Unidad 19

Descargar estos apunte en pdf o html

Índice

Colecciones

Introducción

Uno de los casos más comunes de usu de tipos genéricos o parametrizados, son el de las **colecciones**. Donde se nos pide que implementemos una colección de objetos de un tipo concreto, pero las operaciones de añadir, eliminar, buscar, etc. son idénticas independientemente del tipo de objeto que se almacene en la colección. Por esta razón, C# utilizará el mecanismo de **clases parametrizadas** o **clases genéricas** para poder solucionar este problema.

Una **colección** es un tipo de dato cuyos objetos almacenan otros objetos. Un ejemplo típico son las tablas, aunque en la BCL se incluyen muchas otras clases de colecciones que iremos viendo a lo largo de este tema. En las versiones recientes de C# las podemos encontrar en **System.Collections.Generic**.

Aunque las colecciones predefinidas incluidas en la BCL disponen de miembros propios con los que manipularlas, todas incluyen al menos los miembros de `ICollection<T>`. En realidad la interfaz `ICollection<T>` hereda de:

1. La interfaz `IEnumerable<T>` que permite que sean recorridas con la instrucción `foreach` usando el **patrón iterador**.
2. La interfaz `IClonable`, formada por un único método `object Clone()` que devuelve una copia del objeto sobre el que se aplica.

A lo largo del tema veremos que **hay muchas colecciones parecidas, donde la única diferencia es la '*eficiencia*' de un cierto tipo de operaciones sobre otras**. Por eso, deberemos escoger cuidadosamente el tipo de colección dependiendo de las casuísticas que se nos puedan dar en nuestro programa.

Con esto en mente, veamos algunas de las colecciones más utilizadas en C# y cuando utilizarlas.

Lista simple List<T>

Ya hemos hablado de ellas y las venimos usando desde la unidad 14 donde vimos las relaciones todo-parte. Recordemos que se implementan a través del tipo `List<T>`, que **son equivalentes a los arrays**. Esto es, internamente se almacenan como un array, y **su tamaño crecerá automáticamente** cuando se añadan más elementos de los que puede almacenar el array interno. Pero estos cambios de tamaño estarán optimizados para que no se realicen con demasiada frecuencia y serán transparentes para el usuario.

por esta razón, **permitirá accesos y modificaciones eficientes** a través del operador `[]` y un índice entero. Sin embargo, **añadir y borrar elementos puede ser más costoso**. Recordemos por encima su uso...

Crear una lista

```
List<int> numeros1 = []; // Lista vacía de enteros
List<int> numeros2 = [1, 2, 3, 4, 5]; // Lista de enteros con 5 elementos
List<int> numeros3 = [..numeros1, ..numeros2]; // Lista de enteros con los elementos de numeros1 y numeros2
```

Añadir elementos List<T>

```
numeros1.Add(6); // Añade el elemento 6 al final de la lista
numeros1.AddRange([7, 8]); // Añade los elementos 7 y 8 al final de la lista
numeros1.Insert(0, 5); // Inserta el elemento 5 en la posición 0
numeros1.InsertRange(0, [1,2,3,4]); // Inserta los elementos 1,2,3 y 4 en la posición 0

// numeros1 ahora es [1,2,3,4,5,6,7,8]
```

Acceder y modificar elementos List<T>

```
int primero = numeros1[0]; // primero es 1
numeros1[0] = 10; // Modifica el primer elemento a 10

// numeros1 ahora es [10,2,3,4,5,6,7,8]
```

Eliminar elementos List<T>

```
numeros1.Remove(10); // Elimina el elemento 10
numeros1.RemoveAt(0); // Elimina el elemento en la posición 0
numeros1.RemoveRange(0, 2); // Elimina 2 elementos a partir de la posición 0
// numeros1 ahora es [3,4,5,6,7,8]
```

Ordenar elementos List<T>

Ya hemos visto en unidades anteriores y cuando explicamos los interfaces que podemos ordenarlas con el método `Sort()`. Si el tipo de dato almacenado implementa la interfaz `IComparable<T>` (como los tipos numéricos y cadenas) se ordenarán de forma natural.

```
List<string> nombres = ["Juan", "Pedro", "Luis", "Ana"];
nombres.Sort();
```

Pero,... ¿Qué sucede si el tipo de dato es una clase que no hemos definido nosotros o no podemos modificar porque está en una librería y por tanto no podemos implementar la interfaz `IComparable<T>`?

En este caso, podemos usar el patrón '*Strategy*' pasándole al método `Sort()` un objeto que implemente la interfaz `IComparer<T>`. Esta interfaz define el método `int Compare(T x, T y)` que devuelve un valor negativo si `x < y`, cero si son iguales y un valor positivo si `x > y`.

Ejemplo:

Supongamos que tenemos la siguiente clase `Empleado` que no implementa la interfaz `IComparable<T>` y la cual no podemos modificar:

```
public record Empleado(string Nombre, double Sueldo);
```

Si ejecutamos el siguiente código:

```
public static void Main()
{
    List<Empleado> empleados =
    [
        new(Nombre: "Manuel", Sueldo: 2000),
        new(Nombre: "Mónica", Sueldo: 2800),
        new(Nombre: "Francisco", Sueldo: 2400)
    ];

    empleados.Sort();

    Console.WriteLine(string.Join("\n", empleados));
}
```

❌ **Obtendremos un error en tiempo de ejecución:**

```
System.InvalidOperationException: 'Failed to compare two elements in the array.'
```

Ahora implementamos la clase `OrdenaPorSueldo` que implementa la interfaz `IComparer<Empleado>` y la usamos para ordenar la lista de empleados por sueldo:

```
public class ComparaEmpleadoPorSueldo : IComparer<Empleado>
{
    public int Compare(Empleado? x, Empleado? y) => (x, y) switch
    {
        (null, null) => 0,
        (null, _) => -1,
        (_, null) => 1,
        _ => x.Sueldo.CompareTo(y.Sueldo)
    };
}
```

Ahora si ejecutamos el siguiente código...

```
empleados.Sort(new ComparaEmpleadoPorSueldo());
```

✅ **Obtendremos la lista de empleados ordenada por sueldo.**

Buscar lineal elementos List<T>

> **Nota**
>
> Es importante destacar que más adelante en el curso, **cuando veamos programación funcional**, veremos que en C# las listas implementan muchos otros métodos de extensión para buscar, filtrar, transformar, etc. como `Find()`, `FindAll()`, `Where()`, `Select()`, etc y que **de momento no podemos usar**.

`Contains(T item)`: nos permite saber si un elemento está en la lista haciendo una **búsqueda lineal o secuencial**. Solo deberíamos usarla si:

1. Es un tipo de dato simple como un número o una cadena.
2. Es una un `record class` o `record struct`.
3. Es una clase que hemos definido nosotros y hemos implementado la interfaz `IEquatable<T>`
4. Es una clase que hemos definido nosotros y hemos sobrescrito los métodos `Equals(object? obj)` y `GetHashCode()`.

En el siguiente ejemplo, la clase `Empleado` es un `record class` y por tanto podemos usar el método `Contains()` para buscar un empleado en la lista:

```
public record Empleado(string Nombre, double Sueldo);

empleados.Contains(new (Nombre: "Carmen", Sueldo: 2800));
```

Pero..., ¿Qué sucede si no podemos modificar la clase `Empleado` es una clase con estado, no tiene implementada la interfaz `IEquatable<T>` y no hemos sobrescrito los métodos `Equals(object? obj)` y `GetHashCode()`?

```
public class Empleado
{
    public string Id {get;}
    public string Nombre {get;}
    public double Sueldo {get;}

    public Empleado(
        string id,
        string nombre,
        double sueldo)
    {
        Id = id;
        Nombre = nombre;
        Sueldo = sueldo;
    }
}
```

Usaremos el patrón '*Strategy*' pasándole al método `Contains()` un objeto que implemente la interfaz `IEqualityComparer<T>` como por ejemplo:

```
public class IgualdadEmpleadosPorId: IEqualityComparer<Empleado>
{
    public bool Equals(Empleado? x, Empleado? y) => (x, y) switch
    {
        (null, null) => true,
        (null, _) => false,
        (_, null) => false,
        _ => x.Id == y.Id   // Compara por el Id
    };
    public int GetHashCode(Empleado obj) => obj.Id.GetHashCode();
}
```

Ahora el siguiente programa principal funcionaria correctamente...

```
public static void Main()
{
    List<Empleado> empleados =
    [
        new(id: "001", nombre: "Manuel", sueldo: 2000),
        new(id: "002", nombre: "Mónica", sueldo: 2800),
        new(id: "003", nombre: "Francisco", sueldo: 2400)
    ];

    bool encontrado = empleados.Contains(
                                value: new("002", "Mónica", 2800),
                                comparer: new IgualdadEmpleadosPorId());
}
```

`IndexOf(T item)` : Si queremos encontrar un elemento por posición. El esquema de pasar una clase que implemente la estrategia de búsqueda no sería posible. Deberíamos poder modificar el tipo `T` para que implemente `EqualityComparer<T>` o debería ya implementarlo. Por lo que no es un método tan flexible como `Contains(T item)` .

Buscar binaria elementos List<T>

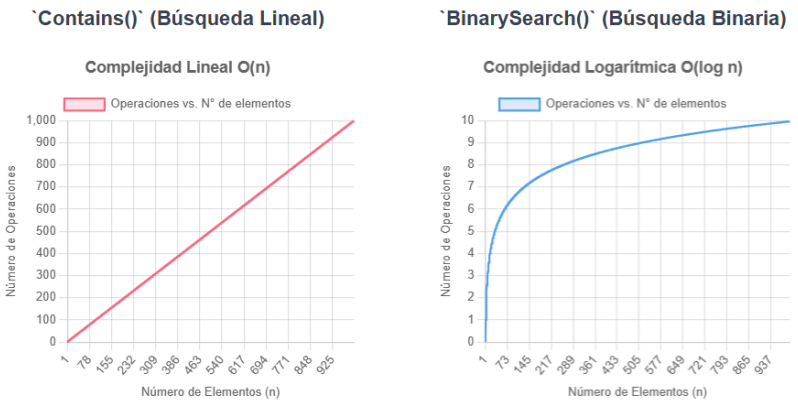
`BinarySearch(T item)` : Nos devolverá **el índice del elemento buscado si lo encuentra** o un valor negativo si no lo encuentra. Para ello usará **una búsqueda binaria** como su nombre indica, pero **solo funcionará si la lista está ordenada** y el tipo de dato almacenado implementa la interfaz `IComparable<T>` o le pasamos un objeto que implemente la interfaz `IComparer<T>` al método `BinarySearch(T item, IComparer<T> comparer)` como sucedió con el método `Sort()` . Por tanto, deberemos usarlo en combinación con este último método. Por ejemplo, si partimos del ejemplo de la clase `Empleado` que no implementa la interfaz `IComparable<T>` y no podemos modificarla, podríamos **ordenas y buscar un empleado por su** `Id` de la siguiente forma:

```
public class ComparaEmpleado : IComparer<Empleado>
{
    public int Compare(Empleado? x, Empleado? y) => (x, y) switch
    {
        (null, null) => 0,
        (null, _) => -1,
        (_, null) => 1,
        _ => x.Id.CompareTo(y.Id)
    };
}
IComparer<Empleado> comparaEmpleados = new ComparaEmpleado();
empleados.Sort(comparaEmpleados);
bool encontrado = empleados.BinarySearch(
                item: new Empleado(id: "002", nombre: "Mónica", sueldo: 2800),
                comparer: comparaEmpleados) >= 0;
```

¿Qué ventaja nos aporta `BinarySearch()` frente a `Contains()` si tengo que ordenar antes?

Pues que `BinarySearch()` tiene una **complejidad logarítmica** frente a la **complejidad de lineal** de `Contains()`. Por tanto, si vamos a realizar muchas búsquedas en una lista o esta tiene muchos elementos, es mejor ordenarla y usar `BinarySearch()`.

Fíjate que el número de operaciones que realiza `BinarySearch()` apenas crece al aumentar el número de elementos, mientras que el número de operaciones de `Contains()` crece linealmente.



Puedes descargar todo el código visto a lo largo del ejemplo del siguiente enlace **busquedas_list.cs**

Tablas Hash, Mapas o Diccionarios

Las tablas hash, también conocidas como Mapas o **Diccionarios**, son otro tipo de colecciones, pero que tienen un comportamiento particular.

Hasta ahora, todos los elementos de una colección se acceden a través de un **índice numérico**. Si tenemos una lista, el primer elemento ocupa la posición 0, el siguiente la 1, etc. Si queremos acceder al cuarto elemento de una lista llamada `miLista`, tenemos que poner `miLista[3]`, y **si no sabemos la posición** debemos usar un bucle.

Para evitar el bucle y **realizar un acceso directo podemos usar tablas hash**. En este tipo de colecciones, cada dato que agregamos a ella no tiene asociado un índice numérico, sino un objeto **clave** que lo identifica. De esta manera, si conocemos la **clave** del dato, podemos acceder directamente a sus datos sin tener que recorrer toda la lista.

Por jemplo, imaginemos que queremos guardar una serie de personas en una tabla hash, donde la clave sea su DNI (un `string`) y el valor un objeto de tipo `Persona` que contenga su nombre y edad.

```
class Persona
{
    public string Nombre { get; }
    public int Edad { get; private set; }
    public Persona(
        string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }
    public override string ToString()
    => $"{Nombre} {Edad} años";
}
```

Clave (`string`)	Valor (`Persona`)
"11224441K"	{ Nombre = "Pepe" Edad = 30 }
"11335499M"	{ Nombre = "María" Edad = 22 }
"12345678O"	{ Nombre = "Juan" Edad = 33 }
"13898743Y"	{ Nombre = "Sara" Edad = 27 }

> **Importante**
>
> Si quiero consultar los datos de María, buscaré por su clave que es su dni. Fíjate que la clave puede ser cualquier tipo de dato. En este caso es un `string`, pero podrían ser enteros u otro tipo cualquiera, siempre que nos aseguremos que **no haya dos claves repetidas**.

Si nos fijamos, el funcionamiento es similar a un diccionario real. Si quiero consultar el significado de una palabra y sé cuál es esa palabra, voy a la página donde está y la consulto, sin tener que ir palabra a palabra comprobando si es esa la que busco.

Como ya hemos comentado, las tablas hash en C# se manejan con el TAD **Dictionary<K, V>** y **SortedDictionary<K, V>**. La diferencia entre ambos es que el primero no garantiza ningún orden en los

elementos, mientras que el segundo los **mantiene ordenados por la clave**. Se pierde un poco más de tiempo insertando elementos, pero se gana en rapidez en las búsquedas porque **todas las búsquedas por clave serán binarias**.

Crear e inicializar un Diccionario

Es importante tener en cuenta que los elementos de esta colección serán objetos del tipo `KeyValuePair<TClave, TValor>` que guardará una clave y su valor. No obstante, muy raramente los vamos a trabajar a través de él.

Podemos **crear un diccionario** en C# de la siguiente formas...

```
Dictionary<TClave, TValor> tabla = new Dictionary<TClave, TValor>();

Dictionary<TClave, TValor> tabla = [];
```

Podemos **inicializarlos por extensión** de la siguiente formas...

```
Dictionary<TClave, TValor> tabla =
new Dictionary<TClave, TValor>()
{
    {clave1, valor1},
    {clave2, valor2},
    ...
};
```

```
Dictionary<TClave, TValor> tabla = new()
{
    [clave1] = valor1,
    [clave2] = valor2,
    ...
};
```

```
Dictionary<string, Persona> personas
= new Dictionary<string, Persona>()
{
    {"11224441K", new Persona("Pepe", 30)},
    {"11335499M", new Persona("María", 22)},
    {"123456780", new Persona("Juan", 33)},
    {"13898743Y", new Persona("Sara", 27)}
};
```

```
Dictionary<string, Persona> personas = new()
{
    ["11224441K"] = new ("Pepe", 30),
    ["11335499M"] = new ("María", 22),
    ["123456780"] = new ("Juan", 33),
    ["13898743Y"] = new ("Sara", 27)
};
```

La forma en la que usamos el operador `[]` para inicializar se da en versiones más modernas de C# y es un poco más clara.

Añadir y modificar elementos Dictionary<K, V>

Se puede realizar de varias formas, una de ellas es usar el método `Add`, indicando la clave que queremos asociar a cada elemento y el elemento en sí.

> ⚠ **Cuidado**
>
> La operación `Add` **generará una excepción** si añadimos una clave ya existente.

Si por ejemplo si queremos ir leyendo datos de personas desde la consola y guardándolas en el diccionario propuesto anteriormente, podríamos hacerlo de la siguiente forma:

```
public static Dictionary<string, Persona> LeeDatos()
{
    Dictionary<string, Persona> personas = [];
    Console.Write("Introduce los datos de las personas: ");
    int numeroPersonas = int.Parse(Console.ReadLine()!);
    for (int i = 0; i < numeroPersonas; i++)
    {
        Console.Write($"Dni {i + 1}: ");
        string dni = Console.ReadLine()!;
        Console.Write($"Nombre {i + 1}: ");
        string nombre = Console.ReadLine()!;
        Console.Write($"Edad {i + 1}: ");
        int edad = int.Parse(Console.ReadLine()!);
        // Añadir con el método add.
        personas.Add(dni, new(nombre, edad));
    }
    return personas;
}
```

El valor de un dato almacenado en el diccionario, también se puede acceder y modificar a través de su clave usando el operador `[]`. Por ejemplo, si queremos **modificar** el sueldo de un empleado con Id `"66668743G"` haríamos lo siguiente:

```
Dictionary<string, Persona> personas = LeeDatos();
personas["66668743G"] = new Persona("Susana", 27);
```

En este caso, si la clave `"66668743G"` no existe, se **añadirá** un nuevo elemento al diccionario. Básicamente podemos resumir diciendo que funciona igual que el `Add` solo que si la clave existe modificará su valor asociado sin generar una excepción.

> **Cuidado**
>
> Además, como en el caso del borrado, si intentamos acceder a un valor del `Dictionary` del que no existe la clave el sistema lanzará una excepción. Por lo que es buena práctica utilizar el método `ContainsKey(clave)`, para comprobar si existe la clave antes de acceder al valor a través de ella.

Borrar elementos Dictionary<K, V>

Usaremos el método **Remove** con la clave del valor que queremos eliminar como argumento. Si no existe la clave obtendremos una excepción.

```
personas.Remove("66668743G");
```

Recorriendo elementos en un Dictionary<K, V>

Los diccionarios disponen de propiedades para **obtener claves y valores por separado**. Estas son las propiedades `Keys` y `Values` respectivamente. Las cuales, me devolverán una secuencia que implementa `IEnumerable<T>` y que por ende puedo transformar a algún tipo de colección de las que conocemos ya sea un array o una lista como hemos visto a lo largo del tema.

```
List<string> litaDnis = [..personas.Keys];
Persona[] arrayPersonas = [..personas.Values];
```

Diccionarios no es lo común, ya que su acceso es mediante clave. Aunque se puede realizar el acceso a todos los elementos usando un `foreach`.

Por ejemplo, este bucle saca las edades de todas las personas:

```
foreach (string dni in personas.Keys)
    Console.WriteLine(personas[dni].Edad);
```

> **Nota**
>
> Es posible que el orden de salida no sea el mismo que cuando se introdujeron los datos, ya que las tablas hash tienen un mecanismo de ordenación diferente.

Realmente si tenemos en cuenta que nuestro diccionario realmente es una secuencia de valores del tipo `KeyValuePair<string, Persona>`. Podríamos recorrer sus valores también de la siguiente forma...

```
foreach (KeyValuePair<string, Persona> par in personas)
    Console.WriteLine($"{par.Key}: {par.Value}");
```

Sin embargo, esta forma es menos habitual y en C# **podemos usar tuplas** para descomponer el par clave-valor en dos variables independientes de forma más **'*natural*' y legible**...

```
foreach ((string dni, Persona persona) in personas)
    Console.WriteLine($"{dni}: {persona}");
```

Ejemplo:

Veamos un ejemplo sencillo donde se ponen en práctica algunas de las operaciones básicas de un diccionario vistas hasta ahora.

```
Dictionary<string, Persona> personas = new()
{
    ["11224441K"] = new("Pepe", 30),
    ["11335499M"] = new("María", 22),
    ["12345678O"] = new("Juan", 33),
    ["13898743Y"] = new("Sara", 27)
};

Console.WriteLine($"Los datos almacenados son:");
foreach ((string dni, Persona persona) in personas)
    Console.WriteLine($"- {persona} y DNI {dni}");

Console.WriteLine("Introduce un DNI para borrar:");
string dniBuscado = Console.ReadLine()!;
string salida = personas.ContainsKey(dniBuscado)
    ? $"{personas[dniBuscado]}ha sido borrado"
    : $"No se ha encontrado el DNI {dniBuscado}";
personas.Remove(dniBuscado);
Console.WriteLine(salida);

Console.WriteLine($"Los DNIs almacenados son:");
Console.WriteLine($"- {string.Join("\n -", personas.Keys)}");
```

La salida del programa sería similar a la siguiente:

```
Los datos almacenados son:
- Pepe 30 años y DNI 11224441K
- María 22 años y DNI 11335499M
- Juan 33 años y DNI 12345678O
- Sara 27 años y DNI 13898743Y
Introduce un DNI para borrar:
12345678O
Juan 33 años ha sido borrado
Los DNIs almacenados son:
- 11224441K
- 11335499M
- 13898743Y
```

Puedes descargar el código del ejemplo desde el siguiente enlace **diccionario_dni_persona_ejemplo.cs**

🚀 **Ampliación opcional:**

Vamos a implementar el ejemplo anterior en otros lenguajes de programación como **JavaScript** y **Kotlin** y así ver si sabemos encontrar las similitudes en la definición de esta colección y sus operaciones básicas en otros lenguajes.

JavaScript (Node.js):

```
const personas = new Map([
    ["11224441K", new Persona("Pepe", 30)],
    ["11335499M", new Persona("María", 22)],
    ["123456780", new Persona("Juan", 33)],
    ["13898743Y", new Persona("Sara", 27)]
]);

console.log("Los datos almacenados son:");
for (const [dni, persona] of personas) {
    console.log(`- ${persona.toString()} y DNI ${dni}`);
}

readline.question('\nIntroduce un DNI para borrar: ', (dniBuscado) => {
    let salida;

    if (personas.has(dniBuscado)) {
        const personaABorrar = personas.get(dniBuscado);
        salida = `${personaABorrar.toString()} ha sido borrado`;
        personas.delete(dniBuscado);
    } else {
        salida = `No se ha encontrado el DNI ${dniBuscado}`;
    }

    console.log(salida);

    console.log("\nLos DNIs almacenados son:");
    const dnisRestantes = Array.from(personas.keys()).join("\n- ");
    console.log(`- ${dnisRestantes}`);

    readline.close();
});
```

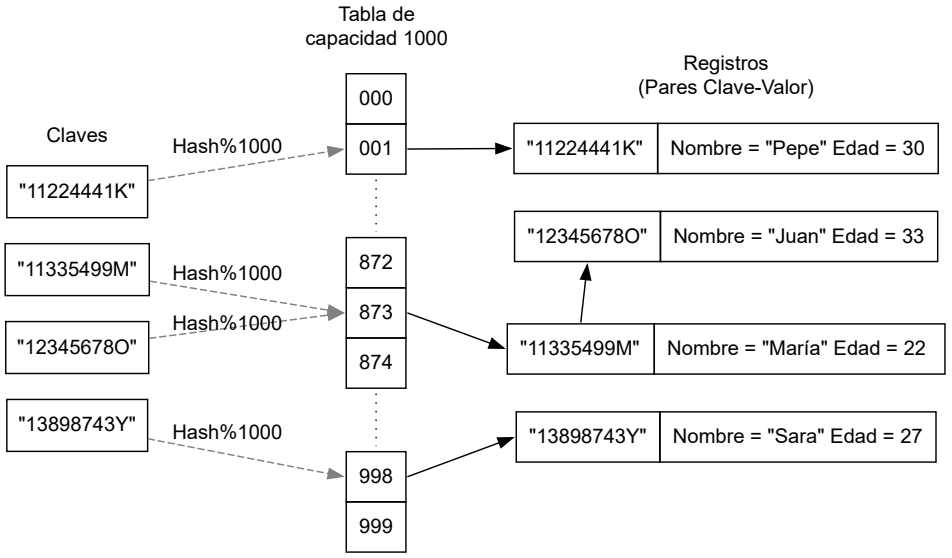
Kotlin:

```kotlin
val personas = mutableMapOf(
    "11224441K" to Persona("Pepe", 30),
    "11335499M" to Persona("María", 22),
    "12345678O" to Persona("Juan", 33),
    "13898743Y" to Persona("Sara", 27)
)

println("Los datos almacenados son:")
for ((dni, persona) in personas) {
    println("- $persona y DNI $dni")
}

println("\nIntroduce un DNI para borrar:")
val dniBuscado = readln()

val personaABorrar = personas[dniBuscado]
val salida = if (personaABorrar != null) {
    "$personaABorrar ha sido borrado"
} else {
    "No se ha encontrado el DNI $dniBuscado"
}

personas.remove(dniBuscado)
println(salida)

println("\nLos DNIs almacenados son:")
println("- ${personas.keys.joinToString("\n- ")}")
```

Profundizando en el uso de los diccionarios

Para entender alguna de las propiedades y métodos que implementa la clase `Dictionary<K, V>` deberemos entender como funcionan los mismos internamente. Para ello, supongamos la tabla de pares clave-valor que hemos puesto antes de ejemplo. Donde la clave era un `string` con el DNI y el valor un objeto de tipo `Persona`.

Una posible **aproximación** a como se organiza la información internamente podría ser el siguiente....



Si nos fijamos, internamente el diccionario guarda una tabla de `n` elementos de capacidad y esto es importante para que funcionen de manera eficiente. De hecho si a priori conocemos cuantos elementos va a tener diccionario podremos dimensionarla en el constructor. Por ejemplo, si queremos que tenga una capacidad de **1000** como en nuestro ejemplo podríamos construir el diccionario de la siguiente manara ...

```
Dictionary<string, Persona> personas = new (1000);
```

Internamente calcula el Hash de la **clave**. En C# el Hash es un valor numérico entero que se obtiene a través del método **GetHashCode()** que cualquier objeto implemente por estar definido en la clase `Object` como virtual y que por tanto podremos invalidar en nuestras definiciones de tipos.

Si consultamos como funciona la **función de Hash** por ejemplo en la **Wikipedia** podemos deducir que, para un determinado estado de un objeto, esta me devuelve un valor numérico '*único*'.

Pero... ¿Para que nos sirve calcular el Hash de la clave?. Si nos fijamos en el diagrama, lo que hace el diccionario es calcular el módulo del Hash entre la capacidad de la tabla `Hash % 1000` esto nos asegurará obtener un resto entre `0 y 999` que son índices válidos para la tabla definida. Por tanto, podemos inferir que el Hash me permitirá crear una **correspondencia entre** los **objetos usados como clave** (`string` en nuestro caso) y un **índice** dónde guardar **el valor asociado** en la tabla, tal y como se muestra en la ilustración.

Pero... ¿Si la tabla tiene poca capacidad entonces, las posibilidades de que el resto de dividir el Hash de dos claves por el tamaño me de el mismo índice es muy alta?. Efectivamente, por eso si nos fijamos en la ilustración hay dos DNI que al dar el mismo índice se deben guardar en la misma posición, por ejemplo a través de una lista enlazada. Una vez vamos a esa posición de la tabla deberemos buscar la clave en la lista. Es por esa

razón, que cada registro de la lista guarda pares de clave-valor y no únicamente el valor. Además, para saber buscar la clave en la lista deberemos saber si dos claves son iguales y por ello **los objetos que usemos como clave deben implementar** la interfaz **IEquatable<T>**. Ten en cuenta que no debes preocuparte por **los tipos básicos y los definidos en las BCL**, puesto que `IEquatable<T>` está implementado en todos ellos.

Resumen

Podemos decir que **para que un tipo pueda hacer de clave** debe invalidar `GetHashCode()` y debe implementar `IEquatable<T>`. Afortunadamente, no debemos preocuparnos porque **los tipos básicos y los definidos en las BCL** cumplen estas condiciones.

Además, del funcionamiento descrito podemos entender los siguiente métodos de optimización y rendimiento de los diccionarios ...

1. `int EnsureCapacity(int capacity)`
   EnsureCapacity me permitirá definir el tamaño de la tabla interna del diccionario si tengo muchas entradas evitando así que se repitan índices en las claves.
2. `void TrimExcess()`
   TrimExcess si he realizado un Clear() o voy a tener pocos elementos en el diccionario me permitirá reducir el tamaño de la tabla adecuándolo al número de entradas.

Usando tipos propios como claves

Imaginemos que queremos definir un diccionario donde la **clave** ahora será la clase `Persona` que hemos utilizado a lo largo del tema y **el valor será una lista de cadenas con los nombres de las mascotas de esa persona**. La definición básica del tipo `Persona` para usarlo como clave sería la siguiente...

```
class Persona : IEquatable<Persona>
{
    public string Nombre { get; }
    public int Edad { get; private set; }
    public Persona(string nombre, int edad)
    {
        Nombre = nombre;
        Edad = edad;
    }
    // Es interesante que redefinamos el ToString para que represente el estado del
    // objeto mediante una cadena.
    public override string ToString() => $"{Nombre} {Edad} años";
    // Invalidamos GetHashCode() y una forma simple es usar la función
    // HashCode.Combine(...) para generar el hashcode a partir de los parámetros.
    public override int GetHashCode() => HashCode.Combine(Nombre, Edad);
    // Implementamos el interfaz, que nos obliga a implementar Equals y podemos
    // comparar fácilmente dos objetos, viendo si tienen el mismo Hash o no.
    public bool Equals(Persona? o)
    => o != null && Nombre == o.Nombre && Edad == o.Edad;
}
```

> ⚠ **Cuidado**
>
> Comparar por `HasCode` en el `Equals` **no sería una opción válida** ya que según la longitud del Hash existe la posibilidad de que objetos diferentes me devuelvan el el mismo Hash.

Analicemos la siguiente propuesta código de ejemplo comentado donde usamos la clase `Persona` que hemos definido como clave...

```
// Definimos el diccionario donde la clave es una persona y el valor una lista de mascotas.
Dictionary<Persona, List<string>> mascotasXPersona = [];

// Creamos un objeto persona pepe y para ese objeto añadimos una lista vacía de mascotas.
Persona pepe = new("Pepe", 30);
mascotasXPersona.Add(pepe, []);
```

```
// Usamos la misma referencia al objeto pepe para acceder
// a su lista de mascotas y añadir dos nombres.
mascotasXPersona[pepe].Add("Snowball");
mascotasXPersona[pepe].Add("Velvet");

// Creamos un objeto persona de nombre María del que no nos guardamos la
// referencia y añadimos una lista inicializada en la definición con dos mascotas.
mascotasXPersona.Add(new("María", 22), ["Simba", "Bella"]);

// Añadimos una tercera mascota a María, pero volvemos a instanciar otro objeto
// Persona para María porque no nos guardamos la referencia como con pepe.
// No debería ser problema porque ambos deberían generar el mismo Hash y además
// sabemos comparar objetos persona con Equal.
mascotasXPersona[new("María", 22)].Add("Lucy");

// Mostramos la lista de mascotas por persona.
foreach (Persona p in mascotasXPersona.Keys)
    Console.WriteLine($"{p}: {string.Join(", ", mascotasXPersona[p])}");
```

Prueba a ejecutar **este código** y comprueba que funciona correctamente. La salida debería ser similar a la siguiente:

```
Pepe 30 años: Snowball, Velvet
María 22 años: Simba, Bella, Lucy
```

Prueba ahora a eliminar la invalidación de `GetHashCode()` de `Persona` y ver que sucede. Deberías obtener un error al añadir la mascota **Lucy** a **María** porque no encontraría la clave en el diccionario.

Consejo

En este caso como persona no es una entidad al no tener un Id como pudiera ser un DNI. Podríamos definir la clase `Persona` como un `record` y ya no tendremos que preocuparnos por implementar `IEquatable<T>` ni invalidar `GetHashCode()` ya que el compilador lo hará por nosotros basándose en los parámetros del constructor. Por tanto, la definición de `Persona` quedaría mucho más simple...

```
record Persona (string Nombre, int Edad)
{
    public override string ToString() => $"{Nombre} {Edad} años";
}
```

Haz la prueba y comprueba que el código funciona igual.

🎓 Caso de estudio:

Vamos su uso a través de un simple **programa de ejemplo** que realice un pequeño examen sobre las capitales de la UE. Para ello, el programa preguntará 5 capitales. Puntuando con 2 puntos cada pregunta acertada. Veamos una posible solución...

```
Dictionary<string, string> capitalesPorPais = new()
{
    ["España"] = "Madrid", // Par clave país, valor capital.
    ["Portugal"] = "Lisboa",
    ["Francia"] = "Paris",
    ["Irlanda"] = "Dublin"
};
// Aunque hemos definido por extensión. Podemos añadir elemetos a posteriori.
capitalesPorPais.Add("Belgica", "Bruselas");
capitalesPorPais["Alemania"] = "Berlin";

// Obtenemos una lista de claves indizable por un entero.
List<string> paises = [..capitalesPorPais.Keys];
// Lista donde almacenaré los países ya preguntados para no repetirnos.
List<string> paisesPreguntados = [];
const int NUMERO_PREGUNTAS = 5;
Random semilla = new();
int puntos = 0;
for (int i = 0; i < NUMERO_PREGUNTAS; i++)
{
    string paisPreguntado;
    // Buscamos un país que ún no hayamos preguntado.
    do
    {
        paisPreguntado = paises[semilla.Next(0, paises.Count)];
    } while (paisesPreguntados.Contains(paisPreguntado));
    paisesPreguntados.Add(paisPreguntado);

    Console.Write($"¿Cual es la capital de {paisPreguntado}? > ");
    string capitalRespondida = Console.ReadLine()!.ToUpper();
    bool respuestaCorrecta = capitalRespondida == capitalesPorPais[paisPreguntado].ToUpper();
    if (respuestaCorrecta) puntos += 2;
    string mensaje = (respuestaCorrecta
                        ? "Correcto !!"
                        : $"Incorrecto !!\nLa respuesta es {capitalesPorPais[paisPreguntado]}.")
                        + $"\nLlevas {puntos} puntos.\n";
    Console.WriteLine(mensaje);
}
Console.WriteLine($"Tu nota final es {puntos}.");
```

Lista doblemente enlazada o vinculada

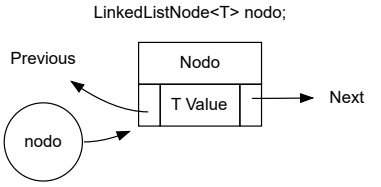
Implementan el TAD (Tipo Abstracto de Datos) de programación clásico denominado: **'lista doblemente enlazada'**.

Este tipo está definido de forma análoga en ostros lenguajes como **Java** o **Kotlin** y es el tipo más adecuado, si voy a necesitar hacer muchas **inserciones** y **borrados** pues tienen un coste `O(1)` sin en embargo, el acceso a un elemento por índice es `O(n)` por lo que s**i necesito hacer muchas búsquedas por índice** es mejor usar una `List<T>`.

Tipo nodo doblemente enlazado LinkedListNode<T>

La clase `LinkedListNode<T>` representará un **nodo de la lista** y contendrá tres propiedades principales:

1. `Value`: que contendrá el valor del nodo de tipo `T`.
2. `Next`: que contendrá una referencia al siguiente nodo de la lista o `NULL` si es el **último nodo**.
3. `Previous`: que contendrá una referencia al nodo anterior de la lista o `NULL` si es el **primer nodo**.

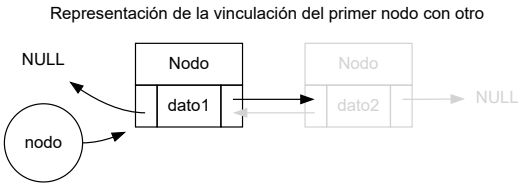


Representación de un nodo doblemente enlazado

Básicamente, cada nodo sabe cual es el siguiente y cual es el anterior y además contiene el valor o el **dato que queremos almacenar** que podrá ser de cualquier tipo `T` ya sea valor o referencia.

Esta estructura, va a permitir que **los datos no estén almacenados de forma contigua en memoria** como en el caso de los arrays o las listas, sino que **cada nodo podrá estar en cualquier parte de la memoria** y se accederá a ellos a través de las referencias `Next` y `Previous`.

Si te fijas en la ilustración, el primer nodo tiene su referencia `Previous` a `NULL` y el último nodo tiene su referencia `Next` a `NULL` y los nodos no están almacenados de forma contigua en memoria. Sino que la referencia `Next` de cada nodo apunta a la dirección de memoria del siguiente nodo y me permitirá acceder a él de forma dinámica. Lo mismo sucede con la referencia `Previous` que me permitirá retroceder en la lista.

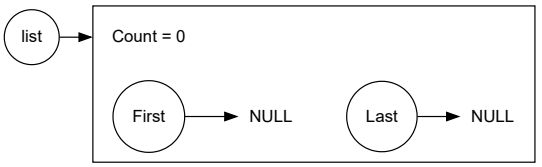


La **ventaja del doble enlace es que puedo recorrer la lista en ambos sentidos**, desde el primero al último nodo y viceversa.

Tipo lista doblemente enlazada LinkedList<T>

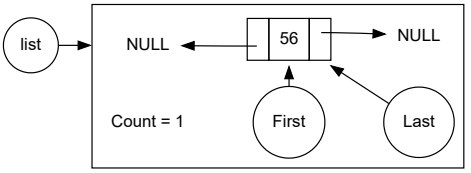
Representa la lista doblemente enlazada en sí misma y contendrá referencias al **primer nodo** a través de la propiedad `First` y al **último nodo** de la lista a través de la propiedad `Last`. También tendrá una propiedad `Count` que me indicará el número de nodos que tiene la lista. Además, implementa la interfaz `ICollection<T>` por lo que tiene las operaciones básicas de cualquier colección.

Si creamos una lista doblemente enlazada vacía, las referencias `First` y `Last` apuntarán a `NULL` y el contador `Count` valdrá 0.



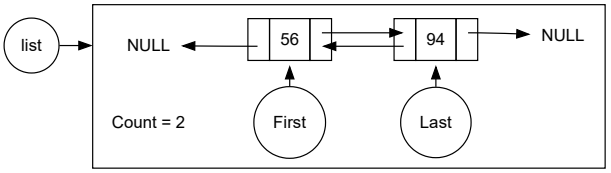

```
LinkedList<int> list = new ();
```

Si inicializamos la lista con un solo elemento, las referencias `First` y `Last` apuntarán al mismo nodo y el contador `Count` valdrá 1.




```
LinkedList<int> list = new ([56]);
```

Si inicializamos la lista con dos elementos, la referencia `First` apuntará al primer nodo y la referencia `Last` al segundo nodo. El contador `Count` valdrá 2.



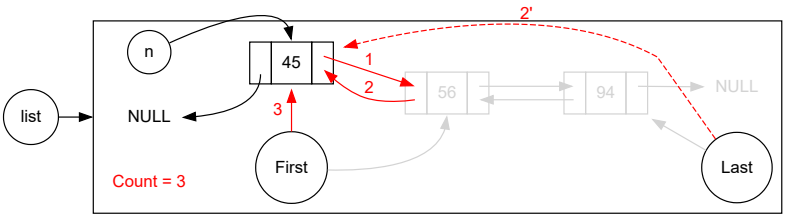

```
LinkedList<int> list = new ([56, 94]);
```

Este proceso de **mantener las referencias al primer y último nodo** y la **cuenta de nodos**, se irá repitiendo a medida que vayamos añadiendo o borrando nodos de la lista. Para ello dispondremos de diferentes métodos que realizarán las operaciones necesarias para mantener la integridad de las referencias en la lista.

Añadir elementos al principio con AddFirst

Añadirá un nodo al principio o la '*cabeza*' de la lista.

```
LinkedList<int> list = new ([56, 94]);
list.AddFirst(45);
```



Pero si la lista ya tenía nodos, se realizará un proceso por pasos como el descrito en la imagen:

Paso 1: Se crea un nuevo nodo `n` con el valor a añadir y haremos que `Next` apunte al nodo que actualmente es el primero de la lista (`First`).
Paso 2: Si la lista no estaba vacía, haremos que el nodo que actualmente es el primero de la lista (`First`) tenga su referencia `Previous` apuntando al nuevo nodo `n` y si la lista estaba vacía, haremos que la referencia `Last` apunte al nuevo nodo `n` (**Paso 2'**).
Paso 3: Finalmente, actualizaremos la referencia `First` para que apunte al nuevo nodo `n` y aumentaremos el contador `Count` en 1.

Todo este proceso se realizará de forma transparente al usuario cuando llame al método `list.AddFirst(45);` .

```
LinkedList<int> list = new ([56, 94]);
LinkedListNode<int> n = new (45);
list.AddFirst(n);
```

También podremos crea nosotros mismos el nodo y pasarlo al método `AddFirst(LinkedListNode<T> node)` . El proceso interno será el mismo que en el caso anterior pero con el nodo ya creado.

> ⚠ **Cuidado**
>
> Una vez añadido el nodo a la lista, pasará a '*ser de su propiedad*' y se encargará de gestionarlo. Por tanto, **no debemos usar más la referencia al nodo** que hemos pasado como argumento.
> Si intentamos añadir el mismo nodo a otra lista o a la misma lista de nuevo, obtendremos una excepción.
> Lo mismo sucederá si intentamos añadir un nodo que ya pertenece a otra lista.

Añadir elementos al final con AddLast

Añadirá un nodo al final o la '*cola*' de la lista.

```
LinkedList<int> list = new ([45, 56, 94]);
list.AddLast(78);
```

El proceso será análogo al de `AddFirst` pero en este caso actualizaremos las referencias `Last` y `Next` de los nodos implicados.

Añadir o insertar antes o después de un nodo

Podremos añadir un nuevo nodo antes o después de un nodo **ya existente** en la lista. Para ello usaremos los métodos `AddBefore(LinkedListNode<T> node, T value)` y `AddAfter(LinkedListNode<T> node, T value)` respectivamente.

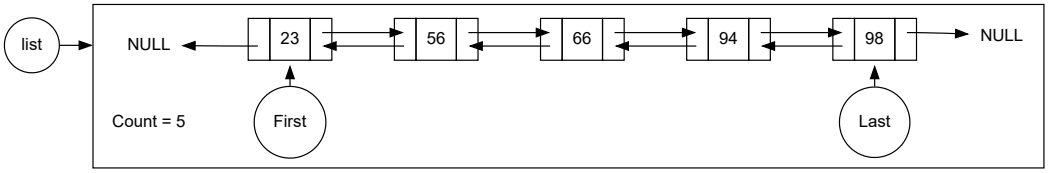
En el ejemplo de código anterior. Buscaremos el nodo con el valor `94` y si lo encontramos, añadiremos un nuevo nodo con el valor `66` antes de ese nodo y otro con el valor `98` después de ese nodo (fíjate que es como hacer un `AddLast`). Finalmente, añadiremos un nuevo nodo con el valor `23` antes del primer nodo de la lista, equivalente a hacer un `AddFirst` pero a diferencia de este último, la lista no está vacía y por eso nos

```
LinkedList<int> list = new ([56, 94]);
LinkedListNode<int>? nodo94 = list.Find(94);
if (nodo94 != null)
{
    list.AddBefore(nodo94, 66);
    list.AddAfter(nodo94, 98);
}
list.AddBefore(list.First!, 23);
Console.WriteLine(string.Join(", ", list));
```

aseguramos de pasar el nodo primero con `list.First!` (operador de supresión de null). El código **mostrará por consola** ...

Mostrará por consola:

```
23, 56, 66, 94, 98
```



Borrar elementos con Remove

Podremos borrar directamente el primero o el último nodo de la lista con los métodos `RemoveFirst()` y `RemoveLast()` respectivamente.

Además, como sucede con otras operaciones, tendremos la posibilidad de borrar un nodo a través de su valor o a través de una referencia al nodo.

1. `bool LinkedList<T>.Remove(T value)`: Busca el primer nodo con el valor indicado y si lo encuentra lo borra devolviendo `true`, en caso contrario devuelve `false`.
2. `void LinkedList<T>.Remove(LinkedListNode<T> node)`: Borra el nodo indicado a través de su referencia. Si el nodo no pertenece a la lista se generará una excepción.

```
LinkedList<int> list = new([23, 56, 66, 94, 98]

LinkedListNode<int>? nodo66 = list.Find(66);
if (nodo66 != null)
    list.Remove(nodo66);
list.Remove(56);
list.RemoveFirst();
list.RemoveLast();

Console.WriteLine(string.Join(", ", list));
```

En el ejemplo de código anterior. Buscaremos el nodo con el valor `66` y si lo encontramos, lo borraremos a través de su referencia. A continuación, borraremos el nodo con el valor `56` a través de su valor. Finalmente, borraremos el primer y el último nodo de la lista.
El código **mostrará por consola** ...

```
94
```

Como vemos tras todos los borrados, únicamente queda el nodo con el valor `94` en la lista y las referencias `First` y `Last` apuntarán a ese nodo.

Recorriendo una LinkedList<T>

No podremos usar un índice entero para acceder a los elementos de la lista y por tanto el operador `[]` no está definido para este tipo. Esto es debido a que el acceso por índice es costoso en este tipo de listas ya que deberíamos recorrer la lista desde el primer nodo hasta llegar al índice indicado y esto tiene un coste `O(n)`.

Podremos recorrer los datos con un `foreach` de la siguiente manera:

```
LinkedList<int> list = new([23, 56, 66, 94, 98]);
foreach (int valor in list)
    Console.WriteLine(valor);
```

También podremos recorrerla en ambos sentidos **con un nodo a modo de iterador**.

```
LinkedList<int> list = new([23, 56, 66, 94, 98]);

for (LinkedListNode<int>? it = list.First; it != null; it = it.Next)
    Console.WriteLine(it.Value);

// Recorrido inverso.
for (LinkedListNode<int>? it = list.Last; it != null; it = it.Previous)
    Console.WriteLine(it.Value);
```

Transformando LinkedList<T> en otras colecciones

Ya hemos visto que el constructor me permite crear una lista a partir de una array u otra colección. Pero, ¿Cómo transformamos de nuevo la lista en un array **si hemos terminado de por ejemplo de hacer inserciones y borrados**?

En C# es tan sencillo como usar expresiones de colección con el operador de extensión `[..]`.

```
LinkedList<int> linkedList = new([23, 56, 66, 94, 98]);

int[] array = [.. linkedList];
List<int> list = [.. linkedList];
```

List vs LinkedList

Aunque al principio del tema hemos comentado que las listas enlazadas son más apropiadas para realizar muchas inserciones y borrados, en la práctica **las diferencias de rendimiento no son tan evidentes**. Es por ello que salvo **Java**, **C#**. Otros lenguajes como **Python**, **JavaScript** o **Kotlin** no disponen de este tipo de listas en sus librerías estándar.

¿Por qué en la práctica en un 95% de los casos es mejor y más eficiente usar `List<T>` en lugar de `LinkedList<T>`?

1. **El castigo del "*Cache Miss*"**: Las listas enlazadas no almacenan los datos de forma contigua en memoria. Por tanto, el procesador no puede aprovechar la localidad espacial de los datos y se producen más fallos de caché (cache misses) al acceder a los nodos, lo que ralentiza el rendimiento.
2. **Mayor sobrecarga de memoria *"Overhead"***: Las listas enlazadas son "despilfarradoras" con la memoria. Ya que por ejemplo, en un `List<int>` solo guardas los enteros. Sin embargo, en una `LinkedList<int>` por cada entero guardas...
    - El valor entero (**4 bytes**).

- Un puntero al nodo siguiente (**8 bytes**).
- Un puntero al nodo anterior (**8 bytes**).
- Referencia al objeto LinkedList al que pertenece el nodo (**8 bytes**).
- Metadatos del objeto Nodo que contiene los datos (**8 bytes**).

Con todo ello, **una simple lista doblemente enlazada consume aproximadamente 5 o 6 veces la memoria que una lista normal**.

3. **Operaciones de inserción y borrado no tan rápidas**: La teoría dice que insertar en una `LinkedList` es `O(1)` (tiempo constante), mientras que en una `List` es `O(n)` porque hay que desplazar elementos. Sin embargo:
   - Salvo en los extremos, para insertar en una lista enlazada, primero tienes que encontrar el nodo. Esa búsqueda es `O(n)`.
   - En la práctica, añadir a una listas es bastante rápido gracias a que **los procesadores están optimizados para mover bloques de memoria contiguos**. Sobre todo si la lista no es muy grande.
   - Crear un nodo y gestionar las referencias en una lista enlazada también tiene su coste temporal

Entonces..., ¿Cuándo usar `LinkedList<T>` en lugar de `List<T>`?

1. **Inserciones y borrados muy frecuentes**: Si la aplicación requiere muchas inserciones y borrados **en posiciones arbitrarias de la lista** y el **acceso por índice es inexistente**.
2. **Sistemas Operativos**: Para gestionar colas de procesos donde el tamaño cambia constantemente y no puedes permitirte "re-alojar" grandes bloques de memoria.
3. **Implementación de otras estructuras**: Son la base para crear **Queues** (Colas) o **Stacks** (Pilas) eficientes.
4. **Sistemas con memoria muy fragmentada**: Donde no hay espacio para un bloque contiguo grande, pero sí para muchos trozos pequeños.
5. **Almacenamos un '*value type*' muy grande**: Si el valor que almacenamos es un **record struct** muy grande, puede ser más eficiente usar una `LinkedList` para evitar copiar grandes bloques de memoria al redimensionar una `List`.

Resumen

Operación	List<T>	LinkedList<T>	Ganador
Acceso por índice	$O(1)$ (Instantáneo)	$O(n)$ (Debe recorrer)	List
Añadir al final	$O(1)$	$O(1)$	Empate
Insertar/Borrar en el medio	$O(n)$ (Desplaza datos)	$O(1)$ (Solo referencias) *	LinkedList
Uso de Memoria	Bajo / Eficiente	Alto (Punteros extra)	List
Caché del CPU	Excelente	Pobre	List

* Asumiendo que ya tienes el nodo localizado*

Pilas (Stacks) y Colas (Queues)

Se trata de dos tipos de colecciones muy usadas en la programación tradicional que implementan los TAD (Tipos Abstractos de Datos) **Pila** y **Cola** respectivamente. Son tipos muy sencillos que **nos permiten almacenar datos de forma ordenada y acceder a ellos siguiendo unas reglas muy concretas**.

De hecho, ambos tipos son **concreciones** del TAD **Lista** pero con restricciones en las operaciones que podemos realizar sobre ellos. De esta manera nos aseguramos que los datos se gestionan siguiendo dichas operaciones y no otras.

Existen muchos algoritmos clásicos que usan ambos tipos de colecciones para resolver problemas concretos. Como hemos comentado, podríamos usar una `List<T>` o una `LinkedList<T>` pero al tener estos tipos las operaciones específicas que necesitamos, el código es más claro y sencillo de mantener.

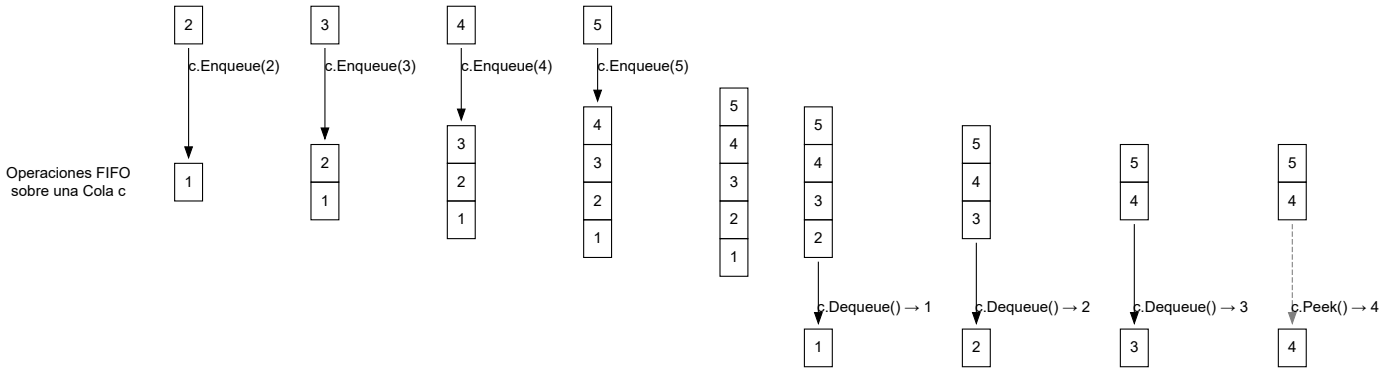
Internamente en C#, ambos tipos usan un `Array` para almacenar los datos como base al igual que `List<T>`. Por tanto, las operaciones de inserción y borrado tienen un coste amortizado de `O(1)` y el acceso a los elementos es `O(1)`.

Las colas en CSharp con (Queue<T>)

Los elementos se añaden por el final y se suprimen por el principio denominado **frente de la cola**. Por esta razón, también se les conoce como estructuras **FIFO**, acrónimo en ingles de *'**Primero** en Entrar, **Primero** en Salir'*.

Operaciones del TAD Cola y su equivalente en C# a través de `Queue<T> c = new()`:

- **Encolar** (`c.Enqueue(T dato)`).
- **Desencolar** (`c.Dequeue() → T`).
- **Borrar** toda la cola (`c.Clear()`)
- **Consultar el frente de la cola sin desencolar** (`c.Peek() → T`).
- Ver si está **vacía** (`c.Count == 0`).



El siguiente código equivaldría a las operaciones descritas en la ilustración y **mostrará por consola**:

```
Cola -> 1
Encolando el 2
Cola -> 1, 2
Encolando el 3
Cola -> 1, 2, 3
Encolando el 4
Cola -> 1, 2, 3, 4
Encolando el 5
Cola -> 1, 2, 3, 4, 5
Desencolado el 1
Cola -> 2, 3, 4, 5
Desencolado el 2
Cola -> 3, 4, 5
Desencolado el 3
Cola -> 4, 5
El frente de la cola es 4
Cola -> 4, 5
```

```
public static void Main()
{
    Queue<int> c = new ([1]);
    Console.WriteLine($"Cola -> {string.Join(", ", c)}");

    for (int i = 2; i <= 5; i++)
    {
        c.Enqueue(i);
        Console.WriteLine($"Encolando el {i}");
        Console.WriteLine($"Cola -> {string.Join(", ", c)}");
    }

    while (c.Count > 2)
    {
        Console.WriteLine($"Desencolado el {c.Dequeue()}");
        Console.WriteLine($"Cola -> {string.Join(", ", c)}");
    }

    Console.WriteLine($"El frente de la cola es {c.Peek()}");
    Console.WriteLine($"Cola -> {string.Join(", ", c)}");
}
```

Como hemos comentado, aunque a simple vista son una concreción de las listas, puede ser interesante tener solo un subconjunto de operaciones más limitado, enfocado a ciertos problemas y que nos pueda proporcionar más '*robustez*' de cara a errores, legibilidad del código e incluso rendimiento en la ejecución del mismo. Normalmente las colas se usar para gestionar datos, eventos o procesos asíncronos por orden de llegada. El ejemplo más paradigmático sería la **cola de impresión**, que irá encolando los '*trabajos*' de impresión y desencolándolos por orden conforme quede libre un recurso de impresión.

Ejemplo:

Vamos a implementar con una Cola programa que simule **'*El juego de la bomba*'** en el cual una bomba programada para explotar en un tiempo aleatorio. Es pasada entre los jugadores de forma rotativa del tal manera que cada jugador la retendrá un tiempo 'aleatorio' que el estime oportuno para que le explote a un compañero y no le vuelva a llegar.

Esto es, el jugador que coge la bomba se d**esencolará del frente de la cola**, esperará un tiempo aleatorio y si la bomba no ha explotado, **volverá a encolarse** pasándosela al siguiente jugador que desencolaremos a continuación y así sucesivamente hasta que la bomba explote. Una posible solución sería la siguiente...

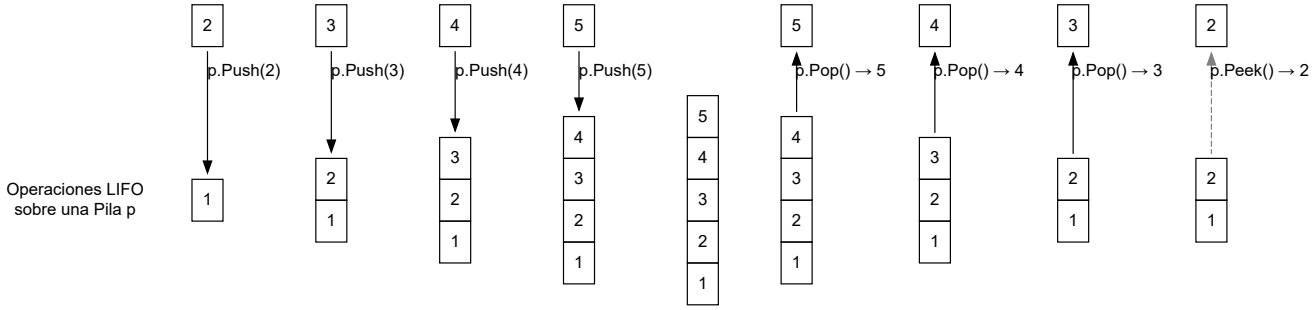
```
public static void Main()
{
    Queue<string> jugadores = new Queue<string>([
        "Pepe", "María", "Juan", "Sara"
    ]);
    Random seed = new ();
    int segundosHastaExplosion = seed.Next(30, 60);
    bool explosion;
    do
    {
        // Desencolamos al jugador que recibirá la bomba.
        string jugador = jugadores.Dequeue();
        int espera = seed.Next(5, 10);
        Console.WriteLine($"{jugador} esperando para pasar la bomba.");
        Console.WriteLine($"En cola quedán {string.Join(", ", jugadores)}");

        Thread.Sleep(espera * 1000); // Espera aleatoria.
        segundosHastaExplosion -= espera;
        explosion = segundosHastaExplosion <= 0; // Comprobamos si ha explotado.
        if (!explosion)
            // Si no ha explotado encolamos de nuevo al jugador para que vuelva a recibir la bomba.
            jugadores.Enqueue(jugador);
        else
            Console.WriteLine($"La bomba le ha explotado a {jugador}.");
    }
    while (!explosion);
    jugadores.Clear();
}
```

Las pilas en CSharp con (Stack<T>)

Los elementos se añaden y extraen **por el mismo extremo** que denominaremos **cabeza de la pila**. Por esta razón, también se les conoce como estructuras **LIFO**, acrónimo en ingles de *'**Último** en Entrar, **Primero** en Salir'*.

Operaciones del TAD Pila y su equivalente en C# a través de `Stack<T> p = new()`:

- **Apilar** (`p.Push(T dato)`).
- **Desapilar** (`p.Pop() → T`).
- **Borrar** toda la pila (`p.Clear()`)
- **Consultar la cabeza** de la cola sin desapilar (`p.Peek() → T`).
- Ver si está **vacía** (`p.Count == 0`).



El siguiente código equivaldría a las operaciones descritas en la ilustración y **mostrará por consola**:

```
Pila -> 1
Apilando el 2
Pila -> 2, 1
Apilando el 3
Pila -> 3, 2, 1
Apilando el 4
Pila -> 4, 3, 2, 1
Apilando el 5
Pila -> 5, 4, 3, 2, 1
Desapilando el 5
Pila -> 4, 3, 2, 1
Desapilando el 4
Pila -> 3, 2, 1
Desapilando el 3
Pila -> 2, 1
La cabeza de la pila es 2
Pila -> 2, 1
```

```csharp
public static void Main()
{
    Stack<int> p = new ([1]);

    Console.WriteLine($"Pila -> {string.Join(", ", p)}");

    for (int i = 2; i <= 5; i++)
    {
        p.Push(i);
        Console.WriteLine($"Apilando el {i}");
        Console.WriteLine($"Pila -> {string.Join(", ", p)}");
    }

    while (p.Count > 2)
    {
        Console.WriteLine($"Desapilando el {p.Pop()}");
        Console.WriteLine($"Pila -> {string.Join(", ", p)}");
    }

    Console.WriteLine($"La cabeza de la pila es {p.Peek()}");
    Console.WriteLine($"Pila -> {string.Join(", ", p)}");
}
```

Ejemplo:

Un típico ejemplo de uso de pilas es comprobar si una **expresión está balanceada** o no. Por ejemplo, $((3+4)*(2-5/(2*4)))$ está balanceada porque se abren tantos paréntesis como se cierran de forma correcta.

Para ello, vamos a definir un método estático denominado `bool VerificaParentesis(string expresion)` que me diga si la cadena con la expresión de entrada lo está o no.

El algoritmo consistirá en recorrer la cadena y cada vez que encontremos el caracter `'('` **apilarlo** y cuando encontremos un `')'` **desapilar** de la cadena. Si desapilamos y no hay nada en la pila, significará que no hay ningún paréntesis de apertura para el de cierre que acabamos de encontrar. Además, si tras recorrer todas la expresión quedan elementos en la pila significará que no todos los paréntesis de apertura se has logrado cerrar.

```
static bool VerificaParentesis(string expresion)
{
    Stack<char> p = new ();
    bool balanceados = true;

    for (int i = 0; i < expresion.Length && balanceados; i++)
    {
        if (expresion[i] == '(')
            p.Push('(');
        else if (expresion[i] == ')')
            balanceados = p.Count > 0 && p.Pop() == '(';
    }

    balanceados = balanceados && p.Count == 0;
    return balanceados;
}

static void Main()
{
    Console.WriteLine(VerificaParentesis("((3 + 4) * (2 - 5 / (2 * 4)))")); // true
    Console.WriteLine(VerificaParentesis("(3 + 4) * (2 - 5 / (2 * 4)))")); // false
    Console.WriteLine(VerificaParentesis("((3 + 4) * (2 - 5 / (2 * 4))"));  // false
}
```

🚀 **Ampliación opcional:**

Vamos a implementar la función `VerificaParentesis` el ejemplo anterior en otros lenguajes de programación como **JavaScript** y **Python**. Fíjate que aunque en ambos lenguajes no existe un tipo específico para pilas, podemos usar los arrays que tienen métodos equivalentes a `push` y `pop` para simular su comportamiento.

JavaScript (Node.js):

```javascript
function verificaParentesis(expresion) {
    const pila = [];
    let balanceados = true;

    for (let i = 0; i < expresion.length && balanceados; i++) {
        if (expresion[i] === '(') {
            pila.push('(');
        } else if (expresion[i] === ')') {
            balanceados = pila.length > 0 && pila.pop() === '(';
        }
    }

    balanceados = balanceados && pila.length === 0;
    return balanceados;
}
```

Python:

```python
def verifica_parentesis(expresion):
    p = []
    balanceados = True
    i = 0

    while i < len(expresion) and balanceados:
        caracter = expresion[i]
        if caracter == '(':
            p.append('(')
        elif caracter == ')':
            if len(p) > 0:
                p.pop()
            else:
                balanceados = False
        i += 1

    return balanceados and len(p) == 0
```

El patrón iterador

Es un **patrón de diseño orientado a objetos** ideado para recorrer cualquier colección de datos. Además, en el lenguaje C# cobra especial importancia para aplicar ciertos esquemas de **programación funcional** que trataremos en temas posteriores.

Este patrón se implementa en C# a través de la definición de la interfaz genérica `IEnumerable<T>` así como su antecesora `IEnumerable`. Ambas ofrecen un mecanismo para la **iteración sobre los elementos de una secuencia**, generalmente con la vista puesta en aplicar a esa secuencia la instrucción de programación `foreach` para recorrerla.

Dicho interfaz deberá ser implementada por la clase o tipo que implemente la secuencia o contenga una serie de datos que nos interese recorrer de forma secuencial. Por tanto, todas las colecciones genéricas que hemos visto en este tema como `List<T>`, `LinkedList<T>`, `Stack<T>` o `Queue<T>` implementan dicha interfaz.

Representará pues la abstracción máxima de una secuencia iterable de datos y nos permitirá recorrerlos sin preocuparnos de la implementación interna de la colección que los contiene. Por tanto, es usada como abstracción en multitud de métodos definidos en las librerías estándar de C# que necesitan recorrer secuencias de datos como por ejemplo el método estático `public static string Join<T>(string? separator, IEnumerable<T> values)` que venimos usando durante todo el curso para mostrar colecciones por consola sparadas por comas u otro separador. **Este métiodo, solo necesita recorrer la secuencia de datos que le pasemos sin importar si es una lista, una pila, una cola o cualquier otra colección que implemente la interfaz** `IEnumerable<T>`.

Interfaz IEnumerable<T>

Definirá un **método que me permitirá ontener intanciás de un iterador** que implemente la interfaz `IEnumerator<T>` que veremos a continuación.

La definición de `IEnumerable<T>` e `IEnumerable` es la siguiente:

```
// Definido dentro de System.Collections
public interface IEnumerable
{
    IEnumerator GetEnumerator();
}

// Definido dentro de System.Collections.Generic
public interface IEnumerable<T> : IEnumerable
{
    IEnumerator<T> GetEnumerator();
}
```

> **Nota**

> Fíjate que la interfaz se apoya en su contrapartida no genérica, para tener compatibilidad con la misma y por tanto hacia versiones anteriores de las BCL.

Operaciones y uso básico de la abstracción IEnumerable<T>

Como yahemos comentado en la introducción, todas las colecciones implementan este interfaz, podremos hacer la **sustitución** de cualquier colección concreta por su interfaz genérica `IEnumerable<T>`. Por ejemplo, podríamos definir un método que reciba como parámetro cualquier colección que implemente dicha interfaz y nos muestre sus elementos por consola.

```
static void MostrarElementos<T>(IEnumerable<T> secuencia)
{
    foreach (T elemento in secuencia)
    {
        Console.WriteLine(elemento);
    }
}

static void Main()
{
    int[] array = [23, 56, 66, 94, 98];
    List<int> lista = new([23, 56, 66, 94, 98]);
    LinkedList<string> linkedList = new (["Hola", "Mundo", "Desde", "CSharp"]);

    MostrarElementos(array);
    MostrarElementos(lista);
    MostrarElementos(linkedList);
}
```

Podremos hacer el proceso inverso y generar cualquier colección a partir de una secuencia que implemente `IEnumerable<T>`. Fíjate que según el tipo tendremos diferentes alternativas ...

```
IEnumerable<int> secuencia = [23, 56, 66, 94, 98];

int[] array2 = secuencia.ToArray();
int[] array3 = [.. secuencia];

List<int> lista = new (secuencia);
List<int> lista2 = secuencia.ToList();
List<int> lista3 = [.. secuencia];

LinkedList<int> listaEnlazada1 = new (secuencia);
```

Aunque profundizaremos más adelante en este aspecto. Dispondremos de multitud de métodos de utiliadad aplicables a `IEnumerable<T>`. Algunos ejemplos son:

- Método de clase, `IEnumerable<int> Enumerable.Range(int start, int count)` que genera una secuencia de enteros empezando en `start`, con `count` elementos.

- Método de instancia, `IEnumerable<T> IEnumerable<T>.Skip(int count)` que devolverá la secuencia resultante de haber saltado `count` posiciones en el objeto secuencia al que aplicamos la operación.
- Método de instancia, `T IEnumerable<T>.First()` que devolverá el primer elemento del objeto secuencia al que aplicamos la operación.

```
List<int> secuencia = Enumerable.Range(2, 4).ToList();
List<int> secuencia = [2, 3, 4, 5]; // Equivalente a la línea anterior.
secuencia.Skip(2).First(); // Se evaluará al entero 4
```

Interfaz IEnumerator<T>

Clase **iterador** que nos permitirá recorrer la secuencia de datos de forma ordenada a través de los objetos instanciados de la misma. Es

La definición de `IEnumerator<T>` e `IEnumerator` es la siguiente:

```
// Definido dentro de System.Collections
public interface IEnumerator
{
    object Current { get; }
    void Reset();
    bool MoveNext();
}

// Definido dentro de System.Collections.Generic
public interface IEnumerator<T> : IDisposable, IEnumerator
{
    T Current { get; }
}
```

Como vemos en la definición, las clases que implementen este interfaz deben implementar los **siguientes miembros**:

- `Current` : Devuelve el elemento actual apuntado por el iterador interno en la secuencia.
- `Reset()` : Establecerá el iterador a un estado inicial **justo antes del primer elemento**. Este método, será llamado desde el constructor del objeto y el estado indicará en siguiente `MoveNext()` que debo ir al primer elemento.
- `MoveNext()` : Desplaza el enumerador al siguiente elemento de la secuencia devolviendo `true` si he podido hacerlo o `false` si no he podido avanzar porque ya estaba al final de la misma. Como hemos comentado antes, en la primera llamada tras el `Reset()` se situará al principio de la secuencia.
- `Dispose()` : Restablece el iterador a su valor inicial y libera cualquier recurso no administrado asociado al mismo.

Uso de el patrón iterador

Como ya hemos comentado nos permite recorrer secuencias con la instrucción `foreach` y **pasar como parámetro o guardar** cualquier colección como la **generalización** secuencia iterable `IEnumerable<T>`.

Una instrucción `foreach` ...

```
List<int> lista = new([23, 56, 66, 94, 98]);


foreach (int valor in lista)
{
    Console.WriteLine(valor);
}
```

realmente está haciendo...

```
List<int> lista = new([23, 56, 66, 94, 98]);

IEnumerator<int> it = lista.GetEnumerator();
while (it.MoveNext())
{
    Console.WriteLine(it.Current);
}
```

Pero ... ¿Por qué crear un objeto aparte para iterar y no dejar que las colecciones implementen directamente `IEnumerator<T>` ?

La respuesta tiene que ver con la necesidad de permitir la ejecución de **iteraciones anidadas sobre una misma secuencia**. Si la secuencia implementara directamente la interfaz enumeradora, solo se dispondría de un único '*estado de iteración*' en cada momento y sería imposible implementar bucles anidados `foreach` sobre una misma secuencia.

Veamoslo a través de una ejemplo de uso de dos iteradores sobre un mismo objeto `LinkedList<T>` :

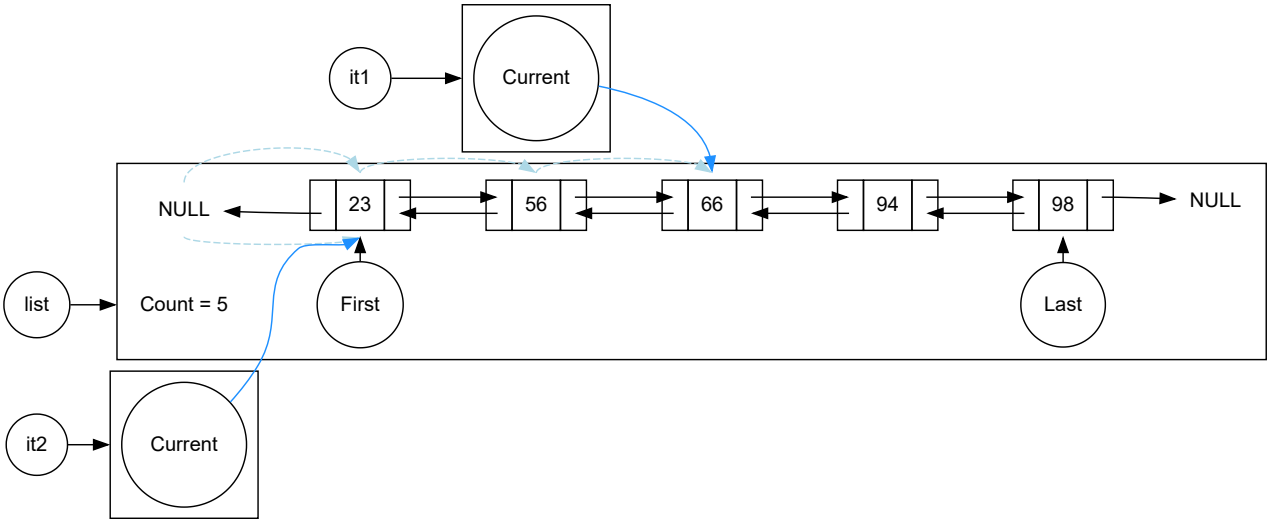
Un doble `foreach` ...

```
LinkedList<int> list = new([23, 56, 66, 94, 98]


foreach (int val1 in list)
{
    foreach (int val2 in list)
    {
        Console.Write($"[{val1}, {val2}]");
    }
}
```

realmente está haciendo...

```
LinkedList<int> list = new([23, 56, 66, 94, 98]);

IEnumerator<int> it1 = list.GetEnumerator();
IEnumerator<int> it2 = list.GetEnumerator();
while (it1.MoveNext())
{
    while (it2.MoveNext())
        Console.Write($"[{it1.Current}, {it2.Current}]");
    it2.Reset();
}
```

Cada objeto iterador `it1` , `it2` mantiene su propio estado de iteración y por tanto, podemos tener varios iteradores activos sobre la misma secuencia sin que interfieran entre ellos.


it1
Current
NULL
23
56
66
94
98
NULL
list
Count = 5
First
Last
it2
Current

Implementando IEnumerable<T> en nuestras colecciones o clases

Siguiendo la definición de los interfaces. La clase que queremos que sea recorrible debería implementar la interfaz `IEnumerable<T>` y definir el método `GetEnumerator()` que devolverá una instancia de una clase que implemente la interfaz `IEnumerator<T>`. Además deberemos definir otra clase que implemente `IEnumerator<T>` con los métodos y propiedades necesarias para recorrer la colección.

Esto en ocasiones puede llegar a ser tedioso y repetitivo. Por suerte, C# nos proporciona una forma mucho más sencilla de implementar el patrón iterador a través del uso de la palabra clave `yield` que veremos a continuación.

Concepto de generación perezosa de secuencias con yield

El concepto de **generador en informática** la vamos a además de en C# en otros lenguajes de programación tales como: **JavaScript**, **Python**, **Kotlin**, etc. Asociado a la normalmente a la palabra reservada **yield** que tendrá un significado y uso similar, aunque con variaciones en la sintaxis.

Es interesante indicar que una posible traducción del inglés del verbo **to yield** es **'*generar o producir*'**. Por lo que normalmente esta instrucción se utiliza para **generar secuencias de forma perezosa** (*lazy evaluation*) **sin necesidad de definir explícitamente las clases** que implementan los interfaces `IEnumerable<T>` e `IEnumerator<T>`.

¿Por qué usar generación perezosa de secuencias?

Imaginemos un escenario, cada día más común, de **big data**, donde vamos a disponer de una gran cantidad de datos a procesar y donde no es tan importante el tiempo de proceso. Si vamos a procesar estos datos en forma de secuencia y los cargamos todos en una colección se nos pueden dar ciertos problemas en el proceso, como:

1. Nos quedamos sin memoria ya que hay demasiados datos y debamos realizar el proceso, cargando en varias secuencias con todo lo que ello conlleva de complejidad final.
2. Los datos pueden cargar en memoria pero tenemos que solicitarlos a un determinado servicio en Internet (endpoint). Sin embargo, son tantos datos que va a tardar mucho en mandármelos todos a la vez, además de que lo vamos a sobrecargar con nuestra petición.
3. Derivado del anterior, no sabemos el tiempo que puede tardar el endpoint en generar cada dato y debemos procesar la secuencia de forma asíncrona. Esto es, el procesador estará atendiendo a otras tareas mientras se genera cada dato y en el momento que se genere un dato lo procesa en la secuencia.
4. Tenemos un stream a un fichero en disco con Terabytes (TB) de información a tratar y queremos aprovechar las funcionalidades de las secuencias para hacerlo.
5. Necesitemos hacer un búsqueda en una gran colección de objetos, de los que solo vamos a necesitar unos pocos hasta encontrar lo que buscábamos. Sin embargo, hemos tenido que cargar previamente todos en la colección para realizar la búsqueda.

En estos casos es mejor ir generando los elementos de la secuencia, conforme los necesitemos para su proceso y no todos a la vez como sucedería al cargarlos en una colección.

Resumen

Cómo puedes ver en el cuadro resumen, si vamos a usar la abstracción `IEnumerable<T>` es mejor usar un generador con `yield` en lugar de crear una colección intermedia como una `List<T>` para obtener la secuencia.

IEnumerable<T> obtenido a partir de ...	**una colección (List)**	**un generador (yield)**
Carga de datos	Todo a la vez (Eager)	Uno a uno (Lazy)
Consumo de RAM	Proporcional al tamaño total	Constante (solo un elemento)
Tiempo de inicio	Lento (debe llenar la lista)	Instantáneo (entrega el primero)
Acceso aleatorio	Al convertirse en IEnumerable<T> pierde la indexación directa y debo iterar secuencialmente	Debo iterar secuencialmente

Uso de yield en CSharp

En C# `yield` irá seguido de las clausulas `return` o `break`.

```
yield return <expression>;
yield break;
```

Vamos a ver a través de un ejemplo sencillo los conceptos vistos en el puntoa anterior. Para ello, supongamos el siguiente código simple donde **NO usamos** `yield` ...

Vamos a definir un método estático denominado `IEnumerable<int> ObtieneMultiplosDeN(int n, int ini, int fin)` que me devuelve **una secuencia genérica** con los múltiplos de un número `n` en el rango `[ini, fin)`.

```
public static IEnumerable<int> ObtieneMultiplosDeN(
                            int n, int ini, int fin)
{
    List<int> multimplos = [];
    // Para ello, vamos añadiendo a una colección dichos números.
    for (int i = ini; i < fin; i++)
    {
        if (i % n == 0)
        {
            // Vamos generando un log del proceso.
            Console.WriteLine($"Obtenido {i}");
            multimplos.Add(i);
        }
    }

    // Hemos tenido que rellenar toda la colección y la
    // retornaremos en su forma de secuencia IEnumerable<T>
    return multimplos;
}
```

Si ejecutamos el **programa principal** y vemos la salida por la consola, comprobado que **hemos generado todos los múltiplos** en el rango dado. Para el rango del ejemplo son 8 pero **podría ser un número muy elevado y que además deberemos tener en un lista en memoria**.

```
Obtenido 320
Obtenido 322
Obtenido 324
Obtenido 326
Obtenido 328
Obtenido 330
Obtenido 332
Obtenido 334
El 4to multiplo es 326
```

```
// En el programa principal, vamos a obtener el 4º múltiplo de 2 entre 320 y 335
// pero ObtieneMultiplosDeN nos devuelve ya toda la secuencia de múltiplos cargada en memoria.
public static void Main()
{
    int cuartoMultObt = ObtieneMultiplosDeN(2, 320, 335).Skip(3).First();
    Console.WriteLine($"El 4to multiplo es {cuartoMultObt}");
}
```

Al generar la secuencia con una colección intermedia, hemos tenido que generar todos los múltiplos y almacenarlos en memoria haciendo un único return al final del método.

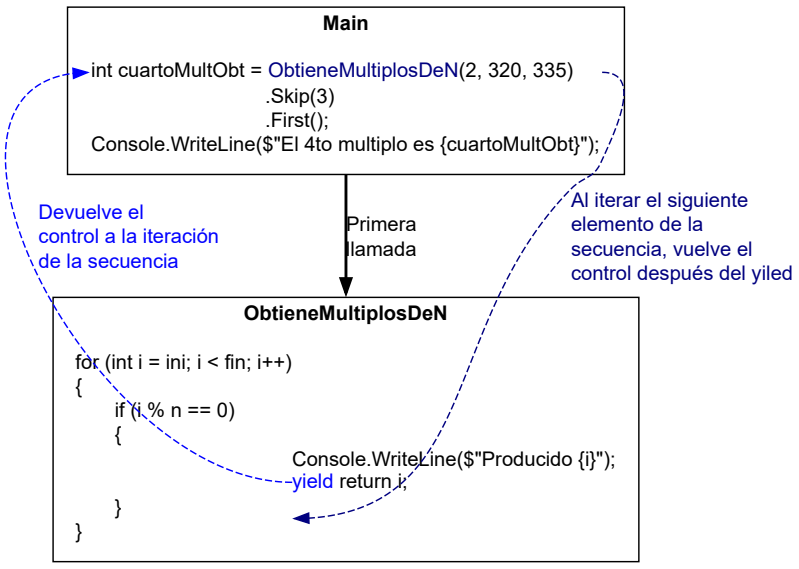
Vamos ahota a **reimplementar el código** de nuestro método y ahora **SÍ usaremos** `yield` ...

```
    static IEnumerable<int> ObtieneMultiplosDeN(int n, int ini, int fin)
    {
        for (int i = ini; i < fin; i++)
        {
            if (i % n == 0)
            {
                Console.WriteLine($"Producido {i}");
8               yield return i;
            }
        }
    }

    static void Main()
    {
15      int cuartoMultObt = ObtieneMultiplosDeN(2, 320, 335).Skip(3).First();
        Console.WriteLine($"El 4to multiplo es {cuartoMultObt}");
    }
```

> **Importante**
>
> Fíjate que al hacer un `yield return dato;` C# lo interpretará como que el método retorna una secuencia `IEnumerable<TipoDato>`, esto es, la abstracción de una secuencia iterable.

El flujo de ejecución frente al anterior sería....

1. `Línea 15`: Llamamos al métodos y la pasamos el control de ejecución.
2. `Líneas 8`: Obtenemos el siguiente valor de la secuencia y retornamos el control a la `línea 15` en ella si procesará el elemento de la secuencio y si necesitamos otro cuando llamemos al `it.MoveNext()` del iterador (`IEnumerator`) proporcionado por la secuencia devuelta, se volverá a pasar el control de ejecución a la `línea 9` para que el algoritmo me vuelva a generar otro elemento de la secuencia de llegar al `yield return i;` y de no ser así porque ha acabado el `for` entonces el `it.MoveNext()` devolverá `false`.



Si ejecutamos el programa y comprobamos el log de salida, veremos que **hemos generado solo los múltiplos hasta el 4** en el rango dado, en lugar de todos como antes. Esto será así independientemente del rango que le pasemos al método para obtener los múltiplos.

```
Producido 320
Producido 322
Producido 324
Producido 326
El 4to multiplo es 326
```

Bien ya hemos visto cómo funciona `yield` y hemos conseguido que un método devuelva una secuencia perezosa en forma de `IEnumerable<T>` sin necesidad de definir las clases que implementan los interfaces del patrón iterador. Pero, **¿Cómo podríamos implementar `IEnumerable<T>` en una clase propia usando `yield`?**

Veamoslo a través de una clase que no es una colección propiamente dicha pero que contiene una serie de datos que nos interesa recorrer de forma secuencial. Por ejemplo, una clase que represente una dirección `IPv4Address` cómo por ejemplo `192.168.1.1` y que contenga los **4 bytes que la componen**.

Definimos las clase como un `record class` ('*value object*') con 4 propiedades de tipo `byte` que representan cada uno de los octetos de la dirección IP. Además, implementamos la interfaz `IEnumerable<byte>` para poder iterar secuencialmente los bytes que componen la dirección IP.

> **Importante**
>
> En este caso los diferentes `yield return` de los bytes los implementamos en el método `GetEnumerator()` y en lugar de generarnos un `IEnumerable<byte>`, nos va a devolver un iterador `IEnumerator<byte>` que es el que nos permitirá recorrer la secuencia de bytes de forma perezosa.

Puedes descargar el código completo de este ejemplo desde **este enlace**.

```
public record class IPv4Address(byte Byte1, byte Byte2, byte Byte3, byte Byte4) : IEnumerable<byte>
{
    public IEnumerator<byte> GetEnumerator()
    {
        yield return Byte1;
        yield return Byte2;
        yield return Byte3;
        yield return Byte4;
    }

    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator() => GetEnumerator();
```

```
        // Constrctor secundario para crear la IP a partir de una cadena.
        public IPv4Address(string ip)
            : this(
                byte.Parse(ip.Split('.')[0]), byte.Parse(ip.Split('.')[1]),
                byte.Parse(ip.Split('.')[2]), byte.Parse(ip.Split('.')[3]))
        { }

        public override string ToString() => $"{Byte1}.{Byte2}.{Byte3}.{Byte4}";

        public string ToBinaryString()
        {
            StringBuilder ipBinario = new();
            // Podemos recorrer los bytes de la IP (this) gracias a que implementa IEnumerable<byte>
            foreach (byte b in this)
                ipBinario.Append($"{b:b8}.");
            return ipBinario.ToString().TrimEnd('.');
        }
    }
```

Si ejecutamos el siguiente programa principal de test. Debería **mostrar por consola**:

```
    IP1: 192.168.1.1     11000000.10101000.00000001.00000001
Mascara: 255.255.255.0   11111111.11111111.11111111.00000000
```

```
class Program
{
    static void Main()
    {
        IPv4Address ip = new(192, 168, 1, 1);
        IPv4Address mascara = new("255.255.255.0");

        Console.WriteLine($"    IP1: {ip,-15} {ip.ToBinaryString()}");
        Console.WriteLine($"Mascara: {mascara,-15} {mascara.ToBinaryString()}");
    }
}
```

🚀 **Ampliación opcional:**

¿Serías capaz de interpretar los conceptos del **uso del patrón iterador** y la **generación perezosa de secuencias** con `yield` en otros lenguajes de programación como **JavaScript**, **Kotlin** o **Python** a través de la conversión del ejemplo sencillo propuesto en C#?

Ejemplo C#:

```
static IEnumerable<int> Datos()
{
    yield return 23;
    yield return 56;
    yield return 66;
    yield return 94;
}

var it = Datos().GetEnumerator();
while (it.MoveNext())
{
    Console.WriteLine(it.Current);
}

// Recorriendo la secuencia con foreach
foreach (int valor in Datos())
{
    Console.WriteLine(valor);
}
```

Equivalente en JavaScript:

```
function* datos() {
    yield 23;
    yield 56;
    yield 66;
    yield 94;
}

const it = datos();
while (!(current = it.next()).done) {
    console.log(current.value);
}

// Recorriendo la secuencia con for...of
for (const valor of datos()) {
    console.log(valor);
}
```

Equivalente en Kotlin:

```
fun datos(): Sequence<Int> = sequence {
    yield(23)
    yield(56)
    yield(66)
    yield(94)
}

fun main() {
    val it = datos().iterator()
    while (it.hasNext()) {
        val current = it.next()
        println(current)
    }

    // Recorriendo la secuencia con for
    for (valor in datos()) {
        println(valor)
    }
}
```

Equivalente en Python:

```
def datos():
    yield 23
    yield 56
    yield 66
    yield 94

it = datos()
while True:
    try:
        current = next(it)
        print(current)
    except StopIteration:
        break

# Recorriendo la secuencia con for
for valor in datos():
    print(valor)
```