

Tema 2.1

Descargar estos apuntes en [pdf](#) o [html](#)

Índice

- [Índice](#)
- ▼ [Creando aplicaciones en .NET en la actualidad](#)
 - [Kit de desarrollo](#)
 - [Entornos de desarrollo](#)
- ▼ [Creación de un proyecto de consola](#)
 - [Definiendo una solución para asociar nuestro proyecto](#)
 - [¿Qué pasa si tenemos más de una versión de .NET instalada?](#)
 - [Integración con VS Code](#)
- [💡 'Tips' de teclado para Visual Studio \(y otros editores\)](#)
- ▼ [Estructura de una programa](#)
 - [Nuestro primer programa Hello World](#)
 - [Si TODO son clases ¿Cómo se define una?](#)
 - [¿Qué es el Main?](#)
 - [Espacios de Nombres](#)
 - [Cláusula Using](#)
- ▼ [Salida y Entrada Estándar](#)
 - [Salida Por Consola](#)
 - [Entrada Por Consola](#)
- ▼ [Definición de Identificadores en CSharp](#)
 - [Nomenclaturas más habituales](#)
 - [Palabras reservadas del lenguaje CSharp](#)
- [Tipos De Datos Básicos](#)
- ▼ [Definición de Literales en CSharp](#)
 - [Literales Booleanos](#)
 - [Literales de Carácter](#)
 - [Literales de Cadena](#)
- ▼ [Definición de variables y almacenamiento en memoria](#)
 - [Tipos de Memoria](#)
 - [Tipos de Variables](#)
 - [Funcionamiento básico del Stack y el Heap](#)

Creando aplicaciones en .NET en la actualidad

Kit de desarrollo

Además, del .NET Runtime que es lo mínimo necesario para ejecutar una aplicación, dispondremos del **Kit de Desarrollo** o SDK [.NET SDK](#) de la versión LTS actual.

Nosotros deberemos descargarnos este último, puesto que trae las herramientas y librerías necesarias para el desarrollo y depuración de aplicaciones.

Entornos de desarrollo

Editores de desarrollo generalistas

Si queremos **trabajar en cualquier SO e incluso On-Line** la mejor opción sería **Visual Studio Code** pues es un editor **Open Source** bastante ligero con una gran cantidad de plugins creados por la '*comunidad*'. Aunque no es tan potente como un IDE, nos permitirá afrontar una **gran cantidad de desarrollos en lenguajes y tecnologías diferentes**.

Aunque el uso de un IDE hace transparente para nosotros los comandos y los flujos de trabajo del proceso de desarrollo, siempre es interesante tener claro como son los procesos para así gestionar su automatización a través de herramientas de [CI/CD](#).

 **Nota:** Nosotros vamos a utilizar este último y más adelante daremos instrucciones para su instalación.

Entornos integrados de desarrollo

El IDE oficial '*gratuito*' sería [Visual Studio Community](#). Este, aunque gratuito, es muy potente y sería la opción adecuada si nos especializamos únicamente en desarrollos con .NET.

Tiene varios inconvenientes:

- Instalación pesada en tamaño y tiempo.
- Solo está disponible para **Windows** y **Mac**.

Otro IDE bastante popular es [Rider](#). Desarrollado por JetBrains, se trata de un IDE multiplataforma disponible también en Linux. Pero con el inconveniente de que no es gratuito.

Creación de un proyecto de consola

La gestión de proyectos se hace con la CLI **dotnet**, así pues para crear un proyecto desde consola usaremos **dotnet new** y en concreto para crea una aplicación de consola ejecutaremos **dotnet new console -n <nombre>**

```
B:\>dotnet new console -n ejemplo
La plantilla "Aplicación de consola" se creó correctamente.

Procesando acciones posteriores a la creación...
Ejecutando "dotnet restore" en B:\ejemplo\ejemplo.csproj...
  Determinando los proyectos que se van a restaurar...
  Se ha restaurado B:\ejemplo\ejemplo.csproj (en 77 ms).
Restauración realizada correctamente.
```

Tras ejecutar este comando, se creará un proyecto definido en el fichero **ejemplo.csproj** donde, si no especificamos la versión de .NET, nos lo creará con la que hemos instalado y el lenguaje será C# con la última versión posible de esa plataforma.

Como podemos ver, para nuestro ejemplo **la plataforma de destino es .NET 8 y la versión de C# es la 12 (aunque no lo indica porque es la de defecto para .NET 8)**

```
<Project Sdk="Microsoft.NET.Sdk">

  <PropertyGroup>
    <OutputType>Exe</OutputType>
    <TargetFramework>net8.0</TargetFramework>
    <ImplicitUsings>enable</ImplicitUsings>
    <Nullable>enable</Nullable>
  </PropertyGroup>

</Project>
```

🔴 **Nota:** Para más información ver [la documentación oficial](#).

Definiendo una solución para asociar nuestro proyecto

Un proyecto debería estar asociado a una solución. Así mismo, una solución podría tener más de un archivo asociado. De hecho, si hubiéramos creado el proyecto de consola con el Visual Studio Community nos crearía por defecto una solución a la que estaría asociado el mismo.

- Con `dotnet new sln -n <nombre>` crearemos una solución.
- Con `dotnet sln <nombre>.sln add <nombre>.csproj` añadiremos una proyecto a una solución.

```
B:\>cd ejemplo
B:\ejemplo>dotnet new sln -n ejemplo
The template "Solution File" was created successfully.

B:\ejemplo>dotnet sln ejemplo.sln add ejemplo.csproj
Se ha agregado el proyecto "ejemplo.csproj" a la solución.
B:\ejemplo>dotnet sln list
Proyectos
-----
ejemplo.csproj
```

Además, siempre es interesante añadir un fichero de metainformación para el **control de versiones de git** y la **versión de .NET esperada**. Para ello usaremos...

```
B:\ejemplo>dotnet new gitignore
The template "dotnet gitignore file" was created successfully.
B:\ejemplo>dotnet new globaljson
La plantilla "archivo global.json" se creó correctamente.
```

Nos habrá creado los ficheros `.gitignore` y `global.json` que añadirá información extra importante al proyecto.

También es posible compilar el proyecto y generar el ensamblado con `dotnet build` y ejecutar el programa usando `dotnet run`. Aunque este proceso lo podremos hacer a través de **VS Code** usando la **extensión oficial para C# de Microsoft**.

```
B:\ejemplo>dotnet build
Determining projects to restore...
Todos los proyectos están actualizados para la restauración.
ejemplo -> B:\ejemplo\bin\Debug\net8.0\ejemplo.dll
Compilación correcta.
    0 Advertencia(s)
    0 Errores
Tiempo transcurrido 00:00:00.64

B:\ejemplo>dotnet run
Hello World!
```

¿Qué pasa si tenemos más de una versión de .NET instalada?

Podemos tener más de una versión instalada del .NET SDK en nuestro sistema. Para saber cuáles tenemos instaladas y cuál es la que se está usando por defecto, podemos usar el comando `dotnet --info`. Entre otras cosas nos mostraría la siguiente información.

🔴 **Nota:** Fíjate que hablamos del SDK y no del Runtime. Ya que del Runtime podríamos tener muchas más instaladas.

```
SDK DE .NET:
Version:      8.0.301
Commit:      1a0e9c0300
Workload version: 8.0.300-manifests.f6879a9a
MSBuild version: 17.10.4+10fbfbf2e

...

.NET SDKs installed:
 6.0.406 [C:\Program Files\dotnet\sdk]
 8.0.301 [C:\Program Files\dotnet\sdk]
```

Donde nos indica que tenemos las LTS 6 y 8 instaladas y que la que se está usando por defecto es la 8.

Como si no indicamos nada, `dotnet` usará la versión por defecto. Si queremos usar una versión concreta, deberemos indicarlo al crear el proyecto con el parámetro `--framework o -f`.

```
B:\ejemplo>dotnet new console -f "net6.0" -n ejemplo
```

y dentro de la carpeta del proyecto `ejemplo` deberemos crear un fichero `global.json` con la siguiente información.

```
{
  "sdk": {
    "version": "6.0.406"
  }
}
```

Para indicarle que use ese SDK en concreto en lugar de el por defecto para compilar al hacer `dotnet build` o `dotnet run`, previamente con `dotnet --info` deberemos asegurar que la versión que hemos indicado en el `global.json` está instalada en nuestro sistema.

⚠ Importante

Deberemos asegurarnos de que durante el curso no tenemos instala una versión de .NET posterior a la 8 y si es así deberemos desinstalarla o crear el fichero en la versión 8 con el `global.json` orrespondiente.

Integración con VS Code

Para abrir una proyecto de VSCode deberemos abrir la carpeta donde se encuentra el proyecto. En nuestro caso `C:\ejemplo`.

🔴 **Nota:** Aquí dispones de un enlace con la documentación oficial sobre [la integración con VS Code](#).

Básicamente se recomienda instalar las siguientes extensiones:

1. **C# Dev Kit** Como indica su nombre es una extensión de extensiones. Nos instalará las **siguientes extensiones** que son las que realmente nos permitirán trabajar con C# en VS Code.
 - **.NET Runtime Install Tool** Como su nombre indica nos instalará automáticamente la versión de .NET para que las extensiones funcionen correctamente.

⚠ Importante

Con el fin de que la extensión no esté constantemente instalando versiones de .NET, es recomendable indicarle a la extensiones de VSCode que usen ya la versión de .NET que tenemos instalada en nuestro sistema. Por ejemplo, si hemos instalado .NET 8 para todos los usuarios.

- Iremos a la configuración de VSCode.
- Buscaremos **Shared Existing Dotnet Path**
- Indicaremos la ruta `C:\Program Files\dotnet\sdk\8.0.301` o la versión `8.0.xyz` que tengamos instalada nosotros.



Dotnet Acquisition Extension: Shared Existing Dotnet Path

The path of the preexisting .NET SDK and .NET Runtime you'd like to use for ALL extensions. Restart VS Code to apply changes.

`C:\Program Files\dotnet\sdk\8.0.301`

- **C# Microsoft** Anteriormente conocida como **OmniSharp**, es la extensión oficial del Microsoft para VSCode para el desarrollo de aplicaciones en .NET con C#.
 - Ya no será necesario usar extensiones antiguas de explorador de soluciones porque esta ya está incluida en C# Microsoft. Al abrir un proyecto de C# nos aparecerá junto al explorador del espacio de trabajo una pestaña depletable denominada **EXPLORADOR DE SOLUCIONES** que nos permitirá crear soluciones y proyectos de forma sencilla sin usar el CLI., así como añadir referencias a otros proyectos y librerías como veremos más adelante en el curso.



Importante: Con **Ctrl + Shift + P** VS Code nos ofrecerá ejecutar comandos y acciones de las extensiones instaladas. Si escribimos `.NET` nos ofrecerá todas la opciones posibles de la extensión según el contexto.

- **Ctrl + Shift + P** → **.NET: Nuevo proyecto**

Este comando creará un nuevo proyecto de C# sin necesidad de usar el CLI. Si no existe una solución en el espacio de trabajo, nos creará una nueva solución y nos añadirá el proyecto al mismo automáticamente. Pero si ya existe una solución, nos preguntará si queremos añadir el proyecto a la solución existente.

- **Ctrl + Shift + P** → **.NET: Abrir Solución** Abrirá una solución ya existente.
- **Ctrl + Shift + P** → **.NET: Cerrar Solución** Cerrará una de las soluciones abiertas.

Si tenemos un proyecto seleccionado en el explorador de soluciones, al pulsar **Ctrl + Shift + P** nos ofrecerá nuevos comandos como:

- **.NET: Compilar** todas las soluciones abiertas.
- **.NET: Recompilar** todas las soluciones abiertas.
- **.NET: Limpiar** todas las soluciones abiertas.

Básicamente, la extensión detectará que tenemos abierta una carpeta (workspace) con un proyecto o solución de .NET y **arrancará C# Microsoft automáticamente**. Si el **Servidor del Lenguaje** arrancado correctamente, nos aparecerá en la esquina inferior izquierda el símbolo de **C#** en un hexágono.

- **Ctrl + Shift + P** → **.NET: Restart Language Server**


Para reiniciar el servidor del lenguaje para la solución actual.

- **Ctrl + Shift + P** → **.NET: Generate assets for build and debug**

Este comando creará una carpeta denominada **.vscode** y en su interior los archivos **task.json** y **launch.json** que contendrán la configuración para compilar y depurar el proyecto desde VSCode sin usar el CLI.

Si queremos que al ejecutar la aplicación VSCode abra un terminal integrados que nos permita interactuar con el programa. Una vez generado el fichero **launch.json** deberemos configurar el parámetro **console** al valor **integratedTerminal** tal y como se muestra en el siguiente código.

```
"configurations": [
  {
    "console": "integratedTerminal",
  },
  ...
]
```

 **Importante:** Este tipo de depuración se ha mejorado desde 2023 y **ahora podemos ejecutar y depurar desde el menú contextual sobre un proyecto** en el **EXPLORADOR** de **SOLUCIONES**.

- **IntelliCode for C# Dev Kit** Nos ayudará a escribir código de forma más rápida y eficiente. Si tenemos algún otro asistente de código mediante IA instalado como GitHub Copilot o Google Gemini, deberemos desactivarlo para que no interfiera con IntelliCode.
2. **Visual Studio Keymap** Me permitirá configurar el teclado como si estuviéramos usando "su hermano mayor" el IDE de Visual Studio.

💡 'Tips' de teclado para Visual Studio (y otros editores)

A lo largo del curso se darán diferentes tips de uso del teclado y atajos para refactorizar.

Puesto que hemos instalado "*Visual Studio Keymap*". Se nos habrán configurados automáticamente las siguientes extensiones de teclado.

👉 **Importante:** Es importantísimo que empieces a utilizarla ya si quieres coger cierta soltura desarrollando código. Tarde o temprano lo tendrás que hacer.

Acción	Teclado	Acción	Teclado
Salta letra	→/←	Salta palabra	Ctrl+→ o Ctrl+←
Salta al inicio	Ctrl+Inicio	Salta al final	Ctrl+Fin
Selecciona letra	Shift+→ o Shift+←	Selecciona palabra	Shift+Ctrl+→ o Shift+Ctrl+←
Selecciona línea	Inicio , Shift+↓	Selecciona columna	Shift+Alt+↓ y/o Ctrl+Alt+→
Selecciona hasta inicio	Ctrl+Shift+Inicio	Selecciona todo	Ctrl+A o Ctrl+Inicio , Ctrl+Shift+Fin
Selecciona hasta fin	Ctrl+Shift+Fin	Copiar línea o selección	Ctrl+C
Mover línea o selección actual	Alt+↑ o Alt+↓	Pegar	Ctrl+V
Cortar línea o selección	Ctrl+X	Duplica selección	Ctrl+D
Pegar sin formato	Ctrl+Shift+V	Des-tabula selección	Shft+Tab
Tabula selección	Tab	Buscar en todo	Ctrl+Shift+F
Buscar	Ctrl+F	Buscar y Reemplazar	Ctrl+H
Buscar siguiente	F3	Formatea Código	Ctrl+K , Ctrl+D
Buscar y Reemplazar en todo	Ctrl+Shift+H	Des-comenta Selección	Ctrl+K , Ctrl+U
Comenta Selección	Ctrl+K , Ctrl+C	Ejecuta sin depurar	Ctrl+F5
Ir a siguiente error o aviso	F8	Re-nombra identificador	Ctrl+R , Ctrl+R
Ejecuta depurando	F5	Ir a la definición	F12
Ir a la declaración	Ctrl+F12	Ver Información sobrecargas	Ctrl+shift+Space
Ver Sugerencia contextual	Ctrl+Space	Ver Refactorización contextual	Ctrl+.

Acción	Teclado	Acción	Teclado
Muestra información in-line	Ctrl+K , Ctrl+I		

Estructura de una programa

Además, de estos apuntes. Puedes seguir, de forma complementaria, este [Tutorial Básico](#) para aprender las bases de lenguaje.

Nuestro primer programa Hello World

Si abrimos el fichero **Program.cs** que se ha creado junto a nuestro proyecto podrás ver el siguiente código...

```
Console.WriteLine("Hello, World!");
```

Como ves es una simple línea donde muestra el típico "*Hola Mundo*" en inglés.

Sin embargo, en versiones anteriores a C# 10 verás ejemplos de programa básico como el siguiente...

```
using System;
namespace ejemplo
{
    class Program
    {
        static void Main()
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

En el fondo tiene que haber una clase básica que en el código anterior hemos llamado **Program** y dentro de ella un método estático (ya veremos lo que son) denominado obligatoriamente **Main** (Principal). Este método es el programa principal y se denomina así en C# y muchos otros lenguajes como herencia del lenguajes C.

Realmente, todo el código escrito en C# se ha de escribir **dentro** de una definición de clase y aunque nosotros no la definamos como en el segundo ejemplo. Internamente C# meterá el **Console.WriteLine("Hello, World!");** dentro de un método **Main** que a su vez estará definido en una clase que C# definirá por nosotros.

Si TODO son clases ¿Cómo se define una?

Utilizaremos la palabra reservada **class**

La definiremos con la siguiente sintaxis...

```
class Identificador
{
    <miembros>
}
```

👉 **Importante:** Es posible definir varias clases en un archivo, pero lo normal **en todos los lenguajes**, es que haya una por archivo o fichero fuente. De hecho, si prescindimos de la clase **Program** y el **Main** de nuestro ejemplo. No podremos definir ningún tipo o clase antes del **Console.WriteLine("Hello World!");**

¿Qué es el Main?

Es el punto de entrada de un programa, esto es, donde empieza a ejecutarse. Ya hemos vistos que si no lo definimos y ponemos instrucciones directamente. El compilador de C# definirá un **Main** por defecto y meterá todas esas instrucciones en el mismo.

Sólo **puede haber uno** y deberemos declararlo de las siguientes maneras, pero siempre con el modificador **static**:

```
static void Main()
static int Main()
static int Main(string[] args)
static void Main(string[] args)
...
```

Espacios de Nombres

Aunque **lo vamos a ver en más profundidad cuando veamos el concepto de paquete** y librería. Puede ser interesante hacer esta primera introducción. Podemos decir que un espacio de nombres es un **identificador** o etiqueta que me ayuda a agrupar y clasificar un conjunto de definiciones de tipos relacionadas.

En C# se definen de la siguiente manera:

```
namespace Espacio1 // Espacio1 es el identificador o etiqueta del namespace
{
    // Definiciones de tipos dentro del namespace Espacio1
    namespace Espacio11
    {
        // Definiciones de dentro del namespace Espacio1.Espacio11
        class MiTipoEnEspacio11
        {
            ...
        }
    }
}
```

También podría hacer...

```
namespace Espacio1.Espacio12
{
    // Definiciones de dentro del namespace Espacio1.Espacio12
    class MiTipoEnEspacio12
    {
        ...
    }
}
```

Nombre Completamente Calificado

Para hacer referencia a un tipo, como por ejemplo una determinada clase, debo indicar la jerarquía de espacios de nombres seguidos por puntos. A esto se le denomina Nombre Completamente Calificado o **NCC**.

```
Espacio1.Espacio11.MiTipoEnEspacio11
```

Esto permite que diferentes librerías puedan dar el mismo nombre a sus tipos o clases y poder distinguir entre una y otra.

```
System.Console.WriteLine("Hola Mundo");
MisDefiniciones.Console.Escribe("Hola Mundo");
```

En el ejemplo anterior tenemos definidas dos clases **Console** con el mismo identificador. Si estuvieran definidas en el mismo espacio de nombres, se produciría un conflicto por la ambigüedad de nombres. Sin embargo, podemos definir otra clase **Console** dentro del espacio de nombres **MisDefiniciones** y cuando queramos usar una u otra usaremos el **NCC** como se muestra en el ejemplo.

Cláusula Using

Para no tener que poner siempre el NCC cuando esté usando tipos y clases definidas dentro de un determinado namespace, puedo usar la cláusula **using**.

```
using System;
class Program
{
    static void Main()
    {
        Console.WriteLine("Hello World!");
        // en lugar de tener que poner.
        System.Console.WriteLine("Hello World!");
    }
}
```

Aunque **no es muy común**, podremos definir **alias** para clases repetidas en diferentes namespaces mediante la siguiente sintaxis: `using <alias> = <NCC>;`

```
using System;
using MiConsole = MisDefiniciones.Console;
...
Console.WriteLine("Hola Mundo"); // Equivale a System.Console.WriteLine("Hola Mundo");
MiConsole.Escribe("Hola Mundo"); // MisDefiniciones.Console.Escribe("Hola Mundo");
```

Using Global

Volvamos al `program.cs` que se ha creado en C# 10 donde solo hay un simple `Console.WriteLine("Hello, World!");` y no tenemos ningún `using System;`.

Recordemos que incluir el espacio de nombres `System` es imprescindible para conocer la clase `Console` que es donde está definido.

👉 ¿Cómo puedes ser que el compilador no proteste y nos diga que no conoce la clase `Console` si no hemos hecho el `using System;` para que nuestro programa conozca todos los tipos definidos en dicho espacio de nombres?

La respuesta es la siguiente...

Si volvemos al `ejemplo.csproj` que creamos al crear el programa de consola, veremos la siguiente línea ..

```
...
<ImplicitUsings>enable</ImplicitUsings>
...
```

En la cual le estamos indicando al compilador que permita la definición de `using implícitas`, esto es, cláusulas using que se suelen repetir en todos los ficheros fuente y que ahora se incluirán de forma **global** en todos ellos de forma automática o implícita. Pero **¿Cuales son estas directivas using que se usarán de forma global en todos los archivos de proyecto?**

Al compilar por primera vez el proyecto si tenemos `<ImplicitUsings>enable</ImplicitUsings>` se generará el archivo `Debug/net6.0/ejemplo.GlobalUsings.g.cs` y en el tendremos definidos los espacios de nombres cuyos tipos serán conocidos en todos los fuentes del proyecto.

```
// <auto-generated/>
global using global::System;
global using global::System.Collections.Generic;
global using global::System.IO;
...
```

🔴 **Nota:** El contenido de este archivo es **auto-generated** lo cual significa que es generado automáticamente y aunque lo modifiquemos se volverá a generar por el compilador.

Fíjate que antes de la cláusula `using` aparece el **modificador global** el cual le indicará al compilador que ese using estará implícito en todos nuestros fuentes.

🔔 Resumen:

A partir de **C# 10** no hace falta poner el `using System;` o incluir definiciones de otros espacios de nombres al tener la directiva `<ImplicitUsings>enable</ImplicitUsings>` que se añade por defecto.

Sin embargo en versiones anteriores, deberemos indicarla al principio de nuestro programa si queremos que el compilador reconozca la clase `Console`.

Using Estático

Aunque mucho más adelante en el curso volveremos aquí y profundizaremos en este concepto para entenderlo correctamente.

En un primer momento apuntaremos, que también tenemos la posibilidad de indicar que todos los miembros de una clase estática (**veremos lo que son más adelante**) sean accesibles de forma global en todo nuestro proyecto.

¿Qué significa esto? Bueno, fíjate que delante de la instrucción `WriteLine("Hello, World!");` hemos tenido que poner `Console.` porque `Console` es una **clase estática** que define diferentes métodos de utilidad relacionados con la consola. Pero siempre tenemos que poner `Console.` antes de un `WriteLine(..)`. Pero..., ¿Es posible hacer implícito el `Console.` igual que hemos quitado el `System.` al añadir el `global using System;`?

La respuesta es sí, si añadimos el **modificador** `static` después del `using` indicando que deseamos usar los métodos definidos en una determinada clase estática.

Por tanto, si definimos nuestros propios usings globales en un nuevo archivo en el proyecto denominado `ejemplo/GlobalUsings.cs` añadiendo la línea ...

```
global using static System.Console;
```

ahora podremos modificar nuestro `program.cs` y poner simplemente ...

```
WriteLine("Hello, World!");
```

y seguirá funcionando. Ya que es como si estuviéramos poniendo de forma implícita el NCC

```
System.Console.WriteLine("Hello, World!");
```

Salida y Entrada Estándar

Salida Por Consola

Como hemos visto, utilizaremos la clase **Console** definida en el namespace **System**.

Esta clase solo tiene sentido usarla en aplicaciones de consola.

Para la salida estándar utilizaremos los métodos `Write` y `WriteLine` (añade un salto de línea al final) que además me permiten formatear una salida numérica o de texto con parámetros.

```
System.Console.Write("Total: ");
System.Console.WriteLine(200);
// Salida
// Total: 200<Salto Línea>
```

Formatos compuestos de texto

Referencia

La sintaxis es: `Console.Write("<texto>{N,M:FormatString}<texto>");`

- **N** indicará el dato a sustituir entre llaves y con el número de orden del parámetro que le corresponde empezando desde 0.
- **M** la alineación. (+ A la derecha y – A la izquierda)
- **FormatString** indicará como se deben mostrar un dato numérico.
Podrá tomar los siguientes valores:

Valor	Significado
C	Muestra el número como una unidad monetaria, usando el símbolo y las convenciones de la moneda local.
D	Muestra el número como un entero decimal.
E	Muestra el número usando notación exponencial (científica).
F	Muestra el número como un valor en coma fija.
G	Muestra el número como un valor entero o en coma fija, dependiendo del formato que sea más compacto.
N	Muestra el número con comas incorporadas.
X	Muestra el número utilizando notación hexadecimal.

Ejemplos formato compuesto

```
1. Console.WriteLine("\Justificación a la {0} con anchura 10: {1,-10}\", "izquierda", 99); `
// Salida
// "Justificación a la izquierda con anchura 10: 99      "
```

```
2. Console.WriteLine("\Justificación a la {0} con anchura 10: {1,10}\", "derecha", 99);
// Salida
// "Justificación a la derecha con anchura 10:          99"
```

```
3. Console.WriteLine("Formato entero - {0:D5}", 88);
// Salida
// Formato entero - 00088
```

```
4. Console.WriteLine("Formado exponencial - {0:E}", 888.8);
// Salida
// Formado exponencial - 8,888000E+002
```

```
5. Console.WriteLine("Formato de punto fijo - {0:F3}", 888.8888);
// Salida
// Formato de punto fijo - 888,889
```

```
6. Console.WriteLine("Formato general - {0:G}", 888.8888000);
// Salida
// Formato general - 888,8888
```

```
7. Console.WriteLine("Formato de número - {0:N}", 8888888.8);
// Salida
// Formato de número - 8.888.888,80
```

```
8. Console.WriteLine("Formato hexadecimal - {0:X4} Hex", 88);
// Salida
// Formato hexadecimal - 0058 Hex
```


Entrada Por Consola

Entrada de texto

Para la entrada estándar utilizaremos los métodos **Read** que lee el siguiente carácter desde stdin (-1 si esta vacía) y **ReadLine** que lee hasta final de línea.

```
string texto = Console.ReadLine();
Console.WriteLine(texto);
```

🚩 **Nota:** Solo funcionará si leemos cadenas.

Entrada de datos otros tipos

Todos los tipos del CTS tienen definido el método tipo `Parse(string texto)` que transforma de cadena al tipo.

```
int entero = int.Parse(Console.ReadLine() ?? "0");
double real = double.Parse(Console.ReadLine() ?? "0");
```

🚩 **Nota:** Si el texto leído no se puede convertir al tipo correspondiente se producirá un error de ejecución y el programa finalizará.

👉 **Importante:** Puesto que `ReadLine()` devuelve un `string?` que puede ser **anulable** y `Parse(string p)` espera un string **no anulable**. Debemos añadir `?? "0"` indicando que la cadena a evaluar en el `Parse` será `"0"` (del tipo que espere el parse) para no obtener un aviso de compilación. Este concepto lo veremos en profundidad en el **tema 5.1** y de momento hasta ese tema puedes no ponerlo, pero obtendrán dicho aviso.

Definición de Identificadores en CSharp

- Con ellos damos nombre a elementos de un programa, como clases, constantes, variables, métodos, etc...
- Un identificador está formado por letras (a|b|c|...|z|A|B|C|...|Z), y dígitos (0|1|2|3|...|9), la única restricción que se nos pone, es que **el primer carácter empiece por una letra** o signo de subrayado '_'.
- C# Distingue entre mayúsculas y minúsculas.
- Es valido cualquier carácter UNICODE como la 'ñ'.
- No son válidos caracteres propios del lenguaje como '"', '#', '?', '@', etc...
- Es importante utilizar una notación o convenio al nombrar, y siempre que haya algún tipo de unidad indicarlo.

Resumen: Deberían ser **sustantivos** significativos, evitando contracciones en identificadores cortos y redundancia dentro del contexto. Buscaremos pues **nombres autodocumentados**.

Nomenclaturas más habituales

- **PascalCasing**
 - Consiste en poner en mayúscula el primer carácter de todas las palabras.
 - Se utiliza para clases, métodos, propiedades, enumeraciones, interfaces, campos constantes y de sólo lectura, espacios de nombres, propiedades, etc...
 - Ejemplo: void **I**nicializa**D**atos(...);
- **camelCasing**
 - Consiste en poner en mayúscula el primer carácter de todas las palabras excepto la primera.
 - Se utiliza para variables que definan campos, parámetros y variables miembro privadas.
 - Ejemplo: int **n**umero**D**e**P**lantas;

'INCORRECTO'	'CORRECTO'
6Caja	Caja6
tot	TotalUnidades
ventlun	VentasLunes
longitudSalto	longitudSalto_m

Palabras reservadas del lenguaje CSharp

- Son identificadores o cláusulas propias del lenguaje.
- No se pueden utilizar como identificadores. En realidad si se puede, siempre y cuando las precedamos del carácter '@', pero no deberíamos usarlas nunca.
- Son siempre minúsculas.
- Los **editores de CSharp normalmente nos las resaltarán** con algún color determinado.

```
abstract, as, base, bool, break, byte, case, catch, char, checked, class,  
const, continue, decimal, default, delegate, do, double, else, enum, event,  
explicit, extern, false, finally, fixed, float, for, foreach, goto, if,  
implicit, in, int, interface, internal, lock, is, long, namespace, new, null,  
object, operator, out, override, params, private, protected, public, readonly,  
ref, return, sbyte, sealed, short, sizeof, stackalloc, static, string, struct,  
switch, this, throw, true, try, typeof, uint, ulong, unchecked, unsafe, ushort,  
using, virtual, void, while, ...
```

”

There are only two hard things in computer science: cache invalidation and naming things.

- Phil Karlton.

”

Tipos De Datos Básicos

C# es (**type-safe**), esto significa que se garantizan los valores almacenados.

Tipo	Descripción	Bits	Rango de valores	Alias
SByte	Bytes con signo	8	[-128, 127]	sbyte
Byte	Bytes sin signo	8	[0, 255]	byte
Int16	Enteros cortos con signo	16	[-32.768, 32.767]	short
UInt16	Enteros cortos sin signo	16	[0, 65.535]	ushort
Int32	Enteros normales	32	[-2.147.483.648, 2.147.483.647]	int
UInt32	Enteros normales sin signo	32	[0, 4.294.967.295]	uint
Int64	Enteros largos	64	[-9.223.372.036.854.775.808, 9.223.372.036.854.775.807]	long
UInt64	Enteros largos sin signo	64	[0-18.446.744.073.709.551.615]	ulong
Single	Reales con 7 dígitos de precisión	32	[1,5×10-45 - 3,4×10+38]	float
Double	Reales de 15-16 dígitos de precisión	64	[5,0×10-324 - 1,7×10+308]	double
Decimal	Reales de 28-29 cifras decimales	128	[1,0×10-28 - 7,9×10+28]	decimal
Boolean	Valores lógicos	8	true, false	bool
Char	Caracteres Unicode	16	['\u0000', '\uFFFF']	char
String	Cadenas de caracteres (pág 120)	Variable	El permitido por la memoria	string
Object	Cualquier objeto	Variable	Cualquier objeto	object

Definición de Literales en CSharp

Entero sin signo: Los literales llevarán el carácter **U** ó **u**

```
uint variable = 4000000000U;
```

Entero largo y con signo: Los literales llevarán el carácter **L** y **U**

```
long variable = 4000000000000000L;  
ulong variable = 1800000000000000000UL;
```

Hexadecimal: Pondremos el prefijo **0x** al valor en hexadecimal.

```
ushort n = 0x070F; // n = 0000 0111 0000 1111(2  
ushort n = 0x_07_0F; // Desde C# 7
```

Binario desde CSharp 7: Pondremos el prefijo **0b** al valor en binario.

```
ushort n = 0b0000011100001111; // n = 070F(16  
ushort n = 0b_0000_0111_0000_1111; // Desde C# 7
```

Real de simple precisión: los literales llevarán el carácter **F** ó **f**

```
float variable = 333.44f;  
float variable = 4.5E-15F;
```

Real de doble precisión: los literales llevarán el carácter **D** ó **d**

```
double variable = 124325.456859D;  
double variable = 4.5e127d;
```

Decimal: Los literales llevarán el carácter **M** ó **m**

```
decimal variable = 64538756498374657493847594.45M;
```

Literales Booleanos

Son las palabras reservadas `true` y `false`

```
bool estaFraccionado = true;
```

Literales de Carácter

- Podemos inicializarlo a cualquier carácter UNICODE o UTF-8
- Podemos consultar sus valores en [UNICODE](#)
- Los literales de carácter se definirán siempre entre comillas simples.
- Podremos representar cualquier literal a través de códigos de escape.

Carácter	Código UNICODE	Código escape
Comilla simple	\u0027	'
Comilla doble	\u0022	"
Carácter nulo	\u0000	\0
Alarma	\u0007	\a
Retroceso	\u0008	\b
Salto de página	\u000C	\f
Retorno de carro	\u000D	\r
Nueva línea	\u000A	\n
Tabulación horizontal	\u0009	\t
Tabulación vertical	\u000B	\v
Barra invertida	\u005C	\

```
char c1 = 'F';  
char c2 = '\\';  
char c3 = '\\u20AC'; \\ €
```

Literales de Cadena

- Los literales se definirán entre comillas dobles "literal"

```
string texto = "Esto es una cadena de caracteres.";
```

- Dentro del literal de la cadena podremos incluir secuencias de escape de carácter, como las que hemos visto anteriormente.

```
string texto = "El precio es 20 \u20AC\nGRACIAS";
```


- Si colocamos el carácter `@` antes del literal de la cadena, se escapará cualquier secuencia de escape que se encuentre..

```
string texto = @"El precio es 20 \u20AC\nGRACIAS";  
string texto = "El precio es 20 \\u20AC\\nGRACIAS";
```

- A partir de **C# 6** se pueden **interpolar cadenas** con las reglas de formato que ya hemos visto en la salida por pantalla.

Para ello colocaremos el carácter `$` delante de la misma.

```
string t1 = "derecha";  
int n = 99;  
string t2 = $"Justificación a la {t1} con anchura 10: {n, 10}\n";  
// Salida  
// "Justificación a la derecha con anchura 10:           99"  
  
int n = 88;  
string t = $"Formato entero - {n:D5}";  
// Salida  
// "Formato entero - 00088"
```

 **Importante:** A partir de ahora usaremos este formato pues, además de hacer más legibles las cadenas, nos evitará errores derivados de una mala indexación en la sustitución.

- A partir de **C# 11** se han incluido los **literales de cadena sin formato** que pueden contener varias líneas y no hará falta usar caracteres de escape por lo podremos incluir cualquier carácter incluida la propia comillas doble. Son ideales para definir cadenas con objetos en formato JSON, XML, YAML etc.

```
var json = """  
{  
    "nombre": "Juan",  
    "edad": 25  
}  
""";
```

Como puedes ver se definen con tres comillas dobles y se cierran con otras tres comillas dobles. Esta sintaxis es similar a la de otros lenguajes como **Kotlin**.

Caso de estudio:

Supongamos el siguiente código donde definimos una variables que representan el nombre de una fruta y su cantidad en en kilogramos...

```
global using static System.Console;

string fruta1 = "Peras";
double cantidadFruta1_kg = 12.5d;

string fruta2 = "Manzanas";
double cantidadFruta2_kg = 2d;
```

Ahora nos piden que se muestren los datos con el siguiente especificación:

Usando **interpolación de cadenas**, representa cada fruta y su cantidad en una sola línea de tal manera que primero se mostrará el nombre alineado a la izquierda y con un ancho de columna mínimo de 10. A continuación irá la cantidad alineada a la derecha, con 2 decimales, un ancho de columna mínimo de 7 y seguido de un espacio y el texto de las unidades.

Dicha especificación la podremos representar de la siguiente forma ...

```
      10      7
FFFFFFFFFCCCCC
Fruta      X,XX Kg
```

Por lo que la salida final quedará...

```
Peras      12,50 Kg
Manzanas   2,00 Kg
```

Una propuesta de solución podría ser la siguiente...

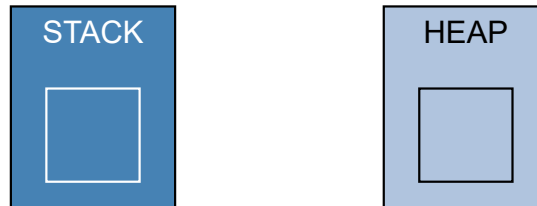
```
// Definiremos como constantes lo que sabemos que no cambia como son el
// texto con las unidades y las longitudes de ambas columnas.
const string UNIDADES = "Kg";
const int COLUMNA_NOMBRE = 10;
const int COLUMNA_CANTIDAD = 7;

// Posteriormente mostraremos cada fruta siguiendo las especificaciones
WriteLine($"{fruta1, -COLUMNA_NOMBRE}{cantidadFruta1_kg, COLUMNA_CANTIDAD:F2} {UNIDADES}");
WriteLine($"{fruta2, -COLUMNA_NOMBRE}{cantidadFruta2_kg, COLUMNA_CANTIDAD:F2} {UNIDADES}");
```


Definición de variables y almacenamiento en memoria

En este bloque vamos a hacer **una introducción** a una serie de **conceptos y definiciones muy importantes** y que deberemos recordar, pues iremos hablando de ellos a lo largo del curso y **son imprescindibles de saber**.

Tipos de Memoria



Como consecuencia del lenguaje C, en la mayoría de lenguajes posteriores como C++, Java, C#, etc., el almacenamiento de los datos de un programa se divide en dos zonas las cuales vamos intentar explicar de forma simplificada o conceptual...

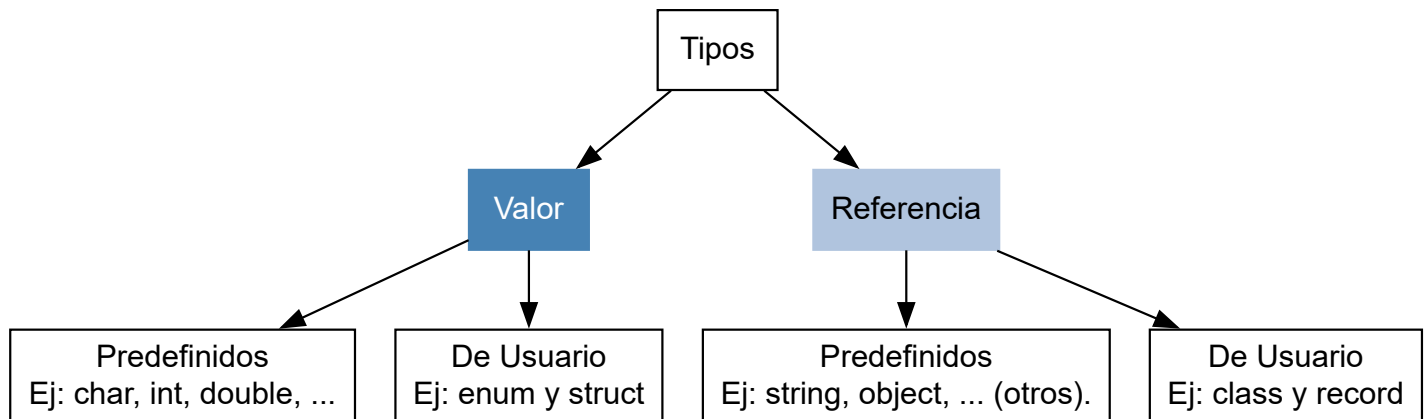
La Pila (Stack)

- Donde se almacena la definición cualquier variable junto con su identificador.
- Su tamaño es limitado y si lo sobrepasamos obtendremos un **StackOverflow**.
- No debería almacenarse más de **16 bytes** de tamaño **por dato**.
- Los datos en la **pila** se van guardando **de forma ordenada**, de abajo hacia arriba como si apiláramos algo y solo podremos acceder a ellos en cualquier momento sino cuando estén a nuestro alcance (**scope**).

La Memoria Montón (Heap)

- Su idea es almacenar gran cantidad de información como si '*amontonáramos*' los datos en diferentes sitios de forma desordenada. Es por eso que se llama **Heap** o **montón**.
- Podremos acceder '*en cualquier momento*' a los datos almacenados en ella.
- Por simplificar comentaremos que su tamaño, aunque limitado, es muy superior al de la pila. Si lo sobrepasamos o nos solapamos con la que usa otro programa obtendremos un **OutOfMemoryException**.
- Podremos almacenar datos de '*cualquier tamaño*' si hemos logrado encontrar espacio libre suficiente para ello.

Tipos de Variables



Variables De Tipo Valor

- El dato se almacenan en el **Stack**.
- No deberían ser mayores de 16 bytes.
- Son **Type-Safe**, lo cual nos asegura que el valor de la variable de ese tipo tiene una valor de su rango para el tipo válido.
- Un tipo valor seguido del carácter `?` se convertirá en **anulable** y por tanto se podrá asignar a **null**.
- La palabra reservada `default` se evaluará al valor por defecto para el tipo o a null si es anulable.
- Cada variable almacena *'su propia copia del dato'*.
- Las operaciones sobre una no afectan a otra.
- Serán de tipo valor todos los tipos básicos `char`, `int`, `double`, ... menos `string` y unas tipos que veremos más adelante como `struct` y `enum`.

Ejemplo Declaración de Tipos Valor

```
// Entero no anulable al que asigno 5 y con valores posibles de 0 a 255
byte a = 5;

// Entero no anulable al que asigno default para byte, que es 0 y con valores posibles de 0 a 255
byte b = default;

// Entero anulable al que asigno default para byte, que ahora es null por ser anulable
// y con valores posibles null o de 0 a 255

byte? c = default;
// Tras la inicialización tendré en memoria: a(5) b(0) c null

// Muestra: a(5) b(0) c()
Console.WriteLine($"a({a}) b({b}) c({c})");

c = a;
// Muestra: a(5) b(0) c(5)
Console.WriteLine($"a({a}) b({b}) c({c})");

c = 255;
// Muestra: a(5) b(0) c(255)
Console.WriteLine($"a({a}) b({b}) c({c})");

// Como 256 no es un valor válido para byte da la vuelta y empieza en 0
c = (byte)(c + 1);
// Muestra: a(5) b(0) c(0)
Console.WriteLine($"a({a}) b({b}) c({c})");

a = null; // Error de compilación
c = null; // Válido
```

Variables De Tipo Referencia

- En el **Stack** se almacena **una referencia** a donde se guarda el dato en el **Heap**.
- Dos variables de tipo referencia pueden '*apuntar*' a la misma zona en el Heap.
- 🧠 La modificación de una variable, pueden afectar a la otra, si ambas '*apuntan*' al mismo dato en el Heap.
- Estos datos se denominarán objetos y serán de tipo referencia el resto de datos que no son tipos valor incluido el tipo simple **string**. Ya usaremos y definiremos algunos de estos tipos durante el curso.
- La liberación de espacio ocupado por los datos en el Heap, la hace el Recolector de Basura o Garbage Collector (GC) cuando esos datos dejan de estar referenciados por una variable en el Stack.
- En **C# 10**, los proyectos tienen la siguiente configuración por defecto.

```
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    ...
    <Nullable>enable</Nullable>
    ...
  </PropertyGroup>
</Project>
```

Esto hace que se genere un aviso (Warning) si asigna un literal **null** a un tipo referencia en algún momento de la ejecución.

Por tanto si hago las siguientes asignaciones ...

```
string texto1 = null;
string texto2 = default; // El valor por defecto para un tipo referencia es null.
```

Se generará un aviso similar a este: *"Se va a convertir un literal nulo o un posible valor nulo en un tipo que no acepta valores NULL"*

Por eso al igual que sucede con los tipos valor, deberé de marcarlos como anulable (nullable) en la definición con la **?**.

```
string? texto1 = null;
string? texto2 = default;
```

Este comportamiento es similar al de otros lenguajes modernos como Rust o Kotlin.

Ejemplo Declaración de Tipos Referencia

```
// La variable referencia a un objeto string en memoria, que contiene la cadena hola a -> (hola)
string a = "hola";

// Aviso de compilación al estar activado <Nullable>enable</Nullable> pues default vale null y con este
// directiva activada no se puede asignar null directamente a un tipo referencia.
string b = default;

// No generará ningún aviso porque estamos especificando de forma explícita que
// se pueda asignar un null al declarar la variable.
// Se comportará como si hubiéramos hecho string c = default;
string? c = default;

// Tras la inicilización tendré en memoria: a->(hola) b->>() c->>null

// Muestra: a->(hola) b->>() c->>()
Console.WriteLine($"a->({a}) b->({b}) c->({c})"); // Muestra: a->(hola) b->>() c->>()

c = a;
// Muestra: a->(hola) b->>() c->a y por tanto c->a->(hola)
Console.WriteLine($"a->({a}) b->({b}) c->a y por tanto c->a->({c})");

c = "Pepe";
// Muestra: a->(hola) b->>() c->(Pepe)
Console.WriteLine($"a->({a}) b->({b}) c->({c})");

a = null; // Aviso de compilación.
c = null; // Válido en todos los casos.
```

Funcionamiento básico del Stack y el Heap

Para entender bien el funcionamiento vamos a definir el concepto de **bloque de instrucciones**...

Usaremos llaves para delimitar bloques donde va a ir uno o más instrucciones.

```
{  
    // Varias instrucciones de código.  
}
```

Se pueden **anidar bloques** pero no pueden declararse variables con el mismo nombre en ellos.

Un bloque **anidado** irá tabulado o *'indentado'* dentro del bloque que lo contiene.

```
{  
    int i;  
    ...  
    {  
        int i; //ERROR, i esta declarada en un ámbito envolvente  
        ...  
    }  
}
```

Bloques hermanos pueden tener variables con el mismo nombre.

```
{  
    int i;  
    ...  
}  
...  
{  
    int i;  
    ...  
}
```

Como hemos comentado al principio del tema, tenemos un bloque de instrucciones principal denominado **Main()** y si no lo indicamos, supondremos que cualquier variable está definido dentro del él.

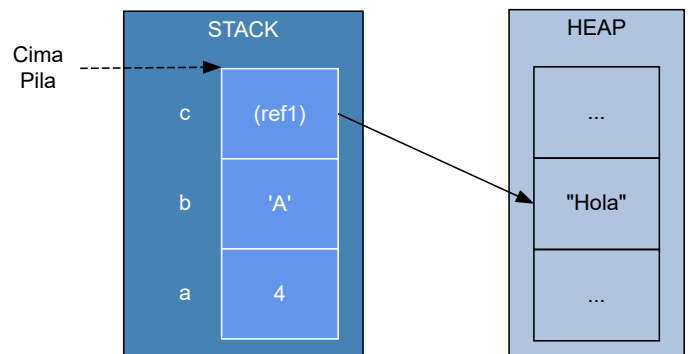
Supongamos el siguiente código y vemos como se comporta la memoria.

```
1  class Program
2  {
3      static void Main()
4      {
5          int a = 4;           // Valor
6          char b = 'A';       // Valor
7          string c = "Hola";  // Referencia
8
9          // Bloque anidado.
10         {
11             int d = 8;       // Valor
12             string e = "Adios"; // Referencia
13             string f = c;    // Referencia
14         }
15     }
16 }
```

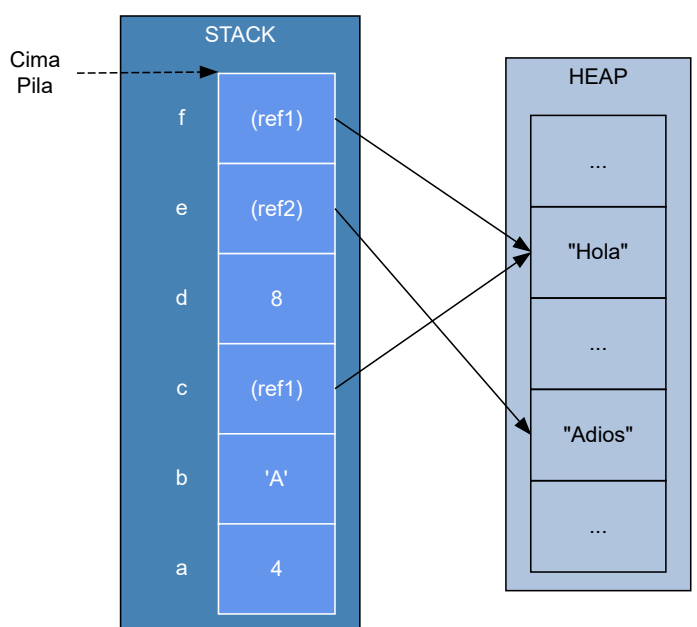
👉 **Importante:** Tras ver este ejemplo, repasa los puntos de las definiciones de tipo valor y referencia y trata de asociarlos con el mismo.

Como se ve en el esquema de memoria, tras entrar en el bloque del **Main** (línea 4), se apilan en el **Stack** las declaraciones de las líneas 5, 6 y 7. Pero **c** al ser un tipo referencia, apunta a donde se encuentre el dato en el **Heap**.

El bloque se cierra en la línea 15. Hasta ahí será el **scope** (alcance) o **ámbito** de las variables y al salir de él bloque se 'desapilarán'.



En la línea 10 se abre un nuevo bloque y se vuelven a apilar las declaraciones. La variable **e** al ser un tipo referencia su dato se almacenará en el **Heap**. Sin embargo **f** tomará el valor de **c** y por tanto ambas referenciarán o 'apuntarán' al mismo sitio en el **Heap**.

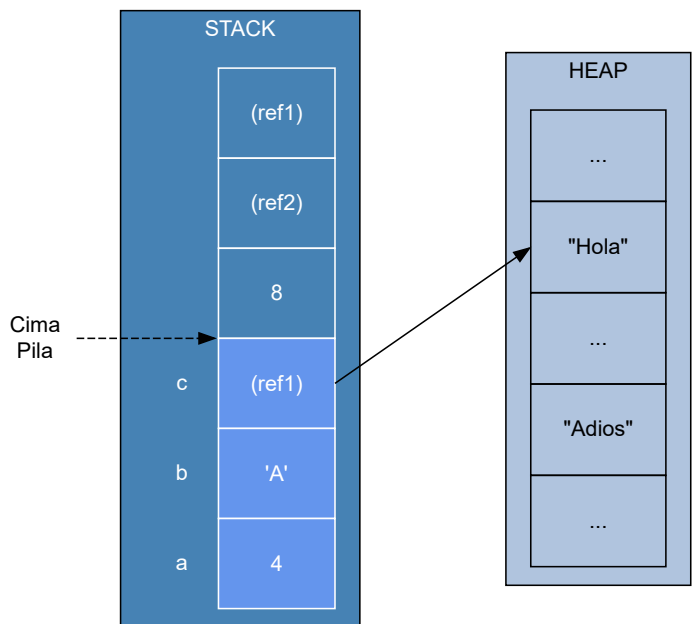


En la **línea 14** se cierra el **bloque anidado** y por tanto acaba el **scope** o **ámbito** de las variables declaradas dentro del mismo. Sucediendo lo siguiente...

1. Se desapilan del **Stack** las variables **d**, **e** y **f** pasando a apuntar la cabeza de la pila a **c**.

🚩 **Nota:** Realmente no se *'borran'*, simplemente el espacio que ocupaban en la pila queda libre para ser usado en futuras declaraciones.

2. Puesto que el espacio que ocupa el dato **"Adios"** en el **Heap** ya no es referenciado por ninguna referencia en el **Stack**, este será liberado en el futuro por el recolector de basura GC para ser usado más adelante. Sin embargo no sucede lo mismo con **"Hola"** que aún sigue referenciado por la variable **c**.



En la **línea 15** se cierra el **bloque del Main** y por tanto acaba el **scope** o **ámbito** de las variables declaradas dentro del mismo. Sucediendo lo mismo que antes...

1. El **Stack** se vacía, actualizando la la cabeza de la misma.
2. El dato en el **Heap** **"Hola"** referenciado por **c** deja de estarlo y por tanto será eliminado en la siguiente pasada del GC.

