?

```
int[] array = { 1, 2, 3 };
Console.WriteLine(array.Length);
```

a) 3
 b) 1, 2, 3
 c) 1
 d) 2

La función Trim() elimina cualquier carácter del principio y el final de la cadena.

a) Verdadero
 b) Falso



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:

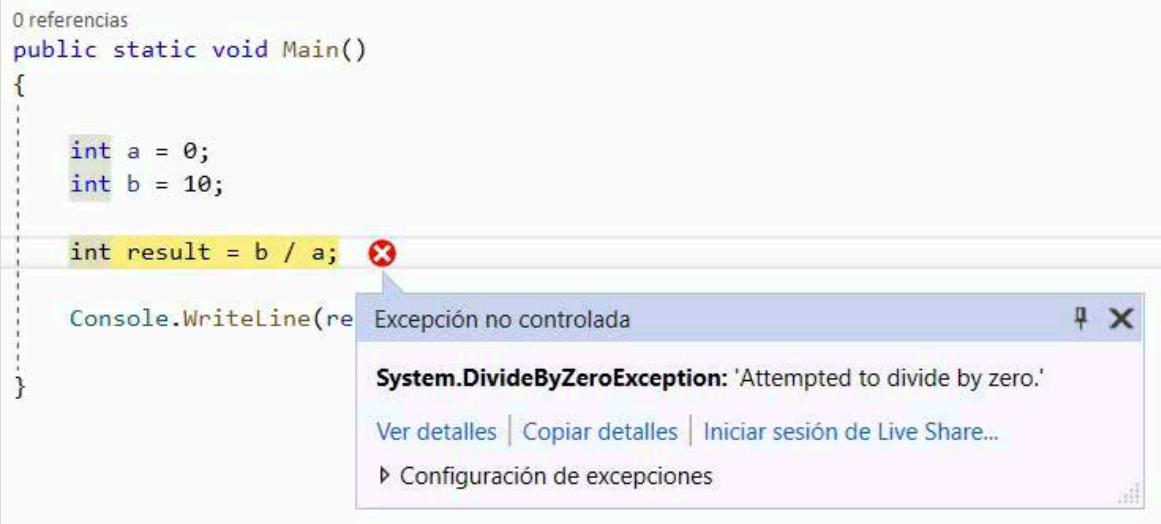
- De **compilación o sintaxis**: errores en el código.
- De **tiempo de ejecución**: los que producen un fallo a la hora de ejecutar el programa. Se trata de fragmentos de código que parecen estar correctos y que no tienen ningún error de sintaxis, pero que no se ejecutarán. Por ejemplo, se podría escribir correctamente una línea de código para abrir un archivo, pero si el archivo no existe, la aplicación no puede llevar a cabo la apertura de ese archivo, por lo que se detendrá la ejecución.
- **Lógicos**: son aquellos errores que dan resultados erróneos, diferentes a los esperados o no deseados, 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 su seguimiento, o podemos comprobar cuándo finalizan los bucles, entre otras cosas. Para depurar el código, teclearemos *F10* (paso a paso).

- **Errores de compilación**: estos errores impiden la ejecución de un programa. Inicialmente se compila el programa, y, si el compilador 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.



```

0 referencias
public static void Main()
{
    int a = 0;
    int b = 10;

    int result = b / a; ✖

    Console.WriteLine(re)
}
  
```

Excepción no controlada

System.DivideByZeroException: 'Attempted to divide by zero.'

[Ver detalles](#) | [Copiar detalles](#) | [Iniciar sesión de Live Share...](#)

[Configuración de excepciones](#)

- **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 lo que su corrección es más difícil. Tenemos que encontrar el error en un programa que sí funciona, pero que funciona de forma diferente a la que debería.



ponte a prueba

¿Qué errores podemos encontrar en la etapa de la depuración de un programa?

- Errores lógicos
- Errores de compilación
- Errores en tiempo de ejecución
- Todas las respuestas son correctas

El programador está realizando un programa en el cual quiere introducir dos datos enteros. Uno, que sea la base y otro que sea el exponente para realizar la operación potencia. Introduce como base el número 5 y como exponente 2. El resultado es 10. ¿Qué tipo de error le está dando a nuestro programador?

- De compilación
- Lógico
- De ejecución
- No hay errores. El programa es correcto

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.

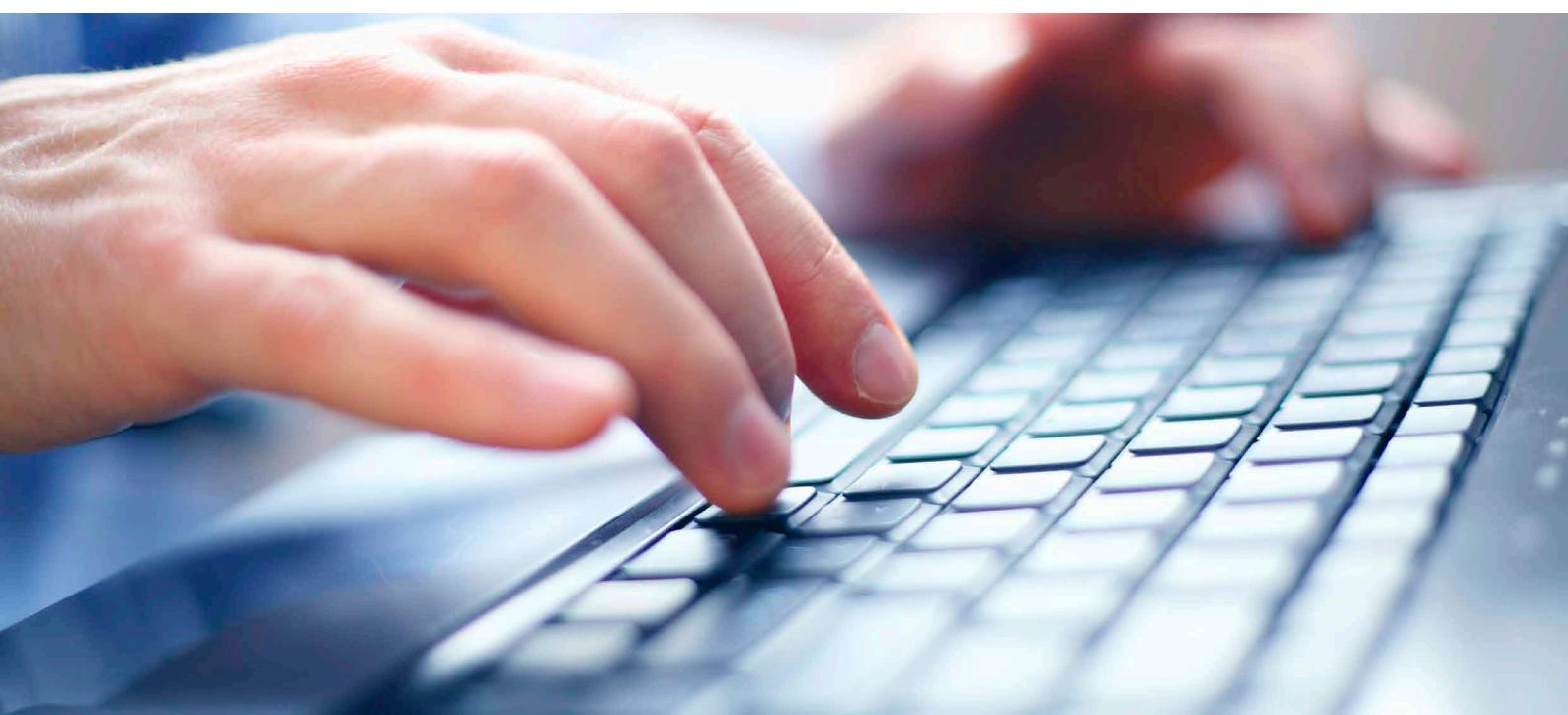
Los ficheros que entreguemos que formen parte del proyecto deben estar testeados y tener un correcto funcionamiento. Habrá que solventar los errores descritos anteriormente.

summary

La etiqueta `<summary>` debe usarse para describir un tipo y para agregar información adicional dentro de nuestro código. Gracias a herramientas de documentación como GhostDoc y Sandcastle, creamos hipervínculos internos a las páginas de documentación de los elementos de código:



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



```
namespace ConsoleApp33
{
    // Implementamos los métodos a utilizar
    class Program
    {
        /// <summary>
        ///
        /// 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();
        }
    }
}
```

```
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>ConsoleApp33</name>
    </assembly>
    <members>
        <member name="M:ConsoleApp33.Program.Main(System.String[])">
            <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.

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.
- Actúan como soporte para diferentes lenguajes de programación.
- Reconocen sintaxis.
- Están integrados con sistemas de control de diferentes versiones.
- Tienen un depurador.
- 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.
- Muestran en pantalla errores y *warnings*.



ponte a prueba

¿Qué ventajas nos ofrecen los IDE?

- a) Una baja curva de aprendizaje.
- b) Crear un proyecto desde cero.
- c) Un intelligence que nos marca qué funciones podemos utilizar.
- d) Todas las opciones son correctas.



4

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.

4.1. 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**.



4.2. 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.
- Escritura y *testing*.
- Reutilización de módulos.
- Independencia de fallos.

- **Inconvenientes:**

- ¿Cuánto hay que dividir nuestro problema?
- Aumenta el uso de memoria y el tiempo de ejecución.



ponte a prueba

¿Qué ventajas nos proporciona la programación modular?

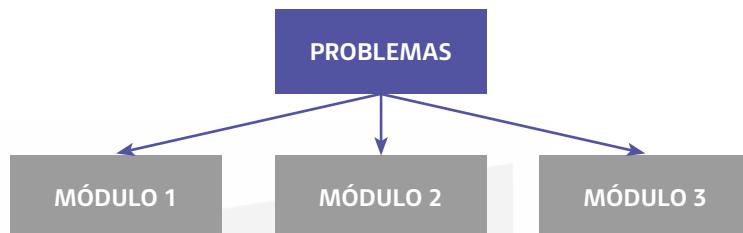
- a) Facilita el mantenimiento.
- b) Facilita el testing.
- c) Podemos reutilizar los módulos.
- d) Todas las opciones son correctas.

4.3. 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)**. Esta solución se divide en módulos que se estructuran e integran jerárquicamente.

Este diseño se basa en el principio **divide y vencerás**:

- Va de lo más general a lo más específico.
- Se basa en una representación por niveles: el primer nivel resuelve el problema, y los sucesivos van depurando el primero.
- El programa, por tanto, tendrá una estructura de árbol.



ponte a prueba

¿En qué consiste el diseño top down?

- a) Empezar desde los módulos más pequeños hasta el módulo mayor.
- b) Dividir un gran problema en subproblemas.
- c) Inventar el código.
- d) Ninguna de las opciones es la correcta.



4.4. MODULACIÓN DE PROGRAMAS.

SUBPROGRAMAS

Podemos denominar los **módulos** como subprogramas, y estos como las diferentes partes del problema que puede resolverse de forma independiente. Los módulos, al ser independientes unos de otros, permiten que podamos centrarnos en una de sus partes (sin tener en cuenta el resto). También ofrecen la posibilidad de que se puedan utilizar las soluciones obtenidas en otras partes del programa.

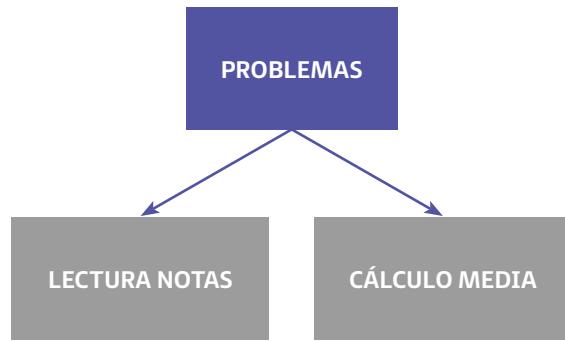
Cada módulo codificado dentro del programa como subprograma se puede definir como una parte del código independiente, que realiza una tarea asignada.

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

```
//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 [ i ];
}
//Finalmente, dividimos el sumatorio por el número 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.

En esta parte, el usuario va introduciendo las notas de los alumnos. Mostraremos por pantalla la nota media.

```

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.

```

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

Podemos comprobar que en C# los programas se dividen en subprogramas, denominados **funciones o métodos**. 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.

4.5. LLAMADAS A FUNCIONES (MÉTODOS). TIPOS Y FUNCIONAMIENTO

Funciones en C#

Los métodos con los que trabajamos en C# se denominan *funciones*.

De hecho, cuando nos referimos a una función, un método o un subprograma en C#, nos estamos refiriendo al mismo concepto.

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

Sintaxis:

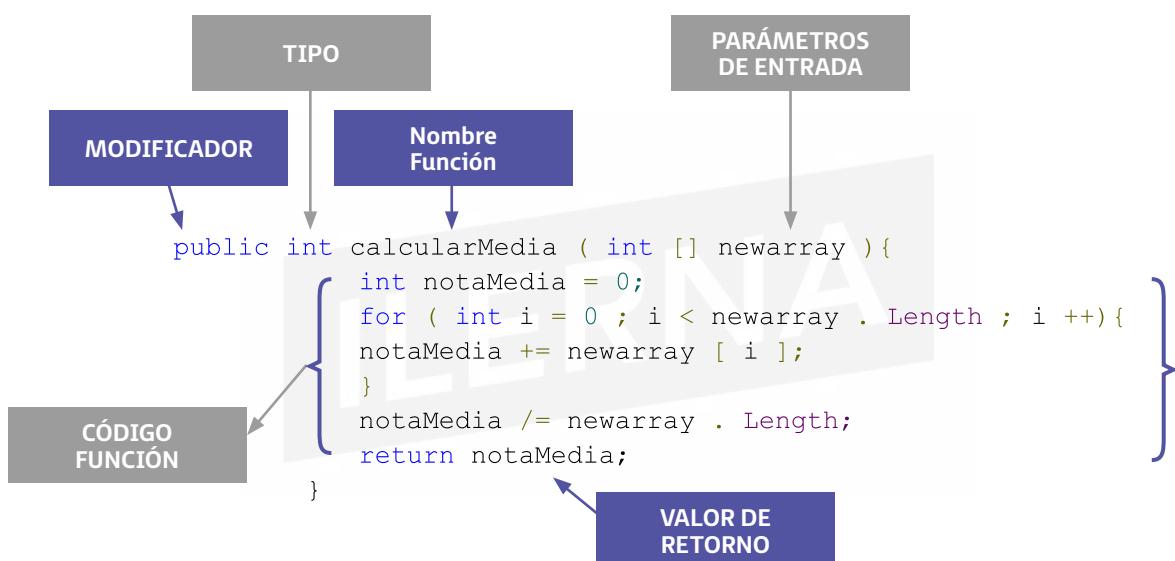
```
Modificadores Tipo NombreFuncion (Parámetros de entrada ) {
    Código de la función
    return expresión;
}
```

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*:
 - **Public:** el acceso no está restringido.
 - **Private:** el acceso está limitado a la misma clase.
 - **Protected:** el acceso está limitado a la clase contenedora o a los tipos derivados de la clase contenedora.
 - **Internal:** el acceso no está restringido, siempre y cuando sea dentro del mismo ensamblado. Un ensamblado es un conjunto de tipos que se compilan para obtener una misma solución. Si no indicamos un modificador, por defecto, nuestros métodos tendrán un nivel de accesibilidad *internal*.

- **Tipo:** las funciones, al igual que ocurría con las variables, poseen un tipo, es decir, el dato que devuelven es de un tipo determinado.
- **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.
- **Return:** si nuestro método está tipado, es obligatorio que devuelvan un valor 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.

Métodos no tipados. Void

Los métodos no tipados son aquellos que realizan una tarea específica y que pueden recibir parámetros, pero no necesitan devolver un valor.

Para referenciar que ese método no nos devuelve un valor, utilizamos la sentencia **void**.

```

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

CÓDIGO EJEMPLO:

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

EJERCICIO 1

Realice un programa modular donde se calcule la suma de los dígitos de un número que el usuario introduzca por pantalla.

Por ejemplo, si el usuario introduce el número 369, el resultado sería: $3+6+9=18$

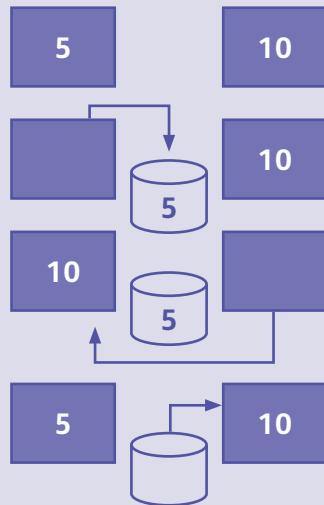
Solución

```
public static int Calcular( int a )
{
    string cad = Convert.ToString(a);
    // Convertimos el parámetro a en string
    int sum = 0;
    for (int i = 0; i < cad.Length; i++)
        // recorremos el número y vamos tomando las subcadenas (dígitos) del mismo
        // con la función Substring almacenándolo en una variable entera para ir haciendo la
        // suma de cada uno de sus dígitos
        sum += Convert.ToInt32(cad.Substring(i, 1));
    return sum;
}

public static void Main()
{
    int numero;
    Console.Write("Introduzca el número: ");
    numero = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine(Calcular(numero));
}
```

 **EJERCICIO 2**

Realice un programa modular donde el usuario introduzca dos números enteros y se intercambien de posición.



Solución

```

public static void intercambio(ref int numero1, ref int numero2)
{
    int aux; // creamos una variable auxiliar para guardar temporalmente en número
    1

    aux = numero1;
    numero1 = numero2;
    numero2 = aux;
}

public static void Main()
{
    int numero1,numero2;

    Console.WriteLine("Introduzca el primer número");
    numero1 = Convert.ToInt32(Console.ReadLine());

    Console.WriteLine("Introduzca el Segundo número ");
    numero2 = Convert.ToInt32(Console.ReadLine());

    // pasamos los parámetros por referencia con la palabra reservada ref (veremos el
    // paso de parámetros en la siguiente sección)
    intercambio( ref numero1, ref numero2 );
    Console.WriteLine("{0}\n{1}",numero1,numero2);
}

```

Paso de parámetros. Paso por valor y paso por referencia

En C#, podemos distinguir **dos tipos de paso de parámetros** diferentes:

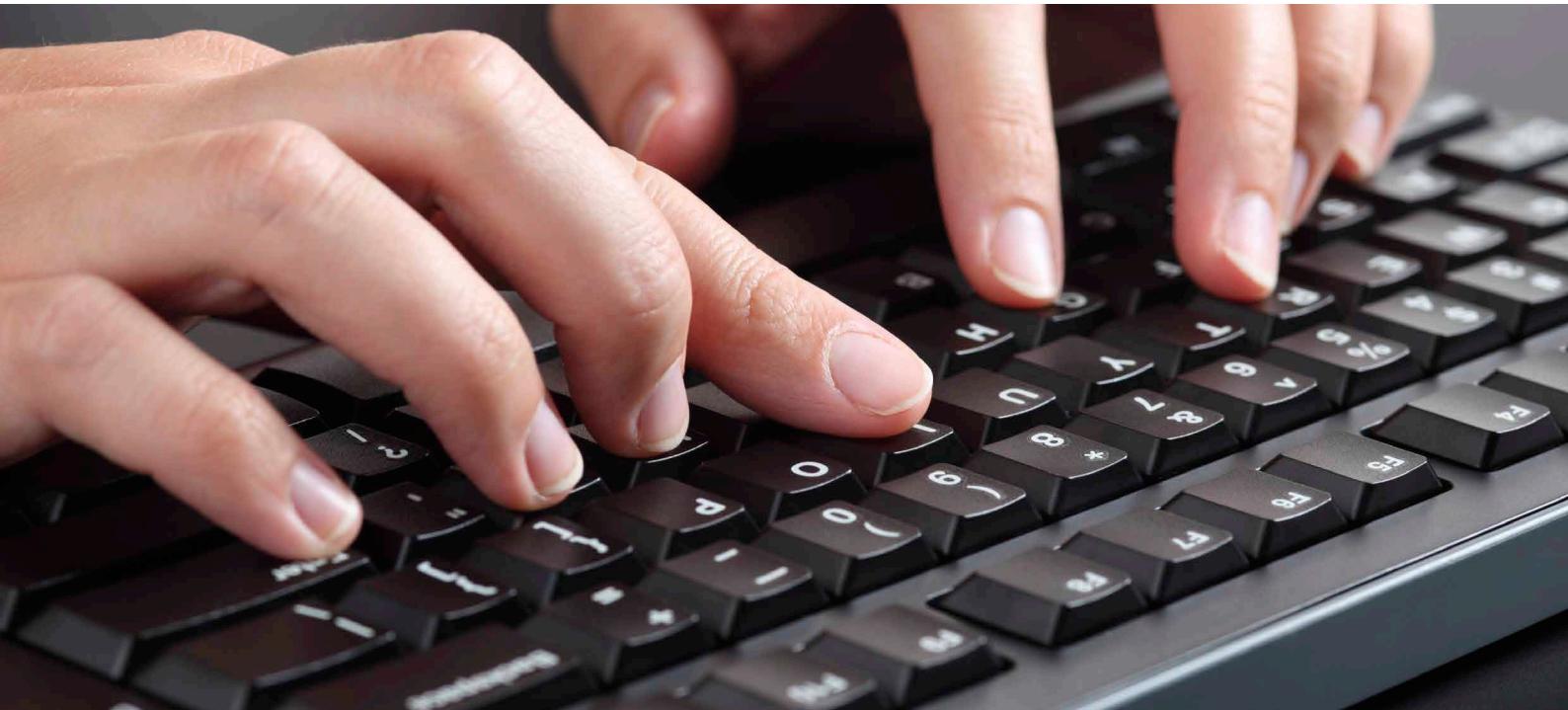
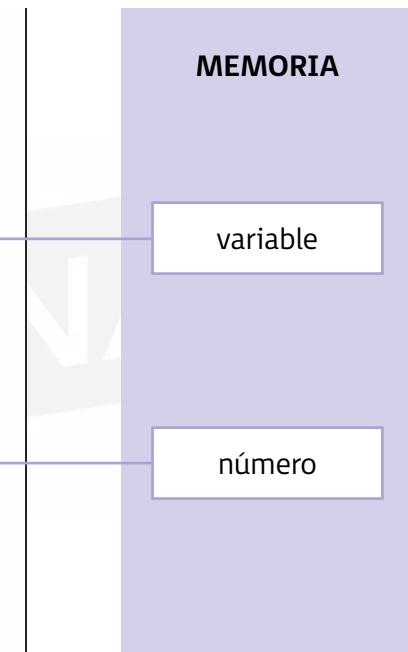
- Por valor o copia.
- Por referencia.

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.

```

1  using System;
2
3  namespace IlernaOnline
4  {
5      class Program
6      {
7          static void MiMetodo(int variable)
8          {
9              variable = variable + 5;
10         }
11
12         static void Main(string[] args)
13         {
14             int numero = 1;
15             MiMetodo(numero);
16             Console.WriteLine(numero);
17             Console.ReadKey();
18         }
19     }
20 }
```



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:  using System;
2:  using System.Collections.Generic;
3:  using System.Linq;
4:  using System.Text;
5:  using System.Threading.Tasks;
6:
7:  namespace ConsoleApp36
8:  {
9:      class Program
10:     {
11:         static void MiMetodo(ref int variable)
12:         {
13:             variable = variable + 5;
14:         }
15:
16:         static void Main(string[] args)
17:         {
18:
19:             int numero = 1;
20:             MiMetodo(ref numero);
21:             Console.WriteLine(numero);
22:             Console.ReadKey();
23:
24:         }
25:     }
26: }
```

MEMORIA

número

CÓDIGO EJEMPLO:

```

static void Main(string[] args)
{
    int a = 0;
    metodo(ref a);

/* cuando pasamos por valor la variable 'a' al método,
tanto la variable 'a' como la variable 'numero' ocupan
el mismo espacio de memoria, pero lo que el valor se so-
breescibirá.

En este caso, inicialmente, tenemos el valor 0, pero al
asignar a 'numero' el valor 1, éste sobreescribe al 0 */
    Console.WriteLine(a);
}

static void metodo(ref int numero) {
    numero = 1;
}
```

params, in y out

Cuando trabajamos con la palabra clave 'params' estamos especificando un paso por valor. En este caso, el tipo de parámetro debe ser una matriz unidimensional (array).

No se permiten pasar más parámetros después de la palabra clave 'params' en una declaración de método. Si el tipo declarado del parámetro 'params' no es un vector, se produciría un error (Error del compilador CS0225).

CÓDIGO EJEMPLO:

```
public static void miMetodo(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        Console.Write(list[i] + " ");
    }
    Console.WriteLine();
}
static void Main()
{
    int[] miArrayInt = { 5, 6, 7, 8, 9 };
    miMetodo(miArrayInt);
}
```

La palabra clave 'in' sirve para pasar parámetros por referencia. Las variables que se han pasado como argumentos con la palabra clave 'in' deben inicializarse antes de pasarse en una llamada de método o función. Si pasamos un parámetro con 'in', este no puede ser modificado dentro del método.

CÓDIGO EJEMPLO:

```
0 referencias
static void Main()
{
    int soloLectura = 44;
    ejemplo(soloLectura);
    Console.WriteLine(soloLectura); // el valor es 44
}

1 referencia
static void ejemplo(in int number)
{
    // error CS8331
    number = 19;
}
```

[] (parámetro) in int number
No se puede asignar a variable "in int" porque es una variable readonly.

La palabra clave ‘out’ también sirve para pasar parámetros por referencia. No es necesario iniciar la variable antes de pasarla al método.

CÓDIGO EJEMPLO:

```
static void Main()
{
    int entera;
    OutEjemplo(out entera);
    Console.WriteLine(entera);      // la salida por pantalla
será 40

}

static void OutEjemplo(out int numero)
{
    numero = 40;
}
```



ponte a prueba

Las funciones deben tener un tipo asociado.

- a) Verdadero
- b) Falso

¿Cuál es la salida del siguiente código?

```
static void Main(string[] args)
{
    int a = 10;
    int b = 5;
    Console.WriteLine(suma(a,b));
}

static int suma(int x,int y) { return x + y; }
```

- a) 15
- b) 10
- c) 5
- d) La función no está bien construida

Cuando realizamos un paso por valor de una variable, creamos dos posiciones de memoria distintas.

- a) Verdadero
- b) Falso

4.6. ÁMBITO DE LAS LLAMADAS A FUNCIONES

Hasta ahora, hemos trabajado tanto con variables dentro de nuestro Main, con variables creadas dentro de nuestros métodos y con el paso de parámetros. Veamos cuál es el ámbito o el alcance de estas variables.

Variables globales

Una variable global es creada fuera de nuestro Main y fuera de cualquier método. Podemos acceder a ella en cualquier momento de nuestro programa.

```
class Program
{
    static int n = 10; // Variable global
    public static void CambiaValorN()
    {
        n++;
    }
    public static void Main()
    {
        Console.WriteLine("n vale {0}", n); // el valor de n es 10
        CambiaValorN();
        Console.WriteLine("Ahora n vale {0}", n); // el valor de n es 11
    }
}
```

En términos generales, debemos intentar construir nuestros programas con variables locales. Así hacemos que cada parte del programa trabaje con sus propios datos, y ayudamos a evitar que un error en un trozo de programa pueda afectar al resto. Si modificamos una variable global en cualquier punto del programa, podemos perdernos en su flujo.

La forma correcta de pasar datos entre distintos métodos no es a través de variables globales, sino usando los parámetros de cada función y los valores devueltos (como hemos visto en ejemplos anteriores con el paso de parámetros por valor y por referencia).

Variables locales

Las variables que se declaran dentro de un método (o de una función), y que, por tanto, solo este fragmento de código las conocerá, son las llamadas *variables locales*. No se podrán usar desde ningún otro método del programa.

```

public static int Potencia(int nBase, int nExponente)
{
    int temporal = 1;          // Valor local de mi método Potencia
    for (int i = 1; i <= nExponente; i++) // Multiplico "n" veces
        temporal *= nBase;      // Para aumentar el valor temporal
    return temporal; // Al final, obtengo el valor que buscaba
}

public static void Main()
{
    int num1, num2; // variables locales de mi método Main
    Console.WriteLine("Introduzca la base: ");
    num1 = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("Introduzca el exponente: ");
    num2 = Convert.ToInt32(Console.ReadLine());
    Console.WriteLine("{0} elevado a {1} vale {2}", num1, num2,
    Potencia(num1, num2));
}

```

En este caso, las variables *temporal* e *i* son locales a la función *potencia*. Para *Main* no existen. Si en *Main* intentáramos hacer *i*=10, obtendríamos un mensaje de error.

Lo mismo ocurre con las variables, *num1* y *num2*, que son locales para *Main*. Desde la función *potencia* no podemos acceder a su valor (ni para leerlo ni para modificarlo).

En este ejemplo, no encontramos ninguna variable global.

Conflictos de nombres en las variables

¿Qué ocurre si trabajamos con la misma variable global en dos métodos diferentes?

```

static int n = 7;

public static void CambiaN()
{
    n++;
}

public static void Main()
{
    Console.WriteLine("n vale {0}", n); // el valor será 7
    CambiaN();
    Console.WriteLine("Ahora n vale {0}", n); // el valor cambiará a 8
}

```



En este ejemplo, vemos que tanto en el método CambiaN como en el método Main estamos trabajando con una variable global.

Ahora veamos otro ejemplo:

```
static int n = 7;

public static void CambiaN()
{
    int n = 10;
    n++;
    Console.WriteLine("Ahora n vale {0} dentro del método CambiaN", n);
}

public static void Main()
{
    Console.WriteLine("n vale {0}", n);
    CambiaN();
    Console.WriteLine("Ahora n vale {0}", n); // el valor que estamos
    sacando de n es el de la variable global
}
```

En el siguiente ejemplo, estamos trabajando con dos variables 'n' distintas. Hemos creado una variable local 'n' dentro del método CambiaN, en el cual modificamos su valor asignando 10 e incrementando después.

En Main, estamos trabajando con la variable global creada al inicio, por lo que el valor no varía siendo 7.

En este último ejemplo, vemos que podemos llegar a tener conflicto de variables a la hora de trabajar con variables globales (el ejemplo son pocas líneas de código. Si trabajáramos con un proyecto de cientos de líneas, la confusión podría ser mayor). Por ello, se recomienda trabajar con variables locales y paso de parámetros entre métodos.



ponte a prueba

Una variable local puede ser accedida desde cualquier función o método.

- a) Verdadero.
- b) Falso.

¿Qué muestra por pantalla el siguiente código?

```
public static string miVar="";
static void Main(string[] args)
{
    int a = 10;
    int b = 5;
    Console.WriteLine(suma(a,b));
    Console.WriteLine(miVar);
}
static int suma(int x,int y) { miVar = "HOLA ILERNA"; re-
turn x + y; }
```

- a) 15.
- b) HOLA ILERNA.
- c) 15
HOLA ILERNA.
- d) El paso de parámetros a la función no es correcto.

Según el siguiente código, ¿qué ocurrirá con la función?

```
public static float Mayor (float num1, float num2)
{
    if (num1 > num2)
        return num1;
    else
        return num2;
}
```

- a) La función nos devolverá dos valores: num1 y num2.
- b) Es incorrecto porque hay dos return.
- c) La función nos devolverá num1 si es mayor que num2 o num2 si es mayor que num1.
- d) Ninguna de las opciones es correcta.

4.7. PRUEBA, DEPURACIÓN Y COMENTARIOS DE PROGRAMAS

Una vez tenemos terminado nuestro programa con su código ya implementado, es momento de **probarlo** para comprobar que no tiene fallos y, sobre todo, que su funcionamiento es correcto. En muchos casos no existen errores, pero el programa no realiza la tarea que debería cuando se ejecuta.

Por esto es muy conveniente que nuestro programa tenga comentarios, al menos, en todo lo importante que realiza. De esta manera, nos será más fácil manejarlos 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.

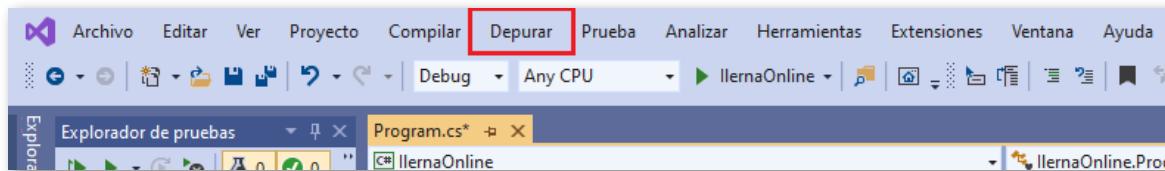
```

1  using System;
2
3  namespace IlernaOnline
4  {
5      class Program
6      {
7          static void MiMetodo(int variable)
8          {
9              variable = variable + 5;
10         }
11
12         static void Main(string[] args)
13         {
14             int numero = 1;
15             MiMetodo(numero);
16             Console.WriteLine(numero);
17             Console.ReadKey();
18         }
19     }
20 }
21

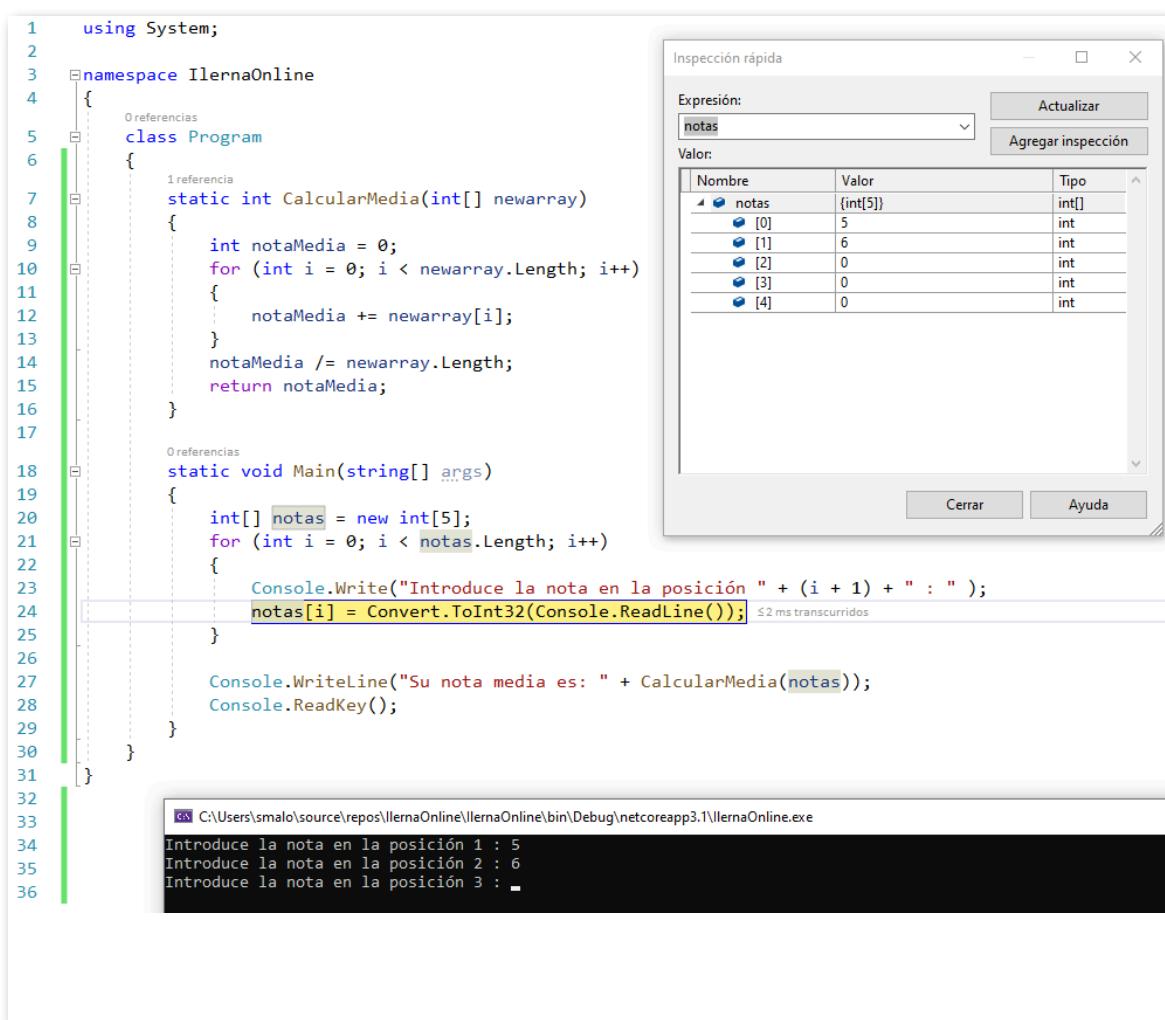
```

The screenshot shows a code editor with a C# script named 'Program.cs'. The code defines a namespace 'IlernaOnline' and a class 'Program' containing a method 'MiMetodo' and the main entry point 'Main'. In the 'Main' method, there is a call to 'Console.WriteLine(numero)' followed by a tilde (~). This tilde is highlighted in yellow, indicating it's a syntax error. A tooltip box appears on the right side of the screen with the message 'CS1002: Se esperaba ;', which translates to 'Expected ;' in English. The code editor interface includes line numbers from 1 to 21, code highlighting, and a vertical yellow bar on the left margin.

Por ejemplo, el programa Visual Studio cuenta con un menú **Depurar** que permite acceder a las herramientas que dispone el depurador. El **C#**, además, nos facilita 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 (**F11**), nuestro código para analizarlo de forma más detallada.



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.

• 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 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 invocuen 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 el resultado es el correcto. Los errores lógicos son los 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 funcionar nuestro programa. Así, nos proporcionará el resultado deseado.



ponte a prueba

Podemos poner un punto de interrupción en nuestros programas y depurar línea a línea.

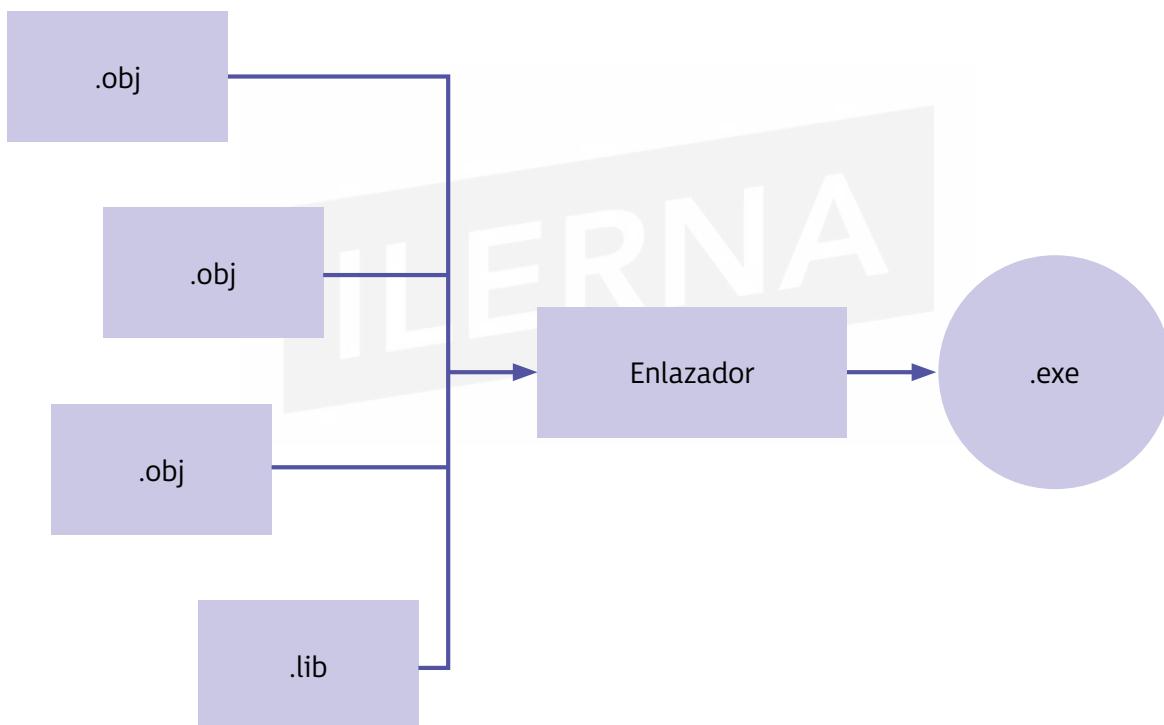
- a) Verdadero
- b) Falso

4.8. 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 todavía no se ha insertado en el programa. Va a ser el **enlazador o linkador** el que realice esta inserción y, por tanto, será 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 es, a veces, una tarea complicada, por lo que los IDE nos proporcionan una herramienta denominada "Depurador" que nos facilita la tarea.

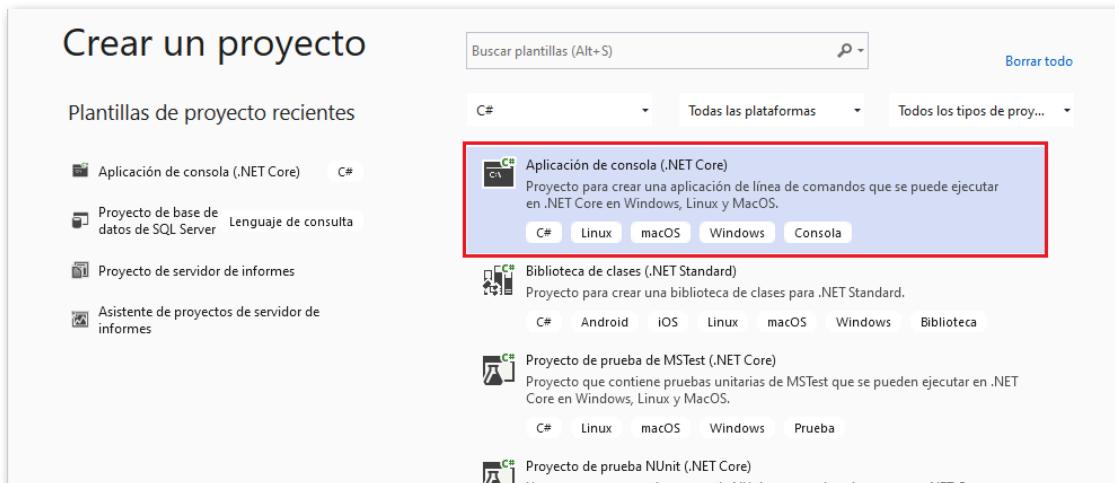
Uno de los motivos por lo que es recomendable utilizar el lenguaje de programación C# es que permite la **interoperación con otros lenguajes**, siempre que estos tengan:

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

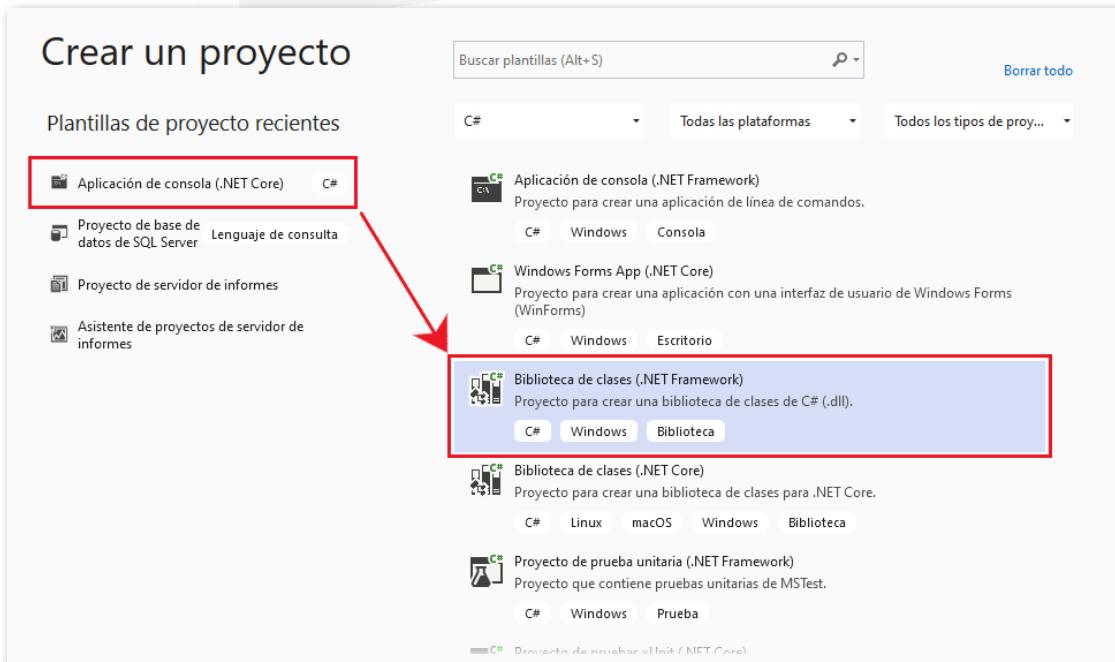
Podemos crear nuestras propias librerías en C# con VS2019.

Veámoslo paso a paso:

1. Creamos un proyecto nuevo como aplicación de consola.



2. Agregamos a nuestra solución una biblioteca de clases (ir a la ventana del explorador de soluciones, botón derecho y agregar).



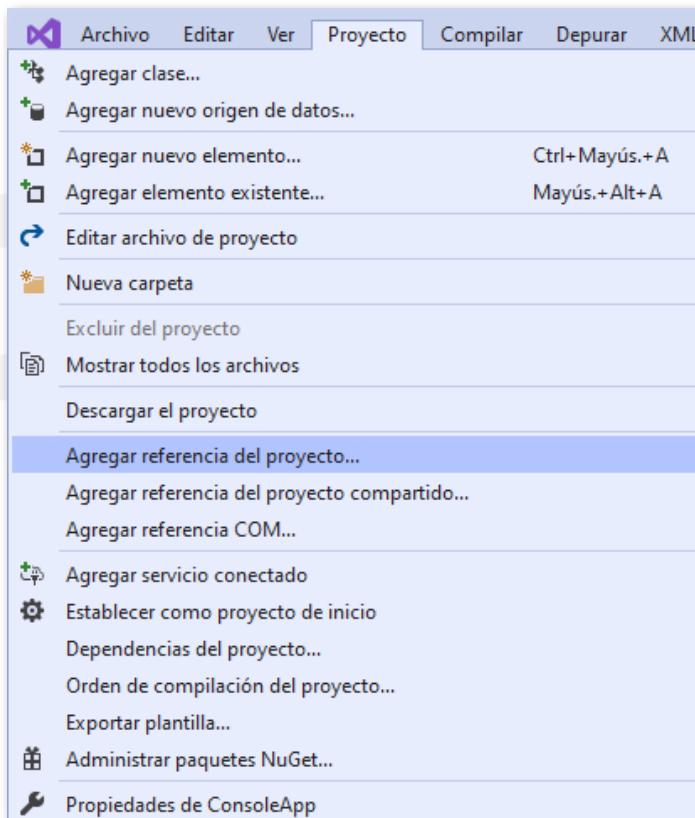
3. Creamos nuestra dll, realizando un módulo de 'Multiplicar'. Allí crearemos una función que realizará la multiplicación de dos números.

```

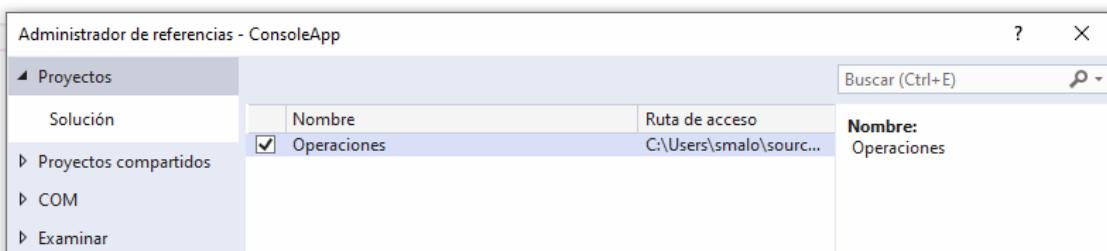
1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using System.Text;
5  using System.Threading.Tasks;
6
7  namespace Operaciones
8  {
9      public class Multiplica
10     {
11         public static int Multiplicar(int x, int y)
12         {
13             return x * y;
14         }
15     }
16 }
17

```

4. A mi proyecto, agregaremos como referencia (botón derecho -> referencia) mi dll con el nombre que habíamos puesto con anterioridad 'Operaciones'.



5. Ahora ya podemos llamar desde nuestro programa a nuestra referencia.



```

1  using System;
2
3  namespace ConsoleApp
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Operac...
10         }
11     }
12 }
13

```

The screenshot shows a code editor in Visual Studio. The cursor is at the end of the word 'Operac...' in the line 'Operac...'. A tooltip window is open, displaying the suggestion 'Operaciones' with a small preview of the code inside the 'Operaciones' class. The background code shows a basic console application structure.

```

1  using System;
2
3  namespace ConsoleApp
4  {
5      class Program
6      {
7          static void Main(string[] args)
8          {
9              Console.WriteLine(Operaciones.Multiplica.Multiplicar(5, 6));
10             Console.ReadKey();
11         }
12     }
13 }
14

```

The screenshot shows the completed code. The line 'Console.WriteLine(Operaciones.Multiplica.Multiplicar(5, 6));' has been replaced by the call 'Multiplica.Multiplicar(5, 6)'. The code now prints the result of the multiplication to the console and waits for a key press.

4.9. 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.

```

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.

```

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 calcula el número aleatorio que está dentro de un intervalo
        especificado devolviendo un número menor que el valor máximo de ese intervalo.
        Mostraremos los números dejando una separación de 6 unidades entre ellos */

        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();
    }
}

// Un ejemplo de salida de números enteros de -100 a 100
//      5     -5    -20     94    -86
//     -3     -16   -45    -19     47
//    -67    -93     40     82     68
//    -60    -55     67    -51    -11
// Un ejemplo de salida de números enteros de 1000 a 10000
//    4857    9897    4405    6606    1277
//    9238    9113    5151    8710    1187
//    2728    9746    1719    3837    3736
//    8191    6819    4923    2416    3028
// Un ejemplo de salida de números enteros de 1 a 10
//      4      4      5      9      9
//      5      1      2      3      5
//      5      4      5     10      5
//      7      7      7     10      5

```



ponte a prueba

Gracias a la librería de matemáticas, el IDE nos proporciona métodos ya implementados. Según esto, ¿qué realiza el siguiente código?

```
double grados =90;
double angulo    = Math.PI * grados / 180.0;
double result = Math.Cos(angulo);
Console.WriteLine(result);
```

- a) Saca por pantalla el coseno de un ángulo.
- b) Saca por pantalla el seno de un ángulo.
- c) El código no es correcto.
- d) Ninguna de las opciones es correcta.

4.10. 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:

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

Podemos definir la recursividad de la siguiente manera:

- Un **caso base** que permita la finalización del programa (sería nuestra condición de salida).
- **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:

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

Debemos saber qué 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:

```
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:

$$3! = 3 * 2 * 1$$

$$4! = 4 * 3 * 2 * 1$$

$$5! = 5 * 4 * 3 * 2 * 1$$

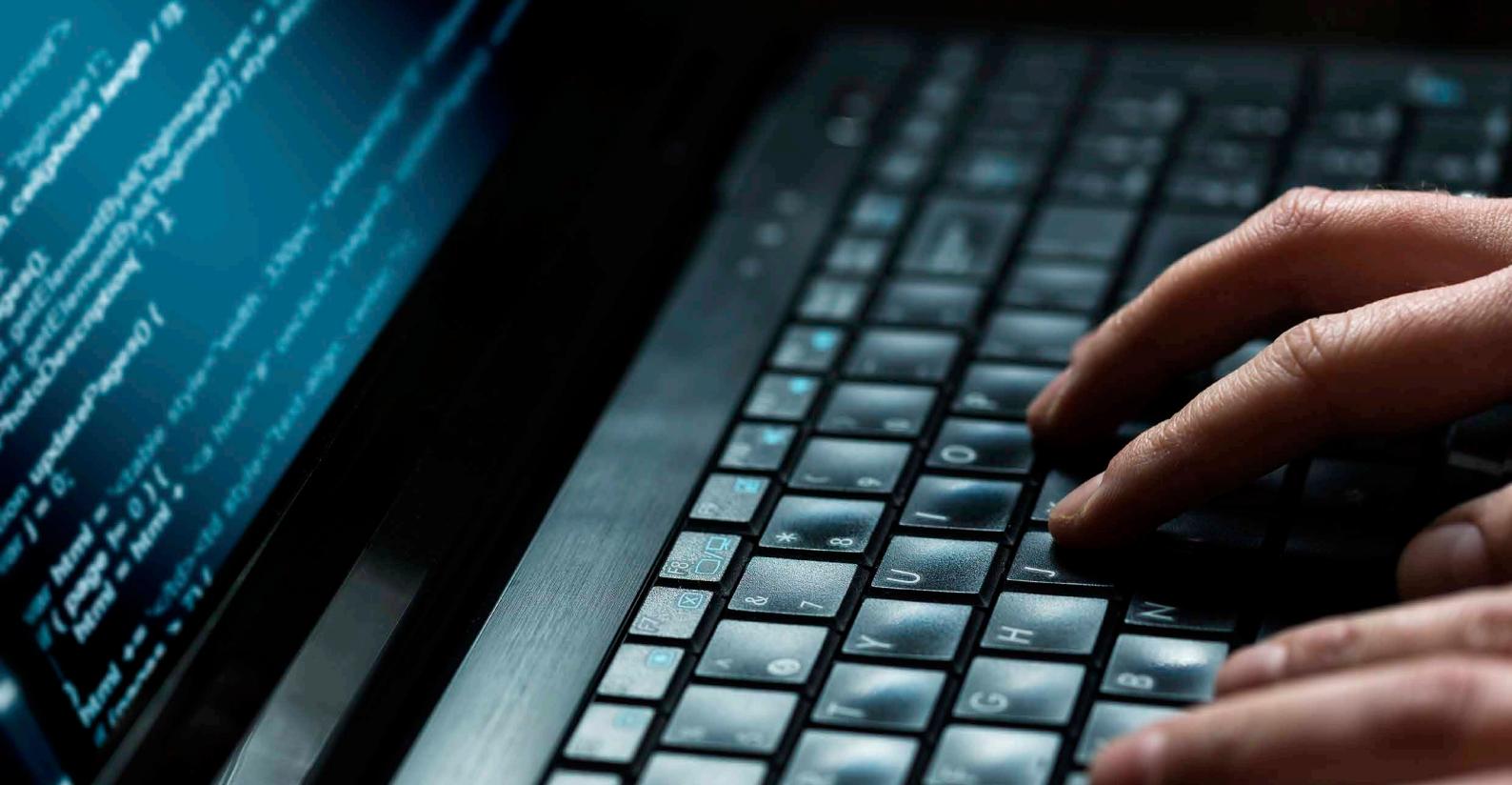
Podemos apreciar lo siguiente:

$$3! = 3 * 2 * 1$$

$$4! = 4 * 3!$$

$$5! = 5 * 4!$$

....



Vemos que en todos los casos sucede lo mismo, se multiplica el número por el número restando una unidad. Por lo que ya tenemos el caso recursivo:

$$\text{factorial}(n) = n * \text{factorial}(n-1)$$

De modo que nos quedaría de la siguiente forma:

Caso base → Si $n=1$, $\text{factorial}(1)$ devuelve 1.

Caso genérico → Si $n > 1$, $\text{factorial}(n) = n * \text{factorial}(n-1)$.

Si hacemos una función llamada “factorial” (entero n) retorna entero:

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

Si lo traducimos al lenguaje de programación C#:

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

```

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

```

A continuación, analizaremos otro ejemplo recursivo. En este caso, vamos a imprimir por pantalla n números naturales. Será el usuario quién decida qué números naturales querrá imprimir. Por ejemplo, si el usuario introduce el número 6, se imprimirá por pantalla la siguiente secuencia:

```
1 2 3 4 5 6
```

- **Caso base:** el usuario introduce valores que no son números naturales (es decir, menores que uno), el programa devolverá siempre 1.
- **Caso recursivo:** vamos a llamar a la función con dos parámetros. Así, uno de ellos controlará la salida e irá incrementando en 1, y el otro irá decrementando hasta llegar al caso base.

```

int salida = 1;
{
    si n>1 entonces
        // Caso recursivo
        n- -;
        Escribimos salida
        devuelve naturales (salida +1, n)
    Si no
        // Caso base
        devuelve salida;
    Fin Si;
}

```

El código en C# quedaría de la siguiente forma:

```

static int naturales (int salida, int n)
{
    if (n > 1)
    {
        n--;
        Console.WriteLine(" {0} ", salida);
        return printNatural (salida + 1, n);

    } else
        return salida
}

```



ponte a prueba

¿A qué se refiere el concepto de recursividad?

- a) Un tipo de método.
- b) La visibilidad de una función.
- c) Llamada de una función a sí misma.
- d) Una librería.

La recursividad necesita un caso base que permita la finalización del programa.

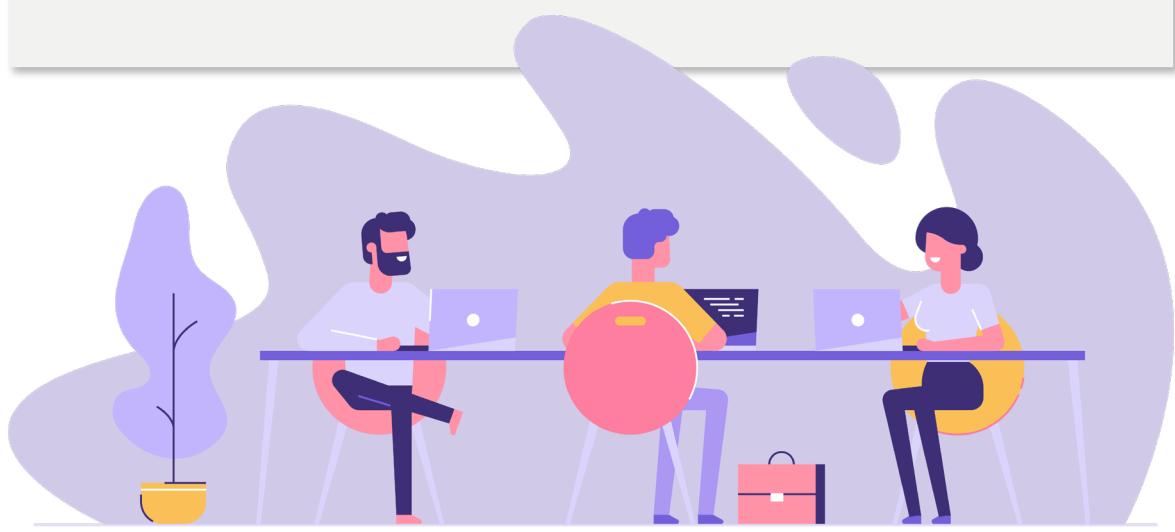
- a) Verdadero.
- b) Falso.

¿Qué realiza el siguiente código?

```
static int printNatural(int ctr, int stval) {
    if (ctr < 1) {return stval; }
    Console.WriteLine(" {0} ", ctr);
    ctr--;
    return printNatural(ctr, stval);
}
```

```
static void Main() {
    Console.Write(" ¿Cuántos números quieres imprimir? : ");
    int ctr = Convert.ToInt32(Console.ReadLine());
    printNatural(ctr, 1);
}
```

- a) Muestra por pantalla los n primeros números naturales.
- b) Muestra por pantalla los números de n a 1.
- c) Muestra la resta de dos números naturales.
- d) Muestra la suma de dos números naturales.



```
var RPC = new fastXDM.Client();
onInit: function() {
    setTimeout(function() {
        Community.resize();
    }, 500);
},
authorised: function(href) {
    var href = location.href;
    if (href.indexOf('http://') > -1)
        else href = href + '?';
    if (href.indexOf('http://') > -1)
        location.href = href;
    return true;
}
}, {
    unauthorised: function(href) {
        var href = location.href;
        if (href.indexOf('http://') > -1)
            href = href + '?';
        href = href + href;
        location.href = href;
    }
}
```

5

GESTIÓN DE FICHEROS

Hasta aquí, hemos visto el uso de variables y las estructuras de datos y hemos trabajado con información que, 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 ejecutemos una aplicación que trabaja con esos datos podrá leer del fichero aquellos que sean necesarios.

5.1. 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 volátiles de la información, la BCL (*Base Class Library*) reserva un espacio de nombres denominados System.IO, destinado a trabajar con ellos.

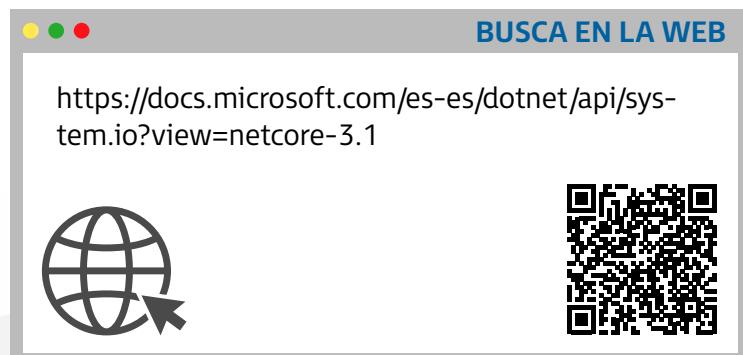
El espacio de nombres System.IO contiene tipos que permiten leer y escribir en los archivos. Además, ese espacio de nombres nos proporciona compatibilidad entre archivos y directorios.

Algunas de las clases que nos proporciona son:

CLASE	DESCRIPCIÓN
BinaryReader	Lee tipos de datos primitivos como valores binarios.
BinaryWriter	Escribe tipos primitivos en formato binario en una secuencia y admite la escritura de cadenas.
File	Proporciona métodos estáticos para crear, copiar, eliminar, mover y abrir un solo archivo. Contribuye a la creación de objetos FileStream.
BufferedStream	Almacena en un búfer los datos necesarios para las operaciones de lectura y escritura en otra secuencia. No se puede heredar esta clase.
FileStream	Proporciona un Stream para un archivo, lo que permite operaciones de lectura y escritura sincrónica y asincrónica.

StreamReader	Implementa un TextReader que lee los caracteres de una secuencia de bytes.
StreamWriter	Implementa TextWriter para escribir los caracteres de una secuencia.
TextReader	Representa un lector que puede leer una serie secuencial de caracteres.
TextWriter	Representa un escritor que puede escribir una serie secuencial de caracteres.

Para obtener más información de las clases que del espacio de nombres de System.IO, podemos acceder a la documentación de Microsoft:



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 el 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 manejar esas estructuras (creación de ficheros, añadir información en un fichero ya existente...).



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

- Abrir el fichero.
- Escribir o leer registros del fichero.
- Cerrar el fichero.

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 un conjunto de bytes con una 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

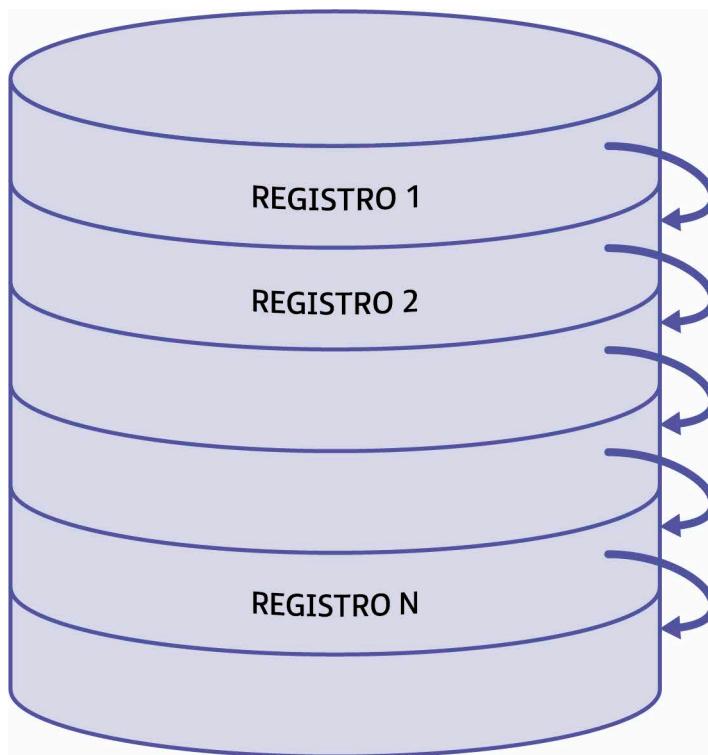
En el apartado anterior hemos visto la relación entre fichero y directorio. Para la identificación inequívoca de este fichero, habrá que nombrar el camino que nos lleva hasta él. A este camino lo denominamos **ruta**.

Dependiendo de cómo empecemos la ruta de directorio para nombrar el archivo, podemos tener dos tipos de rutas bien diferenciados:

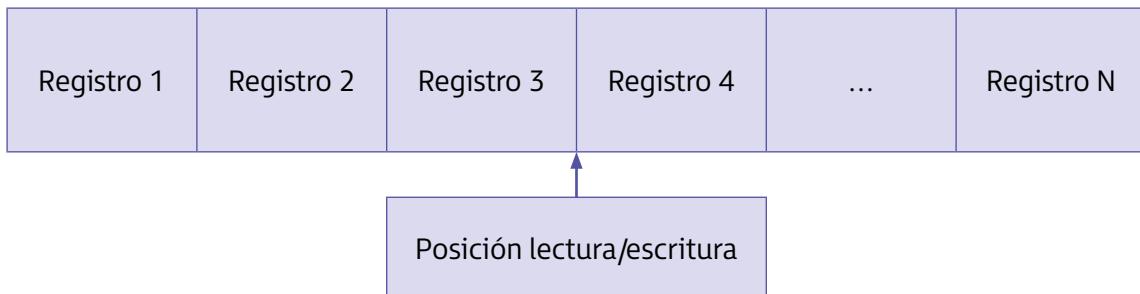
- Ruta absoluta o completa: se indica el camino del directorio desde el comienzo. En el sistema operativo Linux, empieza por "/". En el sistema operativo Windows o MS-DOS, cada partición posee un directorio raíz (por ejemplo, C:\). No existe un directorio raíz común que los contenga a todos ellos como en Linux.
- Ruta relativa: se le indica el camino del directorio desde la posición actual. Por tanto, la ruta no empezaría desde el directorio raíz.

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 recorriendolos de uno en uno.



- **Aleatorio o directo:** en los ficheros aleatorios, 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 (índice) para ser identificados. Permiten el acceso secuencial y aleatorio a un fichero de la siguiente forma:

- Primero busca de forma secuencial el campo clave o índice.
- 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.

Campo clave o índice

1	1510
2	77
3	3680

Archivo de datos



- **Según su estructura:**

- **Ficheros de texto:** formados por texto plano con formato legible por el usuario.
- **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. Los datos se encriptan (codifican) en ceros y unos, de manera que se hace mucho más eficiente su almacenamiento.



ponte a prueba

¿En qué tipo de dispositivo de almacenamiento se guarda el contenido de los ficheros?

- a) Primario.
- b) Secundario.
- c) Terciario.
- d) Ninguna de las opciones es correcta.

¿Cuál de las siguientes afirmaciones sobre un fichero secuencial indexado es correcta?

- a) Está compuesto por un índice secuencial.
- b) Está formado por un puntero que nos indica el campo que queremos acceder.
- c) No podemos programar en C# este tipo de ficheros.
- d) A y B son correctas.

¿Qué tipo de ruta es esta: ./carpeta/fichero.dat?

- a) Es una ruta relativa.
- b) Es una ruta absoluta.
- c) Es una ruta secuencial.
- d) Ninguna respuesta es correcta.

5.2. DISEÑO Y MODULACIÓN DE LAS OPERACIONES SOBRE FICHEROS

5.2.1. 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.

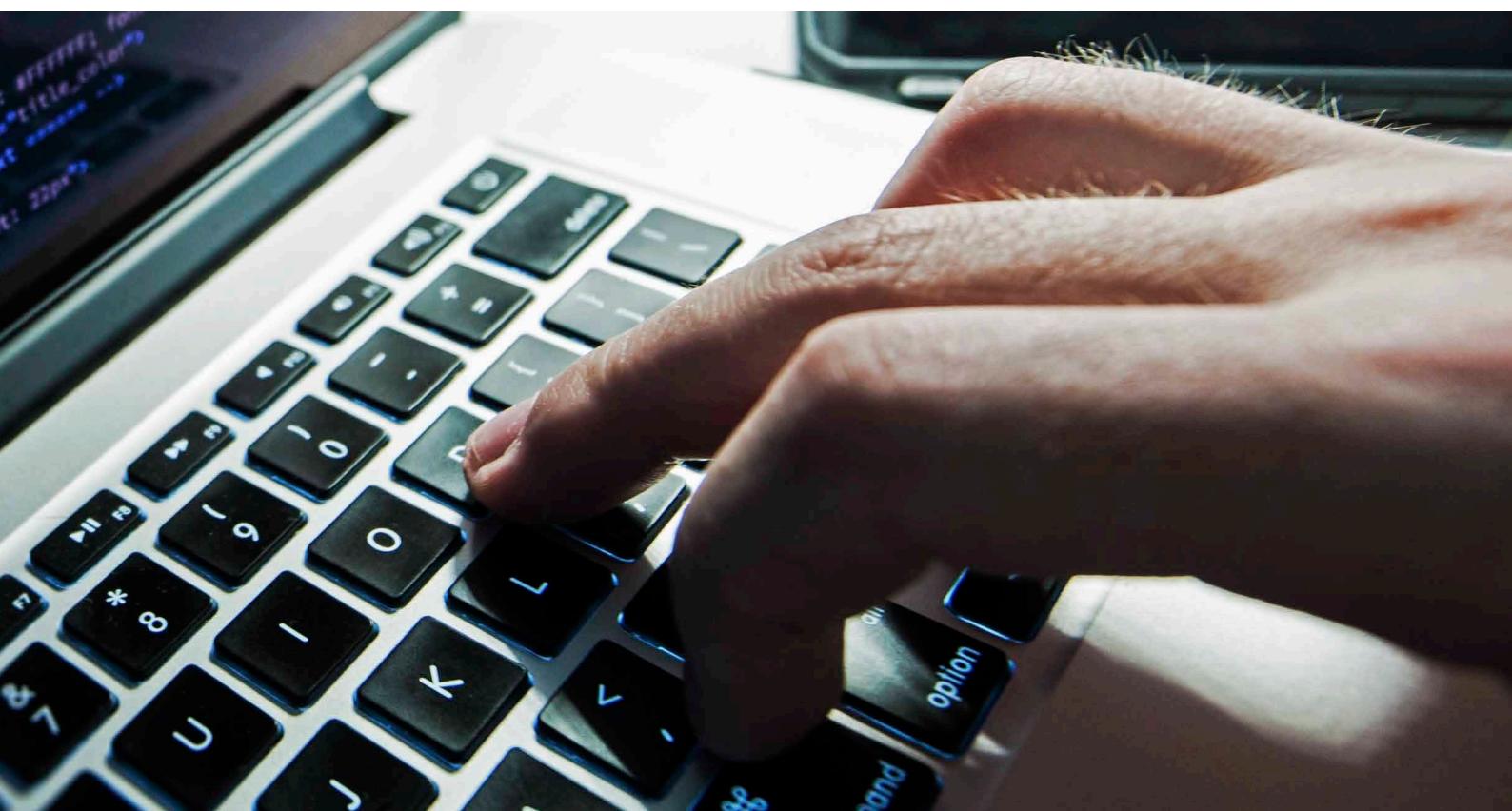
La clase Stream nos permite abstraer de una secuencia de bytes como, por ejemplo, un archivo, un dispositivo de entrada/salida, una canalización de comunicación entre procesos o un socket.

La clase FileStream deriva de la clase Stream y nos proporciona los siguientes métodos:

```
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 está guardado o donde se guardará el fichero.



Puede escribirse:

```
c:\\users\\AppData\\file.txt
@c:\\users\\AppData\\file.txt
```

La ventaja de utilizar la @ es que las secuencias de escape no se procesan. Una secuencia de escape son combinaciones de caracteres que consisten en una barra diagonal inversa (\) seguida de una letra como, por ejemplo, \n que representa una nueva línea.

FileMode nos indica cómo se debe abrir un archivo. Los campos pueden 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.

En el flujo de entrada de datos (lectura), solo podemos realizar la operación de lectura de un fichero, es decir, existe una comunicación unilateral desde el fichero al programa.

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.



ponte a prueba

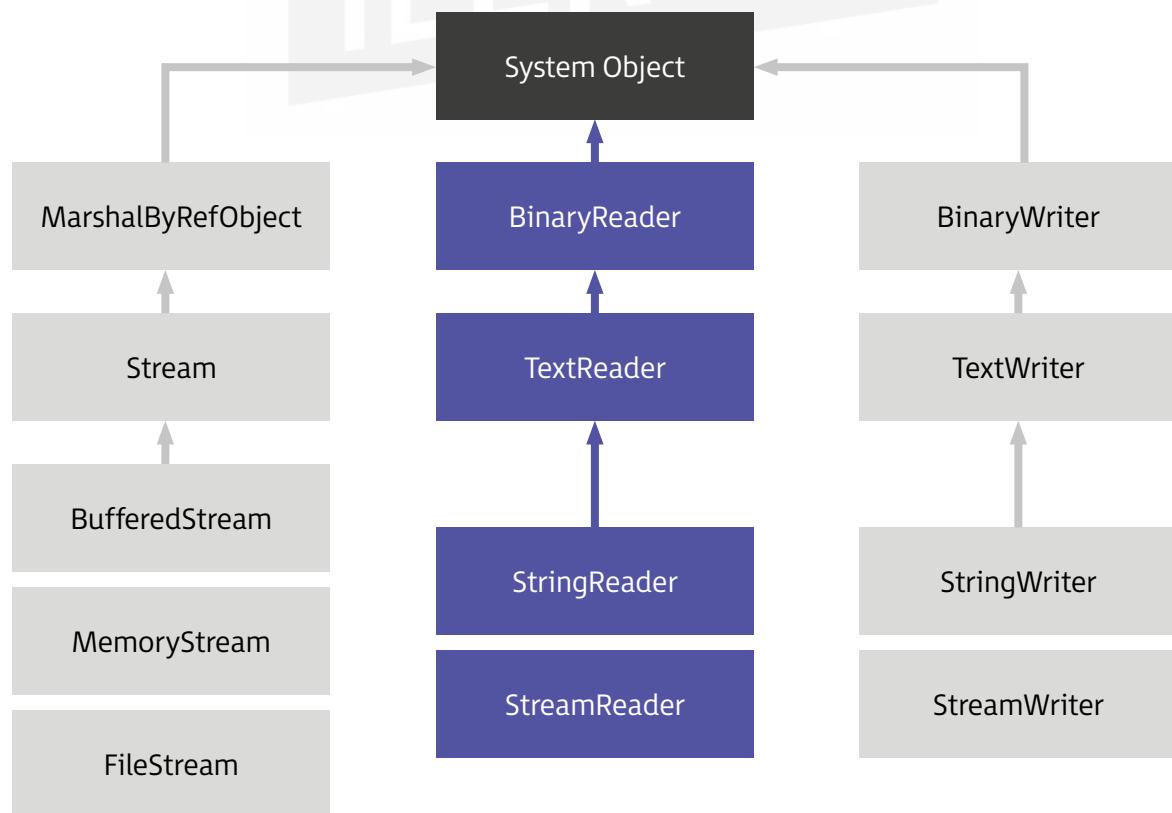
¿Qué es FileStream?

- a) Es un tipo de variable de ficheros.
- b) Es un tipo de visibilidad de métodos en el manejo de ficheros.
- c) Es una clase que permite operaciones de lectura y escritura.
- d) Todas las opciones son correctas.

¿Qué hace la directiva Open?

- a) Abre un fichero si existe. Si no existe, se crea un fichero nuevo.
- b) Abre un fichero existente. Si el fichero no existe, lanzará un error.
- c) Crea un nuevo fichero. Si existe, lanzará un error.
- d) Abre un fichero para añadir datos al final de este.

5.2.2. CLASES DE FLUJOS



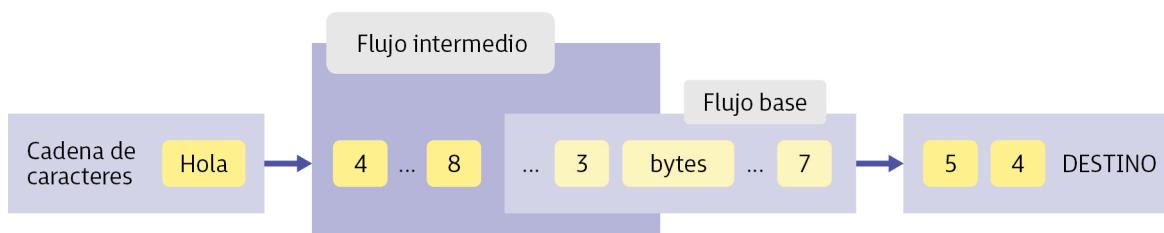
Flujos base

Flujos intermedios

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**:

- **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 pueden combinar con el flujo base de manera que este 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.



5.3. 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:

- **Open**: realizamos la apertura de un fichero. Recordemos que se desencadena una excepción si el fichero no existe (FileNotFoundException).
- **Create**: creamos un fichero nuevo. Si el archivo ya existe, se sobrescribirá. Es equivalente a solicitar que se utilice CreateNew si no existe el archivo, y que se utilice Truncate en caso contrario.
- **CreateNew**: creamos un fichero nuevo. Si el fichero ya existe, lanzará una excepción (IOException).

- **Truncate**: abrimos un archivo que ya existe. Cuando se abra, se debe truncar su tamaño a cero bytes. Al intentar leer un abierto en este modo, se producirá una excepción (ArgumentException).
- **OpenOrCreate**: se abre un archivo si ya existe. En caso contrario, se crea uno nuevo.
- **Append**: se abre el archivo si existe y realiza una búsqueda hasta el final. Si el archivo no existe, crea uno nuevo.

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

```
// PSEUDOCÓDIGO
// 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 leido;
FinMientras

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

Escritura secuencial

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

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

Mientras existan datos a escribir
    f1.escribir(registro);
FinMientras

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



Lectura aleatoria

```
// PSEUDOCÓDIGO  
// Declaración de la variable del fichero  
fichero f1;  
  
// Abrimos el fichero para leerlo  
f1.abrir(lectura);  
  
Situar el puntero del fichero justo antes del  
registro requerido  
f1.leer(registro);  
Otras operaciones con registro leído;  
  
// Cerramos el fichero  
f1.cerrar();
```

Escritura aleatoria

```
// 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);  
    otras operaciones con registro leído;  
finMientras  
  
//Cerramos el fichero  
f1.cerrar();
```

3. Cierre

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



ponte a prueba

¿Qué realiza el siguiente código?

```
FileStream fichero = new FileStream("C:/fichero/ejercicio1.txt", FileMode.Open, FileAccess.Read);
StreamReader fs = new StreamReader(fichero);
string linea = "";
while ((linea = fs.ReadLine()) != null)
    Console.WriteLine(linea);
fs.Close();
fichero.Close();
```

- a) Muestra por pantalla una cadena vacía.
- b) Muestra por pantalla el contenido del fichero "ejercicio1.txt".
- c) Escribe línea en el fichero "ejercicio1.txt".
- d) Ninguna de las opciones es correcta.

La programación en C# de los ficheros secuenciales y aleatorios se realiza teniendo en cuenta la posición del puntero.

- a) Verdadero.
- b) Falso.

5.3.1. CLASE FILESTREAM

En el apartado 5.2.1 vimos los tipos de métodos que nos proporciona esta clase y los diferentes valores que pueden tomar el parámetro *modo* y *acceso*.

```
FileStream (string nombre, FileMode modo)
FileStream (string nombre, FileMode modo, FileAccess 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 utilizando los flujos de las clases **StreamWriter** y **StreamReader**:

- StreamReader implementa un TextReader que lee los caracteres de una secuencia de bytes.
- StreamWriter implementa un TextWriter para escribir los caracteres de una secuencia.

Lectura de un fichero de texto en C#:

```
using System.IO; // Importación de la clase System.IO
// Creamos la variable fichero con la clase FileStream pasando la ruta
// del fichero, el modo de apertura (Open) y cómo vamos a acceder a él
// (lectura)
FileStream fichero = new FileStream("C:/fichero/ejercicio1.txt", FileMode.Open, FileAccess.Read);
// Abrimos el flujo de lectura
// Proporciona una vista genérica de una secuencia de bytes
StreamReader fs = new StreamReader(fichero);
string linea = "";
while ((linea = fs.ReadLine()) != null) // Mientras no lleguemos a final de fichero
    Console.WriteLine(linea); //sacamos por pantalla lo que contenga el fichero
fs.Close(); // cerramos el flujo
fichero.Close(); // cerramos el fichero
}
```



Escritura de un fichero de texto en C#

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

// Creamos la variable fichero con la clase FileStream pasando la ruta
// del fichero, el modo de apertura (en este caso, vamos a añadir datos al
// final del fichero -> Append) y cómo vamos a acceder a él (escritura)

FileStream fichero = new FileStream("C:/fichero/ejercicio1.txt", FileMode.Append, FileAccess.Write);

// abrimos el flujo de escritura

StreamWriter fs = new StreamWriter(fichero);

//pedimos al usuario una cadena de texto la escribiremos en el fichero

Console.WriteLine("Introduce una frase: ");

string frase = Console.ReadLine();

fs.WriteLine(frase);

// cerramos el flujo y el fichero

fs.Close();

fichero.Close();
```

Ficheros binarios

Generalmente, los usuarios trabajamos con ficheros de texto, pero también es frecuente encontrarnos con información que esté organizada como una secuencia de bytes. Nos referimos a ficheros de imágenes, ficheros de sonido, ficheros de video, etc.

Lo primero que vamos a analizar es cómo abrir un fichero de forma que podamos leer los bytes que contiene.

Comenzaremos por utilizar la clase FileStream, diseñada para acceder a un byte o a bloques de byte. Así, esta clase, proporciona un método para el acceso:

- Read (Byte[], int offset, int count) lee un bloque de bytes de la secuencia y escribe los datos en un búffer dado.

Los parámetros:

- array Byte []: contiene la matriz de bytes especificada con los valores de offset y reemplazados por los bytes leídos desde el origen actual.
- offset: se colocarán los bytes leídos.
- Count: el número máximo de bytes que se deben leer.

Por ejemplo:

```
byte[] buffer = new byte[fichero.Length];
// creamos un array unidimensional con la cantidad de bytes que se
// compone el fichero
fichero.Read(buffer, 0, buffer.Length);
// Vamos a leer el buffer donde todavía no hemos consumido ningún byte
y el número máximo que vamos a leer es lo que contiene el buffer
```

Ejemplo de lectura en ficheros binarios

```
using System.IO;

// Vamos a trabajar con un fichero imagen, de tal forma que lo abriremos de modo lectura

FileStream fichero = new FileStream("C:/fichero/pelota.jpg", FileMode.
Open, FileAccess.Read);

// Creamos un buffer de tipo byte para poder trabajar con los datos del
fichero

Byte [] buffer = new byte[fichero.Length]; //un entero de 8 bits sin
signo.

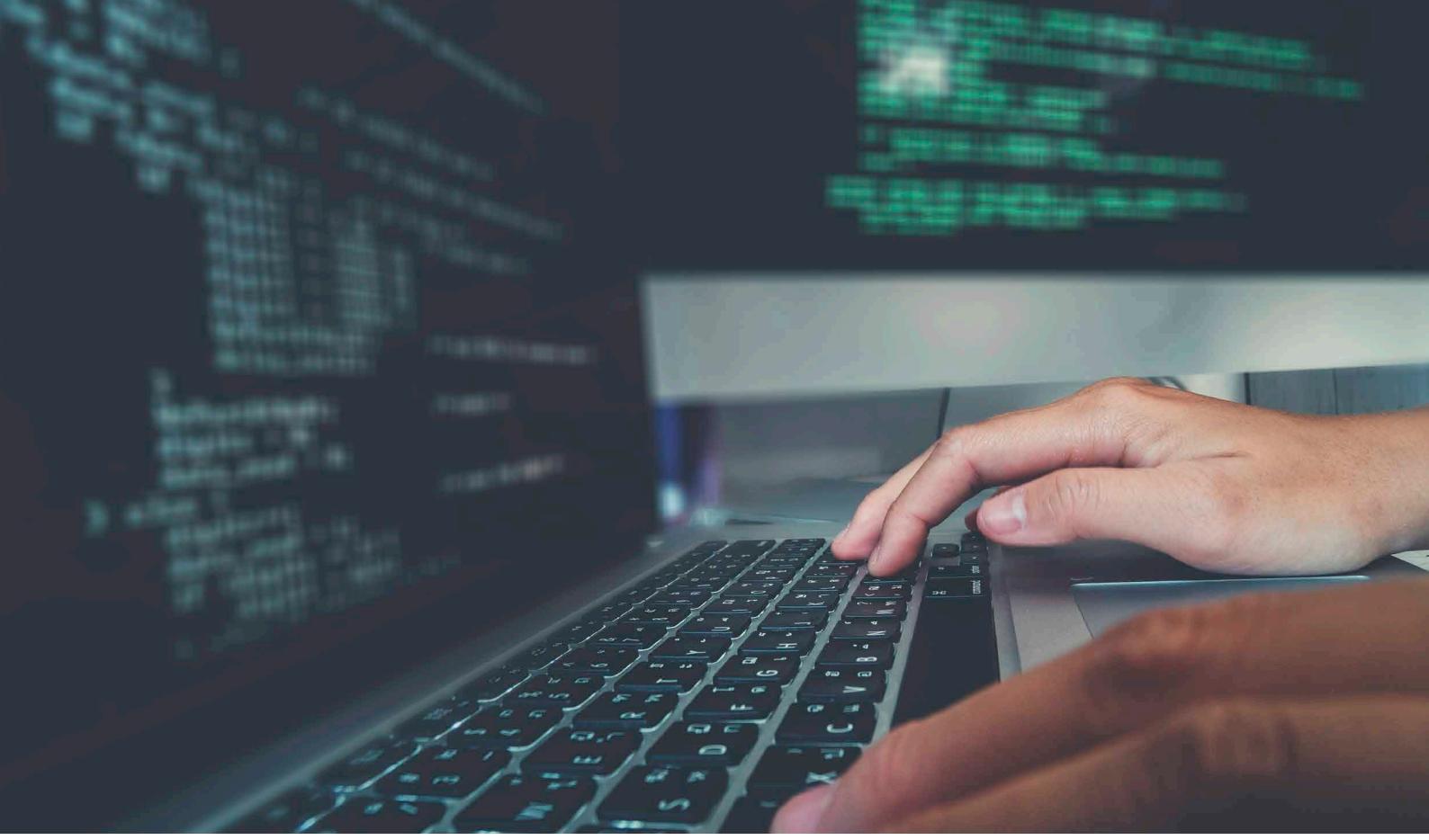
// Abrimos el fichero para la lectura y vamos a ir consumiendo todos
los bytes del fichero y mostrándolos por pantalla

fichero.Read(buffer, 0, buffer.Length);

for (int i = 0; i < buffer.Length; i++)
{
    Console.WriteLine(buffer[i]);
}

//fichero.Close();
```





Ejemplo de escritura en ficheros binarios

```
using System.IO;

// Vamos a trabajar con un fichero imagen, modificando su contenido.
Primero lo abriremos de modo lectura y escritura

FileStream fichero = new FileStream("C:/fichero/pelota.jpg", FileMode.
Open, FileAccess.ReadWrite);

// Creamos un buffer de tipo byte para poder trabajar con los datos del
fichero

Byte[] buffer = new byte[fichero.Length]; //un entero de 8 bits sin signo.

// Abrimos el fichero para la lectura y vamos a ir consumiendo todos
los bytes del fichero

fichero.Read(buffer, 0, buffer.Length);

// Ahora modificamos algunos bytes

for (int i = 1000; i < buffer.Length; i++)

{
    buffer[i] = 0;
}

// Vamos a sustituir los caracteres en el fichero, volcamos el contenido
// del buffer en el fichero y lo cerramos

fichero.Write(buffer, 0, buffer.Length);

fichero.Close();
```

ponte a prueba

¿Qué realiza el siguiente código?

```
FileStream fichero = new FileStream("C:/fichero/ejercicio1.txt",
    FileMode.Append, FileAccess.Write);

StreamWriter fs = new StreamWriter(-
fichero);

Console.WriteLine("Introduce una
frase: ");

string frase = Console.ReadLine();

fs.WriteLine(frase);

fs.Close();

fichero.Close();
```

- a) Escribirá la frase introducida por el usuario en un fichero binario.
- b) Escribirá la frase introducida por el usuario en un fichero de texto.
- c) Escribirá la frase introducida por el usuario en un fichero de datos.
- d) El código contiene errores porque falta cerrar el flujo de datos.

La clase StreamReader es utilizada para leer texto de un archivo.

- a) Verdadero.
- b) Falso.

¿Cuál es el objetivo de utilizar un buffer de lectura para ficheros de datos?

- a) Para almacenar el conjunto de bytes formado por el fichero de datos.
- b) Para guardar línea a línea el fichero de texto.
- c) Para controlar el fin de fichero.
- d) Ninguna opción es correcta.

5.4. CONTROL DE EXCEPCIONES

Hasta ahora, hemos trabajado con ficheros para realizar su lectura y escritura. El problema es que no hemos comprobado si ese fichero realmente existe o no, lo que puede suponer que nuestro programa falle en caso de que el fichero no se encuentre donde esperamos.

Una primera solución es usar el método `exists` para comprobar si está, antes de intentar abrirlo.

- **Exists (string path):** le pasamos la ruta del fichero y devolverá un booleano. Si no existe, devolverá un false. Si la ruta no existe o es incorrecta, también devolverá un false.

```
using System.IO;
if (File.Exists(@"D:\myfile.txt")) {
    Console.WriteLine("El fichero existe...");
} else {
    Console.WriteLine("El fichero no existe en el directorio D...");
```

- **Try/catch:** el bloque `try` contiene el código protegido que puede producir la excepción.

Los mensajes de error y/o acciones para corregir el error estarán en el bloque `catch`.

```
using System.IO;
string linea;

try
{
    FileStream fichero = new FileStream("C:/fichero/ejercicio1.
txt", FileMode.Open, FileAccess.Read);
    StreamReader fs = new StreamReader(fichero);
    if ((linea= fs.ReadLine())!= null)
        Console.WriteLine(linea);
    fs.Close();
    fichero.Close();
}
catch (Exception exp)
{
    Console.WriteLine("Ha habido un error: {0}", exp.Message);
}
```

BIBLIOGRAFÍA / WEBGRAFÍA

- ❑ Ceballos, F. J. (2011). *Microsoft C#. Curso de programación*. 2^aEdición. RA-MA Editorial.
- ❑ Serna, M. E. (2009). *Edsger Wybe Dijkstra*. Revista Digital Lámpsakos, no. 2, pp. 107-11.
- ❑ Nacho Cabanes. *Introducción a C#*. 2020. Sitio web: <http://www.nachocabanes.com/csharp/curso2015/index.php>
- ❑ Pildorasinformaticas. Canal de Youtube. Sitio web: <https://www.youtube.com/user/pildorasinformaticas>
- ❑ Corrado Böhm, Giuseppe Jacopini. (1966). *Flow diagrams, turing machines and languages with only two formation rules*. 2020, de ACM Digital Library, Association for Computing Machinery. Sitio web: <https://dl.acm.org/doi/pdf/10.1145/355592.365646>

solucionario

1. Introducción a la programación

¿Cómo debe ser un programa informático?

d) Todas las opciones anteriores son correctas.

¿Cuál de las siguientes opciones sobre el siguiente diagrama de flujo es correcta?

a) Muestra los n primeros números impares.

C# tiene dos categorías tipos de datos integrados: por valor y por referencia.

a) Verdadero.

2.1. Bloques de un programa informático

¿Qué es un IDE?

c) Es un software diseñado para el desarrollo de aplicaciones.

Siempre que ejecutemos un programa, el primer método que se ejecutará será el Main.

a) Verdadero.

¿Qué realiza el siguiente código?

c) Muestra por pantalla Hello World!

2.2. Variables. Usos y tipos

¿Cuál de las siguientes opciones es una característica de una variable?

d) Todas las respuestas son correctas.

¿Qué tipo de dato es un char?

a) Carácter.

Las variables globales son aquellas que están declaradas dentro de un método o función

b) Falso.

Las variables globales son aquellas declaradas fuera de un método o función.

2.3. Constantes y literales. Tipos y utilidades

¿Con qué palabra reservada se define una constante?

a) Const.

¿Qué representa el literal \n?

a) Una nueva línea.

2.4. Operadores del lenguaje de programación

¿Cuál es el resultado del siguiente código?

d) Ninguna de las repuestas es correcta.

La variable num no se ha declarado, por lo que daría un error.

¿Cuál de las siguientes opciones sobre el operador ! es correcta?

a) Calcula la negación lógica de nuestro operando.

2.5. Conversiones de tipos de clase

Las conversiones implícitas son aquellas que se producen cuando el valor que se va a almacenar se puede almacenar en la variable sin pérdida de información.

a) Verdadero.

¿Qué convierte el método ToString?

d) El valor especificado en su representación de cadena.

2.6. Comentarios al código

Según el siguiente código, ¿qué saldrá por pantalla?

d) Nada, porque el código está comentado.

Para hacer comentarios de una sola línea, debemos utilizar el operador /.

b) Falso.

Hay que utilizar la doble barra //.

3.1. Fundamentos de la programación

Dijkstra concluyó que la combinación de varios tipos de instrucciones, podemos crear un programa estructurado. ¿Cuáles son?

d) Todas las opciones son correctas.

Según Böhm y Jacopini, ¿cuál de las siguientes opciones sobre un programa es cierta?

d) Todas las respuestas son correctas.

3.3. Diseño de algoritmos

En la etapa de diseño, tomamos los requisitos de los clientes.

b) Falso.

Es en la etapa de análisis donde tomamos los requisitos.



solucionario

Después de la fase de pruebas, ¿qué etapa se lleva a cabo?

c) Mantenimiento.

3.5. Tipos de datos: simples y compuestos

¿Qué tipo de datos es el tipo enumerado?

b) Simple.

¿Cómo puedo acceder al dato 2 si tenemos el siguiente array? int[] a = new int[] {1, 2, 3};

a) a[1]

¿Cuántas columnas contendrá la siguiente matriz? int [,] matriz = new int[2,3];

b) 3

3.6. Estructuras de selección

Según el siguiente código, ¿cuándo imprimimos la cadena "Hola a todos"?

c) Cuando se cumpla la condición a y b.

Según el siguiente código:

a) Escribimos por pantalla "Estamos en el if".

3.7. Estructuras de repetición

Según el siguiente código, ¿ejecutaríamos el código de la salida por pantalla?

a) Sí, con que se cumpla una condición es suficiente.

¿Cuántas veces ejecutaríamos un bucle do-while?

b) Mínimo, una vez.

¿Es correcto el siguiente código?

b) No. La variable i no está declarada.

3.8. Estructuras de salto

Según el siguiente código, ¿qué saldrá por pantalla?

a) Case 1

3.9. Tratamiento de cadenas

¿Qué mostrará por pantalla el siguiente código?

a) 3.

La función Trim() elimina cualquier carácter del principio y el final de la cadena.

b) Falso.

Si no se pasa argumentos a la función Trim, elimina todos los caracteres de espacio en blanco del principio y el final de la cadena.

3.10. Depuración de errores

¿Qué errores podemos encontrar en la etapa de la depuración de un programa?

d) Todas las respuestas son correctas.

El programador está realizando un programa en el cual quiere introducir dos datos enteros. Uno, que sea la base y otro que sea el exponente para realizar la operación potencia. Introduce como base el número 5 y como exponente 2. El resultado es 10. ¿Qué tipo de error le está dando a nuestro programador?

b) Lógico.

3.12. Entornos de desarrollo de programas

¿Qué ventajas nos ofrecen los IDE?

d) Todas las opciones son correctas.

4.2. Ventajas e inconvenientes

¿Qué ventajas nos proporciona la programación modular?

d) Todas las opciones son correctas.

4.3. Análisis descendente (top down)

¿En qué consiste el diseño top down?

b) Dividir un gran problema en subproblemas.

4.5. Llamadas a funciones. Tipos y funcionamiento

Las funciones deben tener un tipo asociado.

a) Verdadero.

¿Cuál es la salida del siguiente código?

a) 15.

Cuando realizamos un paso por valor de una variable, creamos dos posiciones de memoria distintas.

a) Verdadero.



solucionario

4.6. Ámbito de las llamadas a funciones

Una variable local puede ser accedida desde cualquier función o método.

- b) Falso.

Solo se pueden acceder a variables locales desde el método donde están creadas.

¿Qué muestra por pantalla el siguiente código?

- c) 15
HOLA ILERNA.

Según el siguiente código, ¿qué ocurrirá con la función?

- c) La función nos devolverá num1 si es mayor que num2 o num2 si es mayor que num1.

4.7. Prueba, depuración y comentarios de programas

Podemos poner un punto de interrupción en nuestros programas y depurar línea a línea.

- a) Verdadero.

4.9. Uso de librerías

Gracias a la librería de matemáticas, el IDE nos proporciona métodos ya implementados.

Según lo comentado, ¿qué realiza el siguiente código?

- a) Saca por pantalla el coseno de un ángulo.

4.10. Recursividad

¿A qué se refiere el concepto de recursividad?

- c) Llamada de una función a sí misma.

La recursividad necesita un caso base que permita la finalización del programa.

- a) Verdadero.

¿Qué realiza el siguiente código?

- b) Muestra por pantalla los números de n a 1.

5.1. Concepto y tipos de ficheros

¿En qué tipo de dispositivo de almacenamiento se guarda el contenido de los ficheros?
b) Secundario.

¿Cuál de las siguientes afirmaciones sobre un fichero secuencial indexado es correcta?

- d) A y B son correctas.

¿Qué tipo de ruta es esta: ./carpeta/fichero.dat?

- a) Es una ruta relativa.

5.2.1. Fundamentos de los flujos

¿Qué es FileStream?

- c) Es una clase que permite operaciones de lectura y escritura.

¿Qué hace la directiva Open?

- b) Abre un fichero existente. Si el fichero no existe, lanzará un error.

5.3. Operaciones sobre ficheros secuenciales

¿Qué realiza el siguiente código?

- b) Muestra por pantalla el contenido del fichero "ejercicio1.txt".

La programación en C# de los ficheros secuenciales y aleatorios se realiza teniendo en cuenta la posición del puntero.

- b) Falso.

Solo hay que controlar el puntero en los ficheros aleatorios.

5.3.1. Clase FileStream

¿Qué realiza el siguiente código?

- b) Escribirá la frase introducida por el usuario en un fichero de texto.

La clase StreamReader es utilizada para leer texto de un archivo.

- a) Verdadero.

¿Cuál es el objetivo de utilizar un buffer de lectura para ficheros de datos?

- a) Para almacenar el conjunto de bytes formado por el fichero de datos.