.Net
Teoría 9

Genéricos

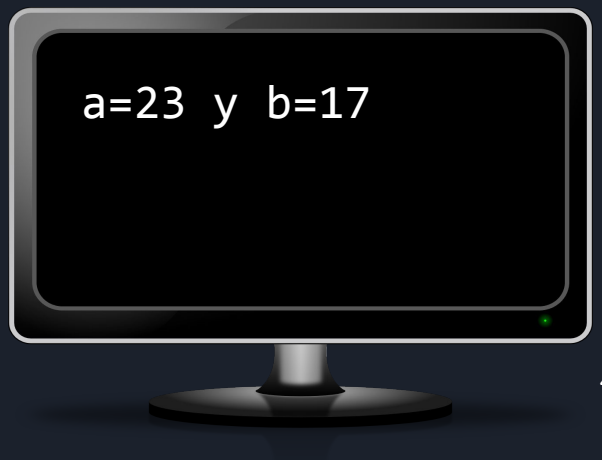
Vamos a presentar la utilidad de los genéricos a partir de un ejemplo

1. Abrir una terminal del sistema operativo
2. Cambiar a la carpeta proyectosDotnet
3. Crear la aplicación de consola Teoria9
4. Abrir Visual Studio Code sobre este proyecto

Codificar el método Swap faltante en el siguiente código:

----------Program.cs----------

```
int a = 17;
int b = 23;
Swap(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
```



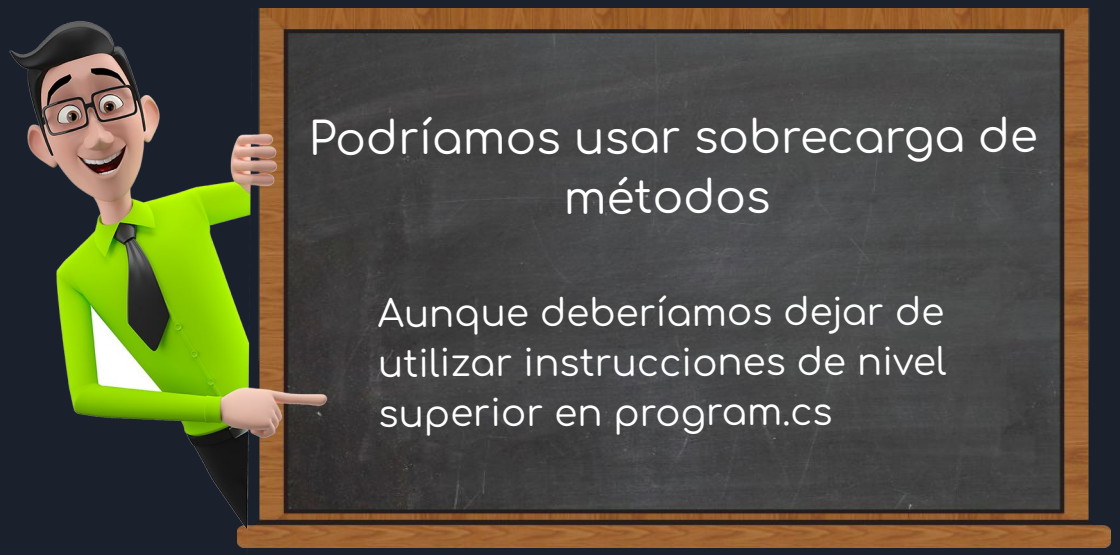
Código en el archivo
09 Teoria-Recursos.txt

Posible solución

```
----------Program.cs----------

int a = 17;
int b = 23;
Swap(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");

void Swap(ref int i, ref int j)
{
    int auxi = i;
    i = j;
    j = auxi;
}
```

Planteo de una situación

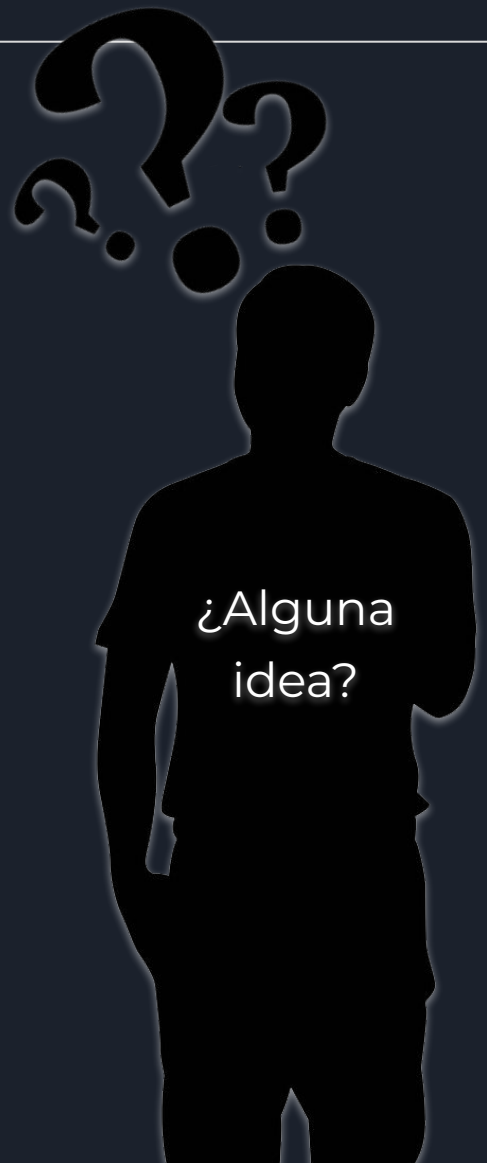
¿Qué podríamos hacer si ahora queremos intercambiar dos variables de tipo string?



Planteo de una situación

¿Y si luego necesitamos intercambiar dos variables de tipo char, double, byte, ArrayList, Stringbuilder, Auto, Persona, etc?

En lugar de codificar tantas sobrecargas del método Swap sería deseable poder codificar un único Swap que nos sirva para todos los casos.



Probar si alcanza con esta sencilla modificación. (Eliminar las otras versiones de Swap)

```
int a = 17;
int b = 23;
Swap(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola";
string st2 = "mundo";
Swap(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");

void Swap( ref object i, ref object j )
{
   object auxi = i;
   i = j;
   j = auxi;
}
```

Error de compilación

No funciona. Error de compilación

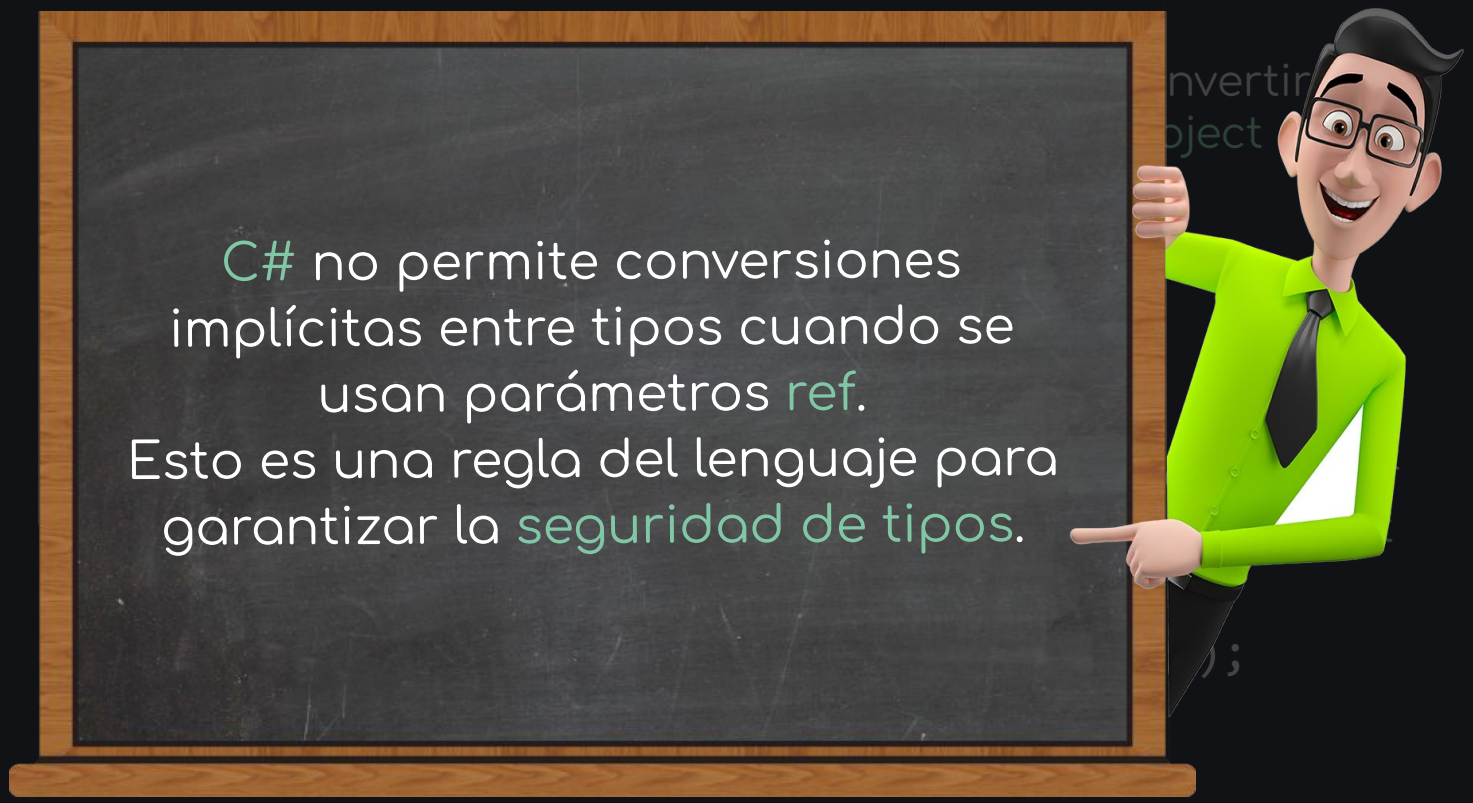
```
int a = 17;
int b = 23;
Swap(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola";
string st2 = "mundo";
Swap(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");
```

No se puede convertir ref int a ref object

No se puede convertir ref string a ref object

Error de compilación

No funciona. Error de compilación



Error de compilación

No funciona. Error de compilación

¿Qué ocurriría con las variables a y b si el compilador permitiese este código?

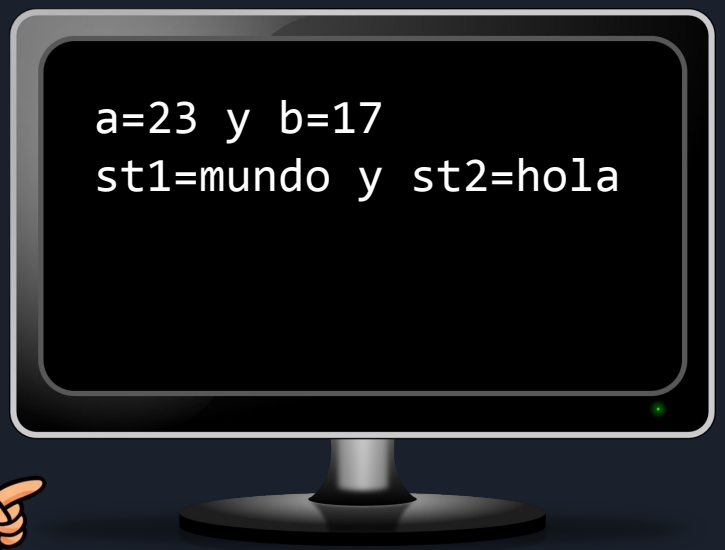
```
int a = 17;
int b = 23;
Swap(ref a, ref b);
//a y b son int, qué valor tendrían aquí?

void Swap(ref object i, ref object j)
{
    i = new Persona();
    j = new List<string> { "uno", "dos" };
}
```

Podríamos usar variables de tipo object para pasar los parámetros

```
object o1, o2;
int a = 17; int b = 23;
o1 = a; o2 = b;
Swap(ref o1, ref o2);
a = (int)o1; b = (int)o2;
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola"; string st2 = "mundo";
o1 = st1; o2 = st2;
Swap(ref o1, ref o2);
st1 = (string)o1; st2 = (string)o2;
Console.WriteLine($"st1={st1} y st2={st2}");
```

```
object o1, o2;
int a = 17; int b = 23;
o1 = a; o2 = b;
Swap(ref o1, ref o2);
a = (int)o1; b = (int)o2;
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola"; string st2 = "mundo";
o1 = st1; o2 = st2;
Swap(ref o1, ref o2);
st1 = (string)o1; st2 = (string)o2;
Console.WriteLine($"st1={st1} y st2={st2}");

void Swap( ref object i, ref object j )
{
   object auxi = i;
   i = j;
   j = auxi;
}
```

¡Funciona!



Genéricos - Introducción

```
object o1, o2;
int a = 17; int b = 23;
o1 = a; o2 = b;
Swap(ref o1, ref o2);
a = (int)o1; b = (int)o2;
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola"; string st2 = "mundo";
o1 = st1; o2 = st2;
Swap(ref o1, ref o2);
st1 = (string)o1; st2 = (string)o2;
Console.WriteLine($"st1={st1} y st2={st2}");

void Swap( ref object i, ref object j )
{
   object auxi = i;
   i = j;
   j = auxi;
}
```

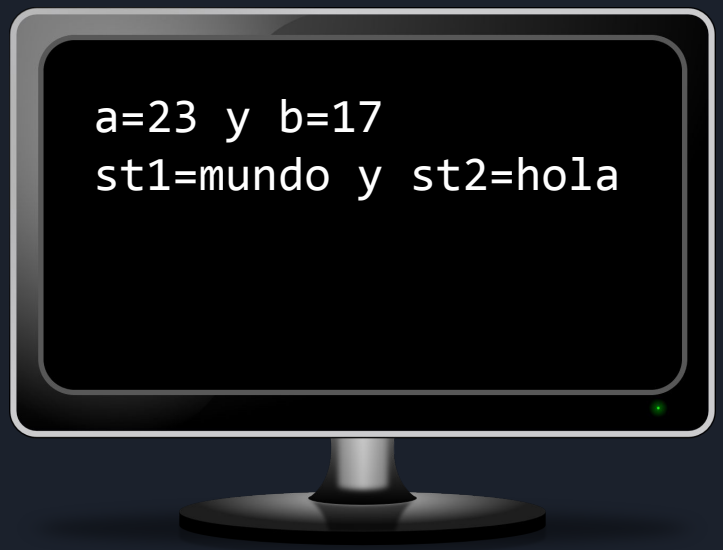
Planteo de otra solución

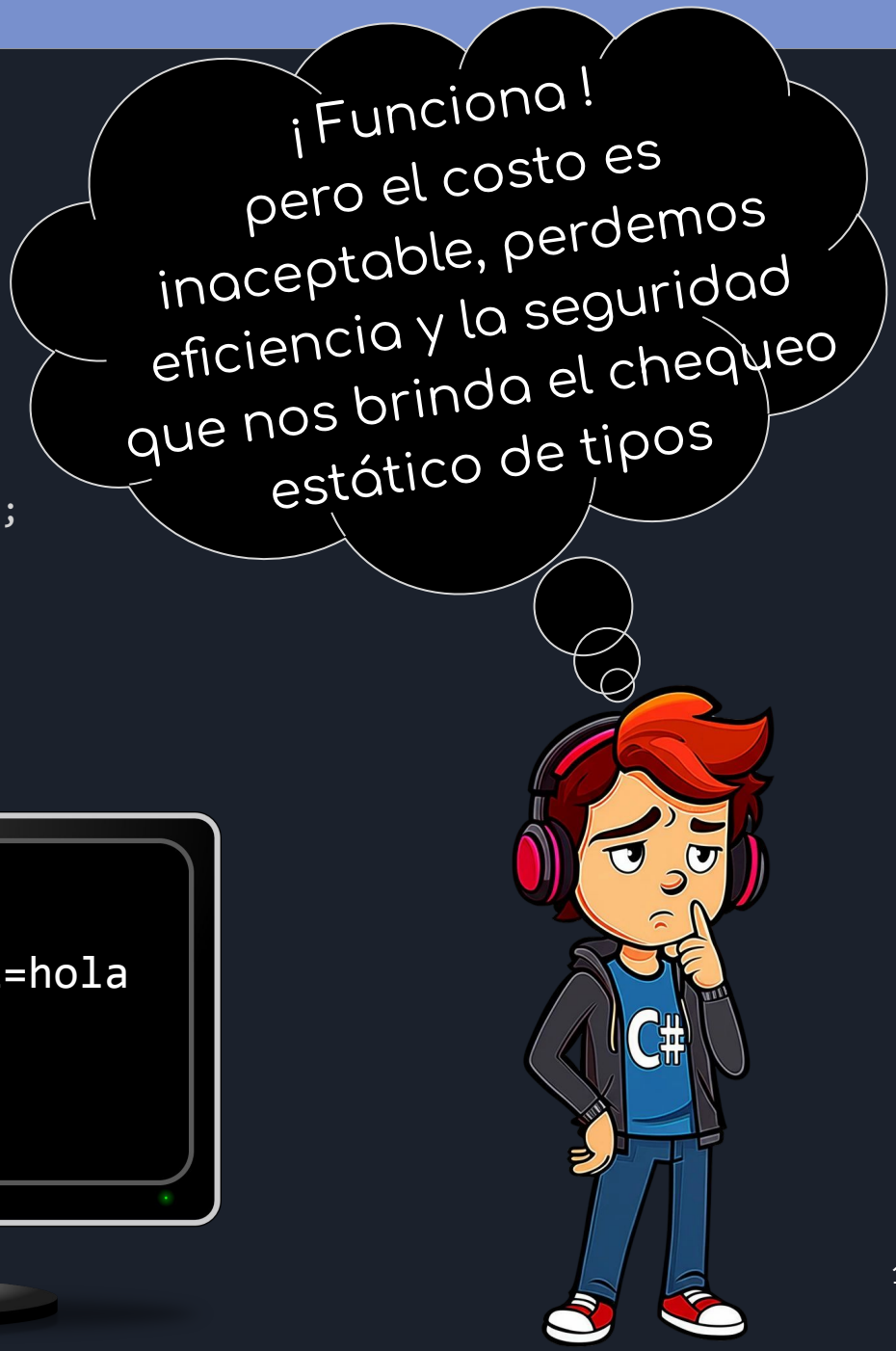
```
dynamic a = 17;
dynamic b = 23;
Swap(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
dynamic st1 = "hola";
dynamic st2 = "mundo";
Swap(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");

void Swap(ref dynamic i, ref dynamic j) {
    dynamic auxi = i;
    i = j;
    j = auxi;
}
```

Usar tipos dynamic

Genéricos - Introducción

```
dynamic a = 17;
dynamic b = 23;
Swap(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
dynamic st1 = "hola";
dynamic st2 = "mundo";
Swap(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");

void Swap(ref dynamic i, ref dynamic j)
{
    dynamic auxi = i;
    i = j;
    j = auxi;
}
```





Métodos Genéricos

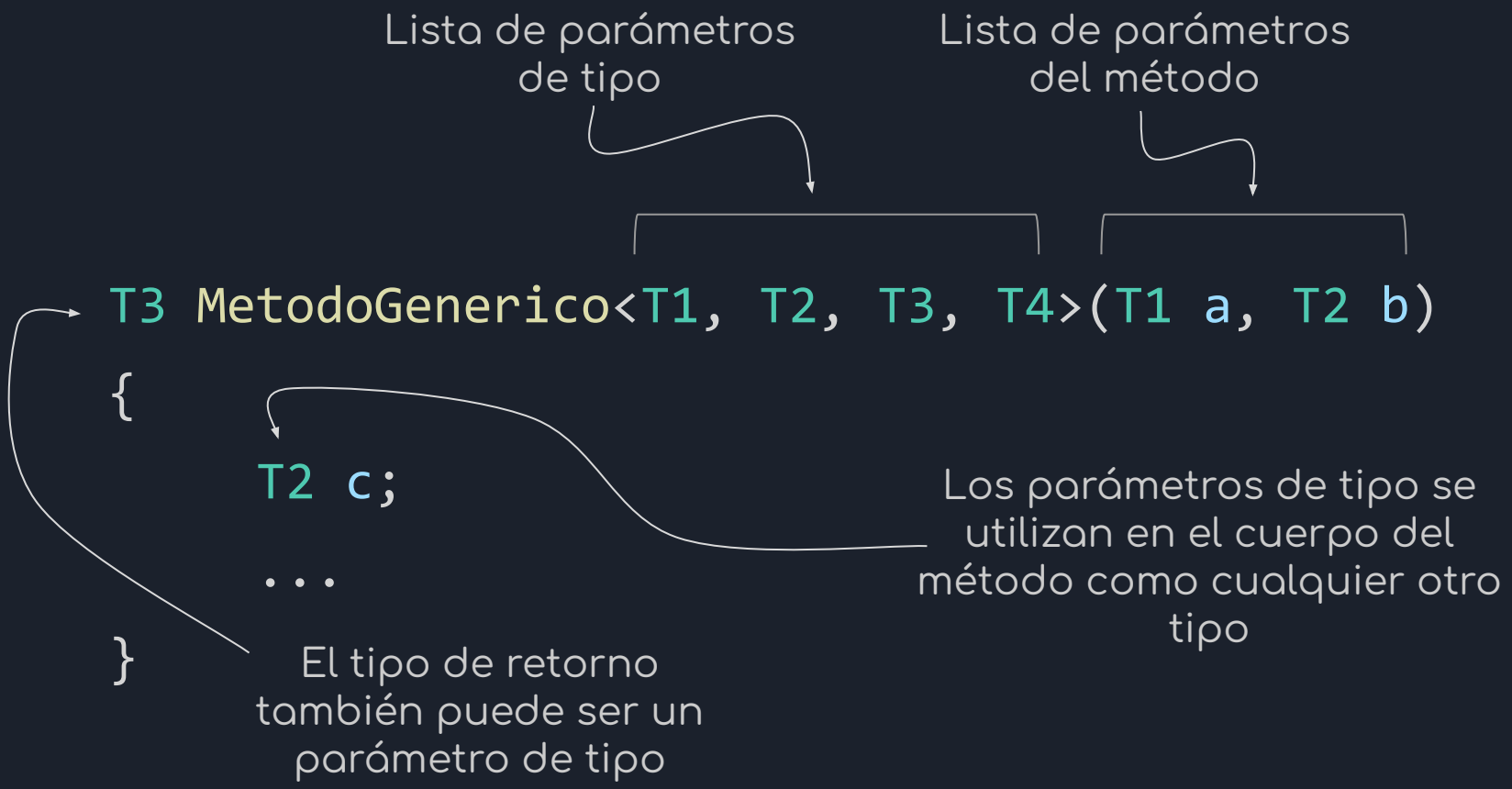
Métodos Genéricos

- Afortunadamente existe una mejor solución, que permite definir métodos genéricos sin perder las ventajas de la verificación estática de tipos (eficiencia y seguridad de tipos)

- Los métodos genéricos permiten pasar los tipos como parámetros

Métodos Genéricos

Declaración de un método genérico

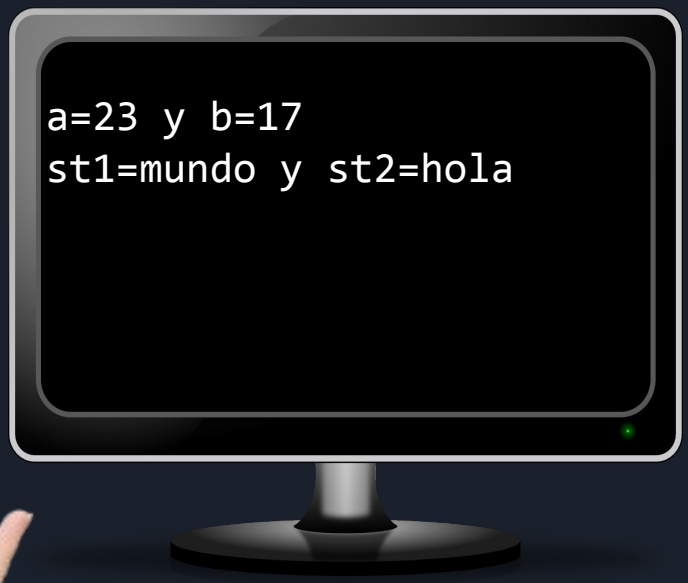


Solución al ejercicio planteado con método Swap genérico

```
int a = 17;
int b = 23;
Swap<int>(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola";
string st2 = "mundo";
Swap<string>(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");

void Swap<T>(ref T i, ref T j)
{
    T auxi = i;
    i = j;
    j = auxi;
}
```

```
int a = 17;
int b = 23;
Swap<int>(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola";
string st2 = "mundo";
Swap<string>(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");

Swap<T>(ref T i, ref T j)
{
     T auxi = i;
     i = j;
     j = auxi;
}
```

¡Funciona!
Es eficiente y provee seguridad de tipos



Parámetros de tipos inferidos en los métodos genéricos

Si se pasan parámetros en la invocación a un método genérico, el compilador a veces puede inferir a partir de ellos los parámetros de tipo.

Ejemplo:

```
void Swap<T>(ref T i, ref T j)
```

A partir del tipo de `i` y `j` pasados en la invocación, puede inferirse `T`

Por lo tanto, el parámetro de tipo puede omitirse en la invocación

Probar eliminando los argumentos de tipo en la invocación a Swap

```
int a = 17;
int b = 23;
Swap(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola";
string st2 = "mundo";
Swap(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");
```

Se invoca Swap<int>

Se invoca Swap<string>

Otro planteo de situación

Se desea codificar el método Mayor que devuelva el mayor de dos instancias de un tipo que admite la comparación de sus elementos, pero evitando la sobrecarga

```
int Mayor(int a, int b) { ... }
long Mayor(long a, long b) { ... }
string Mayor(string a, string b) { ... }
double Mayor(double a, double b) { ... }
char Mayor(char a, char b) { ... }
...
```

Codificar Program.cs de la siguiente manera (resolviendo con interfaces)

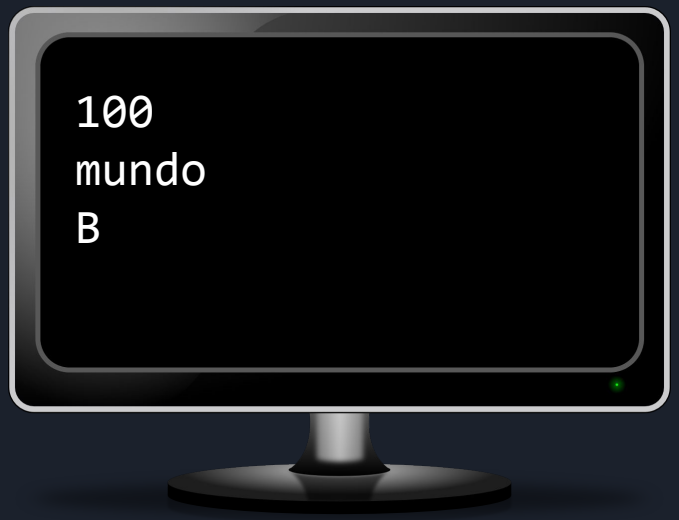
```
int i = Mayor(100, 55);
Console.WriteLine(i);
string st = Mayor("hola", "mundo");
Console.WriteLine(st);
Console.WriteLine(Mayor('A','B'));

IComparable Mayor(IComparable a, IComparable b)
{
    if (a.CompareTo(b) > 0)
    {
        return a;
    }
    return b;
}
```

Y solucionar errores de compilación

Código en el archivo
09 Teoria-Recursos.txt

```
int i = (int)Mayor(100, 55);
Console.WriteLine(i);
string st = (string)Mayor("hola", "mundo");
Console.WriteLine(st);
Console.WriteLine(Mayor('A','B'));

IComparable Mayor(IComparable a, IComparable b)
{
    if (a.CompareTo(b) > 0)
    {
        return a;
    }
    return b;
}
```

es necesario hacer casting



Consideraciones

La solución presentada tiene detalles que podrían mejorarse.

Por ejemplo son necesarias conversiones de tipo, algunas incluso provocan boxing y unboxing lo que supone pérdida de rendimiento.

```
int i = (int)Mayor(100, 55);
```

Consideraciones

Al usar IComparable en lugar de los tipos más específicos se pierde cierta seguridad de tipo. Por ejemplo el compilador no detecta ningún problema con:

```
Console.WriteLine(Mayor("hola",55));
```

Sin embargo hay error en tiempo de ejecución porque el método CompareTo espera trabajar con dos objetos del mismo tipo

Probar esta solución con métodos genéricos

```
int i = Mayor<int>(100, 55);
Console.WriteLine(i);
string st = Mayor<string>("hola", "mundo");
Console.WriteLine(st);
Console.WriteLine(Mayor<char>('A', 'B'));

T Mayor<T>(T a, T b)
{
   if (a.CompareTo(b) > 0)
   {
       return a;
   }
   return b;
}
```

Código en el archivo
09 Teoria-Recursos.txt

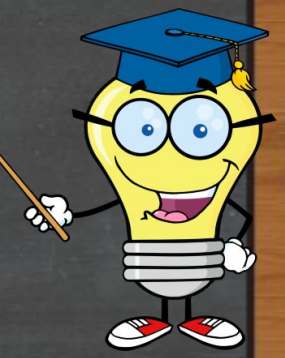
Probar esta solución con métodos genéricos

```
int i = Mayor<int>(100, 55);
Console.WriteLine(i);
string st = Mayor<string>("hola", "mundo");
Console.WriteLine(st);
Console.WriteLine(Mayor<char>('A', 'B'));

T Mayor<T>(T a, T b)
{
    if (a.CompareTo(b) > 0)
    {
        return a;
    }
    return b;
}
```

El compilador no puede asegurar que el tipo T tiene definido el método CompareTo

Solución con métodos genéricos

```
T Mayor<T>(T a, T b) where T : IComparable
{
    if (a.CompareTo(b) > 0)
    {
        return a;
    }
    return b;
}
```



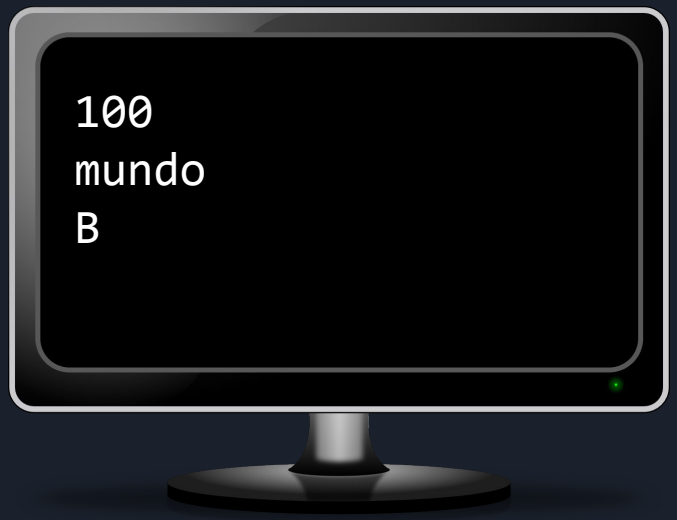
Se resuelve imponiendo una restricción sobre el parámetro de tipo T que debe implementar la interfaz IComparable

Codificar el método genérico de la siguiente manera:

```
T Mayor<T>(T a, T b) where T : IComparable
{
    if (a.CompareTo(b) > 0)
    {
        return a;
    }
    return b;
}
```

```
int i = Mayor<int>(100, 55);
Console.WriteLine(i);
string st = Mayor<string>("hola", "mundo");
Console.WriteLine(st);
Console.WriteLine(Mayor<char>('A', 'B'));

T Mayor<T>(T a, T b) where T : IComparable
{
   if (a.CompareTo(b) > 0)
   {
      return a;
   }
   return b;
}
```

Para este caso, el parámetro de tipo puede omitirse en las invocaciones porque el compilador puede inferirlo



Solución con tipos genéricos

La solución que usa un método genérico es más eficiente:

- No hay conversiones de tipo innecesarias, ni boxing ni unboxing.

La seguridad de tipos es más fuerte,

- un intento como `Mayor("hola",55)` sería detectado en tiempo de compilación

Restricciones sobre los parámetros de tipo - Cláusulas where

Las restricciones se enumeran como cláusulas where.

- Cada parámetro de tipo que tiene restricciones tiene su propia cláusula where.
- Si un parámetro tiene múltiples restricciones, se enumeran en la cláusula where, separadas por comas.

Restricciones sobre los parámetros de tipo - Cláusulas where

```
where ParamTipo : restriccion, restriccion, . . .
```

Palabra clave

Parámetro de tipo

Lista de restricciones

- Si hay más de una cláusula where no se separan por comas ni ningún otro token.
- Se pueden enumerar en cualquier orden.

Restricciones sobre los parámetros de tipo - Cláusulas where

Ejemplo:

```
T3 MetodoGenerico<T1, T2, T3, T4, T5>(T2 a)
   where T3 : IEnumerable, new()
   where T5 : class
   where T1 : Persona
   where T4 : struct
   where T2: notnull

{
   . . .
}
```

T3 debe implementar la interfaz IEnumerable y poseer un constructor público sin parámetros

T5 debe ser cualquier tipo referencia

T1 debe ser de clase `Persona` o derivada de `Persona`

T2 debe ser un tipo que no admite null

T4 debe ser cualquier tipo valor

Tipos Genéricos

Tipos genéricos

Además de los métodos ya vistos, C# provee cuatro categorías más de genéricos, todos ellos son tipos:

1. Clases
2. Estructuras
3. Interfaces
4. Delegados

Un tipo genérico no es un tipo real, sino una plantilla para un tipo real que se construye cuando se proporcionan los argumentos para los parámetros de tipo definidos

Clases Genéricas

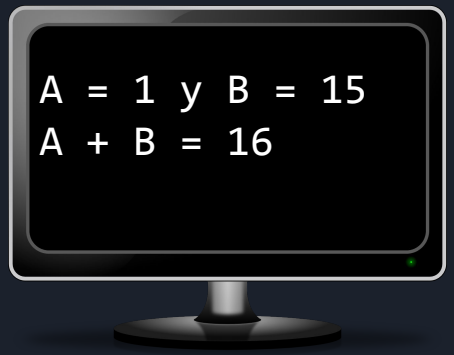
Ejemplo de una clase simple

Se desea codificar la clase Par para almacenar dos valores enteros en sus propiedades A y B que se asignan en el constructor. Fuera de la clase las propiedades son de sólo lectura.

```
class Par
{
    public int A {get; private set; }
    public int B {get; private set; }
    public Par(int a, int b)
    {
        this.A = a;
        this.B = b;
    }
}
```

Ejemplo de una clase simple

La clase Par puede utilizarse de la siguiente manera:




```
Par par = new Par(1, 15);
Console.WriteLine($"A = {par.A} y B = {par.B}");
Console.WriteLine($"A + B = {par.A + par.B} ");
```

Clases Genéricas

¿Qué deberíamos hacer si ahora se necesita que las propiedades A y B de la clase Par sean de tipo double?

¿Y si se necesita que A sea de tipo string y B de tipo float?

Afortunadamente las clases también admiten parámetros de tipo, estas clases se denominan clases genéricas

Clases Genéricas

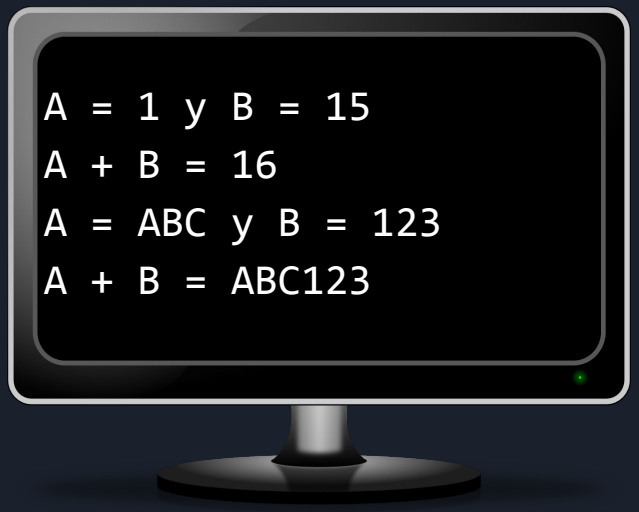
Transformamos la clase Par en una clase genérica

```
class Par<T1, T2>
{
    public T1 A { get; private set; }
    public T2 B { get; private set; }
    public Par(T1 a, T2 b)
    {
        this.A = a;
        this.B = b;
    }
}
```

Agregamos parámetros de tipo en la definición de la clase Par

Observar que el constructor lleva el nombre de la clase sin la lista de parámetros de tipo, es decir `Par(...)` no `Par<T1,T2>(...)`

```
Par<int, int> par = new Par<int, int>(1, 15);
Console.WriteLine($"A = {par.A} y B = {par.B}");
Console.WriteLine($"A + B = {par.A + par.B} ");
Par<string, double> par2 = new Par<string, double>("ABC", 123);
Console.WriteLine($"A = {par2.A} y B = {par2.B}");
Console.WriteLine($"A + B = {par2.A + par2.B} ");
```



Del tipo genérico al construído

Al indicar cuáles son los tipos reales (argumentos de tipo) que deben sustituir a los parámetros de tipo, el compilador JIT toma esos argumentos y crea el tipo real (que se llama tipo construído) del cual se podrán instanciar objetos.

Del tipo genérico al construído

```
Par<string, double>
```

```
class Par<T1, T2>
{
    public T1 A
    { get; private set; }
    public T2 B
    { get; private set; }
    public Par(T1 a, T2 b)
    {
        this.A = a;
        this.B = b;
    }
}
```

Clase genérica

produce

```
class Par<string, double>
{
    public string A
    { get; private set; }
    public double B
    { get; private set; }
    public Par(string a,double b)
    {
        this.A = a;
        this.B = b;
    }
}
```

Clase construída

Cláusulas where en clases genéricas

Valen los mismos considerando expuestos para el caso de métodos genéricos.

Las cláusulas where se colocan antes del cuerpo de la clase, ejemplo:

```
class ClaseGenerica<T1, T2, T3, T4>
                where T2 : IEnumerable
                where T3 : class
                where T4 : struct
{
    . . .
}
```

Colecciones genéricas

El espacio de nombres System.Collections.Generic define varias colecciones genéricas:

- List<T>: una de las más utilizadas, es la versión genérica de ArrayList
- Dictionary<TKey,TValue>: es una versión genérica de Hashtable
- SortedDictionary<TKey,TValue>: Idem al anterior pero ordenado según la clave
- Queue<T>:versión genérica de Queue
- Stack<T>: versión genérica de Stack
- SortedSet<T>: colección de elementos ordenados y sin duplicación
- HashSet<T>: conjunto de elementos sin duplicados sin orden en particular

Cantidad de código creado por los tipos genéricos

- La creación de instancias de clases genéricas con tipos específicos no duplica estas clases en el código IL.
- Sin embargo, cuando el compilador JIT compila las clases genéricas a código nativo, se crea una nueva clase para cada tipo de valor específico. Los tipos de referencia comparten la misma implementación de la misma clase nativa.

Interfaces Genéricas

Interfaces genéricas

Las interfaces genéricas permiten usar parámetros de tipo genérico en la declaración de sus miembros.

Ejemplo de una interfaz genérica sencilla:

```
interface IDuplicador<T>
{
    T Duplicar(T valor);
}
```

Interfaces genéricas

Al establecer diferentes argumentos de tipo en una interfaz genérica se construyen distintas interfaces

Ejemplo:

```
class Duplica : IDuplicador<int>, IDuplicador<string> {...}
```

2 interfaces construidas

```
interface IDuplicador<int>
{
    int Duplicar(int valor);
}
```

```
interface IDuplicador<string>
{
    string Duplicar(string valor);
}
```

Interfaces genéricas

```
Duplica dup = new Duplica();
int i = dup.Duplicar(3);
string st = dup.Duplicar("hola");
Console.Write($"{st} {i}");
```


```
class Duplica : IDuplicador<int>, IDuplicador<string> {
    public int Duplicar(int valor) {
        return valor * 2;
    }
    public string Duplicar(string valor) {
        return valor + valor;
    }
}
```

Observar que la clase Duplica no es una clase genérica

Interfaces genéricas con clases genéricas

Un parámetro de tipo de una clase genérica, puede usarse también como tipo de una interfaz genérica implementada por esa clase. Ejemplo

```
class Generica<T1, T2> : IDuplicador<T2>
{
    public T2 Duplicar(T2 valor)
    {
        return valor;
    }
    . . .
}
```

Cuando se construya la clase **Generica** con tipos reales, por ejemplo **Generica<char, int>** se construye también la interfaz **IDuplicador<int>**

Interfaces genéricas con clases genéricas

```
Generica<char, int>
```

```
interface IDuplicador<int>
{
    int Duplicar(int valor);
}
```

Interfaz construida

```
class Generica<char, int> : IDuplicador<int>
{
    public int Duplicar(int valor)
    {
        return valor;
    }
    . . .
}
```

Clase construida

Atención! esto no está permitido

```
class ClaseGen<T1, T2> : IDuplicador<T1>, IDuplicador<int>
{
    . . .
}
```

No se puede implementar una interfaz con parámetro de tipo genérico y la misma interfaz construida con un tipo real.
Observar que clase construída `ClaseGen<int,char>` tendría 2 interfaces `IDuplicador<int>` lo cual no se permite.

Atención! esto no está permitido

```
class ClaseGen<T1, T2> : IDuplicador<T1>, IDuplicador<T2>
{
    . . .
}
```

Por la misma razón, esto tampoco es válido.
Observar que clase construída `ClaseGen<int,int>` tendría 2 interfaces `IDuplicador<int>` lo cual no se permite.

Interfaces genéricas en la BCL

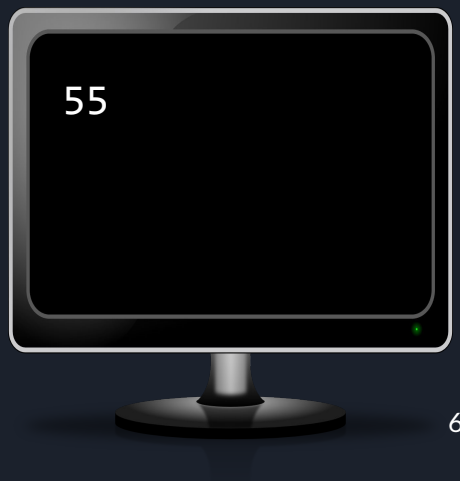
- .NET ofrece muchas interfaces genéricas para diferentes escenarios. Entre muchas otras están IComparer<T>, IComparable<T>, IEnumerator<T> e IEnumerable<T>
- A menudo existen versiones anteriores no genéricas de la misma interfaz basadas en el tipo object. Las respectivas versiones genéricas son preferibles pues evitan conversiones de tipos como boxing y unboxing

Los iteradores también funcionan con IEnumerator<T> e IEnumerable<T>

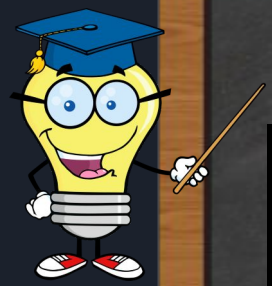
Ejemplo:

No es necesario hacer casting porque r.Current es de tipo int

```
int suma = 0;
var r = Rango(1, 10);
while (r.MoveNext()) {
    suma += r.Current;
}
Console.WriteLine(suma);


IEnumerator<int> Rango(int a, int b) {
    for (int i = a; i <= b; i++) {
        yield return i;
    }
}
```



Extensión de métodos en interfaces genéricas



¿Se acuerdan de la extensión de métodos?

```
public static class ExtensionMetodos
{
    public static int Factorial(this int n)
    {
        return (n == 0) ? 1 : n * Factorial(n - 1);
    }
}
```

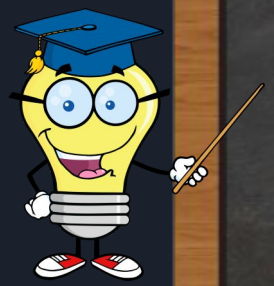
Luego podemos usar

```
int fac5 = 5.Factorial();
```

Extensión de métodos en interfaces genéricas

Extender métodos para una interfaz genérica resulta de mucha utilidad ya que los métodos de extensión podrán invocarse en todos los tipos que implementan dicha interfaz

Extensión de métodos en interfaces genéricas



Ejemplo

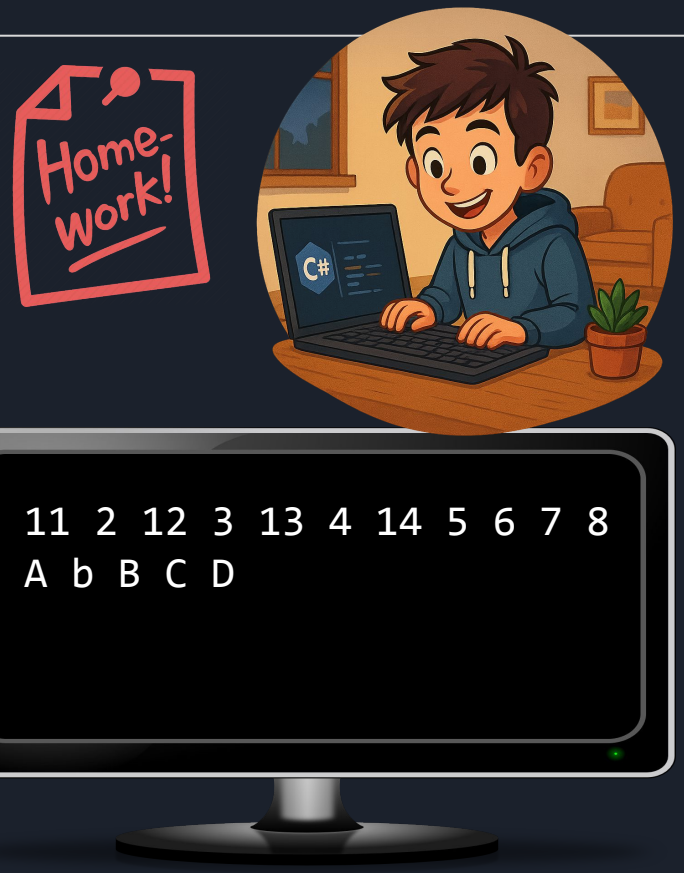
```
public static string EnUnaLinea<T>(this IEnumerable<T> secuencia)
{
    string resultado = "";
    foreach (T t in secuencia) resultado += $"{t} ";
    return resultado;
}
```

Luego podemos usar

```
int[] vector = [1, 2, 3, 4, 5];
List<string> lista = ["uno", "dos", "tres"];
string st1 = vector.EnUnaLinea(); // st1 <== "1 2 3 4 5"
string st2 = lista.EnUnaLinea();  // st2 <== "uno dos tres"
```

Codificar el método de extensión Intercalar que haga posible la ejecución de este código

```
int[] vector = [1, 2, 3, 4, 5, 6, 7, 8];
List<int> lista = [11, 12, 13, 14];
var secuencia1 = vector.Intercalar(lista);
foreach (int i in secuencia1)
{
   Console.Write(i + " ");
}

Console.WriteLine();

char[] vector2 = ['A', 'B', 'C', 'D'];
List<char> lista2 = ['a', 'b'];
var secuencia2 = lista2.Intercalar(vector2);
foreach (char c in secuencia2)
{
   Console.Write(c + " ");
}
```



```
public static class ExtensionMetodos
{
    public static IEnumerable<T> Intercalar<T>(this IEnumerable<T> scuencia1,
                                               IEnumerable<T> scuencia2)
    {
        IEnumerator<T> enum1 = scuencia1.GetEnumerator();
        IEnumerator<T> enum2 = scuencia2.GetEnumerator();
        bool continuar = true;
        while (continuar)
        {
            continuar = false;
            if (enum1.MoveNext())
            {
                yield return enum1.Current;
                continuar = true;
            }
            if (enum2.MoveNext())
            {
                yield return enum2.Current;
                continuar = true;
            }
        }
    }
}
```

Posible solución

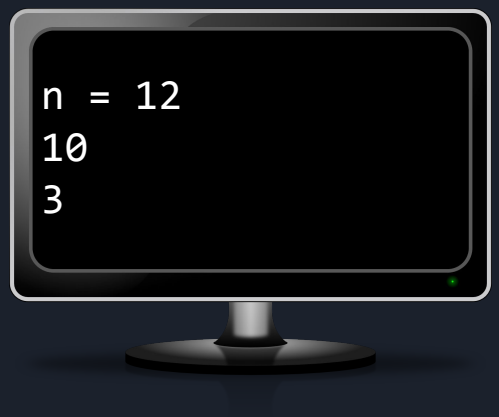
Delegados Genéricos

Delegados genéricos

Los delegados genéricos se declaran como los delegados no genéricos pero utilizando parámetros de tipo en lugar de los tipos reales.

Delegados genéricos - Ejemplo

```
----------Funcion.cs----------
delegate T2 Funcion<T1, T2>(T1 valor);

----------Program.cs----------
Funcion<int, string> lambda1 = n => $"n = {n}";
Funcion<string, int> lambda2 = st => st.Length;
Funcion<int[],double> f = Promedio;
Console.WriteLine(lambda1(12));
Console.WriteLine(lambda2("hola mundo"));
Console.WriteLine(f([1,2,3,4,5]));

double Promedio(int[] vector) {
      int sum=0;
      foreach(int i in vector) sum+=i;
      return sum/vector.Length;
}
```



Delegados genéricos en la BCL
Action<T>

.NET ofrece el delegado genérico Action<T> para métodos con tipo de retorno void

```
Action<T1> métodos con 1 parámetro;

Action<T1,T2> métodos con 2 parámetros;

. . .

Action<T1,...,T16> métodos con 16 parámetros
```

Delegados genéricos en la BCL
Func<TResult>

.NET ofrece el delegado genérico Func<TResult> para métodos con tipo de retorno TResult

```
Func<TResult> métodos sin parámetros

Func<T1,TResult> métodos con 1 parámetro

Func<T1,T2,TResult> métodos con 2 parámetros

. . .

Func<T1,...,T16,TResult> métodos con 16
                         parámetros
```

Delegados Action<T> y Func<T> Ejemplos

```
Action<int, string[], double> a = HacerAlgo;
Func<int, string, List<int>> f = DevolverLista;
. . .

void HacerAlgo(int n, string[] vector, double d) {
    . . .
}

List<int> DevolverLista(int i, string st) {
   . . .
}
```

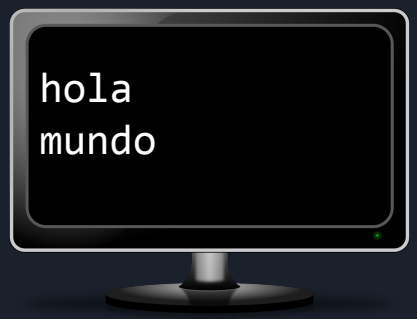
Delegados Action<T> y Func<T> Ejemplos

- La clase List<T> posee un método llamado ForEach que recibe un Action<T> como parámetro:

```
public void ForEach(Action<T> action);
```

- Este método realiza la acción especificada por cada elemento de List<T>. Ejemplo:

```
...
List<string> lista = [ "hola", "mundo" ];
lista.ForEach(st => Console.WriteLine(st));
...
```



Delegado Predicate<T>

Representa un método que determina si un objeto de tipo T cumple con determinados criterios

```
public delegate bool Predicate<T>(T obj);
```

Existen varios métodos de la clase List<T> que reciben como parámetro un objeto Predicate<T>

Delegado Predicate<T>

CURIOSIDAD

Observar que los delegados genéricos Predicate<T> y Func<T, bool> aceptan métodos con la misma firma. Sin embargo no representan el mismo tipo.

```
Func<int, bool> f1 = n => n % 2 == 0;
Predicate<int> f2 = n => n % 2 == 0;
f1 = f2; //Error de compilación
```

Algunos métodos de List<T> que utilizan el delegado Predicate<T>

- Find(Predicate<T>) Devuelve el primer elemento que coincide con las condiciones definidas por el predicado especificado, si no existe devuelve el valor predeterminado para el tipo T.

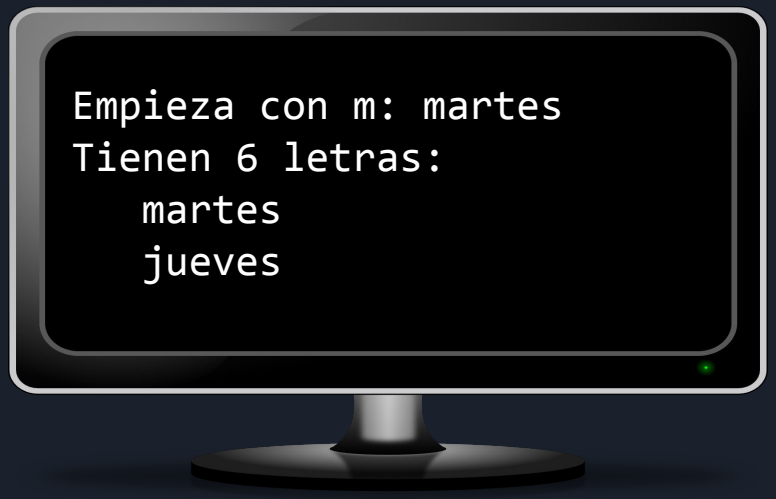
```
public T Find (Predicate<T> match);
```

- FindAll(Predicate<T>) Devuelve un objeto List<T> que contiene todos los elementos que cumplen las condiciones definidas por el predicado especificado, si se encuentran; de lo contrario, devuelve un objeto List<T> vacío

```
List<T> FindAll (Predicate<T> match);
```

Ejemplo:

```
List<string> dias = new List<string>() {
    "lunes","martes","miércoles","jueves","viernes"
 };
string? st = dias.Find(st => st[0] == 'm');
Console.WriteLine($"Empieza con m: {st}");
Console.WriteLine("Tienen 6 letras:");
List<string> lista = dias.FindAll(st => st.Length == 6);
lista.ForEach(st => Console.WriteLine("   " + st));
```



Delegado EventHandler<TEventArgs>

Para el manejo de eventos se provee el delegado genérico EventHandler<TEventArgs> que define un controlador que devuelve void y acepta dos parámetros, el primer parámetro debe ser de tipo objeto, y el segundo parámetro es de tipo TEventArgs.

```
public delegate void EventHandler<TEventArgs>
      (object sender, TEventArgs e);
```

Codificar el método de extensión Seleccionar que haga posible la ejecución de este código


```
Persona[] personas =
[
    new Persona("Juan", 20),
    new Persona("María", 25),
    new Persona("Juana", 21)
];
IEnumerable<string> nombres = personas.Seleccionar(p => p.Nombre);
//nombres <== "Juan", "María", "Juana"

IEnumerable<int> edades = personas.Seleccionar(p => p.Edad);
// edades <== 20, 25, 21

List<string> palabras = ["hola", "mundo"];
IEnumerable<int> longitudes = palabras.Seleccionar(st => st.Length);
// longitudes <== 4, 5

class Persona(string nombre, int edad)
{
    public string Nombre { get; set; } = nombre;
    public int Edad { get; set; } = edad;
}
```

```
public static class ExtensionMetodos
{
    public static IEnumerable<TResult> Seleccionar<T, TResult> (
        this IEnumerable<T> secuencia,
        Func<T, TResult> f )
    {
        foreach (var elemento in secuencia)
        {
            yield return f(elemento);
        }
    }
}
```

Posible solución

Fin teoría 9

Práctica sobre la teoría 9

Práctica sobre la teoría 9

1) Codificar el método genérico **Get** para que el siguiente código produzca la salida en la consola indicada.

```
List<object> lista = [ "hola", 7, 'A' ];
string st = Get<string>(lista, 0);
int i = Get<int>(lista, 1);
char c = Get<char>(lista, 2);
Console.WriteLine($"{st} {i} {c}");
```

Salida por consola

```
hola 7 A
```

2) Codificar los métodos que faltan para que el siguiente código produzca la salida en la consola indicada.

```
int[] vector1 = [1, 2, 3];
bool[] vector2 = [true, true, true];
string[] vector3 = ["uno", "dos", "tres"];
Set<int>(vector1, 110, 2);
Set<bool>(vector2, false, 1);
Set<string>(vector3, "Hola Mundo!", 0);
Imprimir(vector1);
Imprimir(vector2);
Imprimir(vector3);
```

Salida por consola

```
1 2 110
True False True
Hola Mundo! dos tres
```

Debe evitarse que durante la ejecución del método **Imprimir** se produzca *boxing* o *unboxing*. Tip El método **Imprimir** también es un método genérico, no se advierte fácilmente porque no se ha explicitado el parámetro de tipo (el compilador lo infiere)

Práctica sobre la teoría 9

3) Codificar los métodos genéricos **CrearArreglo** y **GetNuevoObjetoDelMismoTipo** que faltan para que el siguiente código produzca la salida en la consola indicada. El método **GetNuevoObjetoDelMismoTipo** debe crear y devolver un nuevo elemento del mismo tipo del que recibe como parámetro. Tip: Para codificar el método **CrearArreglo** tener presente el uso de **params**

```
string[] vector1 = CrearArreglo<string>("uno", "dos");
foreach (string st in vector1) Console.Write(st + " - ");
Console.WriteLine();
double[] vector2 = CrearArreglo<double>(1, 2.3, 4.1, 6.7);
foreach (double valor in vector2) Console.Write(valor + " - ");
Console.WriteLine();

var stb = new System.Text.StringBuilder();
var a = GetNuevoObjetoDelMismoTipo(stb);
var b = GetNuevoObjetoDelMismoTipo(17);
Console.WriteLine(a.GetType());
Console.WriteLine(b.GetType());
```

Salida por consola

```
uno - dos -
1 - 2,3 - 4,1 - 6,7 -
System.Text.StringBuilder
System.Int32
```

Nota: el método **GetNuevoObjetoDelMismoTipo** sólo funciona con parámetros cuyo tipo debe ser no abstracto y con un constructor público sin parámetros.

Práctica sobre la teoría 9

4) Dada la siguiente clase genérica

```
class Nodo<T>
{
   public T Valor { get; private set; }
   public Nodo<T>? Proximo { get; set; } = null;
   public Nodo(T valor) => Valor = valor;
}
```

Utilizar la clase **Nodo<T>** para codificar codificar una lista enlazada genérica tal manera que el código siguiente produzca la salida indicada:

```
var lista = new ListaEnlazada<int>();
lista.AgregarAdelante(3);
lista.AgregarAdelante(100);
lista.AgregarAtras(10);
lista.AgregarAtras(11);
lista.AgregarAdelante(0);
IEnumerator<int> enumerador = lista.GetEnumerator();
while (enumerador.MoveNext())
{
   int i = enumerador.Current;
   Console.Write(i + " ");
}
Console.WriteLine();
```

Salida por consola

```
0 100 3 10 11
```

Práctica sobre la teoría 9

5) En esta teoría vimos que la clase **List<T>** implementa el método **Foreach** que recibe un **Action<T>** como parámetro y que permite código como el siguiente:

```
List<int> lista = [1, 2, 3];
lista.ForEach(i => Console.Write(i + ","));
```

Salida por consola

```
1,2,3
```

Sin embargo, **Foreach** no está definido para los arreglos. Se desea implementar un método de extensión para todas las clases que implementen la interfaz **IEnumerable<T>**, para hacer posible el siguiente código:

```
int[] vector = [1, 2, 3];
vector.ForEach(i => Console.Write(i + ","));
"Hola Mundo".ForEach(c => Console.Write(c + ","));
```

Salida por consola

```
1,2,3,H,o,l,a, ,M,u,n,d,o,
```

Para ello completar la siguiente clase estática

```
public static class Extensiones
{
   public static void ForEach<T>(this IEnumerable<T> secuencia, . . .
. . .
```

Práctica sobre la teoría 9

6) Codificar los métodos de extensión **Donde** y **Seleccionar** para todas las clases que implementen la interfaz **IEnumerable<T>**, que permite código como el siguiente:

```
int[] vector = [1, 2, 3, 4, 5];
vector.Donde(i => i > 3) // IEnumerable<int> : 4, 5
      .Seleccionar(i => i * 10) // IEnumerable<int> : 40, 50
      .Seleccionar(i => $"({i})") //  IEnumerable<string> : "(40)", "(50)"
      .ForEach(st => Console.WriteLine(st));
```

Salida por consola

```
(40)
(50)
```

Para ello seguir completando la clase estática **Extensiones** del ejercicio anterior

```
public static class Extensiones
{
   public static void ForEach<T>(this IEnumerable<T> secuencia, . . .

   public static IEnumerable<T> Donde<T>(this IEnumerable<T> secuencia, . . .

   public static IEnumerable<TResult> Seleccionar<T, TResult>(this IEnumerable<T> secuencia, . . .


. . .
```

Práctica sobre la teoría 9

7) Dada la siguiente clase que representa una regla de validación con un mensaje de error asociado

```
class ReglaValidacion<T>(Func<T, bool> predicado, string mensajeError)
{
   Func<T, bool> _predicado = predicado;
   public bool Ok(T instancia) => _predicado(instancia);
   public string MensajeError { get; } = mensajeError;
}
```

El campo **`_predicado`** es una función que toma un objeto de tipo **`T`** y devuelve un valor booleano indicando si la regla se cumple o no.

La propiedad **`MensajeError`** es un mensaje descriptivo que se mostrará si la regla de validación no se cumple.

Se debe implementar la clase **`Validador<T>`** que permitirá la validación de objetos de tipo **`T`** según las reglas proporcionadas.

Métodos:

1. **`AgregarRegla(ReglaValidacion<T> regla)`:** Este método deberá agregar una regla de validación a la lista de reglas que el validador aplicará.
2. **`Validar(T objeto, out List<string> errores)`:** Este método deberá validar el objeto proporcionado según todas las reglas de validación agregadas. Los errores encontrados deben almacenarse en la lista **`errores`**. Esta lista debe contener los mensajes de error correspondientes a las reglas de validación que no se cumplieron.

El siguiente código, debe producir la salida por la consola indicada:

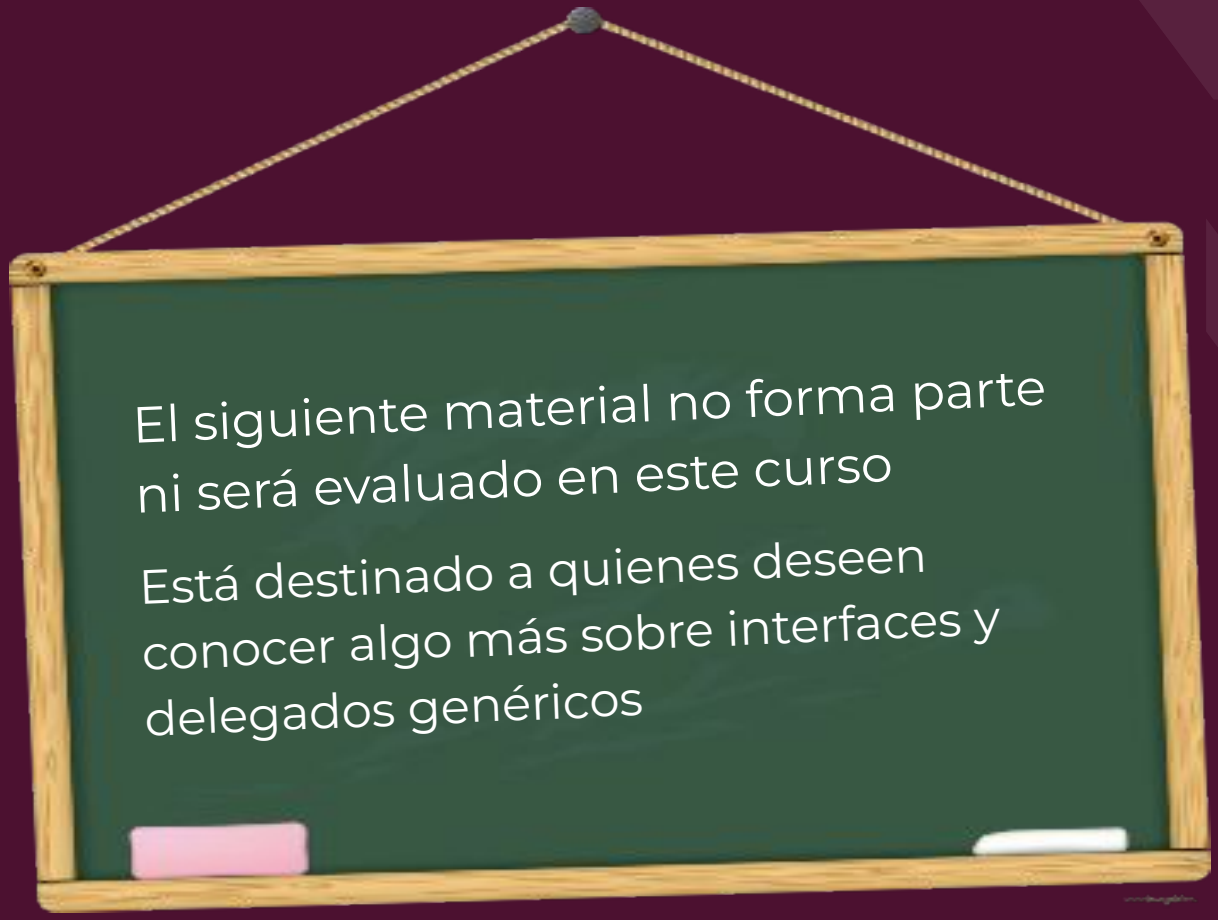
Práctica sobre la teoría 9

```
var regla1 = new ReglaValidacion<Persona>(p => p.Nombre != "", "Falta el nombre");
var regla2 = new ReglaValidacion<Persona>(p => p.Edad <= 35, "Edad excedida");
var regla3 = new ReglaValidacion<Persona>(p => p.Edad >= 18, "Menor de edad");

var validadorPersona = new Validador<Persona>()
                       .AgregarRegla(regla1)
                       .AgregarRegla(regla2)
                       .AgregarRegla(regla3);

var pedro = new Persona { Nombre = "Pedro", Edad = 17 };
var ana = new Persona { Nombre = "", Edad = 16 };
var maria = new Persona { Nombre = "María", Edad = 44 };
var jose = new Persona { Nombre = "José", Edad = 30 };

Validar(pedro, validadorPersona);
Validar(ana, validadorPersona);
Validar(maria, validadorPersona);
Validar(jose, validadorPersona);

void Validar(Persona p, Validador<Persona> validadorPersona)
{
  List<string> listaErrores;
  bool esValido = validadorPersona.Validar(p, out listaErrores);
  Console.WriteLine($"Validando a {p} ==> {esValido}");
  listaErrores.ForEach(st => System.Console.WriteLine("  * " + st));
}
```

Salida por consola

```
Validando a Pedro de 17 años ==> False
  * Menor de edad
Validando a  de 16 años ==> False
  * Falta el nombre
  * Menor de edad
Validando a María de 44 años ==> False
  * Edad excedida
Validando a José de 30 años ==> True
```

Para conocer más



Varianza en interfaces y delegados genéricos
(Covarianza y Contravarianza)

El Problema de la Asignación con Genéricos

Consideremos una jerarquía simple:

```
public class Animal { }
public class Perro : Animal { }
```

Sabemos que podemos hacer:

```
Perro miPerro = new Perro();
Animal miAnimal = miPerro; // ¡Perfectamente válido!
                           //  Un Perro ES UN Animal.
```

Pero lo que sigue, ¿podemos hacerlo?

```
List<Perro> listaDePerros = new List<Perro>();
List<Animal> listaDeAnimales = listaDePerros; // ¿Esto es válido?
```

El Problema de la Asignación con Genéricos

Respuesta: ¡**NO**! Por defecto, los tipos genéricos como List<T> son **INVARIANTES**. List<Perro> NO es un List<Animal>, aunque Perro sea un Animal.

Si el compilador nos dejara hacer esto

```
List<Animal> listaDeAnimales = listaDePerros;
```

La variable `listaDeAnimales` estaría apuntando a una lista de perros en la Heap, entonces, con la siguiente línea válida para el compilador

```
listaDeAnimales.Add(new Gato());
```

Estaríamos agregando un Gato a una lista de Perros y esto rompería la seguridad de tipos, obteniendo un error en tiempo de ejecución.

El Problema de la Asignación con Genéricos

Un ejemplo de este problema de seguridad de tipos, lo constituyen los arreglos que, en lugar de ser INVARIANTES son COVARIANTES (aunque sólo para los tipos de referencia).
Esto significa que si tenemos una clase Perro que hereda de Animal, se puede hacer lo siguiente:

```
Perro[] perros = new Perro[3];
Animal[] animales = perros; // ¡Esto es VÁLIDO! el compilador lo permite
```

La razón histórica para esto fue, en parte, facilitar la interoperabilidad con Java (que tenía arreglos covariantes) y permitir que algunos algoritmos genéricos de ordenamiento y búsqueda (que operaban sobre object[]) pudieran funcionar con arreglos de tipos más específicos.

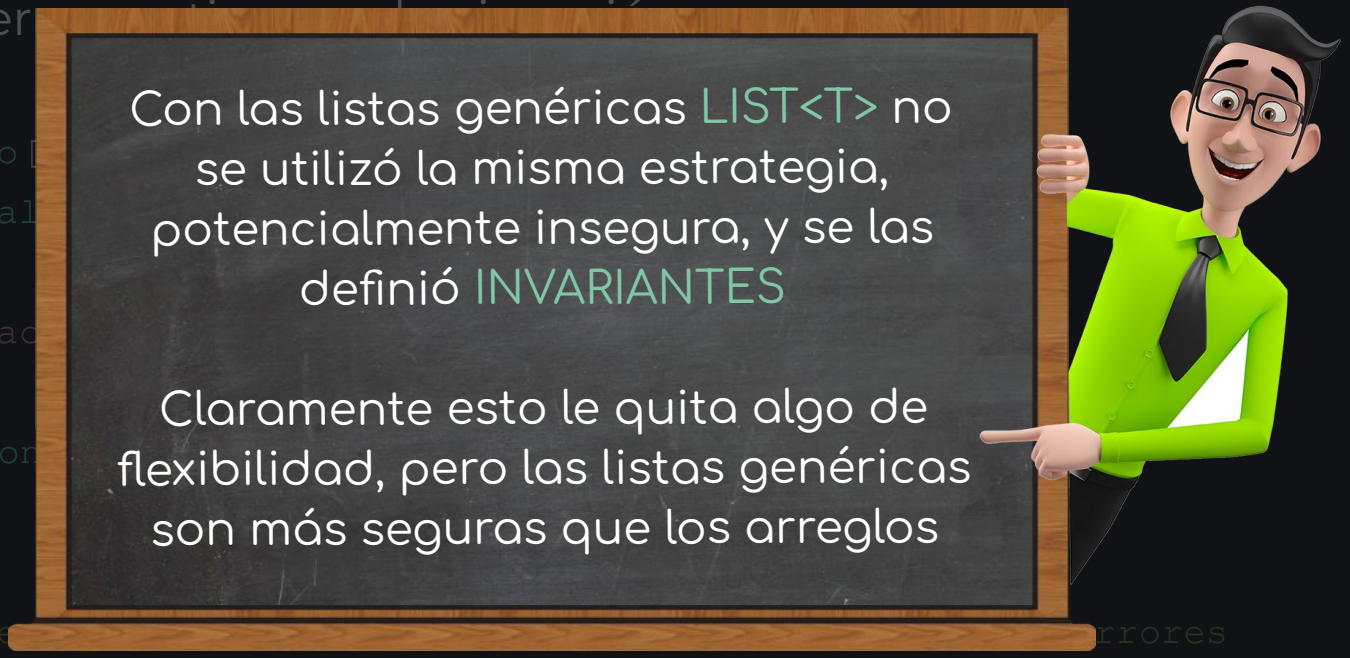
El Problema de la Asignación con Genéricos

Sin embargo esto rompe la seguridad de tipos, y podemos tener un error en tiempo de ejecución:

```
Perro[] perros = new Perro[3];
Animal[] animales = perros; // ¡Esto es VÁLIDO!

foreach(Animal a in animales)
{
   Console.Write(a); // Funciona sin problemas
}

// Pero si queremos asignar elementos podemos tener errores
animales[0] = new Perro(); //ok
animales[1] = new Gato();  // Compila pero produce error en ejecución
                           // ArrayTypeMismatchException
```

El Problema de la Asignación con Genéricos

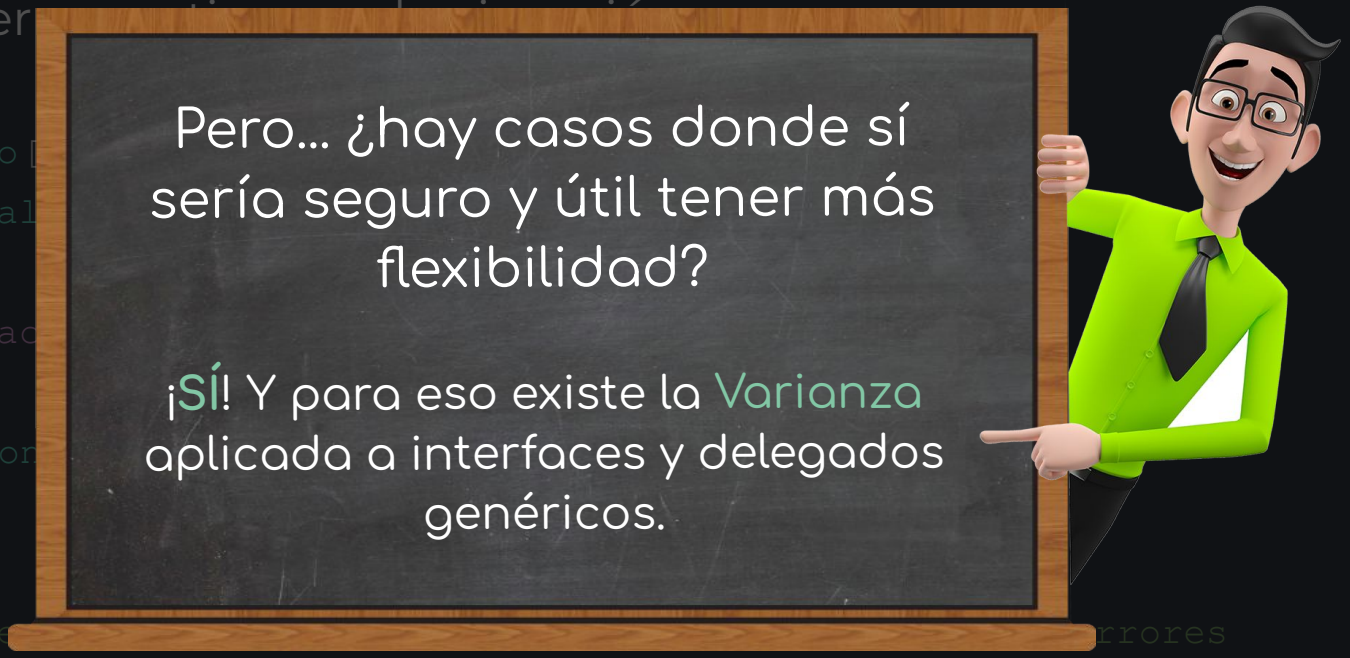
Sin embargo esto rompe la seguridad de tipos, y podemos tener un er




```
Perro[
Animal

foreac
{
    Con
}

// Pe                                    rrores
animales[0] = new Perro(); //ok
animales[1] = new Gato();  // Compila pero produce error en
ejecución
                           // ArrayTypeMismatchException
```

El Problema de la Asignación con Genéricos

Sin embargo esto rompe la seguridad de tipos, y podemos tener un er



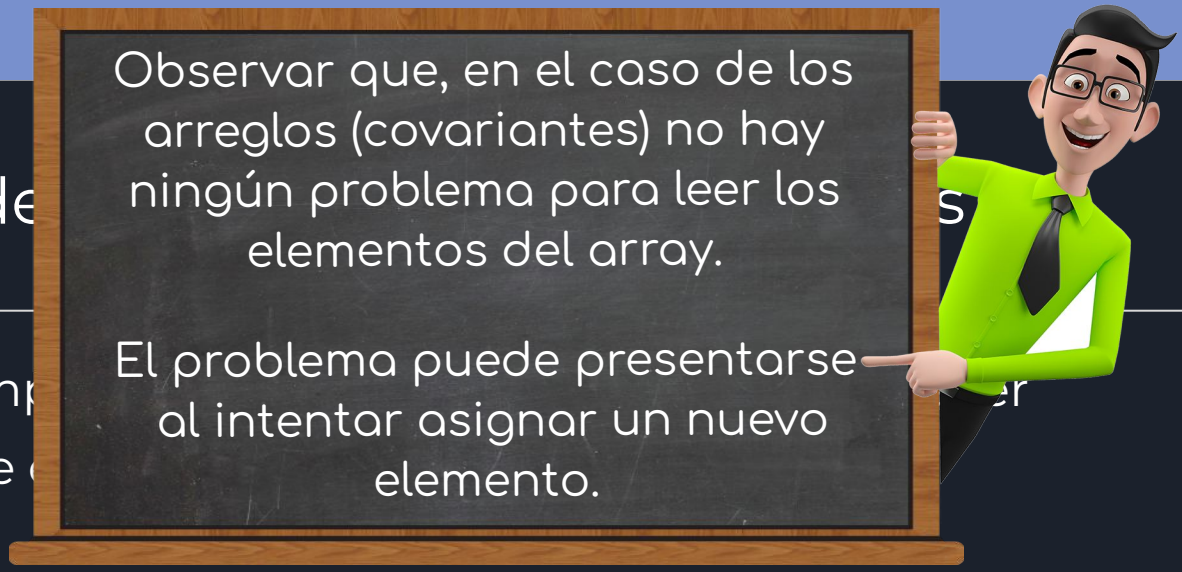

```
Perro[
Animal

foreac
{
    Cor
}

// Pe                                    rrores
animales[0] = new Perro(); //ok
animales[1] = new Gato();  // Compila pero produce error en
ejecución
                           // ArrayTypeMismatchException
```

Covarianza

El Problema de

Sin embargo esto romp

un error en tiempo de




```
Perro[] perros = new Perro[3];
Animal[] animales = perros; // ¡Esto es VÁLIDO!

foreach(Animal a in animales)
{
   Console.Write(a); // Funciona sin problemas
}

// Pero si queremos asignar elementos podemos tener errores
animales[0] = new Perro(); //ok
animales[1] = new Gato();  // Compila pero produce error en ejecución
                           // ArrayTypeMismatchException
```

Covarianza - Cuando el Tipo "Sale" (Out)

- **Concepto**: La covarianza permite usar (como argumento de tipo) un tipo más derivado que el especificado originalmente.
- Aplica cuando el parámetro genérico T solo se usa como tipo de SALIDA (retorno de métodos, propiedades de solo lectura). La interfaz/delegado produce T.
- **Palabra Clave**: out (ej. interface IProductor<out T>)
- **Analogía**: Si una fábrica promete "producir un Automotor" (IProductor<Automotor>), es seguro que entregue un "Auto" (FabricaDeAutos : IProductor<Automotor>), porque un Auto es un Automotor.

Ejemplo de interfaz covariante

```
// T solo se usa como tipo de retorno
public interface IProductor<out T>
{
    T ProducirElemento();
}
```

```
public class CreadorDeStrings : IProductor<string>
{
    public string ProducirElemento() => "Hola Mundo";
}
```

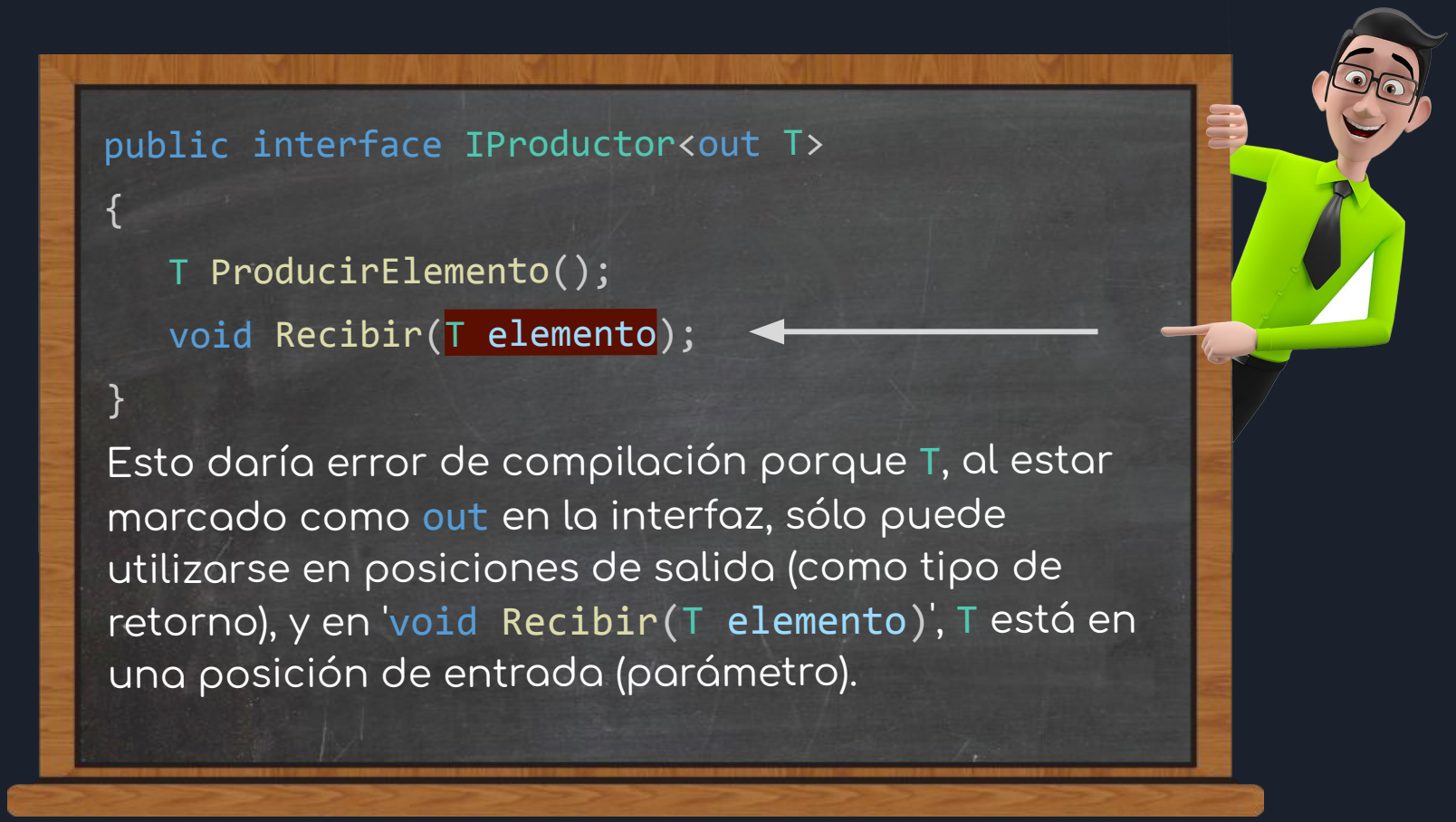
¿Qué permite la covarianza?

```
public interface IProductor<out T> // T solo se usa como tipo de retorno
{
    T ProducirElemento();
}

public class CreadorDeStrings : IProductor<string>
{
    public string ProducirElemento() => "Hola Mundo";
}
```

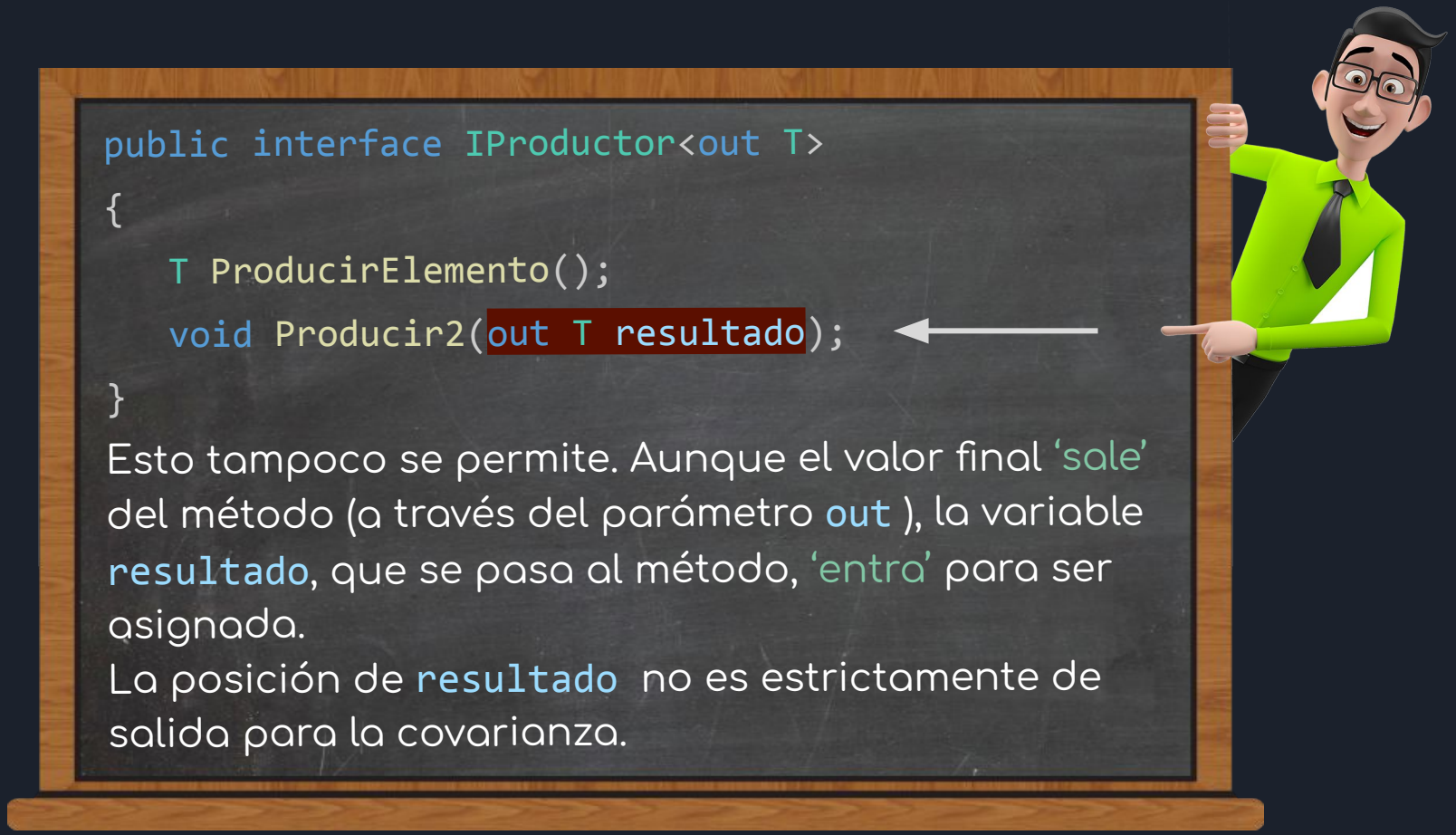
Podemos asignar IProductor<string> a IProductor<object> porque 'string' hereda de 'object' y T está marcado con 'out'.

```
// Uso:
IProductor<string> productorConcreto = new CreadorDeStrings();
IProductor<object> productorAbstracto;

productorAbstracto = productorConcreto;

object resultado = productorAbstracto.ProducirElemento(); // Obtiene "Hola Mundo"
Console.WriteLine(resultado);
```

Restricciones al definir una interfaz covariante



Restricciones al definir una interfaz covariante



Restricciones al definir una interfaz covariante

Si se necesita que un método "devuelva" un valor a través de un parámetro de salida (out) y ese parámetro es del tipo genérico T, la interfaz/delegado debe ser invariante respecto a T (es decir, no usar ni 'in' ni 'out' en la declaración de T para la interfaz/delegado).

```
public interface IProductor<T>
{
    void Producir2(out T parametro);
}
```

Es invariante

Restricciones al definir una interfaz covariante

Los parámetros de método marcados con 'ref', 'in', y 'out' implican que el tipo de dato de ese parámetro (si es un parámetro de tipo genérico T) se está utilizando de una manera que no es puramente de salida (para covarianza) ni puramente de entrada (para contravarianza).

Por lo tanto, si un parámetro de tipo genérico 'T' de una interfaz o delegado se usa como el tipo de un parámetro de método con 'ref', 'in', u 'out', la interfaz/delegado se vuelve **invariante** con respecto a ese 'T'.

Covarianza en Acción - IEnumerable<T>

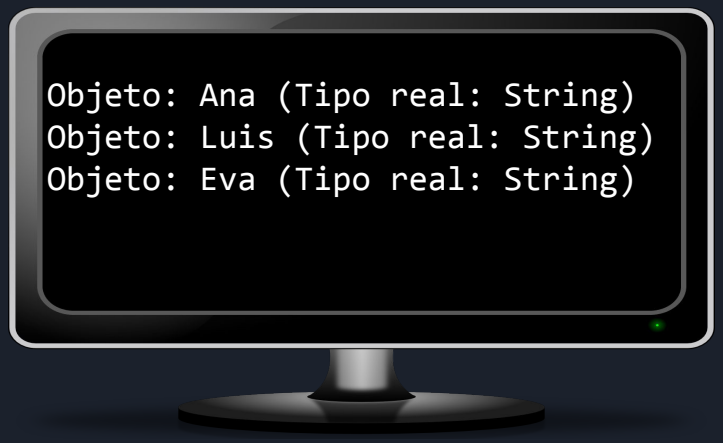
- Ya hemos utilizado interfaces covariantes, quizá sin saberlo.
- La interfaz IEnumerable<T> de .NET es covariante:

```
public interface IEnumerable<out T> : IEnumerable
```

- Esto posibilita, por ejemplo, que a una colección de string (que es IEnumerable<string>), se la pueda tratar como una colección de object (IEnumerable<object>).

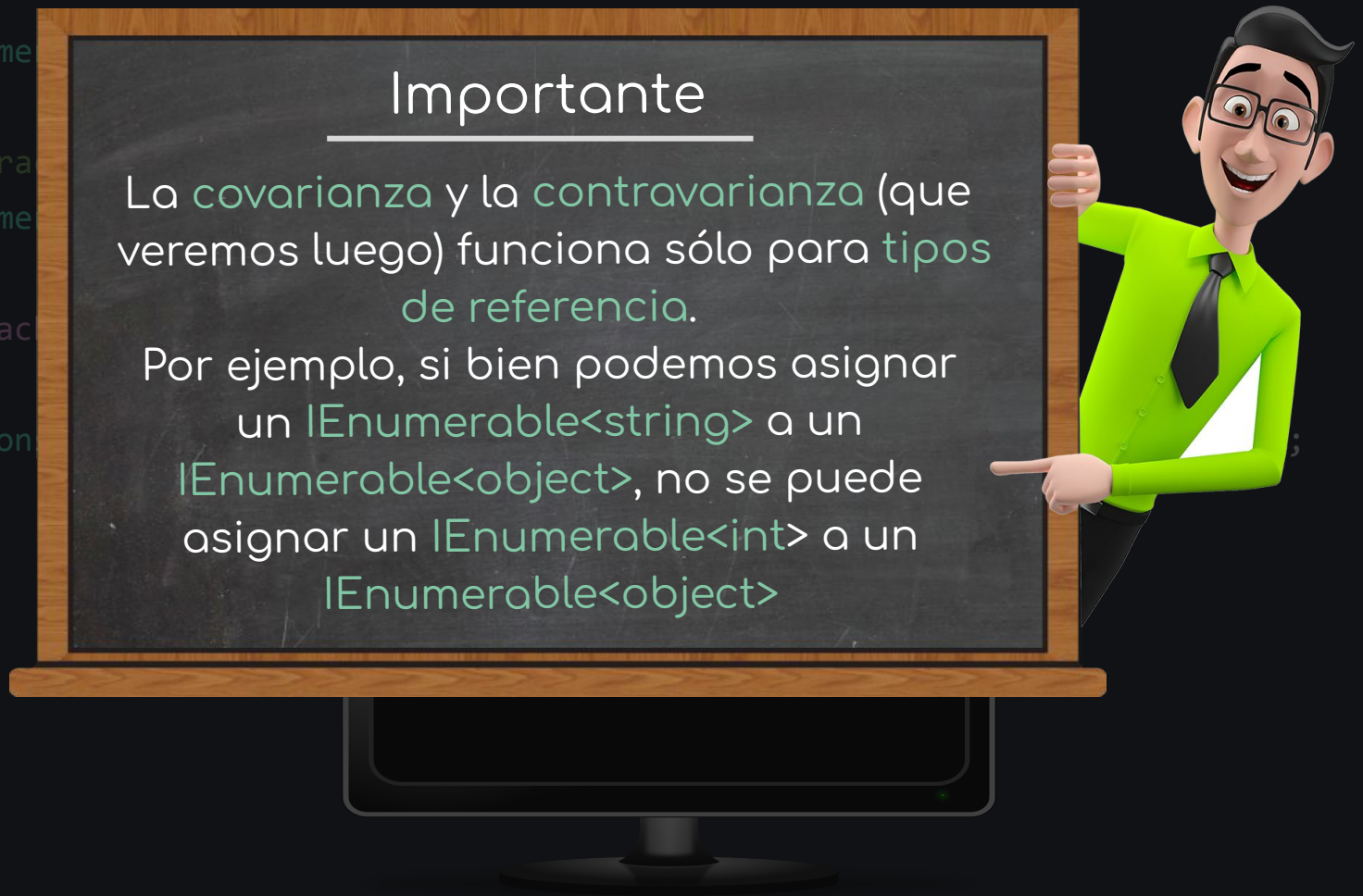
Covarianza en Acción - IEnumerable<T>

```
List<string> listaDeNombres = [ "Ana", "Luis", "Eva" ];

IEnumerable<string> secuenciaDeStrings = listaDeNombres;

// Gracias a que IEnumerable<T> es covariante en T:
IEnumerable<object> secuenciaDeObjetos = secuenciaDeStrings;

foreach (object obj in secuenciaDeObjetos)
{
    Console.WriteLine($"Objeto: {obj} (Tipo real: {obj.GetType().Name})");
}
```



Covarianza en Acción - IEnumerable<T>

```
List<string> listaDeNombres = [ "Ana", "Luis", "Eva" ];

IEnume

// Gra
IEnume

foreac
{
    Con
}
```



Covarianza en Acción - Func<T>

- Ya hemos utilizado delegados covariantes, quizá sin saberlo.
- El delegado Func<T> de .NET es covariante:

```
public delegate TResult Func<out T>();
```

- Esto posibilita que, si Perro deriva de Animal, se pueda asignar un Func<Perro> a un Func<Animal>.

Covarianza en Acción - Func<T>

```
class Animal
{
    public override string ToString() => "Soy un animal";
}
class Perro : Animal
{
    public override string ToString() => "soy un Perro";
}
class Gato : Animal
{
    public override string ToString() => "Soy un Gato";
}
```

```
Func<Perro> fp = () => new Perro();
Func<Gato> fg = () => new Gato();
Func<Animal> fa = () => new Animal();

Console.WriteLine(fa.Invoke()); // imprime: Soy un animal
fa = fg;  // covarianza: se asigna un Func<Gato> a un Func<Animal>
Console.WriteLine(fa.Invoke()); // imprime: Soy un Gato
fa = fp; // covarianza: se asigna un Func<Perro> a un Func<Animal>
Console.WriteLine(fa.Invoke()); // imprime: Soy un Perro
```

Contravarianza

Contravarianza - Cuando el Tipo "Entra" (in)

- **Concepto**: La contravarianza permite usar (como argumento de tipo) un tipo menos derivado que el especificado originalmente.
- **Cuándo aplica**: Cuando el parámetro genérico T solo se usa como tipo de ENTRADA (parámetro de métodos). La interfaz/delegado consume T.
- **Palabra Clave**: in (ej. interface IConsumidor<in T>)
- **Analogía**: Si tienes una máquina que sabe "procesar cualquier Fruta" (IConsumidor<Fruta>), entonces seguro que también sabe "procesar una Manzana" (y podrías asignar IConsumidor<Fruta> a IConsumidor<Manzana>). La asignación parece "al revés" de la herencia.

Ejemplo de interfaz contravariante

```
public interface IProcesador<in T>
// T solo se usa como parámetro
{
    void ProcesarElemento(T elemento);
}
```

```
public class ProcesadorDeObjetos : IProcesador<object>
{
    public void ProcesarElemento(object elemento)
        => Console.WriteLine($"Procesando objeto: {elemento}");
}
```

¿Qué permite la contravarianza?

```
public interface IProcesador<in T>
// T solo se usa como parámetro
{
    void ProcesarElemento(T elemento);
}

public class ProcesadorDeObjetos : IProcesador<object>
{
    public void ProcesarElemento(object elemento)
        => Console.WriteLine($"Procesando objeto: {elemento}");
}
```

```
// Uso:
IProcesador<object> procesadorGeneral = new ProcesadorDeObjetos();
IProcesador<string> procesadorDeStrings;

procesadorDeStrings = procesadorGeneral;

// Llama a ProcesadorDeObjetos.ProcesarElemento
procesadorDeStrings.ProcesarElemento("Texto de prueba");
```

Podemos asignar IProcesador<object> a IProcesador<string> porque 'object' es clase base de 'string' y T está marcado con 'in'

¿Qué permite la contravarianza?

```
public interface IProcesador<in T>
// T solo se usa como parámetro
{
    void ProcesarElemento(T elemento);
}

public class ProcesadorDeObjetos : IProcesador<object>
{
    public void ProcesarElemento(object elemento)
        => Console.WriteLine($"Procesando objeto: {elemento}");
}
```

Seguridad: Es seguro porque si un método espera un string, y le pasas un método que sabe manejar object, este último podrá manejar el string sin problemas.

```
// Uso:
IProcesador<object> procesadorGeneral = new ProcesadorDeObjetos();
IProcesador<string> procesadorDeStrings;

procesadorDeStrings = procesadorGeneral;

// Llama a ProcesadorDeObjetos.ProcesarElemento
procesadorDeStrings.ProcesarElemento("Texto de prueba");
```

Restricciones al definir una interfaz contravariante

```
public interface IProcesador<in T>
{
    void ProcesarElemento(T elemento);
    void Procesar3(in T elemento);
    void Procesar4(out T elemento);
    void Procesar5(ref T elemento);
    T Procesar2();
}
```

Los últimos 4 métodos de la interfaz provocan error de compilación porque T, al estar marcado como in en la interfaz, sólo puede utilizarse en posiciones estrictamente de entrada.

Contravarianza en Acción - IComparer<T>

```
public interface IComparer<in T>
```

```
// Comparador que sabe comparar cualquier 'object'
// basándose en su ToString()
public class ComparadorObjetos : IComparer<object>
{
    public int Compare(object x, object y)
    {
        return x.ToString().CompareTo(y.ToString());
    }
}
```

```
List<string> nombres = ["Eva", "Ana", "Luis"];
IComparer<object> comparadorGeneral = new ComparadorObjetos();

// Gracias a la contravarianza:
IComparer<string> comparador = comparadorGeneral;
nombres.Sort(comparador);
nombres.ForEach(st=>Console.WriteLine(st));
```

Contravarianza en Acción - Action<T>

```
public delegate void Action<in T>(T obj);
```

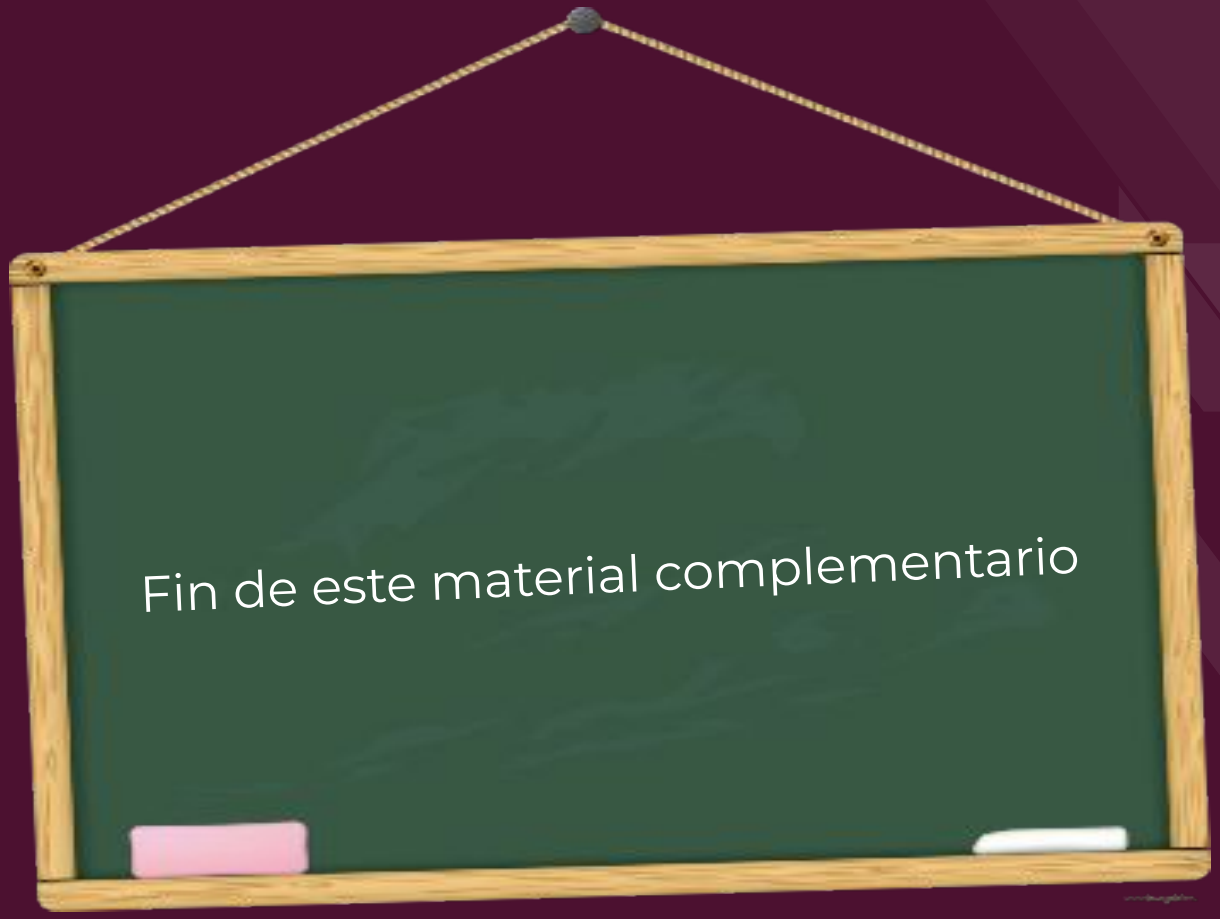
```
public class Animal
{
    public string Nombre { get; set; }
}
public class Perro : Animal
{ }
```

```
Perro miPerro = new Perro { Nombre = "Chopper" };

Action<Animal> accionSobreAnimal =
    a => Console.WriteLine($"Animal: {a.Nombre}");

// Un método que procesa Animales puede procesar Perros
// Gracias a la contravarianza:
Action<Perro> accionSobrePerro = accionSobreAnimal;

accionSobrePerro(miPerro); // Llama a accionSobreAnimal
```

Invarianza

Invarianza - El caso por defecto (Sin in ni out)

- Si un parámetro genérico T en una interfaz o delegado se usa tanto como entrada (parámetro) Y como salida (retorno), o como tipo de una propiedad de lectura/escritura, entonces es INVARIANTE.
- No se pueden usar las palabras clave in ni out.
- Esto significa que no hay flexibilidad de asignación como la vimos con covarianza o contravarianza. IMiInterfaz<Perro> NO es asignable a IMiInterfaz<Animal> ni viceversa.
- List<T> es un ejemplo clásico de tipo genérico invariante, porque puedes Add(T item) (T es entrada) y T Get(int index) (T es salida).


Fin de este material complementario