

# Tema 9.5

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

## Índice

- ▼ [Clases parametrizadas o genéricos](#)
  - ▼ [Clases parametrizadas en C#](#)
    - [Inicializar un dato genérico a su valor por defecto](#)
    - [Consideraciones especiales](#)
    - [Ejemplo de clase parametrizada](#)
    - [Métodos parametrizados](#)
    - [Tipos parametrizados en la BCL](#)
    - [Definiendo restricciones en tipos parametrizados](#)

# Clases parametrizadas o genéricos

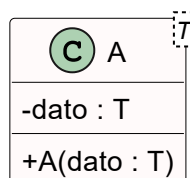
Hasta ahora, cuando queríamos referirnos a un objeto de forma '**genérica**' nos referenciábamos a través de la clase **object**, sin embargo este tipo de generalización no nos asegura, hasta hacer el downcasting, que el tipo de objeto referenciado es del tipo que esperábamos. Esto puede ser fuente de errores y lo peor aún, **no serían detectados hasta el momento de la ejecución**, puesto que el programa compilaría correctamente.

Además de esto, en ocasiones se nos darán clases o métodos cuya funcionalidad y lógica es idéntica cambiando únicamente uno o más tipos usados. En estos casos se nos generará **código prácticamente repetido**, donde cambian únicamente algunos tipos usados. Esta situación es poco deseable y necesitaremos de algún mecanismo para hacer definir clases o métodos '*illa*' donde se defina la lógica y funcionalidad a falta de concretar los tipos.

Para solucionar ambos problemas muchos lenguajes orientados a objetos, incluido C# desde sus versiones mas tempranas, nos permiten **definir los tipos dentro de una clase de forma parametrizada al instanciar un objeto de la misma**. Expresaremos el tipo o los tipos genéricos a través de una o más letras mayúsculas usadas a lo largo de la definición de la clase. Aunque se suele usar la letra **T**, podremos usar cualquier otra que nos represente el tipo parametrizado. A estas letras se les denomina **parámetros tipo** y podremos usarlas en clases, métodos, interfaces, delegados, etc...

A través de este tipo de definiciones, se aparecerá un nuevo tipo de polimorfismo en la POO, denominado **polimorfismo paramétrico** y lo definiremos como aquel que nos permite definir el tipo dentro de una clase de forma parametrizada, al instanciar un objeto de la misma. De tal manera que, para objetos diferentes, el tipo con el que se instancia podrá cambiar.

La forma de representar el tipo parametrizado en los diagramas de clases UML, es a través de un recuadro en la parte superior derecha de la definición.



## Clases parametrizadas en C#

Veamos como se define en C# para acabar de dar forma al concepto.

- **Definición:** Definiremos los tipos genéricos **justo después del identificador** de la clase, entre `< >`.

```
public class A<T> // Un parámetro genérico
{
    private T dato;
    public A(T dato) {
        this.dato = dato;
    }
}
```

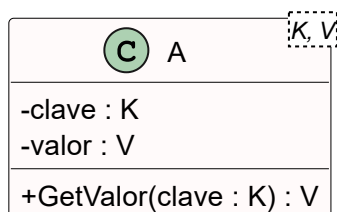
- **Uso:** Cuando yo instancie un objeto de la clase genérica A ...

```
A<int> objA = new A<int>(4);
```

En tiempo de ejecución C# construirá una objeto de la clase **A** como sustituyendo el tipo parametrizado por el que le estamos indicando en el momento de la instanciación. En este caso un **int** de la forma siguiente ...

```
public class A
{
    private int dato;
    public A(int dato) {
        this.dato = dato;
    }
}
```

Si vamos a usar **más de un tipo a parametrizar**. Los separaremos por comas.



```
public class A<K, V> // Dos parámetros genéricos
{
    private K clave;
    private V valor;
    public V GetValor(K clave) { ; }
}
```

## Inicializar un dato genérico a su valor por defecto

```
class A<T>
{
    T dato = null; // Es correcto ¿?
}
```

En principio no sabemos si el tipo que le vamos a indicar a la clase es valor o referencia. Por lo que deberíamos usar la expresión `default(T)`

```
class A<T>
{
    T dato = default(T);
}
```

## Consideraciones especiales

1. No podremos usar los parámetros tipo (`T`, `U`, `K`, etc..) como nombre o identificadores de clases, interfaces, atributos.
2. Deberemos llevar cuidado con el **polimorfismo funcional**. Por ejemplo supongamos la siguiente definición...

```
class A<T> {
    public void IdMétodo(int p1, string p2) { ; }
    public void IdMétodo(T p1, string p2) { ; }
}
```

Si instanciamos `A` de la siguiente forma ...

```
A<string> obj = new A<string>();
```

sería correcto y tendríamos dos signaturas. Una con `p1` como `int` y otra con `p1` como `string`. Pero ... **¿Qué pasa si declaramos?**

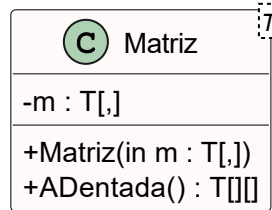
```
A<int> obj = new A<int>();
```

En este caso tendremos **dos signaturas iguales** y aunque no se produzca error, cuando llamemos a `IdMétodo` se ejecutará la **no genérica**.

3. A un objeto declarado a partir de un parámetro tipo:
  - En principio, solo podremos aplicarle operaciones como si fuera de tipo `object` y **no podremos usar operadores**.
  - No podremos realizarle un cast explícito, sin pasar previamente a `object`.

## Ejemplo de clase parametrizada

Supongamos una clase **Matriz** que haga de envoltorio o *'Wrapper'* sobre una matriz bi-dimensional de **cualquier tipo**. Esta clase nos permitirá definir operaciones de utilidad para este tipo de matrices. Por ejemplo, su transformación a una tabla dentada equivalente.



```
1  class Matriz<T>
2  {
3      private readonly T[,] m;
4      public Matriz(T[,] m)
5      {
6          this.m = m;
7      }
8      public T[][] ADentada()
9      {
10         T[][]? d = null;
11         d = new T[m.GetLength(0)][];
12         for (int i = 0; i < d.Length; i++)
13         {
14             d[i] = new T[m.GetLength(1)];
15             for (int j = 0; j < d[i].Length; j++)
16                 d[i][j] = m[i, j];
17         }
18         return d;
19     }
20 }
21 class Ejemplo
22 {
23     static void Main()
24     {
25         // Aquí la matriz a envolver será de enteros.
26         Matriz<int> m1 = new Matriz<int>(new int[,] { {1, 2, 3}, {3, 2, 1} });
27         int[][] d1 = m1.ADentada();
28
29         // Aquí la matriz a envolver será de caracteres.
30         Matriz<char> m2 = new Matriz<char>(new char[,] { {'a'}, {'b'}, {'c'} });
31         char[][] d2 = m2.ADentada();
32     }
33 }
```

## Métodos parametrizados

C# permite definir no sólo clases genéricas, sino que también puede hacerse genéricos **métodos individuales**, tanto de instancia como estáticos, sin necesidad de que lo sea la clase o estructura en la que el método está definido.

La sintaxis para parametrizar un método será análoga a la que hemos usado para las clases.


En la mayoría de los casos, este tipo de parametrización de métodos, tendrá más sentido con métodos de utilidad estáticos. Ya que como operaciones sobre un objeto no será *'inmediato'* relacionarlos con los tipos de los campos que definen el estado de la clase.

```
// La clase puede o no estar parametrizada.
static class A
{
    public static void Metodo<T>(T parametro)
    {
        // Podremos usar el tipo genérico T tanto en
        // los parámetros formales, como en el cuerpo
        // del método.
    }
    public static void Metodo(int parametro)
    {
        // Para C# aunque este método tenga el mismo id
        // que el anterior, tendrán signaturas diferentes
        // y por tanto sabrá distinguirlos como diferentes.
    }
}
```

## Ejemplo de método parametrizado

Supongamos la clase **Matriz** del ejemplo anterior pero esta vez será **estática** y contendrá métodos de utilidad parametrizados para matrices bi-dimensionales.

Si quisiéramos hacer un método **T[][] ADentada<T>(T[,] m)** equivalente al del ejemplo anterior, ahora tendríamos ...

 «static» Matriz
<u>+ADentada&lt;T&gt;(in m:T[,]); T[][]</u>

```
1  static class Matriz
2  {
3      static public T[][] ADentada<T>(T[,] m)
4      {
5          T[][]? d = null;
6          d = new T[m.GetLength(0)][];
7          for (int i = 0; i < d.Length; i++)
8          {
9              d[i] = new T[m.GetLength(1)];
10             for (int j = 0; j < d[i].Length; j++)
11                 d[i][j] = m[i, j];
12         }
13         return d;
14     }
15 }
16 class Program
17 {
18     static void Main()
19     {
20         // En este caso indicamos el tipo la matriz de entrada y tabla de salida
21         // indicándolo el tipo en la llamada del método.
22         int[][] d1 = Matriz.ADentada<int>(new int[,]{ {1, 2, 3}, {3, 2, 1} });
23
24         // También podremos hacer la llamada obviando indicar el tipo parametrizado
25         // ya que C# lo infiere de la matriz de entrada.
26         char[][] d2 = Matriz.ADentada(new char[,]{ {'a'}, {'b'}, {'c'} });
27     }
28 }
```

## Tipos parametrizados en la BCL

Supongamos que tenemos la clase `Hora` que implementa el interfaz `IComparable` que vimos anteriormente en el tema. Recordemos que esto supondría que podremos pasar cualquier objeto, pues todos heredan de `object`. Por esta razón, deberemos hacer un downcast del objeto al comparar a `Hora` pero no se puede asegurar que el objeto que estamos comparando es una hora, por tanto deberemos añadir algún tipo de código de control de errores...

```
class Hora : IComparable
{
    public int H { get; }
    public int M { get; }
    public Hora(int h, int m)
    {
        H = h;
        M = m;
    }
    public override string ToString() => $"{H:D2}:{M:D2}";

    public int CompareTo(object? objHora)
    {
        // Nadie nos asegura que el downcast se pueda realizar.
        Hora hora = objHora as Hora
        ?? throw new ArgumentException("El objeto a comparar no es una hora.", "obj");

        int comparacion = H - hora.H;
        if (comparacion == 0)
            comparacion = M - hora.M;
        return comparacion;
    }
}
```

Sin embargo, C# añadió una definición parametrizada para dicho interfaz `IComparable` y si la utilizamos, nos avisará en tiempo de compilación de que no estamos pasando el tipo correcto y además, no necesitaremos hacer el repetitivo código de control anterior.



```

class Hora : IComparable<Hora>
{
    // ... código omitido para abreviar.

    // Ahora tenemos seguridad de que nos llega una hora.
    public int CompareTo(Hora? hora)
    {
        // Esta comprobación solo la haríamos si no quisieramos comparar con valores null.
        // if (hora == null)
        //     throw new ArgumentNullException("La hora debe ser distinto de null.");
        int comparacion = (hora == null) ? 1 : H - hora.H;
        if (comparacion == 0 && hora != null)
            comparacion = M - hora.M;
        return comparacion;
    }
}

```

Como hemos comentado, dispondremos de métodos estáticos de utilidad que estarán parametrizados, definiendo los tipos de entrada en el momento de su utilización.

Por ejemplo recordemos el método estático para ordenar arrays **Sort**. Tendremos una versión parametrizada con el siguiente **interfaz** `public static void Sort<T>(T[] array);`

Por ejemplo, podríamos usar este método con un array de instancias de nuestra clase **Hora**, que además sabríamos como ordenarlo puesto que la clase implementa el interfaz **IComparable<Hora>**.

```

static void Main()
{
    Hora[] horas =
    {
        new Hora(9, 55), new Hora(10, 50),
        new Hora(8, 30), new Hora(7, 15)
    };

    Array.Sort<Hora>(horas);
    foreach (var h in horas) Console.Write($"{h} ");

    // También sería posible esta llamada donde el tipo
    // parametrizado se inferirá del parámetro de entrada.
    Array.Sort(horas);
}

```

## Definiendo restricciones en tipos parametrizados

Podremos definir restricciones de tipo asociadas a su definición para los parámetros tipo. Se especifican con la palabra reservada **where** al final de la definición.

```
<T> where T : restricción
```

```
class A<T> where T : struct
{
    // T sólo podrán ser tipos valor.
}
```

Tendremos diferentes tipos de restricciones entre las que podemos destacar las siguientes ...

Tipo Restricción	Descripción
De herencia	El tipo debe heredar de una clase base determinada. <code>&lt;T&gt; where T : ClaseBaseDeT</code>
De interfaz ★	El tipo debe implementar una <b>interfaz</b> determinada. <code>&lt;T&gt; where T : IinterfazAImplementar</code>
De tipo referencia	El tipo debe ser <b>referencia</b> . <code>&lt;T&gt; where T : class</code>
De tipo valor	El tipo debe ser <b>valor</b> . <code>&lt;T&gt; where T : struct</code>
De constructor	El tipo debe tener un constructor sin parámetros. <code>&lt;T&gt; where T : new</code>

Veamos un primer **ejemplo** de restricción de tipo con un método parametrizado **Swap** que me permitirá intercambiar **referencias** a objetos del mismo tipo.

```
public static void Swap<T>(ref T c1, ref T c2)
{
    T aux = c1;
    c1 = c2;
    c2 = aux;
}
```

Si no le indico nada me permitirá intercambiar tanto tipos **referencia** como tipos **valor**.

```
static void Main()
{
    var h1 = new Hora(9, 55);
    var h2 = new Hora(10, 50);
    int e1 = 2;
    int e2 = 6;

    Console.WriteLine($"h1 = {h1} h2 = {h2}\ne1 = {e1} e2 = {e2}");
    Swap(ref h1, ref h2); // Tipos referencia.
    Swap(ref e1, ref e2); // Tipos valor.
    Console.WriteLine($"h1 = {h1} h2 = {h2}\ne1 = {e1} e2 = {e2}");
}
```

Sin embargo, si ahora restringimos el tipo genérico a solo tipos referencia con **where T: class**

```
public static void Swap<T>(ref T c1, ref T c2) where T: class { ... }

static void Main()
{
    ...
    // Ahora la siguiente línea ...
    Swap(ref e1, ref e2);
    // Me producirá el error ...
    // El tipo 'int' debe ser un tipo de referencia para poder
    // usarlo como parámetro 'T' en el tipo o método genérico
    // 'Program.Swap<T>(ref T, ref T)' csharp(CS0452)
}
```

## Ejemplo restricción en tipos parametrizados

Vamos a recuperar la clase **class Matriz<T>** y a usarla junto a la clase **Hora : IComparable<Hora>**. Para ello, vamos a invalidar el método de la clase **object bool Equals(object obj)** que me devolverá **true** si todos los elementos de la matriz contenida son iguales a los de la que me llegan para comparar. Devolviendo **false** en caso contrario.

Pero... ¿Cómo comparo los elementos de de la matriz si son de tipo **T**?

Lo que haremos es añadir la restricción de que los **T** con se instancie **Matriz<T>** implementen el interfaz **IComparable<T>** de la siguiente manera:

```

class Matriz<T> where T : IComparable<T>
{
    private readonly T[,] m;
    public Matriz(T[,] m) { this.m = m; }
    // ... código omitido por abreviar.
    public override int GetHashCode() => String.Join("", m).GetHashCode();

    public override bool Equals(object obj)
    {
        bool iguales = obj != null; // Si comparo con null no son iguales.
        if (iguales)
        {
            Matriz<T> m = obj as Matriz<T> ?? throw new();

            for (int i = 0; i < m.m.GetLength(0) && iguales; i++)
                for (int j = 0; j < m.m.GetLength(1) && iguales; j++)
                    // Fíjate que al añadir la restricción, estoy añadiendo
                    // funcionalidad para mi tipo T, porque ahora se que voy a
                    // disponer para todos ellos del método CompareTo a parte de los
                    // básicos por heredar de object.
                    iguales = m.m[i, j].CompareTo(this.m[i, j]) == 0;
        }
        return iguales;
    }
}

```

Ahora, además de poder crear matrices de `int` o `string` que implementan `IComparable<...>`, podré crear matrices de objetos `Hora` ya que dicha clase también implementa dicho interfaz. Sin embargo, al crearla de cualquier otro tipo que no lo implemente, obtendremos un error.

```

static void Main()
{
    Matriz<Hora> m1 = new Matriz<Hora>(new Hora[,]
    {
        { new Hora(0, 0), new Hora(0, 30) },
        { new Hora(12, 0), new Hora(12, 30) },
        { new Hora(18, 0), new Hora(18, 30) }
    });
    Matriz<Hora> m2 = new Matriz<Hora>(new Hora[,]
    {
        { new Hora(0, 0), new Hora(0, 30) },
        { new Hora(12, 0), new Hora(12, 30) },
        { new Hora(18, 0), new Hora(18, 30) }
    });
    Console.WriteLine(m1.Equals(m2));
}

```