



.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

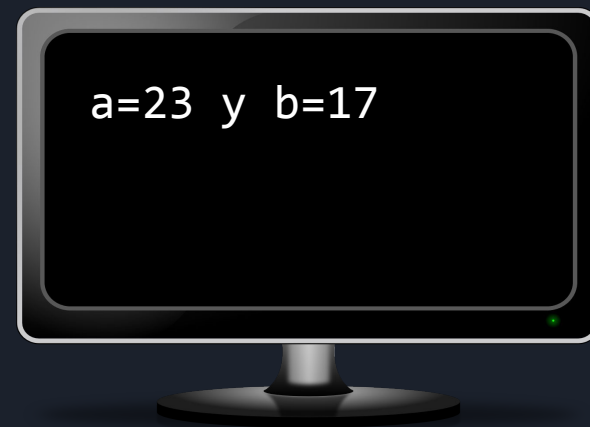


Codificar el método Swap faltante en el siguiente código:



```
-----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_RecursosParaLaTeoria.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`?



Podríamos usar sobrecarga de métodos

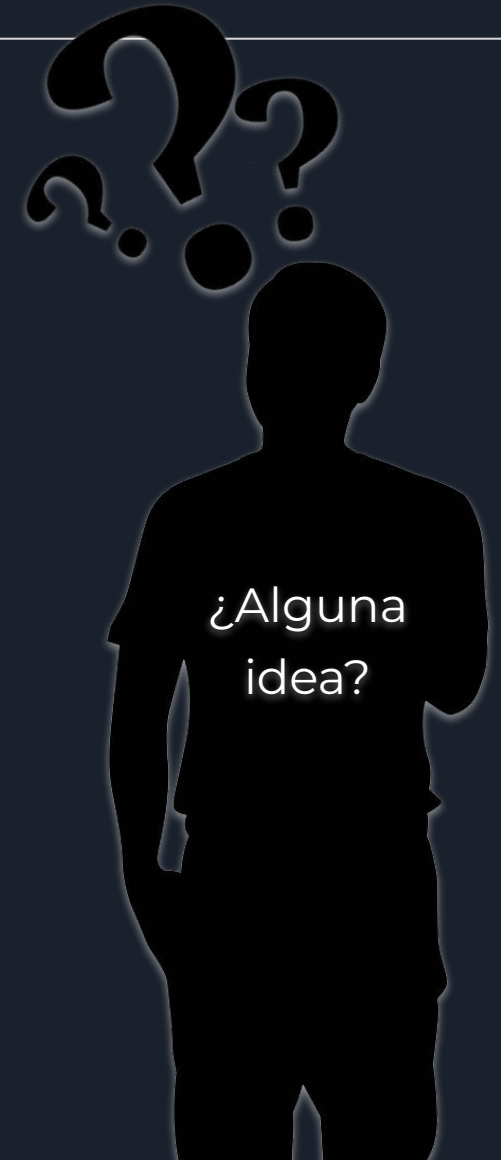
Aunque deberíamos dejar de utilizar instrucciones de nivel superior en `program.cs`



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.



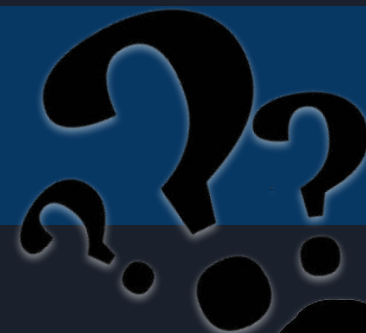


Probar si alcanza con esta sencilla modificación.
(Eliminar las otras versiones de Swap)



```
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



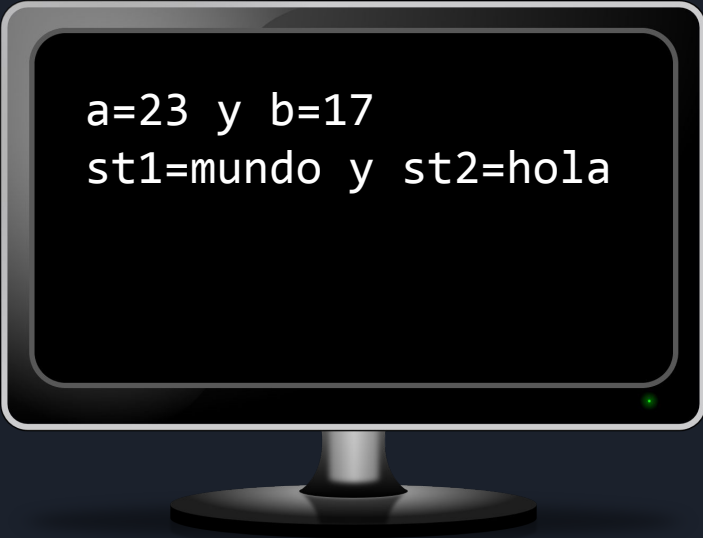
Podríamos usar variables de tipo object
para pasar los parámetros

```
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!



a=23 y b=17
st1=mundo y st2=hola



```
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;  
}
```

Sin embargo la solución
es demasiado engorrosa
además de ineficiente
por requerir
conversiones de tipo



Planteo de otra solución

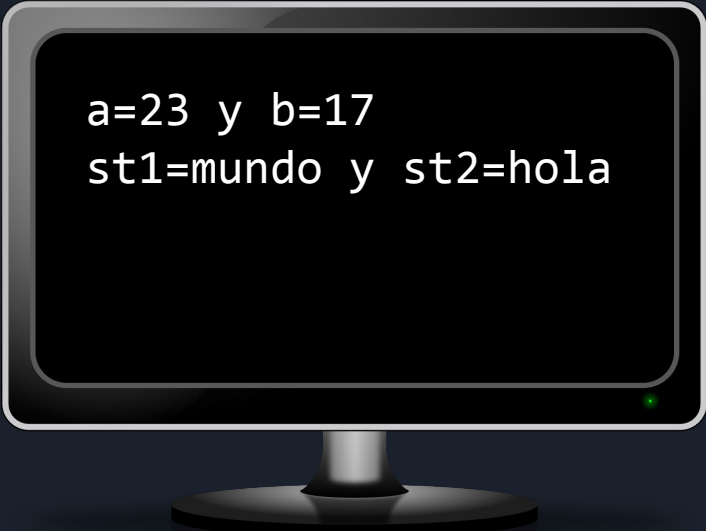
```
dynamic a = 17;
dynamic b = 23;
Swap(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
dynamic st1 = "hola";
dynamic st2 = "mundo";
Swap(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");
```

Usar tipos
dynamic

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

```
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;
}
```



a=23 y b=17
st1=mundo y st2=hola

¡Funciona!
pero el costo es
inaceptable, perdemos
eficiencia y la seguridad
que nos brinda el
chequeo estático de
tipos



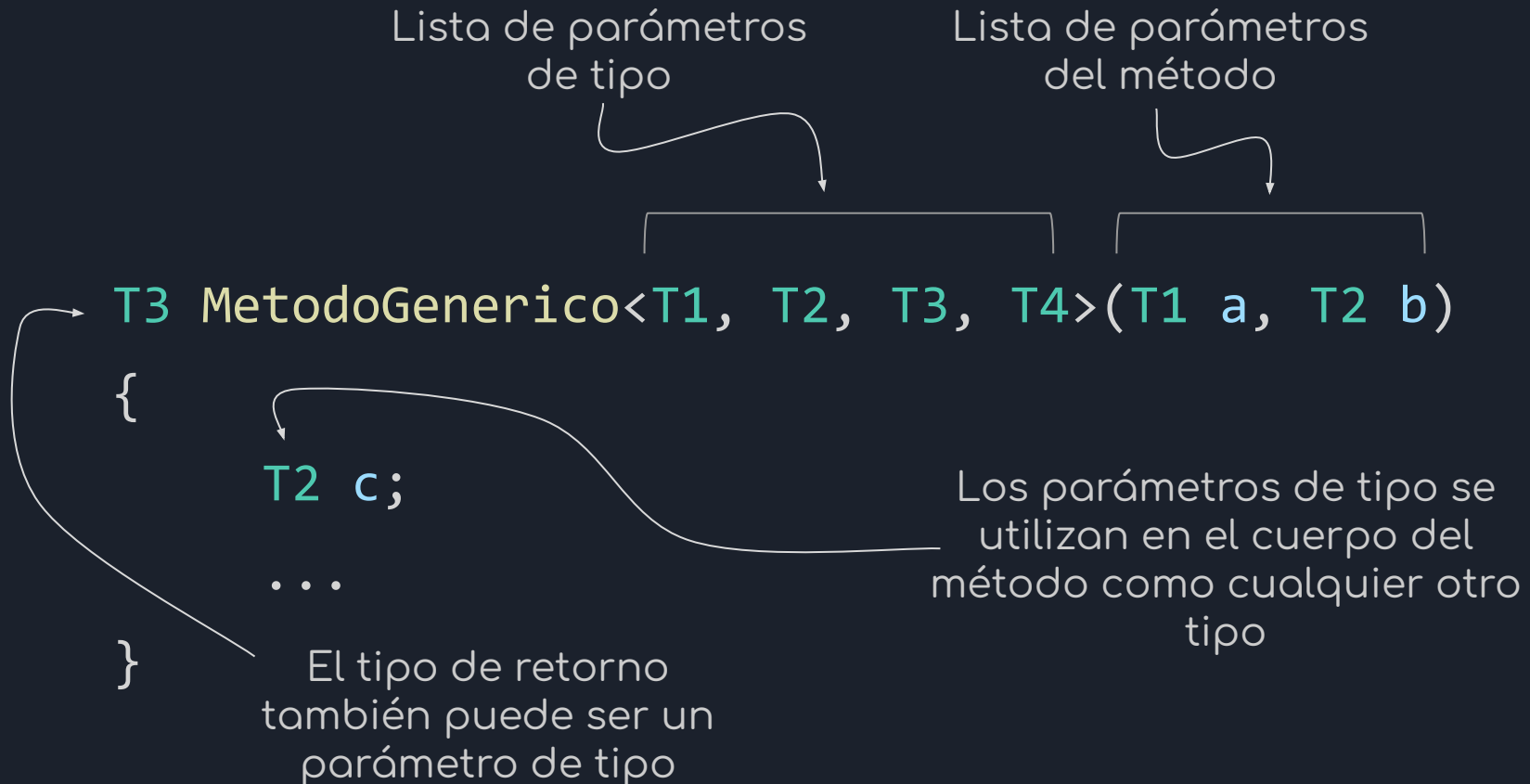
Métodos Genéricos

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



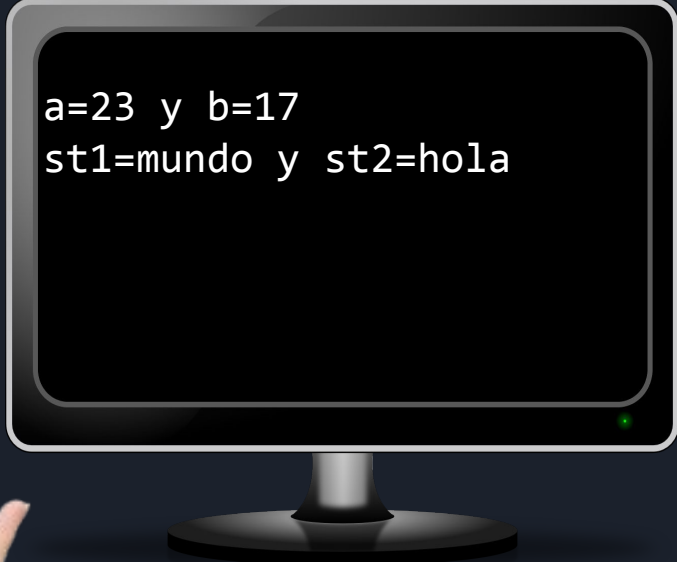
```
int a = 17;
int b = 23;
Swap<int>(ref a, ref b);
Console.WriteLine($"a={a} y b={b}");
string st1 = "hola";
string st2 = "mundo";
Swap<string>(ref st1, ref st2);
Console.WriteLine($"st1={st1} y st2={st2}");

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



a=23 y b=17
st1=mundo y st2=hola



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 `Maximo` que devuelva el mayor de dos instancias de un tipo que admite la comparación de sus elementos, pero evitando la sobrecarga

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



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



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

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

Solucionar
errores de
compilación

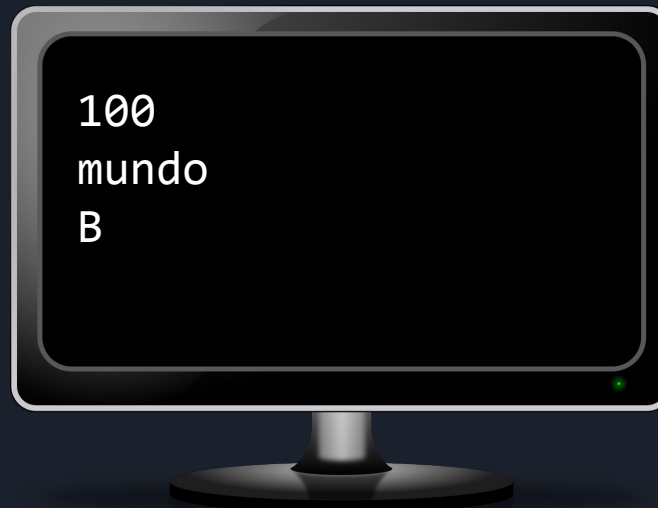


Código en el archivo
09_RecursosParaLaTeoria.txt

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

es necesario
hacer casting

```
Comparable Maximo(Comparable a, Comparable b)  
{  
    if (a.CompareTo(b) > 0)  
    {  
        return a;  
    }  
    return b;  
}
```





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)Maximo(100, 55);
```



Consideraciones

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

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

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

Código en el archivo
09_RecursosParaLaTeoria.txt

Probar esta solución con métodos genéricos

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

```
T Maximo<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 Maximo<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:

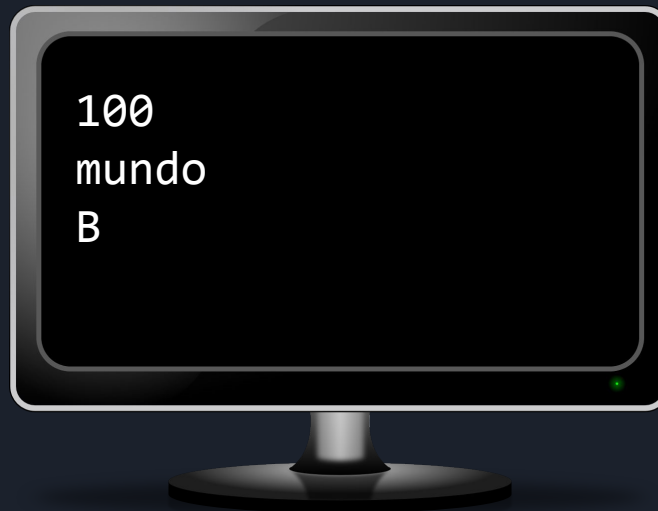


```
T Maximo<T>(T a, T b) where T : IComparable
{
    if (a.CompareTo(b) > 0)
    {
        return a;
    }
    return b;
}
```

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

```
Maximo<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 **Maximo**("hola",55) sería detectado en tiempo de compilación



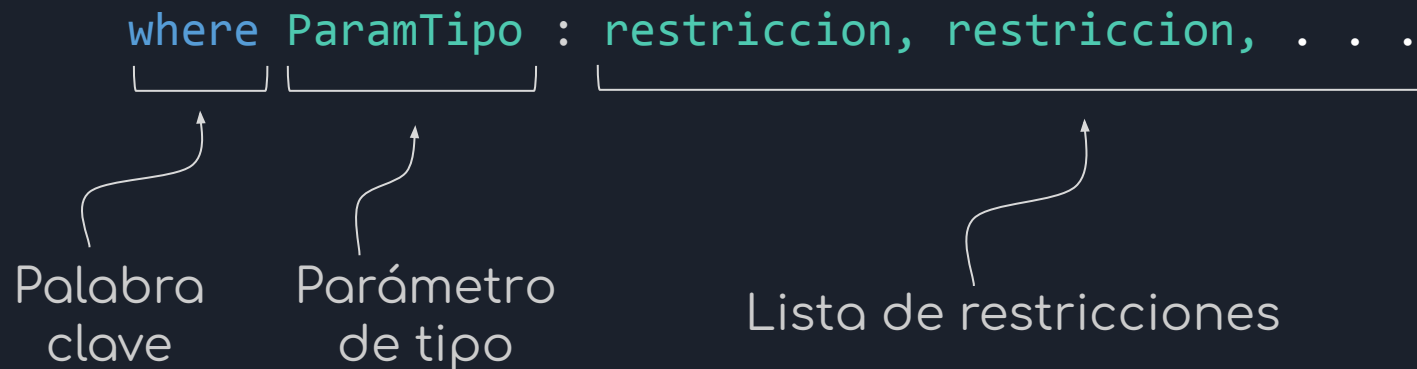


Restricciones sobre los parámetros de tipo - Cláusulas where

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



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

```
T3 MetodoGenerico<T1, T2, T3, T4, T5>(T2 a)
```

```
  where T3 : IEnumerable, new()
```

```
  where T5 : class
```

```
  where T1 : Persona
```

```
  where T4 : struct
```

```
{
```

```
  . . .
```

```
}
```

T5 debe ser cualquier tipo referencia

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

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

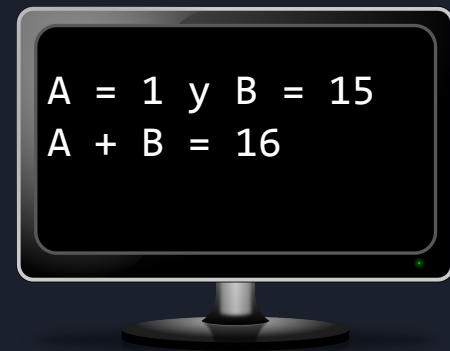
Ejemplo de una clase simple

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

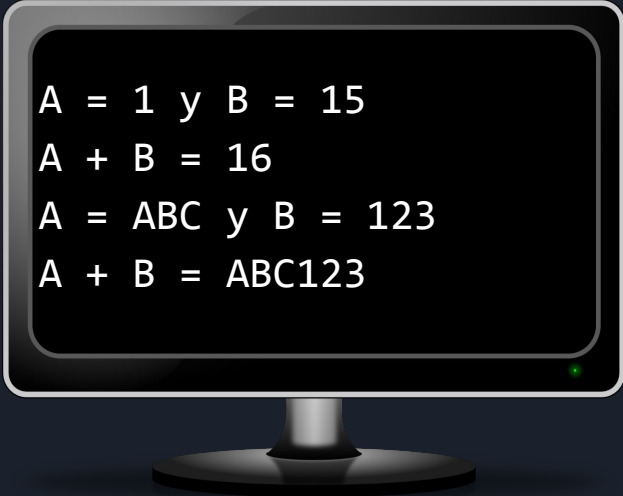
Transformamos la clase **Par** en una **clase genérica**

```
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} ");
```



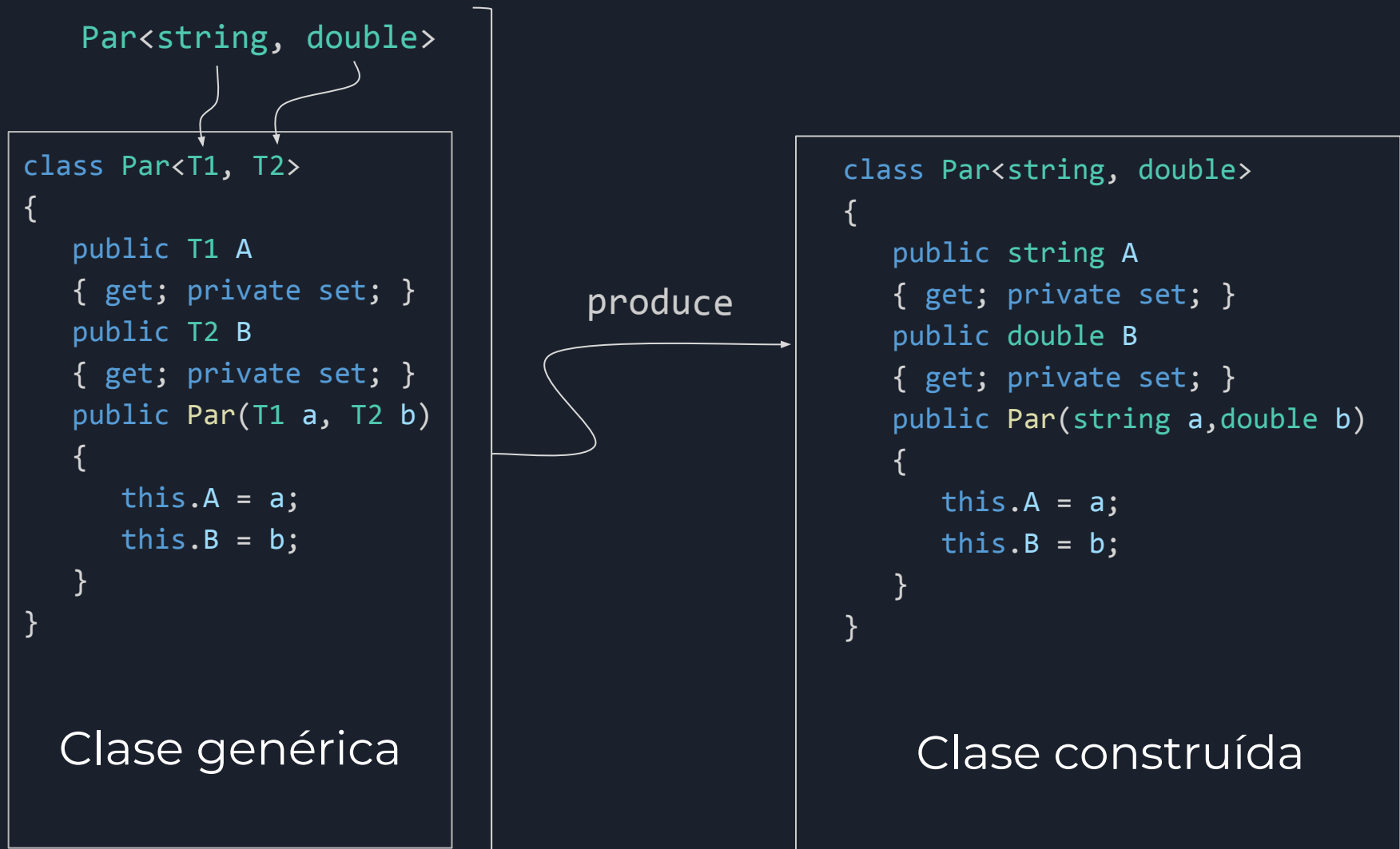
```
A = 1 y B = 15  
A + B = 16  
A = ABC y B = 123  
A + B = ABC123
```

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





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 IRetornador<T>
{
    T Retornar(T valor);
}
```

Interfaces genéricas

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

Ejemplo:

```
class Simple : IRetornador<int>, IRetornador<string> {...}
```

2 interfaces
construidas

```
interface IRetornador<int>
{
    int Retornar(int valor);
}
```

```
interface IRetornador<string>
{
    string Retornar(string valor);
}
```

Interfaces genéricas

```
Simple s = new Simple();  
int i = s.Retornar(3);  
string st = s.Retornar("hola");  
Console.WriteLine($"{st} {i}");
```

```
class Simple : IRetornador<int>, IRetornador<string> {  
    public int Retornar(int valor) {  
        return valor;  
    }  
    public string Retornar(string valor) {  
        return valor.ToUpper();  
    }  
}
```




Observar que
la clase Simple
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 ClaseGen<T1, T2> : IRetornador<T2>
{
    public T2 Retornar(T2 valor)
    {
        return valor;
    }
    . . .
}
```



Cuando se construya
ClaseGen con tipos reales,
por ejemplo
ClaseGen<char, int> se
construye también la
interfaz **IRetornador<int>**



Interfaces genéricas con clases genéricas

`ClaseGen<char, int>`

```
class ClaseGen<char, int> :  
    IRetornador<int>  
{  
    public int Retornar(int valor)  
    {  
        return valor;  
    }  
    . . .  
}
```


Clase construida

```
interface IRetornador<int>  
{  
    int Retornar(int valor);  
}
```

Interfaz
construida

Atención! esto no está permitido


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



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

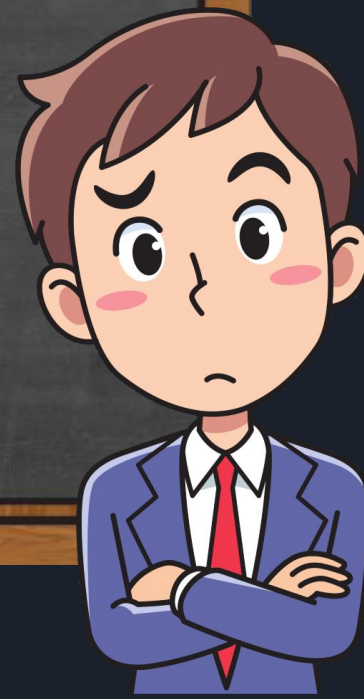
Atención! esto no está permitido

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



Por la misma razón, esto tampoco es válido.

Observar que clase construida `ClaseGen<int,int>` tendría 2 interfaces `IRetornador<int>` lo cual no se permite.



Interfaces genéricas

- .NET ofrece muchas interfaces genéricas para diferentes escenarios. Entre muchas otras están `IComparer<T>`, `Comparable<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:

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

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

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



Delegados Genéricos

Delegados genéricos

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

The diagram shows the declaration `delegate T3 DelGenerico<T1, T2, T3>(T1 a, T2 b);` with three annotations:

- An arrow points from the text "Lista de parámetros de tipo" to the angle brackets and the type parameters `T1, T2, T3`.
- An arrow points from the text "El tipo de retorno también puede ser un parámetro de tipo" to the return type `T3`.
- An arrow points from the text "Lista de parámetros formales de los métodos que se aceptarán" to the parameter list `(T1 a, T2 b)`.

```
delegate T3 DelGenerico<T1, T2, T3>(T1 a, T2 b);
```

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;
```

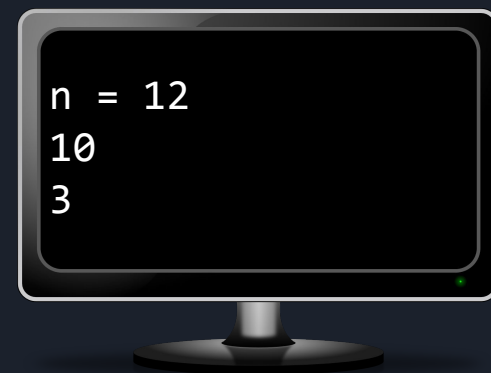
```
Funcion<int[],double> f = Promedio;
```

```
Console.WriteLine(lambda1(12));
```

```
Console.WriteLine(lambda2("hola mundo"));
```

```
Console.WriteLine(f(new int[]{1,2,3,4,5}));
```

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



Delegado 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

Delegado 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) {  
    ...  
}
```

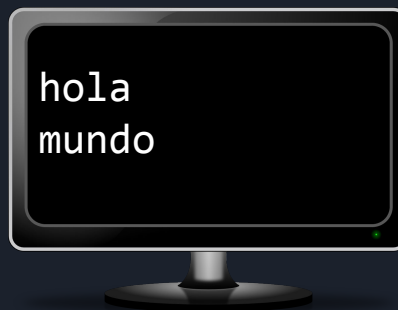

Delegados Action<T> y Func<T> Ejemplos

- 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 = new List<string>() { "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>`



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`.

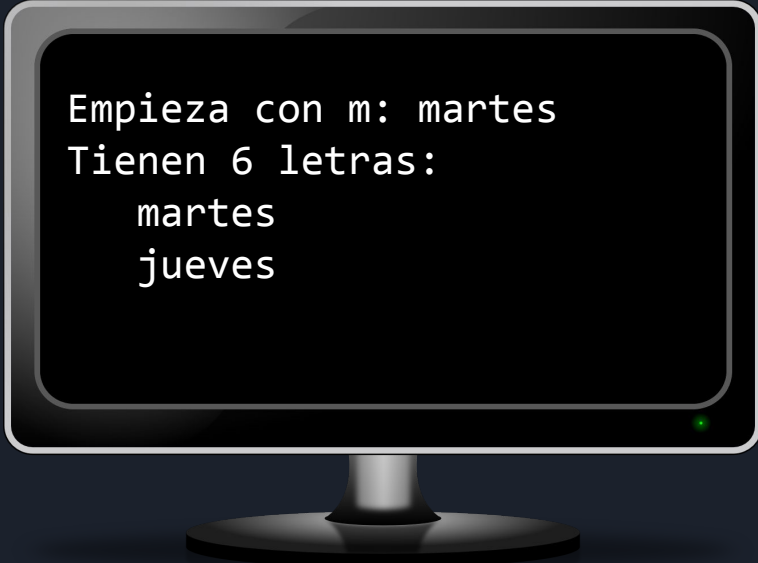
```
public T Find (Predicate<T> match);
```

- `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));
```



```
Empieza con m: martes  
Tienen 6 letras:  
    martes  
    jueves
```



Delegado Converter<TInput,TOutput>

Representa un método que convierte un tipo de objeto a otro tipo.

```
public delegate TOutput Converter<TInput,TOutput>(TInput input);
```

A este delegado lo usa el método `ConvertAll` de la clase `List<T>` para convertir cada elemento de la colección de un tipo a otro.

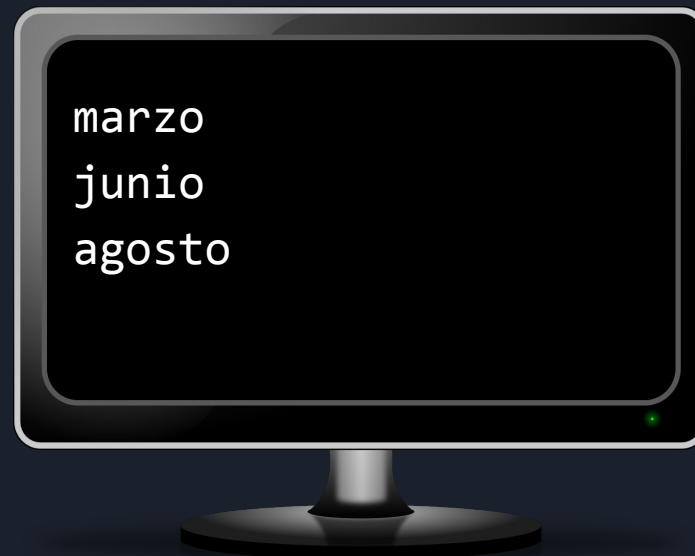
```
public List<TOutput> ConvertAll<TOutput>(Converter<T,TOutput> converter);
```

Ejemplo:

```
List<int> lista = new List<int>() { 3, 6, 8 };  
List<string> listaMeses = lista.ConvertAll<string>(NroToNombre);  
listaMeses.ForEach(st => Console.WriteLine(st));
```

```
string NroToNombre(int i) {  
    DateTime fecha = new DateTime(2000, i, 1);  
    return fecha.ToString("MMMM");  
}
```

El parámetro de tipo se puede omitir porque el compilador puede inferirlo



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);
```

Fin