Unidad

4

DIPLOMATURA EN PROGRAMACION .NET

Capítulo 7

Genéricos

Genéricos

En este capítulo

- Genéricos
- Restricciones en los tipos que son argumentos genéricos
- Interfaces genéricas del Framework de .Net
- Invarianza
- Covariancia

Genéricos

La producción de software siempre busca la disminución de errores provocados por descuidos al programar. Uno de los problemas más recurrente es el de la conversión de tipos de objetos en tiempo de ejecución.

Por lo visto hasta el momento, convertir tipos es el arma más importante al utilizar polimorfismo (es más, se basa en ello). Sin embargo esto deja abierta una puerta peligrosa, sobre todo cuando se utiliza Object, porque cualquier tipo puede ser convertido a esta clase.

Uno de los temas más importantes al aprender polimorfismo es no olvidar que cada objeto conserva su identidad a pesar que la referencia que lo señala haya sido convertida al tipo de una superclase y, por lo tanto, esa conversión puede ser revertida. Esta herramienta se convierte en un arma de doble filo cuando se trata de recuperar el tipo original que fue convertido y se comete un error al realizar la conversión inversa, originando errores en tiempo de ejecución.

Esto impone como meta tratar de poseer una herramienta capaz de detectar este tipo de errores al menos en tiempo de compilación, para que estos sean más fáciles de corregir. Esta es la finalidad de los genéricos.

Los genéricos restringen al código para que los tipos se conviertan en tiempo de compilación en lugar de tiempo de ejecución. Dado este hecho, si un tipo no es “convertible” se detecta en tiempo de compilación y no de ejecución.

A primera vista se puede pensar que esto atenta contra el polimorfismo, sin embargo es todo lo contrario, asegura un mejor desarrollo de software. La razón es que aquello que es diseñado para ser polimórfico se utilice como tal y aquello que no, se especifique con que tipo se quiere utilizar.

Por ejemplo, si se quiere realizar una clase de pila que maneja enteros un primer diseño puede ser intentar manejar esto con un vector.

Ejemplo

C#

```
namespace pila
{
    class ExcepcionPila: Exception
    {
        public ExcepcionPila(String arg0)
            : base(arg0)
        {
        }
    }
}


namespace pila
```

```
{
    public class Pila
    {
        private int[] vec = new int[10];
        private int indice = 0;

        public void Agregar(int valor)
        {
            if (indice == 9)
                throw new ExcepcionPila("Pila llena");
            vec[indice] = valor;
            indice++;
        }

        public int Sacar()
        {
            if (indice == 0)
                throw new ExcepcionPila("Pila vacía");
            indice--;
            return vec[indice];
        }
    }
}
```

VB

```
Public Class ExcepcionPila
    Inherits Exception
    Public Sub New(arg0 As String)
        MyBase.New(arg0)
    End Sub
End Class

Public Class Pila
    Private vec(10) As Integer
    Private indice As Integer = 0

    Public Sub Agregar(valor As Integer)
        If indice = 9 Then
            Throw New ExcepcionPila("Pila llena")
        End If
        vec(indice) = valor
        indice += 1
    End Sub

    Public Function Sacar() As Integer
        If indice = 0 Then
            Throw New ExcepcionPila("Pila vacía")
        End If
        indice -= 1
        Return vec(indice)
    End Function
End Class
```

El primer problema que se puede apreciar realmente es que el código no refleja una **pila** sino una **pila de enteros**, lo cual es bastante limitado.

En un primer intento de generalizar este código, se puede intentar un nuevo diseño de la clase basada en objetos para generalizar la clase Pila. Esta puede además manejar los tipos primitivos por medio del autoboxing, aunque esto signifique una pérdida de rendimiento considerable en cada transformación.

Ejemplo

C#

```
namespace pila2
{
    public class Pila
    {
        private Object[] vec = new Object[10];
        private int indice = 0;

        public void Agregar(Object valor)
        {
            if (indice == 9)
                throw new ExcepcionPila("Pila llena");
            vec[indice] = valor;
            indice++;
        }

        public Object Sacar()
        {
            if (indice == 0)
                throw new ExcepcionPila("Pila vacía");
            indice--;
            return vec[indice];
        }
    }
}
```

VB

```
Public Class Pila
    Private vec(10) As Object
    Private indice As Integer = 0

    Public Sub Agregar(valor As Object)
        If indice = 9 Then
            Throw New ExcepcionPila("Pila llena")
        End If
        vec(indice) = valor
        indice += 1
    End Sub

    Public Function Sacar() As Object
        If indice = 0 Then
            Throw New ExcepcionPila("Pila vacía")
```

```
        End If
        indice -= 1
        Return vec(indice)
    End Function
End Class
```

Una posible clase para utilizar este código se presenta a continuación. Notar que no da ningún tipo de error de compilación.

Ejemplo

C#

```
namespace pila2
{
    class Program
    {
        static void Main(string[] args)
        {
            Pila p = new Pila();

            try
            {
                p.agregar("Hola");
                p.agregar(8); // Autoboxing
                p.agregar(8.8); // Autoboxing
                p.agregar(9);     // Autoboxing

                double total = (double)p.sacar() +
                        (double)p.sacar() +
                        (double)p.sacar() +
                        (double)p.sacar();
                Console.WriteLine(total);
                Console.ReadKey();
            }
            catch (ExcepcionPila e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
        }
    }
}
```

VB

```
Module Module1
```

```
    Sub Main()
        Dim p As New Pila()

        Try
            p.Agregar("Hola")
            p.Agregar(8) ' Autoboxing
            p.Agregar(8.8000000000000007) ' Autoboxing
            p.Agregar(9) ' Autoboxing

            Dim total As Double = DirectCast(p.Sacar(), Double) + _
                                  DirectCast(p.Sacar(), Double) + _
                                  DirectCast(p.Sacar(), Double) + _
                                  DirectCast(p.Sacar(), Double)
            Console.WriteLine(total)
            Console.ReadKey()
        Catch e As ExcepcionPila
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        Catch e As Exception
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        End Try
    End Sub
End Module
```

Cuando se intenta ejecutar el programa se recibe (los directorios cambian según donde se ejecute) el siguiente mensaje:

```
Specified cast is not valid.

   at pila2.Program.Main(String[] args) in
C:\Educacion\Acomodo\Trabajo\Microsoft\Programación en
.Net\EjemplosNuevos\2010\CS\Capitulo7\Capitulo7\Capitulo7\pila2\Program.cs:line
21
```

Se asume que error cometido es intentar convertir un objeto a un tipo base, entonces se utiliza una clase de envoltorio como se muestra a continuación.

Ejemplo

C#

```
namespace pila3
{
    class Program
    {
        static void Main(string[] args)
        {
            Pila p = new Pila();

            try
```

```
            {
                p.Agregar("Hola");
                p.Agregar(8); // Autoboxing
                p.Agregar(8.8); // Autoboxing
                p.Agregar(9);     // Autoboxing

                double total = (Double)p.Sacar() +
                        (Double)p.Sacar() +
                        (Double)p.Sacar() +
                        (Double)p.Sacar();
                Console.WriteLine(total);
                Console.ReadKey();
            }
            catch (ExcepcionPila e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim p As New Pila()

        Try
            p.Agregar("Hola")
            p.Agregar(8) ' Autoboxing
            p.Agregar(8.8000000000000007) ' Autoboxing
            p.Agregar(9) ' Autoboxing

            Dim total As Double = DirectCast(p.Sacar(), System.Double) + _
                                  DirectCast(p.Sacar(), System.Double) + _
                                  DirectCast(p.Sacar(), System.Double) + _
                                  DirectCast(p.Sacar(), System.Double)
            Console.WriteLine(total)
            Console.ReadKey()
        Catch e As ExcepcionPila
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        Catch e As Exception
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
```

```
        End Try
    End Sub
End Module
```

Sin embargo al ejecutar el programa se obtiene el mismo resultado.

```
Specified cast is not valid.

   at pila3.Program.Main(String[] args) in
C:\Educacion\Acomodo\Trabajo\Microsoft\Programación en
.Net\EjemplosNuevos\2010\CS\Capitulo7\Capitulo7\Capitulo7\pila3\Program.cs:line
21
```

En este caso la respuesta parece obvia. Se está tratando de sumar una cadena en el primer elemento que se agrega a la pila, por lo tanto se quita dicho valor. Este último mensaje se hace evidente al examinar el código de cerca en este programa simple, pero puede ser muy difícil de encontrar si el código fuera más extenso.

Ejemplo

C#

```
namespace pila4
{
    class Program
    {
        static void Main(string[] args)
        {
            Pila p = new Pila();

            try
            {
               // p.agregar("Hola");
                p.Agregar(8); // Autoboxing
                p.Agregar(8.8); // Autoboxing
                p.Agregar(9);     // Autoboxing

                double total = //(Double)p.sacar() +
                       (Double)p.Sacar() +
                       (Double)p.Sacar() +
                       (Double)p.Sacar();
                Console.WriteLine(total);
                Console.ReadKey();
            }
            catch (ExcepcionPila e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
```

```
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim p As New Pila()

        Try
            ' p.Agregar("Hola")
            p.Agregar(8) ' Autoboxing
            p.Agregar(8.8000000000000007) ' Autoboxing
            p.Agregar(9) ' Autoboxing

            Dim total As Double = DirectCast(p.Sacar(), System.Double) + _
                                  DirectCast(p.Sacar(), System.Double) + _
                                  DirectCast(p.Sacar(), System.Double) '+ _
            'DirectCast(p.Sacar(), System.Double)
            Console.WriteLine(total)
            Console.ReadKey()
        Catch e As ExcepcionPila
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        Catch e As Exception
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        End Try
    End Sub
End Module
```


Con lo cual se debería de solucionar el problema. Sin embargo al ejecutar el programa, se obtiene nuevamente un error en tiempo de ejecución.

```
Specified cast is not valid.

   at pila4.Program.Main(String[] args) in
C:\Educacion\Acomodo\Trabajo\Microsoft\Programación en
.Net\EjemplosNuevos\2010\CS\Capitulo7\Capitulo7\Capitulo7\pila4\Program.cs:line
21
```

Este último mensaje puede resultar confuso. Si lo que se recibe al extraer de la pila es un objeto y se fuerza una conversión de tipo a Double, ¿por qué no realiza la operación aritmética? La respuesta es que el unboxing se realiza en una asignación, no dentro de una operación aritmética. Para lograr un resultado en la operación aritmética, se debe forzar más aún las cosas para que sea un `double` el que participe de la operación.

Ejemplo

C#

```
namespace pila5
{
    class Program
    {
        static void Main(string[] args)
        {
            Pila p = new Pila();

            try
            {
                // p.agregar("Hola");
                p.Agregar(8); // Autoboxing
                p.Agregar(8.8); // Autoboxing
                p.Agregar(9);     // Autoboxing

                double total = //(Double)p.sacar() +
                 Double.Parse(p.Sacar().ToString()) +
                 Double.Parse(p.Sacar().ToString()) +
                 Double.Parse(p.Sacar().ToString());
                Console.WriteLine(total);
                Console.ReadKey();
            }
            catch (ExcepcionPila e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim p As New Pila()

        Try
            ' p.Agregar("Hola")
            p.Agregar(8) ' Autoboxing
            p.Agregar(8.8000000000000007) ' Autoboxing
            p.Agregar(9) ' Autoboxing

            Dim total As Double = Double.Parse(p.Sacar().ToString()) + _
```

```
                Double.Parse(p.Sacar().ToString()) + _
                Double.Parse(p.Sacar().ToString()) '+ _
            'DirectCast(p.Sacar(), System.Double)
            Console.WriteLine(total)
            Console.ReadKey()
        Catch e As ExcepcionPila
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        Catch e As Exception
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        End Try
    End Sub
End Module
```

Y por fin se obtiene el siguiente resultado *que es un valor numérico*.

```
25,8
```

Esto tiene varios inconvenientes. Primero, los errores de cálculo provienen de los algoritmos de punto flotante que no son exactos, y para poder realizar la cuenta completa se fuerza toda la operación al valor más grande posible. Segundo se hicieron cálculos con los últimos tres elementos de la pila, *pero aún queda un elemento en la pila que es de otro tipo, es más, ni siquiera es un número y se tuvo que eliminar para poder operar con la pila*.

El verdadero problema es en realidad que la pila no puede diferenciar los tipos de datos utilizados, con lo cual no tiene los tipos asegurados (type safe) por estar basada en Object que es superclase de todas las clases de .Net y realizar una serie de conversiones en tiempo de ejecución. Por lo tanto, cuando se produzcan errores se detectarán indefectiblemente en tiempo de ejecución también.

Existen una serie de extensiones a los lenguajes llamados genéricos. Estos son similares a los *templates* de C++. Los genéricos permiten abstraerse de los tipos de datos. Los ejemplos más comunes de esto son las clases contenedoras, como por ejemplo las colecciones (que se explicarán posteriormente).

Si se quisiera diseñar una clase que pudiera manejar coordenadas sin importar como fueran los valores que almacena, enteros o puntos flotantes, se puede utilizar a los genéricos para que el compilador identifique el tipo de datos a utilizar en tiempo de compilación y cualquier intento de operación que sea inadecuada para el tipo, lo reporte como un error en tiempo de compilación.

Los genéricos exigen que se indique el parámetro que define el tipo a utilizar en la declaración de la clase entre los símbolos de mayor y menor que, y ese parámetro define en sí mismo un ***tipo genérico***, que como cualquier tipo se puede utilizar para declarar variables, parámetros o valores

de retorno de los métodos. Una clase para manejar coordenadas con genéricos quedaría de la siguiente manera.

Ejemplo

C#

```
namespace coordenadas
{
    public class Coordenada<T>
    {
        private T x;
        private T y;
        private T z;

        public Coordenada(T x, T y, T z)
        {
            this.x = x;
            this.y = y;
            this.z = z;
        }

        public T X
        {
            get
            {
                return x;
            }
            set
            {
                x = value;
            }
        }

        public T Y
        {
            get
            {
                return y;
            }
            set
            {
                y = value;
            }
        }

        public T Z
        {
            get
            {
                return z;
            }
            set
            {
                z = value;
```

```
            }
        }
    }
}
```

VB

```
Public Class Coordenada(Of T)
    Private _x As T
    Private _y As T
    Private _z As T

    Public Sub New(_x As T, _y As T, _z As T)
        Me._x = _x
        Me._y = _y
        Me._z = _z
    End Sub

    Public Property X() As T
        Get
            Return _x
        End Get
        Set(ByVal value As T)
            _x = value
        End Set
    End Property

    Public Property Y() As T
        Get
            Return _y
        End Get
        Set(ByVal value As T)
            _y = value
        End Set
    End Property

    Public Property Z() As T
        Get
            Return _z
        End Get
        Set(ByVal value As T)
            _z = value
        End Set
    End Property
End Class
```


Para utilizar la clase, por ejemplo con objetos del tipo Int32 (aunque puede ser con cualquier tipo), sólo se debe indicar en la creación del objeto cual es el tipo de parámetro con el cual el compilador debe utilizar las declaraciones dentro de la clase.

La declaración de tipos genéricos no sólo sirve para aquellos que son referenciados. Con un simple cambio en la declaración, la clase se puede convertir en una estructura y estará bien definida. Sin

embargo, el cambio es conceptualemente profundo puesto que en lugar de obtener un tipo referenciado para en CLR, **cuando se lo utilice se crea un tipo por valor**.

Ejemplo

C#

```
public struct Coordenada<T>
{
    private T x;
    private T y;
    private T z;

    public Coordenada(T x, T y, T z)
    {
        this.x = x;
        this.y = y;
        this.z = z;
    }

    public T X
    {
        get
        {
            return x;
        }
        set
        {
            x = value;
        }
    }

    public T Y
    {
        get
        {
            return y;
        }
        set
        {
            y = value;
        }
    }

    public T Z
    {
        get
        {
            return z;
        }
        set
        {
            z = value;
        }
    }
```

```
    }
```

VB

```
    Public Structure Coordenada(Of T)
        Private _x As T
        Private _y As T
        Private _z As T

        Public Sub New(_x As T, _y As T, _z As T)
            Me._x = _x
            Me._y = _y
            Me._z = _z
        End Sub

        Public Property X() As T
            Get
                Return _x
            End Get
            Set(ByVal value As T)
                _x = value
            End Set
        End Property

        Public Property Y() As T
            Get
                Return _y
            End Get
            Set(ByVal value As T)
                _y = value
            End Set
        End Property

        Public Property Z() As T
            Get
                Return _z
            End Get
            Set(ByVal value As T)
                _z = value
            End Set
        End Property
    End Structure
```


Otro punto notable es que si bien la clase Coordenada y la estructura Coordenada son iguales excepto por el manejo que hace de ellas el CLR, cuando se las utiliza puede simplemente no notarse la diferencia desde el código cliente y la única forma de apreciar la diferencia es viendo las respectivas declaraciones. El siguiente ejemplo muestra como el código puede ser exactamente igual tanto si es una clase como una estructura, ya que sirve para ambos tipos sin ninguna modificación.

Ejemplo

C#

```
class Program
{
    static void Main(string[] args)
    {
        Coordenada<Int32> c = new Coordenada<Int32>(2, 3, 4);
        Coordenada<long> c2 = new Coordenada<long>(2, 3, 4);
        Console.WriteLine("X: " + c.X + ", y: "
                + c.Y + ", Z: " + c.Z);
        int suma1 = c.X + c.Y + c.Z;
        long suma2 = c2.X + c2.Y + c2.Z;
        Console.WriteLine("La suma 1 de las coordenadas es: " + suma1);
        Console.WriteLine("La suma 2 de las coordenadas es: " + suma2);
        Console.ReadKey();
    }
}
```

VB

```
Sub Main()
    Dim c As New Coordenada(Of Int32)(2, 3, 4)
    Dim c2 As New Coordenada(Of Long)(2, 3, 4)
    Console.WriteLine("X: " + c.X.ToString + ", y: " _
            + c.Y.ToString + ", Z: " + c.Z.ToString)
    Dim suma1 As Integer = c.X + c.Y + c.Z
    Dim suma2 As Long = c2.X + c2.Y + c2.Z
    Console.WriteLine("La suma 1 de las coordenadas es: " + suma1.ToString)
    Console.WriteLine("La suma 2 de las coordenadas es: " + suma2.ToString)
    Console.ReadKey()
End Sub
```

El resultado obtenido es

```
X: 2, y: 3, Z: 4
La suma 1 de las coordenadas es: 9
La suma 2 de las coordenadas es: 9
```

Como se puede apreciar, se solucionan varios problemas, como todos aquellos que devienen de convertir tipos de datos. Por eso se puede afirmar que el uso de genéricos **asegura el tipo, pero no las operaciones que se realizan con estos**. Por ejemplo si se define un tipo String, la clase Coordenada lo acepta como parámetro válido, por más que no tenga sentido un objeto de ese tipo.

Notar que se puede utilizar genéricos tanto con objetos (como Int32) como con tipos base (como `long`).

Parámetros de los tipos genéricos

Cuando se crea la definición de tipo o método genérico, el parámetro es un marcador de posición que indica los lugares en los cuales el argumento pasado como tipo específico a la declaración

genérica será reemplazado por lo que determine el código cliente que se deba utilizar. Una clase genérica, tal como Coordenada<T> no se puede utilizarse tal cual, ya que no es realmente un tipo, sino que es más bien una plantilla para un tipo al que hay que especificarle un argumento, el cual será tomado como parámetro y a partir de él el compilador infiere como reemplazar las declaraciones en la clase con el argumento recibido. Para utilizar Coordenada<T>, el código cliente debe declarar y crear instancias de un tipo construido mediante la especificación de un argumento de tipo dentro de los símbolos de "mayor que" y "menor que". El argumento de tipo para esta clase en particular puede ser de cualquier tipo reconocido por el compilador. Se pueden crear cualquier número de instancias y cada una puede utilizar un argumento de tipo diferente:

Ejemplo

C#

```
Coordenada<Int32> c = new Coordenada<Int32>(2, 3, 4);
Coordenada<long> c2 = new Coordenada<long>(2, 3, 4);
```

VB

```
Dim c As New Coordenada(Of Int32)(2, 3, 4)
Dim c2 As New Coordenada(Of Long)(2, 3, 4)
```

Genéricos y tipos base

Una pregunta válida es ¿se pueden utilizar genéricos con los tipo base como se los utiliza normalmente en las declaraciones de variables o vectores? La respuesta es un categórico **SI**.

Sin embargo, la duda persiste porque el lenguaje posee la capacidad de autoboxing, ¿no debería manejar automáticamente su capacidad de convertir los tipos base? Nuevamente la respuesta es **SI**. Dado que los tipos genéricos se resuelven en tiempo de compilación, cuando se define el parámetro genérico que se pasa a la clase, todas las operaciones deben quedar bien definidas. Si se realizan operaciones aritméticas se debe ser lo suficientemente precavido para que no se deba resolver un unboxing en ella. Se deben limitar las operaciones de boxing y umboxing a las asignaciones directas para evitar errores inesperados porque, por ejemplo, en una operación aritmética se realiza una instrucción a la vez y realizar un unboxing para luego operarlo implicaría dos operaciones al mismo tiempo.

Para comprender mejor lo expuesto, se crea una versión genérica de la clase Pila.

Ejemplo

C#

```
namespace pila6
{
    public class Pila<T>
    {
        private T[] vec = new T[10];
        private int indice = 0;

        public void Agregar(T valor)
        {
```

```
            if (indice == 9)
                throw new ExcepcionPila("Pila llena");
            vec[indice] = valor;
            indice++;
        }

        public T Sacar()
        {
            if (indice == 0)
                throw new ExcepcionPila("Pila vacía");
            indice--;
            return vec[indice];
        }
    }
}
```

VB

```
Public Class Pila(Of T)
    Private vec(10) As T
    Private indice As Integer = 0

    Public Sub Agregar(valor As T)
        If indice = 9 Then
            Throw New ExcepcionPila("Pila llena")
        End If
        vec(indice) = valor
        indice += 1
    End Sub

    Public Function Sacar() As T
        If indice = 0 Then
            Throw New ExcepcionPila("Pila vacía")
        End If
        indice -= 1
        Return vec(indice)
    End Function
End Class
```

Una clase que utilice a la clase Pila genérica definida anteriormente debe definir los parámetros genéricos apropiadamente como se muestra a continuación.

Ejemplo

C#

```
namespace pila6
{
    class Program
    {
        static void Main(string[] args)
        {
            Pila<double> p = new Pila<double>();

            try
            {
```

```
                p.Agregar(8);
                p.Agregar(8.8);
                p.Agregar(9);

                double total = p.Sacar() + p.Sacar() + p.Sacar();
                Console.WriteLine(total);
                Console.ReadKey();
            }
            catch (ExcepcionPila e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
            catch (Exception e)
            {
                Console.WriteLine(e.Message);
                Console.WriteLine(e.StackTrace);
                Console.ReadKey();
            }
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim p As New Pila(Of Double)()

        Try
            p.Agregar(8) ' Autoboxing
            p.Agregar(8.8000000000000007) ' Autoboxing
            p.Agregar(9) ' Autoboxing

            Dim total As Double = p.Sacar() + p.Sacar() + _
                 p.Sacar()

            Console.WriteLine(total)
            Console.ReadKey()
        Catch e As ExcepcionPila
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        Catch e As Exception
            Console.WriteLine(e.Message)
            Console.WriteLine(e.StackTrace)
            Console.ReadKey()
        End Try
    End Sub
End Module
```

Notar como se simplifica notablemente el código. Cuando se define el objeto cuya referencia es p, del tipo Pila, se utiliza `double` para que no de un error de conversión de tipo cuando se agrega el segundo número a la pila.

Definición de métodos genéricos

Un método genérico es aquel al que se le debe especificar al menos un parámetro de tipo. De esta manera, cuando código cliente utilice el método puede adaptar los tipos de datos con los que funcionará el mismo.

Un método no es genérico simplemente porque se declara en una clase que define un parámetro de tipo genérico. Para serlo basta con que tome por lo menos un parámetro de tipo, además de los parámetros normales que pueda tener definidos. Un método genérico puede utilizar el o los parámetros de tipo junto con los argumentos normales que pueda definir o en su tipo retornado, si lo tiene, y manejarlo en el cuerpo del método.

> ***Es importante tener en cuenta que un método puede ser genérico aunque la clase que lo contenga no lo sea. Es decir, que un método sea genérico no está limitado a pertenecer a una clase genérica. Más aún, los métodos genéricos pueden ser estáticos.***

Para especificar los parámetros de tipo en un método genérico, añadir la notación <T> u (Of T) (C # o VB) después del nombre del método. Se pueden definir tantos parámetros de tipo como sea necesario.

Para invocar un método genérico en el código que lo usa, se declara el nombre del método y se le pasa el o los parámetros de tipo a utilizar por éste. El siguiente ejemplo de código se muestra cómo definir un método genérico y su utilización.

Ejemplo

C#

```
namespace metodos
{
    class Program
    {
        static void Main(string[] args)
        {
            int a = 1;
            int b = 2;

            Console.WriteLine(a + " " + b);
            Intercambiar<int>(ref a, ref b);
            Console.WriteLine(a + " " + b);
            Console.ReadKey();
        }
        public static void Intercambiar<T>(ref T a, ref T b)
        {
            T aux;
```

```
            aux = a;
            a = b;
            b = aux;
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim a As Integer = 1
        Dim b As Integer = 2

        Console.WriteLine(a.ToString + " " + b.ToString)
        Intercambiar(Of Integer)(a, b)
        Console.WriteLine(a.ToString + " " + b.ToString)
        Console.ReadKey()
    End Sub

    Public Sub Intercambiar(Of T)(ByRef a As T, ByRef b As T)
        Dim aux As T
        aux = a
        a = b
        b = aux
    End Sub
End Module
```

Las mismas reglas para la inferencia de tipos (cuando el compilador determina por lógica el tipo al que pertenece una determinada declaración) se aplican tanto a los métodos estáticos como a los métodos de instancia. El compilador puede inferir los parámetros de tipo en base a los argumentos que se pasan al método, no puede deducir los parámetros de tipo a partir sólo de un valor de restricción (lo que se explicará más adelante) o de retorno. Por lo tanto, la inferencia de tipos no funciona con métodos que no tienen parámetros. La inferencia de tipos se realiza en tiempo de compilación antes de que el compilador intente resolver firmas de métodos sobrecargados. El compilador aplica la lógica inferencia de tipos para todos los métodos genéricos que comparten el mismo nombre. En la etapa de resolución de sobrecarga, el compilador incluye sólo aquellos métodos genéricos en los que la inferencia de tipos fue exitosa.

El siguiente ejemplo muestra como el compilador puede inferir tipos cuando se utiliza un método genérico sin especificar el parámetro de tipo y comparándolo cuando se lo declara

Ejemplo

C#

```
namespace inferencia
{
    class MetodoGenerico
    {
        public static void MostrarVector<T>(T[] vector)
```

```
        {
            foreach (T item in vector)
            {
                Console.WriteLine("{0}", item);
            }
        }
    }
}


namespace inferencia
{
    class Program
    {
        static void Main(string[] args)
        {
            // Crear un vector de strings (cadenas).
            string[] amigos = { "Juan", "Sebastián", "Adrián" };
            int[] vec = { 1, 2, 3, 4 };
            // Invocar al método genérico y especificar implícitamente
            // el tipo del parámetro (inferencia)
            MetodoGenerico.MostrarVector(amigos);
            MetodoGenerico.MostrarVector(vec);
            // Invocar al método genérico y especificar explícitamente
            // el tipo del parámetro
            MetodoGenerico.MostrarVector<string>(amigos);
            MetodoGenerico.MostrarVector<int>(vec);
            System.Console.ReadKey();
        }
    }
}
```

VB

```
Public Class MetodoGenerico
    Public Shared Sub MostrarVector(Of T)(ByVal vector As T())
        For Each item As T In vector
            Console.WriteLine("{0}", item)
        Next
    End Sub
End Class

Module Module1

    Sub Main()
        Dim vecNombres() As String = {"Juan", "María", "Pedro"}
        MetodoGenerico.MostrarVector(vecNombres)
        Console.ReadKey()
    End Sub
End Module
```

El programa produce la siguiente salida (notar que es la misma cuando se infiere el tipo que cuando se especifica el parámetro).

```
Juan
Sebastián
Adrián
1
2
3
4
Juan
Sebastián
Adrián
1
2
3
4
```

Si dentro de una clase un método genérico define un parámetro de tipo con el mismo nombre que el de la clase, se oculta el parámetro que pertenece a la clase. Esta es una cuestión simple similar al de visibilidad donde la variable local prevalece siempre respecto de la que no lo es, ampliando el mismo concepto a parámetros de tipo genéricos. Igualmente ante dicha situación el compilador genera una advertencia CS0693, la cual puede ser evitada simplemente renombrando el parámetro de tipo genérico que recibe el método.

Delegados genéricos

Un delegado puede definir sus propios parámetros de tipo. El código que hace referencia al delegado genérico puede especificar el tipo como argumento para crear un tipo cerrado durante la compilación, al igual que al crear instancias de una clase genérica o llamar a un método genérico, como se muestra en el siguiente ejemplo.

Ejemplo

C#

```
namespace delegados
{
    delegate void ElDelegado<T>(T s);

    class Program
    {
        public static void Hola<U>(U s)
        {
            Console.WriteLine("  Hola, {0}!", s);
        }

        public static void Adios<U>(U s)
        {
            Console.WriteLine("  Adiós, {0}!", s);
        }
```

```
        public static void Main()
        {
            ElDelegado<string> a, b, c, d;

            // Crear el objeto a del tipo delegate que referencia
            // al método Hola:
            a = new ElDelegado<string>(Hola<string>);
            // Crear el objeto b del tipo delegate que referencia
            // al método Adios:
            b = new ElDelegado<string>(Adios<string>);
            // Los dos delegates, a y b, se componen para formar c,
            // el cual llama a ambos métodos en orden:
            c = a + b;
            // Remover a del delegate compuesto, dejando sólo b
            // en d, el cual llama al método Adios
            d = c - a;

            Console.WriteLine("Invocando al delegado a:");
            a("A");
            Console.WriteLine("Invocando al delegado b:");
            b("B");
            Console.WriteLine("Invocando al delegado c:");
            c("C");
            Console.WriteLine("Invocando al delegado d:");
            d("D");

            System.Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
    Delegate Sub ElDelegado(Of T)(s As T)

    Public Sub Hola(Of U)(s As U)
        Console.WriteLine("  Hola, {0}!", s)
    End Sub

    Public Sub Adios(Of U)(s As U)
        Console.WriteLine("  Adiós, {0}!", s)
    End Sub

    Sub Main()
        Dim a, b, c, d As ElDelegado(Of String)

        ' Crear el objeto a del tipo delegate que referencia
        ' al método Hola:
        a = New ElDelegado(Of String)(AddressOf Hola)
        ' Crear el objeto b del tipo delegate que referencia
        ' al método Adios:
        b = New ElDelegado(Of String)(AddressOf Adios)
        ' Los dos delegates, a y b, se componen para formar c,
        ' el cual llama a ambos métodos en orden:
        c = [Delegate].Combine(a, b)
```

```
        ' Remover a del delegate compuesto, dejando sólo b
        ' en d, el cual llama al método Adios

        d = [Delegate].Remove(c, a)

        Console.WriteLine("Invocando al delegado a:")
        a("A")
        Console.WriteLine("Invocando al delegado b:")
        b("B")
        Console.WriteLine("Invocando al delegado c:")
        c("C")
        Console.WriteLine("Invocando al delegado d:")
        d("D")
        System.Console.ReadKey()
    End Sub
End Module
```

Los delegados definidos dentro de una clase genérica pueden utilizar los parámetros que se le pasen a esta de la misma manera que lo hacen los métodos. El código que referencie al delegado debe especificar el argumento de tipo de la clase contenedora. En el ejemplo anterior, el delegado fue declarado fuera de una clase y se utilizó sin dependencias de nombres con un tipo especificado para clase alguna, por eso, para evitar confusiones en las declaraciones, notar que los métodos utilizan un nombre distinto para los parámetros de tipo.

Valores por defecto en los tipos genéricos

Nota: si bien los valores por defecto están definidos para C#, `Nothing` cumple la misma función en VB

En las clases y métodos genéricos, una cuestión que se plantea es cómo asignar un valor predeterminado a un tipo parametrizado T cuando no se conoce con antelación lo siguiente:

- Si T será un tipo por referencia o por valor.
- Si T es un tipo por valor, si éste va a ser un valor numérico o una estructura.

Dada una variable t del parámetro de tipo T, la declaración `t = null` sólo es válida si T es un tipo por referencia y `t = 0` sólo funciona para los tipos numéricos, pero no para las estructuras. La solución consiste en utilizar la palabra clave `default`, que retornará `null` para los tipos por referencia y cero para los tipos que sean valores numéricos. Para las estructuras, retornará cada miembro de la estructura inicializado a cero o nulo, dependiendo de si son o no tipos por valor o referencia. Para los tipos por valor que aceptan valores `null`, `default` retorna un System.Nullable<T>, que se inicializa como cualquier estructura.

Ejemplo

C#

```
namespace defecto
{
    // T es el tipo de datos almacenado en una
```

```
    // instancia particular de ListaGenerica.
    public class ListaGenerica<T>
    {
        private class Nodo
        {
            // Cada nodo tiene una referencia
            // al próximo nodo en la lista.
            public Nodo proximo;
            // Cada nodo almacena un valor tde tipo T
            public T dato;
        }

        // La lista se encuentra inicialmente vacía.
        private Nodo cabecera = null;

        // Agregar un nodo al comienzo de la lista
        // usando a t como el valor del dato
        public void AgregarNodo(T t)
        {
            Nodo nuevoNodo = new Nodo();
            nuevoNodo.proximo = cabecera;
            nuevoNodo.dato = t;
            cabecera = nuevoNodo;
        }

        // El siguiente método retorna el valor del dato
        // almacenado en el último nodo de la lista. Si la
        // lista está vacía, se retorna el valor por defecto
        // para el tipo T especificado
        public T ObtenerUltimo()
        {
            // El valor de temp se retorna como el del método.
            // La siguiente declaración inicializa temp con el
            // valor por defecto apropiado para el tipo T. Si la
            // lista está vacía se retorna el valor por defecto.
            T temp = default(T);

            Nodo actual = cabecera;
            while (actual != null)
            {
                temp = actual.dato;
                actual = actual.proximo;
            }
            return temp;
        }
    }
}

namespace defecto
{
    class Program
    {
        static void Main(string[] args)
        {
            // Verificar con una lista NO vacía de enteros
```

```
            ListaGenerica<int> lista1 = new ListaGenerica<int>();
            lista1.AgregarNodo(5);
            lista1.AgregarNodo(4);
            lista1.AgregarNodo(3);
            int valorEntero = lista1.ObtenerUltimo();
            // La siguiente línea de código imprime 5
            System.Console.WriteLine(valorEntero);

            //  Verificar con una lista vacía de enteros
            ListaGenerica<int> lista2 = new ListaGenerica<int>();
            valorEntero = lista2.ObtenerUltimo();
            // La siguiente línea imprime 0.
            System.Console.WriteLine(valorEntero);

            // Verificar con una lista NO vacía de strings.
            ListaGenerica<string> lista3 = new ListaGenerica<string>();
            lista3.AgregarNodo("cinco");
            lista3.AgregarNodo("cuatro");
            string valorCadena = lista3.ObtenerUltimo();
            // La siguiente línea de código imprime cinco.
            System.Console.WriteLine(valorCadena);

            // Verificar con una lista vacía de strings.
            ListaGenerica<string> lisat4 = new ListaGenerica<string>();
            valorCadena = lisat4.ObtenerUltimo();
            // La siguiente línea de código imprime una línea en blanco
            System.Console.WriteLine(valorCadena);
            Console.ReadKey();
        }
    }
}
```

VB

```
' T es el tipo de datos almacenado en una
' instancia particular de ListaGenerica.
Public Class ListaGenerica(Of T)
    Private Class Nodo
        ' Cada nodo tiene una referencia
        ' al próximo nodo en la lista.
        Public proximo As Nodo
        ' Cada nodo almacena un valor tde tipo T
        Public dato As T
    End Class

    ' La lista se encuentra inicialmente vacía.
    Private cabecera As Nodo = Nothing

    ' Agregar un nodo al comienzo de la lista
    ' usando a t como el valor del dato
    Public Sub AgregarNodo(_t As T)
        Dim nuevoNodo = New Nodo()
        nuevoNodo.proximo = cabecera
        nuevoNodo.dato = _t
        cabecera = nuevoNodo
    End Sub
```

```
    ' El siguiente método retorna el valor del dato
    ' almacenado en el último nodo de la lista. Si la
    ' lista esta vacía, se retorna el valor por defecto
    ' para el tipo T especificado
    Public Function ObtenerUltimo() As T
        ' El valor de temp se retorna como el del método.
        ' La siguiente declaración inicializa temp con el
        ' valor por defecto apropiado para el tipo T. Si la
        ' lista está vacía se retorna el valor por defecto.
        Dim temp As T = Nothing 'Equivlente a default en C#
        Dim actual As Nodo = cabecera
        While actual IsNot Nothing
            temp = actual.dato
            actual = actual.proximo
        End While
        Return temp
    End Function
End Class

Module Module1

    Sub Main()
        ' Verificar con una lista NO vacía de enteros
        Dim lista1 As New ListaGenerica(Of Integer)
        lista1.AgregarNodo(5)
        lista1.AgregarNodo(4)
        lista1.AgregarNodo(3)
        Dim valorEntero As Integer = lista1.ObtenerUltimo()
        ' La siguiente línea de código imprime 5
        System.Console.WriteLine(valorEntero)

        '  Verificar con una lista vacía de enteros
        Dim lista2 As New ListaGenerica(Of Integer)
        valorEntero = lista2.ObtenerUltimo()
        ' La siguiente línea imprime 0.
        System.Console.WriteLine(valorEntero)

        ' Verificar con una lista NO vacía de strings.
        Dim lista3 As New ListaGenerica(Of String)()
        lista3.AgregarNodo("cinco")
        lista3.AgregarNodo("cuatro")
        Dim valorCadena As String = lista3.ObtenerUltimo()
        ' La siguiente línea de código imprime cinco.
        System.Console.WriteLine(valorCadena)

        ' Verificar con una lista vacía de strings.
        Dim lisat4 As New ListaGenerica(Of String)()
        valorCadena = lisat4.ObtenerUltimo()
        ' La siguiente línea de código imprime una línea en blanco
        System.Console.WriteLine(valorCadena)
        Console.ReadKey()
    End Sub
End Module
```

Interfaces genéricas

Las interfaces pueden definir tipos genéricos. Esto tiene como consecuencia que la declaración de los métodos en la interfaz deben definir algoritmos en sus métodos independientes de tipo de datos a utilizar, sea por valor o referencia.

Existen muchas interfaces genéricas en el Framework de .Net que son de gran utilidad para asegurar los tipos en el código. Éstas tienen por lo general dos formatos para las cuales están definidas: una con genéricos y otra sin ellos. Existen dos problemas principales con las interfaces que no son definidas con genéricos:

- Trabajan con Object, lo cual implican conversiones de tipo al utilizarlas.
- En ciertos algoritmos, como las clases de colección que se verán posteriormente y que sirven para agrupar objetos en una clase contenedora, generan operaciones de boxing y unboxing

El espacio de nombres donde se definen la mayoría de las interfaces genéricas es System.Collections.Generic. Más adelante se retomará el tema de las interfaces genéricas del Framework de .Net, luego de explicar las restricciones que pueden tener los parámetros de tipo genérico.

Ejemplo

C#

```
interface IParaGenericos<T>
{
        T MiMetodo();
}
```

VB

```
Public Interface IParaGenericos(Of T)
       Function MiMetodo() As T
Interface
```


Restricciones en los tipos que son argumentos genéricos

Cuando se define un tipo genérico, normalmente se trata que sea lo más independiente posible respecto de otros tipos. Sin embargo, a veces es necesario asegurar que los argumentos de tipo son específicos para que suministren, por ejemplo, una cierta capacidad. Para entender esto se puede plantear la necesidad de comparar dos elementos para ordenar o cotejarlos, y se especifica que el argumento genérico que reciba la clase, por ejemplo, debe implementar la interfaz IComparable.

Se puede hacer cumplir este requisito mediante la definición de restricciones en los tipos de los argumentos genéricos que se definen en las clases y que el código cliente está obligado a

especificar para utilizarla. Estas limitaciones se denominan restricciones. Si el código cliente utiliza un tipo que una limitación no permite, el resultado es un error en tiempo de compilación.

La siguiente tabla muestra cómo se deben definir las restricciones.

C#	VB	Descripción
where T : struct	**T As Structure**	El argumento del tipo debe ser un tipo por valor
where T : class	**T As Class**	El argumento del tipo debe ser una referencia, incluyendo cualquier clase, interfaz o vector
where T : new()	**T As New**	El argumento del tipo debe tener un constructor sin parámetros accesible. En C# cuando se especifica esta restricción se coloca al final si hay otras restricciones
where T : class name	**T As class name**	El argumento del tipo debe ser una clase específica o subclase de esta
where T : interface name	**T As interface name**	El argumento del tipo debe ser la interfaz especificada o al menos la debe implementar. Se pueden especificar múltiples interfaces con esta restricción
where T : U	**T As U**	El argumento que se provee para T debe ser U o derivado del argumento provisto para U

Se puede imponer, si es apropiado, más de una restricción sobre un parámetro de tipo. Por ejemplo, puede determinar que un parámetro de tipo debe implementar varias interfaces y que debe proporcionar un constructor sin parámetros. Para imponer más de una restricción, utilizar una lista separada por comas para definir las restricciones. Si se está utilizando Visual Basic, se debe incluir las limitaciones dentro de llaves ({}).

En el siguiente ejemplo se declara una clase Empleado que implementa distintas interfaces (los detalles respecto de la implementación de las diferentes interfaces se explicarán posteriormente) y posee un constructor por defecto (sin argumentos).

Ejemplo

C#

```
namespace restricciones
{
    public class Empleado : IDisposable, IComparable
    {
        private string nombre;
        private string apellido;

        public Empleado()
        {
        }
```

```
public Empleado(string nombre, string apellido)
{
    this.nombre = nombre;
    this.apellido = apellido;
}

public string Apellido
{
    get
    {
        return apellido;
    }
    set
    {
        apellido = value;
    }
}
public string Nombre
{
    get
    {
        return nombre;
    }
    set
    {
        nombre = value;
    }
}
#region IDisposable Members

private bool disposedValue = false;

protected virtual void Dispose(bool disposing)
{
    if (!this.disposedValue)
    {
        if (disposing)
        {
            apellido = null;
            nombre = null;
        }
    }
    this.disposedValue = true;
}

public void Dispose()
{
    Dispose(true);
    GC.SuppressFinalize(this);
}

#endregion
#region IComparable Members
```

```
        public int CompareTo(object obj)
        {
            Empleado e = null;
            if (obj == null) return 1;
            if (obj.GetType() == typeof(Empleado)) e = (Empleado)obj;
            int res1 = this.apellido.CompareTo(e.apellido);
            int res2 = this.nombre.CompareTo(e.nombre);
            if (res1 != 0) return res1;
            else if (res2 != 0) return res2;
            return 0;
        }

        #endregion
        public override string ToString()
        {
            return "[" + apellido + ", " + nombre + "]";
        }
    }
}
```

VB

```
Public Class Empleado : Implements IDisposable, IComparable
    Private _nombre As String
    Private _apellido As String

    Public Sub New()

    End Sub

    Public Sub New(_nombre As String, _apellido As String)
        Me._nombre = _nombre
        Me._apellido = _apellido
    End Sub

    Public Property Nombre() As String
        Get
            Return _nombre
        End Get
        Set(ByVal value As String)
            _nombre = value
        End Set
    End Property

    Public Property Apellido() As String
        Get
            Return _apellido
        End Get
        Set(ByVal value As String)
            _apellido = value
        End Set
    End Property

    Public Overrides Function ToString() As String
        Return "[" + _apellido + ", " + _nombre + "]"
    End Function
```

```
    Public Function CompareTo(ByVal obj As Object) As Integer Implements _
              System.IComparable.CompareTo
        Dim e As Empleado = Nothing
        If (obj Is Nothing) Then Return 1
        If (TypeOf obj Is Empleado) Then e = obj
        Dim res1 As Integer = Me.Apellido.CompareTo(e.Apellido)
        Dim res2 As Integer = Me.Nombre.CompareTo(e.Nombre)
        If (res1 <> 0) Then
            Return res1
        ElseIf (res2 <> 0) Then
            Return res2
        End If
        Return 0
    End Function

    Private disposedValue As Boolean = False

    Protected Overridable Sub Dispose(ByVal disposing As Boolean)
        If Not Me.disposedValue Then
            If disposing Then
                _apellido = Nothing
                _nombre = Nothing
            End If
        End If
        Me.disposedValue = True
    End Sub

#Region " IDisposable Support "
     Public Sub Dispose() Implements IDisposable.Dispose
        Dispose(True)
        GC.SuppressFinalize(Me)
    End Sub
#End Region
End Class
```

La clase ListaDeEmpleados es genérica e impone restricciones sobre los parámetros de tipo que tiene definido. Como dichos parámetros coinciden con la implementación de la clase Empleado, se utiliza un objeto de su tipo como argumento genérico de dicha clase.

Ejemplo

C#

```
namespace restricciones
{
    class ListaDeEmpleados<T>
        where T : Empleado, IDisposable, IComparable, new()
    {
        private T[] vec = new T[5];
        private int indice = 0;
        // Miembros.
        public ListaDeEmpleados()
        {
```

```
        }

        public void AgregaEmpleado(T e)
        {
            if (indice < 10){
                vec[indice] = e;
                indice++;
            }
        }

        public T ObtieneEmpleado(int indice)
        {
            return vec[indice];
        }

        public void Ordenar()
        {
            Array.Sort<T>(vec);
        }
    }
}
```

VB

```
Public Class ListaDeEmpleados(Of T As {Empleado, IDisposable, IComparable, New})
    ' Miembros.
    Private vec(5) As T
    Private indice As Integer = 0

    Public Sub New()
    End Sub

    Public Sub AgregaEmpleado(e As T)
        If indice < 10 Then
            vec(indice) = e
            indice += 1
        End If
    End Sub

    Public Function ObtieneEmpleado(indice As Integer) As T
        Return vec(indice)
    End Function

    Public Sub Ordenar()
        Array.Sort(Of T)(vec)
    End Sub
End Class
```


Para utilizar la clase ListaDeEmpleados se declara el siguiente programa el cual cumple con las restricciones que impone la clase para sus parámetros de tipo y la utiliza.

Ejemplo

C#

```
namespace restricciones
{
    class Program
    {
        static void Main(string[] args)
        {
            ListaDeEmpleados<Empleado> lista = new ListaDeEmpleados<Empleado>();
            Empleado e = new Empleado("Juan", "Perez");
            lista.AgregaEmpleado(e);
            e = new Empleado("Pedro", "García");
            lista.AgregaEmpleado(e);
            e = new Empleado("Marcelo", "Añez");
            lista.AgregaEmpleado(e);
            e = new Empleado("Alejo", "García");
            lista.AgregaEmpleado(e);
            e = new Empleado("Pedro", "García");
            lista.AgregaEmpleado(e);

            lista.Ordenar();

            for (int j = 0; j < 5; j++)
            {
                Console.WriteLine(lista.ObtieneEmpleado(j));
            }
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim lista As New ListaDeEmpleados(Of Empleado)
        Dim e As New Empleado("Juan", "Perez")
        lista.AgregaEmpleado(e)
        e = New Empleado("Pedro", "García")
        lista.AgregaEmpleado(e)
        e = New Empleado("Marcelo", "Añez")
        lista.AgregaEmpleado(e)
        e = New Empleado("Alejo", "García")
        lista.AgregaEmpleado(e)
        e = New Empleado("Pedro", "García")
        lista.AgregaEmpleado(e)

        lista.Ordenar()

        For j As Integer = 1 To 5
            Console.WriteLine(lista.ObtieneEmpleado(j))
        Next
```

```
        Console.ReadKey()
    End Sub

End Module
```

La salida obtenida es una lista ordenada de los elementos que posee el vector que es atributo de la clase ListaDeEmpleados

```
[Añez, Marcelo]
[García, Alejo]
[García, Pedro]
[García, Pedro]
[Perez, Juan]
```

Usando restricciones en los métodos

Los métodos pueden definir sus propias restricciones de la misma manera que pueden definir sus propios parámetros genéricos.

Sin embargo, no se puede declarar a nivel de método limitaciones para los parámetros de tipo genérico de nivel de clase. Todas las restricciones parámetros de tipo genérico de nivel de clase se debe definir en el ámbito de la clase.

Ejemplo

C#

```
namespace metodosRestringidos
{
    public interface IParaGenericos<T>
    {
        T MiMetodo();
    }
}

namespace metodosRestringidos
{
    class MiClase<T>: IParaGenericos<T> where T: new()
    {
        private T t = default(T);
        private static int contador = 0;

        public MiClase()
        {
            contador++;
            Console.WriteLine("Creando la clase MiClase número " + contador);
            t = new T();
        }

        public T MiMetodo()
        {
            Console.WriteLine("T: " + t.GetType().Name);
            return t;
```

```
        }
    }
}

namespace metodosRestringidos
{
    class MiOtraClase<T> : IParaGenericos<T> where T : new()
    {
        private T t = default(T);
        private static int contador = 0;

        public MiOtraClase()
        {
            contador++;
            Console.WriteLine("Creando la clase MiOtraClase número " + contador);
            t = new T();
        }

        public T MiMetodo()
        {
            Console.WriteLine("T: " + t.GetType().Name);
            return t;
        }
    }
}

namespace metodosRestringidos
{
    class Program
    {
        static void Main(string[] args)
        {
            MiClase<MiOtraClase<int>> mc = new MiClase<MiOtraClase<int>>();
            MetodoPolimorfico<MiOtraClase<int>>(mc);
            MiOtraClase<MiClase<int>> moc = new MiOtraClase<MiClase<int>>();
            MetodoPolimorfico<MiClase<int>>(moc);

            IParaGenericos<MiOtraClase<int>> ig = mc;
            mc.MiMetodo();
            Console.ReadKey();
        }

        public static void MetodoPolimorfico<T>(IParaGenericos<T> obj)
              where T: new()
        {
            T o = new T();
            obj.MiMetodo();
        }
    }
}
```

VB

```
Public Interface IParaGenericos(Of T)
    Function MiMetodo() As T
End Interface
```

```
Public Class MiClase(Of T As New)
    Implements IParaGenericos(Of T)

    Private _t As T = Nothing
    Private Shared contador As Integer = 0

    Public Sub New()
        contador += 1
        Console.WriteLine("Creando la clase MiClase número " + contador.ToString)
        _t = New T()
    End Sub

    Public Function MiMetodo() As T Implements IParaGenericos(Of T).MiMetodo
        Console.WriteLine("T: " + _t.GetType().Name)
    End Function
End Class

Public Class MiOtraClase(Of T As New)
    Implements IParaGenericos(Of T)

    Private _t As T = Nothing
    Private Shared contador As Integer = 0

    Public Sub New()
        contador += 1
        Console.WriteLine(
              "Creando la clase MiOtraClase número " + contador.ToString)
        _t = New T()
    End Sub

    Public Function MiMetodo() As T Implements IParaGenericos(Of T).MiMetodo
        Console.WriteLine("T: " + _t.GetType().Name)
    End Function
End Class

Module Module1

    Sub Main()
        Dim mc As New MiClase(Of MiOtraClase(Of Integer))()
        MetodoPolimorfico(Of MiOtraClase(Of Integer))(mc)
        Dim moc As New MiOtraClase(Of MiClase(Of Integer))()
        MetodoPolimorfico(Of MiClase(Of Integer))(moc)

        Dim ig As IParaGenericos(Of MiOtraClase(Of Integer)) = mc
        mc.MiMetodo()
        Console.ReadKey()
    End Sub
    Public Sub MetodoPolimorfico(Of T As New)(obj As IParaGenericos(Of T))
        Dim o As T = New T()
        obj.MiMetodo()
    End Sub
End Module
```

Interfaces genéricas del Framework de .Net

El Framework .NET define varias interfaces que las clases pueden implementar para ofrecer un modelo de contrato a los consumidores de dichas clases.

Existen definiciones de interfaces para que usen genéricos o no

Algunas de las más importantes son:

- IComparable.
- IEquatable.
- IConvertible.
- ICloneable.
- IFormattable.
- IDisposable.

IComparable

La interfaz IComparable permite código cliente comparar dos instancias de una clase o una estructura. Implementar esta interfaz si se desea habilitar a las instancias de una clase o estructura para que sean clasificadas u ordenadas.

Por ejemplo, si desea ordenar los elementos de una matriz mediante el método Array.Sort, los elementos deben ser de un tipo que implementa la interfaz IComparable.

La versión 2.0 del Framework. NET introdujo una versión genérica de la interfaz IComparable que trabaja con objetos fuertemente tipados (esto quiere decir que maneja tipos asegurados). En contraste, la interfaz no genérica IComparable funciona con el tipo System.Object, lo que significa que el código del cliente puede pasar objetos de diferentes tipos de datos. Si se implementa la interfaz no genérica IComparable, se debe probar que los objetos recibidos como argumentos son del mismo tipo y lanzar una excepción ArgumentException si no lo son.

Método	Descripción
CompareTo	Compara la instancia actual con otro objeto del mismo tipo. En la interfaz no genérica IComparable, este método recibe un parámetro de tipo System.Object. En la interfaz genérica IComparable, este método recibe un parámetro de fuertemente tipado. El método devuelve un valor negativo si el primer objeto es lógicamente menor que el segundo objeto, cero si los dos objetos son lógicamente iguales, o un valor positivo si el primer objeto es lógicamente mayor que el segundo objeto.

Ejemplo

C#

```
namespace comparaciones
{
    public class Dinero : IComparable<Dinero>
    {
        private int _pesos, _centavos;

        public Dinero(int pesos, int centavos)
        {
            _pesos = pesos;
            _centavos = centavos;
        }

        public int CompareTo(Dinero otroObj)
        {
            int esteValor = (this._pesos * 100) + this._centavos;
            int otroValor = (otroObj._pesos * 100) + otroObj._centavos;
            return esteValor.CompareTo(otroValor);
        }

        public override String ToString()
        {
            String res = "[pesos = " + _pesos + ", centavos = " + _centavos + "]";
            return res;
        }
    }
}

namespace comparaciones
{
    class Program
    {
        static void Main(string[] args)
        {
            Dinero[] salarios = { new Dinero(100, 0),
                    new Dinero(57, 23),
                    new Dinero(250, 21) };
            Array.Sort(salarios);
            foreach (Dinero d in salarios)
                System.Console.WriteLine(d.ToString());
            System.Console.WriteLine("Comparando");
            Dinero aux = salarios[0];
            foreach (Dinero d in salarios)
            {
                if (aux.CompareTo(d) < 0) aux = d;
            }
            System.Console.WriteLine("El mayor es: " + aux.ToString());
            System.Console.ReadKey();
        }
    }
}
```

VB

```
Public Class Dinero
    Implements IComparable(Of Dinero)
    Private _pesos, _centavos As Integer

    Public Sub New(ByVal pesos As Integer, ByVal centavos As Integer)
        _pesos = pesos
        _centavos = centavos
    End Sub

    Public Function CompareTo(ByVal otroObj As Dinero) As Integer _
    Implements IComparable(Of Dinero).CompareTo
        Dim esteValor As Integer = (Me._pesos * 100) + Me._centavos
        Dim otroValor As Integer = (otroObj._pesos * 100) + otroObj._centavos
        Return esteValor.CompareTo(otroValor)
    End Function

    Public Overrides Function ToString() As String
        Dim res As String
        res = "[pesos = " + CStr(_pesos) + ", centavos = " + CStr(_centavos) + "]"
        Return res
    End Function
End Class

Module Module1

    Sub Main()
        Dim salarios As Dinero() = {New Dinero(100, 0), _
            New Dinero(57, 23), _
            New Dinero(250, 21)}
        Array.Sort(salarios)
        For Each d As Dinero In salarios
            System.Console.WriteLine(d.ToString())
        Next
        System.Console.WriteLine("Comparando")
        Dim aux As Dinero = salarios(0)
        For Each d As Dinero In salarios
            If (aux.CompareTo(d) < 0) Then
                aux = d
            End If
        Next
        System.Console.WriteLine("El mayor es: " + aux.ToString())
        System.Console.ReadKey()
    End Sub
End Module
```


La salida obtenida es

```
[pesos = 57, centavos = 23]
[pesos = 100, centavos = 0]
[pesos = 250, centavos = 21]
Comparando
El mayor es: [pesos = 250, centavos = 21]
```

IEquatable

La interfaz genérica IEquatable permite comparar al código cliente instancias de una clase o estructura por la igualdad. A diferencia del método de Object, esta posee la posibilidad de implementar el parámetro genérico para ser específica y asegurar los tipos. Sin embargo, es recomendable rescribir el método de Object y GetHashCode para evitar confusiones en los llamados realizados por el código cliente que utilice la clase.

Método	Descripción
Equals	Indica si el objeto actual es igual a otro objeto del mismo tipo.

La implementación del método Equals esté destinada a realizar una prueba de igualdad con otro objeto de tipo T, del mismo tipo que el objeto actual. El método Equals se llama en las siguientes circunstancias (tener en cuenta que colecciones se explicará posteriormente, sin embargo se introduce el tema para entender el uso de la interfaz):

- Cuando el método Equals se llama y el parámetro del método representa un objeto fuertemente tipado del tipo T. (Si el argumento es del tipo Object, se llama al método de la clase base, Object.Equals(Object). De los dos métodos, IEquatable <T>.Equals ofrece un rendimiento ligeramente mejor.)
- Cuando los métodos de búsqueda de un número de colecciones de objetos genéricos son llamados. Algunos de estos tipos y sus métodos incluyen los siguientes:
    - Algunas de las sobrecargas genéricas del método BinarySearch.
    - Los métodos de búsqueda de la clase List <T>, incluidos List <T>.Contains (T), List<T>.IndexOf, List<T>.LastIndexOf, y List<T>.Remove.
    - Los métodos de búsqueda de la clase Dictionary <TKey, TValue>, incluyendo ContainsKey y Remove.
    - Los métodos de búsqueda de la clase genérica LinkedList <T>, incluyendo LinkedList <T>.Contains y LinkedList <T>.Remove.

En otras palabras, para gestionar la posibilidad de que objetos de una clase se almacenan en un vector o un objeto de colección genérico, es una buena idea para implementar IEquatable <T> de modo que el objeto puede ser fácilmente identificado y manipulado.

Al aplicar el método Equals, definir el parámetro de tipo apropiadamente para el tipo especificado por el argumento genérico. Por ejemplo, si el argumento es de tipo Int32, definir la igualdad apropiadamente para la comparación de dos enteros de 32 bits con signo.

Ejemplo

C#

```
namespace igualdades
{
    public class Producto : IEquatable<Producto>
```

```
    {
        private int _id;
        private string _nombre;
        private double _precio;

        public Producto(int id, string nombre, double precio)
        {
            _id = id;
            _nombre = nombre;
            _precio = precio;
        }

        public bool Equals(Producto otroObj)
        {
            return (this._id == otroObj._id) &&
            (this._nombre == otroObj._nombre) &&
            (this._precio == otroObj._precio);
        }
    }
}



namespace igualdades
{
    class Program
    {
        static void Main(string[] args)
        {
            Producto p1 = new Producto(1, "Chai", 18.00);
            Producto p2 = new Producto(1, "Chai", 18.00);
            if (p1.Equals(p2))
                Console.WriteLine("Los objetos son iguales.");
            else
                Console.WriteLine("Los objetos son diferentes.");
            System.Console.ReadKey();
        }
    }
}
```

VB

```
Public Class Producto
    Implements IEquatable(Of Producto)
    Private _id As Integer
    Private _nombre As String
    Private _precio As Double

    Public Sub New(ByVal id As Integer, ByVal nombre As String, _
              ByVal precio As Double)
        _id = id
        _nombre = nombre
        _precio = precio
    End Sub

    Public Overloads Function Equals(ByVal otroObj As Producto) As Boolean _
```

```
    Implements IEquatable(Of Producto).Equals
        Return (Me._id = otroObj._id) AndAlso _
        (Me._nombre = otroObj._nombre) AndAlso _
        (Me._precio = otroObj._precio)
    End Function
End Class

Module Module1

    Sub Main()
        Dim p1 As New Producto(1, "Chai", 18.0)
        Dim p2 As New Producto(1, "Chai", 18.0)
        If p1.Equals(p2) Then
            Console.WriteLine("Los objetos son iguales.")
        Else
            Console.WriteLine("Los objetos son diferentes.")
        End If
        System.Console.ReadKey()
    End Sub
End Module
```

IConvertible

La interfaz IConvertible define métodos que convierten el valor de una clase o estructura en una serie de diferentes tipos de datos. Loa tipos soportados son los que se muestran en la siguiente tabla.

Método	Descripción
GetTypeCode	Devuelve la enumeración TypeCode para esta instancia.
ToBoolean	Convierte el valor de esta instancia en un valor booleano equivalente.
ToSByte, ToByte	Convierte el valor de esta instancia en un equivalente entero de 8 bits con o sin signo.
ToInt16, ToUInt16, ToInt32, ToUInt32, ToInt64, ToUInt64	Convierte el valor de esta instancia en un equivalente de 16 bits, 32 bits o 64 bits con o sin signo.
ToSingle, ToDouble	Convierte el valor de esta instancia en un número de punto flotante equivalente de simple o doble precisión.
ToDecimal	Convierte el valor de esta instancia a un valor Decimal equivalente.
ToDateTime	Convierte el valor de esta instancia con un valor equivalente DateTime.
ToChar, ToString	Convierte el valor de esta instancia en un valor Unicode equivalente de caracter o cadena.
ToType	Convierte el valor de esta instancia en un objeto del tipo especificado que tiene un valor equivalente.

Ejemplo

C#

```
namespace conversiones
{
    public class Dinero : IConvertible
    {
        private int _pesos, _centavos;

        public Dinero(int pesos, int centavos)
        {
            _pesos = pesos;
            _centavos = centavos;
        }

        public override String ToString()
        {
            String res = "[pesos = " + _pesos + ", centavos = " + _centavos + "]";
            return res;
        }

        // Indica que el TypeCode (tipo de un objeto) para Dinero
        // es TypeCode.Object.
        public TypeCode GetTypeCode()
        {
            return TypeCode.Object;
        }
        // Propiedad de ayuda utilizada por los métodos de conversión.
        private double Valor
        {
            get { return _pesos + (_centavos / 100.0); }
        }
        // Propiedad de ayuda utilizada por los métodos de conversión.
        private double Redondeo
        {
            get {
                int p = _pesos;
                if (_centavos >= 50) p++;
                return p;
            }
        }

        public bool ToBoolean(IFormatProvider proveedorFormato)
        {
            if (_pesos != 0 || _centavos != 0)
                return true;
            else
                return false;
        }

        // La mayoría de los métodos de conversión son variaciones de este.
        public byte ToByte(IFormatProvider proveedorFormato)
        {
            // Convert convierte un tipo de datos base a otro
```

```
        return Convert.ToByte(Valor);
    }

    // No tiene significado esta conversión, entonces lanza la excepción.
    public DateTime ToDateTime(IFormatProvider proveedorFormato)
    {
        throw new InvalidCastException(
          "Conversión inválida. No se puede tranformar en DateTime");
    }
    // Retorna una cadena especifica de la localidad formateada
    // como moneda en curso.
    public string ToString(IFormatProvider proveedorFormato)
    {
        return String.Format(proveedorFormato, "{0:c}", Valor);
    }
    // Convierte un objeto del tipo Dinero a un tipo especifico.
    public object ToType(Type tipoDeConversion,
          IFormatProvider proveedorFormato)
    {
        // Convert convierte un tipo de datos base a otro
        return Convert.ChangeType(Valor, tipoDeConversion);
    }

    public sbyte ToSByte(IFormatProvider proveedorFormato)
    {
        if (!(Valor <= 127.0))
        {
            throw new InvalidCastException(
                "Conversión inválida. Valor muy grande");
        }

        return Convert.ToSByte(Redondeo);
    }

    public char ToChar(IFormatProvider proveedorFormato)
    {
        // Convert convierte un tipo de datos base a otro
        return Convert.ToChar(Redondeo);
    }

    public decimal ToDecimal(IFormatProvider proveedorFormato)
    {
        // Convert convierte un tipo de datos base a otro
        return Convert.ToDecimal(Valor);
    }
    public float ToSingle(IFormatProvider proveedorFormato)
    {
        // Convert convierte un tipo de datos base a otro
        return Convert.ToSingle(Valor);
    }

    public double ToDouble(IFormatProvider proveedorFormato)
    {
        // Convert convierte un tipo de datos base a otro
        return Convert.ToDouble(Valor);
```

```
        }

        public short ToInt16(IFormatProvider proveedor)
        {
            // Convert convierte un tipo de datos base a otro
            return Convert.ToInt16(Redondeo);
        }

        public ushort ToUInt16(IFormatProvider proveedor)
        {
            // Convert convierte un tipo de datos base a otro
            return Convert.ToUInt16(Redondeo);
        }

        public int ToInt32(IFormatProvider proveedor)
        {
            // Convert convierte un tipo de datos base a otro
            return Convert.ToInt32(Redondeo);
        }

        public uint ToUInt32(IFormatProvider proveedor)
        {
            // Convert convierte un tipo de datos base a otro
            return Convert.ToUInt32(Redondeo);
        }

        public long ToInt64(IFormatProvider proveedor)
        {
            // Convert convierte un tipo de datos base a otro
            return Convert.ToInt64(Redondeo);
        }

        public ulong ToUInt64(IFormatProvider proveedor)
        {
            // Convert convierte un tipo de datos base a otro
            return Convert.ToUInt64(Redondeo);
        }
    }
}

namespace conversiones
{
    class Program
    {
        static void Main(string[] args)
        {
            Dinero dinero1 = new Dinero(10, 50);

            try
            {
                // Convierte el dinero a double.
                double d = Convert.ToDouble(dinero1);
                // Convierte el dinero a String.
                string s = Convert.ToString(dinero1);
                // Convierte el dinero a sbyte.
```

```
                sbyte sb1 = Convert.ToSByte(dinero1);
                float f = Convert.ToSingle(dinero1); // Convierte el dinero a float.
                int i = Convert.ToInt32(dinero1); // Convierte el dinero a int.

                System.Console.WriteLine("Dinero: {0}", d);
                System.Console.WriteLine("Dinero: {0}", s);
                System.Console.WriteLine("Dinero: {0}", sb1);
                System.Console.WriteLine("Dinero: {0}", f);
                System.Console.WriteLine("Dinero: {0}", i);

                DateTime dt = Convert.ToDateTime(dinero1); // Causa una excepción.
            }
            catch (InvalidCastException e)
            {
                System.Console.WriteLine("Error: {0} \nPresione una tecla ...",
                     e.Message);
                System.Console.ReadKey();
                System.Console.WriteLine("");
            }
            finally
            {
                try
                {
                    Dinero dinero2 = new Dinero(127, 50);
                    // Convierte el dinero a sbyte.
                    sbyte sb2 = Convert.ToSByte(dinero2);
                    System.Console.WriteLine("Dinero: {0}", sb2);
                    System.Console.ReadKey();
                }
                catch (InvalidCastException e)
                {
                    System.Console.WriteLine(
                    "Programa Terminado. Error: {0} \nPresione una tecla ...",
                            e.Message);
                    System.Console.ReadKey();
                }
            }
        }
    }
}
```

VB

```
Public Class Dinero
    Implements IConvertible
    Private _pesos, _centavos As Integer

    Public Sub New(ByVal pesos As Integer, ByVal centavos As Integer)
        _pesos = pesos
        _centavos = centavos
    End Sub
    ' Indica que el TypeCode (tipo de un objeto) para Dinero es TypeCode.Object.
    Public Function GetTypeCode() As TypeCode Implements IConvertible.GetTypeCode
        Return TypeCode.Object
    End Function
    ' Propiedad de ayuda utilizada por los métodos de conversión.
```

```
Private ReadOnly Property Valor() As Double
    Get
        Return _pesos + (_centavos / 100.0)
    End Get
End Property

' Propiedad de ayuda utilizada por los métodos de conversión.
Private ReadOnly Property Redondeo() As Double
    Get
Dim p As Integer = _pesos
        If _centavos >= 50 Then
            p = p + 1
        End If
        Redondeo = p
    End Get
End Property

Public Function ToBoolean(ByVal proveedorFormato As IFormatProvider) _
   As Boolean Implements IConvertible.ToBoolean
    If _pesos <> 0 OrElse _centavos <> 0 Then
        Return True
    Else
        Return False
    End If
End Function
' La mayoría de los métodos de conversión son variaciones de este.
Public Function ToByte(ByVal proveedorFormato As IFormatProvider) As Byte _
   Implements IConvertible.ToByte
    ' Convert convierte un tipo de datos base a otro
    Return Convert.ToByte(Valor)
End Function
' No tiene significado esta conversión, entonces lanza la excepción.
Public Function ToDateTime(ByVal proveedorFormato As IFormatProvider) _
   As DateTime Implements IConvertible.ToDateTime
    Throw New InvalidCastException("Conversión inválida.")
End Function
' Retorna una cadena especifica de la localidad formateada como moneda en curso.
Public Overloads Function ToString(ByVal proveedorFormato As IFormatProvider) _
   As String Implements IConvertible.ToString
    Return String.Format(proveedorFormato, "{0:c}", Valor)
End Function
' Convierte un objeto del tipo Dinero a un tipo específico.
Public Function ToType(ByVal tipoDeConversion As Type, _
ByVal provider As IFormatProvider) As Object _
   Implements IConvertible.ToType
    ' Convert convierte un tipo de datos base a otro
    Return Convert.ChangeType(Valor, tipoDeConversion)
End Function
Public Function ToSByte(ByVal proveedorFormato As IFormatProvider) _
   As SByte Implements IConvertible.ToSByte
    ' Convert convierte un tipo de datos base a otro
    If Not (Valor <= 127.0) Then
        Throw New InvalidCastException("Conversión inválida. Valor muy grande")
    End If
```

```
        Return Convert.ToSByte(Redondeo)
    End Function
    Public Function ToChar(ByVal proveedorFormato As IFormatProvider) _
       As Char Implements IConvertible.ToChar
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToChar(Redondeo)
    End Function

    Public Function ToDecimal(ByVal proveedorFormato As IFormatProvider) _
       As Decimal Implements IConvertible.ToDecimal
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToDecimal(Valor)
    End Function
    Public Function ToSingle(ByVal proveedorFormato As IFormatProvider) _
       As Single Implements IConvertible.ToSingle
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToSingle(Valor)
    End Function
    Public Function ToDouble(ByVal proveedorFormato As IFormatProvider) _
       As Double Implements IConvertible.ToDouble
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToDouble(Valor)
    End Function
    Public Function ToInt16(ByVal proveedor As IFormatProvider) _
       As Short Implements IConvertible.ToInt16
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToInt16(Redondeo)
    End Function
    Public Function ToUInt16(ByVal proveedor As IFormatProvider) _
       As UShort Implements IConvertible.ToUInt16
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToUInt16(Redondeo)
    End Function
    Public Function ToInt32(ByVal proveedor As IFormatProvider) _
       As Integer Implements IConvertible.ToInt32
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToSByte(Redondeo)
    End Function
    Public Function ToUInt32(ByVal proveedor As IFormatProvider) _
       As UInteger Implements IConvertible.ToUInt32
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToUInt32(Redondeo)
    End Function
    Public Function ToInt64(ByVal proveedor As IFormatProvider) _
       As Long Implements IConvertible.ToInt64
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToInt64(Redondeo)
    End Function
    Public Function ToUInt64(ByVal proveedor As IFormatProvider) _
       As ULong Implements IConvertible.ToUInt64
        ' Convert convierte un tipo de datos base a otro
        Return Convert.ToUInt64(Redondeo)
    End Function
End Class
```

```
Module Module1

    Sub Main()
        Dim dinero1 As New Dinero(10, 50)
        Dim d As Double = Convert.ToDouble(dinero1) ' Convierte el dinero a Double.
        Dim s As String = Convert.ToString(dinero1) ' Convierte el dinero a String.

        System.Console.WriteLine("Dinero: {0}", d)
        System.Console.WriteLine("Dinero: {0}", s)

        Try
            Dim dt As DateTime = Convert.ToDateTime(dinero1) ' Causa una excepción.
            Exit Try

        Catch e As InvalidCastException
            System.Console.WriteLine(
              "Programa Terminado. Error: {0} \nPresione una tecla ...", e.Message)
            System.Console.ReadKey()
            Exit Try
        Finally
            Try
                Dim dinero2 As New Dinero(127, 50)
                ' Convierte el dinero a sbyte.
                Dim sb2 As SByte = Convert.ToSByte(dinero2)
                System.Console.WriteLine("Dinero: {0}", sb2)
                System.Console.ReadKey()
                Exit Try
            Catch e As InvalidCastException
                System.Console.WriteLine(
                    "Programa Terminado. Error: {0}. Presione una tecla ...",
                    e.Message)
                System.Console.ReadKey()
                Exit Try
            End Try
        End Try
    End Sub
End Module
```


ICloneable

La interfaz ICloneable permite al código cliente crear una nueva instancia de una clase con el mismo valor que una instancia existente. Implementar esta interfaz en las clases que requieren un comportamiento de copia en profundidad. Esto quiere decir que si la clase almacena en cu interior un atributo que es una referencia a un objeto, copiar la referencia a dicho objeto sería una copia superficial, puesto que no se “clonaría” realmenter el objeto. En cambio, si sobre ese atributo se crea un nuevo objeto, en una locación de memoria diferente, y se lo inicializa con los valores que almacena al momento de la clonación, el objeto que se mandó a clonar inicialmente (el que contiene al atributo que es una referencia a un objeto), la clonación se realizará en profundidad.

Si la clonación es superficial, los cambios en los atributos que sean referencias se verán en ambos objetos luego de la clonación. Si en cambio es en profundidad, como todos los objetos son

diferentes y no hay una referencia al mismo objeto, los cambios son independientes a partir del momento de la clonación.

Si sólo desea obtener una copia superficial de un objeto, no es necesario implementar la interfaz ICloneable. En su lugar, invocar en un objeto el método MemberwiseClone.

Método	Descripción
Clone	Crea un nuevo objeto que es una copia en profundidad de la instancia actual.

Ejemplo

C#

```
namespace clonacion
{
    class DetallesPersonales : ICloneable
    {
        private string _nombre;
        private List<string> _numerosContactos;
        public DetallesPersonales(string n)
        {
            _nombre = n;
            _numerosContactos = new List<string>();
        }
        public void AgregarNumeroContacto(string numero)
        {
            _numerosContactos.Add(numero);
        }
        public object Clone()
        {
            DetallesPersonales objectoClonado = new DetallesPersonales(_nombre);
            foreach (string num in _numerosContactos)
            {
                objectoClonado._numerosContactos.Add(num);
            }
            return objectoClonado;
        }
        public List<string> NumerosContactos
        {
            get
            {
                return _numerosContactos;
            }
        }
    }
}

namespace clonacion
{
    class Program
    {
        static void Main(string[] args)
        {
```

```
            DetallesPersonales detallesOriginales = new DetallesPersonales("Ruby");
            detallesOriginales.AgregarNumeroContacto("Número1");
            detallesOriginales.AgregarNumeroContacto("Número2");
            DetallesPersonales detallesClonados =
                (DetallesPersonales)detallesOriginales.Clone();
            // Agregar un contacto al objeto original.
            detallesOriginales.AgregarNumeroContacto("Número3");
            // Agregar un contacto al objeto clonado.
            detallesClonados.AgregarNumeroContacto("Número4");
            List<string> lista1 = detallesOriginales.NumerosContactos;
            List<string> lista2 = detallesClonados.NumerosContactos;

            Console.WriteLine("Imprimiendo lista original");
            foreach (string s in lista1)
            {
                Console.WriteLine(s);
            }

            Console.WriteLine("Imprimiendo lista clonada");
            foreach (string s in lista2)
            {
                Console.WriteLine(s);
            }

            Console.ReadKey();
        }
    }
}
```

VB

```
Public Class DetallesPersonales
    Implements ICloneable
    Private _nombre As String
    Private _numerosContactos As List(Of String)
    Public Sub New(ByVal n As String)
        _nombre = n
        _numerosContactos = New List(Of String)()
    End Sub
    Public Sub AgregarNumeroContacto(ByVal numero As String)
        _numerosContactos.Add(numero)
    End Sub
    Public Function Clone() As Object Implements ICloneable.Clone
        Dim objectoClonado As New DetallesPersonales(_nombre)
        For Each num As String In _numerosContactos
            objectoClonado._numerosContactos.Add(num)
        Next
        Return objectoClonado
    End Function
    Public ReadOnly Property NumerosContactos() As List(Of String)
        Get
            Return _numerosContactos
        End Get
    End Property
End Class
```

```
Module Module1

    Sub Main()
        Dim detallesOriginales As New DetallesPersonales("Ruby")
        detallesOriginales.AgregarNumeroContacto("Número1")
        detallesOriginales.AgregarNumeroContacto("Número2")
        Dim detallesClonados As DetallesPersonales = _
        CType(detallesOriginales.Clone(), DetallesPersonales)
        ' Agregar un contacto al objeto original.
        detallesOriginales.AgregarNumeroContacto("Número3")
        ' Agregar un contacto al objeto clonado.
        detallesClonados.AgregarNumeroContacto("Número4")

        Dim lista1 As List(Of String) = detallesOriginales.NumerosContactos
        Dim lista2 As List(Of String) = detallesClonados.NumerosContactos

        Console.WriteLine("Imprimiendo lista original")
        For Each s As String In lista1
            Console.WriteLine(s)
        Next

        Console.WriteLine("Imprimiendo lista clonada")
        For Each s As String In lista2
            Console.WriteLine(s)
        Next
        Console.ReadKey()
    End Sub
End Module
```


IFormattable

La interfaz IFormattable permite al código cliente dar formato al valor de un objeto en una representación de cadena.

Implementar esta interfaz en clases o estructuras que se desee dar formato como cadenas en métodos tales como Console.WriteLine y String.Format.

Método	Descripción
ToString	Formatea el valor de la instancia actual utilizando el formato especificado. El método recibe dos argumentos: una cadena de formato y un objeto del tipo IFormatProvider.

El ejemplo de código siguiente muestra cómo implementar la interfaz IFormattable. La estructura Punto implementa el método ToString de la interfaz IFormattable de la siguiente manera:

- Si la cadena de formato es una referencia nula, retornar las coordenadas x e y de la estructura del tipo Punto.
- Si la cadena de formato es "x", retornar la coordenada x del objeto Punto.
- Si la cadena de formato es "y", retornar la coordenada y del objeto Punto.

- Si la cadena de formato es otra cosa, lanzar una FormatException para indicar que la cadena de formato no es válida.

Ejemplo

C#

```
namespace formateo
{
    public struct Punto : IFormattable
    {
        private int _x, _y;

        public Punto(int x, int y)
        {
            _x = x;
            _y = y;
        }

        // Sobrescribe el método ToString de la clase System.Object.
        public override string ToString()
        {
            return ToString(null, null);
        }
        // Implementa el método ToString de la interfaz IFormattable.
        public String ToString(string formato, IFormatProvider fp)
        {
            try
            {
                if (formato == null)
                    return String.Format("({0}, {1})", _x, _y);
                else if (formato == "x")
                    return _x.ToString();
                else if (formato == "y")
                    return _y.ToString();
                else
                    throw new FormatException(
                    String.Format("Formato inválido de string: '{0}'.", formato));

            }
            catch (FormatException fe)
            {
                Console.WriteLine("Error: " + fe.Message);
            }
            return "Error en " + formato + " para " + fp.ToString();
        }
    }
    class Program
    {
        static void Main(string[] args)
        {
            Punto p = new Punto(10, 20);
            Console.WriteLine("Esto imprime la coordenada X: {0:x}", p);
            Console.WriteLine("Esto imprime la coordenada Y: {0:y}", p);
            Console.WriteLine("Esto imprime las coordenadas del punto: {0}", p);
```

```
            Console.WriteLine("Esto causa una FormatException: {0:z}", p);

            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
    Public Structure Point
        Implements IFormattable
        Private _x, _y As Integer

        Public Sub New(ByVal x As Integer, ByVal y As Integer)
            _x = x
            _y = y
        End Sub
        ' Sobrescribe el método ToString de la clase System.Object.
        Public Overrides Function ToString() As String
            Return ToString(Nothing, Nothing)
        End Function
        ' Implementa el método ToString de la interfaz IFormattable.
        Public Overloads Function ToString( _
        ByVal formato As String, ByVal fp As IFormatProvider) _
        As String Implements IFormattable.ToString
            Try
                If formato Is Nothing Then
                    Return String.Format("({0}, {1})", _x, _y)
                ElseIf formato = "x" Then
                    Return _x.ToString()
                ElseIf formato = "y" Then
                    Return _y.ToString()
                Else
                    Throw New FormatException( _
                    String.Format("Formato inválido de string: '{0}'.", formato))
                End If
            Catch fe As FormatException
                Console.WriteLine("Error: " + fe.Message)
            End Try
            Return "Error en " + formato + " para " + fp.ToString()
        End Function
    End Structure

    Sub Main()
        Dim p As New Point(10, 20)
        Console.WriteLine("Esto imprime la coordenada X: {0:x}", p)
        Console.WriteLine("Esto imprime la coordenada Y: {0:y}", p)
        Console.WriteLine("Esto imprime las coordenadas del punto: {0}", p)
        Console.WriteLine("Esto causa una FormatException: {0:z}", p)

        Console.ReadKey()
    End Sub
End Module
```

IDisposable

Si un objeto utiliza un conjunto específico de recursos, puede liberar estos recursos mediante el uso de un destructor. El recolector de basura ejecuta el destructor cuando libera la memoria que ocupa el objeto.

Sin embargo, algunos recursos son demasiado valiosos para esperar a que el recolector de basura los libere. Los recursos escasos deben libertarse tan pronto como sea posible. En estos casos, la única opción es liberar el recurso en el código en el momento que estos no sean más necesarios. Cuando hay recursos que son necesarios liberarlos rápidamente no se puede esperar a que en un tiempo no determinado el GC llame al destructor

Para liberar los recursos rápidamente, se puede utilizar esta interfaz y rescribir el método Dispose

Implementar la interfaz IDisposable

Cuando se escribe una clase, ¿qué es lo mejor, escribir un destructor o implementar la interfaz IDisposable? Una llamada a un destructor pasará, pero no sé sabe cuándo. Por el contrario, se sabe exactamente cuando sucede una llamada al método Dispose. Es posible asegurar que el método Dispose siempre se ejecuta se llame o no desde el destructor. Esta actúa como una copia de seguridad útil. Es posible que se olvide de llamar al método Dispose, pero al menos se puede estar seguro de que se va a llamar, incluso si sólo sucede cuando el programa se cierra. Los ejemplos de código siguientes muestran esto.

Ejemplo

C#

```
namespace disponer
{
    public class ContenedorDelRecurso : IDisposable
    {
        private bool _dispuesto = false;
        public void Dispose()
        {
            // El parámetro True indica que
            // el objeto es eliminado a causa de una llamada explícita
            // al método público Dispose en lugar que el CLR
            // invoque código de finalización en el objeto.
            Dispose(true);

            // quitar el objeto de la cola de finalización de CLR.
            // Este método frena al recolector de basura para
            // que no llame al destructor de este objeto,
            // porque el mismo ha sido finalizado.
            GC.SuppressFinalize(this);
        }
        protected virtual void Dispose(bool disponiendo)
        {
            if (!_dispuesto)
            {
```

```
                // Si se llamó desde este objeto (disponiendo=true),
                // disponer los recursos manejados y los No manejados.
                // Sino sólo los últimos porque lo llamó el GC por
                // medio del método Finalize y ya eliminó los primeros.
                if (disponiendo)
                {
                    // Disponer los recursos manejados acá.
                }
                // Disponer los recursos No manejados acá (Por ejemplo,
                // si se intanció un objeto COM o ActiveX no codificado
                // con .Net). Fijar el valor de la bandera _dispuesto
                // en true para evitar futuras disposiciones
                _dispuesto = true;
            }
        }
        // Código de finalización.
        ~ContenedorDelRecurso()
        {
            Dispose(false);
        }
    }
}
```

VB

```
Public Class ContenedorDelRecurso
    Implements IDisposable
    Private _dispuesto As Boolean = False
    Public Sub Dispose() Implements IDisposable.Dispose
        ' El parámetro True indica que
        ' el objeto es eliminado a causa de una llamada explícita
        ' al método público Dispose en lugar que el CLR
        ' invoque código de finalización en el objeto.
        Dispose(True)
        ' quitar el objeto de la cola de finalización de CLR.
        ' Este método frena al recolector de basura para
        ' que no llame al destructor de este objeto,
        ' porque el mismo ha sido finalizado.
        GC.SuppressFinalize(Me)
    End Sub
    Protected Overridable Sub Dispose(ByVal disponiendo As Boolean)
        If Not _dispuesto Then
            ' Si se llamó desde este objeto (disponiendo=true),
            ' disponer los recursos manejados y los No manejados.
            ' Sino sólo los últimos porque lo llamó el GC por
            ' medio del método Finalize y ya eliminó los primeros.
            If disponiendo Then
                ' Disponer los recursos manejados acá.
            End If
            ' Disponer los recursos No manejados acá (Por ejemplo,
            ' si se intanció un objeto COM o ActiveX no codificado
            ' con .Net). Fijar el valor de la bandera _dispuesto
            ' en true para evitar futuras disposiciones
            _dispuesto = True
        End If
    End Sub
```

```
    ' Código de finalización (actúa como un destructor).
    Protected Overrides Sub Finalize()
        Dispose(False)
        MyBase.Finalize()
    End Sub
End Class
```

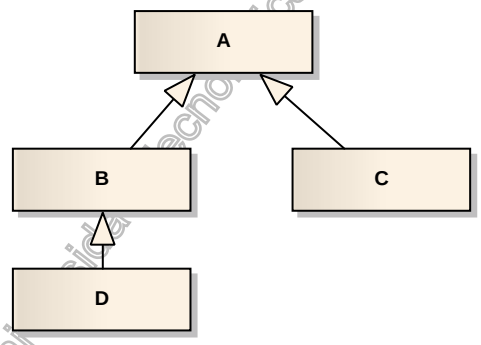
Se debe notar que VB no maneja el concepto de destructor, sin embargo posee una forma de finalizar el código en memoria a través de un método de Object, Finalize. Por lo tanto, se sobrescribe dicho método para realizar las tareas de liberación de recursos. Este método es invocado también en C# internamente por el destructor.

Invariancia

El uso de genéricos asegura el tipo. Es más, su principal objetivo es asegurar el tipo utilizado por un determinado objeto. Pero, como se puede asignar cualquier tipo al parámetro genérico, ¿qué pasa si se crea una referencia a la que se asigna un parámetro de tipo cuando se crea un objeto y luego se trata de asignar a una referencia de ese tipo otro objeto que se creó con un parámetro diferente?

Es claro que nunca estos objetos pueden ser “exactamente del mismo tipo” porque el compilador asignó diferentes tipos al parámetro genérico y es como si con declaraciones estándar se hubiesen creado dos objetos diferentes. Al mecanismo que asegura el tipo de esta forma se lo llama invariancia y ésta es su principal función.

Suponiendo las cadenas de herencia que muestra el gráfico, se van a utilizar como parámetros de



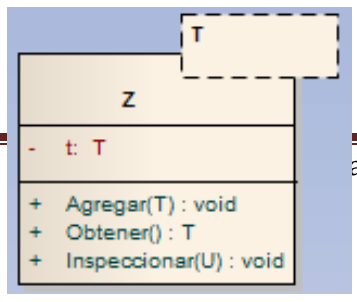
tipo en una clase genérica Z:

El código de la clase Z es el siguiente

Ejemplo

C#

```
namespace invariancia
{
```



```
    class Z<T>
    {
        private T t;

        public void Agregar(T t)
        {
            this.t = t;
        }

        public T Obtener()
        {
            return t;
        }

        public void Inspeccionar<U>(U u)
        {
            Console.WriteLine("T: " + t.GetType().Name);
            Console.WriteLine("U: " + u.GetType().Name);
        }
    }
}
```

VB

```
Class Z(Of T)
    Private _t As T

    Public Sub Agregar(_t As T)
        Me._t = _t
    End Sub

    Public Function Obtener() As T
        Return _t
    End Function

    Public Sub Inspeccionar(Of U)(_u As U)
        Console.WriteLine("T: " + _t.GetType().Name)
        Console.WriteLine("U: " + _u.GetType().Name)
    End Sub
End Class
```

Al tratar de utilizar Z con las clases de las cadenas de herencia que muestra el gráfico anterior, se puede observar el problema que detecta el compilador al tratar de asignar tipos incompatibles.

Ejemplo

C#

```
namespace invariancia
{
    class Program
    {
        static void Main(string[] args)
        {
            Z<A> z1 = new Z<A>();
```

```
            Z<B> z2 = new Z<B>();
            Z<C> z3 = new Z<C>();

            //z1 = z2; // ERROR
            //z2 = z3; // ERROR
            Console.WriteLine("Todas las declaraciones son correctas");
            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim z1 As New Z(Of A)()
        Dim z2 As New Z(Of B)()
        Dim z3 As New Z(Of C)()

        'z1 = z2 ' ERROR
        'z2 = z3 ' ERROR
        Console.WriteLine("Todas las declaraciones son correctas")
        Console.ReadKey()
    End Sub
End Module
```

Las asignaciones marcadas como comentarios comenten exactamente ese error. En el primer caso z1 se creó con un parámetro genérico del tipo A, por lo tanto intentar asignar un objeto que se creó con un parámetro del tipo B es incompatible. El error que se registra es el siguiente:

> Cannot implicitly convert type 'invariancia.Z<invariancia.B>' to 'invariancia.Z<invariancia.A>'

El segundo error es similar, lo que cambia son las clases con lo que se intenta hacer el mismo tipo de conversión no admitida entre referencias. El error en este caso es el siguiente:

> Cannot implicitly convert type 'invariancia.Z<invariancia.C>' to 'invariancia.Z<invariancia.B>'

Covarianza

Se define a la covarianza de la siguiente manera

> *Cuando en las asignaciones se preserva el orden natural de los subtipos respecto a sus súper tipos en una asignación (de lo más específico a lo más genérico) la asignación es covariante*

Para ampliar un poco el concepto, se puede suponer la misma cadena de herencia, pero restringiendo ahora los parámetros de tipo para definir asignaciones que puedan ser compatibles según los parámetros de tipos utilizados.

Si se tratan de establecer los límites en un método de una clase Z para que utilice estas dos cadenas de herencia se puede optar por poner a la clase A como restricción superior y ver como reaccionaría un programa al intentar utilizar los diferentes tipos de objetos.

La clase Z puede tener definidos a la vez parámetros de tipo, pero se restringe el parámetro en el método Inspeccionar para verificar las asignaciones de manera de generalizar el ejemplo expuesto anteriormente

Ejemplo

C#

```
namespace limitaciones
{
    class Z<T> where T : A
    {
        private T t;

        public void Agregar(T t)
        {
            this.t = t;
        }

        public T Obtener()
        {
            return t;
        }

        public void Inspeccionar<U>(U u) where U : A
        {
            Console.WriteLine("T: " + t.GetType().Name);
            Console.WriteLine("U: " + u.GetType().Name);
        }
    }
}
```

VB

```
Class Z(Of T As A)
    Private _t As T

    Public Sub Agregar(_t As T)
        Me._t = _t
    End Sub

    Public Function Obtener() As T
        Return _t
    End Function

    Public Sub Inspeccionar(Of U As A)(_u As U)
        Console.WriteLine("T: " + _t.GetType().Name)
        Console.WriteLine("U: " + _u.GetType().Name)
    End Sub
End Class
```

Se debe tener mucho cuidado de entender correctamente los cambios. Al definir las restricciones le indicamos al compilador que infiera si la conversión de tipo de un parámetro a otro tipo de parámetro es posible. Si el compilador puede realizar la conversión, no generará errores.

La siguiente clase verifica el correcto funcionamiento de la clase Z para la cadena de herencia del gráfico UML.

Ejemplo

C#

```
namespace limitaciones
{
    class Program
    {
        static void Main(string[] args)
        {
              Z<A> z1 = new Z<A>();
              Z<B> z2 = new Z<B>();
              Z<C> z3 = new Z<C>();
              Z<D> z4 = new Z<D>();

              z1.Agregar(new A());
              z1.Inspeccionar(new A());
              z1.Inspeccionar(new B());
              z1.Inspeccionar(new C());
              z1.Inspeccionar(new D());

              // La siguiente línea da Error porque el parámetro
              // del tipo para z2 es B
              // z2.Agregar(new A());
              z2.Agregar(new B());
              z2.Inspeccionar(new A());
              z2.Inspeccionar(new B());
              z2.Inspeccionar(new C());
              z2.Inspeccionar(new D());

              z3.Agregar(new C());
              z3.Inspeccionar(new A());
              z3.Inspeccionar(new B());
              z3.Inspeccionar(new C());
              z3.Inspeccionar(new D());

              z4.Agregar(new D());
              z4.Inspeccionar(new A());
              z4.Inspeccionar(new B());
              z4.Inspeccionar(new C());
              z4.Inspeccionar(new D());

        Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1

    Sub Main()
        Dim z1 As New Z(Of A)()
        Dim z2 As New Z(Of B)()
        Dim z3 As New Z(Of C)()
        Dim z4 As New Z(Of D)()
        ' Dim z5 As New Z(Of Double)() ' Error

        z1.Agregar(New A())
        z1.Inspeccionar(New A())
        z1.Inspeccionar(New B())
        z1.Inspeccionar(New C())
        z1.Inspeccionar(New D())

        ' La siguiente línea da Error porque el parámetro
        ' del tipo para z2 es B
        ' z2.Agregar(New A())
        z2.Agregar(New B())
        z2.Inspeccionar(New A())
        z2.Inspeccionar(New B())
        z2.Inspeccionar(New C())
        z2.Inspeccionar(New D())

        z3.Agregar(New C())
        z3.Inspeccionar(New A())
        z3.Inspeccionar(New B())
        z3.Inspeccionar(New C())
        z3.Inspeccionar(New D())

        z4.Agregar(New D())
        z4.Inspeccionar(New A())
        z4.Inspeccionar(New B())
        z4.Inspeccionar(New C())
        z4.Inspeccionar(New D())

        Console.ReadKey()
    End Sub
End Module
```

Si se ejecuta el programa se obtiene la siguiente salida.

```
T: A
U: A
T: A
U: B
T: A
U: C
T: A
U: D
T: B
```

```
U: A
T: B
U: B
T: B
U: C
T: B
U: D
T: C
U: A
T: C
U: B
T: C
U: C
T: C
U: D
T: D
U: A
T: D
U: B
T: D
U: C
T: D
U: D
```

Notar como el argumento declarado en el método admitió los objetos de las dos cadenas de herencia como parámetros.

El compilador puede inferir correctamente el tipo utilizado en las asignaciones cuando los parámetros de tipo son covariantes, es decir, una supuesta asignación de un parámetro de tipo a otro parámetro de tipo esta bien definida.

Que el compilador determine que un parámetro de tipo es coveriante respecto de otro parámetro de tipo es independiente de la clase o método que define dicho parámetro.

Con una declaración de este tipo, si se intenta usar la clase Z con un parámetro que no sea una subclase de A, se obtiene un error. Así una declaración como la siguiente:

C#

```
Z<Double> z5 = new Z<Double>(); // Error
```

VB

```
Dim z5 As New Z(Of Double)() ' Error
```

Genera errores como los siguientes, uno por cada parámetro mal usado:

1 The type 'double' cannot be used as type parameter 'T' in the generic type or method 'limitaciones.Z<T>'. There is no boxing conversion from 'double' to 'limitaciones.A'.
2 The type 'double' cannot be used as type parameter 'T' in the generic type or method 'limitaciones.Z<T>'. There is no boxing conversion from 'double' to 'limitaciones.A'.

Si se desea se pueden especificar restricciones adicionales que deben ser implementadas por los tipos usados como parámetros, se deben declarar utilizando el operador coma, como en el siguiente ejemplo.

Ejemplo

C#

```
        public void Inspeccionar<U>(U u) where U : A, I
```

VB

```
        Public Sub Inspeccionar(Of U As {A, I})(_u As U)
```

De esta manera, la limitación en el parámetro debe cumplir ambas condiciones. Se puede suponer como ejemplo la siguiente interfaz

Ejemplo

C#

```
namespace interfaces
{
    public interface I
    {
        void Met();
    }
}
```

VB

```
Public Interface I
    Sub Met()
End Interface
```

Una clase, que inclusive puede ser ella misma genérica, puede implementar la interfaz y a la vez heredar de otra clase como A, según se muestra a continuación:

Ejemplo

C#

```
namespace interfaces
{
    class F<H> : A, I
    {
        private H h;
```

```
        public F(H h)
        {
            this.h = h;
        }

        public void Met()
        {
            Console.WriteLine("H: " + h.GetType().Name +
                    ". Invocado polimórficamente");
        }
    }
}
```

VB

```
Public Class F(Of H)
    Inherits A
    Implements I

    Private _h As H

    Public Sub New(_h As H)
        Me._h = _h
    End Sub

    Public Sub Met() Implements I.Met
        Console.WriteLine("H: " + _h.GetType().Name + _
                    ". Invocado polimórficamente")
    End Sub
End Class
```

En este punto es donde se debe modificar la clase Z para restringir los parámetros del método Inspeccionar, quedando la clase de la siguiente manera.

Ejemplo

C#

```
namespace interfaces
{
    class Z<T>
    {
        private T t;

        public void Agregar(T t)
        {
            this.t = t;
        }

        public T Obtener()
        {
            return t;
        }
```

```
        public void Inspeccionar<U>(U u) where U : A, I
        {
            Console.WriteLine("T: " + t.GetType().Name);
            Console.WriteLine("U: " + u.GetType().Name);
            u.Met();
        }
    }
}
```

VB

```
Class Z(Of T)
    Private _t As T

    Public Sub Agregar(_t As T)
        Me._t = _t
    End Sub

    Public Function Obtener() As T
        Return _t
    End Function

    Public Sub Inspeccionar(Of U As {A, I})(_u As U)
        Console.WriteLine("T: " + _t.GetType().Name)
        Console.WriteLine("U: " + _u.GetType().Name)
        _u.Met()
    End Sub

End Class
```

Ahora se puede verificar como actúan las restricciones con las declaraciones realizadas para admitir parámetros genéricos

Ejemplo

C#

```
namespace interfaces
{
    class Program
    {
        static void Main(string[] args)
        {
            Z<F<B>> z1 = new Z<F<B>>();

            z1.Agregar(new F<B>(new B()));
            z1.Inspeccionar(new F<D<A>>(new D<A>()));

            Console.ReadKey();
        }
    }
}
```

VB

```
Module Module1
```

```
    Sub Main()
        Dim z1 As New Z(Of F(Of B))

        z1.Agregar(New F(Of B)(New B()))
        z1.Inspeccionar(New F(Of D(Of A))(New D(Of A)()))

        Console.ReadKey()
    End Sub
End Module
```

Notar el uso de parámetros anidados en las declaraciones de los objetos utilizados.

Este último programa produce la siguiente salida (el apóstrofe surge de la forma en que el compilador infiere parámetros de tipos anidados).

```
T: F`1
U: F`1
H: D`1. Invocado polimórficamente
```

Subtipos

Como se mencionara anteriormente, es posible asignar un objeto de un tipo a uno de otro tipo, siempre que estos tengan bien definida una conversión de tipo en su cadena de herencia. ***Sin embargo los tipos genéricos no son covariantes por si mismos salvo que existan restricciones que permitan la covarianza.***

Por ejemplo, puede asignar un número entero a un objeto, ya que Object es superclase de Int32.

Ejemplo

C#

```
Object objeto = new Object();
Int32 entero = new Int32();
objeto = entero; // Bien
```

VB

```
Dim objeto As New Object()
Dim entero As New Int32()
objeto = entero ' Bien
```

En la terminología orientada a objetos, esto se llama una relación "es un". Dado que un número entero es un tipo de objeto, la asignación está permitida. Pero Int32 es también del tipo Object, por lo que el siguiente código es válido también.

Ejemplo

C#

```
public static void unMetodo(Object n)
{
```

```
            // se omite el cuerpo
        }

        public static void otroMetodo(Object n)
        {
            unMetodo(new Int32()); // Bien
            unMetodo(new Double()); // Bien
        }
```

VB

```
    Public Sub UnMetodo(n As Object)
           ' se omite el cuerpo
    End Sub

    Public Sub OtroMetodo(n As Object)
           UnMetodo(New Int32()) ' Bien
           UnMetodo(New Double()) ' Bien
    End Sub
```

Lo mismo ocurre con los genéricos. Se puede realizar una invocación de tipo genérica, pasando como argumento de tipo Object, y cualquier llamada posterior de complemento se permitirá si el argumento es compatible con Object.

Ejemplo

C#

```
    A<Object> a = new A<Object>();
    a.agregar(new Int32()); // Bien
    a.agregar(new Double()); // Bien
```

VB

```
    Dim a As New A(Of Object)()
    a.Agregar(New Int32()) ' Bien
    a.Agregar(New Double()) ' Bien
```

Ahora considérese el siguiente método en la clase A.

Ejemplo

C#

```
    public void unMetodo(A<Object> a)
    {
           // Hace algo
    }
```

VB

```
    Public Sub UnMetodo(a As A(Of Object))
           '  Hace algo
    End Sub
```

La clase B que se define de la siguiente manera.

Ejemplo

C#

```
public class B<T> : A<T>
{
}
```

VB

```
Public Class B(Of T)
       Inherits A(Of T)

End Class
```

Y la siguiente invocación

Ejemplo

C#

```
A<Double> a2 = new A<Double>();
// a.unMetodo(a2); //Error
B<Double> b1 = new B<Double>();
// a2.unMetodo(b1); //Error
B<Object> b2 = new B<Object>();
a2.unMetodo(b2); //Correcto
```

VB

```
Dim a2 As New A(Of Double)()
' a.unMetodo(a2); ' Error
Dim b1 As New B(Of Double)()
' a2.unMetodo(b1); ' Error
Dim b2 = New B(Of Object)
a2.UnMetodo(b2) ' Correcto
```


¿Por qué dan error las invocaciones de métodos que se encuentran como comentarios? En el primer error se intenta pasar como parámetro al método un tipo A<Double> cuando admite como argumento A<Object>. Por más que Object es superclase de Double, ***las conversiones de tipo están definidas para las clases, no para los parámetros genéricos***. Esto se realizó de esta manera para mantener la consistencia del lenguaje. Se debe recordar que la finalidad de los parámetros genéricos es asegurar el tipo, pero si los parámetros admitieran subtipos de ellos, con sólo declarar a Object como el tipo del parámetro se podría realizar cualquier asignación y se perdería la seguridad de tipo buscada.

Por otra parte, en el segundo error comentado, si bien B es un subtipo de A, el parámetro asignado a la creación de B no es exactamente Object, y el problema se repite. Sin embargo, en la última línea, como B es subtipo de A y el parámetro utilizado es Object, la conversión de tipo entre A y B está bien definida.