POO How-To

Cómo crear una clave primaria

- Crear una propiedad de solo lectura basada en un campo de respaldo estático para llevar la cuenta de objetos creados
- Permite hacer saber cuántos productos se han creado en total
- No es útil en entornos multithread porque varios hilos podrían leer su valor a la vez

```
public class Producto
{
    private static int _contadorProductos = 0; //Campo estático: solo hay una copia comp

    public int Id { get; } // Esto no es multithread-safe
    public string Nombre { get; }

    public Producto(string nombre)
    {
        _contadorProductos++;
        Id = _contadorProductos;
        Nombre = nombre; // Se debería comprobar que nombre no sea ni null ni vacío
    }

    public static int ProductosCreados => _contadorProductos;  // Propiedad estática: pe

    public override string ToString() => $"ID: {Id}, Nombre: {Nombre}";
}

public class Program
{
    public static void Main()
    {
        Producto producto1 = new("Producto A");
        Console.WriteLine(producto1);
        Producto producto2 = new("Producto B");
        Console.WriteLine(producto2);
        Console.WriteLine($"Total de productos creados: {Producto.ProductosCreados}");
    }
}
```

- También se puede crear con un GUID
- Ventajas: Multithread-safe. Útil en sistemas distribuidos
- Desventajas: ocupa 16 *Bytes* frete a los 4 de un `int`. Es más difícil de leer para humanos

```
public class Producto
{
    public Guid Id { get; }
    public string Nombre { get; }

    public Producto(string nombre)
    {
        Id = Guid.NewGuid(); // Multithread-safe. Útil en sistemas distribuidos
        Nombre = nombre; // Se debería comprobar que nombre no sea ni null ni vacío
    }
    public override string ToString() => $"Id={Id}, Nombre={Nombre}]";
}

public class Program
{
    public static void Main()
    {
        Producto producto1 = new("Producto A");
        Console.WriteLine(producto1);
        Producto producto2 = new("Producto B");
        Console.WriteLine(producto2);
    }
}
```

Cómo hacer que un campo sea obligatorio-opcional

- Campo obligatorio: no es nulable y se asigna en el constructor. Si el campo es un tipo referencia, en el constructor se comprobará que no es ni vacío ni `null`
- Campo opcional: es nulable y se inicializa a null

```
public class Persona
{
    public string Nombre { get; } // Obligatorio
    public string DNI { get; } // Obligatorio
    public DateOnly? FechaNacimiento { get; set; } // Opcional. Permite fechas de nacim

    public int Edad => FechaNacimiento.HasValue ? DateTime.Now.Year - FechaNacimiento.Va

    public Persona(string nombre, string dni)
    {
        if (string.IsNullOrEmpty(nombre))
        {
            throw new ArgumentException("El nombre no puede estar vacío.", nameof(nombre
        }
        Nombre = nombre;
        if (string.IsNullOrEmpty(dni))
        {
            throw new ArgumentException("El DNI no puede estar vacío.", nameof(DNI));
        }
        DNI = dni;
        FechaNacimiento = null;
    }

    private string DescripcionEdad() => FechaNacimiento.HasValue ? $"{Edad} años" : "Eda
    public override string ToString() => $"{Nombre} ({DNI}) - Edad a fecha ({DateTime.No
}

public class Program
{
    public static void Main()
    {
        Persona persona1 = new Persona("Juan Pérez", "12345678A");
        persona1.FechaNacimiento = new DateOnly(1990, 1, 1);
        Console.WriteLine(persona1);

        Persona persona2 = new Persona("María López", "87654321B");
        Console.WriteLine(persona2);
    }
}
```

Cómo crear una relación uno a muchos

Vamos a trabajar el ejemplo de un *Blog* que puede contener cero o muchos *Post*

Sólo navegación del lado uno al lado muchos

```
// Clase principal (padre)
public class Blog
{
    public string CodigoHtml { get; set; }
    public List<Post> Posts { get; } // Desde el Blog puedo acceder a todos sus Posts

    public Blog(string codigoHtml)
    {
        CodigoHtml = codigoHtml;
        Posts = new List<Post>();
    }

    public void AgregarPost(Post post)
    {
        Posts.Add(post);
    }
}

// Clase dependiente (hija)
// Desde un Post no puedo acceder al Blog
public class Post
{
    public string CodigoHtml { get; set; }
    public Post(string codigoHtml) => CodigoHtml = codigoHtml;
}

public class Program
{
    public static void Main()
    {
        Blog miBlog = new Blog("Mi Blog");
        Post post1 = new ("Mi primer post");
        miBlog.AgregarPost(post1);
        Post post2 = new ("Mi segundo post");
        miBlog.AgregarPost(post2);

        Console.WriteLine("Contenido del Blog:");
        Console.WriteLine(miBlog.CodigoHtml);
        foreach (var post in miBlog.Posts)
        {
            Console.WriteLine("- " + post.CodigoHtml);
```

```
            }
        }
}
```

- Con esta estructura solo el *Blog* conoce a sus *Post* por lo que hay una mejor encapsulación
- Si hacemos una búsqueda de *Posts* que contienen una palabra clave, será más complicado obtener los *Blogs* en los que se publicaron dichos *Posts*

Sólo navegación desde el lado muchos al lado uno

```
public class Blog
{
    public string CodigoHtml { get; set; }

    public Blog(string codigoHtml) => CodigoHtml = codigoHtml;
}

public class Post
{
    public string CodigoHtml { get; set; }
    public Blog Blog { get; } // Desde un Post se puede acceder al Blog asociado

    public Post(string codigoHtml, Blog blog)
    {
        CodigoHtml = codigoHtml;
        Blog = blog;
    }
}

public class Program
{
    public static void Main()
    {
        Blog miBlog1 = new Blog("Mi Blog");
        Blog miBlog2 = new Blog("Otro Blog");

        List<Post> posts = [];

        Post post = new Post("Post 1", miBlog1);
        posts.Add(post);
        post = new Post("Post 2", miBlog1);
        posts.Add(post);
        post = new Post("Post 3", miBlog2);
        posts.Add(post);
        post = new Post("Post 4", miBlog1);
        posts.Add(post);

        foreach (Post p in posts)
        {
            Console.WriteLine($"Post: {p.CodigoHtml}, pertenece al Blog: {p.Blog.CodigoH
        }
```

```
    }
}
```

- El *Post* conoce a su *Blog*, pero este no conoce a sus *Posts*.

Navegación bidireccional

```
public class Blog
{
    public string CodigoHtml { get; set; }
    public List<Post> Posts { get; } // Desde el Blog puedo acceder a todos sus Posts

    public Blog(string codigoHtml)
    {
        CodigoHtml = codigoHtml;
        Posts = new List<Post>();
    }

    public void AgregarPost(Post post)
    {
        post.Blog = this; // Asignar el Blog al Post
        Posts.Add(post);
    }

    public override string ToString() => $"Blog: {CodigoHtml}";
}

public class Post
{
    public string CodigoHtml { get; set; }
    public Blog? Blog { get; set; } // Desde un Post se puede acceder al Blog asociado
    // Al crear un Post, todavía no está asignado a ningún Blog

    public Post(string codigoHtml) => CodigoHtml = codigoHtml;

    public override string ToString() => $"Post: {CodigoHtml}";
}

public class Program
{
    public static void Main()
    {
        Blog miBlog1 = new Blog("Mi Blog");
        Blog miBlog2 = new Blog("Otro Blog");

        List<Post> posts = [];

        Post post = new Post("Post 1");
```

```
            posts.Add(post);
            miBlog1.AgregarPost(post);
            post = new Post("Post 2");
            posts.Add(post);
            miBlog1.AgregarPost(post);
            post = new Post("Post 3");
            posts.Add(post);
            miBlog2.AgregarPost(post);
            post = new Post("Post 4");
            posts.Add(post);
            miBlog2.AgregarPost(post);

            Console.WriteLine("Partiendo de un Blog puedo obtener sus Posts:");
            Console.WriteLine(miBlog1);
            foreach (Post p in miBlog1.Posts)
            {
                Console.WriteLine("\t" + p);
            }

            Console.WriteLine("Partiendo de un Post puedo obtener su Blog:");
            foreach (Post p in posts)
            {
                Console.Write(p + " pertenece a ");
                Console.WriteLine(p.Blog);
            }
        }
    }
```

- Esta opción tiene más riesgo de inconsistencias porque la información está duplicada
- Requiere mantener la sincronía de manera manual a la hora de crear, eliminar o cambiar *Blogs* y *Posts*

Cómo hacer una Relación muchos a muchos

- Vamos a ver un ejemplo en el que un *Post* puede tener muchas *Etiquetas* y una *Etiqueta* puede estar asignada a muchos *Posts*

```
public class Post
{
    public string CodigoHtml { get; set; }
    public List<Etiqueta> Etiquetas { get; } = [];

    public Post(string codigoHtml) => CodigoHtml = codigoHtml;

    public void AgregarEtiqueta(Etiqueta etiqueta)
    {
        Etiquetas.Add(etiqueta);
        etiqueta.Posts.Add(this);
    }

    public override string ToString() => CodigoHtml;

    public string PostYEtiquetas()
    {
        string resultado = $"Post: {CodigoHtml}\nEtiquetas:\n";
        foreach (var etiqueta in Etiquetas)
        {
            resultado += $"- {etiqueta}\n";
        }
        return resultado;
    }
}

public class Etiqueta
{
    public string TextoEtiqueta { get; set; }
    public List<Post> Posts { get; } = [];

    public Etiqueta(string textoEtiqueta) => TextoEtiqueta = textoEtiqueta;

    public override string ToString() => TextoEtiqueta;

    public string EtiquetaYPosts()
    {
        string resultado = $"Etiqueta: {TextoEtiqueta}\nPosts:\n";
        foreach (var post in Posts)
        {
            resultado += $"- {post}\n";
        }
        return resultado;
```

```
    }
}

class Program
{
    static void Main(string[] args)
    {
        // Crear etiquetas
        Etiqueta etiquetaCSharp = new ("C#");
        Etiqueta etiquetaDotNet = new (".NET");
        Etiqueta etiquetaProgramacion = new ("Programación");
        List<Etiqueta> etiquetas = new List<Etiqueta> { etiquetaCSharp, etiquetaDotNet,

        // Crear posts
        Post post1 = new Post ("Introducción a C#");
        Post post2 = new Post ("Desarrollo con .NET");
        List<Post> posts = new List<Post> { post1, post2 };

        // Asociar etiquetas a los posts
        post1.AgregarEtiqueta(etiquetaCSharp);
        post1.AgregarEtiqueta(etiquetaProgramacion);
        post2.AgregarEtiqueta(etiquetaDotNet);
        post2.AgregarEtiqueta(etiquetaProgramacion);

        // Mostrar resultados
        Console.WriteLine("Listado de posts y sus etiquetas:");
        foreach (var post in posts)
        {
            Console.WriteLine(post.PostYEtiquetas());
        }
        Console.WriteLine("Listado de etiquetas y sus posts:");
        foreach (var etiqueta in etiquetas)
        {
            Console.WriteLine(etiqueta.EtiquetaYPosts());
        }
    }
}
```

Cómo hacer que una relación sea de agregación o de composición

Agregación

En el apartado "Sólo navegación del lado uno al lado muchos" del punto "Cómo crear una relación uno a muchos" hemos visto el siguiente código:

```
public class Blog
{
    public string CodigoHtml { get; set; }
    public List<Post> Posts { get; } // Desde el Blog puedo acceder a todos sus Posts

    public Blog(string codigoHtml)
    {
        CodigoHtml = codigoHtml;
        Posts = new List<Post>();
    }

    public void AgregarPost(Post post)
    {
        Posts.Add(post);
    }
}

public class Post
{
    public string CodigoHtml { get; set; }
    public Post(string codigoHtml) => CodigoHtml = codigoHtml;
}

public class Program
{
    public static void Main()
    {
        Blog miBlog = new Blog("Mi Blog");
        Post post1 = new ("Mi primer post");
        miBlog.AgregarPost(post1);
        Post post2 = new ("Mi segundo post");
        miBlog.AgregarPost(post2);

        Console.WriteLine("Contenido del Blog:");
        Console.WriteLine(miBlog.CodigoHtml);
        foreach (var post in miBlog.Posts)
        {
            Console.WriteLine("- " + post.CodigoHtml);
        }
    }
}
```

- Es importante fijarse cómo, en este ejemplo, el *Post* tiene una vida independiente del Blog, es decir, si un *Blog* desaparece, no tienen porqué desaparecer todos sus *Posts*.
- Por ejemplo, "Mi primer post" está guardado en `post1`. Si desaparece su blog: `miBlog`, el post continuará estando guardado en `post1`
- A este tipo de relación, se le llama **Agregación**.

Composición

- En este tipo de relación queremos que al desaparecer un *Blog* desaparezcan junto con él todos sus *Posts*. Para conseguirlo, la única referencia a un *Post* debe estar guardada dentro del *Blog* y, por tanto, es necesario crear (`new`) el *Post* dentro del *Blog*. A este tipo de relación, se le llama **Composición** y a los métodos que crean objetos se les llama **Métodos factoría**

```
public class Blog
{
    public string CodigoHtml { get; set; }
    public List<Post> Posts { get; } // Desde el Blog puedo acceder a todos sus Posts

    public Blog(string codigoHtml)
    {
        CodigoHtml = codigoHtml;
        Posts = new List<Post>();
    }

    public void AgregarPost(string codigoHtml)
    {
        Post nuevoPost = new (codigoHtml);
        Posts.Add(nuevoPost);
    }
}

public class Post
{
    public string CodigoHtml { get; set; }
    public Post(string codigoHtml) => CodigoHtml = codigoHtml;
}

public class Program
{
    public static void Main()
    {
        Blog miBlog = new Blog("Mi Blog");
        miBlog.AgregarPost("Mi primer post");
        miBlog.AgregarPost("Mi segundo post");

        Console.WriteLine("Contenido del Blog:");
        Console.WriteLine(miBlog.CodigoHtml);
        foreach (var post in miBlog.Posts)
        {
            Console.WriteLine("- " + post.CodigoHtml);
        }
    }
}
```