

1. Toma de contacto con C#

1.1. *Conceptos básicos sobre programación*

1.1.1. Programa y lenguaje

Un **programa** es un conjunto de órdenes para un ordenador.

Estas órdenes se le deben dar en un cierto **lenguaje**, que el ordenador sea capaz de comprender.

El problema es que los lenguajes que realmente entienden los ordenadores resultan difíciles para nosotros, porque son muy distintos de los que nosotros empleamos habitualmente para hablar. Escribir programas en el lenguaje que utiliza internamente el ordenador (llamado "**lenguaje máquina**" o "código máquina") es un trabajo duro, tanto a la hora de crear el programa como (especialmente) en el momento de corregir algún fallo o mejorar lo que se hizo.

Por ejemplo, un programa que simplemente guardara un valor "2" en la posición de memoria 1 de un ordenador sencillo, con una arquitectura propia de los años 80, basada en el procesador Z80 de 8 bits, sería así en código máquina:

```
0011 1110 0000 0010 0011 1010 0001 0000
```

Prácticamente ilegible. Por eso, en la práctica se emplean lenguajes más parecidos al lenguaje humano, llamados "lenguajes de **alto nivel**". Normalmente, estos son muy parecidos al idioma inglés, aunque siguen unas reglas mucho más estrictas.

Ejercicios propuestos

- **(1.1.1.1)** Hay fragmentos del anterior programa en código máquina que sí se podrían llegar a entender si los analizas con atención. Con la ayuda de alguna calculadora que sea capaz de convertir de números en el sistema binario a números en el sistema decimal (la propia calculadora de Windows puede ser suficiente), mira cual de los bloques de 4 cifras anteriores equivale al número "2" (el valor que se deseaba guardar) y cual equivale al número "1" (la posición de memoria en la que se deseaba guardar ese valor).

1.1.2. Lenguajes de alto nivel y de bajo nivel

Vamos a ver en primer lugar algún ejemplo de lenguaje de alto nivel, para después comparar con lenguajes de bajo nivel, que son los más cercanos al ordenador.

Uno de los lenguajes de **alto nivel** más sencillos es el lenguaje **BASIC**. En este lenguaje, escribir el texto Hola en pantalla, sería tan sencillo como usar la orden

```
PRINT "Hola"
```

Otros lenguajes, como **Pascal**, nos obligan a ser algo más estrictos y detallar ciertas cosas como el nombre del programa o dónde empieza y termina éste, pero, a cambio, hacen más fácil descubrir errores (ya veremos por qué):

```
program Saludo;
begin
    write('Hola');
end.
```

El equivalente en lenguaje **C** resulta algo más difícil de leer, porque los programas en C suelen necesitar incluir bibliotecas externas y devolver códigos de error al sistema operativo (incluso cuando todo ha ido bien):

```
#include <stdio.h>

int main()
{
    printf("Hola");
    return 0;
}
```

En **C#** hay que dar todavía más pasos para conseguir lo mismo, porque, como veremos, cada programa será lo que llamaremos "una clase":

```
class Saludo
{
    static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

Como se puede observar, a medida que los lenguajes evolucionan, los programas sencillos tienden a volverse más complicados, aunque a cambio los nuevos lenguajes son capaces de ayudar al programador en más tareas y de mayor complejidad. Por ejemplo, cuando se creó BASIC, en 1964, sus diseñadores no se

plantearon que un programa pudiera necesitar conectar a Internet, igual que no se consideraba que la facilidad para crear "interfaces gráficas de usuario" fuera algo relevante para un lenguaje cuando se diseñó C en 1972.

Afortunadamente, no todos los lenguajes siguen esta tendencia de avanzar hacia una mayor complejidad, y algunos se han diseñado de forma que las tareas simples sean (de nuevo) sencillas de programar. Por ejemplo, para escribir algo en pantalla usando el lenguaje **Python** haríamos:

```
print("Hola")
```

Y lo mismo ocurre en versiones recientes de C#, que permitirían simplificar nuestro programa hasta llegar a:

```
Console.WriteLine("Hola");
```

Por el contrario, los lenguajes de **bajo nivel** son más cercanos al ordenador que a los lenguajes humanos. Eso hace que sean más difíciles de aprender y también que los fallos sean más difíciles de descubrir y corregir, a cambio de que podemos optimizar al máximo la velocidad (si sabemos cómo), e incluso llegar a un nivel de control del ordenador que a veces no se puede alcanzar con otros lenguajes. Por ejemplo, escribir Hola en **lenguaje ensamblador** de un ordenador equipado con el sistema operativo MsDos y con un procesador de la familia Intel x86 sería algo como

```
dosseg
.model small
.stack 100h

.data
saludo db 'Hola',0dh,0ah,'$'

.code
main proc
    mov     ax,@data
    mov     ds,ax

    mov     ah,9
    mov     dx,offset saludo
    int     21h

    mov     ax,4C00h
    int     21h
main endp
end main
```

Resulta bastante más difícil de seguir. Pero eso todavía no es lo que el ordenador entiende, aunque existe una equivalencia casi directa. Lo que el ordenador realmente es capaz de comprender son secuencias de ceros y unos. Por ejemplo, las órdenes "mov ds, ax" y "mov ah, 9" (en cuyo significado no vamos a entrar) se convertirían en lo siguiente:

```
1000 0011 1101 1000 1011 0100 0000 1001
```

(Nota: los colores de los ejemplos anteriores son una ayuda que nos dan algunos entornos de programación, para que nos sea más fácil descubrir ciertos errores. Los colores dependerán del entorno de desarrollo utilizado, y frecuentemente serán personalizables por el usuario).

Ejercicios propuestos

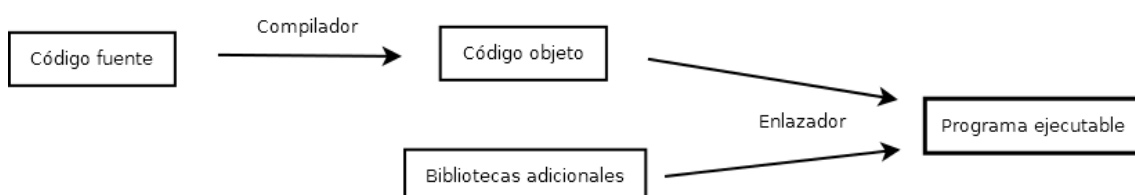
- **(1.1.2.1)** Analiza el programa que está escrito en lenguaje C#. ¿Qué cambio esperas que se deba realizar para escribir "Hasta luego" en vez de "Hola"?

1.1.3. Ensambladores, compiladores e intérpretes

Como hemos visto, las órdenes que nosotros hemos escrito (lo que se conoce como "programa **fuentes**") deben convertirse a lo que el ordenador comprende (obteniendo un "programa **ejecutable**").

Si elegimos un lenguaje de bajo nivel, como el ensamblador (en inglés *Assembly*, abreviado como *Asm*), la traducción es sencilla, y de hacer esa traducción se encargan unas herramientas llamadas **ensambladores** (en inglés *Assembler*).

Cuando el lenguaje que hemos empleado es de alto nivel, la traducción es más complicada, y a veces implicará varios pasos: generar un "código máquina" que todavía no será utilizable, quizá recopilar varios fuentes distintos y quizá incluir funcionalidades adicionales que se encuentran en otras bibliotecas que no hemos preparado nosotros. Las herramientas encargadas de realizar todo esto son los **compiladores**.



El programa ejecutable que se ha obtenido con un compilador o un ensamblador se podría hacer funcionar **en otro ordenador** similar al que habíamos utilizado para crearlo, sin necesidad de que ese otro ordenador tenga instalado el compilador o el ensamblador.

Por ejemplo, en el caso de Windows y del programa que nos saluda en lenguaje Pascal, tendríamos un fichero fuente llamado SALUDO.PAS. Este fichero inicialmente no serviría de nada en un ordenador que no tuviera un compilador de Pascal. En cambio, después de compilarlo obtendríamos un fichero SALUDO.EXE, capaz de funcionar en cualquier otro ordenador que tuviera el mismo sistema operativo (Windows, en este caso), aunque dicho ordenador no tenga un compilador de Pascal instalado. Eso sí, no funcionaría en otro ordenador que tuviera un sistema operativo distinto (por ejemplo, Linux o Mac OS X).

Un **intérprete** es otro tipo de traductor, una herramienta parecida a un compilador, con la diferencia de que en los intérpretes no se crea ningún "programa ejecutable" capaz de funcionar "por sí solo", de modo que si queremos distribuir nuestro programa a alguien, deberemos entregarle el programa fuente y también el intérprete que es capaz de entenderlo, o no le servirá de nada. Cuando ponemos el programa en funcionamiento, el intérprete se encarga de convertir el programa que hemos escrito en lenguaje de alto nivel a su equivalente en código máquina, orden por orden, justo en el momento en que hay que procesar cada una de las órdenes.

En principio, es esperable que un intérprete sea más sencillo que un compilador. Por eso eran muy frecuentes en los años 70 u 80, cuando la capacidad de proceso y la memoria disponible en los ordenadores eran muy limitadas.

Aun así, hoy en día los intérpretes siguen siendo muy habituales, especialmente en los entornos en los que es importante responder a las primeras órdenes cuanto antes, sin perder tiempo de analizar todo el programa de principio a fin. Por ejemplo, en un servidor web es habitual crear programas usando lenguajes como PHP, ASP o Python, y que estos programas no se conviertan a un ejecutable, sino que sean analizados y puestos en funcionamiento en el momento en el que se solicita la correspondiente página web.

Actualmente existe una alternativa más, algo que parece intermedio entre un compilador y un intérprete. Existen lenguajes que no se compilan para dar lugar a un ejecutable diseñado para un ordenador concreto, sino que se crea un ejecutable "genérico", que es capaz de funcionar en distintos tipos de

ordenadores, a condición de que en ese ordenador exista una "**máquina virtual**" capaz de entender esos ejecutables genéricos. Esta es la idea que se aplica en Java: los fuentes son ficheros de texto, con extensión ".java", que se compilan a ficheros ".class". Estos ficheros ".class" se podrían llevar a cualquier ordenador que tenga instalada una "máquina virtual Java" (y las hay para la mayoría de sistemas operativos). A cambio, esa "capa intermedia" que supone la máquina virtual se suele traducir en que los programas funcionen algo más despacio que su equivalente compilado para una plataforma específica.

Esta misma idea se sigue en el lenguaje C#, que se apoya en una máquina virtual llamada "Dot Net Framework" (algo así como "**plataforma punto net**"): los programas que creemos con herramientas como Visual Studio serán unos ejecutables que funcionarán en cualquier ordenador que tenga instalada dicha "plataforma .Net", algo que suele ocurrir en las versiones recientes de Windows y que se puede conseguir de forma un poco más artesanal en plataformas Linux y Mac, gracias a un "clon" de la "plataforma .Net" que es de libre distribución, conocido como "proyecto Mono".

Ejercicios propuestos

- **(1.1.3.1)** Localiza en Internet el intérprete de BASIC llamado Bywater Basic, en su versión para el sistema operativo que estés utilizando y prueba el primer programa de ejemplo que se ha visto en el apartado 0.1. También puedes usar cualquier "ordenador clásico" (de principios de los años 80) y otros muchos BASIC modernos, como Basic256. **(Nota:** no es necesario realizar este ejercicio para seguir adelante con el curso; puedes omitirlo si no encuentras ningún intérprete de BASIC).
- **(1.1.3.2)** Localiza en Internet el compilador de Pascal llamado Free Pascal, en su versión para el sistema operativo que estés utilizando, instálalo y prueba el segundo programa de ejemplo que se ha visto en el apartado 0.1. **(Nota:** no es necesario realizar este ejercicio para seguir adelante con el curso; puedes omitirlo si no quieres instalar en tu ordenador software que luego no vayas a seguir utilizando).
- **(1.1.3.3)** Localiza un compilador de C para el sistema operativo que estés utilizando (si es Linux o alguna otra versión de Unix, es fácil que se encuentre ya instalado) y prueba el tercer programa de ejemplo que se ha visto en el apartado 0.1. **(Nota:** no es necesario realizar este ejercicio para

seguir adelante con el curso; puedes omitirlo si no quieres instalar en tu equipo software que después quizá no vuelvas a utilizar).

- **(1.1.3.4)** Descarga un intérprete de Python (ya estará preinstalado si usas Linux) o busca en Internet "try Python web" para probarlo desde tu navegador web, y prueba el quinto programa de ejemplo que se ha visto en el apartado 0.1. (**Nota:** nuevamente, no es necesario realizar este ejercicio para seguir adelante con el curso).

1.1.4. Pseudocódigo y algoritmo

A pesar de que los lenguajes de alto nivel se acercan al lenguaje natural (inglés), es habitual no usar inicialmente ningún lenguaje de programación concreto cuando queremos plantear los pasos necesarios para resolver un problema, sino emplear un lenguaje de programación ficticio, no tan estricto, muchas veces incluso en español. Este lenguaje recibe el nombre de **pseudocódigo**.

La secuencia de pasos para resolver un problema es lo que se conoce como **algoritmo**. Realmente es algo un poco más estricto que eso: por ejemplo, un algoritmo debe estar formado por un número finito de pasos y poderse resolver en un tiempo finito.

Por tanto, un **programa** de ordenador es un algoritmo expresado usando un lenguaje de programación.

Por ejemplo, un algoritmo que controlase los pagos que se realizan en una tienda con tarjeta de crédito, escrito en pseudocódigo, podría ser:

```

Leer banda magnética de la tarjeta
Conectar con central de cobros
Si hay conexión y la tarjeta es correcta:
    Pedir código PIN
    Si el PIN es correcto
        Comprobar saldo_existente
        Si saldo_existente >= importe_compra
            Aceptar la venta
            Descontar importe del saldo
        En caso contrario
            Avisar de saldo insuficiente
        Fin Si
    En caso contrario
        Avisar de PIN incorrecto
    Fin Si
En caso contrario
    Avisar de fallo de conexión
Fin Si

```

Como se ve en este ejemplo, el pseudocódigo suele ser menos detallado que un lenguaje de programación "real" y expresar las acciones de forma más general, buscando concretar las ideas más que la forma real de llevarlas a cabo. Por ejemplo, ese "conectar con central de cobros" correspondería a varias órdenes individuales en cualquier lenguaje de programación.

Ejercicios propuestos

- **(1.1.4.1)** ¿Qué esperas que escriba en pantalla el siguiente pseudocódigo? Localiza en Internet el intérprete de Pseudocódigo llamado PseInt y prueba a escribirlo y lanzarlo. (**Nota:** no es necesario realizar este ejercicio para seguir adelante con el curso; puedes omitirlo si no te apetece instalar en tu equipo software que luego no vayas a seguir utilizando).

```
Proceso EjemploDeSuma
    Escribir 2+3
FinProceso
```

1.2. ¿Qué es C #? ¿Qué entorno usaremos?

C# es un lenguaje de programación de ordenadores. Se trata de un lenguaje moderno, evolucionado a partir de C y C++, y con una sintaxis muy similar a la de Java. Los programas creados con C# no suelen ser tan rápidos como los creados con C, pero a cambio la productividad del programador es mucho mayor y es más difícil cometer errores.

Se trata de un lenguaje creado por Microsoft cerca del año 2000, para realizar programas para su plataforma .NET, pero fue estandarizado posteriormente por ECMA y por ISO, y existe una implementación alternativa de "código abierto", llamada "proyecto Mono", que está disponible para Windows, Linux, Mac OS X y otros sistemas operativos.

Nosotros comenzaremos por usar en los primeros temas el compilador que incorpora la propia plataforma .Net, junto con un editor de texto para programadores. Cuando los conceptos básicos estén asentados, pasaremos a emplear Visual Studio, de Microsoft, que requiere un ordenador más potente pero a cambio incluye un entorno de desarrollo muy avanzado, y está disponible también en una versión gratuita (Visual Studio Community Edition). Existen otros entornos integrados alternativos, como SharpDevelop o MonoDevelop, que también comentaremos.

Los **pasos** que seguiremos para crear un programa en C# serán:

- Escribir el programa en lenguaje C# (**fichero fuente**), con cualquier editor de textos.
- Compilarlo, con nuestro compilador. Esto creará un "**fichero ejecutable**".
- Lanzar el fichero ejecutable.

La mayoría de los compiladores actuales permiten dar todos estos pasos desde un único **entorno**, en el que escribimos nuestros programas, los compilamos, y los depuramos en caso de que exista algún fallo.

Tras el siguiente apartado veremos un ejemplo de entorno desde el que realizar nuestros programas, dónde localizarlo y cómo instalarlo.

Ejercicios propuestos

- **(1.2.1)** Investiga en qué año se crearon los lenguajes C, C++, Java y C#.

1.3. Escribir un texto en C#

Vamos con un primer ejemplo de programa en C#, posiblemente el más sencillo de los que "hacen algo útil". Se trata de escribir un texto en pantalla. La apariencia de este programa la vimos en el tema anterior. Vamos a analizarlo ahora con más detalle:

```
class Ejemplo_01_03a
{
    static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

Esto escribe "Hola" en la pantalla. Pero hay muchas "cosas raras" alrededor de ese "Hola", de modo vamos a comentarlas antes de proseguir, aunque muchos de los detalles los aplazaremos para más adelante. En este primer análisis, iremos desde dentro hacia fuera:

- `WriteLine("Hola");` : "Hola" es el texto que queremos escribir, y `WriteLine` es la orden encargada de escribir (Write) una línea (Line) de texto en pantalla.
- `Console.WriteLine("Hola");` : `WriteLine` siempre irá precedido de "Console." porque es una orden de manejo de la "consola" (la "pantalla negra" en modo texto del sistema operativo).

- `System.Console.WriteLine("Hola");` : Las órdenes relacionadas con el manejo de consola (Console) pertenecen a la categoría de órdenes del sistema (System).
- Las llaves { y } se usan para delimitar un bloque de programa. En nuestro caso, se trata del bloque principal del programa (Main).
- `static void Main()` : Main indica cual es "el cuerpo del programa", la parte principal (porque un programa puede estar dividido en varios fragmentos, como veremos más adelante). Todos los programas deben tener un bloque "Main". Los detalles de por qué hay que poner delante "static void" y de por qué se añade después un paréntesis vacío los iremos aclarando más tarde. De momento, deberemos memorizar que ésa será la forma habitual (aunque no la única) de escribir "Main".
- `class Ejemplo_01_03a` : De momento pensaremos que "Ejemplo_01_03a" es el nombre de nuestro programa. Una línea como esa deberá existir también siempre en nuestros programas (aunque el nombre no tiene por qué ser tan "rebuscado"), y eso de "class" será obligatorio. Nuevamente, aplazamos para más tarde los detalles sobre qué quiere decir "class".

Nota: en ocasiones verás programas en los que aparece la palabra "**public**" antes de "class" y antes de "static void Main". Ya veremos más adelante por qué, pero uno de esos "public" no es necesario en programas tan sencillos y el otro está incluso desaconsejado por la propia Microsoft en sus últimas recomendaciones. Aun así, será aceptable que aparezcan ambas palabras "public" (o una de ellas), como ocurre en esta variante del ejemplo anterior:

```
public class Ejemplo_01_03a2
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

Como se puede ver, mucha parte de este programa todavía es casi un "acto de fe" para nosotros. Debemos creernos que "se debe hacer así". Poco a poco iremos detallando el por qué de "static", de "void", de "class"... Por ahora nos limitaremos

a utilizar un esqueleto como ese y "rellenar" el cuerpo del programa para entender los conceptos básicos de programación.

Ejercicio propuesto (1.3.1): Crea un programa en C# que te salude por tu nombre (por ejemplo, "Hola, Nacho").

Sólo un par de detalles más antes de seguir adelante:

- Cada orden de C# debe terminar con un **punto y coma (;)**
- C# es un lenguaje de **formato libre**, de modo que puede haber varias órdenes en una misma línea, u órdenes separadas por varias líneas o espacios entre medias. Lo que realmente indica donde termina una orden y donde empieza la siguiente son los puntos y coma y las llaves. Por ese motivo, el programa anterior se podría haber escrito también así (aunque no es aconsejable, porque puede resultar menos legible):

```
class Ejemplo_01_03b {
static
void Main() { System.Console.WriteLine("Hola"); } }
```

De hecho, hay dos formas especialmente frecuentes de colocar la llave de comienzo, y yo usaré ambas indistintamente. Una es como hemos hecho en el primer ejemplo: situar la llave de apertura en una línea, sola, y justo encima de la llave de cierre correspondiente. Esto es lo que muchos autores llaman el "estilo C". La segunda forma habitual es situándola a continuación del nombre del bloque que comienza (el "estilo Java"), así:

```
class Ejemplo_01_03c {
    static void Main(){
        System.Console.WriteLine("Hola");
    }
}
```

(esta es la forma que se empleará preferentemente en este texto en el caso de que estemos trabajando con fuentes de mayor tamaño, para que ocupe un poco menos de espacio; en la mayoría de programas usaremos el "estilo C", que tiende a resultar más legible).

La gran mayoría de las órdenes que encontraremos en el lenguaje C# son palabras en inglés o abreviaturas de éstas, pero hay que tener en cuenta que C# **distingue**

entre mayúsculas y minúsculas, por lo que "WriteLine" es una palabra reconocida, pero "writeLine", "WRITELINE" o "Writeline" no lo son.

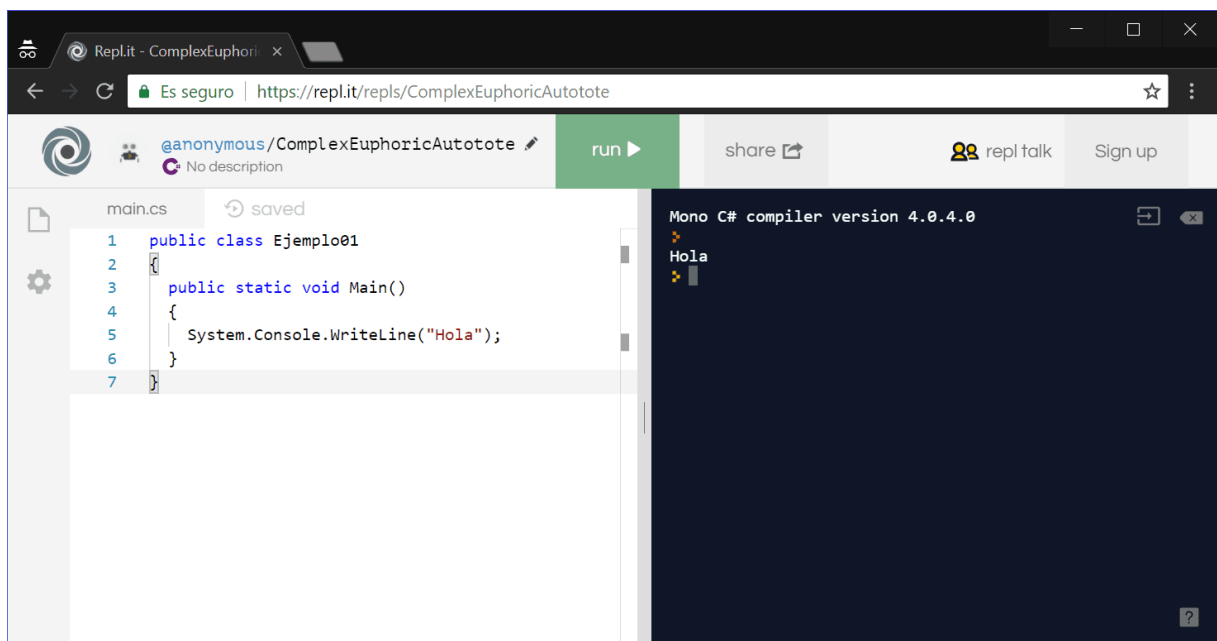
1.4. Cómo probar este programa

Para probar este programa será necesario tener un editor de texto y un compilador de C#, o bien un entorno desarrollo integrado, que incluya ambos. Vamos a ver los pasos requeridos para instalar y usar ambas herramientas, tanto en Linux como en Windows.

1.4.1. Cómo probarlo con un compilador online

Cada vez son más frecuentes sitios web que incluyen "compiladores online", en los que podemos teclear nuestro programa y probarlo, sin necesidad de instalar nada en nuestro equipo ni de más herramientas que un navegador web.

Uno de ellos es "repl.it", que no necesita registro y muestra un panel izquierdo en el que podemos teclear nuestro programa y un panel derecho en el que, si pulsamos el botón "run", podemos ver su resultado:



Esta forma de trabajar tendrá dos problemas:

- No será adecuada para proyectos grandes, que estén formados por varios ficheros.

- Por lo general, no será posible "depurar el programa" (avanzar paso a paso, ver valores de variables, y una serie de operaciones avanzadas que veremos más adelante).

Un tercer problema que encontraremos en muchos entornos online es que no se podrán usar con comodidad para programas interactivos, que respondan a datos introducidos por el usuario. Algunos de ellos, como el de "repl.it", sí permitirán introducir datos de forma sencilla.

Ejercicio propuesto (1.4.1.1): Usa el compilador online de "repl.it" (o algún otro sitio web similar) para crear y probar un programa en C# que escriba en pantalla "Bienvenido a C#".

1.4.2. Cómo probarlo con Mono en Linux

Para alguien acostumbrado a sistemas como Windows o Mac OS, hablar de Linux puede sonar a que se trata de algo apto sólo para expertos. Eso no necesariamente es así, y, de hecho, para un aprendiz de programador puede resultar justo al contrario, porque Linux tiene compiladores e intérpretes de varios lenguajes ya preinstalados, y otros son fáciles de instalar en unos pocos clics.

La instalación de Linux, que podría tener la dificultad de crear particiones para coexistir con nuestro sistema operativo habitual, hoy en día puede realizarse de forma simple, usando software de virtualización gratuito, como VirtualBox, que permite tener un "**ordenador virtual**" dentro del nuestro, e instalar Linux en ese "ordenador virtual" sin interferir con nuestro sistema operativo habitual.

Así, podemos instalar VirtualBox, descargar la imagen ISO del CD o DVD de instalación de algún Linux que sea reciente y razonablemente amigable, arrancar VirtualBox, crear una nueva máquina virtual y "cargar esa imagen de CD" para instalar Linux en esa máquina virtual.

En este caso, yo comentaré los pasos necesarios para usar Linux Mint como entorno de desarrollo (en su versión 17 Cinnamon, pero los cambios deberían ser mínimos para versiones posteriores):

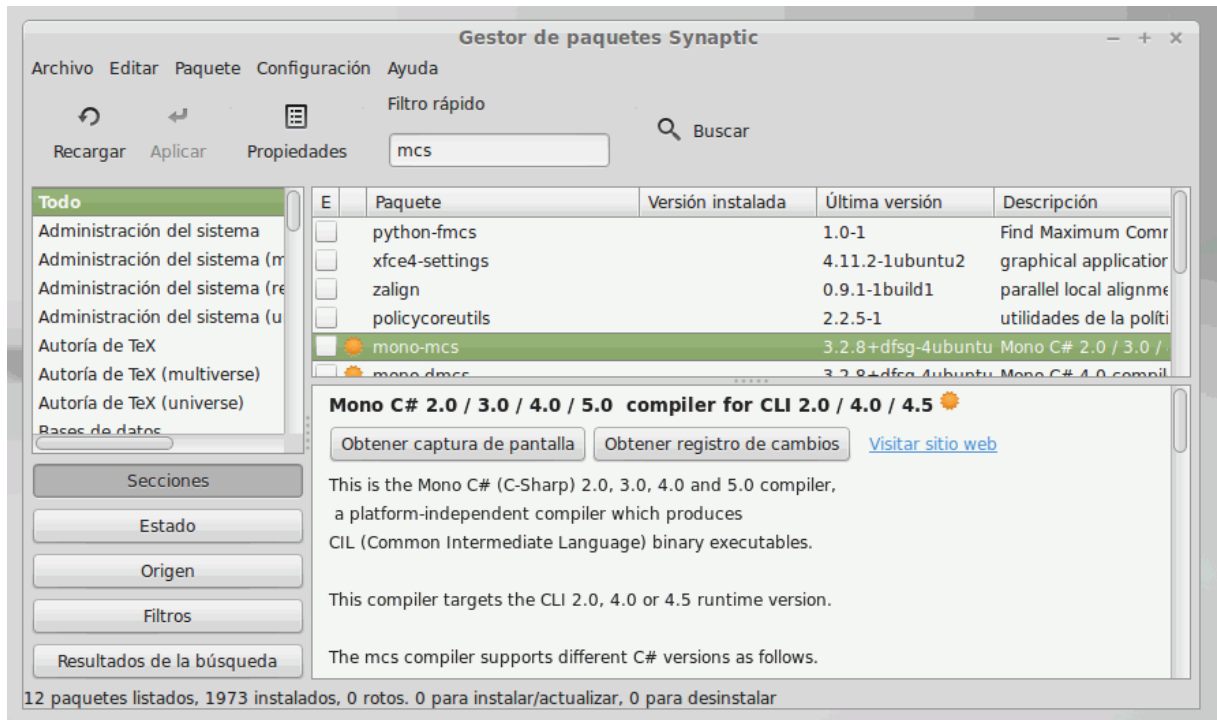


En primer lugar, deberemos entrar al instalador de software de nuestro sistema, que para las versiones de Linux Mint basadas en escritorios derivados de Gnome (como es el caso de Mint 17 Cinnamon), suele ser un tal "Gestor de paquetes Synaptic":

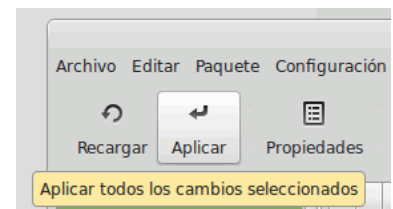


Se nos pedirá nuestra contraseña de usuario (la que hayamos utilizado en el momento de instalar Linux), y aparecerá la pantalla principal de Synaptic, con una enorme lista de software que podríamos instalar. En esta lista, aparece una casilla

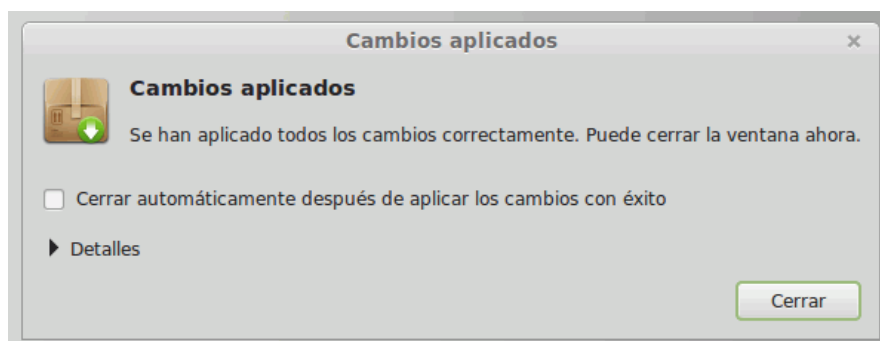
de texto llamada "Filtro rápido", en la que podemos teclear "mcs" para que nos aparezca directamente nuestro compilador Mono:



Entre otros paquetes, posiblemente veremos uno llamado "mono-mcs", en cuya descripción se nos dirá que es el "Mono C# Compiler". Al hacer doble clic se nos avisará en el caso (habitual) de que sea necesario instalar algún otro paquete adicional y entonces ya podremos pulsar el botón "Aplicar":



Se descargarán los ficheros necesarios, se instalarán y al cabo de un instante se nos avisará de que se han aplicado todos los cambios:



Ya tenemos instalado el compilador, que es la herramienta que convertirá nuestros programas en algo que el ordenador realmente entienda. Para teclear los programas necesitaremos un editor de texto, pero eso es algo que viene

preinstalado en cualquier Linux. Por ejemplo, en esta versión de Linux encontraremos un editor de textos llamado "gedit" dentro del apartado de accesorios:



En este editor podemos teclear nuestro programa, que inicialmente se verá con letras negras sobre fondo blanco:

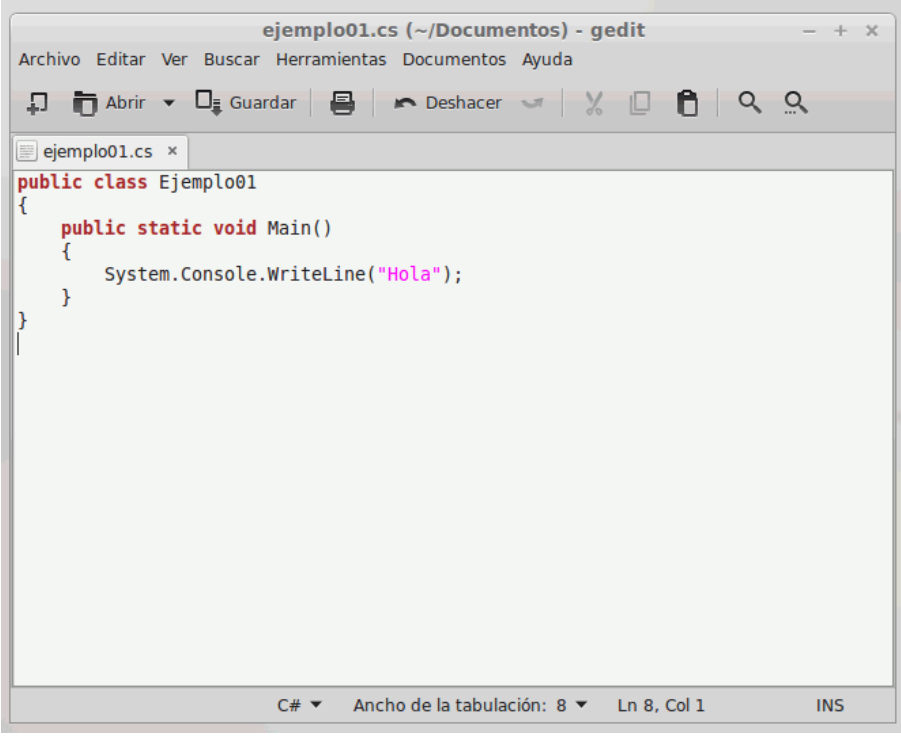


The screenshot shows the gedit text editor window titled "*Documento sin título 1 - gedit". The menu bar includes Archivo, Editar, Ver, Buscar, Herramientas, Documentos, and Ayuda. The toolbar contains icons for opening, saving, printing, undo, redo, cut, copy, paste, and search. The editor area contains the following C# code:

```
public class Ejemplo01
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

The status bar at the bottom indicates "Texto plano", "Ancho de la tabulación: 8", "Ln 8, Col 1", and "INS".

Cuando lo guardemos con un nombre terminado en ".cs" (como "ejemplo01.cs"), el editor sabrá que se trata de un fuente en lenguaje C# y nos mostrará cada palabra en un color que nos ayude a saber la misión de esa palabra:

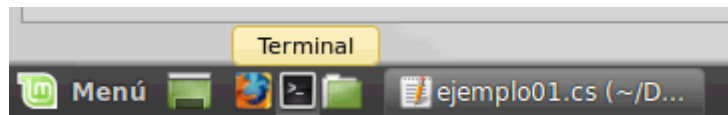


The screenshot shows the gedit text editor window titled "ejemplo01.cs (~/Documentos) - gedit". The menu bar and toolbar are the same as in the previous image. The editor area shows the same C# code, but with syntax highlighting:

```
public class Ejemplo01
{
    public static void Main()
    {
        System.Console.WriteLine("Hola");
    }
}
```

The status bar at the bottom indicates "C#", "Ancho de la tabulación: 8", "Ln 8, Col 1", and "INS".

Para compilar y lanzar el programa usaremos un "terminal", que habitualmente estará accesible en la parte inferior de la pantalla:



En esa "pantalla negra" ya podemos teclear las órdenes necesarias para compilar y probar el programa:

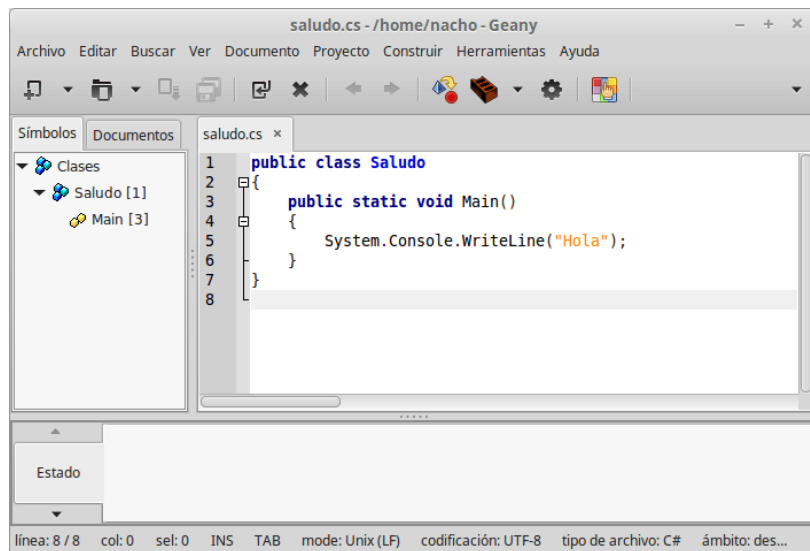
- Si hemos guardado el fuente en la carpeta "Documentos", el primer paso será entrar a esa carpeta con la orden "cd Documentos".
- Después lanzaremos el compilador con la orden "mcs" seguida del nombre del fuente: "mcs ejemplo01.cs" (recuerda que en Linux debes respetar las mayúsculas y minúsculas tal y como las hayas escrito en el nombre del fichero).
- Si no aparece ningún mensaje de error, ya podemos lanzar el programa ejecutable, con la orden "mono" seguida del nombre del programa (terminado en ".exe"): "mono ejemplo01.exe", así:

```
Terminal
nacho@nacho-mint17 ~ $ cd Documentos/
nacho@nacho-mint17 ~/Documentos $ mcs ejemplo01.cs
nacho@nacho-mint17 ~/Documentos $ mono ejemplo01.exe
Hola
nacho@nacho-mint17 ~/Documentos $
```

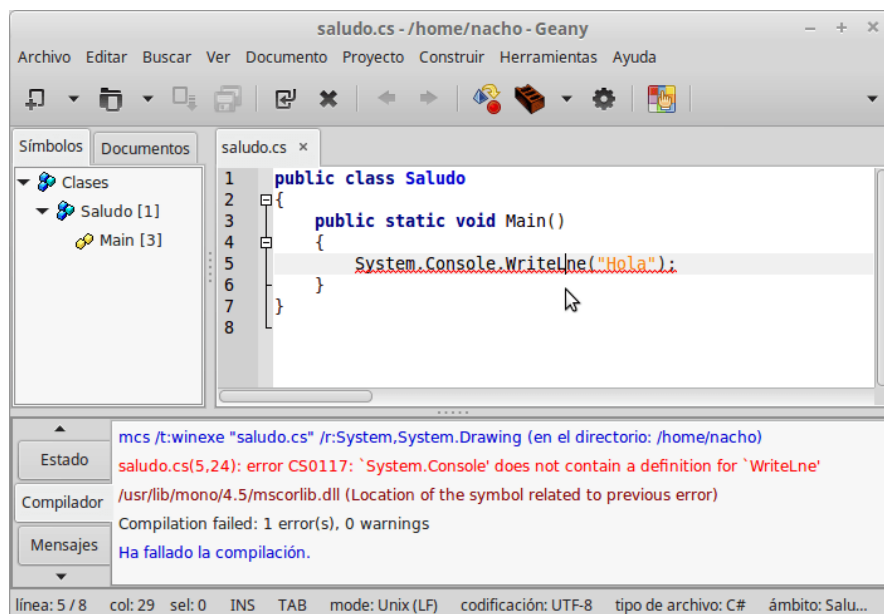
Si obtienes algún mensaje de error, tendrás que comprobar si has dado los pasos anteriores de forma correcta y si tu fuente está bien tecleado.

Ejercicio propuesto (1.4.2.1): Si vas a utilizar Linux, instala el software de desarrollo y después crea y prueba un programa en C# que escriba en pantalla "Bienvenido a Linux".

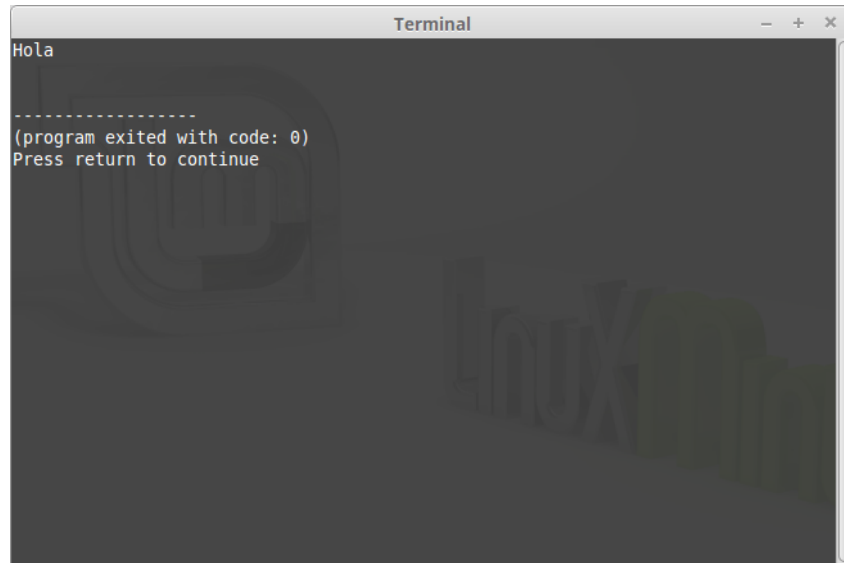
Una alternativa aún más cómoda es no abrir una consola para compilar el fuente y probar el resultado, sino utilizar un editor un poco más avanzado, que permita realizar ambas tareas desde el propio editor. Uno de los que lo incluye estas posibilidades (y que además es muy ligero) es **Geany**, que no viene preinstalado en la mayoría de distribuciones de Linux, pero se puede instalar en pocos segundos empleando Synaptic o el gestor de paquetes de nuestro sistema.



En Geany encontrarás un botón "Compilar", que lanza el compilador por ti, e incluso destaca en color rojo las líneas que contengan algún error:



y también tiene un botón "Ejecutar", que lanza el programa y espera a que se pulse una tecla antes de volver al editor:



Ejercicio propuesto (1.4.2.2): Si vas a utilizar Linux, instala también Geany y crea y prueba un programa en C# que escriba en pantalla "Bienvenido a Linux y a Geany".

1.4.3. Windows y equipos potentes: Visual Studio

Si usas Windows y tienes un equipo reciente (al menos 4 GB de memoria RAM -recomendable 8 GB o más- y al menos 2.0 GHz de velocidad de procesador de 64 bits -recomendable quad core o más-), la alternativa más profesional es utilizar Visual Studio, de Microsoft. La versión "Community" de Visual Studio es gratuita para educación y para desarrolladores individuales:

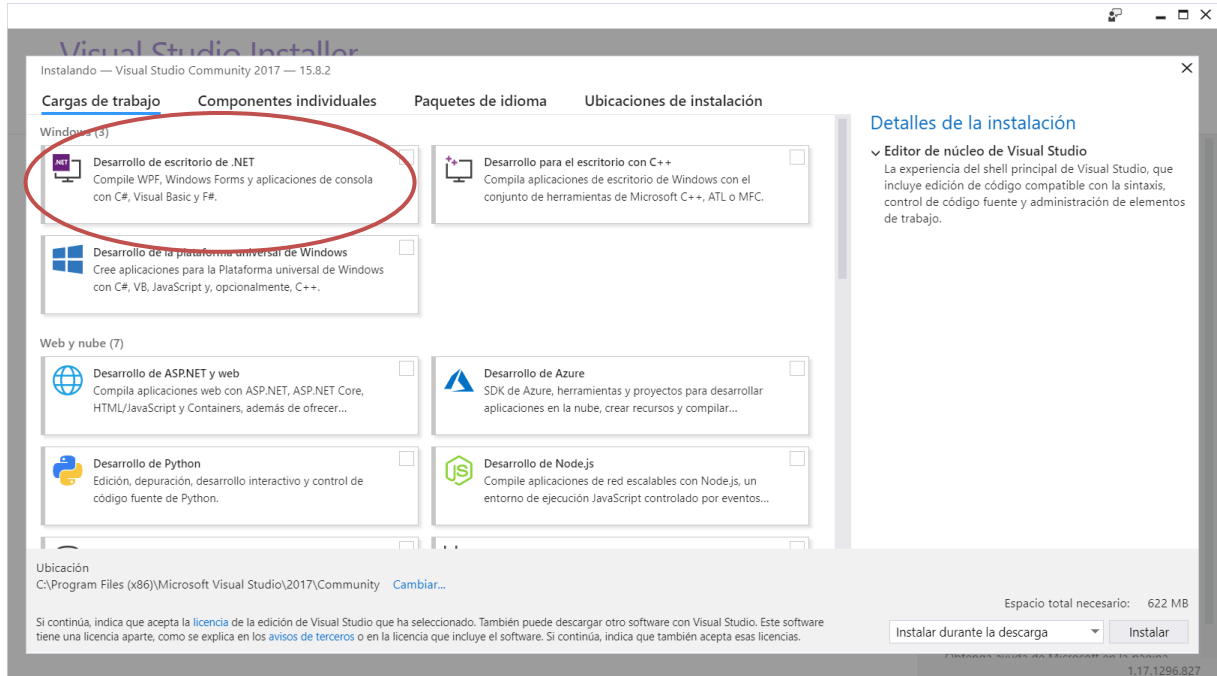
<https://visualstudio.microsoft.com/es/vs/community/>

(Si tu centro de estudios es parte del programa Microsoft para estudiantes, quizá eso te dé acceso gratuito a las versiones Professional y/o Enterprise, pero éstas consumen muchos más recursos y no aportan gran cosa para un principiante, por lo que en principio son menos recomendables para seguir este texto).

En versiones anteriores a la 2017, era posible descargar una imagen ISO de un DVD, para poder instalar Visual Studio en equipos en los que la conexión de Internet sea lenta o tenga un límite de datos. En la versión 2017 y posteriores, por el contrario, sólo existe un instalador online¹, pero, a cambio, éste permite descargar sólo las herramientas que realmente vayamos a utilizar.

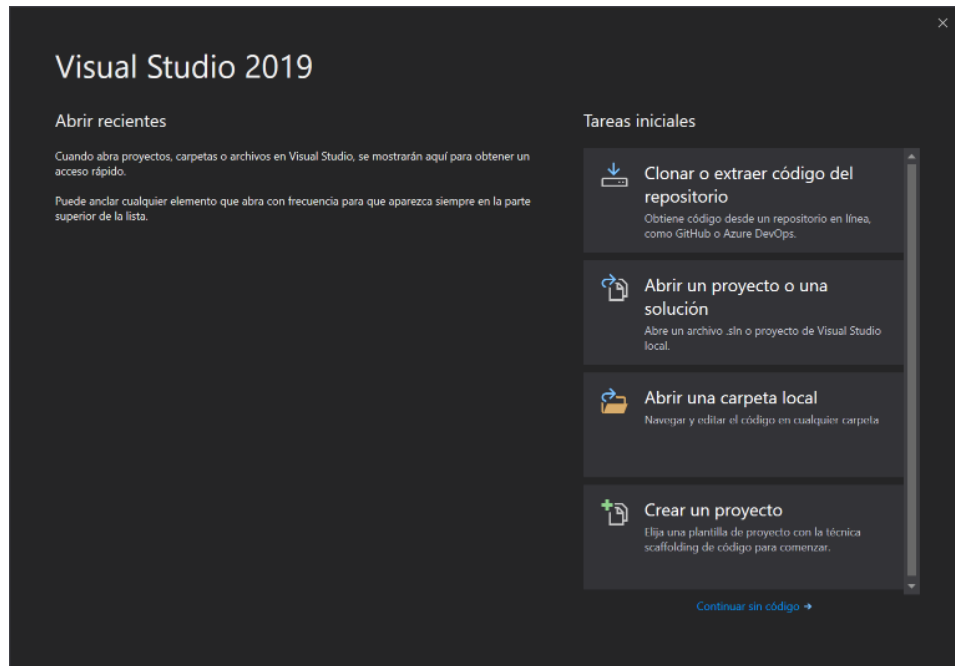
¹ Realmente, es posible crear un "instalador offline" para Visual Studio 2017 y posteriores, descargando los ficheros necesarios mediante el instalador predeterminado, y luego se podrían llevar todos esos ficheros a

El primer paso de la instalación será escoger qué herramientas deseamos instalar. Para el propósito de este texto nos bastaría con la primera opción, "Desarrollo de escritorio .Net":

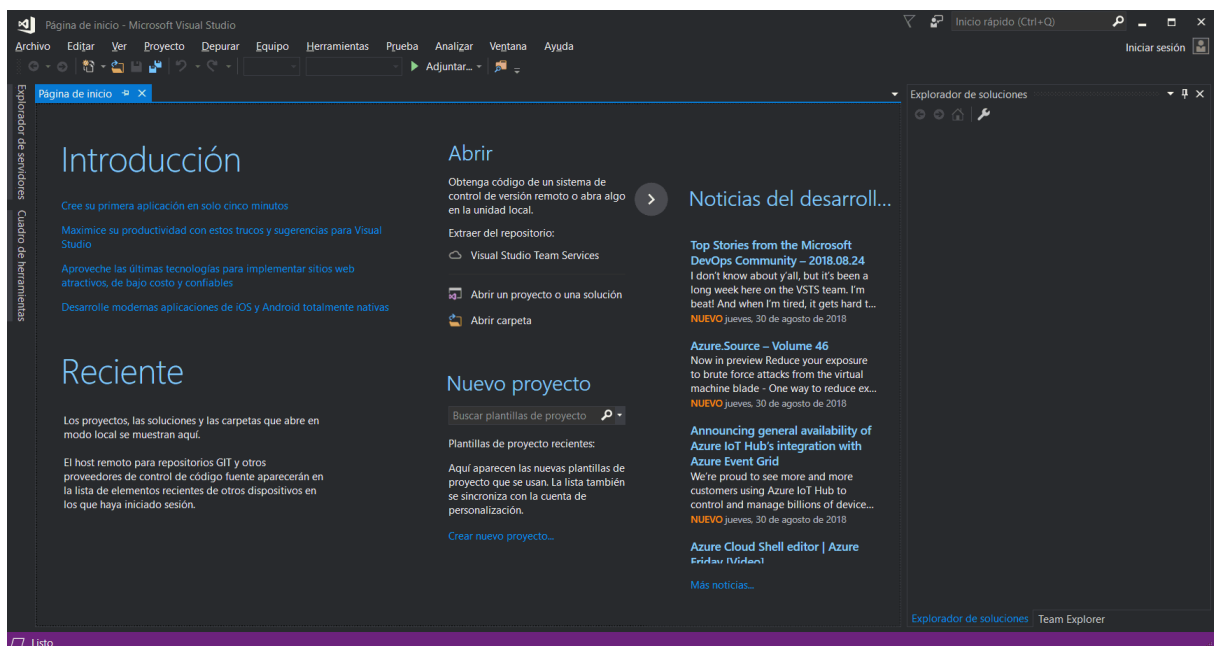


Cuando lo instales, verás una pantalla como ésta (la combinación de colores puede ser distinta en tu caso):

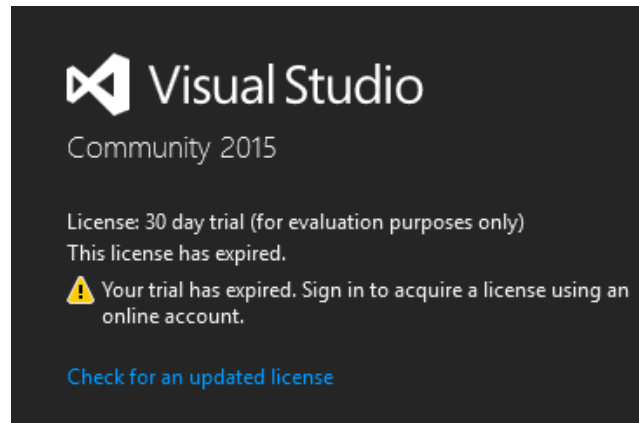
cualquier otro equipo, aunque no tuviera conexión a Internet. En cualquier caso, se trata de un proceso relativamente engorroso, por lo que no entraremos en más detalle.



O, en versiones más antiguas, como Visual Studio 2017, quizá sea como ésta:

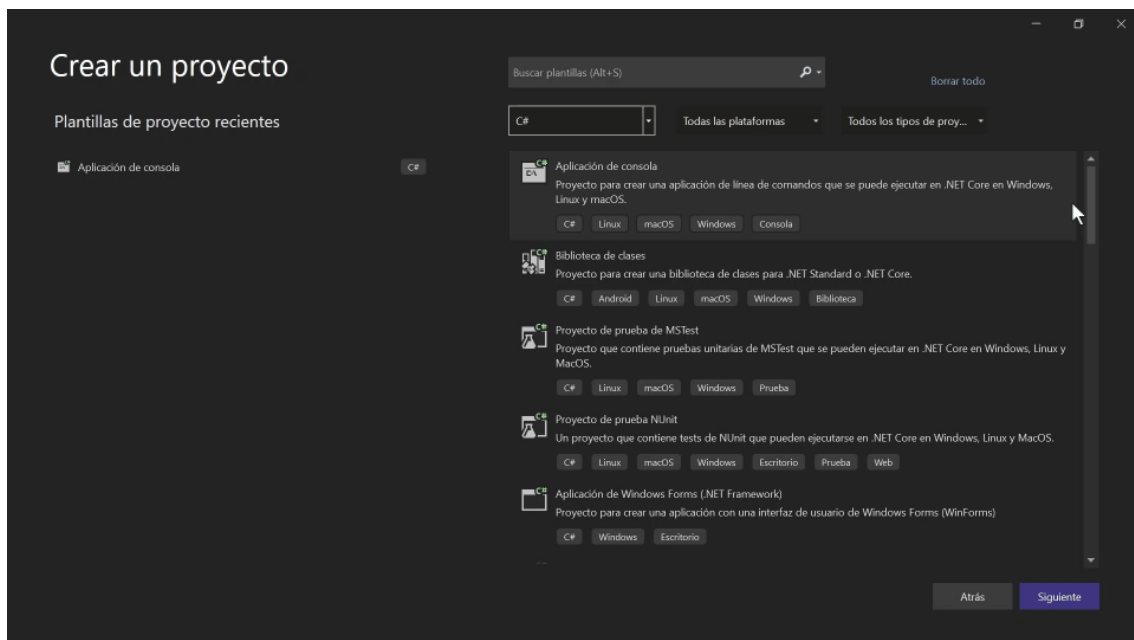


Cuidado: es probable que tengas un **mes de prueba** inicial. A partir de ese momento, Visual Studio Community sigue siendo gratis, pero quizá tengas que registrar tu copia para que siga activada:



Para crear un programa, deberás comenzar por crear un proyecto, bien desde la correspondiente opción de la pantalla principal, o bien desde Archivo, en la opción "Nuevo / Proyecto".

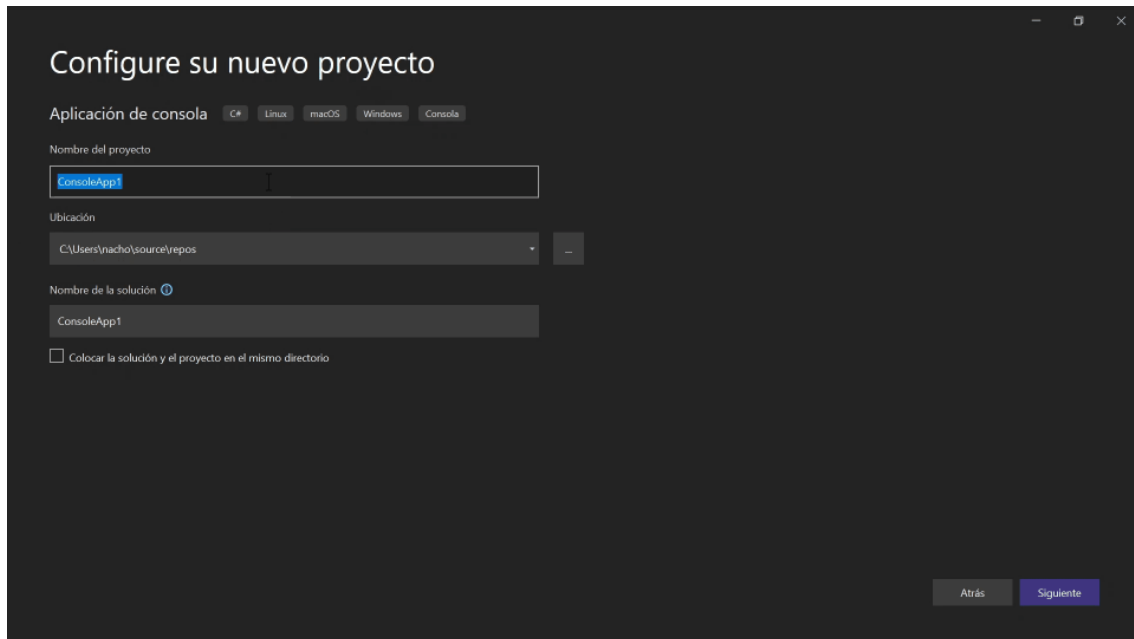
Entre los tipos de proyectos disponibles, los que deberás comenzar realizando como aprendiz de programación serán "Aplicación de consola":



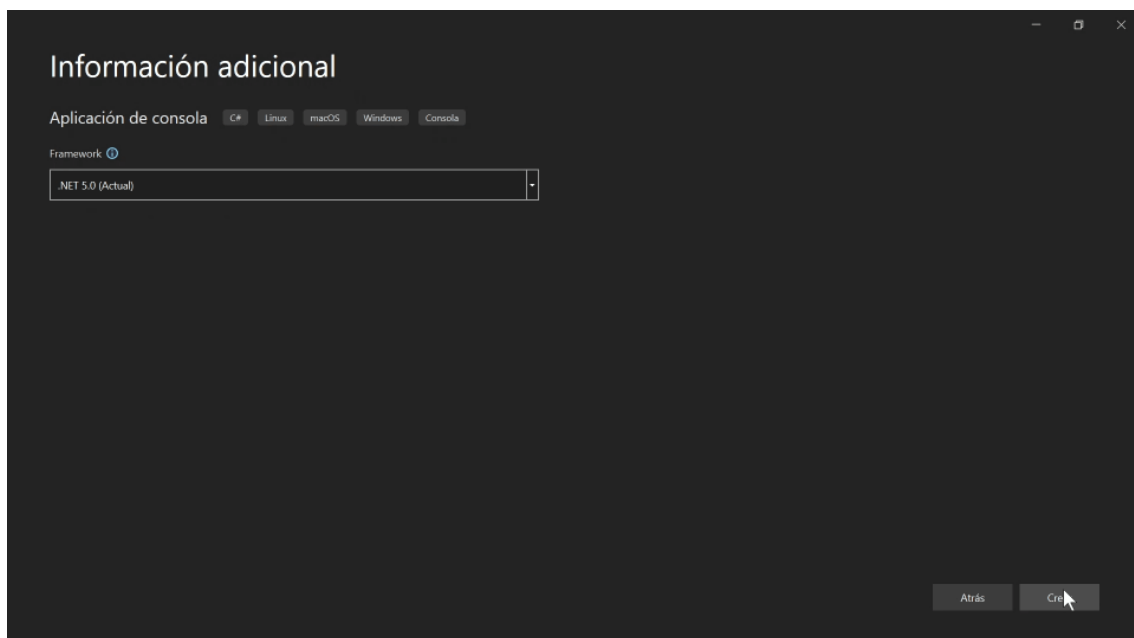
Se te propondrá el nombre ConsoleApp1, que deberás cambiar por otro nombre que indique la misión del programa que estás creando (en este ejemplo, "Saludo").

También es importante que te fijes en la casilla "ubicación", que te indica en qué carpeta de tu ordenador se va a guardar tu programa. Necesitarás conocer esa carpeta si se trata de un programa que debas entregar.

En el caso de **Visual Studio 2022**, en un primer paso se te pedirá el nombre y la ubicación:

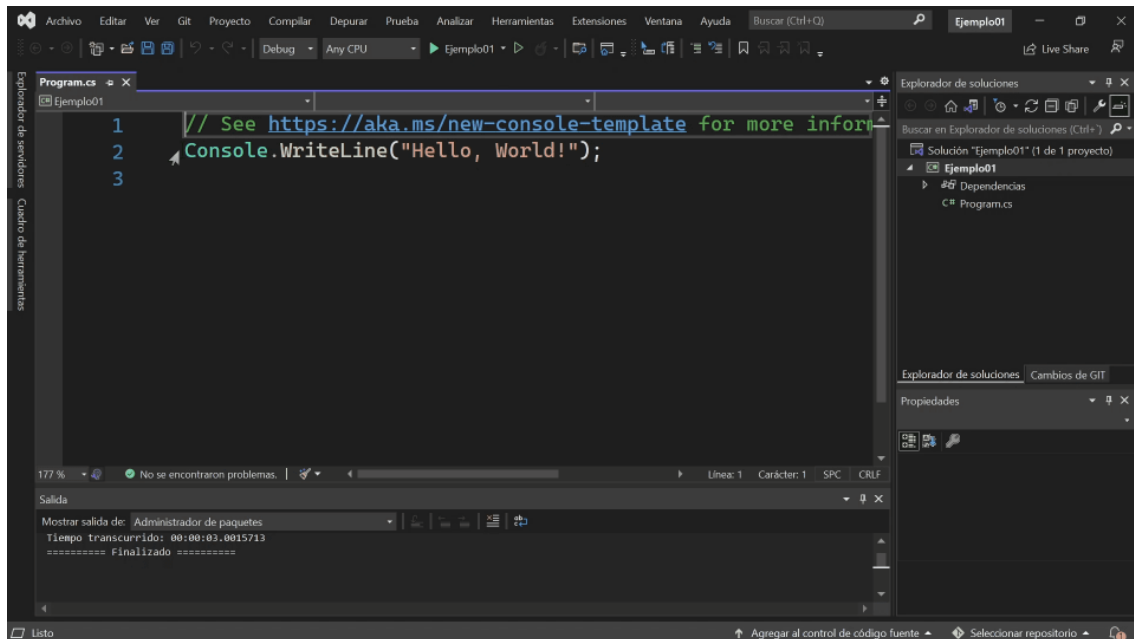


Y en un segundo paso se te preguntará la plataforma de destino. Por motivos que veremos enseguida, será recomendable escoger la plataforma ".Net 5.0" o una versión anterior:

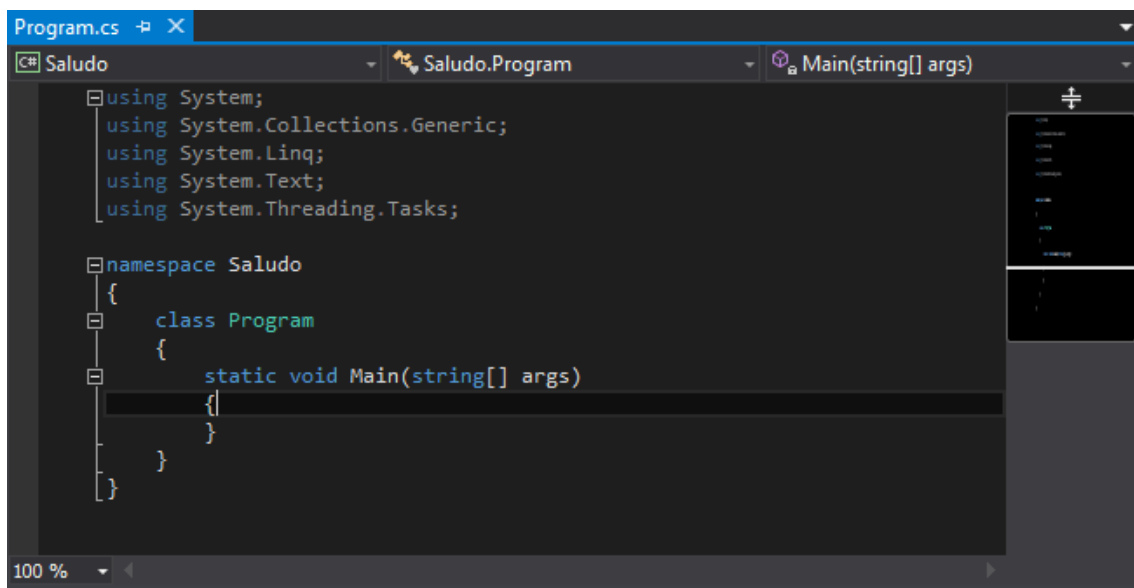


El problema de escoger una versión más moderna (la 6.0, en este momento) es que el programa estará más simplificado, llegando casi al nivel de Python, lo que

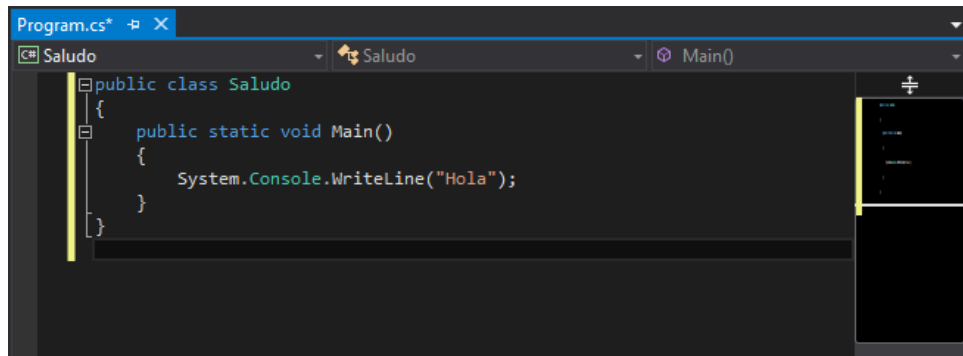
permitiría esquivar ciertos detalles que un programador "de carrera" debe conocer, así que trabajaremos con un esqueleto de programa "más convencional".



Si has escogido la versión 5.0 o anterior, aparecerá un esqueleto de programa en el que "**sobran cosas**": varias líneas "**using**", que habría que borrar (en un programa normal sólo conservaremos una, como veremos dentro de poco), así como un "**namespace**", que ayuda a evitar colisiones de nombres en proyectos de gran tamaño pero que será innecesario en programas tan sencillos:

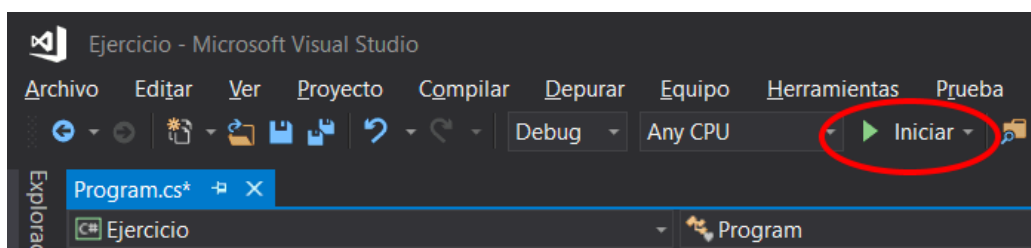


La apariencia real de tu programa finalizado debería ser algo como:

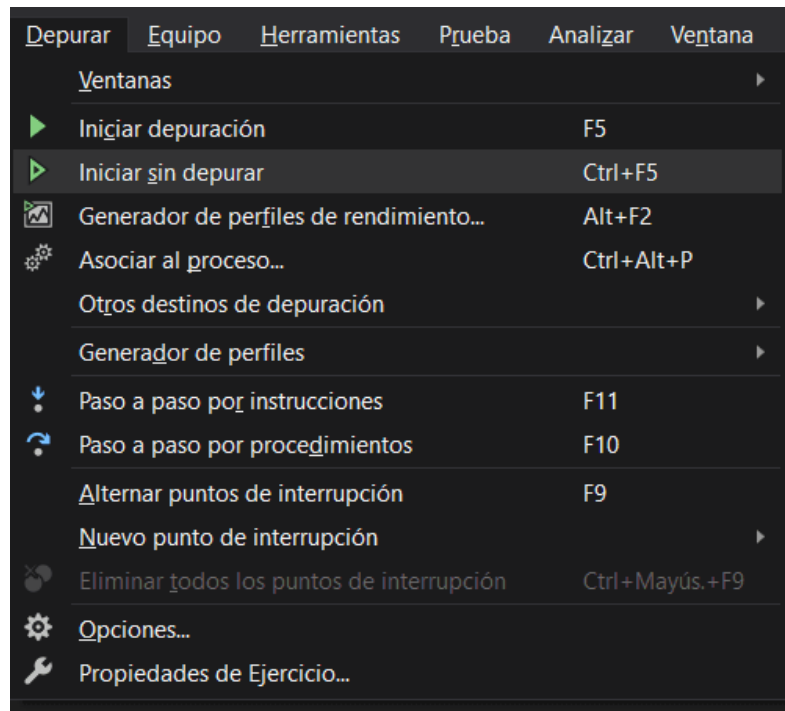


(tras eliminar el "namespace", si tu programa queda demasiado a la derecha, puedes desplazar todo el cuerpo del programa un poco más a la izquierda, seleccionándolo y pulsando Mays+Tab).

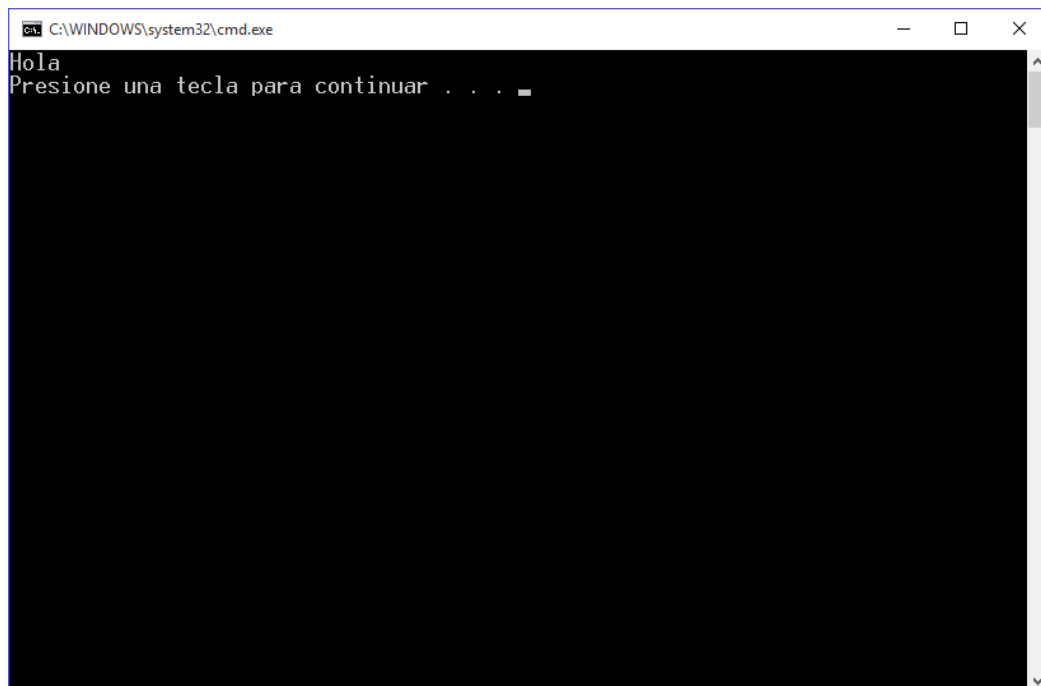
Para **lanzar** un programa, puedes usar el botón "Iniciar", o pulsar la tecla F5, o la correspondiente opción del menú Depurar:



Pero esa opción tiene un problema en los programas tan sencillos y no interactivos: el programa se lanza y **la ventana de ejecución se cierra** inmediatamente, por lo que probablemente no tendrás tiempo de comprobar los resultados. La alternativa es pulsar **Ctrl+F5** o usar la opción "Iniciar sin depurar" del menú Depurar:



De modo que se podrá comprobar el resultado, que sería:



Otra **alternativa** (poco elegante y que deberías intentar evitar), si usas una versión de Visual Studio antigua, que no permita pulsar Ctrl+F5 para hacer una pausa al final de la ejecución, es añadir una orden "ReadLine", de modo que sea tu propio programa el que se encargue de esperar a que el usuario pulse Intro:

```

class SaludoPausa
{
    static void Main()
    {
        System.Console.WriteLine("Hola");
        System.Console.ReadLine();
    }
}

```

Eso sí, esa línea "ReadLine" no debería ser parte de un programa definitivo, y **deberías borrarla** antes de entregar un ejercicio que la contenga.

Visual Studio es muy cómodo para el trabajo diario en proyectos grandes, especialmente por detalles que veremos más adelante, como el autocompletado y la facilidad para depurar, pero tiene un par de **inconvenientes**:

- Requiere bastantes recursos: Ocupa mucho espacio en disco, consume mucha memoria durante su funcionamiento y necesita un procesador relativamente potente.
- Hay que crear un nuevo proyecto para cada fuente, lo que puede suponer tener decenas de carpetas si haces todos los ejercicios propuestos (y sí, deberías hacer todos ellos si quieres asegurarte de asentar tus conocimientos).

Ejercicio propuesto (1.4.3.1): Si vas a emplear Visual Studio, instálalo y después crea y prueba un programa en C# que escriba en pantalla "Bienvenido a Visual Studio" (recuerda pulsar Ctrl+F5 para que se haga una pausa al final del programa sin necesidad de emplear la orden ReadLine).

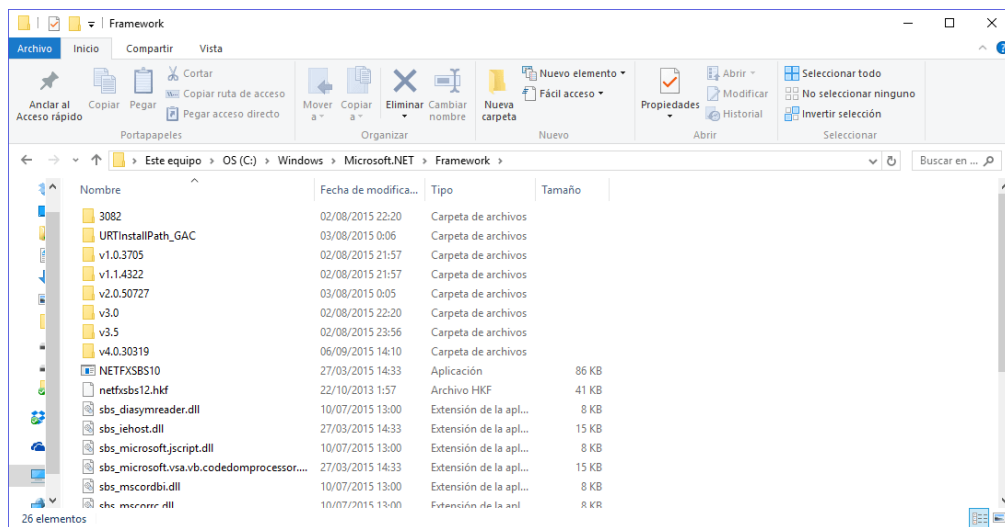
1.4.4. Windows y equipos poco potentes: .Net y Geany

Si tu equipo informático no es muy reciente, o si prefieres un entorno de desarrollo más ligero, hay una alternativa relativamente sencilla y bastante parecida a la que hemos visto para Linux: instalar Geany y utilizar el compilador de C# que es probable que ya tengas en tu sistema. Ésta es la alternativa recomendada para los primeros temas de este curso (1 al 5).

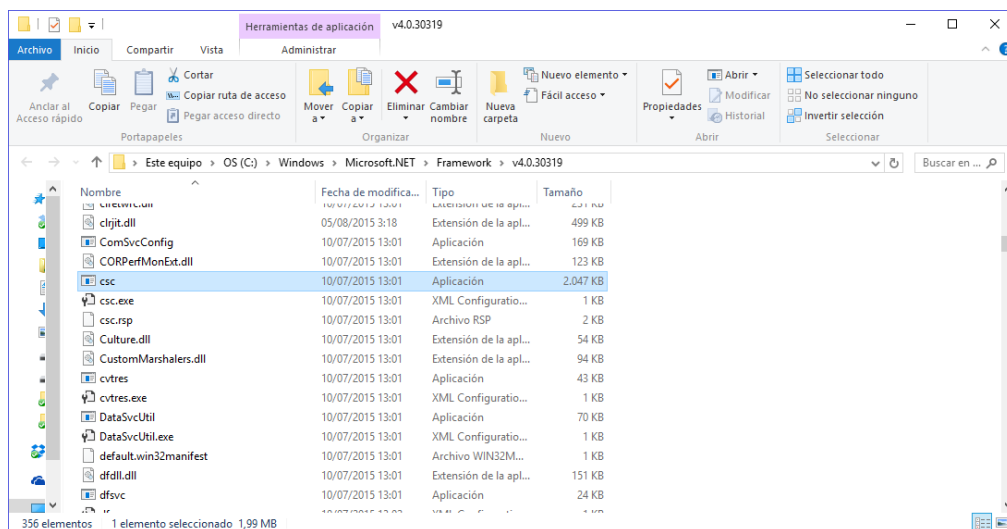
La plataforma .Net, en una u otra versión, viene preinstalada en los equipos con versiones de Windows posteriores a XP (e incluso en Windows XP, si has instalado

los "Service Pack"). Es más, esta plataforma no incluye sólo lo necesario para utilizar programas creados en C#, sino también un compilador para crearlos.

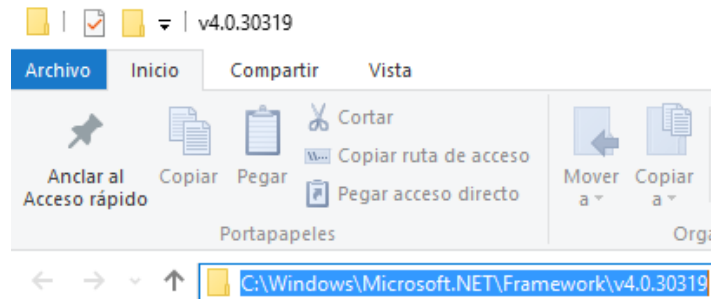
Para encontrar el compilador, deberás abrir un Explorador de archivos, y pasear por la carpeta Windows de tu equipo, subcarpeta "Microsoft .Net", donde habrá una carpeta "Framework" y, si el sistema operativo es de 64 bits, una "Framework64". En ellas existirán distintas subcarpetas, una para cada versión de la plataforma .Net que tengamos instalada (por ejemplo, "v2.0.50727" para la versión 2 y "v4.0.300319" para la versión 4).



En algunas de esas carpetas existirá un fichero llamado "csc" (de tipo "Aplicación"), que es el compilador de C#:

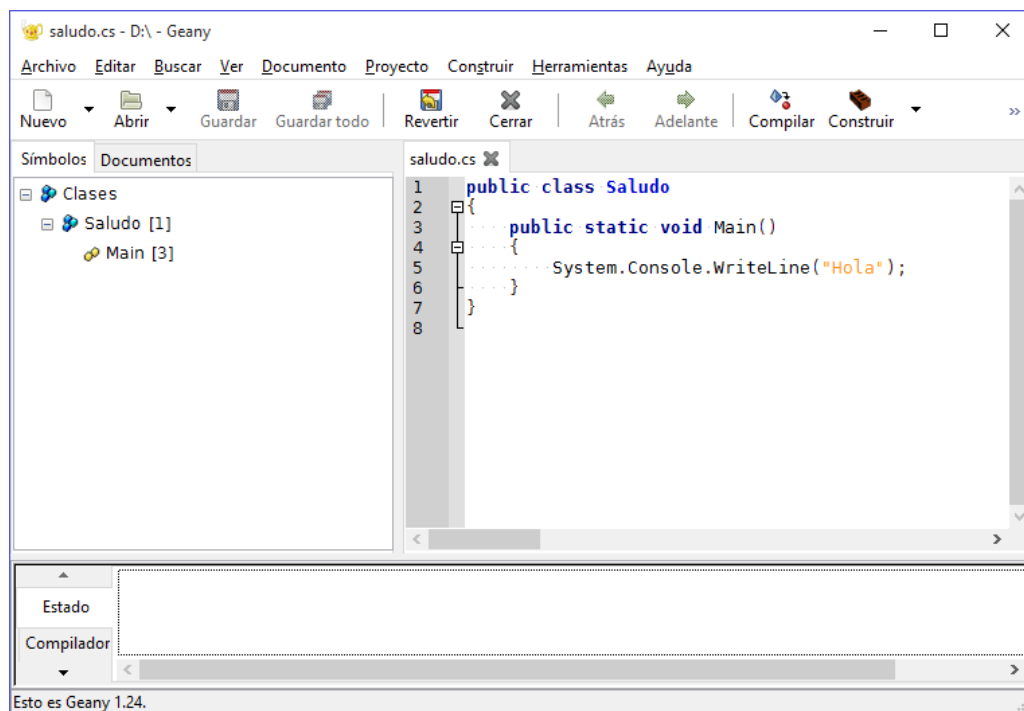


Nos interesará "memorizar esa carpeta". La forma más sencilla posiblemente será hacer clic en la barra de direcciones para ver su ruta completa y luego pulsar Ctrl+C para "copiar" esa dirección:



Si hemos encontrado el compilador, podremos instalar **Geany para Windows** y configurarlo, siguiendo los siguientes pasos:

- Descargar Geany desde www.geany.org e instalarlo.
- Teclear un programa en C# y guardarlo con un nombre que termine en ".cs" (a partir de entonces, su sintaxis se verá destacada con colores, como hemos visto para el caso de Linux).

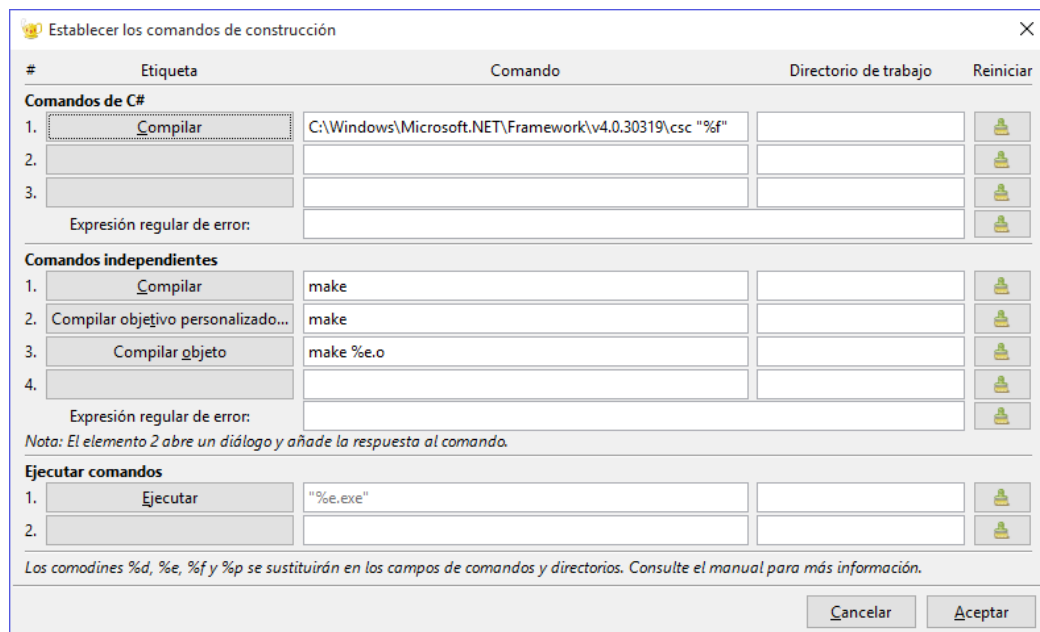


- A continuación, entraremos al menú "Construir" y la opción "Establecer comandos de construcción". En la casilla "Compilar", deberemos "pegar" la

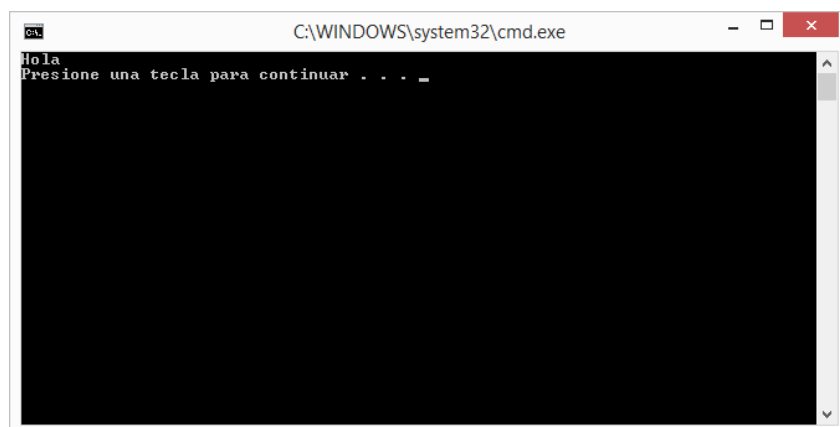
ruta en la que estaba el compilador, terminada en "\csc" y luego, entre comillas, "%f" (que es el símbolo que usa Geany para referirse al nombre del fichero actual), de modo que quedaría algo como

```
C:\Windows\Microsoft.NET\Framework64\v4.0.30319\csc.exe "%f"
```

De igual modo, en la casilla Ejecutar debería aparecer algo como "%e.exe" (incluyendo las comillas, para que se comporte correctamente incluso si el fuente está en una carpeta cuyo nombre contenga espacios):



- A partir de ese momento, ya podremos usar el botón Compilar y el botón Ejecutar para generar el ejecutable de nuestro programa y para lanzarlo (con una pausa automática al terminar), como vimos en el caso de Linux.



Ejercicio propuesto (1.4.4.1): Si vas a utilizar Geany, descárgalo y configúralo. Después crea y prueba un programa en C# que escriba en pantalla "Bienvenido a C#".

1.5. Mostrar números enteros en pantalla

Cuando queremos escribir un texto "tal cual", como en el ejemplo anterior, lo encerramos entre comillas. Pero no siempre queremos escribir textos prefijados. En muchos casos, se tratará de algo que habrá que calcular.

El ejemplo más sencillo es el de una operación matemática. La forma de realizarla es simple: no usar comillas en WriteLine. Entonces, C# intentará analizar el contenido para ver qué puede significar. Por ejemplo, para sumar 3 y 4 bastaría hacer:

```
class Ejemplo_01_05a
{
    static void Main()
    {
        System.Console.WriteLine(3+4);
    }
}
```

Ejercicios propuestos:

(1.5.1) Crea un programa que diga el resultado de sumar 118 y 56.

(1.5.2) Crea un programa que diga el resultado de sumar 12345 y 67890.

(Recomendación: no "copies y pegues" aunque dos ejercicios se parezcan. Volver a teclear cada nuevo ejercicio te ayudará a memorizar las estructuras básicas del lenguaje, ahora que todavía estás empezando).

1.6. Operaciones aritméticas básicas

1.6.1. Operadores

Parece evidente que el símbolo de la suma será un +, y podemos esperar cual será el de la resta, pero algunas de las operaciones matemáticas habituales tienen símbolos menos intuitivos. Veamos cuales son los más importantes:

Operador	Operación
----------	-----------

+	Suma
-	Resta, negación
*	Multiplicación
/	División
%	Resto de la división ("módulo")

Así, podemos calcular el resto de la división entre dos números de la siguiente forma:

```
class Ejemplo_01_06_01a
{
    static void Main()
    {
        System.Console.WriteLine("El resto de dividir 19 entre 5 es");
        System.Console.WriteLine(19 % 5);
    }
}
```

(Nota: como posiblemente imaginas y como podrás haber comprobado con el ejemplo anterior, es habitual que un programa esté formado por más de una orden, y en ese caso se analizarán de la primera -la que está más arriba- a la última -la que se encuentra más abajo-, de una en una).

Por ahora nos centraremos en los números enteros. Verás que el resultado de la división de dos números enteros es otro número que tampoco tiene cifras decimales. Por ejemplo, $15/2$ dará como resultado 7. Si quisieras realizar una división con decimales, alguno de los dos operandos deberá tener decimales, aunque no los necesite realmente, como en $15.0/2$ o en $15/2.0$.

Ejercicios propuestos:

(1.6.1.1) Haz un programa que calcule el producto de los números 12 y 13.

(1.6.1.2) Un programa que calcule la diferencia (resta) entre 321 y 213.

(1.6.1.3) Un programa que calcule el resultado de dividir 301 entre 3.

(1.6.1.4) Un programa que calcule el resto de la división de 301 entre 3.

1.6.2. Orden de prioridad de los operadores

Debería resultar sencillo porque coincide con la prioridad habitual en matemáticas:

- En primer lugar se realizarán las operaciones indicadas entre paréntesis.
- Luego la negación.
- Después las multiplicaciones, divisiones y el resto de la división.
- Finalmente, las sumas y las restas.

- En caso de tener igual prioridad, se analizan de izquierda a derecha.

Así, el siguiente ejemplo da como resultado 23 (primero se multiplica $4*5$ y luego se le suma 3) en vez de 35 (no se suma $3+4$ antes de multiplicar, aunque aparezca a la izquierda, porque la prioridad de la suma es menor que la de la multiplicación).

```
class Ejemplo_01_06_02a
{
    static void Main()
    {
        System.Console.WriteLine("Ejemplo de precedencia de operadores");
        System.Console.WriteLine("3+4*5=");
        System.Console.WriteLine(3+4*5);
    }
}
```

Ejercicios propuestos: Calcula (a mano y después comprueba desde C#) el resultado de las siguientes operaciones:

- (1.6.2.1) Calcula el resultado de $-2 + 3 * 5$
- (1.6.2.2) Calcula el resultado de $(20+5) \% 6$
- (1.6.2.3) Calcula el resultado de $15 + -5*6 / 10$
- (1.6.2.4) Calcula el resultado de $2 + 10 / 5 * 2 - 7 \% 1$

1.6.3. Introducción a los problemas de desbordamiento

El espacio del que disponemos para almacenar los números es limitado. Veremos los límites exactos más adelante, pero de momento nos basta saber que si el resultado de una operación es un número "demasiado grande", obtendremos un mensaje de error o un resultado erróneo. Por eso en los primeros ejemplos usaremos números pequeños. Más adelante veremos a qué se debe realmente este problema y cómo evitarlo. Como anticipo, el siguiente programa ni siquiera compila, porque el compilador sabe que el resultado va a ser "demasiado grande":

```
class Ejemplo_01_06_03a
{
    static void Main()
    {
        System.Console.WriteLine(10000000*10000000);
    }
}
```

1.7. Introducción a las variables: *int*

El primer ejemplo nos permitía escribir "Hola". El segundo llegaba un poco más allá y nos permitía sumar dos números que habíamos prefijado en nuestro programa. Pero esto tampoco es "lo habitual", sino que esos números dependerán de valores que haya tecleado el usuario o de cálculos anteriores.

Por eso necesitaremos reservar zonas de memoria a las que daremos un nombre y en las que guardemos los datos con los que vamos a trabajar y también los resultados temporales. A estas "zonas de memoria con nombre" les llamaremos **variables**.

Como primer ejemplo, vamos a ver lo que haríamos para sumar dos números enteros que fijásemos en el programa.

1.7.1. Definición de variables: números enteros

Para usar una cierta variable primero hay que **declararla**: indicar su nombre y el tipo de datos que queremos guardar.

El primer tipo de datos que usaremos serán números enteros (sin cifras decimales), que se indican con "int" (abreviatura del inglés "integer"). Después de esta palabra se indica el nombre que tendrá la variable:

```
int primerNumero;
```

Esa orden reserva espacio para almacenar un número entero, que podrá tomar distintos valores, y al que nos referiremos con el nombre "primerNumero".

Ejercicio propuesto (1.7.1.1): Amplía el "Ejemplo 01.06.02a" para declarar tres variables, llamadas n1, n2, n3.

1.7.2. Asignación de valores

Podemos darle un valor a esa variable durante el programa haciendo

```
int primerNumero;  
...  
primerNumero = 234;
```

Hay que tener en cuenta que esto **no es una igualdad matemática**, sino una "asignación de valor": el elemento de la izquierda recibe el valor que indicamos a la derecha. Por eso **no se puede hacer `234 = primerNumero`**, y se puede cambiar el valor de una variable tantas veces como queramos

```
primerNumero = 234;
primerNumero = 237;
```

También podemos dar un valor inicial a las variables ("inicializarlas") antes de que empiece el programa, en el mismo momento en que las definimos:

```
int primerNumero = 234;
```

Si varias variables son del mismo tipo, podemos declararlas a la vez

```
int primerNumero, segundoNumero;
```

O incluso podemos definir e inicializar más de una variable a la vez

```
int primerNumero = 234, segundoNumero = 567;
```

(esta línea reserva espacio para dos variables, que usaremos para almacenar números enteros; una de ellas se llama `primerNumero` y tiene como valor inicial 234 y la otra se llama `segundoNumero` y tiene como valor inicial 567).

Después ya podemos hacer operaciones con las variables, igual que las hacíamos con los números:

```
suma = primerNumero + segundoNumero;
```

Ejercicio propuestos:

(1.7.2.1) Amplía el ejercicio 1.7.1.1, para que las tres variables `n1`, `n2`, `n3` estén declaradas en la misma línea y tengan valores iniciales.

(1.7.2.2) Amplía el ejercicio 1.7.2.1, declarando también una variable `"suma"` y guardando en ella el resultado de sumar `n1`, `n2` y `n3`.

1.7.3. Mostrar el valor de una variable en pantalla

Una vez que sabemos cómo mostrar un número en pantalla, es sencillo mostrar el valor de una variable. Para un número hacíamos cosas como

```
System.Console.WriteLine(3+4);
```

pero si se trata de una variable es idéntico (sin comillas, para que el compilador analice su valor de antes de escribir):

```
System.Console.WriteLine(suma);
```

O bien, si queremos **mostrar un texto prefijado además del valor de la variable**, podemos indicar el texto entre comillas, detallando con `{0}` en qué parte de dicho texto queremos que aparezca el valor de la variable, de la siguiente forma:

```
System.Console.WriteLine("La suma es {0}.", suma);
```

Si queremos mostrar de más de una variable, detallaremos en el texto dónde debe aparecer cada una de ellas, usando `{0}`, `{1}` y tantos números sucesivos como sea necesario, y tras el texto incluiremos los nombres de cada una de esas variables, separados por comas:

```
System.Console.WriteLine("La suma de {0} y {1} es {2}",  
    primerNumero, segundoNumero, suma);
```

Ya sabemos todo lo suficiente para crear nuestro programa que suma dos números usando variables:

```
class Ejemplo_01_07_03a  
{  
    static void Main()  
    {  
        int primerNumero;  
        int segundoNumero;  
        int suma;  
  
        primerNumero = 234;  
        segundoNumero = 567;  
        suma = primerNumero + segundoNumero;  
  
        System.Console.WriteLine("La suma de {0} y {1} es {2}",  
            primerNumero, segundoNumero, suma);  
    }  
}
```

Repasemos lo que hace:

- (Aplazamos todavía los detalles de qué significan "public", "class", "static" y "void").
- *Main()* indica donde comienza el cuerpo del programa, que se delimita entre llaves: { y }
- *int primerNumero;* reserva espacio para guardar un número entero, al que llamaremos primerNumero.

- *int segundoNumero;* reserva espacio para guardar otro número entero, al que llamaremos segundoNumero.
- *int suma;* reserva espacio para guardar un tercer número entero, al que llamaremos suma.
- *primerNumero = 234;* da el valor del primer número que queremos sumar
- *segundoNumero = 567;* da el valor del segundo número que queremos sumar
- *suma = primerNumero + segundoNumero;* halla la suma de esos dos números y la guarda en otra variable, en vez de mostrarla directamente en pantalla.
- *System.Console.WriteLine("La suma de {0} y {1} es {2}", primerNumero, segundoNumero, suma);* muestra en pantalla el texto y los valores de las tres variables (los dos números iniciales y su suma).

El resultado de este programa sería:

La suma de 234 y 567 es 801

Ejercicios propuestos:

(1.7.3.1) Crea un programa que calcule el producto de los números 121 y 132, usando variables.

(1.7.3.2) Crea un programa que calcule la suma de 285 y 1396, usando variables.

(1.7.3.3) Crea un programa que calcule el resto de dividir 3784 entre 16, usando variables.

(1.7.3.4) Amplía el ejercicio 1.7.2.1, para que se muestre el resultado de la operación $n1+n2*n3$.

(1.7.3.5) Amplía el ejercicio 1.7.2.2, para que se muestre la suma de los tres números.

1.8. Identificadores

Los nombres de variables (lo que se conoce como "**identificadores**") pueden estar formados por letras, números o el símbolo de subrayado (_) y deben comenzar por letra o subrayado. No deben tener espacios intermedios. También hay que recordar que las vocales acentuadas y la ñe son problemáticas, porque no son letras "estándar" en todos los idiomas, así que no se pueden utilizar como parte de un identificador en la mayoría de lenguajes de programación.

Por eso, no son nombres de variable válidos:

1numero	(empieza por número)
un numero	(contiene un espacio)

Año1	(tiene una eñe)
MásDatos	(tiene una vocal acentuada)

(**Nota:** algunos entornos de programación modernos sí permitirán variables que contengan eñe y vocales acentuadas, pero como no es lo habitual en todos los lenguajes de programación, durante este curso introductorio nosotros no consideraremos válido un nombre de variable como "año", aun sabiendo que si estamos programando en C# con Visual Studio, el sistema sí lo consideraría aceptable).

Tampoco podremos usar como identificadores las **palabras reservadas** de C#. Por ejemplo, la palabra "int" se refiere a que cierta variable guardará un número entero, así que esa palabra "int" no la podremos usar tampoco como nombre de variable (pero no vamos a incluir ahora una lista con todas las palabras reservadas de C#, ya nos iremos encontrando con ellas).

Hay que recordar que en C# las **mayúsculas y minúsculas** se consideran diferentes, de modo que si intentamos hacer

```
PrimerNumero = 234;
primernumero = 234;
```

o cualquier variación similar, el compilador protestará y nos dirá que no conoce esa variable, porque la habíamos declarado como

```
int primerNumero;
```

Ejercicios propuestos:

(1.8.1) Crea un programa que calcule el producto de los números 87 y 94, usando variables llamadas "numero1" y "numero2".

(1.8.2) Intenta crear una nueva versión del programa que calcula el producto de los números 87 y 94, usando esta vez variables llamadas "1numero" y "2numero".

(1.8.3) Intenta crear una nueva versión del programa que calcula el producto de los números 87 y 94, usando esta vez variables llamadas "numero 1" y "numero 2".

(1.8.4) Crea una nueva versión del programa que calcula el producto de los números 87 y 94, usando esta vez variables llamadas "número1" y "número2".

1.9. Comentarios

Podemos escribir comentarios, que el compilador ignorará, pero que pueden ser útiles para nosotros mismos, haciendo que sea más fácil recordar el cometido un

fragmento del programa más adelante, cuando tengamos que ampliarlo o corregirlo.

Existen dos formas de indicar comentarios. En su forma más general, los escribiremos entre `/* y */`:

```
int suma; /* Guardaré el valor para usarlo más tarde */
```

Es conveniente escribir comentarios que aclaren la misión de las partes de nuestros programas que puedan resultar menos claras a simple vista. Incluso suele ser aconsejable que el programa comience con un comentario, que nos recuerde qué hace el programa sin que necesitemos mirarlo de arriba a abajo. Un ejemplo casi exagerado podría ser:

```
/* ---- Ejemplo en C#: sumar dos números prefijados ---- */

class Ejemplo_01_09a
{
    static void Main()
    {
        int primerNumero = 234;
        int segundoNumero = 567;
        int suma; /* Guardaré el valor para usarlo más tarde */

        /* Primero calculo la suma */
        suma = primerNumero + segundoNumero;

        /* Y después muestro su valor */
        System.Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}
```

Un comentario puede empezar en una línea y terminar en otra distinta, así:

```
/* Esto
   es un comentario que
   ocupa más de una línea
*/
```

También es posible declarar otro tipo de comentarios, que comienzan con doble barra y terminan cuando se acaba la línea (estos comentarios, claramente, no podrán ocupar más de una línea). Son los "comentarios de una línea" o "comentarios al estilo de C++" (a diferencia de los "comentarios de múltiples líneas" o "comentarios al estilo de C" que ya hemos visto):

```
// Este es un comentario "al estilo C++"
```


De modo que el programa anterior se podría reescribir usando comentarios de una línea:

```
// ---- Ejemplo en C#: sumar dos números prefijados ----

class Ejemplo_01_09b
{
    static void Main()
    {
        int primerNumero = 234;
        int segundoNumero = 567;
        int suma; // Guardaré el valor para usarlo más tarde

        // Primero calculo la suma
        suma = primerNumero + segundoNumero;

        // Y después muestro su valor
        System.Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}
```

En este texto, a partir de ahora los fuentes comenzarán con un comentario que resuma su cometido, y en ocasiones incluirán también comentarios intermedios.

Ejercicios propuestos:

(1.9.1) Crea un programa que convierta una cantidad prefijada de metros (por ejemplo, 3000) a millas. La equivalencia es 1 milla = 1609 metros. Usa comentarios donde te parezca adecuado.

1.10. Datos por el usuario: ReadLine

Hasta ahora hemos utilizado datos prefijados, pero eso es poco frecuente en el mundo real. Es mucho más habitual que los datos los introduzca el usuario, o que se lean desde un fichero, o desde una base de datos, o se reciban de Internet o cualquier otra red. El primer caso que veremos será el de interaccionar directamente con el usuario.

Si queremos que sea el usuario de nuestro programa quien teclee los valores, necesitamos una nueva orden, que nos permita leer desde teclado. Pues bien, al igual que tenemos `System.Console.WriteLine` ("escribir línea"), también existe `System.Console.ReadLine` ("leer línea"). Para leer textos, haríamos

```
texto = System.Console.ReadLine();
```

pero eso ocurrirá un poco más adelante, cuando veamos cómo manejar textos. De momento, nosotros sólo sabemos manipular números enteros, así que deberemos convertir ese dato a un número entero, usando `Convert.ToInt32`:

```
primerNumero = System.Convert.ToInt32( System.Console.ReadLine() );
```

Un ejemplo de programa que sume dos números tecleados por el usuario sería:

```
// Ejemplo en C#: sumar dos números introducidos por el usuario
class Ejemplo_01_10a
{
    static void Main()
    {
        int primerNumero;
        int segundoNumero;
        int suma;

        System.Console.WriteLine("Introduce el primer número");
        primerNumero = System.Convert.ToInt32(
            System.Console.ReadLine());
        System.Console.WriteLine("Introduce el segundo número");
        segundoNumero = System.Convert.ToInt32(
            System.Console.ReadLine());
        suma = primerNumero + segundoNumero;

        System.Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}
```

Ejercicios propuestos:

- (1.10.1)** Crea un programa que calcule el producto de dos números introducidos por el usuario.
- (1.10.2)** Crea un programa que calcule la división de dos números introducidos por el usuario, así como el resto de esa división.
- (1.10.3)** Suma tres números tecleados por usuario.
- (1.10.4)** Pide al usuario una cantidad de "millas náuticas" y muestra la equivalencia en metros, usando: 1 milla náutica = 1852 metros.

1.11. using System

Va siendo hora de hacer una pequeña mejora: no es necesario repetir "System." al principio de la mayoría de las órdenes que tienen que ver con el sistema (por ahora, las de consola y las de conversión), si al principio del programa utilizamos "using System":

```
// Ejemplo en C#: "using System" en vez de "System.Console"
using System;
```

```

class Ejemplo_01_11a
{
    static void Main()
    {
        int primerNumero;
        int segundoNumero;
        int suma;

        Console.WriteLine("Introduce el primer número");
        primerNumero = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Introduce el segundo número");
        segundoNumero = Convert.ToInt32(Console.ReadLine());
        suma = primerNumero + segundoNumero;

        Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}

```

Si además declaramos varias variables a la vez, como vimos en el apartado 1.5.2, el programa podría ser aún más compacto:

```

// Ejemplo en C#: "using System" y declaraciones múltiples de variables
using System;

class Ejemplo_01_11b
{
    static void Main()
    {
        int primerNumero, segundoNumero, suma;

        Console.WriteLine("Introduce el primer número");
        primerNumero = Convert.ToInt32(Console.ReadLine());
        Console.WriteLine("Introduce el segundo número");
        segundoNumero = Convert.ToInt32(Console.ReadLine());
        suma = primerNumero + segundoNumero;

        Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}

```

Ejercicios propuestos:

(1.11.1) Crea una nueva versión del programa que calcula el producto de dos números introducidos por el usuario (1.10.1), empleando "using System". El programa deberá contener un comentario al principio, que recuerde cual es su objetivo.

(1.11.2) Crea una nueva versión del programa que calcula la división de dos números introducidos por el usuario, así como el resto de esa división (1.10.2), empleando "using System". Deberás incluir un comentario con tu nombre y la fecha en que has realizado el programa.

1.12. Escribir sin avanzar de línea

En el apartado 1.7.3 vimos cómo usar {0} para escribir en una misma línea datos calculados y textos prefijados. Pero hay otra alternativa, que además nos permite también escribir un texto y pedir un dato a continuación, en la misma línea de pantalla: emplear "Write" en vez de "WriteLine", así:

```
// Ejemplo en C#: escribir sin avanzar de línea
using System;

class Ejemplo_01_12a
{
    static void Main()
    {
        int primerNumero, segundoNumero, suma;

        Console.Write("Introduce el primer número: ");
        primerNumero = Convert.ToInt32(Console.ReadLine());
        Console.Write("Introduce el segundo número: ");
        segundoNumero = Convert.ToInt32(Console.ReadLine());
        suma = primerNumero + segundoNumero;

        Console.WriteLine("La suma de {0} y {1} es {2}",
            primerNumero, segundoNumero, suma);
    }
}
```

Incluso el último "WriteLine" de varios datos se podría convertir en varios Write (aunque generalmente eso hará el programa más largo y no necesariamente más legible), así

```
// Ejemplo en C#: escribir sin avanzar de línea (2)
using System;

class Ejemplo_01_12b
{
    static void Main()
    {
        int primerNumero, segundoNumero, suma;

        Console.Write("Introduce el primer número: ");
        primerNumero = Convert.ToInt32(Console.ReadLine());
        Console.Write("Introduce el segundo número: ");
        segundoNumero = Convert.ToInt32(Console.ReadLine());
        suma = primerNumero + segundoNumero;
        Console.Write("La suma de ");
        Console.Write(primerNumero);
        Console.Write(" y ");
        Console.Write(segundoNumero);
        Console.Write(" es ");
        Console.WriteLine(suma);
    }
}
```

}

Ejercicios propuestos:

- **(1.12.1)** El usuario tecleará dos números (a y b), y el programa mostrará el resultado de la operación $(a+b)*(a-b)$ y el resultado de la operación a^2-b^2 . Ambos resultados se deben mostrar en la misma línea.
- **(1.12.2)** Pide al usuario un número y muestra su tabla de multiplicar, usando {0},{1} y {2}. Por ejemplo, si el número es el 3, debería escribirse algo como

$$3 \times 0 = 0$$

$$3 \times 1 = 3$$

$$3 \times 2 = 6$$

...

$$3 \times 10 = 30$$

- **(1.12.3)** Crea una variante del programa anterior, que pide al usuario un número y muestra su tabla de multiplicar. Esta vez no deberás utilizar {0}, {1}, {2}, sino "Write".
- **(1.12.4)** Crea un programa que convierta de grados Celsius (centígrados) a Kelvin y a Fahrenheit: pedirá al usuario la cantidad de grados centígrados y usará las siguiente tablas de conversión: $\text{kelvin} = \text{celsius} + 273$; $\text{fahrenheit} = \text{celsius} \times 18 / 10 + 32$. Emplea "Write" en vez de "{0}" cuando debas mostrar varios datos en la misma línea.