Unidad 12

Descargar estos apunte en pdf o html

Índice

Tipos valor vs Tipos referencia en POO

Introducción

Muchos lenguajes de programación utilizan tanto **tipos por valor y tipos por referencia** para gestionar la memoria y el comportamiento de las variables. La diferencia fundamental radica en cómo se almacena y se pasa la información. Básicamente podemos decir que:

- **Tipos por Valor (Value Types)**: La variable contiene directamente el dato. Al asignarla o pasarla a una función, se crea una copia independiente.
- **Tipos por Referencia (Reference Types)**: La variable contiene una "referencia" a la ubicación en memoria donde se almacena el dato. Al asignarla o pasarla, se copia la referencia, pero ambas apuntan al mismo dato original.

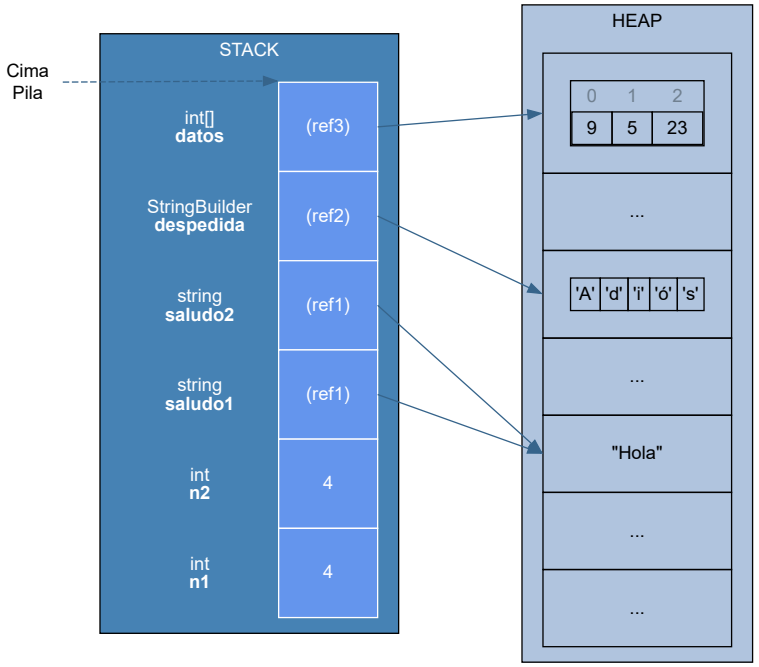
Es importante entender cómo funcionan y cuándo se utilizan cada uno de ellos. Pues la forma de pasar datos, el manejo de la memoria, la eficiencia y la robustez del código pueden variar significativamente entre ambos tipos. En el siguiente diagrama, podemos ver la diferencia de gestión de la que hemos ido tratando en las unidades anteriores...

```
public static void Main()
{
    // Tipo valor
    int n1 = 4;
    // Copia del valor de n1
    int n2 = n1;

    // Tipo referencia
    string saludo1 = "Hola";
    // Copia de la referencia a saludo1
    string saludo2 = saludo1;

    // Objeto tipo referencia
    StringBuilder despedida = new("Adiós");

    // Tipo referencia
    int[] datos = [9, 5, 23];
}
```



> **Nota**
>
> Para ver la diferencia en entre Pila (Stack) y Montón (Heap) puede ver o repasar el **Anexo I de la unidad 3**, donde tratábamos el tema.

En **CSharp**, Además de los **tipos primitivos que hemos usado** en las primeras unidades cómo `int`, `double`, `char`, `bool`, etc (Todos excepto `string`). C# define una serie de tipos valor más complejos como `DateTime`, `TimeOnly`, `DateOnly`, `TimeSpan`, `Guid`, `ValeTuple`, `Enum`, etc. Además, **no es habitual definir tipos valor personalizados**, ya que la mayoría de las veces se usan los tipos predefinidos ya mencionados.

Debemos destacar que en muchos lenguajes de programación no se pueden definir tipos valor personalizados. Sin embargo, más adelante ahondaremos en el concepto de uso, características y definición desde el punto de vista de la POO moderna en C# y otros lenguajes.

Tipos valor relacionados con fechas y tiempo

Uno de los objetos valor predefinidos en las BCL y que más vamos usar, es el tipo de dato **fecha**: **System.DateTime**. Son **objetos inmutables** que representan un instante en el tiempo. El objeto `DateTime` almacena la fecha y la hora, y puede representar fechas desde el 1 de enero del año 0001 hasta el 31 de diciembre del año 9999.

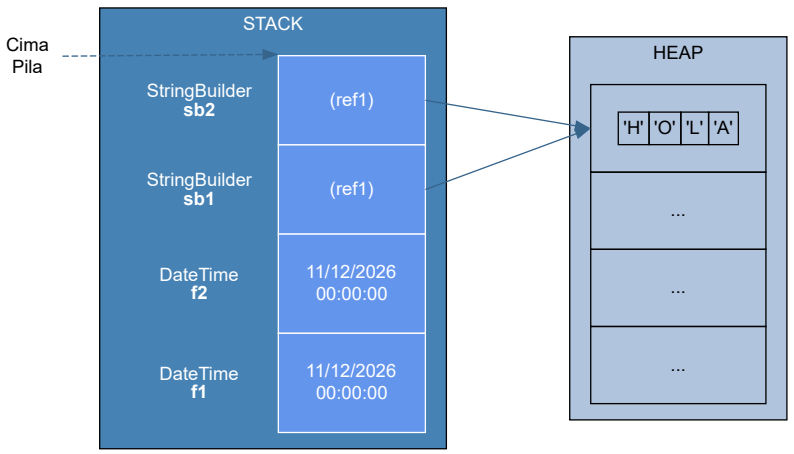
Formas de instanciar objetos valor fecha

```
DateTime f1 = new (2026, 3, 19);             // 19/03/2026 a las 00:00:00
DateTime f2 = new (2026, 3, 19, 8, 30, 00);  // 19/03/2026 a las 8:30:00
DateTime f3 = DateTime.Today;                // Hoy a las 00:00:00
DateTime f4 = DateTime.Now;                  // Hoy a a la hora actual en mi zona horaria.
DateTime f5 = DateTime.UtcNow;               // Hoy a a la hora actual en zona UTC.
```

Al **compararlo con la instancia y asignación de un tipo referencia** como `StringBuilder`. El comportamiento en memoria es diferente...

```
DateTime f1 = new(2026, 12, 11);
DateTime f2 = f1;

StringBuilder sb1 = new("HOLA");
StringBuilder sb2 = sb1;
```



Operaciones más comunes con fechas

Podemos usar todos los operadores de comparación `==, != , > , <, >= y <=`

En las operaciones de suma `+` y resta `-`, interviene otro tipo de dato valor denominado **TimeSpan** que representará un periodo de tiempo que podremos representar en las siguiente magnitudes `años, meses, días, horas, minutos, segundos` según la precisión que necesitemos.

```
DateTime f1 = new (year: 2026, month: 3, day: 19);
 // Periodo de tiempo con precisión de 25 días
TimeSpan periodo = new (days: 25, hours: 0, minutes: 0, seconds: 0);

DateTime f2 = f1 + periodo;

// Muestra Del 19/03/2026 al 13/04/2026 hay 25 días
Console.WriteLine($"Del {f1.ToShortDateString()} al {f2.ToShortDateString()} hay {(f2 - f1).Days} días");
```

Estas operaciones también las podremos hacer, a través de diferentes métodos `f.AddHours(...)`, `f.AddDays(...)`, etc. y así evitarnos usar `TimeSpan`

```
DateTime f1 = new (year: 2026, month: 8, day: 1);

// Recuerda el método no cambia el estado de f1,
// sino que devuelve un objeto valor fecha que copiamos en f2
DateTime f2 = f1.AddDays(25);

// Muestra: "01/08/2026"
Console.WriteLine(f1.ToShortDateString());
// Muestra: "miércoles, 26 de agosto de 2026"
Console.WriteLine(f2.ToLongDateString());
```

Fechas y cadenas

Una aproximación sencilla es usar las propiedades del objeto para dar el formato:

```
DateTime f = new (year: 2026, month: 3, day: 19);
Console.WriteLine($"{f.Day:D2}-{f.Month:D2}-{f.Year:D4}"); // Muestra 19-03-2020
```

Pero el lenguaje provee una sintaxis especial para cadenas de fecha y hora personalizadas de DateTime a string y viceversa. Usaremos los siguientes **formatos definidos por el lenguaje**.

Aquí dispones un **cuadro resumen** de los principales formatos descritos en el enlace:

Formato	Descripción	Formato	Descripción
d	Día del mes con el mínimo de dígitos	**/**	Separador fecha según el país
dd	Día del mes con 2 dígitos	**gg**	Indicar la era según el país AC/DC
ddd	Día de la semana abreviado según el país	**h**	Hora de 0 a 12 horas
dddd	Día de la semana completo según el país	**hh**	Hora de 00 a 12 horas
M	Mes con el mínimo de dígitos	**z**	Indicador Zona Horaria
MM	Mes con 2 dígitos	**H**	Hora de 0 a 23 horas
MMM	Nombre del mes abreviado según el país	**HH**	Hora de 00 a 23 horas
MMMM	Nombre del mes completo según el país	**mm**	Minutos con 2 dígitos
yyyy	Año con 4 dígitos	**ss**	Segundos con 2 dígitos

> **Nota**
>
> Aquellos caracteres que tienen un significado especial como **d**, **h**, **M**, etc. si queremos que se representen tal cual y no sean sustituidos por un valor de DateTime deberemos escaparlos con `\\`. Por ejemplo, `\\d` es el caracter `d` y no el día del mes en formato numérico.

```
DateTime f = new (year: 2026, month: 3, day: 19, hour: 19, minute: 0, second: 0, kind: DateTimeKind.Utc);
// Muestra: "19-03-2026"
Console.WriteLine(f.ToString("dd-MM-yyyy"));
// Muestra: "jueves 19 de marzo de 2026 a las 19 horas UTC+0"
Console.WriteLine(f.ToString(@"dddd d \de MMMM \de yyyy a la\s H \hora\s UTCz"));
```

Fíjate que en el ejemplo anterior hemos usado `DateTimeKind.Utc` para indicar que la fecha es en formato UTC. Esto es importante **si queremos hacer operaciones con fechas y horas en diferentes zonas horarias**. **UTC** es el **Tiempo Universal Coordinado** y es la referencia estándar para todas las zonas horarias del mundo, vendría a representar la **hora en el meridiano de Greenwich** (GMT+0). Si

no especificamos el tipo de fecha, por defecto se usará `DateTimeKind.Unspecified` y el lenguaje no sabrá si la fecha es UTC o local. El otro tipo de fecha que podemos usar es `DateTimeKind.Local` que indica que la fecha es en la **zona horaria local del sistema**.

Para transformar un DateTime Local a UTC, podemos usar el método `ToUniversalTime()`.
Por ejemplo:

```
// DateTime.Now() devuelve la fecha y
// hora actual en la zona horaria local del sistema.
DateTime fechaUtc = DateTime.Now.ToUniversalTime();

// Equivaldría ha haer lo siguiente...
DateTime fechaUtc2 = DateTime.UtcNow;
```

Tendremos también la conversión en sentido inverso si hacemos un Parse. Aunque en este último caso **también podemos utilizar ER**.

```
public static void Main()
{
    while (true)
    {
        Console.Write("Introduce una fecha con formato (dd/MM/yyyy): ");
        string texto = Console.ReadLine();
        if (DateTime.TryParseExact(texto, "dd/MM/yyyy", null,
                                   System.Globalization.DateTimeStyles.AllowWhiteSpaces
                                   | System.Globalization.DateTimeStyles.AdjustToUniversal,
                                   out DateTime fecha))
            Console.WriteLine($"Has introducido {fecha.ToShortDateString()}");
        else
            Console.WriteLine($"{texto} es un fecha incorrecta.");
    }
}
```

¿Qué pasa si solo queremos guardar la hora o la fecha?

AL igual que sucede en otros lenguajes `DateTime` me obliga a guardar una fecha asociada si solo quiero guardar una hora. Para solucionar esto, disponemos también de los tipos valor `TimeOnly` y `DateOnly` similares a `DateTime` pero solo guardaremos u hora o fecha respectivamente.

Debemos tener en cuenta que estos tipos valor también son **inmutables** y no permiten modificar su estado una vez creados. Además, para crearlos **a partir de los datos de fecha y hora actuales** tanto local como UTC, podemos usar los métodos `TimeOnly.FromDateTime(DateTime)` y `DateOnly.FromDateTime(DateTime)` y al revés, para convertirlos a `DateTime` podemos usar los métodos `TimeOnly.ToDateTime()` y `DateOnly.ToDateTime()`.

Ejemplo:

Vamos a crear un programa que muestre la hora actual en diferentes zonas horarias del mundo de forma aproximada sin tener en cuenta horarios de verano o cambios de hora. Para ello, usaremos el tipo valor `TimeOnly` para representar la hora local y las horas en diferentes ciudades del mundo.

```
public class Program
{
    public static void Main()
    {
        (string, int)[] horarios = [
            ("Los Ángeles", -7), // UTC-7
            ("Nueva York", -4),  // UTC-4
            ("Londres", 0),      // UTC+0
            ("Berlín", 2),       // UTC+2
            ("Moscú", 3),        // UTC+3
            ("Pekín", 8),        // UTC+8
        ];

        TimeOnly horaLocal = TimeOnly.FromDateTime(DateTime.Now);
        Console.WriteLine($"Hora local: {horaLocal:HH:mm:ss}");

        // Para trabajar con zonas horarios lo hago respecto a UTC
        TimeOnly horaUtc = TimeOnly.FromDateTime(DateTime.UtcNow);
        foreach (var (ciudad, offset) in horarios)
        {
            TimeOnly horaCiudad = horaUtc.AddHours(offset);
            Console.WriteLine($"En {ciudad} son las {horaCiudad:HH:mm:ss}");
        }
    }
}
```

Tipo valor GUID

Otro tipo valor que vamos a usar mucho es `Guid`. Un **GUID (Globally Unique Identifier)** es un identificador único global que se utiliza para identificar de forma única objetos, registros o entidades en sistemas distribuidos. En C#, el tipo `Guid` es un tipo valor que representa un GUID y proporciona métodos para crear, comparar y manipular GUIDs.

Está compuesto por 128 bits (16 bytes) y se representa comúnmente como una cadena de 32 caracteres hexadecimales, divididos en cinco grupos separados por guiones. Por ejemplo: `bce3fec8-e3d2-4b77-af17-56f38dde4589`.

En C# se puede generar un GUID de la siguiente manera:

```
// Para generar un GUID, usamos el método estático NewGuid()
Guid guid = Guid.NewGuid();

Console.WriteLine($"Versión del GUID: {guid.Version}");
Console.WriteLine($"Variante del GUID: {guid.Variant}");

// "N" es el formato de 32 caracteres sin guiones
string textoGuidSinSeparar = guid.ToString("N");
Console.WriteLine($"GUID sin separar: {textoGuidSinSeparar}");

// "D" es el formato de 32 caracteres con guiones
// Es el formato por defecto
string textoGuidSeparadoPorGuiones = guid.ToString("D");
Console.WriteLine($"GUID separado por guiones: {textoGuidSeparadoPorGuiones}");
```

Mostrará por consola algo como:

```
Versión del GUID: 4
Variante del GUID: 2
GUID sin separar: bce3fec8e3d24b77af1756f38dde4589
GUID separado por guiones: bce3fec8-e3d2-4b77-af17-56f38dde4589
```

Fíjate que el GUID tiene una **versión**. La versión está indicada en el propio GUID y se describen en el estándar **RFC 4122 (Version)**.