Tema 9.3

Descargar estos apuntes

Índice

Profundizando en la Programación Orientada a Objetos

Interfaces

Básicamente un Interfaz es la definición de un conjunto de interfaces de métodos, accesores o mutadores (como **Propiedades**), indizadores, etc. Es muy parecido a definir una **clase abstracta pura**, pero **sin ningún tipo de atributo** o campo, constructor, ni modificador de acceso (public, private, etc…). Como en las clases abstractas, las interfaces son tipos referencia, no puede crearse objetos de ellas sino sólo de tipos que deriven de ellas, y participan del polimorfismo.

Pueden implementarse en muchos lenguajes OO con idénticas características:

- Es posible la **herencia múltiple** de interfaces.
- No pueden definir atributos pero sí propiedades.
- Un interfaz **puede heredar de otro interfaz**.
- Si una clase hereda de un interfaz. Esta, deberá invalidar todo lo que hayamos definido en el mismo.

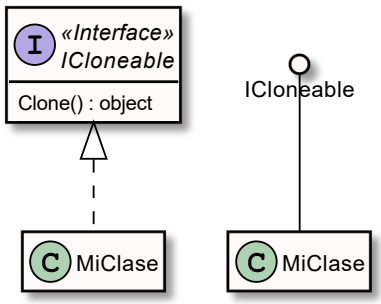
Podemos resumir diciendo que es la forma más recomendable y común de **definir la abstracción de un comportamiento**.

Interfaces en los diagramas de clases UML

Para expresar que la clase `MiClase` **implementa** el interfaz `ICloneable`.

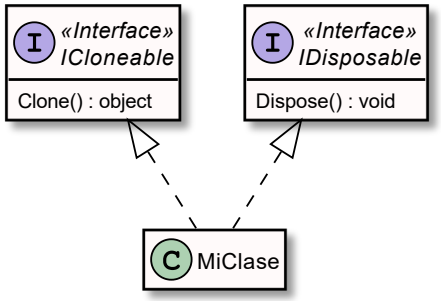
> 📌 **Nota:** Usamos la palabra **implementa** en lugar de "*hereda de*" ya que, como hemos comentado, más que responder `MiClase` a la pregunta "*es un*", un interfaz define un comportamiento abstracto que `MiClase` deberá implementar.

Podremos expresarlo de las forma siguientes formas ...



`MiClase` ahora está obligada a **implementar** el método público `Clone` con **idéntica signatura**.

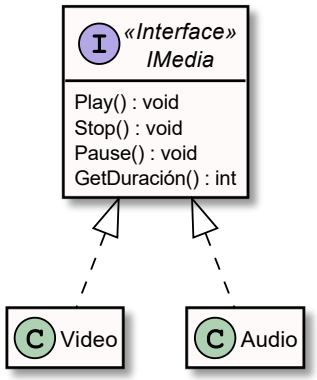
Podremos hacer que una clase **implemente** o "*herede*" de más de un interfaz.



Interfaces en C#

Como ya habrás podido apreciar en el diagrama anterior, según el convenio de nomenclatura de C#, el identificador o nombre de la clase irá siempre precedido por la letra mayúscula `I` (`I nterface`) para distinguirlo de otro tipo de clases.

```
<modificadores> interface I<identificador> : <interfacesBase>
{
    <interfaces de métodos, propiedades o indizadores>
}
```




```
interface IMedia
{
    void Play();
    void Stop();
    void Pause();
    int Duración { get; }
}
```

Para aplicar un interfaz a una clase. Haremos que esta herede del interfaz con la sintaxis de herencia que hemos usado hasta ahora.

```
class Video : IMedia
{
    // ...

    public int Duración => 0;
    public void Pause() => Console.WriteLine("Pausando el vídeo.");
    public void Play() => Console.WriteLine("Reproduciendo el vídeo.");
    public void Stop() => Console.WriteLine("Parando el vídeo.");
}
```

Interfaces de utilidad predefinidos en las BCL

Podemos decir que me permiten definir comportamientos para mis própios tipos, que serán reconocidos por otras clases o tipos ya implementadas en las BCL.

> 📌 **Nota:** Podríamos utilizar interfaces própios para hacer lo mismo, **pero perderíamos interoperabilidad con el resto de clases de las BCL**.

IEnumerable

Lo veremos más adelante, al usar o definir colecciones.

ICloneable

Me indicará que puedo crear copias del objeto, puesto que me obliga a implementar un "*constructor copia*" con el interfaz `object Clone()` el cual me permitirá hacer **copias en profundidad** de objetos de tipo referencia.

IComparable

Me indicará que el objeto debe implementar el método `int CompareTo(Object otro)` que me servirá para comparar dos objetos de la misma clase y que ya usamos en el **tema 7** para comparar cadenas.

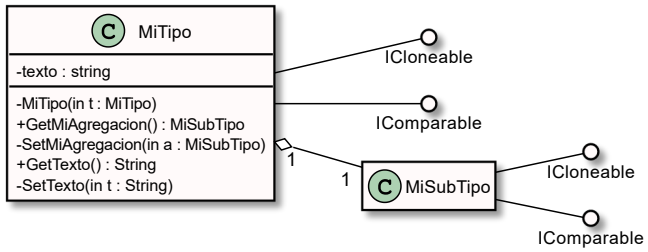
> 📌 **Nota:** Recordemos brevemente que...

```
Tipo o1 = new Tipo(...);
Tipo o2 = new Tipo(...);

// Si Tipo es IComparable entonces ...
int comparacion = o1.CompareTo(o2);

// comparacion = 0 si o1 y o2 son iguales.
// comparacion > 0 si o1 > o2.
// comparacion < 0 si o1 < o2.
```

Veamos un ejemplo "*genérico*" comentado el uso de este tipo de interfaces. Para ello, supongamos la siguiente agregación con tipos definidos por el usuario, donde ambos implementan los interfaces `ICloneable` e `IComparable`.



Vamos a suponer que la clase `MiSubTipo` ya la hemos definido e implementa ambos interfaces. Una implementación '*genérica*' posible para `MiTipo` podría ser ...

```
class MiTipo : ICloneable, IComparable
{
    // Propiedad autoimplementada que representa la agregación del UML.
    public MiSubTipo MiAgregacion { get; private set; }
    // Propiedad autoimplementada que representa el atributo texto con su
    // accesor y mutador
    public string Texto { get; private set; }
    // Constructor copia privado que usaremos para hacer el clonado
    // en profundidad.
    private MiTipo(MiTipo t)
    {
        // Debo hacer un downcasting porque Clone() me devuelve un object.
        MiAgregacion = t.MiAgregacion.Clone() as MiSubTipo;
        // No hay problema en igualar los string porque es inmutable.
        Texto = t.Texto;
    }
    // Estoy forzado a implementarlo por ser ICloneable
    // Llamamos al constructor copia mediante un cuerpo de expresión.
    public object Clone() => new MiTipo(this);

    // Estoy forzado a implementarlo por ser IComparable
    public int CompareTo(object obj)
    {
        // Si obj de es de MiTipo genero una excepción.
        MiTipo t = obj as MiTipo
        ?? throw new ArgumentException("No es del tipo MiClase", "obj");
        // Para realizar la comparación voy llamando a los CompareTo de cada
        // tipo en el orden adecuado.
        int comparacion = MiAgregacion.CompareTo(t.MiAgregacion);
        if (comparacion == 0)
            comparacion = Texto.CompareTo(t.Texto);
        return comparacion;
    }
}
```

IDisposable

Me indicará que el objeto debe implementar el método `void Dispose()` que **se encargará de liberar los recursos usados por el objeto.** No confundir con el destructor `~<Tipo>()`.
Indicaremos a las BCL que nuestro objeto tiene el comportamiento de liberar recursos y lo utilizaremos junto a la **instrucción** `using`, la cual ya hemos usado para cerrar automáticamente los flujos al producirse una excepción cuando vimos la entrada y salida de datos. Esto es, porque en el fondo lo único que espera el `using` es que las clases marcadas con está instrucción o cláusula implementen este interfaz. En el fondo la clase base **Stream** lo implementa y por tanto tiene un método `Dispose()` que es llamado en el `finally`.

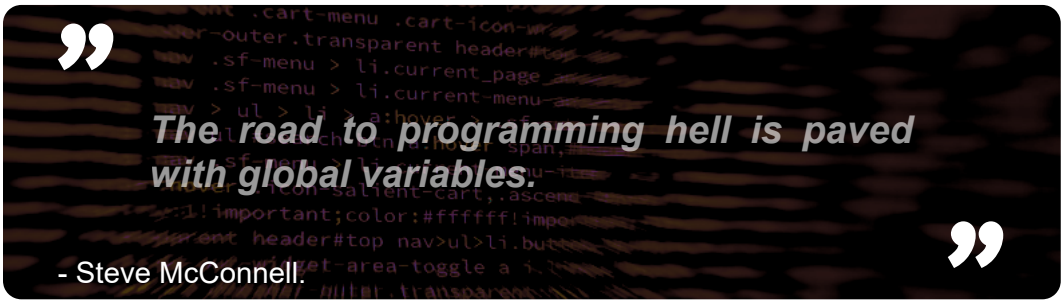
Repasando la instrucción using

- Se utiliza para instanciar objetos que contienen recursos no gestionados, esto es, que no son liberados por el recolector de basura.
- Como acabamos de comentar, estos objetos deben implementar el interfaz `IDisposable` y por tanto el método de liberación `Dispose()`.
- `using` garantiza que se llama a `Dispose()` **aunque se produzca una excepción**.
  > ✋ **Importante:** Dentro del bloque using, el objeto es de solo lectura y no se puede modificar ni reasignar puesto que dejaría de tener una referencia y no se liberaría.

Sintaxis básica:

```
// Podemos usar varios recursos liberables en el mismo ámbito así ...
using (var r1 = new TipoIDisposable())
using (var r2 = new TipoIDisposable())
...
using (var rN = new TipoIDisposable())
{
    // Ámbito de uso de solo lectura de r1, r2, … , rN
}

// También podremos anidarlos.
```



En **C#8.0** evolucionó el uso de esta instrucción. De tal manera que como comentamos con los flujos, **podemos aprovechar un bloque ya definido para decidir cuando va a estar disponible un recurso**.

```
// Esté código desde C#8
if (…)
{
    using var r = new TipoIDisposable();
    // Bloque...
}

// Equivale al siguiente en versiones anteriores...
if (…)
{
    using (var r = new TipoIDisposable() )
    {
      // Bloque...
    }
}
```

Interpretación real de la instrucción `using`:

```
// Cuando instanciemos un objeto disposable de la siguiente manera en un método...
void Metodo()
{
    using var r = new TipoIDisposable();
    // Cuerpo del método ...
}

// Realmente será un 'syntactic sugar' del siguiente código.
void Metodo()
{
    TipoIDisposable r;
    try
    {
        r = new TipoIDisposable();
        // Cuerpo del método ...
    }
    finally
    {
        if (r != null) ((IDisposable)r).Dispose();
    }
}
```

Recordemos su uso a través de un **ejemplo**...

En el ejemplo siguiente creamos una clase de utilidad para generar **logs** de nuestros programas a un determinado fichero y visualizarlos.

```
static class Log
{
    public static void Escribe(string fichero, string texto)
    {
        using TextWriter w = File.AppendText($"{fichero}.log");
        w.WriteLine(DateTime.Now.ToString("dd/MM/yyy HH:mm:ss ") + texto);
    }

    public static void Muestra(string fichero)
    {
        using TextReader r = File.OpenText($"{fichero}.log");
        string s;
        while ((s = r.ReadLine()) != null) Console.WriteLine(s);
    }
}

static class Ejemplo
{
    static void Main()
    {
        Log.Escribe("DEBUG", "Empieza Main");
        Log.Escribe("DEBUG", "Finaliza Main");
        Log.Muestra("DEBUG");
    }
}
```

Dado que las clases `TextWriter` y `TextReader` implementan la interfaz `IDisposable`, podremos usar la instrucción `using` que garantizará que el archivo subyacente se cierre correctamente después de las operaciones de lectura o escritura.

Caso de aplicación de interfaces

🎓 **Caso de estudio:**

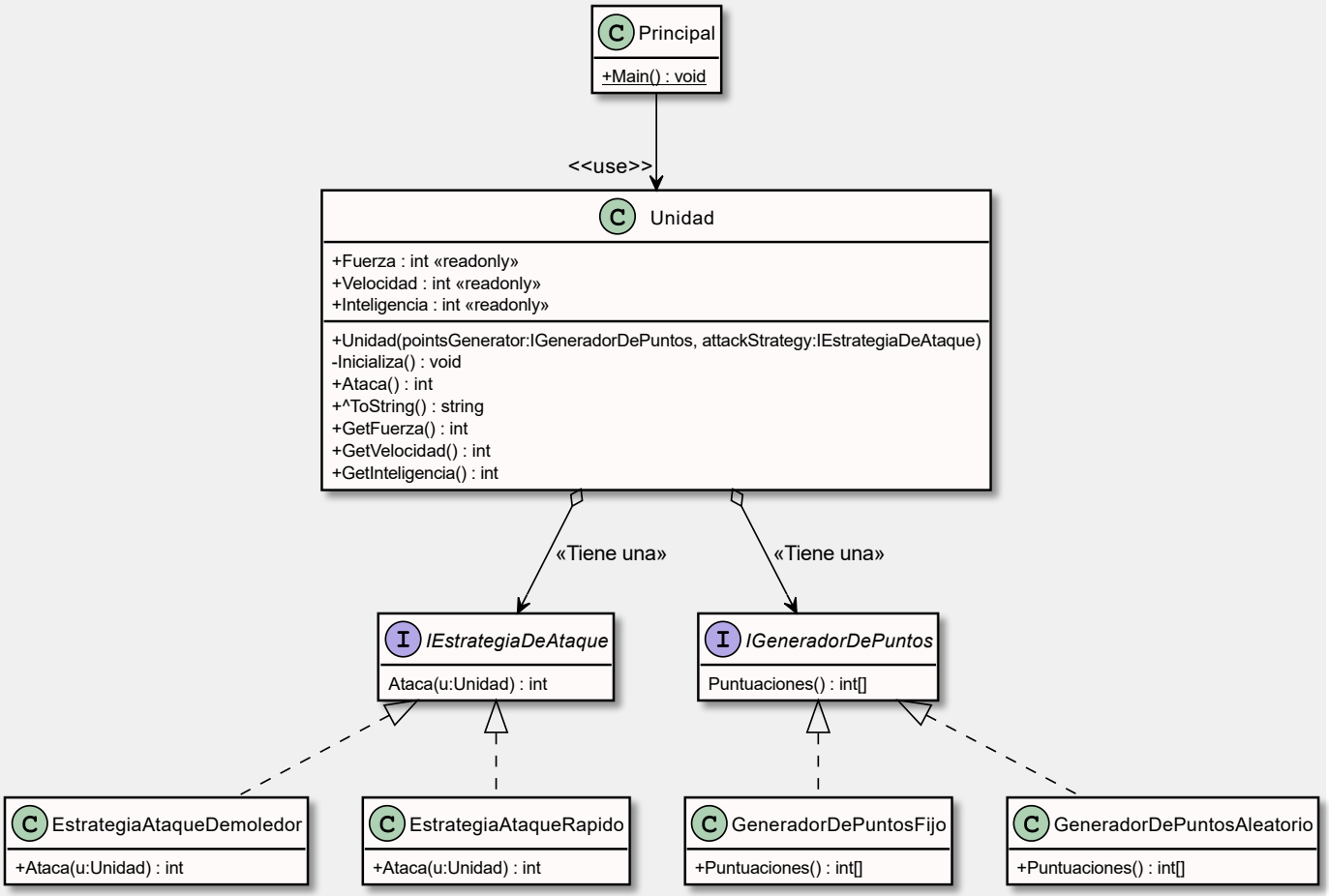
Vamos a ver un ejemplo de uso de interfaces a través de un ejemplo de aplicación del patrón de diseño **Strategy**.

📌 **Nota:** La idea no es aprender a usar el patrón, si no más bien ver un **caso práctico de uso de interfaces**.

Este patrón, nos permite definir una familia de algoritmos, encapsulando cada uno de ellos y haciéndolos intercambiables. Por tanto, nos permite que un algoritmo pueda variar de forma dinámica en el uso que hacen los clientes.

Supongamos que en un juego de estrategia en tiempo real, tenemos una **unidad** de ataque con una serie de propiedades como son `Fuerza`, `Velocidad` e `Inteligencia`. Sin embargo, hasta que no instanciemos el objeto unidad, no sabremos cómo se inicializarán estos valores y que '*estrategia*' va a seguir, es decir cómo se calcularán los puntos de ataque que va a tener la unidad de acuerdo a los valores anteriores.

Para cumplir esta especificación, implementaremos el siguiente patrón Strategy, conforme se expresa en al diagrama de clases siguiente.



Si nos fijamos, definimos 2 objetos agregados en mi `Unidad` que se corresponderán con 2 **generalizaciones** de **objetos que sabemos que implementan un determinado comportamiento** (interfaz). Uno tiene la funcionalidad de inicializar los puntos de las propiedades y otro la de calcular los puntos de ataque (estos són los algoritmos que hemos dicho antes que se definen al instanciar el objeto, en esta caso unidad).

```
// Definimos las generalizaciones de los comportamientos
public interface IEstrategiaDeAtaque
{
        int Ataca(Unidad u);
}
public interface IGeneradorDePuntos
{
        int[] Puntuaciones();
}

// Definimos las concreciones de dichos comportamientos
//    * Formas de inicializar las propiedades de la unidad.
public class GeneradorDePuntosFijo : IGeneradorDePuntos
{
    public int[] Puntuaciones() => new int[] { 6, 6, 6 };
}
public class GeneradorDePuntosAleatorio : IGeneradorDePuntos
{
    public int[] Puntuaciones()
    {
        int[] values = new int[3];
        var semilla = new Random();
        for (int i = 0; i < 3; i++)
        {
            values[i] = semilla.Next(0, 8);
        }
        return values;
    }
}
//    Formas de atacar de mi unidad y de calcular la fuerza de su ataque.
public class EstrategiaAtaqueDemoledor : IEstrategiaDeAtaque
{
    public int Ataca(Unidad u) => u.Fuerza * (u.Velocidad / 2);
}
public class EstrategiaAtaqueRapido : IEstrategiaDeAtaque
{
        public int Ataca(Unidad u) => u.Velocidad + u.Inteligencia;
}
```

🎓 Caso de estudio (Continuación)

```
public class Unidad
{
    private readonly IGeneradorDePuntos generadorDePuntos;
    private readonly IEstrategiaDeAtaque estrategiaDeAtaque;

    public int Fuerza { get; private set; }
    public int Velocidad { get; private set; }
    public int Inteligencia { get; private set; }

    public Unidad(IGeneradorDePuntos generadorDePuntos, IEstrategiaDeAtaque estrategiaDeAtaqu
    {
12      // Estos métodos definirán una forma de hacer las cosas genérica.
        this.generadorDePuntos = generadorDePuntos;
        this.estrategiaDeAtaque = estrategiaDeAtaque;
15      // Este método aplica la estrategia de obtención de puntuaciones.
        Inicializa();
    }
    private void Inicializa()
    {
20      // El array de puntuaciones se obtendrá según la concreción.
        int[] values = generadorDePuntos.Puntuaciones();
        Fuerza = values[0];
        Velocidad = values[1];
        Inteligencia = values[2];
    }
    public int Ataca()
    {
        // Los puntos de ataque se obtendrán según la concreción.
29      return estrategiaDeAtaque.Ataca(this);
    }
    public override string ToString()
    {
        return $"Unidad\n\tF={Fuerza}\n\t" +
               $"V={Velocidad}\n\tI={Inteligencia}" +
               $"Ataque={Ataca()}\n";
    }
}
```

Fíjate que ahora instanciamos nuestra unidades de ataque con las **concreciones** que implementan los interfaces esperados por la clase `Unidad` .

```
public static void Main()
{
  var uAleatoriaDeAtaqueRapido = new Unidad(
              new GeneradorDePuntosAleatorio(), new EstrategiaAtaqueRapido());
  var uFijaDeAtaqueDemoledor = new Unidad(
              new GeneradorDePuntosFijo(),      new EstrategiaAtaqueDemoledor());
  Console.WriteLine(uAleatoriaDeAtaqueRapido);
  Console.WriteLine(uFijaDeAtaqueDemoledor);
}
```

Clases parametrizadas o genéricos

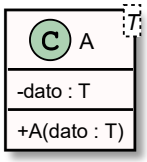
Hasta ahora, cuando queríamos referirnos a un objeto de forma '**genérica**' nos referenciábamos a través de la clase **object**, sin embargo este tipo de generalización no nos asegura, hasta hacer el downcasting, que el tipo de objeto referenciado es del tipo que esperábamos. Esto puede ser fuente de 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. Está 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> {
    private T dato;

    public A(T dato) {
        this.dato = dato;
    }
}
```

- **Uso:** Cuando yo instáncie un objeto de la clase genérica A …

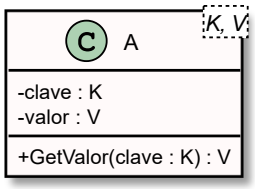
```
A<int> objA = new A<int>(4);
```

  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>
{
    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>();
   ```

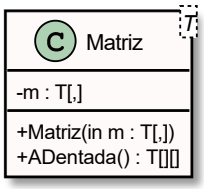
   sería correcto y tendríamos dos signaturas. Una con **p1** como `int` y otra con **p1** como `string` . Pero ... **¿Qué pasa si declaramos?**

   ```
   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.




```
class Matriz<T>
{
    private readonly T[,] m;
    public Matriz(T[,] m)
    {
        this.m = m;
    }
    public T[][] ADentada()
    {
        T[][] d = null;
        if (m != null)
        {
            d = new T[m.GetLength(0)][];
            for (int i = 0; i < d.Length; i++)
            {
                d[i] = new T[m.GetLength(1)];
                for (int j = 0; j < d[i].Length; j++)
                    d[i][j] = m[i, j];
            }
        }
        return d;
    }
}
class Ejemplo
{
    static void Main()
    {
28          // Aquí la matriz a envolver será de enteros.
        Matriz<int> m1 = new Matriz<int>(new int[,]{ {1, 2, 3}, {3, 2, 1} });
        int[][] d1 = m1.ADentada();

32          // Aquí la matriz a envolver será de caracteres.
        Matriz<char> m2 = new Matriz<char>(new char[,]{ {'a'}, {'b'}, {'c'} });
        char[][] d2 = m2.ADentada();
    }
}
```

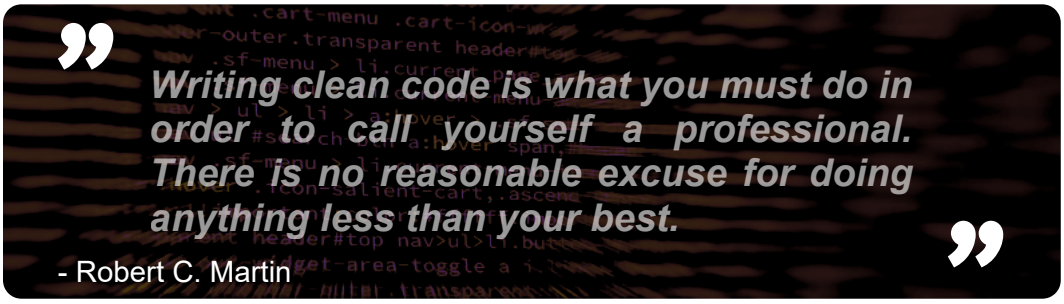
Métodos parametrizados

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.

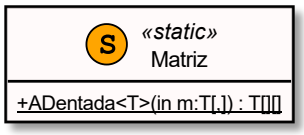
En la mayoría de los casos, este tipo de parametrización de métodos, tendrá más sentido con métodos de utilidad estáticos. Ya que como operaciones sobre un objeto no será '*inmediato*' relacionarlos con los tipos de los campos que definen el estado de la clase.

```
// 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 está 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 class Matriz
{
    static public T[][] ADentada<T>(T[,] m)
    {
        T[][] d = null;
        if (m != null)
        {
            d = new T[m.GetLength(0)][];
            for (int i = 0; i < d.Length; i++)
            {
                d[i] = new T[m.GetLength(1)];
                for (int j = 0; j < d[i].Length; j++)
                    d[i][j] = m[i, j];
            }
        }
        return d;
    }
}
class Program
{
    static void Main()
    {
        // En este caso indicamos el tipo la matriz de entrada y tabla de salida
        // indicándo el tipo en le llamada del método.
        int[][] d1 = Matriz.ADentada<int>(new int[,]{ {1, 2, 3}, {3, 2, 1} });

        // También podremos hacer la llamada obviando indicar el tipo parametrizado
        // ya que C# lo infiere de la matriz de entrada.
        char[][] d2 = Matriz.ADentada(new char[,]{ {'a'}, {'b'}, {'c'} });
    }
}
```

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()
    {
        return $"{H:D2}:{M:D2}";
    }
    public int CompareTo(object objHora)
    {
16      Hora hora = objHora as Hora
17      ?? throw new ArgumentException("El objeto a comparar no es una hora.", "obj");
        int comparacion = 0;
        if (hora.H > H) comparacion = -1;
        else if (hora.H < H) comparacion = 1;
        else if (hora.M < M) comparacion = -1;
        else if (hora.M > M) comparacion = 1;
        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.

    public int CompareTo(Hora hora)
    {
        int comparacion = 0;
        if (hora.H > H) comparacion = -1;
        else if (hora.H < H) comparacion = 1;
        else if (hora.M < M) comparacion = -1;
        else if (hora.M > M) comparacion = 1;
        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. `<T> where T : ClaseBaseDeT`
De interfaz ⭐	El tipo debe implementar una **interfaz** determinada. `<T> where T : IinterfazAImplementar`
De tipo referencia	El tipo debe ser **referencia**. `<T> where T : class`
De tipo valor	El tipo debe ser **valor**. `<T> where T : struct`
De constructor	El tipo debe tener un constructor sin parámetros. `<T> where T : new`

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

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

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

    static void Main()
    {
        ...
        // Ahora la siguiente línea ...
7       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 contrário.

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 instáncie `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 bool Equals(object obj)
    {
        Matriz<T> m = obj as Matriz<T>;
        bool iguales = true;
        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));
}
```